PARTICIONAMENTO AUTOMÁTICO DE RESTRIÇÕES

Marluce Rodrigues Pereira

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Prof. Inês de Castro Dutra, Ph.D.

Prof. Maria Clicia Stelling de Castro, D.Sc.

Prof. Felipe Maia Galvão França, Ph.D.

Prof. Valmir Carneiro Barbosa, Ph.D.

Prof. Cláudio Fernando Resin Geyer, Dr.

Prof. Philippe Yves Paul Michelon, Dr.

RIO DE JANEIRO, RJ - BRASIL

MARÇO DE 2006

PEREIRA, MARLUCE RODRIGUES

Particionamento Automático de Restrições [Rio de Janeiro] 2006

XIV, 235 p. 29,7 cm (COPPE/UFRJ, D.Sc., Engenharia de Sistemas e Computação, 2006)

Tese – Universidade Federal do Rio de Janeiro, COPPE

1 - Particionamento de Restrições

2 - Algoritmos de Consistência de Arcos

3 - Coloração de grafos

I. COPPE/UFRJ II. Título (série)

Ao meu esposo Marcelo de Oliveira Moreira

Agradecimentos

Agradeço:
às minhas orientadoras Inês de Castro Dutra e Maria Clicia Stelling de Castro, pelo incentivo, confiança e apoio em todos os momentos;
à CAPES que deu suporte financeiro à realização deste trabalho;
ao professor Cláudio Amorim que gentilmente permitiu que eu utilizasse o *cluster* do LCP para realizar alguns experimentos e à sua equipe que me deu assistência quando precisei;
ao professor Enrico Pontelli, de *New Mexico State University*, que cedeu a máquina de memória compartilhada (*typhoon*) para eu realizar os experimentos;
aos professores Felipe França, Valmir Barbosa e Márcia Cerioli que contribuíram no estudo dos trabalhos existentes na área de pesquisa;
à amiga Patrícia Kayser pela amizade e pelo trabalho em conjunto;
ao amigo Marco Mangan pelos momentos de desabafo, descontração e compreensão dos trabalhos realizados nos fins de semana com a Patrícia;
ao amigo Luís Otávio Rigo Júnior pela parceria no trabalho realizado no LabIA na montagem do *cluster* como requisito do estágio à docência, pela manutenção do laboratório e pelo apoio recebido;
ao meu esposo Marcelo, pelo carinho, incentivo e compreensão;
aos meus pais e irmãos pelo incentivo;
aos amigos Regiane e Luís Oscar pela companhia e paciência nos momentos difíceis;
a todas as pessoas que de alguma forma auxiliaram-me no desenvolvimento e conclusão desta tese;
a Deus pela proteção, inspiração e por proporcionar as oportunidades em minha vida.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Doutor em Ciências (D. Sc.)

PARTICIONAMENTO AUTOMÁTICO DE RESTRIÇÕES

Marluce Rodrigues Pereira

Março/2006

Orientadoras: Inês de Castro Dutra
Maria Clicia Stelling de Castro

Programa: Engenharia de Sistemas e Computação

Esta tese concentra-se no desenvolvimento de um novo método de particionamento de restrições geral para problemas de satisfação de restrições (CSPs). Uma restrição pode ser descrita em função de mais de um *indexical*, que explicita cada variável da restrição, gerando um grafo de granulosidade mais fina. Por isso, o novo método, denominado *Grouping-Sink*, pode ser aplicado a restrições e a *indexicals*. Ele agrupa nós (restrições ou *indexicals*) de um grafo baseando-se na dependência (variáveis comuns) existente entre os nós, buscando maior concorrência. O particionamento baseia-se na remoção de nós *sinks* num grafo com uma orientação acíclica (gerada por *Alg-Colour*). Os grupos obtidos são distribuídos entre processos ou processadores para uma execução paralela/distribuída.

Os resultados experimentais obtidos mostram que *Grouping-Sink* por *indexicals* e por restrições são métodos gerais para qualquer CSP, pois apresentam resultados melhores ou semelhantes aos outros particionamentos aos quais foram comparados.

As contribuições mais relevantes deste trabalho são o desenvolvimento dos algoritmos *Grouping-Sink* por *indexicals* e por restrições, duas formas de representação do grafo de restrições (com os nós sendo restrições e sendo *indexicals*), a adaptação do sistema PC-SOS para validação do método, a execução numa máquina paralela real e a demonstração empírica de que *Alg-Colour*, o algoritmo de orientação acíclica utilizado neste trabalho, sempre gera nós *sink* que pertencem ao período.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

AUTOMATIC CONSTRAINT PARTITIONING

Marluce Rodrigues Pereira

March/2006

Advisors: Inês de Castro Dutra
Maria Clicia Stelling de Castro

Department: Systems Engineering and Computer Sciences

This thesis focuses on the study of a new general constraint partitioning method to Constraint Satisfaction Problems (CSPs). A constraint can be represented as a set of indexicals, where each indexical makes explicit each variable of the constraint. An indexical graph is more fine-grained than a constraint graph. So, the new method named Grouping-Sink can be applied to constraints and to indexicals. Grouping-Sink groups nodes (constraints or indexicals) in a graph according to the dependency (common variables) between the nodes. It tries to get maximum concurrency, reducing the communication between the groups. Grouping-Sink removes sink nodes in a graph with acyclic orientation (obtained with Alg-Colour). The generated groups are distributed between processes or processors to execute in a parallel or distributed way.

The experimental results show that Grouping-Sink per indexicals and per constraints are general methods for partitioning CSPs. Their results are better or similar to the other partitioning methods implemented in PCSOS.

Our main contributions are the implementation of Grouping-Sink per indexicals and per constraints, two new forms of representing the constraint graph, the adaptation of the PCSOS system to validate the new methods, the execution in a real parallel machine, and the empirical demonstration that Alg-Colour always generates sink nodes that belong to the period.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Em várias áreas do conhecimento existem problemas cuja formulação se enquadra numa classe de formalização e solução denominada programação por restrições. Normalmente, estes problemas possuem um grande número de restrições que precisam ser satisfeitas e envolvem muitas variáveis. Considere, por exemplo, o problema de definir as salas de aula para um semestre de uma universidade [22]. Existem várias restrições que devem ser satisfeitas, tais como: todos os cursos precisam de uma sala de aula para serem ministrados; não pode haver coincidência de duas aulas na mesma sala, ao mesmo tempo; um professor não pode ser alocado em duas turmas simultaneamente, entre outras.

À medida que introduzimos novas restrições e/ou variáveis ao problema, aumentamos o espaço de busca por uma solução, o que faz com que estes problemas caiam numa classe de resolução de complexidade muito alta, cujos métodos para resolvê-los possuem complexidade temporal e espacial exponencial. Estes problemas são conhecidos como *problemas de satisfação de restrições* ou *redes de restrições* [22].

Um problema formulado como satisfação de restrições possui três componentes principais: variáveis, domínios e restrições. A representação do problema pode ser feita através de conjuntos de equações e/ou inequações que relacionam as variáveis. Muitas vezes, alguns conjuntos de equações e/ou inequações são transformados em uma única restrição, que é implementada de forma eficiente em alguma linguagem/solver (por exemplo, `alldifferent`, `before` etc). Neste trabalho, nos concentramos apenas em restrições primitivas, ou seja, aquelas baseadas em equações e inequações. Cada variável é inicializada com seu domínio respectivo, ou seja, um conjunto de valores. No caso de domínios finitos, o conjunto é discreto, finito e enumerável.

Os problemas de satisfação de restrições sobre domínios finitos são combinatórios. Portanto, apresentam um espaço de busca exponencial. Porém, o espaço de busca pode ser podado através de algoritmos de consistência. A poda do espaço de busca ocorre quando são removidos valores inconsistentes dos domínios das variáveis de acordo com as restrições impostas sobre as mesmas. As podas no espaço de busca podem acelerar a

solução do problema.

Para exemplificar um problema de satisfação de restrições sobre domínios finitos, suponha o problema de se colocar N rainhas em um tabuleiro de xadrez $N \times N$ de tal forma que as rainhas não se ataquem [1]. As rainhas se atacam se estiverem na mesma linha, na mesma coluna, na mesma diagonal principal ou na mesma diagonal secundária. Cada rainha deve ser colocada em uma linha do tabuleiro. O problema consiste em selecionar uma coluna para cada rainha, de forma que elas não se ataquem.

Uma representação em satisfação de restrições deste problema consiste em associar cada rainha a uma variável e fixar cada rainha numa linha do tabuleiro. Cada variável pode assumir valores de 1 a N, que são os valores das colunas que as rainhas podem ocupar e que correspondem ao domínio do problema. As restrições correspondem às condições necessárias e suficientes para que as rainhas não se ataquem. Para exemplificar o conjunto de restrições, suponha que X e Y sejam duas rainhas. As restrições para que a rainha Y seja colocada no tabuleiro de xadrez de forma que não ataque a rainha X, colocada anteriormente, pode ser escrita da seguinte forma:

$Y \neq X$, rainhas X e Y não se atacam na mesma coluna;

$Y \neq X + I$, rainhas X e Y não se atacam na diagonal principal e

$Y + I \neq X$, rainhas X e Y não se atacam na diagonal secundária,

onde $I \in 1, ..., N$ corresponde às posições (colunas) que uma rainha pode ocupar (domínio).

Uma classe importante dos algoritmos para resolver problemas de satisfação de restrições (*Constraint Satisfaction Problems* - CSPs) é a classe dos algoritmos de consistência de arcos [45].

Os algoritmos de consistência de arcos permitem eliminar valores inconsistentes do domínio das variáveis de forma a podar o espaço de busca. De uma forma geral, estes algoritmos são integrados em *solvers* de propósito geral que possuem três fases: consistência de nós, consistência de arcos e *labeling*.

A consistência de nós consiste em executar as restrições que relacionam apenas uma variável. Após esta fase é realizada a consistência de arcos, onde as restrições que relacionam mais de uma variável são executadas para eliminar valores inconsistentes dos domínios das variáveis. Apesar de reduzir o tamanho do domínio, a execução de um algoritmo de consistência de arcos não é suficiente para atingir a solução de um CSP. Isto é, reduzir o domínio de cada variável a um único valor. Então, após a fase de consistência de arcos é necessário realizar a fase de *labeling*.

Na fase de *labeling*, uma variável e um valor do seu domínio são selecionados. Esta variável e seu valor são propagados. Estes dados são utilizados para eliminar valores inconsistentes das outras variáveis através da execução das restrições. Na fase de propa-

gação é examinada cada restrição que relacione a variável que foi escolhida. Se a variável e o valor escolhidos eliminarem todas as soluções possíveis para o CSP é realizado, então, o *backtracking*. Primeiramente, no procedimento de *backtracking* é escolhido um novo valor no domínio da variável. Se todos os valores para a variável falharem, uma nova variável é escolhida. O algoritmo termina quando é encontrada uma atribuição de um único valor para cada variável ou se todos os valores possíveis para as variáveis forem testados e não foi possível encontrar uma solução. Este tipo de algoritmo trata de restrições entre pares de variáveis. Existem outros métodos, tais como consistência de hiper-arcos [44, 22, 43, 49], porém são pouco utilizados por apresentarem complexidade muito alta.

Na literatura, diversos algoritmos de consistência de arcos foram propostos como AC-1 [41], AC-2 [43], AC-3 [41], AC-4 [49], AC-5 [40], AC-6 [9] e AC-7 [10]. Nosso trabalho concentra-se no algoritmo AC-5, que consiste na generalização dos algoritmos de AC-1 a AC-4. O algoritmo AC-5 possui complexidade $O(ed)$, onde e é o número de restrições e d é o tamanho do domínio.

O algoritmo AC-5 apresenta características que facilitam a sua implementação paralela, além de possuir complexidade baixa quando comparado aos demais algoritmos de consistência de arcos [62], cujas complexidades são pelo menos um fator expoente mais alto do que a do AC-5.

Como os CSPs apresentam espaço de busca que cresce exponencialmente em função do tamanho do problema, é necessário obter alternativas para acelerar a sua execução ou mesmo conseguir executar instâncias maiores de problemas. Uma alternativa é particionar um dos componentes do problema: conjunto de variáveis, domínios ou restrições. Pode-se ainda combinar estes particionamentos. Este trabalho concentra-se no particionamento de restrições, isso porque trabalhos anteriores mostraram resultados promissores [62, 57].

O particionamento do conjunto de restrições de um CSP permite obter resultados melhores tanto para a execução seqüencial quanto para a execução paralela, como mostrado em trabalho anterior [56]. Neste trabalho foram realizados experimentos com uma aplicação CSP denominada *Arithmetic* sem particionar o conjunto de restrições e realizando um particionamento manual. Ao executar esta aplicação de forma seqüencial, obtivemos o tempo de execução de 4,74 segundos. Realizando experimentos com esta aplicação para 1 e n processadores do mesmo tipo obtivemos os resultados que encontram-se na Tabela 1.1.

Nesta tabela são apresentados os resultados da execução da aplicação *Arithmetic* para 1 processador, com 2, 4 e 8 processos (segunda coluna) e para 2, 4 e 8 processadores (terceira coluna). Na última coluna é apresentado o *Speedup* obtido. Podemos observar que o particionamento da aplicação para ser executada em um mesmo processador (se-

Número de Processos	1 processador	n processadores	$Speedup$
2	0,55	0,39	1,40
4	0,53	0,37	1,42
8	0,59	0,47	1,24

Tabela 1.1: Tempos de execução de *Arithmetic* para 1 proc. e n procs. [56].

gunda coluna) diminui bastante o tempo de execução em relação ao tempo seqüencial (4,74 seg.). Além disso, o particionamento para a execução paralela da aplicação (terceira coluna) também permite a obtenção de um bom *speedup*. Assim, podemos notar que o particionamento de uma aplicação é importante não só para sua execução paralela mas também para sua execução seqüencial. Na execução seqüencial o número de passos para se chegar a uma solução pode ser menor, quando se usa um particionamento.

O particionamento de um CSP pode melhorar significativamente a sua execução. Assim, torna-se importante utilizar algum mecanismo que realize o particionamento automático de um conjunto de restrições.

A obtenção de subconjuntos de restrições independentes é um problema NP-completo. O particionamento do conjunto de variáveis ou do domínio, também, é um problema NP-completo.

Um CSP pode ser modelado como um grafo de restrições, onde os nós representam variáveis e as arestas, restrições entre as variáveis [57]. Outra forma de representação consiste em considerar os nós como restrições e as arestas como variáveis que são comuns a duas restrições.

A partir do momento que temos o CSP modelado como um grafo, há várias formas de realizar o particionamento.

O problema que queremos resolver é definido da seguinte forma: dado um grafo $G = (C, E)$, onde C é o conjunto de nós (restrições) e E é o conjunto de arestas (variáveis comuns entre as restrições), queremos particionar G em subgrafos (grupos de nós) totalmente conectados (cliques).

Os trabalhos relacionados à consistência de arcos, existentes na literatura, buscam melhorar a complexidade do algoritmo, de forma a executar seqüencialmente um número menor de passos para alcançar a solução [9, 10, 23]. Porém poucos trabalhos se preocupam em particionar a rede de restrições, de forma a obter subgrupos de restrições que possam reduzir a complexidade total para encontrar uma solução.

Em teoria dos grafos existem algoritmos que buscam particionar grafos de forma que se obtenha subgrafos com mesmo número de nós e que o número de arestas entre os subgrafos seja minimizado. Como exemplo destes métodos podemos citar os métodos de particionamento simples (linear, aleatório, *scattered*) e os métodos espectrais de bisseção,

quadrisseção e octosseção [39, 65]. A implementação destes métodos utiliza representação matemática e apresenta complexidade exponencial (ver Apêndice B para um melhor detalhamento destes métodos). Para grafos mais complexos, onde o número de arestas é grande, como nos CSPs, este tipo de particionamento pode não ser o mais adequado. Isso porque as características da aplicação não são consideradas e para grafos mais conexos a comunicação pode ser muito alta, se as arestas estiverem relacionadas à comunicação entre os processadores.

Outro método que pode ser utilizado para particionar restrições é o método de Escalonamento por Reversão de Arestas (SER-*Scheduling by Edge Reversal*) [7], onde observamos algumas características importantes que são apresentadas a seguir.

Este método consiste em manipular orientações acíclicas de um grafo conexo, onde um nó representa um processo e uma aresta entre dois nós representa o compartilhamento de um recurso entre os nós. Um recurso é representado em um grafo G por um clique. Isto é, um subgrafo totalmente conectado, já que todos os processos que compartilham o mesmo recurso estão conectados. Note que um processo pode acessar um número arbitrário de recursos.

A execução deste mecanismo em um grafo acíclico gera agrupamentos de nós (blocos) buscando alcançar uma melhor concorrência para a aplicação. Este fator é bastante importante em particionamento de massas de dados para serem executadas em paralelo.

As características dos blocos gerados após o particionamento de restrições são bastante importantes, pois a execução desnecessária de restrições pode aumentar o tempo de execução total do algoritmo de consistência de arcos. Na fase de propagação dos algoritmos de consistência de arcos, dependendo do momento em que uma variável é rotulada pode-se evitar a realização de um *backtracking* e, conseqüentemente, a execução de muitas outras restrições. Este fato está diretamente ligado à ordem em que as restrições são executadas e em que blocos as restrições estão localizadas.

O Escalonamento por Reversão de Arestas agrupa as restrições baseando-se na dependência existente entre elas, buscando maior concorrência (menor comunicação entre blocos). A desvantagem deste método é que possui um tempo de execução muito alto quando aplicado a problemas muito grandes. Este fato motivou a implementação de outro método que produzisse pelo menos os mesmos resultados, mas que tivesse um tempo de execução menor. Desta forma, surgiu o particionamento *Grouping-Sink*, que utiliza o método de decomposição em *sinks* de um grafo que possua orientação acíclica inicial.

Assim, os objetivos principais deste trabalho são os seguintes:

1. pesquisar métodos de particionamento automáticos de conjuntos de restrições de CSPs para facilitar a execução seqüencial e em paralelo de tais problemas e

2. propor um método de particionamento que seja geral para qualquer problema que possa ser representado como um CSP. Um passo de pré-processamento antes da execução de um CSP é suficiente para transformar um problema no formato necessário para o método de particionamento geral.

O novo método de particionamento proposto é comparado com os métodos existentes para avaliar seu desempenho. Os problemas utilizados para avaliar o novo método *Grouping-Sink* apresentam diferentes grafos de restrições e atuam sobre domínios finitos.

Nosso trabalho difere dos demais porque investiga o particionamento de restrições considerando as características da aplicação. Ou seja, buscamos obter uma reorganização do conjunto de restrições, antes de sua execução seqüencial ou em paralelo. Esta reorganização tem o objetivo de evitar que execuções desnecessárias de restrições sejam realizadas. São consideradas as dependências existentes entre as restrições no momento de se formar os subgrupos.

O algoritmo de particionamento geral e automático proposto neste texto é denominado *Grouping-Sink*. Há duas versões implementadas: *Grouping-Sink* por restrições e *Grouping-Sink* por *indexicals*. *Grouping-Sink* por restrições agrupa as restrições. Uma restrição pode, também, ser representada por um esquema de *indexicals* [20], onde cada variável da restrição é explicitada em função das demais variáveis. Então, uma restrição pode ser representada por mais de um *indexical*. *Grouping-Sink* por *indexicals* agrupa *indexicals*. Este particionamento apresentou os melhores resultados, pois os *indexicals* proporcionam agrupamentos melhores, aumentando a computação local e diminuindo a comunicação entre os processadores.

Para avaliar o particionamento *Grouping-Sink* por restrições e *Grouping-Sink* por *indexicals* nos baseamos num sistema chamado PCSOS desenvolvido por Andino *et al.* [62] na Universidade Complutense de Madri. PCSOS implementa o algoritmo de consistência de arcos AC-5 [40].

Os resultados obtidos com os experimentos mostraram que *Grouping-Sink* por restrições é o melhor particionamento para *Arithmetic*. *Grouping-Sink* por *indexicals* ficou entre os melhores particionamentos para *Queens* e PBCSP, mas apresenta menor quantidade de mensagens do que os demais particionamentos. Essa característica é bastante importante em um particionamento para execução num sistema distribuído. Para *Sudoku*, *Grouping-Sink* por *indexicals* teve um resultado intermediário. Estes resultados mostram que *Grouping-Sink* por *indexicals* apresenta resultados melhores ou semelhantes aos métodos de particionamento utilizados anteriormente [57]. E *Grouping-Sink* por restrições apresenta resultados intermediários aos particionamentos aos quais foi comparado. Porém, *Grouping-Sink* é aplicável a qualquer problema que for modelado com restrições ou *indexicals* e é um método automático.

As principais contribuições deste trabalho são as seguintes:

1. proposição dos algoritmos gerais de particionamento denominados *Grouping-Sink* por *indexicals* e *Grouping-Sink* por restrições;

2. duas novas representações dos grafos de restrições: com os nós representando as restrições e as arestas representando as variáveis comuns entre as restrições, e com os nós representando os *indexicals* e as arestas representando as variáveis comuns entre os *indexicals* (normalmente um CSP é representado por um grafo, onde os nós são as variáveis e as arestas são as restrições. Esta representação é conveniente por causa dos algoritmos de consistência, porém para o nosso propósito, esta representação não é adequada);

3. apresentação de resultados com a utilização dos métodos de particionamento numa máquina paralela real;

4. demonstração empírica de que o algoritmo de orientação acíclica utilizado neste trabalho gera sempre nós *sink* que pertencem ao período.

Os demais capítulos estão organizados da seguinte forma. No Capítulo 2 apresentamos os conceitos de algoritmos de consistência de arcos e realizamos uma análise de alguns deles. No Capítulo 3 são apresentados os tipos de particionamento de restrições existentes, os métodos de particionamento de grafos existentes na literatura de teoria dos grafos, com suas vantagens e desvantagens e apresentamos porque o método de particionamento *Grouping-Sink* pode ser uma boa solução para o problema de particionamento. O Capítulo 4 aborda o método de particionamento *Grouping-Sink*. O Capítulo 5 apresenta as plataformas de *hardware* e *software*, o sistema PCSOS e as aplicações utilizadas. O Capítulo 6 apresenta os resultados obtidos com o particionamento *Grouping-Sink* em relação aos métodos de particionamento existentes e sua análise. O Capítulo 7 apresenta as conclusões deste trabalho e os trabalhos futuros.

Capítulo 2

Problemas de Satisfação de Restrições e Consistência de Arcos

Uma grande variedade de problemas, em Inteligência Artificial e em outras áreas da Ciência da Computação, podem ser considerados Problemas de Satisfação de Restrições (*Constraint Satisfaction Problems-CSPs*). Uma forma de solucionar tais problemas é utilizando algoritmos de consistência de arcos. Esses algoritmos são usados para podar o espaço de busca de um CSP. Este capítulo apresenta o conceito de CSP e como os algoritmos de consistência de arcos podem ser aplicados a tais problemas. Além disso, descreve o algoritmo de consistência de arcos AC-5, que é utilizado em nosso trabalho.

2.1 Considerações iniciais

Um modelo que envolva variáveis, seus domínios e restrições entre variáveis é chamado de problema de satisfação de restrições ou rede de restrições [22] . Neste texto é utilizada a notação problema de satisfação de restrições (*Constraint Satisfaction Problem* - CSP).

Um CSP é um tipo especial de problema de busca que possui estados e domínios. Os estados são conjuntos de variáveis. O estado inicial é um conjunto de variáveis com valores possíveis iniciais. O estado final é um conjunto de variáveis com valores que respeitem as restrições do problema. O domínio é o conjunto possível de valores que uma variável pode assumir, que pode ser discreto ou contínuo e finito ou infinito.

O CSP define restrições sobre variáveis e um domínio que relaciona cada variável a um conjunto de valores.

Um *solver* é um método para resolver CSPs. Vários *solvers* de CSP possuem complexidade polinomial. O objetivo dos *solvers* é transformar um CSP, que tem espaço de busca exponencial, em outro equivalente com os domínios menores para as variáveis [56]. Para encontrar as soluções o *solver* passa por fases de eliminação de valores do domínio das variáveis, de acordo com as restrições.

Os *solvers* dependem do domínio das variáveis. No caso de restrições no domínio real (infinito), são utilizados métodos matemáticos como eliminação de Gauss e Fourier-Moutzkin ou métodos computacionais como o Simplex [68]. No caso de domínios finitos, o método mais utilizado é o de consistência de arcos. Nesta tese nos concentramos em domínios finitos com método de consistência de arcos.

Para solucionar CSPs que possuem domínios finitos são necessárias duas escolhas: a da variável e a do valor da variável. A escolha da variável pode ser *most-constrained*, *most-constraining* ou *least-constrained*. Na escolha *most-constrained* é escolhida a variável de menor domínio. Na *most-constraining*, a variável escolhida é a que restringe ao máximo os domínios das outras variáveis. Na *least-constrained* a variável com maior domínio é escolhida. Além disso, *least-constrained* utiliza uma heurística baseada na minimização do número de falhas para evitar *backtracking*.

A escolha do valor da variável, também, pode ser feita utilizando-se vários métodos: *least-constraining*, menor valor, valor médio, maior valor ou valor seqüencial. Pelo princípio *least constraining*, o valor escolhido é aquele que afeta menos o conjunto de valores das outras variáveis. A escolha dos valores menor, médio ou maior consiste em escolher, respectivamente, o menor, o valor médio ou o maior valor do conjunto de valores do domínio. A escolha de um valor seqüencial consiste em selecionar o próximo valor do domínio que ainda não foi escolhido.

Considerando apenas problemas de satisfação de restrições que possuem um conjunto de variáveis, um domínio finito para cada variável e um conjunto de restrições unárias ou binárias, é possível representar o CSP por um grafo de restrições, onde cada nó representa uma variável e cada arco representa uma restrição entre variáveis. Qualquer CSP com restrições n-árias pode ser convertido em um CSP binário equivalente [41].

Como exemplo de um CSP com restrições binárias, podemos citar o problema de coloração de mapas. Ele pode ser considerado como um CSP com domínio finito. Neste problema precisamos colorir cada região do mapa com uma cor (de um conjunto de cores), tal que duas regiões adjacentes não tenham a mesma cor.

A Figura 2.1 mostra um exemplo do problema de coloração de mapas e seu CSP equivalente, representado por um grafo de restrições tradicional. O mapa tem quatro regiões que devem ser coloridas com as cores verde, azul ou vermelho. O CSP equivalente tem uma variável para cada uma das quatro regiões do mapa (V_1, V_2, V_3, V_4). O domínio de cada uma das variáveis é dado pelo conjunto de cores. Para cada par de regiões que são adjacentes no mapa, há uma restrição binária ($V_i \neq V_j$) entre as variáveis correspondentes que não permite atribuições idênticas para as duas variáveis. No grafo da Figura 2.1, cada nó representa uma variável do problema (V_1, V_2, V_3, V_4) e as arestas representam as restrições existentes entre as variáveis ($V_1 \neq V_2$, $V_1 \neq V_3$, $V_1 \neq V_4$, $V_2 \neq V_3$, $V_3 \neq V_4$).

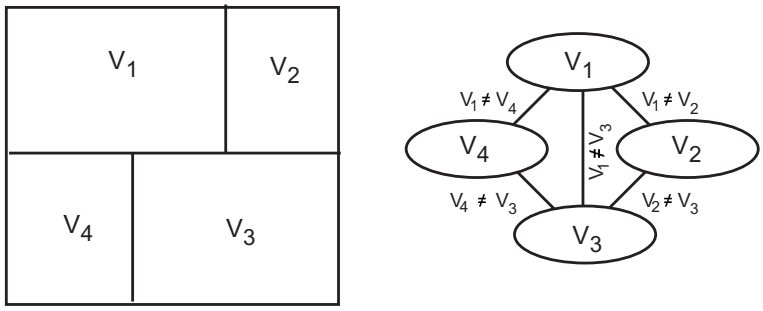


Figura 2.1: Exemplo de um problema de coloração de mapas e seu grafo de restrições equivalente [41]

Na implementação do *solver* para um CSP podem ser identificadas três fases principais: consistência de nós, consistência de arcos e *labeling*.

A consistência de nós elimina aqueles valores dos domínios das variáveis que as tornam inconsistentes de acordo com as restrições. Isto é, as restrições que envolvam apenas uma variável (por exemplo, $V > 0$) [57].

A consistência de arcos é realizada sobre o grafo de variáveis e restrições. Um arco (V_i,V_j) é consistente se para todo valor x no domínio corrente de V_i há algum valor de y no domínio de V_j tal que $V_i = x$ e $V_j = y$ é um par de valores permitidos pela restrição binária entre V_i e V_j. O conceito de consistência de arcos é unidirecional, isto é, se um arco (V_i,V_j) é consistente não implica que o arco (V_j,V_i) também seja consistente [41].

Para realizar a fase de consistência de arcos existem os algoritmos de consistência de arcos. Eles permitem eliminar valores inconsistentes do domínio das variáveis de forma a podar o espaço de busca. Na literatura, diversos algoritmos de consistência de arcos foram propostos como AC-1 [41], AC-2 [43], AC-3 [41], AC-4 [49], AC-5 [40], AC-6 [9] e AC-7 [10]. Nosso trabalho concentra-se no algoritmo AC-5. Os algoritmos de consistência de arcos não são suficientes para se chegar a uma solução, onde cada variável fica com apenas um valor em seu domínio, satisfazendo todas as restrições. Por isso, é necessário realizar a fase de *labeling* associada à consistência de arcos.

Na fase de *labeling* seleciona-se uma variável e um valor do domínio desta variável. Com a variável e o valor escolhidos tenta-se eliminar valores inconsistentes das outras variáveis, examinando cada restrição da variável escolhida (fase de propagação). Neste momento o algoritmo de consistência de arcos é executado novamente. A terminação do algoritmo de resolução do CSP ocorre quando é encontrada uma atribuição de um único valor para cada variável ou quando todas as restrições são executadas mas não é possível encontrar uma solução. Neste momento, o algoritmo alcança o ponto fixo, onde nenhum domínio é modificado numa próxima iteração, ou seja, os domínios permanecem inalterados por duas iterações consecutivas. Para cada variável são consideradas as restrições da variável e o domínio de outra variável envolvida nas mesmas restrições. O domínio

de uma outra variável, então, pode ser manipulado para podar os valores inconsistentes baseado nas restrições. O processo se repete até que nenhum valor do domínio de uma variável possa ser eliminado.

Existem vários algoritmos de consistência de arcos. Cada um dos algoritmos existentes pode ser utilizado para resolver o problema exemplificado na Figura 2.1.

Nosso trabalho utiliza o algoritmo AC-5. A próxima seção apresenta este algoritmo. Detalhes sobre os demais algoritmos podem ser encontrados no Apêndice A.

As seções seguintes estão organizadas da seguinte forma. Iniciamos apresentando o algoritmo de consistência de arcos AC-5 e a análise de sua complexidade. A Seção 2.3 apresenta uma discussão sobre algoritmos de consistência.

2.2 Algoritmo AC-5

O algoritmo AC-5, apresentado em [40], é um algoritmo genérico. As variáveis assumem valores de números naturais e possuem um domínio finito D_{V_i} associado. Todas as restrições são binárias e relacionam duas variáveis distintas. No grafo implementado por AC-5 um arco representa uma restrição entre duas variáveis.

O algoritmo AC-5 está descrito na Figura 2.2. O termo Δ retorna os valores que foram removidos do domínio. O termo $C_{V_iV_j}$ representa uma restrição entre as variáveis V_i e V_j.

AC-5 possui dois passos principais. No primeiro passo, todos os arcos são considerados uma vez e a consistência de nós é realizada para cada um deles. Uma fila Q é gerada com elementos da forma $\langle(V_k, V_i), w\rangle$, onde (V_k, V_i) é o arco que chega em V_i e w é o elemento removido ao se fazer a consistência do arco (V_k, V_i). No segundo passo, computa-se valores em D_{V_i} afetados pela remoção do valor w para cada elemento de Q, possibilitando a geração de novos elementos em Q e a repetição do processo. O procedimento *InitQueue(Q)* inicializa a fila Q vazia. A função *EmptyQueue(Q)* testa se a fila está vazia. O procedimento *Enqueue(V_i, Δ, Q)* é usado quando o conjunto de valores Δ é removido de D_{V_i}. Ele introduz elementos na forma $\langle(V_k, V_i), v\rangle$ na fila Q onde (V_k, V_i) é um arco do grafo de restrições e $v \in \Delta$. O procedimento *Dequeue(Q, V_i, V_j, w)* retira um elemento da fila. O procedimento *ArcCons(V_i, V_j, Δ)* computa o conjunto de valores Δ para a variável V_i que não é suportado por D_{V_j}. O procedimento *LocalArcCons(V_i, V_j, w, Δ)* é usado para computar o conjunto de valores em D_{V_i} não mais suportado porque os valores w de D_{V_j} foram removidos.

A execução deste algoritmo ocorre com a inicialização da fila Q ($Q = \emptyset$), através do procedimento *InitQueue(Q)*. Em seguida, os procedimentos $ArcCons(V_i, V_j, \Delta)$, *Enqueue($V_i, \Delta, Q$)* e *Remove($\Delta, D_{V_i}$)* são realizados, podando o domínio das variáveis afetadas.

Procedure AC-5
Post: let P_0 = D_{1_0} *x ... x* D_{n_0},
$P = D_1$ *x ... x* D_n
G is maximally arc consistent with P in P_0
1 InitQueue(Q)
2 foreach $(V_i, V_j) \in arc(G)$ *do*
3 ArcCons(V_i, V_j, Δ*);*
4 Enqueue(V_i, Δ, Q*);*
5 Remove(Δ, D_{V_i}*);*
6 endforeach
7 while not EmptyQueue(Q) do
8 Dequeue(Q, V_i, V_j, w*);*
9 LocalArcCons(V_i, V_j, w, Δ*);*
10 Enqueue(V_i, Δ, Q*);*
11 Remove(Δ, D_{V_i}*);*
12 endwhile
end_AC5

Procedure ArcCons(in V_i, V_j, *out* Δ*)*
Pre: $(V_i, V_j) \in$ *arc(G),* $D_{V_i} \neq \emptyset$, *and* $D_{V_j} \neq \emptyset$
Post: Δ = {$v \in D_{V_i} \mid \forall w \in D_{V_j}$: $\neg C_{V_iV_j}(v, w)$}

Procedure LocalArcCons(inV_i, V_j, w, *out* Δ*)*
Pre: $(V_i, V_j) \in arc(G), w \notin D_{V_j}, D_{V_i} \neq \emptyset$ *and* $D_{V_j} \neq \emptyset$
Post: $\Delta_1 \subseteq \Delta \subseteq \Delta_2$,
with Δ_1 = {$v \in D_{V_i} \mid C_{V_iV_j}$(v,w) *and* $\forall$ w' $\in D_{V_j}$: $\neg C_{V_iV_j}$(v,w') },
Δ_2 = {$v \in D_{V_i} \mid \forall$ w' $\in D_{V_j}$: $\neg C_{V_iV_j}$(v,w') }

Figura 2.2: Procedimento AC-5

A complexidade deste algoritmo é $O(ed)$, onde e representa o número de arestas e d o tamanho do domínio [40].

A utilização deste algoritmo deve-se à sua baixa complexidade em relação a outros algoritmos e pelo fato de ter sido criado para executar de forma mais adequada restrições funcionais e monotônicas. AC-5 está implementado no *solver* PCSOS [62] que estamos utilizando em nossos experimentos e que é detalhado no Capítulo 5.

Na implementação do AC-5 no PCSOS, as restrições são representadas na forma de *indexicals* [20]. O esquema de *indexicals* consiste em definir as restrições na forma *X in r*, onde X é uma variável e r é uma expressão. Desta forma, uma restrição mais complexa é representada por um conjunto de *indexicals*. Cada *indexical* descreve uma variável X dependente das demais variáveis.

Suponha a restrição aritmética $V_1 = V_2 + 4$ sobre o domínio dos inteiros. Ela pode ser transformada em dois *indexicals*, explicitando apenas uma variável de cada vez:

$$I_1 = V_1 \; in \; min\,(V_2) + 4 \;\; . . \;\; max\,(V_2) + 4$$

$$I_2 = V_2 \; in \; min\,(V_1) - 4 \;\; . . \;\; max\,(V_1) - 4$$

Cada variável é colocada em função de máximos e mínimos. Quando o limite inferior de V_2 ($min\,(V_2)$) ou o limite superior ($max\,(V_2)$) se modifica, devido à eliminação de algum valor do domínio, o *indexical* I_1 remove valores inconsistentes do domínio de V_1. O *indexical* I_2 comporta-se similarmente. Suponha que os domínios iniciais de V_1 e V_2 sejam iguais ao intervalo $[1, \ldots, 10]$. Ao aplicarmos o algoritmo de consistência de nós, o *indexical* I_1 é executado e o valor 1 é atribuído a $min\,(V_2)$ e o valor 10 a $max\,(V_2)$. Então o domínio de V_1 torna-se $[5, \ldots, 14]$. Porém, o valor máximo do domínio de V_1 é 10. Assim, a restrição I_1 poda o domínio de V_1 para $[5, \ldots, 10]$ e I_2, da mesma forma, poda o domínio de V_2 para $[1, \ldots, 6]$. Se outro *indexical* relacionado a V_2 poda seu domínio para $[4, \ldots, 5]$, I_1 substituirá $min\,(V_2)$ por 4 e $max\,(V_2)$ por 5, podando o domínio de V_1 para $[8, \ldots, 9]$. Entretanto, esta modificação é propagada por causa da execução dos *indexicals* que dependem de V_1 na fase de consistência de arcos. Em geral, uma restrição com n variáveis, se converte em n *indexicals*.

Na próxima seção realizamos uma discussão sobre os algoritmos de consistência de arcos e motivamos a necessidade do particionamento de restrições para melhorar o desempenho de aplicações em termos de tempo e quantidade de mensagens.

2.3 Discussão

Na literatura, vários algoritmos de consistência são estudados (Apêndice A). Cada um desses algoritmos possui uma característica particular. O algoritmo que possui maior complexidade de tempo é o AC-1, que é igual a $O(end^3)$, onde e é o número de arestas, n é o número de variáveis e d é o tamanho do domínio. Isto porque é um algoritmo de força bruta, que percorre a fila de arcos a serem executados inúmeras vezes. Desta forma, são realizadas verificações redundantes de arcos. Com o intuito de melhorar este algoritmo surgiu o algoritmo AC-2 e, em seguida, o AC-3.

O algoritmo AC-2 difere do AC-1 porque os nós são previamente ordenados e a execução é realizada seguindo a ordem dos nós. Esta ordenação dos nós deve-se ao fato do AC-2 considerar que um novo nó pode ser inserido no grafo de restrições. A complexidade de AC-2 é $O(ed^3)$.

O algoritmo AC-3 é uma extensão do algoritmo AC-2 que realiza o procedimento de verificação somente dos arcos que possivelmente foram afetados pela eliminação de um valor de uma determinada variável. A complexidade de tempo de AC-3 é $O(ed^3)$ e a complexidade de espaço é $O(e)$.

Com base no conceito de conjunto suporte [1] surgiu o algoritmo AC-4. O algoritmo

[1] Para entendermos melhor o conceito de conjunto suporte, considere um exemplo, onde temos dois nós V_i e V_j, com domínios $\{1, 2, 3, 5\}$ e $\{2, 3, 5\}$, respectivamente. Seja b um valor do domínio de V_i. O valor b somente é considerado um valor viável para V_i se tiver um mínimo de suporte dos valores dos domínios

AC-5 é baseado no AC-4, mas é apresentado de uma forma mais genérica. AC-4 tem complexidade de tempo $O(end^2)$ e AC-5 $O(ed)$. AC-5 foi criado para tratar restrições funcionais e monotônicas. O algoritmo AC-6 é uma variante de AC-4, que busca ter menor complexidade de espaço de busca ($O(ed)$), enquanto AC-4 tem complexidade de espaço $O(ed^2)$.

O algoritmo AC-7 utiliza a noção de bidirecionalidade para restrições. Com isto, ele diminui o número de comparações que devem ser realizadas, cuja complexidade de tempo é $O(ed^2)$ no pior caso e de espaço é $O(ed)$.

O algoritmo AC2001/3.1 [13] é baseado no AC-3, mas possui complexidade de tempo e espaço menores que AC-3. A complexidade de tempo no pior caso é $O(ed^2)$ e a complexidade de espaço é $O(ed)$.

O algoritmo AC-5 é utilizado nesta tese devido à sua complexidade de tempo ser mais baixa, por ser mais adequado para tratar restrições funcionais e monotônicas e estar implementado no sistema PCSOS [61, 62], utilizado em nossos experimentos.

Pesquisas nesta área de algoritmos de consistência de arcos continuam a ser realizadas na tentativa de conseguir algoritmos de complexidades menores. Para a característica de cada problema um algoritmo pode se adaptar melhor do que outro. Esta análise deve ser realizada antes de se implementar o algoritmo de consistência de arcos.

Assim, podemos perceber que os algoritmos de consistência de arcos existentes na literatura foram surgindo com os objetivos de diminuir a complexidade de espaço de busca, por causa da limitação de *hardware* para atender à característica de uma determinada aplicação ou facilitar a sua implementação. Estes algoritmos não têm como principal objetivo dividir a massa de dados a ser executada. Portanto, é de responsabilidade do programador determinar se o algoritmo será aplicado a todas as restrições ou somente a um subgrupo.

Na fase de propagação de um algoritmo de consistência de arcos, se a aplicação for observada, é possível evitar que determinadas restrições sejam executadas desnecessariamente. Isto é possível observando-se a existência de dependências entre as restrições da aplicação. Desta forma, é possível diminuir a quantidade de computação exigida pela aplicação, diminuindo-se o tempo de execução do programa.

Realizando-se um particionamento é possível rearranjar a entrada de dados para que se obtenha diminuição no tempo de execução numa execução seqüencial. Na execução distribuída o objetivo é diminuir o tempo de execução e o número de mensagens trocadas entre processadores. Este fato nos motiva a investigar alguns métodos de particionamento de restrições, que estão abordados no próximo capítulo.

---

de cada um dos outros nós V_j. Considerando a restrição $i \neq j$, o conjunto suporte para $j = 1$ é igual a $S_{V_{j1}}$=$\{(1,2),(1,3),(1,5)\}$.

Capítulo 3

Particionamento

Conforme abordado no Capítulo 2, CSPs podem se beneficiar significativamente, reduzindo o seu tempo de execução, se utilizarmos métodos de particionamento de restrições. Como as restrições podem ser representadas na forma de grafos, pode-se aplicar mecanismos utilizados em outras áreas de pesquisa diferentes de programação lógica com restrições, como por exemplo Teoria dos Grafos, para resolver este problema. Como os métodos de particionamento de grafos de restrições existentes são muito complexos para serem aplicados a CSP, buscamos métodos genéricos que facilitem o particionamento de um CSP.

Na Seção 3.1 apresentamos a definição de particionamento de um CSP. As seções seguintes apresentam a definição do problema de particionamento para execução seqüencial e paralela. Na Seção 3.4 são apresentados os principais trabalhos relacionados.

3.1 Definição de Particionamento de um CSP

Os problemas de satisfação de restrições podem gerar grande quantidade de computação, cujo tempo de execução pode tornar inviável a solução dos problemas.

Uma alternativa para resolver este problema é aproveitar as características das aplicações e particionar um de seus componentes: o conjunto de domínios, de variáveis ou de restrições. Ao realizar o particionamento, subgrupos independentes podem ser executados separadamente em diferentes processos numa mesma máquina ou em diferentes máquinas.

Na literatura, encontramos trabalhos de utilização do particionamento de domínios para realizar pré-processamento de programas com restrições. Müller [50] define o particionamento de domínio como problema de particionamento de conjuntos.

Um problema de particionamento de conjuntos (*Set Partitioning Problem* - SPP) pode ser definido da seguinte forma. Um conjunto *ground* corresponde ao domínio do problema. Dado um conjunto *ground* finito G e um conjunto P de n subconjuntos X_j, com

custo C_j, encontrar a partição de custo mínimo de G, ou seja, subconjuntos de P onde todos os elementos são disjuntos e a união dos elementos é G.

O objetivo de um pré-processamento é reduzir o tamanho do SPP, ou seja, reduzir o número de subconjuntos e cardinalidade do conjunto *ground*.

Existem várias formas de pré-processamento. Como exemplo, podemos citar: detecção de conjuntos múltiplos, detecção de subconjuntos *subsumed*, análise clique e análise de domínio [50].

A técnica de detecção de conjuntos múltiplos busca encontrar conjuntos iguais e manter o conjunto com custo mínimo. A complexidade de um algoritmo de comparação par a par é $O(n^2)$. Para uma comparação com subconjuntos ordenados a complexidade é $O(nlogn)$, onde n é o número de subconjuntos.

Na técnica de detecção de subconjuntos *subsumed*, um subconjunto X_j pode ser descartado de um SPP se ele pode ser particionado por outros subconjuntos com menor custo que C_j. O algoritmo trabalha sobre subconjuntos de um conjunto *ground*. Cada subconjunto possui um custo associado. Os subconjuntos com seus respectivos custos são ordenados pelo custo de forma descendente. O algoritmo utiliza heurísticas gulosas para manter o esforço computacional baixo. A complexidade do algoritmo é $O(n^2)$ [50].

Na técnica de análise clique um grafo é derivado de um SPP, onde os nós do grafo correspondem a subconjuntos e dois nós que compartilham no mínimo um elemento são conectados. Um clique C_e em tal grafo é o conjunto de todos os nós contendo um certo elemento e do conjunto *ground*. Uma propriedade de clique é que somente um membro dele pode ser parte da solução e todos os outros precisam ser descartados. Se for encontrado um clique C que apropriadamente *subsumes* C_e, então, todos os nós contidos somente em C podem ser descartados. Isso porque eles são atendidos pela regra do nó selecionado de C_e. A complexidade deste algoritmo é $O(\#G * n^2)$, onde $\#G$ é o número de elementos de G e n é o número de subconjuntos.

A técnica de análise de domínio tem como objetivo principal reduzir a cardinalidade do conjunto *ground*. A análise de domínio visa encontrar um elemento k do conjunto *ground* que ocorre somente naqueles subconjuntos em que também o elemento l está contido. O elemento k é dominado por l e pode remover k do conjunto *ground* e todos os subconjuntos, já que todos os subconjuntos que poderiam ser descartados pela presença de k já teriam sido descartados devido a l. A complexidade do algoritmo é $O(\#G^2 * n)$.

Na literatura existem alguns trabalhos que realizam o particionamento manual das restrições [62, 57].

Um CSP pode ser modelado na forma de um grafo onde, por exemplo, os nós podem representar as variáveis e as arestas entre os nós podem representar as restrições. Através desta representação torna-se possível aplicar técnicas de outras áreas de pesquisa, dife-

rente de programação lógica com restrições, como por exemplo, Teoria dos Grafos. Na Seção 3.4.1 os algoritmos de particionamento existentes nesta área são discutidos.

O particionamento do conjunto de restrições de um CSP permite obter melhores resultados tanto para a execução seqüencial quanto para a execução paralela, como mostrado em trabalho anterior [56].

O grafo de restrições de um CSP pode ser visto de diferentes formas, que são apresentadas a seguir.

3.1.1 Representação do grafo de restrições

O grafo de restrições tradicional, em CSP, representa as variáveis nos nós e as arestas são as restrições entre as variáveis. Este tipo de grafo é usado em geral com algoritmos de consistência de arcos. Nesta representação, a dependência entre os nós (arestas) são restrições. A Figura 3.1 exemplifica este tipo de grafo para um CSP com 4 variáveis (V_1, V_2, V_3, V_4) e 3 restrições ($V_1 = V_2 + 1$, $V_1 = V_3 + 2$ e $V_1 = V_4 + 3$).

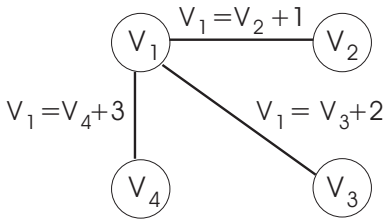


Figura 3.1: Grafo de restrições tradicional

Neste trabalho, o objetivo é realizar o particionamento estático do grafo de restrições para dividi-lo entre processos e realizar execução seqüencial ou paralela. Observando o grafo de restrições tradicional, a primeira idéia de particionamento é considerando os nós, ou seja, dividir o conjunto de nós em grupos. Sendo assim, cada variável é colocada em um grupo diferente. Mas as restrições (que estão nas arestas) devem ser executadas para que haja poda dos domínios das variáveis. Uma massa de dados grande gera um grafo de restrições que pode ter um número de arestas muito maior do que o número de nós. Portanto, é necessário decidir em que grupos as restrições que envolvem mais de uma variável devem ficar. Formar grupos de nós com as restrições relacionadas pode não gerar bons resultados. Por isso, surgiu a idéia de se representar o grafo de restrições de formas diferentes da tradicional em busca de melhores resultados. Foram estudadas duas novas representações para o grafo de restrições.

Uma segunda forma de representar as restrições considera as restrições como nós e as arestas são as variáveis comuns entre os nós. Com esta representação, a dependência entre os nós são as variáveis comuns entre eles. Considerando o mesmo exemplo da Figura 3.1, a nova representação do grafo de restrições é apresentada na Figura 3.2. Note

que, para este exemplo, as 3 restrições possuem a mesma variável em comum ($V1$). Com esta representação o particionamento das restrições pode utilizar como fator para agrupar restrições as variáveis comuns entre as restrições.

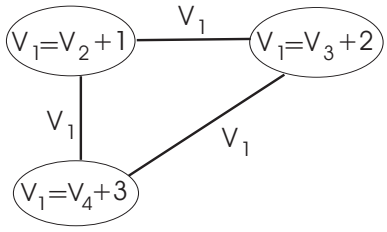


Figura 3.2: Grafo de restrições para particionamento

Como mencionado no Capítulo 2, estamos utilizando o esquema de *indexicals* para representar restrições, onde as restrições são pré-processadas e uma restrição pode estar representada por mais de um *indexical*. Com o esquema de *indexicals*, a granulosidade das restrições torna-se menor, permitindo agrupar *indexicals* ao invés de restrições. Com isso, podemos representar o grafo de restrições de uma terceira forma. Considerando o mesmo exemplo da Figura 3.1, os *indexicals* são os seguintes:

$I_1 = V_1\ in\ min\,(V_2) + 1\ \ ..\ max\,(V_2) + 1$
$I_2 = V_2\ in\ min\,(V_1) - 1\ \ ..\ max\,(V_1) - 1$
$I_3 = V_1\ in\ min\,(V_3) + 2\ \ ..\ max\,(V_3) + 2$
$I_4 = V_3\ in\ min\,(V_1) - 2\ \ ..\ max\,(V_1) - 2$
$I_5 = V_1\ in\ min\,(V_4) + 3\ \ ..\ max\,(V_4) + 3$
$I_6 = V_4\ in\ min\,(V_1) - 3\ \ ..\ max\,(V_1) - 3$

O grafo de restrições em função de *indexicals* é apresentado na Figura 3.3. Pode-se perceber que é um grafo totalmente conectado, pois há uma variável comum entre todos os *indexicals*, a variável V_1, que está representada nas arestas. Com esta representação o agrupamento dos *indexicals* é realizado considerando as variáveis comuns entre os *indexicals*.

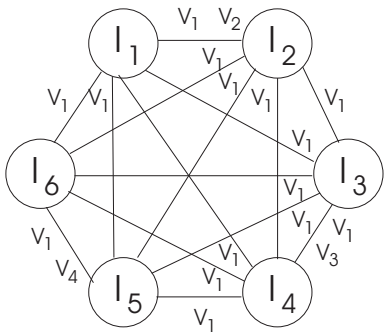


Figura 3.3: Grafo de *indexicals* para particionamento

Nas Seções 3.2 e 3.3 discutimos sobre particionamento para execução seqüencial e particionamento para execução paralela. Além disso, são abordados os trabalhos relacionados a cada um dos problemas.

3.2 Particionamento de CSP para Execução Seqüencial

Em nosso contexto, particionar restrições visa obter subconjuntos de restrições que compartilhem um número mínimo de variáveis. Isto é similar a encontrar subgrafos totalmente conectados (cliques), onde os nós representam as restrições e as arestas as dependências entre restrições. O problema seqüencial pode ser definido da seguinte forma.

Considere um grafo $G = (C, E)$, onde C é o conjunto de nós que representam as restrições e E é o conjunto de arestas entre os nós. Cada aresta representa um conjunto de variáveis comum entre duas restrições. Queremos particionar C em grupos, dividindo o grafo G em cliques.

No particionamento para execução seqüencial estamos interessados em que cada grupo possa trabalhar localmente no seu conjunto de restrições, sem propagação desnecessária de valores de variáveis de um grupo para o outro.

Se as restrições estiverem definidas em função de *indexicals* o grafo terá uma representação como a apresentada na Figura 3.3. Então, a definição de particionamento para execução seqüencial pode ser expressa da seguinte forma.

Considere um grafo $G = (I, E)$, onde I é o conjunto de nós que representam os *indexicals* e E é o conjunto de arestas entre os nós. Cada aresta representa um conjunto de variáveis comum entre dois *indexicals*. Queremos particionar I em grupos, dividindo o grafo G em cliques.

Neste caso, *indexicals*, que representam uma restrição, podem ser atribuídos a conjuntos diferentes. A granulosidade do grafo de *indexicals* é mais fina do que a granulosidade do grafo de restrições.

Na literatura existem alguns métodos de particionamento de problemas de satisfação de restrições para execução seqüencial [14]. Porém, quando se observa a característica da aplicação antes de particioná-la é possível diminuir a quantidade de computação a ser executada pelo algoritmo de consistência de arcos. Assim, apesar do algoritmo ter complexidade alta é possível ter ganho no tempo de execução total. Isso ocorre porque subgrupos independentes de restrições podem ser resolvidos separadamente, ou seja, o número de passos do algoritmo se reduz a uma fração da quantidade total de restrições do problema, com um fator aditivo e não multiplicativo de complexidade.

3.3 Particionamento de CSP para Execução Paralela

O projeto de um modelo de execução paralelo no contexto de CSP depende da política de escalonamento de restrições, isto é, da atribuição de restrições aos processadores. O escalonamento pode ser feito estaticamente ou dinamicamente. No nosso contexto, utilizamos o particionamento estático.

Uma política de escalonamento estático é apropriada para um sistema de memória distribuída porque reduz a comunicação. Neste tipo de escalonamento as restrições são distribuídas entre processadores em tempo de compilação. Na política de escalonamento dinâmico o particionamento das restrições é feito em tempo de execução. Este particionamento pode não ser apropriado para uma arquitetura de memória distribuída, porque a quantidade de trabalho para cada processador pode ser menor do que o custo de comunicação.

O problema de particionamento para execução paralela é definido da seguinte forma.

Considere um grafo $G = (C, E)$, onde C é o conjunto de restrições e E é o conjunto de variáveis comum entre duas restrições. Considere, também $p > 0$, onde p é o número de processadores. Queremos particionar C em grupos, dividindo o grafo G em p cliques.

Se as restrições estiverem definidas em função de *indexicals* o problema é definido da seguinte forma.

Considere um grafo $G = (I, E)$, onde I é o conjunto de restrições e E é o conjunto de variáveis comum entre dois *indexicals*. Considere, também $p > 0$, onde p é o número de processadores. Queremos particionar I em grupos, dividindo o grafo G em p cliques.

3.4 Trabalhos relacionados

Existem vários trabalhos relacionados a particionamento de restrições. A seguir são apresentados alguns dos principais trabalhos existentes na literatura.

3.4.1 Definição de Particionamento em Teoria dos Grafos

Em geral, a Teoria dos Grafos aborda os problemas da computação considerando os cálculos descritos como grafos nos quais os vértices representam computação e as arestas refletem as dependências de dados.

Antes de descrevermos o problema de particionamento em grafos, estabelecemos a notação utilizada para esta definição. Seja $G = (V, E)$ um grafo formado por um conjunto de vértices V e um conjunto de arestas E. Além disso, consideremos $n = |V(G)|$ o número de vértices do grafo G e v_i um vértice do grafo.

Em Teoria dos Grafos, um problema a ser resolvido pode ser definido por uma instância e uma questão a ser respondida sobre esta instância. O problema a ser resolvido pode ser classificado como: de partição em grafos, de partição em grafos versão cardinalidade e k-cut. Cada um destes problemas está descrito a seguir:

Problema de Partição em Grafos
Instância: Grafo $G = (V, E)$, funções peso $l : V \longrightarrow N$ e $w : E \longrightarrow N$ e inteiros L e W.
Questão: Existe uma partição de V em conjuntos $V_1, ..., V_k$ tal que $\sum_{v \in V_i} l(v) \leq L$ para $1 \leq i \leq k$ e tal que $\sum_{e \in E_{ij}, i \neq j} w(e) \leq W$, onde E_{ij} é o conjunto das arestas com um extremo em V_i e outro em V_j?
O problema da partição em grafos é NP-completo mesmo para L fixo, $L \geq 3$ e mesmo se todos os vértices e arestas tiverem peso 1 [29].

Problema de Partição em Grafos Versão Cardinalidade
Instância: Grafo $G = (V, E)$ e inteiros L e W.
Questão: Existe uma partição de V em conjuntos $V_1, ..., V_k$ tal que $\mid V_i \mid \leq L$ para $1 \leq i \leq k$ e $\sum_{1 \leq i < j \leq k} \mid E_{ij} \mid \leq W$?
Quando $k = 2$ e $L = \mid V \mid /2$ o problema continua NP-completo e é denominado Problema da Bisseção. Este problema foi provado ser NP-completo por Garey, Johnson e Stockmeyer [29], em 1976.

Problema k-cut
Instância: Grafo $G = (V, E)$, $w : E \longrightarrow N$ e inteiro W.
Questão: Existe uma partição de V em exatamente k conjuntos tal que $sum_{e \in E_{ij}} w(e) \leq W$?
O problema do k-cut é polinomial [32]. Porém, se k faz parte da entrada, ele é NP-completo. Este problema é apresentado em [32].

Assim, pode-se realizar o particionamento de massas de dados que são representadas através do particionamento de grafos, buscando obter subgrafos com o mesmo número de vértices.

A diferença deste problema para aquele apresentado na Seção 3.3 é que $k - cut$ visa minimizar o peso das arestas entre as k componentes sem considerar as características das dependências entre os nós. Ao encontrar cliques em um grafo, o agrupamento dos nós ocorre devido às dependências existentes entre eles.

Os métodos de particionamento de grafos encontrados na literatura buscam particionar os vértices em subconjuntos de potências de 2, para facilitar o momento da distribuição de

tarefas aos processadores, no caso do particionamento para execução paralela. Eles buscam diminuir o número de arestas entre os subconjuntos (que representa a comunicação entre os processadores após distribuir as tarefas), visando minimizar a comunicação.

Modelando estes problemas em grafos, temos que o conjunto de vértices de um grafo pode ser particionado em 2, 4, 8 ou mais subconjuntos. Existem alguns métodos para fazer esta divisão. Entre eles, podemos citar os métodos de particionamento simples e os métodos espectrais de bisseção, quadrisseção e octosseção [39].

Na literatura foram implementados algoritmos de particionamento simples e espectrais. Entre os métodos de particionamento simples, podemos destacar o método linear, o aleatório e o *scattered* [39].

No método linear os vértices são atribuídos, em ordem, aos processadores de acordo com sua numeração no grafo original. Para um grafo com n vértices sendo divididos em p conjuntos, os primeiros n/p vértices são atribuídos ao conjunto 0, os próximos n/p ao conjunto 1 e assim por diante. Este método produz bons resultados devido à localidade de dados estar implícita na numeração dos vértices.

No método aleatório, os vértices são atribuídos aleatoriamente aos conjuntos de forma que preserve o balanceamento. Enquanto que no método *scattered*, os vértices são tratados em ordem e o próximo vértice é alocado no menor conjunto.

O problema de bisseção pode ser considerado como um caso particular do problema de partição de grafos e é definido da seguinte forma.

Dado um grafo G, queremos particionar o conjunto de vértices $V = V_1 \cup V_2$, $V_1 \cap V_2 = \emptyset$, tal que o número de arestas entre as duas partições seja minimizado. $|V_1| = |V_2|$, se n é par e $|V_1| = |V_2| - 1$, se n é ímpar.

O método de bisseção consiste em particionar um grafo em dois subgrafos, com o mesmo número de vértices, utilizando uma determinada estratégia. A execução recursiva deste particionamento é denominada bisseção recursiva e consiste em dividir o grafo em dois subgrafos, depois dividir cada um dos subgrafos em dois, recursivamente, utilizando a mesma estratégia anterior. Desta forma, a partição de uma grafo em $p = 2^k$ subgrafos é obtida após a execução de k passos de particionamento recursivos.

Na literatura são apresentados alguns algoritmos de bisseção que vão desde os algoritmos mais intuitivos, como o algoritmo de bisseção coordenada recursiva, até os mais elaborados, como os algoritmos de bisseção, quadrisseção e octosseção espectral [54, 55, 65, 19, 30, 37].

O algoritmo de Bisseção Coordenada Recursiva utiliza sempre as coordenadas bi ou tri-dimensionais disponíveis para os vértices do grafo. Este algoritmo busca determinar a direção da coordenada da mais longa expansão do conjunto, para a partir desta informação realizar o particionamento [65].

O algoritmo de Bisseção Grafo Recursiva utiliza o grafo de distâncias entre vértices para realizar o particionamento [65].

Todo método espectral é baseado na determinação de autovalores a partir da matriz Laplaciana associada ao grafo [37].

Para melhorar as partições obtidas com um determinado método de particionamento são definidos os algoritmos de refinamento de partições, tais como Lanczos e Kernighan-Lin [37, 30].

Uma descrição mais detalhada destes algoritmos encontra-se no Apêndice B.

3.4.1.1 Discussão sobre Particionamento em Teoria dos Grafos

No problema de particionamento de grafo, o grafo considerado possui vértices que representam unidades de computação e as arestas são as dependências de dados.

Uma vez tendo o modelo de grafo de uma computação, o particionamento do grafo pode ser usado para determinar como dividir o trabalho e os dados para uma computação paralela eficiente. O objetivo é distribuir as computações sobre p processadores particionando os vértices em p conjuntos com igual peso, para minimizar a comunicação entre processadores, que é representada por arestas entre os conjuntos.

Os métodos de bisseção, quadrisseção e octosseção apresentam alguns problemas que são discutidos nos trabalhos [36, 35, 19]. Estes problemas são apresentados a seguir.

A utilização da métrica de corte de arestas para medir o volume de comunicação existente entre as partições não é uma boa alternativa, pois a quantidade de comunicação requerida por uma aplicação não é proporcional ao número de arestas cortadas no grafo pelo particionamento. Além disso, o volume de comunicação é no máximo uma previsão fraca do custo de comunicação.

O problema com a métrica de corte de arestas é que diversas arestas podem descrever a mesma necessidade de transferência de dados. Assim, reduzir o corte de arestas pode não implicar necessariamente na redução do volume real de comunicação.

Outro efeito que não é considerado pelos métodos de particionamento de grafos é o efeito do congestionamento de comunicação. Tipicamente, em computações científicas, muitas mensagens estão simultaneamente competindo por recursos da rede. O padrão de comunicação coletivo pode ter um impacto significativo no custo de comunicação. Os algoritmos de particionamento de grafos existentes falham neste aspecto.

As ferramentas de particionamento existentes atualmente tentam otimizar um objetivo que é distante do custo verdadeiro provocado por uma computação paralela. Estas ferramentas permitem que instâncias muito grandes de aplicações científicas sejam realizadas com sucesso, pois a grande maioria das aplicações que utiliza algoritmos de particionamento de grafos vem da área de equações diferenciais e o grafo corresponde a uma malha.

Estas características implicam em três fatores principais. Primeiro, as propriedades geométricas das malhas asseguram que boas partições existam. Se uma malha em d dimensões tem n vértices, ela tem separadores da ordem de $n^{(d-1)/d}$. Isto garante que para problemas suficientemente grandes, a taxa de comunicação para computação será pequena. Então, qualquer particionamento razoável permitirá um bom desempenho paralelo. O segundo fator que podemos observar é que os vértices associados a malhas computacionais tipicamente têm um número de vizinhos limitado. Isto limita o dano causado pela aproximação pelo corte de arestas. Finalmente, malhas são geralmente homogêneas, então os vários conjuntos são similares. Isto limita a comunicação possivelmente não homogênea nessas aplicações.

As matrizes provenientes de métodos de ponto interior, decomposição de valor singular para indexação de semântica latente e outras aplicações são, geralmente, muito menos estruturadas que matrizes de malhas. Em particular, o número de arestas associado com um vértice pode ser muito grande. Sem a garantia de obter boas partições, as matrizes altamente não estruturadas possuem o potencial de requerer mais comunicação do que as malhas. Assim, a qualidade da partição pode influenciar mais diretamente no aumento do tempo de execução da aplicação.

A implementação destes métodos utiliza representação matemática e apresenta complexidade exponencial. Para grafos mais complexos, onde o número de arestas é grande, como nos CSPs, este tipo de particionamento pode não gerar o melhor particionamento. Isso porque as características da aplicação não são consideradas e para grafos mais conexos a comunicação pode ser muito alta, se as arestas estiverem relacionadas à comunicação entre os processadores.

Devido a estes fatores buscamos implementar um método que seja geral para particionar grafos de restrições de um CSP de forma automática.

3.4.2 Trabalhos relacionados a execução seqüencial

No trabalho de Bliek *et al.* [14] o enfoque são os CSPs contínuos estruturados definidos por sistemas de equações. Nele são utilizadas técnicas para decompor o grafo de restrições num grafo acíclico orientado com blocos pequenos. Além disso, são apresentados algoritmos para resolver problemas decompostos, que resolvem os blocos em ordem parcial e um *backtracking* inteligente quando um bloco não tem solução.

O grupo de CSPs tratados é definido por equações não lineares e aborda o caso geral em que o sistema não é necessariamente quadrado. São utilizadas técnicas para decompor grafos de restrição com algoritmos de *backtracking* para resolver sistemas decompostos.

3.4.3 Trabalhos relacionados a execução paralela

A primeira tentativa de construir um sistema de programação com restrições num computador SIMD (*Single Instruction, Multiple Data*) maciçamente paralelo foi realizada por Tong e Leung [70]. Uma linguagem de programação com restrições concorrente denominada modelo de computação *Firebird* foi apresentada. Esta linguagem trabalha sobre restrições de domínios finitos e suporta paralelismo de dados e concorrência. A concorrência é derivada do fluxo de paralelismo-E da escolha contida em linguagens de programação lógica. A implementação SIMD suporta paralelismo-Ou mas o fluxo de paralelismo-E é implementado seqüencialmente. Num passo de derivação não determinístico, uma das variáveis do domínio é selecionada para criar um ponto de escolha e todas os valores possíveis do seu domínio são atendidos com paralelismo-Ou. O paralelismo de dados é explorado na execução do paralelismo-Ou resultante e milhares de restrições podem ser resolvidas num único passo.

Uma partição é uma seqüência de derivações independentes, que podem ser mapeadas em um número idêntico de partições físicas, para evitar movimento de dados entre processadores. Cada partição física corresponde a um processador do computador paralelo. Inicialmente todo processador executa exatamente a mesma partição lógica inicial.

Nguyen e Deville [51] apresentaram um algoritmo de consistência de arcos distribuído (DisAC-4), baseado no algoritmo AC-4. Ele é um algoritmo paralelo projetado para computadores de memória distribuída usando comunicação por passagem de mensagens. Considera o grafo de restrições tradicional e somente restrições binárias. Além disso, é feito o particionamento das variáveis (nós do grafo de restrições) e seus respectivos domínios, que são atribuídos aos processos. A complexidade de tempo do DisAC-4 é $O(\frac{n^2d^2}{k})$, onde n é o número de variáveis, d é o tamanho do maior domínio e k é o número de processadores.

Ferris e Mangasarian [24] apresentam uma técnica para paralelizar restrições em um programa matemático. As restrições do programa matemático são distribuídas entre processadores juntamente com um operador Lagrangeano apropriado para cada processador, que contém informações Lagrangeanas sobre as restrições tratadas pelos outros processadores. A informação do multiplicador Lagrangeano é, então, trocada entre os processadores.

Zervoudakis [75] implementou um *solver* CSP para domínios finitos inteiros. O *solver* provê classes para entidades CSP como variáveis, restrições e mecanismos para expressar árvores de busca em termos de conjunções e disjunções de objetivos. O *solver* foi implementado na linguagem C++, permitindo utilizar tanto o algoritmo AC-5 quanto o AC-3.

Andino *et al.* [62] utilizam particionamento de restrições. O particionamento é re-

alizado de forma estática, em tempo de compilação. Foram implementadas três formas de particionamento: *round-robin* por variáveis, *round-robin* por *indexicals* e *round-robin* por restrições. Estas formas de particionamento estão incorporadas num sistema paralelo baseado em memória distribuída. Este sistema utiliza o algoritmo de consistência de arcos sobre domínios finitos, utilizando o esquema de *indexicals*, para uma plataforma com múltiplos processadores.

O particionamento de restrições entre processadores foi realizado em nosso trabalho anterior [57, 56] observando-se as dependências. As restrições são independentes quando não possuem variáveis em comum e são dependentes quando relacionam as mesmas variáveis. A análise da aplicação com relação à dependência entre as restrições permite a realização de particionamentos que explorem melhor o paralelismo da aplicação. Os particionamentos manuais *round-robin* por restrições e particionamento por blocos por variáveis foram os métodos utilizados. No particionamento *round-robin* por restrições a primeira restrição é atribuída ao primeiro processador, a segunda restrição é atribuída ao segundo processador, e assim por diante. O particionamento por blocos por variáveis consiste em dividir o conjunto de variáveis que são atribuídas aos processadores. Então, as restrições que relacionarem o primeiro conjunto de variáveis são atribuídas ao primeiro processador, o segundo conjunto ao segundo processador e assim por diante. Desta forma, somente há comunicação entre as variáveis de ligação entre os blocos.

Existem problemas combinatoriais que já são distribuídos por natureza [46]. Considere um grande hospital composto por muitas salas. Cada sala possui um quadro de horário semanal, definindo cada turno dos enfermeiros. A construção do quadro de horários envolve resolver um CSP para cada sala. Alguns dos enfermeiros são qualificados para trabalhar na Sala de Emergência. Os regulamentos do hospital exigem que haja um certo número de enfermeiros qualificados em cada turno. Isto impõe restrições entre os quadros de horário das diferentes salas e gera um complexo CSP distribuído.

Para resolver problemas deste tipo, foram implementados vários algoritmos. A seção a seguir apresenta um resumo de alguns dos principais algoritmos.

3.4.3.1 CSP Distribuído

Um CSP distribuído é um CSP onde variáveis e restrições são distribuídas entre múltiplos agentes [1] [73, 46]. Cada agente atribui valores às suas variáveis tentando gerar consistência local de forma que seja consistente com todas as restrições. Entre os agentes há restrições. O valor atribuído à variável deve satisfazer estas restrições. A solução de um CSP distribuído envolve atribuições de todos os agentes para todas suas variáveis e

[1] Um agente computacional é algo que age para alcançar um resultado. Mas possui atributos que o difere de um simples programa, tais como operar sob controle autônomo, perceber seu ambiente, persistir por um período de tempo prolongado, adaptar-se a mudanças e ser capaz de assumir metas de outros agentes [63].

Algoritmo	Síncrono/ Assíncrono	Tipo de atribuição	Definição de agente	Tipo de ordenação	Privacidade	Avaliação
Synchronous Backtracking (SBT) [73]	síncrono	seqüencial	cada agente contém apenas uma variável	estática	não	simulação
Asynchronous Backtracking (ABT) [73]	assíncrono	concorrente	cada agente contém apenas uma variável	estática	não	simulação
Asynchronous Weak-Commitment Search [73]	assíncrono	concorrente	cada agente contém apenas uma variável	dinâmica	não	simulação
Distributed Backtracking (DIBT) [34]	assíncrono	concorrente	cada agente contém apenas uma variável	estática	não	execução
Interleaved Parallel Search (IDIBT) [33]	assíncrono	concorrente	cada agente contém apenas uma variável	estática	não	execução
Dynamic Backtracking (DB) [31]	assíncrono	concorrente	cada agente contém apenas uma variável	estática	não	simulação
Distributed Dynamic Backtracking (DisDB) [12]	assíncrono	concorrente	cada agente contém apenas uma variável	estática	não	somente propõe o algoritmo
Concurrent Backtrack Search (ConcBT) [77]	síncrono	concorrente	cada agente contém sua rede de restrições local (múltiplas variáveis)	dinâmica	não	simulação
Concurrent Dynamic Backtracking (ConcDB) [78]	assíncrono	concorrente	cada agente contém sua rede de restrições local (múltiplas variáveis)	dinâmica	não	simulação
Asynchronous Backtracking with Dynamic Ordering (ABT_DO) [80]	assíncrono	concorrente	cada agente contém apenas uma variável	dinâmica	não	simulação
Asynchronous Forward-Checking (AFC) [48]	atribuição síncrona e forward-checking assíncrono	concorrente	cada agente contém sua rede de restrições local (múltiplas variáveis)	dinâmica	não	simulação
Distributed Forward-Checking with Dynamic Ordering (DODFC) [47]	assíncrono	concorrente	cada agente contém sua rede de restrições local (múltiplas variáveis)	dinâmica	pode ser adaptado	simulação
Distributed Forward Checking with Partially Known Constraints (DFC-PKC) [15, 16]	assíncrono	concorrente	cada agente contém apenas uma variável	estática	sim	execução
Asynchronous Backtracking for Asymetric Constraints (ABT-ASC) [79]	assíncrono	concorrente	cada agente contém apenas uma variável	estática	sim	simulação
Asynchronous Aggregation Search (AAS) [64]	assíncrono	concorrente	cada agente contém apenas uma variável	estática	restrições privadas	simulação

Tabela 3.1: Principais algoritmos para resolver CSP distribuído

trocas de informação entre todos os agentes, para verificar a consistência de atribuições com restrições entre agentes.

O modelo de comunicação utilizado para resolver um CSP distribuído consiste em agentes que se comunicam pelo envio de mensagens. O atraso no envio de uma mensagem é finito. As mensagens recebidas numa comunicação entre dois agentes são recebidas na ordem que elas foram enviadas.

Luo *et al.* [42] apresentam em seu trabalho algumas heurísticas de métodos de busca utilizadas para melhorar as estratégias de resolução de CSPs distribuídos. Estas estratégias são denominadas baseada em variável, em domínio e em função. Eles apresentam também os algoritmos relacionados a estas estratégias. Os resultados experimentais mostraram que os métodos de heurística básica para problemas de satisfação de restrições podem ser aplicados, em algoritmos de busca distribuídos, localmente quando os algoritmos são baseados em variável ou globalmente quando os algoritmos são baseados em domínio ou baseados em função.

A Tabela 3.1 apresenta um resumo dos principais algoritmos para resolver CSPs distribuídos. Nesta tabela os algoritmos são classificados como síncrono ou assíncrono; com atribuição de valores seqüencial ou concorrente; qual a definição de agente utilizada; o tipo de ordenação de agentes utilizado; se o algoritmo considera a propriedade de privacidade dos agentes e se os resultados obtidos foram simulados ou obtidos de execução.

Percebemos que a maioria dos algoritmos são baseados no algoritmo *Asynchronous Backtracking* [73]. O que mais varia é o tipo de ordenação das variáveis. A maioria dos resultados obtidos foram através de simulação.

Os algoritmos que constam na Tabela 3.1 são apresentados com mais detalhes no Apêndice C.

3.5 Considerações finais

Estes trabalhos se diferenciam da nossa proposta em alguns aspectos: (1) muitos se concentram em particionamento de domínios ou variáveis [70, 51, 24], enquanto nos concentramos em particionamento de restrições; (2) os trabalhos que se concentram em particionamento de restrições ou o fazem de forma manual [62, 57, 58] ou não consideram a característica das aplicações no que se refere às dependências entre as restrições; (3) têm complexidade muito alta tornando a solução de instâncias maiores inviável e (4) consideram que o CSP já apresenta característica distribuída natural [71, 46, 48, 64, 67, 79, 74, 15, 47, 80, 12, 77, 11, 76, 31, 78, 16, 60], não necessitando realizar um particionamento antes da execução.

Os CSPs distribuídos apresentam a característica de já possuírem naturalmente partições e para execução dos algoritmos basta usar estas informações.

Um CSP seqüencial precisa ser particionado para ser executado de forma paralela ou distribuída. Por isso, é importante haver um algoritmo que possa ser aplicado a qualquer problema.

Outro método que pode ser utilizado para particionamento de um grafo conexo é o Escalonamento por Reversão de Arestas (*Scheduling by Edge Reversal* - SER) [28, 7, 4, 5]. Este mecanismo consiste em manipular orientações acíclicas de um grafo conexo, onde o nó representa o processo e as arestas entre os dois nós representam que os nós compartilham um recurso. Através deste mecanismo é possível obter blocos, considerando as características particulares das aplicações e agrupando os nós que possuam maior dependência. Este método se aplica a outros problemas com grafos mais conectados que os grafos de malhas. Há indicativos de que estes problemas mencionados anteriormente possam ser contornados mais facilmente por este método do que pelos métodos de particionamento de Teoria dos Grafos [59]. Este método possui a desvantagem de possuir um tempo de execução muito alto quando o número de restrições é muito grande. Por isso, foi desenvolvido o método *Grouping-Sink*, baseado no SER, que produz os mesmos particionamentos que o SER, mas com tempo de execução menor. O particionamento *Grouping-Sink* é detalhado no Capítulo 4.

Neste trabalho, o particionamento obtido a partir da decomposição em *sinks* é comparado com outros métodos de particionamento manuais, que são apresentados a seguir.

3.5.1 Tipos de particionamento usados

Um CSP possui três elementos principais em sua modelagem na forma de grafo: variáveis, restrições e domínios. No nosso contexto, as restrições são representadas por *indexicals*. O tamanho do domínio inicial é igual para todas as variáveis. Como estamos interessados em realizar um particionamento estático antes de iniciar a execução, o particionamento dos domínios não é um método muito interessante, pois os domínios são alterados durante a execução. Assim, nos concentramos em particionar três elementos: variáveis, restrições e *indexicals*. Após escolher o elemento que será particionado é necessário definir como distribuí-lo entre os processadores. Estamos nos concentrando em três métodos principais para distribuir o elemento escolhido entre os processadores disponíveis: *Round-Robin*, Blocos e o algoritmo proposto neste trabalho, denominado *Grouping-Sink*.

Round-Robin Consiste em atribuir os elementos aos processadores, em ordem seqüencial do arquivo de entrada. Portanto, se o elemento escolhido for variáveis, então, a primeira variável é atribuída ao primeiro processador, a segunda variável ao segundo processador e assim por diante. Os *indexicals* associados às variáveis são colocados no processador correspondente. No caso de se particionar restrições, ele consiste em atribuir

a primeira restrição (que pode ser representada por mais de um *indexical*) ao primeiro processador, a segunda restrição ao segundo processador e assim por diante. Para *indexicals* o raciocínio é o mesmo: o primeiro *indexical* é alocado ao primeiro processador, o segundo *indexical* ao segundo processador e assim por diante.

Blocos Consiste em dividir a quantidade do elemento escolhido pelo número de processadores disponíveis para execução. Desta forma, são definidos blocos de variáveis, de restrições ou de *indexicals*. Este método agrupa os elementos de forma seqüencial. Então, se o elemento escolhido for variáveis, consiste em dividir o número de variáveis da aplicação pelo número de processadores, calculando assim, a quantidade de variáveis que devem ser atribuídas aos processadores (blocos). Se a divisão não for exata, os elementos do resto da divisão devem ser redistribuídos entre os processadores aumentando em uma unidade o tamanho do bloco. Por exemplo, suponha 126 variáveis para serem distribuídas entre 8 processadores, o resultado são blocos de 15 variáveis e restando 6 variáveis. Essas devem ser alocadas a algum processador. Então, teremos 6 processadores com 16 variáveis seqüencias (como por exemplo, o processador 0 recebe as variáveis de V0 a V15) e 2 processadores recebem 15 variáveis seqüenciais. Quando o elemento a ser particionado for restrições o método consiste em dividir o número de restrições da aplicação pelo número de processadores disponíveis. Em caso de divisão inexata usa-se o mesmo raciocínio utilizado para a distribuição das variáveis.

Algoritmo *Grouping-Sink* Consiste em utilizar um algoritmo que agrupe as restrições ou *indexicals* utilizando as dependências do grafo de restrições e as informações adicionais da aplicação. Estamos utilizando os algoritmos *Grouping-Sink* por restrições e *Grouping-Sink* por *indexicals*. Este algoritmo está melhor descrito no Capítulo 4.

Desta forma, temos 8 formas de particionar um CSP, que esteja representado na forma de *indexicals*.

Para ilustrar os 8 tipos de particionamento, considere o problema *Queens* para 4 variáveis. Este problema apresenta 22 restrições e 40 *indexicals*. A Tabela 3.2 apresenta as restrições, os *indexicals* relacionados e os números atribuídos aos *indexicals*. Os *indexicals* de 0 a 3 representam as variáveis relacionadas a seus domínios. Os demais *indexicals* representam as restrições entre as variáveis, explicitando cada variável em função de máximos e mínimos [20]. Podemos notar, para este exemplo, que as restrições que relacionam as variáveis a seus domínios são idênticas aos *indexicals*, mas cada restrição que envolve duas variáveis resulta em dois *indexicals*.

As Figuras de 3.5 a 3.12 mostram como ficaram os *indexicals* nos 4 processadores nos 8 particionamentos. Os *indexicals* estão representados por seus respectivos números. A Figura 3.4 apresenta a legenda para todas as figuras de particionamento para *Queens* com

Número do *indexical*	*Indexicals*	Restrições
0	$V_0\ in\ 1..4$	$V_0\ in\ 1..4$
1	$V_1\ in\ 1..4$	$V_1\ in\ 1..4$
2	$V_2\ in\ 1..4$	$V_2\ in\ 1..4$
3	$V_3\ in\ 1..4$	$V_3\ in\ 1..4$
4	$V_0\ in\ min(V_1)\ ..\ max(V_1)$	$V_0 \neq V_1$
5	$V_1\ in\ min(V_0)\ ..\ max(V_0)$	
6	$V_0\ in\ min(V_1) - 1\ ..\ max(V_1) - 1$	$V_0 \neq V_1 - 1$
7	$V_1\ in\ min(V_0) + 1\ ..\ max(V_0) + 1$	
8	$V_0\ in\ min(V_1) + 1\ ..\ max(V_1) + 1$	$V_0 \neq V_1 + 1$
9	$V_1\ in\ min(V_0) - 1\ ..\ max(V_0) - 1$	
10	$V_0\ in\ min(V_2)\ ..\ max(V_2)$	$V_0 \neq V_2$
11	$V_2\ in\ min(V_0)\ ..\ max(V_0)$	
12	$V_0\ in\ min(V_2) - 2\ ..\ max(V_2) - 2$	$V_0 \neq V_2 - 2$
13	$V_2\ in\ min(V_0) + 2\ ..\ max(V_0) + 2$	
14	$V_0\ in\ min(V_2) + 2\ ..\ max(V_2) + 2$	$V_0 \neq V_2 + 2$
15	$V_2\ in\ min(V_0) - 2\ ..\ max(V_0) - 2$	
16	$V_0\ in\ min(V_3)\ ..\ max(V_3)$	$V_0 \neq V_3$
17	$V_3\ in\ min(V_0)\ ..\ max(V_0)$	
18	$V_0\ in\ min(V_3) - 3\ ..\ max(V_3) - 3$	$V_0 \neq V_3 - 3$
19	$V_3\ in\ min(V_0) + 3\ ..\ max(V_0) + 3$	
20	$V_0\ in\ min(V_3) + 3\ ..\ max(V_3) + 3$	$V_0 \neq V_3 + 3$
21	$V_3\ in\ min(V_0) - 3\ ..\ max(V_0) - 3$	
22	$V_1\ in\ min(V_2)\ ..\ max(V_2)$	$V_1 \neq V_2$
23	$V_2\ in\ min(V_1)\ ..\ max(V_1)$	
24	$V_1\ in\ min(V_2) - 1\ ..\ max(V_2) - 1$	$V_1 \neq V_2 - 1$
25	$V_2\ in\ min(V_1) + 1\ ..\ max(V_1) + 1$	
26	$V_1\ in\ min(V_2) + 1\ ..\ max(V_2) + 1$	$V_1 \neq V_2 + 1$
27	$V_2\ in\ min(V_1) - 1\ ..\ max(V_1) - 1$	
28	$V_1\ in\ min(V_3)\ ..\ max(V_3)$	$V_1 \neq V_3$
29	$V_3\ in\ min(V_1)\ ..\ max(V_1)$	
30	$V_1\ in\ min(V_3) - 2\ ..\ max(V_3) - 2$	$V_1 \neq V_3 - 2$
31	$V_3\ in\ min(V_1) + 2\ ..\ max(V_1) + 2$	
32	$V_1\ in\ min(V_3) + 2\ ..\ max(V_3) + 2$	$V_1 \neq V_3 + 2$
33	$V_3\ in\ min(V_1) - 2\ ..\ max(V_1) - 2$	
34	$V_2\ in\ min(V_3)\ ..\ max(V_3)$	$V_2 \neq V_3$
35	$V_3\ in\ min(V_2)\ ..\ max(V_2)$	
36	$V_2\ in\ min(V_3) - 1\ ..\ max(V_3) - 1$	$V_2 \neq V_3 - 1$
37	$V_3\ in\ min(V_2) + 1\ ..\ max(V_2) + 1$	
38	$V_2\ in\ min(V_3) + 1\ ..\ max(V_3) + 1$	$V_2 \neq V_3 + 1$
39	$V_3\ in\ min(V_2) - 1\ ..\ max(V_2) - 1$	

Tabela 3.2: Conjunto de *indexicals* para *Queens* com 4 variáveis

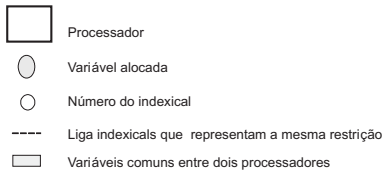


Figura 3.4: Legenda das figuras de particionamento para *Queens* com 4 variáveis

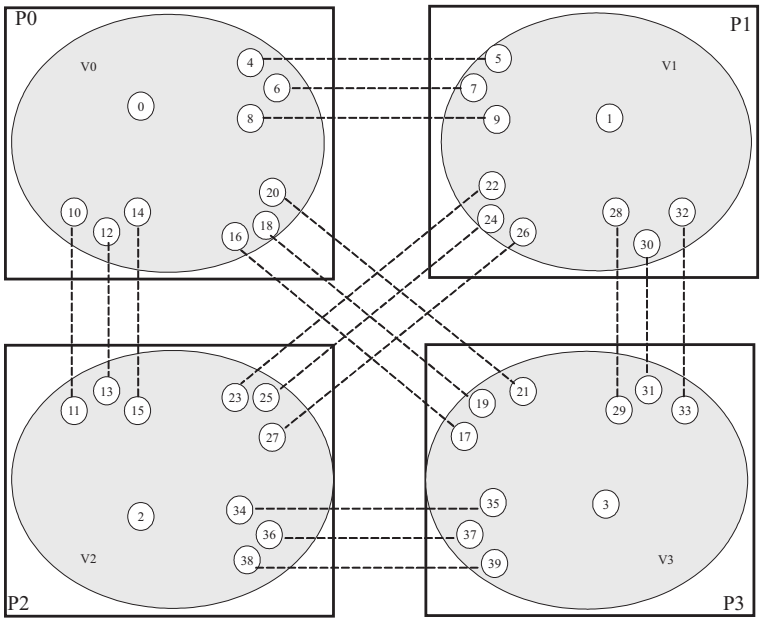


Figura 3.5: Particionamento Blocos por variáveis para *Queens* com 4 variáveis

4 variáveis. Nelas, os retângulos representam os processadores, as elipses cinzas são as variáveis e os círculos numerados são os *indexicals*. As linhas pontilhadas entre *indexicals* indicam que os *indexicals* ligados representam a mesma restrição. Os retângulos cinza entre os processadores representam dependências devido às variáveis indicadas (variáveis comuns entre os processadores).

Para realizar o particionamento Blocos por variáveis (alocação dos *indexicals* aos processadores) é realizada a divisão do número de variáveis (4) pelo número de processadores (4). Assim, é definido que será atribuída uma variável, com seus respectivos *indexicals*, a cada processador. Então, os *indexicals* relacionados a V_0 são atribuídos ao processador P0, aqueles relacionados a V_1 são atribuídos ao processador P1 e assim por diante. A Figura 3.5 ilustra como os *indexicals* ficaram alocados aos processadores.

No particionamento Blocos por *indexicals*, o número de *indexicals* (40) é dividido pelo número de processadores (4). Então, cada processador recebe 10 *indexicals* consecutivos, como ilustrado na Figura 3.6.

No particionamento Blocos por restrições, o número de restrições (22) é dividido pelo número de processadores (4). O quociente resultante é 5 e o resto 2. Então, no proces-

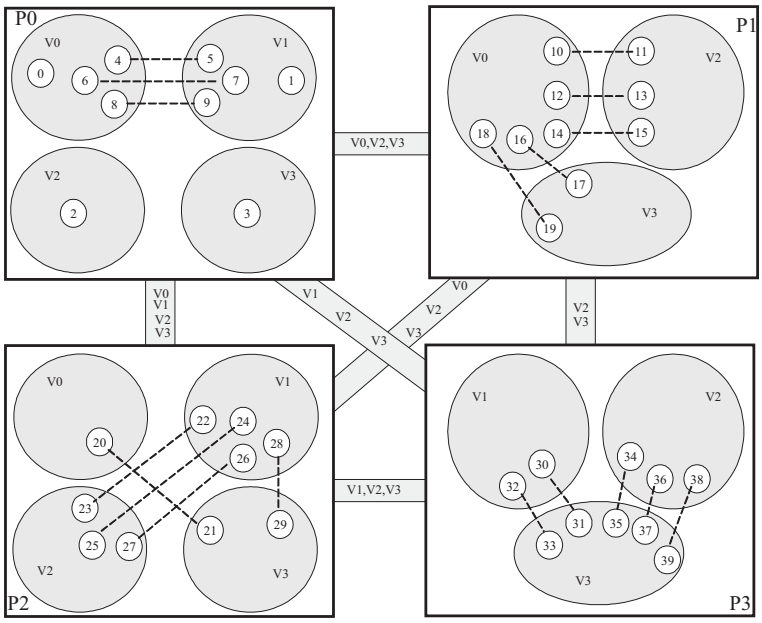


Figura 3.6: Particionamento Blocos por *indexicals* para *Queens* com 4 variáveis

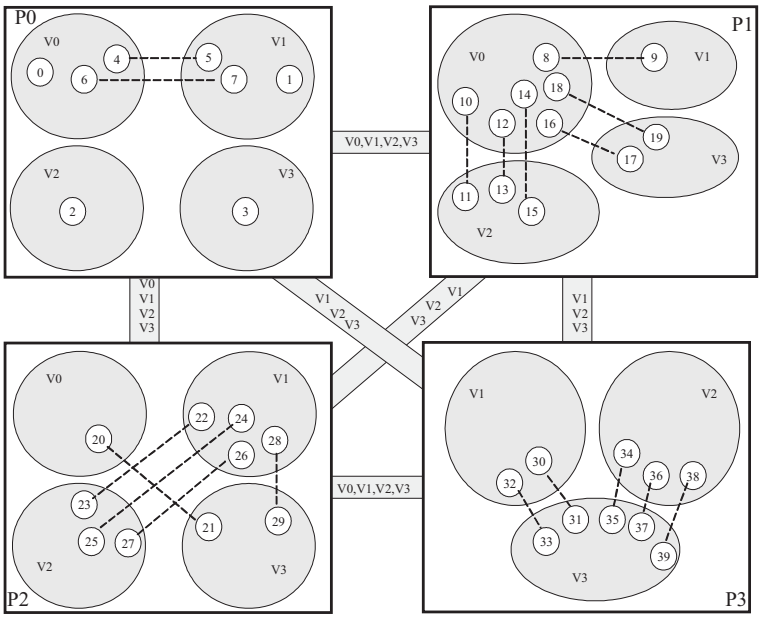


Figura 3.7: Particionamento Blocos por restrições para *Queens* com 4 variáveis

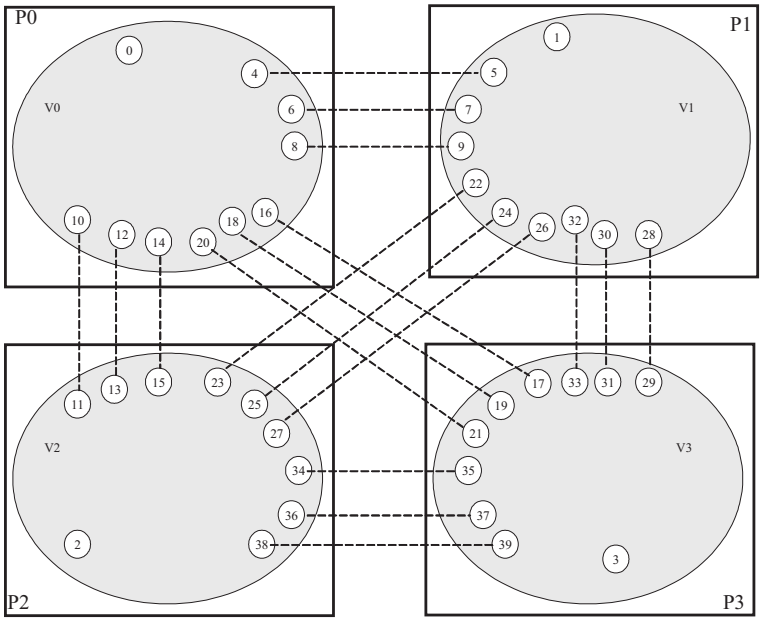


Figura 3.8: Particionamento *Round-Robin* por variáveis para *Queens* com 4 variáveis

sador P0 são alocados os *indexicals* relacionados às 6 primeiras restrições (8 *indexicals*). No processador P1 são alocadas as 6 restrições seguintes (12 *indexicals*). Em P2 são alocadas as 5 restrições seguintes (10 *indexicals*) e P3 recebe 5 restrições (10 *indexicals*). A Figura 3.7 ilustra esta alocação dos *indexicals* aos processadores.

O particionamento *Round-Robin* por variáveis consiste em atribuir os *indexicals* aos processadores de acordo com a atribuição das variáveis aos processadores. Então, a primeira variável (e seus *indexicals* relacionados) é atribuída ao processador P0, a segunda variável ao processador P1, a terceira variável a P2 e assim por diante. Se o número de variáveis for maior que o número de processadores a distribuição recomeça no primeiro processador. Este critério é também válido para *Round-Robin* por *indexicals* e por restrições. Este particionamento está ilustrado na Figura 3.8.

Round-Robin por *indexicals* consiste em atribuir o primeiro *indexical* ao processador P0, o segundo *indexical* a P1, o terceiro *indexical* a P2 e assim por diante. Este particionamento está ilustrado na Figura 3.9.

No particionamento *Round-Robin* por restrições, os *indexicals* relacionados à primeira restrição são atribuídos ao processador P0, os *indexicals* relacionados à segunda restrição são atribuídos a P1 e assim por diante. A Figura 3.10 ilustra este particionamento.

Os particionamentos *Grouping-Sink* por *indexicals* e por restrições estão ilustrados nas Figura 3.11 e 3.12. Este método é melhor explicado no Capítulo 4.

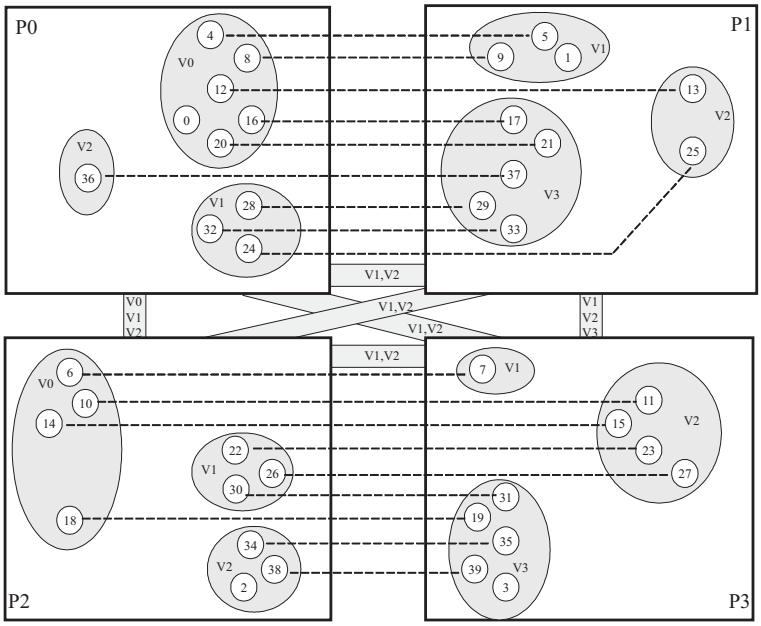


Figura 3.9: Particionamento *Round-Robin* por *indexicals* para *Queens* com 4 variáveis

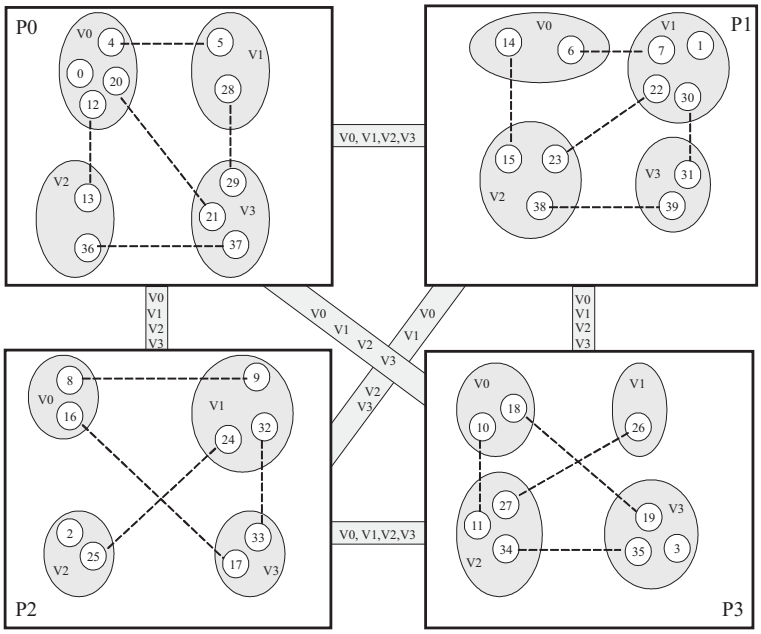


Figura 3.10: Particionamento *Round-Robin* por restrições para *Queens* com 4 variáveis

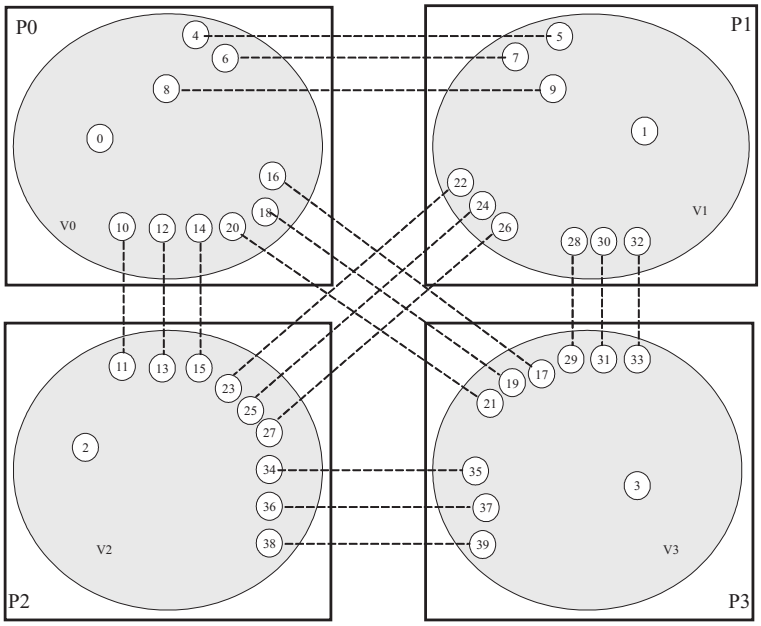


Figura 3.11: Particionamento *Grouping-Sink* por *indexicals* para *Queens* com 4 variáveis

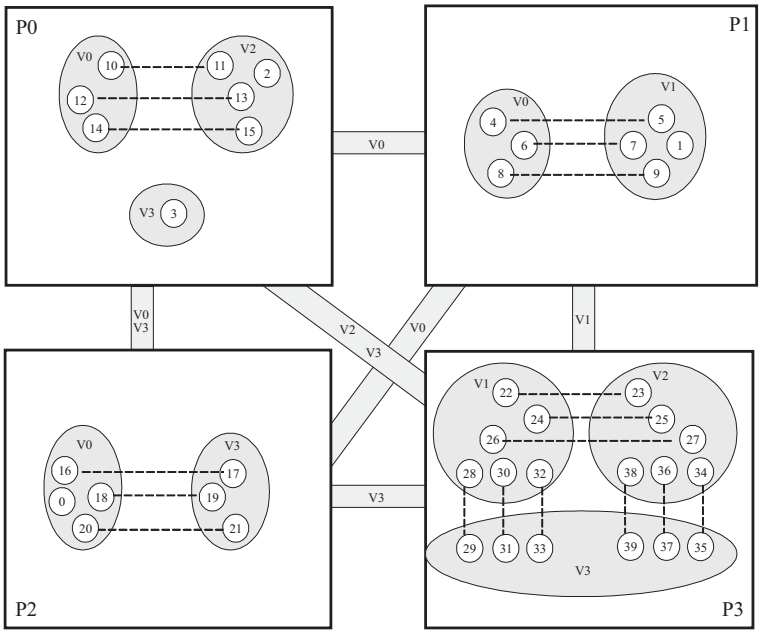


Figura 3.12: Particionamento *Grouping-Sink* por restrições para *Queens* com 4 variáveis

Capítulo 4

Particionamento Grouping-Sink

Um método de particionamento que seja geral para todas as aplicações permite que aplicações com diferentes características e tamanhos sejam particionadas de forma automática para execução paralela. O particionamento *Grouping-Sink* se originou a partir do método de Escalonamento por Reversão de Arestas (*Scheduling by Edge Reversal* - SER) [7]. SER agrupa as restrições baseando-se na dependência existente entre elas, buscando maior concorrência (menor comunicação entre os grupos de restrições).

Este capítulo aborda as diferenças entre os particionamentos gerados pelo *Grouping-Sink* e pelo SER, além de apresentar as vantagens de *Grouping-Sink* e sua implementação.

4.1 Escalonamento por Reversão de Arestas

O Escalonamento por Reversão de Arestas ou SER [7] é um mecanismo de escalonamento totalmente distribuído baseado na manipulação de orientações acíclicas de um grafo G. Cada nó representa um processo e dois nós são conectados por uma aresta se, e somente se, os processos compartilham um recurso. Um recurso é representado em G por um clique. Isto é, um subgrafo completamente conectado, já que todos os processos que compartilham o mesmo recurso estão conectados [26]. Note que um processo pode acessar um número arbitrário de recursos. Em nosso contexto, um nó corresponde a uma restrição; um recurso é uma variável (ou conjunto de variáveis) comum entre duas restrições e um nó está operando quando é escolhido um valor para uma variável.

Em qualquer momento, um nó está ocioso ou operando. Um nó está ocioso quando está esperando por um recurso e operando quando utiliza os recursos que compartilha com os outros nós. Assim, o grafo G representa um sistema chamado *neighborhood constrained*. Isto é, um sistema onde alguns processos não podem operar enquanto seus vizinhos estiverem utilizando os recursos compartilhados.

O algoritmo SER distribuído síncrono trabalha sobre um grafo G, com uma orientação acíclica e classifica os nós do grafo como *sinks* (sumidouros). Os nós *sinks* nesta orien-

tação são os nós que possuem somente arestas incidentes a eles (estão operando). Já que toda orientação acíclica tem no mínimo um *sink* não há *deadlock* ou *starvation* [7]. No próximo passo, as arestas incidentes aos *sinks* são revertidas modificando esta orientação. Este passo resulta numa nova orientação acíclica, bem como num novo conjunto de *sinks*. Então, o escalonamento pode ser visto como uma seqüência infinita de orientações.

Na Figura 4.1 é apresentado o algoritmo SER. O primeiro passo é obter uma orientação acíclica do grafo G, utilizando algum algoritmo de geração de orientação acíclica [2, 3, 17] (linha 1). No próximo passo, as arestas incidentes aos *sinks* são revertidas modificando esta orientação e resultando numa orientação acíclica, com um novo conjunto de *sinks* (linhas de 2 a 4). Então, o escalonamento pode ser visto como uma evolução no tempo de orientações acíclicas de G, gerando uma seqüência de orientações (linhas de 2 a 4).

1 Orienta grafo G
2 Enquanto o período não for encontrado
3 Reverte arestas incidentes aos sinks
4 Fim-enquanto

Figura 4.1: Algoritmo SER distribuído síncrono

Num escalonamento guloso, uma nova orientação acíclica é obtida da orientação acíclica anterior revertendo as arestas de todos os *sinks*. Sob esta hipótese, orientações repetem-se periodicamente depois de um certo tempo. Todos os nós operam o mesmo número m de vezes num período. A concorrência de um período p é definida pelo coeficiente m/p [7].

Considere os exemplos de execução do SER apresentados na Figura 4.2. No exemplo da parte superior, o primeiro estado (grafo (a)) não pertence ao período, ou seja, na terceira reversão de arestas volta-se ao segundo estado (grafo (b)) e a partir daí as reversões ficam revezando entre os grafos (b) e (c). Neste exemplo, em cada estado obtém-se um grupo de nós formado pelos *sinks*, que não possuem dependência entre si. Por isso, estes nós podem ser executados em processadores diferentes. Então, o escalonamento obtido apresenta o primeiro grupo formado pelos nós 1, 3 e 5 e o segundo grupo pelos nós 2, 4 e 6.

No exemplo da parte inferior, o primeiro estado (grafo (a)) pertence ao período. Podemos observar que o período começa do estado inicial, formando o primeiro grupo com os nós 1 e 2. Ao reverter as arestas destes dois nós *sinks* obtemos o segundo grupo com o nó 5 (grafo (b)). A reversão das arestas de 5 gera um novo grupo com o nó 3 (grafo (c)). Revertendo as arestas de 3 obtém-se o grupo com o nó 4 (grafo (d)) . Ao reverter as arestas de 4 volta-se ao estado inicial (grafo (a)), completando o período. Podemos

observar que todos os nós operaram o mesmo número de vezes (1 vez) e o número de grupos gerados foi quatro.

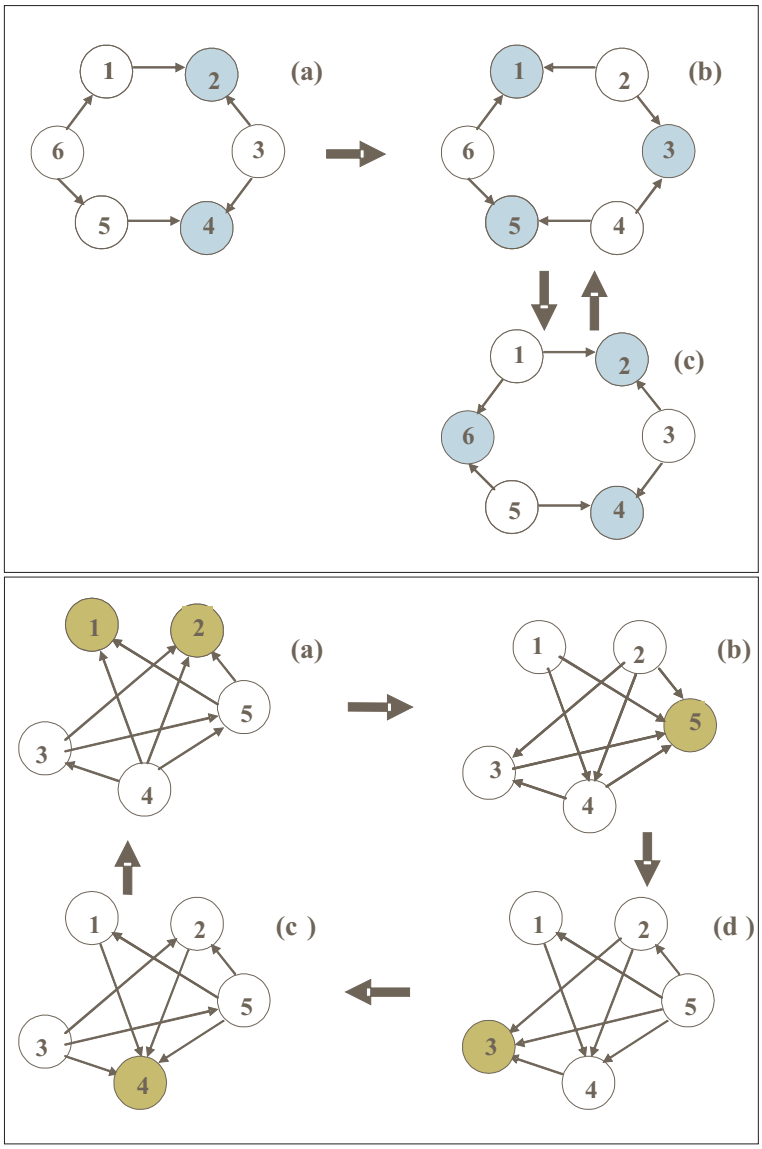


Figura 4.2: Exemplos de execução do SER

A quantidade de concorrência alcançada na execução do SER é altamente dependente da orientação inicial de G. Mas determinar uma orientação acíclica inicial de um grafo que minimize dependências entre grupos é um problema NP-difícil [7, 18]. Existem, na literatura, algoritmos para se obter uma orientação acíclica inicial não ótima [3, 17, 2]. Uma orientação acíclica inicial é ótima quando garante concorrência máxima.

Em trabalho anterior [59] foi mostrado que SER produz bons particionamentos de grafos de redes de restrições, onde os nós são restrições e as arestas representam variáveis comuns entre as restrições. Neste trabalho, primeiro obtemos o grafo complementar e em seguida executamos o SER. Utilizamos o grafo complementar porque os nós que são

sinks ao mesmo tempo no grafo complementar correspondem a nós que compartilham um recurso no grafo original (formam um clique no grafo original [26]). O algoritmo SER apresenta duas fases principais: (1) orientação do grafo e (2) reversão de arestas.

Para aplicar SER ao grafo CSP, precisamos ter uma orientação acíclica inicial. Uma forma simples de se obter uma orientação acíclica é atribuir identificações distintas totalmente ordenadas a todos os nós do grafo e então orientar as arestas de acordo com a ordem total. Entretanto, a menos que possa ser assumido que os nós começam sem estas identificações, pode ser necessário reordená-los através de técnicas probabilísticas para assegurar que eles sejam distintos [7]. A concorrência provida por uma orientação acíclica determinística pode ser pior do que uma orientação acíclica aleatória.

Em [59] foram experimentados três algoritmos de orientação acíclica, sendo que o algoritmo *Alg-Colour* [2, 3] apresentou uma boa solução para orientar um grafo antes de aplicar técnicas baseadas em SER [59]. *Alg-Colour* é o algoritmo que gera períodos de menor tamanho e melhor concorrência. A concorrência refere-se ao agrupamento de nós que possuem dependências para que estes possam ser executados por um mesmo processo. Para nossas aplicações, notamos que *Alg-Colour* é o algoritmo que produz soluções mais estáveis com uma variação muito pequena no tamanho do período entre as execuções. Além disso, ele utiliza o algoritmo de coloração[1]. Como o problema de coloração é equivalente ao problema de se obter concorrência máxima, *Alg-Colour* produz bons resultados.

O algoritmo *Alg-Colour* trabalha sobre um grafo não orientado e possui duas fases: coloração e orientação. Na primeira fase, é usado um dado com f faces para gerar valores aleatórios associados a cada nó. Inicialmente, todos os nós obtêm um valor aleatório no intervalo $[0...f-1]$. Cada nó compara seu valor com os valores de todos os seus vizinhos. Se o nó tiver o maior valor, ele escolhe uma cor representada por um inteiro não negativo e torna-se um nó determinístico. Esta cor deve ser o valor imediatamente menor do que o das cores de seus vizinhos. Todos os nós que ainda não têm uma cor continuam ativos e participarão do próximo passo. Esta fase continua até que todos os nós tornem-se determinísticos. Na segunda fase, cada aresta é orientada do nó com a cor representada pelo maior valor para o nó com a cor de menor valor.

Considere, por exemplo, o grafo conexo formado pelos nós A, B, C, D e E, apresentado na Figura 4.3. A orientação acíclica deste grafo com *Alg-Colour* segue os passos apresentados na figura. Cada nó joga um dado de $f = 15$ faces. Os valores obtidos por cada nó são apresentados fora do nó. Se o nó tiver obtido um valor maior do que os valores de seus vizinhos, este nó escolhe um valor de cor inteiro (de 0 a 4) menor do que o de seus vizinhos já coloridos. O nó que estiver colorido está representado em cinza e

[1]Uma coloração de nós de um grafo é uma atribuição de números naturais (cores) aos nós tal que dois vizinhos não recebam a mesma cor [5]

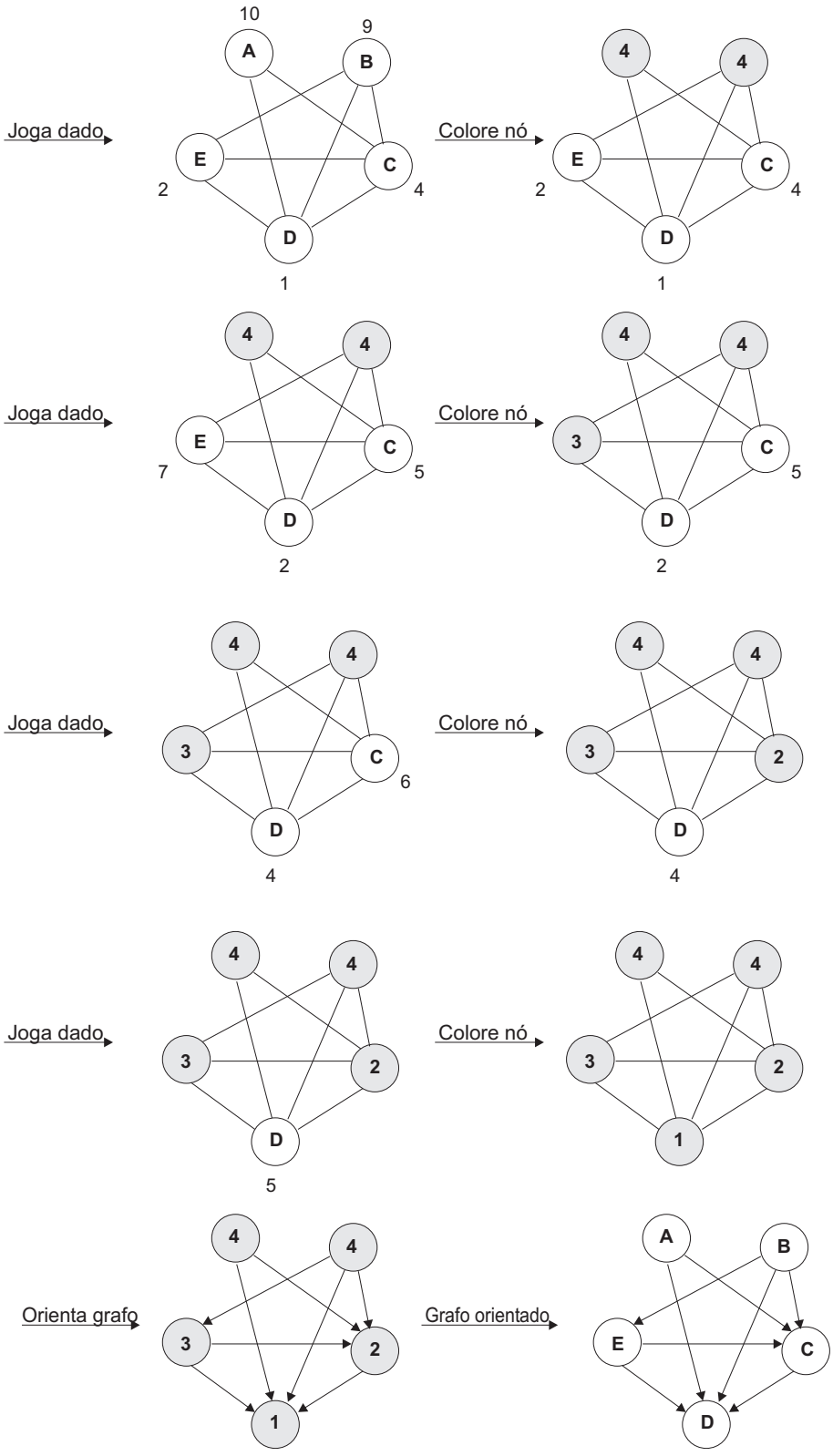


Figura 4.3: Exemplo de orientação acíclica de um grafo gerada por *Alg-Colour*

com o número da cor escolhida. Os nós A e B são os primeiros nós a se colorirem. Eles escolheram a mesma cor (4) para se colorirem pois não são vizinhos. Este passo de jogar o dado e colorir os nós que possuírem maiores valores que seus vizinhos continua até que todos os nós sejam coloridos. Em seguida é realizada a orientação das arestas do nó com cor de maior valor para o nó com cor de menor valor. O resultado da orientação é mostrado no último grafo da figura, onde o nó D é *sink*.

Após a obtenção da orientação acíclica inicial, passamos para a fase de reversão de arestas até que um período seja encontrado. Somente as orientações pertencentes ao período são consideradas como resultado. Isso porque todos os nós operam o mesmo número de vezes. Os nós *sinks* na mesma orientação são associados ao mesmo processo ou processador.

Foi observado nos experimentos realizados em [59] que a orientação acíclica inicial gerada por *Alg-Colour* para as nossas aplicações corresponde ao primeiro estado do período quando se faz as reversões de arestas. Isso acontece porque todo período SER pode ser considerado como atribuindo uma multi-coloração[2] aos nós do grafo [5]. Então esta multi-coloração atribui m cores a cada nó, das p cores do período.

Esta característica é muito importante, pois podemos usar um algoritmo para obter *sinks* mais simples do que as reversões de arestas e continuamos garantindo que todos os nós irão operar. O método que podemos usar para obter os *sinks* é a decomposição em *sinks* [8, 5]. Com este método, os grupos obtidos são idênticos aos grupos de *sinks* obtidos utilizando SER. O exemplo apresentado na Figura 4.4 mostra esta característica. O grafo inicial na Figura 4.4 (a) e (b), são idênticos e sua orientação foi obtida utilizando *Alg-Colour* (apresentado na Figura 4.3). Os grupos de *sinks* obtidos pela execução do SER foram {D}, {C}, {A,E} e {B}. Estes mesmos grupos foram obtidos executando a decomposição em *sinks*.

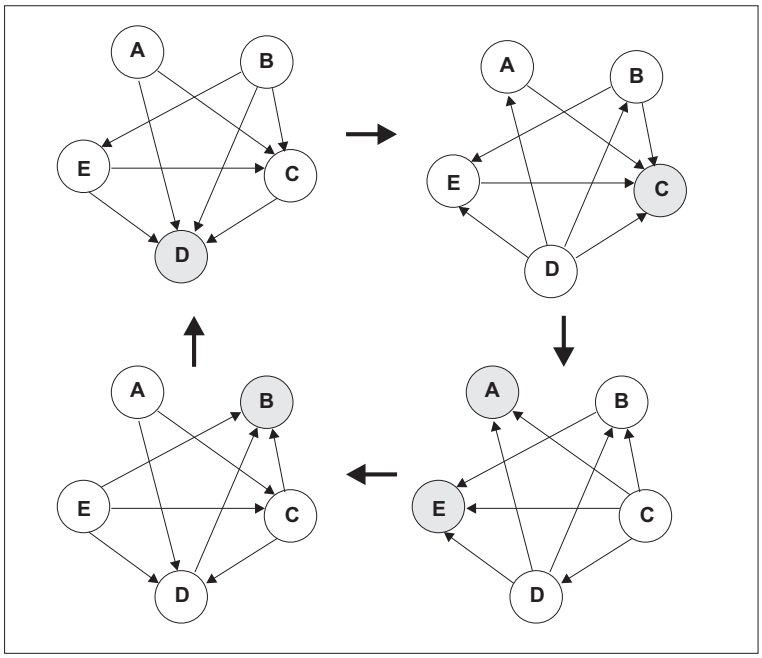
O tempo de execução de decomposição em *sinks* é menor do que o tempo gasto para executar o SER. O particionamento que utiliza decomposição em *sinks* é denominado *Grouping-Sink* e está descrito a seguir.

4.2 Particionamento *Grouping-Sink*

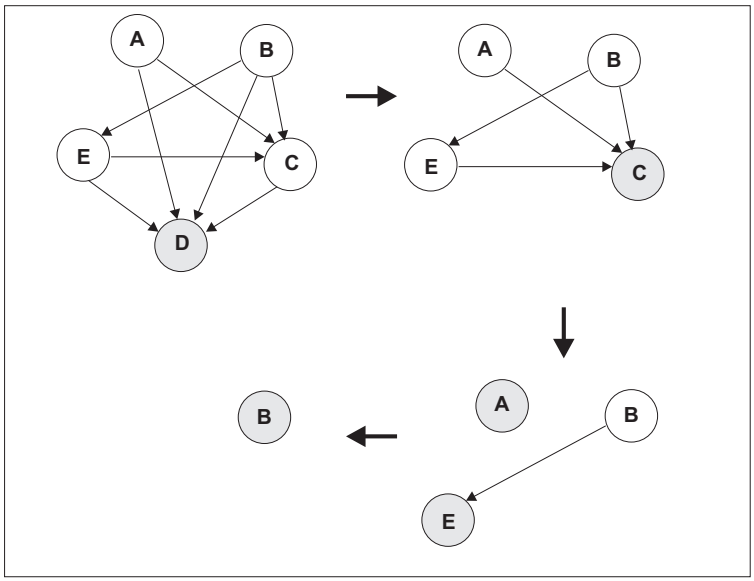
No trabalho de Barbosa e Ferreira [6] é apresentada a relação entre a orientação acíclica e coloração de um grafo da seguinte forma.

Considere um grafo G e um número inteiro positivo k. Considere que os nós de G possam ser coloridos por k cores. Então, é possível atribuir uma orientação acíclica a G, orientando as arestas do nó com a cor representada pelo valor mais alto para o nó com

[2]Multi-coloração é uma coloração que permite que mais de uma cor seja atribuída a cada nó [5]



(a) Execução do SER



(b) Execução da decomposição em *sinks*

Figura 4.4: Exemplo de execução com SER e decomposição em *sinks* do mesmo grafo orientado com *Alg-Colour*

cor de valor mais baixo. Esta orientação induz um caminho não orientado em G contendo mais do que k nós.

Suponha, agora, que G tenha uma orientação acíclica inicial. Se k for o número de nós do mais longo caminho orientado em G, de acordo com esta orientação, G poderá ser colorido com no máximo k cores. Então, a cor 1 é atribuída aos nós *sinks* gerados pela orientação acíclica. A cor 2 é atribuída aos *sinks* que se formariam se os *sinks* originais fossem removidos de G, a próxima cor disponível de valor mais baixo seria atribuída ao conjunto de *sinks* que aparecessem a seguir e assim por diante.

Este processo que começa com uma coloração produz uma única orientação acíclica. Porém, uma orientação acíclica pode produzir mais de uma coloração dos nós de G. O que o processo indica é que é possível buscar colorações ótimas para G, observando as orientações acíclicas de G que são mais curtas em relação à quantidade de nós que há no caminho orientado mais longo.

Dada uma orientação acíclica ω, a decomposição em *sinks* do grafo G, de acordo com ω, é uma partição do conjunto de nós N em $\lambda(\omega)$ conjuntos independentes $S_0, ..., S_{\lambda(\omega)}$, tal que S_0 é o conjunto de *sinks* de acordo com ω, e que para $0 \leq \ell \leq \lambda(\omega) - 1$, S_ℓ é o conjunto de nós *sinks* do grafo que permanece depois que os nós em todos os $S_0, ..., S_{l-1}$ tenham sido removidos de G. $\lambda(\omega)$ é denominado o tamanho da decomposição em *sinks* de G de acordo com w [5].

Observe os dois exemplos de orientação acíclica apresentados na Figura 4.5. Do lado direito de cada orientação acíclica há a partição do grafo em *sinks*. Esta partição é denominada decomposição em *sinks* do grafo de acordo com a orientação acíclica [8]. Em cada retângulo da figura de decomposição em *sinks* estão os nós que são *sinks* ao mesmo tempo, e que vão sendo eliminados do grafo. O número de camadas (retângulos) na decomposição em *sinks* para uma orientação acíclica é igual ao maior caminho orientado no grafo, ou seja, o caminho que possui maior número de nós. Na Figura 4.5, a segunda orientação acíclica produziu a menor decomposição em *sinks* e corresponde à melhor atribuição de cores (uma cor diferente para os nós em cada camada).

O número de camadas numa decomposição em *sinks* para uma orientação acíclica é igual ao caminho orientado no grafo que possua maior número de nós. Com isso, o número de grupos de *sinks* gerados pela decomposição em *sinks* é igual ao número de cores necessárias para colorir o grafo.

Alg-Colour [59] gera uma boa orientação acíclica, com menos nós no caminho orientado mais longo. Ao se decompor em *sinks* o grafo orientado por *Alg-Colour*, obtivemos menor número de camadas do que utilizando outro método de orientação. O número de grupos de *sinks* gerado pela decomposição em *sinks* é igual ao caminho mais longo no grafo orientado.

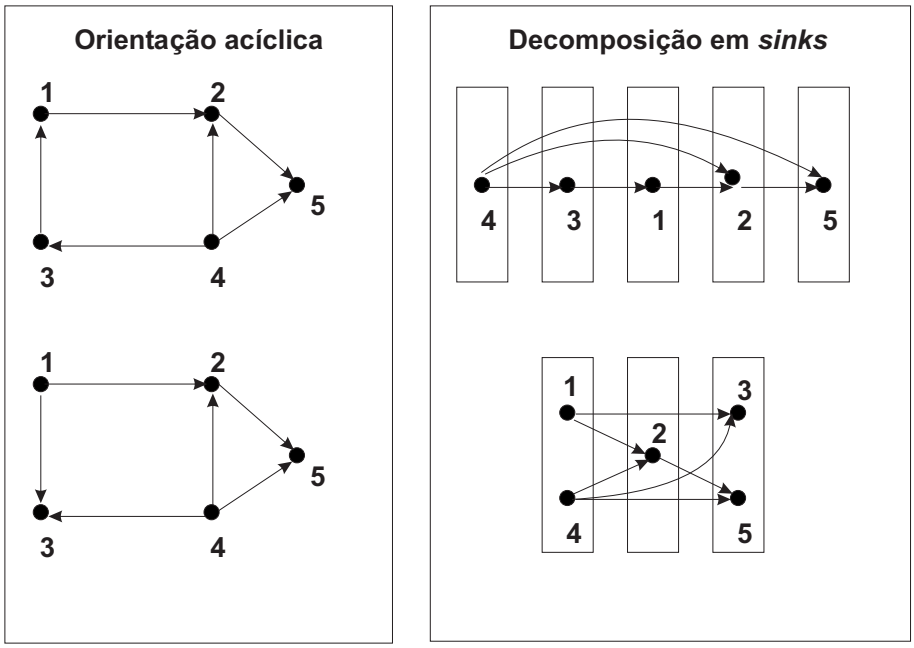


Figura 4.5: Exemplo de duas orientações acíclicas com suas decomposições em *sinks* [6]

O particionamento *Grouping-Sink* utiliza o método de decomposição em *sinks* para gerar grupos de *sinks* a partir de um grafo de restrições com orientação acíclica inicial gerada pelo algoritmo *Alg-Colour*.

O algoritmo de decomposição em *sinks* consiste em ir retirando os nós *sinks* e suas arestas do grafo até que não restem mais nós. Cada conjunto de *sinks* que é retirado do grafo deve pertencer ao mesmo grupo que será alocado a um processador numa execução paralela/distribuída.

Em nosso trabalho, tanto o SER quanto a decomposição em *sinks* atuam sobre o grafo complementar, pois nós *sinks* ao mesmo tempo no grafo complementar são nós que compartilham recursos (dependência) e devem ser colocados num mesmo grupo. Os subconjuntos de nós obtidos em cada passo correspondem ao particionamento do grafo de restrições em k subconjuntos S. O tempo gasto na remoção de nós *sinks* é menor do que o tempo gasto na reversão de arestas devido a (1) baixa complexidade para detectar *sinks* e (2) detecção de terminação mais simples. Já que a detecção de terminação é dependente do tamanho do grafo e o grafo decresce durante a execução, a complexidade também decresce. No algoritmo SER, a terminação do algoritmo é detectada através da comparação de estados para verificar se um período foi encontrado. No algoritmo de decomposição em *sinks*, a detecção de terminação é obtida verificando se o grafo está vazio (sem nós).

A seguir estão apresentados os detalhes da implementação de *Grouping-Sink*.

4.3 Implementação do Particionamento *Grouping-Sink*

Grouping-Sink foi implementado na linguagem Java, pois aproveitou várias classes utilizadas pelo SER, que havia sido implementado em Java. Nosso protótipo possui quatro passos principais:

1. leitura de dados e geração do grafo;

2. orientação do grafo com *Alg-Colour*;

3. decomposição em *sinks*;

4. remapeamento dos grupos de *sinks* aos processadores disponíveis.

No primeiro passo, é lido o arquivo de entrada que é gerado por um programa Prolog [62]. Os dados são representados na forma de *indexicals* [20], que explicitam as variáveis de cada restrição.

Este arquivo contém a rede de restrições necessária para executar a aplicação CSP. A rede de restrições consiste de uma lista de restrições que relaciona diversas variáveis.

Um grafo complementar não orientado é construído a partir do grafo original porque nós que são *sinks* ao mesmo tempo representam um clique no grafo original. Estes nós representam restrições que compartilham uma ou mais variáveis e não podem realizar a atribuição de valor para essas variáveis ao mesmo tempo. Então, é melhor manter estas restrições no mesmo grupo. Esta estratégia usando grafo complementar, também, foi utilizada no trabalho de França *et al.* [27].

Uma matriz nxn (onde n é o número de restrições) é criada para representar o grafo de restrições complementar. Em seguida, o grafo é orientado utilizando o algoritmo *Alg-Colour* [59]. Após obter a orientação acíclica inicial, a decomposição em *sinks* é executada considerando um grafo orientado.

O algoritmo decomposição em *sinks* é apresentado na Figura 4.6.

1 Enquanto houver nós no grafo
2 Detecta sinks verificando as linhas da matriz;
3 Armazena sinks;
4 Remove os sinks do grafo;
5 Fim-enquanto
6 Apresenta o resultado final

Figura 4.6: Algoritmo de decomposição em *sinks*

Este algoritmo remove nós *sinks* de um grafo e suas arestas, reduzindo o tamanho do grafo. Este passo continua enquanto existir nós no grafo. Cada grupo de *sinks* obtido

deve ser associado ao mesmo processador para diminuir a quantidade de comunicação numa execução paralela/distribuída. Numa execução seqüencial é diminuído o número de passos.

Os grupos de *sinks* gerados são apresentados ao usuário como um programa Prolog, que consiste de uma base de dados de fatos Prolog. Este formato é apropriado para ser usado como entrada para o PCSOS [62] (solver que utilizamos em nossos experimentos), que é apresentado no Capítulo 5. O número de grupos gerados pode ser maior do que o número de processadores disponíveis para a execução da aplicação. Assim, a última fase de *Grouping-Sink* consiste em remapear os grupos aos processadores disponíveis. Este remapeamento é realizado de forma *round-robin*, ou seja, o primeiro grupo é atribuído ao primeiro processador, o segundo grupo ao segundo processador e assim por diante. Esta fase é realizada dentro do PCSOS.

Antes de escolher esta forma de remapeamento, foram testadas outras três formas: (1) considerando a seqüência numérica dos grupos gerados, dividir o número de grupos pelo número de processadores disponíveis. Em caso de divisão não exata, o primeiro processador ficaria com um número maior de grupos que os demais; (2) considerando a seqüência numérica dos grupos gerados, dividir o número de grupos pelo número de processadores disponíveis. Em caso de divisão não exata, o último processador ficaria com um número maior de grupos e (3) juntando os grupos de acordo com o número de variáveis comuns entre eles. Mas, *round-robin* foi o método que apresentou o melhor resultado. Novas formas de remapeamento podem ser estudadas, mas está fora do escopo deste trabalho.

Para ilustrar o comportamento do método *Grouping-Sink*, comparado com os outros particionamentos, considere um exemplo pequeno de *Arithmetic* com 6 variáveis (Aa, Ab, VAb, VBa, Ba e Bb). Na Tabela 4.1, a primeira coluna apresenta o número dos *indexicals* que estão na segunda coluna. Na terceira coluna são apresentadas as 11 restrições do problema, sendo que 5 delas representam restrições relacionando duas ou mais variáveis e 6 representam as restrições associadas ao domínio de cada variável. Os *indexicals* estão representados em função de máximos e mínimos das variáveis.

Este problema pode ser representado com um grafo de restrições ou um grafo de *indexicals*. Como a granulosidade do grafo de restrições é menor, exemplificamos os passos de *Grouping-Sink* por restrições. São ilustrados o grafo original deste problema, o grafo complementar, a orientação do grafo, a decomposição em *sinks* e a realocação das restrições nos processadores.

Num.	*Indexicals*	Restrições
0	$Aa\ in\ 1..8$	$Aa\ in\ 1..8$
1	$Ab\ in\ 1..8$	$Ab\ in\ 1..8$
2	$Aa\ in\ 126/11 - (13/11) * min(Ab)\ ..\ 126/11 - (13/11) * max(Ab)$	$11 * Aa + 13 * Ab \neq 126$
3	$Ab\ in\ 126/13 - (11/13) * min(Aa)\ ..\ 126/13 - (11/13) * max(Aa)$	
4	$Aa\ in\ 139/10 - (14/10) * min(Ab) - (1/10) * min(VAb)\ ..\ 139/10 - (14/10) * max(Ab) - (1/10) * max(VAb)$	$10{*}Aa{+}14{*}Ab{+}VAb \neq 139$
5	$Ab\ in\ 139/14 - (10/14) * min(Aa) - (1/14) * min(VAb)\ ..\ 139/14 - (10/14) * max(Aa) - (1/14) * max(VAb)$	
6	$VAb\ in\ 139 - 10 * min(Aa) - 14 * min(Ab)\ ..\ 139 - 10 * max(Aa) - 14 * max(Ab)$	
7	$VAb\ in\ 1..8$	$VAb\ in\ 1..8$
8	$VBa\ in\ 1..8$	$VBa\ in\ 1..8$
9	$VAb\ in\ 9 - min(VBa)\ ..\ 9 - max(VBa)$	$VAb + VBa \neq 9$
10	$VBa\ in\ 9 - min(VAb)\ ..\ 9 - max(VAb)$	
11	$Ba\ in\ 1..8$	$Ba\ in\ 1..8$
12	$Bb\ in\ 1..8$	$Bb\ in\ 1..8$
13	$Ba\ in\ 59/3 - (12/3) * min(Bb) - (1/3) * min(VBa)\ ..\ 59/3 - (12/3) * max(Bb) - (1/3) * max(VBa)$	$3{*}Ba{+}12{*}Bb{+}VBa \neq 59$
14	$Bb\ in\ 59/12 - (3/12) * min(Ba) - (1/12) * min(VBa)\ ..\ 59/12 - (3/12) * max(Ba) - (1/12) * max(VBa)$	
15	$VBa\ in\ 59 - 3 * min(Ba) - 12 * min(Bb)\ ..\ 59 - 3 * max(Ba) - 12 * max(Bb)$	
16	$Ba\ in\ 40/4 - (7/4) * min(Bb)\ ..\ 40/4 - (7/4) * max(Bb)$	$4 * Ba + 7 * Bb \neq 40$
17	$Bb\ in\ 40/7 - (4/7) * min(Ba)\ ..\ 40/7 - (4/7) * max(Ba)$	

Tabela 4.1: Conjunto de *indexicals* para *Arithmetic* com 6 variáveis

A Figura 4.7 (a) apresenta o grafo de restrições referente a este problema e a Figura 4.7 (b) apresenta o grafo complementar. Cada nó possui uma restrição representada na forma de *indexicals*. Por exemplo, o nó que possui os números 2 e 3 está representando uma restrição através de seus dois *indexicals*. A utilização do grafo complementar ocorre porque queremos agrupar restrições que possuem dependências num mesmo processador. Desta forma, buscamos minimizar comunicação entre processadores.

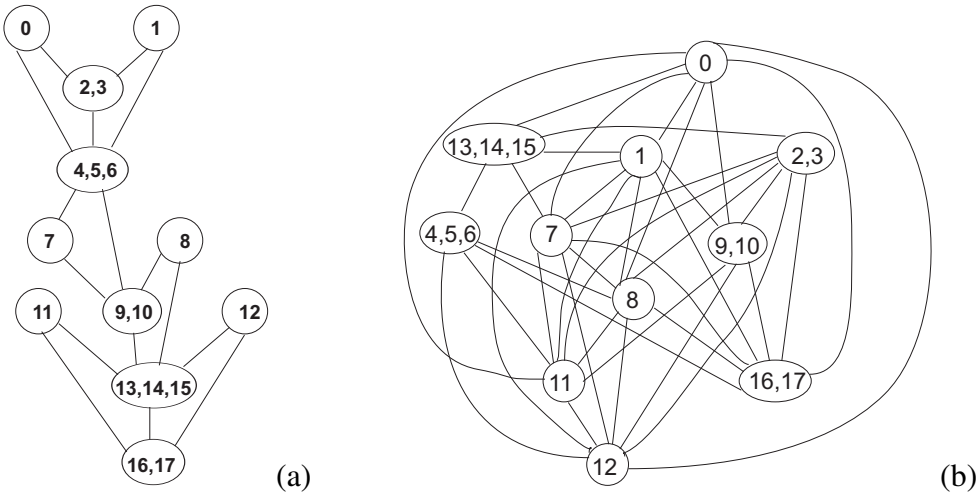


Figura 4.7: *Arithmetic*: grafo de restrições original (a) e grafo complementar (b)

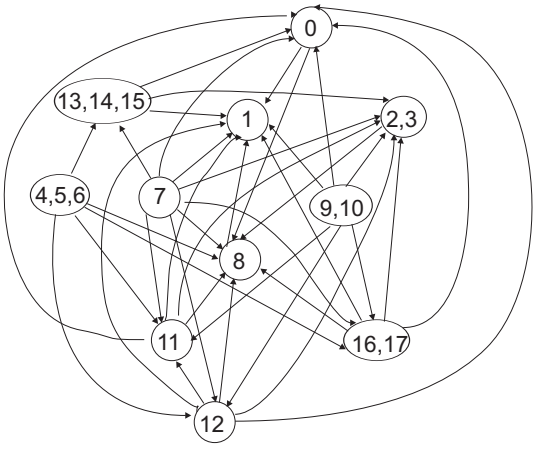


Figura 4.8: *Arithmetic*: grafo complementar orientado com *Alg-Colour*

Para realizar a decomposição em *sinks* é necessário que o grafo tenha uma orientação acíclica inicial. A Figura 4.8 apresenta o grafo da Figura 4.7 (b) com uma orientação acíclica que foi gerada utilizando o algoritmo *Alg-Colour*.

A Figura 4.9 apresenta os passos de decomposição em *sinks* do grafo orientado com *Alg-Colour*, onde os nós *sinks* vão sendo retirados do grafo e formando os grupos, que estão representados como conjuntos de *indexicals*. Neste exemplo, formaram-se 6 grupos: {1}, {8}, {0,2,3}, {11,13,14,15,16,17}, {12} e {4,5,6,7,9,10}. Para alocar estes grupos

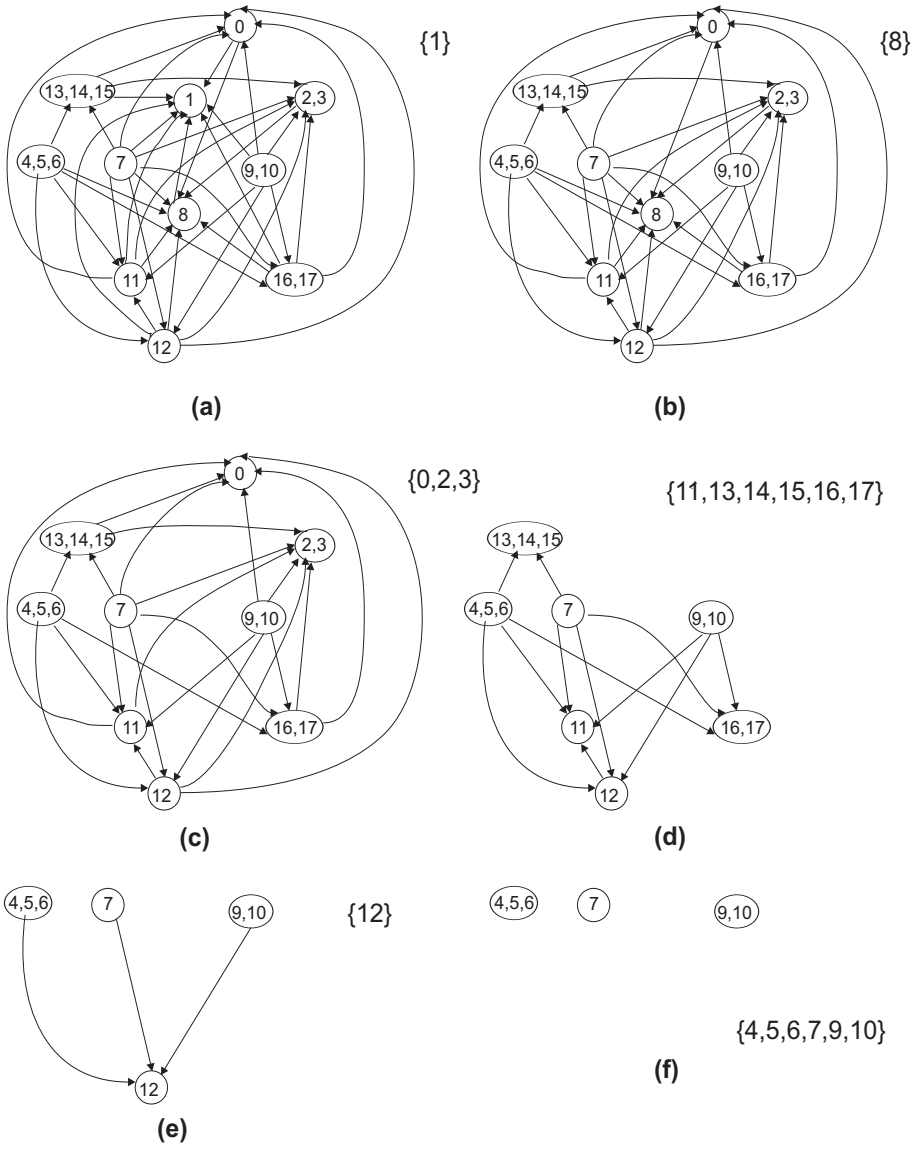


Figura 4.9: *Arithmetic*: decomposição em *sinks*

em 4 processadores os processadores P0 e P1 ficam com dois grupos e os demais com um grupo. Assim, os *indexicals* 1 e 8 ficam no processador P0 e 0, 2, 3, 11, 13, 14, 15, 16 e 17 ficam no processador P1.

A alocação dos *indexicals* aos 4 processadores, gerada por *Grouping-Sink* por restrições, é apresentada na Figura 4.10. Nesta figura, os retângulos representam os processadores, as elipses cinzas são as variáveis e os círculos numerados são os *indexicals*. As linhas pontilhadas entre *indexicals* indicam que os *indexicals* ligados representam a mesma restrição. Os retângulos cinza entre os processadores representam dependências devido às variáveis indicadas (variáveis comuns entre os processadores).

O particionamento, desta mesma aplicação, gerado por *Grouping-Sink* por *indexicals*, Blocos por variáveis, *indexicals*, restrições e *Round-Robin* por variáveis, *indexicals* e

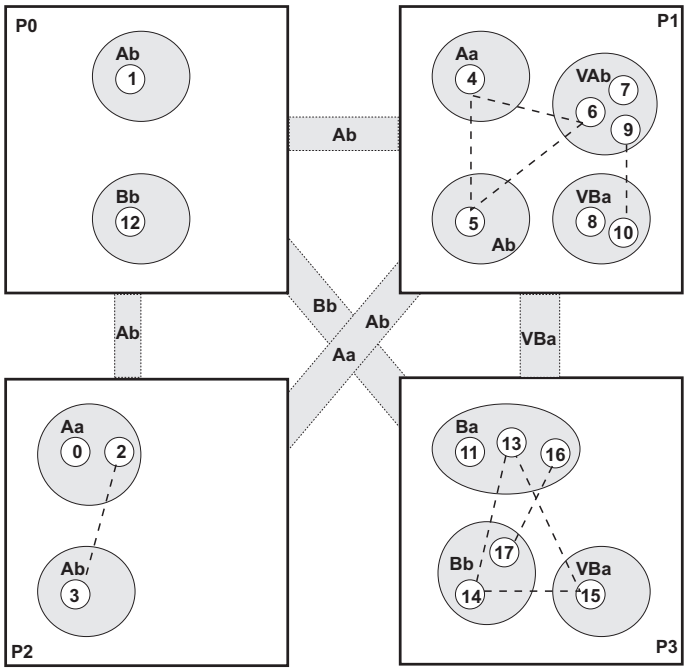


Figura 4.10: *Arithmetic*: particionamento gerado por *Grouping-Sink* por restrições

restrições, está ilustrado nas Figuras de 4.11 a 4.17.

Podemos observar que *Grouping-Sink* por *indexicals* busca agrupar os *indexicals* de acordo com as váriáveis envolvidas (Figura 4.11), minimizando o número de variáveis compartilhadas entre os processadores. Assim, uma restrição pode ter cada um de seus *indexicals* num processador diferente. No *Grouping-Sink* por restrições (Figura 4.10), os *indexicals* de uma mesma restrição ficam num mesmo processador, mas a quantidade de variáveis compartilhadas entre os processadores pode ser alta. Dependendo da restrição, pode haver o compartilhamento de todas as variáveis. Por isso, *Grouping-Sink* por *indexicals* realiza um particionamento melhor do que *Grouping-Sink* por restrições. Observando todos os particionamentos, verificamos que *Grouping-Sink* por *indexicals* possui comportamento similar a Blocos por variáveis e *Round-Robin* por variáveis.

Os particionamentos Blocos por *indexicals* e por restrições, *Round-Robin* por *indexicals* (Figura 4.16) e por restrições (Figura 4.17) apresentam a maioria das variáveis em todos os processadores, gerando uma dependência entre os processadores muito grande, que pode gerar muita comunicação entre os processadores.

Grouping-Sink é um método que pode ser aplicado para particionar qualquer problema de satisfação de restrições. Para validar o particionamento gerado por *Grouping-Sink* foi utilizado o *solver* para domínios finitos denominado PCSOS [62]. *Grouping-Sink* é comparado com métodos de particionamento utilizados em outros trabalhos [62, 57]. O Capítulo 5 apresenta a metodologia experimental utilizada para realizar nossos experimentos com *Grouping-Sink*.

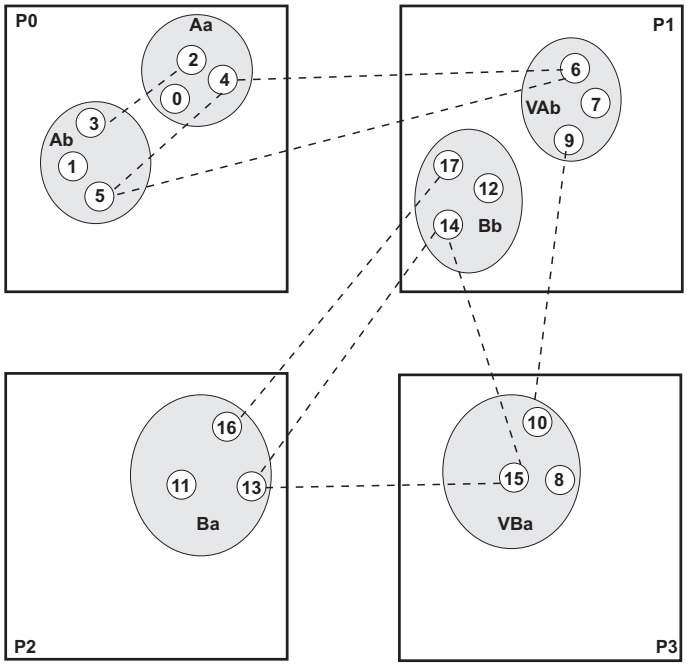


Figura 4.11: *Arithmetic*: particionamento gerado por *Grouping-Sink* por *indexicals*

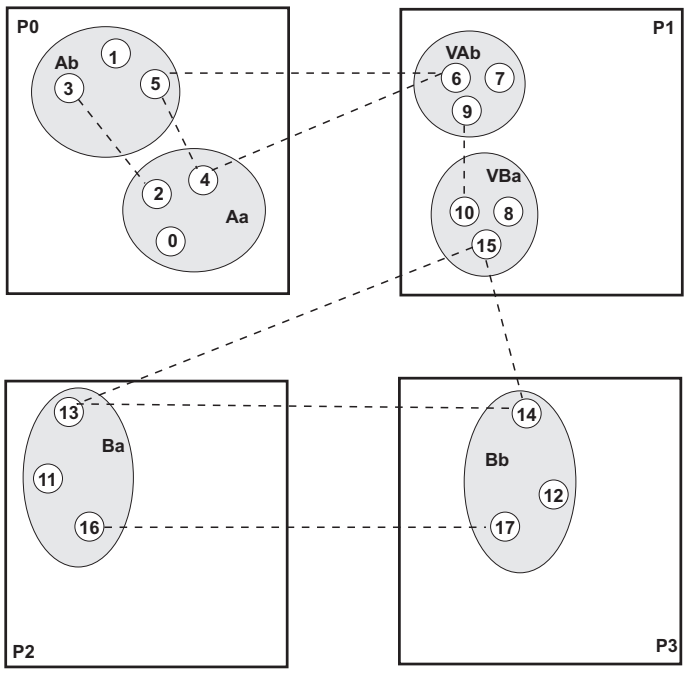


Figura 4.12: *Arithmetic*: particionamento gerado por Blocos por variáveis

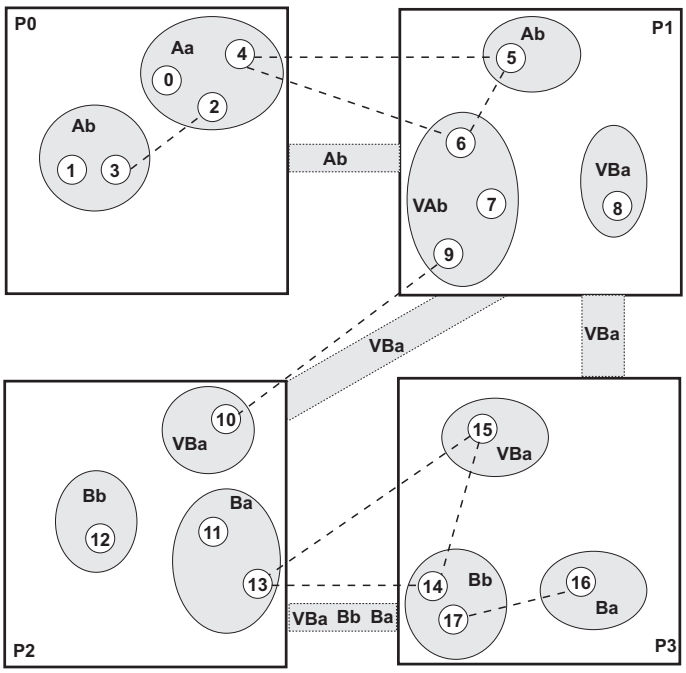


Figura 4.13: *Arithmetic*: particionamento gerado por Blocos por *indexicals*

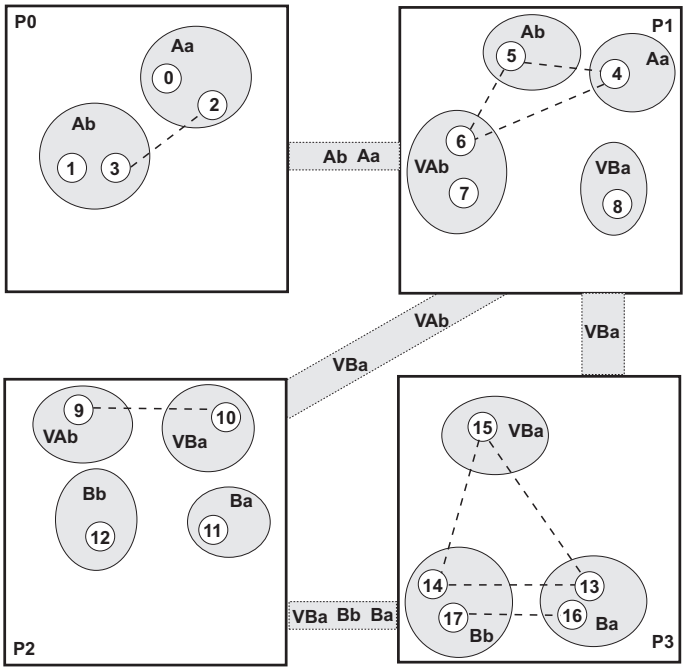


Figura 4.14: *Arithmetic*: particionamento gerado por Blocos por restrições

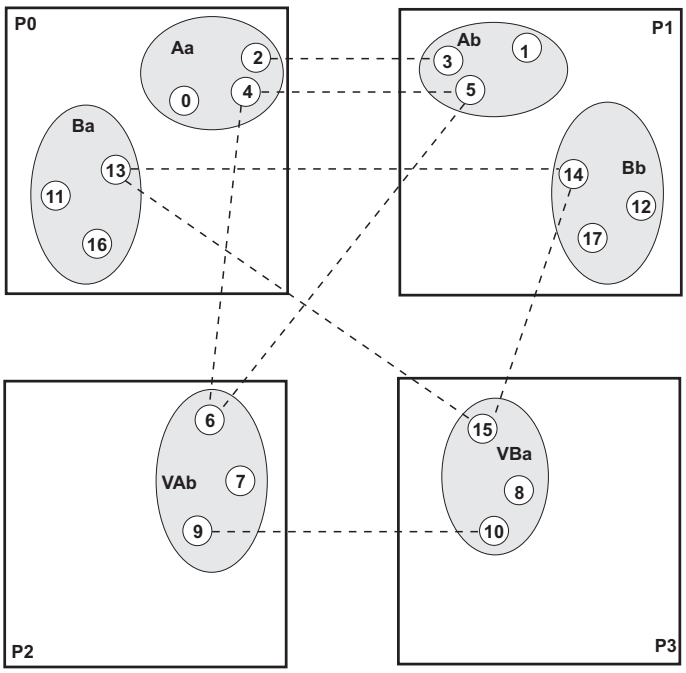


Figura 4.15: *Arithmetic*: particionamento gerado por *Round-Robin* por variáveis

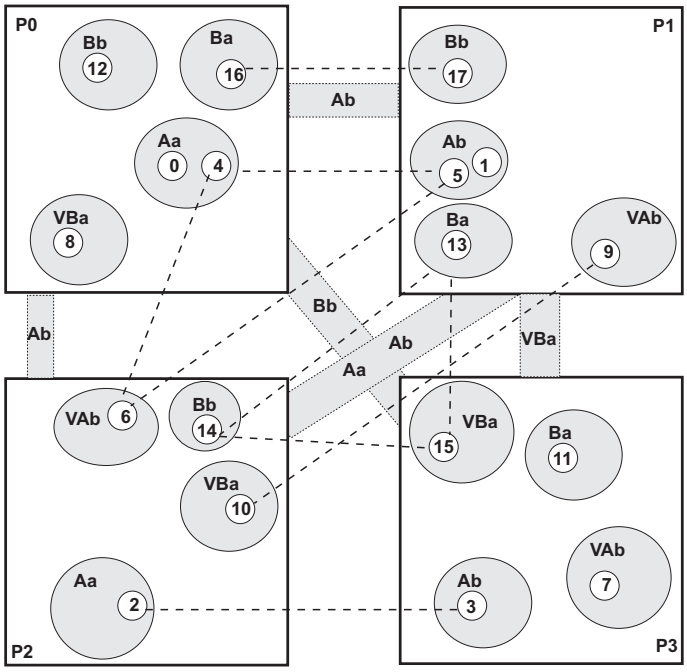


Figura 4.16: *Arithmetic*: particionamento gerado por *Round-Robin* por *indexicals*

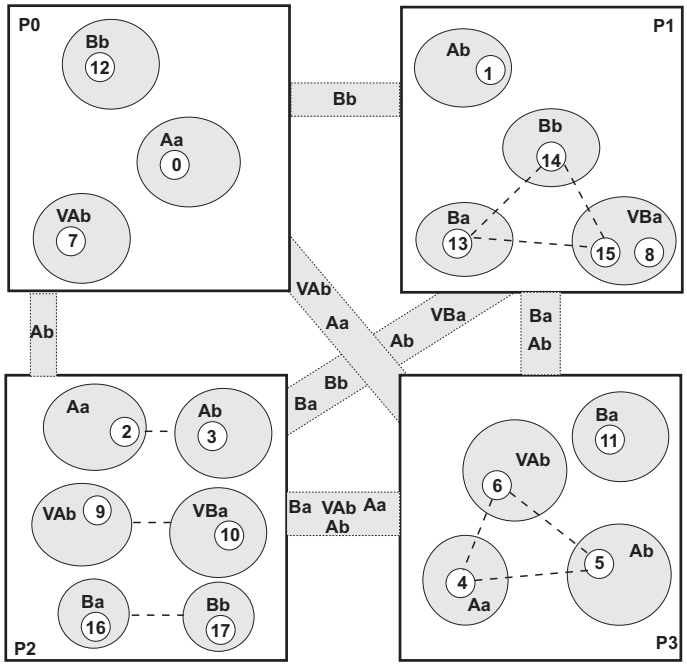


Figura 4.17: *Arithmetic*: particionamento gerado por *Round-Robin* por restrições

Capítulo 5

Metodologia Experimental

Neste capítulo descrevemos a metodologia utilizada neste trabalho. Na primeira seção, apresentamos as aplicações e suas características. Na segunda seção, apresentamos o PCSOS, o sistema que utilizamos para realizar nossos experimentos e a plataforma de *hardware*.

5.1 Descrição das aplicações

Foram utilizadas quatro aplicações para estudar o particionamento de restrições produzido pelos métodos estudados: *Arithmetic*, *Queens*, PBCSP e *Sudoku*. Estas aplicações foram utilizadas em outros trabalhos que abordam diferentes técnicas para particionar as restrições [61, 62, 57, 59].

Arithmetic O primeiro *benchmark*, *Arithmetic*, é um *benchmark* sintético. São definidos x blocos de relações aritméticas, $\{B_1, \ldots, B_x\}$, onde cada bloco contém y equações e inequações relacionando z variáveis. Os blocos B_i e B_{i+1} são conectados por uma equação adicional entre um par de variáveis, um de B_i e outro de B_{i+1}. Os coeficientes foram gerados aleatoriamente. O objetivo é encontrar um vetor solução inteiro. Este tipo de programação com restrições é muito usado para decomposição de problemas grandes de otimização.

A Figura 5.1 apresenta a instância de *Arithmetic* com 16 blocos de relações aritméticas. Cada bloco possui 15 equações e inequações entre 6 variáveis mais as variáveis de folga. No total são 126 variáveis, que estão representadas na figura com os valores de 0 a 125. Em cinza são apresentadas as variáveis de ligação entre os blocos (variáveis de folga).

Queens O segundo *benchmark*, *Queens*, consiste em colocar N rainhas em um tabuleiro de xadrez $N \times N$ de maneira que as rainhas não se ataquem na mesma linha, coluna e

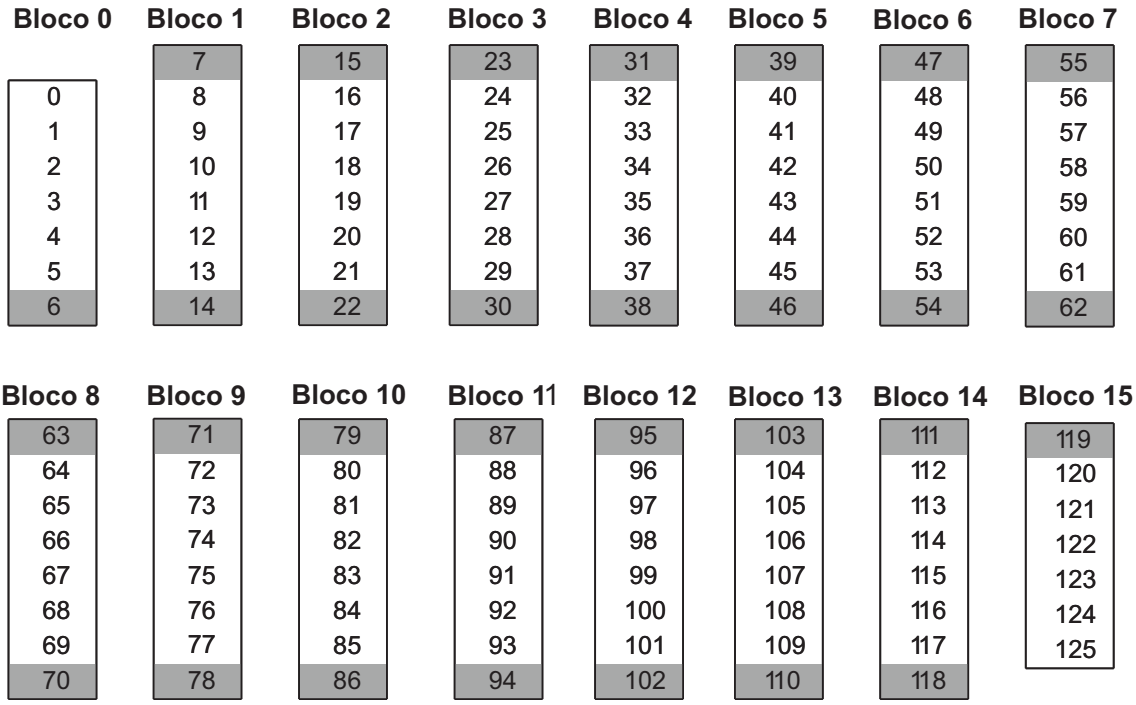


Figura 5.1: Variáveis de *Arithmetic* distribuídas nos 16 blocos

diagonal principal ou secundária. As restrições são todas inequações. Em nossos experimentos utilizamos um tabuleiro com 111 rainhas, pois gera um grafo com 18.426 restrições nos nós e outro com 36.741 *indexicals* nos nós, permitindo avaliar melhor os particionamentos.

Para exemplificar, considere a Figura 5.2, que ilustra um tabuleiro 4×4, onde V_i é a coluna a ser ocupada pela rainha da linha i. A rainha representada em preto está colocada na posição 1 e pode atacar as demais na mesma linha, coluna ou diagonal primária e secundária. Seu movimento está representado em linha pontilhada.

As restrições que descrevem este problema são as seguintes:

$$
\begin{array}{lll}
V_0 \neq V_1 & V_0 \neq V_3 & V_1 \neq V_3 \\
V_0 \neq V_1 - 1 & V_0 \neq V_3 - 3 & V_1 \neq V_3 - 2 \\
V_0 \neq V_1 + 1 & V_0 \neq V_3 + 3 & V_1 \neq V_3 + 2 \\
V_0 \neq V_2 & V_1 \neq V_2 & V_2 \neq V_3 \\
V_0 \neq V_2 - 2 & V_1 \neq V_2 - 1 & V_2 \neq V_3 - 1 \\
V_0 \neq V_2 + 2 & V_1 \neq V_2 + 1 & V_2 \neq V_3 + 1
\end{array}
$$

A rainha em preto e as que estão representadas em cinza estão atendendo a todas estas restrições. Por isso, as rainhas colocadas nestas posições no tabuleiro correspondem a uma solução do problema. Desta forma, as variáveis recebem os valores $V_0 = 1$, $V_1 = 3$, $V_2 = 0$ e $V_3 = 2$.

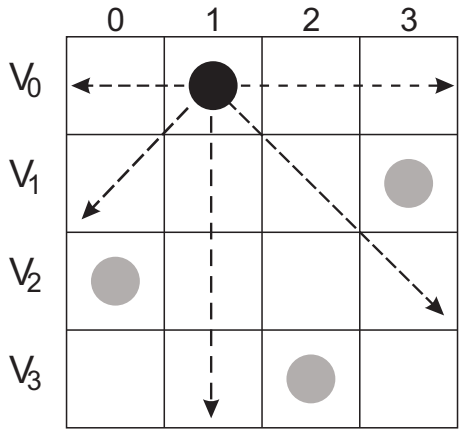


Figura 5.2: Exemplo de *Queens* para 4 rainhas

PBCSP O terceiro *benchmark*, *Parametrizable Binary Constraint Satisfaction Problem* (PBCSP), é um *benchmark* sintético. Instâncias deste problema são geradas aleatoriamente de acordo com quatro parâmetros: número de variáveis (nv), o tamanho dos domínios iniciais (ds), densidade e *tightness*. Densidade e *tightness* são definidos como $\frac{nc}{nv-1}$ e $1 - \frac{np}{ds^2}$, respectivamente, onde nc é o número de restrições envolvendo uma variável (é a mesma para todas elas) e np é o número de pares que satisfazem as restrições. O *tightness* representa o grau de dificuldade para se chegar a uma solução. Em nossos experimentos, o tamanho do domínio foi fixado em 20, *tightness* é igual a 0,85, que também foram utilizados em [62]. PBCSP com 50 variáveis possui densidade 0,65 e com 100 variáveis densidade 0,75. A primeira instância executa mais passos do que a segunda para chegar a uma solução, pois os grafos de restrições são diferentes. PBCSP é utilizado na área de programação com restrições, com outros nomes, pois a variação dos parâmetros permite a geração de conjuntos de restrições que correspondem a classes diferentes de grafos de restrições.

Sudoku É um problema cripto-aritmético japonês. Dada uma grade de 25×25 quadrados, onde 317 deles são totalmente preenchidos com um número entre 1 e 25, a idéia é preencher o restante dos quadrados tal que cada linha e coluna seja uma permutação dos números de 1 a 25. Entretanto, cada uma das 25 grades de 5×5 quadrados começa nas colunas (linhas) 1, 6, 11, 16, 21 e deve ser uma permutação de números de 1 a 25.

Um exemplo de *Sudoku*, em tamanho menor (9×9), é apresentado na Figura 5.3. Há 36 números para serem preenchidos, que correspondem às variáveis do problema (variáveis de V_0 a V_{35} na figura). As restrições são as seguintes: os números a serem colocados nos quadrados vazios devem ser de 1 a 9 e diferentes dos números já preenchidos na mesma linha e coluna da grade e mesmo quadrado 3×3. Isto é, numa mesma

linha/coluna da grade um mesmo número não pode se repetir. Esta restrição é válida para cada quadrado de tamanho 3×3.

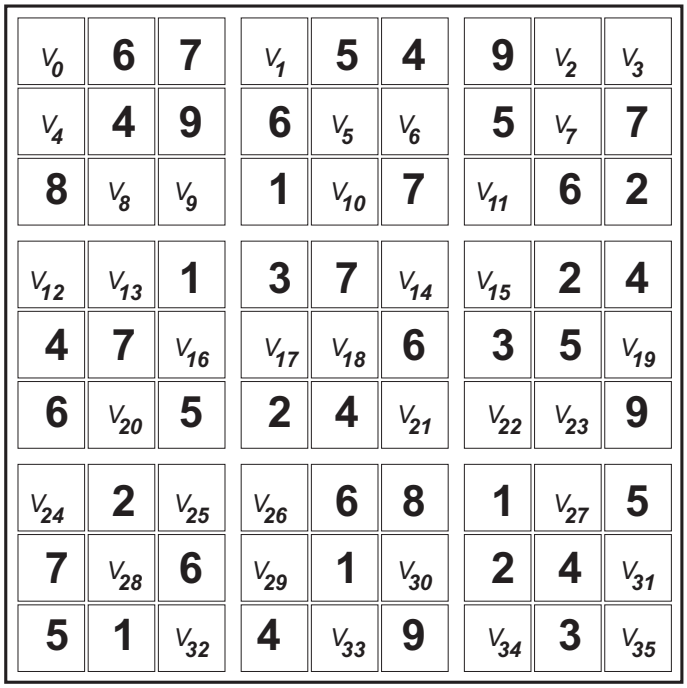


Figura 5.3: Exemplo de *Sudoku* 9×9 [21]

Aplicação	Vars.	Número de Restrições	Número de *Indexicals*	Num. de Arestas do Grafo Complementar		Densidade do Grafo Complementar	
				restr.	ind.	restr.	ind.
Arithmetic	126	381	1.626	69.180	1.309.515	0,96	0,99
Queens	111	18.426	36.741	163.705.575	668.869.905	0,96	0,99
PBCSP	50	1.620	3.190	1.211.985	4.986.265	0,92	0,98
PBCSP	100	7.538	14.976	27.296.710	111.018.838	0,96	0,99
Sudoku	308	17.099	22.484	145.370.463	251.944.462	0,99	0,99

Tabela 5.1: Características das Aplicações

A Tabela 5.1 apresenta as principais características das aplicações usadas nos experimentos: número de variáveis, de restrições, de *indexicals*, número de arestas e conectividade dos grafos de restrições e de *indexicals*. Estas informações estão relacionadas ao grafo complementar.

A densidade representa o percentual de conexão entre os nós e está relacionada ao grafo complementar (que é utilizado por *Grouping-Sink*). O grafo de restrições ao qual nos referimos é o grafo onde um nó representa uma restrição e uma aresta é uma variável comum entre dois nós e o grafo de *indexicals* corresponde ao grafo onde um nó representa um *indexical* e uma aresta é uma variável comum entre dois nós. O cálculo

da densidade é realizado dividindo-se o número de arestas e pelo número de arestas num grafo correspondente completo com n nós (densidade = $\frac{e}{[n*(n-1)]/2}$).

A utilização de *benchmarks* sintéticos deve-se à dificuldade em se encontrar aplicações reais que sejam de domínio público. Além disso, as aplicações de um modo geral não estão escritas em Prolog e então exigem um esforço adicional para escrevê-las nesta linguagem. É importante ressaltar que no contexto deste trabalho utilizamos Prolog, porque nosso pré-processador para gerar a rede de restrições está escrito em Prolog e lê programas Prolog. Porém, outros pré-processadores poderiam ser utilizados.

Note, também, que as aplicações utilizadas possuem diferentes características em relação ao número de variáveis, e natureza dos grafos, apresentando uma diversidade e podendo representar muitos problemas reais.

5.2 Plataforma de *hardware* e *software*

A plataforma experimental utilizada consiste em uma máquina de memória compartilhada com 14 processadores. A máquina é uma *Sun Enterprise* 4500 com sistema operacional Solaris 8, situada na *New Mexico State University, USA*. Utilizamos uma máquina com memória compartilhada para que pudéssemos controlar melhor o ambiente para fazer experimentos. A idéia é implementar o particionamento e testá-lo para, então, utilizar uma plataforma mais complexa, como por exemplo, um ambiente de memória distribuída.

O objetivo de um *solver* é encontrar uma ou várias soluções para um problema de satisfação de restrições ou então concluir que não há solução. O *solver* que utilizamos é o PCSOS. Este sistema foi implementado por Andino *et al.* [62, 57] e estendido em nosso trabalho para dar suporte aos particionamentos implementados. Além disso, teve que ser adaptado para executar na plataforma de *hardware* descrita através da introdução das primitivas de sincronização adequadas para a máquina utilizada [56].

O arquivo de entrada do PCSOS é um arquivo texto que passou por um passo de pré-processamento para ter o formato adequado. O CSP é escrito em Prolog. Existe um pré-processador Prolog que lê o CSP em Prolog e gera dois arquivos: um arquivo de restrições e outro de *indexicals* [20]. Quando o particionamento utilizado não é *Grouping-Sink*, este arquivo de *indexicals* é a entrada para o PCSOS. Quando utilizamos Grouping-Sink, estes dois arquivos são utilizados pelo algoritmo de decomposição em *sinks* para gerar o agrupamento de *sinks*. O número de cada grupo ao qual cada *indexical* pertence é adicionado adequadamente ao arquivo de *indexicals*, que é a entrada para o PCSOS. Dentro do PCSOS, os *indexicals* são alocados aos processadores de acordo com o número do grupo definido no arquivo.

O *solver* PCSOS para domínios finitos contém três fases principais: a fase de consistência de nós, onde são eliminados os valores inconsistentes do domínio das variáveis

devido à execução do conjunto de restrições unárias; a fase de consistência de arcos, onde os valores eliminados são devido à execução de restrições binárias e a fase de *labeling*.

Cada restrição está representada por *indexicals*. No esquema de *indexicals* somente uma variável da restrição é evidenciada de cada vez. Portanto, uma restrição com duas variáveis é representada por dois *indexicals*. Assim, ao executar a consistência de nós e a consistência de arcos, é realizada a eliminação dos valores inconsistentes dos domínios das variáveis através da execução do conjunto de todos os *indexicals*.

A fase de *labeling* consiste em escolher uma variável e um valor do domínio para esta variável e realizar a consistência desta atribuição com o restante das variáveis. Para isso é executado todo o conjunto de *indexicals*. Quando um valor escolhido gera a poda do domínio de uma variável, que se torna vazio, é gerada uma falha. Pois não há como encontrar uma solução. Então, é realizado o procedimento de *backtrack*, onde são recuperados a última variável e o último valor escolhido para esta variável e restaurados os domínios das demais variáveis para o estado anterior. É escolhido um novo valor para a variável e realizada a nova consistência. Este processo continua até se encontrar uma solução (os domínios das variáveis são todos podados para um único valor obedecendo todas as restrições) ou atingir um ponto em que se conclua que não há solução para a aplicação.

A implementação é feita de forma distribuída para uma plataforma de memória compartilhada. Desta forma, as rotinas implementadas com a finalidade de atualizar as variáveis compartilhadas simulam uma troca de mensagens. Assim, podemos avaliar melhor a quantidade de vezes que estas variáveis são atualizadas. Este número é equivalente, num sistema distribuído, ao número de mensagens trocadas entre processadores. O sistema PCSOS está implementado em linguagem C e a entrada de dados é realizada através de um arquivo texto, que contém o código de *indexicals*.

PCSOS apresenta alguns parâmetros para serem configurados antes da sua execução, como por exemplo, o número de soluções a serem encontradas, o tipo de particionamento estático (*Round-Robin* por restrições, por variáveis ou por *indexicals*, Blocos por restrições, por variáveis ou por *indexicals* e *Grouping-Sink* por *indexicals* ou por restrições), o momento de acesso às informações compartilhadas (imediatamente ou num ponto de sincronização), para quais processadores difundir as informações compartilhadas, para todos os processadores ou somente para o processador que possuir dependência com a variável alterada, entre outros. Em nossos experimentos encontramos somente a primeira solução do CSP. Realizamos experimentos com todos os tipos de particionamento citados.

Quando o domínio de uma variável é atualizado em memória compartilhada é necessário informar aos outros processadores qual a variável alterada e o novo domínio. Isto equivale a realizar o envio de mensagem num sistema distribuído. Como o PCSOS per-

mite que o usuário escolha o momento que a variável compartilhada será atualizada e quais os processadores serão comunicados, realizamos um estudo com estes parâmetros, que está detalhado no Apêndice D. Neste estudo foram executadas todas as aplicações descritas anteriormente com todos os particionamentos. Foram obtidos o tempo de execução (média de 10 execuções), o total de "mensagens" (quantidade de escrita em memória compartilhada da variável alterada e de seu novo valor de domínio) e o total de trabalho (total de execução de *indexicals* em cada processador).

Neste estudo foi concluído que para a execução em memória compartilhada a quantidade de escrita em memória compartilhada não interfere muito no tempo de execução. Mas queremos obter um particionamento que diminua a quantidade de comunicação entre os processadores para posterior execução num sistema distribuído. Então, adotamos a combinação que difunde informações num determinado ponto de sincronização e somente para o(s) processador(es) que possuir(em) dependências com aquela variável que teve seu domínio alterado. O processador que realiza a alteração no valor da variável não difunde as informações para si mesmo. Na maioria dos particionamentos esta combinação foi a melhor opção quando observamos o tempo de execução e o total de mensagens.

Utilizando os *benchmarks* e a plataforma de *hardware* e *software*, descritos neste capítulo, foram realizados os experimentos para validar o particionamento *Grouping-Sink*, que são apresentados no Capítulo 6.

Capítulo 6

Resultados experimentais e discussão

A avaliação do novo método para particionamento de restrições proposto foi realizada utilizando uma máquina de memória compartilhada com 14 processadores e o sistema PCSOS. As aplicações utilizadas foram *Arithmetic*, *Queens*, PBCSP com 50 e 100 variáveis e *Sudoku* (Capítulo 5). Para cada aplicação são apresentados os gráficos de tempo de execução, total de trabalho, total de mensagens, balanceamento de carga e número de falhas para 8 e 14 processadores, para cada tipo de particionamento (Blocos por variáveis, por *indexicals*, por restrições, *Round-Robin* por variáveis, por *indexicals*, por restrições e *Grouping-Sink* por *indexicals* e por restrições). Para cada aplicação foi encontrada somente a primeira solução e em todas as execuções a solução encontrada foi a mesma.

O tempo de execução apresentado nas tabelas encontra-se em segundos. Os valores que estão entre parênteses correspondem ao desvio padrão das 10 execuções que foram realizadas. O menor tempo de execução está destacado em negrito. E o menor tempo entre as duas implementações de *Grouping-Sink* está sublinhado.

O total de trabalho e o total de mensagens referem-se à soma do trabalho e mensagens de cada execução. Lembrando que numa máquina de memória compartilha uma mensagem corresponde à escrita em memória das informações que devem ser compartilhadas entre os processadores. A menor quantidade de trabalho/mensagens está ressaltada em negrito e a menor quantidade entre as implementações de *Grouping-Sink* está sublinhada.

O balanceamento de carga corresponde ao percentual de diferença entre a maior e a menor quantidade de trabalho realizado em cada processador. Assim, o cálculo é realizado da seguinte forma: $(\frac{maior\_valor - menor\_valor}{maior\_valor}) * 100$. Com este valor é possível determinar numa execução se a quantidade de trabalho realizada pelos processadores foi muito diferente. O tempo de execução é limitado pelo processador que realizar maior quantidade de trabalho. O menor valor, o maior valor e o menor percentual estão destacados em negrito e o menor percentual entre as implementações de *Grouping-Sink* está sublinhado.

O número de falhas corresponde à soma total de falhas em todos os processadores. A menor diferença entre o número de falhas nos processadores e o menor total de falhas

estão em negrito. Para as implementações de *Grouping-Sink*, a menor diferença entre o número de falhas nos processadores e o menor total de falhas estão sublinhados.

6.1 Introdução

Antes de apresentar os resultados obtidos para cada aplicação, é importante ressaltar a seguinte observação sobre o particionamento *Grouping-Sink*.

Cada vez que o algoritmo de orientação acíclica inicial é executado a orientação gerada é diferente. Conseqüentemente, o particionamento gerado com *Grouping-Sink* e o arquivo de entrada para o PCSOS são diferentes. Por isso, foi avaliada a qualidade dos particionamentos obtidos com *Grouping-Sink* por *indexicals* e por restrições. Utilizamos a aplicação *Arithmetic*, uma vez que esta apresentou as maiores discrepâncias. Consideramos que com um particionamento de boa qualidade a execução da aplicação no PCSOS apresenta tempo de execução menor. Assim, comparamos os tempos de execução no PCSOS para 10 particionamentos diferentes usando *Grouping-Sink* por *indexicals* e por restrições.

As Tabelas 6.1 e 6.2 apresentam os tempos de execução dos 10 particionamentos, gerados a partir de 10 orientações acíclicas iniciais diferentes (F0 a F9), para *Grouping-Sink* por *indexicals* e por restrições, respectivamente. Cada tempo de execução corresponde à média de 10 execuções no PCSOS e o tempo está em segundos.

	Número de Processadores					
Arquivos	1	2	4	8	12	14
F0	8,96	10,58	10,24	9,40	9,40	10,29
F1	8,98	11,90	10,25	9,69	10,05	11,05
F2	9,00	10,75	10,52	9,67	9,95	12,29
F3	8,99	10,58	11,05	10,17	10,73	11,65
F4	9,01	10,95	10,20	9,52	9,29	11,59
F5	9,01	10,33	9,71	9,50	9,96	11,26
F6	8,99	11,24	10,26	9,95	9,95	10,97
F7	9,04	10,59	8,76	8,95	9,10	12,14
F8	9,01	11,42	10,67	10,11	9,97	10,88
F9	8,98	10,36	9,72	9,55	9,59	12,65
Perc. entre maior e menor valor	0,88	13,19	20,72	12,00	15,19	18,66

Tabela 6.1: *Arithmetic* - Tempos de execução (s.) para *Grouping-Sink* por *indexicals*

Na Tabela 6.1, a diferença entre o maior e o menor tempo de execução dos arquivos é inferior a 21% e na Tabela 6.2 é inferior a 6,5%. O grafo considerado para executar *Grouping-Sink* por *indexicals* possui mais nós e arestas do que o grafo de *Grouping-Sink* por restrições. O grafo usado por *Grouping-Sink* por *indexicals* possui 1626 nós, enquanto que o grafo de *Grouping-Sink* por restrições possui 381 nós. Com grafos maiores o número de orientações acíclicas possíveis tende a ser maior, pois o número de orientações acíclicas está relacionado ao número de vértices do grafo [5].

A diferença entre o maior e o menor tempo de execução no PCSOS para os diferentes arquivos é pequena. Portanto, podemos escolher um ou outro arquivo com agrupamentos gerados pela decomposição em *sinks* para ser executado no PCSOS. Então, os arquivos que foram executados no PCSOS em nossos experimentos foram escolhidos aleatoriamente. Isso significa que nos resultados das próximas seções não foi escolhida a melhor orientação acíclica inicial.

	Número de Processadores					
Arquivos	1	2	4	8	12	14
F0	8,92	9,71	9,93	10,54	11,66	12,66
F1	8,99	9,55	9,79	10,48	11,45	12,05
F2	8,96	9,66	9,79	10,52	11,51	12,29
F3	8,95	9,74	9,79	10,46	11,40	12,33
F4	8,93	9,65	9,73	10,47	11,42	12,18
F5	9,01	9,80	9,98	10,47	11,39	12,32
F6	9,23	9,90	10,17	10,72	11,65	12,49
F7	8,94	9,70	9,95	10,46	11,59	12,49
F8	8,94	9,68	9,84	10,63	11,48	12,32
F9	9,07	9,75	9,65	10,06	11,06	12,01
Perc. entre maior e menor valor	3,36	3,54	5,11	6,16	5,15	5,13

Tabela 6.2: *Arithmetic* - Tempos de execução (s.) para *Grouping-Sink* por restrições

6.2 *Arithmetic*

A Figura 6.1 (tabela) apresenta os tempos de execução para a aplicação *Arithmetic* com todos os particionamentos. A diferença entre os tempos de execução dos oito particionamentos para 1 processador é pequena, menor do que 0,7%.

Para 2, 4 e 8 processadores o menor tempo de execução (Figura 6.1) é apresentado por *Round-Robin* por restrições. Mas com 12 processadores *Round-Robin* por *indexicals* apresenta o menor tempo de execução e com 14 processadores o menor tempo é de *Round-Robin* por variáveis.

Comparando o tempo de execução de *Grouping-Sink* por *indexicals* com *Grouping-Sink* por restrições percebemos que para 2 e 4 processadores *Grouping-Sink* por restrições apresenta o menor tempo e para 8, 12 e 14 processadores o menor tempo é de *Grouping-Sink* por *indexicals*.

Percebe-se na Figura 6.1 que para 2, 4 e 8 processadores há um desempenho dos particionamentos um pouco diferente do que o obtido para 12 e 14 processadores. Então, uma análise diferenciada pode ser feita até 8 processadores e depois para 12 e 14 processadores.

A Figura 6.2 apresenta a quantidade de trabalho executado. Com 1 processador todos os particionamentos apresentam a mesma quantidade de trabalho. Para 2, 4 e 8 processadores, Blocos por variáveis realizou menos quantidade de trabalho em relação aos demais particionamentos. Observe que *Grouping-Sink* por restrições apresenta resultados

similares a Blocos por variáveis. Enquanto que para 12 e 14 processadores *Grouping-Sink* por restrições apresenta a menor quantidade de trabalho. Este resultado também está refletido no gráfico de mensagens (Figura 6.3). Para 2 e 4 processadores Blocos por variáveis apresentou a menor quantidade de mensagens. Mas para 8, 12 e 14 processadores *Grouping-Sink* por restrições apresentou a menor quantidade de mensagens.

Apesar de terem as menores quantidades de trabalho e mensagens, Blocos por variáveis e *Grouping-Sink* por restrições não apresentam o menor tempo de execução. Este fato pode ser melhor entendido através dos gráficos de falhas e de balanceamento de carga. Para facilitar a análise, escolhemos a execução com 8 processadores, que possui um comportamento similar à execução para 2 e 4 processadores, e a execução com 14 processadores (similar a 12 processadores).

A Figura 6.4 mostra a quantidade de *indexicals* executados em cada um dos 8 processadores. Estão destacados em negrito a maior e a menor quantidade de *exec-ind* (*indexical* executado) de cada particionamento. Na última linha é apresentada a diferença percentual entre o maior e o menor número de *indexicals* executados. Num balanceamento de carga ótimo, todos os processadores executariam o mesmo número de *indexicals* e a diferença entre eles seria de 0%. Na Figura 6.4, o particionamento que apresenta o melhor balanceamento de carga para 8 processadores é *Round-Robin* por *indexicals* (a diferença percentual entre o maior e o menor número de *indexicals* executados é 27,42%). Mas, além do total de mensagens, total de trabalho e balanceamento de carga, a quantidade de falhas ocorridas em cada processador influencia no tempo de execução total. Pois, o tempo de execução total é relativo ao processador que gastou mais tempo para terminar a execução.

A última linha das Figuras 6.6 e 6.7 apresenta a diferença entre o processador com maior número de falhas e aquele com menor número de falhas, para 8 e 14 processadores, respectivamente. *Round-Robin* por *indexicals* apresentou a menor diferença entre o número de falhas ocorridas para 8 processadores. Para 14 processadores, *Round-Robin* por restrições apresentou a menor diferença. Mas o número total de falhas na execução (soma de todos os processadores) não é o menor. Isso pode justificar o fato desse particionamento ter um balanceamento de carga melhor.

Numa máquina de memória compartilhada o tempo gasto em escrita em memória compartilhada (mensagem) não é significativo quando comparado com a troca de mensagens em uma máquina com memória distribuída. Por isso, os particionamentos *Round-Robin* por variáveis, *indexicals* e restrições apresentaram melhores tempos de execução mesmo tendo um total de mensagens (Figura 6.3) bem mais alto do que os demais particionamentos. Para 8 processadores o fator que está sendo determinante no tempo de execução é o balanceamento de carga. A pequena diferença entre *Round-Robin* por *in-*

dexicals e por restrições deve-se à quantidade de trabalho e total de falhas. *Round-Robin* por restrições realiza cerca de 6% menos de trabalho do que *Round-Robin* por *indexicals* e o total de falhas é cerca de 6,5% menor.

Embora o nosso foco no momento não seja a obtenção de *speedup*, *Arithmetic* apresenta um pequeno *speedup* somente para 4, 8 e 12 processadores. O grafo de restrições padrão desta aplicação é fracamente conectado. Os particionamentos que agrupam restrições pelas variáveis (Blocos por variáveis, *Grouping-Sink* por restrições) tendem a apresentar menor número de mensagens, pois conseguem aumentar a computação local e diminuir a comunicação. Estes particionamentos apresentam menor número de falhas, mas não possuem bom balanceamento de carga.

Para facilitar o entendimento destes resultados, as Figuras 6.8 e 6.9 apresentam um resumo dos resultados de tempo de execução, total de trabalho, total de mensagens, balanceamento de carga e número de falhas para cada particionamento para a execução com 8 e 14 processadores. A tabela superior mostra a legenda dos símbolos das faixas de valores dos parâmetros. As faixas variam do melhor resultado (+ + + +) para o pior resultado (- - - -). O critério adotado para construir esta tabela foi baseado nos resultados obtidos dos 8 particionamentos. O tamanho do intervalo é encontrado subtraindo o maior valor pelo menor valor e dividindo pelo número de particionamentos (8).

Na tabela inferior da Figura 6.8, Blocos por variáveis e *Grouping-Sink* por restrições apresentam os melhores resultados de uma forma geral para 8 processadores. Isto é, apresentam a maior quantidade de "+". Porém, *Grouping-Sink* por restrições, em relação a *Round-Robin* por restrições (que apresentou o menor tempo de execução), possui a vantagem de reduzir o total de mensagens em mais de 85% com um atraso no tempo de execução total de apenas 31%. Em um sistema distribuído o ganho de 85% em troca de mensagens é muito significativo. Vale ressaltar, também, que não foi escolhida a melhor orientação acíclica inicial para executar *Grouping-Sink* por restrições. Na Tabela 6.1 percebemos que particionamentos gerados a partir de orientações acíclicas iniciais diferentes produzem tempos de execução diferentes. O arquivo que utilizamos foi $F0$ e este não apresentou os mais baixos tempos de execução. Isso significa que é possível obter um resultado ainda melhor com *Grouping-Sink* por restrições utilizando uma orientação acíclica diferente. Uma orientação acíclica pior poderia gerar resultados ainda piores.

Para 14 processadores (Figura 6.9) *Grouping-Sink* por restrições tem um ganho em troca de mensagens de 81,2% e um atraso no tempo de execução de apenas 23,5%, em relação a *Round-Robin* por variáveis, que tem o menor tempo de execução.

Arithmetic - tempo de execução total																
Procs.	Blocos						*Round-Robin*						*Grouping-Sink*			
	Variáveis		*Indexicals*		Restrições		Variáveis		*Indexicals*		Restrições		*Indexicals*		Restrições	
1	8,97	(0,05)	8,97	(0,04)	8,93	(0,02)	8,95	(0,02)	8,95	(0,03)	8,93	(0,02)	8,96	(0,04)	**8,92**	(0,02)
2	9,83	(0,10)	9,83	(0,06)	9,88	(0,16)	10,95	(0,06)	10,67	(0,19)	**9,19**	(0,19)	10,58	(0,08)	9,71	(0,05)
4	10,05	(0,17)	9,96	(0,09)	9,94	(0,08)	9,45	(0,05)	8,04	(0,18)	**6,93**	(0,12)	10,24	(0,11)	9,93	(0,09)
8	10,69	(0,06)	10,59	(0,09)	10,59	(0,08)	8,50	(0,38)	7,65	(0,20)	**7,28**	(0,13)	9,40	(0,12)	10,54	(0,06)
12	12,01	(0,18)	11,11	(0,08)	11,53	(0,07)	8,67	(0,06)	**8,40**	(0,30)	9,30	(0,05)	9,40	(0,02)	11,66	(0,36)
14	13,65	(0,24)	11,31	(0,26)	11,27	(0,27)	**9,66**	(0,24)	9,68	(0,24)	11,66	(0,29)	10,29	(0,29)	12,66	(0,27)

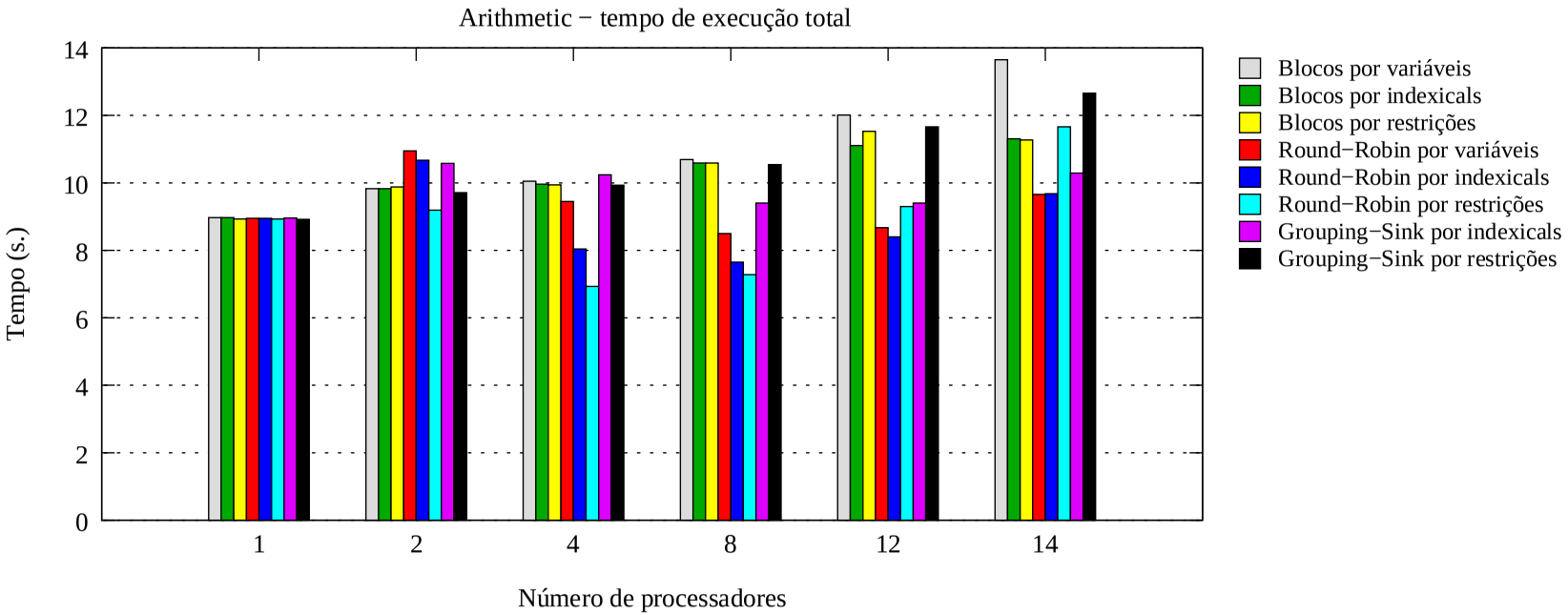


Figura 6.1: *Arithmetic* - tempo de execução

Arithmetic - total de trabalho								
Procs.	Blocos			*Round-Robin*			*Grouping-Sink*	
	Variáveis	*Indexicals*	Restrições	Variáveis	*Indexicals*	Restrições	*Indexicals*	Restrições
1	1.953.660	1.953.660	1.953.660	1.953.660	1.953.660	1.953.660	1.953.660	1.953.660
2	**1.969.644**	2.377.467	2.377.579	3.210.551	3.126.205	2.723.582	2.556.185	1.970.964
4	**2.001.559**	2.956.427	2.952.055	4.448.041	4.140.722	3.673.889	3.305.636	2.005.024
8	**2.067.224**	3.767.013	3.752.547	4.898.001	5.217.647	4.914.465	4.051.532	2.069.955
12	2.468.378	3.491.112	3.552.485	4.644.974	4.876.680	5.674.234	4.234.548	**2.165.583**
14	2.799.952	2.804.473	2.773.651	4.705.374	4.969.890	5.579.794	4.566.031	**2.196.129**

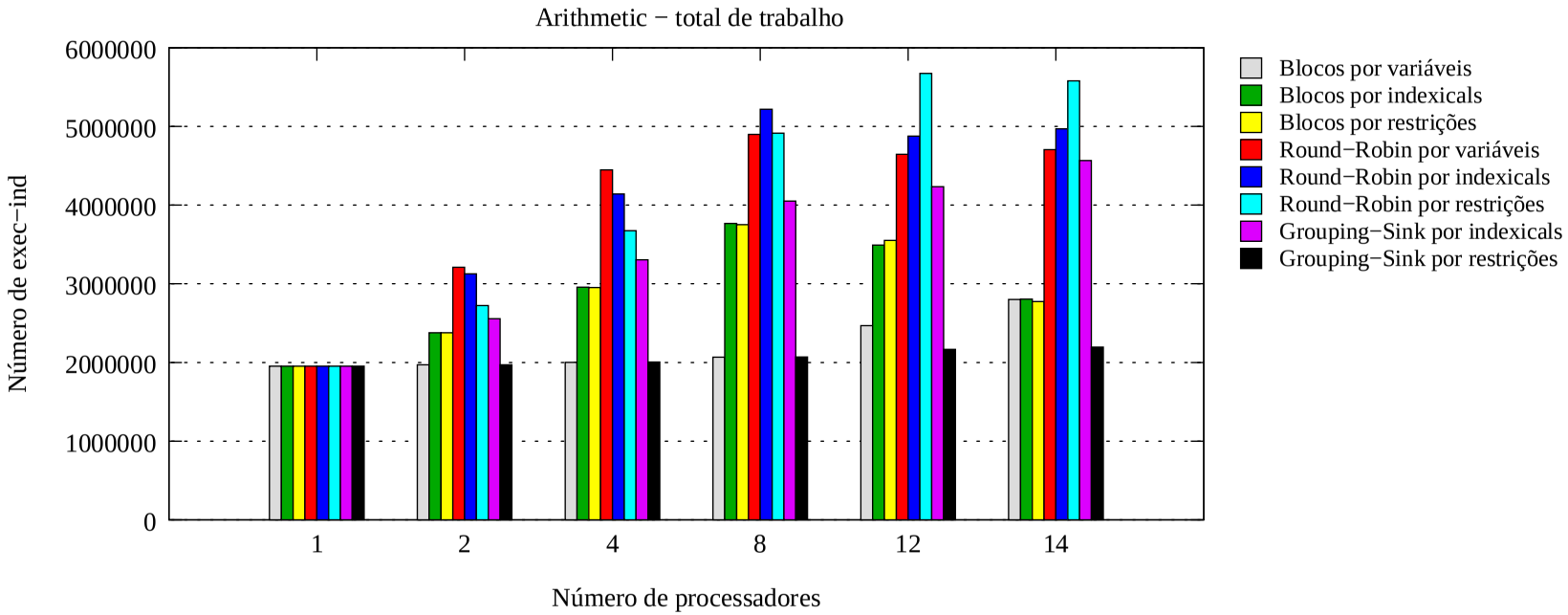


Figura 6.2: *Arithmetic* - total de trabalho

Arithmetic - total de mensagens								
Procs.	Blocos			*Round-Robin*			*Grouping-Sink*	
	Variáveis	*Indexicals*	Restrições	Variáveis	*Indexicals*	Restrições	*Indexicals*	Restrições
1	0	0	0	0	0	0	0	0
2	**30.334**	32.190	32.193	66.083	60.229	48.675	52.130	30.556
4	**91.393**	114.044	114.038	398.075	321.960	317.922	299.214	92.335
8	237.379	298.679	298.643	1.554.018	1.481.530	1.490.863	840.461	**215.805**
12	477.065	546.217	528.219	2.139.567	3.474.695	3.991.929	1.508.569	**388.156**
14	603.837	595.131	530.672	2.437.810	4.829.650	6.072.948	1.729.617	**459.106**

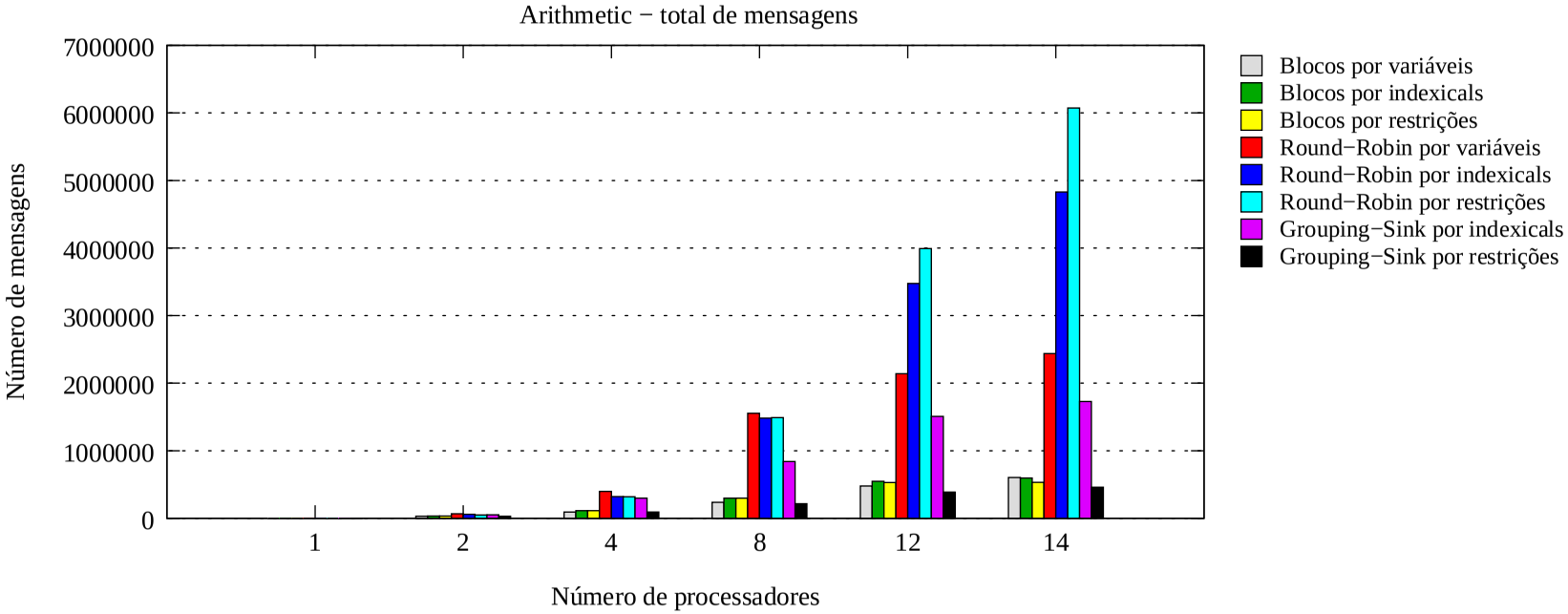


Figura 6.3: *Arithmetic* - total de mensagens

Arithmetic - balanceamento de carga para 8 processadores								
Procs.	Blocos			*Round-Robin*			*Grouping-Sink*	
	Variáveis	*Indexicals*	Restrições	Variáveis	*Indexicals*	Restrições	*Indexicals*	Restrições
0	437.122	416.548	416.069	621.184	**503.452**	494.350	**255.923**	464.613
1	610.432	1.023.512	1.023.921	833.125	666.659	663.001	582.563	22.083
2	**723.335**	**1.063.409**	**1.052.768**	810.771	675.395	678.490	338.075	91.089
3	37.586	287.332	280.888	**854.982**	684.445	662.055	500.744	382.791
4	54.443	95.462	95.622	783.816	682.156	**704.913**	724.386	**796.997**
5	50.946	706.247	708.278	824.595	**693.682**	**378.116**	497.507	275.425
6	**20.868**	**41.440**	**41.936**	85.202	649.453	683.303	415.157	19.158
7	132.492	133.063	133.065	**84.326**	662.405	650.237	**737.177**	**17.799**
Diferença percentual	97,12	96,10	96,02	90,14	**27,42**	46,36	65,28	97,77

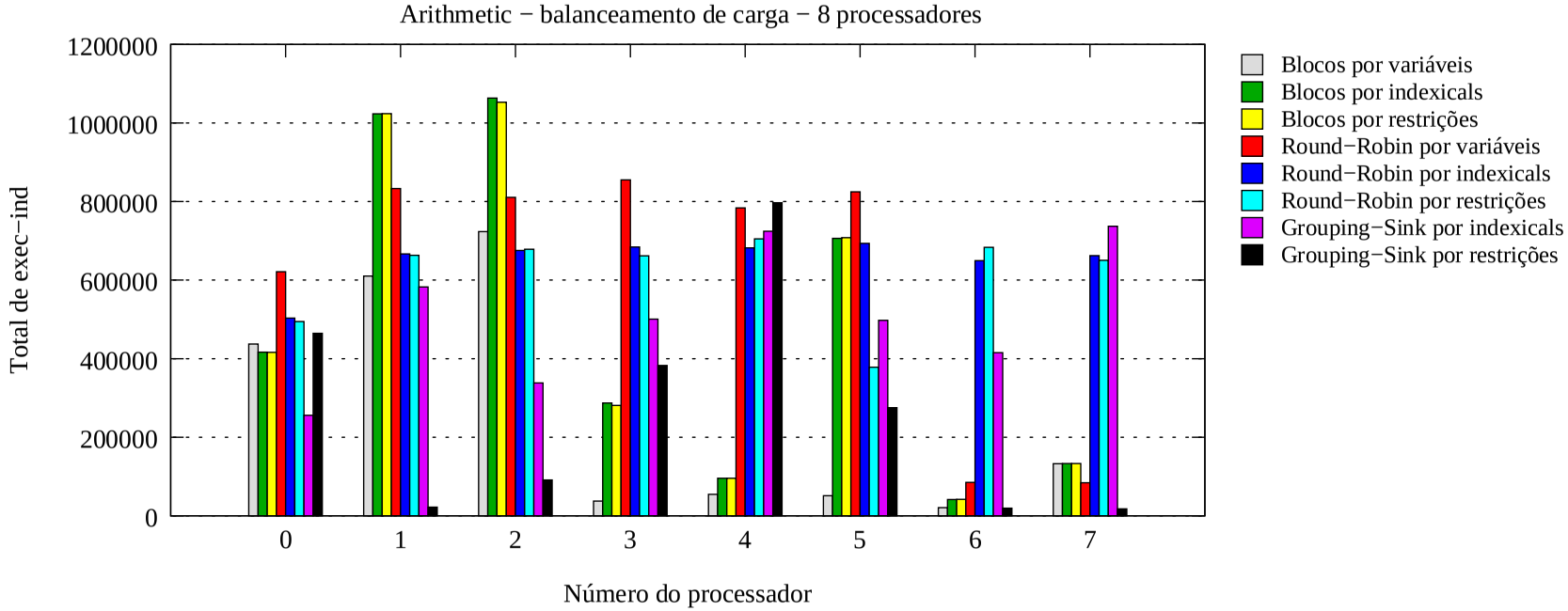


Figura 6.4: *Arithmetic* - balanceamento de carga para 8 processadores

Arithmetic - balanceamento de carga para 14 processadores								
Procs.	Blocos			*Round-Robin*			*Grouping-Sink*	
	Variáveis	*Indexicals*	Restrições	Variáveis	*Indexicals*	Restrições	*Indexicals*	Restrições
0	107.392	78.237	22.296	381.847	**260.708**	**279.904**	237.546	408.014
1	637.444	**569.230**	493.885	473.081	360.956	381.152	67.865	18.660
2	564.965	447.917	415.819	467.898	366.494	458.470	300.919	38.424
3	213.729	272.710	271.329	461.611	371.562	396.874	136.543	**16.000**
4	**782.912**	514.376	**567.777**	187.985	369.380	379.724	231.441	17.292
5	110.027	505.473	566.590	176.227	332.853	350.886	218.533	24.106
6	**16.861**	37.025	37.429	**110.821**	349.250	409.913	485.563	95.322
7	17.223	**19.571**	**17.428**	162.935	356.827	365.257	636.280	18.751
8	59.853	59.595	54.129	212.790	358.279	465.172	186.141	64.922
9	62.276	76.922	78.224	224.312	359.853	408.658	379.160	362.515
10	33.567	49.740	50.209	309.113	365.860	407.718	**40.309**	143.961
11	20.006	29.686	30.165	340.040	372.672	399.963	**812.962**	37.534
12	134.816	109.504	119.493	597.627	353.680	**460.306**	505.907	**681.996**
13	38.881	34.487	48.878	**599.087**	**391.516**	415.797	326.862	268.632
Diferença percentual	97,85	96,56	96,93	81,50	**33,41**	39,19	95,04	97,65

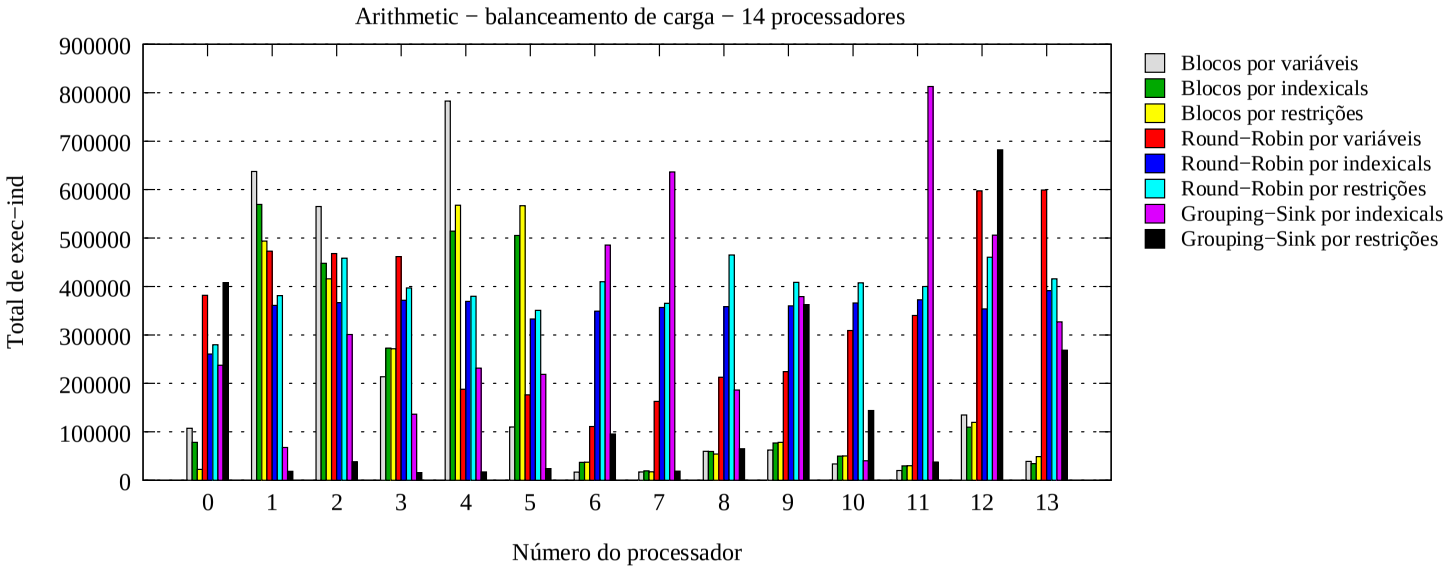


Figura 6.5: *Arithmetic* - balanceamento de carga para 14 processadores

Arithmetic - número de falhas para 8 processadores								
Procs.	Blocos			*Round-Robin*			*Grouping-Sink*	
	Variáveis	*Indexicals*	Restrições	Variáveis	*Indexicals*	Restrições	*Indexicals*	Restrições
0	3.403	31	23	5.867	1.548	1.715	399	3.613
1	4.622	7.699	7.929	2.630	3.315	1.388	1.699	27
2	3.202	3.604	3.050	1.614	1.861	4.537	532	432
3	221	1.429	93	1.226	1.012	2.310	3.210	2.535
4	234	345	328	2.129	826	556	2.721	3.490
5	280	1.483	1.806	567	1.659	15	332	2.330
6	16	42	231	421	2.303	1.722	2.546	11
7	460	460	460	239	1.137	509	1.552	0
Diferença entre < e >	4.606	7.668	7.906	5.628	2.489	4.522	2.878	3.613
Total	**12.438**	15.093	13.920	14.693	13.661	12.752	12.991	**12.438**

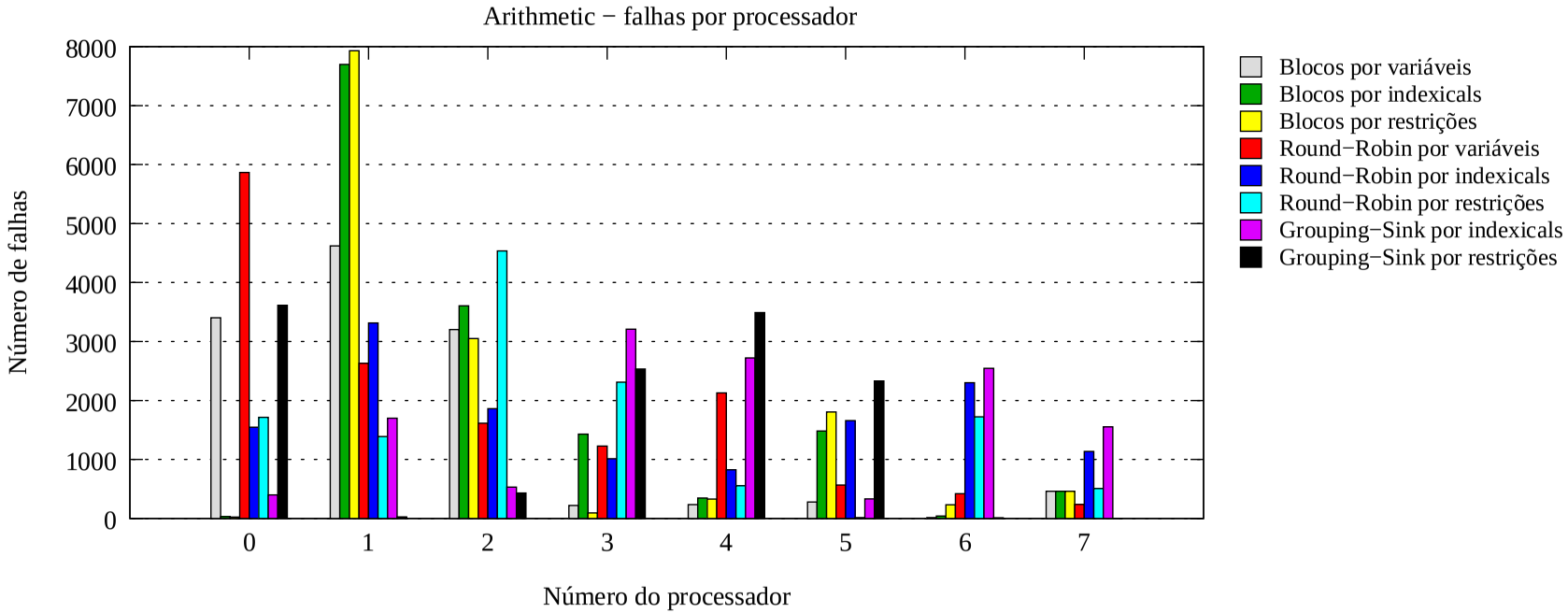


Figura 6.6: *Arithmetic* - número de falhas com 8 processadores

Arithmetic - número de falhas para 14 processadores								
Procs.	Blocos			*Round-Robin*			*Grouping-Sink*	
	Variáveis	*Indexicals*	Restrições	Variáveis	*Indexicals*	Restrições	*Indexicals*	Restrições
0	22	22	22	1.978	365	2.101	852	3.381
1	4.988	4.113	4.073	2.975	2.513	2.295	464	5
2	2.647	2.380	2.583	2.112	1.258	977	1.514	208
3	499	1.645	1.519	288	799	858	605	0
4	2.975	3.013	3.002	907	2.259	1.720	2.166	2
5	317	255	359	429	657	1.381	342	22
6	0	134	132	489	662	1.218	1.723	395
7	2	2	2	1.326	639	713	579	22
8	292	229	150	777	1.159	662	104	0
9	207	211	394	623	908	631	2.731	2.314
10	14	149	71	758	1.233	494	47	510
11	11	26	54	998	810	579	1.792	221
12	443	438	73	686	307	1.621	514	3.050
13	22	24	391	545	820	415	14	2.308
Diferença entre < e >	4.988	4.111	4.071	2.687	2.206	1.880	2.717	3.381
Total	12.439	12.641	12.825	14.891	14.389	15.665	13.447	**12.438**

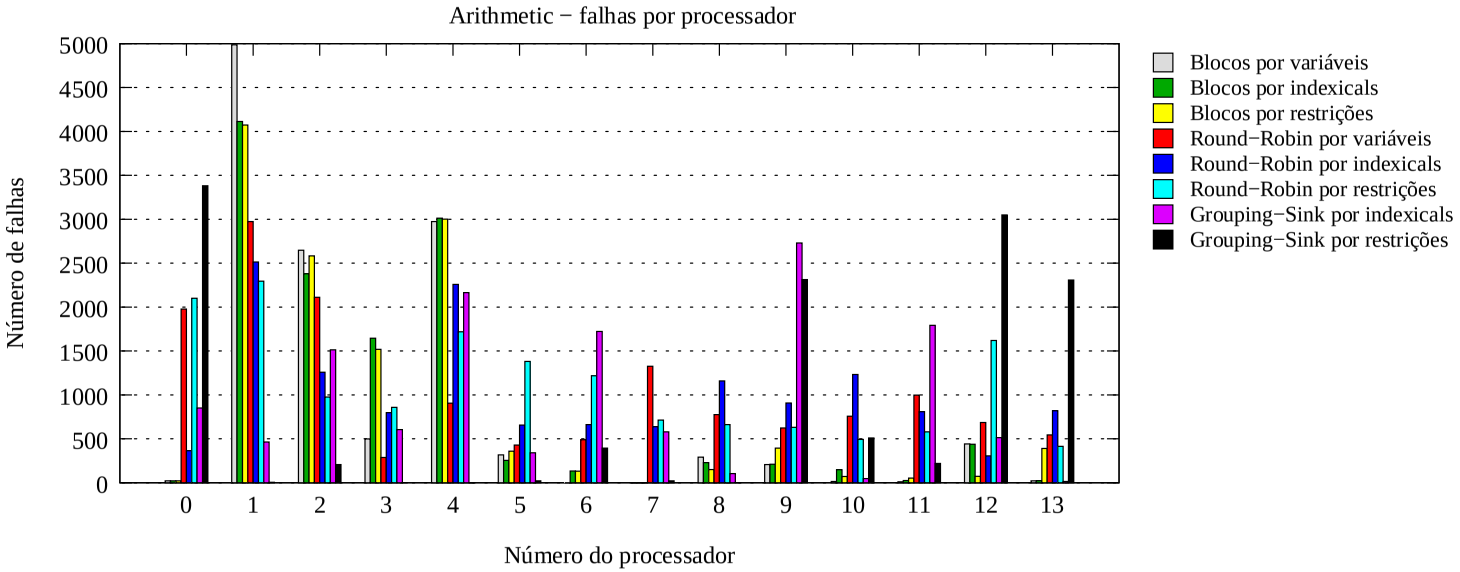


Figura 6.7: *Arithmetic* - número de falhas com 14 processadores

Símbolo	Tempo de execução	Total de trabalho	Total de mensagens	Balanceamento de carga (%)	Número de falhas
+ + + +	7,28 a 7,71	2.067.224 a 2.461.027	215.805 a 383.082	27,42 a 36,21	12.438 a 12.769
+ + +	7,72 a 8,13	2.461.028 a 2.854.831	393.083 a 550.359	36,22 a 45,00	12.770 a 13.101
+ +	8,14 a 8,56	2.854.832 a 3.248.634	550.360 a 717.637	45,01 a 53,79	13.102 a 13.433
+	8,57 a 8,98	3.248.635 a 3.642.438	717.638 a 884.915	53,80 a 62,58	13.434 a 13.765
-	8,99 a 9,41	3.642.439 a 4.036.241	884.916 a 1.052.192	62,59 a 71,37	13.766 a 14.097
- -	9,42 a 9,84	4.036.242 a 4.430.044	1.052.193 a 1.219.469	71,38 a 80,16	14.098 a 14.429
- - -	9,85 a 10,26	4.430.045 a 4.823.847	1.219.470 a 1.386.746	80,17 a 88,95	14.430 a 14.761
- - - -	10,27 a 10,69	4.823.848 a 5.217.647	1.386.746 a 1.554.018	88,96 a 97,77	14.762 a 15.093

Particionamento	Tempo de execução	Total de trabalho	Total de mensagens	Balanceamento de carga	Número de falhas	Total geral
Blocos por variáveis	- - - -	+ + + +	+ + + +	- - - -	+ + + +	**+4**
Blocos por *indexicals*	- - - -	-	+ + + +	- - - -	- - - -	-9
Blocos por restrições	- - - -	-	+ + + +	- - - -	-	-6
Round-Robin por variáveis	+ +	- - - -	- - - -	- - - -	- - -	-13
Round-Robin por *indexicals*	+ + + +	- - - -	- - - -	+ + + +	+	+1
Round-Robin por restrições	+ + + +	- - - -	- - - -	+ +	+ + + +	+2
Grouping-Sink por *indexicals*	-	- -	+	-	+ + +	0
Grouping-Sink por restrições	- - - -	+ + + +	+ + + +	- - - -	+ + + +	**+4**

Figura 6.8: *Arithmetic* - resumo para 8 processadores

Símbolo	Tempo de execução	Total de trabalho	Total de mensagens	Balanceamento de carga (%)	Número de falhas
+ + + +	9,66 a 10,16	2.196.129 a 2.619.087	459.106 a 1.160.836	33,41 a 41,46	12.438 a 12.841
+ + +	10,17 a 10,66	2.619.088 a 3.042.045	1.160.837 a 1.862.566	41,47 a 49,52	12.842 a 13.244
+ +	10,67 a 11,16	3.042.046 a 3.465.003	1.862.567 a 2.564.296	49,53 a 57,57	13.245 a 13.648
+	11,17 a 11,65	3.465.004 a 3.887.961	2.564.297 a 3.266.027	57,58 a 65,63	13.649 a 14.051
-	11,66 a 12,15	3.887.962 a 4.310.919	3.266.028 a 3.967.757	65,64 a 73,69	14.052 a 14.454
- -	12,16 a 12,65	4.310.920 a 4.733.877	3.967.758 a 4.669.487	73,70 a 81,74	14.455 a 14.858
- - -	12,66 a 13,15	4.733.878 a 5.156.835	4.669.488 a 5.371.217	81,75 a 89,80	14.859 a 15.261
- - - -	13,16 a 13,65	5.156.836 a 5.579.794	5.371.218 a 6.072.948	89,81 a 97,85	15.262 a 15.665

Particionamento	Tempo de execução	Total de trabalho	Total de mensagens	Balanceamento de carga	Número de falhas	
Blocos por variáveis	- - - -	+ + +	+ + + +	- - - -	+ + + +	+3
Blocos por *indexicals*	+	+ + +	+ + + +	- - - -	+ + + +	**+8**
Blocos por restrições	+	+ + +	+ + + +	- - - -	+ + + +	**+8**
Round-Robin por variáveis	+ + + +	- -	+ +	- -	- - -	-1
Round-Robin por *indexicals*	+ + + +	- - -	- - -	+ + + +	-	+1
Round-Robin por restrições	-	- - - -	- - - -	+ + + +	- - - -	-9
Grouping-Sink por *indexicals*	+ + +	- -	+ + +	- - - -	+ +	+2
Grouping-Sink por restrições	- - -	+ + + +	+ + + +	- - - -	+ + + +	+5

Figura 6.9: *Arithmetic* - resumo para 14 processadores

6.3 *Queens*

Na Figura 6.10 são apresentados os tempos de execução obtidos para *Queens* com todos os particionamentos. Para 1 processador a diferença no tempo de execução entre os particionamentos é de 2,2%.

Para 2, 4, 8 e 14 processadores o menor tempo de execução é obtido com Blocos por variáveis. Para 12 processadores o menor tempo é de *Round-Robin* por restrições.

Blocos por *indexicals* e por restrições apresentam menor quantidade de trabalho (Figura 6.11) e de mensagens (Figura 6.12), mas os piores balanceamentos de carga (Figuras 6.13 e 6.15).

Observando o gráfico de número de falhas para 8 processadores (Figura 6.14), percebemos que Blocos por variáveis (menor tempo de execução) apresenta menor total de falhas (30% a menos) do que *Round-Robin* por restrições, que apresenta o segundo melhor tempo de execução para 8 processadores.

Para 14 processadores, o segundo melhor tempo é de *Grouping-Sink* por *indexicals*. A diferença percentual no número total de falhas para 14 processadores entre Blocos por variáveis e *Grouping-Sink* por *indexicals* é de 1,7%.

Para *Queens*, o balanceamento de carga e o total de falhas são os fatores que causam maior impacto no tempo de execução. Como *Grouping-Sink* por *indexicals* teve 0,3% e 1,7% de falhas a mais que Blocos por variáveis, para 8 (Figuras 6.13 e 6.14) e 14 processadores (Figuras 6.15 e 6.16), o impacto no tempo de execução foi de 10,8 % (8 processadores) e 4,4% (14 processadores).

O grafo de restrições padrão de *Queens* é fortemente conectado, ou seja, há muita dependência entre os nós. Por isso, Blocos por variáveis, que agrupa os *indexicals* em função das variáveis que eles explicitam, apresenta melhores tempos de execução e menos falhas.

As Figuras 6.17 e 6.18 apresentam um resumo dos resultados de tempo de execução, total de trabalho, total de mensagens, balanceamento de carga e número de falhas para *Queens* com todos os particionamentos.

Para 8 processadores (Figura 6.17), o melhor particionamento é Blocos por variáveis. Mas *Grouping-Sink* por *indexicals* é o segundo melhor particionamento. A diferença percentual em número de mensagens entre estes dois particionamentos é de 9,1% e em tempo de execução é 10,84%, sendo que em ambos os casos Blocos por variáveis é o melhor.

Para 14 processadores (Figura 6.18), o melhor particionamento também é Blocos por variáveis e o segundo melhor é *Grouping-Sink* por *indexicals*. A diferença percentual em número de mensagens é 5,6% e em tempo de execução é 26,2%.

Queens - tempo de execução total																
Procs.	Blocos						*Round-Robin*						*Grouping-Sink*			
	Variáveis		*Indexicals*		Restrições		Variáveis		*Indexicals*		Restrições		*Indexicals*		Restrições	
1	9,77	(0,06)	9,98	(0,26)	**9,76**	(0,04)	9,78	(0,09)	9,77	(0,06)	9,83	(0,16)	9,84	(0,11)	**9,77**	(0,05)
2	**7,96**	(0,06)	8,14	(0,06)	8,23	(0,11)	10,97	(0,09)	13,89	(0,39)	9,13	(0,07)	10,38	(0,05)	8,16	(0,03)
4	**5,93**	(0,05)	6,91	(0,09)	7,01	(0,18)	7,59	(0,34)	10,38	(0,21)	6,68	(0,50)	7,41	(0,41)	6,89	(0,05)
8	**5,18**	(0,04)	6,78	(0,22)	6,69	(0,06)	5,88	(0,42)	7,89	(0,07)	5,66	(0,65)	5,81	(0,08)	6,96	(0,25)
12	6,64	(0,83)	7,81	(0,05)	7,81	(0,04)	6,02	(0,08)	7,79	(0,04)	**5,45**	(1,61)	6,04	(0,04)	7,61	(0,32)
14	**6,36**	(0,50)	9,08	(0,69)	8,95	(0,59)	6,86	(0,71)	8,71	(0,58)	6,92	(0,68)	6,65	(0,52)	8,62	(0,21)

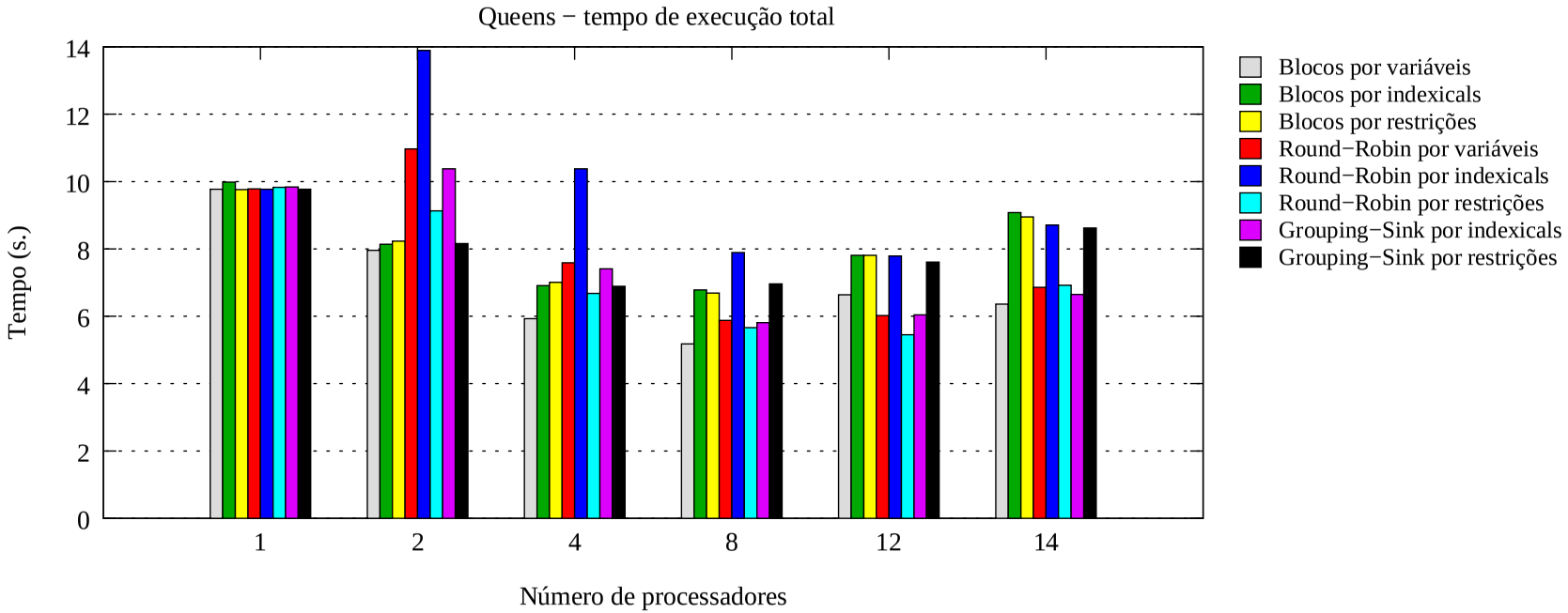


Figura 6.10: *Queens* - tempo de execução

Queens - total de trabalho								
Procs.	Blocos			*Round-Robin*			*Grouping-Sink*	
	Variáveis	*Indexicals*	Restrições	Variáveis	*Indexicals*	Restrições	*Indexicals*	Restrições
1	718.886	718.886	718.886	718.886	718.886	718.886	718.886	718.886
2	2.296.769	**727.546**	**727.546**	2.992.435	5.577.763	795.830	2.791.964	742.678
4	3.774.585	**730.065**	730.140	4.694.513	6.434.491	819.023	4.615.969	767.923
8	5.304.876	758.670	**756.210**	5.655.351	6.778.171	857.853	5.664.988	812.508
12	5.453.743	829.432	**828.100**	6.043.515	6.944.218	889.662	6.124.502	875.936
14	5.971.573	**843.704**	844.067	6.276.625	6.992.144	903.490	6.261.716	897.645

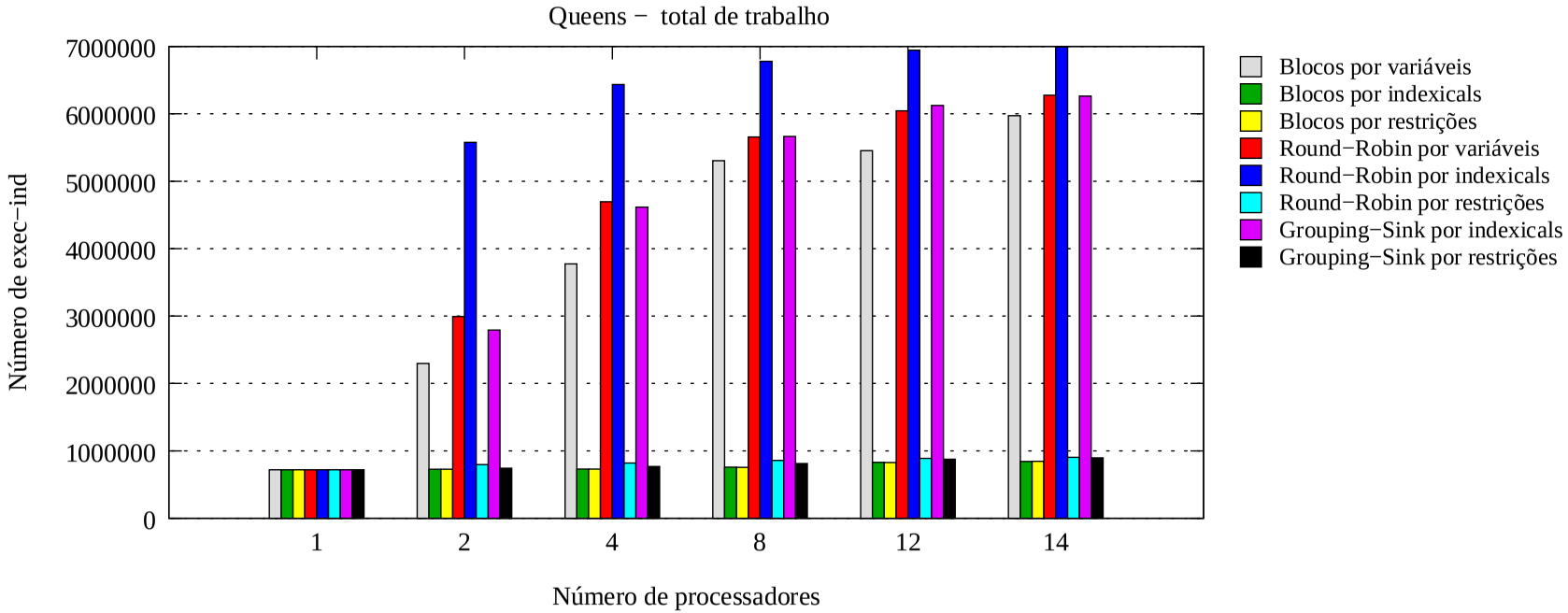


Figura 6.11: *Queens* - total de trabalho

Queens - total de mensagens								
Procs.	Blocos			*Round-Robin*			*Grouping-Sink*	
	Variáveis	*Indexicals*	Restrições	Variáveis	*Indexicals*	Restrições	*Indexicals*	Restrições
1	0	0	0	0	0	0	0	0
2	46.395	**44.063**	**44.063**	69.913	65.299	98.298	64.621	72.246
4	209.922	203.722	**203.715**	299.112	364.498	403.152	276.510	308.892
8	852.173	764.031	**762.785**	961.653	1.191.335	1.241.590	937.167	986.292
12	1.690.744	1.629.903	**1.629.449**	1.899.755	2.270.193	2.362.987	1.846.097	1.954.422
14	2.320.578	2.161.071	**2.160.905**	2.491.632	2.926.216	3.039.322	2.458.898	2.521.202

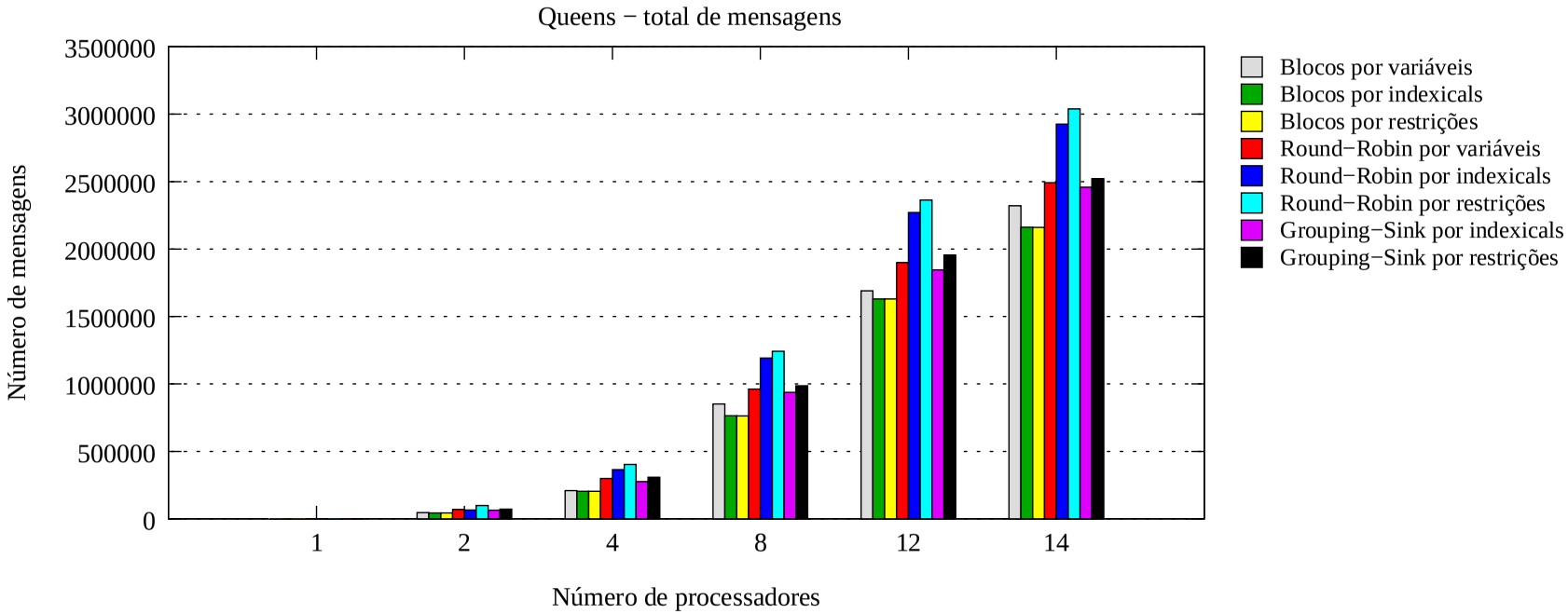


Figura 6.12: *Queens* - total de mensagens

Queens - balanceamento de carga para 8 processadores								
Procs.	Blocos			*Round-Robin*			*Grouping-Sink*	
	Variáveis	*Indexicals*	Restrições	Variáveis	*Indexicals*	Restrições	*Indexicals*	Restrições
0	740.808	11.012	10.964	585.031	625.371	102.385	645.078	198.128
1	761.656	10.957	10.964	740.141	1.074.968	111.276	611.135	129.162
2	651.121	10.957	10.963	722.564	641.370	97.936	735.626	107.374
3	760.151	10.957	10.963	780.524	1.067.437	98.331	767.800	124.986
4	542.836	145.575	148.288	716.815	637.029	103.570	729.389	57.881
5	530.174	10.956	10.963	715.038	1.053.599	106.274	655.797	87.722
6	653.179	313.399	314.649	711.224	626.278	118.255	767.338	61.429
7	664.951	244.857	238.456	684.014	1.052.119	119.826	752.825	45.826
Diferença percentual	30,39	96,50	96,52	25,05	41,82	**18,27**	20,40	76,87

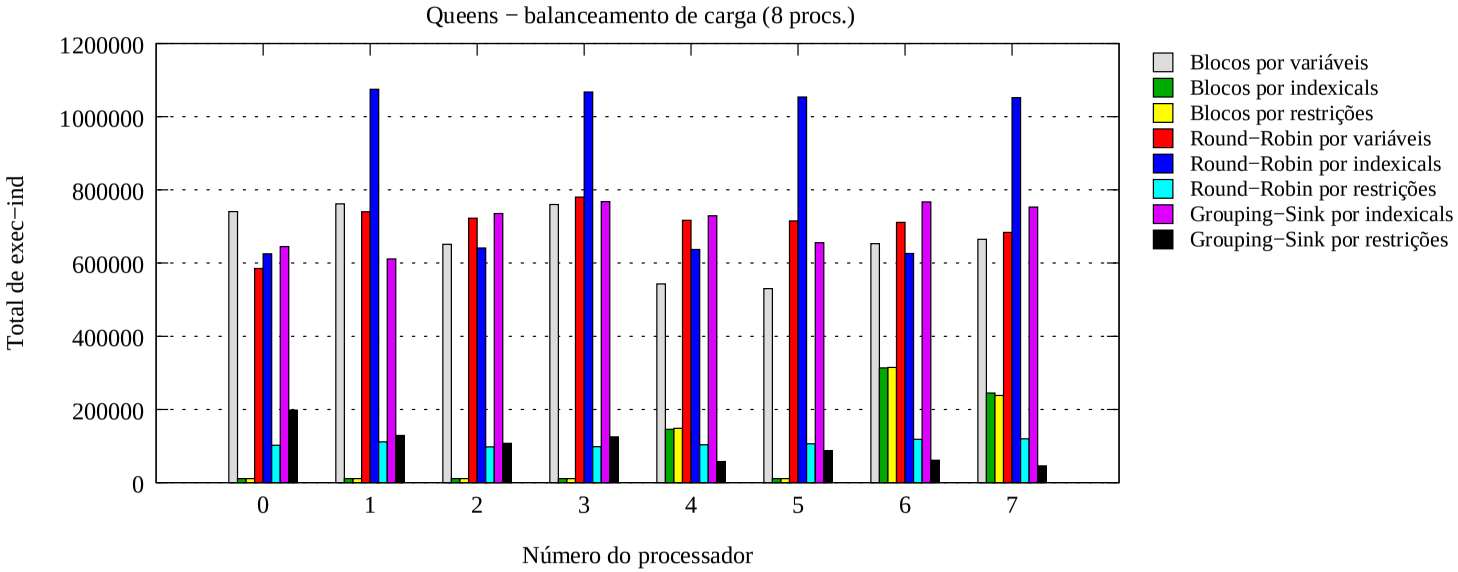


Figura 6.13: *Queens* - balanceamento de carga para 8 processadores

Queens - número de falhas para 8 processadores								
Procs.	Blocos			*Round-Robin*			*Grouping-Sink*	
	Variáveis	*Indexicals*	Restrições	Variáveis	*Indexicals*	Restrições	*Indexicals*	Restrições
0	0	1	3	966	1.281	1.560	1.191	1.473
1	0	2	4	431	568	989	675	871
2	144	2	3	1.035	814	861	1.185	791
3	0	2	2	590	481	682	298	1.010
4	560	212	232	502	779	649	598	293
5	1.078	0	0	638	366	598	508	179
6	1.302	1.485	1.492	278	562	500	77	126
7	1.406	2.778	2.747	179	354	580	0	35
Diferença entre < e >	1.406	2.778	2.747	856	927	1.060	1.191	1.438
Total	4.490	**4.482**	4.483	4.619	5.205	6.419	4.532	4.778

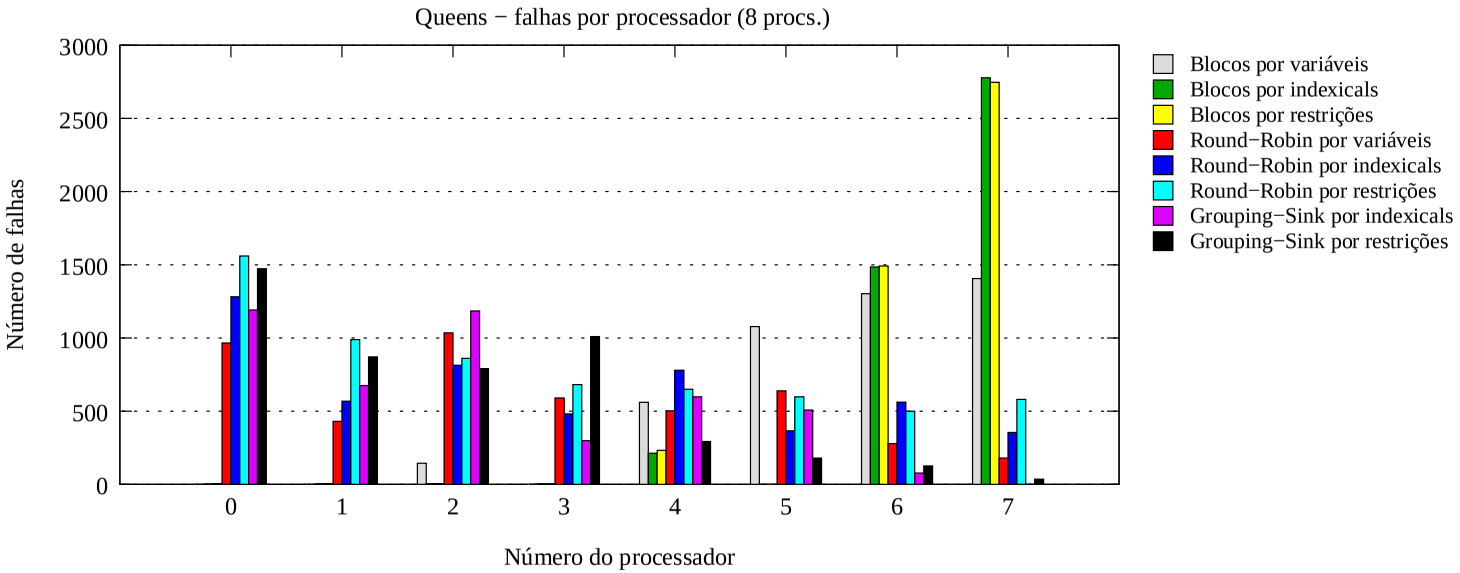


Figura 6.14: *Queens* - total de falhas para 8 processadores

Queens - balanceamento de carga para 14 processadores								
Procs.	Blocos			*Round-Robin*			*Grouping-Sink*	
	Variáveis	*Indexicals*	Restrições	Variáveis	*Indexicals*	Restrições	*Indexicals*	Restrições
0	426.617	10.028	9.977	397.267	344.128	53.184	400.367	80.266
1	477.756	9.973	9.977	443.802	615.330	59.301	524.968	9.919
2	475.592	9.973	9.976	451.691	378.462	57.979	481.456	61.495
3	469.361	9.973	9.976	473.849	638.088	60.601	502.899	52.746
4	467.543	9.973	9.976	431.354	374.431	61.474	416.773	132.613
5	406.031	9.972	9.976	441.789	627.280	58.996	431.913	19.931
6	457.247	9.972	9.976	448.788	369.392	60.629	502.215	38.575
7	403.189	9.972	9.976	485.406	656.172	59.435	449.395	9.834
8	334.680	151.015	151.677	464.732	369.527	73.107	438.740	48.211
9	392.275	9.972	9.976	445.113	641.471	79.751	465.461	115.421
10	461.954	9.972	9.976	493.593	360.429	73.091	431.180	112.658
11	398.539	225.676	226.716	489.747	614.158	70.548	377.527	119.659
12	406.638	242.056	240.428	414.100	360.424	67.225	446.491	10.087
13	394.151	125.177	125.484	395.394	642.852	68.169	392.331	86.230
Diferença percentual	29,95	95,88	95,85	**19,89**	47,56	33,31	28,09	92,52

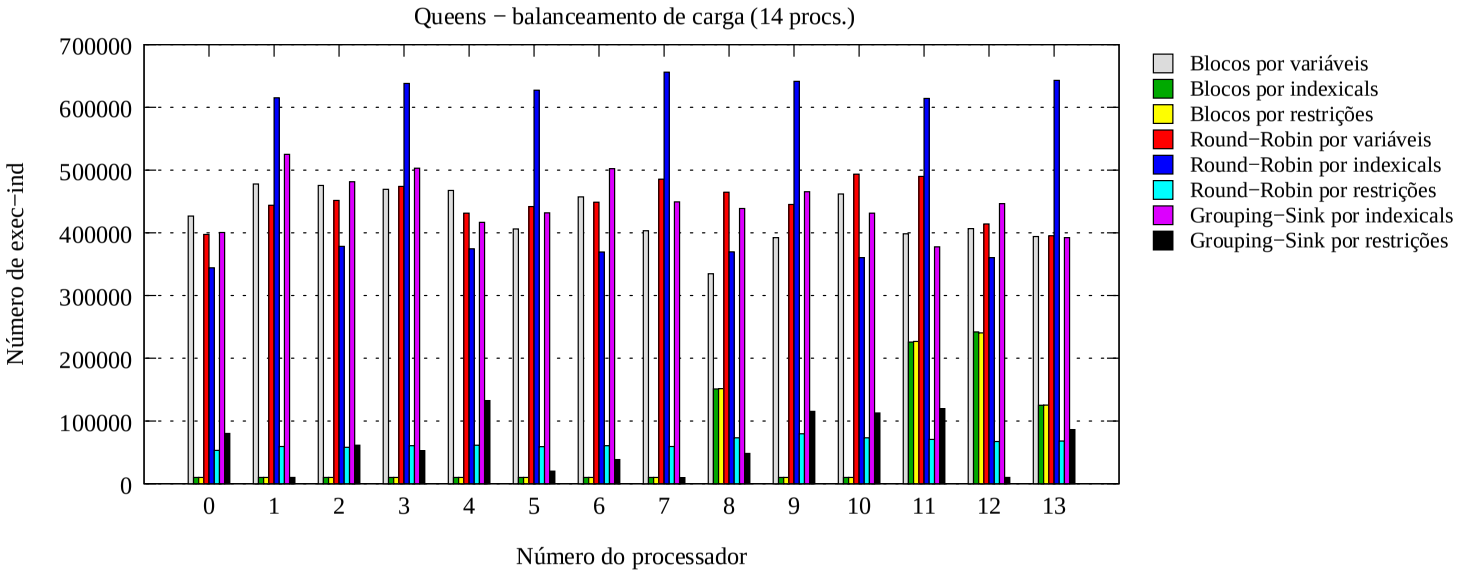


Figura 6.15: *Queens* - balanceamento de carga para 14 processadores

Queens - número de falhas para 14 processadores								
Procs.	Blocos			*Round-Robin*			*Grouping-Sink*	
	Variáveis	*Indexicals*	Restrições	Variáveis	*Indexicals*	Restrições	*Indexicals*	Restrições
0	0	65	62	879	1.177	1.313	409	791
1	0	69	67	752	699	1173	0	503
2	0	65	62	570	666	927	339	581
3	0	54	50	264	621	802	0	397
4	0	32	26	562	475	562	964	724
5	139	7	4	531	365	526	355	271
6	0	2	1	413	511	286	0	268
7	218	0	0	0	206	259	77	2
8	911	225	226	78	241	233	563	388
9	529	0	0	313	158	362	126	606
10	0	0	0	0	124	287	302	512
11	1.327	765	776	0	317	80	835	601
12	1.278	1.143	1.133	172	320	253	363	0
13	112	2.285	2.286	68	259	192	260	205
Diferença entre < e >	1.327	2.285	2.286	879	1.177	1.313	964	791
Total	**4.514**	4.712	4.693	4.602	6.139	7.255	4.593	5.849

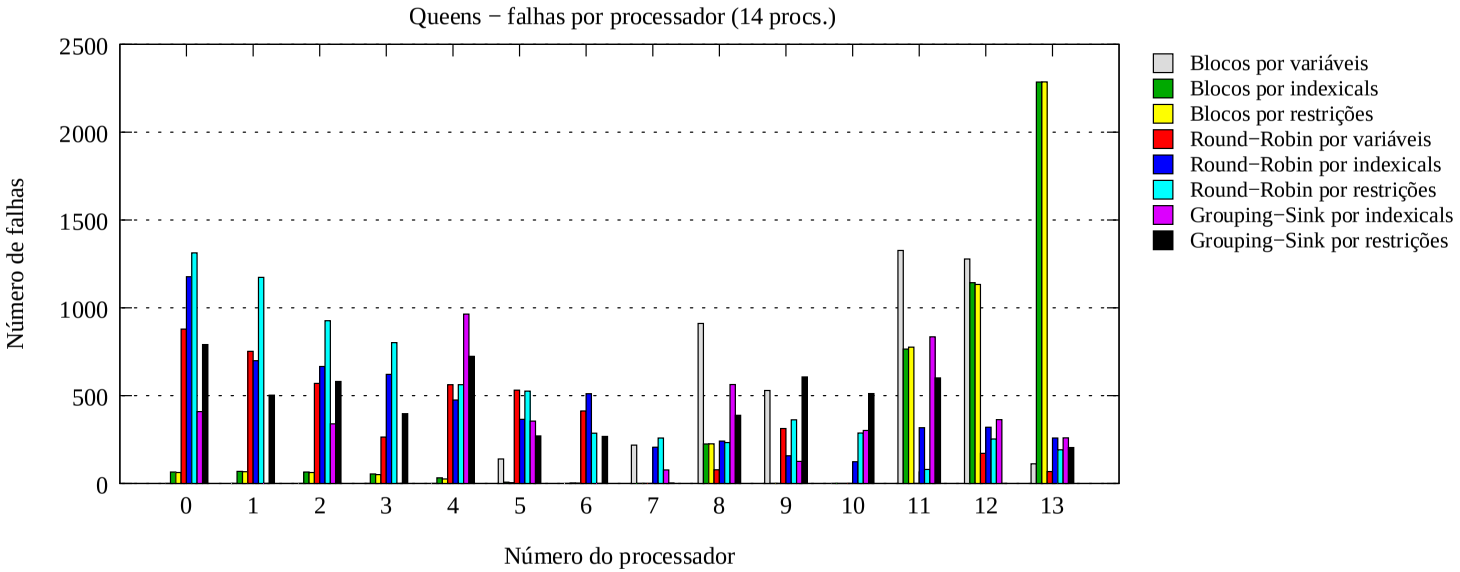


Figura 6.16: *Queens* - total de falhas para 14 processadores

Símbolo	Tempo de execução	Total de trabalho	Total de mensagens	Balanceamento de carga (%)	Número de falhas
+ + + +	5,18 a 5,52	756.210 a 1.508.955	762.785 a 822.635	18,27 a 28,05	4.482 a 4.724
+ + +	5,53 a 5,86	1.508.956 a 2.261.700	822.636 a 882.486	28,06 a 37,83	4.725 a 4.966
+ +	5,87 a 6,20	2.261.701 a 3.014.445	882.487 a 942.336	37,84 a 47,61	4.967 a 5.208
+	6,21 a 6,54	3.014.446 a 3.767.190	942.337 a 1.002.187	47,62 a 57,39	5.209 a 5.450
-	6,55 a 6,87	3.767.191 a 4.519.935	1.002.188 a 1.062.038	57,40 a 67,18	5.451 a 5.692
- -	6,88 a 7,21	4.519.936 a 5.272.680	1.062.039 a 1.121.888	67,19 a 76,96	5.693 a 5.934
- - -	7,22 a 7,55	5.272.681 a 6.025.425	1.121.889 a 1.181.739	76,97 a 86,74	5.935 a 6.176
- - - -	7,56 a 7,89	6.025.426 a 6.778.171	1.181.740 a 1.241.590	86,75 a 96,52	6.177 a 6.419

Particionamento	Tempo de execução	Total de trabalho	Total de mensagens	Balanceamento de carga	Número de falhas	Total geral
Blocos por variáveis	+ + + +	- - -	+ + +	+ + +	+ + + +	**+11**
Blocos por *indexicals*	-	+ + + +	+ + + +	- - - -	+ + + +	+7
Blocos por restrições	-	+ + + +	+ + + +	- - - -	+ + + +	+7
Round-Robin por variáveis	+ +	- - -	+	+ + + +	+ + + +	+8
Round-Robin por *indexicals*	- - - -	- - - -	- - - -	+ +	+ +	-8
Round-Robin por restrições	+ + +	+ + + +	- - - -	+ + + +	- - - -	+3
Grouping-Sink por *indexicals*	+ + +	- - -	+ +	+ + + +	+ + + +	**+10**
Grouping-Sink por restrições	- -	+ + + +	+	- -	+ + +	+4

Figura 6.17: *Queens* - resumo para 8 processadores

Símbolo	Tempo de execução	Total de trabalho	Total de mensagens	Balanceamento de carga (%)	Número de falhas
+ + + +	6,36 a 6,70	843.704 a 1.612.259	2.160.905 a 2.270.707	19,89 a 29,39	4.514 a 4.856
+ + +	6,71 a 7,04	1.612.260 a 2.380.814	2.270.708 a 2.380.509	29,40 a 38,89	4.857 a 5.199
+ +	7,05 a 7,38	2.380.815 a 3.149.369	2.380.510 a 2.490.311	38,90 a 48,39	5.200 a 5.541
+	7,39 a 7,72	3.149.370 a 3.917.924	2.490.312 a 2.600.113	48,40 a 57,89	5.542 a 5.884
-	7,73 a 8,06	3.917.925 a 4.686.479	2.600.114 a 2.709.915	57,90 a 67,38	5.885 a 6.227
- -	8,07 a 8,40	4.686.480 a 5.455.034	2.709.916 a 2.819.717	67,39 a 76,88	6.228 a 6.569
- - -	8,41 a 8,74	5.455.035 a 6.223.589	2.819.718 a 2.929.519	76,89 a 86,38	6.570 a 6.912
- - - -	8,75 a 9,08	6.223.590 a 6.992.144	2.929.520 a 3.039.322	86,39 a 95,88	6.913 a 7.255

Particionamento	Tempo de execução	Total de trabalho	Total de mensagens	Balanceamento de carga	Número de falhas	Total geral
Blocos por variáveis	+ + + +	- - -	+ + +	+ + +	+ + + +	**+11**
Blocos por *indexicals*	- - - -	+ + + +	+ + + +	- - - -	+ + + +	+4
Blocos por restrições	- - - -	+ + + +	+ + + +	- - - -	+ + + +	+4
Round-Robin por variáveis	+ + +	- - - -	+	+ + + +	+ + + +	+8
Round-Robin por *indexicals*	- - -	- - - -	- - -	+ +	-	-9
Round-Robin por restrições	+ + +	+ + + +	- - - -	+ + +	- - - -	+2
Grouping-Sink por *indexicals*	+ + + +	- - - -	+ +	+ + + +	+ + + +	+10
Grouping-Sink por restrições	- - -	+ + + +	+	- - - -	+ + + +	+2

Figura 6.18: *Queens* - resumo para 14 processadores

6.4 PBCSP (50 vars. - 0,65)

A Figura 6.19 apresenta os tempos de execução para todos os particionamentos. Para 1 processador, a diferença de tempo entre os particionamentos é de no máximo 0,6%.

O menor tempo de execução para 1, 2, 4, 12 e 14 processadores é de Blocos por variáveis. *Round-Robin* por variáveis tem o menor tempo para 8 processadores. Para todos os particionamentos o tempo de execução diminuiu com o aumento do número de processadores.

Blocos por variáveis apresenta a menor quantidade de trabalho (Figura 6.20) e o total de mensagens (Figura 6.21). Mas o melhor balanceamento de carga para 8 e 14 processadores é de *Round-Robin* por *indexicals* (Figuras 6.22 e 6.24). O total de falhas de Blocos por variáveis para 8 e 14 processadores é menor do que para *Round-Robin* por *indexicals* (Figuras 6.23 e 6.25).

Grouping-Sink por *indexicals* apresenta uma diferença de 9,4% para 8 processadores em tempo de execução em relação a Blocos por variáveis e 2,2% para 14 processadores.

Em total de mensagens, *Grouping-Sink* por *indexicals* tem 3,4% maior que Blocos por variáveis, de mensagens, para 8 processadores e 5,4% para 14 processadores.

As Figuras 6.26 e 6.27 apresentam um resumo para PBCSP (50 vars. - 0,65) com tempo de execução, total de trabalho, total de mensagem, balanceamento de carga e total de falhas.

Para 8 processadores, os dois melhores particionamentos são *Round-Robin* por variáveis e *Grouping-Sink* por *indexicals*.

Para 14 processadores, os dois melhores particionamentos são Blocos por variáveis e *Round-Robin* por variáveis. Mas *Grouping-Sink* por *indexicals* so é inferior no balanceamento de carga (20,8% em relação a *Round-Robin* por variáveis e 15,5% em relação a Blocos por variáveis). Blocos por variáveis é melhor que *Grouping-Sink* por *indexicals* em tempo de execução em 2,2% e em número de mensagens em 5,4%. E *Grouping-Sink* por *indexicals* é melhor, em número de mensagens, que *Round-Robin* por variáveis em 0,7%.

Estas diferenças no tempo de execução e no número de mensagens não são significativas, portanto, *Grouping-Sink* por *indexicals* pode ser uma boa alternativa.

PBCSP (50 vars. - 0,65) - tempo de execução total																
Procs.	Blocos						*Round-Robin*						*Grouping-Sink*			
	Variáveis		*Indexicals*		Restrições		Variáveis		*Indexicals*		Restrições		*Indexicals*		Restrições	
1	**604,01**	(2,91)	604,55	(3,31)	605,58	(4,94)	605,22	(3,45)	604,59	(3,49)	606,47	(4,27)	607,83	(1,31)	604,27	(2,35)
2	**479,33**	(2,67)	536,36	(1,57)	539,86	(1,33)	510,67	(2,86)	732,06	(2,47)	654,36	(1,59)	486,99	(0,97)	506,87	(1,78)
4	**239,81**	(0,89)	318,23	(1,05)	322,33	(0,99)	246,17	(0,86)	382,97	(1,22)	335,99	(1,31)	240,51	(0,69)	287,51	(1,94)
8	126,85	(0,52)	179,43	(0,54)	181,96	(0,28)	**113,33**	(0,20)	194,07	(0,43)	164,71	(0,58)	114,89	(0,30)	157,54	(0,51)
12	**74,61**	(0,36)	129,07	(0,71)	131,82	(0,52)	80,20	(0,33)	129,33	(0,61)	110,23	(0,48)	80,66	(0,15)	111,35	(0,31)
14	**67,61**	(0,47)	117,75	(0,50)	119,02	(0,32)	68,17	(0,31)	114,14	(0,36)	97,18	(0,54)	69,13	(0,21)	104,29	(0,59)

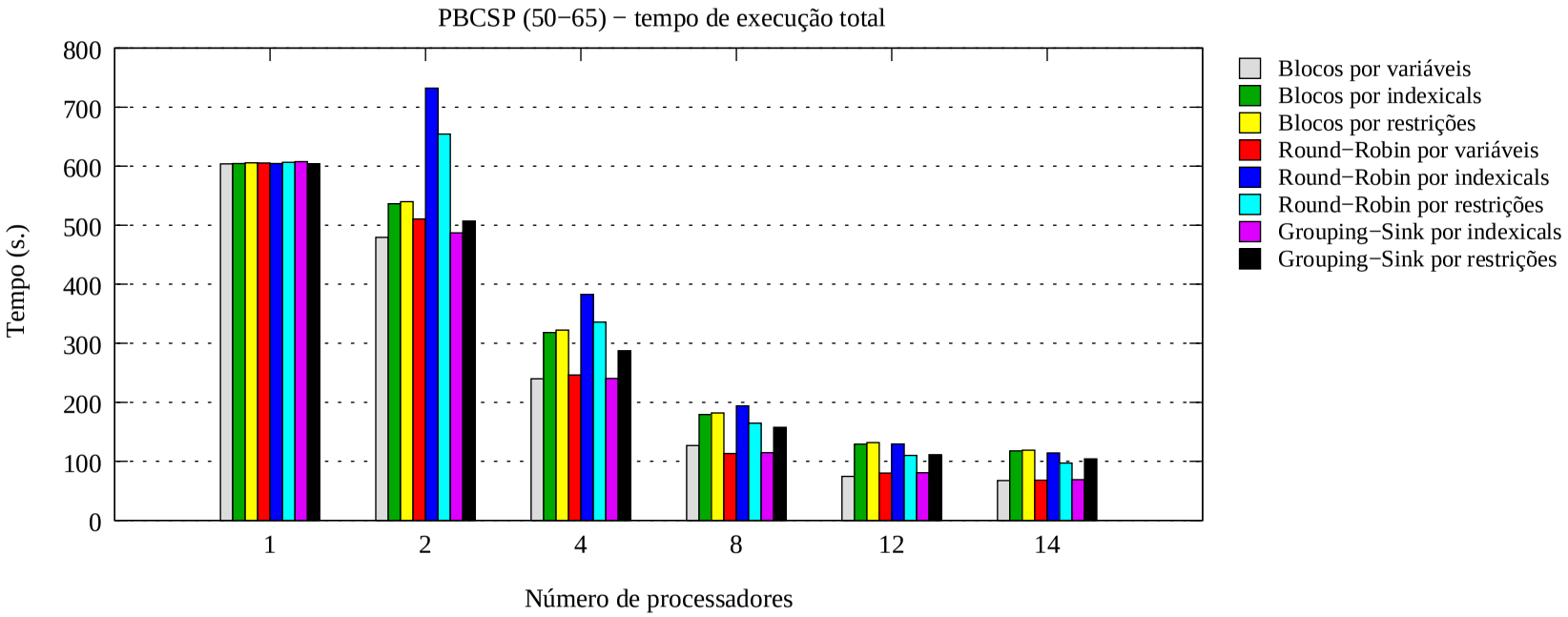


Figura 6.19: PBCSP (50 vars. - 0,65) - tempo de execução

PBCSP (50 vars. - 0,65) - total de trabalho								
Procs.	Blocos			*Round-Robin*			*Grouping-Sink*	
	Variáveis	*Indexicals*	Restrições	Variáveis	*Indexicals*	Restrições	*Indexicals*	Restrições
1	11.989.918	11.989.918	11.989.918	11.989.918	11.989.918	11.989.918	11.989.918	11.989.918
2	**12.320.672**	14.695.864	14.701.015	12.660.399	18.815.499	16.221.290	12.375.187	12.988.614
4	**14.110.967**	17.286.286	17.295.128	14.753.104	23.275.247	20.587.789	14.471.230	14.630.249
8	**14.691.694**	19.147.864	19.132.527	14.912.027	26.425.985	22.400.960	15.060.571	15.984.653
12	**14.422.089**	19.867.072	19.805.185	14.930.079	26.815.241	22.515.753	14.910.912	15.922.032
14	**14.186.891**	20.099.092	20.068.576	14.785.527	26.665.450	22.429.209	14.932.960	15.858.491

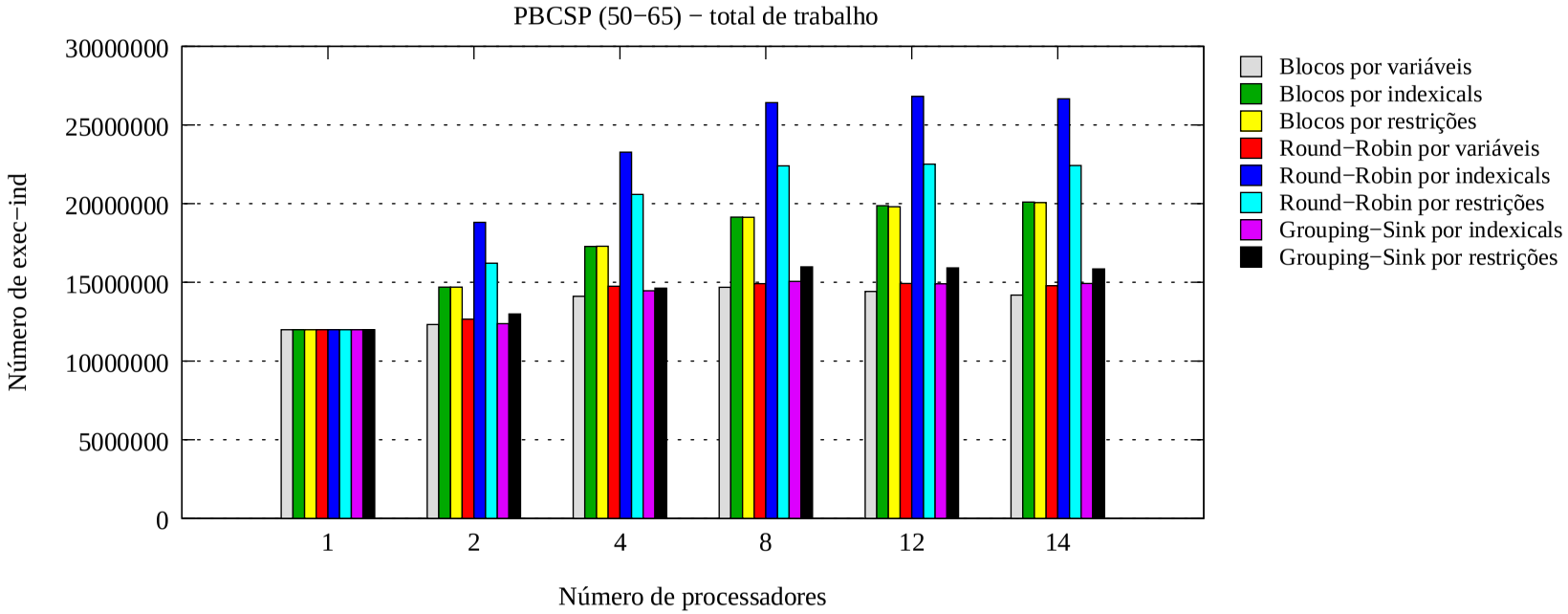


Figura 6.20: PBCSP (50 vars. - 0,65) - total de trabalho

PBCSP (50 vars. - 0,65) - total de mensagens								
Procs.	Blocos			*Round-Robin*			*Grouping-Sink*	
	Variáveis	*Indexicals*	Restrições	Variáveis	*Indexicals*	Restrições	*Indexicals*	Restrições
1	0	0	0	0	0	0	0	0
2	**167.695**	208.972	209.237	176.212	246.439	259.214	169.090	177.674
4	**720.072**	979.869	981.051	764.997	1.217.064	1.335.342	746.205	788.590
8	**1.996.546**	3.045.047	3.035.416	2.036.440	3.598.280	3.729.936	2.066.757	2.298.888
12	**3.397.452**	5.296.816	5.242.148	3.560.070	6.157.459	6.263.026	3.579.757	3.757.911
14	**4.103.691**	6.334.812	6.358.904	4.368.048	7.449.666	7.586.306	4.339.150	4.360.857

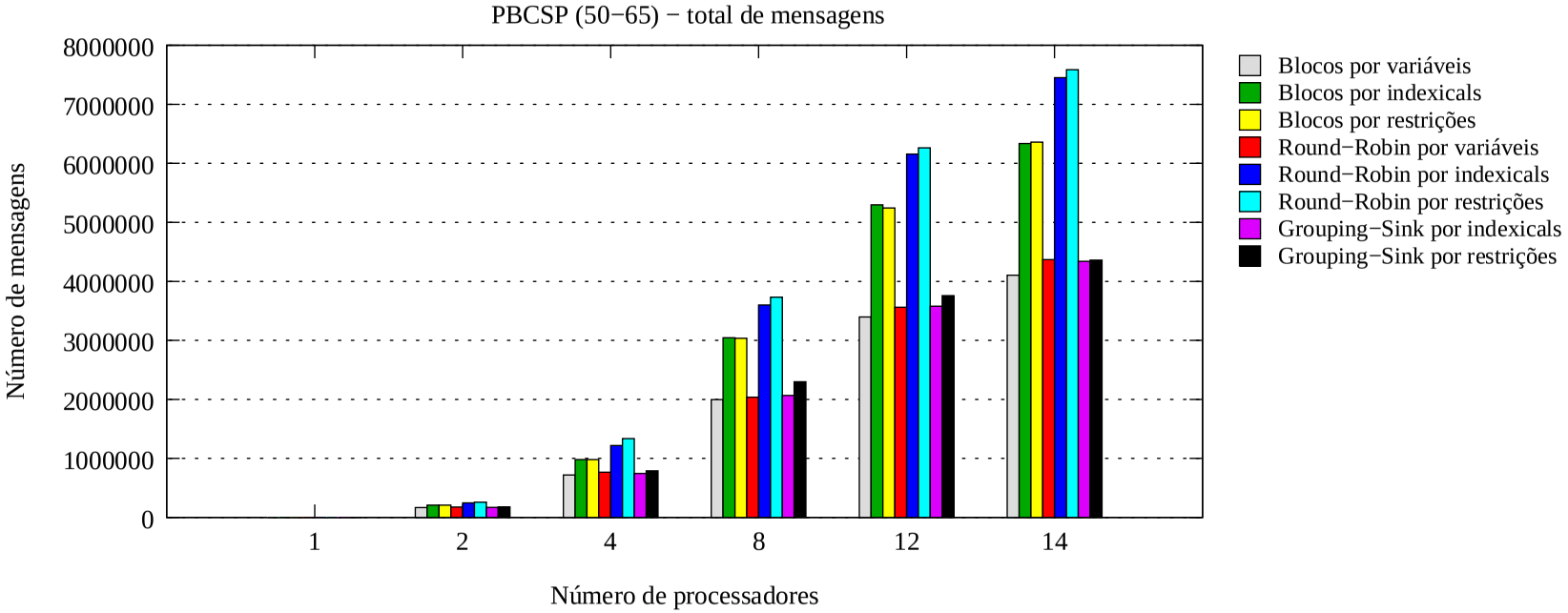


Figura 6.21: PBCSP (50 vars. - 0,65) - total de mensagens

PBCSP (50 vars. - 0,65) - balanceamento de carga para 8 processadores								
Procs.	Blocos			*Round-Robin*			*Grouping-Sink*	
	Variáveis	*Indexicals*	Restrições	Variáveis	*Indexicals*	Restrições	*Indexicals*	Restrições
0	1.948.174	1.696.837	1.452.131	2.014.218	3.318.387	2.821.845	2.118.302	2.192.508
1	1.982.661	1.996.229	2.003.547	2.029.743	3.275.706	2.719.591	2.059.040	2.128.238
2	1.673.318	2.184.818	2.273.459	1.770.232	3.369.371	2.856.987	1.842.810	2.080.641
3	1.882.711	2.797.086	2.804.633	1.787.958	3.281.461	2.846.009	1.795.762	1.903.877
4	1.888.550	2.726.289	2.758.559	1.829.742	3.314.045	2.750.405	1.801.275	1.931.129
5	1.775.463	2.705.435	2.737.593	1.891.364	3.273.837	2.862.511	1.833.625	2.106.620
6	1.833.434	2.644.832	2.669.566	1.736.498	3.325.483	2.783.670	1.824.286	1.565.927
7	1.707.383	2.396.338	2.433.039	1.852.272	3.267.695	2.759.942	1.785.471	2.075.713
Diferença percentual	15,60	39,34	48,22	14,45	**3,02**	4,99	15,71	28,58

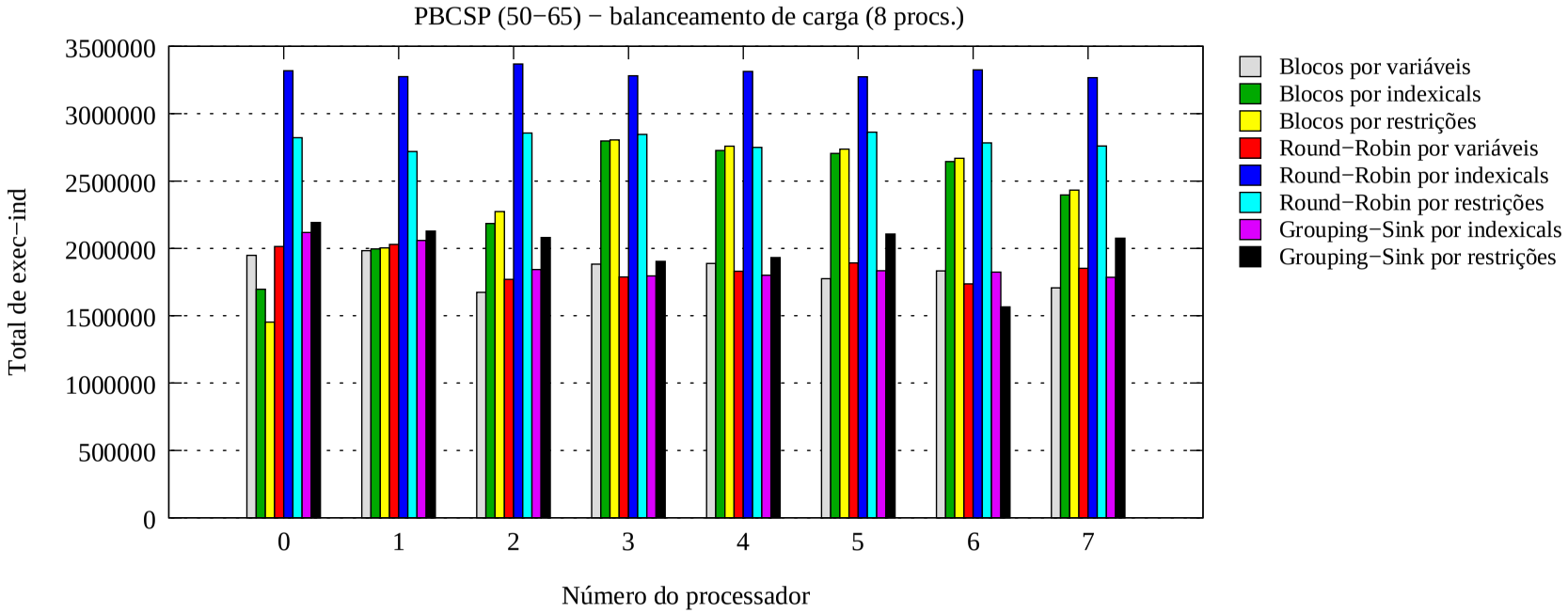


Figura 6.22: PBCSP (50 vars. - 0,65) - balanceamento de carga para 8 processadores

PBCSP (50 vars. - 0,65) - número de falhas para 8 processadores								
Procs.	Blocos			*Round-Robin*			*Grouping-Sink*	
	Variáveis	*Indexicals*	Restrições	Variáveis	*Indexicals*	Restrições	*Indexicals*	Restrições
0	445	236	191	439	846	820	787	734
1	502	502	481	570	438	550	524	611
2	345	403	462	518	758	687	574	654
3	656	670	649	498	436	620	461	533
4	739	725	716	713	685	526	581	723
5	544	647	654	711	509	692	649	547
6	930	919	953	632	763	563	700	411
7	645	701	707	717	472	535	524	587
Diferença entre < e >	585	683	762	278	410	294	326	323
Total	4.806	4.803	4.813	**4.798**	4.907	4.993	4.800	4.800

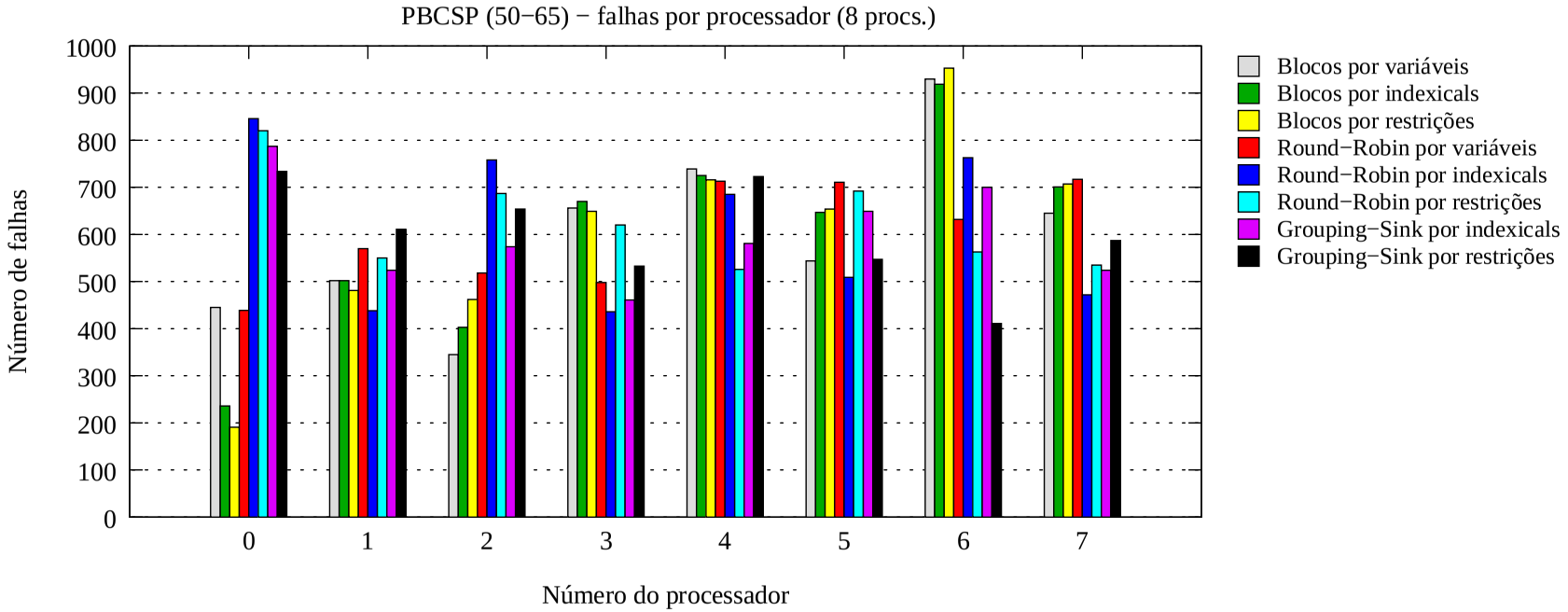


Figura 6.23: PBCSP (50 vars. - 0,65) - total de falhas para 8 processadores

PBCSP (50 vars. - 0,65) - balanceamento de carga para 14 processadores								
Procs.	Blocos			*Round-Robin*			*Grouping-Sink*	
	Variáveis	*Indexicals*	Restrições	Variáveis	*Indexicals*	Restrições	*Indexicals*	Restrições
0	1.035.932	645.483	388.863	1.087.104	1.842.385	1.549.076	1.170.317	1.249.431
1	1.137.344	1.475.259	1.485.640	1.203.504	1.918.720	1.598.739	1.235.563	1.234.670
2	1.086.394	1.086.812	1.248.033	1.154.651	1.906.193	1.663.331	1.217.023	1.280.586
3	1.143.286	1.560.116	1.411.312	1.159.734	1.928.471	1.613.037	1.156.347	1.205.685
4	1.100.299	1.235.591	1.441.100	1.164.159	1.870.525	1.615.782	1.164.450	845.605
5	1.174.293	1.439.216	1.281.751	1.167.012	1.925.574	1.628.815	1.172.173	1.311.654
6	1.132.062	1.687.675	1.730.818	1.183.787	1.872.218	1.605.773	1.206.162	1.448.098
7	1.169.016	1.662.714	1.680.849	1.181.333	1.914.945	1.543.590	1.207.464	1.370.461
8	869.846	1.601.885	1.595.208	914.014	1.900.234	1.603.575	902.449	1.287.748
9	886.459	1.600.916	1.630.825	902.739	1.914.168	1.653.128	827.045	1.041.907
10	865.972	1.603.479	1.631.986	892.181	1.971.960	1.579.454	954162	752.788
11	892.112	1.599.316	1.608.037	888.360	1.914.347	1.594.312	893.424	950116
12	847.551	1.573.414	1.595.477	932.262	1.888.809	1.588.052	914.358	792.985
13	846.325	1.327.216	1.338.677	954.687	1.896.901	1.592.545	912.023	1.086.757
Diferença percentual	27,93	61,75	76,87	26,19	**6,57**	7,20	33,06	48,02

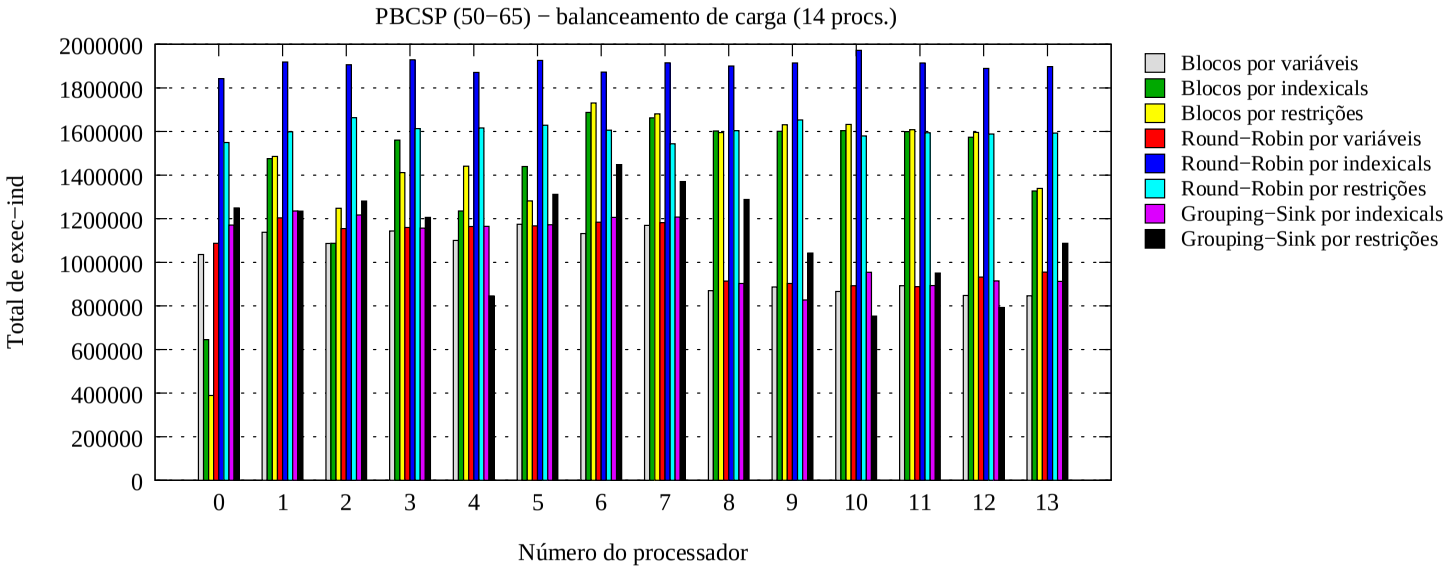


Figura 6.24: PBCSP (50 vars. - 0,65) - balanceamento de carga para 14 processadores

PBCSP (50 vars. - 0,65) - número de falhas para 14 processadores								
Procs.	Blocos			*Round-Robin*			*Grouping-Sink*	
	Variáveis	*Indexicals*	Restrições	Variáveis	*Indexicals*	Restrições	*Indexicals*	Restrições
0	169	104	58	245	596	631	388	317
1	349	361	300	426	327	411	501	354
2	280	274	365	462	486	481	538	591
3	388	422	340	338	324	449	378	249
4	200	282	334	350	380	380	563	260
5	405	307	205	376	327	457	324	445
6	367	382	427	414	383	392	418	457
7	442	405	442	406	241	255	353	456
8	263	495	430	303	508	418	295	418
9	394	326	341	214	323	416	157	307
10	536	582	602	286	464	295	315	266
11	380	439	442	338	310	343	282	327
12	411	406	431	371	387	306	234	306
13	292	328	313	366	277	330	122	278
Diferença entre < e >	367	478	544	248	355	336	441	342
Total	4.876	5.113	5.030	4.895	5.333	5.564	**4.868**	5.031

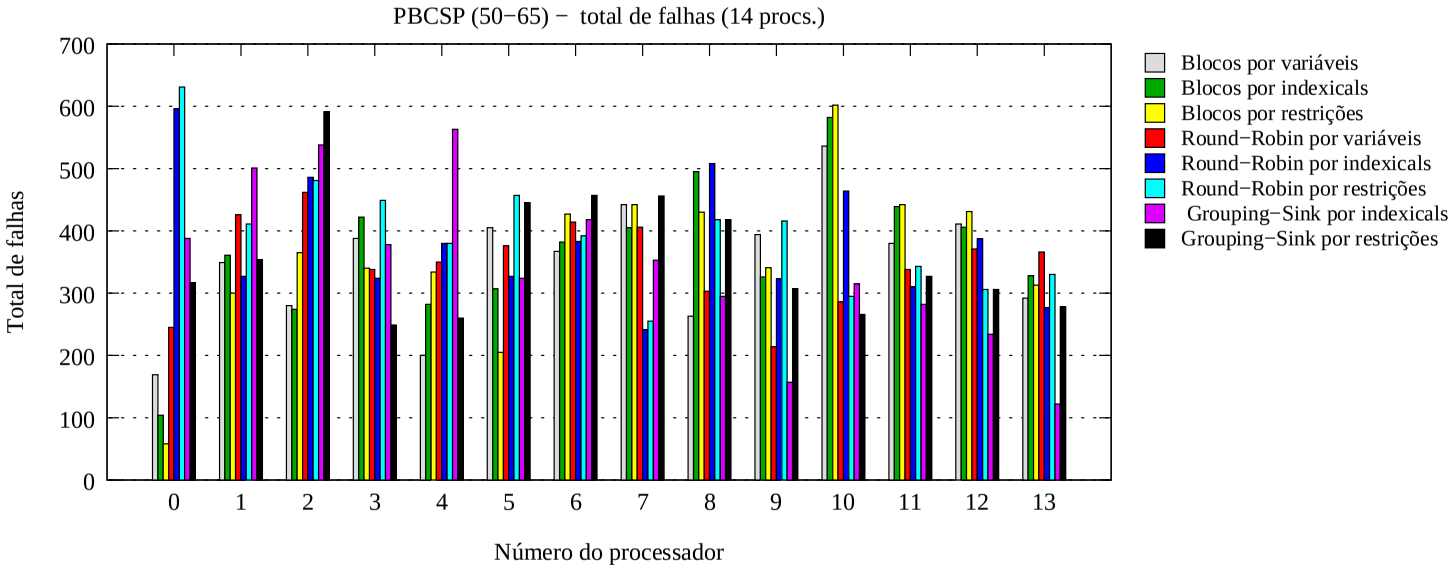


Figura 6.25: PBCSP (50 vars. - 0,65) - total de falhas para 14 processadores

Símbolo	Tempo de execução	Total de trabalho	Total de mensagens	Balanceamento de carga (%)	Número de falhas
+ + + +	113,33 a 123,42	14.691.694 a 16.158.480	1.996.546 a 2.213.219	3,02 a 8,67	4.798 a 4.822
+ + +	123,52 a 133,51	16.158.481 a 17.625.266	2.213.220 a 2.429.893	8,68 a 14,32	4.823 a 4.846
+ +	133,61 a 143,61	17.625.267 a 19.092.053	2.429.894 a 2.646.567	14,33 a 19,97	4.847 a 4.871
+	143,71 a 153,70	19.092.054 a 20.558.839	2.646.568 a 2.863.241	19,98 a 25,62	4.872 a 4.895
-	153,80 a 163,79	20.558.840 a 22.025.625	2.863.242 a 3.079.914	25,63 a 31,27	4.896 a 4.919
- -	163,89 a 173,88	22.025.626 a 23.492.412	3.079.915 a 3.296.588	31,28 a 36,92	4.920 a 4.944
- - -	173,98 a 183,98	23.492.413 a 24.959.198	3.296.589 a 3.513.262	36,93 a 42,57	4.945 a 4.968
- - - -	184,08 a 194,07	24.959.199 a 26.425.985	3.513.263 a 3.729.936	42,58 a 48,22	4.969 a 4.993

Particionamento	Tempo de execução	Total de trabalho	Total de mensagens	Balanceamento de carga	Número de falhas	Total geral
Blocos por variáveis	+ + +	+ + + +	+ + + +	+ +	+ + + +	+17
Blocos por *indexicals*	- - -	+	-	- - -	+ + + +	-1
Blocos por restrições	- - -	+	-	- - - -	+ + + +	-3
Round-Robin por variáveis	+ + + +	+ + + +	+ + + +	+ +	+ + + +	**+18**
Round-Robin por *indexicals*	- - - -	- - - -	- - - -	+ + + +	-	-9
Round-Robin por restrições	- -	- -	- - - -	+ + + +	- - - -	-8
Grouping-Sink por *indexicals*	+ + + +	+ + + +	+ + + +	+ +	+ + + +	**+18**
Grouping-Sink por restrições	-	+ + + +	+ + +	-	+ + + +	+9

Figura 6.26: PBCSP (50 vars. - 0,65) - resumo para 8 processadores

Símbolo	Tempo de execução	Total de trabalho	Total de mensagens	Balanceamento de carga (%)	Número de falhas
+ + + +	67,61 a 74,04	14.186.891 a 15.746.710	4.103.691 a 4.539.017	6,57 a 15,36	4.868 a 4.955
+ + +	74,14 a 80,46	15.746.711 a 17.306.530	4.539.018 a 4.974.344	15,37 a 24,15	4.956 a 5.042
+ +	80,56 a 86,89	17.306.531 a 18.866.350	4.974.345 a 5.409.671	24,16 a 32,93	5.043 a 5.129
+	86,99 a 93,31	18.866.351 a 20.426.170	5.409.672 a 5.844.998	32,94 a 41,72	5.130 a 5.216
-	93,41 a 99,74	20.426.171 a 21.985.990	5.844.999 a 6.280.325	41,73 a 50,51	5.217 a 5.303
- -	99,84 a 106,17	21.985.991 a 23.545.810	6.280.326 a 6.715.652	50,52 a 59,30	5.304 a 5.390
- - -	106,27 a 112,59	23.545.811 a 25.105.630	6.715.653 a 7.150.979	59,31 a 68,08	5.391 a 5.477
- - - -	112,69 a 119,02	25.105.631 a 26.665.450	7.150.980 a 7.586.306	68,09 a 76,87	5.478 a 5.564

Particionamento	Tempo de execução	Total de trabalho	Total de mensagens	Balanceamento de carga	Número de falhas	Total geral
Blocos por variáveis	+ + + +	+ + + +	+ + + +	+ +	+ + + +	**+18**
Blocos por *indexicals*	- - - -	+	- -	- - -	+ +	-6
Blocos por restrições	- - - -	-	- -	- - - -	+ + +	-8
Round-Robin por variáveis	+ + + +	+ + + +	+ + + +	+ +	+ + + +	**+18**
Round-Robin por *indexicals*	- - - -	- - - -	+ + + +	+ + + +	- -	-2
Round-Robin por restrições	-	- -	- - - -	+ + + +	- - - -	-7
Grouping-Sink por *indexicals*	+ + + +	+ + + +	+ + + +	-	+ + + +	+15
Grouping-Sink por restrições	- -	+ + + +	+ + + +	+	+ + +	+12

Figura 6.27: PBCSP (50 vars. - 0,65) - resumo para 14 processadores

6.5 PBCSP (100 vars. - 0,75)

A Figura 6.28 apresenta os tempos de execução para todos os particionamentos. Para 1 processador a diferença entre os particionamentos em tempo de execução é de no máximo 0,6%. Para 2, 8, 12 e 14 processadores, Blocos por variáveis tem o menor tempo de execução. *Grouping-Sink* por *indexicals* tem o menor tempo de execução para 1 e 4 processadores.

Blocos por variáveis apresentou a menor quantidade de trabalho (Figura 6.29) para todas as quantidades de processadores.

Para PBCSP com 100 variáveis, a quantidade de trabalho realizado gera um impacto no tempo de execução total.

Em termos de total de mensagens (Figura 6.30), Blocos por variáveis tem a menor quantidade. Para 4, 8 e 14 processadores *Round-Robin* por variáveis tem a menor quantidade.

O melhor balanceamento de carga para 8 e 14 processadores (Figuras 6.31 e 6.33) é de *Round-Robin* por restrições.

O total de falhas para 8 processadores (Figura 6.32) é o mesmo para os particionamentos com melhor tempo de execução. E para 14 processadores (Figura 6.34), a diferença é pequena (1 ou 2 falhas).

As Figuras 6.35 e 6.36 apresentam um resumo dos resultados de tempo de execução, total de trabalho, total de mensagens, balanceamento de carga e número de falhas para PBCSP (100 vars. - 0,75) com todos os particionamentos.

Para 8 processadores (Figura 6.35), *Grouping-Sink* por *indexicals* foi o melhor particionamento.

Para 14 processadores (Figura 6.36), os dois melhores particionamentos foram *Round-Robin* por variáveis e *Grouping-Sink* por *indexicals*. Em tempo de execução *Grouping-Sink* por *indexicals* é 0,8% melhor que *Round-Robin* por variáveis. E em total de mensagens *Round-Robin* por variáveis é melhor que *Grouping-Sink* por *indexicals* em 5,7%.

PBCSP (100 vars. - 0,75) - tempo de execução total																
Procs.	Blocos						*Round-Robin*						*Grouping-Sink*			
	Variáveis		*Indexicals*		Restrições		Variáveis		*Indexicals*		Restrições		*Indexicals*		Restrições	
1	37,52	(0,17)	37,46	(0,10)	37,41	(0,12)	37,46	(0,10)	37,49	(0,16)	37,38	(0,07)	**37,31**	(0,03)	37,53	(0,21)
2	**27,72**	(0,08)	34,85	(0,11)	35,23	(0,15)	28,68	(0,11)	40,75	(0,18)	38,26	(0,14)	27,87	(0,09)	31,32	(0,34)
4	14,34	(0,13)	20,04	(0,18)	20,18	(0,12)	14,12	(0,15)	22,95	(0,14)	22,32	(0,19)	**14,22**	(0,06)	17,61	(0,10)
8	**6,67**	(0,08)	10,16	(0,07)	10,30	(0,05)	6,93	(0,09)	11,66	(0,15)	11,95	(0,16)	6,99	(0,07)	9,39	(0,06)
12	**4,03**	(0,07)	6,84	(0,04)	6,97	(0,05)	4,20	(0,07)	8,05	(0,06)	7,64	(0,11)	4,31	(0,04)	6,78	(0,05)
14	**3,47**	(0,16)	6,07	(0,25)	5,85	(0,04)	3,59	(0,18)	6,46	(0,07)	6,31	(0,13)	3,56	(0,05)	5,74	(0,06)

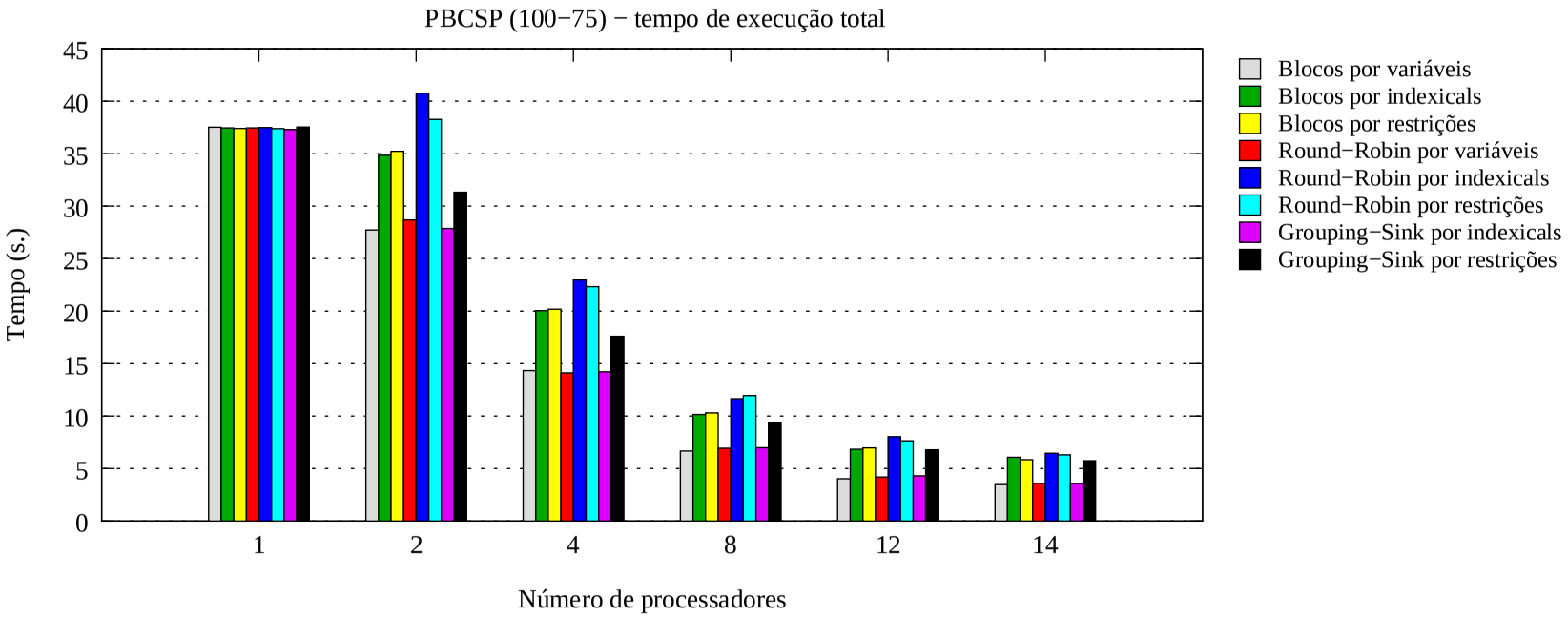


Figura 6.28: PBCSP (100 vars. - 0,75) - tempo de execução

PBCSP (100 vars. - 0,75) - total de trabalho								
Procs.	Blocos			*Round-Robin*			*Grouping-Sink*	
	Variáveis	*Indexicals*	Restrições	Variáveis	*Indexicals*	Restrições	*Indexicals*	Restrições
1	626.000	626.000	626.000	626.000	626.000	626.000	626.000	626.000
2	**606.781**	780.629	791.484	629.220	883.944	835.688	622.756	700.926
4	**709.829**	970.567	978.256	721.054	1.106.991	1.148.968	742.249	777.823
8	**824.055**	1.131.714	1.146.736	831.330	1.411.977	1.463.700	837.955	934.788
12	**748.524**	1.200.365	1.201.575	778.405	1.594.434	1.486.073	813.232	1.001.878
14	**741.156**	1.221.116	1.160.413	743.624	1.555.279	1458.966	779.274	1.008.371

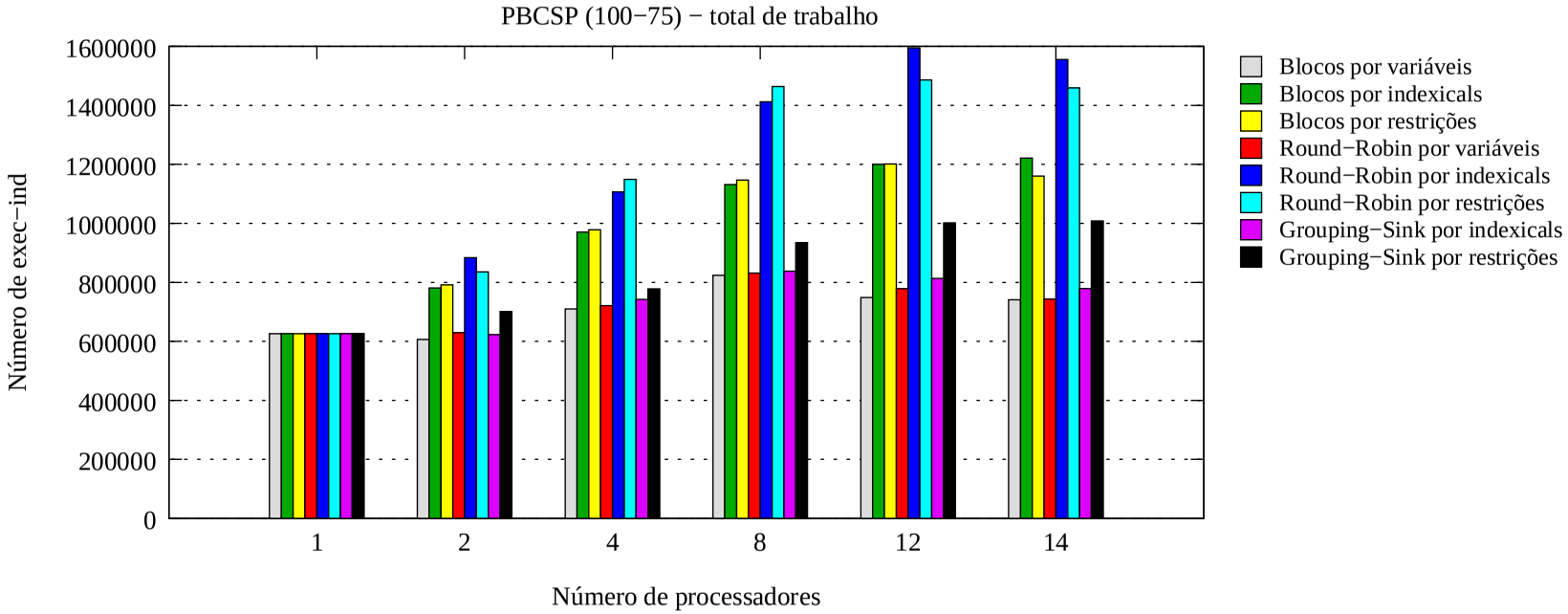


Figura 6.29: PBCSP (100 vars. - 0,75) - total de trabalho

PBCSP (100 vars. - 0,75) - total de mensagens								
Procs.	Blocos			*Round-Robin*			*Grouping-Sink*	
	Variáveis	*Indexicals*	Restrições	Variáveis	*Indexicals*	Restrições	*Indexicals*	Restrições
1	0	0	0	0	0	0	0	0
2	**2.878**	4.082	4.136	3.080	4.700	4.904	3.011	3.215
4	13.440	19.611	19.893	**13.344**	23.487	26.595	13.611	15.413
8	39.774	63.644	65.653	**39.557**	80.094	86.835	39.858	51.997
12	**59.400**	114.884	117.667	61.622	150.920	153.274	64.614	94.538
14	71.825	144.768	141.778	**71.448**	180.570	180.076	75.738	116.732

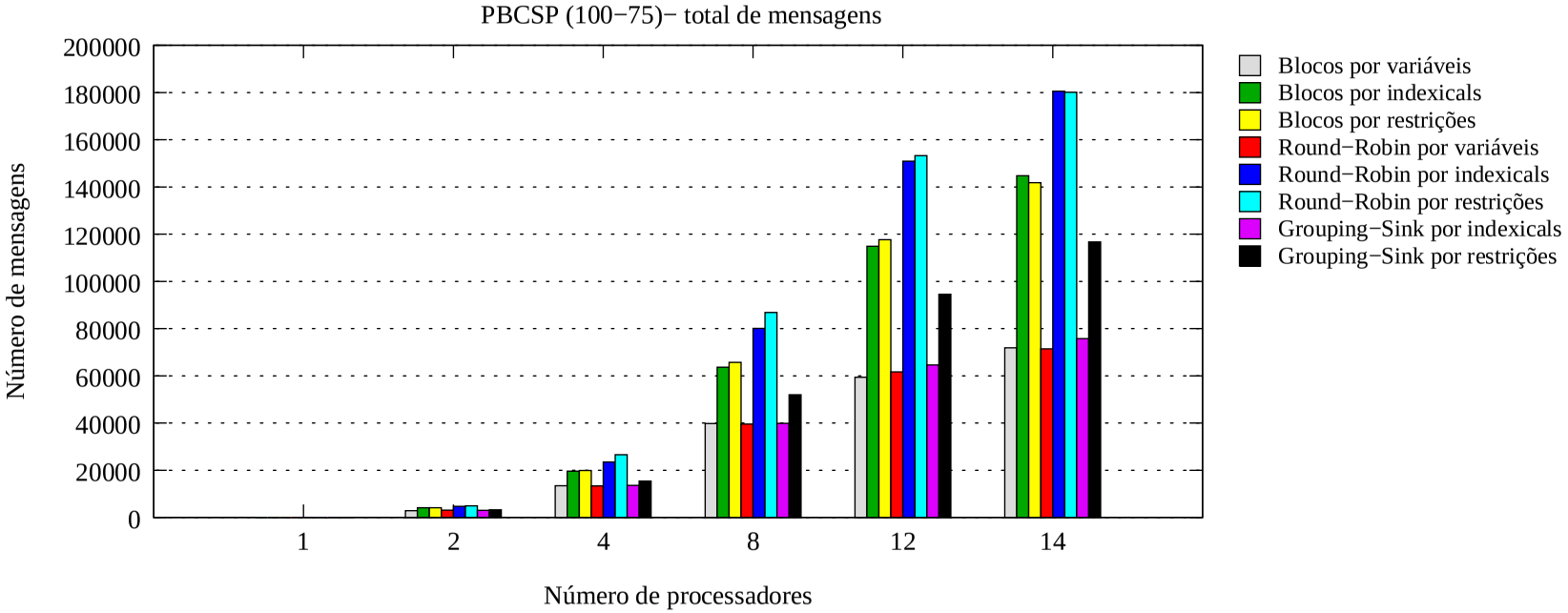


Figura 6.30: PBCSP (100 vars. - 0,75) - total de mensagens

PBCSP (100 vars. - 0,75) - balanceamento de carga para 8 processadores								
Procs.	Blocos			*Round-Robin*			*Grouping-Sink*	
	Variáveis	*Indexicals*	Restrições	Variáveis	*Indexicals*	Restrições	*Indexicals*	Restrições
0	104.126	132.317	126.179	106.065	178.835	181.667	110.220	117.362
1	108.498	154.528	154.649	110.031	170.767	184.061	110.241	120.871
2	105.381	128.659	136.146	109.336	185.567	182.240	107.806	127.989
3	107.281	149.246	151.683	106.279	171.585	181.857	106.326	116.351
4	108.439	132.571	136.296	100.022	181.308	183.719	103.416	118.107
5	108.436	140.430	142.654	99.299	171.447	186.111	99.878	112.921
6	110.265	146.303	147.668	100.078	182.244	182.868	100.880	115.589
7	71.629	147.660	151.461	100.220	170.224	181.177	99.188	105.598
Diferença percentual	35,04	16,74	18,41	9,75	8,27	**2,65**	10,03	17,49

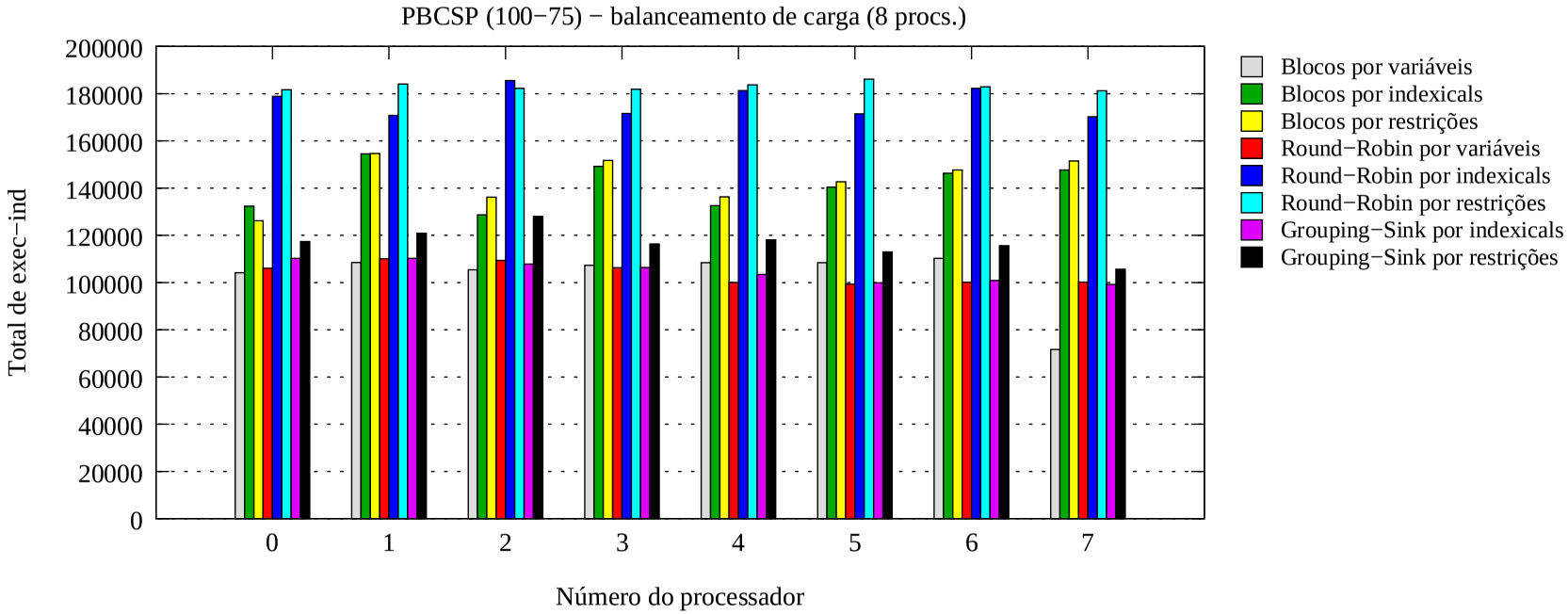


Figura 6.31: PBCSP (100 vars. - 0,75) - balanceamento de carga para 8 processadores

PBCSP (100 vars. - 0,75) - número de falhas para 8 processadores								
Procs.	Blocos			*Round-Robin*			*Grouping-Sink*	
	Variáveis	*Indexicals*	Restrições	Variáveis	*Indexicals*	Restrições	*Indexicals*	Restrições
0	1	5	7	6	5	0	6	3
1	2	2	0	6	1	7	1	2
2	7	5	3	5	11	3	3	6
3	6	7	11	5	2	7	3	4
4	9	4	4	5	7	2	4	8
5	6	4	5	4	0	6	5	3
6	2	6	3	2	4	3	4	3
7	2	2	2	4	5	7	9	6
Diferença entre < e >	8	5	11	4	11	7	8	6
Total	**35**	**35**	**35**	37	**35**	**35**	35	35

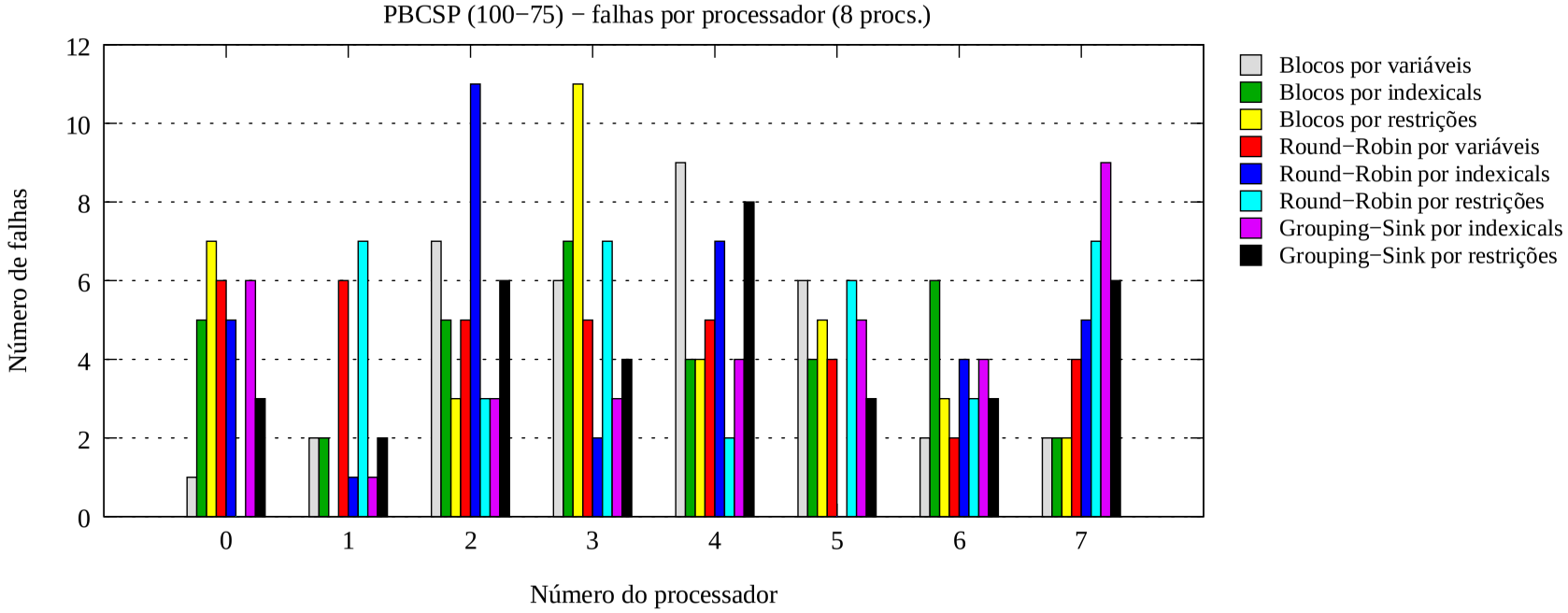


Figura 6.32: PBCSP (100 vars. - 0,75) - total de falhas para 8 processadores

PBCSP (100 vars. - 0,75) - balanceamento de carga para 14 processadores								
Procs.	Blocos			*Round-Robin*			*Grouping-Sink*	
	Variáveis	*Indexicals*	Restrições	Variáveis	*Indexicals*	Restrições	*Indexicals*	Restrições
0	58.623	74.938	65.118	57.685	111.297	105.356	60.820	79.167
1	58.875	91.831	86.477	59.510	109.930	105.034	62.182	76.992
2	53.612	95.168	89.548	52.301	113.068	103.167	54.981	74.401
3	50.328	80.662	79.842	52.496	110.017	105.866	54.201	53.977
4	51.325	88.796	83.704	53.438	113.926	102.348	55.397	69.343
5	51.721	83.181	79.874	53.229	109.883	106.171	53.800	69.480
6	51.114	92.214	88.591	52.964	111.491	103.959	55.376	73.810
7	50.387	75.696	73.336	52.233	110.462	105.117	54.022	69.970
8	51.847	88.370	85.324	52.491	112.445	103.510	55.431	72.453
9	53.066	87.428	83.022	50.431	110.595	103.421	54.707	77.748
10	51.523	89.321	83.808	51.372	107.363	100.652	54.176	65.063
11	51.436	90.153	85.937	51.683	110.498	104.427	56.008	74.078
12	53.863	92.575	88.220	52.291	113.709	105.180	54.284	71.396
13	53.436	90.783	87.612	51.500	110.595	104.758	53.889	80.493
Diferença percentual	14,42	21,26	27,28	15,26	5,76	**5,20**	13,48	32,94

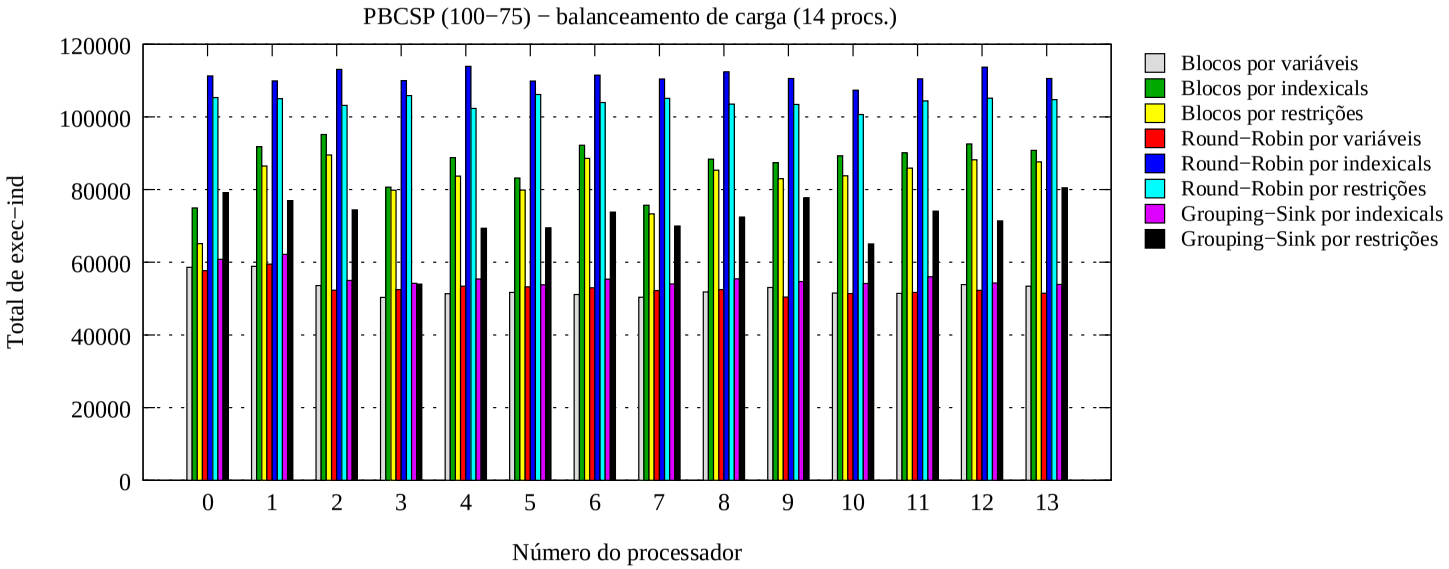


Figura 6.33: PBCSP (100 vars. - 0,75) - balanceamento de carga para 14 processadores

PBCSP (100 vars. - 0,75) - número de falhas para 14 processadores								
Procs.	Blocos			*Round-Robin*			*Grouping-Sink*	
	Variáveis	*Indexicals*	Restrições	Variáveis	*Indexicals*	Restrições	*Indexicals*	Restrições
0	3	1	0	3	3	4	0	0
1	3	3	5	3	1	3	6	3
2	0	0	2	3	4	1	1	4
3	3	2	0	3	1	4	2	1
4	3	4	1	0	4	1	0	1
5	3	8	9	3	5	6	4	2
6	4	3	1	1	3	2	7	9
7	2	1	1	4	1	3	1	1
8	5	1	4	5	3	1	0	1
9	5	9	6	0	1	3	3	4
10	1	1	2	1	5	3	2	2
11	0	1	1	2	3	1	1	2
12	3	0	2	5	0	3	3	2
13	1	1	1	2	3	2	5	3
Diferença entre < e >	5	9	9	5	5	5	7	9
Total	36	**35**	**35**	**35**	37	37	35	35

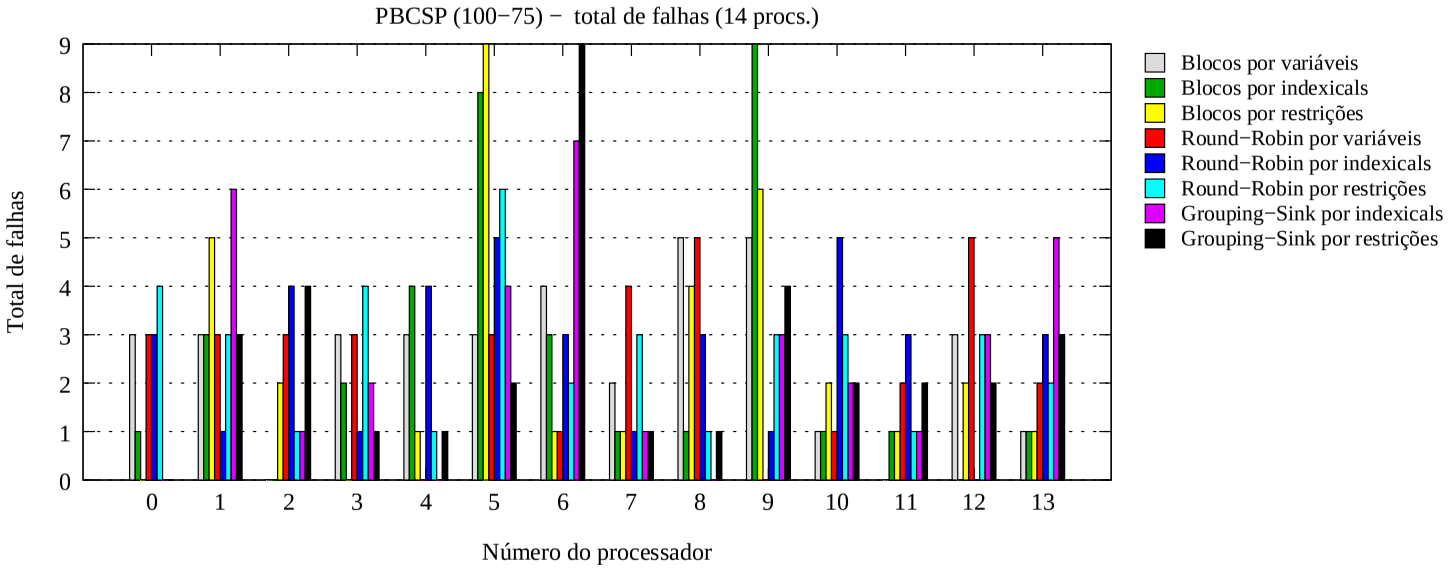


Figura 6.34: PBCSP (100 vars. - 0,75) - total de falhas para 14 processadores

Símbolo	Tempo de execução	Total de trabalho	Total de mensagens	Balanceamento de carga (%)	Número de falhas
+ + + +	6,67 a 7,33	824.055 a 904.010	39.557 a 45.466	2,65 a 6,70	35 a 35
+ + +	7,34 a 7,99	904.011 a 983.966	45.467 a 51.376	6,71 a 10,75	36 a 36
+ +	8,00 a 8,65	983.967 a 1.063.921	51.377 a 57.286	10,76 a 14,80	37 a 37
+	8,66 a 9,31	1.063.922 a 1.143.877	57.287 a 63.196	14,81 a 18,84	38 a 38
-	9,32 a 9,97	1.143.878 a 1.223.833	63.197 a 69.105	18,86 a 22,89	39 a 39
- -	9,98 a 10,63	1.223.834 a 1.303.788	69.106 a 75.015	22,90 a 26,94	40 a 40
- - -	10,64 a 11,29	1.303.789 a 1.383.744	75.016 a 80.925	26,95 a 30,99	41 a 41
- - - -	11,30 a 11,95	1.383.745 a 1.463.700	80.926 a 86.835	31,00 a 35,04	42 a 42

Particionamento	Tempo de execução	Total de trabalho	Total de mensagens	Balanceamento de carga	Número de falhas	Total geral
Blocos por variáveis	+ + + +	+ + + +	+ + + +	- - - -	+ + + +	+12
Blocos por *indexicals*	- -	+	-	+	+ + + +	+3
Blocos por restrições	- -	-	-	+	+ + + +	+1
Round-Robin por variáveis	+ + + +	+ + + +	+ + + +	+ + +	+ +	+17
Round-Robin por *indexicals*	- - - -	- - - -	- - -	+ + +	+ + + +	-4
Round-Robin por restrições	- - - -	- - - -	- - - -	+ + + +	+ + + +	-4
Grouping-Sink por *indexicals*	+ + + +	+ + + +	+ + + +	+ + +	+ + + +	**+19**
Grouping-Sink por restrições	-	+ + +	+ +	+	+ + + +	+9

Figura 6.35: PBCSP (100 vars. - 0,75) - resumo para 8 processadores

Símbolo	Tempo de execução	Total de trabalho	Total de mensagens	Balanceamento de carga (%)	Número de falhas
+ + + +	3,47 a 3,84	741.156 a 842.921	71.448 a 85.088	5,20 a 8,67	35 a 35
+ + +	3,85 a 4,22	842.922 a 944.686	85.089 a 98.728	8,68 a 12,13	36 a 36
+ +	4,23 a 4,59	944.687 a 1.046.452	98.729 a 112.368	12,14 a 15,60	37 a 37
+	4,60 a 4,97	1.046.453 a 1.148.217	112.369 a 126.009	15,61 a 19,07	38 a 38
-	4,98 a 5,34	1.148.218 a 1.249.982	126.010 a 139.649	19,08 a 22,54	39 a 39
- -	5,35 a 5,71	1.249.983 a 1.351.748	139.650 a 153.289	22,55 a 26,01	40 a 40
- - -	5,72 a 6,09	1.351.749 a 1.453.513	153.290 a 166.929	26,02 a 29,47	41 a 41
- - - -	6,10 a 6,46	1.453.514 a 1.555.279	166.930 a 180.570	29,48 a 32,94	42 a 42

Particionamento	Tempo de execução	Total de trabalho	Total de mensagens	Balanceamento de carga	Número de falhas	Total geral
Blocos por variáveis	+ + + +	+ + + +	+ + + +	+ +	+ + +	+17
Blocos por *indexicals*	- - -	-	- -	-	+ + + +	-3
Blocos por restrições	- - -	-	- -	- - -	+ + + +	-5
Round-Robin por variáveis	+ + + +	+ + + +	+ + + +	+ +	+ + + +	**+18**
Round-Robin por *indexicals*	- - - -	- - - -	- - - -	+ + + +	+ +	-6
Round-Robin por restrições	- - - -	- - - -	- - - -	+ + + +	+ +	-6
Grouping-Sink por *indexicals*	+ + + +	+ + + +	+ + + +	+ +	+ + + +	**+18**
Grouping-Sink por restrições	- - -	+ +	+	- - - -	+ + + +	0

Figura 6.36: PBCSP (100 vars. - 0,75) - resumo para 14 processadores

6.6 *Sudoku*

A Figura 6.37 apresenta os tempos de execução para todos os particionamentos. O tempo de execução para 1 processador apresenta o percentual de diferença entre todos os particionamentos de 1,5%. Para 2 processadores o menor tempo de execução é apresentado pelo particionamento Blocos por *indexicals*. Para 4 processadores o menor tempo é do particionamento Blocos por variáveis. Para 8, 12 e 14 processadores Blocos por restrições tem o menor tempo. *Grouping-Sink* por restrições apresenta menores tempos do que *Grouping-Sink* por *indexicals*. A diferença entre os tempos de execução de Blocos por restrições e Blocos por *indexicals* é menor do que 3%. E entre Blocos por restrições e *Grouping-Sink* por restrições é menor do que 27,5%.

A quantidade de trabalho realizado, apresentado na Figura 6.38, mostra que o particionamento Blocos por restrições apresenta a menor quantidade de trabalho para 2, 8, 12 e 14 processadores. Os resultados de total de mensagens (Figura 6.39) também mostram que Blocos por restrições gera menos mensagens que os demais particionamentos, a partir de 8 processadores. Em relação a Blocos por *indexicals* a diferença é menor do que 5% em quantidade de trabalho e do que 2% em total de mensagens. A diferença percentual de Blocos por restrições para *Grouping-Sink* por restrições, em quantidade de trabalho é menor do que 31% e em total de mensagens chega a 66,4% para 14 processadores, favorável a Blocos por restrições.

Pelas Figuras 6.40 e 6.41 podemos observar que o balanceamento de carga não é bom para nenhum dos três tipos de particionamentos em Blocos. *Round-Robin* por variáveis apresenta o menor desbalanceamento de carga para 8 processadores (diferença de 22,55% entre os processadores) e *Round-Robin* por restrições para 14 processadores (diferença de 17,47% entre os processadores).

As Figuras 6.42 e 6.43 mostram que Blocos por variáveis apresenta a menor quantidade de falhas para 8 processadores (36.192 falhas) e para 14 processadores (36.217 falhas). Blocos por restrições apresenta 0,1% a mais de falha do que Blocos por variáveis, para 8 e para 14 processadores.

Analisando estes dados, percebemos que o que mais interfere no tempo de execução de *Sudoku* é a quantidade de trabalho e o total de mensagens.

As Figuras 6.44 e 6.45 apresentam um resumo dos resultados obtidos para *Sudoku*. Ambas as figuras mostram que Blocos por restrições e por *indexicals* apresentam os melhores resultados. *Grouping-Sink* apresenta resultados intermediários. Sendo que *Grouping-Sink* por restrições é melhor do que *Round-Robin* em número de mensagens e tempo de execução.

Sudoku - tempo de execução total																
Procs.	Blocos						*Round-Robin*						*Grouping-Sink*			
	Variáveis		*Indexicals*		Restrições		Variáveis		*Indexicals*		Restrições		*Indexicals*		Restrições	
1	52,91	(0,18)	52,84	(0,24)	53,05	(0,48)	52,64	(0,36)	**52,52**	(0,41)	52,58	(0,36)	52,85	(0,45)	53,30	(1,74)
2	59,38	(0,40)	**58,37**	(0,70)	58,39	(0,47)	89,45	(0,82)	79,16	(0,49)	90,93	(0,46)	87,90	(0,40)	79,93	(0,23)
4	**57,74**	(2,83)	60,09	(2,32)	59,36	(0,40)	76,95	(8,56)	75,77	(0,42)	81,44	(0,29)	77,23	(0,35)	77,22	(0,38)
8	64,06	(0,09)	57,75	(2,47)	**56,87**	(0,18)	82,29	(0,55)	83,51	(5,34)	89,35	(8,37)	83,30	(0,42)	77,87	(5,07)
12	76,87	(4,37)	64,05	(3,21)	**62,20**	(1,69)	89,14	(0,60)	97,76	(8,30)	106,03	(12,11)	95,73	(12,11)	84,39	(0,42)
14	82,83	(4,72)	67,13	(0,24)	**66,76**	(2,88)	104,62	(17,10)	105,26	(7,57)	107,95	(7,58)	98,47	(6,33)	92,03	(5,27)

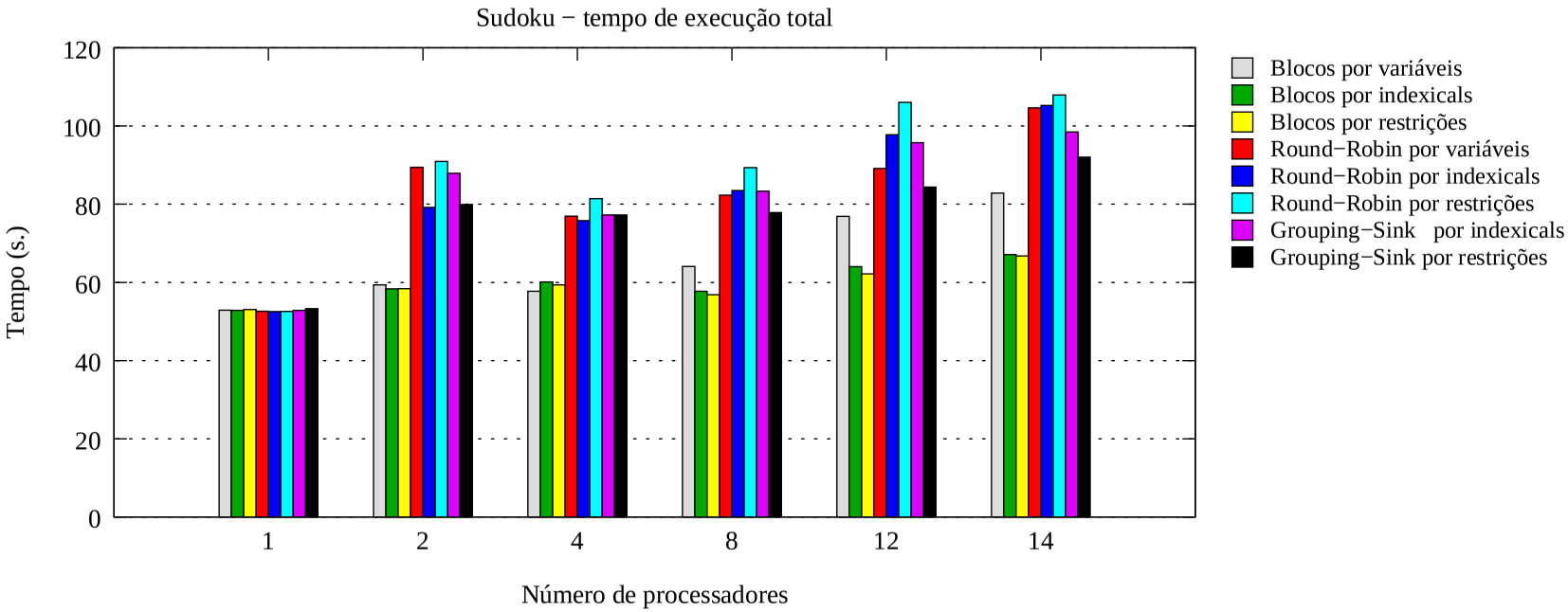


Figura 6.37: *Sudoku* - tempo de execução

Sudoku - total de trabalho								
Procs.	Blocos			*Round-Robin*			*Grouping-Sink*	
	Variáveis	*Indexicals*	Restrições	Variáveis	*Indexicals*	Restrições	*Indexicals*	Restrições
1	9.765.277	9.765.276	9.765.276	9.765.277	9.765.277	9.765.277	9.765.277	9.765.277
2	8.820.585	8.059.753	**8.056.210**	12.504.721	14.356.744	10.779.887	12.821.960	11.568.763
4	8.728.034	**8.191.212**	8.199.866	14.041.379	15.711.721	10.791.757	14.471.306	11.781.274
8	9.892.316	8.176.227	**8.142.964**	16.478.662	16.015.393	11.155.295	16.445.182	11.139.206
12	10.865.352	8.895.011	**8.664.729**	16.098.728	16.264.218	10.937.834	15.262.122	11.781.512
14	10.553.390	8.938.505	**8.542.615**	16.550.449	16.513.840	11.165.328	16.030.094	11.672.103

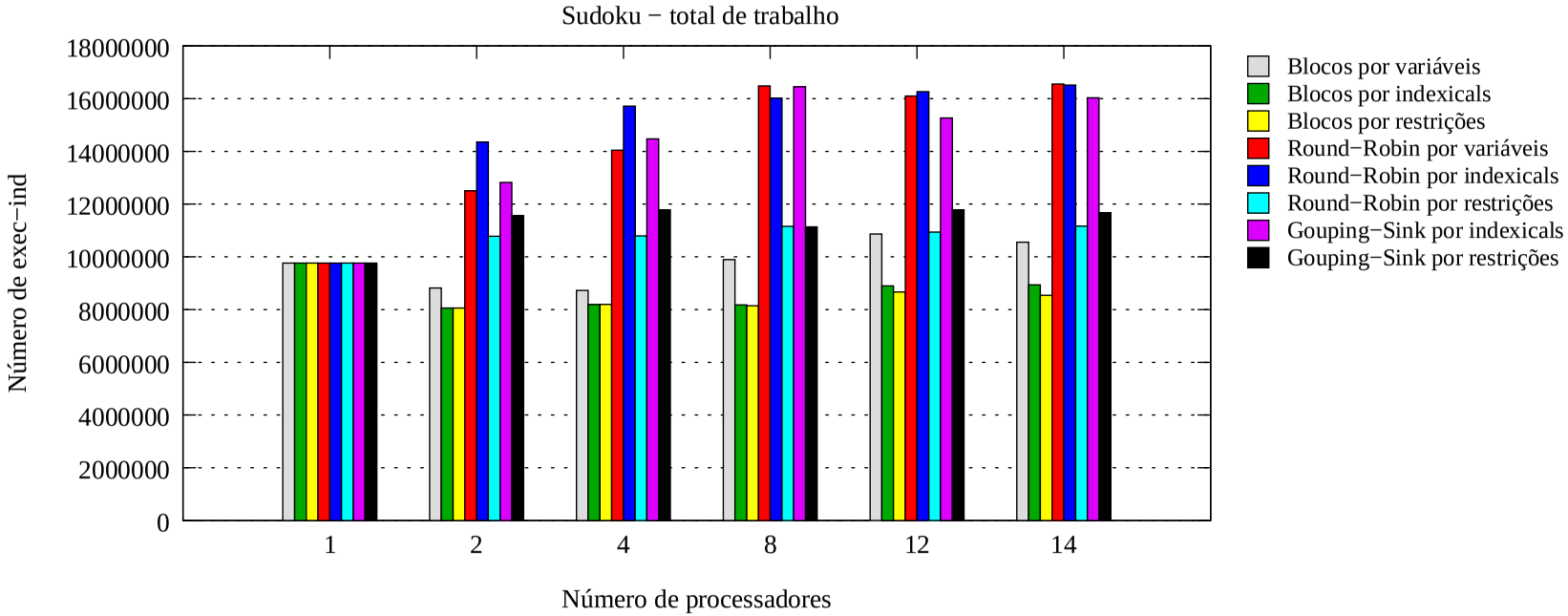


Figura 6.38: *Sudoku* - total de trabalho

Sudoku - total de mensagens								
Procs.	Blocos			*Round-Robin*			*Grouping-Sink*	
	Variáveis	*Indexicals*	Restrições	Variáveis	*Indexicals*	Restrições	*Indexicals*	Restrições
1	0	0	0	0	0	0	0	0
2	**706.004**	822.977	822.158	1.517.243	1.248.293	1.699.663	1.470.117	1.297.510
4	3.153.888	**3.081.901**	3.090.537	5.402.673	5.901.045	6.783.939	5.569.989	5.481.045
8	10.413.547	5.849.523	**5.762.555**	16.560.468	17.968.592	19.275.621	16.645.438	14.440.882
12	19.381.046	8.765.970	**8.646.962**	29.024.075	31.162.310	32.382.523	27.789.001	24.425.720
14	22.713.659	10.189.443	**10.004.831**	36.964.981	38.924.160	40.270.201	34.574.303	29.780.203

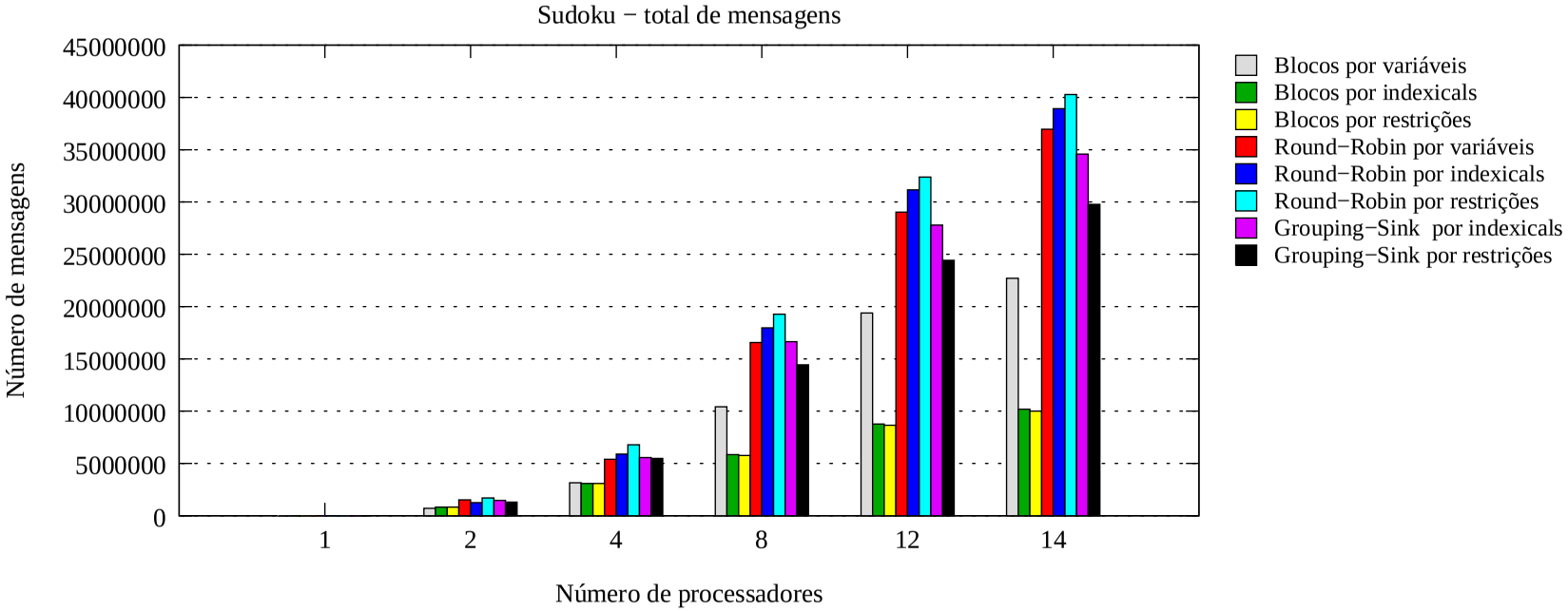


Figura 6.39: *Sudoku* - total de mensagens

Sudoku - balanceamento de carga para 8 processadores								
Procs.	Blocos			*Round-Robin*			*Grouping-Sink*	
	Variáveis	*Indexicals*	Restrições	Variáveis	*Indexicals*	Restrições	*Indexicals*	Restrições
0	1.050.099	1.070.821	1.023.213	1.744.435	1.840.578	1.138.661	1.491.598	2.213.141
1	1.428.153	1.057.964	1.108.796	2.186.641	2.376.231	1.426.104	2.086.604	1.404.781
2	2.570.125	565.479	566.065	2.244.349	2.178.072	1.487.039	2.115.361	1.438.698
3	907.156	1.751.818	1.732.965	2.252.323	2.248.636	1.378.260	2.389.322	1.462.041
4	873.811	600.830	603.411	2.063.510	832.459	1.444.111	1.849.513	1.113.443
5	1.124.023	1.317.257	1.293.686	2.022.629	2.175.595	1.369.796	2.105.787	1.129.771
6	846.808	570.987	563.859	1.992.907	2.263.230	1.431.038	2.351.482	1.136.116
7	1.092.141	1.241.071	1.250.969	1.971.868	2.100.592	1.480.286	2.055.515	1.241.215
Diferença percentual	67,05	67,72	67,46	**22,55**	64,97	23,43	37,57	49,69

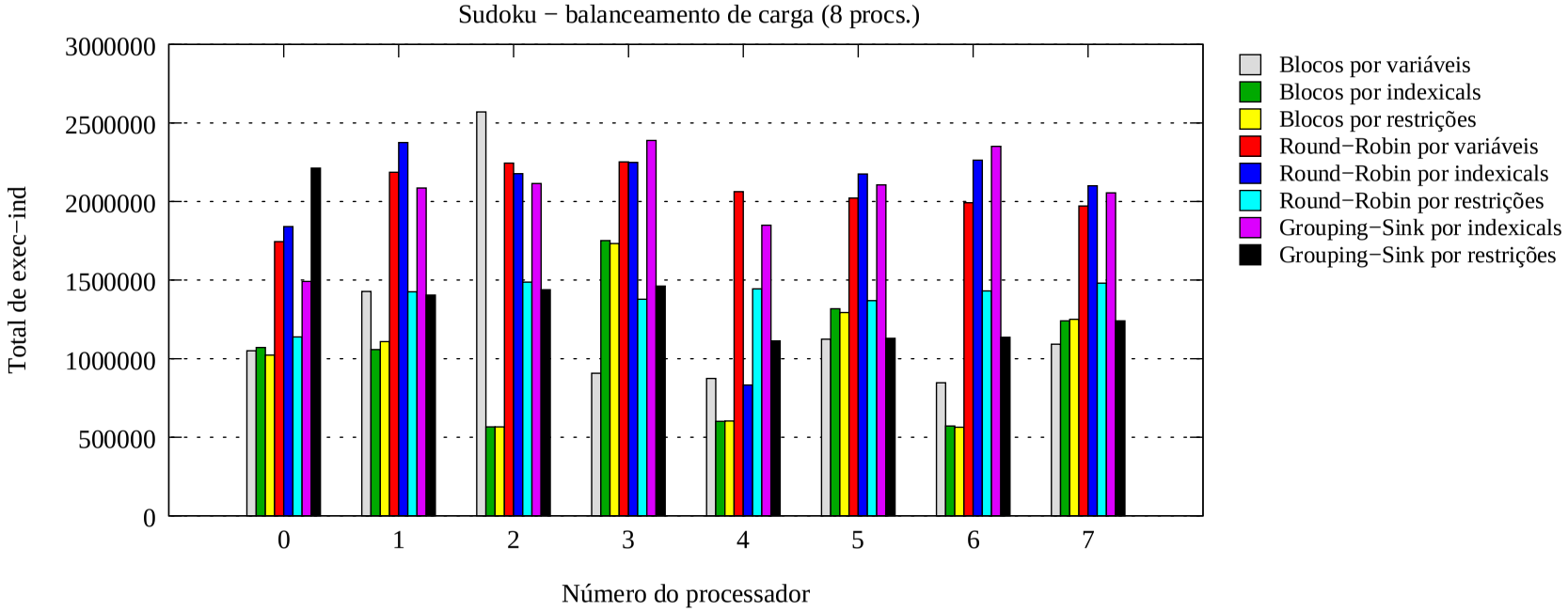


Figura 6.40: *Sudoku* - balanceamento de carga para 8 processadores

Sudoku - balanceamento de carga para 14 processadores								
Procs.	Blocos			*Round-Robin*			*Grouping-Sink*	
	Variáveis	*Indexicals*	Restrições	Variáveis	*Indexicals*	Restrições	*Indexicals*	Restrições
0	702.490	269.711	251.165	1.047.402	919.537	740.698	981.104	1.623.805
1	672.387	1.238.954	1.114.823	1.273.395	1.231.415	811.847	1.389.588	728.379
2	897.421	532.418	641.144	1.213.568	1.194.358	806.537	1.163.813	836.397
3	1.319.647	459.935	417.349	1.160.035	1.248.609	779.430	1.149.424	756.856
4	1.496.743	373.064	334.223	1.232.420	1.158.930	803.700	1.177.551	855.752
5	823.260	814.741	696.595	1.128.506	1.237.014	760.888	1.066.169	755.779
6	512.853	993.163	1.040.820	1.106.667	1.245.980	788.896	1.299.265	706.420
7	518.603	411.982	343.151	1.138.630	1.201.867	824.756	1.322.000	786.193
8	547.478	425.803	401.062	1.250.260	1.152.119	817.373	1.141.102	887.666
9	578.855	1.006.035	930.799	1.229.857	1.152.589	859.536	1.165.703	783.715
10	661.092	652.338	685.128	1.224.796	1.216.149	870.181	1.056.491	764.678
11	588.824	283.947	295.758	1.220.819	1.193.547	830.367	1.191.694	890.030
12	551.476	643.063	613.189	1.216.234	1.239.438	752.971	1.019.404	664.463
13	682.261	833.351	777.409	1.107.860	1.122.288	718.148	906.786	631.970
Diferença percentual	65,74	78,23	77,47	17,75	26,36	**17,47**	34,74	61,08

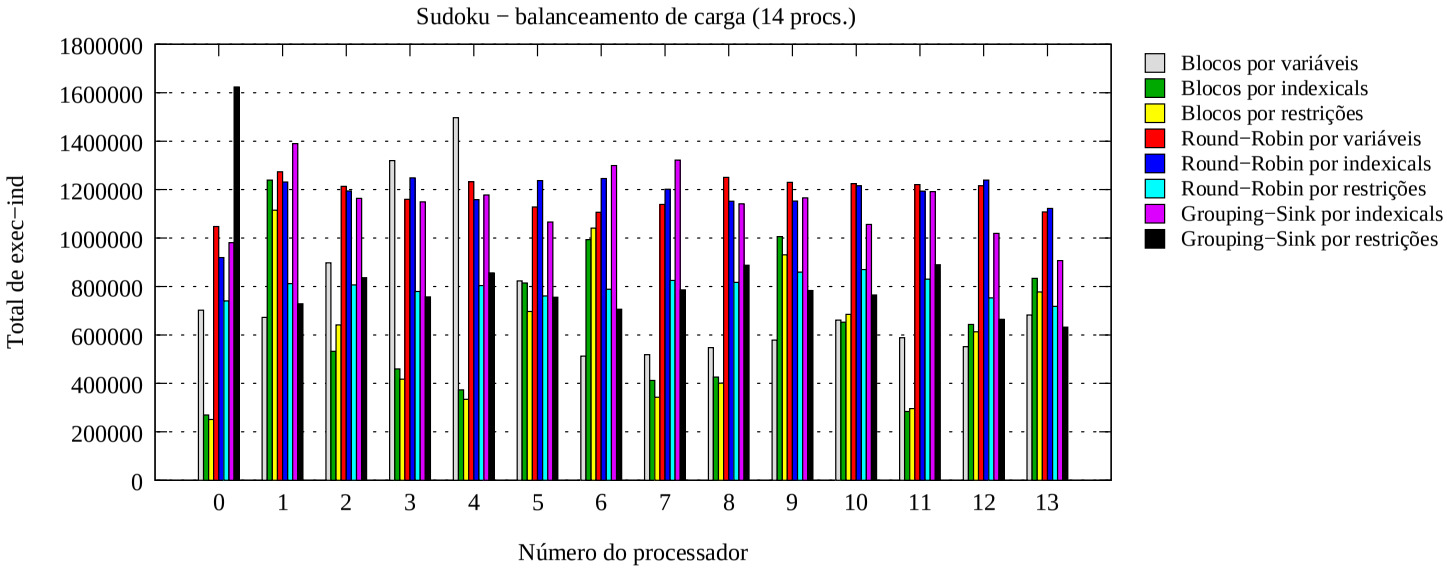


Figura 6.41: *Sudoku* - balanceamento de carga para 14 processadores

Sudoku - número de falhas para 8 processadores								
Procs.	Blocos			*Round-Robin*			*Grouping-Sink*	
	Variáveis	*Indexicals*	Restrições	Variáveis	*Indexicals*	Restrições	*Indexicals*	Restrições
0	707	4.822	4.707	4.996	4.887	8.717	4.824	8.602
1	13.538	5.037	4.985	4.987	4.095	5.892	3.281	9.888
2	15.827	1.544	1.613	6.512	3.901	7.247	2.018	9.630
3	654	17.046	16.988	7.399	5.822	5.924	6.648	6.200
4	835	2.510	2.462	3.155	80	6.425	2.574	2.346
5	2.442	2.191	2.189	3.693	8.722	2.159	5.342	2.917
6	834	372	559	4.431	5.655	3.211	3.294	2.462
7	1.355	2.781	2.719	2.415	5.172	7.470	8.877	4.785
Diferença entre < e >	15.173	16.674	16.429	4.984	8.642	6.558	6.859	7.542
Total	**36.192**	36.303	36.222	37.588	38.334	47.045	36.858	46.830

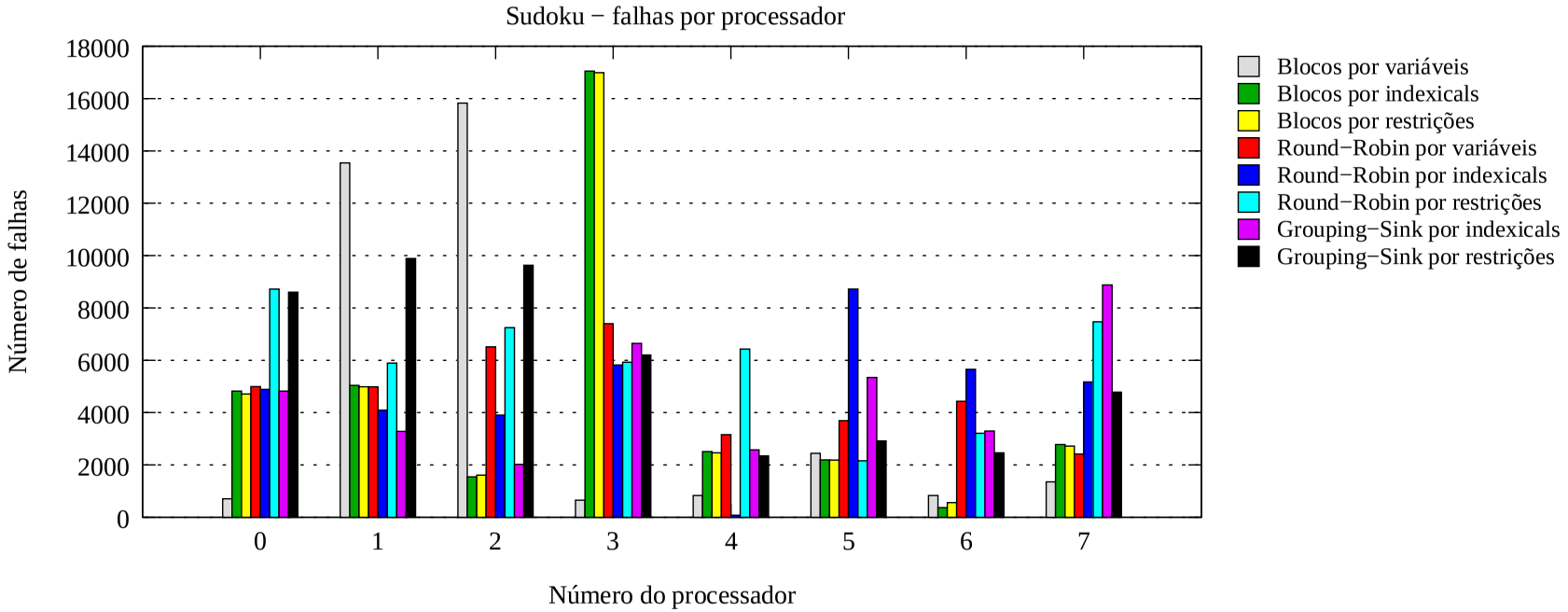


Figura 6.42: *Sudoku* - total de falhas para 8 processadores

Sudoku - número de falhas para 14 processadores								
Procs.	Blocos			*Round-Robin*			*Grouping-Sink*	
	Variáveis	*Indexicals*	Restrições	Variáveis	*Indexicals*	Restrições	*Indexicals*	Restrições
0	478	2.085	1.897	3.614	5.508	9.312	1.262	8.652
1	529	4.947	4.207	989	6.094	4.202	4.258	2.694
2	4.555	2.020	4.810	3.754	5.891	3.823	2.211	5.765
3	10.141	642	516	3.449	5.218	2.182	3.169	5.906
4	13.595	1.337	758	3.860	3.414	4.114	2.082	5.707
5	440	11.186	9.936	2.047	2.859	3.519	4.667	3.085
6	279	5.732	5.187	1.247	2.871	3.329	2.001	6.135
7	360	1.524	1.049	1.442	1.915	2.314	4.784	3.682
8	287	1.831	2.040	1.702	4.939	1.785	2.909	4.218
9	647	1.959	1.843	994	3.513	3.305	513	1.291
10	3.314	536	1.365	2.327	2.199	2.777	977	1.751
11	288	31	25	6.940	1.090	970	3.951	5.270
12	321	1.659	1.653	2.163	1.174	1.010	1.359	664
13	983	1.276	1.113	2.280	1.996	1.776	2.484	1.730
Diferença entre < e >	13.316	11.155	9.911	5.951	5.004	8.342	4.271	7.988
Total	**36.217**	36.765	36.399	36.808	48.681	44.418	36.627	56.550

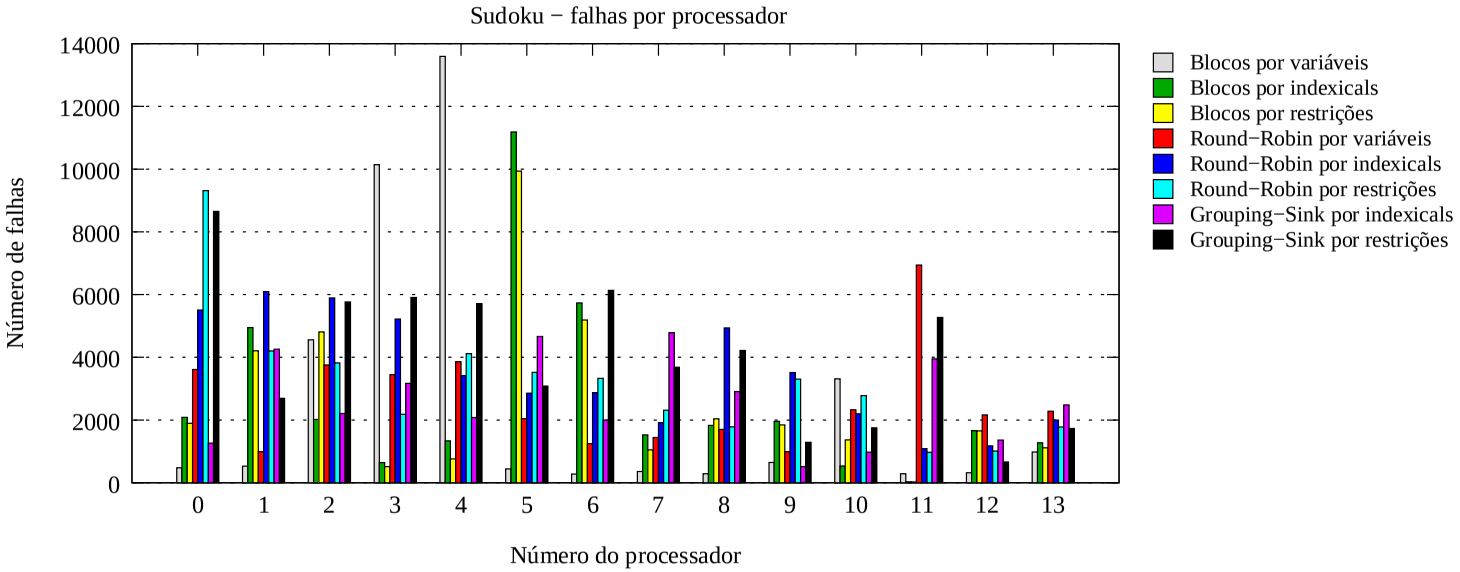


Figura 6.43: *Sudoku* - total de falhas para 14 processadores

Símbolo	Tempo de execução	Total de trabalho	Total de mensagens	Balanceamento de carga (%)	Número de falhas
+ + + +	56,87 a 60,93	8142.964 a 9.184.926	5.762.555 a 7.451.688	22,55 a 28,20	36.192 a 37.521
+ + +	60,94 a 64,99	9.184.927 a 10.226.888	7.451.689 a 9.140.821	28,21 a 33,84	37.522 a 38.851
+ +	65,00 a 69,05	10.226.889 a 11.268.850	9.140.822 a 10.829.954	33,85 a 39,49	38.852 a 40.181
+	69,06 a 73,11	11.268.851 a 12.310.813	10.829.955 a 12.519.088	39,50 a 45,14	40.182 a 41.511
-	73,12 a 77,17	12.310.814 a 13.352.775	12.519.089 a 14.208.221	45,15 a 50,78	41.512 a 42.840
- -	77,18 a 81,23	13.352.776 a 14.394.737	14.208.222 a 15.897.354	50,79 a 56,43	42.841 a 44.170
- - -	81,24 a 85,29	14.394.738 a 15.436.699	15.897.355 a 17.586.487	56,44 a 62,07	44.171 a 45.500
- - - -	85,30 a 89,35	15.436.700 a 16.478.662	17.586.488 a 19.275.621	62,08 a 67,72	45.501 a 46.830

Particionamento	Tempo de execução	Total de trabalho	Total de mensagens	Balanceamento de carga	Número de falhas	Total geral
Blocos por variáveis	+ + +	+ + +	+ +	- - - -	+ + + +	+8
Blocos por *indexicals*	+ + + +	+ + + +	+ + + +	- - - -	+ + + +	**+12**
Blocos por restrições	+ + + +	+ + + +	+ + + +	- - - -	+ + + +	**+12**
Round-Robin por variáveis	- - -	- - - -	- - -	+ + + +	+ + +	-3
Round-Robin por *indexicals*	- - -	- - - -	- - - -	- - - -	+ + +	-12
Round-Robin por restrições	- - - -	+ +	- - - -	+ + + +	- - - -	-6
Grouping-Sink por *indexicals*	- - -	- - - -	- - -	+ +	+ + + +	-4
Grouping-Sink por restrições	- -	+	- -	-	- - - -	-8

Figura 6.44: *Sudoku* - resumo para 8 processadores

Símbolo	Tempo de execução	Total de trabalho	Total de mensagens	Balanceamento de carga (%)	Número de falhas
+ + + +	66,76 a 71,91	8.542.615 a 9.543.594	10.004.831 a 13.788.002	17,47 a 25,06	36.217 a 38.758
+ + +	71,92 a 77,06	9.543.595 a 10.544.573	13.788.003 a 17.571.173	25,07 a 32,66	38.759 a 41.300
+ +	77,07 a 82,21	10.544.574 a 11.545.552	17.571.174 a 213.54344	32,67 a 40,25	41.301 a 43.841
+	82,22 a 87,36	11.545.553 a 12.546.532	21.354.345 a 25.137.516	40,26 a 47,85	43.842 a 46.383
-	87,37 a 92,50	12.546.533 a 13.547.511	25.137.517 a 28.920.687	47,86 a 55,44	46.384 a 48.925
- -	92,51 a 97,65	13.547.512 a 14.548.490	28.920.688 a 32.703.858	55,45 a 63,04	48.926 a 51.466
- - -	97,66 a 102,80	14.548.491 a 15.549.469	32.703.859 a 36.487.029	63,05 a 70,63	51.467 a 54.008
- - - -	102,81 a 107,95	15.549.470 a 16.550.449	36.487.030 a 40.270.201	70,64 a 78,23	54.009 a 56.550

Particionamento	Tempo de execução	Total de trabalho	Total de mensagens	Balanceamento de carga	Número de falhas	Total geral
Blocos por variáveis	+	+ +	+	- - -	+ + + +	+5
Blocos por *indexicals*	+ + + +	+ + + +	+ + + +	- - - -	+ + + +	**+12**
Blocos por restrições	+ + + +	+ + + +	+ + + +	- - - -	+ + + +	**+12**
Round-Robin por variáveis	- - - -	- - - -	- - - -	+ + + +	+ + + +	-4
Round-Robin por *indexicals*	- - - -	- - - -	- - - -	+ + +	-	-10
Round-Robin por restrições	- - - -	+ +	- - - -	+ + + +	+	-1
Grouping-Sink por *indexicals*	- - -	- - - -	- - -	+ +	+ + + +	-4
Grouping-Sink por restrições	-	+	-	- -	- - - -	-7

Figura 6.45: *Sudoku* - resumo para 14 processadores

6.7 Discussão

As aplicações utilizadas nos experimentos apresentam características de grafos de restrições diferentes. Por isso, um particionamento que pode ser considerado bom para uma aplicação pode não ser bom para as demais. Os experimentos foram realizados em uma máquina de memória compartilhada, onde a troca de informação entre os processadores é realizada através de escrita em memória compartilhada. Então, a escrita em memória compartilhada (que chamamos de mensagem), não causa impacto significativo no tempo de execução. Os resultados mostram que o balanceamento de carga e o total de falhas ocorridas nos processadores são os fatores que causam mais atraso na execução. O tempo de execução é determinado pelo processo que demorar mais tempo para realizar todo o seu processamento. Um processador com maior número de falhas tende a ter mais computação a realizar e demora mais tempo para terminar sua execução, impactando no tempo de execução total.

É importante destacar que o foco destes experimentos não é obter *speedup*, mas avaliar os particionamentos.

Num sistema distribuído, o número de mensagens trocadas entre os processadores causa um impacto grande no tempo de execução. Por isso, o total de mensagens geradas é um dos fatores principais para escolhermos um determinado particionamento. Um particionamento que minimize a troca de mensagens, sem ter um grande impacto no tempo de execução devido ao total de trabalho realizado, ao balanceamento de carga e ao número de falhas, é melhor.

A Tabela 6.3 apresenta um resumo de todas as aplicações e dos particionamentos que mais se destacaram.

Aplicação	Blocos			*Round-Robin*			*Grouping-Sink*	
	Variáveis	*Indexicals*	Restrições	Variáveis	*Indexicals*	Restrições	*Indexicals*	Restrições
Arithmetic								✓
Queens	✓						✓	
PBCSP (50 vars.)	✓			✓			✓	
PBCSP (100 vars.)				✓			✓	
Sudoku		✓	✓					

Tabela 6.3: Resumo de todos os particionamentos

Para *Arithmetic*, *Grouping-Sink* por restrições apresentou os melhores resultados. Mas os resultados de *Grouping-Sink* por *indexicals* é intermediário aos particionamentos. Para *Arithmetic*, o balanceamento de carga é o fator que gera maior impacto no tempo de execução. Porém, se observarmos o número de mensagens, *Grouping-Sink* é o que apresenta a menor quantidade. Numa arquitetura distribuída isto pode ser uma grande vantagem do *Grouping-Sink* por restrições.

Para *Queens*, Blocos por variáveis apresentou resultados um pouco melhores que *Grouping-Sink* por *indexicals* tanto em tempo de execução quanto em total de mensagens. Mas *Grouping-Sink* por *indexicals* é o melhor para 8 processadores. Para *Queens* a quantidade de falhas e o balanceamento de carga durante a execução geram um forte impacto no tempo de execução.

Para PBCSP (50 vars.), os particionamentos Blocos por variáveis, *Round-Robin* por variáveis e *Grouping-Sink* por *indexicals* apresentaram os melhores resultados. *Grouping-Sink* por *indexicals* apresenta um percentual inferior a 2,5% em tempo de execução, inferior a 5,5% em total de mensagens e a 21% em balanceamento de carga em relação a Blocos por variáveis e *Round-Robin* por variáveis.

Para PBCSP (100 vars.), os particionamentos *Round-Robin* por variáveis e *Grouping-Sink* por *indexicals* apresentam os melhores resultados.

A quantidade de trabalho realizado e o balanceamento de carga na execução de PBCSP gera um impacto no tempo de execução total. Para 8 processadores, com 50 variáveis, *Round-Robin* por restrições só foi melhor que Blocos por variáveis porque apresentou um balanceamento de carga melhor. Mas para as outras quantidades de processadores Blocos por variáveis realiza menos trabalho e tem um balanceamento de carga razoável e, por isso, apresentou um tempo de execução melhor.

Sudoku apresentou os melhores resultados com o particionamento Blocos por restrições, seguido por Blocos por *indexicals*. Mas *Grouping-Sink* por restrições apresenta quantidade de mensagens e tempo de execução intermediários aos particionamentos.

Com os dados apresentados para todas as aplicações, percebemos que *Grouping-Sink* por *indexicals* e por restrições tiverem um desempenho, em termos de tempo de execução, intermediário ou melhor do que os outros particionamentos. Desta forma, *Grouping-Sink* pode ser considerado o candidato mais próximo de um particionamento geral para todas as aplicações.

Ao contrário do que ocorre na execução num sistema de memória compartilhada, num sistema distribuído o total de mensagens pode impactar mais no tempo de execução do que o balanceamento de carga. *Grouping-Sink* busca minimizar a quantidade de mensagens trocadas durante a execução, tornando-o ainda mais forte como candidato ao particionamento de aplicações para um ambiente de *hardware* distribuído.

Para aplicações com características semelhantes a *Arithmetic*, *Grouping-Sink* por restrições é a melhor opção.

É importante ressaltar que para executar *Grouping-Sink* por *indexicals* é necessário que a aplicação esteja expressa na forma de *indexicals*. Para isso, é necessário utilizar um pré-processador para realizar a conversão de restrições em *indexicals*.

Os resultados que foram obtidos, utilizaram a realocação dos grupos gerados por

Grouping-Sink atribuindo o primeiro grupo ao primeiro processador, o segundo grupo ao segundo processador e assim por diante. Foram estudados alguns métodos para realizar a realocação, mas este foi o que apresentou melhores resultados.

O primeiro método estudado foi dividir o número de grupos pelo número de processadores disponíveis. Quando a divisão não for exata, é atribuído um grupo a mais aos processadores iniciais. Desta forma, temos fatias a serem atribuídas aos processadores que possuem grupos seqüenciais. Por exemplo, considere que há 10 grupos e 4 processadores disponíveis. Os dois primeiros processadores ficariam com 3 grupos e os demais com 2 grupos. Comparando os resultados de tempo de execução obtidos com este método e com o *Round-Robin*, este método apresentou tempos mais altos.

O segundo método estudado considerou que os grupos que sobrassem na divisão pelos processadores disponíveis deveriam ser atribuídos aos processadores finais. Considerando o exemplo anterior, os 2 primeiros processadores receberiam 2 grupos consecutivos e os 2 processadores finais receberiam 3 grupos. Neste caso também os tempos obtidos foram piores do que atribuir de maneira *Round-Robin*.

O terceiro método consistiu num algoritmo que juntava os grupos considerando as variáveis já atribuídas aos processadores disponíveis. Mas este método também não produziu melhores resultados. Outras formas de realocação podem ser estudadas para se conseguir melhores resultados de realocação de grupos, inclusive reutilizar decomposição em *sinks*, mas não faz parte do escopo desta tese.

A orientação acíclica inicial utilizada para orientar um grafo sobre o qual *Grouping-Sink* atua é aleatória. Portanto, a orientação que foi utilizada nos experimentos não foi necessariamente a melhor. Desta forma, estes resultados ainda podem ser melhorados se a orientação acíclica inicial utilizada for a melhor possível.

A análise dos particionamentos foi realizada para um sistema de memória compartilhada. Uma outra análise que pode ser realizada é a utilização dos particionamentos em um ambiente de *hardware* distribuído, para avaliar melhor o impacto da troca de mensagens entre os processadores. Porém, consideramos que o estudo realizado neste trabalho já oferece subsídios suficientes para julgar *Grouping-Sink* como sendo uma boa alternativa para sistemas distribuídos, visto que na maioria dos casos estudados, há redução em quantidade de mensagens comparado com os outros particionamentos. Para 8 processadores a redução no número de mensagens atinge 86% e para 14 processadores 92%.

Capítulo 7

Conclusões e trabalhos futuros

Problemas de Satisfação de Restrições são problemas de busca NP-completos mas que, de acordo com suas características, podem ser resolvidos beneficiando-se de uma execução local distribuída. Sendo assim, o particionamento de restrições em CSPs torna-se importante, podendo gerar bons resultados em ambientes de execução seqüencial e paralela/distribuída.

Este texto apresenta um método de particionamento automático de restrições com a finalidade de melhorar a eficiência de execução de CSPs. A pesquisa de um método automático partiu do estudo de métodos manuais que exploravam as características das aplicações. Em trabalho anterior [59], o método de particionamento utilizado foi baseado em Escalonamento por Reversão de Arestas. Este método toma como entrada um grafo orientado e iterativamente reverte arestas de nós sumidouros, formando grupos de nós sumidouros que podem operar simultaneamente, até que um período seja encontrado. Naquele trabalho, utilizamos três algoritmos de orientação acíclica e constatamos que: o método de reversão de arestas tem um custo computacional alto e o número de grupos é fortemente influenciado pelo algoritmo de orientação acíclica utilizado. Analisando os resultados daquele trabalho, implementamos um novo método de particionamento automático, denominado *Grouping-Sink*, que gera o mesmo resultado que o SER, mas com tempo de execução menor. Há duas versões implementadas: *Grouping-Sink* por restrições e por *indexicals*.

Grouping-Sink utiliza o algoritmo de orientação acíclica *Alg-Colour*, que produz o menor número de grupos, e o algoritmo de decomposição em *sinks* para gerar o particionamento. Resultados experimentais mostraram que *Grouping-Sink* por *indexicals* gera resultados similares aos melhores resultados apresentados pelos outros métodos de particionamento aos quais foi comparado. A vantagem de *Grouping-Sink* por *indexicals* é que podemos utilizar um único método de particionamento para particionar diferentes CSPs (com diferentes grafos de restrições).

Este trabalho apresenta, também, novas formas de visualizar o grafo de restrições.

Tradicionalmente, o grafo de restrições considera que os nós são as variáveis e as arestas são as restrições. Porém, o grafo pode ser considerado com os nós representando as restrições e as arestas são as variáveis comuns entre as restrições (dependências entre as restrições). Utilizando o esquema de *indexicals*, as restrições são representadas na forma de *indexicals* (que explicitam cada uma das variáveis da restrição). O grafo de restrições então é elaborado baseado nos *indexicals*. Desta forma, os nós representam os *indexicals* e as arestas são as variáveis comuns entre os *indexicals*. Este grafo possui uma granulosidade mais fina e permite agrupamentos melhores dos *indexicals*, aumentando a computação local. Os resultados experimentais apresentados por *Grouping-Sink* por *indexicals* foram um pouco melhores do que *Grouping-Sink* por restrições devido à granulosidade mais fina permitir agrupamentos melhores.

Neste trabalho, apresentamos ganhos com *Grouping-Sink* em relação a outros métodos de particionamento normalmente utilizados na literatura, mesmo com uma implementação não otimizada. Um dos caminhos naturais deste trabalho seria a otimização da implementação de *Grouping-Sink*. Atualmente, a realocação dos grupos é feita atribuindo o primeiro grupo ao primeiro processador, o segundo grupo ao segundo processador e assim por diante. Pode-se investigar a melhor forma de realizar esta realocação aproveitando, por exemplo, a informação de quais variáveis estão contidas nos grupos, numa tentativa de aumentar a computação local e diminuir a comunicação entre os processadores.

Um outro caminho que poderia ser investigado seria a implementação da geração da orientação acíclica inicial com *Alg-Colour* de forma distribuída. Em aplicações que possuem grafos com densidade muito alta, como *Queens*, a execução seqüencial requer muita memória para armazenar os dados, o que pode requerer um tempo grande para a geração da orientação acíclica.

O SER pode ser usado para fazer alocação dinâmica das restrições aos processadores, usando o grafo original. Em cada estado gerado pelas reversões de arestas os nós *sinks* seriam alocados a processadores diferentes.

O grafo de restrições considerado neste trabalho possui o mesmo peso para todos os nós e as arestas. Mas se forem considerados pesos diferentes para os nós e para as arestas, baseado no número de variáveis existentes em um nó e a quantidade de dependências existentes entre os nós, a orientação acíclica inicial poderá gerar *sinks* que possuam maior dependência. Assim, a quantidade de comunicação pode ser reduzida, pois haverá mais computação local.

Neste trabalho, por problemas de acesso a outras aplicações escritas em programação lógica com restrições, nos restringimos apenas a 4 aplicações. Embora consideremos que estas 4 aplicações representem um espectro variado de grafos, seria interessante estender este trabalho para outras aplicações e aplicações reais.

Nesta tese a análise dos particionamentos foi realizada para um sistema de memória compartilhada. Uma outra análise que pode ser realizada é a utilização dos particionamentos em um ambiente de *hardware* distribuído, para avaliar melhor o impacto da troca de mensagens entre os processadores.

Um aspecto não abordado nesta tese está relacionado com outras formas de particionamento. A princípio podemos particionar conjuntos de restrições, assim como o domínio das variáveis. Em trabalho anterior, mostramos que o particionamento do domínio (ao qual denominamos *labeling* distribuído) poderia oferecer ganhos com aceleração superlinear [56]. Este tipo de particionamento é ortogonal ao particionamento de restrições e poderia ser uma alternativa interessante a ser combinada com nosso trabalho.

Apêndice A

Descrição de Algoritmos de Consistência de Arcos

Este apêndice apresenta uma classificação dos algoritmos de consistência de arcos e a descrição de alguns deles. Para facilitar a compreensão do texto, a Seção A.1 apresenta a notação utilizada. A Seção A.2 apresenta a classificação dos algoritmos e como os algoritmos se enquadram nesta classificação. Em cada seção seguinte é descrito um algoritmo, com sua respectiva complexidade.

A.1 Notação utilizada

No decorrer deste texto há referência a variáveis, domínios, restrições, etc. A notação que está sendo utilizada é a seguinte:

1. Variável: V_i, V_j, V_k, V_l, V_m, V_p
2. Domínio de variável: D_{V_i}, D_{V_j}, D_{V_k}, D_{V_l}
3. Valor de variável: a, b, w, x, y
4. Restrição: $R_{V_iV_j}$ (restrição entre as variáveis V_i e V_j), $R_{V_iV_j}(x, b)$ (restrição entre V_i e V_j para $V_i = x$ e $V_j = b$)
5. Arco: (V_i, V_j)
6. Conjunto de arcos: S
7. Fila de arcos: Q, Q'
8. Par variável-valor: (V_i, a) (a é o valor atribuído a V_i)
9. Conjunto suporte: $S_{V_{j1}}$ (conjunto suporte para $V_j = 1$)

10. Grau do vértice: g_{V_j} (grau do vértice do nó V_j)

11. Tamanho do domínio: d

12. Número de arestas: e

13. Número de variáveis: n

A.2 Classificação dos algoritmos de consistência de arcos

Segundo Bessiere *et al.* [13] , os algoritmos de consistência de arcos podem ser classificados de acordo com seus métodos de propagação. Existem dois métodos principais de propagação que são utilizados: propagação orientada a arco e orientada a valor. Na propagação orientada a arco o grafo de restrições possui variáveis nos nós e as arestas são as restrições. Esta abordagem surgiu com o AC-1 [43] .

Na propagação orientada a valor o grafo de restrições é baseado em valor, ou seja, cada restrição é representada como um subgrafo, onde um vértice é um valor e uma aresta é uma tupla permitida pela restrição correspondente. Este grafo é também conhecido como grafo de consistência ou micro-estrutura. Esta abordagem surgiu com o AC-4 [49].

Um nome mais específico para o grafo de restrições tradicional pode ser grafo de restrições baseado em variável.

A idéia principal da propagação baseada em valor é que uma vez que um valor é removido somente a viabilidade daqueles valores dependentes dele será verificada. A granulosidade deste algoritmo é mais fina.

Podemos classificar os algoritmos de propagação de restrições de acordo com a granulosidade. Segundo esta classificação há duas classes de algoritmos: os de granulosidade grossa e os de granulosidade fina. Os algoritmos de granulosidade grossa recebem um conjunto de valores removidos por uma das variáveis envolvidas na restrição e propaga esta remoção às outras variáveis da restrição. A propagação é orientada a restrições.

Nos algoritmos de granulosidade fina a remoção de um valor do domínio de uma variável é propagado somente para os valores afetados nos domínios das outras variáveis. Neste caso a propagação é orientada a valor.

Podemos citar como exemplos de algoritmos de granulosidade grossa AC-1, AC-2 [43], AC-3 [41] e AC2001/3.1 [13]. E como algoritmos de granulosidade fina AC-4, AC-6 [9] e AC-7 [10].

Uma grande vantagem dos algoritmos de granulosidade grossa em relação aos algoritmos de granulosidade fina é que são fáceis de serem integrados na implementação de um *solver* de restrições. Porém os algoritmos de granulosidade fina geralmente possuem

complexidade de tempo ótima no pior caso, o que não ocorre com os de granulosidade grossa.

AC-3 comparado com AC-4, AC-6 ou AC-7 possui a vantagem de não exigir estruturas de dados complexas em sua implementação.

É difícil para algoritmos de granulosidade grossa trabalhar com restrições funcionais e para algoritmos de granulosidade fina tratar restrições monotônicas. Por isso foi introduzido o algoritmo AC-5 [40]. Este algoritmo utiliza ambos os tipos de grafos.

No algoritmo AC-3 a fila a ser executada possui arcos (V_i, V_j). Em AC-4 a fila contém pares (V_i, a), onde V_i é um nó e a é um valor. Na fila de AC-5 há elementos do tipo $< (V_i, V_j), w >$, onde (V_i, V_j) é um arco e w é um elemento que foi removido de D_{V_j} e justifica a necessidade de considerar o arco (V_i, V_j) [40]. Portanto, AC-5 utiliza características de AC-3 e AC-4.

A.3 Algoritmo AC-1

As idéias que deram origem ao algoritmo AC-1 são apresentadas por Mackworth em [43] . AC-1 é um algoritmo de consistência de arcos de força bruta. Ele utiliza um procedimento denominado $REVISE(V_i, V_j)$, apresentado na Figura A.1. Este procedimento expressa a seguinte observação de Fikes [43] :

"Dados domínios discretos, D_{V_i} e D_{V_j}, para duas variáveis V_i e V_j com as quais já foi realizada a fase de consistência de nós, se x $\in D_{V_i}$ e não há y $\in D_{V_j}$ tal que a restrição entre os valores x e y (aresta entre os nós do grafo) seja satisfeita, então x pode ser retirado de D_{V_i}. Quando esta consistência for realizada para cada valor de $x \in D_{V_i}$ então o arco (V_i, V_j) (mas não necessariamente o arcoV_j, V_i) será consistente."

```
Procedure REVISE(V_i,V_j);
1    DELETE ←false;
2    foreach x ∈ D_Vi do
3        if there is no such V_j ∈ D_Vj
4            such that (x,V_j) is consistent,
5        then
6            delete x from D_Vi;
7            DELETE ←true;
8        endif;
9    endforeach;
10   return DELETE;
end_REVISE
```

Figura A.1: Procedimento $REVISE(V_i, V_j)$ [41]

O procedimento $REVISE(V_i, V_j)$ elimina valores inconsistentes dos domínios das variáveis, de acordo com as restrições. Este procedimento deve ser executado mais de

uma vez para cada arco do grafo de restrições para que o grafo se torne consistente. Isto significa que para cada valor x do domínio de uma variável V_k (D_{V_k}) é verificado se existe um valor y de uma variável V_l (D_{V_l}) que satisfaça à restrição (V_k, V_l). Caso não exista, o valor x é eliminado de D_{V_k}. A fila de arcos é fixa. Como o procedimento $REVISE(V_i, V_j)$ pode reduzir o domínio de alguma variável V_k, cada arco (V_l, V_k) tem que ser verificado novamente. Isto porque a poda de D_{V_k} pode ter removido o único valor que tornava algum dos elementos do domínio de V_l compatível com V_k.

O procedimento $REVISE(V_i, V_j)$ é chamado pelo algoritmo AC-1 para ser executado para cada arco da fila Q. O algoritmo AC-1 é mostrado na Figura A.2.

```
Procedure AC-1
1   Q ← { (Vi,Vj) ∈ arcs(G), i ≠ j};
2   repeat
3       CHANGE ← False;
4       foreach (Vi, Vj) ∈ Q do
5           CHANGE ← (REVISE (Vi,Vj) or CHANGE);
6       endforeach;
7   until not (CHANGE);
end_AC-1;
```

Figura A.2: Procedimento AC-1 [41]

Para realizar uma análise da complexidade deste algoritmo, consideramos que d é o tamanho do domínio, e é o número de arestas e n é o número de nós (número de variáveis).

A complexidade do procedimento $REVISE(V_i, V_j)$, na Figura A.1, é $O(d^2)$. Isto porque o *loop* $foreach$ (linhas 2 a 9) é executado para todos os valores de D_{V_i} e todos os valores de D_{V_j}. Desta forma, será percorrido todo o domínio d para cada par de variáveis do arco executado. Isto implica em obter d^2 iterações.

No procedimento AC-1, todos os arcos (V_i, V_j) da fila Q são executados pelo *loop* $for\ each$ (linhas de 4 a 6). Assim, a complexidade para este *loop* é $O(e)$. Cada vez que um valor do domínio de alguma variável é eliminado, a variável $CHANGE$ é alterada. Se para uma determinada variável V_i apenas um valor do seu domínio é eliminado de cada vez e todos os valores do domínio de V_i, de tamanho d, são alterados e se isto ocorrer para todas as variáveis n, o *loop* $repeat$ (linhas 2 a 7) será executado $O(nd)$ vezes. Portanto, no pior caso a complexidade do algoritmo AC-1 é $O(end^3)$.

A desvantagem deste algoritmo é que a eliminação de algum valor do domínio de alguma variável durante a revisão de um arco numa iteração particular força a re-execução de todos os outros arcos na próxima iteração. Embora, somente uma pequena quantidade dos arcos possa ser afetada.

Observando este fato Waltz [43] implementou um outro algoritmo, que considera que os nós estão em ordem numérica. A descrição deste algoritmo encontra-se na seção A.4.

A.4 Algoritmo AC-2

O algoritmo AC-2 é baseado no algoritmo de força bruta AC-1. Foi desenvolvido por Waltz [43] , que considera que o grafo de restrições possui seus nós numerados. O algoritmo foi descrito por Waltz da seguinte forma:

1. Juntar a um nó todos os valores que não conflitam com valores de nós previamente atribuídos. Isto é, se é conhecido previamente que um arco deve ser rotulado de um conjunto S, então, não deve ser agrupado qualquer valor do nó que possa requerer que o arco seja rotulado com um elemento que não pertença a S;

2. Verificar os vizinhos deste nó que já tenham sido rotulados. Se algum de seus valores não tiver uma atribuição correspondente para o mesmo nó, então este valor deve ser eliminado;

3. Quando algum valor é retirado de um nó, verificar todos os seus vizinhos e se algum de seus valores pode ser eliminado. Caso possa, continuar este processo iterativamente até que nenhuma mudança possa ser realizada. Então, vá para o próximo nó que foi organizado numericamente.

A idéia deste algoritmo é tornar todos os arcos do grafo consistentes numa única passagem através dos nós. Para isso, é necessário assegurar que com a introdução de um nó V_i todos os arcos (V_k, V_m), onde $k, m \leq i$ e $k \neq m$ tenham tornado-se consistentes. Quando um nó V_{i+1} é introduzido todos os arcos que partem dele e chegam nele (partem e chegam de nós introduzidos anteriormente) podem estar inconsistentes e devem ser verificados. Se ao executar o procedimento REVISE((V_k, V_m)) é eliminado algum valor do domínio de V_k, então, os únicos arcos adicionais que precisam ser verificados são todos aqueles que chegam em V_k com a exceção de (V_m, V_k), $\{(V_p, V_k) \mid (V_p, V_k) \in arcs(G),\ p \leq i\ \ p \neq m\}$. O arco (V_m, V_k) não precisa ser verificado porque não pode ter se tornado inconsistente como um resultado direto da retirada de valores realizada em D_{V_k} pelo procedimento REVISE((V_k, V_m)). Os valores são eliminados por não haver valor correspondente no domínio D_{V_m}.

Estas idéias estão resumidas no algoritmo AC-2 [43], que está ilustrado na Figura A.3.

Quando um nó V_i é introduzido na i-ésima iteração das linhas 2-14, Q e Q' são inicializados nas linhas 2 e 3 para conter todos os arcos orientados de e para o nó V_i, respectivamente. Quando Q torna-se vazio pela iteração das linhas 5-10, o conteúdo de Q' é atribuído a Q e Q' torna-se vazio. No início da s-ésima iteração das linhas 4-13, Q consiste de todos os arcos orientados para nós dos $s-2$ arcos removidos de V_i que estão para serem verificados enquanto Q' está pronto para assegurar todos os arcos direcionados para nós com $(s-1)$ arcos removidos de V_i que precisam ser verificados como um resultado

```
Procedure AC-2
1 for i ← 1 until n do
2     Q ← {(V_i, V_j) | (V_i, V_j) ∈ arcs(G), j <i};
3     Q' ← {(V_j, V_i) | (V_j, V_i) ∈ arcs(G), j <i};
4     while Q not empty do
5         while Q not empty do
6             pop(V_k, V_m) from Q;
7             if REVISE((V_k, V_m)) then
8                 Q' ← Q' ∪ {(V_p, V_k) | (V_p, V_k) ∈ arcs(G), p≤i, p≠m};
9             end-if;
10        end-while;
11        Q ← Q';
12        Q' ← empty;
13    end-while;
14 end-for;
end_AC-2;
```

Figura A.3: Procedimento AC-2 [43]

```
Procedure AC-3
1   Q ← { (V_i,V_j) ∈ arcs(G), i ≠ j };
2   while Q not empty
3       select and delete any arc (V_k,V_m) from Q;
4       if (REVISE (V_k,V_m )) then
5           Q ← ∪ { (V_i,V_k) such that (V_i,V_k) ∈ arcs(G), i ≠ k, i ≠ m }
6       endif;
7   endwhile;
end_AC-3;
```

Figura A.4: Procedimento AC-3 [41]

da revisão dos arcos de Q. Este processo inicialmente espalha do nó V_i mas pode retornar para V_i se houver ciclos no grafo de arco de tamanho maior que 2. A forma particular de AC-2 deriva, em parte, da consideração de apenas uma situação em que o espalhamento da onda de revisão do arco será através de si mesmo.

Outro algoritmo que foi desenvolvido baseado em AC-1 é o AC-3, que é mostrado na próxima seção.

A.5 Algoritmo AC-3

O algoritmo AC-3 [41] também é baseado no AC-1. Porém, ele realiza o procedimento de verificação somente dos arcos que possivelmente foram afetados pela eliminação de um valor de uma determinada variável. Este algoritmo é apresentado na Figura A.4.

Como observado, na seção A.3, a complexidade do procedimento $REVISE(V_i, V_j)$ é $O(d^2)$. Suponha que o arco (V_k, V_m) seja executado até que sejam removidos todos

os valores do domínio de V_k (de tamanho d). O *loop* $while$ (linhas 2 a 7) é executado em função dos arcos (totalizando e arestas, no pior caso) e para cada arco é executado d vezes. Portanto, a complexidade do *loop* $while$ é $O(ed)$. Como a complexidade do $REVISE(V_i, V_j)$ é $O(d^2)$, verificamos que a complexidade do algoritmo AC-3 é $O(ed^3)$.

A.6 Algoritmo AC-4

O algoritmo AC-4 é baseado em técnicas utilizadas no sistema ALICE [49]. Este algoritmo foi projetado para resolver problemas combinatórios usando uma estratégia geral e unificada. O sistema ALICE utiliza o algoritmo para realizar a consistência de arcos.

AC-4 difere de AC-1, AC-2 e AC-3 pois utiliza a noção de conjunto suporte. Para entendermos melhor o conceito de conjunto suporte, considere um exemplo, onde temos dois nós V_i e V_j, com domínios $\{1, 2, 3, 5\}$ e $\{2, 3, 5\}$, respectivamente. Seja b um valor do domínio de V_i. O valor b somente é considerado um valor viável para V_i se tiver um mínimo de suporte dos valores dos domínios de cada um dos outros nós V_j. Considerando a restrição $i \neq j$, o conjunto suporte para $j = 1$ é igual a $S_{V_{j1}}$=$\{(1, 2), (1, 3), (1, 5)\}$.

Este algoritmo utiliza uma estrutura de dados representada por uma matriz M para armazenar as variáveis com os respectivos valores que foram eliminados. Além disso, utiliza uma estrutura de dados S para armazena o conjunto suporte de cada variável. A estrutura denominada $List$ é utilizada para controlar a propagação das restrições.

O algoritmo AC-4 [49] está descrito na Figura A.5. Este algoritmo possui dois passos: construção das estruturas de dados (linhas 1 a 15) e poda de valores inconsistentes (linhas 16 a 25).

No passo de construção das estruturas de dados, a matriz M, os conjuntos suporte S e a estrutura $List$ são preenchidos. A matriz M é da ordem do tamanho do domínio d. Entre as linhas 2 a 14 na matriz M são armazenados os valores que foram removidos do domínio. O preenchimento do conjunto suporte S para cada valor do domínio de uma variável é realizado após a execução da restrição R. Após executar todas as restrições, a estrutura $List$ é preenchida com os pares (V_i, b), V_i é uma variável e b é o valor do domínio desta variável que foi eliminado. A estrutura $List$ é utilizada no segundo passo do algoritmo.

O segundo passo consiste em eliminar os valores inconsistentes dos domínios das variáveis, baseado nos dados armazenados em $List$.

Podemos observar, que no primeiro passo do algoritmo, o comando *if* (linha 6) é executado no máximo ed^2 vezes, onde e é o número de arestas e d é o tamanho do domínio. O número de elementos nos conjuntos S_{V_jc} é da ordem de $O(ed^2)$. O comando *else* (linha 12) é executado no máximo ed vezes e o número total de contadores é da ordem de ed. A linha 14 é da ordem de nd, onde é o número de variáveis, pois insere pares únicos

Procedure AC-4
Step 1: Construction of the data structures
1 M:=0; $S_{V_i b}$:= Empty_set;
2 for $(V_i, V_j) \in G$ do
3 for b$\in D_{V_i}$ do
4 Total:=0;
5 for c $\in D_{V_j}$ do
6 if R(V_i, b, V_j, c) then
7 Total:=Total +1;
8 Append($S_{V_j c}$, (V_i, b));
9 endif
10 endfor
11 if Total=0 then M[V_i,b]:= 1; D_{V_i}= D_{V_i} - {b};
12 else Counter[(V_i, V_j),b]:= Total;
13 endfor
14 endfor
15 Initialize List with {(V_i, b) | M(V_i,b)=1};

Step 2: Prunning the inconsistent labels
16 while List not Empty do
17 choose (V_j, c) from List and remove(V_j , c)from List;
18 for (V_i, V_j)$\in$ $S_{V_j c}$ do
19 Counter[(V_i, V_j),b]:=Counter[(V_i, V_j),b]-1;
20 if Counter[(V_i, V_j),b]=0 and M[V_i,b]=0 then
21 Append(List, (V_i, b));
22 M[V_i,b]:= 1; D_{V_i}= D_{V_i} - {b};
23 endif
24 endfor
25 endwhile
end_AC-3

Figura A.5: Procedimento AC-4 [49]

(V_i, b) na lista. Observando o comando *while*, (linhas 16 a 25), verificamos que um valor é eliminado no máximo uma vez, porque a matriz M grava esta informação. Dado que o valor c seja eliminado do nó V_j, os únicos valores que podem ser afetados são aqueles nós V_i que têm uma aresta para V_j. Seja g_{V_j} o grau do vértice do nó V_j. Então, já que V_j pode aparecer no máximo d vezes (linha 17) e já que há no máximo $g_{V_j} d$ elementos de $S_{V_j c}$ para um dado V_j, então a linha 19 pode ser executada no máximo $d^2 e$. O limite inferior da complexidade para AC-4 é $O(ed^2)$.

A.7 Algoritmo AC-6

O algoritmo AC-6 é utilizado para alcançar consistência de arcos em grafos de restrições binárias [9].

Este algoritmo trabalha com uma estrutura de dados (matriz) de booleanos, denominada M, que mantém quais os valores do domínio inicial estão no domínio corrente

($M(V_i, a) = true \Leftrightarrow a \in D_{V_i}$). Nesta matriz cada domínio inicial D_{V_i} é considerado como o intervalo de inteiros entre $1..\mid D_{V_i} \mid$. Mas pode ser um conjunto de valores de qualquer tipo que tenha ordenação total desses valores [9]. São utilizadas algumas funções e procedimentos de tempo constante para tratar os conjuntos do domínio. Estas funções e procedimentos estão descritos a seguir.

1. A função $last(D_{V_i})$ retorna o maior valor no domínio de V_i (D_{V_i}), se $D_{V_i} \neq \emptyset$, senão retorna 0;
2. Se $a \in D_{V_i}$, $next(a, D_{V_i})$ retorna o menor valor em D_{V_i} maior que a;
3. O procedimento $remove(a, D_{V_i})$ remove o valor a de D_{V_i};
4. $S_{V_j b} = \{(V_i, a) \mid (V_j, b)$ é o menor valor em D_{V_j} suportando (V_i, a) para a restrição $R_{V_i V_j}\}$;
5. $Waiting - List$ contém valores eliminados do domínio mas para aqueles que a propagação não tenha sido processada.

No algoritmo AC-4 quando um valor (V_j, b) é removido este é adicionado à $Waiting - List$ para aguardar a propagação em conseqüência de sua eliminação. Em AC-6, a conseqüência da eliminação do elemento conjunto suporte (V_j, b) é encontrar outro suporte para todo $(V_i, a) \in S_{V_j b}$. Como D_{V_j} está ordenado, é verificado um outro $c \in D_{V_j}$, depois de b, que possa suportar (V_i, a) para a restrição $R_{V_i V_j}$. Quando um valor c é encontrado, (V_i, a) é adicionado a $S_{V_j c}$, desde que (V_j, c) seja o menor suporte para (V_i, a) em D_{V_j}. Se tal valor não existir, (V_i, a) é eliminado.

O procedimento $nextsupport$, descrito na Figura A.6, é utilizado para encontrar o menor valor em D_{V_j}, que não seja menor que b e dê suporte para (V_i, a) para a restrição $R_{V_i V_j}$.

O algoritmo AC-6 é apresentado na Figura A.7.

Neste algoritmo, no passo de inicialização é calculado o conjunto suporte $S_{V_j b}$ para todo valor (V_i, a) de cada restrição $R_{V_i V_j}$ para provar que (V_i, a) é viável. Se não houver uma restrição $R_{V_i V_j}$ da qual (V_i, a) não tenha suporte, este é removido do domínio de i e colocado no $Waiting - List$.

No passo de propagação os valores (V_j, b) são retirados de $Waiting - List$ para propagar as conseqüências de sua eliminação. Isto é, encontrar outro suporte (V_j, c) para os valores (V_i, a) que eles estavam dando suporte. Quando tal valor $c \in D_{V_j}$ não é encontrado, (V_i, a) é removido de D_{V_i} e colocado na $Waiting - List$.

A matriz M tem tamanho proporcional ao número de valores em D, $O(nd)$. Os pares arco-valor $[(V_i, V_j), a]$ têm no máximo um suporte (V_j, b) com (V_i, a) pertencendo a $S_{V_j b}$.

```
Procedure nextsupport(in i,j,a: integer; in out b: integer; out emptysupport: boolean)
1 if b≤last(D_Vj) then
2   emptysupport ← false;
3   {search of the smallest value greater (or equal) than b that belongs to D_Vj}
4   while not M(V_j, b) do b ← b + 1;
5   {search of the smallest support for (i,a) in D_j}
6   while not R_ij(a,b) and not emptysupport do
7       if b < last(D_j) then b ← next(b,D_j)
8       else emptysupport ← true;
9       endif
10 else emptysupport ← true;
11 endif
end_nextsupport
```

Figura A.6: Procedimento *nextsupport*

Então, o tamanho total dos conjuntos $S_{V_j b}$ é no máximo igual ao número de pares arco-valor que é $O(ed)$. A complexidade do espaço de busca do AC-6, no pior caso, é igual a $O(ed)$, enquanto que no AC-4 é $O(ed^2)$.

Nos passos de inicialização e propagação, o $loop$ mais interno é uma chamada para o procedimento $nextsupport$ que computa um suporte para um valor sobre uma restrição, começando de um valor corrente. Assim, para cada par arco-valor $[(V_i, V_j), a]$, cada valor em D_{V_j} será verificado no máximo uma vez. Há ed pares arco-valor. Assim $O(ed^2)$ é a complexidade de tempo no pior caso para o AC-6.

Na fase de inicialização das estruturas de dados o conjunto suporte de cada uma das variáveis para cada um de seus valores é inicializado com vazio e a matriz M é preenchida.

Na fase de propagação são executados os pares pertencentes ao conjunto $List$. Desta forma, são eliminados os valores inconsistentes das variáveis que ainda não tiverem um único valor.

A.8 Algoritmo AC-7

AC-7 é um algoritmo de consistência de arcos que utiliza a idéia de suporte bidirecional: (V_i, a) dá suporte a (V_j, b) se e somente se (V_j, b) suporta (V_i, a). Além disso, se (V_i, a) já tiver sido verificado, não é necessário verificar (V_j, b). Ao explorar a bidirecionalidade, o algoritmo AC-7 pode ser implementado com complexidade de espaço menor [10]. Este algoritmo é baseado no algoritmo *AC-Inference*, descrito na Figura A.8.

A estrutura de dados do AC-Inference é uma estrutura híbrida entre AC-4 e AC-6. Para cada variável V_i para cada um dos seus possíveis valores, a, e cada variável V_j, que compartilha uma restrição com V_i, *AC-Inference* mantém:

```
Procedimento AC-6
1 Waiting-List ← Empty-List;
2 for (i,a) ∈ D do S_ia ← { }; M(i,a) ← true;
3 for(i, j) ∈ arcs(G) do
4    for a ∈ D_i do
5        b ← 1; nextsupport(i,j,a,b,emptysupport);
6        if emptysupport
7        then remove(a, D_i); M(i,a) ← false;
8            Add-to(Waiting-List,(i,a) )
9        else Add-to(S_jb(i,a))
10   end-for
11 end-for
12 while Waiting-List ≠ { } do
13     Pick (j,b) from Waiting-List ;
14     for (i,a) ∈ S_jb do
15         Delete (i,a) from S_jb ;
16         if M(i,a) then
17           c ←b; nextsupport(i,j,a,c,emptysupport);
18           if emptysupport
19           then remove(a, D_i); M(i,a) ← false;
20               Add-to(Waiting-List,(i,a) )
21           else Add-to(S_jb(i,a))
22           endif
23         endif
24     endfor
25 endwhile

end_AC-6
```

Figura A.7: Procedimento AC-6

- $CS[V_i, a, V_j]$, um conjunto atualmente suportado, de valores do domínio de V_j atualmente suportados por (V_i, a);
- $S[V_i, a, V_j]$, um conjunto suporte, de valores do domínio de V_j que suporta (V_i, a);
- $U[V_i, a, V_j]$, um conjunto não verificado, de valores do domínio de V_j que não tenham sido verificados para verificar se eles suportam (V_i, a);
- Valores que tenham sido verificados e não suportam (V_i, a) não aparecem em nenhum destes conjuntos.

Manter os conjuntos S e U permite que o esquema grave informações de verificação de restrições inferidas. Resultados positivos são lembrados movendo-os dos conjuntos não verificados para os conjuntos suportes. Resultados negativos são lembrados pela eliminação dos conjuntos não verificados. Como AC-6, o algoritmo AC-7 trabalha através de conjuntos não verificados uma vez e faz verificações sobre um único suporte. Porém, AC-7 evita algumas verificações de valores via inferência, o que não é realizado em AC-6.

É importante destacar que quanto mais breve a bidirecionalidade do suporte for inferida, o conjunto atualmente suportado de cada par valor-variável é incluído no conjunto suporte.

AC-7 refina o algoritmo *AC-Inference*, restringindo as inferências àquelas baseadas na bidirecionalidade. Já que bidirecionalidade é uma propriedade geral de restrições, AC-7 é um algoritmo de consistência de arcos de propósito geral. Restringindo inferências para bidirecionalidade a complexidade do espaço de busca é mantida na mesma ordem de AC-6, $O(ed)$.

AC-7 apresenta as seguintes propriedades desejáveis:

- Nunca verifica restrições $R_{V_iV_j}(a, b)$ se existir b' ainda em D_{V_j} tal que $R_{V_iV_j}(a, b')$ já tenha sido verificada com sucesso.
- Nunca verifica restrições $R_{V_iV_j}(a, b)$ se existir b' ainda em D_{V_j} tal que $R_{V_iV_j}(b', a)$ já tenha sido verificada com sucesso.
- Nunca verifica restrições $R_{V_iV_j}(a, b)$ se esta restrição já tiver sido verificada ou $R_{V_jV_i}(b, a)$ tiver sido verificada.
- Tem complexidade de espaço de busca igual a $O(ed)$.

Em termos de estrutura de dados, AC-7 representa o domínio D_{V_i} de uma variável V_i como uma tabela ordenada de valores booleanos para responder em tempo constante se um valor de um domínio inicial está no domínio atual ou não. Nesta tabela, o domínio inicial D_{V_i} é considerado como um intervalo inteiro $1..\mid D_{V_i} \mid$. Nesta tabela também é armazenado o mais alto valor do domínio D_{V_i} e para cada valor em D_{V_i}, os índices dos valores anteriores e seguintes de D_{V_i}, de forma a utilizar as seguintes funções e procedimentos de tempo constante:

- $highest(D_{V_i})$ retorna o mais alto valor em D_{V_i} se $D_{V_i} \neq \emptyset$.
- $next(a, D_{V_i})$ retorna o menor valor de D_{V_i} maior que a se $a \in D_{V_i}$ $highest(D_{V_i})$; se $a = highest(D_{V_i})$, nil é retornado.
- $remove(a, D_{V_i})$ remove o valor a de D_{V_i}.

Para todo a no domínio de D_{V_i}, $CS[V_i, a, V_j]$ contém todos os valores b em D_{V_j} para que (V_i, a) seja atribuído como seu suporte atual. O suporte atual não é necessariamente o menor.

Para todo a no domínio de D_{V_i}, $last[V_i, a, V_j]$ é atualizado por AC-7 para ser o último valor de D_{V_j} que já tenha sido verificado quando procurando por um suporte para (V_i, a). Então, os vetores $last$ asseguram que todo b em D_{V_j} tal que $R_{V_iV_j}$ já tenha sido verificado

é menor ou igual a $last[V_i, a, V_j]$ e que todo b em D_{V_j} compatível com (V_i, a) é maior ou igual a $last[V_i, a, V_j]$.

O Algoritmo AC-7 é apresentado na Figura A.9.

A função $SeekSupportSet$ tem o mesmo comportamento que em *AC-Inference*. Tratando o $SeekSupportSet$ como uma pilha parece ser uma heurística eficiente porque ela propaga as conseqüentes remoções de valores tão breve quanto elas são realizadas, e descobre domínios vazios mais cedo.

A complexidade de espaço no pior caso do AC-6 é $O(ed)$ devido ao tamanho dos conjuntos $S[V_j, b]$. Em AC-7, os conjuntos $S[V_j, b]$ são divididos em $CS[V_j, b, V_i]$. Isto não aumenta o seu tamanho total: cada par arco-valor $[(V_i, V_j), a]$ tem no máximo um suporte atual (V_j, b). O tamanho total dos conjuntos $CS[V_j, b, V_i]$ é $O(ed)$, mesmo adicionando os arrays $last$, que é $2ed$ em espaço e o tamanho de ***SeekSupportSet***, que é $O(ed)$. Mesmo assim, a complexidade de espaço do AC-7 permanece em $O(ed)$.

Os dois *loops* mais internos de AC-7 estão em ***SeekCurrentSupport***. Já que cada $R_{V_iV_j}$ é verificado no máximo uma vez, no segundo *loop* de ***SeekCurrentSupport***, a complexidade devido às chamadas deste *loop* é $O(ed^2)$. No decorrer do algoritmo, um valor de $a \in D_{V_i}$ é colocado no máximo uma vez em $CS[V_j, b, V_i]$: quando (V_j, b) é atribuído a um conjunto suporte corrente de (V_i, a). ***SeekSupportSet***, no seu primeiro *loop* mais interno, é chamado no máximo d vezes para cada $CS[V_i, a, V_j]$, para um total de trabalho de no máximo d eliminações em cada $CS[V_i, a, V_j]$. Aquele *loop* é O(d) sobre cada $CS[V_i, a, V_j]$ e então a complexidade devido às chamadas a este *loop* tem limite inferior $2ed$x$O(d)$, isto é, $O(ed^2)$. Então a complexidade de tempo AC-7 no pior caso é $O(ed^2)$. Finalmente, surgiu o AC-7, que também é baseado no AC-6.

A.9 Algoritmo AC2001/3.1

O algoritmo AC2001/3.1 [13] é um algoritmo de consistência de arcos de granulosidade grossa com complexidade de tempo ótima no pior caso. Este algoritmo mantém a simplicidade de AC-3, mas obtém melhores resultados em termos de verificação de restrições e tempo de cpu. O procedimento principal de AC-3 é o procedimento $REVISE(V_i, V_j)$ (Figura A.1). AC2001/3.1 busca melhorar a implementação deste procedimento. No procedimento $REVISE(V_i, V_j)$ uma restrição (V_i, V_j) pode ser revisada várias vezes. Para melhorar a eficiência do algoritmo é necessário que se encontre um valor suporte no domínio de V_j para o valor $x \in D_{V_i}$ na primeira revisão do arco (V_i, V_j) e armazene o suporte numa estrutura $Last((V_i, x), V_j)$. Ao verificar a viabilidade de $x \in D_{V_i}$ em revisões subseqüentes do arco (V_i, V_j) é necessário somente verificar se o suporte (b) armazenado em $Last((V_i, x), V_j)$ ainda está em D_{V_j}. Se devido à revisão de outras restrições o valor b

tiver sido removido de D_{V_j} é necessário explorar somente os valores que ocorrem depois de b. Os valores que ocorrem antes de b em D_{V_j} já foram verificados anteriormente.

Assuma, sem perda de generalidade, que os domínios D_{V_i} de todas as variáveis estão associados com uma ordenação total. A função $succ(x, D_{V_j})$ retorna o primeiro valor em D_{V_j} que está depois de x de acordo com a ordenação total ou NIL se não existe elemento. NIL é um valor que não pertence a qualquer domínio mas precede qualquer valor em qualquer domínio.

A Figura A.10 apresenta o procedimento $REVISE(V_i, V_j)$ (do algoritmo AC-3) adaptado para AC2001/3.1, que é denominado $REVISE2001/3.1(V_i, V_j)$. Nesta figura, a linha 1 verifica se o suporte em $Last$ ainda é válido. A linha 2 utiliza a ordenação total do domínio para encontrar o próximo valor suporte.

Para revisar o arco (V_i, V_j) é necessário encontrar um suporte para cada valor de D_{V_i}. Como há d valores em D_{V_i}, no máximo $O(d^2)$ vezes serão necessárias para revisar o arco (V_i, V_j). O número de arcos na rede de restrições é $2e$ (uma restrição corresponde a dois arcos). Então, a complexidade de tempo é $O(ed^2)$.

A complexidade de espaço é limitada pelo tamanho da fila Q e pela estrutura $Last$. Q pode ter complexidade $O(n)$ ou $O(e)$, dependendo da implementação da fila. O tamanho de $Last$ é $O(ed)$, já que cada valor $x \in D_{V_i}$ precisa de um espaço em $Last$ com relação a cada restrição envolvendo V_i. Assim a complexidade de espaço global é $O(ed)$.

A complexidade de tempo do AC2001/3.1 é $O(ed^2)$ e a complexidade de espaço é $O(ed)$.

Resultados experimentais mostram que AC2001/3.1 é competitivo com AC-6 e AC-7 que são algoritmos de granulosidade fina [13] .

Algoritmo AC-Inference
1 Initialize the SeekSupportSet with all the $[(V_i, a), V_j]$*;*
2 Establish the initial S sets and U sets using initial inferences;
3 Initialize the CS *sets with the empty set;*
4 while SeekSupportSet $\neq \emptyset$ *do*
5 pick $[(V_i, a), V_j]$ *from SeekSupportSet;*
6 if $a \in D_{V_i}$ *then*
7 $c \leftarrow$ *SeekCurrentSupport(*V_i, a, Vj*);*
8 if $c \neq nil$ *then*
9 put $a \in CS[V_j, c, V_i]$*;*
10 else ProcessDeletion$(V_i, a, SeekSupportSet)$*;*
11 end-if;
12 end-while;
end_AC-Inference

function SeekCurrentSupport (in V_i*: variable; in* a*: value; in* V_j*:*
variable):value
*/*returns a value* b *supporting* (V_i, a)*, or nil if not found*/;*
*/*updates* S *and* U *if necessary*/;*
1 found $\leftarrow$ *false;*
2 while $(S[V_i, a, V_j] \neq \emptyset)$ *and* $(\neg found)$ *do*
3 $b \leftarrow$ *an element of* $S[V_i, a, V_j]$*;*
4 if $b \in D_{V_j}$ *then found* $\leftarrow$ *true;*
5 else delete b *from* $S[V_i, a, V_j]$*;*
6 while $(U[V_i, a, V_j] \neq \emptyset) and (\neg found)$ *do*
7 pick b *from* $U[V_i, a, V_j]$*;*
8 if $b \in D_{V_j}$ *then*
9 $found \leftarrow R_{V_i,V_j}(a, b)$*;*
10 make inferences from ($(V_i, a)(V_j, b)$*);*
11 if found then return b*;*
12 else return nil;
end_Seek-Current-Support

procedure ProcessDeletion(in Vi: variable; in a:value; in out SeekSupportSet:set)
1 remove a *from* D_{V_i}
2 if $D_{V_i} = \emptyset$ *then exit ("wipe out");*
3 for each $V_j / R_{V_i V_j}$ *exists do*
4 for each $b \in CS[V_i, a, V_j]$*;*
5 delete b *from* $CS[V_i, a, V_j]$*;*
6 put $[(V_j, b), V_i]$ *in SeekSupportSet;*
end_ProcessDeletion

Figura A.8: Algoritmo *AC-Inference*

```
Algoritmo AC-7;
1 SeekSupportSet ← ∅;
2 for each (V_i, V_j) / R_{V_i V_j} exists do
3     for each a ∈ D_{V_i}
4         CS[V_i, a, V_j] ← ∅;
5         last[V_i, a, V_j] ← 0 /* first value de D_{V_j} − 1*/;
6         put[(V_i, a), V_j] in SeekSupportSet;
7     end-for-each;
8 end-for-each;
9 while SeekSupportSet ≠ ∅ do
10 pick [(V_i, a), V_j] from SeekSupportSet;
11 if a ∈ D_{V_i} then
12    c ← SeekCurrentSupport(V_i, a, V_j);
13    if c ≠ nil then
14    put a ∈ CS[V_j, c, V_i];
15    else ProcessDeletion(V_i, a, SeekSupportSet);
16    end-if;
17 end-while;
end_AC-7

function SeekCurrentSupport (in V_i: variable; in a: integer; in V_j:variable):integer
/*returns a value supporting (V_i, a), or nil if not found*/;
/*updates last[V_i, a, V_j] if necessary*/;
begin
1   found ← false;
2   while (S[V_i, a, V_j] ≠ ∅) and (¬found) do
3       b ← an element of S[V_i, a, V_j];
4       if b ∈ D_j then found ← true;
5       else delete b from S[V_i, a, V_j];
6   if found then return b;
7   b ← last[V_i, a, V_j];
8   if b > highest(D_{V_j}) then return nil;
9   while b ∉ D_{V_j} do b ← b + 1;
10  while (b ≠ nil) and (¬found) do
11      if (last[V_j, b, V_i]) ≤ a) and then (R_{V_i V_j}(a, b))
12        found ← true;
13        else b ← next(b, D_{V_j}));
14  last[V_i, a, V_j] ← b;
15  return b;
end_Seek
```

Figura A.9: Procedimento AC-7

```
Procedure REVISE2001/3.1(V_i,V_j);
      DELETE ←false;
      foreach x ∈ D_{V_i} do
          b ← Last((V_i, x), V_j);
1         if b ∉ D_{V_j}
            b ← succ(b,D_{V_j});
2           while (b ≠ NIL) and (not (R_{V_iV_j}(x, b))) do
                b ← succ(b,D_{V_j});
            endwhile;
            if b ≠ NIL then
              Last((V_i, x), V_j) ← b;
            else
              delete x from D_{V_i};
              DELETE ←true;
            endif;
          endif;
      endforeach;
      return DELETE;
end_REVISE2001/3.1
```

Figura A.10: Procedimento $REVISE2001/3.1(V_i, V_j)$ para AC2001/3.1 [13]

Apêndice B

Métodos de Particionamento em Teoria dos Grafos

Este apêndice busca detalhar alguns métodos de particionamento utilizados em teoria dos grafos. A Seção B.1 apresenta os métodos de bisseção intuitivos. São descritos os algoritmos de bisseção coordenada recursiva e de bisseção grafo recursiva. Na Seção B.2 são apresentados os método de bisseção espectral. Na Seção B.3 são apresentados os métodos de quadrisseção e octosseção espectrais.

B.1 Métodos de Bisseção Intuitivos

Em [65] são descritos três algoritmos recursivos de particionamento de conjuntos: bisseção coordenada recursiva, bisseção grafo recursiva e bisseção espectral recursiva. A diferença entre eles é a estratégia de particionamento de um conjunto em subconjuntos.

B.1.1 O Algoritmo de Bisseção Coordenada Recursiva

O Algoritmo de Bisseção Coordenada Recursiva é conceitualmente o algoritmo mais fácil. Porém, considera que existem coordenadas bi ou tri-dimensionais disponíveis para os vértices. Isto é, para cada $v_i \in V$ temos associado uma tupla $v_i = (x_i, y_i)$ ou tripla $v_i = (x_i, y_i, z_i)$, dependendo se o modelo é bi ou tri-dimensional. Esta estratégia simples de bisseção para um conjunto consiste em determinar a direção da coordenada da mais longa expansão do conjunto. Sem perda de generalidade, assuma que esta seja a direção x. Então todos os vértices são ordenados de acordo com a sua coordenada x. Metade dos vértices com coordenadas x é atribuída a um subconjunto e a outra metade é atribuída ao segundo subconjunto.

Resumidamente, o algoritmo de Bisseção Coordenada Recursiva pode ser definido como na Figura B.1:

1- Determine a mais longa expansão do conjunto (direção x, y ou z)
2- Ordene os vértices de acordo com a coordenada na direção selecionada
3- Atribua metade dos vértices a cada subconjunto
4- Repita recursivamente (divisão e conquista)

Figura B.1: Algoritmo de Bisseção Coordenada Recursiva

1- Use o algoritmo SPARSPAK RCM para computar a estrutura de nível
2- Ordene os vértices de acordo com a estrutura de nível RCM
3- Atribua metade dos vértices a cada subconjunto
4- Repita recursivamente (divisão e conquista)

Figura B.2: Algoritmo de Bisseção Grafo Recursiva

Para particionar um grafo em oito subgrafos, por exemplo, utilizando este algoritmo, é só executá-lo recursivamente. Primeiro basta dividir o grafo em dois subgrafos. Em seguida, aplicar o algoritmo a cada subgrafo, obtendo quatro subgrafos e finalmente, aplicá-lo a cada um dos subgrafos e obter oito subgrafos.

Este algoritmo tem a desvantagem de não aproveitar as informações de conectividade fornecidas pelo grafo. Desta forma, podem ser gerados subconjuntos desconexos e até mesmo vértices isolados.

B.1.2 O Algoritmo de Bisseção Grafo Recursiva

O algoritmo de Bissecção Grafo Recursiva utiliza o grafo de distâncias entre vértices, dado por: $d(v_i, v_j)$ = |caminho mais curto entre v_i e v_j|. Desta forma, as informações de conectividade do grafo são utilizadas.

Primeiramente, é construída uma estrutura em nível utilizando o algoritmo SPARSPAK RCM (Reverse Cuthill-Mc-Kee) [65, 30].

Este algoritmo primeiro encontra os dois vértices pseudo-periféricos no grafo. Isto é, os vértices que tenham uma distância muito grande, mas que não é necessariamente o par de vértices com distância máxima. Começando por um dos vértices (o vértice raiz) é construída uma estrutura em nível (*level structure*), que permite organizar os vértices do grafo em conjuntos de distância crescente a partir da raiz.

A estrutura em níveis produzida pelo algoritmo é a base para o algoritmo recursivo de bisseção de grafo. A metade dos vértices, aqueles que estão mais próximos da raiz, são atribuídos a um subconjunto e os demais vértices são atribuídos ao outro subconjunto. Se o grafo inicial é conexo, então, é garantido que no mínimo um dos subconjuntos é conexo, aquele formado pelos vértices mais próximos da raiz.

A Figura B.2 apresenta o algoritmo de forma resumida.

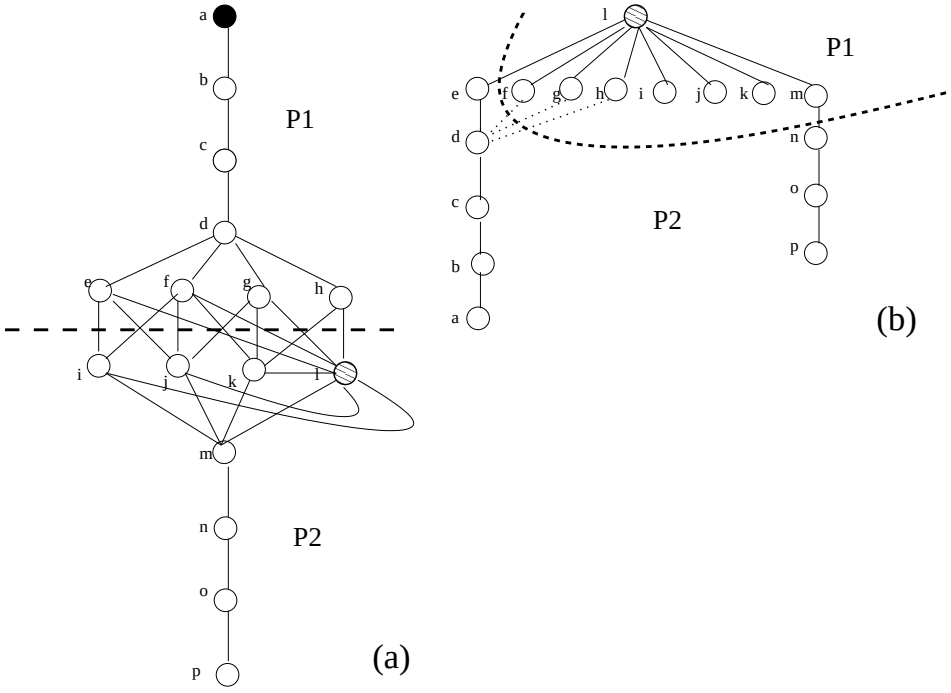


Figura B.3: Exemplo de Bisseção Grafo

A escolha de vértices diametrais não é, necessariamente, a melhor escolha. Na figura B.3, podemos observar um exemplo de grafo em que a escolha de um vértice intermediário é melhor do que de um vértice diametral. Na Figura B.3(a) do exemplo, o algoritmo de Bisseção Grafo Recursiva escolhe o vértice a como raiz da árvore que gera a estrutura em níveis, gerando uma bisseção com 12 arestas entre as partições. Na Figura B.3(b) do exemplo, podemos observar a escolha de um vértice intermediário l e a bisseção realizada é melhor do que aquela quando o vértice escolhido foi um diametral.

B.2 Métodos de Bisseção Espectral

Todo método espectral é baseado na determinação de autovalores e é utilizada a matriz Laplaciana , pois este tipo de matriz possui propriedades importantes [37], como segue:

Seja $u_1, u_2 ... u_n$ autovetores normalizados de L com autovalores $\lambda_1 \leq \lambda_2 \ldots \leq \lambda_n$ correspondentes, a matriz L tem as seguintes propriedades:

1- L é simétrica e semidefinida positiva;

2- Os autovetores u_i são dois a dois ortogonais;

3- $u_1 = n^{-\frac{1}{2}}\mathbf{1}$, $\lambda_1 = 0$

4- Se G é conexo, então λ_1 é o único autovalor nulo de L.

Na literatura, o método de bisseção espectral é apresentado por diferentes autores. As subseções seguintes, descrevem os algoritmos de Simon e de Hendrickson e Leland.

1- Compute o vetor de Fiedler para o grafo usando o algoritmo de Lanczos
2- Ordene os vértices de acordo com o tamanho de entrada no vetor de Fiedler
3- Atribua metade dos vértices a cada subconjunto
4- Repita recursivamente (divisão e conquista)

Figura B.4: Algoritmo de Bisseção Espectral Recursiva

B.2.1 Método de Bisseção Espectral de Simon

Simon [65] define o método de bisseção espectral recursivo como sendo derivado de uma estratégia de bisseção de grafo que é baseada na computação de um autovetor específico da matriz Laplaciana de um grafo G.

A matriz Laplaciana $L(G) = (l_{ij})$, onde $i, j = 1..n$ (n inteiro) é definida da seguinte forma:

$$l_i j = \begin{cases} +1 & \text{se } v_i v_j \in E \\ \text{-grau } (v_i) & \text{se } i = j \\ 0 & \text{caso contrário} \end{cases}$$

onde grau(v_i) representa o grau do vértice v_i.

Podemos notar que $L(G) = -D + A$, onde A é a matriz de adjacência do grafo e D é a matriz diagonal dos graus dos vértices.

A matriz $L(G)$ é semidefinida negativa. Assim, seus autovalores são reais. Segue da definição de L que o maior autovalor λ_1 é zero. Isto é conseqüência da escolha particular dos elementos da diagonal em $L(G)$. Se G é conexo então λ_2, o segundo maior autovalor é negativo. A magnitude de λ_2 é uma medida de conectividade do grafo. O segundo maior autovetor também é conhecido como vetor de Fiedler [25].

O autovetor x_2 associado a λ_2 fornece a informação direcional do grafo. Se as componentes de x_2 são associadas com os vértices correspondentes do grafo, elas produzem um peso para os vértices. As diferenças nestes pesos fornece uma informação de distância sobre os vértices do grafo. Ordenando os vértices de acordo com os seus respectivos pesos gera outra forma de particionar o grafo.

O maior desafio computacional do algoritmo de bisseção espectral recursivo é o cálculo do segundo maior autovetor. Por isso, o algoritmo de Lanczos [54, 55] é usado por Simon, pois não requer qualquer manipulação da matriz Laplaciana $L(G)$. São necessárias apenas multiplicações de matrizes vetor por $L(G)$, que não gera custo de armazenamento adicional. Isto porque a matriz reflete a estrutura do grafo. Numa versão especial do algoritmo de Lanczos somente um autovetor é requerido. Uma função de aproximação racional para computar aproximações de λ_2 é utilizada.

Na Figura B.4 podemos observar o algoritmo de bisseção espectral recursivo.

Na subseção B.2.2 são feitas algumas considerações sobre o algoritmo de Lanczos.

B.2.2 Considerações Sobre o Algoritmo de Lanczos

Em [55], os autores utilizam o algoritmo de Lanczos, que é aplicado para uma matriz definida positiva e produz boas aproximações para os autovalores dos extremos do espectro após poucas iterações.

O algoritmo de Lanczos simples, para uma matriz simétrica A, $n \times n$, computa uma seqüência de vetores de Lanczos, $v_1, v_2, v_3, \ldots$, como mostrado na Figura B.5.

Algoritmo LANCZOS;
1 Escolhe um v_1 *arbitrário,* $\| v_1 \| = 1$*;*
2 $u_1 = Av_1$*;*
3 Para $j = 1, 2, \ldots$ *faça*
$\alpha_j = u_j^T v_j$*;*
$r_j = u_j - \alpha_j v_j$*;*
$\beta_j = \| r_j \|$*;*
$v_j = r_j / \beta_j$*;*
$u_{j+1} = Av_{j+1} - \beta_j v_j$*;*
end-LANCZOS

Figura B.5: Algoritmo de Lanczos

As equações mostradas na Figura B.5 podem ser condensadas na forma de matriz como

$$AV_j - V_j T_j = \beta_j v_{j+1} e_j^T,$$

onde $V_j = (v_1, v_2, ..., v_j)$, $e_j^T = (0, 0, 0, \ldots, 1)$ e

$$T_j = \begin{bmatrix} \alpha_1 & \beta_1 & 0 & \cdot & \cdot \\ \beta_1 & \alpha_2 & \beta_2 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \beta_{j-1} & \alpha_{j-1} & \beta_{j-1} \\ \cdot & \cdot & 0 & \beta_{j-1} & \alpha_j \end{bmatrix}$$

O algoritmo termina se $\beta_j = 0$, e isto ocorre somente para algum $j \leq n$ com aritmética exata. Os autovalores da matriz tridiagonal T_j, também chamados valores de Ritz, são as aproximações de Rayleigh-Ritz dos autovalores de A do subespaço gerado pelos vetores $v_1, v_2, \ldots, v_j$ [55].

Em [53] é feita uma análise do algoritmo de Lanczos, que está detalhada a seguir.

Suponha que tenhamos que computar os n-vetores b, Ab, $A^2b, \cdots, A^{m-1}b$, onde A é uma matriz real simétrica $n \times n$. Lanczos demonstrou, em 1950, como construir uma base melhor para o espaço originado por estes vetores com um pequeno custo extra. A nova base $\{q_1, q_2, \cdots, q_m\}$, chamada de base de Lanczos, tem duas propriedades: (i) é ortogonal, (ii) a representação da projeção de A é uma matriz simétrica tridiagonal T_m.

A propriedade (ii) é sinônima com a recorrência de 3 termos governando os vetores de Lanczos. Entretanto, alguns dos autovalores de T_m, são excelentes aproximações para alguns autovalores de A, mesmo quando $m \ll n$. Estas propriedades implicam em que é mais fácil encontrar o maior dos poucos autovalores de A do que resolver $Ax = b$.

Ao implementar o algoritmo de Lanczos, a propriedade (i) falha completamente para m menor ou igual a 20 ou 30. Conseqüentemente, a relação computada de T_m para A não fica clara. Lanczos propôs manter a aplicação do processo de Gram-Schmidt para cada novo vetor de Lanczos como ele é computado. Todos os $\{q_i\}$ devem ser mantidos em fácil acesso já que em aritmética exata somente os 3 últimos vetores de Lanczos são necessários e os primeiros q's podem ser descartados. O custo aritmético desta reortogonalização total cresce quadraticamente com m. Então, a esperança de computar T_n eficientemente e precisamente pelo algoritmo de Lanczos foi deixada de lado. Em aritmética exata T_n é similar a A e o algoritmo pára.

Durante os cálculos há uma perda de ortogonalidade e o T_m computado retém informação sobre A. Assim, a descoberta dos autovalores de A pelo algoritmo de Lanczos simples é atrasada, mas não previne o alcance da precisão total se passos suficientes são executados. Na aritmética de precisão finita o algoritmo de Lanczos simples irá executar para sempre e tem-se início com modelos que descrevam como T_m relaciona-se a A quando $m \gg n$.

B.2.3 Algoritmo de Bisseção Espectral de Hendrickson e Leland

Outra definição de algoritmo de bisseção espectral que aparece na literatura é apresentada por Bruce Hendrickson e Robert Leland [37].

A métrica utilizada neste trabalho é a *hop* ou *Manhattan* num hipercubo. Nesta métrica o peso de qualquer uma das arestas é 1 [37]. Em máquinas com arquitetura hipercubo, minimizar esta métrica corresponde a minimizar o congestionamento dentro de uma rede de comunicação.

O particionamento de uma tarefa computacional entre os processadores corresponde à atribuição de cada vértice do grafo a um processador. A soma dos pesos de vértices atribuídos a um processador representa a quantidade de esforço computacional do processador. Além disso, a soma dos pesos de todas as arestas conectando vértices atribuídos a dois processadores diferentes representa o total de quantidade de informação que deve ser comunicada entre os dois processadores.

Considere o grafo $G(V, E)$ não orientado e conexo. Atribua uma variável x_i a cada v_i tal que $x_i = \pm 1$ e $\sum_{V_i} x_i = 0$. A primeira condição estipula uma partição de V em dois conjuntos disjuntos. A segunda condição requer que os conjuntos sejam de tamanho igual (assumindo um número par de vértices). O vetor x cujos elementos satisfazem

estas condições é denominado vetor indicador. Isto porque ele indica o conjunto ao qual pertence cada vértice do grafo, em uma bipartição.

A função $f(x) = \frac{1}{4}\sum_{E_{ij}} (x_i - x_j)^2$ conta o número de arestas entre os conjuntos.

A matriz Laplaciana do grafo G $L(G) = L_{ij}$, é uma matriz de ordem n, definida da seguinte forma:

$$l_{ij} = \begin{cases} \text{-1} & \text{se } v_i v_j \in E \\ d_i & \text{se } i = j \\ 0 & \text{caso contrário} \end{cases}$$

onde, d_i corresponde ao grau do vértice i.

Nesta definição podemos notar que $L(G) = D - A$, diferente da definição dada por [65], onde $L(G) = -D + A$.

A função $f\,(x)$ pode também ser definida como $f\,(x) = \frac{1}{4}x^T Lx$. Juntando este fato com as restrições de x, é definido o problema da bisseção, caso discreto, como:

Minimizar $\frac{1}{4}x^T Lx$
Sujeito a: $x^T \mathbf{1} = 0$, $x_i = \pm 1$, onde $\mathbf{1}$ é o n-vetor $(1, 1, \ldots, 1)^T$.

O particionamento de grafos é um problema NP-difícil. Portanto, há a necessidade de relaxar as restrições dos valores discretos sobre o vetor x e definir o problema de bisseção do vetor x com restrições com valores contínuos da seguinte forma.

Minimizar $\frac{1}{4}x^T Lx$
Sujeito a: $x^T \mathbf{1} = 0$, $x^T x = n$
onde os elementos de x podem assumir qualquer valor satisfazendo as restrições.

O problema da bisseção contínuo é uma relaxação do problema discreto e sua solução deve ser mapeada para o caso discreto por algum esquema apropriado para definir uma partição. É claro que o mapeamento no caso discreto não é a melhor solução.

A partir das propriedades da matriz Laplaciana, pode-se provar que $x = \sqrt{n}u_2$ satisfaz as restrições e minimiza $f(x)$, sendo a solução do problema contínuo [65]. Se $\lambda_2 \neq \lambda_3$, esta solução é única ($x = -\sqrt{n}u_2$ define a mesma partição).

O passo seguinte é mapear a solução do problema contínuo para uma partição. Para isto, basta encontrar a mediana dos valores de x_i e mapear os vértices com os valores de x_i correspondentes acima da mediana para um conjunto e aqueles abaixo da mediana para o outro conjunto. Os vértices com valores iguais à mediana são atribuídos aos conjuntos de forma que se mantenha o mesmo número de vértices para cada conjunto.

Na figura B.6 é mostrado o algoritmo de bisseção espectral implementado por Hendrickson e Leland.

Algoritmo BISSECAO(L,n);
1 Calcular o segundo autovetor da matriz L;
2 Calcular o vetor $x = \sqrt{n}u_2$*;*
3 Calcular a mediana m *dos valores de* x*;*
4 Calcular o vetor x'*, onde* $x'_i = -1$*, se* $x_i < m$ *ou* $x'_i = +1$*, se* $x_i > m$*.*
Se houver valores iguais a m*, manter o balanceamento;*
5 Calcular o valor de $f(x') = \frac{1}{4}\sum_{e_{ij}} \left(x'_i - x'_j\right)^2$
6 Balancear as partições com o algoritmo de KL
end-BISSECAO

Figura B.6: Algoritmo de Bisseção Espectral de Hendrickson e Leland

$$\text{L(G)} = \begin{bmatrix} 2 & -1 & 0 & -1 \\ -1 & 3 & -1 & -1 \\ 0 & -1 & 1 & 0 \\ -1 & -1 & 0 & 2 \end{bmatrix}$$

Figura B.7: Matriz Laplaciana de um grafo G com quatro vértices

Os parâmetros de entrada do algoritmo BISSECAO são a matriz Laplaciana L e n que corresponde ao número de vértices do grafo, que também é a ordem da matriz. Se o grafo dado tiver um número ímpar de vértices é necessário acrescentar mais um vértice para termos o mesmo número de vértices nos dois subconjuntos após realizarmos a bisseção.

Na Figura B.14 é apresentado um grafo G com quatro vértices. Sua matriz Laplaciana é representada pela Figura B.7:

Sejam v um autovetor e λ um autovalor da matriz L. Pela definição de autovetor, podemos afirmar que $Lv = \lambda v$. Considere $v = Iv$, onde I é a matriz identidade. Portanto,

$$Lv - \lambda v = 0$$
$$(L - \lambda I)v = 0$$

Neste caso o autovetor v não pode ser nulo.

O polinômio característico $P(\lambda) = det(L - \lambda I) = 0$. Ao resolvermos este determinante para o exemplo dado encontramos uma equação em λ:

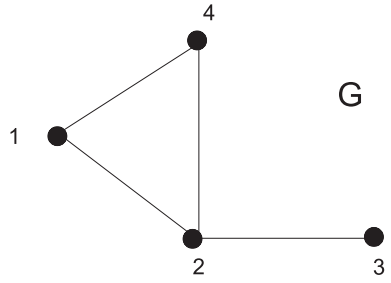


Figura B.8: Exemplo de um grafo G com 4 vértices

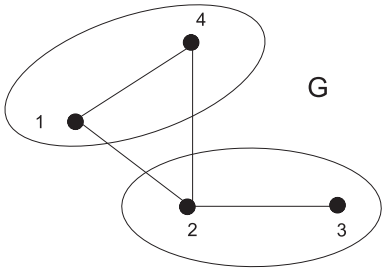


Figura B.9: Bisseção do grafo G

$$\lambda^4 - 8\lambda^3 + 19\lambda^2 - 12\lambda = 0.$$

A solução desta equação nos fornece 4 autovalores: $\lambda_1 = 0$, $\lambda_2 = 1$, $\lambda_3 = 3$ e $\lambda_4 = 4$.

Estamos interessados no segundo autovalor, que é $\lambda_2 = 1$. Queremos encontrar o autovetor correspondente para este autovalor.

Ao resolvermos a igualdade $Lv = \lambda v$, encontramos uma família de autovetores, $u_2 = (w, 0, -2w, w)$.

Se considerarmos $w = 1$, então, $u_2 = (1, 0, -2, 1)$ é um segundo autovetor associado ao autovalor $\lambda_2 = 1$.

O vetor indicador x é dado por $x = \sqrt{n}u_2$, onde $n = 4$ é o número de vértices do grafo. Assim, $x = (2, 0, -4, 2)$.

Para passarmos a solução do problema contínuo para uma partição discreta é necessário calcular a mediana e escrever este vetor em função de $1´$s e $-1´$s.

Uma maneira utilizada para calcular a mediana consiste em ordenar os elementos do vetor x em ordem crescente $(-4, 0, 2, 2)$. Depois, a partir dos dois valores centrais, tirar a média (que é 1). Em seguida, basta escrever o vetor x', de forma que os elementos maiores que 1 tenham o valor 1 em x' e aqueles menores que 1 tenham valor -1. Para valores iguais a mediana é necessário fazer uma análise para decidir em que conjunto estes vértices serão atribuídos de modo que os conjuntos fiquem balanceados. Neste exemplo, $x' = (1, -1, -1, 1)$. Portanto, os vértices v_1 e v_4 estão num conjunto e os demais estão em outro conjunto.

A função $f(x) = \frac{1}{4}\sum_{e_{ij}}(x_i - x_j)^2 = 2$ calcula o número de arestas que existem entre os conjuntos.

Percebemos, então, que existem duas arestas entre os conjuntos na partição gerada, como mostra a Figura B.9.

Como são heurísticas, os métodos de bisseção espectral muitas vezes não produzem partições que sejam localmente boas. Desta forma, estes métodos podem ser melhorados associando a eles heurísticas para melhorar a partição localmente. Uma destas heurísticas é o algoritmo de *Kernighan-Lin* (KL) [37].

O algoritmo de *K*ernighan-Lin é considerado bom para encontrar respostas localmente ótimas, devendo ser inicializado com uma boa partição global. Este algoritmo é mostrado na Figura B.10, considerando que $P(V_i)$ é a partição formada pelo conjunto de vértices V_i.

Algoritmo KL(partição global);
  Repita
    Melhor Partição $\leftarrow$ *Partição Atual;*
    $S_1 \leftarrow$ *Vertices da Partição 1;*
    $S_2 \leftarrow$ *Vertices da Partição 2;*
    Para todo V_i
      Computar o valor de preferência $q(i)$*;*
    Enquanto $S_1 \neq \emptyset$ *e* $S_2 \neq \emptyset$
      $V_i \leftarrow$ *vértice em* S_1 *com maior preferência;*
      $P(V_i) \leftarrow 2$*;*
      $S_1 \leftarrow S_1 \setminus V_i$*;*
      Atualiza preferência dos vizinhos de V_i*;*
      $V_j \leftarrow$ *vértice em* S_2 *com maior preferência;*
      $P(V_j) \leftarrow 1$*;*
      $S_2 \leftarrow S_2 \setminus V_j$*;*
      Atualiza preferências dos vizinhos de V_j*;*
      Se Partição Atual for melhor que Melhor Partição
      Então Melhor Partição $\leftarrow$ *Partição Atual;*
    fim-enquanto;
    Partição Atual $\leftarrow$ *Melhor Partição;*
  Até não haver partição a ser descoberta
fim-Algoritmo KL

Figura B.10: Algoritmo de Kernighan-Lin

No algoritmo KL, o *loop* mais externo itera até que partições melhores sejam descobertas. Este *loop* começa computando um valor de preferência q para cada vértice. O valor de preferência é a redução do número de arestas entre os conjuntos se o vértice estiver ligando os conjuntos, ou seja:

$$q(V_i) = \sum_{(V_i,V_j)\in E} \begin{cases} +1 & \text{se } P(V_j) \neq P(V_j) \\ -1 & \text{caso contrário} \end{cases}$$

onde $P(V_i)$ é a partição atual para o vértice V_i.

O algoritmo entra num *loop* em que dois vértices são trocados entre partições e atualiza os valores de preferência de seus vizinhos. Ele sempre troca os vértices com as preferências maiores e uma vez que um vértice é movido, este não é reconsiderado. Depois de cada troca a partição resultante é avaliada e a melhor encontrada é gravada e torna-se a nova partição, mesmo saindo do *loop* mais interno. Já que os vértices são trocados, se uma partição inicial está balanceada então todas as partições geradas estarão balanceadas também.

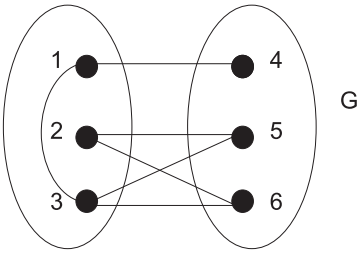


Figura B.11: Grafo G com 6 vértices

Para ilustrarmos o comportamento deste algoritmo, considere o grafo G apresentado na Figura B.11, onde os vértices 1, 2 e 3 pertencem à partição P_1 do grafo e os vértices 4, 5 e 6 pertencem à partição P_2.

Sejam S_1 e S_2 os conjuntos de vértices, que ainda não foram trocados entre as partições 1 e 2, respectivamente: $S_1 = \{1, 2, 3\}$ e $S_2 = \{4, 5, 6\}$.

Para executarmos o algoritmo de KL, são necessários os seguintes passos:

1) Computar o valor de preferência q para cada vértice. O valor de preferência é dado pelo número de arestas entre os 2 conjuntos, como mostrado na Tabela B.1.

Val. de pref.	1^a iteração	2^a iteração	3^a iteração
$q(1)$	0	-2	0
$q(2)$	2	-	-
$q(3)$	1	-	-
$q(4)$	1	-	-
$q(5)$	2	0	-2
$q(6)$	2	0	-2

Tabela B.1: Tabela de valores de preferência

2) Trocar 2 vértices entre partições e atualizar valores de preferência dos vizinhos. Sempre trocar vértices com maior preferência e uma vez que o vértice é movido, este não é reconsiderado. Após a troca, a partição resultante é avaliada e se for melhor do que a anterior, esta é gravada e torna-se a nova partição.

Na primeira iteração do algoritmo foram trocados os vértices 2 e 4, resultando no grafo da Figura B.12, com $S_1 = \{1, 3\}$ e $S_2 = \{5, 6\}$.

Nesta nova partição existem apenas duas arestas entre as duas partições. Portanto, esta partição é gravada e tenta-se uma nova troca para conseguir uma partição melhor a partir desta. Trocando os vértices 3 e 5, resulta em duas partições com três arestas entre elas. Portanto, a partição anterior é melhor e permanece. Resulta em $S_1 = \{1\}$ e $S_2 = \{6\}$. Ao trocarmos 1 e 6 o número de arestas entre as partições é maior, então, da mesma forma, permanece a partição anterior. Não há mais trocas possíveis. Assim, a melhor partição

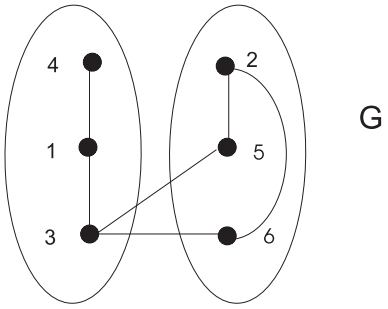


Figura B.12: Grafo G após primeira iteração

com o menor número de arestas entre as partições é a que produziu dois conjuntos com duas arestas entre esses conjuntos.

Este algoritmo requer um número de passos pequeno, geralmente menor do que 5, sendo executado em tempo linear. O espaço requerido é proporcional ao número de vértices do grafo [37].

O algoritmo de KL pode ser generalizado para um número arbitrário de conjuntos. Desta forma, pode ser aplicado para quadrisseção e octosseção espectral, que estão descritos na Seção B.3. Em [37] uma generalização do algoritmo de Kernighan-Lin(GKL) é apresentada. Este algoritmo apresenta as seguintes diferenças em relação ao algoritmo KL:

1- Ao invés de um único valor, as preferências precisam ser computadas para transferir vértices para qualquer outro conjunto, os valores são denotados por $q^k(i)$.

2- Os valores de preferência incluem um fator para a métrica entre conjuntos, que é escolhida a métrica *hop*.

3- Movimentos são feitos de maneira única e não em pares.

4- As partições geradas podem ser desbalanceadas, então, a seleção do movimento tenta encorajar a geração de conjuntos balanceados.

5- Já que partições intermediárias podem ser desbalanceadas, as partições são somente consideradas se elas satisfazem a um critério de balanceamento.

O algoritmo GKL é mostrado na Figura B.13.

Para b conjuntos, este algoritmo pode ser implementado para executar em tempo $O(b^2|E|)$ e requer espaço $O(b|V|)$.

B.2.4 Comparação entre os Métodos de Bisseção

Comparando os três algoritmos de bisseção recursiva implementados por [65] podemos observar que o método de bisseção coordenada recursiva cria conjuntos longos, limitados e desconexos. O algoritmo de bisseção grafo recursivo produz conjuntos mais compactos. Porém, seus limites são *fuzzy* e algumas vezes eles podem ser desconexos. O algoritmo

Algoritmo GKL;
Repita
Melhor Partição $\leftarrow$ *Partição Atual;*
Para todo $k \in \{1, ..., n\}$, $S_k \leftarrow$ *Vértices na partição* k
Para todo V_i
Computar $n-1$ *preferências* $q^k(i)$*;*
Enquanto existe movimentos permitidos
$V_i \leftarrow$ *vértice com maior preferência;*
$l \leftarrow$ *conjunto de vértices* V_i *que querem ir;*
$S_P(V_i) \leftarrow S_P(V_i) \setminus V_i$*;*
$P(V_i) \leftarrow l$*;*
Atualiza preferências dos vizinhos de V_i*;*
Se Partição Atual for a melhor e for balanceada
Então Melhor Partição $\leftarrow$ *Partição Atual;*
fim-enquanto;
Partição Atual $\leftarrow$ *Melhor Partição;*
Até não haver partição a ser descoberta
fim-Algoritmo KL

Figura B.13: Algoritmo generalizado de Kernighan-Lin

de bisseção espectral recursivo cria conjuntos balanceados e conexos, que produzem um particionamento visualmente mais agradável, porém não necessariamente melhor. Entre os três algoritmos recursivos, o algoritmo de bisseção espectral é o que apresenta o menor número de arestas entre os subconjuntos.

Os algoritmos de bisseção apresentam alguns problemas. Por exemplo, eles não aceitam um corte inicial menos atrativo, que produziria, mais tarde, redes com cortes melhores. Ou seja, estes algoritmos não possuem *look ahead*. Em bisseção, a tarefa de dividir o grafo em conjuntos de vértices (decomposição do problema) é separado da atribuição de vértices para um processador específico (problema de atribuição). O *overhead* de comunicação em um programa depende da decomposição e da atribuição. Conseqüentemente, é preferível considerar estes aspectos do problema juntos. Por exemplo, poderíamos escolher dois conjuntos com maior volume de comunicação entre eles para colocá-los topologicamente juntos numa arquitetura [38].

Utilizando o método de bisseção recursiva, quanto maior o número de partes que queremos obter de um grafo, maior é o número de autovetores computados. Na Figura B.14, podemos observar o particionamento de um grafo G em 8 subgrafos, através de bisseção recursiva. No primeiro nível, o grafo é dividido em 2 subgrafos (G_1 e G_2), sendo computados 2 autovetores. No segundo nível, G_1 e G_2 são subdivididos, utilizando o algoritmo de bisseção recursivo, gerando $G_3,\ G_4,\ G_5,\ G_6$, respectivamente. Desta forma, são computados mais 4 autovetores, neste nível. No terceiro nível os 4 subgrafos são subdivididos, gerando 8 partições. Neste nível, são computados mais 8 autovetores. Para particionar o grafo G em 8 partes foi necessário computar um total de 14 autovetores.

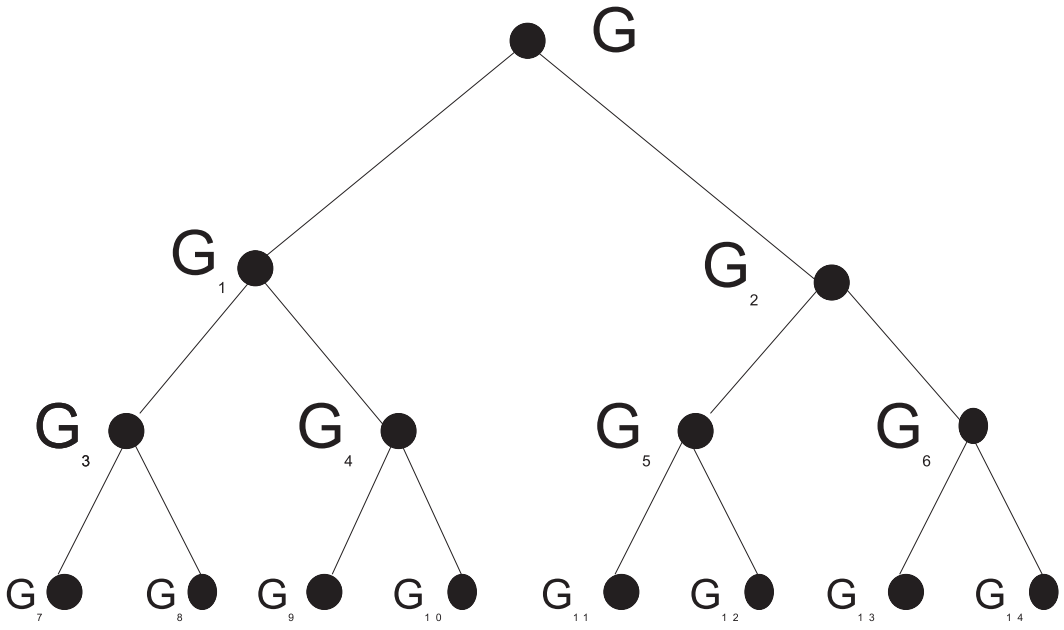


Figura B.14: Partição do grafo G em 8 subgrafos

Em [66] é feita uma análise entre os métodos de bisseção recursiva e um algoritmo de bisseção ótimo.

O método de bisseção recursivo (RB), mesmo quando assumido num algoritmo de bisseção ótimo, pode produzir uma partição *p-way* que é muito distante do ótimo [66]. Este resultado negativo é complementado por dois outros positivos. Primeiro, é mostrado que para algumas classes importantes de grafos que ocorrem em aplicações práticas, tais como elementos finitos com elementos de contorno bem definidos e malhas de diferenças finitas, o RB está quase sempre dentro de um fator constante de um ótimo. Segundo, é mostrado que se a condição de balanço é relaxada tal que cada bloco na partição *p-way* é limitada por $2n/p$, onde n é o número de vértices do grafo, então, um RB modificado encontra uma partição *p-way* aproximadamente balanceada cujo custo está dentro de um fator $O\left(log\ p\right)$ do custo da partição *p-way* ótima [66].

Uma partição *p-way* de um grafo G é uma divisão de seu conjunto de vértices em *p* subconjuntos disjuntos de tamanho n/p, onde n é o número de vértices em G (n é múltiplo de p). O custo de uma partição *p-way* é o número de arestas cujos extremos estão em subconjuntos diferentes.

B.3 Métodos de Quadrisseção e Octosseção Espectral

Na tentativa de minimizar a quantidade de autovetores calculados, surgiram os métodos de quadrisseção e octosseção espectral. Em [65] são apresentados estes métodos de quadrisseção e octosseção espectral. Através destes dois algoritmos a computação pode

ser dividida em 4 ou 8 partes de uma vez, ao invés de usar a bisseção espectral de maneira recursiva.

Estes métodos utilizam múltiplos autovetores de forma a dividir um problema em 4 ou 8 partes de uma vez só. O primeiro autovetor define uma superfície que faz a bisseção do grafo, o segundo define uma superfície que faz bisseção destas duas partes e assim por diante. Isto permite realizar poucos passos recursivos, enquanto se está dividindo um problema num dado número de partes. Desta forma, o problema de *look ahead* apresentado na bisseção é reduzido. A utilização de múltiplos autovetores provoca a diminuição do custo computacional causado pelos cálculos de autovetores. Estes métodos consideram a arquitetura da máquina, minimizando uma função que considera o custo de comunicação numa topologia hipercúbica [65].

Um multiprocessador hipercúbico com d dimensões é composto por um conjunto de 2^d processadores. Os processadores são identificados por números binários distintos de 0 a $2^d - 1$. A informação é transmitida entre eles passando mensagens através de uma rede em que os fios conectam os processadores cujos valores binários diferem de um único bit. No trabalho de [38] foi considerado que não há memória global e que os fios podem transmitir dados em ambos os sentidos simultaneamente. A Figura B.15 ilustra um multiprocessador hipercúbico com 3 dimensões.

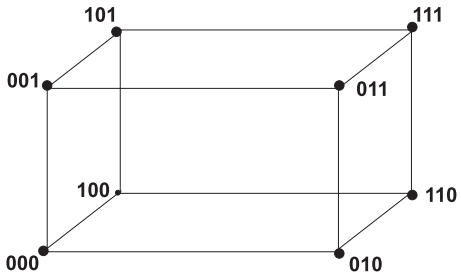


Figura B.15: Hipercubo com 3 dimensões

Os multiprocessadores hipercúbicos possuem uma rede regular, onde cada processador está conectado a d fios de comunicação. Uma mensagem transmitida entre dois processadores usa o fio do bit em que as representações binárias dos dois processadores diferem.

B.3.1 Quadrisseção Espectral

Este método é baseado na bisseção espectral. Porém, é utilizado para realizar a partição do grafo em 4 subgrafos. Para isso, é necessário um segundo vetor indicador, y, que associa 2 coordenadas de dois estados a cada vértice. Para tornar a atribuição explícita, foi definido um mapeamento do vetor indicador para dígitos binários $\hat{x}_i = \frac{1}{2}(x_i - 1)$, $\hat{y}_i = \frac{1}{2}(y_i - 1)$, atribuindo v_i ao conjunto $0, 1, 2$ ou 3, interpretando $\hat{x}_i\hat{y}_i$ como um número binário.

A função objetivo $f(x,y) = \frac{1}{4}(x^T Ly)$ conta as arestas cruzando de uma partição para outra no hipercubo.

No problema de bisseção contínua a restrição $x^T\mathbf{1} = 0$ assegura o balanceamento de carga no vetor indicador x. Na quadrisseção é necessário acrescentar o vetor $y^T\mathbf{1} = 0$ para assegurar o balanceamento no vetor indicador y. A restrição $\sum_{V_i} x_i y_i = x^T y = 0$ garante o balanceamento entre os quatro conjuntos.

Ao passar o problema do modelo discreto para o contínuo, a quadrisseção é definida da seguinte forma:

Minimizar $\frac{1}{4}(x^T Lx + y^T Ly)$
Sujeito a: $x^T\mathbf{1} = y^T\mathbf{1} = x^T y = 0$, $x^T x = y^T y = n$.

Por extensão do argumento algébrico usado na bisseção, o valor mínimo possível de $f(x,y) = n(\lambda_2 + \lambda_3)/4$ é obtido através de $x = \sqrt{n}u_2$ e $y = \sqrt{n}u_3$.

Se $x = \sqrt{n}u_2 cos\theta + \sqrt{n}u_3 sen\theta$ e $y = \sqrt{n}u_2 sen\theta + \sqrt{n}u_3 cos\theta$, x e y continuam satisfazendo as equações de restrições e produzem o mesmo valor mínimo de $f(x,y)$. Então há uma família de soluções, correspondendo a várias escolhas para o parâmetro livre θ. Diferentes valores de θ correspondem a diferentes soluções para o problema contínuo que tem o mesmo valor, mas especifica partições diferentes. O parâmetro θ é escolhido de forma que haja menor perda de precisão quando o problema é relaxado do modelo discreto para o contínuo. Para mapear a solução novamente para o discreto, é necessário, primeiro, definir uma função de distância de um ponto contínuo (x_i, y_i) para um ponto discreto $(\pm 1, \pm 1)$. Em seguida, basta atribuir um quarto dos pontos contínuos para os pontos discretos de tal forma que a soma das distâncias dos valores contínuos de suas atribuições no modelo discreto será minimizada. Esta é uma instância do problema de atribuição de custo mínimo, para o qual algoritmos eficientes são conhecidos. Em [65] foi implementado o algoritmo apresentado em [69] que executa em tempo de execução $O(n\ log(n))$.

B.3.2 Octosseção Espectral

O algoritmo de octosseção espectral segue do algoritmo de quadrisseção. A diferença entre eles é que um terceiro vetor indicador z é definido e mapeado para um *bit* $\hat{z}_i = \frac{1}{2}(z_i - 1)$ e atribui cada v_i a um dos oito conjuntos pela interpretação de $\hat{x}_i, \hat{y}_i, \hat{z}_i$ como um número binário. A função $f(x,y) = \frac{1}{4}(x^T Lx + y^T Ly + z^T Lz)$ conta as arestas que cruzam entre partições num hipercubo de 3 dimensões. Para garantir o balanceamento é necessário adicionar uma restrição: $\sum_{V_i} x_i y_i z_i = 0$. Ao relaxar o problema de octosseção do modelo discreto para o modelo contínuo, o problema se apresenta da seguinte forma:

Minimizar $\frac{1}{4}(x^T Lx + y^T Ly + z^T Lz)$
Sujeito a: $x^T \mathbf{1} = y^T \mathbf{1} = z^T \mathbf{1} = x^T z = y^T z = \sum_{V_i} x_i y_i z_i = 0$,
$x^T x = y^T y = z^T z = n$.

Uma solução para o problem é $x = \sqrt{n}u_2$, $y = \sqrt{n}u_3$ e $z = \sqrt{n}u_4$ com o mínimo correspondente de $n(\lambda_2 + \lambda_3 + \lambda_4)/4$. Porém há redundância no espaço de solução já que qualquer rotação desses autovetores gera outra solução de valor igual. Como estamos trabalhando no espaço R^3 há três graus de liberdade de rotação. A solução é mapeada de volta para o modelo discreto utilizando o mesmo algoritmo da quadrisseção espectral.

Para problemas de particionamento maiores que $d = 4$ dimensões existe um número grande de graus de liberdade de rotação, sendo, geralmente, impossível construir uma partição balanceada de $d + 1$ menores autovalores de L.

Apêndice C

Algoritmos de Satisfação de Restrições Distribuídos

Um CSP distribuído (DisCSP) é um CSP onde variáveis e restrições são distribuídas entre múltiplos agentes [73, 46]. Cada agente atribui valores às suas variáveis tentando gerar consistência local de forma que seja consistente com todas as restrições. Entre os agentes há restrições. O valor atribuído à variável deve satisfazer estas restrições. A solução de um CSP distribuído envolve atribuições de todos os agentes para todas suas variáveis e trocas de informação entre todos os agentes, para verificar a consistência de atribuições com restrições entre agentes. Um CSP distribuído pode ser visto como um modelo elegante para muitos problemas combinatoriais que já são distribuídos por natureza [46].

Formalmente, um CSP distribuído baseado em variável [12] é definido por uma 5-tupla (V, D, R, A, ϕ), onde:

$V = V_1, ..., V_n$ é um conjunto de n variáveis;

$D = D_{V_1}, ..., D_{V_n}$ é uma coleção de domínios finitos;

R é um conjunto de restrições entre variáveis. Uma restrição especifica a relação de combinações permitidas de valores das variáveis envolvidas na restrição;

$A = 1, ..., p$ é um conjunto de agentes de p agentes;

$\phi : V \rightarrow A$ é uma função que mapeia cada variável em seu agente.

O modelo de comunicação utilizado para resolver um CSP distribuído consiste em agentes que se comunicam pelo envio de mensagens. Um agente pode enviar mensagem para outro agente se, e somente se, ele conhece seu endereço na rede. O atraso no envio de uma mensagem é finito. As mensagens recebidas numa comunicação entre dois agentes são recebidas na ordem que elas foram enviadas.

Segundo Luo *et al.* [42], os algoritmos para resolver DisCSPs são formulados de acordo com três elementos principais:

1. Estrutura de controle: centralizada ou descentralizada.

   O uso de controle centralizado com múltiplos processos permite um nível maior de controle, pois o conhecimento global está disponível para os processos que estão resolvendo o problema (agentes). O conhecimento global pode ser algo como uma ordenação de agente que dá aos agentes uma posição relativa dentro de uma ordem global.

   Quando o controle é descentralizado esta ordem global não está disponível, então o nível de controle é mais baixo. Se um conflito surge durante a busca, então um mecanismo de controle que não exija conhecimento global deve ser usado. Isto pode ser uma forma de negociação que tenta resolver o conflito com somente conhecimento local.

2. Tipos de espaço de busca: compartilhada ou separada.

   Se o espaço de busca é compartilhado por diferentes processadores então cada processador pode ser responsável por uma ou mais variáveis e tentará atribuir valores que não conflitem com as variáveis dos outros processadores. Decisões sobre um processador são afetadas pelo estado corrente de outros processadores e então a comunicação é essencial neste caso.

   Num espaço de busca separado, cada processador é responsável por uma área separada do espaço de busca (cada um tem um ramo único da árvore de busca para percorrer). Neste caso, os agentes podem processar de forma autônoma e precisam se comunicar somente quando uma solução é encontrada ou o espaço de busca já foi explorado exaustivamente. Assim, a necessidade de comunicação é pequena.

   O tipo de espaço de busca é dependente das características do problema. Problemas que são naturalmente ou logicamente distribuídos podem ser facilmente mapeados a uma plataforma multiprocessada usando um espaço de busca compartilhado, pois há pouca necessidade de comunicação entre os agentes. Usar este método para um problema que possui alta interconexão entre os agentes gera muita comunicação.

3. Comunicação: passagem de mensagem ou memória compartilhada.

   O método de comunicação pode ser de passagem de mensagem, memória global compartilhada ou uma combinação dos dois. O custo de comunicação é dependente do tipo de problema, do método de controle usado e do tipo de espaço de busca escolhido. Para controle centralizado o custo de comunicação é mais baixo pois o uso do dado global compartilhado para passar a informação entre agentes é mais baixo. Também, quando o espaço de busca é dividido em sub-espaços de busca desconectados pode não haver necessidade de comunicação.

Existem vários algoritmos para resolver CSPs distribuídos. Segundo Yokoo *et al.* [72] os métodos para resolver CSPs podem ser divididos em 2 grupos: algoritmos de busca (por exemplo, algoritmos de *backtracking*) e algoritmos de consistência. Os algoritmos de consistência são procedimentos de pré-processamento que são invocados antes da busca. Os algoritmos de consistência podem ser aplicados a CSPs distribuídos. Este texto concentra-se em algoritmos de busca.

O algoritmo *Backtracking* padrão [22] possui duas fases. Na primeira fase as variáveis são selecionadas em seqüência e uma solução parcial é estendida pela atribuição de um valor consistente, caso exista, para a próxima variável. Na segunda fase, quando não existe solução consistente para a variável corrente, o algoritmo retorna para a variável atribuída anteriormente.

Backtracking freqüentemente sofre de *trashing* [22], ou seja, repetidamente redescobrir as mesmas inconsistências e sucessos parciais durante a busca. Como resolver este problema é NP-completo, a melhoria do algoritmo pode ser alcançada através de algoritmos de pré-processamento que reduzem o tamanho do espaço de busca e dinamicamente melhoram a estratégia de controle do algoritmo durante a busca. Os procedimentos que melhoram dinamicamente o poder de poda do *backtracking* durante a busca são os esquemas *look-ahead* e *look-back*.

Os esquemas *look-ahead* podem ser invocados quando o algoritmo está preparando para atribuir um valor à próxima variável. O objetivo é tentar descobrir, através de uma inferência restrita (propagação de restrição), como as decisões correntes sobre a seleção de uma variável e valor restringirão buscas futuras. Após a propagação de restrição futura, o algoritmo pode usar a informação para (i) decidir qual a próxima variável a ser instanciada, se a ordem não for pré-determinada e (ii) decidir que valor atribuir à próxima variável.

Geralmente, instanciar primeiro aquelas variáveis que maximalmente restringem o resto do espaço de busca é vantajoso. Mas a variável mais restringida tem um número menor de valores viáveis e por isso é usualmente selecionada.

Quando a busca é por uma única solução, uma tentativa é feita para atribuir o valor que maximize o número de opções disponíveis para atribuições futuras.

Os esquemas *look-back* são invocados quando o algoritmo prepara para realizar *backtrack* após encontrar um *dead-end*. Estes esquemas realizam duas funções. A primeira função é decidir quão profundo voltar. Analisando as razões para o *dead-end*, pontos de *backtrack* irrelevante podem freqüentemente ser evitados tal que os algoritmos dirijam-se diretamente para a fonte de falha. Este procedimento é conhecido como *backjumping*. A segunda função é gravar as razões para o *dead-end* na forma de novas restrições, para que o mesmo conflito não apareça novamente na busca. Os termos usados para descrever esta

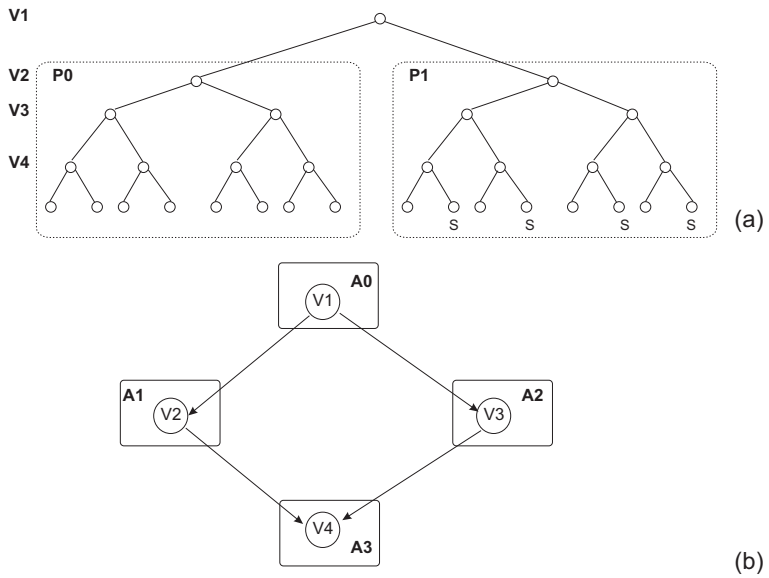


Figura C.1: Árvores de busca: (a) busca paralela; (b) busca distribuída [33]

função são *constraint recording and learning*.

A busca *backtracking* pode ser classificada em busca paralela e busca distribuída [33]. A busca *backtracking* paralela é usada para acelerar o processo de resolução. A busca *backtracking* distribuída é usada em situações onde o problema todo não está totalmente acessível. A resolução é forçada pela colaboração entre sub-problemas.

Ambos os métodos utilizam diversas unidades de processamento. Na busca paralela n processadores executam *backtracking* concorrentemente sobre partes disjuntas da árvore de busca. Na busca distribuída subproblemas distintos são espalhados entre diversas unidades processantes e o *backtracking* é feito de forma colaborativa.

Para ilustrar melhor, a Figura C.1 apresenta as árvores de busca paralela e distribuída. Na Figura C.1 (a) é apresentado um exemplo de exploração do paralelismo. O problema é duplicado em dois processadores $P0$ e $P1$. $P0$ se encarrega do sub-espaço caracterizado por $V1 = a$ e $P1$ explora o restante do espaço. Durante a computação a passagem de mensagem não é usada. Entretanto, um processador pode terminar sua tarefa antes do outro, por isso o balanceamento de carga deve ser realizado. Geralmente uma unidade ociosa solicita a uma ocupada parte da tarefa para executar. A Figura C.1 (b) apresenta a distribuição de um problema com 4 variáveis entre quatro agentes. A exploração do espaço de busca usa resolução local para cada subproblema com negociação sobre restrições compartilhadas.

O algoritmo mais simples para resolver um CSP distribuído consiste em selecionar um agente líder entre todos os agentes e obter toda informação sobre as variáveis, seus domínios e suas restrições, concentrando no agente líder. Assim, o líder resolve o CSP sozinho utilizando algoritmos de satisfação de restrições centralizados. Porém, o custo de

coletar toda informação sobre um problema pode ser muito alto. Em alguns problemas, ter somente um agente para coletar toda informação é indesejável ou impossível por razões de segurança/privacidade.

Muitos problemas do mundo real exigem que haja privacidade das restrições contidas nos agentes. Assim, a consistência deve ser realizada por todos os agentes que possuírem parte de uma restrição [79].

A privacidade pode ser definida para valores atribuídos ou para restrições [46].

A privacidade de valores atribuídos ocorre quando os agentes não divulgam suas atribuições. Neste caso, os agentes querem esconder a existência ou a inexistência de alguns valores de suas variáveis [64].

A privacidade de restrições ocorre quando os agentes mantêm parte de suas restrições entre eles e um agente restritivo privado. Os agentes querem esconder sua concordância ou discordância com certos conjuntos de atribuições [64].

Baseado no critério de considerar privacidade na implementação, as seções seguintes apresentam os principais algoritmos existentes para resolver CSPs distribuídos.

C.1 Algoritmos que não consideram privacidade

C.1.1 *Synchronous Backtracking*

O algoritmo de busca *backtracking* padrão para resolver CSP pode ser modificado para produzir o algoritmo *Synchronous Backtracking (SBT)* [72, 76] para CSPs distribuídos. Considere que agentes assumam uma ordem de instanciação para suas variáveis. Cada agente recebe uma solução parcial (*Current Partial Assignment - CPA*) dos agentes anteriores e instancia sua variável baseado nas restrições que ele conhece. Se for encontrado um valor que seja consistente com as restrições, o agente adiciona este valor à solução parcial e passa a solução parcial para o próximo agente. Se nenhuma instanciação para esta variável que satisfaça todas as restrições for encontrada, então ele envia uma mensagem de *backtracking* com a CPA para o agente anterior. O agente que receber a mensagem de *backtracking* remove a atribuição de sua variável e tenta reatribuí-la com um valor consistente. O algoritmo termina com sucesso se o último agente consegue encontrar uma atribuição consistente para sua variável. O algoritmo termina sem sucesso se o primeiro agente encontra um domínio vazio.

O algoritmo *Synchronous Backtracking* requer um custo de comunicação para determinar a ordem de instanciação. Este algoritmo não se beneficiar das vantagens do paralelismo, porque somente um agente está recebendo a solução parcial e atuando sobre ela. Desta forma, as atribuições são realizadas seqüencialmente e sincronamente, um agente de cada vez na ordem fixada.

C.1.2 *Asynchronous Backtracking*

O algoritmo *Asynchronous Backtracking* (ABT) [72, 73] permite que agentes executem concorrentemente e assincronamente. Cada agente instancia sua variável e comunica o valor da variável para os agentes vizinhos.

O modelo utilizado por este algoritmo é baseado em variável, onde cada variável pertence a um agente e as restrições são compartilhadas entre agentes [11].

Para definir o algoritmo *Asynchronous Backtracking*, considere um CSP distribuído onde todas as restrições são binárias como numa rede de restrições, as variáveis são nós e as restrições são ligações entre os nós. Cada agente tem exatamente uma variável e um nó representa somente um agente. As ligações entre os nós são direcionadas. Uma ligação é direcionada do agente que envia o valor para o agente que avalia a restrição.

O algoritmo *Asynchronous Backtracking* é uma versão distribuída do algoritmo de *backtracking* tradicional. Neste algoritmo, a ordem de prioridade das variáveis é conhecida (segue a ordem alfabética dos identificadores de variáveis). Cada agente comunica a seus vizinhos a sua tentativa de atribuição de valor através de uma mensagem. Cada agente mantém gravado numa estrutura o valor de atribuição dos agentes vizinhos. Um agente troca sua atribuição de valor se o valor atual não for consistente com as atribuições dos agentes de prioridade mais alta. Se não houver valor que seja consistente com os agentes de prioridade mais alta, o agente gera uma nova restrição denominada *Nogood*. Este *Nogood* é comunicado aos agentes de mais alta prioridade, que deverão trocar seu valor. Cada agente atua assincronamente e concorrentemente.

Para detectar terminação são utilizados os algoritmos de detecção de terminação distribuída [72]. O algoritmo termina quando todos os valores das variáveis satisfazem todas as restrições, alcançando uma solução ou então quando todos os agentes estão esperando por uma mensagem que não chega e o algoritmo descobre que não há solução e termina.

Uma limitação do *Asynchronous Backtracking* é que a ordenação da variável/agente é realizada estaticamente. Se a seleção de um valor por um agente de prioridade mais alta for ruim, os agentes de prioridade mais baixa precisam realizar uma busca exaustiva para revisar a decisão ruim.

C.1.3 *Asynchronous Weak-Commitment Search*

O algoritmo *Asynchronous Weak-Commitment Search* (WCS) [73] busca reduzir o risco de decisões ruins, como no *Asynchronous Backtracking*, introduzindo a heurística *min-conflit* [22] . Esta heurística escolhe o valor que remove o menor número de valores dos domínios das próximas variáveis. Considera-se cada valor no domínio da variável atual e associa com ele o número total de valores nos domínios das variáveis futuras com os

quais ele conflita mas que são consistentes com a atribuição parcial atual. Os valores das variáveis atuais são então selecionados em ordem crescente de acordo com esta definição.

Além de usar esta heurística, a ordenação do agente é realizada dinamicamente tal que uma decisão ruim possa ser revisada reduzindo a busca exaustiva.

Para detectar terminação são utilizados os algoritmos de detecção de terminação distribuída.

Nos trabalhos [71, 72] os algoritmos *Synchronous Backtracking*, *Asynchronous Backtracking* e *Asynchronous Weak-Commitment Search* foram avaliados através de simulação.

C.1.4 *Distributed Backtracking*

Este algoritmo é definido considerando que cada agente possui uma variável.

O algoritmo *Distributed Backtracking* (DIBT) [34] é totalmente assíncrono, mas precisa de uma ordenação entre os agentes para aplicar o esquema de *backtracking*, que assegura completude. Então, é necessário encontrar uma ordenação parcial entre os agentes que será seguida pelas instanciações de variável e gerar uma ordenação total para guiar os passos de *backtracking*. A ordenação usada é baseada na topologia do grafo de restrições, que poderá reduzir o espaço de busca e a passagem de mensagem.

É utilizado um método distribuído de ordenação de variável denominado *Distributed Agente Ordering*, DisAO [33]. Neste método os agentes cooperativamente constroem uma ordenação global entre os sub-problemas. Esta ordenação é que define uma relação hierárquica entre os agentes.

Cada agente computa os pais e os filhos no grafo. Durante a busca os agentes enviarão valores instanciados aos filhos e no caso de um *dead-end* o agente realiza *backtrack* para o primeiro agente filho. Então, é necessária uma ordenação dos filhos. Os agentes sem filhos estabelecem que são de nível 1 e comunicam esta informação aos seus conhecidos. Os outros agentes pegam o valor de nível máximo recebido do filho, adiciona 1 a este valor e envia essa informação para os seus vizinhos. Com este novo ambiente de informação, cada agente rearranja os agentes no seu conjunto de filhos local aumentando o nível. Cada agente pode realizar o algoritmo de ordenação localmente com um número constante de mensagens com cada conhecido. Na ordenação resultante, agentes de mesmo nível são independentes, ou seja, não compartilham qualquer restrição e podem realizar computações paralelas ao mesmo tempo.

No processo de busca do algoritmo DIBT, cada agente instancia sua variável de acordo com as restrições de seus pais. Após a instanciação o agente informa o valor escolhido a seus filhos. Se nenhum valor satisfaz as restrições com os agentes filhos, uma mensagem de *backtrack* é enviada para o agente pai mais próximo. Como o *backtrack* ocorre entre variáveis conectadas, é um *backtrack* baseado em grafo. A mensagem de *backtrack* inclui

o conjunto de filhos ordenados dos agentes, as posições de nível e os valores conhecidos pelos agentes. O receptor da mensagem de *backtrack* primeiro verifica a validade da mensagem comparando o valor corrente com o reportado na mensagem. Se o valor for diferente significa que o agente emissor ainda não recebeu informação sobre o último valor do receptor, então a decisão de *backtrack* pode ser obsoleta. Se a comparação for igual e se o agente não pode pegar um valor diferente ele faz *backtrack*. Ele envia uma mensagem de *backtrack* para o agente mais próximo no conjunto união ordenado dos filhos e do conjunto dos emissores. Este novo conjunto é anexado à mensagem com os valores relacionados e posição de nível dos agentes para garantir continuidade do grafo baseado em *backjumping*.

A terminação ocorre quando os agentes são instanciados e esperam por uma mensagem ou quando um agente fonte encontra um conjunto *backtrack* vazio e neste caso o problema não tem solução. A satisfação global ocorre quando agentes instanciados de acordo com restrições pai estão esperando por uma mensagem e quando nenhuma mensagem transita na rede de comunicação.

A avaliação do algoritmo em [34] foi feita através de experimentos realizados numa rede local onde cada agente era atribuído a um processador.

C.1.5 *Interleaved Parallel Search Algorithm*

O algoritmo *Interleaved Parallel Search Algorithm* - IDIBT [33] considera que cada agente possui somente uma variável.

IDIBT mistura busca paralela com busca distribuída e é uma generalização do DIBT, descrito anteriormente. O algoritmo funciona em camadas alternadas de exploração de sub-espaços dentro de cada agente. O paralelismo é alcançado entre agentes distintos porque eles podem considerar sub-espaços distintos ao mesmo tempo.

Para adicionar busca paralela ao DIBT é necessário dividir o espaço de busca em partes independentes e em cada parte uma busca *backtrack* distribuída será realizada. No sistema há dois tipos distintos de agentes: o agente *fonte* que particiona o seu espaço de busca em diversos sub-espaços denominados *contexto*; os demais agentes tentam instanciar o espaço de busca em cada *contexto*.

Não há duplicação de unidades de processamento. Os agentes irão sucessivamente considerar busca nos diferentes *contextos*. Essas camadas de exploração serão alcançadas por operações de passagem de mensagem. O *contexto* de resolução adicionado a cada mensagem permite a um agente sucessivamente explorar espaços de busca disjuntos.

Em cada *contexto* c cada agente instancia sua variável de acordo com as restrições dos agentes pai. Depois da instanciação o agente comunica aos seus filhos o valor instanciado. Cada mensagem é interpretada no seu *contexto* particular. O valor reportado na mensagem

é armazenado e um *timestamp* é associado a ele. Finalmente, o agente tenta conseguir um valor que seja compatível com a nova mensagem. Se um valor compatível for encontrado, uma mensagem com o *contexto* c é enviado aos filhos. Se nenhum valor satisfizer as restrições com os agentes pai uma mensagem de *backtrack* é enviada no *contexto* c ao agente pai mais próximo.

O agente que recebe a mensagem de *backtrack* verifica a validade comparando seu *timestamp* com o reportado na mensagem e verifica que conhecimento compartilhado é reportado com o mesmo *timestamp*. No caso de valores diferentes, o emissor ou o receptor ainda não receberam alguma informação. A decisão de *backtrack* poderia ser obsoleta ou mal interpretada.

Quando a mensagem é válida, há dois comportamentos possíveis. Se o agente descobrir um valor compatível no restante do espaço de busca este valor é endereçado aos agentes filho. Se o valor não pode ser encontrado, há dois casos possíveis. O primeiro é porque é um agente sem possibilidade de fazer *backtracking*. Este agente detecta problema no sub-espaço c. Uma mensagem de que não é possível encontrar solução no *contexto* c é enviada ao agente de *sistema*. Este agente extra pára a computação distribuída no *contexto* c e faz um *broadcasting* de uma mensagem *stop* para todo o sistema de agentes. Se todo o *contexto* não tiver solução, a computação é terminada. Além disso, ele pára a computação quando a solução é encontrada. Um algoritmo para detecção de estado global é executado. A satisfação global ocorre quando num *contexto* c particular, agentes instanciados de acordo com as restrições com os agentes pai estão esperando por uma mensagem e quando não há mensagem transiente com *contexto* c na rede de comunicação, ou seja, cada instanciação local satisfaz as restrições locais.

Se houver um agente pai para realizar *backtracking*, o agente envia uma mensagem de *backtrack* para o agente mais próximo no conjunto união ordenado dos pais e do conjunto de emissores. esse novo conjunto é anexado à mensagem com informação relacionada sobre agentes para garantir continuidade do *backjumping* baseado em grafo. O emissor está esperando receber uma mensagem, mas se o objetivo global for maximizar a satisfação no sistema, o agente pode conseguir de volta seu espaço de busca local inicial e um valor compatível com os anteriores.

IDIBT possui balanceamento de carga implícito, pois quando um *contexto* c detecta insolubilidade a busca interna é cancelada. Na busca paralela o comportamento básico quando um sub-espaço é terminado é rearranjar a distribuição de carga entre as unidades processantes. Sem realocação, algumas unidades se tornam ociosas e outras podem possuir muito trabalho. Quando o *contexto* c é terminado, mais tempo de CPU e banda são alocados para os sub-espaços restantes. O agente *fonte* é que tem que realocar seus sub-espaços.

Os resultados apresentados em [33] foram obtidos de experimentos realizados numa rede Linux, com algoritmos implementados em C++ e biblioteca de passagem de mensagens MPI [52]. Cada variável do DisCSP foi dedicada a um único computador.

C.1.6 *Dynamic Backtracking*

Este algoritmo é definido considerando que cada agente possui exatamente uma variável; o número do agente é identificado com o índice de sua variável; todas as restrições são entre agentes e são restrições binárias; o grafo de restrições possui agentes como nós e as arestas são restrições entre agentes.

O algoritmo *Dynamic Backtracking* (DB) [31] é um procedimento de busca que mantém para cada valor c removido de uma variável V_k como um *Nogood* orientado, que possui a forma $V_i = a \wedge V_j = b \wedge ... \Rightarrow V_k \neq c$, de acordo com a atribuição dos valores $a, b, ...$ às variáveis $V_i, V_j, ...$ Os lados direito e esquerdo do *Nogood* são separados pela seta $\Rightarrow$. O *Nogood* orientado é mantido no *store* de *Nogood*. O *Dynamic Backtracking* seleciona uma variável não atribuída e tenta atribuir valores para ela. Se um valor é inconsistente com uma variável atribuída anteriormente, aquele valor é descartado e o *Nogood* correspondente é adicionado ao *store* de *Nogood*. Quando todos os valores da variável V_k forem englobados por algum *Nogood*, eles são resolvidos computando um novo *Nogood*. Seja V_j a variável mais recente cronologicamente do lado esquerdo dos *Nogoods*, com o valor b. O lado esquerdo do *Nogood* é a conjunção dos lados esquerdos de todos os *Nogoods* para valores de V_k removendo a variável V_j. O lado direito do *Nogood* é $V_j \neq b$. O novo *Nogood* é adicionado ao *store*, removendo aqueles *Nogoods* com a variável V_j do seu lado esquerdo. A variável V_j é não atribuída e o procedimento itera selecionando a nova variável para atribuição.

Os resultados obtidos em [31] foram gerados a partir de simulação.

C.1.7 *Distributed Dynamic Backtracking*

O algoritmo *Distributed Dynamic Backtracking* (DisDB) [12] é definido considerando que cada agente possui exatamente uma variável; o número do agente é identificado com o índice de sua variável; todas as restrições são entre agentes e são restrições binárias; o grafo de restrições possui agentes como nós e as arestas são restrições entre agentes.

Este algoritmo possui características dos algoritmos *Asynchronous Backtracking* (ABT) [72, 73], *Dynamic Backtracking* (DB) [31] e *Distributed Backtracking* (DIBT) [34].

No grafo de restrições, as restrições são orientadas formando um grafo acíclico orientado, onde a hierarquia de um agente pode ser definida usando um critério de heurística (grau máximo ou grau mínimo, por exemplo) ou os identificadores dos agentes ordenados em ordem total.

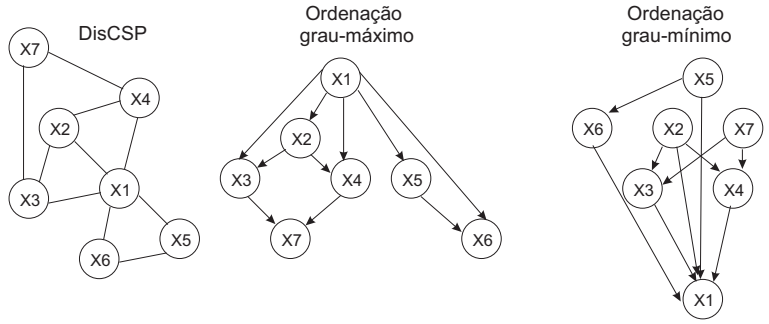


Figura C.2: DisCSP e topologias de agentes com ordenação por variáveis [60]

A Figura C.2 apresenta um CSP distribuído (DisCSP) e as topologias de agentes com ordenação por variáveis em ordem de grau máximo e grau mínimo.

Este algoritmo é executado em cada agente, mantendo sua própria visão de agente e seu *Nogood*. Uma visão de agente consiste no conjunto de valores que o agente armazena como valores que foram atribuídos anteriormente aos outros agentes de acordo com a ordem total. A visão de um agente é sempre consistente com os *Nogoods* no *store*. Os agentes trocam atribuições e *Nogoods*. Quando um agente recebe um *Nogood* ele é aceito se for consistente com a visão do agente, caso contrário é eliminado. Um *Nogood* é aceito para atualizar a visão do agente e o *store* de *Nogoods*. Quando todos os valores de um agente são descartados por algum *Nogood*, o conjunto de *Nogoods* armazenados é resolvido como um caso centralizado, gerando um novo *Nogood*, que é enviado para a variável no seu lado direito. Esta variável soma todas as variáveis do lado esquerdo do novo *Nogood*. Estas variáveis têm a sua atribuição desfeita na visão do agente e os *Nogoods* são atualizados. O processo termina quando uma solução é encontrada ou quando um *Nogood* vazio é gerado, significando que o problema não tem solução.

O algoritmo utiliza a topologia da rede o máximo possível evitando a adição de ligações de comunicação entre agentes que não compartilham restrições entre agentes. Isso evita o envio de mensagens a agentes que não possuem conhecimento comum e, portanto, não precisam receber a informação.

Em [12] o algoritmo não é avaliado experimentalmente. A completude e terminação do algoritmos são provadas de maneira formal.

C.1.8 *Concurrent Backtrack Search*

O algoritmo *Concurrent Backtrack Search* (ConcBT) [77] atua sobre uma rede de restrições distribuída composta por agentes que possuem sua própria rede de restrições local. O algoritmo executa em cada um dos agentes no CSP distribuído.

O algoritmo é composto por diversos processos de busca concorrentes que atuam sobre partes disjuntas do espaço de busca. Cada agente possui os dados relevantes para

seu estado em cada subespaço de busca numa estrutura de dados separada denominada Processo de Busca. Os agentes passam as suas atribuições para os outros agentes numa estrutura de dados denominada CPA (*Current Partial Assignment*). Cada CPA representa um processo de busca. Uma CPA contém uma lista ordenada de triplas $< A_i, V_j, val >$, onde A_i é o agente que possui a variável V_j e val é um valor, do domínio de V_j, atribuído a V_j.

Um agente que recebe uma CPA tenta atribuir valores às suas variáveis locais que não conflitarem com as atribuições na CPA, usando somente os domínios atuais no processo de busca relacionados à CPA recebida. A unicidade da CPA para cada espaço de busca garante que as atribuições não sejam feitas concorrentemente num único subespaço de busca.

Cada agente que recebe uma CPA adiciona a ela um conjunto de atribuições de suas variáveis locais que seja consiste com as atribuições anteriores. Se o agente conseguir fazer a atribuição corretamente a CPA é passada para o próximo agente. Caso contrário, o agente faz *backtrack* enviando a CPA para o agente do qual foi recebida.

O algoritmo ConcBT é capaz de dividir o espaço de busca dinamicamente. Dividir o espaço de busca de uma variável implica em dividir os valores do domínio desta variável em diversos grupos. Cada sub-domínio define um subespaço de busca único e uma CPA única que passa pelo espaço de busca.

Cada agente pode gerar um conjunto de CPAs que dividem o espaço de busca de uma CPA que passou através daquele agente, dividindo o domínio de uma de suas variáveis. Os agentes podem realizar divisões independentemente e manter as estruturas de dados resultantes (Processos de Busca) de forma privada. Múltiplas CPAs são passadas entre os agentes, cada uma representando um único processo de busca. CPAs são criadas pelo agente iniciante, no início da execução do algoritmo, ou dinamicamente por qualquer agente que divida um espaço de busca ativo durante a execução do algoritmo.

Uma operação de *backtrack* é realizada por um agente que falha para encontrar uma atribuição consistente no espaço de busca correspondente à atribuição parcial na CPA. *Backtrack* é realizado retornando a CPA ao agente que atribuiu por último valor às suas variáveis. Agentes que fazem a divisão dinâmica, têm que coletar todas as CPAs de retorno antes de realizar uma operação de *backtrack*.

O procedimento de *backtracking* é síncrono. Os processos de busca são gerados dinamicamente. O domínio de uma ou mais variáveis pode ser particionado. A divisão do espaço de busca é feita independentemente pelos agentes. Cada agente possui sua rede local de restrições. Há diversos processos de busca concorrente, mas o procedimento de *backtrack* é síncrono. Este algoritmo permite o balanceamento de carga automático.

A busca termina quando uma mensagem inclui uma atribuição completa de todas as

variáveis de todos os agentes e retorna a solução. Há a circulação de múltiplas CPAs.

Em [77] o algoritmo é avaliado através de simulação.

C.1.9 *Concurrent Dynamic Backtracking*

O algoritmo *Concurrent Dynamic Backtracking* (ConcDB) [78] considera que cada agente possui sua rede de restrições local, que são conectadas por restrições entre variáveis de diferentes agentes.

ConcDB realiza *backtracking* dinâmico em cada subespaço de busca concorrente, em partes do espaço de busca global, que não possuem interseção. Então, há múltiplos processos concorrentes executando busca concorrente sobre o CSP distribuído. Todos os processos de busca são realizados assincronamente por todos os agentes. Agentes e variáveis são ordenados aleatoriamente em cada processo de busca, de acordo com o subespaço de busca. Os agentes geram e terminam processos de busca dinamicamente durante a execução do algoritmo, criando um algoritmo assíncrono distribuído. O grau de concorrência durante a busca muda dinamicamente e permite o balanceamento de carga automático.

Cada agente que remove um valor de seu domínio atual armazena a atribuição parcial que causou a remoção do valor. Quando o domínio atual de um agente torna-se vazio, o agente constrói uma mensagem de *backtrack* da união de todas as atribuições parciais que causaram remoção de valor (*Nogood*). A mensagem de *backtrack* é enviada ao último agente que gerou uma atribuição que causou remoção de valor. Um *Nogood* pode pertencer a múltiplos espaços de busca e todos eles não contêm solução. Para terminar o processo de busca correspondente um agente que recebe uma mensagem de *backtrack* realiza o seguinte procedimento:

- detectar a qual processo de busca a CPA recebida pertence ou do qual foi dividida;
- verificar se o Processo de Busca foi dividido;
- se o Processo de Busca foi dividido:
  - enviar uma mensagem de que não há solução para o próximo agente relacionado à CPA
  - escolher um novo ID único para a CPA recebida e seu Processo de Busca relacionado
  - continuar a busca usando o Processo de Busca e a CPA com o novo ID
- verificar se há outros Processos de Busca que contêm atribuição parcial inconsistente recebida, enviar mensagem de que não há solução e terminar a busca sobre os processos com novas CPAs geradas.

A mudança do ID faz o processo independente se a mensagem de *backtrack* incluir a CPA original ou uma de suas CPAs derivadas.

Um agente que recebe uma mensagem de que não há solução realiza as seguintes operações para o Processo de Busca sem solução e cada Processo de Busca derivado:

- marca o Processo de Busca como sem solução;
- envia uma mensagem de que não há solução que carrega o ID do Processo de Busca para o agente ao qual a CPA relacionada foi enviada por último.

Agentes que recebem uma CPA primeiro verificam se o Processo de Busca relacionado não está marcado como sem solução. Se estiver marcado a CPA e seu Processo de Busca relacionado são finalizados.

Como os processos são dinamicamente gerados pelo *Concurrent Dynamic Backtracking* é possível detectar que espaços de busca não possuem solução e realizar a terminação mais cedo destes processos.

Em [78] o algoritmo é avaliado através de simulação.

C.1.10 *Asynchronous Backtracking with Dynamic Ordering*

O algoritmo *Asynchronous Backtracking with Dynamic Ordering* (ABT-DO) foi proposto por Zivan e Meisels [80] e considera que cada agente possui somente uma variável.

Em ABT-DO os agentes do CSP distribuído escolhem a ordem dinamicamente e de forma assíncrona. Cada ordem é definida de acordo com o *timestamp* de atribuição dos agentes. Um vetor de contadores representa a prioridade de uma ordem proposta, de acordo com a árvore de busca global. Cada agente pode trocar a ordem de todos os agentes que tenham menor prioridade. Um agente pode propor uma troca de ordem cada vez que ele troca a sua atribuição. Existem algumas regras que precisam ser seguidas para propor a troca de ordem de um agente A_i:

- Agentes com maior prioridade que A_i e o próprio A_i não trocam prioridade na nova ordem;
- Agentes com menor prioridade que A_i, na ordem atual, podem trocar suas prioridades na nova ordem, mas não com agentes de maior prioridade que A_i.

Os contadores anexados a cada ID de agente na lista de ordem forma um *timestamp*. Inicialmente, todo contador de *timestamp* começa com zero e todos os agentes começam com a mesma ordem. Cada agente que propõem uma nova ordem troca a ordem dos pares na lista ordenada e atualiza os contadores, da seguinte forma:

- Os contadores de agentes com prioridade mais alta que A_i não são alterados;

- O contador de A_i é incrementado por um;
- Os contadores de agentes com prioridade mais baixa que A_i são zerados.

Em [80] o algoritmo é avaliado através de simulação.

C.1.11 *Asynchronous Forward-Checking*

O algoritmo *Asynchronous Forward-Checking* (AFC) [48] considera que cada agente possui sua rede de restrições local, que é conectada por restrições entre variáveis de diferentes agentes.

Este algoritmo processa sincronamente somente uma atribuição parcial consistente, mas realiza *forward-checking* assincronamente. O estado do processo de busca é representado por uma estrutura de dados chamada *Current Partial Assignment* (CPA). O agente de inicialização grava suas atribuições na estrutura e a envia para o próximo agente. Cada agente receptor verifica se há um valor consistente para ser atribuído e a adiciona na CPA. Se não houver uma atribuição consistente, o agente desfaz a atribuição e envia a CPA para o agente anterior para revisar sua atribuição.

Cada agente que faz uma atribuição na CPA envia uma cópia da estrutura atualizada para os agentes vizinhos, requisitando que todos os agentes realizem *forward-checking*. Agentes que receberem cópias de atribuições filtram seus domínios removendo valores inconsistentes e no caso de encontrar um domínio vazio envia de volta uma mensagem *Not_OK* que contém a atribuição parcial inconsistente que causou o domínio vazio. A mensagem *Not_OK* é enviada a todos os agentes com variáveis não atribuídas na CPA inconsistente. Um agente que receber a CPA e tiver uma mensagem *Not_OK* envia a CPA de volta numa mensagem *backtrack*. A unicidade da CPA garante que somente um único *backtrack* seja inicializado, mesmo para múltiplas mensagens *Not_OK*. Somente um agente que receber qualquer das mensagens *Not_OK* realizará o *backtrack* e será o primeiro agente que receber a CPA e contiver a mensagem *Not_OK*.

A concorrência do AFC ocorre porque *forward-checking* é feito concorrentemente por todos os agentes. Todos os agentes processam mensagens *forward-checking* concorrentemente e bloqueiam o processo de atribuição de agentes que violarem consistência com variáveis futuras.

Asynchronous Forward-Checking realiza atribuição de um agente de cada vez, mas verifica consistência num processo assíncrono.

Em [48] o algoritmo é avaliado através de simulação.

C.2 Algoritmos que consideram privacidade

C.2.1 *Distributed Forward Checking with Dynamic Ordering*

O algoritmo *Distributed Forward Checking with Dynamic Ordering* (DODFC) proposto por Meisels *et al.* [47] utiliza o conceito de CSP distribuído onde um agente contém um conjunto variáveis, cada variável com seu domínio e os agentes são conectados por restrições entre variáveis que pertencem a diferentes agentes.

Este é um algoritmo de busca distribuído para redes de restrições distribuídas que ordena as variáveis dinamicamente durante a busca. A ordenação da variável e do valor são determinadas de forma distribuída por todos os agentes. Os agentes cooperam trocando mensagens com variáveis e valores propostos e recebendo respostas que avaliam estes candidatos. Em cada estado o melhor candidato é cooperativamente selecionado.

O algoritmo DODFC começa com os domínios iniciais das variáveis. Em cada passo do algoritmo há um agente denominado *agente decisor* que recebe mensagens de variáveis candidatas de todos os agentes. O primeiro *agente decisório* deve ser definido antes do início do algoritmo. Ele pode ser, por exemplo, o agente com menor identificação. Em cada passo a próxima variável para atribuição é selecionada e o agente que inclui a variável selecionada na sua rede local torna-se o *agente decisório*. Na segunda parte do mesmo passo o valor a ser atribuído à variável é escolhido pelo *agente decisório*, baseado numa troca de mensagens com os outros agentes. Um procedimento similar é usado para a propagação da atribuição para todos os agentes. A seleção da próxima variável é realizada novamente por todos os agentes enviando mensagens com variáveis candidatas para o *agente decisório*. O agente que possui a variável selecionada torna-se o *agente decisório*.

É assumido que todas as mensagens são enviadas com sucesso.

Este algoritmo termina quando os valores são atribuídos a todas as variáveis de todos os agentes ou quando não há solução, ou seja, quando algum agente tenta executar *backtrack* de uma iteração número zero.

O algoritmo DODFC apresentado em [47] pode ser adaptado para suportar privacidade de valores atribuídos e restrições. A alteração necessária para obter a privacidade de valores atribuídos consiste em fazer com o agente envie pares variável-valor de todas as variáveis locais (ao invés de enviar somente da atribuição candidata) do agente que está recebendo a mensagem, mas que não está compatível com o candidato. Para alcançar a privacidade das restrições é necessário deixar que o agente refine os domínios das suas variáveis (em resposta a uma mensagem de refinamento) usando somente sua parte da matriz de restrições. A noção de compatibilidade de qualquer atribuição deve ser verificada com variáveis atribuídas e não atribuídas, ou seja, os valores inconsistentes

nos domínios das variáveis não atribuídas serão removidos pela parte conhecida das restrições ou recebendo uma mensagem de incompatibilidade para estes valores de agentes com variáveis atribuídas que estão conflitando com esses valores.

Em [47] o algoritmo é avaliado através de simulação.

C.2.2 *Distributed Forward Checking for Partially Known Constraints*

Este algoritmo considera que cada agente possui apenas uma variável por agente.

Distributed Forward Checking [16] é baseado no algoritmo ABT, que após a atribuição de uma variável de um agente envia o subconjunto do domínio que for compatível com o valor atribuído, ao invés de simplesmente enviar o valor atribuído, para os agentes de prioridade mais baixa. Além disso, ele troca valores verdadeiros por seqüências de números em mensagens *backtracking*. Desta forma, a atribuição de um agente é mantida privada.

Existem dois modelos que consideram privacidade de restrições [16]: *Totally Known Constraints* (TKC) e *Partially Known Constraints* (PKC). O modelo *Totally Known Constraints* assume que quando dois agentes i e j compartilham uma restrição C_{ij} ambos conhecem o escopo da restrição e um deles conhece a parte relacional da restrição. O modelo *Partially Known Constraints* assume que quando dois agentes i e j compartilham uma restrição C_{ij}, num dos dois conhece completamente a restrição. Cada agente conhece a parte da restrição que é capaz de construir, baseado na sua própria informação.

Brito *et al.* [15] propuseram um modelo para restrições assimétricas denominado *Distributed Forward Checking for Partially Known Constraints* (DFC-PKC) [79]. Neste modelo, cada restrição binária é dividida entre dois agentes restritivos. Para resolver o CSP distribuído resultante com restrições assimétricas foi proposto um algoritmo de *backtracking* assíncrono de duas fases. Neste algoritmo, uma ordem de prioridade estática é definida entre todos os agentes.

Na primeira fase, o algoritmo de *backtracking* assíncrono é executado examinando somente as restrições contidas em agentes de prioridade mais baixa. Ou seja, somente um agente de cada restrição verifica consistência. Quando uma solução é alcançada, uma segunda fase é realizada em que a consistência da solução é verificada novamente de acordo com as restrições contidas nos agentes de prioridade mais alta, para cada restrição binária. Se nenhuma restrição for violada, uma solução é reportada. Se houver restrições violadas, a primeira fase termina após a gravação de *Nogoods*.

A desvantagem deste algoritmo é o esforço de produzir soluções em cada primeira fase. Como restrições em direções opostas não são examinadas, grande parte do espaço de busca pode ser varrido exaustivamente. Outra desvantagem é a troca entre as duas fases, que exige um mecanismo de detecção de terminação, que é complicado para um algoritmo de *backtracking* assíncrono. Para que todos os agentes sejam informados sobre

a troca entre fases é necessário um monitoramento global que afeta a independência dos agentes num sistema assíncrono distribuído.

Em [16] os resultados obtidos foram empíricos.

C.2.3 *Asynchronous Backtracking for Asymetric Constraints*

Este algoritmo considera que cada agente possui apenas uma variável por agente. Além disso, as restrições são assimétricas e binárias.

O algoritmo *Asynchronous Backtracking for Asymetric Constraints* (ABT-ASC) foi proposto por Zivan *et al.* [79] e visa realizar *backtracking* assíncrono em uma única fase em CSPs distribuídos assimétricos. Para isso, todas as restrições devem ser satisfeitas incluindo as restrições pertencentes a agentes de mais alta prioridade. Em CSPs distribuídos assimétricos as restrições são somente parcialmente conhecidas de cada agente participante. Conseqüentemente, ambos os agentes que possuem a restrição precisam verificar consistência de suas atribuições um com o outro. Isso inclui agentes de maior e menor prioridade.

O algoritmo ABT-ASC verifica restrições entre agentes, assincronamente, em ambos os agentes restritivos. Os agentes enviam suas atribuições propostas para todos os seus vizinhos no grafo de restrições. Os agentes fazem a atribuição de suas variáveis locais de acordo com a ordem de prioridade (como no *Asynchronous Backtracking*), mas verifica as restrições também de acordo com as atribuições dos agentes de prioridade mais baixa. Quando um agente detecta um conflito entre sua própria atribuição e a atribuição de um agente de prioridade mais baixa, este agente envia um *Nogood* para o agente de prioridade mais baixa, mas mantém sua atribuição. Agentes que recebem um *Nogood* de um agente de prioridade mais alta, realizam as mesmas operações como se eles tivessem produzido este *Nogood*. Os agentes removem sua atribuição atual de seu domínio atual, armazena o *Nogood* de eliminação e reatribui sua variável.

A principal diferença entre ABT e ABT-ASC é que depois de uma atribuição, que é consistente com todas as atribuições dos agentes com maior prioridade na visão do agente é verificada consistência com os agentes de menor prioridade. Para cada violação da restrição entre a atribuição e uma atribuição de um agente de menor prioridade, um *Nogood* contendo ambas as atribuições conflitantes é enviado ao agente de menor prioridade.

O processamento assíncrono de todas as restrições da rede numa única fase gera um espaço de busca menor.

Em [79] os resultados obtidos foram através de simulação.

C.2.4 *Asynchronous Aggregations Search*

Este algoritmo considera que cada agente possui somente uma variável.

Os agentes trocam mensagens não sobre suas atribuições às suas variáveis individuais, mas sobre tuplas de variáveis. Isto permite eliminar restrições de acordo com que as restrições são tratadas. Cada agente mantém os valores para o conjunto de variáveis que as suas restrições envolvem. Uma atribuição corresponde a uma lista de domínios, um para cada variável envolvida que representa todas as tuplas do seu produto cartesiano.

Não há restrições entre agentes. A forma de consistência de valores é garantida para uma variável compartilhada por duas restrições que não se encontram no mesmo agente através da duplicação da variável nos agentes [12].

Este algoritmo assume o modelo baseado em restrições, onde cada restrição pertence a um agente [11].

Uma agregação é uma lista de atribuições (conjunto de pares de variáveis e valores atribuídos). O algoritmo *Asynchronous Aggregations Search* (AAS) [64] permite a filtragem de uma atribuição global por agentes de acordo com suas restrições privadas. Entretanto, neste algoritmo não há privacidade dos domínios, isto é, todos os agentes conhecem os domínios de todas as variáveis. Portanto, este algoritmo viola a privacidade exigida por muitos problemas do mundo real.

Em [64] os resultados obtidos foram através de simulação.

C.3 Considerações Finais

A principal característica do ABT [12] é a forma como ele processa *dead-ends* (quando um agente não consegue um valor consistente para sua variável) para garantir completude da busca. O ABT cria um *Nogood* e envia para o agente de prioridade menor. Quando um agente j recebe um *Nogood* verifica se os valores estão obsoletos. Caso não haja variável desconhecida pelo agente o *Nogood* pode ser eliminado. Se os valores estiverem obsoletos mas o *Nogood* contiver uma variável V_k que ele não conheça (pois havia uma ligação entre V_i e V_k, mas não entre V_j e V_k), o agente j irá adicionar uma ligação de k para j, que não havia anteriormente. O número de ligações que podem ser adicionadas por ABT pode ser muita alta e depende da ordenação da rede.

No AWS [12] a ordenação dos agentes é dinâmica, podendo ser alterada durante a busca. Para completude do algoritmo, *Nogoods* não podem ser eliminados. Assim, AWC tem uma complexidade exponencial.

DIBT [12] tenta preservar a estrutura distribuída da rede. Ele constrói um hierarquia na rede sem adicionar novas ligações. DIBT não armazena *Nogoods*, mas não garante completude.

A busca concorrente do algoritmo *Concurrent Dynamic Backtracking* e do *Interleaved Parallel Search Algorithm - IDIBT* apresentam estratégias de paralelismo similares [78]. Mas IDIBT executa múltiplos processos de *backtracking* assíncrono e sua multiplicidade

é fixada no início de sua execução. A divisão dinâmica do espaço de busca faz com que a busca seja melhor.

Segundo Meisels [46], o procedimento distribuído que mantém consistência local em AFC é mais eficiente do que ABT. Ele necessita verificar, na média, menos restrições por passo de computação. AFC, também, apresenta carga de comunicação (medida pelo número de mensagens enviadas durante a busca) mais baixa do que ABT.

A comparação de ConcDB com ABT [46] mostra que ConcDB apresenta resultados melhores em termos de número de passos concorrentes de computação. ABT apresenta uma alta quantidade de computação local em cada passo porque lê todas as mensagens que ele recebeu para esse passo. A carga de comunicação de ABT é 4 vezes maior do que ConcDB.

Segundo Zivan e Meisels [79], o algoritmo ABT-ASC apresenta melhores resultados que DisFC-PKC. ABT-ASC apresenta uma carga na rede menor do que DisFC-PKC.

Em [48] eles compararam experimentalmente AFC com ABT. O número de verificações de restrições concorrentes por passo de computação de AFC é menor do que em ABT. Isso indica que o procedimento distribuído que mantém consistência local em AFC é eficiente. Além disso, a carga de mensagem de AFC é menor do que de ABT.

Em [80] os autores apresentaram uma comparação de ABT-DO com ABT, usando três heurísticas diferentes: ordenação aleatória dos agentes com prioridades mais baixas; cálculo do tamanho do domínio baseado no fato que os agentes que realizam uma atribuição enviam o tamanho do domínio atual para todos os outros agentes; *Nogood* disparado: agentes trocam a ordem dos agentes de prioridade mais baixa somente quando eles recebem um *Nogood* que elimina suas atribuições atuais. Os resultados mostraram que ABT-DO com ordenação aleatória não melhora os resultados de ABT. Mas ABT-DO com heurística de tamanho de domínio para ordenar agentes de mais baixa prioridade é um pouco melhor que ABT. A heurística *Nogood* disparado aplicada a ABT-DO foi que apresentou os melhores resultados em relação a ABT. Esta heurística também apresenta o menor número de mensagens geradas para ABT-DO em relação a ABT.

Em [77] foi realizada uma comparação entre ConcBT, ABT e DisDB. Os resultados mostraram que ConcBT apresenta menos verificações de restrições atuais do que DisDB e ABT. ConcBT também apresenta menor número de mensagens do que os outros dois algoritmos.

Meisels e Razgon [47] mostraram que DODFC apresenta menor número de verificação de restrições e atribuições do que AWC.

Apêndice D

Estudo sobre atualização de variáveis compartilhadas no PCSOS

O sistema PCSOS possui vários parâmetros que podem ser configurados. Este apêndice apresenta um estudo sobre dois deles: o momento de atualização das variáveis compartilhadas (equivalente a troca de mensagens num sistema distribuído) e a quais os processadores serão comunicados da atualização. Uma atualização pode ser comunicada logo que a alteração é gerada (I - Imediatamente) ou podem ser acumuladas alterações e enviadas num determinado ponto de sincronização (S - Sincronização). A alteração da variável pode ser comunicada a todos os processadores (T - Todos) ou somente para aqueles que possuem dependência com a variável alterada (G - Grupo). Para simplificar, nos gráficos utilizamos as letras I, S, T e G para representarem estes parâmetros.

Os resultados apresentados são referentes ao tempo de execução, ao total de mensagens e ao total de trabalho para cada particionamento em cada aplicação. Portanto, os particionamentos são: Blocos por variáveis, Blocos por *indexicals*, Blocos por restrições, *Round-Robin* por variáveis, *Round-Robin* por *indexicals*, *Round-Robin* por restrições, *Grouping-Sink* por *indexicals* e *Grouping-Sink* por restrições. As aplicações são *Arithmetic*, *Queens*, PBCSP (com 50 e 100 variáveis) e *Sudoku*.

O tempo de execução corresponde à média de 10 execuções e está expresso em segundos (s.). O total de "mensagens" é a soma de todas as atualizações das variáveis compartilhadas (equivalente a troca de mensagem num sistema distribuído) em cada processador. O total de trabalho corresponde à soma de todos os *indexicals* executados (*exec-ind*) em todos os processadores.

D.1 *Arithmetic*

As Figuras de D.1 a D.7 apresentam os gráficos com os resultados de cada particionamento em tempo de execução, total de trabalho e total de mensagens obtidos com *Arith-*

metic.

Blocos por variáveis Para este particionamento o menor tempo de execução para 1, 2, 4 e 8 processadores é apresentado pela combinação ST na Figura D.1. A diferença entre SG, ST e IG é inferior a 6%. Para 12 e 14 processadores o menor tempo de execução fica com IG e a diferença com SG e ST é cerca de 20%. A menor quantidade de mensagens é apresentada pela combinação SG. O total de trabalho apresentado até 12 processadores é praticamente o mesmo para todas as combinações. Considerando o tempo de execução, o total de trabalho e o total de mensagens, as combinações que se destacaram foram SG e IG.

Blocos por *indexicals* No gráfico de tempo de execução da Figura D.2, observamos que até 8 processadores o tempo de execução de SG e ST são os menores. O tempo de execução de IG é mais alto do que SG e ST, mas a diferença é inferior a 11%. Para 12 processadores, o tempo de IG é menor do que SG cerca de 5%. No gráfico de total de mensagens, fica claro que SG e IG apresentam a menor quantidade de mensagens. O gráfico de total de trabalho mostra que IG e IT apresentam a menor quantidade de trabalho. Considerando os 3 gráficos, as combinações que se destacaram foram SG e IG.

Blocos por restrições O tempo de execução apresentado na Figura D.3 mostra que as combinações SG, ST e IG apresentam os menores tempos de execução. A diferença entre eles é inferior a 9%. No gráfico de total de mensagens, SG e IG apresentam a menor quantidade de mensagens. No gráfico de total de trabalho, IG e IT apresentam a menor quantidade de trabalho. De acordo com os 3 gráficos, as combinações que se destacam são IG e SG.

Round-Robin **por variáveis** Os menores tempos de execução apresentados no gráfico de tempo de execução da Figura D.4 são IG e IT até 8 processadores e SG e IG para 12 e 14 processadores. No gráfico de total de mensagens SG e IG apresentam a menor quantidade de mensagens. No gráfico de total de trabalho IG e IT apresentam as menores quantidades. Considerando os 3 gráficos, as combinações que se destacam são IG e SG.

Round-Robin **por** *indexicals* O menor tempo de execução para 1, 2, 4 e 8 processadores é apresentado com a combinação IT, mostrado no gráfico de tempo de execução da Figura D.5. Para 12 e 14 processadores o menor tempo foi para ST. SG e ST apresentam os menores números de mensagens e IG e IT apresentam a menor quantidade de trabalho. Observando os 3 gráficos, notamos que as combinações que se destacam são ST e IT.

Arithmetic										
Particion.	Variáveis			*Indexicals*			Restrições			Avalia
	Tempo	Trabalho	Mensagens	Tempo	Trabalho	Mensagens	Tempo	Trabalho	Mensagens	
Blocos	SG ST IG	SG ST IG IT	SG IG	SG ST IG	IG IT	SG IG	SG ST IG	IG IT	SG IG	**SG** **IG**
Round-Robin	SG IG IT	IG IT	SG IG	ST IG IT	IG IT	SG ST	ST IT	IG IT	SG ST	**ST** **IT**
Grouping-Sink	- - -	- - -	- - -	IG	IG IT	SG IG	SG ST IG	SG ST	SG IG	**SG** **IG**

Tabela D.1: Resumo dos resultados para *Arithmetic*

Round-Robin **por restrições** Na Figura D.6, o gráfico de tempo de execução mostra que ST e IT apresentam os menores tempos de execução. Mas a diferença em relação à combinação que apresentou maior tempo de execução é inferior a 13%. SG e ST apresentam as menores quantidades de mensagens. IG e IT realizaram as menores quantidades de trabalho. Analisando os 3 gráficos em conjunto, as combinações IT e ST são as que mais se destacam.

Grouping-Sink **por** *indexicals* As combinações IG apresenta o menor tempo de execução no gráfico de tempo de execução apresentado na Figura D.7. No gráfico de total de mensagens podemos observar que SG e IG apresentam as menores quantidades de mensagens. IG e IT são as combinações com a menor quantidade de trabalho no gráfico de total de trabalho. Pelos 3 gráficos, observamos que a combinação que mais se destacou foi IG .

Grouping-Sink **por restrições** O gráfico de tempo de execução da Figura D.8 mostra que as combinações SG, ST e IG apresentam os melhores resultados. O gráfico de total de mensagens mostra que SG e IG geram menos mensagens. Em termos de total de trabalho todas as combinações apresentam aproximadamente a mesma quantidade de trabalho até 8 processadores. Mas para 12 e 14 processadores SG e ST apresentam menos total de trabalho. De acordo com os 3 gráficos SG foi a combinação que mais se destacou.

A Tabela D.1 apresenta um resumo de todos os particionamentos para *Arithmetic*. Na última coluna (Avalia), estão as combinações que mais se destacaram. Como SG e IG foram as combinações que mais se destacaram para Blocos e *Grouping-Sink* vamos escolher uma delas. Como SG é a que gera, de uma forma geral, a menor quantidade de mensagens está é a escolhida para realizar os experimentos, cujos resultados são apresentados no Capítulo 6.

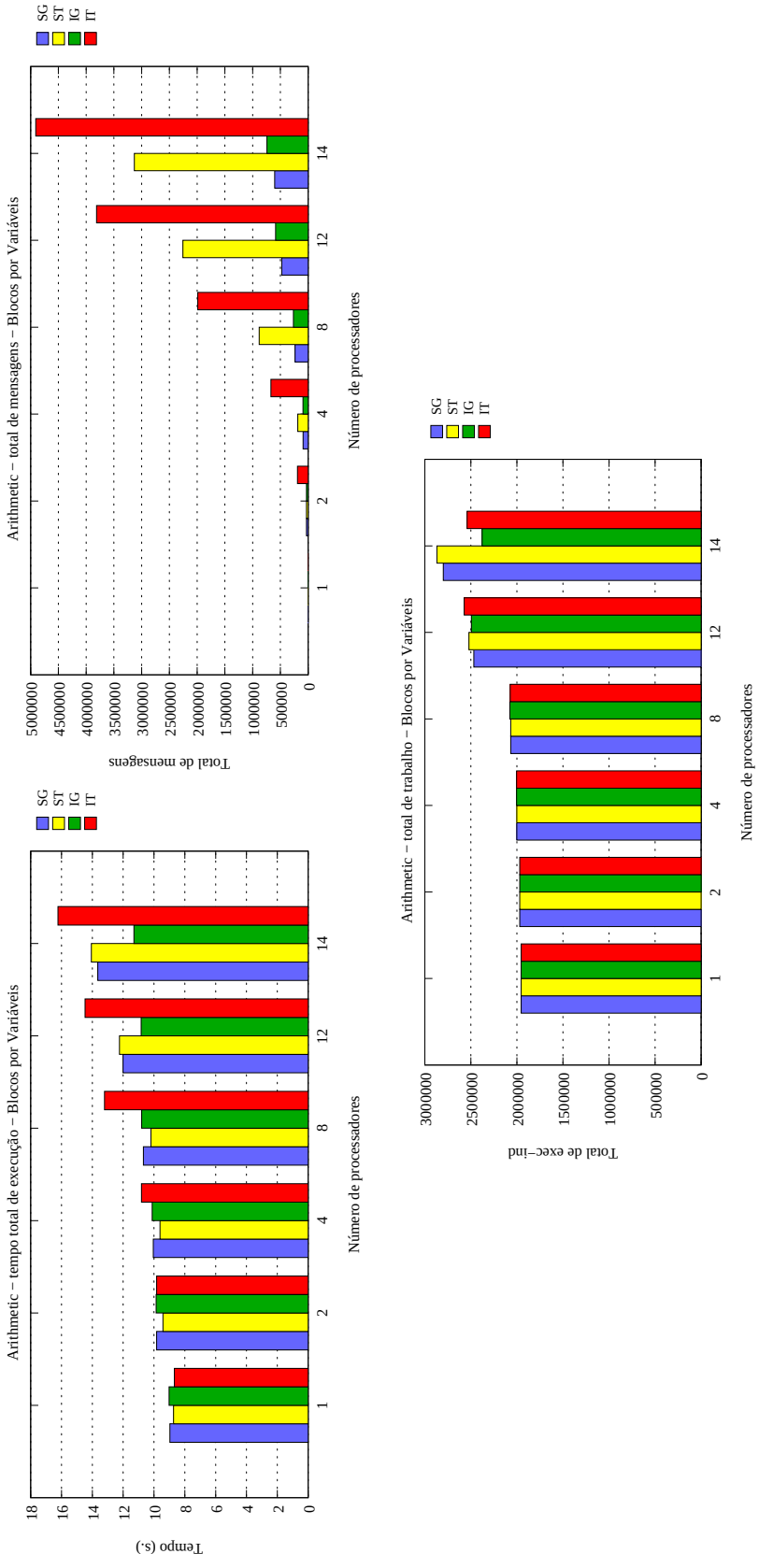


Figura D.1: *Arithmetic* - Blocos por variáveis

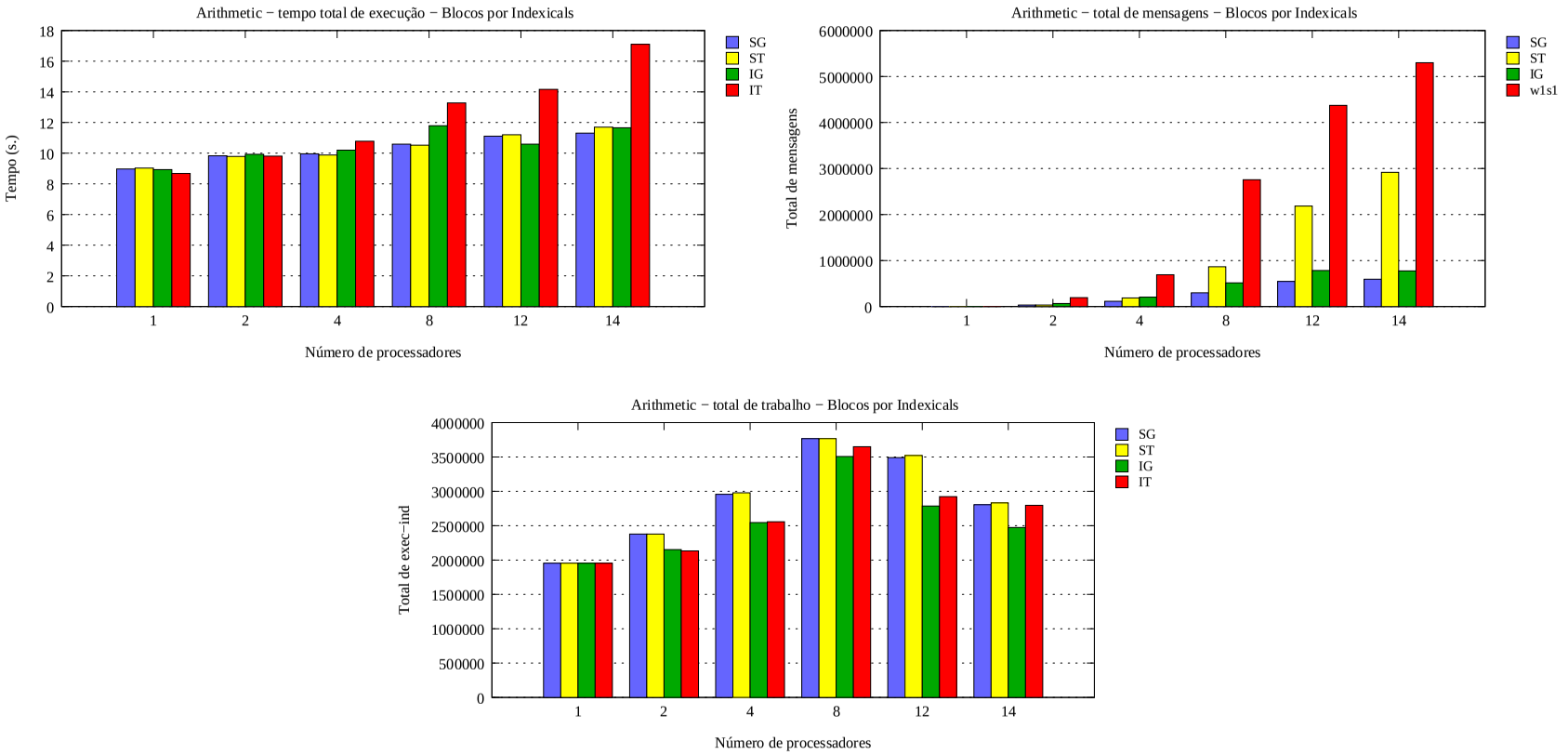


Figura D.2: *Arithmetic* - Blocos por *indexicals*

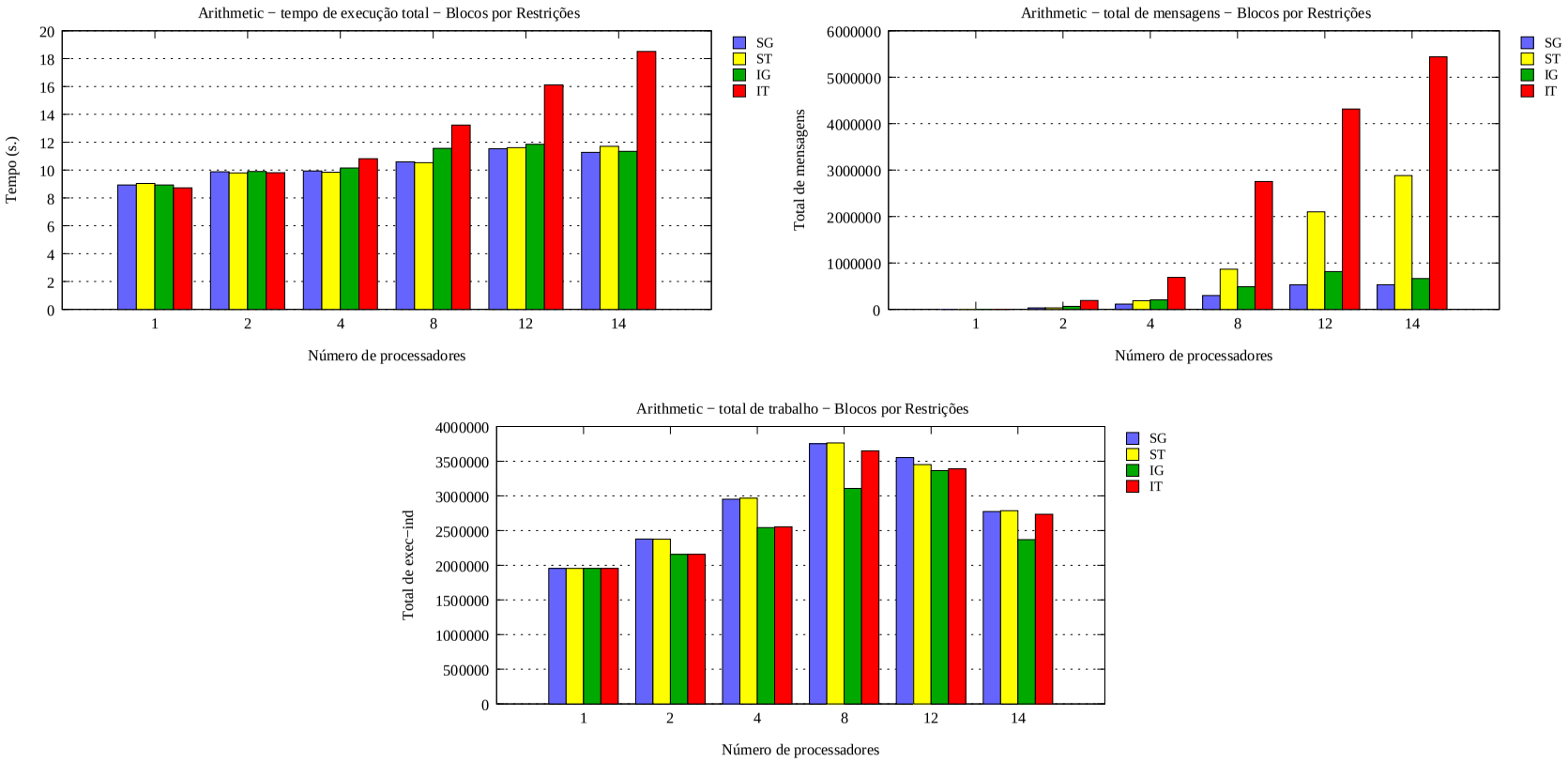


Figura D.3: *Arithmetic* - Blocos por restrições

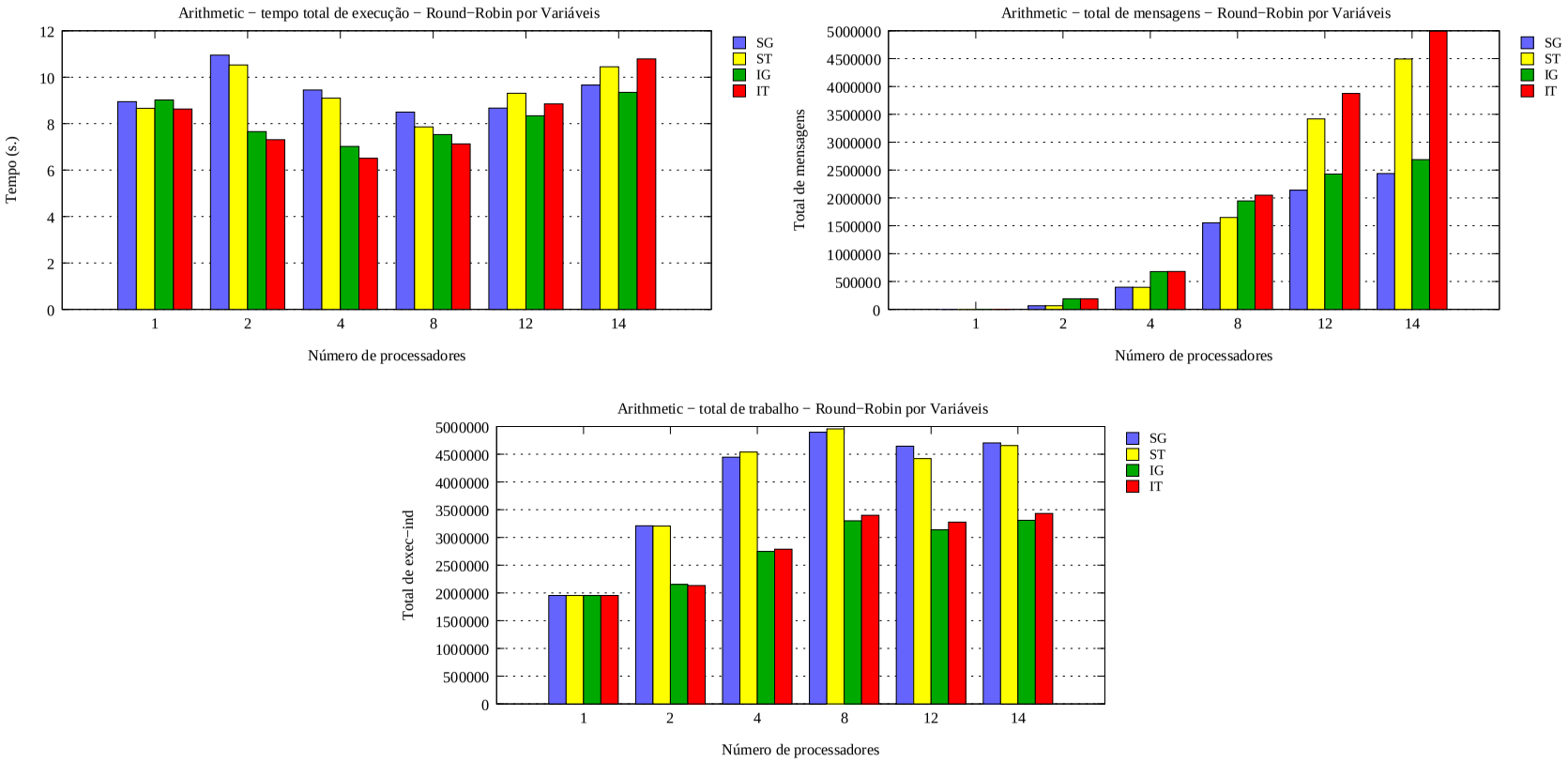


Figura D.4: *Arithmetic* - *Round-Robin* por variáveis

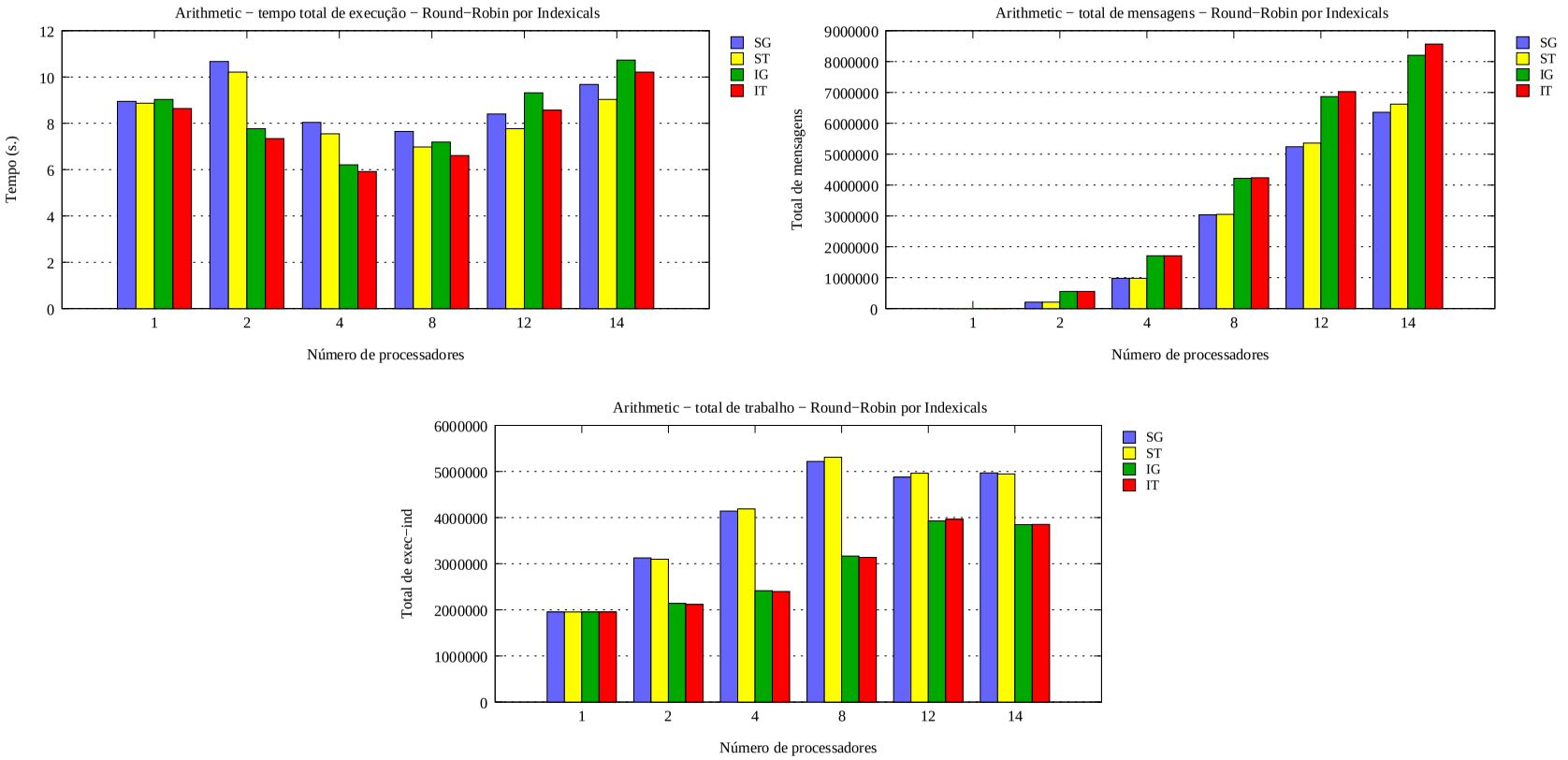


Figura D.5: *Arithmetic - Round-Robin* por *indexicals*

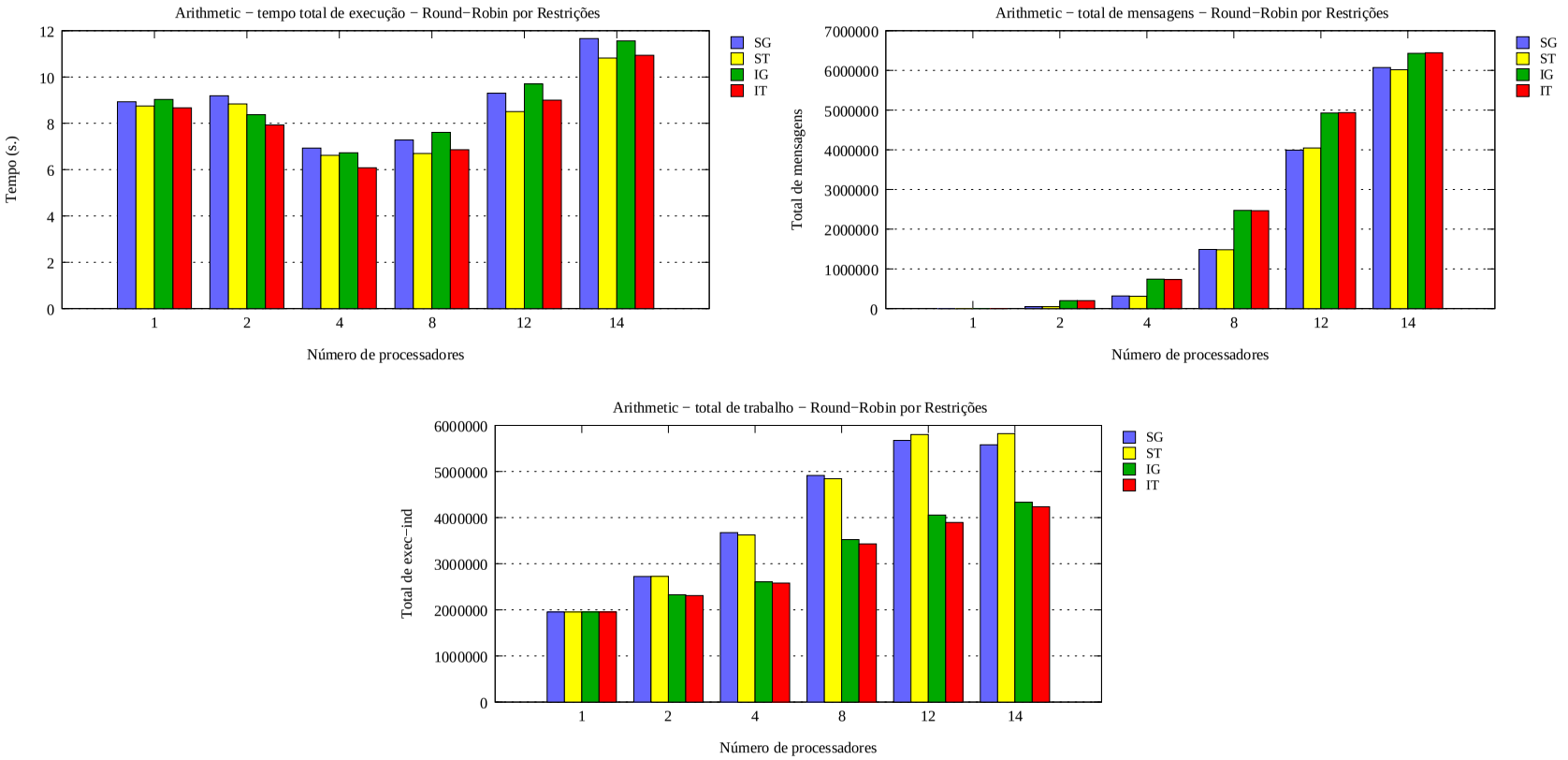


Figura D.6: *Arithmetic - Round-Robin* por restrições

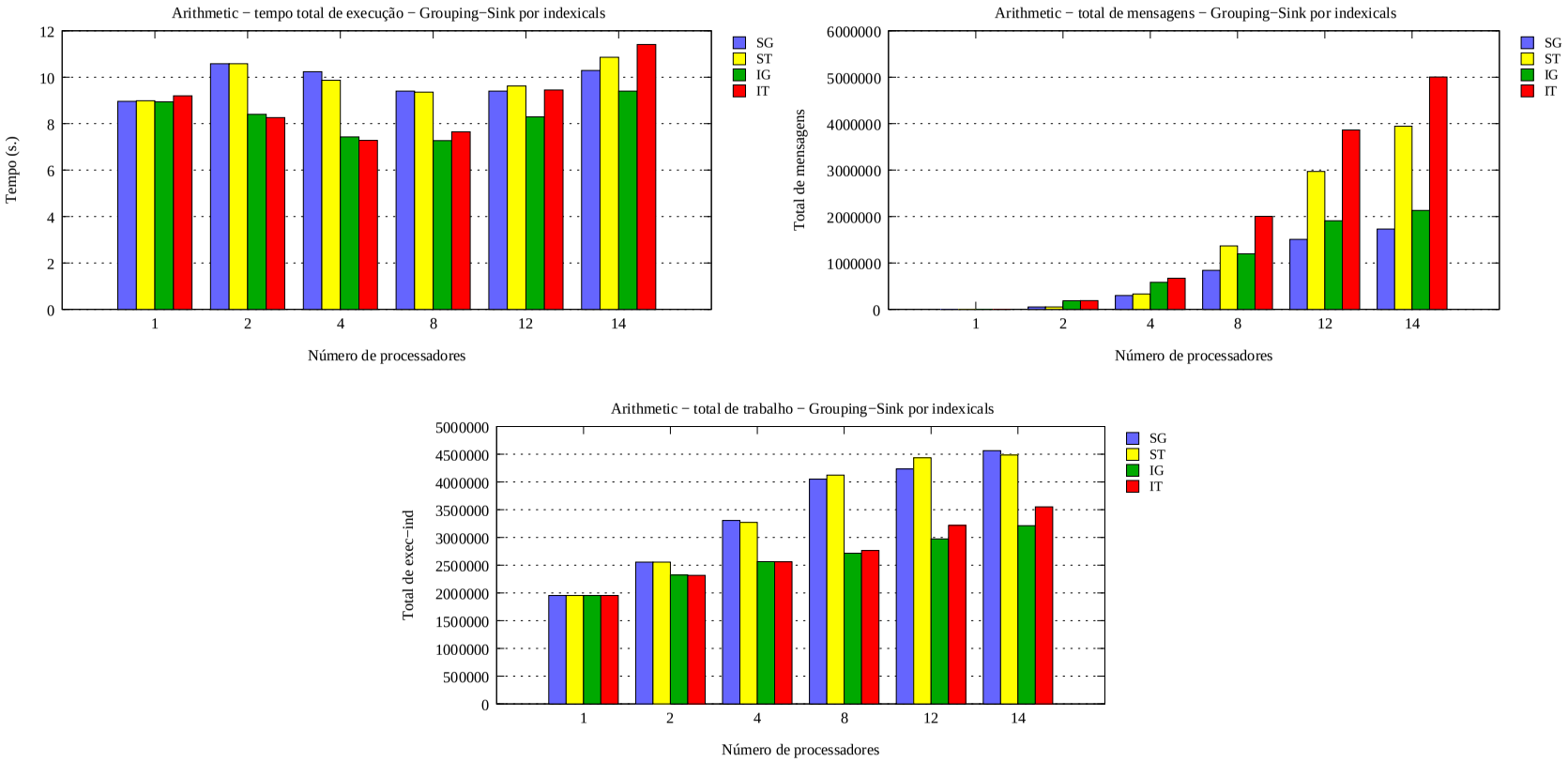


Figura D.7: *Arithmetic - Grouping-Sink* por *indexicals*

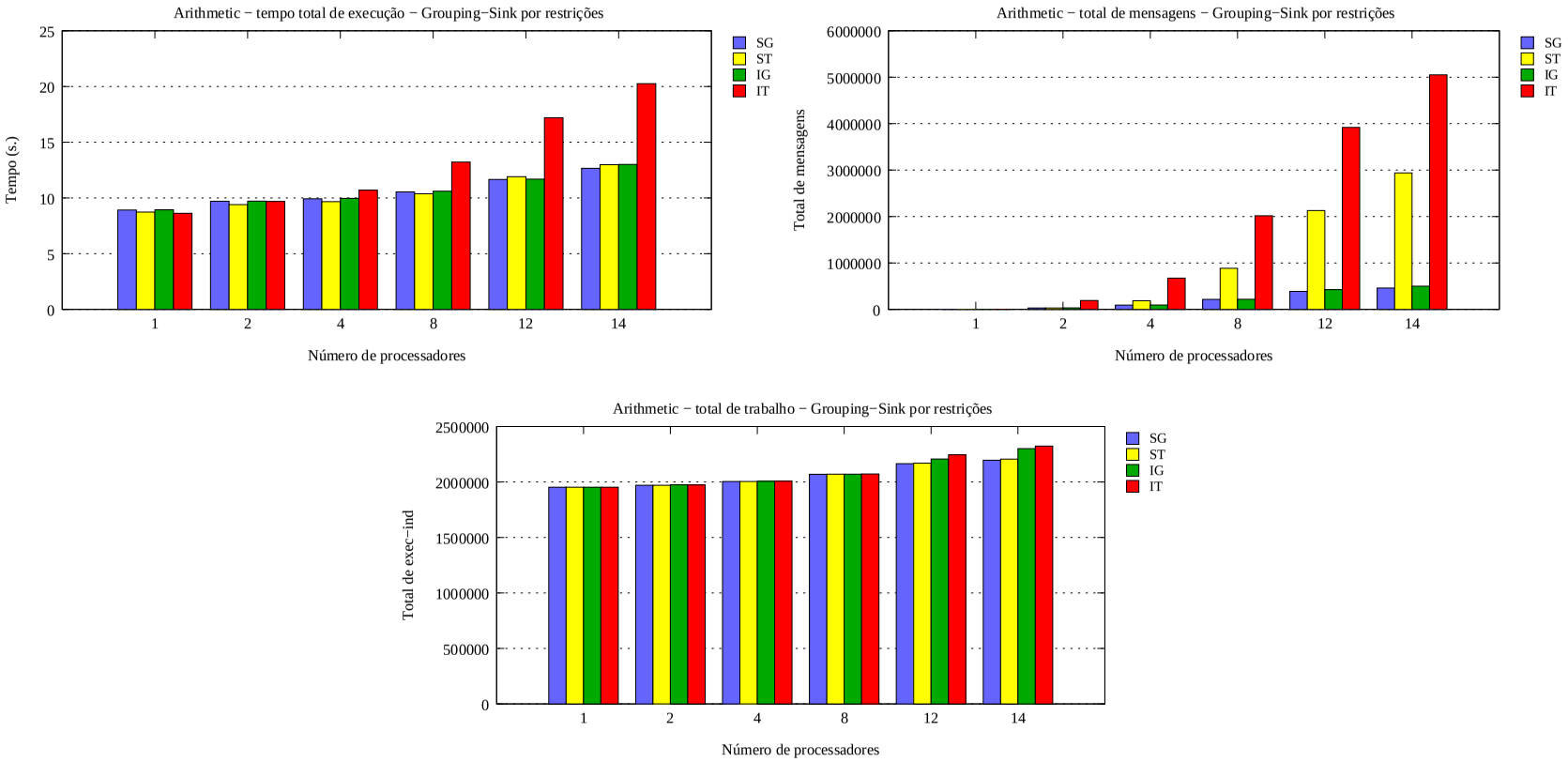


Figura D.8: *Arithmetic - Grouping-Sink* por restrições

D.2 *Queens*

As Figuras de D.9 a D.16 apresentam os gráficos de tempo de execução, total de mensagens e total de trabalho para todos os particionamentos, com as combinações de parâmetros SG, ST, IG e IT.

Blocos por variáveis No gráfico de tempo de execução da Figura D.9, é mostrado que SG e ST apresentam os menores tempos de execução. Nos gráficos de total de mensagens e total de trabalho estas são as melhores combinações. Desta forma, estas são as combinações que mais se destacam para este particionamento.

Blocos por *indexicals* As combinações SG e ST apresentam tempos de execução mais baixos no gráfico de tempo de execução da Figura D.10. Somente para 4 processadores é que IT tem o menor tempo de execução (diferença de inferior a 25%). Mas SG tem a menor quantidade de mensagens. Em quantidade total de trabalho SG e ST foram as que mais se destacaram. Observando os 3 gráficos, a combinação SG foi a que mais se destacou.

Blocos por restrições As combinações SG e ST apresentam o menor tempo de execução (Figura D.11). Mas SG possui o menor total de mensagens. Em total de trabalho, SG e ST apresentam a menor quantidade. Observando os 3 gráficos, podemos dizer que SG foi a combinação que mais se destacou.

Round-Robin **por variáveis** Na Figura D.12 os gráficos de tempo de execução, total de mensagens e total de trabalho apresentam que SG e ST são as combinações que mais se destacam, por apresentarem menores tempos de execução, quantidade de mensagens e total de trabalho.

Round-Robin **por** *indexicals* O menor tempo de execução até 8 processadores foi para IT (gráfico de tempo de execução da Figura D.13). Até 8 processadores SG e ST apresentam praticamente o mesmo tempo de execução. Para 12 e 14 processadores ST apresenta um tempo de execução um pouco menor que IT (diferença inferior a 4%). O total de mensagens é praticamente o mesmo para SG e ST, o mesmo ocorrendo para o total de trabalho. Considerando os 3 gráficos, as combinações que mais se destacaram foram ST e IT.

Round-Robin **por restrições** O gráfico de tempo de execução da Figura D.14 mostra que para 2 e 4 processadores IG e IT apresentam o menor tempo de execução. Para 8 e 14

Queens										
Particion.	Variáveis			*Indexicals*			Restrições			Avalia
	Tempo	Trabalho	Mensagens	Tempo	Trabalho	Mensagens	Tempo	Trabalho	Mensagens	
Blocos	SG ST	SG ST	SG ST	SG ST IT	SG ST	SG	SG ST	SG ST	SG	**SG**
Round-Robin	SG ST	SG ST	SG ST	ST IT	IG IT	SG ST	SG ST IT	IG IT	SG ST	**SG** **ST**
Grouping-Sink	- - - -	- - - -	- - - -	SG ST	SG ST	SG ST	SG ST IT	SG ST IG IT	SG	**SG** **ST**

Tabela D.2: Resumo dos resultados para *Queens*

processadores ST apresentou o menor tempo de execução. E para 12 processadores SG obteve o menor tempo. O total de mensagens é menor para SG e ST e total de trabalho é menor para IG e IT. Considerando os 3 gráficos, as combinações que mais se destacaram foram SG e ST.

Grouping-Sink **por** *indexicals* Pelos gráficos da Figura D.15, SG e ST apresentam os melhores tempos de execução, totais de mensagens e totais de trabalho. Portanto, estas duas combinações foram as que mais se destacaram.

Grouping-Sink **por restrições** Na Figura D.16, o gráfico de tempo de execução, SG, ST e IT apresentam o menor tempo de execução. O menor total de mensagens é apresentado por SG. O total de trabalho é similar para as quatro combinações. Considerando os 3 gráficos, as combinações que mais se destacaram foram SG e ST.

A Tabela D.2 apresenta um resumo de todos os particionamentos para *Queens*, com tempo de execução, total de trabalho e total de mensagens. Na última coluna há um resumo das combinações que se destacaram. Observando todos os particionamentos, observamos que SG foi a combinação que apresentou bons resultados para todos os particionamentos, quando aplicado a *Queens*.

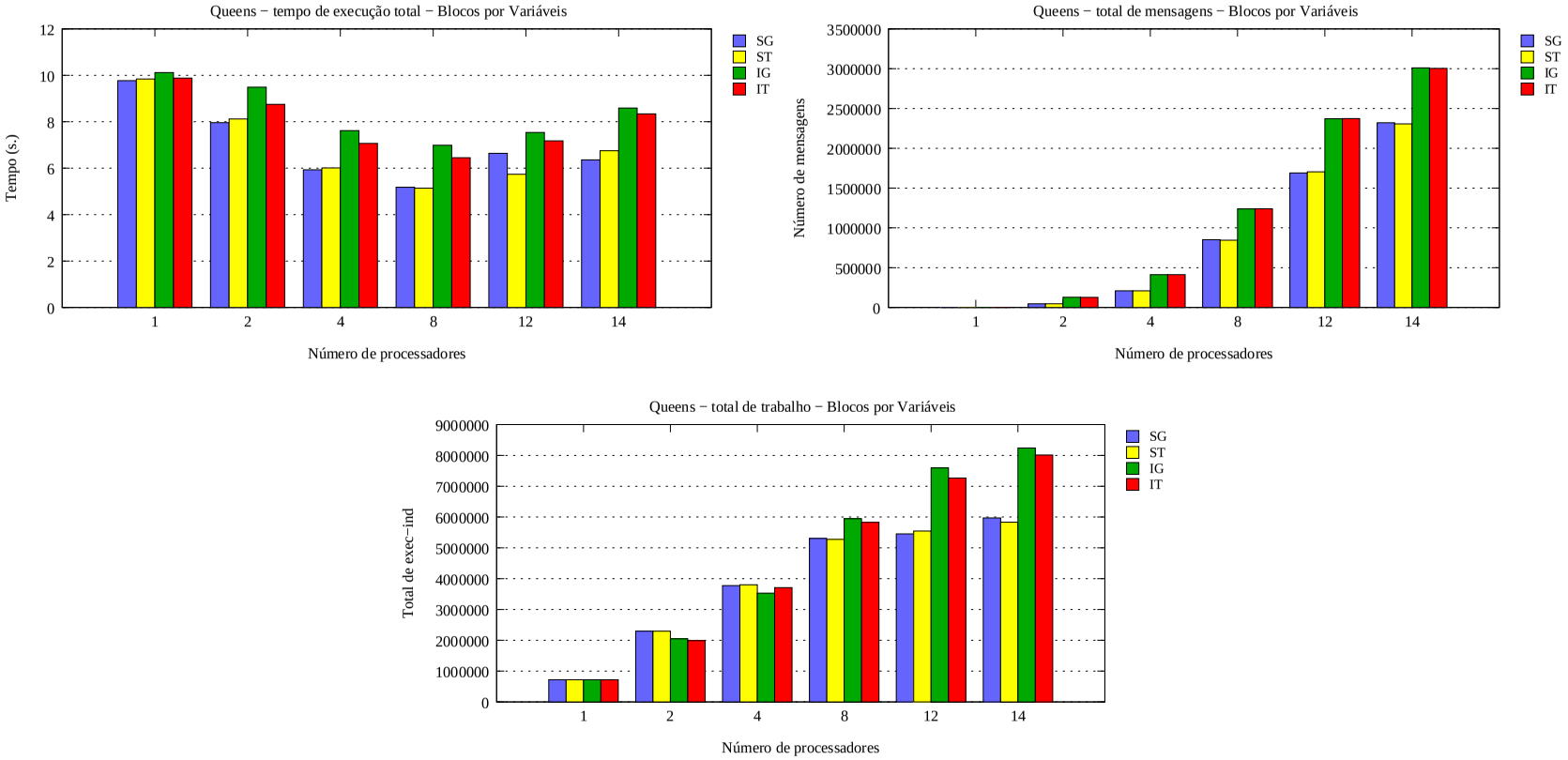


Figura D.9: *Queens* - Blocos por variáveis

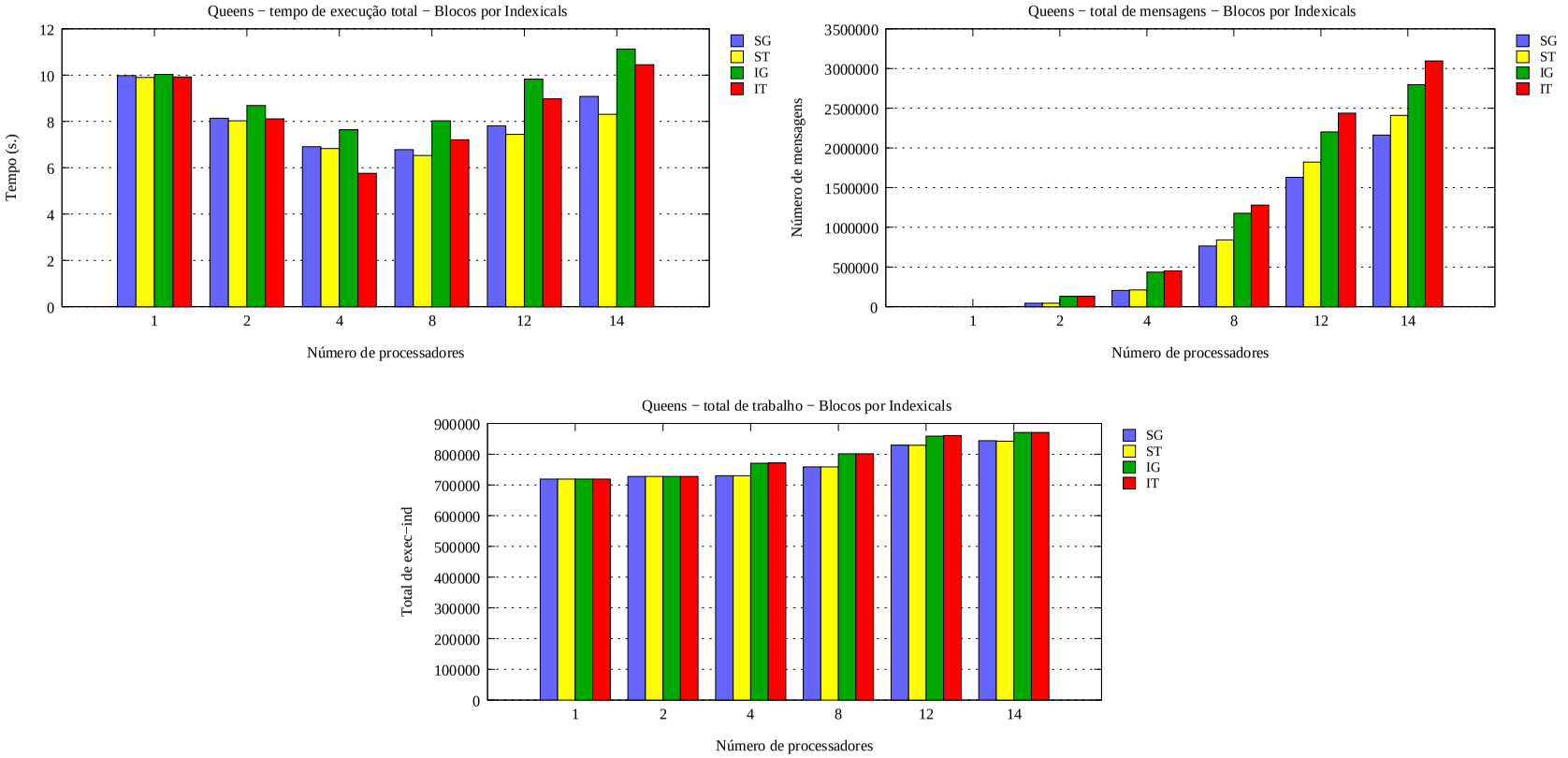


Figura D.10: *Queens* - Blocos por *indexicals*

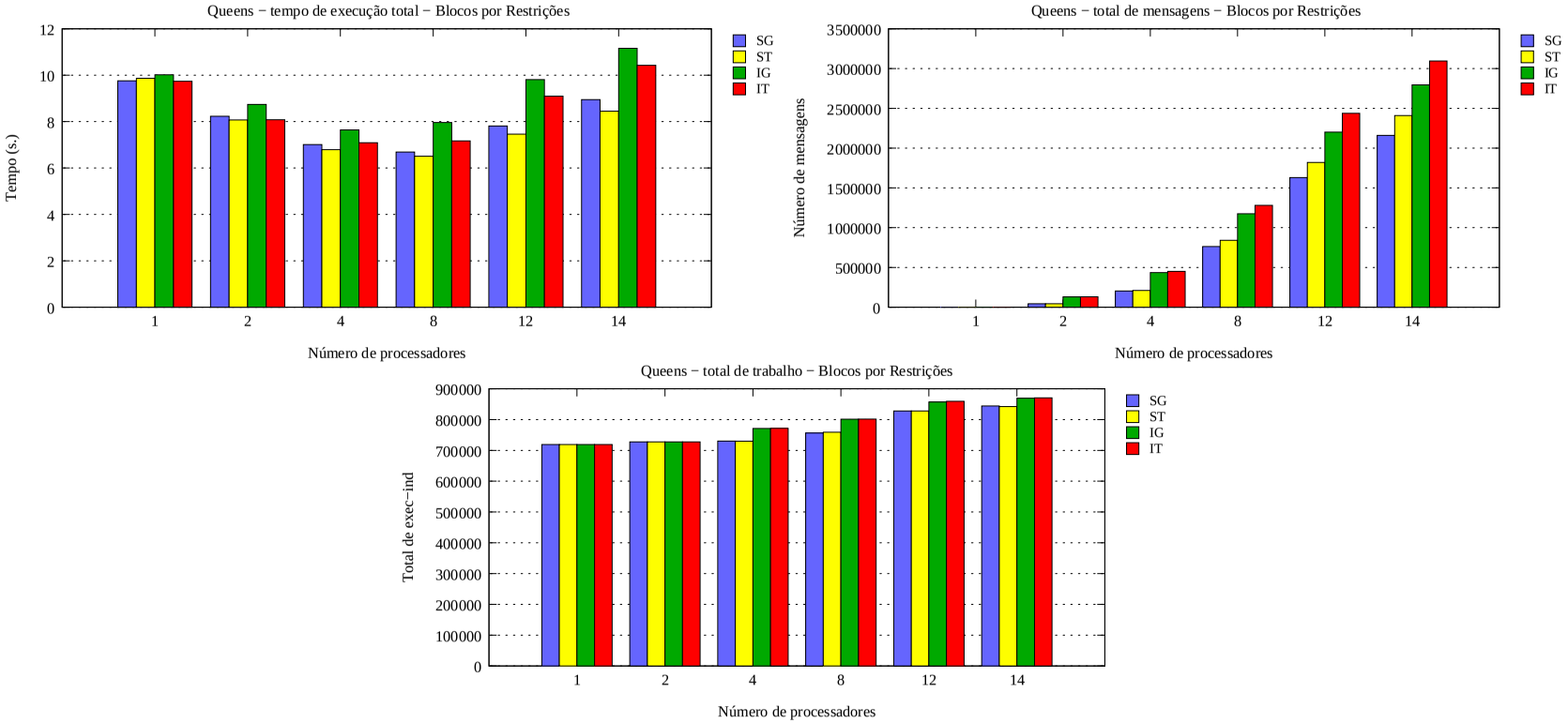


Figura D.11: *Queens* - Blocos por restrições

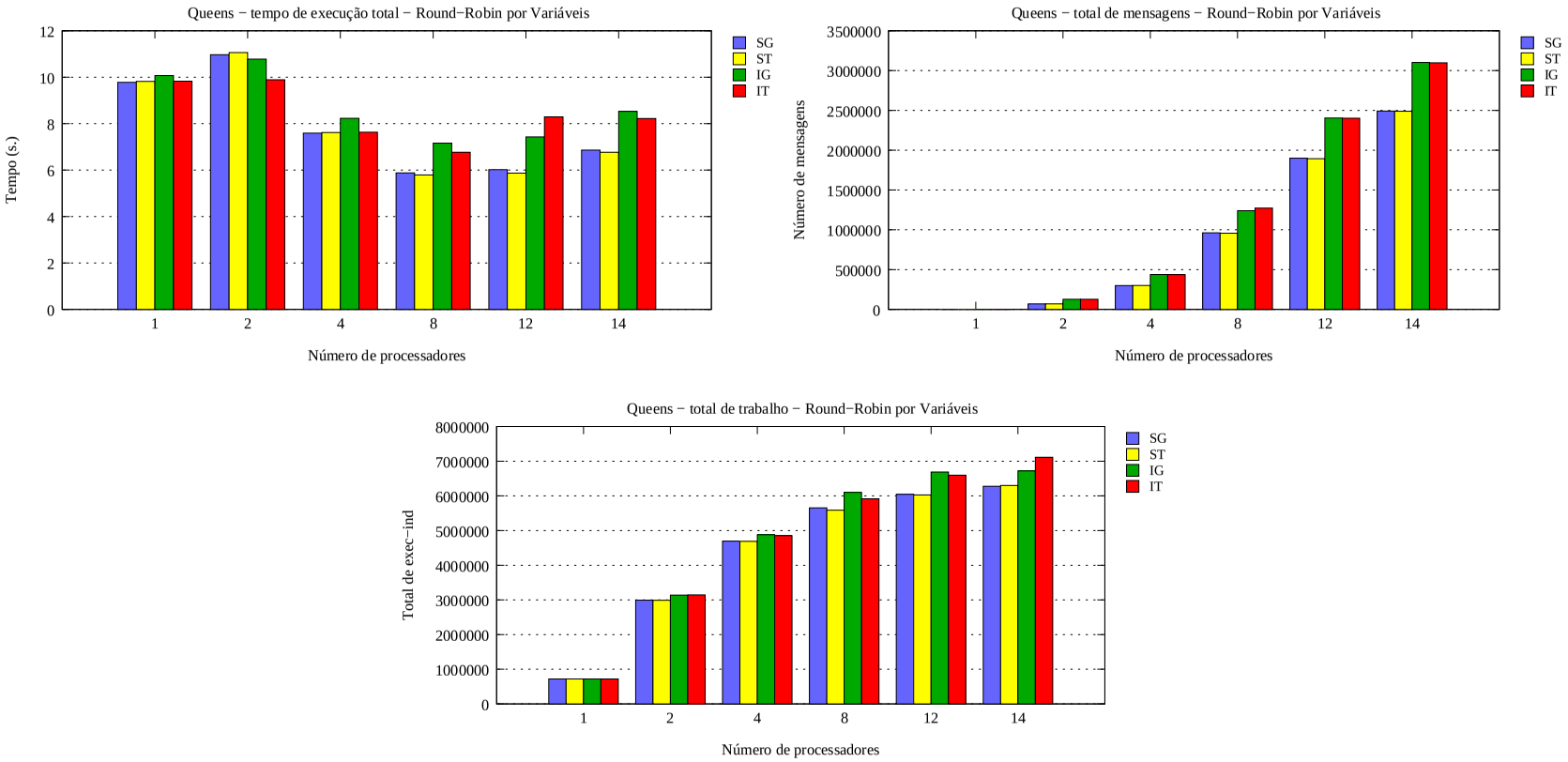


Figura D.12: *Queens - Round-Robin* por variáveis

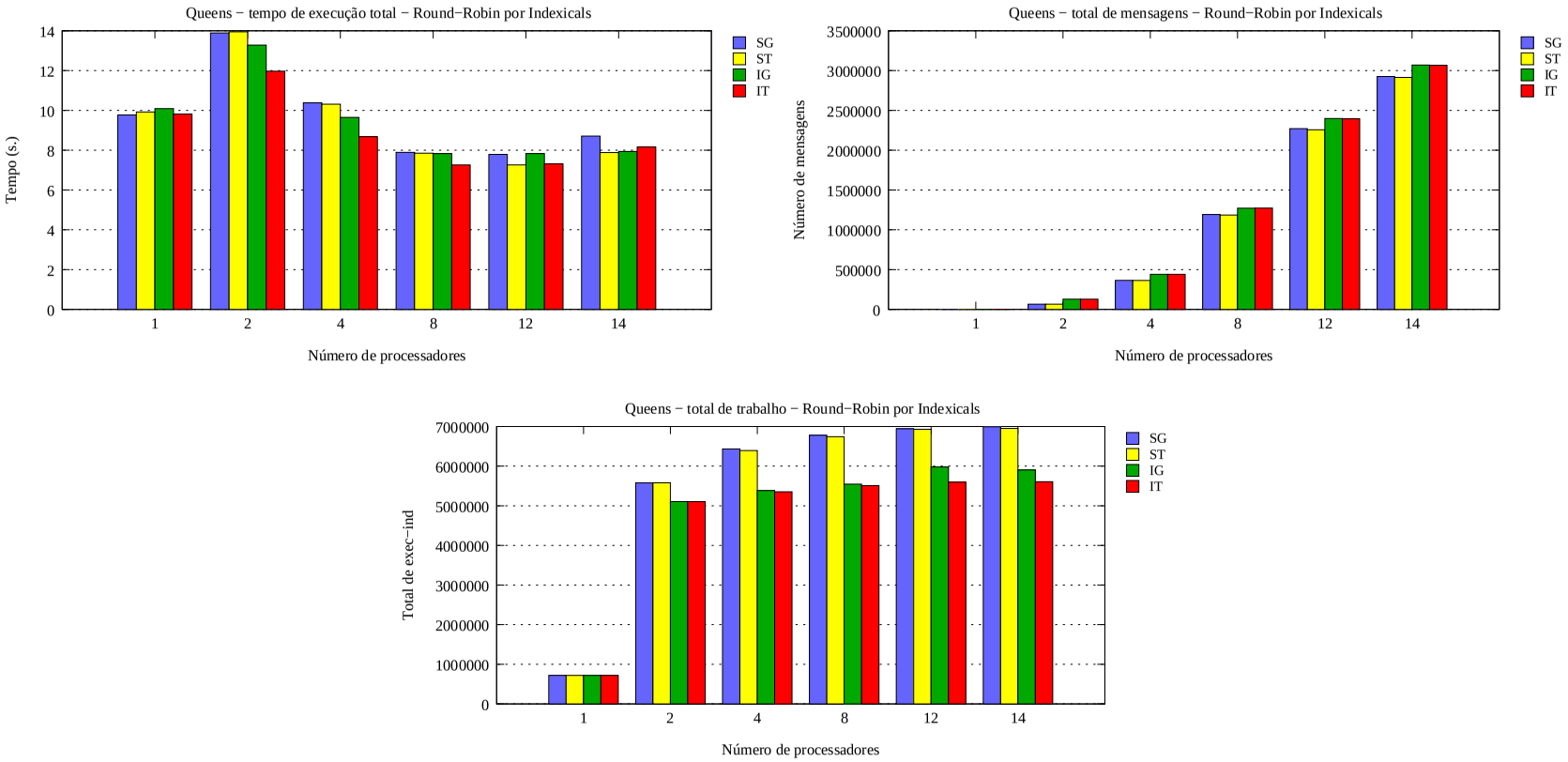


Figura D.13: *Queens - Round-Robin* por *indexicals*

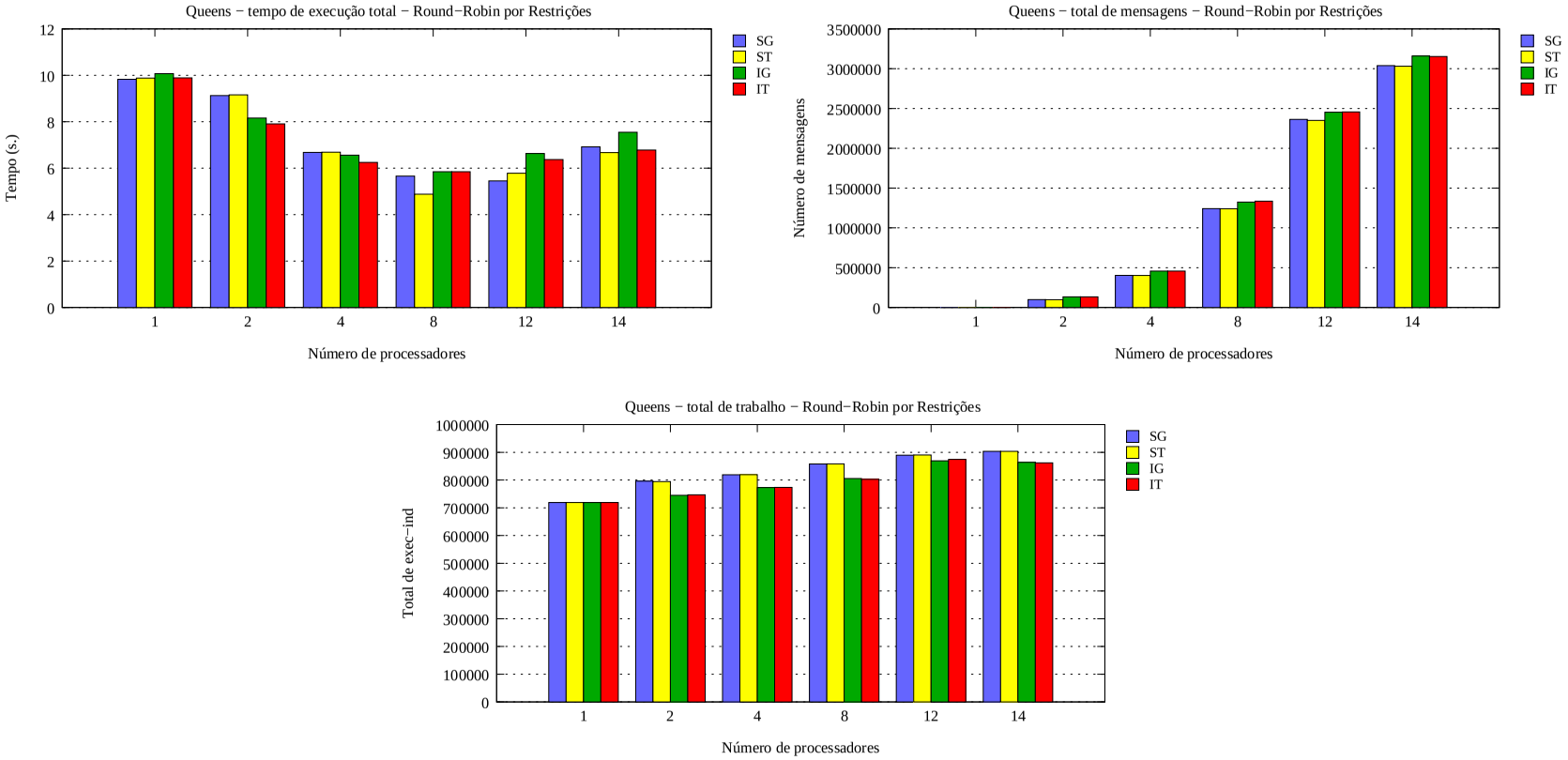


Figura D.14: *Queens - Round-Robin* por restrições

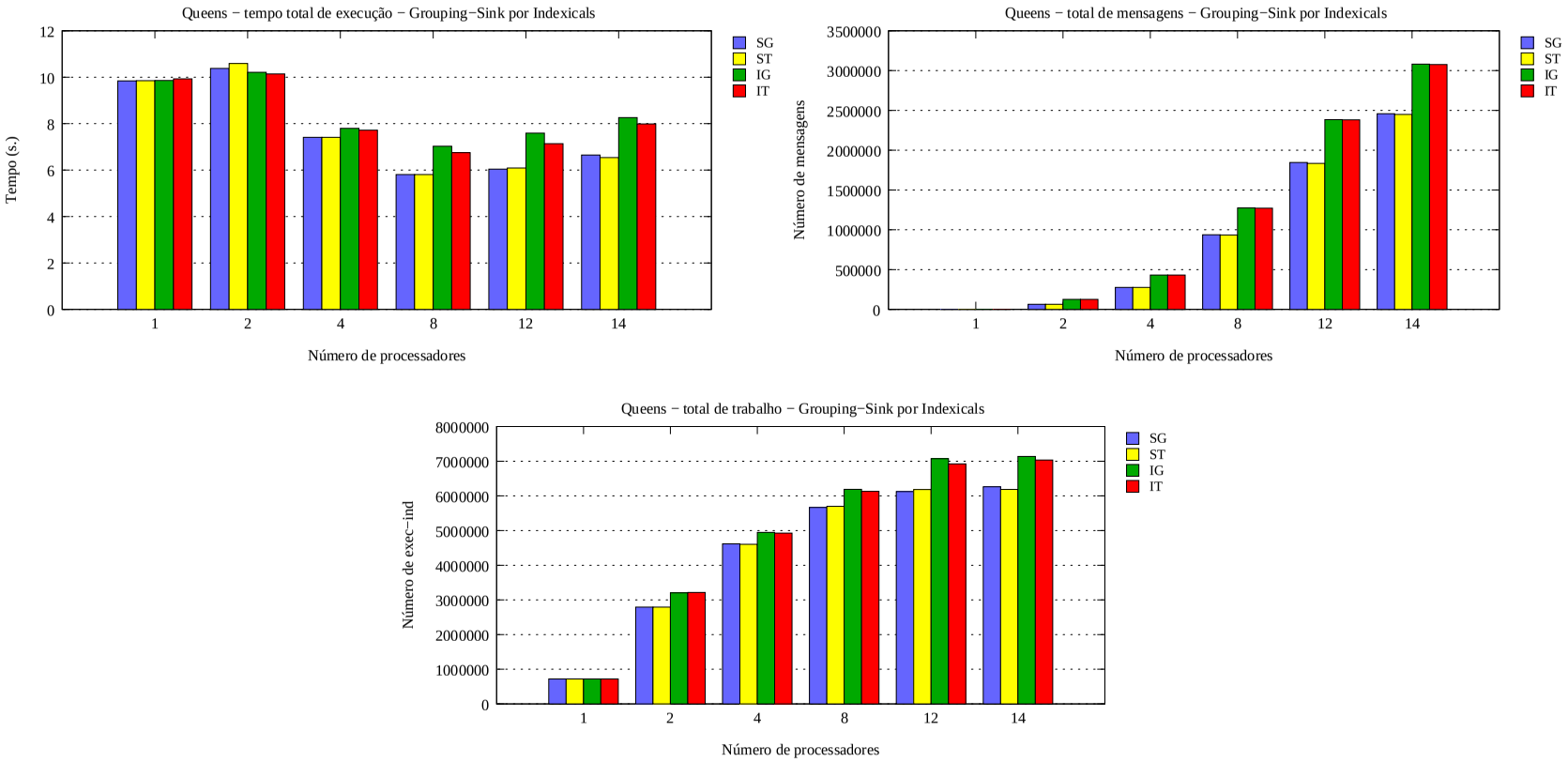


Figura D.15: *Queens - Grouping-Sink* por *indexicals*

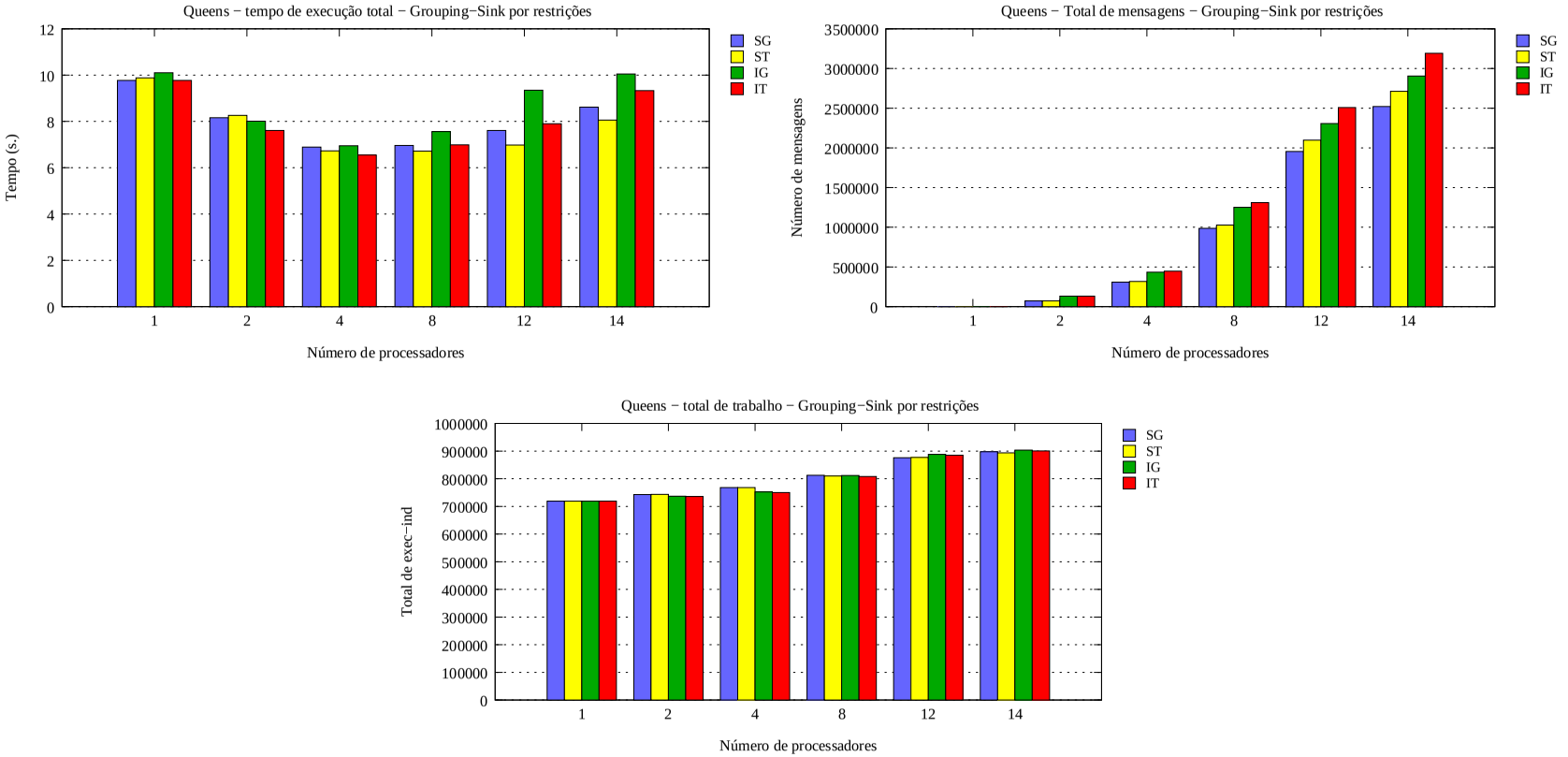


Figura D.16: *Queens - Grouping-Sink* por restrições

D.3 PBCSP (50-65)

As Figuras de D.17 a D.24 apresentam os resultados para PBCSP (50-65) de tempo de execução, total de trabalho e total de mensagens para cada particionamento.

Blocos por variáveis Pelo gráfico de tempo de execução da Figura D.17 podemos observar que o menor tempo de execução ficou com IG e IT. O total de mensagens para SG e ST é bem menor do que para IG e IT. O total de trabalho é praticamente o mesmo para todas as combinações. As combinações que mais se destacaram, considerando os 3 gráficos, foram SG e ST.

Blocos por *indexicals* Os menores tempos de execução foram obtidos com as combinações IG e IT (Figura D.18). Os menores totais de mensagens são de SG e ST. As combinações IG e IT apresentam as menores quantidades de trabalho. Observando os 3 gráficos as combinações que se destacaram foram IG e IT.

Blocos por restrições O menor tempo de execução ocorre com as combinações IG e IT (Figura D.19). Mas os menores totais de mensagens são de SG e ST. Os menores totais de trabalho são apresentados para IT e IG. Observando os 3 gráficos as combinações que se destacaram foram IG e IT.

Round-Robin **por variáveis** Os menores tempos de execução são apresentados por IG e IT (Figura D.20). Os menores totais de mensagens são apresentados por SG e ST. As combinações IG e IT apresentam a menor quantidade de trabalho. Observando os 3 gráficos as combinações que se destacaram foram IG e IT.

Round-Robin **por** *indexicals* O menor tempo de execução é apresentado por IG e IT (Figura D.13). O total de mensagens é menor para SG e ST. As combinações IG e IT apresentam a menor quantidade de trabalho. Observando os 3 gráficos as combinações que se destacaram foram IG e IT.

Round-Robin **por restrições** O menor tempo de execução é apresentado por IG e IT (Figura D.22). O total de mensagens é menor para SG e ST. As combinações IG e IT apresentam a menor quantidade de trabalho. Observando os 3 gráficos as combinações que se destacaram foram IG e IT.

Grouping-Sink **por** *indexicals* O menor tempo de execução foi apresentado pelas combinações IG e IT D.23. O menor número de mensagens foi para SG e IT. As combinações

PBCSP (50-65)										
Particion.	Variáveis			*Indexicals*			Restrições			Avalia
	Tempo	Trabalho	Mensagens	Tempo	Trabalho	Mensagens	Tempo	Trabalho	Mensagens	
Blocos	IG IT	SG ST IG IT	SG ST	IG IT	IG IT	SG ST	IG IT	IG IT	SG ST	**IG** **IT**
Round-Robin	IG IT	IG IT	SG ST	IG IT	IG IT	SG ST	IG IT	IG IT	SG ST	**IG** **IT**
Grouping-Sink	- -	- -	- -	IG IT	IG IT	SG ST	IG IT	IG IT	SG	**IG** **IT**

Tabela D.3: Resumo dos resultados para PBCSP (50-65)

IG e IT apresentam a menor quantidade de trabalho. Observando os 3 gráficos as combinações que se destacaram foram IG e IT.

Grouping-Sink **por restrições** O menor tempo de execução é apresentado por IG e IT (Figura D.24). Mas o menor total de mensagens é apresentado por SG. As combinações IG e IT apresentam a menor quantidade de trabalho. Observando os 3 gráficos as combinações que se destacaram foram IG e IT.

A Tabela D.3 apresenta um resumo dos resultados de todos os particionamentos para PBCSP (50-65).

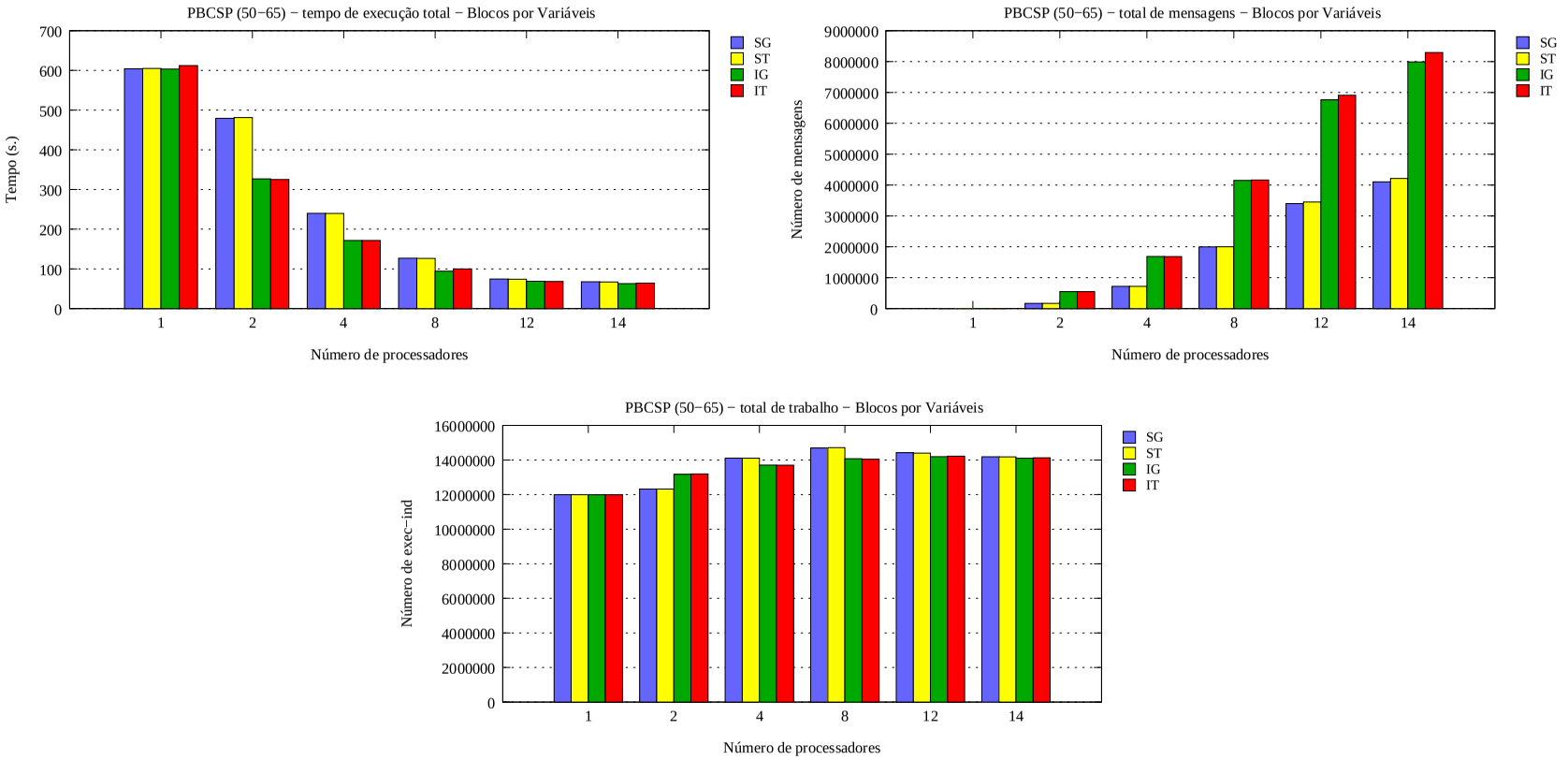


Figura D.17: PBCSP (50-65) - Blocos por variáveis

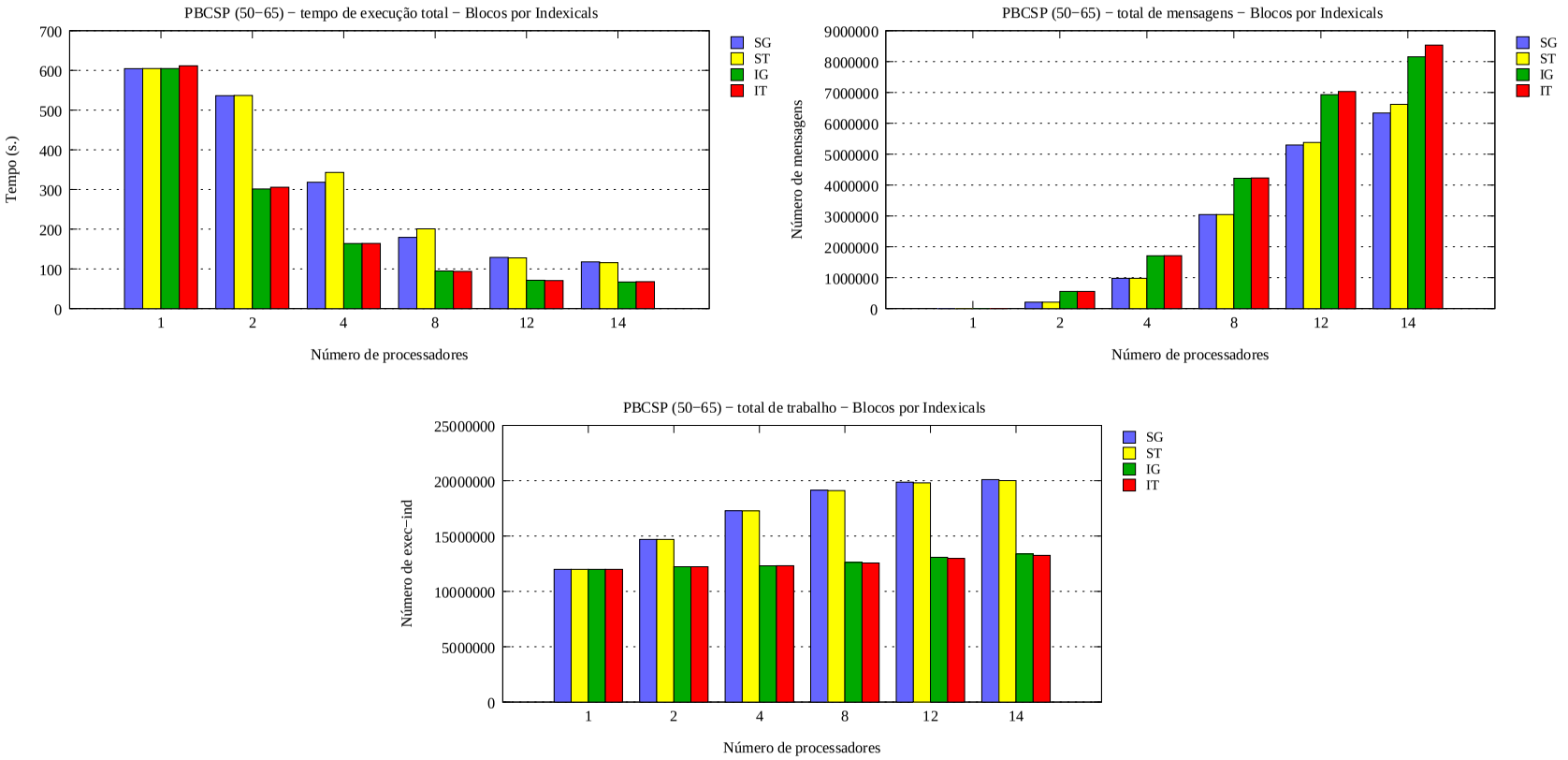


Figura D.18: PBCSP (50-65) - Blocos por *indexicals*

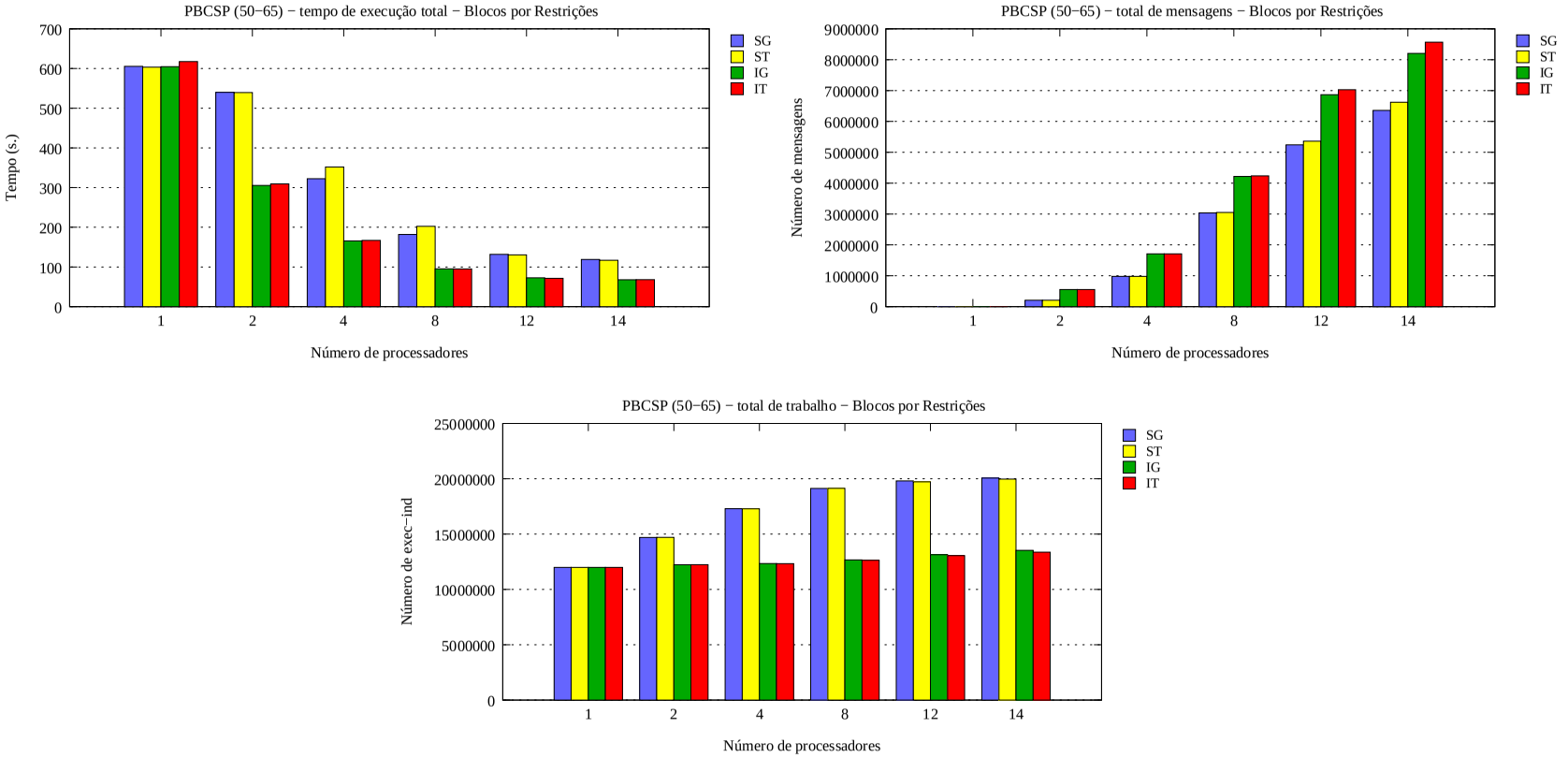


Figura D.19: PBCSP (50-65) - Blocos por restrições

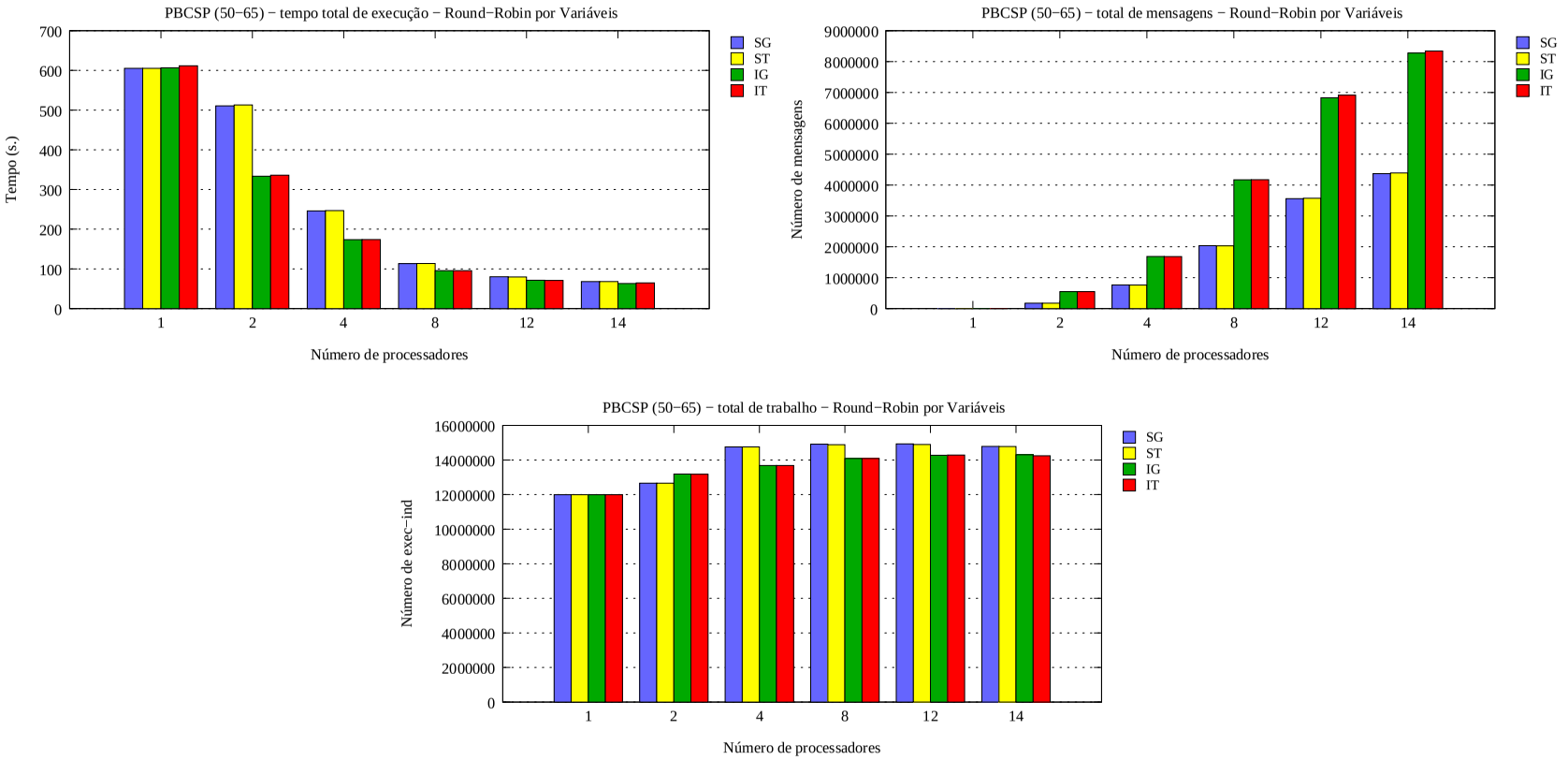


Figura D.20: PBCSP (50-65) - *Round-Robin* por variáveis

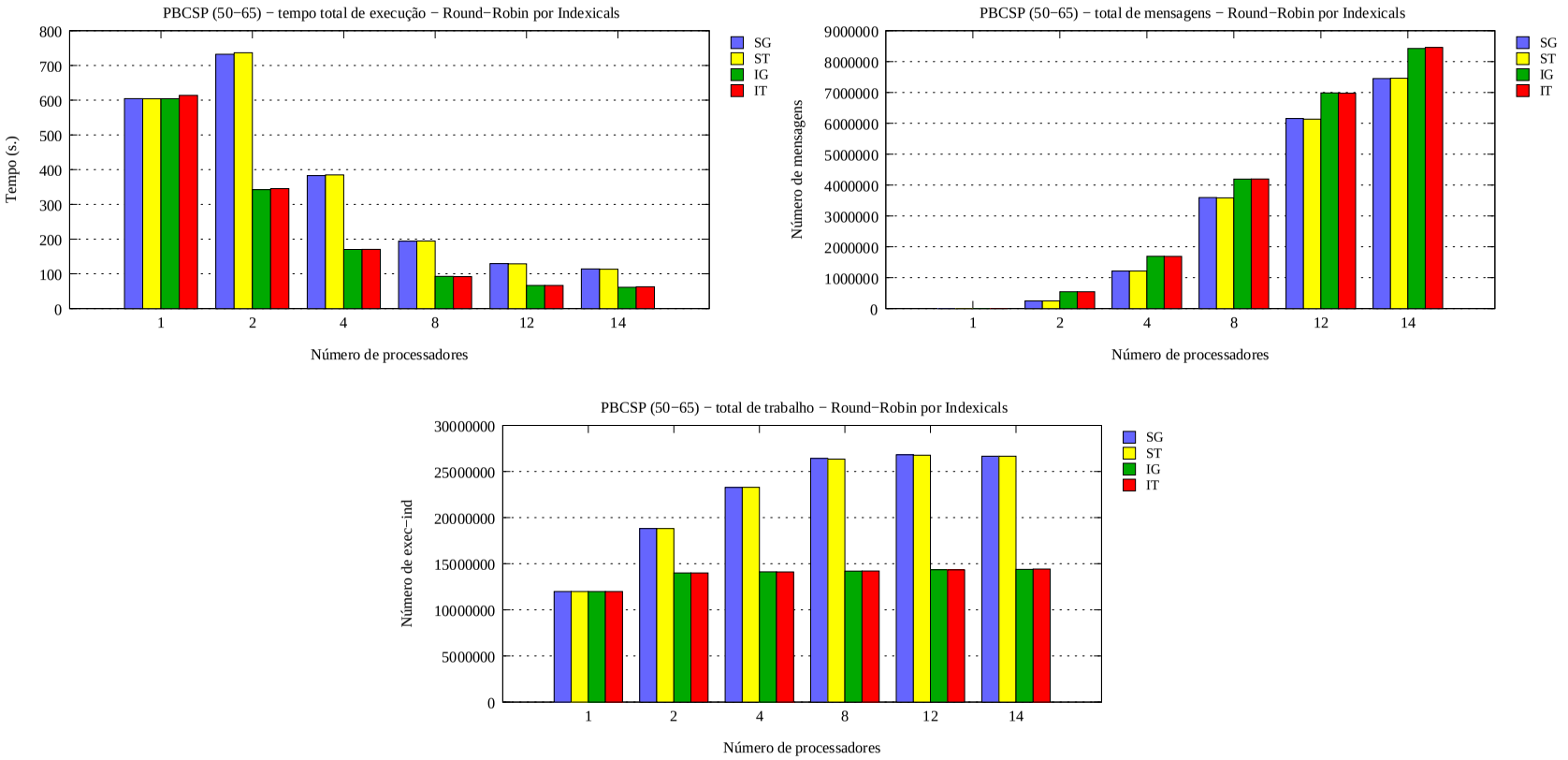


Figura D.21: PBCSP (50-65) - *Round-Robin* por *indexicals*

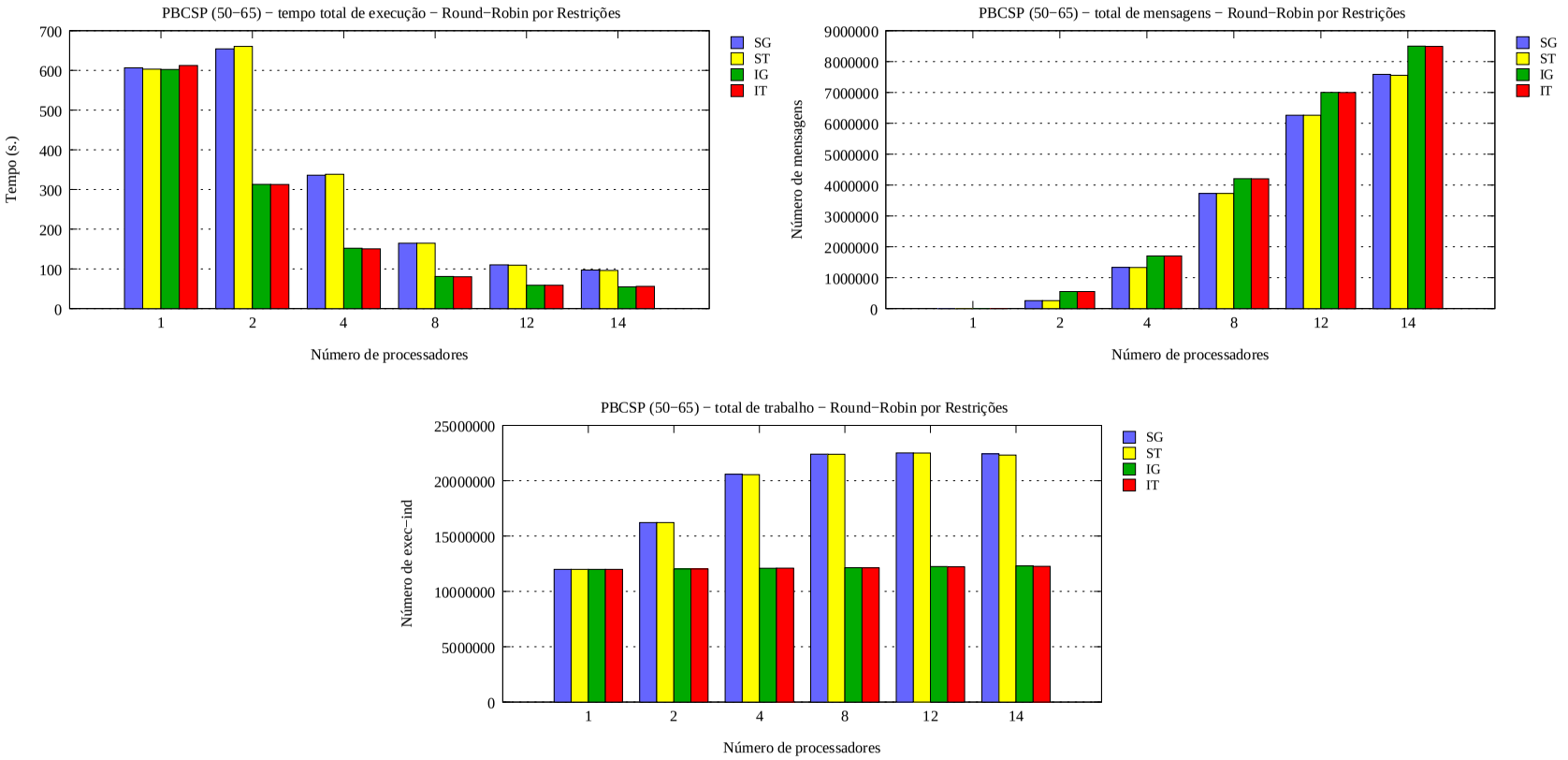


Figura D.22: PBCSP (50-65) - *Round-Robin* por restrições

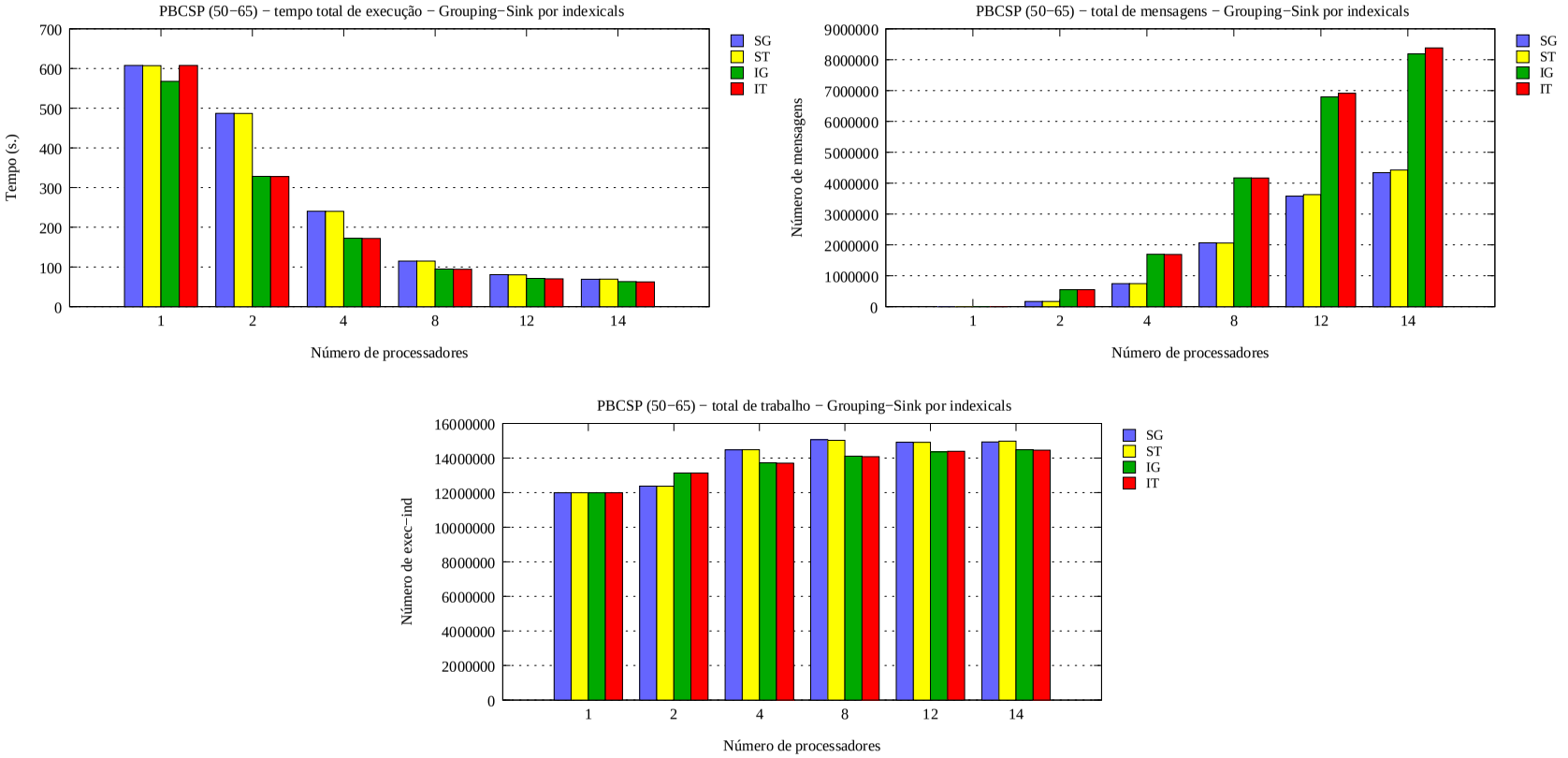


Figura D.23: PBCSP (50-65) - *Grouping-Sink* por *indexicals*

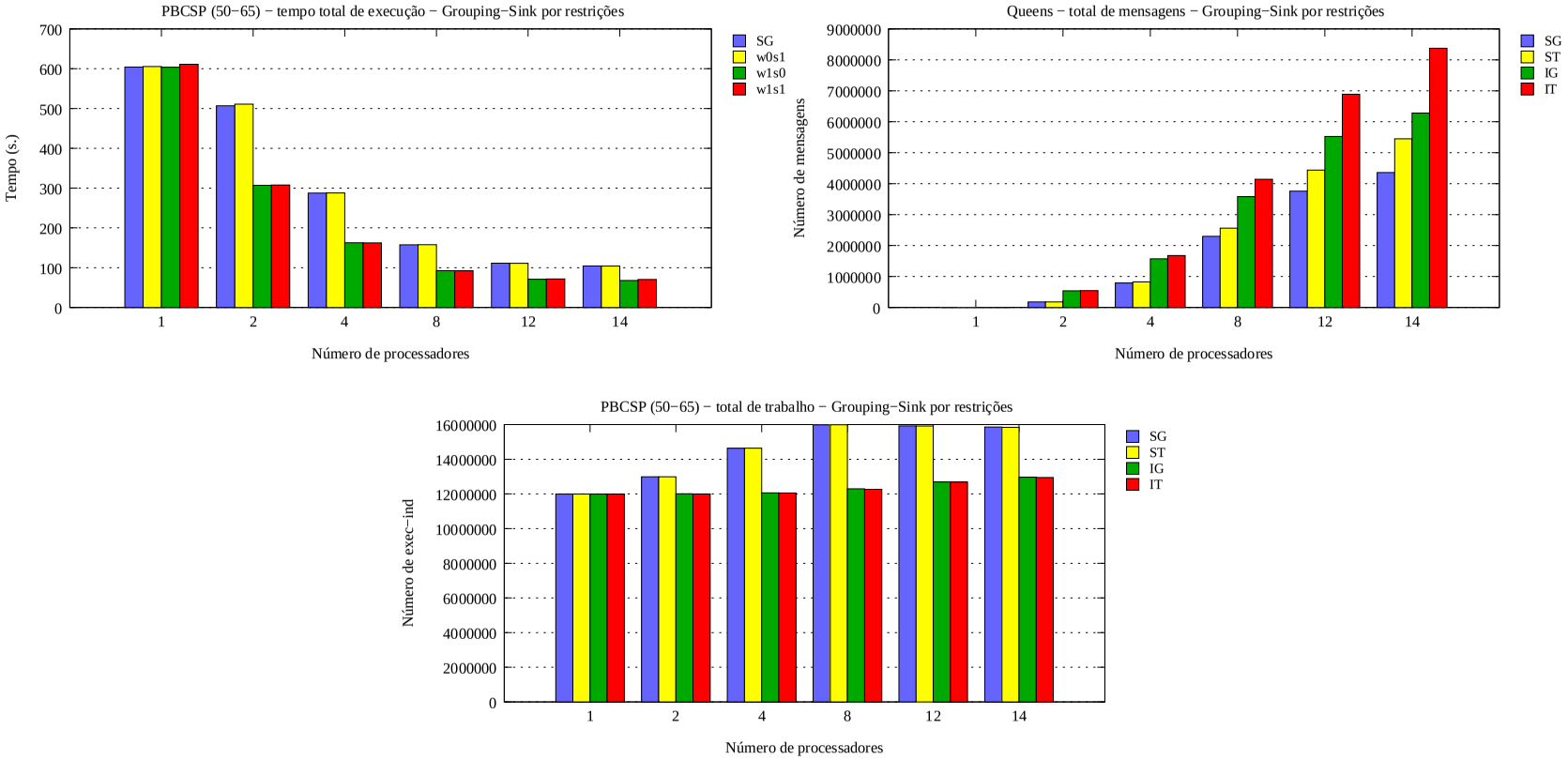


Figura D.24: PBCSP (50-65) - *Grouping-Sink* por restrições

D.4 PBCSP (100-75)

As Figuras de D.25 a D.32 apresentam os resultados de PBCSP (100-75).

Blocos por variáveis O gráfico de tempo de execução da Figura D.25 mostra que os menores tempos de execução são apresentados por IG e IT. Mas o menor total de mensagens é de SG e ST. As combinações IG e IT também apresentam a menor quantidade de trabalho. Observando os 3 gráficos, as combinações que mais se destacaram foram IG e IT.

Blocos por *indexicals* O gráfico de tempo de execução da Figura D.26 mostra que os menores tempos de execução são apresentados por IG e IT. No gráfico de total de mensagens percebemos que SG e ST apresentam a menor quantidade de mensagens. As combinações IG e IT também apresentam a menor quantidade de trabalho. Observando os 3 gráficos, as combinações que mais se destacaram foram IG e IT.

Blocos por restrições O gráfico de tempo de execução da Figura D.27 mostra que os menores tempos de execução são apresentados por IG e IT. No gráfico de total de mensagens percebemos que SG e ST apresentam a menor quantidade de mensagens. As combinações IG e IT também apresentam a menor quantidade de trabalho. Observando os 3 gráficos, as combinações que mais se destacaram foram IG e IT.

Round-Robin **por variáveis** As combinações IG e IT apresentam o menor tempo de execução no gráfico de tempo de execução da Figura D.28. SG e ST apresentam as menores quantidades de mensagens. As combinações IG e IT também apresentam a menor quantidade de trabalho. Observando os 3 gráficos, as combinações que mais se destacaram foram IG e IT.

Round-Robin **por** *indexicals* O gráfico de tempo de execução da Figura D.29 mostra que os menores tempos de execução são apresentados por IG e IT. SG e ST apresentam as menores quantidades de mensagens. As combinações IG e IT também apresentam a menor quantidade de trabalho. Observando os 3 gráficos, as combinações que mais se destacaram foram IG e IT.

Round-Robin **por restrições** Os menores tempos de execução são apresentados por IG e IT no gráfico de tempo de execução da Figura D.30). E o menor total de mensagens é apresentado por SG e ST até 8 processadores. Mas a diferença em total de mensagens

PBCSP (100-75)										
Particion. ~~Blocos~~	Variáveis			*Indexicals*			Restrições			Avalia
	Tempo	Trabalho	Mensagens	Tempo	Trabalho	Mensagens	Tempo	Trabalho	Mensagens	
	IG IT	IG IT	SG ST	IG IT	IG IT	SG ST	IG IT	IG IT	SG ST	**IG** **IT**
Round-Robin	IG IT	IG IT	SG ST	IG IT	IG IT	SG ST	IG IT	IG IT	SG ST	**IG** **IT**
Grouping-Sink	- IT	- IT	- ST	IG IT	IG IT	SG ST	IG IT	IG IT	SG ST	**IG** **IT**

Tabela D.4: Resumo dos resultados para PBCSP (100-75)

com SG e ST é muito pequena. Os menores totais de trabalho são apresentados por IG e IT.

Grouping-Sink **por** *indexicals* Os menores tempos de execução foram apresentados por IG e IT D.31. O menor número de mensagens foi para SG e ST. A menor quantidade de trabalho ocorre com IG e IT. De acordo com os 3 gráficos, as combinações que mais se destacaram foram IG e IT.

Grouping-Sink **por restrições** Os menores tempos de execução são apresentados por IG e IT (Figura D.32). Mas os menores totais de mensagens são de SG e IG. A menor quantidade de trabalho ocorre com IG e IT. De acordo com os 3 gráficos, as combinações que mais se destacaram foram IG e IT.

A Tabela D.4 apresenta um resumo dos resultados de todos os particionamentos para PBCSP (100-75).

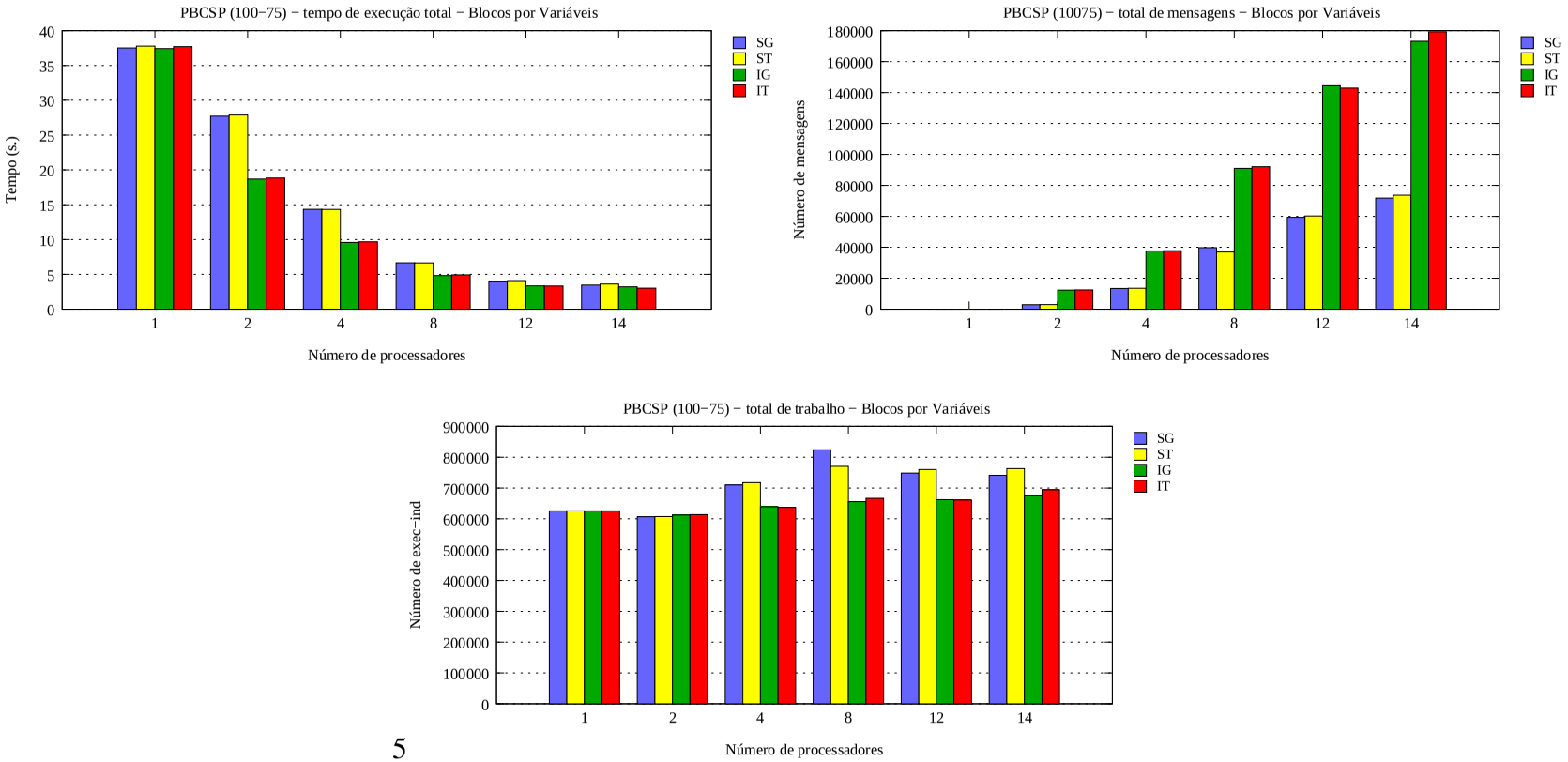


Figura D.25: PBCSP (100-75) - Blocos por variáveis

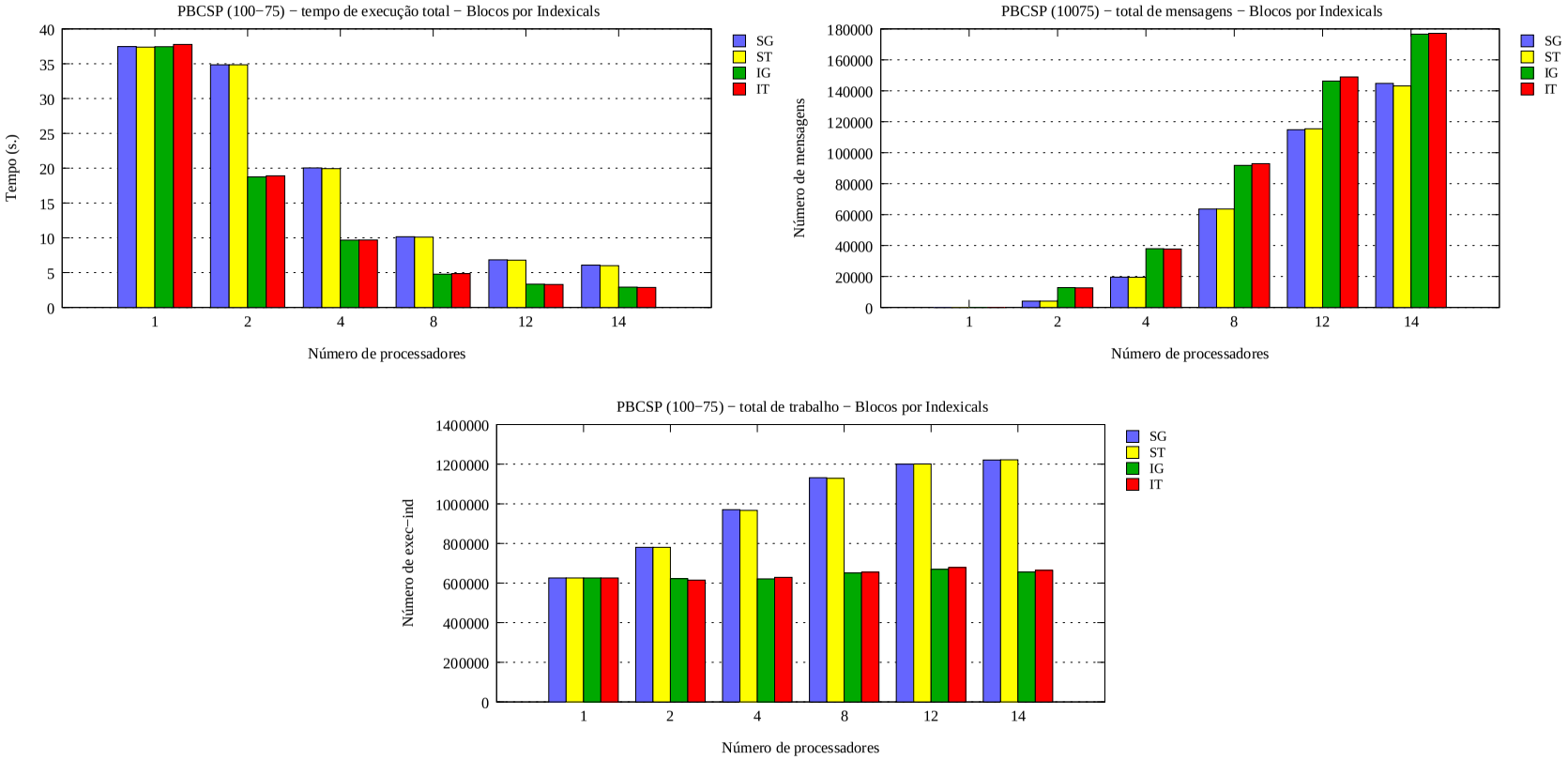


Figura D.26: PBCSP (100-75) - Blocos por *indexicals*

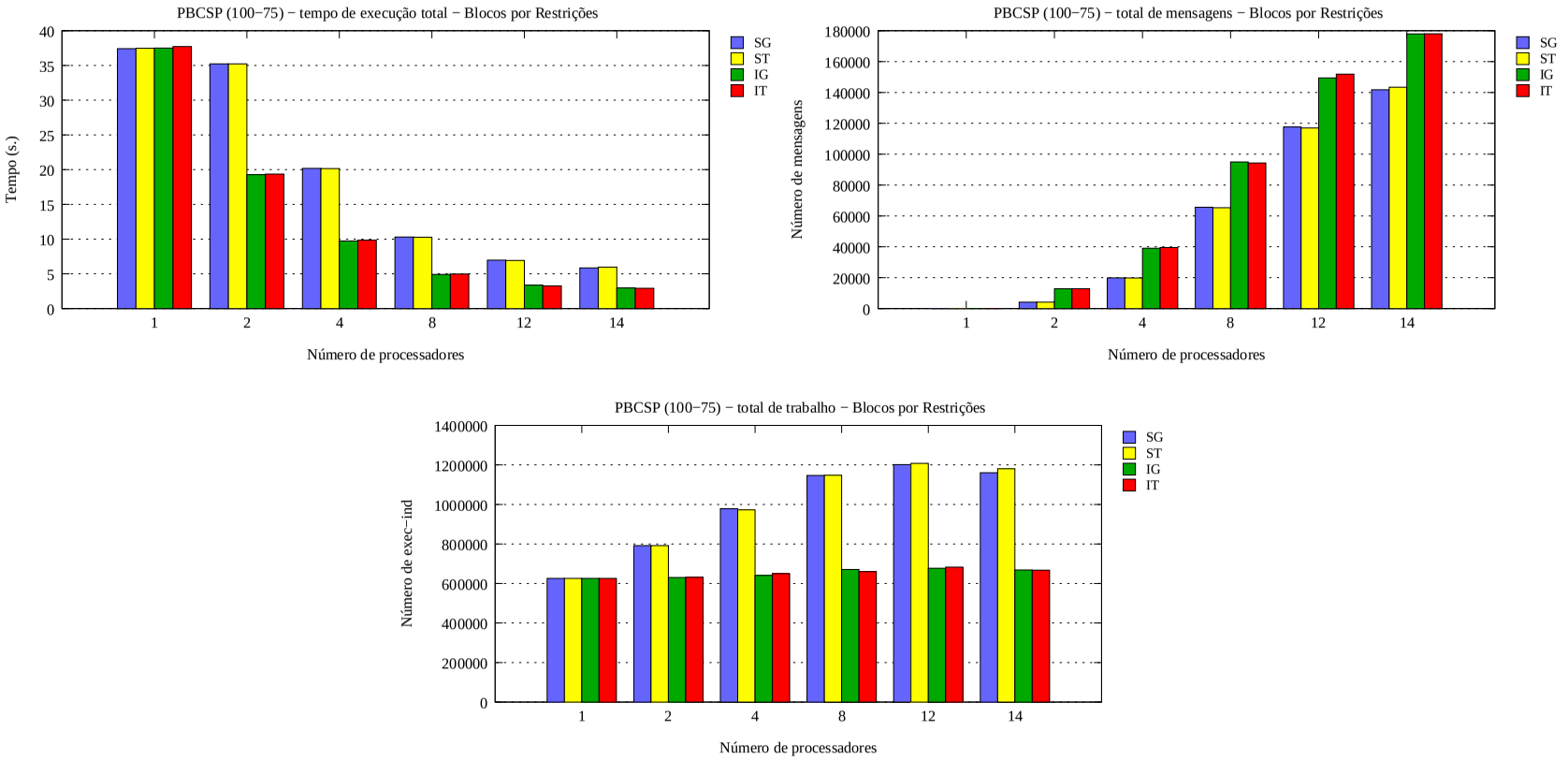


Figura D.27: PBCSP (100-75) - Blocos por restrições

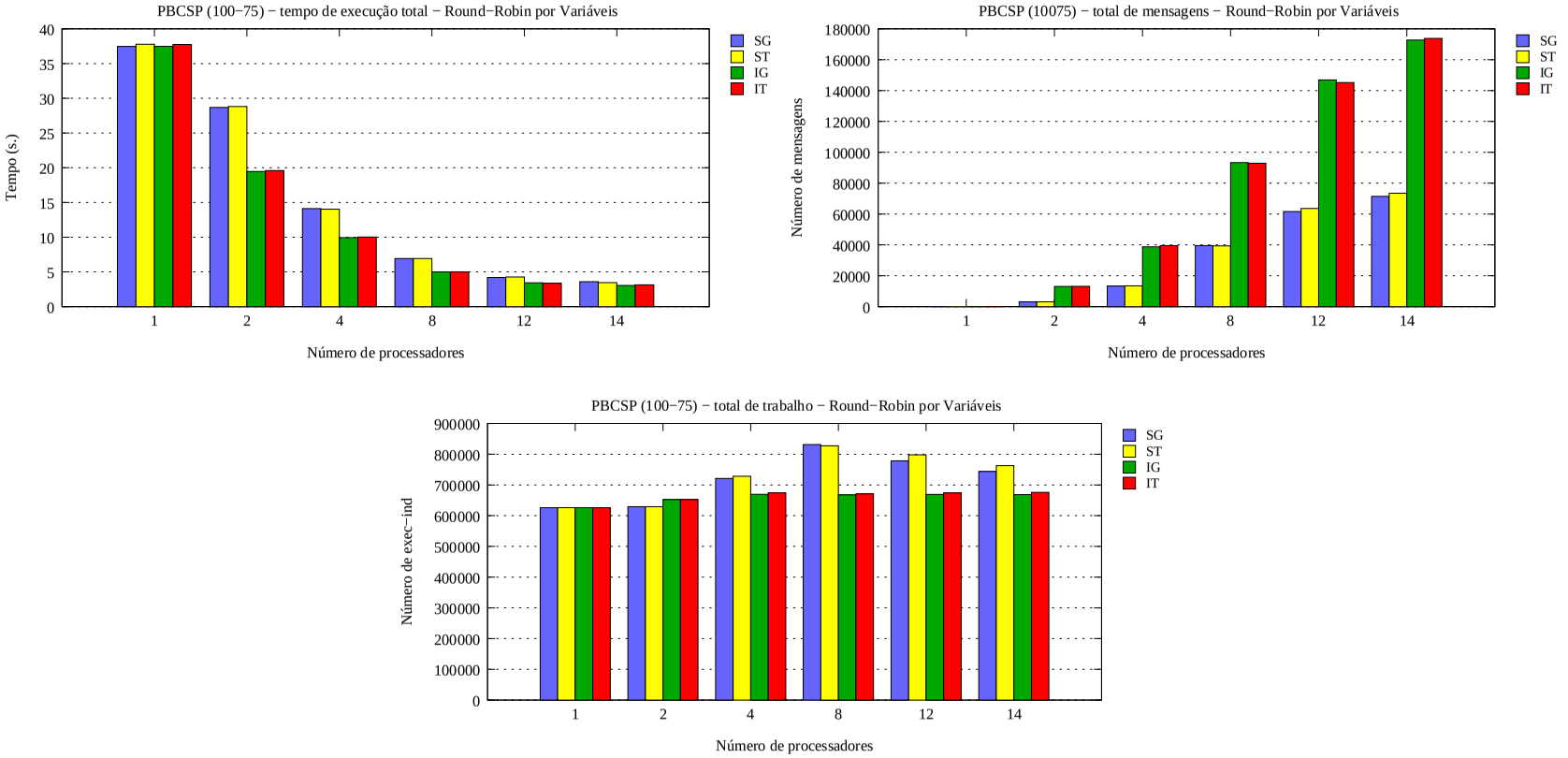


Figura D.28: PBCSP (100-75) - *Round-Robin* por variáveis

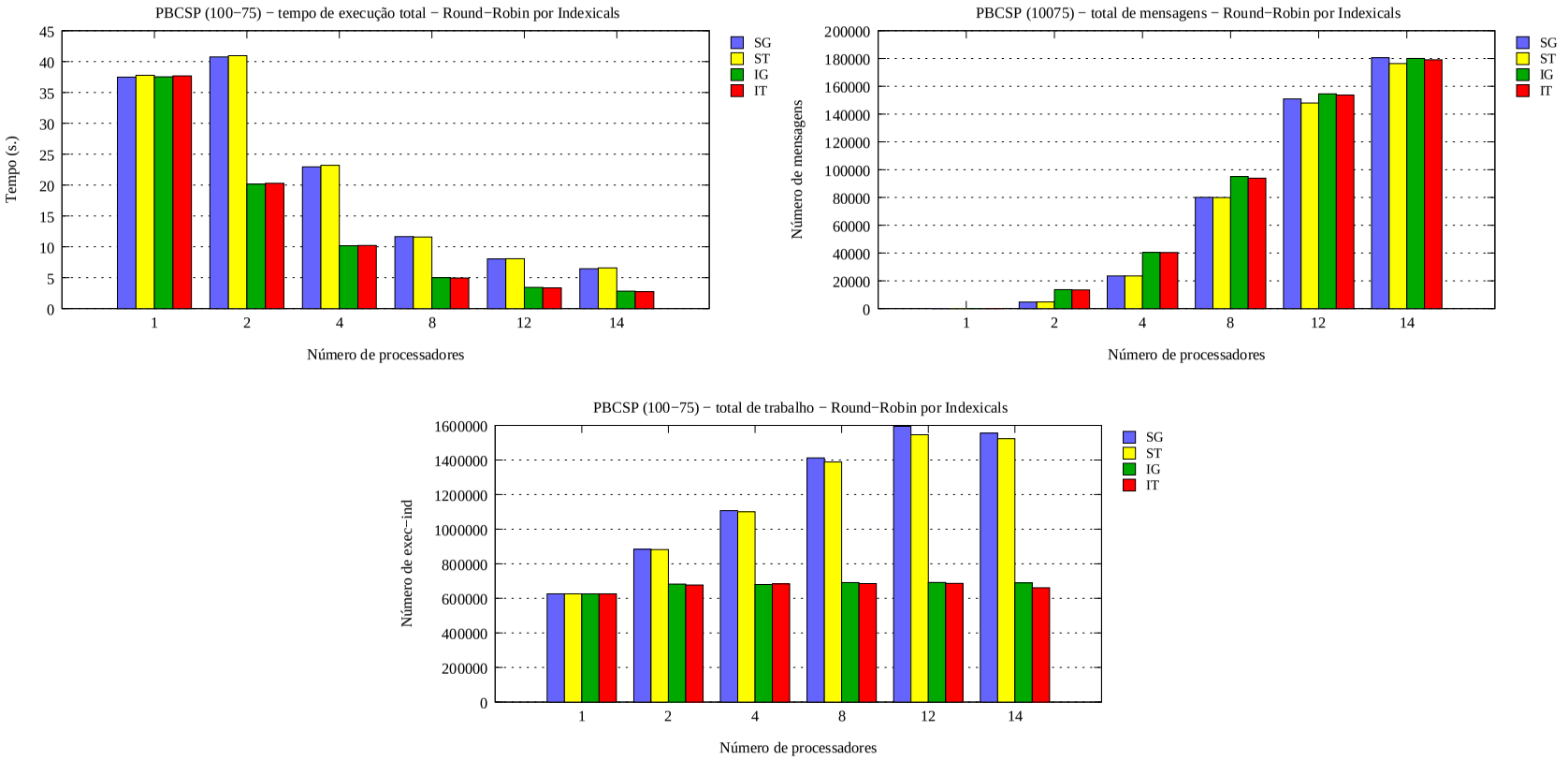


Figura D.29: PBCSP (100-75) - *Round-Robin* por *indexicals*

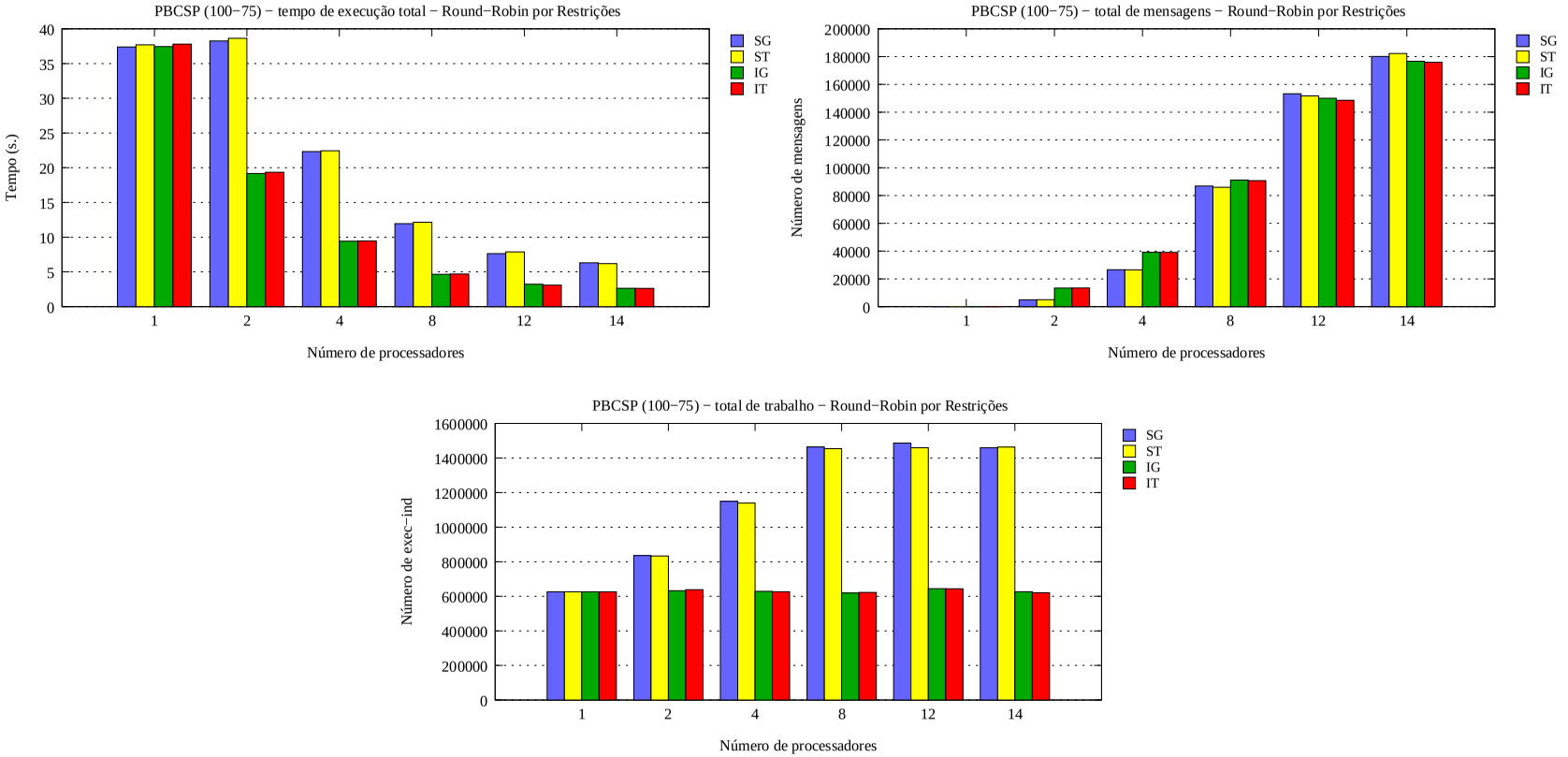


Figura D.30: PBCSP (100-75) - *Round-Robin* por restrições

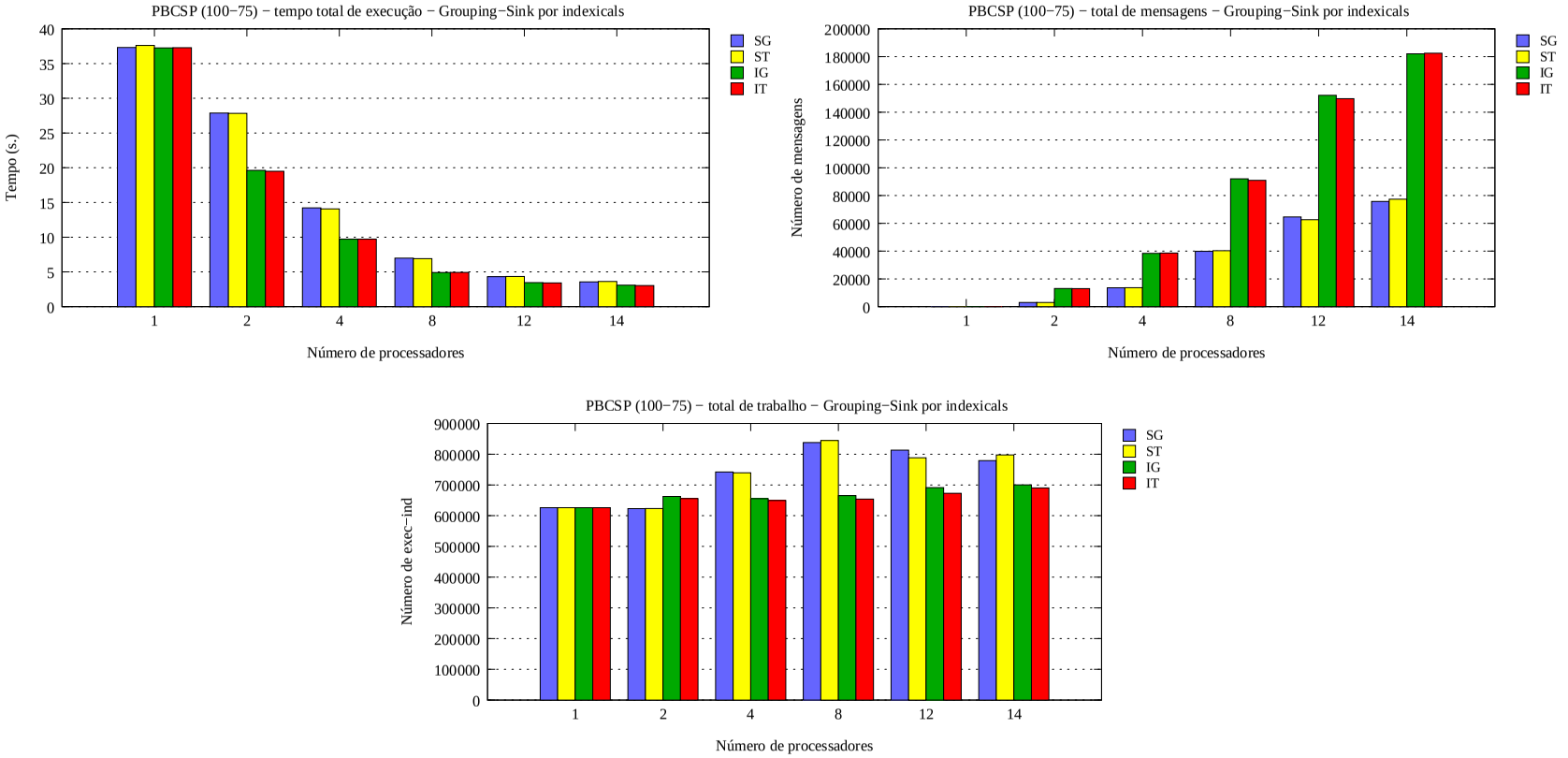


Figura D.31: PBCSP (100-75) - *Grouping-Sink* por *indexicals*

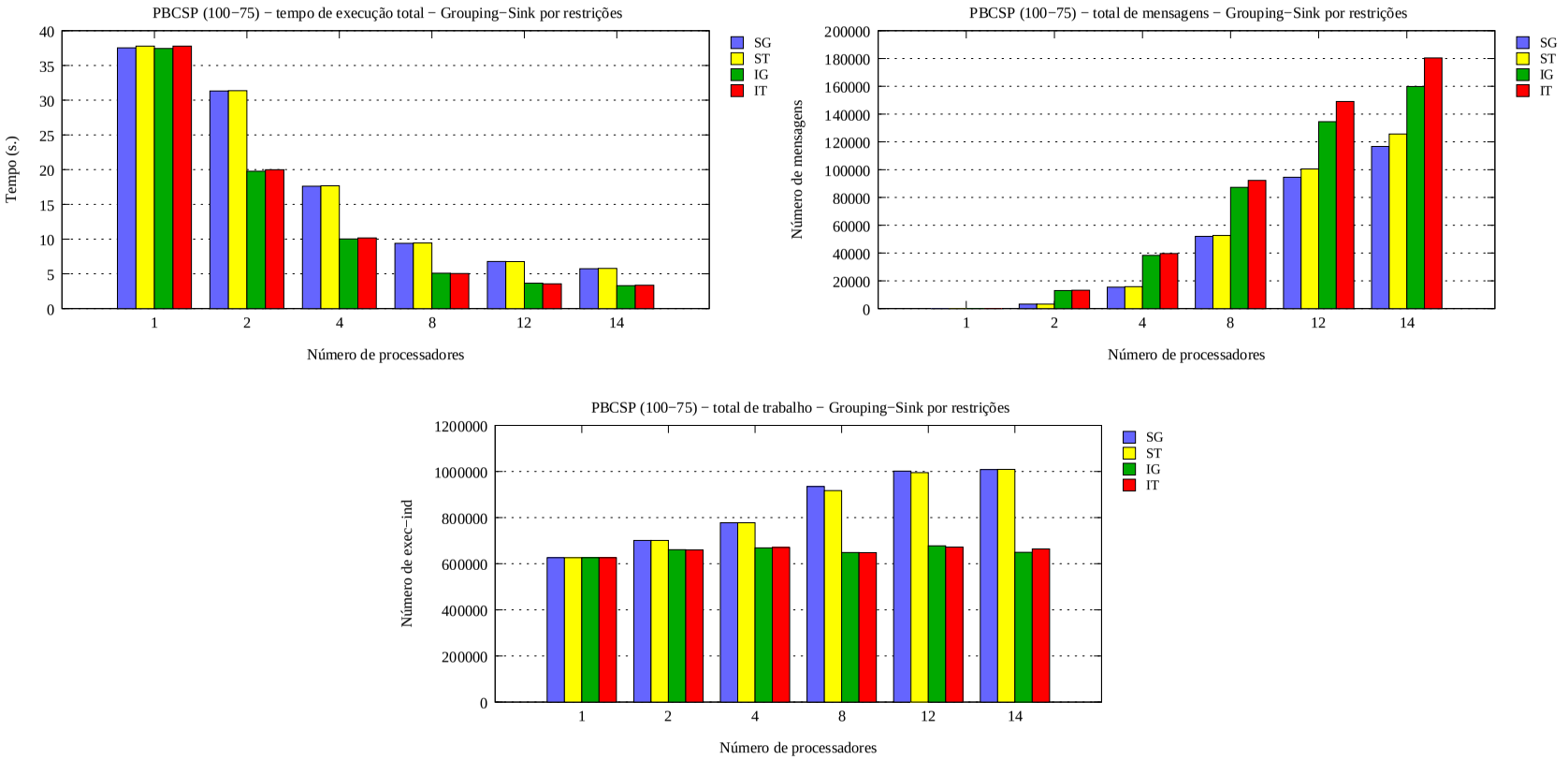


Figura D.32: PBCSP (100-75) - *Grouping-Sink* por restrições

D.5 *Sudoku*

As Figuras de D.33 a D.40 apresentam os resultados de *Sudoku* com todos os particionamentos.

Blocos por variáveis Nos gráficos da Figura D.33 SG e ST apresentaram os melhores resultados em tempo de execução, total de trabalho e total de mensagens.

Blocos por *indexicals* Na Figura D.34, SG, ST e IG apresentaram os melhores resultados em tempo de execução. Em total de mensagens, SG e IG apresentam a menor quantidade. Em total de trabalho SG e ST apresentam os menores valores. Observando os 3 gráficos SG é que mais se destaca.

Blocos por restrições SG, ST e IG apresentam os melhores resultados em tempo de execução, como apresentado na Figura D.35. Os menores totais de mensagens estão em SG e IG. As combinações SG e ST apresentam as menores quantidade de trabalho. Observando os 3 gráficos SG é a combinação que mais se destaca.

Round-Robin **por variáveis** No gráfico de tempo de execução da Figura D.36, os menores tempos são apresentados por SG e ST. Os menores totais de mensagens estão em SG e ST. Todas as combinações apresentam resultados similares em quantidade de trabalho. Observando os 3 gráficos SG e ST são as combinações que mais se destacaram.

Round-Robin **por** *indexicals* No gráfico de tempo de execução da Figura D.37, os menores tempos são apresentados por SG e ST. Os menores totais de mensagens estão em SG e ST. As combinações IG e IT apresentam resultados similares em quantidade de trabalho. Observando os 3 gráficos SG e ST são as combinações que mais se destacaram.

Round-Robin **por restrições** No gráfico de tempo de execução da Figura D.38, os menores tempos são apresentados por SG e ST. Os menores totais de mensagens estão em SG, ST e IG. Todas as combinações apresentam resultados similares em quantidade de trabalho. Observando os 3 gráficos SG e ST são as combinações que mais se destacaram.

Grouping-Sink **por** *indexicals* No gráfico de tempo de execução da Figura D.39, os menores tempos são apresentados por SG e ST. Os menores totais de mensagens estão em SG, ST e IG. Todas as combinações apresentam resultados similares em quantidade de trabalho. Observando os 3 gráficos SG e ST são as combinações que mais se destacaram.

Sudoku										
Particion.	Variáveis			*Indexicals*			Restrições			Avalia
	Tempo	Trabalho	Mensagens	Tempo	Trabalho	Mensagens	Tempo	Trabalho	Mensagens	
Blocos	SG ST	SG ST	SG ST	SG ST IG	SG ST	SG IG	SG ST IG	SG ST	SG IG	**SG**
Round-Robin	SG ST	SG ST IG	SG ST	SG ST	IG IT	SG ST	SG ST IT	SG ST IG IT	SG ST IG	**SG** **ST**
Grouping-Sink	- - - -	- - - -	- - - -	SG ST	IG ST	SG ST	SG ST IG	SG ST IG IT	SG	**SG** **ST**

Tabela D.5: Resumo dos resultados para *Sudoku*

Grouping-Sink **por restrições** No gráfico de tempo de execução da Figura D.37, os menores tempos são apresentados por SG e ST. Os menores totais de mensagens estão em SG, ST e IG. Até 8 processadores IG e IT apresentam a menor quantidade de trabalho. Para 8, 12 e 14 processadores todas as combinações apresentam resultados similares. Observando os 3 gráficos SG e ST são as combinações que mais se destacaram.

A Tabela D.5 apresenta um resumo dos resultados de todos os particionamentos para *Sudoku*. Podemos notar que SG e ST foram as combinações que mais se destacaram, mas SG mostra-se como uma boa combinação de parâmetros em todos os particionamentos.

D.6 Discussão

Os resultados que foram apresentados anteriormente referem-se a uma máquina de memória compartilhada. Mas estamos interessados em usar esta plataforma apenas para conhecer o comportamento do particionamento. Através do número de mensagens podemos inferir como seria o impacto do particionamento num sistema distribuído, onde geralmente o gargalo principal pode estar na rede de comunicação. Por isso, para escolher a combinação de parâmetros relativos às "mensagens" foi realizado este estudo. Percebemos que para *Arithmetic*, *Queens*, PBCSP (50-65), PBCSP (100-75) e *Sudoku* a maioria dos particionamentos apresentam o menor número de mensagens para SG. Por isso, escolhemos SG para realizar os experimentos, com todos os particionamentos em todas as aplicações, apresentados no Capítulo 7.

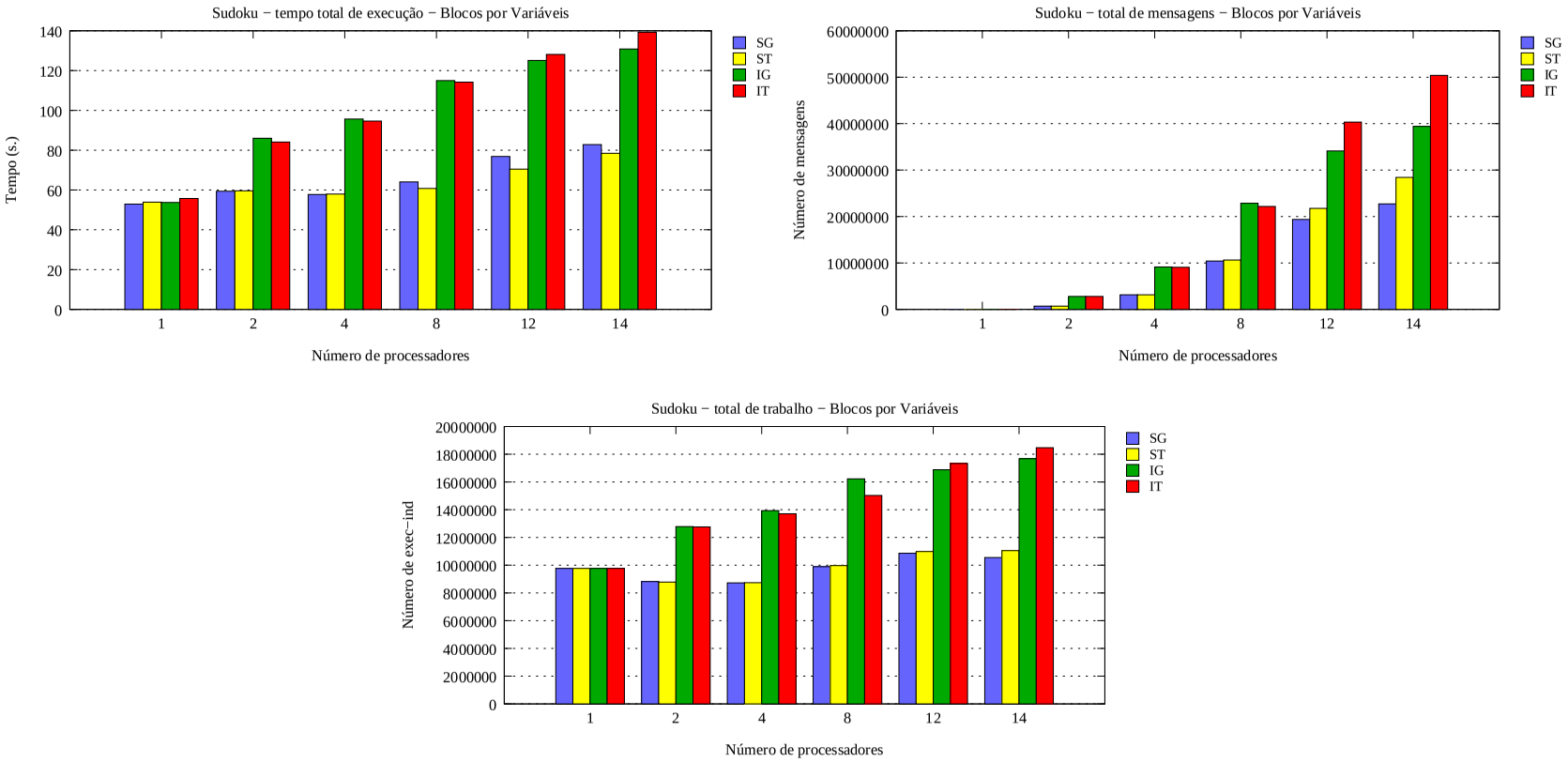


Figura D.33: *Sudoku* - Blocos por variáveis

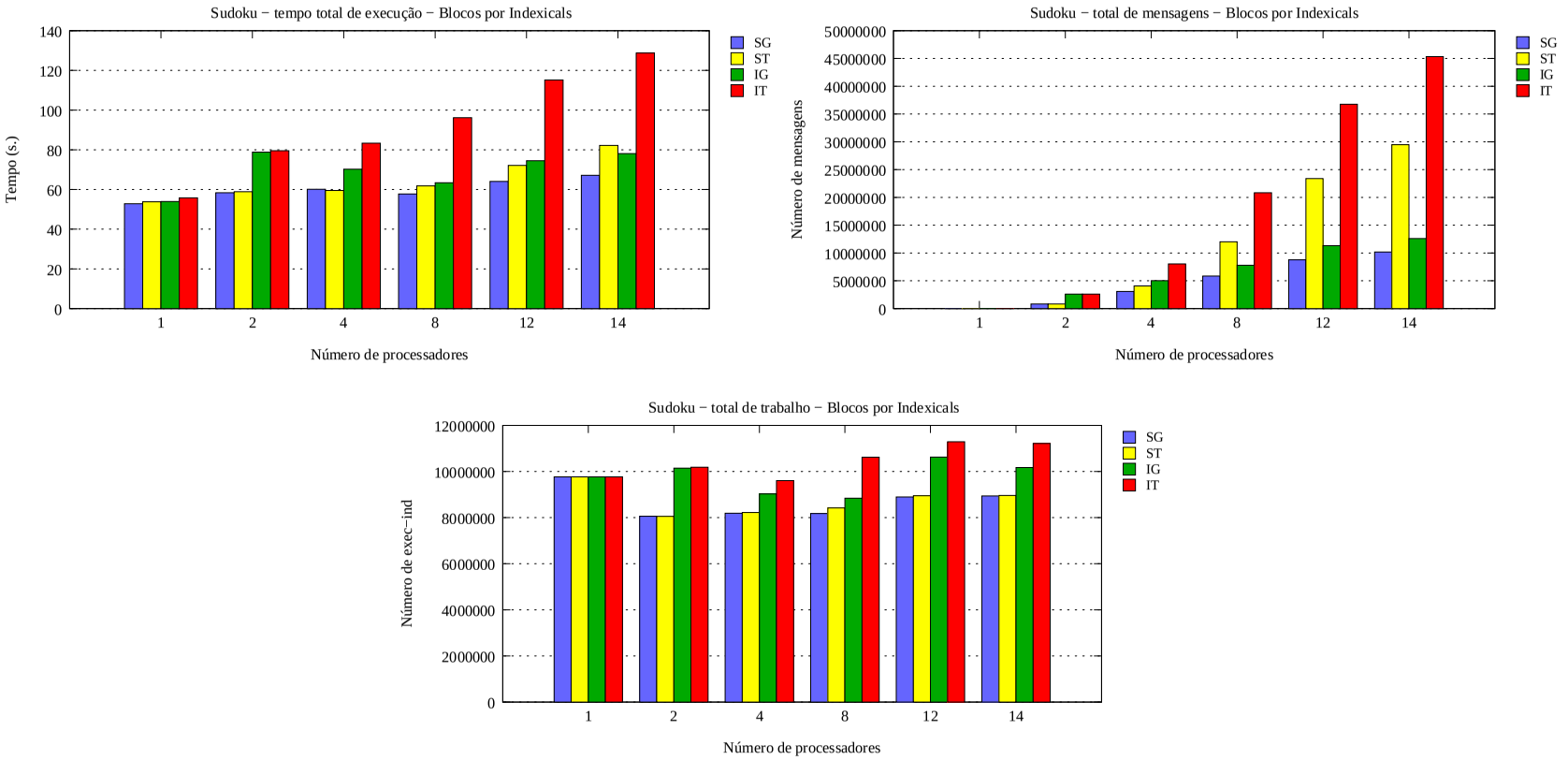


Figura D.34: *Sudoku* - Blocos por *indexicals*

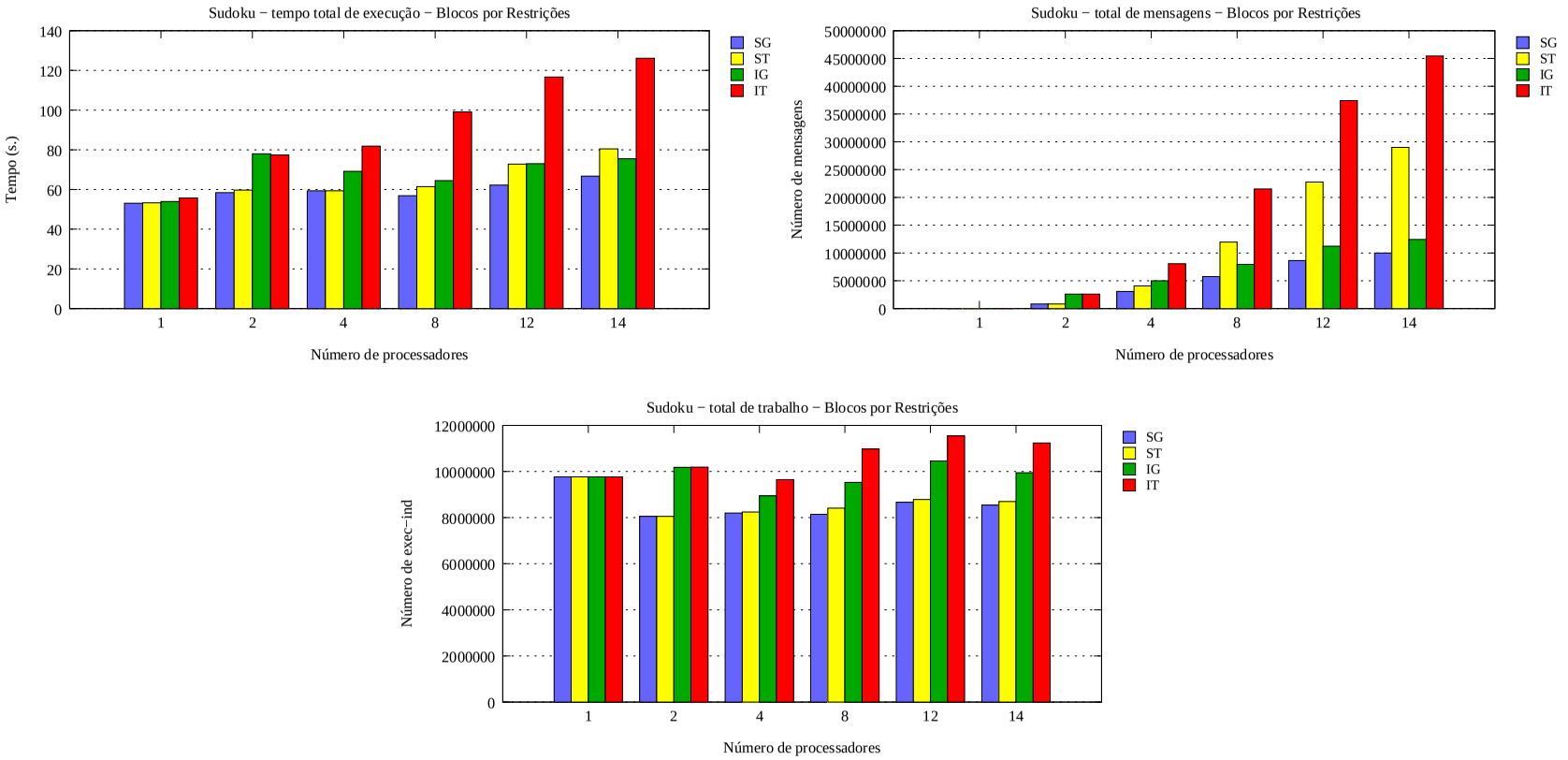


Figura D.35: *Sudoku* - Blocos por restrições

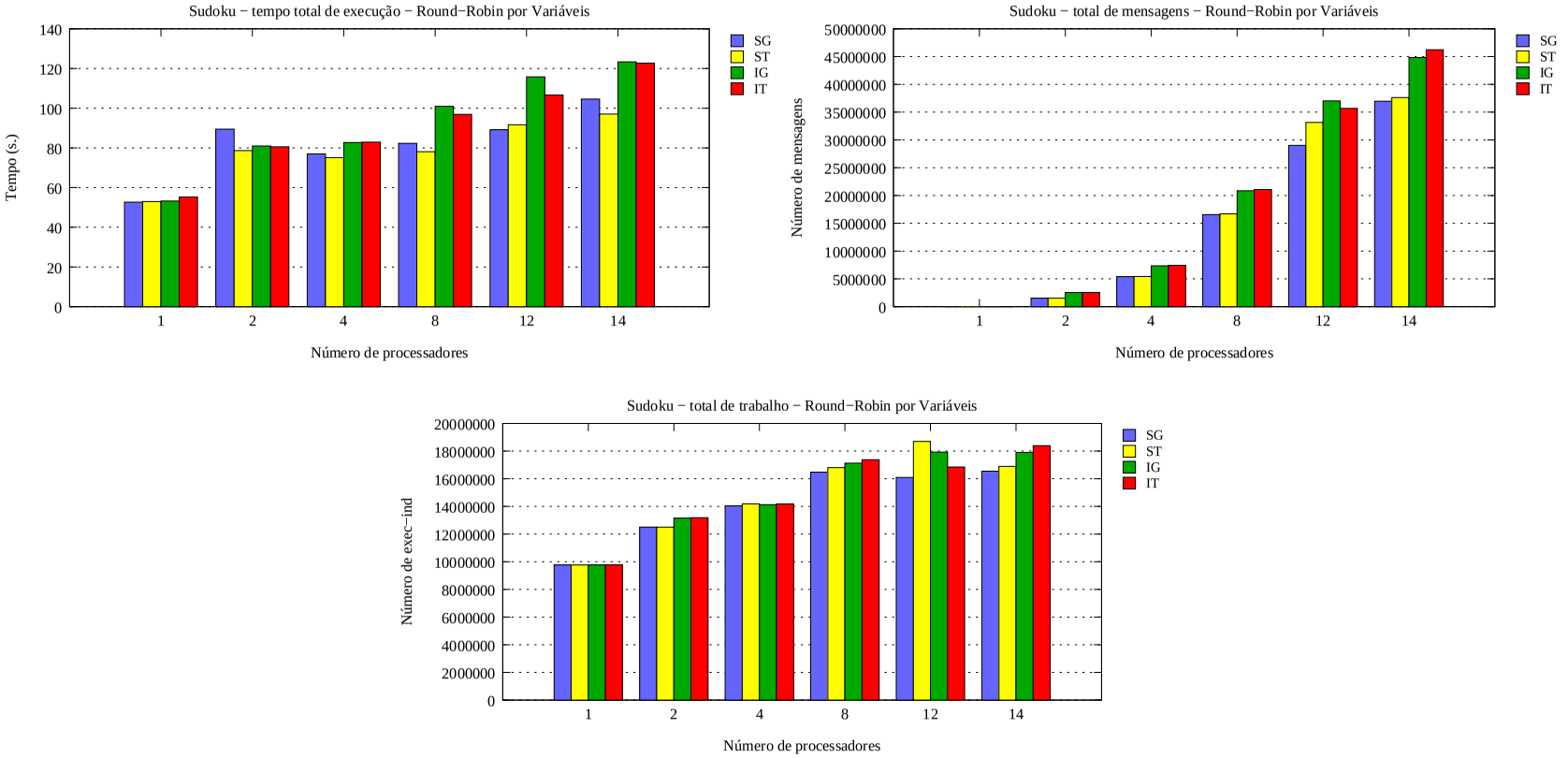


Figura D.36: *Sudoku* - *Round-Robin* por variáveis

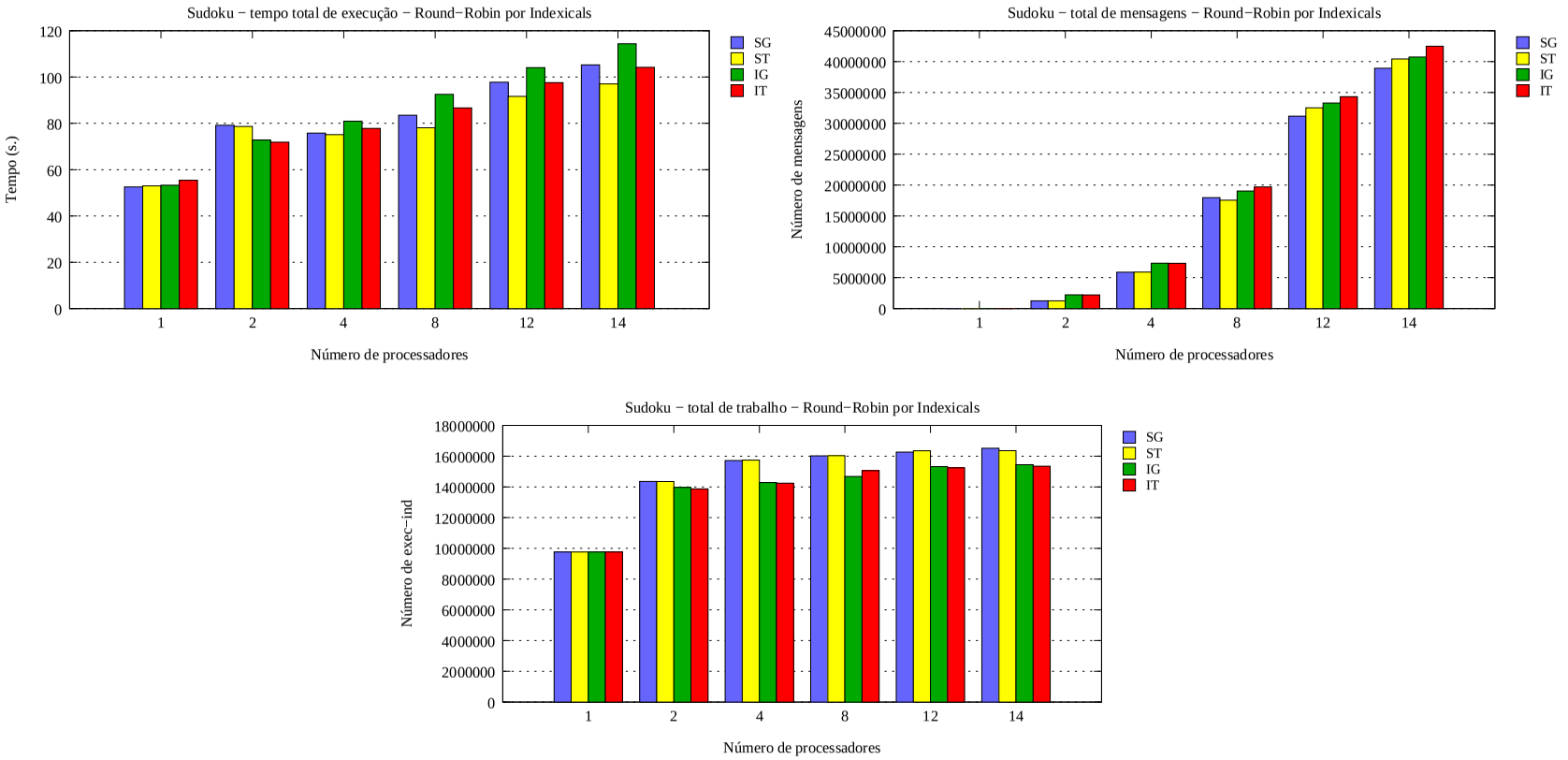


Figura D.37: *Sudoku - Round-Robin* por *indexicals*

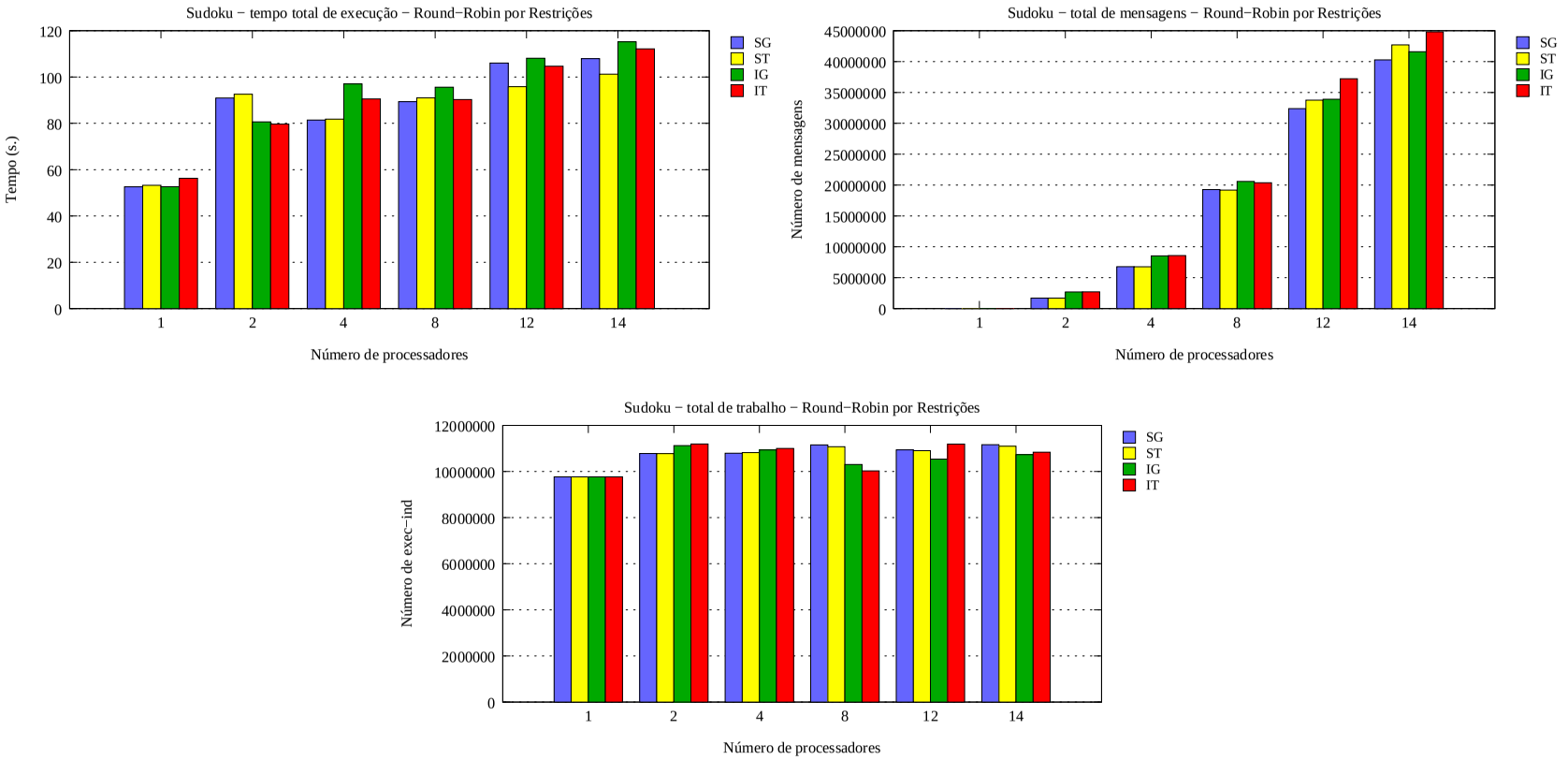


Figura D.38: *Sudoku - Round-Robin* por restrições

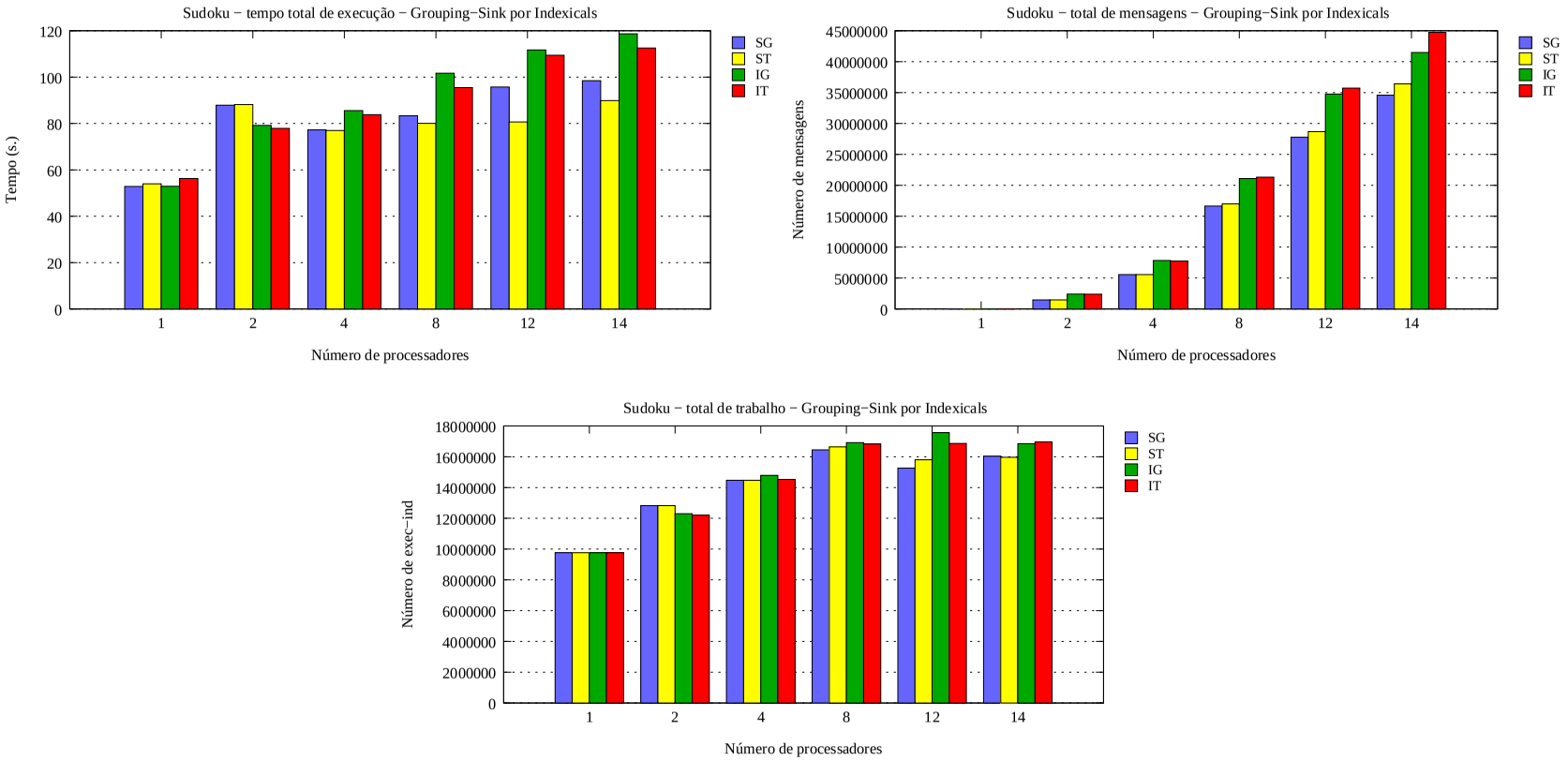


Figura D.39: *Sudoku - Grouping-Sink* por *indexicals*

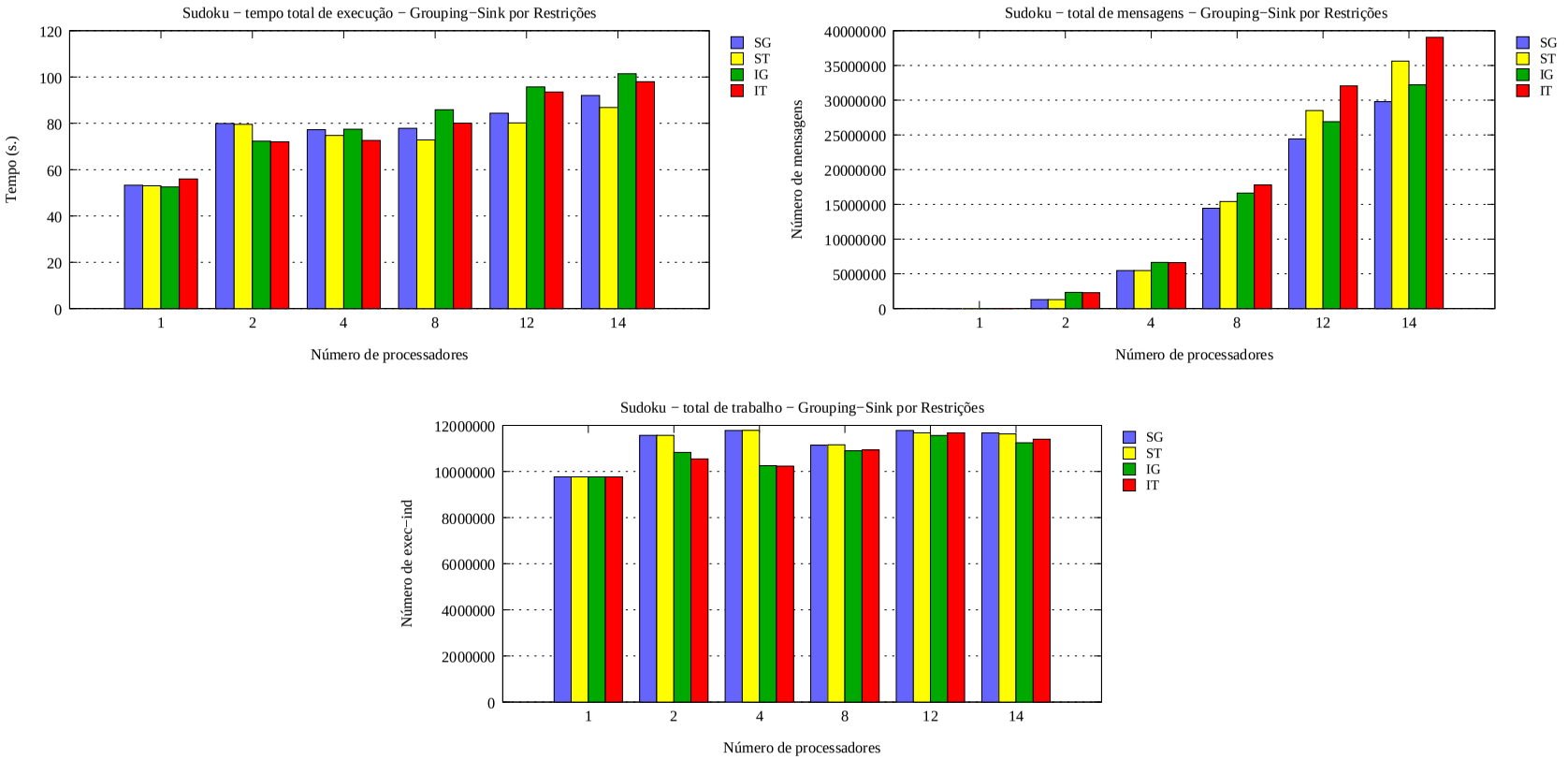


Figura D.40: *Sudoku - Grouping-Sink* por restrições

Referências Bibliográficas

[1] ANDINO, A. R., *CSOS User Manual (Draft Version)*, June 1997.

[2] ARANTES JR., G. M., *Orientações Acíclicas em Sistemas Distribuídos Anônimos e suas Aplicações no Compartilhamento de Recursos*. Dissertação de mestrado, COPPE/Sistemas – UFRJ, Rio de Janeiro, RJ, Brazil, 1999.

[3] ARANTES JR., G. M., FRANÇA, F. M. G., MARTINHON, C. A., "Algoritmos Randômicos para a Geração de Orientações Acíclicas em Sistemas Distribuídos", In: *Anais do Simpósio Brasileiro de Pesquisa Operacional (XXXIV SBPO)*, (RJ, Brazil), pp. 1–12, Novembro 2002.

[4] BARBOSA, V. C., *An Introduction to Distributed Algoritms*. London, England, The MIT Press, 1996.

[5] BARBOSA, V. C., *An Atlas of Edge-Reversal Dynamics*. London, UK, Chapman and Hall/CRC, 2000.

[6] BARBOSA, V. C., FERREIRA, R. G., "On the Phase Transitions of Graph Coloring and Independent Sets", *Physica A: Statistical Mechanics and its Applications*, v. 343, pp. 401–423, 2004.

[7] BARBOSA, V. C., GAFNI, E., "Concurrency in Heavily Loaded Neighborhood-Constrained Systems", *ACM Transactions on Programming Languages and Systems*, v. 11, pp. 562–584, October 1989.

[8] BERGE, C., *Graphs and Hypergraphs*. North-Holland, Amsterdam, The Netherlands, 1976.

[9] BESSIERE, C., "Arc-consistency and Arc-consistency Again", *Artificial Intelligence*, v. 65, pp. 179–190, 1994.

[10] BESSIERE, C., FREUDER, E. C., "Using Constraint Metaknowledge to Reduce Arc-consistency Computation", *Artificial Intelligence*, v. 107, pp. 125–148, January 1999.

[11] BESSIERE, C., MAESTRE, A., BRITO, I., *et al.*, "Asynchronous Backtracking without Adding Links: a New Member in the ABT Family", *Artificial Intelligence*, v. 161, pp. 7–24, 2005.

[12] BESSIERE, C., MAESTRE, A., MESEGUER, P., "Distributed Dynamic Backtracking", *Proceedings of the IJCAI'01 Workshop on Distributed Constraint Reasoning*, pp. 9–16, July 2001.

[13] BESSIERE, C., YAP, R. H. C., RéGIN, J.-C., *et al.*, "An Optimal Coarse-Grained Arc-Consistency Algorithm", *Artificial Intelligence*, v. 165, n. 2, pp. 165–185, 2005.

[14] BLIEK, C., NEVEU, B., TROMBETTONI, G., "Using Graph Decomposition for Solving Continuous CSPs", In: *Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming*, v. 1520 of *Lecture Notes in Computer Science*, (London, UK), pp. 102–116, Springer-Verlag, October 26-30 1998.

[15] BRITO, I., MESEGUER, P., "Distributed Forward Checking", In: *Proceedings in Principles and Practice of Constraint Programming (CP 2003). 9th International Conference, Kinsale, Ireland*, pp. 801–806, Sep.-Oct. 2003.

[16] BRITO, I., MESEGUER, P., "Distributed Stable Marriage Problem", In: *Proceedings of The Sixth International Workshop in Distributed Constraint Resoaning (DCR-05), Edinburgh,Scotland*, pp. 135–147, July 2005.

[17] CALABRESE, A., FRANÇA, F. M. G., "Randomized Distributed Primer for Updating Control of Anonymous ANNs", In: *Proceedings of the Intl. Conf. on Artificial Neural Networks (ICANN)*, (Sorrento, Italy), pp. 26–29, May 1994.

[18] CALABRESE, A., FRANÇA, F. M. G., "Distributed Computing on Neighbourhood Constrained Systems", In: *Proceedings of the 3rd. Intl. Conf. On Principles Of Distributed Systems*, (Hanoi, Vietnam), pp. 219–234, October 1999.

[19] CHAMBERLAIN, B. L., "Graph Partitioning Algorithms for Distributing Workloads of Parallel Computations", Tech. Rep. TR-98-10-03, Univ. of Washington, Dept. of Computer Science & Engineering, October 1998.

[20] CODOGNET, P., DIAZ, D., "Compiling Constraints in CLP(FD)", *Journal of Logic Programming*, v. 27, n. 3, pp. 185–226, 1996.

[21] COQUETEL, *Sudoku - Livro Oficial*. Rio de Janeiro, Ediouro, 2005.

[22] DECHTER, R., *Constraint Processing*. 1st ed. San Francisco, CA, Morgan Kaufmann, 2003.

[23] DONGEN, M., "AC-3d an Efficient Arc-consistency Algorithm with a Low Space-Complexity", *Principles and Practice of Constraint Programming, CP'2002 (Procs.), Pascal Van Hentenryck (Ed.), Lecture Notes in Computer Science - Springer*, pp. 755–760, September 2002.

[24] FERRIS, M. C., MANGASARIAN, O. L., "Parallel Constraint Distribution", *SIAM Journal on Optmization*, v. 1, n. 4, pp. 487–500, 1991.

[25] FIEDLER, M., "A Property of Eigenvectors of Nonnegative Symmetric Matrices and Its Application to Graph Theory", *Czechoslovak Math. J.*, v. 25, n. 100, pp. 619–633, 1975.

[26] FRANÇA, F. M. G., FARIA, L., "Optimal mapping of neighbourhood-constrained systems", In: *Workshop on Parallel Algorithms for Irregularly Structured Problems*, pp. 165–170, 1995.

[27] FRANÇA, F. M. G., MUYLAERT FILHO, J. A., PAILLARD, G. A. L., "Uma Proposta de um Escalonador para Gamma", In: *Anais do II Workshop em Sistemas Computacionais de Alto Desempenho (em conjunto com o SBAC-PAD'2001)*, (Pirenópolis, GO), pp. 47–54, Setembro 2001.

[28] GAFNI, E. M., BERTSEKAS, D. P., "Distributed Algorithms for Generating Loop-free Routes in Networks with Frequently Changing Topology", *IEEE Transactions on Communications*, v. 1, pp. 11–18, January 1981.

[29] GAREY, M., JOHNSON, D., STOCKMEYER, L., "Some Simplified NP-complete Graph Problems", *Theoretical Computer Science*, v. 1, n. 3, pp. 237–267, 1976.

[30] GEORGE, A., LIU, J., *Computer Solution of Large Sparce Positive Definite Systems*. Prentice Hall, Englewood Cliffs, 1981.

[31] GINSBERG, M. L., "Dynamic Backtracking", *Journal of Artificial Intelligence Research*, v. 1, pp. 25–46, 1993.

[32] GOLDSCHMIDT, O., HOCHBAUM, D. S., "A Polynomial Algorithm for the k-cut Problem for Fixed k", *Mathematics of Operations Research*, v. 19, pp. 24–37, February 1994.

[33] HAMADI, Y., "Interleaved Backtracking in Distributed Constraint Networks", *International Journal on Artificial Intelligence Tools*, v. 11, n. 2, pp. 167–188, 2002.

[34] HAMADI, Y., BESSIERE, C., QUINQUETON, J., "Backtracking in Distributed Constraint Networks", In: *Proceedings of the 13th. European Conference on Artificial Intelligence*, (Brighton, UK), pp. 219–223, 1998.

[35] HENDRICKSON, B., "Load Balancing Fictions, Falsehoods and Fallacies", *Appl. Math. Modelling*, v. 25, pp. 99–108, 2000.

[36] HENDRICKSON, B., KOLDA, T. G., "Graph Partitioning Models for Parallel Computing", *Parallel Computing*, v. 26, n. 12, pp. 1519–1534, 2000.

[37] HENDRICKSON, B., LELAND, R., "Multidimensional Spectral Load Balancing", In: *Proceeding of 6th SIAM Conf. Parallel Proc. Sci. Comput., Sandia National Laboratories*, pp. 953–961, January 1993.

[38] HENDRICKSON, B., LELAND, R., "An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations", *SIAM Journal on Scientific Computing*, v. 16, n. 2, pp. 452–469, 1995.

[39] HENDRICKSON, B., LELAND, R., "The Chaco User's Guide Version 2.0", Technical Report SAND95-2344, Sandia National Laboratories, Albuquerque, 1995.

[40] HENTENRYCK, P. V., DEVILLE, Y., TENG, C., "A Generic Arc-consistency Algorithm and its Specializations", *Artificial Intelligence*, v. 57, pp. 291–321, 1992.

[41] KUMAR, V., "Algorithms for Constraint Satisfaction Problems: A Survey", *Artificial Intelligence Magazine*, v. 13, n. 1, pp. 32–44, 1992.

[42] LUO, Q. Y., HENDRY, P. G., BUCHANAN, J. T., "Heuristic Search for Distributed Constraint Satisfaction Problems", Research Report KEG-6-93, Department of Computer Science. University of Strathclyde, 1993.

[43] MACKWORTH, A. K., "Consistency in Networks of Relations", *Artificial Intelligence*, v. 8, n. 1, pp. 99–118, 1977.

[44] MACKWORTH, A. K., FREUDER, E. C., "The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems", *Artificial Intelligence*, v. 25, pp. 65–74, 1985.

[45] MARRIOT, K., STUCKEY, P. J., *Programming with constraints: An Introduction*. MIT Press, 1998.

[46] MEISELS, A., "CP04 Tutorial: Distributed Constraints Satisfaction Algorithms, Performance, Communication". Tutorial presented at Tenth International Conference on Principles and Practice of Constraint Programming, Toronto, 2004.

[47] MEISELS, A., RAZGON, I., "Distributed Forward Checking with Dynamic Ordering", In: *Proceedings of the Workshop on Cooperative Solvers in Constraint Programming, in CP-2001, Paphos, Cyprus*, pp. 21–27, Dec. 2001.

[48] MEISELS, A., ZIVAN, R., "Asynchronous Forward-Checking for Distributed CSPs", In: *Frontiers in Artificial Intelligence and Applications* (ZHANG, W., ed.), pp. 93–109, IOS Press, 2003.

[49] MOHR, R., HENDERSON, T. C., "Arc and Path Consistency Revisited", *Artificial Intelligence*, v. 28, n. 2, pp. 225–233, 1986.

[50] MÜLLER, T., "Solving Set Partitioning Problems with Constraint Programming", In: *Proceedings of the 6th Intl. Conf. on the Practical Application of Prolog and the 4th Intl. Conf. on the Practical Applications of Constraint Technology (PAPPACT)*, (London, UK), pp. 313–332, The Practical Application Company Ltd, March 1998.

[51] NGUYEN, T., DEVILLE, Y., "A Distributed Arc-consistency Algorithm", *Science of Computer Programming*, v. 30, n. 1-2, pp. 227–250, 1998.

[52] PACHECO, P. S., *Parallel Programming with MPI*. Morgan Kaufmann Publishers, Inc., California, 1997.

[53] PARLETT, B., "Do we Fully Understand the Symmetric Lanczos Algorithm yet?", In: *Proceedings of the Cornelius Lanczos International Centenary Conference - SIAM*, (Phyladelphia, PA), pp. 93–107, 1994.

[54] PARLETT, B. N., *The Symmetric Eigenvalue Problem*. Upper Saddle River, NJ, USA, Prentice-Hall, Inc., 1998.

[55] PARLETT, B. N., SIMON, H., STRINGER, L., "Estimating the Largest Eigenvalue with the Lanczos Algorithm", *Mathmatics of Computation*, v. 38, n. 157, pp. 153–165, 1982.

[56] PEREIRA, M. R., *Paralelização de Algoritmos de Consistência de Arcos em um Cluster de PCś*, Dissertação de Mestrado, COPPE - Engenharia de Sistemas e Computação - Universidade Federal do Rio de Janeiro, Rio de Janeiro, Agosto 2001.

[57] PEREIRA, M. R., DUTRA, I. C., CASTRO, M. C. S., "Arc-consistency Algorithms on a Software DSM Platform", In: *Colloquium on Implementation of Cronstraint and Logic Programming Systems - CICLOPS 2001*, (Paphos, Cyprus), pp. 103–117, December 2001.

[58] PEREIRA, M. R., DUTRA, I. C., CASTRO, M. C. S., "Parallelisation of Arc-consistency Algorithms", In: *Jornadas Chilenas de Computación, 2002. VI Workshop on Distributed Systems and Parallelism*, (Copiapó, Chile), Sociedad Chilena de Ciencia de la Computación, November 2002.

[59] PEREIRA, M. R., VARGAS, P. K., FRANÇA, F. M. G., *et al.*, "Applying Scheduling by Edge Reversal to Constraint Partitioning", In: *Symposium on Computer Architecture and High Performance Computing*, v. 15, pp. 134–141, IEEE. Computer Society Press, November 2003.

[60] RINGWELSKI, G., HAMADI, Y., "Multi-Directional Distributed Searches with Aggregation", In: *Proceedings of The Sixth International Workshop in Distributed Constraint Resoaning (DCR-05), Edinburgh,Scotland*, pp. 2–14, July 2005.

[61] RUIZ-ANDINO, A., ARAUJO, L., RUZ, J., "Parallel Solver for Finite Domain Constraints", Technical Report SIP 71/98, Universidade Complutense de Madri, 1998.

[62] RUIZ-ANDINO, A., ARAUJO, L., SÁENZ, F., *et al.*, "Parallel Execution Models for Constraint Programming over Finite Domains", In: *Principles and Practice of Declarative Programming, Intl. Conf. PPDP, Paris, France* (NADATHUR, G., ed.), v. 1702 of *Lecture Notes in Computer Science*, pp. 134–151, Springer, September 29–October 1 1999.

[63] RUSSEL, S., NORVIG, P., *Inteligência Artificial: tradução da segunda edição/ Stuart Russel, Peter Norvig; tradução de PubliCare Consultoria*. Editora Campus, 2004.

[64] SILAGHI, M. C., *Asynchronously Solving Problems with Privacy Requirements*. Phd thesis, Swiss Federal Institute of Technology at Lausanne (EPFL), 2002.

[65] SIMON, H. D., TENG, S. H., "Partitioning of Unstructured Problems for Parallel Processing", *Computing Systems in Engineering*, v. 2, pp. 135–148, March 1991.

[66] SIMON, H. D., TENG, S. H., "How Good Is Recursive Bisection", *SIAM Journal on Scientific Computing*, v. 18, pp. 1436–1445, July 1995.

[67] SOLOTOREVSKY, G., GUDES, E., MEISELS, E., "Modeling and Solving Distributed Constraint Satisfaction Problems (dcsps)", In: *Constraint Processing - 96*, (New Hamphshire), pp. 561–562, October 1996.

[68] SPENDLEY, G. R., HEXT, G. R., HIMSWORTH, F. R., "Sequential Application of Simplex Designs inn Optimization and Evolutionary Operation", *Technometrics*, v. 4, pp. 441–461, 1962.

[69] TOKUYAMA, T., NAKANO, J., "Geometric Algorithms for a Minimum Cost Assignment Problems", In: *Proceedings of the 7th Annual Symposium on Computational Geometry*, (New York, NY, USA), pp. 262–271, ACM Press, 1991.

[70] TONG, B. M., LEUNG, H. F., "Data-parallel Concurrent Constraint Programming", *The Journal of Logic Programming*, v. 35, n. 2, pp. 103–150, 1998.

[71] YOKOO, M., DURFEE, E. H., "Distributed Constraint Satisfaction for Formalizing Distributed Problem Solving", In: *12th IEEE International Conference on Distributed Computing Systems*, pp. 614–621, 1992.

[72] YOKOO, M., DURFEE, E. H., ISHIDA, I., *et al.*, "The Distributed Constraint Satisfaction Problem: Formalization and Algorithms", *IEEE Transactions on Knowledge and Data Engineering*, v. 10, pp. 673–685, September/October 1998.

[73] YOKOO, M., HIRAYAMA, K., "Algorithms for Distributed Constraint Satisfaction: A Review", *Autonomous Agents and Multi-Agent Systems*, v. 3, n. 2, pp. 198–212, 2000.

[74] YOKOO, M., SUZUKI, K., HIRAYAMA, K., "Secure Distributed Constraints Satisfaction: Reaching Agreement without Revealing Private Information", *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP-2002)*, 2002.

[75] ZERVOUDAKIS, K., "An Object-Oriented Implementation of a Finite-Domain Integer CSP Solver", In: *Proceeding of Second International Workshop on Constraint Propagation and Implementation, Volume II, Solver Competition*, (Sitges-Spain), pp. 89–93, October 2005.

[76] ZIVAN, R., MEISELS, A., "Synchronous vs Asynchronous Search on DisCSPs", In: *Proceedings of the First European Workshop on Multi-Agent Systems (EUMA)*, December 2003.

[77] ZIVAN, R., MEISELS, A., "Concurrent Backtrack Search for DisCSPs", In: *Proceedings of the 17th International Florida Artificial Intelligence Research Symposium Conference (FLAIRS-04)*, pp. 776–81, May 2004.

[78] ZIVAN, R., MEISELS, A., "Concurrent Dynamic Backtracking for Distributed CSPs", In: *Proceedings Constraint Programming*, pp. 782–787, September 2004.

[79] ZIVAN, R., MEISELS, A., "Asynchronous Backtracking for Asymmetric DisCSPs", In: *Proceedings of the Sixth International Workshop on Distributed Constraint Reasoning (DCR-05)*, (Edinburgh, Scotland), pp. 148–160, July-August 2005.

[80] ZIVAN, R., MEISELS, A., "Dynamic Ordering for Asynchronous Backtracking on DisCSPs", In: *Proceeding of Sixth International Workshop on Distributed Constraint Reasoning*, (Edinburgh, Scotland), pp. 15–29, July 2005.