SIMULAÇÃO DE MODELOS PARALELOS DE PROGRAMAÇÃO EM LÓGICA

Ruy Marinho da Costa

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 MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Prof. Cláudio Luis de Amorim, Ph.D.
(Presidente)

Prof. Raul Queiroz Feitosa, Ph.D.

Profa. Leila Ripoll Eizirik, D.Sc.

Profa. Sheila Regina Murgel Veloso, D.Sc.

RIO DE JANEIRO, RJ - BRASIL
NOVEMBRO DE 1991

COSTA, RUY MARINHO DA

Simulação de Modelos Paralelos de Programação em Lógica [Rio de Janeiro] 1991

X, 151 p. 29,7 cm (COPPE/UFRJ, M. Sc.,Engenharia de Sistemas, 1991)

Tese - Universidade Federal do Rio de Janeiro, COPPE

1 – Programação em Lógica, 2 – PROLOG, 3 – Processamento Paralelo

I. COPPE/UFRJ II. Título (série)

À minha filha Milena

Agradecimentos

Ao meu orientador, Cláudio Amorim, pela orientação e auxílio prestados durante a elaboração da tese, principalmente pelos trabalhos de organização e correção da redação da tese e pelo tema que criou para a tese.

À amiga Inês, que participou de forma ativa e decisiva do estudo dos modelos e da concepção do simulador, além de fornecer diversos conselhos que foram extremamente importantes.

Ao amigo Ricardo Bianchini, cujas opiniões e observações foram muito úteis no desenvolvimento e implementação do simulador.

Ao amigo Juarez, pelo apoio, amizade e incentivo dispensados, sem os quais este trabalho não seria realizado.

Ao Professor Valmir, pelo auxílio lúcido e eficaz, dispensado em todas as situações nas quais foi solicitado, no decorrer de todo o processo de realização do mestrado.

Às Marias e não-Marias do programa, que com amizade, companheirismo e personalidade, em muito contribuiram para tornar mais amenos os dias de trabalho.

A Ana Paula, Cláudia e Denise, pela paciência e colaboração prestada na área burocrática.

Aos meus irmãos (Augusto, Renato e Ricardo), que através da união e da amizade, contribuiram fortemente para a realização deste trabalho.

Aos meus pais (Ruy e Cândida) e aos de minha esposa (Antônio e Gracinda), pelo total apoio e compreensão dedicados, que tornaram possível atingir o estágio atual.

E, por fim, à minha esposa, Cláudia, por tudo que fez e deixou de receber neste sacrificado período.

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

SIMULAÇÃO DE MODELOS PARALELOS DE PROGRAMAÇÃO EM LÓGICA

Ruy Marinho da Costa

Novembro, 1991

Orientador: Professor Cláudio Luis de Amorim

Programa: Engenharia de Sistemas e Computação

Resumo

Este trabalho propõe e avalia o uso da simulação no estudo e no desenvolvimento de modelos de execução paralela de Programação em Lógica. Um simulador foi projetado e testado utilizando os modelos "Backup" [Furukawa 82] e "Kabu–Wake" [Sohma 85]. A avaliação, comparação e análise do desempenho dos modelos são conduzidas através de vários experimentos e revelam o potencial do simulador para identificar prováveis deficiências de modelos propostos na literatura.

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

SIMULATION OF LOGIC PROGRAMMING PARALLEL MODELS

Ruy Marinho da Costa

November, 1991

Thesis Supervisor: Professor Cláudio Luis de Amorim

Department: Systems and Computing Engineering

Abstract

This work presents and evaluates the use of simulation in the study and development of parallel execution models for Logic Programming. A simulator was designed and tested using the "Backup" [Furukawa 82] and "Kabu–Wake" [Sohma 85] models. The performance of these two models was evaluated and compared through experiments that revealed the potential of the simulator to identify probable deficiences of these and others models proposed in the literature.

Índice

Lista de Algoritmos

Lista de Figuras

Lista de Gráficos

Lista de Tabelas

Capítulo I

Introdução

As linguagens de programação baseadas na lógica de primeira ordem (particularmente PROLOG) têm-se mostrado bastante adequadas para o desenvolvimento de programas utilizados em diversas áreas do conhecimento humano, principalmente às ligadas à Inteligência Artificial. Entretanto o fraco desempenho das implementações disponíveis (quando comparadas ao de linguagens procedimentais conhecidas) dificulta bastante os trabalhos das equipes que resolvem explorar as vantagens e potencialidades desta classe de linguagens como ferramenta de trabalho. Freqüentemente, os ganhos obtidos no desenvolvimento de uma tarefa pouco compensam a perda de desempenho ocasionada pelo processo de execução destes programas em nossas máquinas tradicionais.

Um passo bastante importante para solucionar estes problemas foi dado através do desenvolvimento de compiladores, que comparados aos processos tradicionais de interpretação, proporcionaram um ganho expressivo de desempenho. Outro passo bastante importante tem sido o de desenvolvimento de máquinas virtuais (PLM e WAM são bons exemplos), com conjunto de instruções e arquitetura adequados à execução desta classe de programas. Trilhando este caminho, novas arquiteturas (muitas das quais extremamente especializadas) e técnicas de compilação e otimização têm sido alvo e fruto de grandes investimentos por parte de inúmeras equipes de pesquisa e desenvolvimento em todo o mundo.

Além destas áreas fundamentais, também o paralelismo potencial normalmente presente nestes programas tem chamado a atenção de grupos de pesquisadores e igualmente sido investigado com muito cuidado. A exploração deste paralelismo tem-se mostrado um caminho promissor, principalmente porque possivelmente não descarta o aproveitamento de outras técnicas desenvolvidas, vindo somar-se ao farto

arsenal de trabalhos para propiciar aos usuários da programação em lógica[1] um melhor ambiente de trabalho.

Certamente o grande esforço internacional concentrado no desenvolvimento de máquinas com arquitetura paralela, com resultados significativos tais como a BBN Butterfly, Cray Y-MP e o Cubo Cósmico, tem sido de grande valia também para a programação lógica. A criação de implementações que utilizem e explorem estes ambientes nos darão exemplo da força e do potencial do casamento destes ramos da computação.

O desenvolvimento de técnicas que casam o paralelismo à programação lógica, entretanto, é um ramo extenso, onde um grande número de possibilidades se apresentam aos pesquisadores. É com o intuito de facilitar o desenvolvimento deste tipo de trabalho, de servir como balança em várias das muitas decisões a serem tomadas, que surge a nossa proposta de um Simulador de Modelos para Programação Lógica Paralela. Este simulador torna-se uma ferramenta atraente e útil sob diversos pontos de vista, sobretudo em aspectos como comparação de diferentes modelos / estratégias propostos e a implementação experimental de modelos, fundamentais para se estudar e avaliar os possíveis caminhos a seguir.

O conteúdo dos capítulos seguintes é descrito resumidamente abaixo :

- No capítulo II são descritos sucintamente alguns fundamentos para a exploração de paralelismo, os principais tipos e sub-tipos de paralelismo em utilização e relatados alguns aspectos importantes da exploração do mesmo. Alguns dos principais modelos propostos são também brevemente comentados nesse capítulo;

- No capítulo III encontramos a descrição detalhada do simulador proposto e implementado, em termos de suas unidades funcionais e principais interfaces com o usuário;

- O capítulo seguinte é destinado aos procedimentos fundamentais à análise e à implementação de um modelo no simulador. As implementações realizadas são apresentadas contendo o estudo detalhado dos modelos selecionados e os principais detalhes de sua implementação. Segue a apresentação dos resultados obtidos no processo de simulação pelos modelos selecionados e por um modelo

[1]Por conveniência, as denominações *programação em lógica* e *programação lógica* são utilizadas indistintamente ao longo do texto.

seqüencial, bem como um breve resumo dos ambientes de hardware e software utilizados;

- Finalmente, em nosso último capítulo, apresentamos as conclusões sobre o trabalho realizado e sugestões para outros futuros, bem como uma análise das perspectivas para este ramo da computação;

Nos apêndices serão respectivamente apresentados :

- A representação BNF da sintaxe utilizada pelo compilador PROLOG do simulador;
- As funções PROLOG pré-definidas para o simulador;
- As funções e procedimentos do simulador que são utilizadas na implementação dos modelos;
- As tabelas confeccionadas à partir dos dados obtidos pelas simulações e utilizadas na análise dos modelos; e
- Os gráficos confeccionados à partir dos dados obtidos pelas simulações e utilizados na análise dos modelos.

Capítulo II

Exploração de Paralelismo em Programação em Lógica

Associado a cada programa em lógica existe uma árvore (conhecida como *and / or search tree*) que representa todos os pontos de decisão e instanciação de variáveis que existem no programa. Uma estratégia de controle seria um modo de percorrer-se esta árvore em busca de soluções. O fato de termos uma árvore (cada nó pode ter mais de um filho) torna evidente o potencial de paralelismo existente nestes programas e a facilidade de serem adaptadas estratégias de controle paralelas aos mesmos. É esperado que este potencial de paralelismo possa ser explorado e que traga eficiência e, principalmente, praticidade à programação em lógica.

II.1 Principais Tipos de Paralelismo

II.1.1 Paralelismo no Algoritmo

Dentre as opções possíveis de código para uma tarefa podem existir diferenças nos desempenhos destas opções que podem diferenciá-las razoavelmente entre sí. No caso específico de exploração de paralelismo estas diferenças são, normalmente, acentuadas. Entretanto, esta forma de paralelismo não pode ser explorada sem que haja a execução paralela de programas. O vínculo entre os paralelismos do algoritmo e da execução se torna mais estreito em função de existência de variações de desempenho quando um programa é executado em diferentes modelos e vice-versa. Deste modo, cabe exclusivamente ao elaborador do algoritmo determinar qual o melhor código a ser produzido, sempre considerando o modelo de execução paralela a ser adotado. É evidente que esta relação não é facilmente explorada por poder ser tratada somente

pelo elemento humano.

II.1.2 Paralelismo nos Dados

O paralelismo nos dados é uma modalidade atrativa na medida que não obriga ao uso de um modelo de execução paralela para ser usufruída. Pode-se utilizar diversos modelos de execução seqüencial, agindo cada qual sobre uma parcela independente dos dados, todos simultaneamente. A distribuição dos dados, entretanto, é uma tarefa delicada, que exige um alto grau de compreensão do programa em questão. É difícil imaginar que esta tarefa possa ser mecanizada sem um alto grau de sofisticação, que oneraria em muito o seu uso. Desta forma esta é uma atividade que seria (assim como o paralelismo nos algoritmos) transferida inteiramente ao programador.

II.1.3 Paralelismo na Execução

Dos tipos de paralelismo relacionados este é o único que pode ser implementado sem a intervenção do elemento humano, embora existam implementações como [Gregory 87] que preferem seguir às orientações do programador na exploração de paralelismo. Este é o tipo de paralelismo preferido para estudos e implementações, sendo adotado na maioria dos casos. Será também, por este motivo, adotado em sua versão totalmente automática nos estudos e no desenvolvimento do simulador. Suas principais ramificações serão abordadas nas seções a seguir.

Paralelismo Or

O *paralelismo or* é obtido através da execução em paralelo das diferentes cláusulas cujas cabeças unifiquem com o objetivo corrente. Este paralelismo é conhecido como "OR" porque todo sucesso obtido por uma das cláusulas levará o objetivo também a um sucesso, e este somente falhará quando todas as cláusulas falharem. No exemplo da figura II.1 temos que as cláusulas alternativas para o objetivo proposto são plenamente exploráveis pelo *paralelismo or*.

Paralelismo And

O *paralelismo and* é obtido através da execução em paralelo dos diversos objetivos que compõem o corpo da cláusula corrente. Este paralelismo é conhecido como

$$avô\ (X,Y) \Leftarrow pai\ (X,Z),\ pai\ (Z,Y).$$
$$avô\ (X,Y) \Leftarrow pai\ (X,Z),\ mãe\ (Z,Y).$$

Figura II.1: Cláusulas que unificam com o objetivo $avô\ (X,Y)$

"AND" porque qualquer falha obtida por um dos objetivos levará a cláusula também a uma falha, e esta somente resultará em sucesso quando todos os objetivos resultarem em sucesso. A cláusula apresentada na figura II.2 tem o seu corpo formado por objetivos exploráveis pelo *paralelismo and.*

$$p\ (X,Y,Z) \Leftarrow q\ (X),\ r\ (Y),\ s\ (Z).$$

Figura II.2: Cláusula explorável pelo *paralelismo and*

O *paralelismo and,* entretanto, apresenta uma séria restrição não encontrada no *paralelismo or.* Existe o problema de interdependência entre os objetivos do corpo da cláusula, causado por variáveis comuns aos objetivos, que precisam ser instanciadas com o mesmo valor para comporem uma solução. O problema existe apenas quando na chamada da cláusula, as variáveis comuns não são instanciadas. Para resolver estes problemas de interdependência, o *paralelismo and* é subdividido nas seguintes formas:

All Solution And resolve todos os objetivos em paralelo e cruza as soluções de cada objetivo para compor as possíveis soluções para a cláusula;

Restricted And examina a cláusula para determinar os grupos de objetivos que podem ser executados em paralelo, seqüenciando os grupos, e

Stream And implementa uma forma de *pipeline* com produção e consumo de instanciações para resolver os problemas de interdependência.

II.2 Restrições à Exploração de Paralelismo

Evidentemente, o controle necessário para a utilização destas formas de paralelismo pode provocar um grande *overhead* no processamento. Entretanto este não é o único aspecto a ser levado em consideração no projeto de uma implementação paralela. Alguns dos principais problemas já são inerentes à programação lógica, sendo agravados com a exploração de paralelismo.

Um exemplo típico nos é dado pelo gerenciamento dos ambientes de variáveis. Na modalidade de cópias de variáveis, o paralelismo causa um grande aumento no número de ambientes existentes em um mesmo período de tempo, enquanto que na modalidade de compartilhamento desses ambientes, a realização de instanciações simultâneas a uma mesma variável dificulta enormemente o trabalho.

Também o grau de paralelismo pode vir a ser um grande problema a enfrentar. Muitas vezes este grau pode ser extremamente alto e a não racionalização do seu aproveitamento, via alguma restrição ao seu uso, pode levá-lo a extrapolar a capacidade do equipamento sendo utilizado. Mesmo equipamentos com grande capacidade podem não representar solução em muitos casos, pois a complexidade destes programas tende a ter um desenvolvimento de base exponencial.

Um problema que é fruto de os programadores estarem presos à estratégia de controle do PROLOG (busca em profundidade à esquerda com avaliação na mesma ordem da declaração no programa fonte) seria ocasionado pelo desrespeito do paralelismo à mesma. Este fato leva uma estratégia paralela a percorrer de modo diferente a árvore de soluções e, portanto, a alterar a ordem de encontro das soluções, o que pode representar um sério problema para alguns programas que só funcionariam a contento se alterados seus algoritmos.

Por fim, existem casos em que o grau de paralelismo apresentado pode não justificar a utilização de técnicas complexas para a sua exploração (devido ao *overhead* normalmente gerado pelas mesmas), provocando o questionamento da exploração de paralelismo versus a otimização dos métodos seqüenciais.

II.3 Alguns Modelos Propostos

Vários modelos foram propostos para ser aproveitado o paralelismo dos programas em linguagens lógicas. Alguns modelos propuseram alterações nas linguagens de modo que o programador exercesse controle sobre (ou pelo menos indicasse onde ou como explorar) o paralelismo. Apesar destes modelos terem os seus méritos, devido a fatores de ordem diversa (não uniformidade para compilar, por exemplo), decidimos que neste trabalho apenas os modelos mais puros, que detectassem e explorassem o paralelismo implícito na linguagem, seriam alvo de análise. Sendo assim, apresentamos aqui um pequeno resumo de alguns dos principais trabalhos com estas características:

[Conery 87] explora as modalidades *Or* e *Stream And* de paralelismo. O modelo é bastante complexo e não impõe qualquer espécie de restrição ao aproveitamento do paralelismo encontrado.Utiliza-se de alguns algoritmos complexos para ordenação de literais e verificação de dependências de variáveis entre objetivos, além de contar com um esquema de retrocesso semi-inteligente que pode diminuir consideravelmente o tamanho da árvore de busca. Este modelo pode ser considerado um dos mais completos em termos de aproveitamento de paralelismo e muitos outros modelos são adaptações ou simplificações deste.

[DeGroot 84] prefere abordar unicamente a modalidade de paralelismo *Restrict And*, afim de evitar a possível explosão do número de processos provocada pela exploração do *paralelismo or*. Outro ponto fundamental para esta escolha está sedimentado nas análises estática (realizada em tempo de compilação) e dinâmica (tempo de execução) do programa. Espera-se que a análise estática gere primitivas que quando executadas determinem a seqüência de exploração do paralelismo existente. Encontramos em [Hermenegildo 86] o casamento desta técnica (sofrendo algumas restrições) com a WAM, uma máquina PROLOG descrita em [Warren 83].

[Lipovski 85] explora potencialmente ambas as formas de paralelismo indistintamente. Obtém este resultado devido à sua abordagem das metas em execução (um caminho na árvore e seu contexto são processados e armazenados). Os EPs (Elementos Processadores) utilizam-se de técnicas *Branch and Bound* para decidirem que caminho deverão seguir, sendo o paralelismo advindo do fato dos EPs trabalharem simultanemente.

[Furukawa 82] apresenta uma técnica que pode ser considerada como uma variação do *paralelismo and*. Durante a resolução do corpo de uma cláusula com n objetivos, após a resolução de um dos objetivos, enquanto se tenta resolver o atual, uma (e somente uma) nova solução para o anterior é procurada simultaneamente, com o intuito de acelerar um possível retrocesso, no caso de falha do objetivo corrente. Portanto estando-se no i-ésimo objetivo da cláusula, os $i-1$ anteriores estarão com uma solução alternativa ou em busca da mesma. É importante notar que o processo se repete em cada cláusula sendo executada.

[Sohma 85] procura evitar os problemas gerados pela exploração de *paralelismo or* criando uma arquitetura específica onde cada processador trabalha seqüencialmente. Deste modo o número de tarefas realizadas em paralelo será no máximo igual ao número de processadores no sistema. Dispõe de duas redes de comunicação: uma para troca de contextos e outra para que os processadores ociosos possam realizar pedidos de trabalho. Um dos aspectos mais interessantes deste modelo está no seu modo de determinar os pontos de quebra para processamento paralelo. Dentre os vários que são determinados ao longo do processamento, o ponto escolhido é aquele mais próximo à raiz. Deste modo de divisão vem também o nome do modelo: "Kabu–Wake", nome de uma técnica japonesa para podagem de raizes de árvores.

Os modelos [Furukawa 82] e [Sohma 85], por suas características de simplicidade, de implementação e alteração, como também pelos caminhos completamente opostos que optaram para realizar a exploração de paralelismo, serão alvo de estudo mais detalhado, exemplificando a realização de implementações em nosso simulador.

Capítulo III

Simulação de Modelos Paralelos

Sem dúvida, uma das dificuldades à realização de estudos comparativos entre os modelos de execução paralela propostos na área de Programação em Lógica é a falta de uniformidade entre as implementações realizadas e, por muitas vezes, a própria falta de uma implementação. Estas, quando realizadas, sempre o foram em condições diversas, em máquinas diferentes (muitas das quais experimentais) e utilizando-se de técnicas também diversas, tornado-se assim praticamente impossível realizar qualquer comparação entre os resultados obtidos pelas implementações realizadas dos modelos propostos. Um ambiente comum, dotado de diversas facilidades, onde apenas os modelos, e não as implementações, pudessem ser avaliados, sob as mesmas condições, é a principal motivação para o desenvolvimento de um simulador.

III.1 Objetivos

A simulação possibilita medir as operações fundamentais (quantitade de unificações, comunicações, ...) realizadas pelos modelos e aquelas que possam ser consideradas importantes para o seu desempenho (ordenação de literais em [Conery 87], por exemplo), tornando-se, desta forma, uma ferramenta realmente capaz, não só de proporcionar condições de avaliação adequada do desempenho de um modelo, mas também de se comparar diferentes modelos. Desta forma, existem dois aspectos a serem avaliados : o comportamento do modelo na exploração da árvore de busca e a eficiência de utilização dos recursos disponíveis.

A evolução em tempo de execução de um modelo é um aspecto que reflete como o modelo se comporta na exploração do paralelismo potencial existente na árvore de soluções dos programas a serem executados. As diferenças de comporta-

mento, basicamente residem na exploração dos paralelismos *OR* e *AND* e na metodologia de busca (em largura, profundidade ou heurística). Um ponto importante é poder acompanhar como o modelo cria dinamicamente a sua árvore, avaliando não só a evolução, mas também as alternativas que o modelo teria, criticando as suas decisões (fundamental para um melhor ajuste ou compreensão do método em uso).

A utilização dos recursos é sem dúvida um aspecto importante a ser observado, principalmente por tratarem-se de execuções paralelas. É também desta forma que se sabe o quanto um modelo trabalhou para uma solução e se foi mais rápido que um outro. Pode-se avaliar a utilização dos processadores no decorrer do tempo e a distribuição percentual de operações básicas como criação e término de processos, comunicações internas e externas e unificações, além de outras particularidades geradoras de *overhead* que possam existir em um modelo.

Outro objetivo é o de poder facilmente ajustar ou alterar um modelo e avaliar o efeito, confrontando os resultados obtidos pelas diferentes versões. Vale notar que, na maioria dos casos, os resultados publicados são oriundos de execuções de programas diferentes, dificultando ainda mais qualquer estudo comparativo, e que para ser obtida uniformidade nas medições o mesmo conjunto de programas deve ser empregado. Existem ainda as vantagens naturais de um simulador, à medida que não se fazem necessários a implementação real de um modelo, nem o *hardware* específico do modelo (quando for exigido por este) e pode-se implementar diversos modelos diferentes (ou variações de um mesmo modelo) com pequeno dispêndio de tempo e de recursos. Por fim, lembramos que condições como variação do número de processadores, ficam muito facilitadas de serem realizadas através da simulação.

III.2 A Arquitetura do Simulador

O simulador é composto basicamente pelos seguintes componentes :

Compilador : que atua previamente ao simulador propriamente dito, gerando o **Banco de Cláusulas** (que atuará como código) a ser utilizado por este;

Sincronizador : que funciona como um relógio estabelecendo a ordem dos eventos;

Processadores Virtuais : que são em número determinado pelo usuário e controlam os **Processos** dos modelos simulados; e

Controlador de Comunicações : que manipula as mensagens (internas e externas).

O esquema de ligação desses componentes em um processador real pode ser visto na figura III.1. Cabe aqui observar que a arquitetura do simulador não influencia na dos modelos, sendo possível que qualquer topologia e esquema de memória (distribuída ou compartilhada) sejam aplicadas a um modelo. Também integra o sistema, um conjunto de rotinas próprio para a interpretação de programas PROLOG com o objetivo de facilitar tanto a definição dos algoritmos, quanto a manipulação das estruturas de dados PROLOG fornecidas pelo simulador. O uso destas rotinas permite ainda que o simulador tenha ciência do trabalho que está sendo realizado e possa contabilizá-lo.

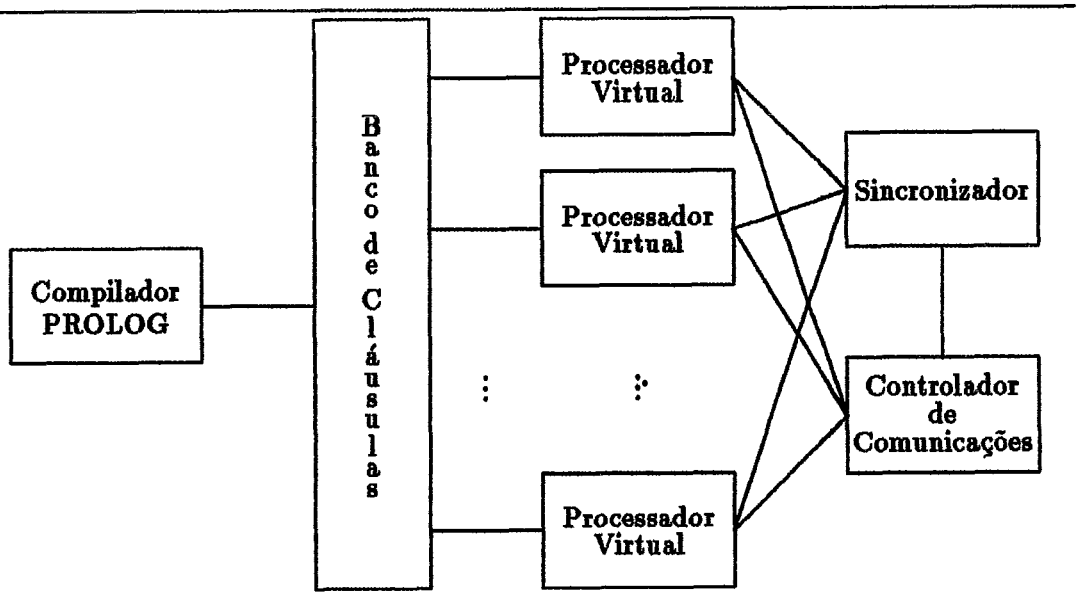


Figura III.1: Arquitetura do Simulador

III.3 O Compilador

As principais funções do compilador são :

- Realizar as análises Léxica e Sintática do programa fonte PROLOG a ser simulado;

- Gerar um grafo que represente este programa e possibilite a sua execução por parte do simulador; e
- Propiciar informações sobre cláusulas pré-definidas ao simulador, para que este possa tratá-las de forma diferenciada das demais.

O código gerado é armazenado em um banco de cláusulas que funciona como uma espécie de memória global, a qual todos os processadores virtuais têm acesso, ou como se cada processador tivesse a sua própria cópia do código, funcionando também com memória distribuida. As cláusulas são postas no banco preservando a ordem de declaração das mesmas no fonte. Este procedimento é fundamental para que determinados algoritmos PROLOG venham a funcionar corretamante em uma estratégia de controle nos moldes da utilizada nas implementações PROLOG. O acesso a este banco de cláusulas por parte do modelo será descrito na seção III.7 e é realizado por intermédio de um grupo de rotinas a ser descrito com mais detalhes no apêndice C.

As diferentes implementações de PROLOG têm mantido uma notação padrão. A sintaxe BNF adotada pelo compilador pode ser encontrada no apêndice A. Esta notação é uma abreviação das normalmente empregadas e possui pequenas restrições em *strings*, caracteres, expressões e comparações. Porém parte destas restrições (expressões e comparações), consideradas fundamentais para a programação, serão supridas via cláusulas pré-definidas, que são um meio prático e eficiente de realizar este trabalho, podendo a relação das mesmas, acompanhadas de suas respectivas descrições, ser encontrada no apêndice B.

III.4 Processadores Virtuais

A utilização de processadores virtuais se faz necessária para que o simulador possa trabalhar, obedecendo à topologia determinada por um modelo. Também desta forma apenas a usual restrição de capacidade de memória imprime um limite ao número de processadores virtuais. Manutenção de estatísticas e controle fiel de atividades são outros fatores que endossam o uso de processadores virtuais por parte do simulador.

O estudo dos diversos modelos citados na seção II.3 nos revelou que de um modo geral os processadores para PROLOG necessitam basicamente fornecer os seguintes recursos :

- Manter diversos processos;
- Permitir que operações de *time-slice* possam ou não, ser realizadas;
- Diferenciar a prioridade dos processos, embora não sejam necessárias múltiplos níveis, bastando diferenciá-los pelo tempo para *time-slice*;
- Fornecer informações sobre seu estado atual (trabalhando, parado ou aguardando evento);
- Bloquear processos; e
- Manter por algum tempo as informações sobre os processos terminados, com o objetivo de melhor se analisar um período do processamento.

O número de processadores existentes tanto pode ser fixo (pré-determinado a cada simulação), como variável (limitado ou não a um número máximo). Evidentemente o primeiro modo é mais simples de ser implementado e será o adotado. O segundo irá requerer que as suas estatísticas sejam feitas de acordo com padrões próprios, pois no caso de uma implementação com número de processadores variável a contagem dos processadores ociosos para efeito de avaliação da eficiência de um modelo, deve seguir uma metodologia especial, uma vez que o número de processadores usado pelo modelo deve variar muito durante a realização da computação. Esta questão está ligada a que tipo de ambiente será utilizado pelo modelo, se uma máquina dedicada ou se poderá alocar e liberar processadores durante a computação.

III.5 O Controlador de Comunicações

Praticamente toda comunicação existente nos modelos estudados é realizada entre pai e filho ou entre processos previamente definidos e distribuídos segundo uma regra bem definida. Desta forma, o envio de uma mensagem torna-se uma tarefa fácil. Já o recebimento de mensagens normalmente exige que estas sejam recebidas de uma fonte específica, embora em algumas ocasiões um processo possa esperar por uma mensagem qualquer. Agrega-se o fato de que um processo pode querer ou não

esperar pela chegada de uma mensagem, ou esperar ou não pela recepção de uma transmissão sua.

Para realizar-se comunicações eficientemente sob estas condições observou-se que é possível realizar todo o endereçamento das mensagens via o endereço dos processos envolvidos, devendo tanto as operações de envio como de recebimento fornecerem remetente e destinatário. Evidentemente, deve ser também indicado o desejo de aguardar ou não a conclusão da operação. Deve ser observado que, o tempo de comunicação será função do tipo de comunicação (interna ou externa a um processador virtual), da distância entre os processadores (no caso de comunicação externa) e do tamanho da mensagem enviada.

O sistema de comunicações deve simular diversos canais de capacidade a ser definida pelo usuário (com default 0). Estes canais seriam criados e eliminados automaticamente, de acordo com as necessidades dos processos, sem a participação explicita do usuário. Sendo assim, os próprios canais ficam responsáveis pelo aramzenamento das mensagens.

Envia (*remetente*, *destinatário*, *espera*, *tamanho*, *mensagem*).

Figura III.2: Envio de mensagem

---

mensagem ← *Recebe* (*destinatário*, *remetente*, *espera*).

Figura III.3: Recebimento de mensagem

III.6 O Sincronizador

O mecanismo de sincronização faz-se necessário para proporcionar um comportamento uniforme (e portanto mais real) aos processadores virtuais. Este mecanismo atua sobre os processos, restringindo-os ou liberando-os na realização de atividades básicas, baseado nas informações que possui sobre os processadores virtuais dos mesmos. Cuida para que haja um constante rodízio na posse da UCP da máquina hospedeira por parte dos processadores virtuais (na verdade dos processos deste), procurando impedir que um processador virtual avance muito em seu processamento

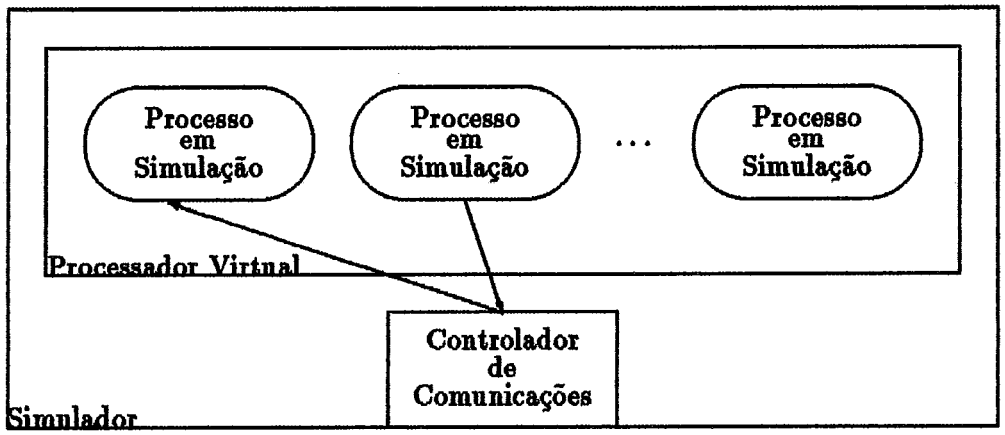


Figura III.4: Comunicação Interna

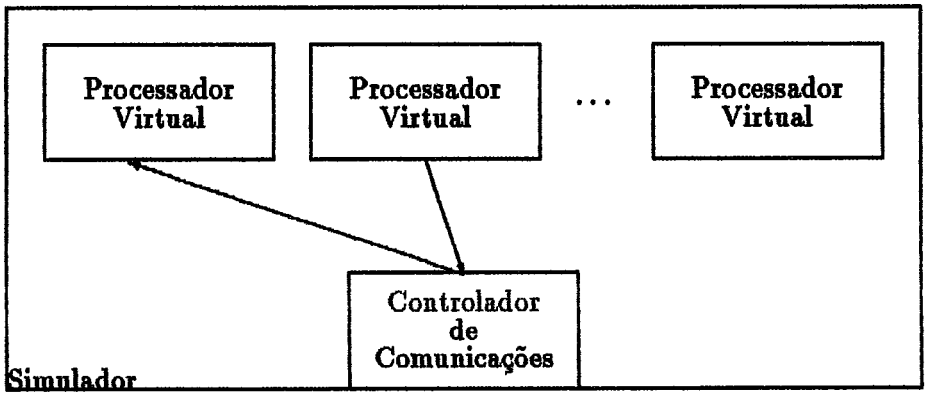


Figura III.5: Comunicação Externa

em relação aos demais, mantendo uniforme a velocidade dos processadores virtuais do sistema.

Para desempenhar tal tarefa, o sincronizador possui um relógio global que determina o tempo para operações dos processadores virtuais. A cada unidade de tempo os processadores virtuais ativos realizam uma ou parte de uma tarefa. Os inativos registram este tempo como ocioso. Como cada tarefa tem complexidade diferente, também o tempo para realização das mesmas difere. São consideradas as seguintes tarefas :

1. Pesquisa no banco de cláusulas;
2. Unificação de termos simples;
3. Criação de processos locais;
4. Criação de processos remotos;
5. Término de processo;
6. Envio de mensagem local;
7. Envio de mensagem remota;
8. Recepção de mensagens [1] e
9. Particularidades do modelo.

Os tempos para cada tarefa devem ser obtidos por medições na máquina alvo e a seguir passados para o simulador, adaptados a uma escala onde o menor valor é 1. As particularidades devem ser definidas pelo usuário e têm como objetivo principal contabilizar os pontos de *overhead* que um modelo possua.

As tarefas são sempre executadas após ter-se decorrido o tempo necessário à sua execução porque o tempo simulado é muito superior ao tempo real de execução da tarefa. Isto equivale a dizer que primeiro se pede para executar uma tarefa chamando um procedimento do sistema (figura III.6) que gasta o tempo necessário a execução da mesma, e a seguir entra o código da tarefa em questão, que é rapidamente executado. Isto visa impedir que resultados sejam obtidos antes do seu

[1] Como nem sempre é possível determinar antecipadamente o tamanho de uma mensagem ou se esta é local ou remota, a diferenciação necessária deve ser feita posteriormente ao recebimento da mesma.

processamento, que mensagens cheguem ao destino antes de serem enviadas e que processos comecem sua execução antes de serem criados, impedindo que a simulação obtenha resultados corretos.

Executa (*processo*, *tarefa*, *tempo*).
Tarefa.

Figura III.6: Execução de tarefa pelo modelo

III.7 Processos

Os processos são formados pelos algoritmos dos modelos, e criados como processos reais da máquina hospedeira. Uma estrutura de dados aliada às condições de bloqueio em determinadas operações fundamentais, garante o controle sobre o processo por parte do processador virtual, que desta forma pode medí-lo, escaloná-lo, bloqueá-lo ou mesmo eliminá-lo, entre outras possibilidades. Tais operações são necessárias para que o comportamento, tanto do processo, quanto do processador, sejam o mais próximo do real possível.

Para auxiliar o modelo no processamento de um programa PROLOG, o simulador dispõe de uma biblioteca de funções que permite a interpretação do programa através do método de cópia de valores e da manutenção de listas de instanciações. As funções em questão permitem :

- Obter uma consulta;
- Obter uma cláusula cuja cabeça unifique com um determinado literal e a lista decorrente desta unificação;
- Obter o número de literais em uma cláusula;
- Obter o i-ésimo literal de uma cláusula;
- Aplicar uma lista a um literal;
- Aplicar uma lista a um literal e aos seguintes na cláusula;
- Obter uma lista com as variáveis de um literal;

- Obter uma lista que seja a união de duas listas;
- Obter uma lista que seja a combinação[2] de duas listas; e
- Obter uma lista que contenha as instanciações das variáveis de uma lista, extraidas de uma outra lista.

Além dessas funções fundamentais, também são supridas rotinas que permitem a cópia, liberação e impressão de cláusulas, literais e listas. Vale lembrar que a básica renomeação de variáveis deve estar embutida na obtenção de cláusulas para evitar este trabalho seja obrigação do modelo.

---

processo ← *CriaProcesso* (*pai*, *processador*, *parâmetros*...).
TerminaProcesso (*processo*).

Figura III.7: Criação e término de processos

III.8 Avaliação de Modelos

A principal finalidade da simulação é propiciar informações sobre a execução dos modelos implementados. Por se tratar de um simulador, qualquer tipo de informação pode ser obtida. Entretanto, existem algumas medidas mais comuns e significativas, que devem ser facilmente obtidas, sem a necessidade de modifição no processo de simulação. Desta forma, o simulador poderá fornecer esses dados independentemente.

As informações consideradas básicas para a produção de medidas são o tempo global de ocorrência de um evento e o número de operações básicas (descritas na seção III.6) realizadas por cada processador até esse instante. Também a periodicidade de obtenção das informações é fator primordial, podendo estas serem fornecidas **constantemente**, a cada n unidades de tempo, ou **eventualmente**, sempre que ocorrer um evento significativo (a obtenção de uma solução ou o término da simulação). Vale ressaltar que a coleta de dados realizada **constantemente** gera uma grande quantidade de dados, que tornam mais trabalhoso o processo de análise, sem que necessariamente sejam mais significativos, sendo, portanto, normalmente

[2]união e aplicação recursiva das listas

aconselhável o uso do modo **eventual**[3].

Os dados fornecidos pelo simulador devem ser então processados (através de planilhas e geradores de gráficos) para que sejam extraidas as informações mais significativas e possam ser realizadas as análises de desempenho dos modelos. As métricas escolhidas para analisar os modelos implementados e realizadas para cada programa simulado de cada modelo, são apresentadas a seguir :

- **Variação Percentual do Tempo do Seqüencial** : são comparados os percentuais do tempo do seqüencial que cada simulação obteve. Os dados são apresentados cumulativamente e vale ressaltar que +90% significa que o modelo chegou ao mesmo resultado em 1,9 vezes o tempo do seqüencial e que -90% que precisou de apenas um décimo desse tempo[4].

- **Distribuição Percentual do Tempo Total** : são apresentadas as fatias de tempo ocupadas com ociosidade, processamento útil PROLOG, processamento de comunicações e manutenção de processos, do tempo total disponível para processamento na configuração utilizada na simulação.

- **Distribuição Percentual do Tempo Efetivo de Processamento** : são apresentadas as fatias de tempo ocupadas com processamento útil PROLOG e *overhead* (processamento de comunicações mais manutenção de processos) do tempo total disponível menos o tempo ocioso.

- **Tempo para Obtenção das Soluções e Término do Processamento** : são comparados os desempenhos das simulações com diferente número de processadores, mais a de um modelo seqüencial PROLOG. Os dados são apresentados em valores absolutos e cumulativos.

- **Percentual do Processamento Total Realizado** : são comparados os percentuais do processamento total[5] realizado a cada evento entre as simulações com diferente número de processadores mais o realizado por um modelo seqüencial PROLOG.

- **Percentual de Processamento Útil PROLOG** : mede o tempo gasto com as operações *unificação* e *pesquisa* no decorrer da simulação, sendo realizada

[3]No caso particular das implementações realizadas, o tempo de evento e as operações básicas dos processadores foram coletadas usando a periodicidade **eventual** devido ao menor volume de dados gerado sem no entanto descaracterizar o modelo.

[4]Para os valores positivos o limite superior é ∞, enquanto para os negativos é -100%.

[5]Aquele que os modelos realizaram ao final de suas respectivas simulações.

com os valores médios por evento obtidas nas simulações com diferente número de processadores e com os valores individuais de cada processador na simulação com 4 processadores.

- **Percentual de Processamento de Comunicações** : mede o tempo gasto com as operações *envio local*, *envio remoto* e *recepção* no decorrer da simulação, sendo realizada com os valores médios por evento obtidas nas simulações com diferente número de processadores e com os valores individuais de cada processador na simulação com 4 processadores.

- **Percentual de Processamento de Manutenção de Processos** : realizada com os valores médios por evento obtidas nas simulações com diferente número de processadores e com os valores individuais de cada processador na simulação com 4 processadores, mede o tempo gasto com as operações *cria local*, *cria remoto* e *termina* no decorrer da simulação.

- **Percentual de Ociosidade dos Processadores** : realizada com os valores médios por evento obtidas nas simulações com diferente número de processadores e com os valores individuais de cada processador na simulação com 4 processadores, mede o tempo passado sem a realização de operações no decorrer da simulação.

Para cada métrica citada acima são gerados gráficos (torta, linha ou barra) para facilitar o trabalho de análise e, nos casos mais significativos, são produzidas tabelas com os valores exatos obtidos. Os gráficos e tabelas mais significativos serão encontrados no decorrer da seção IV.5, enquanto os demais estão disponíveis nos apêndices D e E.

III.9 Algoritmos do Simulador

O simulador é composto por apenas um processo, o sincronizador, que realiza todo o trabalho de sincronização dos processos do modelo simulado através de mecanismos de bloqueio e ativação, além de, nos momentos apropriados, reproduzir as funções que caberiam ao *hardware* em simulação. Todos os demais processos criados pertencem ao modelo, sendo possível ao simulador executar as funções necessárias ao processo de simulação, devido ao uso, por esses processos, de diversas rotinas da biblioteca do simulador que têm embutidas no código de suas funções primárias

também o código que as conecta ao sincronizador. O algoritmo III.1 apresenta em alto nível o funcionamento do simulador, enquanto o algoritmo III.2 apresenta as funções que o modelo deve executar obrigatoriamente.

Iniciação de variáveis e tabelas;
Compilação do programa PROLOG;
Montagem das tabelas de operações básicas;
Iniciação do modelo a ser simulado;
    Repetir
        Incremento do contador do relógio em 1 TICK;
        Realização das operações básicas (ou fração) do TICK;
        Se alguma operação básica (ou fração) foi realizada então
            Se não emite informações periodicamante então
                Elimina os processos mortos;
            Senão se momento de gerar informações então
                Gera informações do periodo;
                Elimina os processos mortos;
        Senão
            Ocorreu DEADLOCK;
    Incondicionalmente;

Algoritmo III.1: Descrição do Simulador

Cria os Processadores que serão utilizados;
Cria os Processos iniciais do modelo para cada processador;

Algoritmo III.2: Iniciação de um Modelo

Capítulo IV

Avaliação do Simulador

Neste capítulo descrevemos o ambiente de implementação do simulador, o ambiente simulado e as simulações dos modelos *Backup* e *Kabu–Wake* realizadas. São estudados detalhadamente os dois modelos e ressaltados os pontos relevantes para a simulação. São também fornecidos e comentados os resultados das avaliações dos modelos em relação a um modelo seqüencial.

IV.1 O Ambiente do Simulador

O ambiente de *hardware* é composto por um micro-computador da família IBM–PC–XT/AT, e por uma placa de expansão contendo um ou mais[1] processadores TRANSPUTER (T-800 ou T-414) da INMOS. Os TRANSPUTERS são processadores de arquitetura RISC, possuem uma pequena memória interna e cada processador pode ser conectado a até outros quatro processadores via um circuito próprio que opera em alta velocidade e permite tráfego de informações em ambas as direções. Além disso, os processadores possuem uma memória local, configurando uma arquitetura com memória distribuída.

O ambiente de *software* é formado pelo sistema operacional HELIOS, que é compatível com o UNIX, e pelo compilador C que acompanha o sistema. Este compilador obedece aos padrões ANSI para o C e é dotado de extensões na sua biblioteca de funções que permitem :

- Criação dinâmica de processos para execução concorrente em um mesmo processador; e

[1]No estágio atual do simulador somente 1 processador será usado

- Sincronização de processos de um mesmo módulo de execução através do uso de semáforos.

O ambiente TRANSPUTER, descrito anteriormente, é favorável á proliferação de processos locais e à troca de mensagens internas e externas. Também o roteamento das mensagens pode ser entregue ao sistema. Devido a estas características e à possibilidade de poder ampliar o simulador para trabalhar com mais processadores, é que este ambiente foi adotado para ser o principal na implementação do simulador.

Um segundo ambiente, utilizado para implementar o simulador, é composto por um *hardware* da família IBM–PC–XT/AT tendo como *softwares* o sistema operacional MS–DOS, o compilador TURBO–C e o montador TURBO–ASSEMBLER. As principais vantagem deste sistema residem nas facilidades para depuração oferecidas pelo compilador e na facilidade do mesmo ser utilizado, por ser muito comum e de fácil aprendizado. As principais desvantagens são a reduzida memória do equipamento e necessidade de criar todo o mecanismo para utilização de processos. Seguindo algumas regras, os programas desenvolvidos neste ambiente são plenamente transportáveis para o ambiente TRANSPUTER. A criação deste ambiente visou criar facilidades para o desenvolvimento e testes do simulador e dos modelos simulados, além de permitir verificar alguns dos resultados obtidos no ambiente principal.

IV.2 Procedimentos para a Simulação de Modelos

Para se avaliar um modelo usando o simulador, alguns procedimentos básicos devem ser adotados. Primeiro, deve ser realizada uma análise do modelo buscando identificar os seguintes aspectos principais :

Topologia : deve ser identificada e analisada as implicações desta nas comunicações e nos processos;

Comunicações : o tamanho e os tipos de mensagens, bem como quais os tipos de processos e o fluxo que estas seguem (se sempre de pai para filho, por exemplo);

Controle : como o modelo decide a distribuição das tarefas entre processos e de processos entre processadores; e

Processos : os diferentes tipos encontrados e seus respectivos algoritmos.

Outros aspectos relevantes, particulares de um modelo, também devem ser analisados e suas implicações levadas em consideração na simulação. A seguir estes dados devem ser programados na linguagem C (ou qualquer outra que possua compatibilidade de código na máquina hospedeira) utilizando-se das rotinas disponíveis na biblioteca do simulador.

IV.3 Análise dos Modelos

IV.3.1 O Modelo de Paralelismo de Backup

A descrição original do Modelo de Paralelismo de *Backup* encontra-se em [Furukawa 82] onde são apresentados os algoritmos originais e diversas considerações importantes sobre o modelo, além das descrições do *hardware* próprio e da implementação realizada pelos autores. Seus formuladores discutem as vantagens proporcionadas, os problemas detectados e indicam algumas alterações para se incrementar o paralelismo.

Topologia

Nenhuma topologia específica é apresentada, mas sugere-se que os todos os processadores estejam conectados uns aos outros, formando uma rede totalmente interconectada. Desta forma, os tempos de comunicação são reduzidos ao mínimo, pois todos os processadores são vizinhos. Este fato também beneficia a criação de processos em outros processadores.

Comunicações

A comunicação entre processos é sempre constituida por uma mensagem cujo conteúdo pode ser SUCESSO ou FALHA. As comunicações podem ser entre processos de um mesmo processador (internas) ou de processadores diferentes (externas). Devido às características de sincronização de processos do modelo (requerida apenas no momento de enviar ao pai a resposta obtida) um modelo de comunicação baseado em mensagens bloqueantes seria considerado ideal, à medida que realizaria ambas as

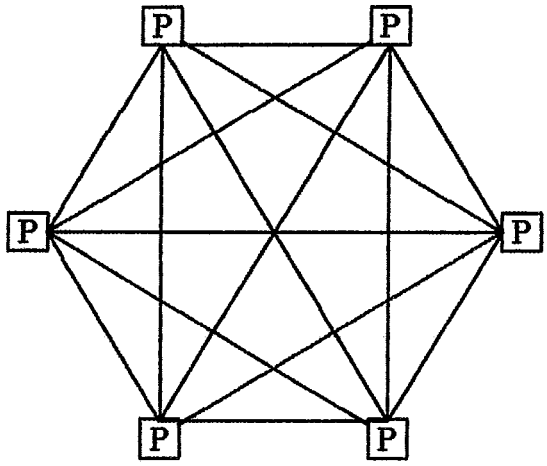


Figura IV.1: Rede totalmente interconectada com 6 processadores

funções simultaneamente, pois bloquearia o processo filho até que o pai recebesse a sua mensagem, liberando-o para continuar seu processamento e bloquearia o pai até que algum filho tenha obtido uma solução. A comunicação neste modelo é sempre realizada do filho para o pai, e sendo que o pai pode receber de vários filhos, deve ser possível a este selecionar de qual filho deseja receber mensagens em um dado momento.

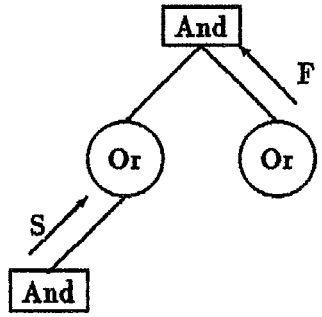


Figura IV.2: Comunicação de resultados entre processos

Controle

A princípio, o Modelo de Paralelismo de *Backup* repete a estrutura de controle de um interpretador seqüencial, sendo tanto o avanço como o retrocesso realizados de forma idêntica ao deste. Entretanto, quando um retrocesso ocorre, a continuidade do processo é acelerada pelo fato de o processo filho estar trabalhando em uma solução alternativa paralelamente ao pai.

Quando o pai cria um filho, ele se bloqueia até que o filho lhe envie uma resposta. O filho trabalha, obtem a resposta e a envia ao pai, bloqueando-se. O pai recebendo a mensagem, prossegue no seu processamento e libera o filho de modo que ambos passam a trabalhar paralelamente, o pai avançando em seu processamento e o filho buscando uma solução alternativa à anterior para o pai. Se o filho processa mais um resultado antes de o pai o desejar, ele se bloqueia, aguardando o pedido do pai por este resultado.

Se por um lado este bloqueio limita o paralelismo realizado, por outro esta atitude mantém apenas o paralelismo útil para aquela ocasião, pois nesse momento apenas uma alternativa será utilizada. O mesmo comportamento se repete a cada nível da árvore de modo que a coleta de uma alternativa pode provocar uma reação em cadeia, possibilitando que as soluções armazenadas sejam enviadas para seus respectivos pais e os processos computem novas soluções. Devido a estas características, um mecanismo de sincronização de processos faz-se indispensável para o perfeito funcionamento de qualquer implementação deste modelo.

Alocação de Processadores

A política de alocação de processadores para o modelo consiste em manter sempre um *processo and* e seus filhos (*processos or*) em um mesmo processador, enquanto um *processo or* cria os seus filhos (*processos and*) em processadores diferentes. No último caso, sempre deve ser alocado um processador ocioso (inativo). Não existe definição por parte do modelo para a situação na qual todos os processadores do sistema estejam em uso (ativos) e seja necessária a alocação de um processador.

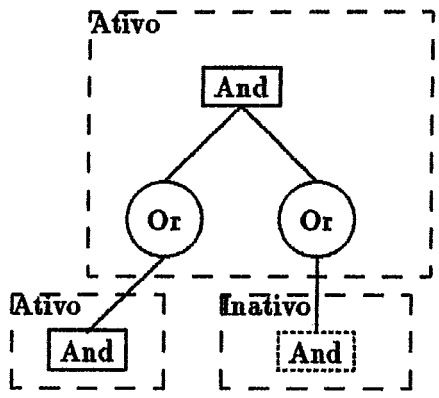


Figura IV.3: Processo And sendo criado em processador ocioso

Comportamento Básico dos Processos

O modelo funciona mantendo *processos and* e *processos or* que trabalham montando soluções. Todo processo recebe uma tarefa para realizar e cria outros processos para trabalharem na solução desta tarefa. Deste modo uma tarefa é subdividida em outras menores, as alternativas para cada tarefa são exploradas e a árvore de busca é coberta. Quando um resultado é produzido este é enviado para o processo pai. Entretanto a emissão de um resultado só é realizada quando o pai autoriza o filho a fazê-lo. Sempre que um processo falha, ele comunica o fato ao pai e morre. Outro aspecto importante é a necessidade de um processo criar outros processos em processadores diferentes do seu.

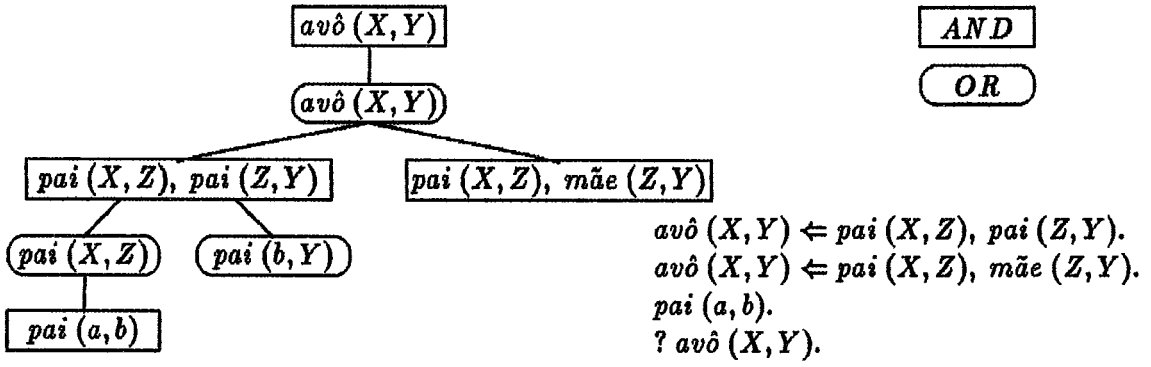


Figura IV.4: Programa e parte da sua árvore *And/Or* de execução

Processos And

Um *processo and* é criado para resolver o corpo de uma cláusula, funcionando como produtor de soluções. As soluções são produzidas através da avaliação dos literais do corpo da cláusula, sendo criados seqüencialmente um *processo or* para cada um dos literais existentes, conforme podemos observar na figura IV.5. Quando a cláusula não possui corpo (quando é um fato), o processo produz um sucesso automaticamente, falhando a seguir. Caso contrário este gera um sucesso quando todos os seus literais obtêm sucesso e falha quando o literal mais a esquerda no corpo da cláusula falhar. O algoritmo IV.1 representa um *processo and.*

```
Processo_And (cláusula)
    Se Número_de_Metas (cláusula) = 0
        Envia (SUCESSO, pai)
        Envia (FALHA, pai)
        Morre
    Senão
        i ← 1
        Repita
            Se filho [i] = nulo
                filho [i] ← Cria_Processo_Or (Submeta (i, cláusula))
            Se Recebe (filho [i]) = FALHA
                Se i = 1
                    Envia (FALHA, pai)
                    Morre
                Senão
                    filho [i] ← nulo
                    i ← (i − 1)
            Senão
                Se i = Número_de_Metas (cláusula)
                    Envia (SUCESSO, pai)
                Senão
                    i ← (i + 1)
        Para_Sempre
Fim
```

Algoritmo IV.1: *Processo And* do Modelo *Backup*

$$avô\,(X,Y) \Leftarrow \underbrace{\overbrace{pai\,(X,Z)}^{Or_0},\ \overbrace{pai\,(Z,Y)}^{Or_1}}_{Corpo\ do\ And}.$$

Figura IV.5: Distribuição dos literais de um *AND* entre os *processos Or*

Processos Or

O *processo or* é criado para buscar alternativas a fim de solucionar um literal de um *processo and*. Ele age como coletor, percorrendo o banco de cláusulas e criando *processos and* para cada cláusula cuja cabeça unifique com o seu literal, conforme ilustrado na figura IV.6. Vale notar que o *processo or* cria um *processo and* mesmo para fatos. As soluções fornecidas pelos seus filhos são então repassadas para o seu pai, *processo and*. O *processo or* falha quanda não encontra uma cláusula para o seu literal. Um *processo or* pode ser implementado segundo o algoritmo IV.2.

```
Processo_Or (literal)
    Enquanto (cláusula ← Busca_Cláusula (literal)) ≠ nulo
        filho ← Cria_Processo_And (cláusula)
        Enquanto Recebe (filho) = SUCESSO
            Envia (SUCESSO, pai)
    Envia (FALHA, pai)
    Morre
Fim
```

Algoritmo IV.2: *Processo Or* do Modelo *Backup*

Detalhes da Implementação

A topologia do modelo *Backup* não exige cuidados especiais para ser implementada no simulador devido à extrema semelhança existente entre esta e a arquitetura do simulador. Também o esquema de comunicações foi implementado sem dificuldades, utilizando-se dos recursos de bloqueio e endereçamento do simulador.

⋮	⋮
avô (X,Y)	*pai* (X,Z), *pai* (Z,Y)
avô (X,Y)	*pai* (X,Z), *mãe* (Z,Y)
⋮	⋮

Figura IV.6: Cláusulas selecionadas para resolver *avô* (X,Y)

A única decisão importante a ser tomada foi relativa à alocação de um processador para um novo processo, quando não existem processadores livres no sistema, sendo adotado o seguinte critério para a escolha do processador :

- Escolhe um processador que esteja parado devido a seus processos estarem bloqueados (aguardando por mensagens); ou
- Escolhe o processador que contenha o menor número de processos ativos.

A estrutura de dados do modelo também é bastante simples, valendo ressaltar apenas a utilizada pelos *processos and* para realizar o controle dos filhos. Esta é formada por um vetor (com uma entrada para cada sub-meta da cláusula) cujos campos contém a identificação do processo filho e a lista de variáveis obtida apos a sua resolução (para esta é utilizada a lista anterior). Desta forma tanto a "desinstanciação" de variáveis quanto o retrocesso são realizados de forma rápida e eficiente.

IV.3.2 O Modelo Kabu-Wake

A descrição do modelo *Kabu–Wake* encontra-se em [Sohma 85]. Os algoritmos não são fornecidos, mas o artigo contém alguns aspectos importantes da implementação realizada e expõe as principais características do modelo. Diversas análises do seu desempenho também são apresentadas no artigo.

Topologia

A topologia proposta para o modelo Kabu-Wake pode ser vista na figura IV.7, onde temos representados os processadores e as duas redes de interconexão existentes : a em anel e a de troca de dados. A existência de duas redes tem por função diminuir o *overhead* de comunicações que seria natural em um ambiente com diversos processadores orientados por demanda.

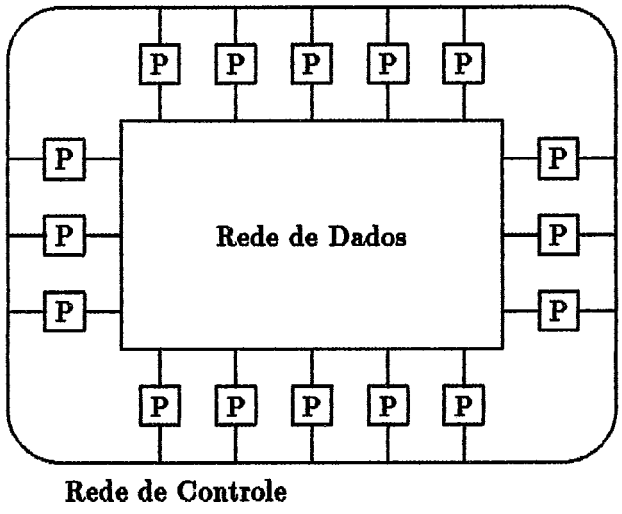


Figura IV.7: Topologia do *Kabu–Wake* para 16 processadores

Comunicação

Pelas duas redes existentes neste modelo trafegam os seguintes tipos de mensagens :

Pedidos de Serviço : que trafegam pela rede de controle, e que partindo de processadores ociosos são retransmitidos por processadores também ociosos e por aqueles que possuam um número de metas inferior a um limite arbitrário.

Metas para Execução : trafegam pela rede de dados, e são enviadas por processadores que possuam metas em número superior ao limite estabelecido e tenham recebido um pedido de serviço.

Soluções Encontradas : trafegam também pela rede de dados, sendo enviadas para o processador raiz por qualquer processador que tenha obtido uma solução e, evidentemente, não seja o processador raiz.

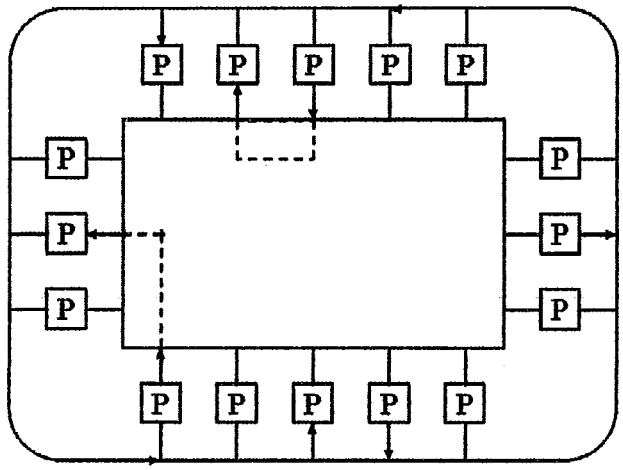


Figura IV.8: Tráfego de mensagens nas redes

Controle

Neste modelo existem apenas duas formas de controle. A primeira trata das metas de execução. Ela seleciona uma meta (priorizando uma busca em profundidade à esquerda) e a partir desta, se possível, constrói novas metas (que são armazenadas) e caso não seja possível a construção de alternativas, fica caracterizada uma falha, sendo a meta atual então descartada e o processo se repete. Quando não existirem mais alternativas no processador, este envia um pedido de serviço e o interpretador fica bloqueado até receber uma meta para execução. A Segunda forma de controle seleciona a meta que será fornecida a um processador ocioso, que deverá ser a mais próxima à raiz da árvore que existir no processador. Não foi definido critério de seleção para o caso de existirem diversas metas igualmente próximas à raiz.

Processos

No modelo *Kabu–Wake* coexistem, em um mesmo processador, um dispositivo para controle de pedidos de tarefa e um processo que trabalha buscando soluções. Também como função do processo interpretador temos a comunicação de instanciações de variáveis e dos pontos da árvore que representam uma alternativa para o processamento. Todo processo interpretador realiza uma busca em profundidade, gerando

alternativas e dando continuidade à localizada mais à esquerda na árvore de soluções. As demais alternativas tanto podem ser realizadas em paralelo (por um outro processador que esteja ocioso) ou seqüencialmente pelo próprio interpretador quando este terminar a alternativa corrente. A criação de uma alternativa se dá através de uma composição no corpo de uma cláusula, realizada substituindo-se o literal que está sendo atualmente executado pelos literais do corpo de uma cláusula, cuja cabeça unifique com o literal em execução. Serão geradas tantas alternativas quantas forem as cláusulas que unifiquem com o literal em execução. O processo se repete para cada literal do novo corpo gerado, até que seja encontrado um fato e o literal seja eliminado do corpo. Quando todos os literais são eliminados, uma solução é encontrada e quando não se consegue encontrar uma cláusula que unifique com o literal, esta alternativa falha e o interpretador procura outra alternativa para executar.

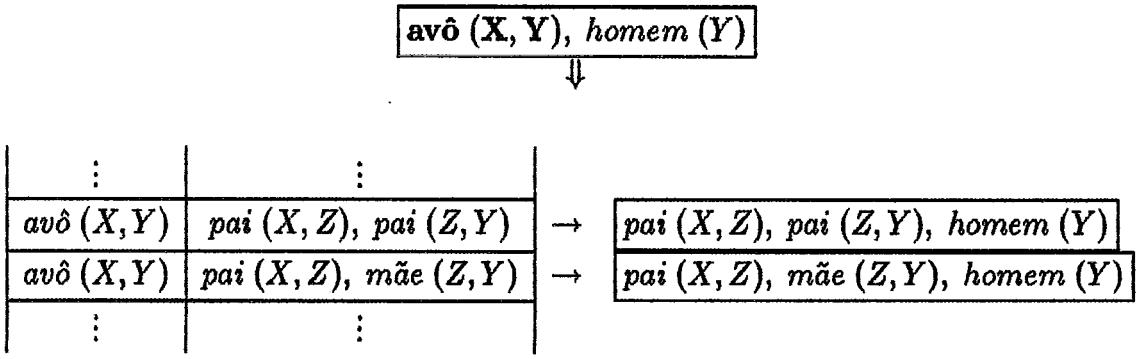


Figura IV.9: Substituição de literal pelo corpo de uma cláusula

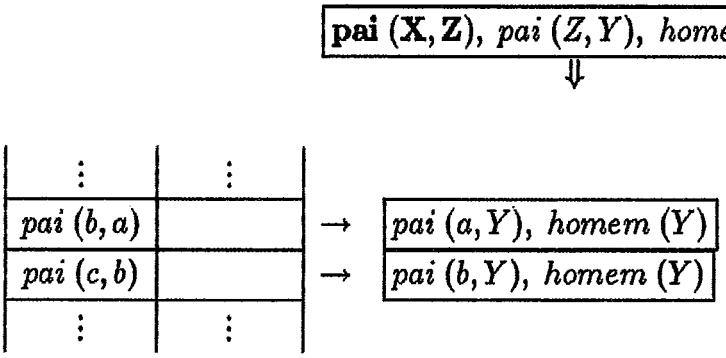


Figura IV.10: Eliminação de um literal da meta em execução

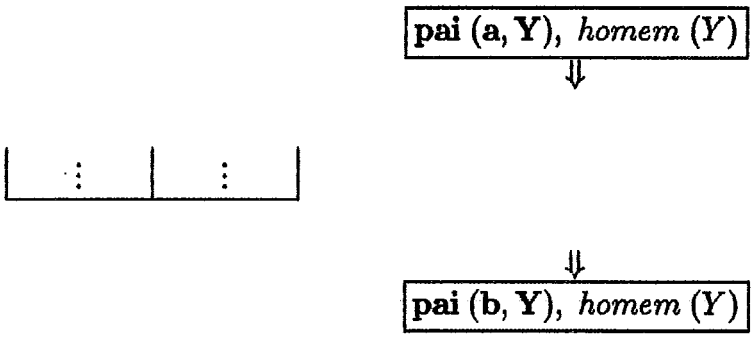


Figura IV.11: Falha da meta em execução e sua substituição

Detalhes da Implementação

A topologia do modelo *Kabu–Wake* foi implementada utilizando-se dois processadores virtuais para emular o funcionamento de um processador real do modelo. Isto se deve ao fato de o modelo contar com um *hardware* específico para comunicações embutido em seus processadores. Sendo assim, a solução adotada procede, desde que os dados gerados pela simulação sejam tratados de forma adequada. Os dados referentes ao processador que realizou o processamento de interpretação (principal) estão aptos ao estudo do desempenho do modelo, enquanto os do processador que realizou as comunicações (co-processador) têm validade apenas para a análise do fluxo de mensagens nas redes de comunicação do modelo.

A comunicação, deste modo, é distribuida entre os dois tipos de processadores conforme abaixo :

Co-processador :

- recepção de pedidos de trabalho;
- recepção de trabalho;
- recepção de soluções (apenas no processador raiz);
- retransmissão de pedidos de trabalho; e
- transmissão de trabalho.

Principal :

- transmissão de soluções (exceto no processador raiz); e
- transmissão de pedidos de trabalho.

Justifica-se a realização destas operações no processador principal, devido à necessidade de aguardar por resposta no caso dos pedidos de trabalho.

Foi adotado o critério de optar-se pela meta mais à direita no caso de empate na seleção de uma meta para ser enviada a outro processador.

Os algoritmos IV.3 e IV.4 foram elaborados para exercerem as funções de co-processador e processador principal, respectivamente.

```
Co_Processador
    Repita
        msg ← Recebe_Mensagem
        Se PEDIDO
            Se Existe_Meta
                Envia_Meta (Seleciona_Meta (ENVIO))
            Senão
                Retransmite (msg)
        Senão Se SOLUCÃO
            Imprime_Solucão
        Senão Se TRABALHO
            Insere_Meta
    Para_Sempre
Fim
```

Algoritmo IV.3: Co-processador do Modelo *Kabu-Wake*

A estrutura de dados do modelo deve comportar basicamente as operações de inserção e retirada de metas segundo os critérios estabelecidos pelo modelo, de modo a facilitar ao máximo as operações. Deve-se observar que na presente implementação tal estrutura será compartilhada por dois processadores, exigindo cuidados especiais.

```
Processador_Principal
    Repita
        Se metas = 0
            Verifica_Término
            Pede_Servico
            Aguarda_Servico
        meta ← Seleciona_Meta (EXECUÇÃO)
        Enquanto Existe_Cláusula (meta)
            Insere_Meta (Monta_Meta (meta, cláusula))
    Para_Sempre
Fim
```

Algoritmo IV.4: Processador Principal do Modelo *Kabu-Wake*

IV.3.3 Comparação dos Modelos

Na tabela IV.1 podemos apreciar as principais diferenças e semelhanças existentes entre os modelos *Backup* e *Kabu-Wake*, sob o ponto de vista dos procedimentos da seção IV.2 e do tipo de paralelismo de execução (seção II.1.3) que exploram.

IV.4 O Ambiente Simulado

O ambiente escolhido para servir de base na determinação dos tempos das operações básicas do simulador foi o TRANSPUTER. A escolha foi baseada nas mesmas características que o elegeram como base para a implementação. Embora possua muitas virtudes, nenhum dos modelos simulados poderia ser implementado em TRANSPUTERS sem modificações, devido principalmente a problemas de topologia. Ressalta-se que qualquer outro ambiente poderia ser escolhido e que o ambiente simulado em nada depende do ambiente no qual o simulador foi implementado[2].

Os tempos das operações básicas foram calculados com base nos valores obtidos nas medições das própias operações[3] na máquina TRANSPUTER escolhida

[2]Confirmado pelo fato de as simulações realizadas no TRANSPUTER obterem os mesmos resultados das realizadas no IBM-PC-XT/AT.

[3]Foram realizadas 100 vezes cada operação

Ítem	Backup	Kabu-Wake
Topologia	Totalmente Conectado	2 Redes : Anel e Similar a Estrela
Comunicações	Capacidade Zero, do Filho para o Pai	Orientada, no Anel
Controle	Busca em Profundidade à Esquerda	Busca em Largura no nó, cont. em Prof. à Esq.
Processos	AND e OR, n por processador, sob demanda	1 Tipo, estático 1 por processador
Paralelismo	AND, para retrocessos	OR, sob demanda

Tabela IV.1: Comparação dos Modelos *Backup* e *Kabu-Wake*

como base. O menor valor obtido por estas medições[4] é usado para dividir todos os demais valores obtidos. Os valores resultantes são então arredondados. Assim, o menor valor será sempre a unidade e todos os valores serão inteiros. A relação dos tempos calculados, utilizada nas simulações dos modelos implementados, pode ser encontrada na tabela IV.2.

Pesquisa no banco de cláusulas	1
Unificação de termos simples	10
Criação de processos locais	6
Criação de processos remotos	7
Término de processo	1
Envio de mensagem local	1
Envio de mensagem remota	2
Recepção de mensagens	2
Particularidades do modelo	1

Tabela IV.2: Tempos das Operações Básicas

É importante notar que devido às características do hardware do ambiente selecionado como base, as operações de troca de mensagens são realizadas em um tempo bastante reduzido e, portanto, a utilização deste ambiente como referência

[4] 454 *ticks* do TRANSPUTER

trará aos resultados das simulacões valores muito diferentes dos observados se máquinas convencionais fossem usadas como referência.

IV.5 Testes e Resultados

Foram realizadas simulações de cada modelo com 2, 4, 8 e 16 processadores e de um modelo seqüencial com 1 processador para cada um dos 4 programas PROLOG selecionados. Vale lembrar que os resultados obtidos no processo de simulação estão ligados aos valores atribuidos aos tempos das operações básicas na máquina alvo e, portanto, as análises realizadas são restritas à mesma.[5].

IV.5.1 Descrição dos Testes

Para os programas (interseção, paper, densidade e coloração) utilizados na simulação dos modelos, apresentados a seguir, são comentados os objetivos, algoritmos, nível de *paralelismo and/or* e a possível exploração deste paralelismo por parte dos modelos. São comparados os desempenhos dos modelos em cada programa e analisada a influência da árvore do programa (espaço de busca) no resultado dos modelos. O número de soluções de cada programa é também fornecido.

Os gráficos de aceleração[6] que relacionam os modelos apresentam os tempos normalizados, com a referência seqüencial igualada à zero. As medidas dos modelos *Backup* e *Kabu-Wake* são expostos na forma de variação percentual em relação ao seqüencial. Desta forma valores positivos significam perda e negativos ganhos desses modelos em relação ao seqüencial, conforme explicado na seção III.8.

Interseção

Este programa relaciona os elementos comuns a duas listas. É gerado um sucesso para cada elemento pertencente à lista de interseção encontrado, e uma falha quando o conjunto de interseção estiver completo. Este é um programa típico de trabalho com listas, envolvendo as principais técnicas de manipulação das mesmas, sendo muito comum a aplicação destas técnicas de trabalho com listas como parte de

[5]Para completar a análise seriam necessárias a realização de simulações com máquinas alvo diferentes e um estudo de sensibilidade dos modelos às variações de tempo das operações básicas

[6]Variação Percentual do Tempo do Seqüencial

programas mais complexos. As listas utilizadas no evento de simulação proveram um conjunto com 3 elementos, redundando no mesmo número de soluções.

? *interseção* (*ELEMENTO*, [*a*, *b*, *c*, *d*], [*f*, *a*, *d*, *c*]).

interseção (*CABEÇA*, *LISTA*, [*CABEÇA*|*RESTO*]) ←
    membro (*CABEÇA*, *LISTA*).
interseção (*ELEMENTO*, *LISTA*, [*CABEÇA*|*RESTO*]) ←
    interseção (*ELEMENTO*, *LISTA*, *RESTO*).

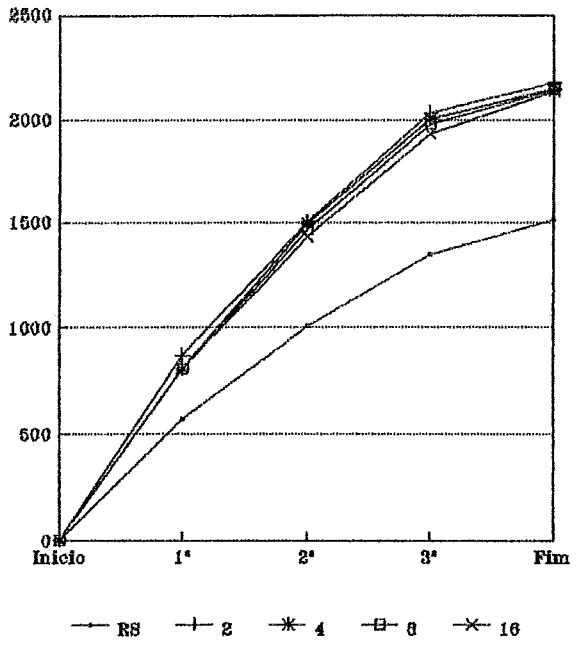
membro (*ELEMENTO*, [*ELEMENTO*|*RESTO*]).
membro (*ELEMENTO*, [*CABEÇA*|*RESTO*]) ←
    diferente (*ELEMENTO*, *CABEÇA*), *membro* (*ELEMENTO*, *RESTO*).

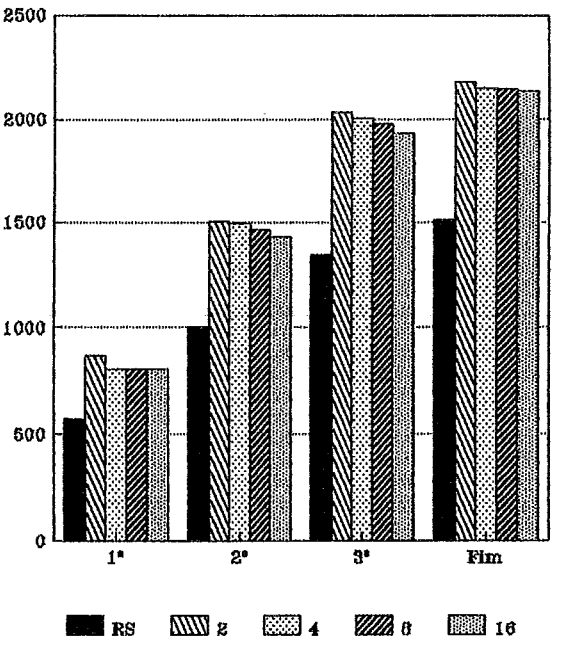
Algoritmo IV.5: Interseção das listas [a,b,c,d] e [f,a,d,c]

O paralelismo disponível é reduzido neste programa, predominando o processamento seqüencial. A modalidade *OR* de paralelismo está presente nas duas funções (*interseção* e *membro*) com duas declarações para cada. O *paralelismo and* é ainda menor, pois em apenas uma das declarações de *membro* esta modalidade existe e, ainda assim, de forma extremamente restrita (apenas duas metas com uma variável comum).

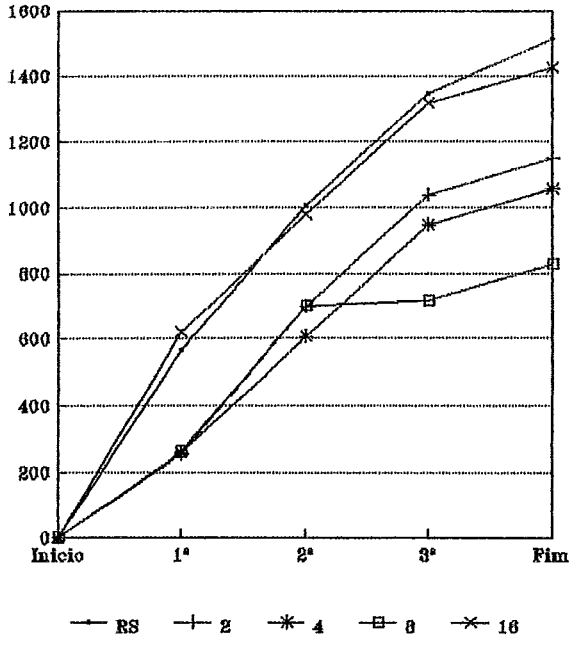
O modelo *Backup* ressente-se do mesmo problema que atinge a modalidade *AND* de paralelismo, mas com um agravante. Como a única cláusula que possui mais de uma meta (onde poderia haver retrocesso) tem por primeira meta a função pré-definida *diferente*, o modelo fica sem como realizar qualquer processamento paralelo. E, de fato, o modelo *Backup* obteve um fraco desempenho neste programa registrando em todos os eventos (soluções ou fim) um tempo sempre superior em pelo menos 40% ao do seqüencial. Um ponto decisivo certamente advém do fato de este modelo seguir à risca a árvore do seqüencial e possuir *overhead* com processos e comunicações. Como não pode explorar paralelismo, os tempos de *overhead* simplesmente somaram-se ao do processamento normal. Isto fica bem claro quando é observada a tabela IV.3, onde nota-se que o *overhead* situou-se na casa dos 36% do tempo do processamento útil (PROLOG), taxa esta muito similar à de desaceleração do modelo.

Backup





Kabu-Wake



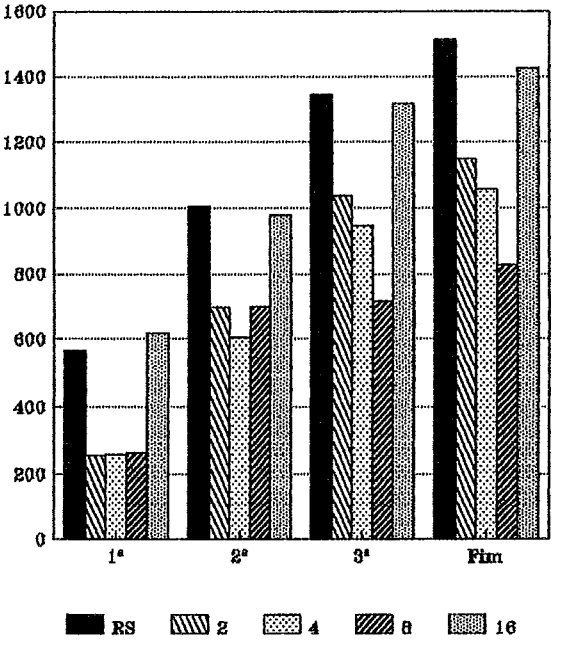


Gráfico IV.1: Tempo para Obtenção de Soluções e Término do Processamento
— Interseção —

Modelo	NP	Útil	Overhead
Backup	2	64,34%	35,66%
	4	63,72%	36,28%
	8	63,72%	36,28%
	16	63,69%	36,31%
Kabu-Wake	2	99,34%	0,66%
	4	98,95%	1,05%
	8	98,44%	1,56%
	16	97,67%	2,33%

Tabela IV.3: Distribuição Percentual do Tempo Efetivo de Processamento
— Interseção —

Por sua vez, seria esperado que o modelo Kabu-Wake tivesse pouco espaço para expandir-se devido às cláusulas possuirem poucas alternativas e de estas serem predominantemente recursões ou fatos. Entretanto, superando às espectativas, o modelo obteve um bom desempenho neste programa sendo de 23% a 54% mais rápido que o seqüencial (ver a tabela IV.4). A conclusão obtida é que o modelo ao dividir a árvore permitiu que os ramos que levam às soluções fossem explorados simultaneamente, mostrando ser, neste caso específico, bastante hábil na exploração de paralelismo, mesmo em condições adversas.

Modelo	NP	1ª	2ª	3ª	Fim
Backup	2	53,00%	49,50%	50,82%	44,05%
	4	42,05%	48,71%	48,52%	41,93%
	8	42,05%	45,72%	46,74%	41,60%
	16	42,05%	42,63%	43,18%	41,20%
Kabu-Wake	2	-54,95%	-30,38%	-23,07%	-24,01%
	4	-54,59%	-39,44%	-29,82%	-30,09%
	8	-54,24%	-30,18%	-46,66%	-45,30%
	16	9,72%	-2,49%	-2,37%	-5,69%

Tabela IV.4: Variação Percentual do Tempo do Seqüencial
— Interseção —

Neste programa, portanto, obtivemos a seguinte classificação em termos de

menor tempo na obtenção de soluções (segundo a tabela IV.5) :

Modelo	NP	1ª	2ª	3ª	Fim
Seqüencial	1	566	1004	1348	1512
Backup	2	866	1501	2033	2178
	4	804	1493	2002	2146
	8	804	1463	1978	2141
	16	804	1432	1930	2135
Kabu-Wake	2	255	699	1037	1149
	4	257	608	946	1057
	8	259	701	719	827
	16	621	979	1316	1426

Tabela IV.5: Tempo para Obtenção de Soluções e Término do Processamento
— Interseção —

1º Kabu–Wake

2º Seqüencial

3º Backup

Paper

Este programa foi extraído de [Conery 87] e implementa um Banco de Dados Relacional, onde estão armazenadas informações sobre artigos diversos. Analogamente ao programa anterior é gerado um sucesso para cada artigo encontrado e uma falha quando não houverem mais artigos em condições de satisfazer à consulta. Este programa representa uma classe de grande aplicação em Programação Lógica, sendo de grande interesse de diversos usuários. São obtidas por este programa um total de 6 soluções.

O paralelismo potencial deste programa pode ser considerado bom, considerando-se o montante de processamento nescessário à sua resolução. Cada uma das 7 funções é declarada de 2 a 8 vezes, perfazendo uma média de 5,7 cláusulas por função, resultando em um bom nível de *paralelismo or*. Já o *paralelismo and* é bem menos disponível, uma vez que apenas 2 cláusulas (ambas declarações de *paper*),

```
? paper (X, Y, Z).

paper (P, D, I) ←
    date (P, D), author (P, A), loc (A, I, D).
paper (P, D, I) ←
    tr (P, I), date (P, D).
paper (xform, 1978, uci).

author (fp, backus).          author (df, arvind).
author (eft, kling).          author (pro, pereira).
author (sem, vanemdem).       author (db, warren).
author (sasl, turner).        author (xform, standish).

date (fp, 1978).              date (df, 1978).
date (eft, 1978).             date (pro, 1978).
date (sem, 1976).             date (db, 1981).
date (sasl, 1979).

title (db, efficient_processing_of_interactive).
title (df, an_asynchronous_programming_language).
title (eft, value_conflicts_and_social_choice).
title (fp, can_programming_be_liberated).
title (pro, dec_ten_prolog_user_manual).
title (sasl, a_new_implementation_technique).
title (sem, the_semantics_of_predicate_logic).
title (xform, irvine_program_transformation_catalog).

loc (arvind, mit, 1980).      loc (backus, ibm, 1978).
loc (kling, uci, 1978).       loc (pereira, lisbon, 1978).
loc (vanemdem, waterloo, 1980).loc (turner, kent, 1981).
loc (warren, edinburgh, 1977). loc (warren, sri, 1982).

journal (fp, cacm).           journal (sasl, spe).
journal (kling, cacm).        journal (sem, jacm).

tr (db, edinburgh).           tr (df, uci).
```

Algoritmo IV.6: Banco de Dados Relacional

possuem mais de uma meta (uma possui 3 e a outra 2). Em ambas as cláusulas existem interdependência entre variáveis, restringindo ainda mais o *paralelismo and.*

O modelo *Backup*, que explora basicamente a modalidade *AND* de paralelismo, apesar ter pouco espaço, beneficia-se da pouca profundidade da árvore de busca associada, uma vez que as metas a serem processadas são todas cláusulas sem corpo (fatos). Apesar de não apresentar bom desempenho com 2 processadores (entre 10 e 164% pior – tabela IV.6), com 4 ou mais processadores o seu desempenho melhorou muito, sempre à partir da segunda solução. Nestas condições, o modelo foi aproximadamente 11% mais rápido que o seqüencial. Apenas a primeira solução foi obtida em tempo 49% superior à do seqüencial.

Modelo	NP	1ª	2ª	3ª	4ª	5ª	6ª	Fim
Backup	2	164,52%	34,55%	25,21%	12,18%	13,74%	10,71%	10,71%
	4	49,68%	-11,43%	-13,57%	-13,32%	-12,61%	-11,47%	-11,14%
	8	49,68%	-11,43%	-13,57%	-13,32%	-12,61%	-11,47%	-11,14%
	16	49,68%	-11,43%	-13,57%	-13,32%	-12,61%	-11,47%	-11,14%
Kabu-Wake	2	-40,00%	-69,48%	-71,10%	-77,89%	-55,78%	-46,46%	-45,28%
	4	-40,00%	-69,22%	-70,91%	-82,16%	-78,33%	-68,04%	-66,54%
	8	-40,00%	-68,96%	-70,91%	-81,90%	-81,04%	-81,46%	-77,17%
	16	-40,00%	-68,44%	-70,36%	-82,11%	-80,96%	-81,38%	-77,17%

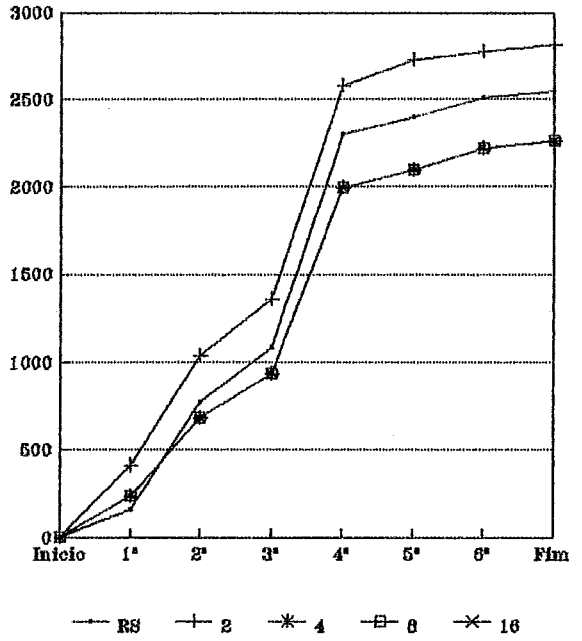
Tabela IV.6: Variação Percentual do Tempo do Seqüencial
— Paper —

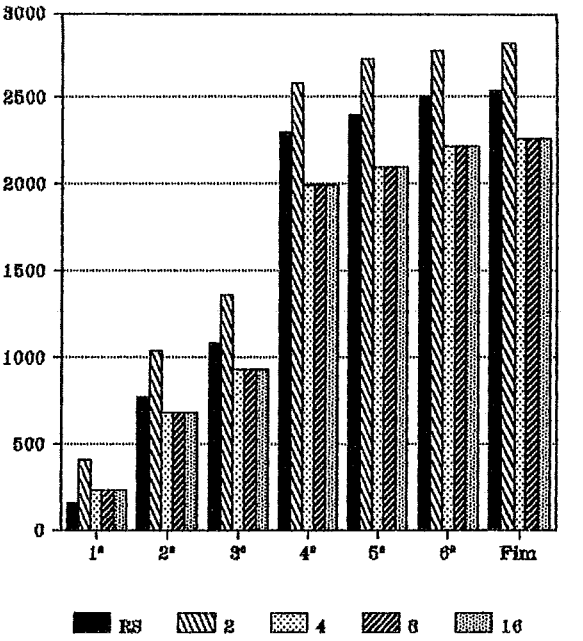
O modelo *Kabu-Wake*, por sua vez, ao explorar o *paralelismo or*, consegue tirar bastante proveito das características da árvore de busca associada a este programa. Sua vantagem se torna mais evidente na obtenção da primeira solução, que é a última dos demais modelos apresentados, obtida rapidamente (devido à busca em largura promovida pelo *paralelismo or* em uma cláusula) enquanto que simultaneamente os demais processadores progridem em busca de novas soluções. O modelo foi entre 40 e 82% mais veloz que o seqüencial a cada evento.

A classificação, nos mesmos moldes do teste anterior (segundo a tabela IV.7), nesse programa foi a seguinte :

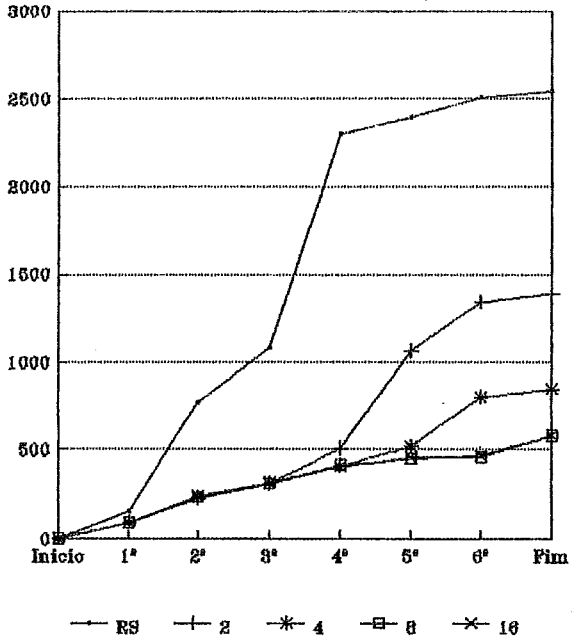
1º Kabu–Wake

Backup





Kabu-Wake



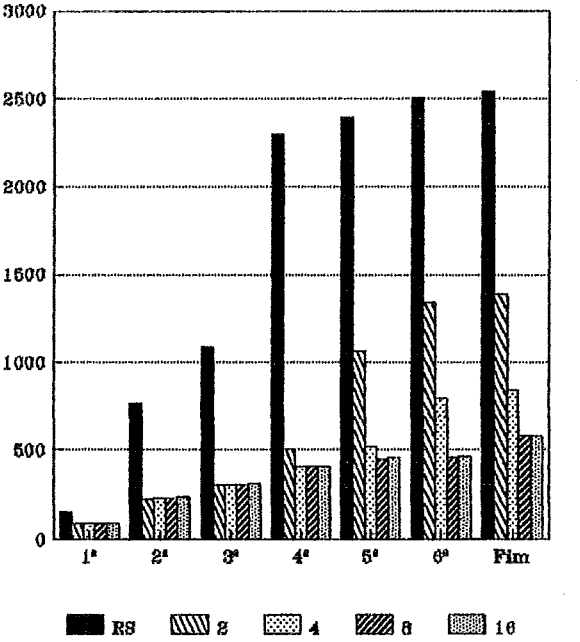


Gráfico IV.2: Tempo para Obtenção de Soluções e Término do Processamento
— Paper —

Modelo	NP	1ª	2ª	3ª	4ª	5ª	6ª	Fim
Seqüencial	1	155	770	1083	2298	2395	2503	2540
Backup	2	410	1036	1356	2578	2724	2771	2812
	4	232	682	936	1992	2093	2216	2257
	8	232	682	936	1992	2093	2216	2257
	16	232	682	936	1992	2093	2216	2257
Kabu-Wake	2	93	235	313	508	1059	1340	1390
	4	93	237	315	410	519	800	850
	8	93	239	315	416	454	464	580
	16	93	243	321	411	456	466	580

Tabela IV.7: Tempo para Obtenção de Soluções e Término do Processamento
— Paper —

2ª Backup

3ª Seqüencial

Densidade

O programa densidade fornece quais os pares de países, dentre os de sua lista, em que a população do primeiro país seja superior ao dobro da população do segundo país e que a área do primeiro país seja inferior à metade da área do segundo país. É gerado um sucesso para cada par de países encontrado e uma falha ao término da busca. Este programa possui complexidade da ordem do número de países ao quadrado (n^2), sendo portanto polinomial e, certamente, grande consumidor de recursos computacionais. Este fato atrai a atenção para estudos de otimização do tempo de processamento. Para a lista de países utilizada no programa são obtidas 3 soluções.

æ

O *paralelismo and* disponível está concentrado nas seguintes cláusulas :

questão com 3 metas, sendo as duas primeiras totalmente independentes e apenas a terceira possui variáveis compartilhadas com as demais;

cálculo com 4 metas, com dependências entre a 1ª e a 2ª e entre a 3ª e a 4ª metas

---

? *densidade* (*X*, *Y*, *Z*, *W*, *R*, *T*).

densidade (*X*, *Y*, *Z*, *W*, *R*, *T*) ←
    informação (*X*, *Y*, *Z*), *informação* (*W*, *R*, *T*), *cálculo* (*Y*, *Z*, *R*, *T*).

cálculo (*Y*, *Z*, *R*, *T*) ←
    multiplica (*R*, 2, *W*), *maior* (*Y*, *W*), *multiplica* (*Z*, 2, *V*), *menor* (*V*, *T*).

informação (*C*, *P*, *A*) ←
    população (*C*, *P*), *área* (*C*, *A*).

população (*china*, 8250).
população (*ussr*, 2521).
população (*india*, 5863).
população (*usa*, 2119).

área (*china*, 3380).
área (*ussr*, 8708).
área (*india*, 1139).
área (*usa*, 3609).

Algoritmo IV.7: Densidade Demográfica

(as duas primeiras são independentes das duas últimas); e

informação com duas metas que possuem uma dependência.

Desta forma esta modalidade de paralelismo está razoavelmente presente no programa, embora com muitas restrições causadas pelas constantes dependências entre variáveis. A modalidade *OR* de paralelismo tem presença em apenas duas funções com 4 cláusulas declaradas para cada, sendo todas fatos.

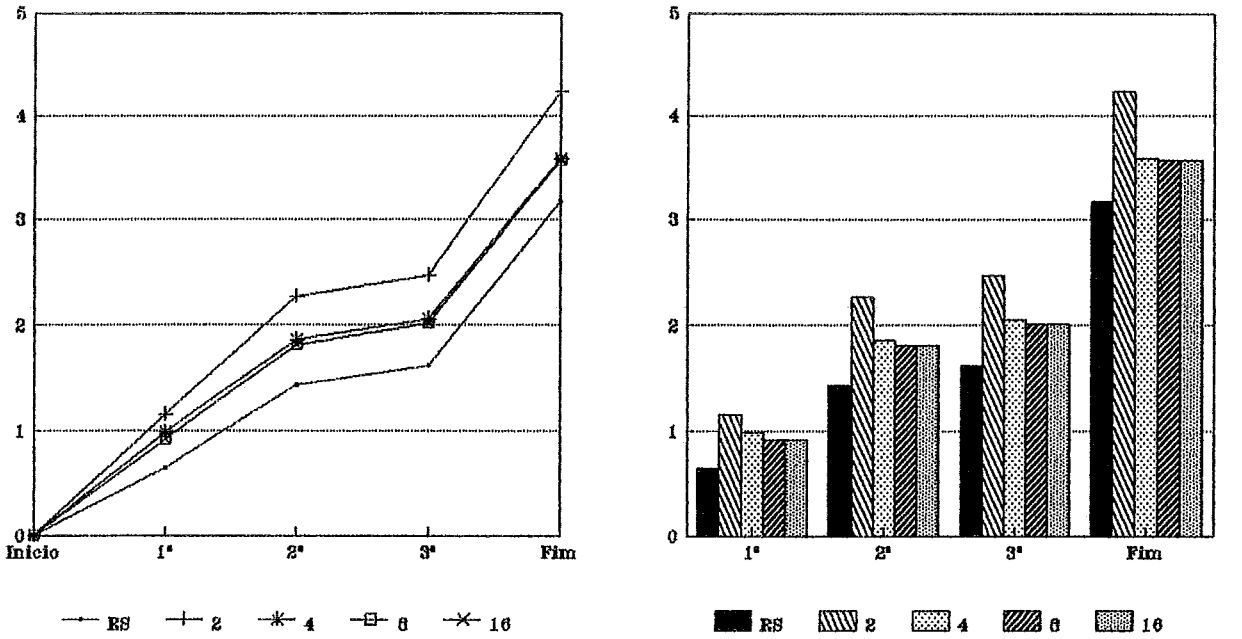
Este programa pode ser compreendido como um grande *AND* obtido pela junção dos corpos das cláusulas para *questão*, *cálculo* e *informação*. Para o modelo *Backup* este grande *AND* é composto por uma seqüência de *processos and*, cada qual com sua sub-árvore particular. Apesar do longo *AND*, em *cálculo* não existe retrocesso viável. Em *informação* só existem uma possibilidade adiante e mais 3 em retrocesos. O auge do paralelismo é obtido através da repetição de *informação* em *questão*, onde são realizadas as n^2 tentativas. Desta forma, tudo indica que o modelo *Backup* encontrou dificuldades em explorar paralelismo nesse programa devido a problemas de iniciação da sua árvore de processos, pois se sua primeira solução foi de 42 a 80% pior que o seqüencial (tabela IV.8, no término esta marca caiu para apenas de 12 a 33% pior, mostrando boa recuperação. Provavelmente, se o programa contivesse mais dados, o modelo superaria o seqüencial e certamente, devido à sensibilidade de PROLOG à ordem de declaração das cláusulas, qualquer mudança desta mudaria o desempenho do modelo, sendo possível obter marcas melhores. Ficou claro, contudo, que o fator determinante do mau desempenho do modelo foi o alto *overhead*, que sempre ficou na casa dos 44% (tabela IV.9), sendo o maior registrado e que supera em muito a diferença entre *Backup* e Seqüencial.

Para o modelo *Kabu–Wake* a característica do programa de formar um grande *AND* é extremamente favorável, por ser assim que este modelo trabalha. Assim como o modelo *Backup*, o *Kabu–Wake* teve problemas com a iniciação do programa, mas apenas com 2 e 4 processadores. Os melhores aresultados, obtidos com 8 e 16 processadores, situaram-se entre 20% melhor no início e 81% melhor ao final. O pior foi obtido pela simulação com 2 processadores que ficou entre 20% pior no início e 43% melhor que o seqüencial ao final.

Nesse programa a classificação dos modelos (segundo a tabela IV.10) foi a seguinte :

1º Kabu–Wake

Backup



Kabu-Wake

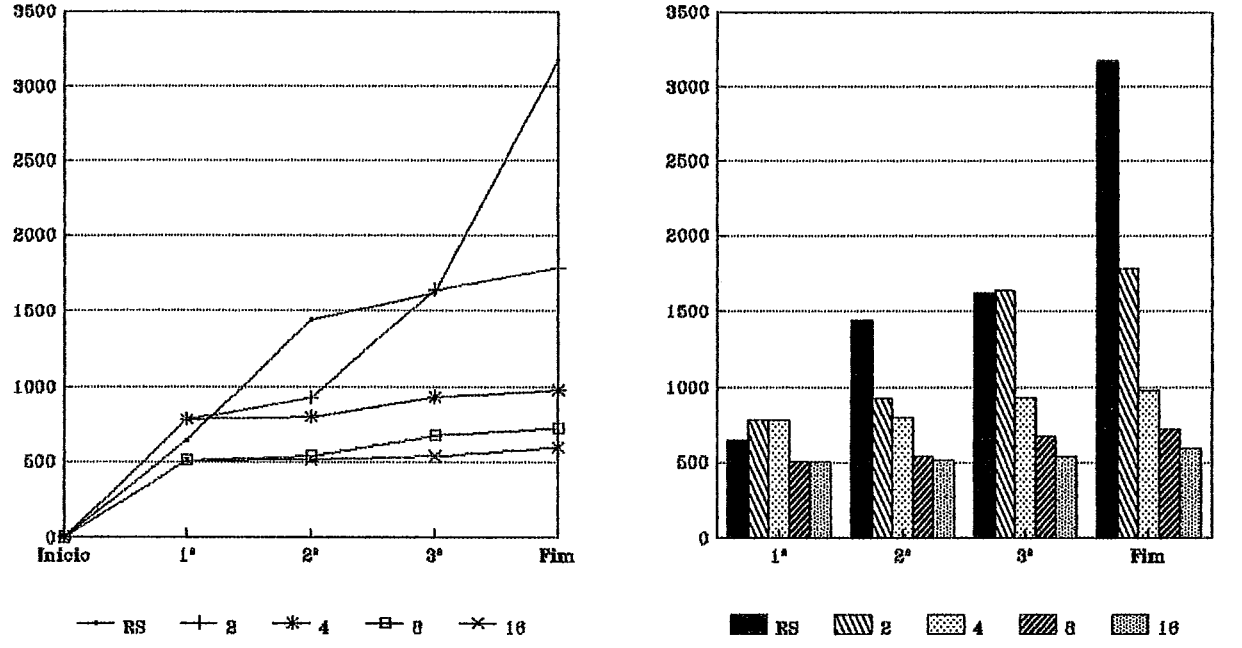


Gráfico IV.3: Tempo para Obtenção de Soluções e Término do Processamento
— Densidade —

Modelo	NP	1ª	2ª	3ª	Fim
Backup	2	80,19%	58,44%	52,72%	33,69%
	4	52,73%	29,71%	27,38%	13,07%
	8	42,75%	26,57%	24,60%	12,66%
	16	42,75%	26,57%	24,60%	12,66%
Kabu-Wake	2	20,59%	-35,36%	1,05%	-43,61%
	4	20,90%	-44,56%	-42,58%	-69,12%
	8	-20,59%	-62,76%	-58,78%	-77,36%
	16	-20,44%	-64,23%	-67,00%	-81,37%

Tabela IV.8: Variação Percentual do Tempo do Seqüencial
— Densidade —

Modelo	NP	Útil	Overhead
Backup	2	55,77%	44,23%
	4	55,07%	44,93%
	8	55,07%	44,93%
	16	55,06%	44,94%
Kabu-Wake	2	99,62%	0,38%
	4	99,43%	0,57%
	8	98,63%	1,37%
	16	97,96%	2,04%

Tabela IV.9: Distribuição Percentual do Tempo Efetivo de Processamento
— Densidade —

Modelo	NP	1ª	2ª	3ª	Fim
Seqüencial	1	641	1434	1618	3167
Backup	2	1155	2272	2471	4234
	4	979	1860	2061	3581
	8	915	1815	2016	3568
	16	915	1815	2016	3568
Kabu-Wake	2	773	927	1635	1786
	4	775	795	929	978
	8	509	534	667	717
	16	510	513	534	590

Tabela IV.10: Tempo para Obtenção de Soluções e Término do Processamento
— Densidade —

2º Seqüencial

3º Backup

Coloração

Este programa foi extraído de [Casanova 87] e determina quais as possíveis combinações de determinadas cores que poderiam colorir uma área plana dividida em regiões sem que duas dessas regiões vizinhas tenham a mesma cor. É gerado um sucesso para cada combinação encontrada e uma falha ao término das combinações possíveis. Igualmente ao programa de densidade, possui complexidade polinomial. O problema proposto está representado na figura IV.12, onde a área é dividida em 4 regiões e dispõe-se de 3 cores para preenchê-las. Existem 6 combinações de cores que satisfazem às condições do problema.

O *paralelismo and* presente no programa está todo contido em uma única cláusula que contém 5 metas. Existem dependências entre variáveis, entretanto são possíveis diversas combinações de metas que minimizam o problema. O *paralelismo or* também está presente nas 6 declarações de *vizinho*. A simplicidade deste programa ajuda a combinar as diferentes modalidades de paralelismo de modo a se obter um bom desempenho.

Para o modelo *Backup* o único ponto desfavorável neste programa advém

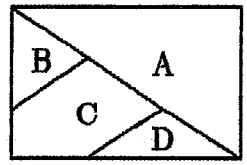


Figura IV.12: Mapa para Coloração

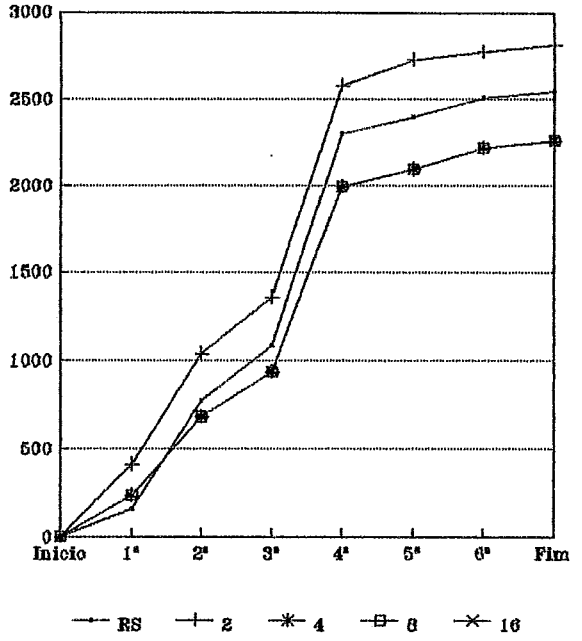
? *cores* (*A*, *B*, *C*, *D*).

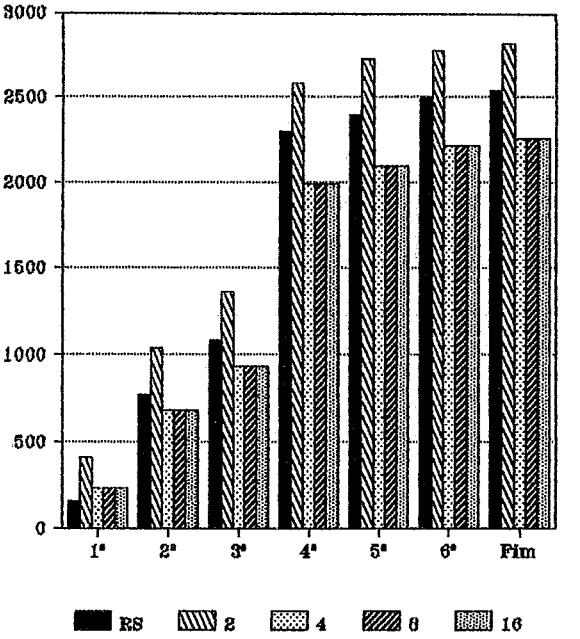
cores (*A*, *B*, *C*, *D*) ←
    vizinho (*A*, *B*), *vizinho* (*A*, *C*), *vizinho* (*A*, *D*), *vizinho* (*B*, *C*), *vizinho* (*C*, *D*).

vizinho (*branco*, *verde*).
vizinho (*verde*, *branco*).
vizinho (*verde*, *vermelho*).
vizinho (*vermelho*, *verde*).
vizinho (*vermelho*, *branco*).
vizinho (*branco*, *vermelho*).

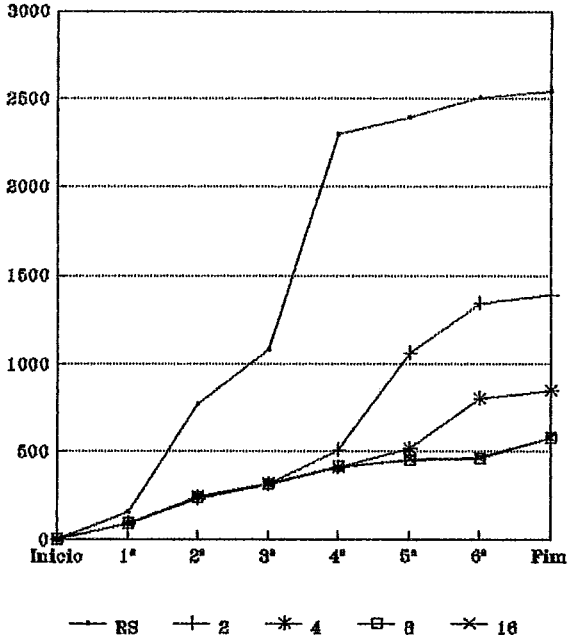
Algoritmo IV.8: Coloração de Mapas

Backup





Kabu-Wake



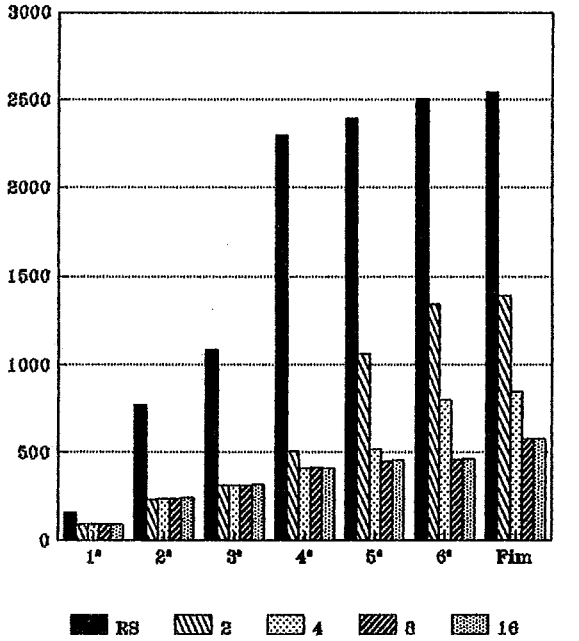


Gráfico IV.4: Tempo para Obtenção de Soluções e Término do Processamento
— Coloração —

do fato de o modelo criar processos or no mesmo processador do *processo and* que é o pai deles. É criada desta forma uma séria limitação na exploração de paralelismo pelo modelo neste programa, que só não é pior porque estes criam *processos and* em outro processador para resolver os fatos. Com 2 processadores o modelo foi pior que o seqüencial, mas com maior número de processadores foi pior apenas na primeira solução (5%), e oscilou entre 1 e 7% melhor da segunda em diante, de acordo com a tabela IV.11. Vale notar que devido às características da árvore gerada e à política de utilização de processadores pelo modelo, o mapeamento da árvore nescessita de 3 a 7 processadores. Qualquer número de processadores superior à este será desnecessário e inútil, obrigando que as simulações com 8 e 16 processadores obtenham sempre o mesmo tempo, o que ocorreu inclusive com 4 processadores, indicando que 4 tenha sido o número máximo de processadores usado pelo modelo.

Modelo	NP	1ª	2ª	3ª	4ª	5ª	6ª	Fim
Backup	2	35,24%	14,73%	12,12%	12,29%	12,92%	11,28%	7,59%
	4	5,17%	-2,61%	-1,15%	-2,50%	-4,92%	-6,12%	-7,84%
	8	5,17%	-2,61%	-1,15%	-2,50%	-4,92%	-6,12%	-7,84%
	16	5,17%	-2,61%	-1,15%	-2,50%	-4,92%	-6,12%	-7,84%
Kabu-Wake	2	28,04%	-49,49%	-30,17%	-48,63%	-40,51%	-50,66%	-48,21%
	4	18,45%	-50,22%	-66,78%	-76,11%	-63,62%	-67,69%	-69,64%
	8	-23,99%	-69,74%	-69,36%	-76,63%	-80,82%	-81,23%	-82,10%
	16	-23,62%	-69,45%	-77,85%	-83,47%	-86,18%	-86,32%	-86,82%

Tabela IV.11: Variação Percentual do Tempo do Seqüencial
— Coloração —

O modelo *Kabu–Wake* é muito beneficiado pela estrutura do programa, pois seu trabalho consistirá em duplicar o corpo de *cores* através do preenchimento das variáveis das 3 primeiras metas, e depois em verificar a existência de fatos equivalentes às duas últimas. Com 2 e 4 processadores o modelo encontrou alguma dificuldade na obtenção da primeira solução, provavelmente devido ao grande número de combinações possível, com tempos 28 e 18% piores, respectivamente. Os demais tempos situaram-se entre 30 e 86% melhores. Com 8 e 16 processadores, o comportamento previsto acima torna-se evidente, com algum volume de processamento até a primeira solução e a seguir as demais sendo obtidas em intervalos curtíssimos.

Os modelos obtiveram a seguinte classificação (segundo a tabela IV.12) nesse programa :

Modelo	NP	1ª	2ª	3ª	4ª	5ª	6ª	Fim
Seqüencial	1	542	1378	2095	2922	3639	4475	4872
Backup	2	733	1581	2349	3281	4109	4980	5242
	4	570	1342	2071	2849	3460	4201	4490
	8	570	1342	2071	2849	3460	4201	4490
	16	570	1342	2071	2849	3460	4201	4490
Kabu-Wake	2	694	696	1463	1501	2165	2208	2523
	4	642	686	696	698	1324	1446	1479
	8	412	417	642	683	698	840	872
	16	414	421	464	483	503	612	642

Tabela IV.12: Tempo para Obtenção de Soluções e Término do Processamento — Coloração —

1º Kabu-Wake

2º Backup

3º Seqüencial

IV.5.2 Avaliação do Desempenho dos Modelos

Backup

O desempenho do modelo *Backup* nos testes realizados foi relativamente ruim. Em dois programas (interseção e densidade) o desempenho foi inferior ao do seqüencial e nos programas em que conseguiu ser mais veloz que o seqüencial, só o foi à partir da segunda solução e por um fator máximo de 13%. Os desempenhos nos programas seguiram a seguinte ordem, do melhor para o pior :

1º Paper

2º Coloração

3º Densidade

4º Interseção

No que se refere ao uso de processadores, o modelo mostrou-se bastante limitado, devido ao fato de que o tempo de processamento normalmente tendeu a uma constante com o crescimento do número de processadores. Isto indica que o modelo tem distribuições de sub-árvores ótimas (para sí) e que atingindo estas distribuições não as modifica. O modelo se adapta ao número de processadores apenas quando este é menor que o ideal para sí, deixando ociosos os processadores que excederem a esse número, levando o modelo a ter sempre o mesmo desempenho para qualquer número de processadores superior ao seu ótimo. Como conseqüência da constância do tempo de processamento, a ociosidade tem crescimento proporcional ao número de processadores. Os programas paper e coloração (a partir de 4 processadores) e densidade (a partir de 8) ilustram bem este comportamento.

Como pontos falhos apresentados pelo modelo enfatizamos a limitação no uso de processadores e adicionamos :

- a manutenção de processos, que é normalmente muito onerosa[7];
- grande *overhead*[8], que oscilou entre 20 e 45% do processamento útil;
- a criação de processos filhos no mesmo processador do pai, que atrapalha a exploraçào do paralelismo e obriga ao uso de escalonadores nos processadores, incrementando o *overhead*;
- a criação de *processos and* para fatos, absolutamente desnecessária;
- a limitação do paralelismo imposta pelos *processos or*, que criam um novo filho apenas quando o atual termina; e
- a comunicação bloqueante, fator de restrição da exploração do paralelismo.

Embora os dois últimos pontos acima sejam usados pelos autores para controlar o paralelismo e evitar uma possível explosão no número de processos, os resultados mostram que houve excesso de precaução, pelo menos no que se refere aos programas testados. Para o modelo tornar-se competitivo é necessário que algumas dessas restrições sejam relaxadas.

[7]Vide a tabela IV.2

[8]Somando o com processos ao com comunicações

Um fator que pode ser considerado favorável ao modelo é este manter a estratégia de busca PROLOG. Embora possa ser também considerada como um defeito por restringir o paralelismo, esta estratégia é a mais conhecida pelos programadores, que a usam em diversos programas, fazendo surgir respostas erradas quando a mesma não é seguida. É plenamente discutível se este fator realmente constitui vantagem.

Realmente, o modelo *Backup*, um dos primeiros a ser publicado, talvez pelas limitações existentes na época, não seja, na forma em que foi apresentado, um modelo competitivo. É provável que apresente em alguns programas, mais particulares e maiores, melhor desempenho sem entretanto exibir valores magníficos de desempenho. Com algumas alterações para incrementar a exploração de paralelismo é possível que para uma classe maior de programas o modelo possa obter melhor desempenho e justificar a sua utilização.

Kabu–Wake

O modelo *Kabu-Wake* obteve, em todos os programas simulados, desempenho superior tanto ao do modelo *Backup* quanto ao do seqüencial. Em poucos eventos (temporários) este modelo foi superado e em 3 dos 4 programas chegou a ser 80% (ou 5 vezes) mais rápido que o seqüencial e no quarto foi 50% (ou 2 vezes) melhor. Desta forma pode-se considerar como satisfatório o desempenho do modelo. A classificação dos programas, segundo o desempenho do modelo, foi a seguinte :

1º Coloração

2º Densidade

3º Paper

4º Interseção

O uso dos processadores por parte do modelo foi sempre razoavelmente eficiente, sempre proporcional à quantidade de trabalho disponível. À favor deste argumento está o fato de a ordem de desempenho ser idêntica à de maior percentual de processamento útil (ou menor tempo ocioso). Em muitas das simulações o processador que inicia o processamento trabalha perto de 100% do tempo e a utilização dos demais varia, às vezes em função da posição na rede em anel (figura IV.7), às

vezes aleatoriamente, tudo depende da árvore gerada. O único ponto invariável é que sempre existe pelo menos um processador servindo de produtor para os demais. Registrou-se apenas um caso, em interseção, onde o aumento do número de processadores decididamente atrapalhou o desempenho do modelo. Tal fato deveu-se, provavelmente, a uma pulverização dos objetivos, sendo estes excessivamente distribuídos. Desta forma o modelo passou a maior parte do tempo distribuindo objetivos, e deixando os processadores ociosos. Em algus outros casos, este problema foi localizado em um ou outro evento, normalmente com diferenças de tempo pouco significativas.

Um ponto que merece destaque como possível problema do modelo, sem que isso signifique que seja uma falha do mesmo, é o de ocorrerem adversidades com programas que foram escritos para rodarem particularmente em PROLOG. Como o modelo não obedece à ordem de avaliação desta linguagem, determinados programas fornecem soluções que não satisfazem ao enunciado do programa, mas no entanto estas falsas soluções seriam obtidas se pedidas novas soluções em PROLOG. O problema está em se esperar que seja obtida apenas a primeira solução, alcançada por busca em profundidade à esquerda, e que depois pare o processamento, o que para o modelo *Kabu–Wake* é impossível. Surge então um problema de compatibilidade com alguns programas existentes.

Outro ponto significativo, que pode ser considerado um entrave à utilização do modelo *Kabu–Wake*, é a sua arquitetura peculiar, que embora praticamente elimine a existência de *overhead* (pois a absoluta maioria da comunicação é realizada pelo co-processador e não existem processos), exige processadores e redes de comunicação de dados específicos e altamente complexos. Certamente seria possível adaptar este sistema em arquiteturas hipercúbicas de memória privativa (ex. rede de TRANSPUTERS), entretanto forçosamente seria introduzido algum *overhead*.

Como ponto forte do modelo (além de desempenho, baixo *overhead* e boa utilização de recursos), destaca-se a possibilidade de serem executados simultaneamente diversos programas diferentes, ou mesmo diversas consultas para um mesmo programa. Devido às características do modelo, este tipo de utilização, na maioria dos casos, não deve provocar grande desaceleração nos programas e certamente baixaria a ociosidade. É evidente que seria aconselhável a junção de programas apenas dispondo-se de muitos processadores e para programas que trabalhem com grande ociosidade. Esta possibilidade de unir programas muito provavelmente resultaria em menor tempo para obter-se todos os resultados que a soma dos tempos indivi-

duais dos programas, sendo portanto um modo de otimizar o tempo e o uso dos processadores.

O modelo *Kabu–Wake* demonstrou possuir diversas qualidades que podem ser muito bem aproveitadas na exploração de paralelismo em Programação Lógica. Apesar de necessitar de um *hardware* complexo e possivelmente inviável para a maioria dos usuários, há a possibilidade deste modelo ser implementado em outras máquinas. No mínimo, pode-se usá-lo como base na confecção de outros modelos mais próximos do estágio atual de desenvolvimento do *hardware*. Certamente este modelo apresenta grandes contribuições à execução paralela de Programas Lógicos.

IV.5.3 Backup x Kabu–Wake

No ítem desempenho o modelo *Kabu–Wake* foi amplamente superior ao *Backup*. Registrou melhores tempos, usou melhor os recursos disponíveis e teve menor *overhead*. A sua superioridade neste item é, portanto, indiscutível. Em termos de facilidade de implementação ambos são razoavelmente simples, demandando pouco trabalho para realizar a implementação de suas principais rotinas (aquelas que caracterizam o modelo). Neste ponto vale ressaltar que o modelo *Backup* é mais fácil de ser modificado, aceitando mais alterações que o *Kabu–Wake*, que é mais fechado. Uma outra vantagem do modelo *Backup* é a maior facilidade para adaptá-lo a um *hardware* qualquer, provavelmente provocando poucas mudanças em seu desempenho. A mesma tarefa se realizada para o modelo *Kabu–Wake*, certamente seria mais complexa e provocaria danos muito maiores ao seu desempenho. Mesmo assim, podemos concluir que o modelo *Kabu–Wake*, de um modo geral, é superior ao *Backup*, sem que isto signifique que seja uma melhor referência para estudos ou uma melhor base para implementações em *hardware*.

Capítulo V

Conclusão

O Simulador de Modelos Paralelos de Programação em Lógica revelou-se uma ferramenta bastante adeqüada e útil para o estudo e a análise de modelos. Mesmo com poucos recursos, não é obrigatório o uso de multi-processadores, é possível realizar a simulação de programas de razoável complexidade, que poderá crescer em função dos recursos disponíveis (principalmente memória).

Os resultados apresentados mostraram-se coerentes e, na sua maioria, muito próximos do esperado. Mesmo onde os resultados representavam surpresa, sempre foi possível verificar a veracidade dos mesmos, lembrando-nos que nossas estimativas, por um mero detalhe, podem estar erradas. A complexidade dos modelos combinada à existente nos programas está, muitas vezes, além da nossa capacidade de realizar análises, tornando obrigatório o uso de ferramentas adeqüadas. A realidade apontada pelo simulador torna-se mais amena, e está sempre dentro da nossa capacidade de compreensão. Os dados obtidos pelo simulador são tratáveis de diversas maneiras, sendo possível realizar as mais complexas e detalhadas análises.

A facilidade de uso é outra característica atraente do simulador. A implementação de um modelo é muito facilitada pelo ambiente provido, de modo que apenas as características do modelo devem ser programadas, reduzindo em muito o trabalho. Do mesmo modo o simulador suporta modelos muito diferentes, orientados para *hardwares* específicos, mostrando ser versátil. Os modelos implementados (Seqüencial, *Backup* e *Kabu–Wake*) demonstram esta versatilidade.

Sendo assim só nos resta concluir que o papel do simulador foi devidamente cumprido. Através do uso do simulador, as reais capacidades dos modelos implementados e a superioridade do *Kabu–Wake* sobre o *Backup* foram reveladas e quantificadas. Este é apenas mais um passo no estudo de paralelismo em Programação em Lógica. Como continuação deste trabalho sugerimos :

- testar os modelos *Kabu–Wake* e *Backup* com diversas questões / programas simultaneamente;
- realizar as alterações do modelo *Backup* propostas na seção IV.5.2; e
- simular outros modelos como [Conery 87] e [Bianchini 90].

Todo este trabalho faz parte de uma obra maior que é a definição de um modelo com alto desempenho para o projeto de uma máquina paralela de inferência. Após todos os estudos necessários este modelo será elaborado e certamente implementado, testado e melhorado no Simulador de Modelos Paralelos de Programação em Lógica.

Appendix A

Sintaxe para os Programas PROLOG

<programa>	::-	<objetivos> <cláusulas> | <cláusulas> <objetivos>
<objetivos>	::-	<objetivo> [<objetivo>]*
<objetivo>	::-	? <literais> .
<literais>	::-	<literal> [, <literal>]*
<literal>	::-	<identificador> [(<termos>)]
<termos>	::-	<termo> [, <termo>]*
<termo>	::-	<constante> | <variável> | <lista> | <termo_composto>
<termo_composto>	::-	<identificador> (<termos>)
<identificador>	::-	<minúscula> [<minúscula> | <dígito> | _]*
<dígito>	::-	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<minúscula>	::-	a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z
<lista>	::-	'[' ']' | '[' <termos> ']' | '[' <variável> '|' <variável> ']'
<variável>	::-	<maiúscula> [<maiúscula> | <dígito> | _]*
<maiúscula>	::-	A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
<constante>	::-	<identificador> | <número>
<número>	::-	[<sinal>] <dígitos>
<dígitos>	::-	<dígito> [<dígito>]*
<sinal>	::-	+ | —
<cláusulas>	::-	<cláusula> [<cláusula>]*
<cláusula>	::-	<fato> | <regra>
<regra>	::-	<literal> :- <literais> .
<fato>	::-	<literal> .

Appendix B

Funções PROLOG Pré-definidas

Descrevemos abaixo as primitivas consideradas fundamentais para o desenvolvimento de programas PROLOG que serão supridas pelo simulador :

$is(X,Y)$: falha se X e Y forem instanciadas com valores diferentes, caso contrário sucede e, se necessário, atribui o valor à não instanciada;

$eq(X,Y)$: sucede se as variáveis estiverem instanciadas com valores idênticos;

$ne\ (X,Y)$: sucede se instanciadas com valores diferentes;

$le\ (X,Y)$: sucede se $X \leq Y$;

$lq\ (X,Y)$: sucede se $X < Y$;

$gt\ (X,Y)$: sucede se $X > Y$;

$ge\ (X,Y)$: sucede se $X \geq Y$;

$add\ (X,Y,R)$: sucede fazendo $R \leftarrow X + Y$, falha se X ou Y não estiverem instanciadas, se o forem com valores não numéricos ou se R estiver instanciada;

$sub\ (X,Y,R)$: sucede fazendo $R \leftarrow X - Y$, falha se X ou Y não estiverem instanciadas, se o forem com valores não numéricos ou se R estiver instanciada;

$mult\ (X,Y,R)$: sucede fazendo $R \leftarrow X * Y$, falha se X ou Y não estiverem instanciadas, se o forem com valores não numéricos ou se R estiver instanciada;

$div\ (X,Y,R)$: sucede fazendo $R \leftarrow X/Y$, falha se X ou Y não estiverem instanciadas, se o forem com valores não numéricos ou se R estiver instanciada;

mod (X, Y, R) : sucede fazendo $R \leftarrow X \ mod \ Y$, falha se X ou Y não estiverem instanciadas, se o forem com valores não numéricos ou se R estiver instanciada;

Appendix C

Estrutura de Dados, Funções e Procedimentos da Interface do Simulador

C.1 Rotinas Globais

void
Mostre (*formato*, ...)
    char * *formato*;
{

- Exibe mensagens;
- Sintaxe idêntica a de *printf*.

}

void
TerminaSimulação (*status*)
    int status;
{ }

void *
AlocaMemória (*tamanho*)
    unsigned int tamanho;
{ }

```
void
LiberaMemória (endereço)
    void * endereço;
{ }

SEMÁFORO *
AlocaSemáforo (valor_inicial)
    int valor_inicial;
{
```

- Cria um semáforo;
- O inicia com *valor_inicial*.

```
}

void
LiberaSemáforo (semáforo)
    SEMÁFORO ** semáforo;
{ }

void
Sinaliza (semáforo)
    SEMÁFORO * semáforo;
{
```

- Equivalente a *Signal*.

```
}
```

```
void
Aguarda (semáforo)
    SEMÁFORO * semáforo;
{
```

- Equivalente a *Wait*.

```
}
```

C.2 Interpretação PROLOG

```
#define SUCESSO      VERDADE
#define FALHA        FALSO

#define CONST(t)     ((t-> tipo = NUM_CONST)OR(t-> tipo = ALFA_CO

enum tipo_termo {
    NUM_CONST,
    ALFA_CONST,
    FUNÇÃO,
    LISTA,
    CONSTRUÇÃO,
    VARIÁVEL
}; typedef enum tipo_termo TIPO_TERMO;

enum tipo_substituição {
    INSTANCIAÇÃO,
    REFERÊNCIA,
    NÃO_INSTANCIADA
}; typedef enum tipo_substituição TIPO_SUBSTITUIÇÃO;

struct termo_função {
    char * identificador;
    struct termo * termos;
};
```

```
struct termo_num_const {
    double valor;
};

struct termo_alfa_const {
    char * string;
};
struct termo_construção {
    long variável;
    char * string;
};

struct termo_lista {
    struct termo * termos;
};

struct termo_variável {
    long variável;
    char * string;
};

struct termo {
    TIPO_TERMO tipo;
    union {
        struct termo_função f;
        struct termo_num_const n;
        struct termo_alfa_const a;
        struct termo_construção c;
        struct termo_lista l;
        struct termo_variável v;
    } u;
    struct termo * próximo;
}; typedef struct termo TERMO;
```

```
struct literal {
    char * identificador;
    int aridade;
    TERMO * termo;
    struct literal * próximo;
}; typedef struct literal LITERAL;

struct substituição {
    long variável;
    char * string;
    TIPO_SUBSTITUIÇÃO tipo;
    union {
        TERMO * termo;
        long referência;
    } u;
    struct substituição * próxima;
}; typedef struct substituição SUBSTITUIÇÃO;

struct cláusula {
    LITERAL * cabeça;
    LITERAL * corpo;
    struct cláusula * próxima;
}; typedef struct cláusula CLÁUSULA;


CLÁUSULA *
CopiaCláusula (cláusula)
    CLÁUSULA * cláusula;
{ }

void
MostreCláusula (cláusula)
    CLÁUSULA * cláusula;
{ }
```

```
void
LiberaCláusula (cláusula)
    CLÁUSULA ** cláusula;
{ }

LITERAL *
CopiaLiterais (literal)
    LITERAL * literal;
{ }

LITERAL *
CopiaLiteral (literal)
    LITERAL * literal;
{ }

void
MostreLiterais (literal)
    LITERAL * literal;
{ }

void
MostreLiteral (literal)
    LITERAL * literal;
{ }

void
LiberaLiteral (literal)
    LITERAL ** literal;
{ }

TERMO *
CopiaTermos (termo)
    TERMO * termo;
{ }
```

```
TERMO *
CopiaTermo (termo)
    TERMO * termo;
{ }

void
MostreTermo (termo)
    TERMO * termo;
{ }

void
LiberaTermo (termo)
    TERMO ** termo;
{ }

SUBSTITUIÇÃO *
CopiaSubstituições (substituição)
    SUBSTITUIÇÃO * substituição;
{ }

void
MostreSubstituições (substituição)
    SUBSTITUIÇÃO * substituição;
{ }

void
LiberaSubstituições (substituição)
    SUBSTITUIÇÃO ** substituição;
{ }
```

int
NúmeroLiteraisCláusula (*cláusula*)
    CLÁUSULA * *cláusula*;
{

- Fornece o número de literais contidos em *cláusula.*

}

LITERAL *
PegaLiteralI (*i*, *cláusula*)
    int i;
    CLÁUSULA * *cláusula*;
{

- Fornece o *i*-ésimo literal de *cláusula.*

}

void
SubstituiLiteral (*literal*, *substituição*)
    LITERAL * *literal*;
    SUBSTITUIÇÃO * *substituição*;
{

- Aplica *substituição* somente ao primeiro literal de *literal.*

}

void
SubstituiLiterais (*literal*, *substituição*)
    LITERAL * *literal*;
    SUBSTITUIÇÃO * *substituição*;
{

- Aplica *substituição* a todos os literais de *literal*.

}

CLÁUSULA *
EncontraCláusula (*partida*, *literal*, *substituição*, *cláusula*, *processo*)
    CLÁUSULA * *partida*;
    LITERAL * *literal*;
    SUBSTITUIÇÃO ** *substituição*;
    CLÁUSULA ** *cláusula*;
    PROCESSO * *processo*;
{

- Fornece em *cláusula* uma cláusula que unifique com *literal*;
- A lista de substituições da unificação é fornecida em *substituição*;
- *partida* deve ser *NULL* na primeira chamada, e conter o valor retornado pela função nas demais;
- Quando a função retorna *NULL* é porque não existe cláusula que unifique com *literal*;
- *processo* é fornecido para que possa ser controlado.

}

SUBSTITUIÇÃO *
UneSubstituições (*primeira*, *segunda*)
    SUBSTITUIÇÃO * *primeira*;
    SUBSTITUIÇÃO * *segunda*;
{

- Inclui a lista *segunda* no final da lista *primeira*;

}

SUBSTITUIÇÃO *
CombinaSubstituições (*base*, *complemento*)
    SUBSTITUIÇÃO * *base*;
    SUBSTITUIÇÃO * *complemento*;
{

- Adiciona a lista *complemento* à lista *base*;
- Aplica a lista resultante à própria lista resultante.

}

SUBSTITUIÇÃO *
ExtraiSubstituições (*lista*, *substituições*)
    SUBSTITUIÇÃO * *lista*;
    SUBSTITUIÇÃO * *substituições*;
{

- Cria uma nova lista com as variáveis de *lista* contidas em *substituições*, que estiverem instanciadas.

}

SUBSTITUIÇÃO *

PegaVariáveisLiteral (*literal*)

LITERAL * *literal*;

{

- Fornece uma lista com as variáveis de em literal.

}

CLÁUSULA *

PegaQuestão (*partida*)

CLÁUSULA * *partida*;

{

- Retorna uma questão do banco de cláusulas;
- *partida* deve ser *NULL* na primeira chamada e conter a cláusula retornada nas demais;
- Retorna *NULL* quando não existirem mais questões.

}

C.3 Estatísticas e Medições

#define MSG_PROCESSOS 0x0001
#define MSG_ESTATUS 0x0002
#define MSG_TEMPO 0x0004

enum amostragem {
    PERÍODO,
    SOLUÇÃO,
    FINAL
}; *typedef enum amostragem AMOSTRAGEM*;

void
AjustaMensagens (*modo*)
    int modo;
{

- Define que tipos de mensagens serão emitidas;
- Configurado pelos *bits* em *MSG_tipo*.

}

void
MostreTrabalho (*amostragem*)
    AMOSTRAGEM amostragem;
{

- Emite informações sobre o processamento;
- *amostragem* determina que tipo de informação está sendo emitida.

}

C.4 Processos e Processadores

```
#define TICK            1L

enum operações {
    UNIFICAÇÃO,
    ENVIA_LOCAL,
    ENVIA_REMOTA,
    RECEBE,
    CRIA_LOCAL,
    CRIA_REMOTO,
    TERMINATE,
    BUSCA,
    USUÁRIO
}; typedef enum operações OPERAÇÕES;

enum estatus_processador {
    PROCESSANDO,
    PARADO,
    AGUARDANDO
}; typedef enum estatus_processador ESTATUS_PROCESSADOR;

enum estatus_processo {
    MORTO,
    EXECUTANDO,
    PRONTO,
    BLOQUEADO
}; typedef enum estatus_processo ESTATUS_PROCESSO;

struct endereço {
    PROCESSADOR * virtual;
    int número_processo;
}; typedef struct endereço ENDEREÇO;
```

```
struct mensagem {
        ENDEREÇO remetente;
        ENDEREÇO destinatário;
        BOOLEAN aguarda;
        int tamanho;
        void * buffer;
}; typedef struct mensagem MENSAGEM;

struct processo {
        ENDEREÇO endereço;
        ENDEREÇO pai;
        long começo;
        long fim;
        long * operações;
}; typedef struct processo PROCESSO;

struct processador {
        int processo_criados;
        long * operações;
        long tempo_perdido;
}; typedef struct processador PROCESSADOR;


void
AjustaTimeSliceProcesso (processo, time_slice)
        PROCESSO * processo;
        int time_slice;
{
```

- Define o *time_slice* de *processo*.

```
}
```

void
*Ajusta*__TimeSliceDefault (time_slice)__
    int __time_slice;__
{

- Define o *time_slice* que será utilizado nos processos criados à partir da chamada da função.

}

PROCESSO *
CriaProcesso (*pai*, *processador*, *função*, *número_parâmetros*, ...)
    PROCESSO * *pai*;
    PROCESSADOR * *processador*;
    void (* *função*) ();
    int número_parâmetros;
{

- Cria um processo, filho de *pai* (que deve ser obrigatoriamente quem executa a rotina), para *função*;
- O processo é criado no processador *processador*;
- São passados para o processo os parâmetros seguintes a *número_parâmetros*.

}

void
TerminaProcesso (*processo*)
    PROCESSO * *processo*;
{

- Computa, para o simulador, o término de um processo;
- Deve ser executada somente por *processo*.

}

PROCESSADOR *

Processador (*número*)

int número;

{

- Fornece o endereço do processador *número*.

}

long

Tick*Corrente* ()

{ }

ESTATUS_PROCESSADOR

EstatusProcessador (*processador*)

PROCESSADOR * *processador*;

void

Processa (*processo*, *operação*)

PROCESSO * *processo*;

int operação;

{

- Sincroniza *processo* pelo tempo de *operação*, que deve ser definida pelo modelo.

}

void

Sincroniza (*processo*)

PROCESSO * *processo*;

{

- Sincroniza um processo com a sua criação;
- Deve obrigatoriamente ser a primeira intrução da função de *processo*.

}

```
void
EnviaMensagem (remetente, destinatário, espera, tamanho, mensagem)
    PROCESSO * remetente;
    ENDEREÇO * destinatário;
    BOOLEAN espera;
    int tamanho;
    void * mensagem;
{
```

- Envia *mensagem* de *remetente* para *destinatário*;
- *tamanho* contém o tamanho da mensagem;
- *espera* informa se deseja aguardar pelo recebimento da mensagem.

```
}

void *
RecebeMensagem (processo, remetente, espera)
    PROCESSO * processo;
    ENDEREÇO * remetente;
    BOOLEAN espera;
{
```

- *processo* realiza o recebimento de uma mensagem de *remetente*;
- Se *remetente* for *NULL*, recebe qualquer mensagem;
- *espera* define se deseja aguardar pela chegada da mensagem;
- Se *espera* for *FALSE* e não existir mensagem, retorna *NULL*.

```
}
```

void
CriaProcessadores (*número*)
    int número;
{

- Cria *número* processadores virtuais;
- Deve fazer parte da rotina de iniciação do modelo (*IniciaModelo*).

}

C.5 Rotinas Definidas no Modelo

void
IniciaModelo (*número_processadores*)
    int número_processadores;
{

- Inicia o modelo;
- Cria os processadores virtuais via *CriaProcessadores* de acordo com *número_processadores*;
- Cria os processos iniciais do modelo via *CriaProcesso*, onde o parâmetro *pai* recebe *NULL*.

}

Appendix D

Tabelas

Modelo	NP	Ocioso	Útil	Comunicação	Processos
Backup	2	46,05%	34,71%	8,82%	10,42%
	4	72,36%	17,61%	4,64%	5,39%
	8	86,15%	8,83%	2,32%	2,70%
	16	93,05%	4,43%	1,17%	1,36%
Kabu-Wake	2	33,77%	65,80%	0,44%	0,00%
	4	63,86%	35,76%	0,38%	0,00%
	8	76,78%	22,85%	0,36%	0,00%
	16	93,22%	6,63%	0,16%	0,00%

Tabela D.1: Distribuição Percentual do Tempo Total
— Interseção —

Modelo	NP	Útil	Overhead
Backup	2	80,15%	19,85%
	4	79,50%	20,50%
	8	79,47%	20,53%
	16	79,50%	20,50%
Kabu-Wake	2	99,45%	0,55%
	4	99,06%	0,94%
	8	98,53%	1,47%
	16	97,92%	2,08%

Tabela D.2: Distribuição Percentual do Tempo Efetivo de Processamento
— Paper —

Modelo	NP	Ocioso	Útil	Comunicação	Processos
Backup	2	43,66%	45,16%	5,67%	5,51%
	4	64,61%	28,13%	3,73%	3,52%
	8	82,30%	14,07%	1,87%	1,77%
	16	91,15%	7,03%	0,93%	0,88%
Kabu-Wake	2	8,16%	91,33%	0,50%	0,00%
	4	24,65%	74,64%	0,71%	0,00%
	8	44,52%	54,66%	0,82%	0,00%
	16	72,05%	27,37%	0,58%	0,00%

Tabela D.3: Distribuição Percentual do Tempo Total
— Paper —

Modelo	NP	Ocioso	Útil	Comunicação	Processos
Backup	2	33,31%	37,19%	14,32%	15,18%
	4	59,85%	22,11%	8,84%	9,20%
	8	79,85%	11,10%	4,44%	4,62%
	16	89,92%	5,55%	2,22%	2,31%
Kabu-Wake	2	11,00%	88,66%	0,34%	0,00%
	4	18,63%	80,91%	0,46%	0,00%
	8	44,06%	55,17%	0,77%	0,00%
	16	65,79%	33,51%	0,70%	0,00%

Tabela D.4: Distribuição Percentual do Tempo Total
— Densidade —

Modelo	NP	Útil	Overhead
Backup	2	74,62%	25,38%
	4	73,65%	26,35%
	8	73,64%	26,36%
	16	73,65%	26,35%
Kabu-Wake	2	99,71%	0,29%
	4	99,39%	0,61%
	8	99,02%	0,98%
	16	98,38%	1,62%

Tabela D.5: Distribuição Percentual do Tempo Efetivo de Processamento
— Coloração —

Modelo	NP	Ocioso	Útil	Comunicação	Processos
Backup	2	37,73%	46,47%	7,75%	8,05%
	4	63,17%	27,13%	4,84%	4,87%
	8	81,58%	13,56%	2,42%	2,44%
	16	90,79%	6,78%	1,21%	1,22%
Kabu-Wake	2	3,19%	96,53%	0,28%	0,00%
	4	17,18%	82,31%	0,51%	0,00%
	8	29,49%	69,82%	0,69%	0,00%
	16	51,81%	47,41%	0,78%	0,00%

Tabela D.6: Distribuição Percentual do Tempo Total
— Coloração —

Appendix E

Gráficos das Simulações

Neste apêndice são apresentados os gráficos descritos na seção III.8, confeccionados a partir dos dados obtidos nas simulações dos modelos e utilizados nas suas análises de desempenho, realizada na seção IV.5.

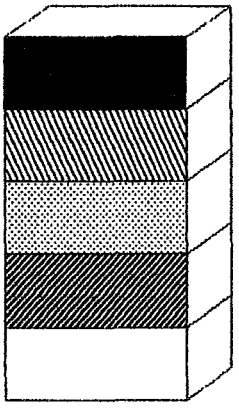
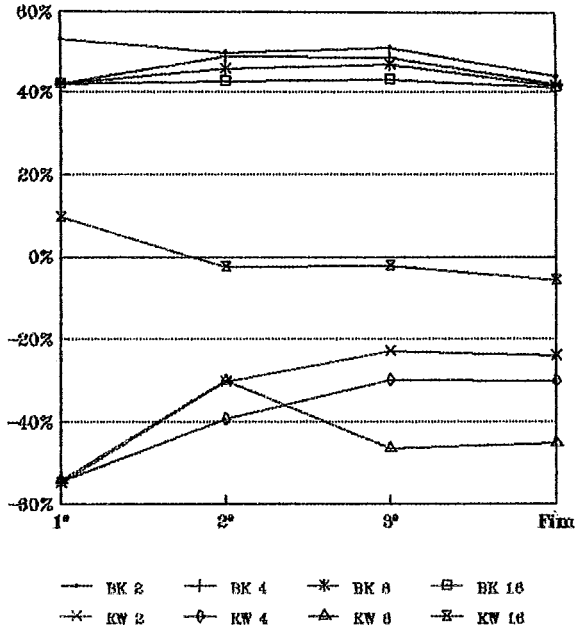


Figura E.1: Legenda da Distribuição Percentual do Tempo



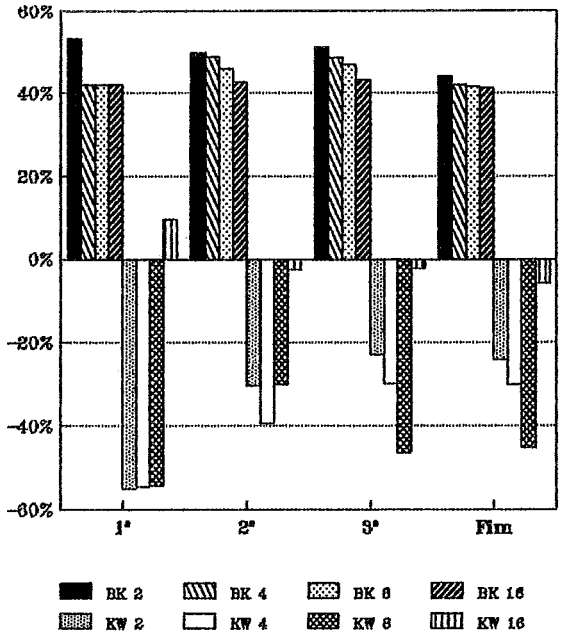
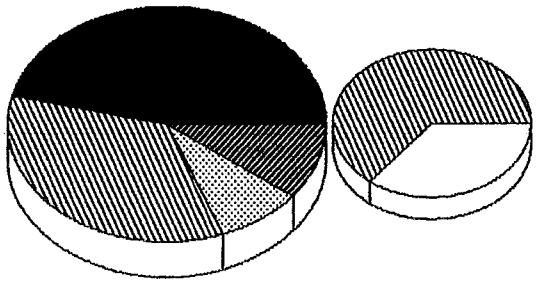
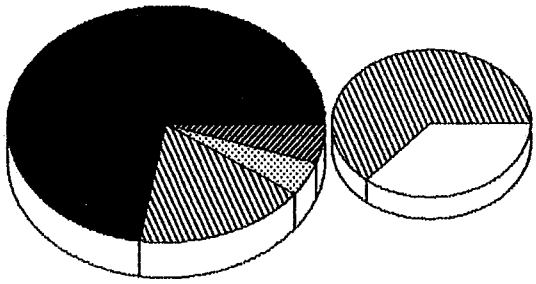


Gráfico E.1: Variação Percentual do Tempo do Seqüencial
— Interseção —

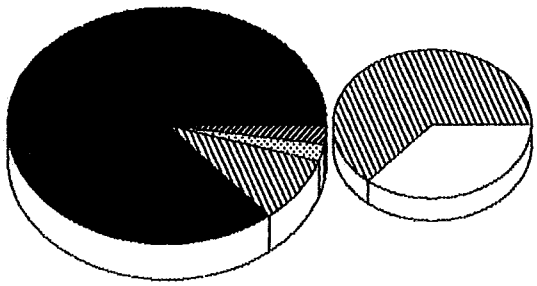
2 Processadores



4 Processadores



8 Processadores



16 Processadores

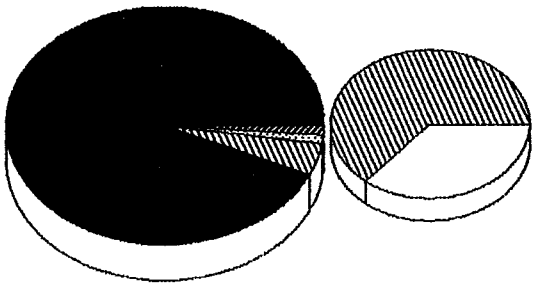
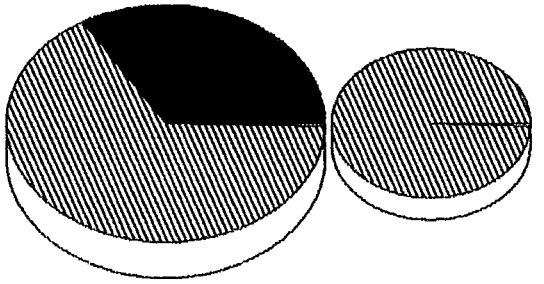
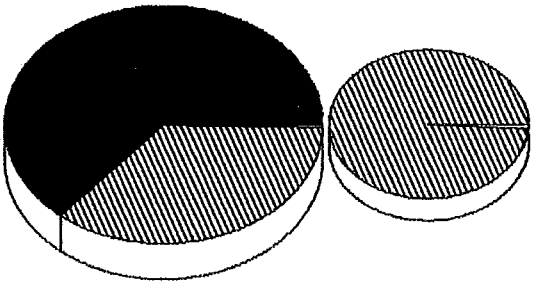


Gráfico E.2: Distribuições Percentuais dos Tempos Total e Efetivo de Processamento
Backup
— Interseção —

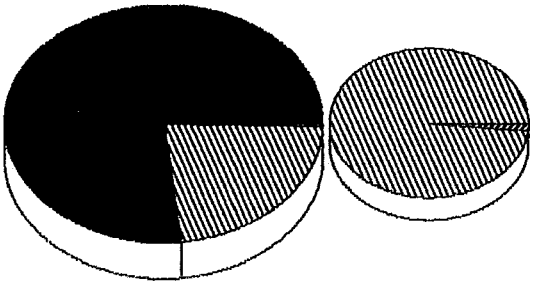
2 Processadores



4 Processadores



8 Processadores



16 Processadores

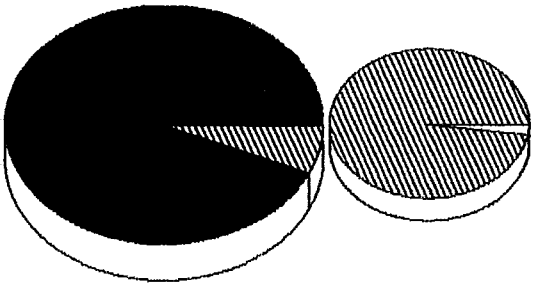
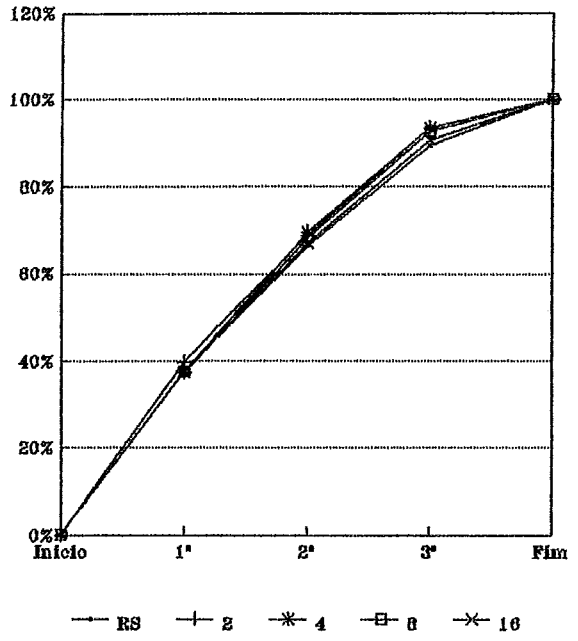
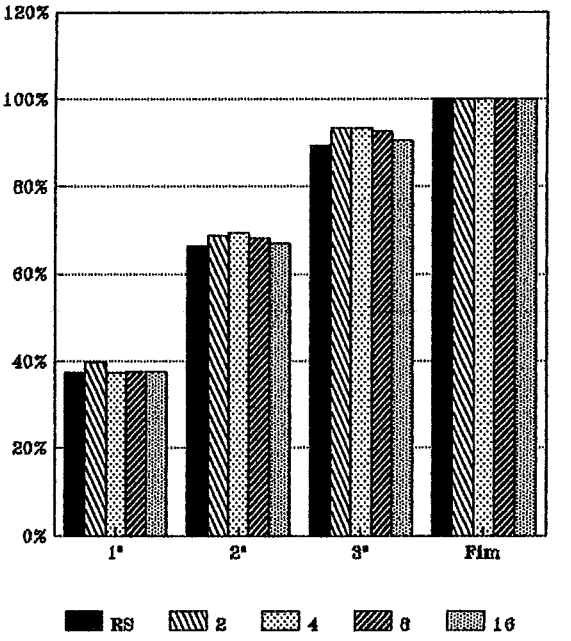


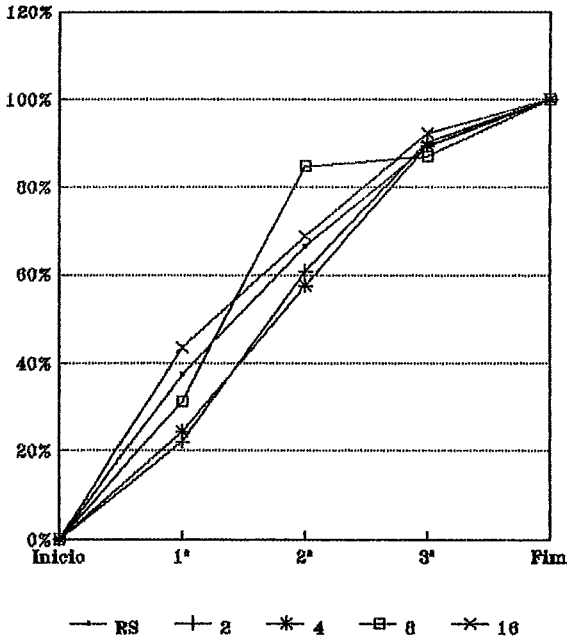
Gráfico E.3: Distribuições Percentuais dos Tempos Total e Efetivo de Processamento
Kabu-Wake
— Interseção —

Backup





Kabu-Wake



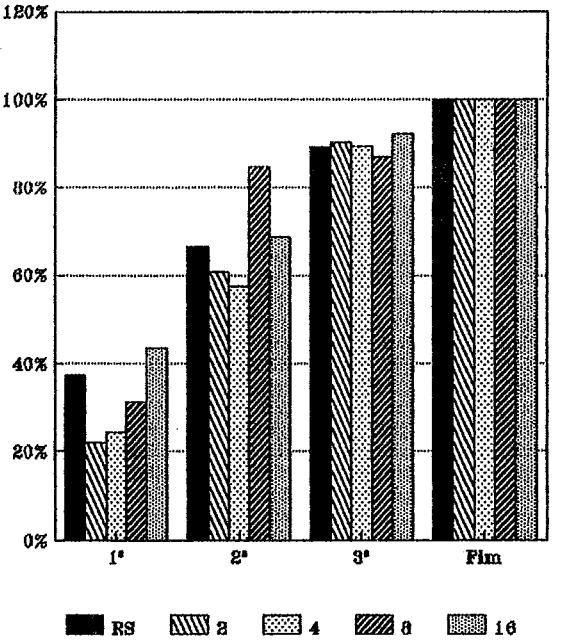
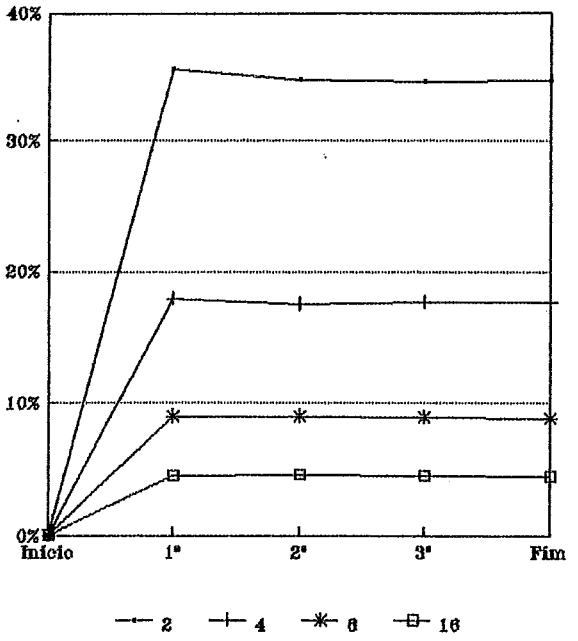
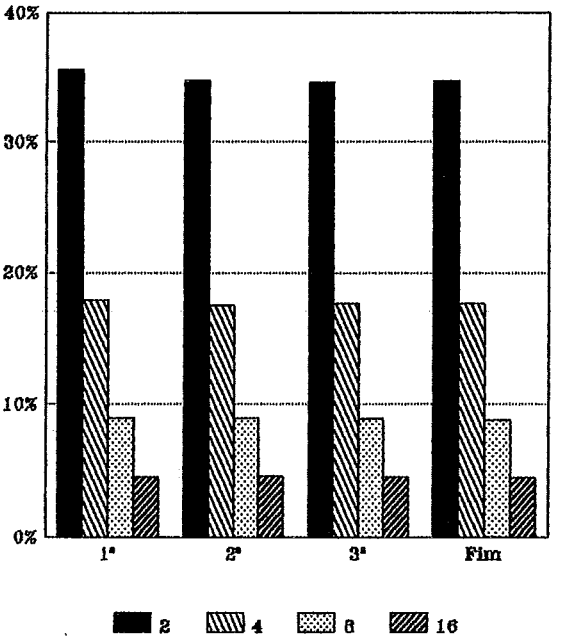


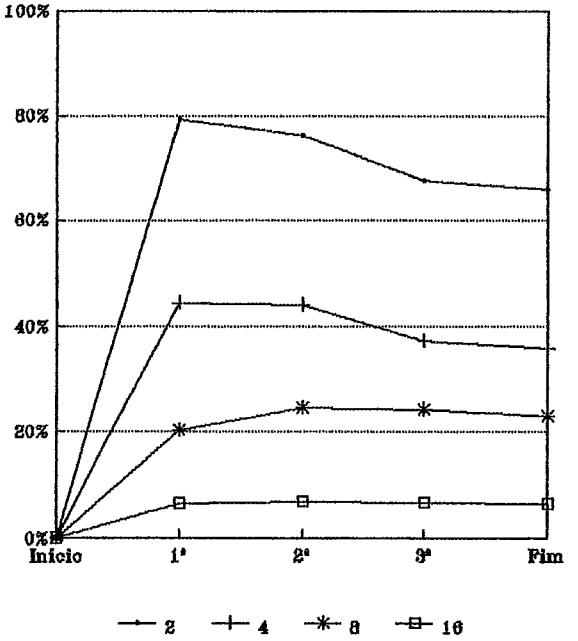
Gráfico E.4: Percentual do Processamento Total Realizado
— Interseção —

Backup





Kabu-Wake



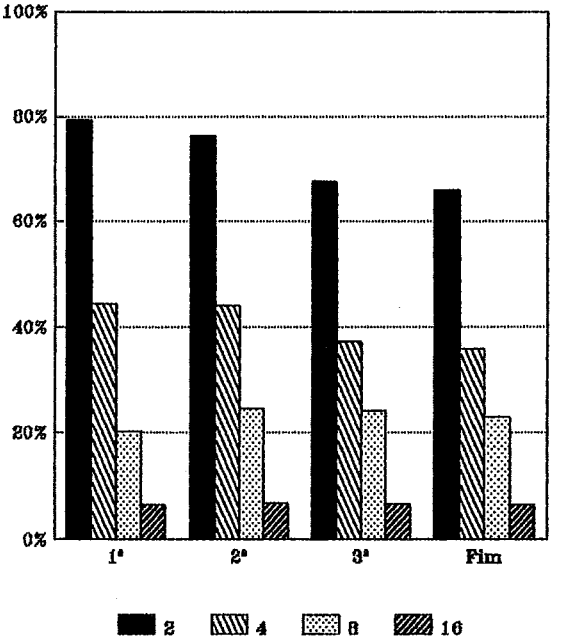
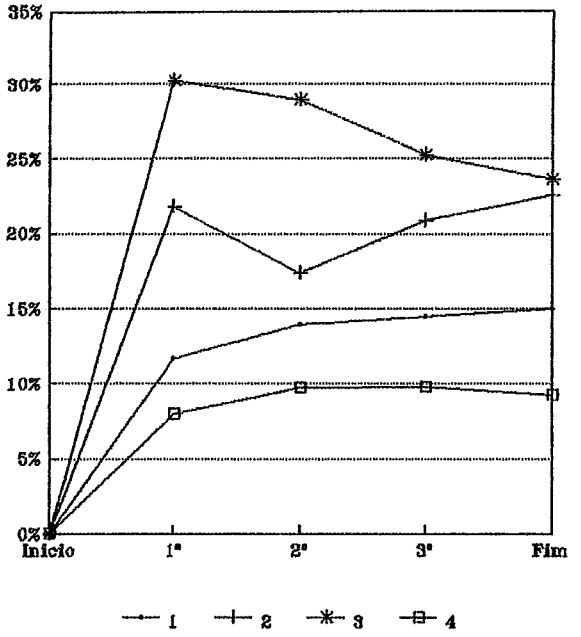
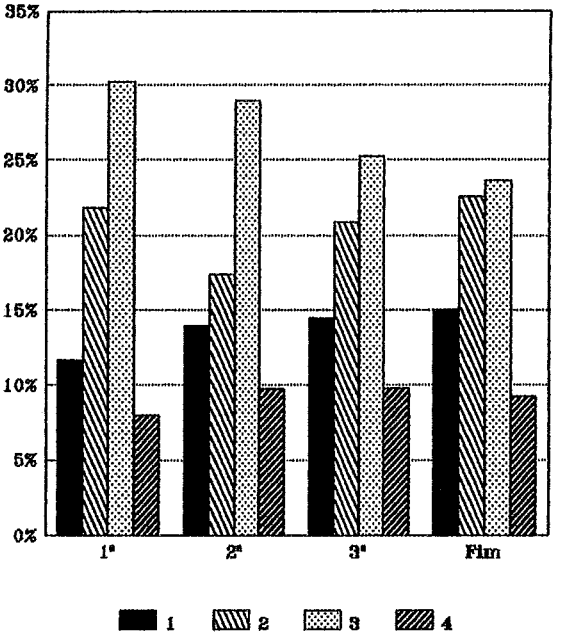


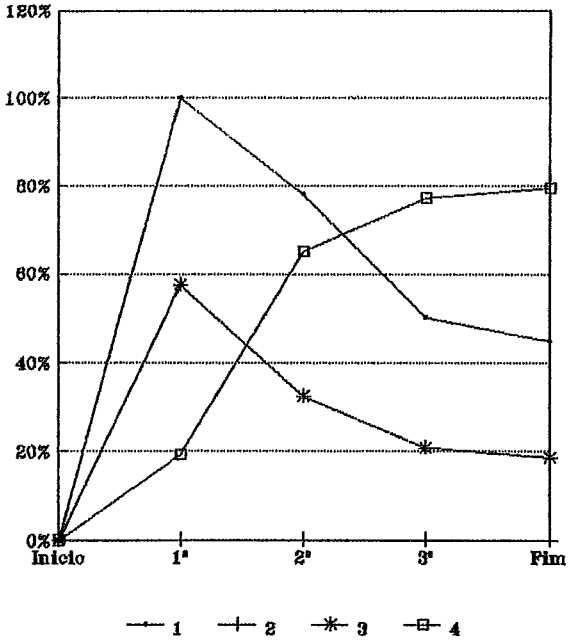
Gráfico E.5: Percentual de Processamento Útil PROLOG
Valores Médios de cada Simulação com Diferentes Número de Processadores
— Interseção —

Backup





Kabu-Wake



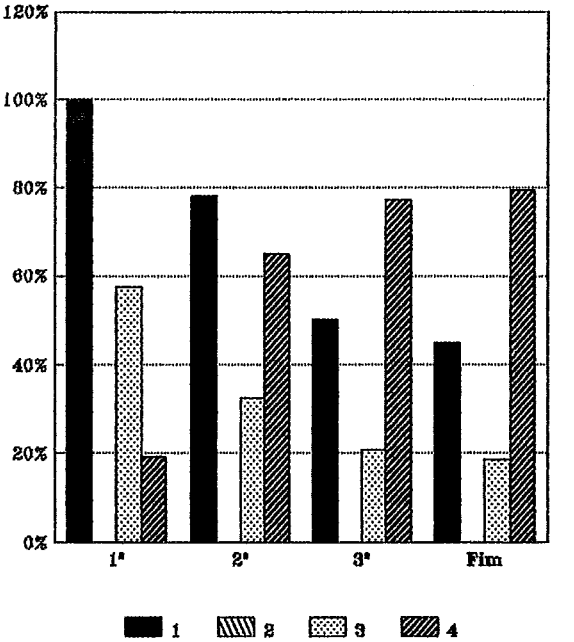
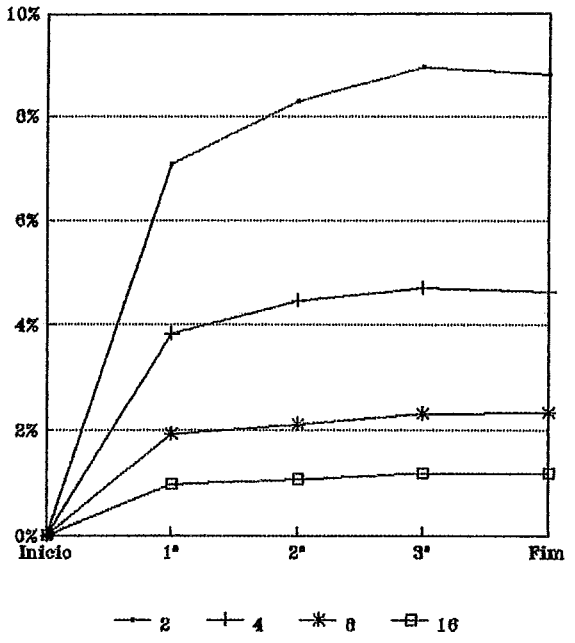
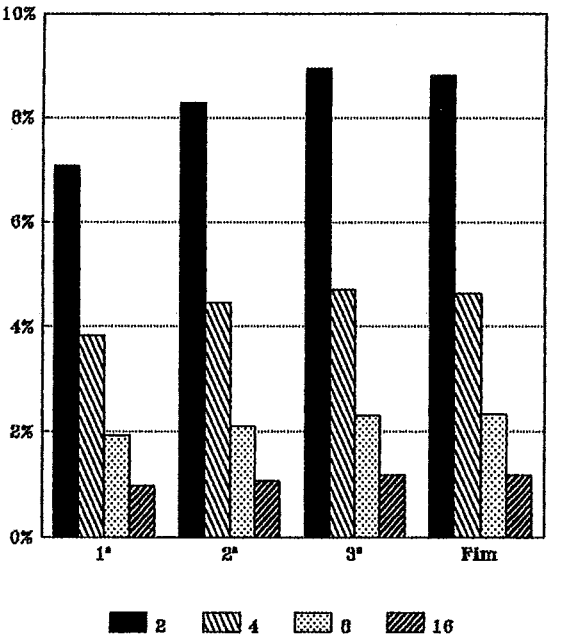


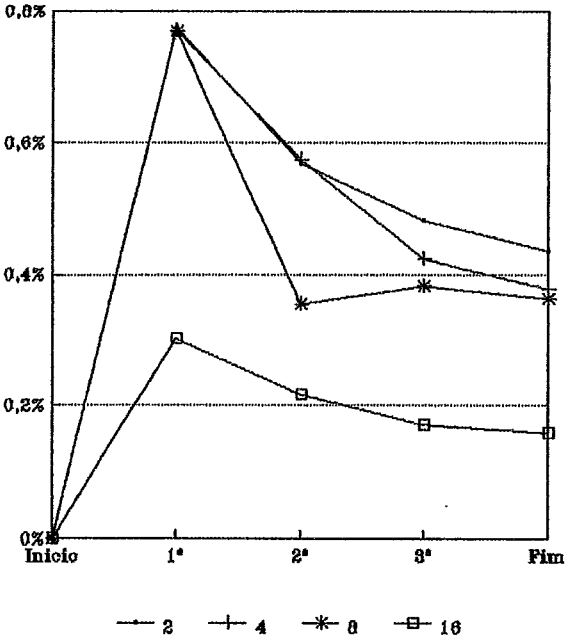
Gráfico E.6: Percentual de Processamento Útil PROLOG
Valores Individuais de cada Processador na Simulação com 4 Processadores
— Interseção —

Backup





Kabu-Wake



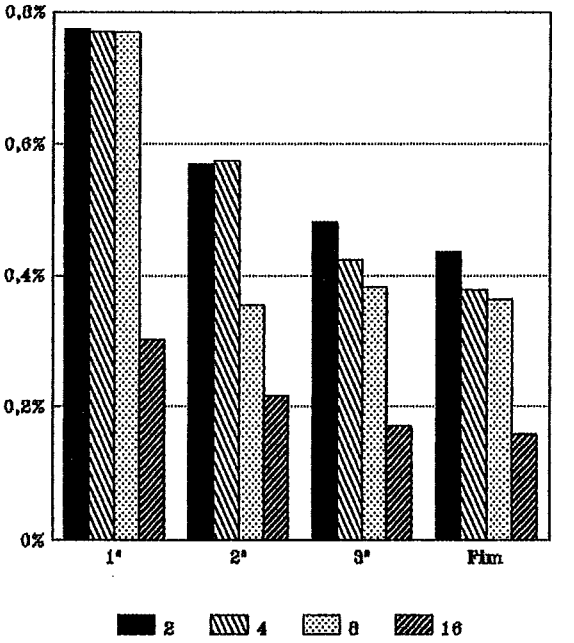
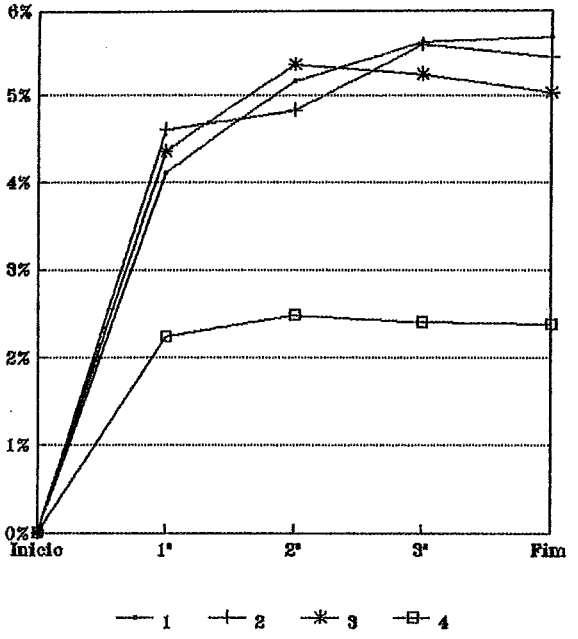
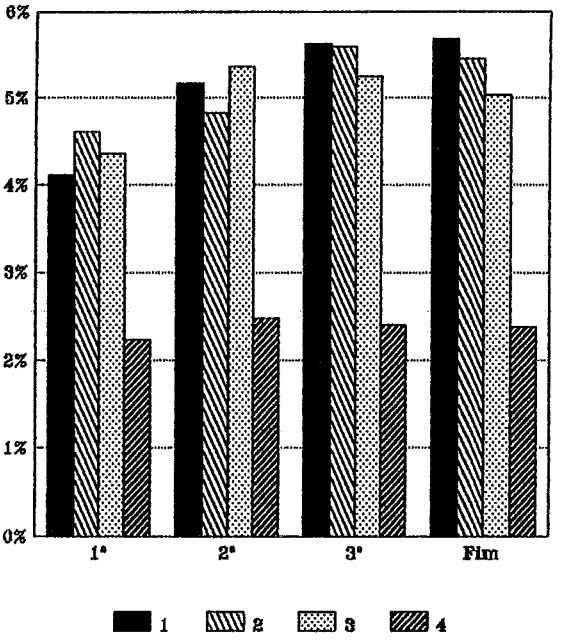


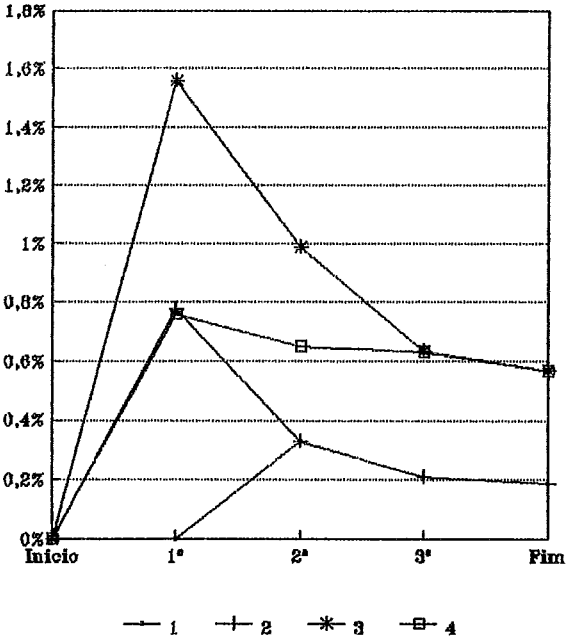
Gráfico E.7: Percentual de Processamento de Comunicações
Valores Médios de cada Simulação com Diferentes Número de Processadores
— Interseção —

Backup





Kabu-Wake



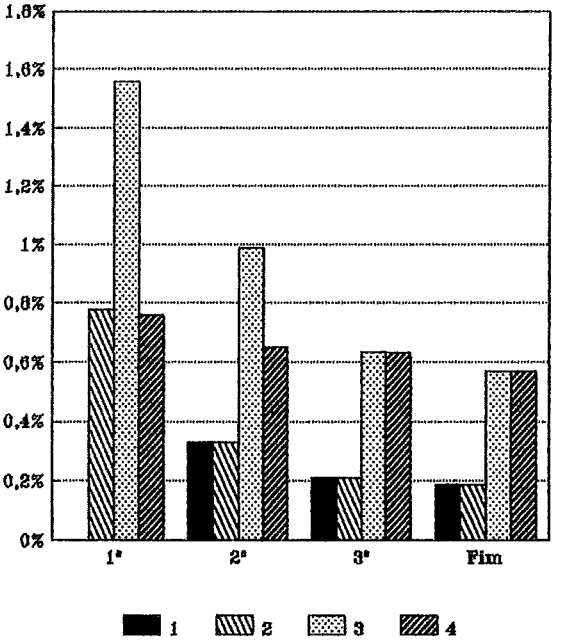
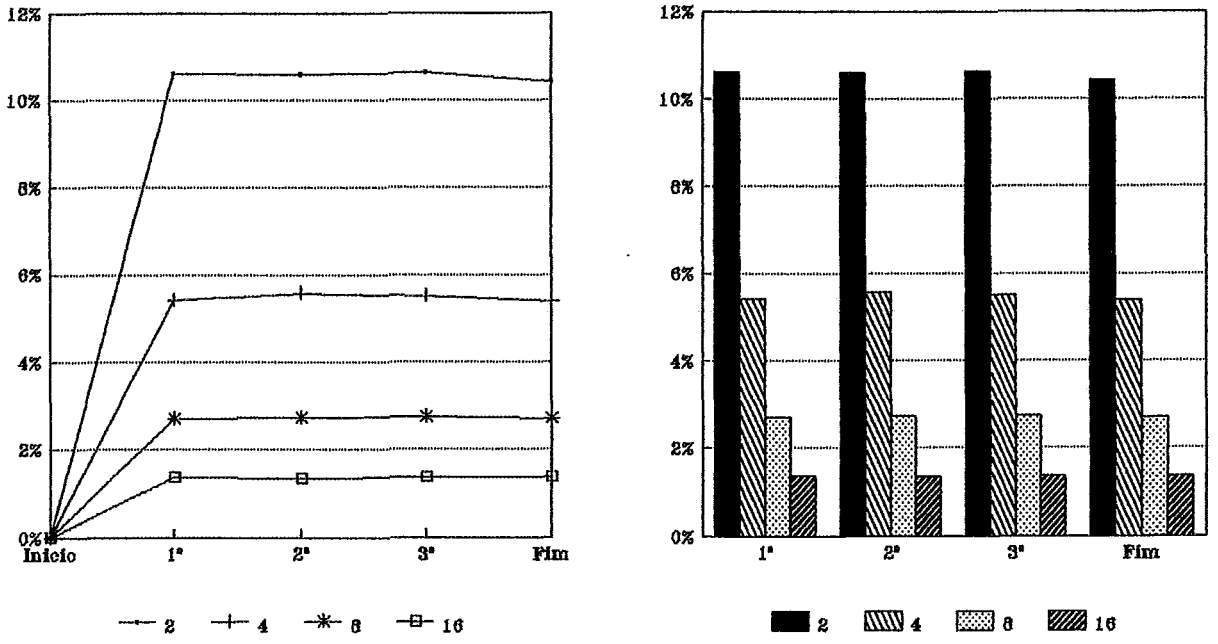


Gráfico E.8: Percentual de Processamento de Comunicações
Valores Individuais de cada Processador na Simulação com 4 Processadores
— Interseção —

Valores Médios de cada Simulação com Diferentes Número de Processadores



Valores Individuais de cada Processador na Simulação com 4 Processadores

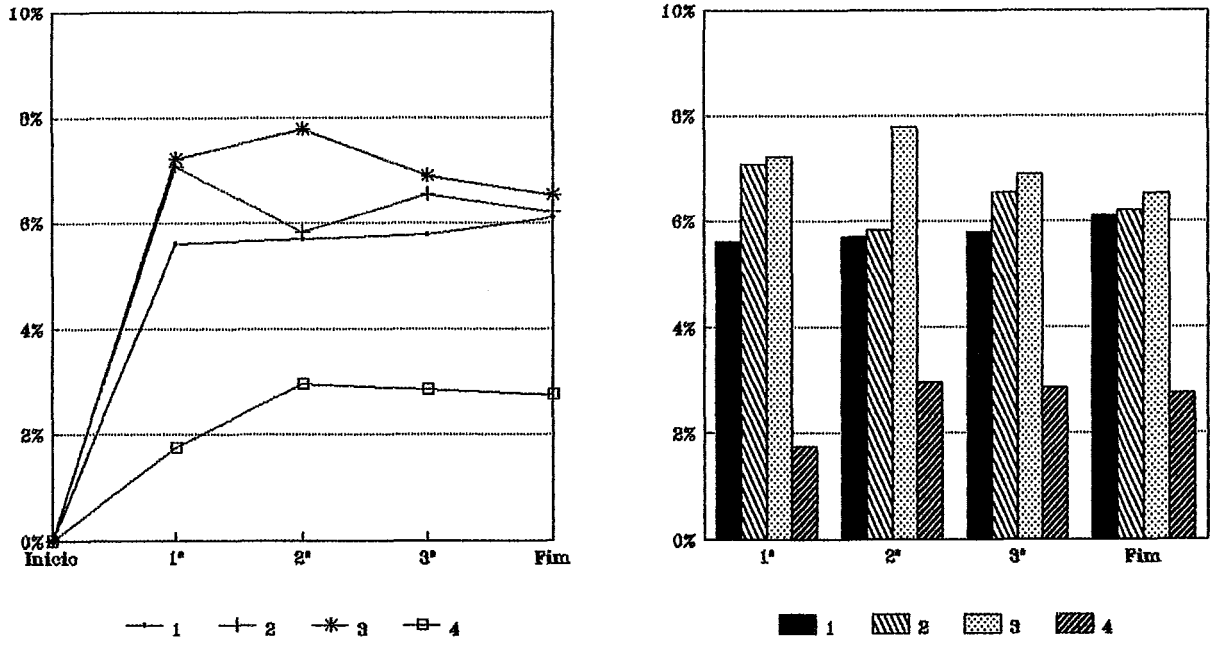
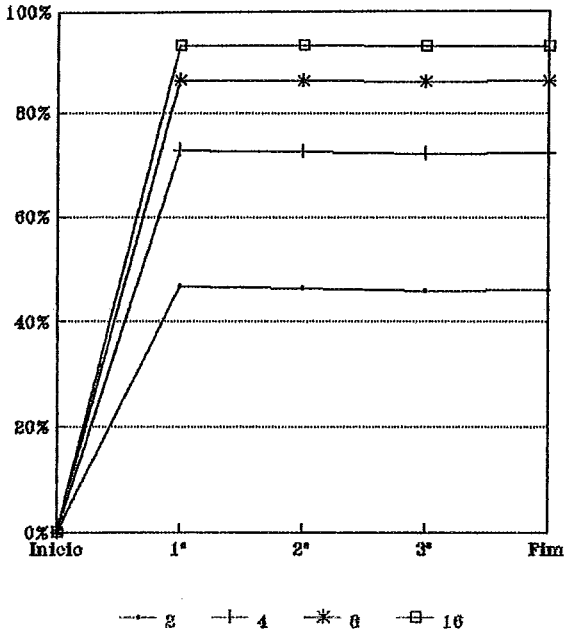
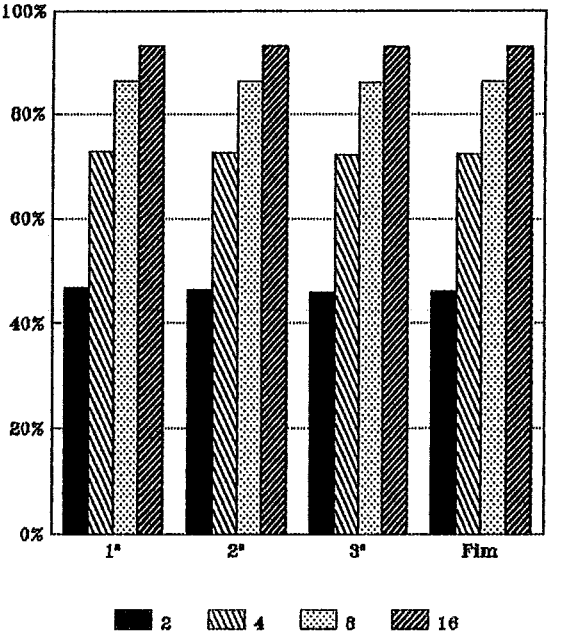


Gráfico E.9: Percentual de Processamento de Manutenção de Processos *Backup*

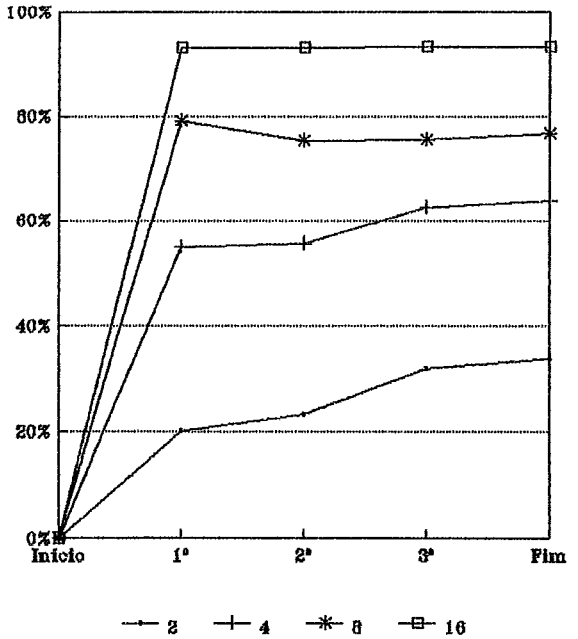
— Interseção —

Backup





Kabu-Wake



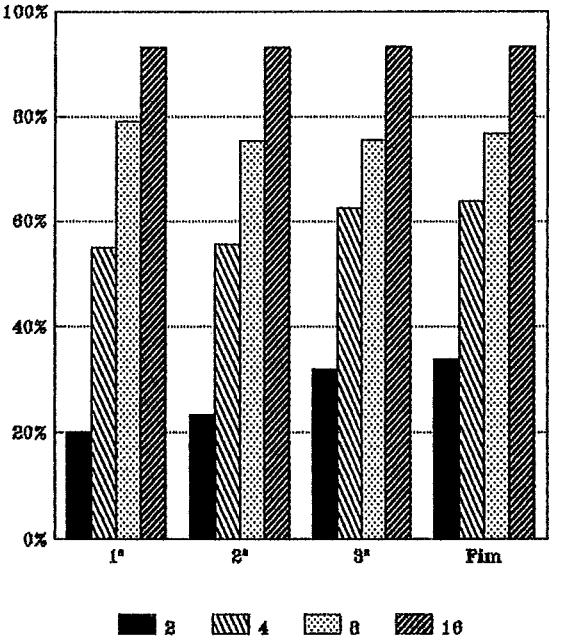
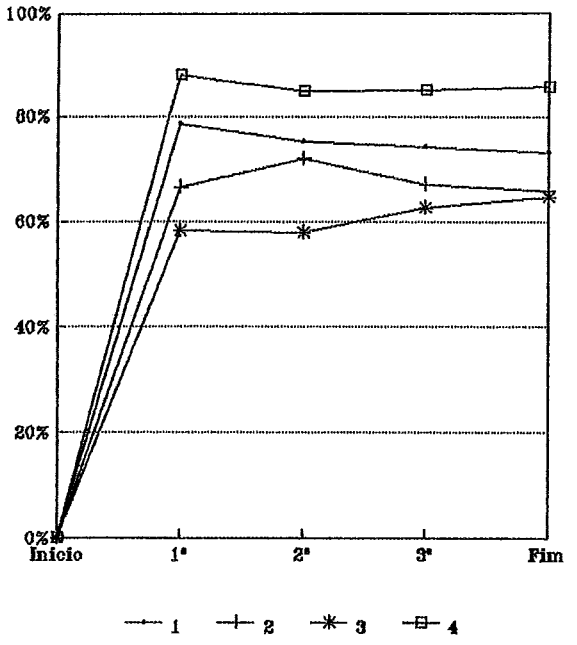
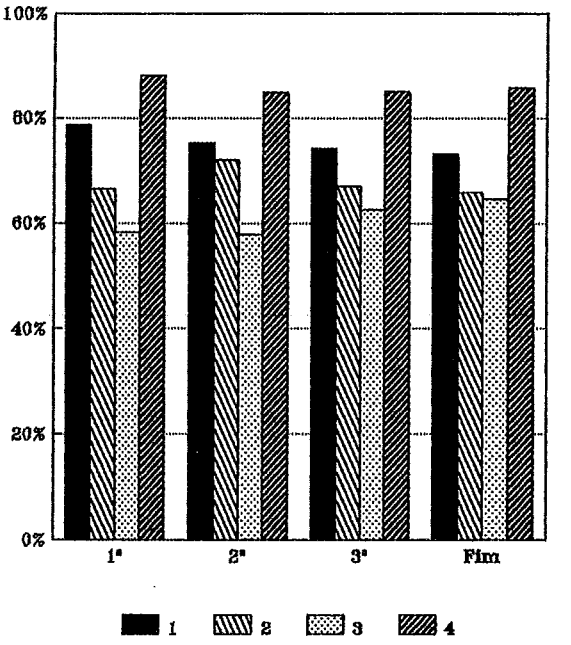


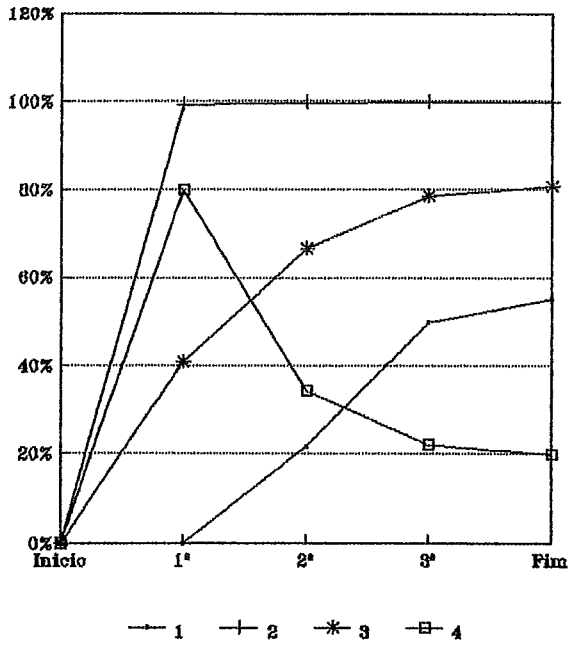
Gráfico E.10: Percentual de Ociosidade dos Processadores
Valores Médios de cada Simulação com Diferentes Número de Processadores
— Interseção —

Backup





Kabu-Wake



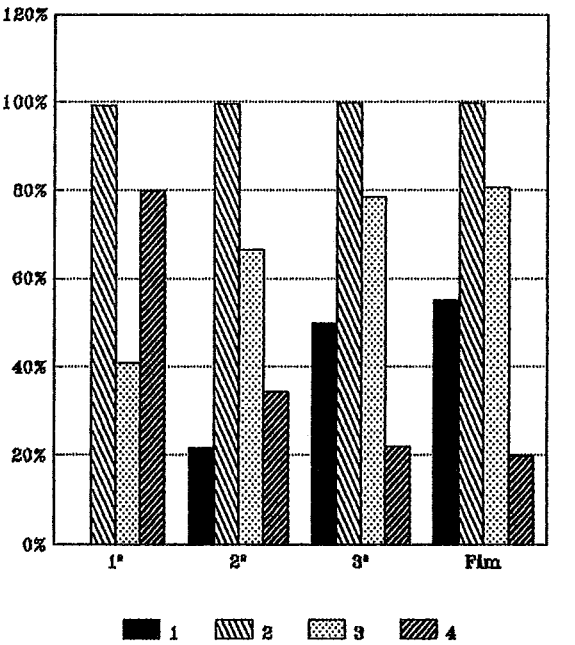
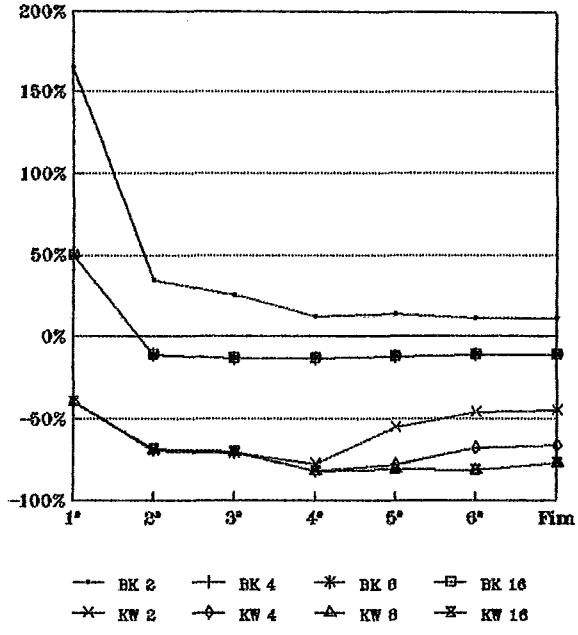


Gráfico E.11: Percentual de Ociosidade dos Processadores
Valores Individuais de cada Processador na Simulação com 4 Processadores
— Interseção —



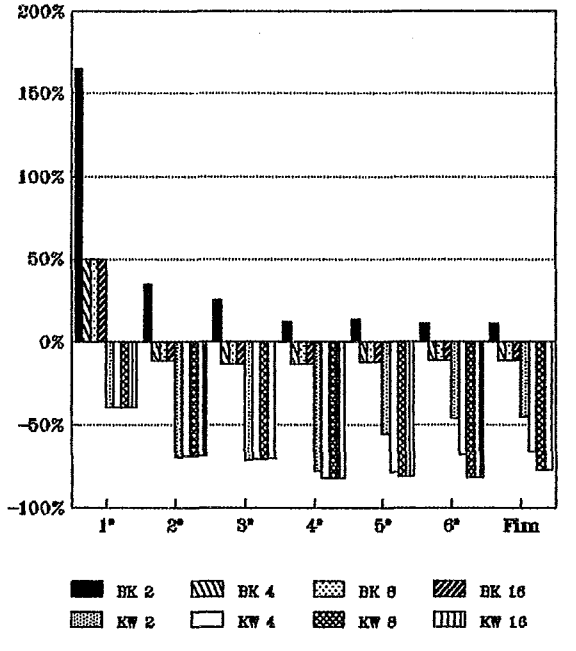
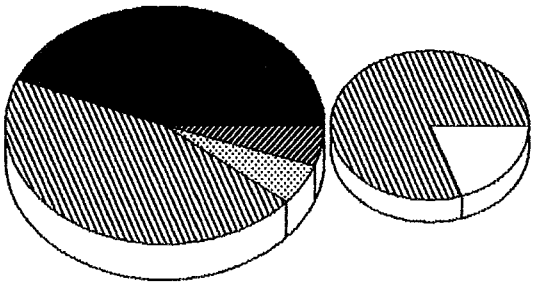
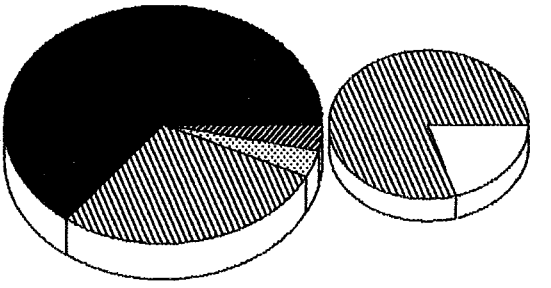


Gráfico E.12: Variação Percentual do Tempo do Seqüencial
— Paper —

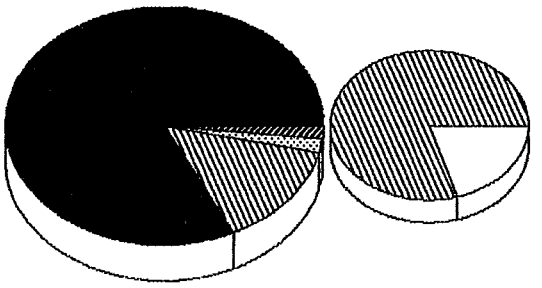
2 Processadores



4 Processadores



8 Processadores



16 Processadores

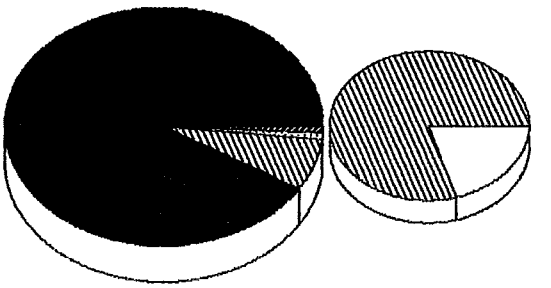
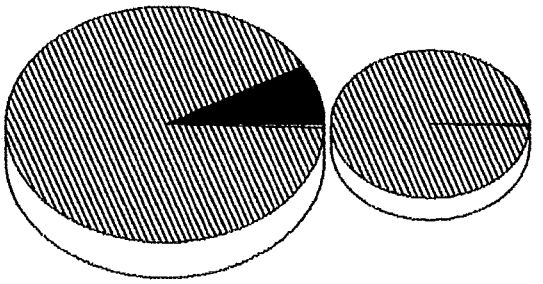


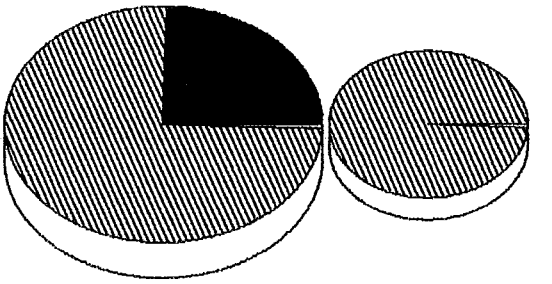
Gráfico E.13: Distribuições Percentuais dos Tempos Total e Efetivo de Processamento

Backup
— Paper —

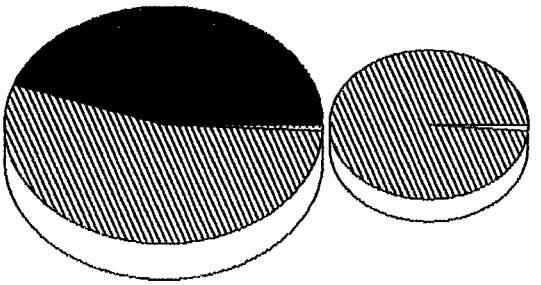
2 Processadores



4 Processadores



8 Processadores



16 Processadores

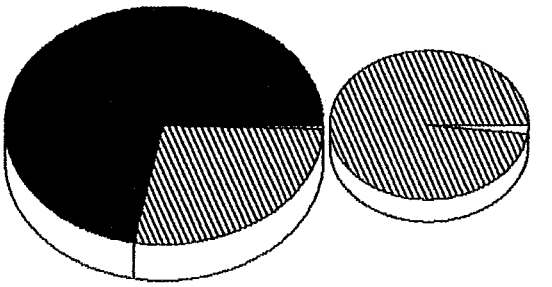
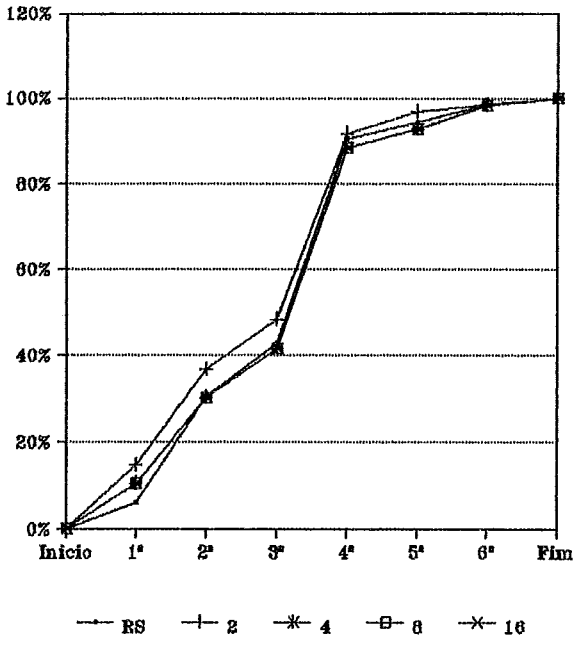
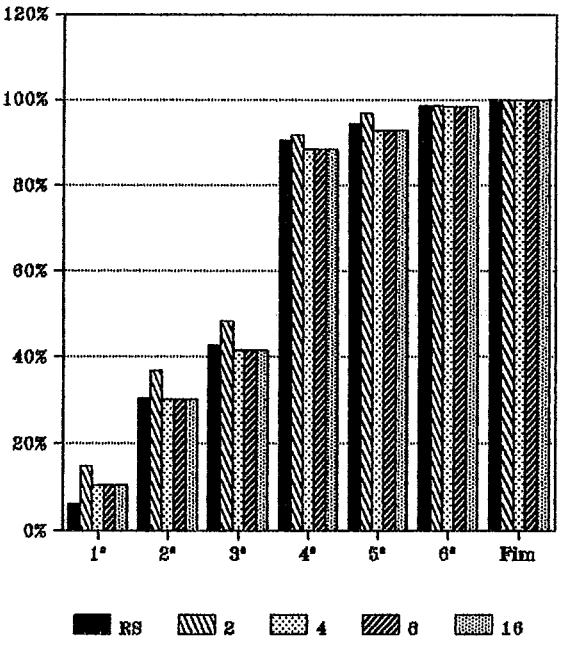


Gráfico E.14: Distribuições Percentuais dos Tempos Total e Efetivo de Processamento

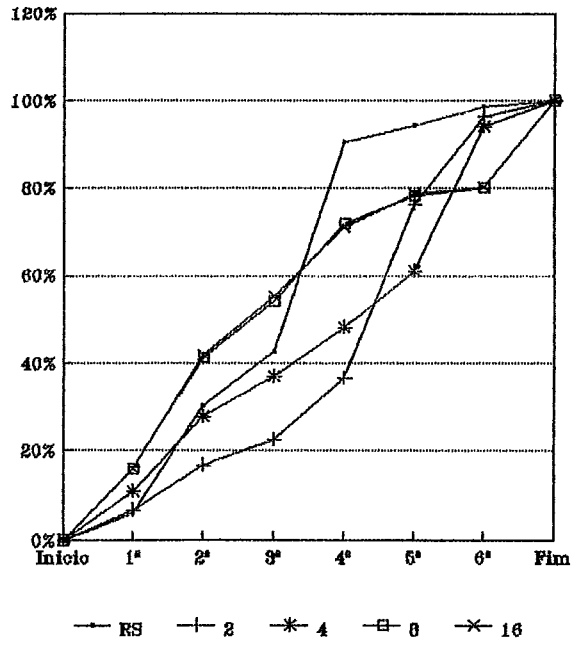
Kabu-Wake
— Paper —

Backup





Kabu-Wake



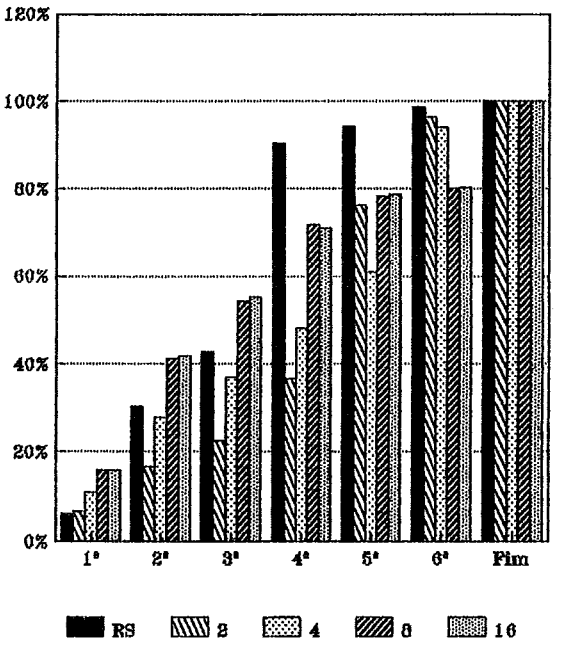
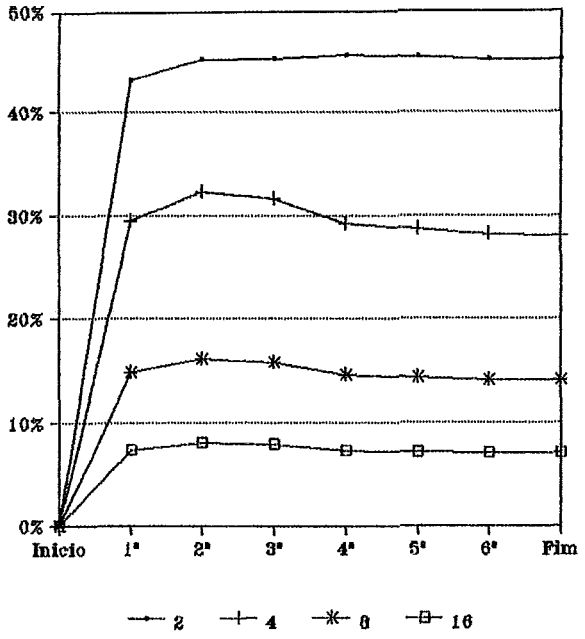
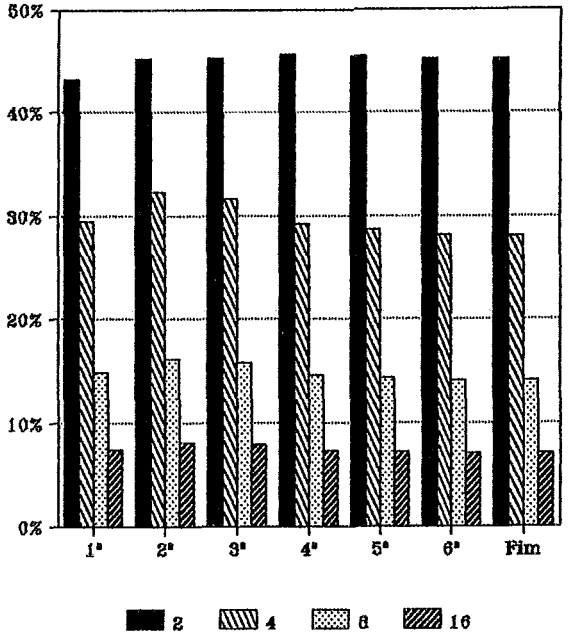


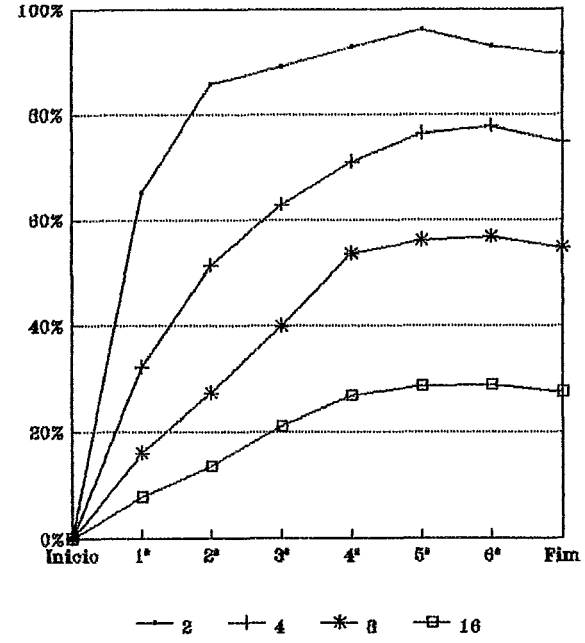
Gráfico E.15: Percentual do Processamento Total Realizado
— Paper —

Backup





Kabu-Wake



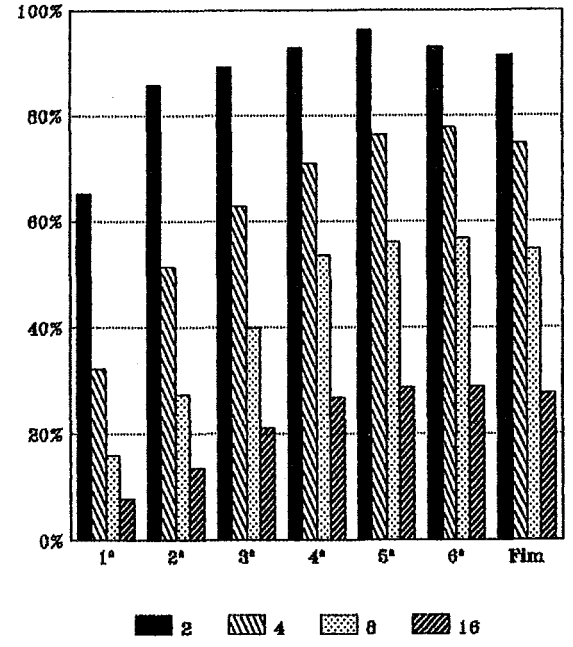
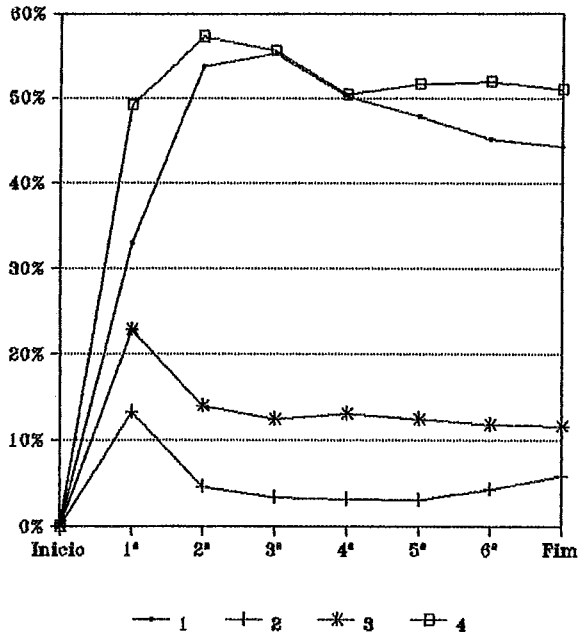
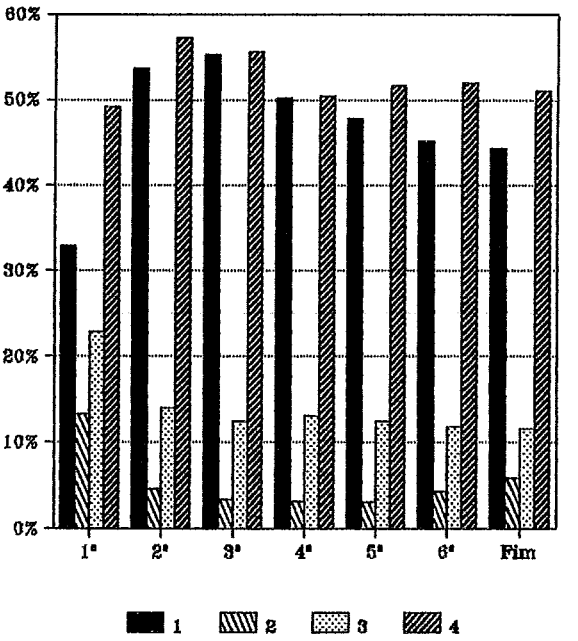


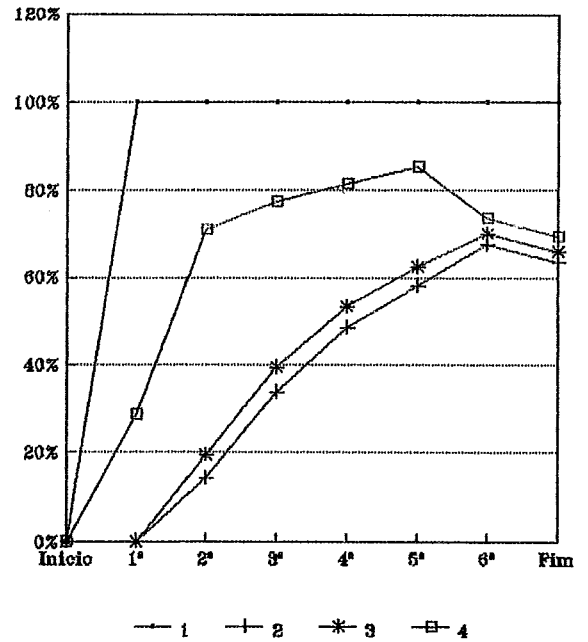
Gráfico E.16: Percentual de Processamento Útil PROLOG
Valores Médios de cada Simulação com Diferentes Número de Processadores
— Paper —

Backup





Kabu-Wake



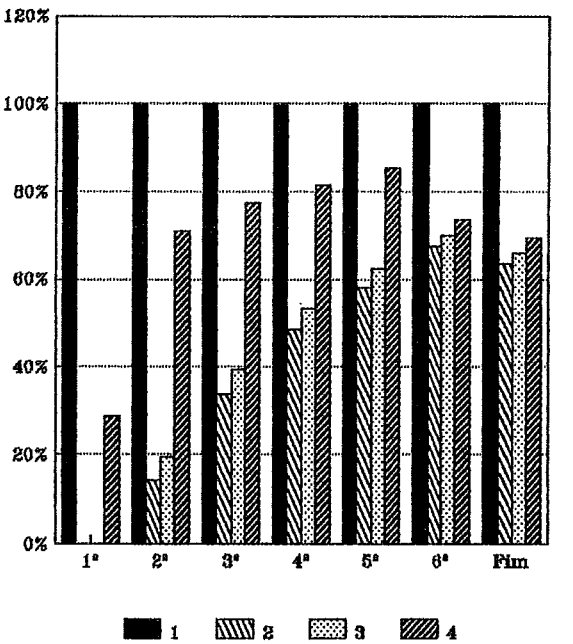
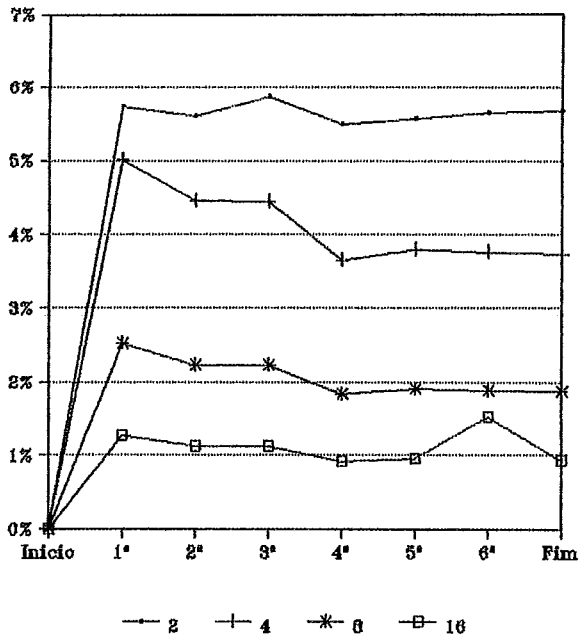
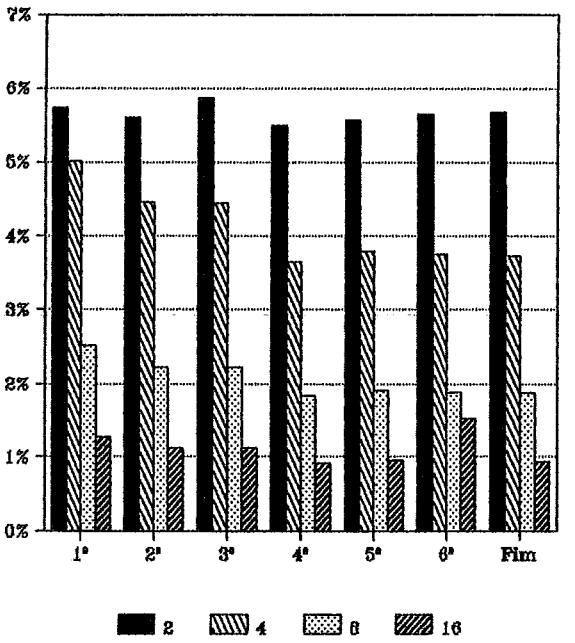


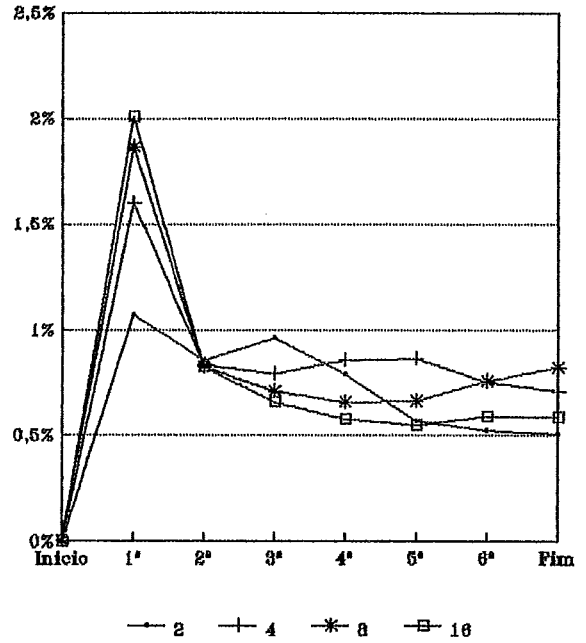
Gráfico E.17: Percentual de Processamento Útil PROLOG
Valores Individuais de cada Processador na Simulação com 4 Processadores
— Paper —

Backup





Kabu-Wake



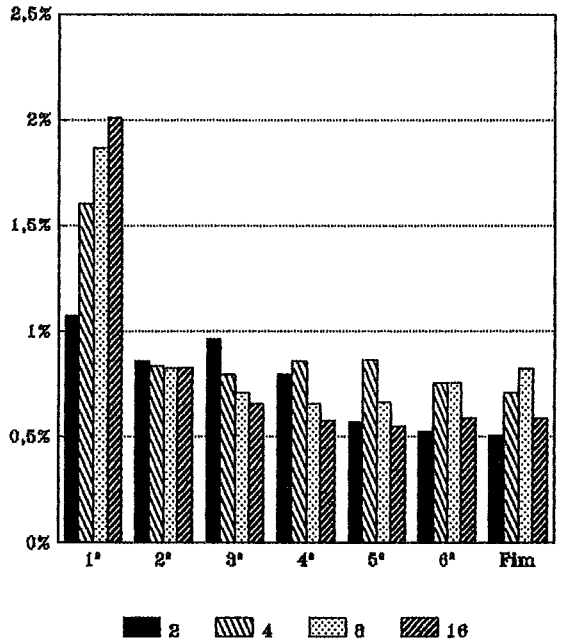
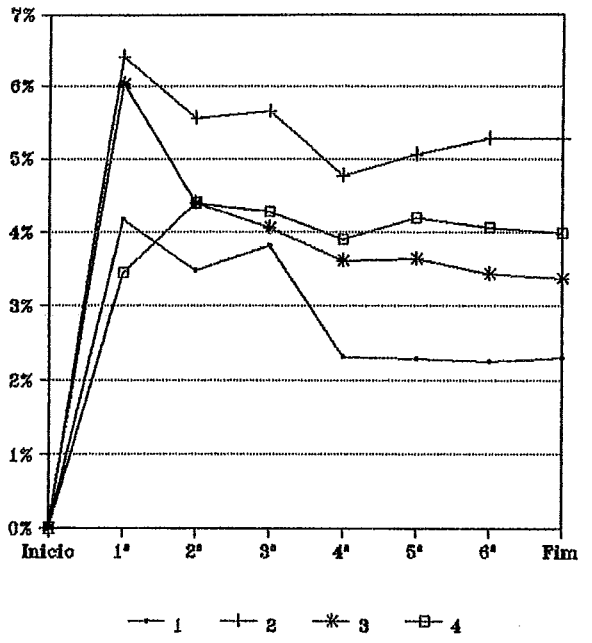
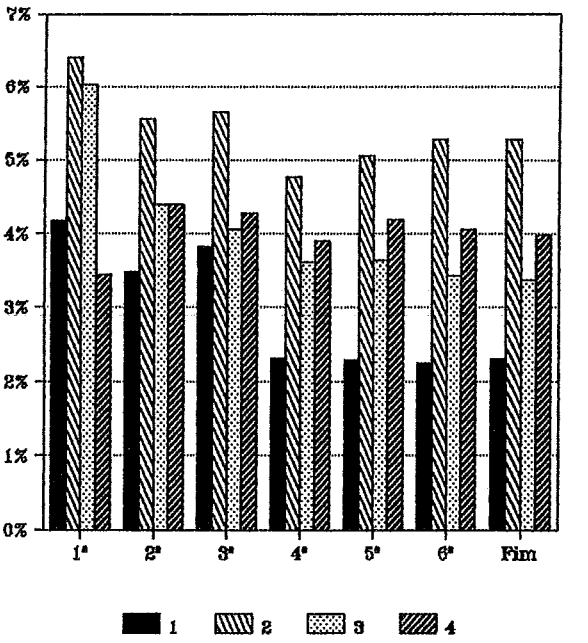


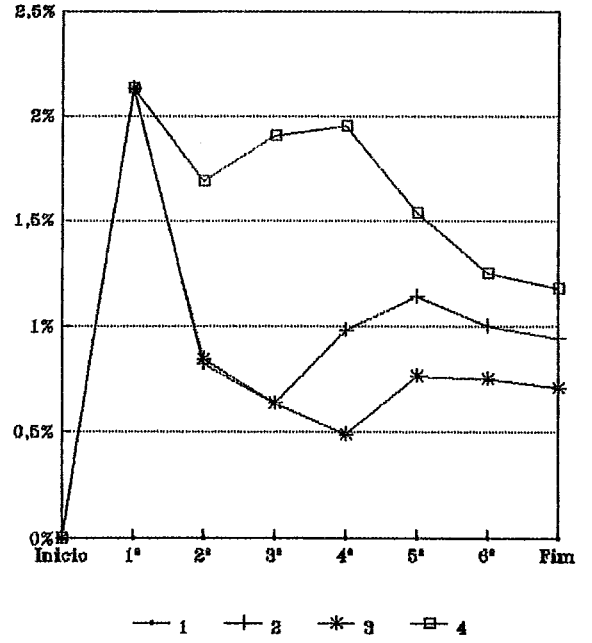
Gráfico E.18: Percentual de Processamento de Comunicações
Valores Médios de cada Simulação com Diferentes Número de Processadores
— Paper —

Backup





Kabu-Wake



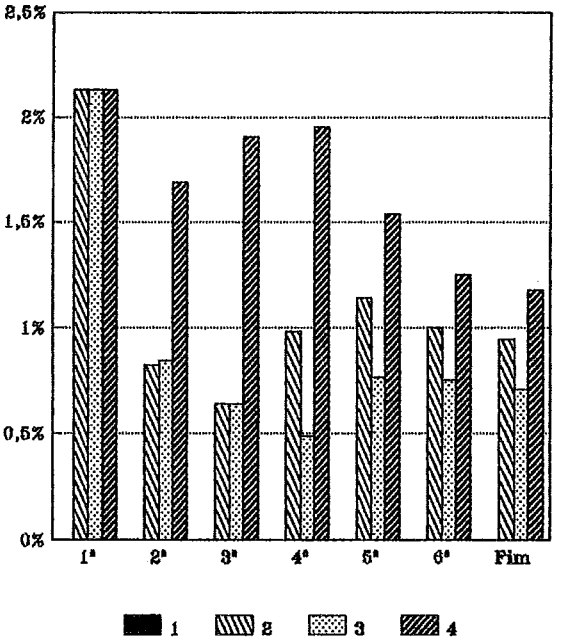
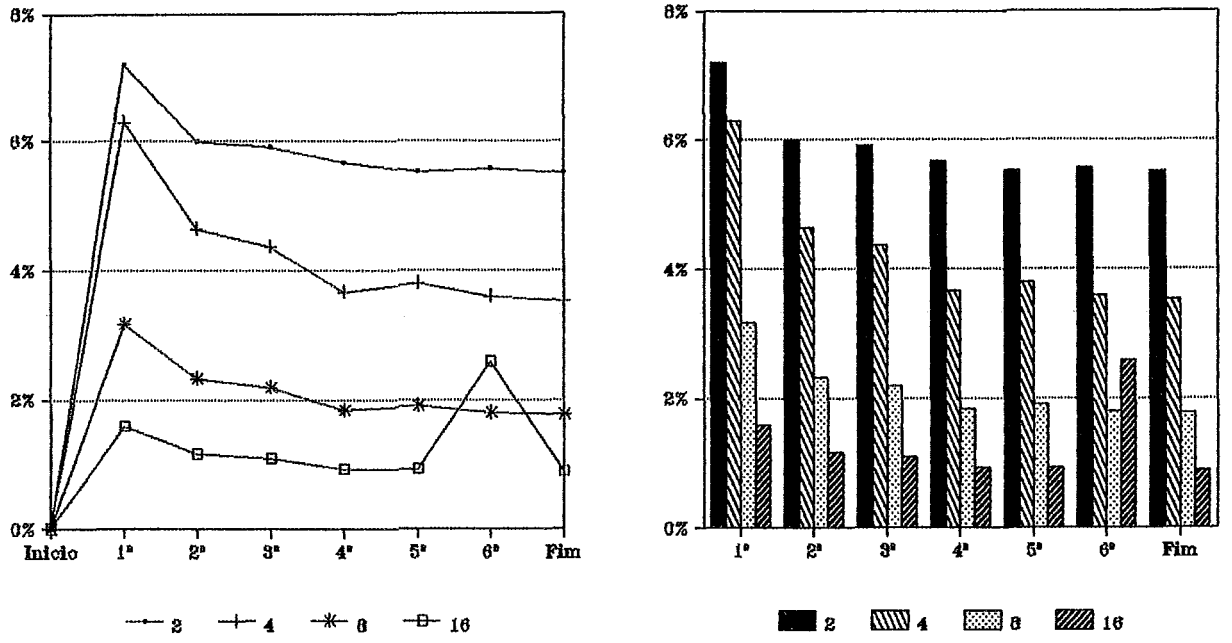


Gráfico E.19: Percentual de Processamento de Comunicações
Valores Individuais de cada Processador na Simulação com 4 Processadores
— Paper —

Valores Médios de cada Simulação com Diferentes Número de Processadores



Valores Individuais de cada Processador na Simulação com 4 Processadores

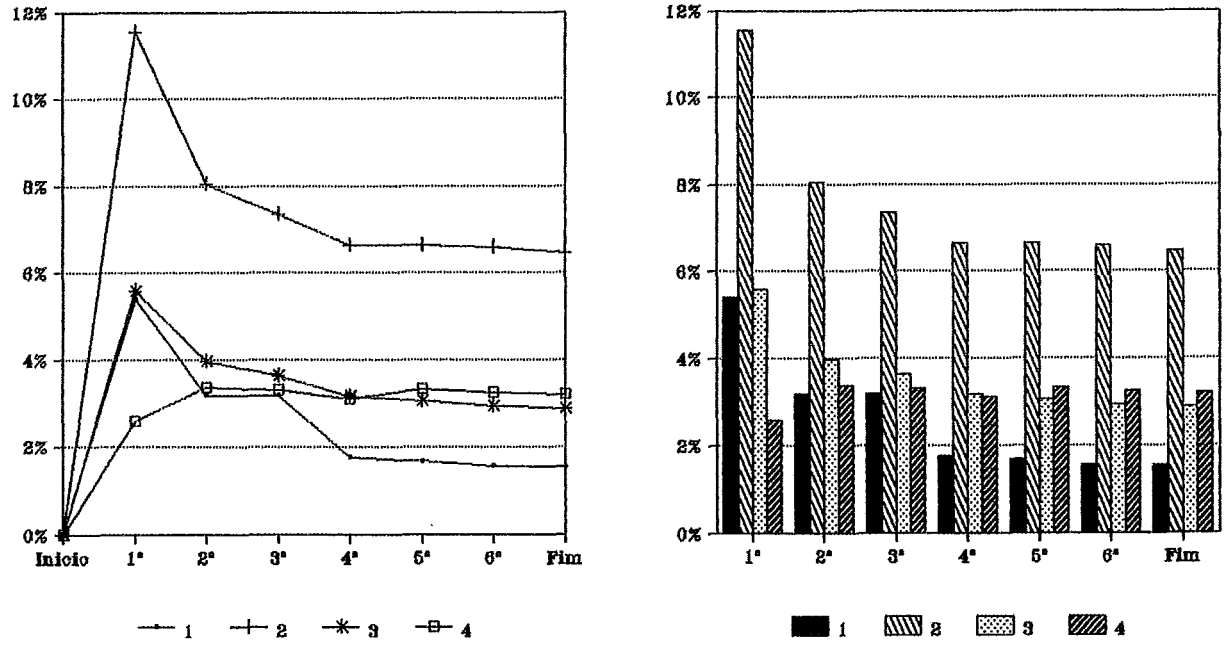
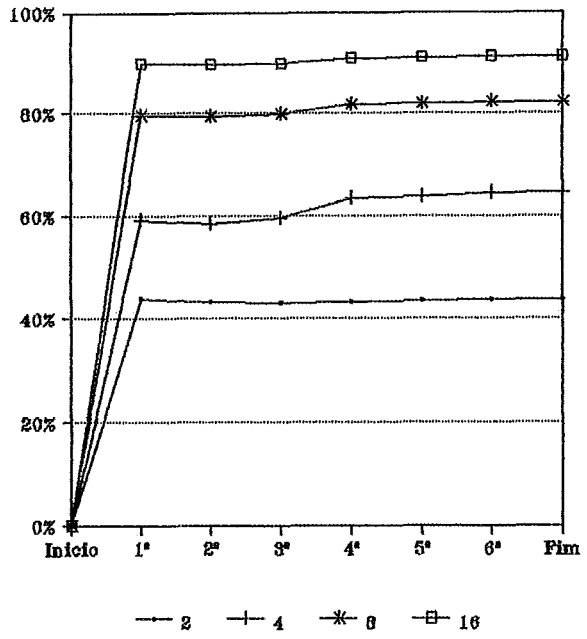
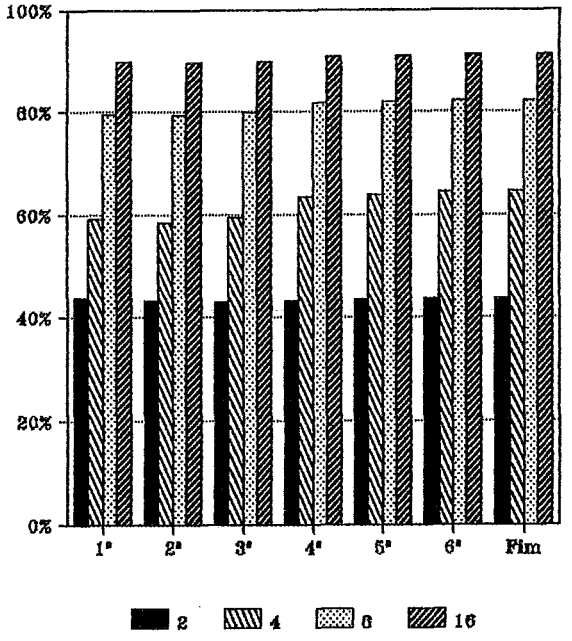


Gráfico E.20: Percentual de Processamento de Manutenção de Processos *Backup*

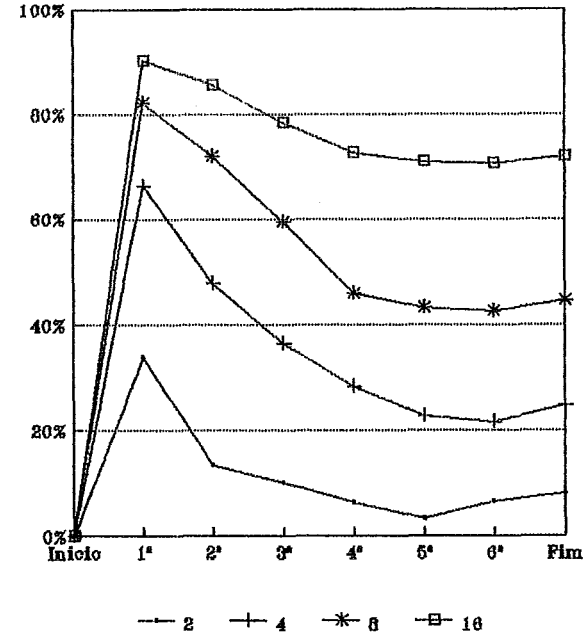
— Paper —

Backup





Kabu-Wake



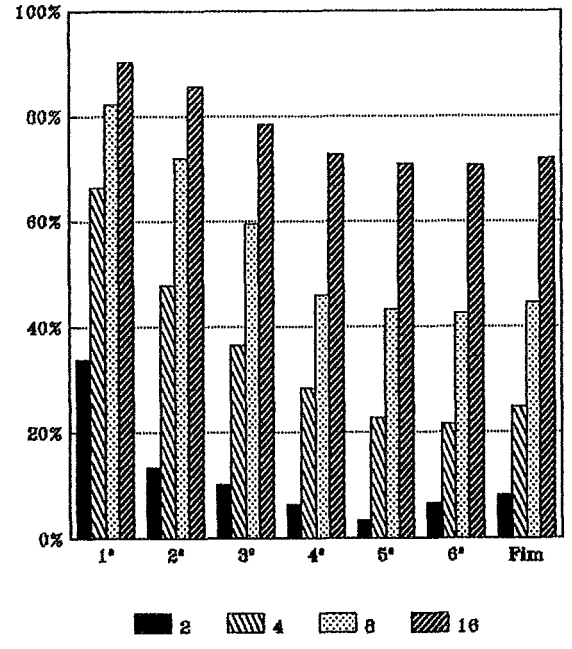
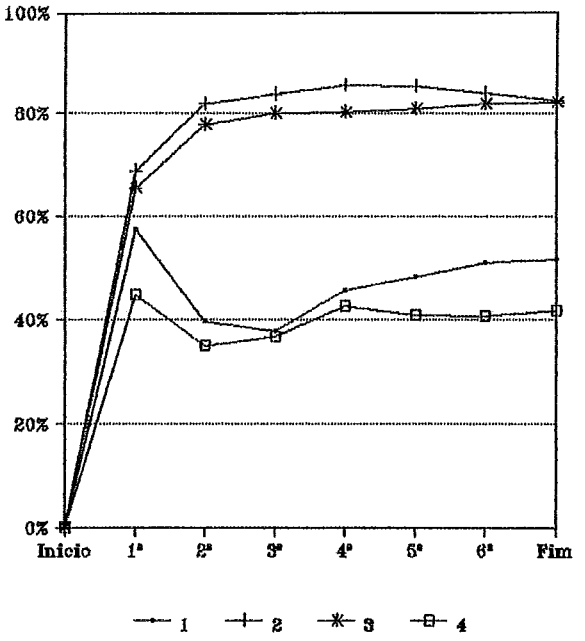
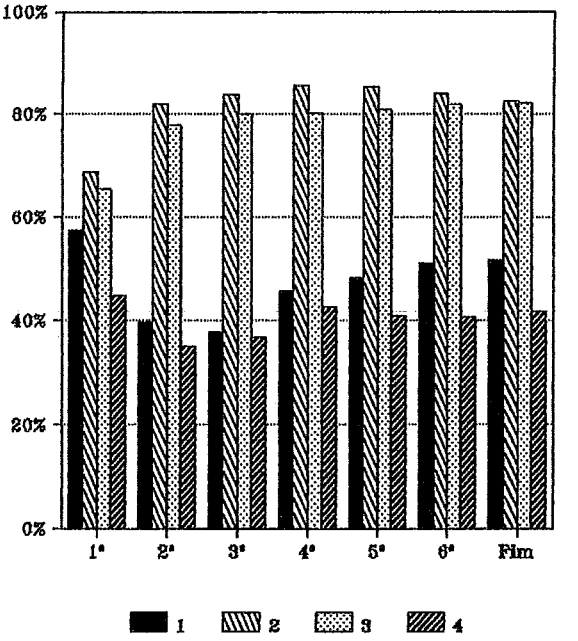


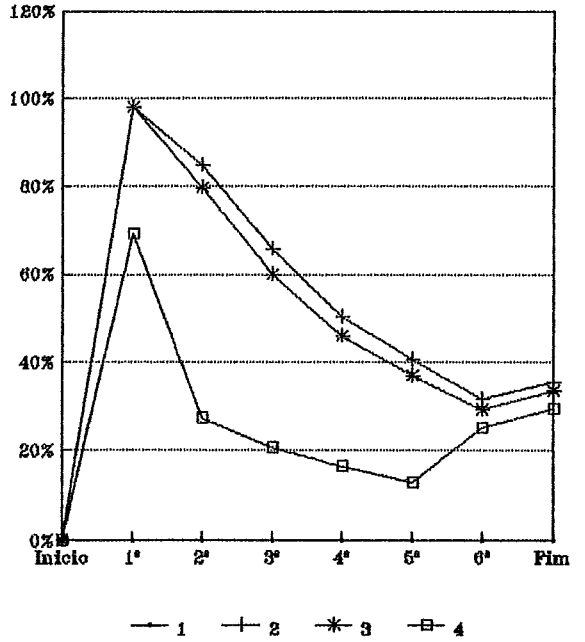
Gráfico E.21: Percentual de Ociosidade dos Processadores
Valores Médios de cada Simulação com Diferentes Número de Processadores
— Paper —

Backup





Kabu-Wake



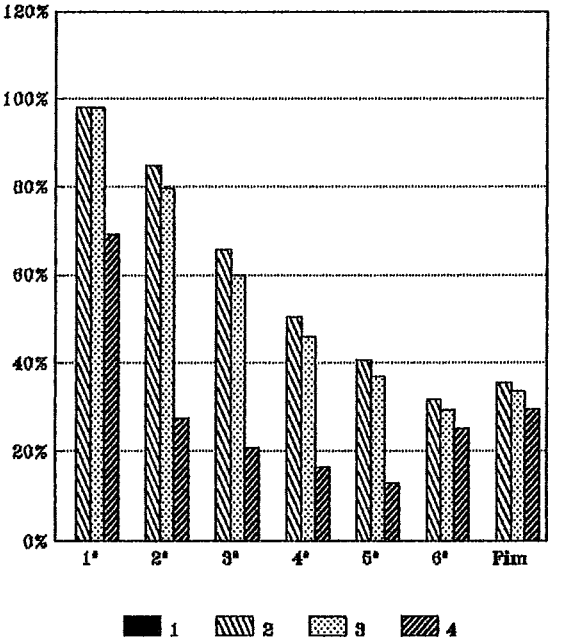
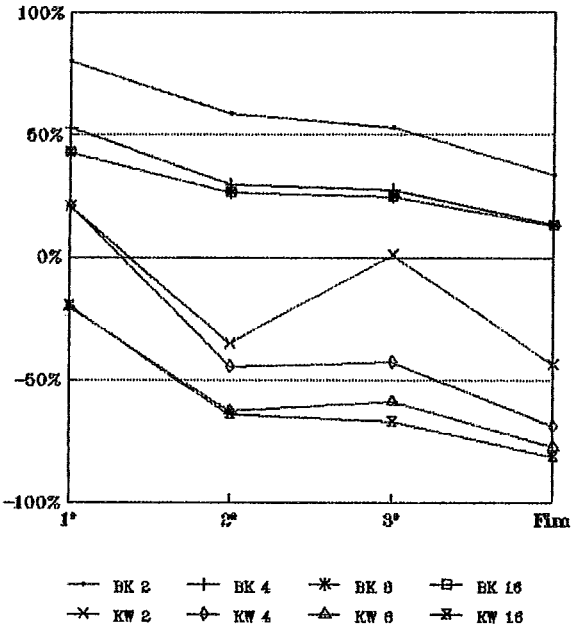


Gráfico E.22: Percentual de Ociosidade dos Processadores
Valores Individuais de cada Processador na Simulação com 4 Processadores
— Paper —



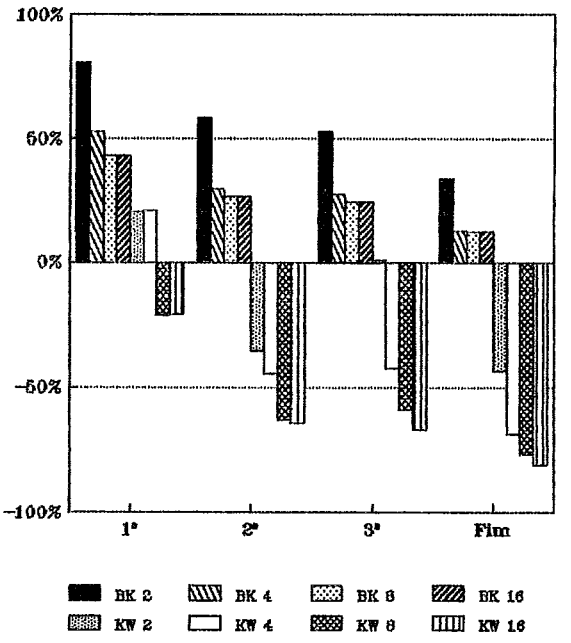
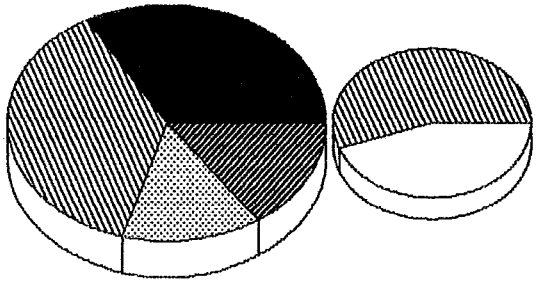
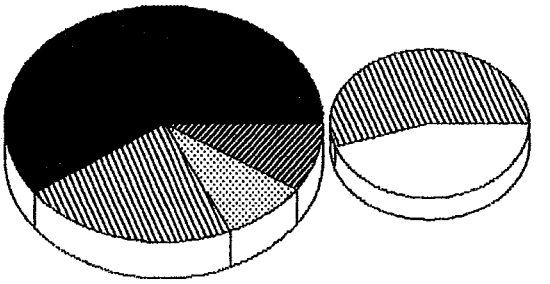


Gráfico E.23: Variação Percentual do Tempo do Seqüencial
— Densidade —

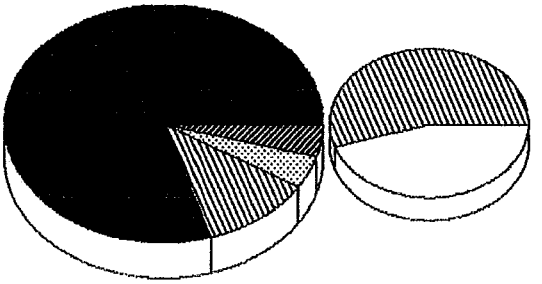
2 Processadores



4 Processadores



8 Processadores



16 Processadores

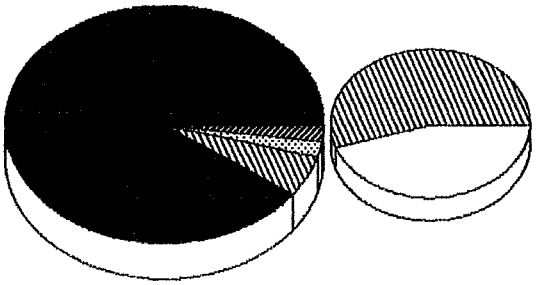
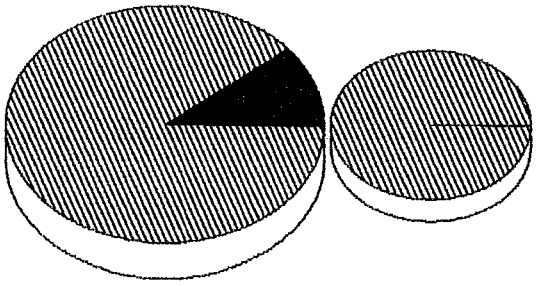


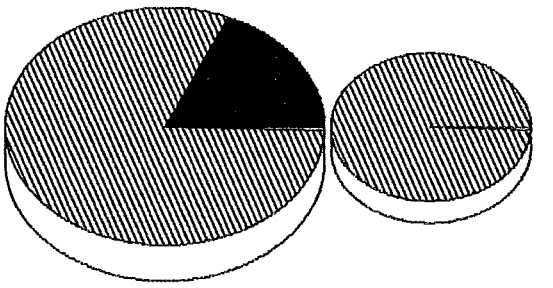
Gráfico E.24: Distribuições Percentuais dos Tempos Total e Efetivo de Processamento

Backup
— Densidade —

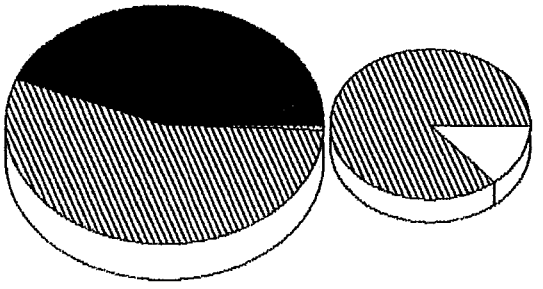
2 Processadores



4 Processadores



8 Processadores



16 Processadores

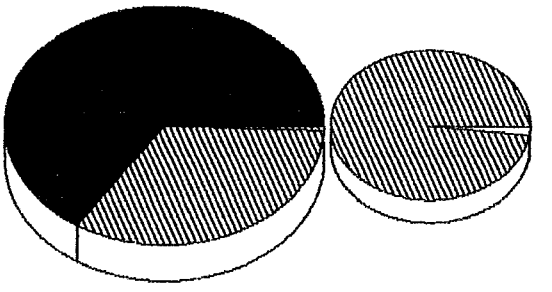
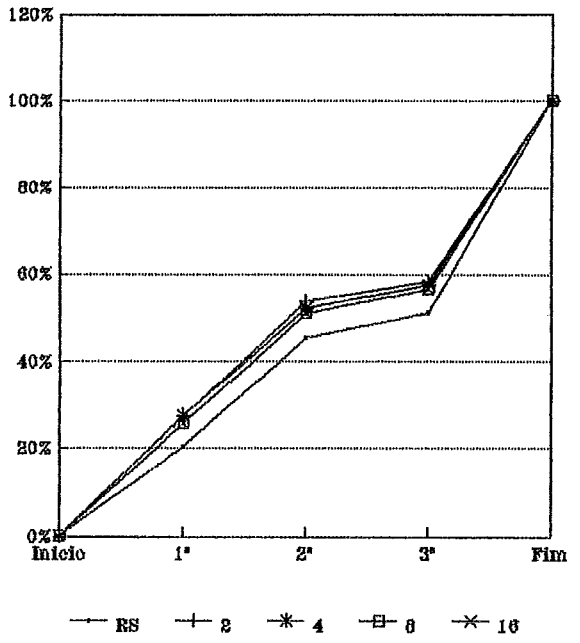
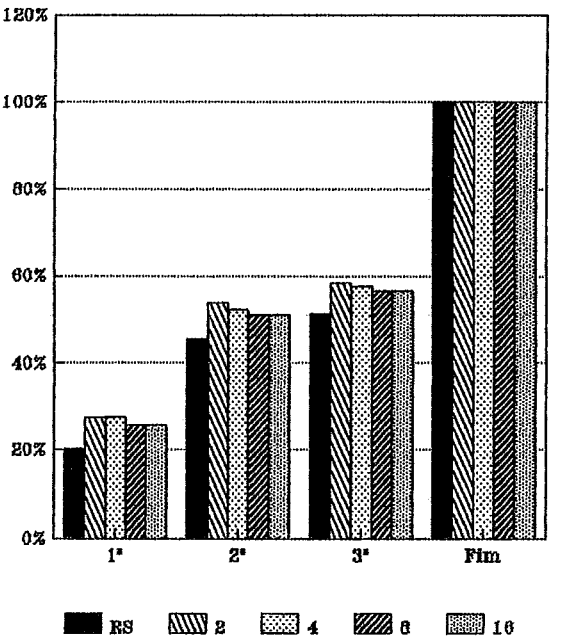


Gráfico E.25: Distribuições Percentuais dos Tempos Total e Efetivo de Processamento

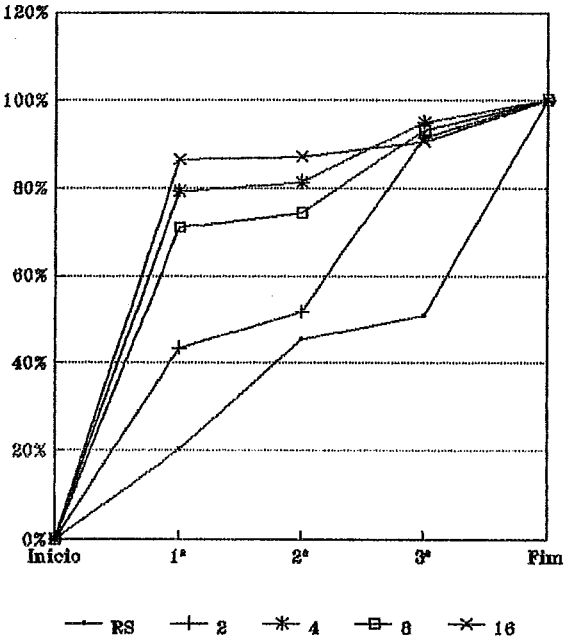
Kabu-Wake
— Densidade —

Backup





Kabu-Wake



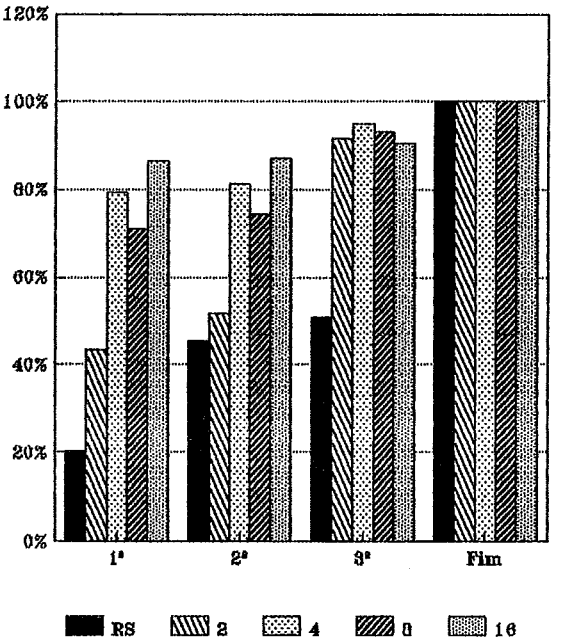
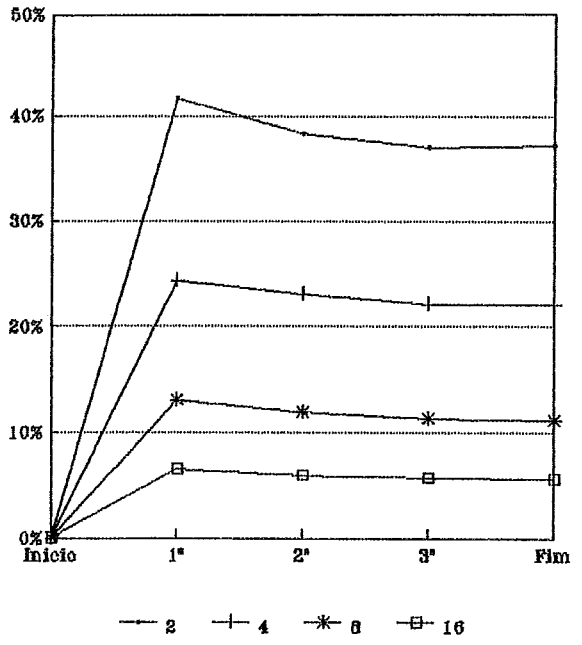
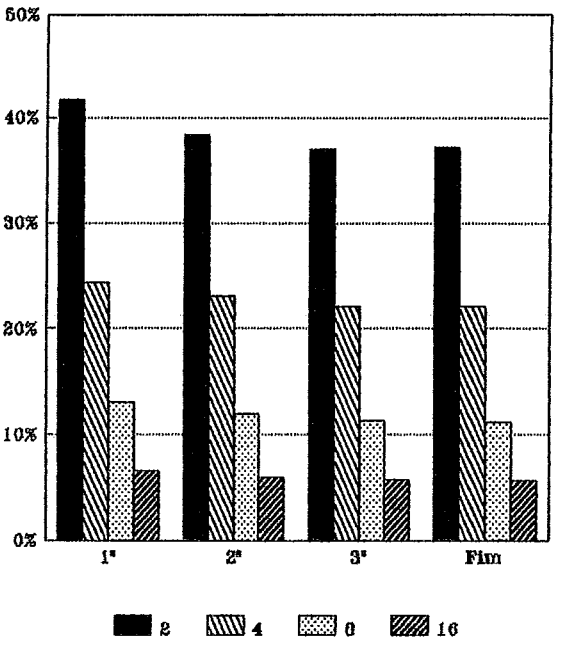


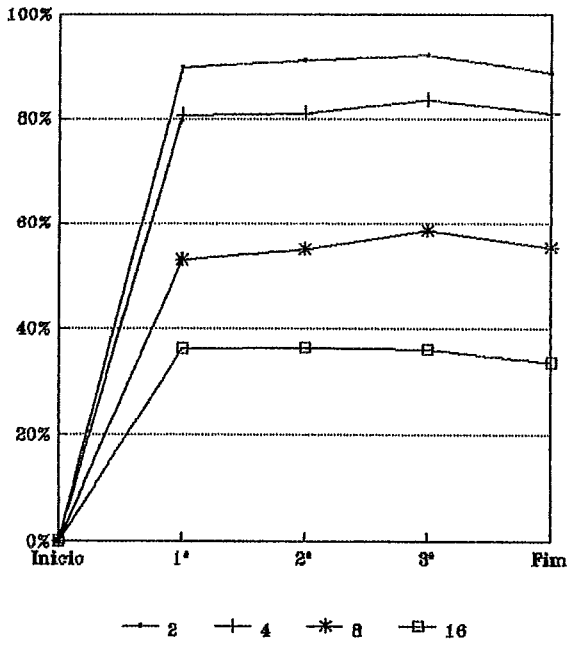
Gráfico E.26: Percentual do Processamento Total Realizado
— Densidade —

Backup





Kabu-Wake



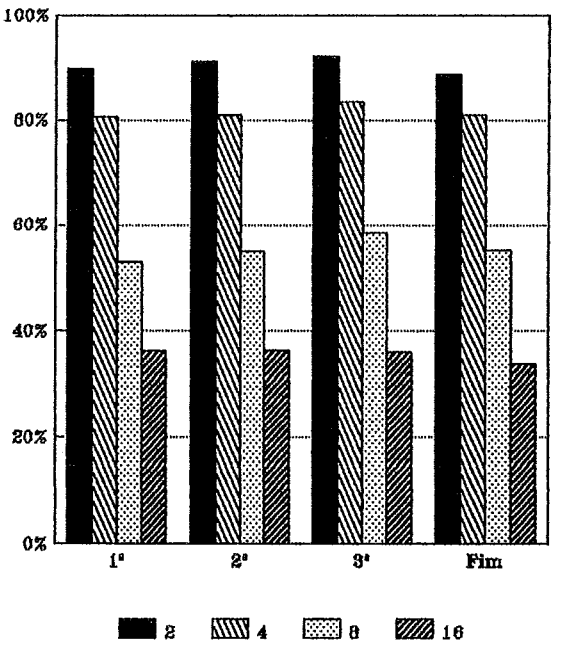
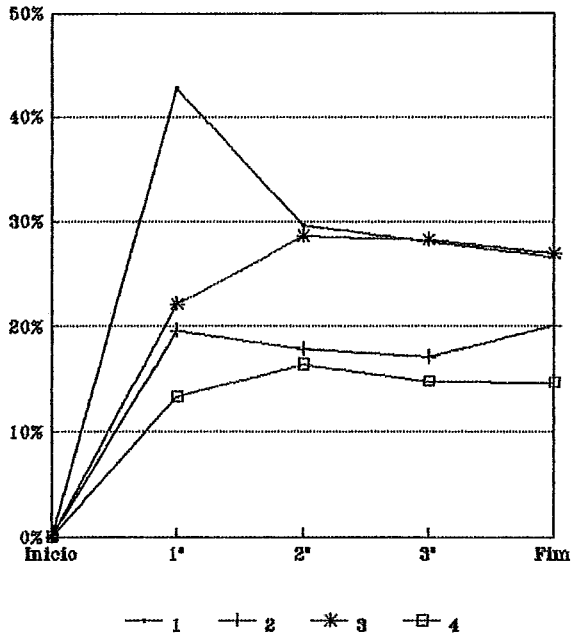
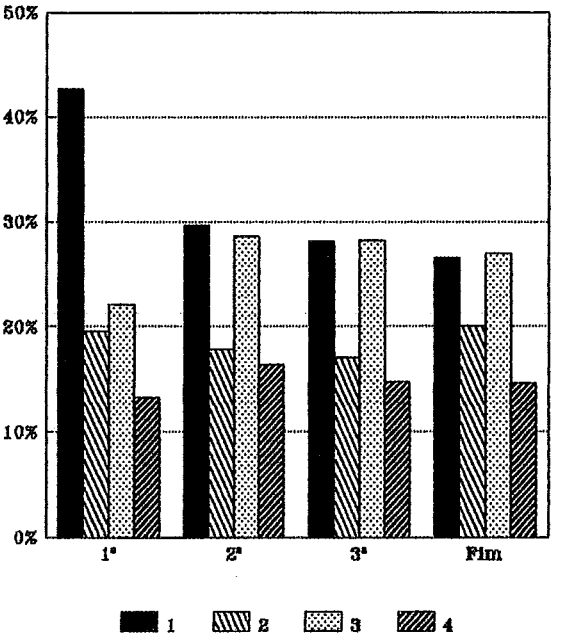


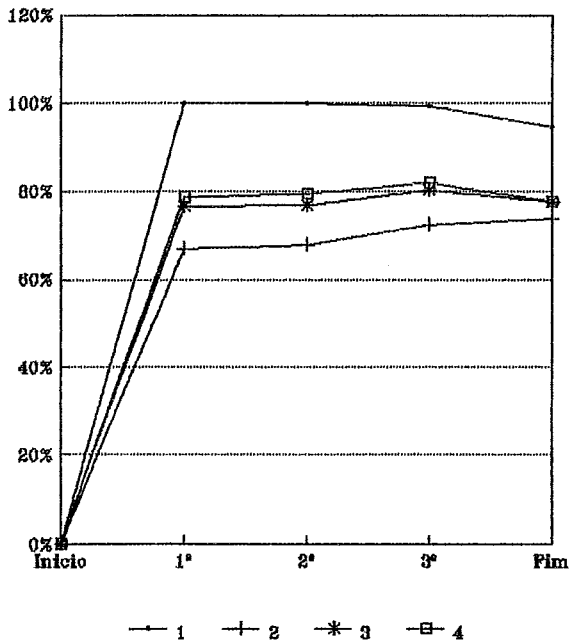
Gráfico E.27: Percentual de Processamento Útil PROLOG
Valores Médios de cada Simulação com Diferentes Número de Processadores
— Densidade —

Backup





Kabu-Wake



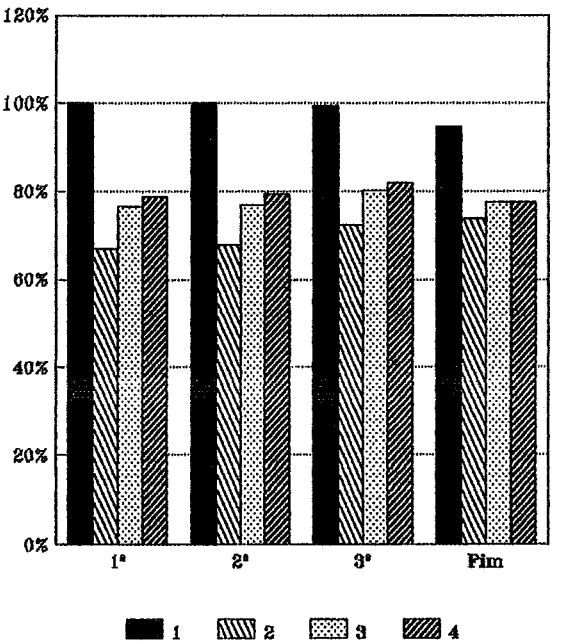
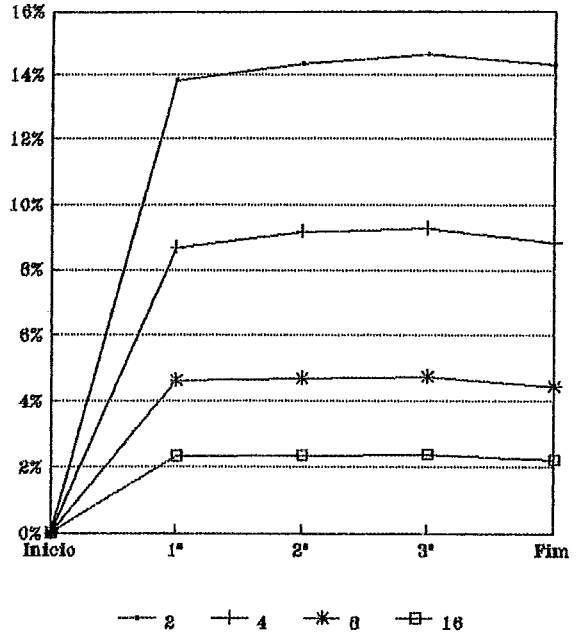
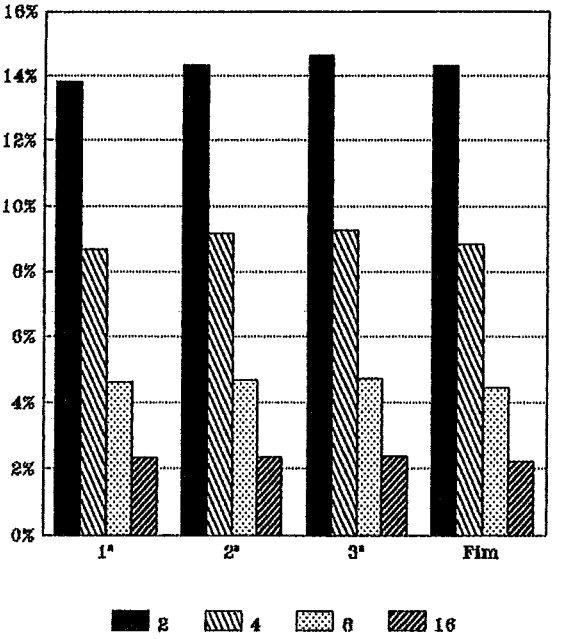


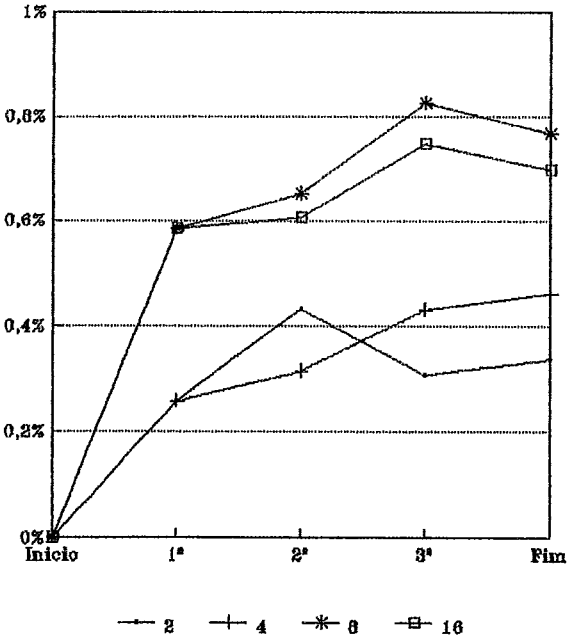
Gráfico E.28: Percentual de Processamento Útil PROLOG
Valores Individuais de cada Processador na Simulação com 4 Processadores
— Densidade —

Backup





Kabu-Wake



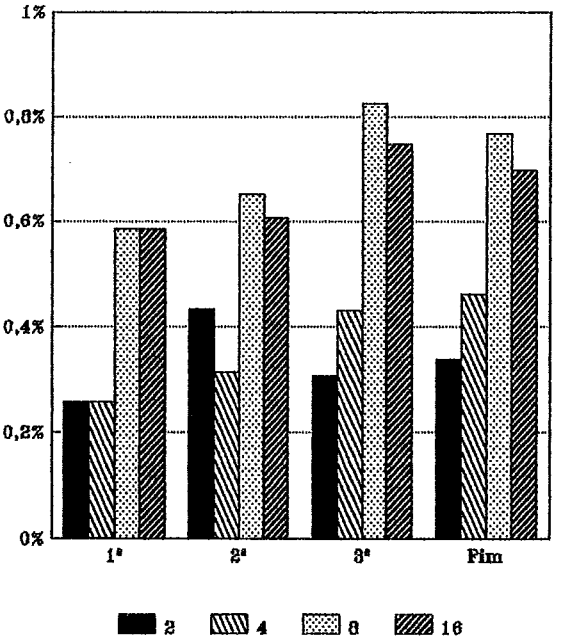
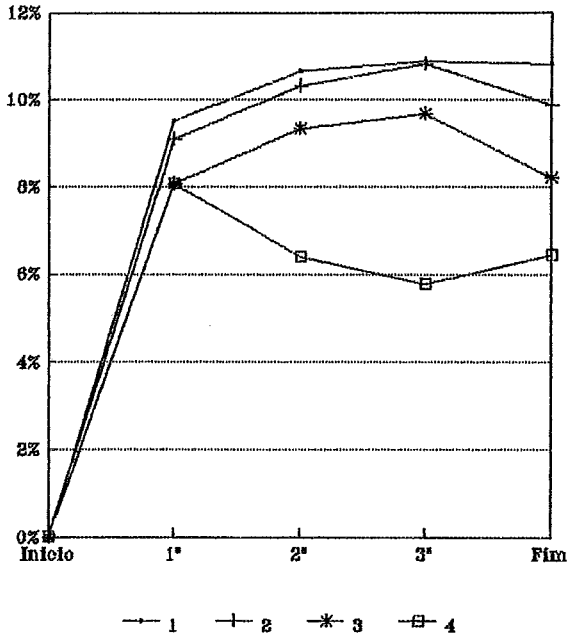
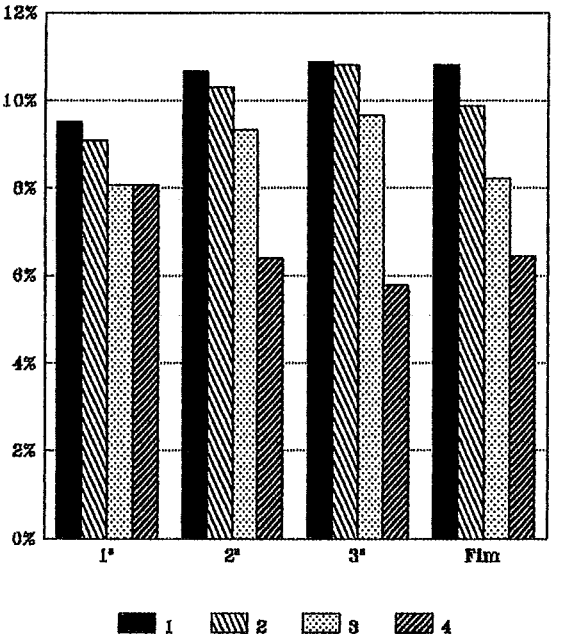


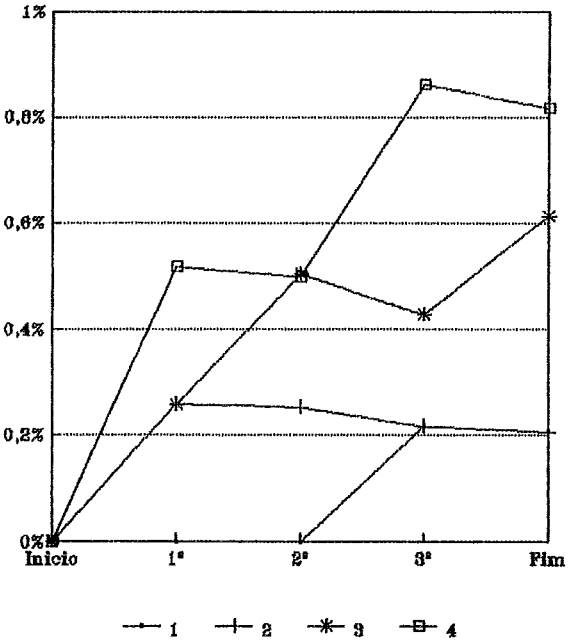
Gráfico E.29: Percentual de Processamento de Comunicações
Valores Médios de cada Simulação com Diferentes Número de Processadores
— Densidade —

Backup





Kabu-Wake



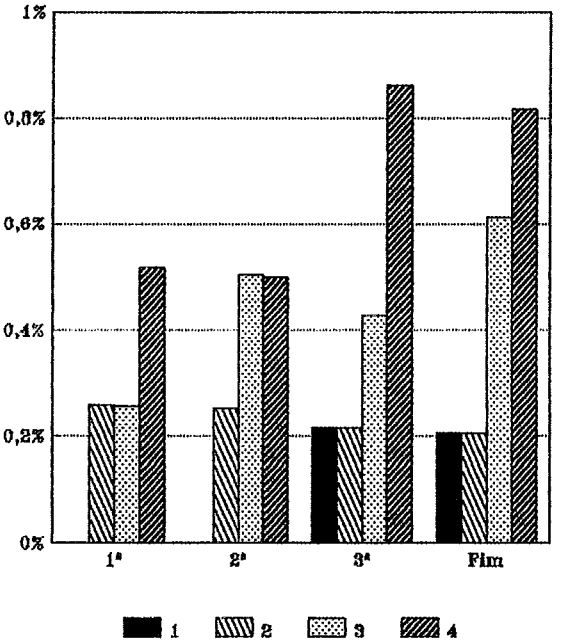
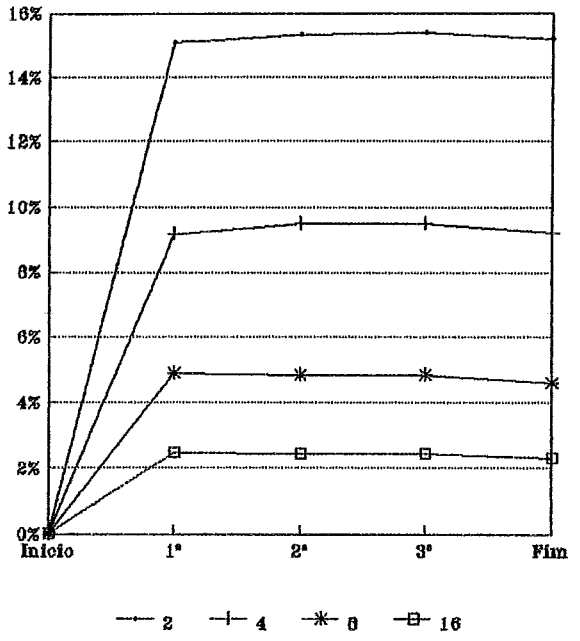
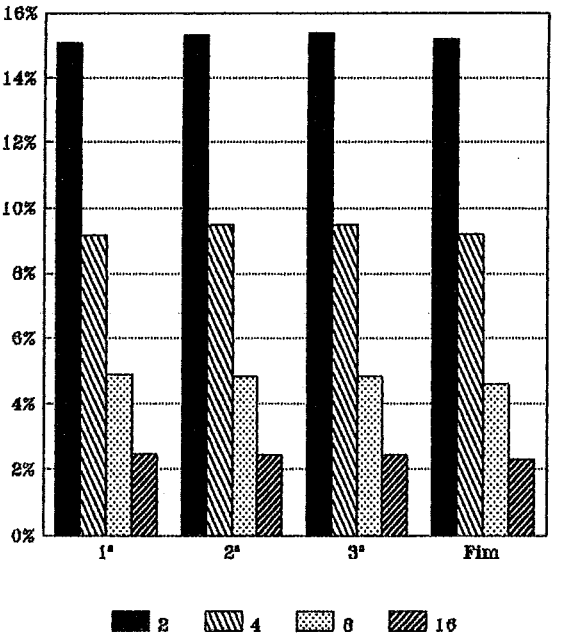


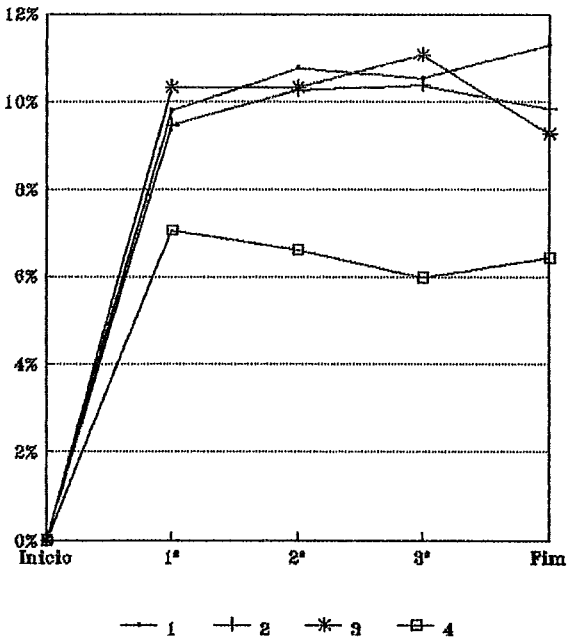
Gráfico E.30: Percentual de Processamento de Comunicações
Valores Individuais de cada Processador na Simulação com 4 Processadores
— Densidade —

Valores Médios de cada Simulação com Diferentes Número de Processadores





Valores Individuais de cada Processador na Simulação com 4 Processadores



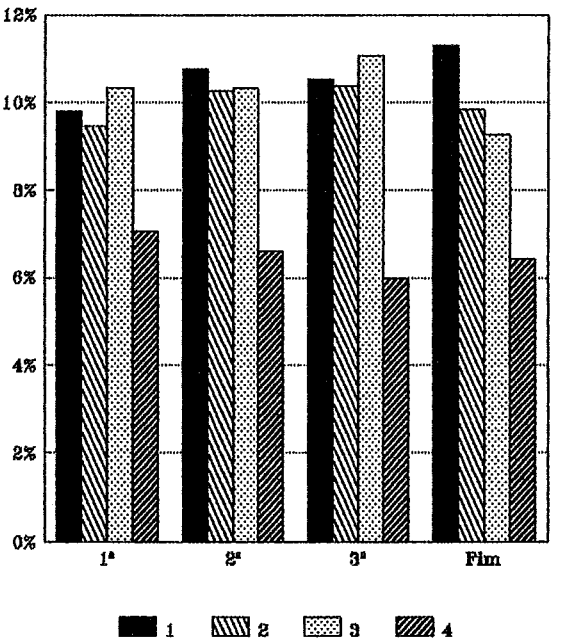
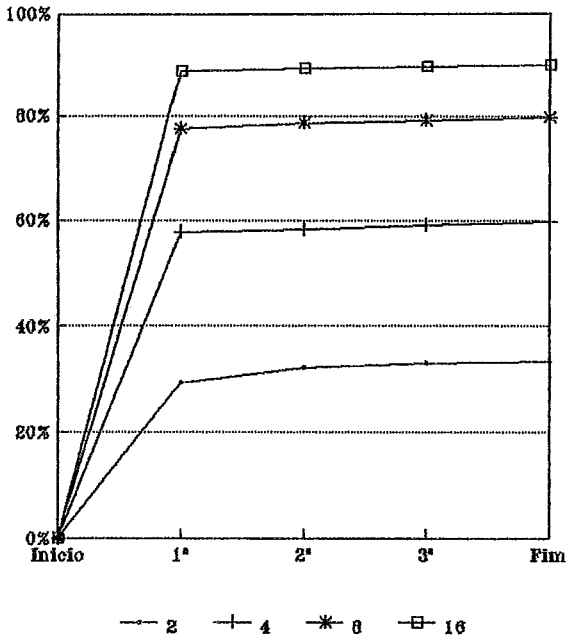
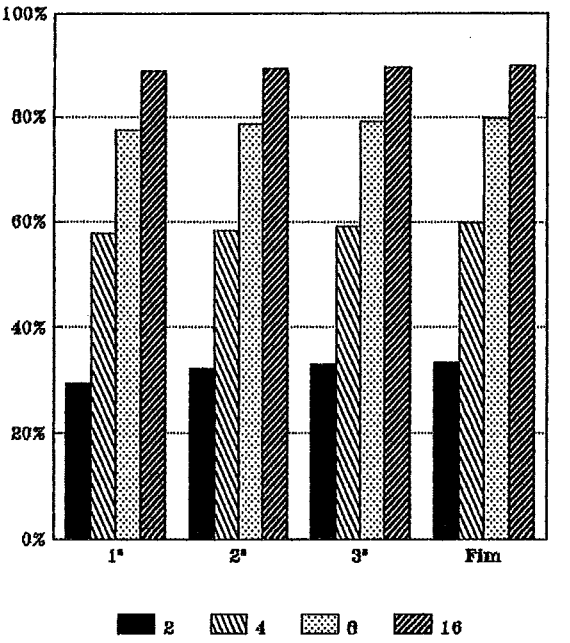


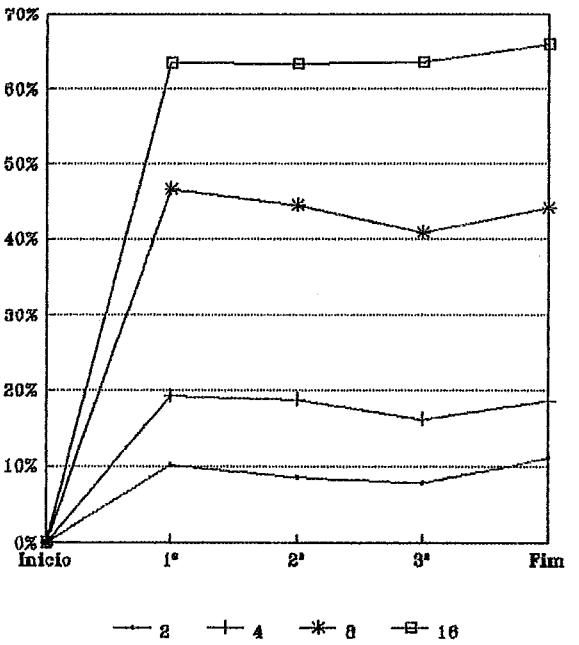
Gráfico E.31: Percentual de Processamento de Manutenção de Processos *Backup* — Densidade —

Backup





Kabu-Wake



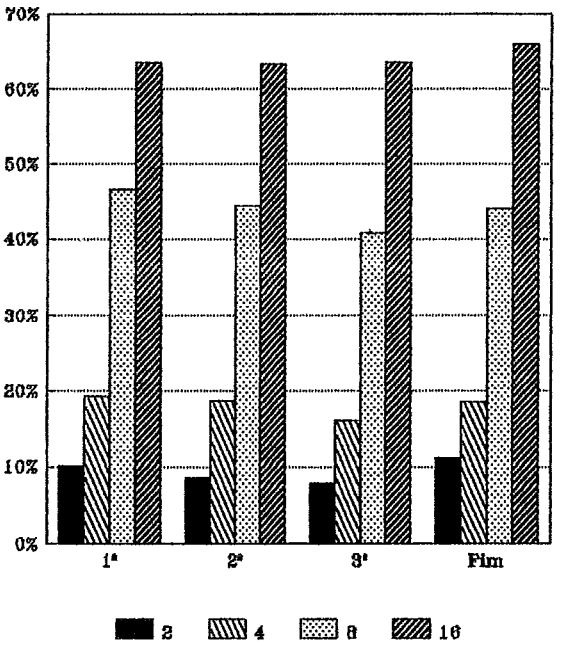
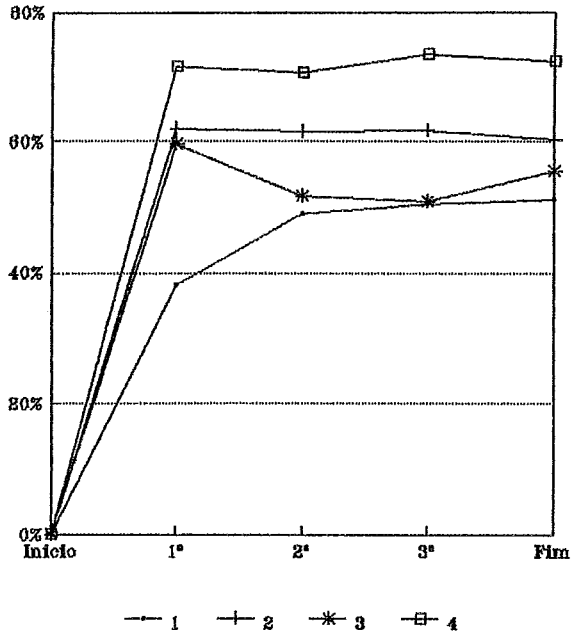
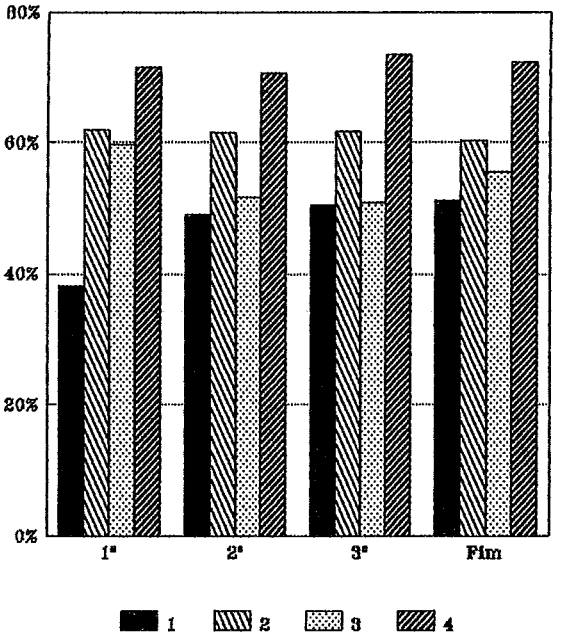


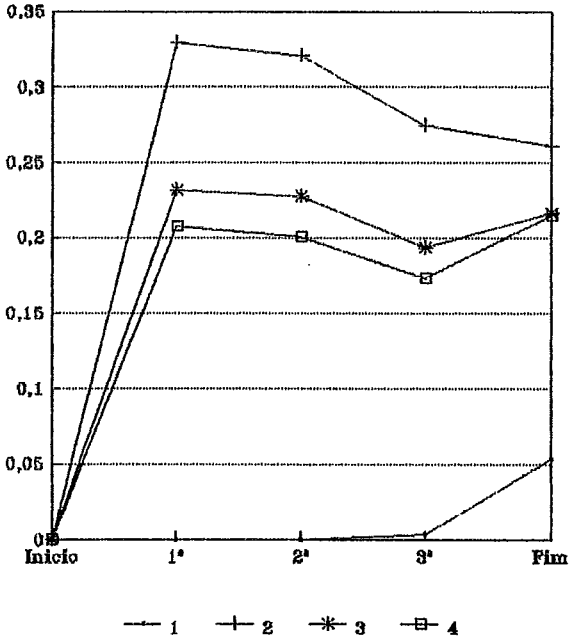
Gráfico E.32: Percentual de Ociosidade dos Processadores
Valores Médios de cada Simulação com Diferentes Número de Processadores
— Densidade —

Backup





Kabu-Wake



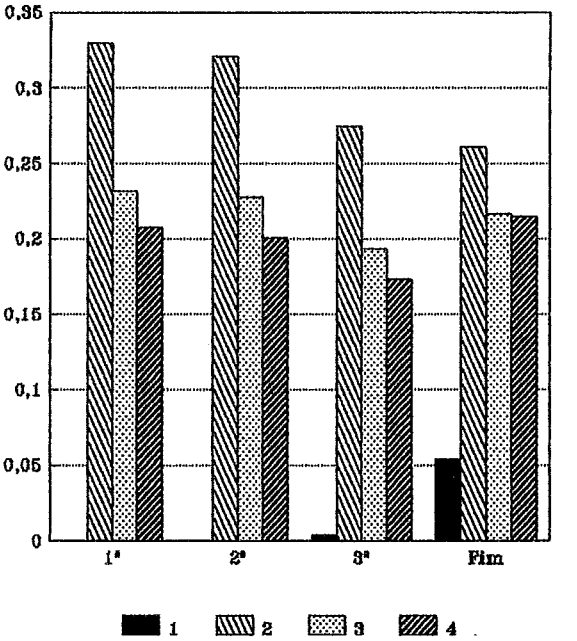
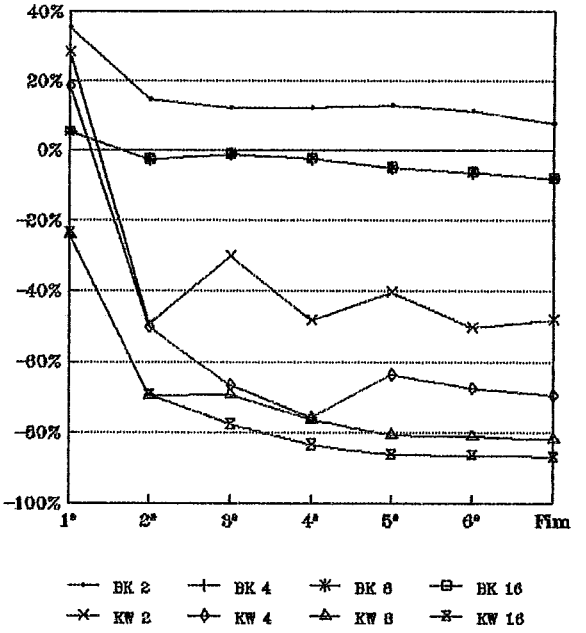


Gráfico E.33: Percentual de Ociosidade dos Processadores
Valores Individuais de cada Processador na Simulação com 4 Processadores
— Densidade —



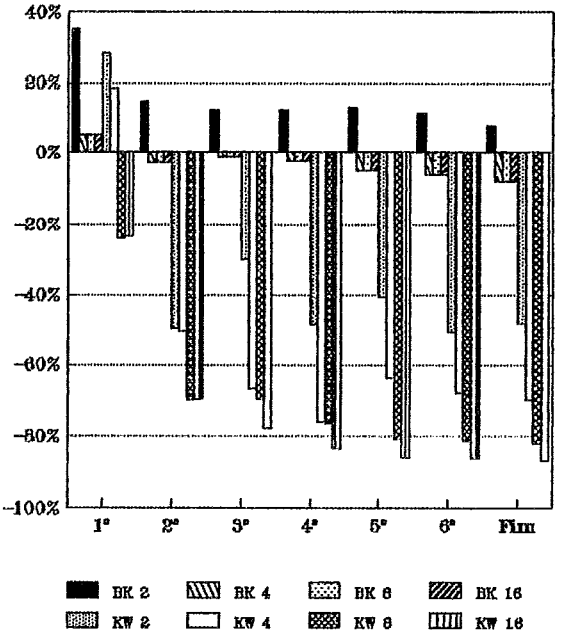
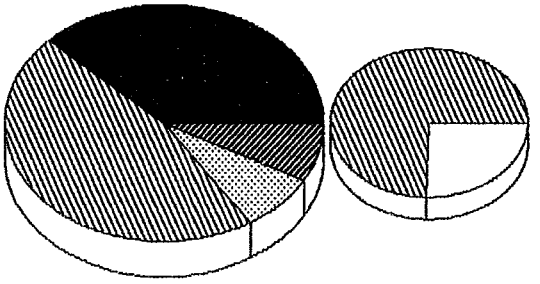
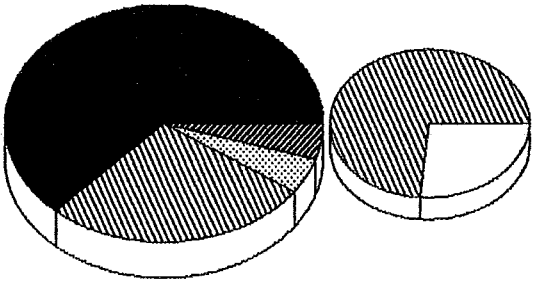


Gráfico E.34: Variação Percentual do Tempo do Seqüencial
— Coloração —

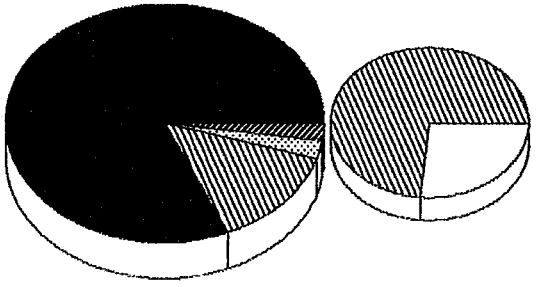
2 Processadores



4 Processadores



8 Processadores



16 Processadores

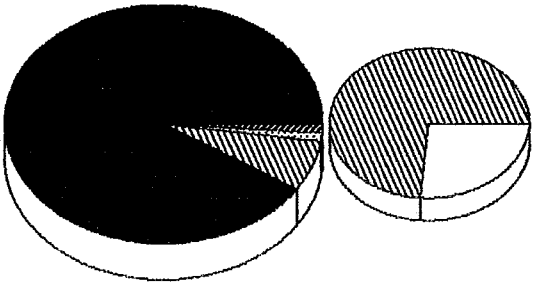
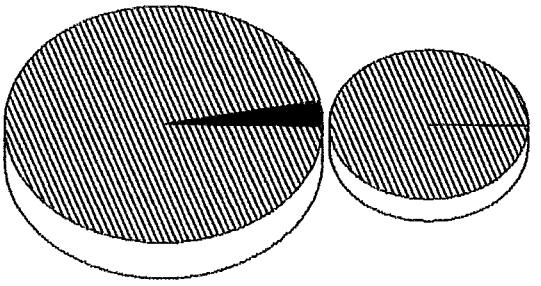


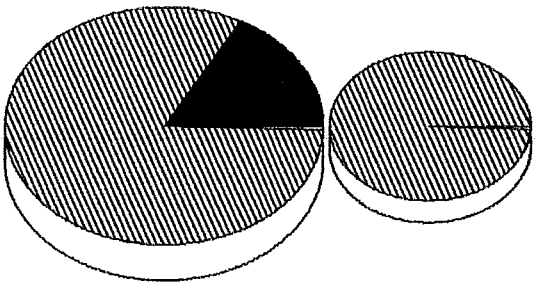
Gráfico E.35: Distribuições Percentuais dos Tempos Total e Efetivo de Processamento

Backup
— Coloração —

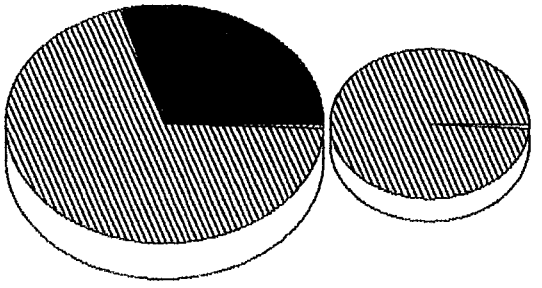
2 Processadores



4 Processadores



8 Processadores



16 Processadores

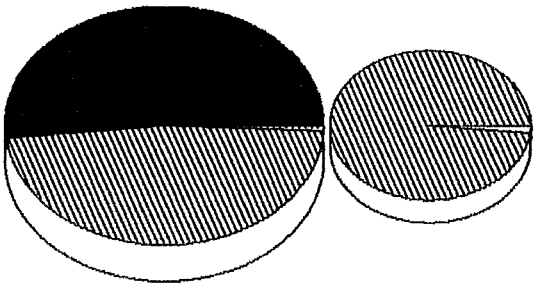
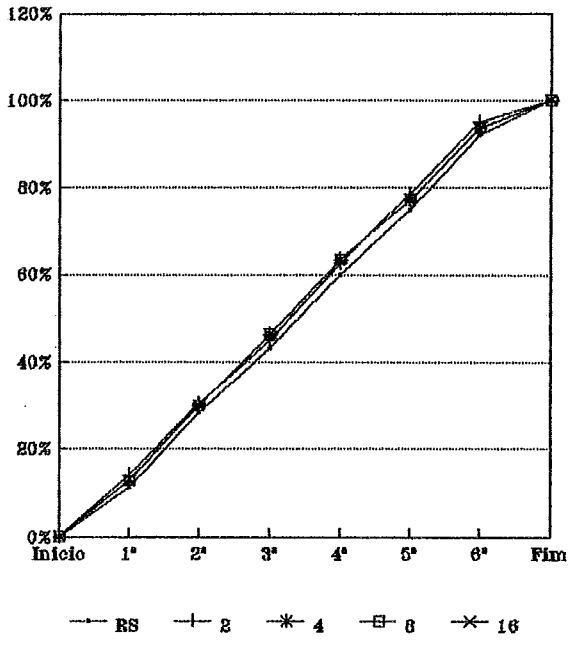
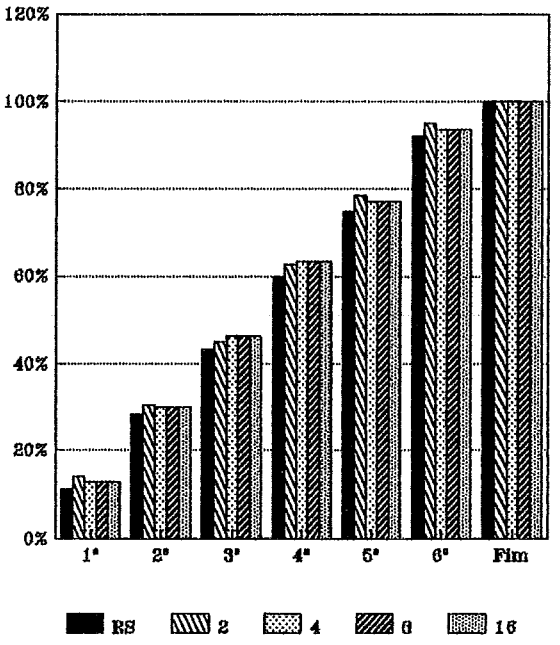


Gráfico E.36: Distribuições Percentuais dos Tempos Total e Efetivo de Processamento

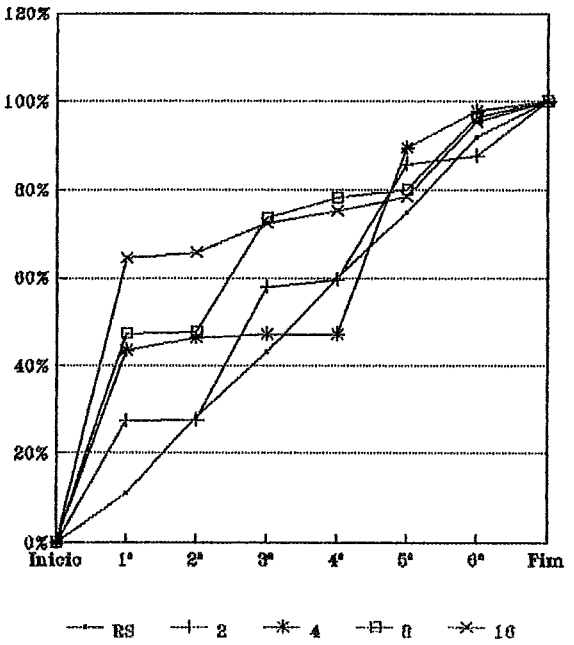
Kabu-Wake
— Coloração —

Backup





Kabu-Wake



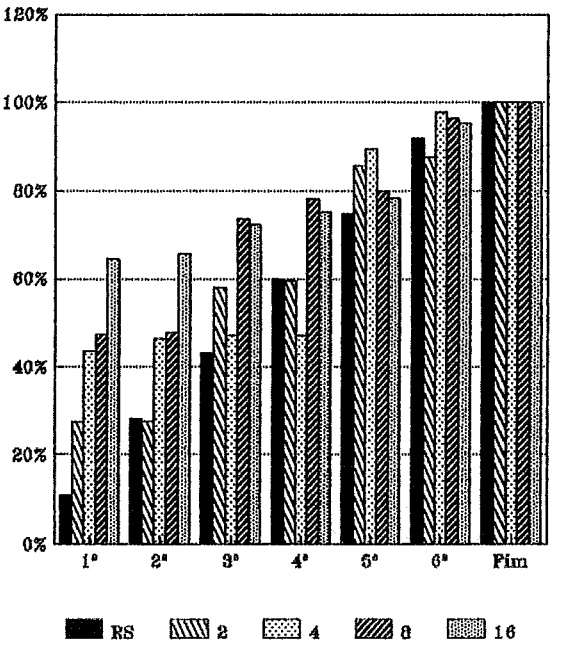
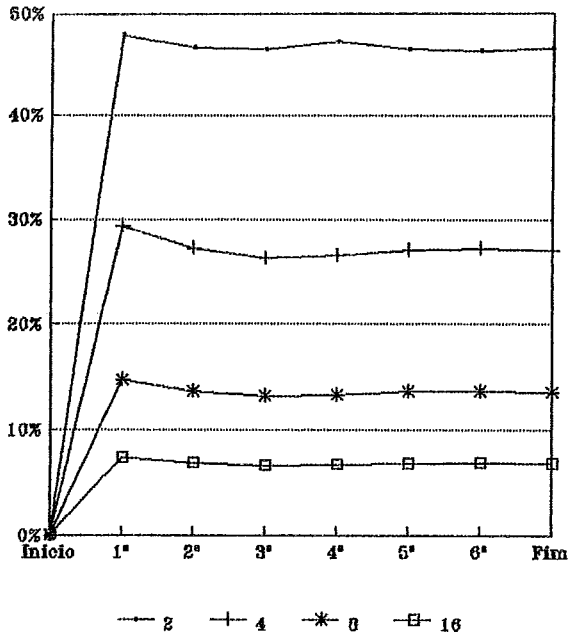
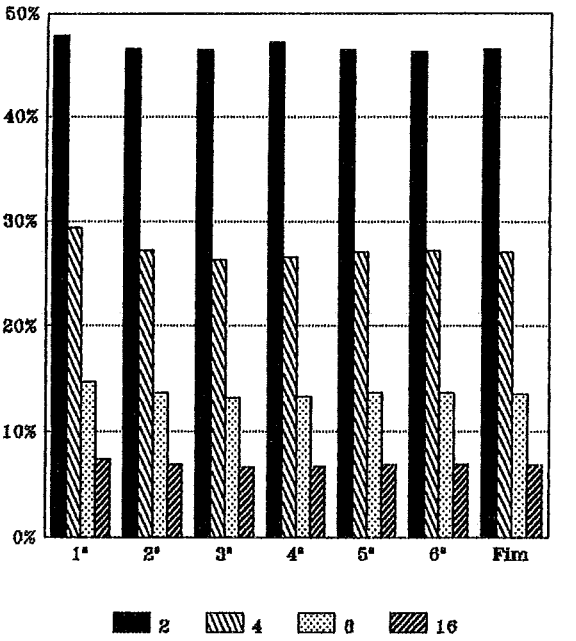


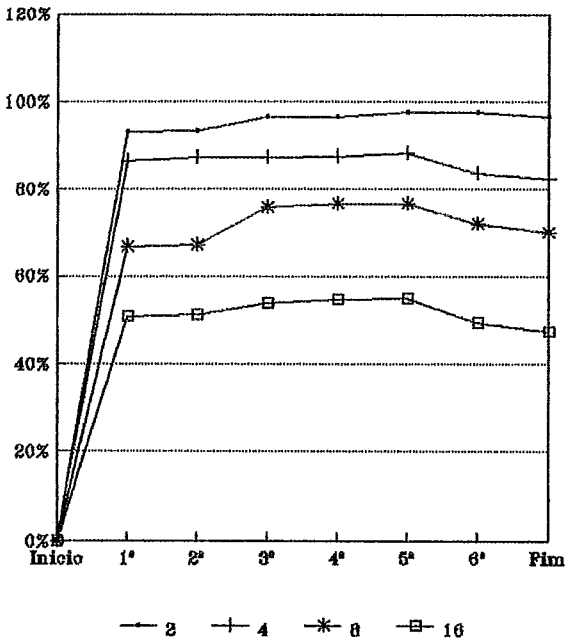
Gráfico E.37: Percentual do Processamento Total Realizado
— Coloração —

Backup





Kabu-Wake



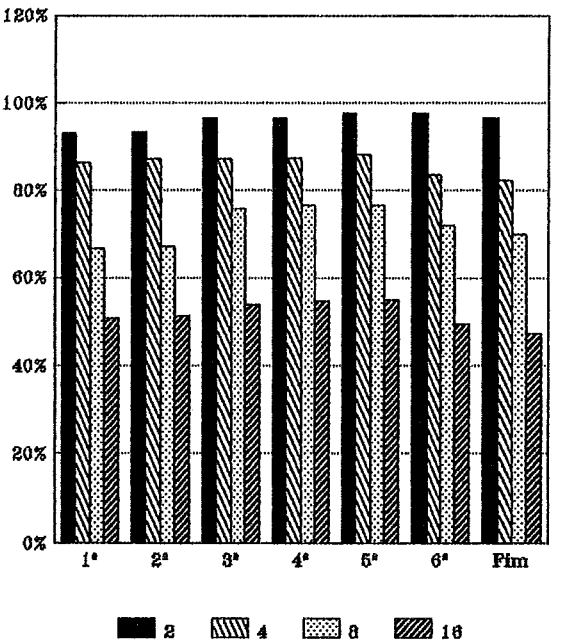
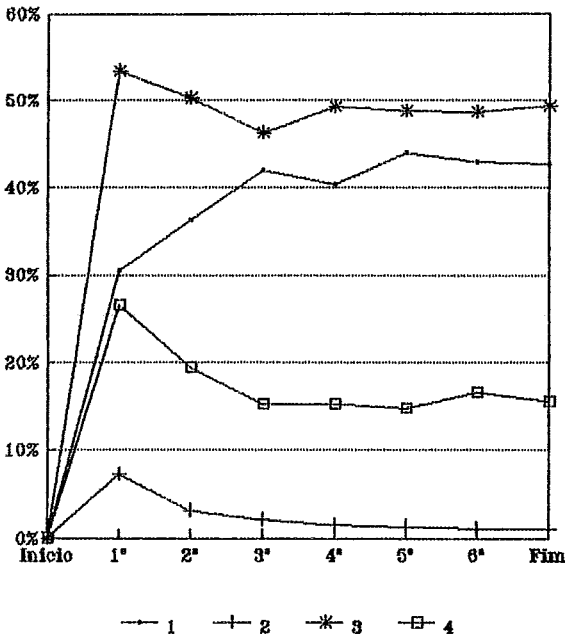
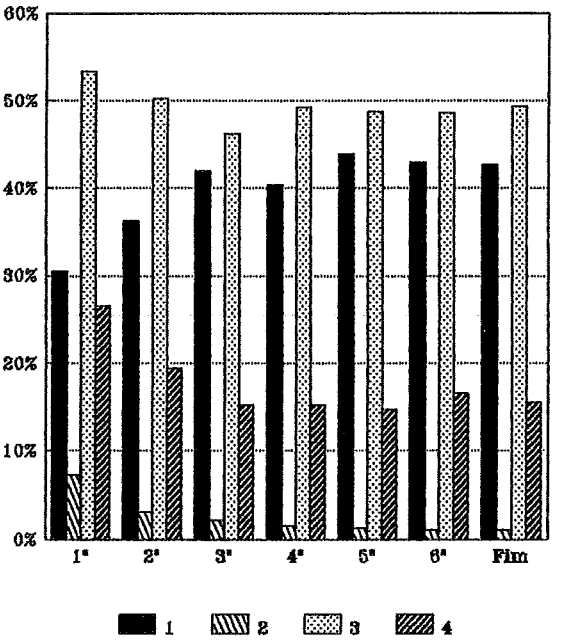


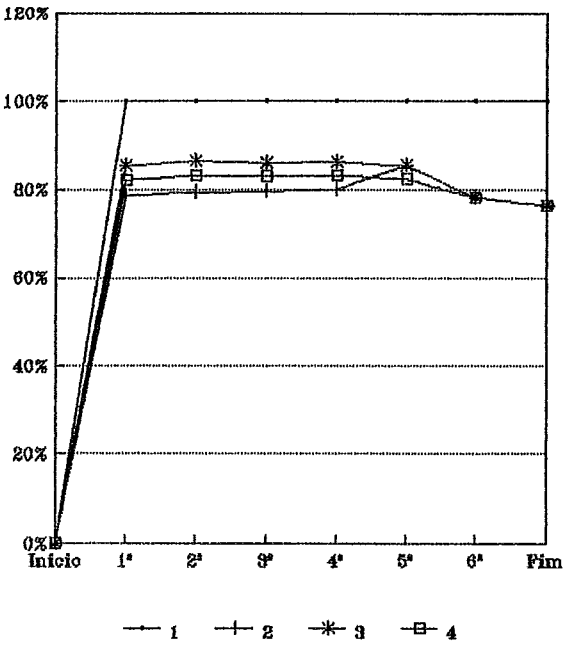
Gráfico E.38: Percentual de Processamento Útil PROLOG
Valores Médios de cada Simulação com Diferentes Número de Processadores
— Coloração —

Backup





Kabu-Wake



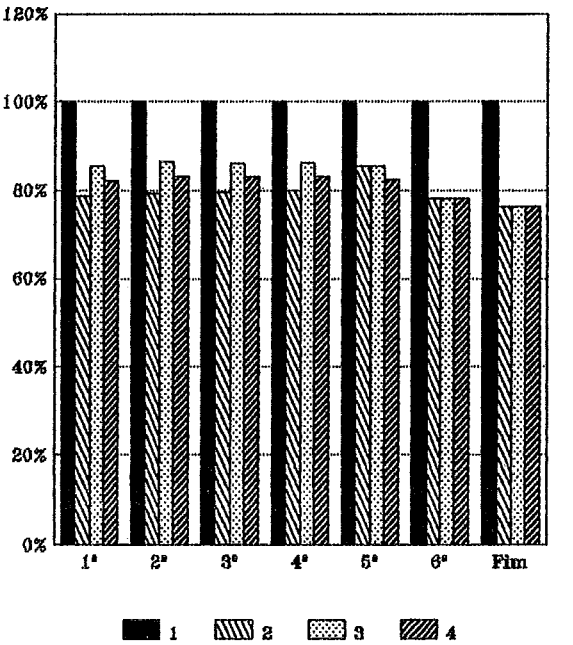
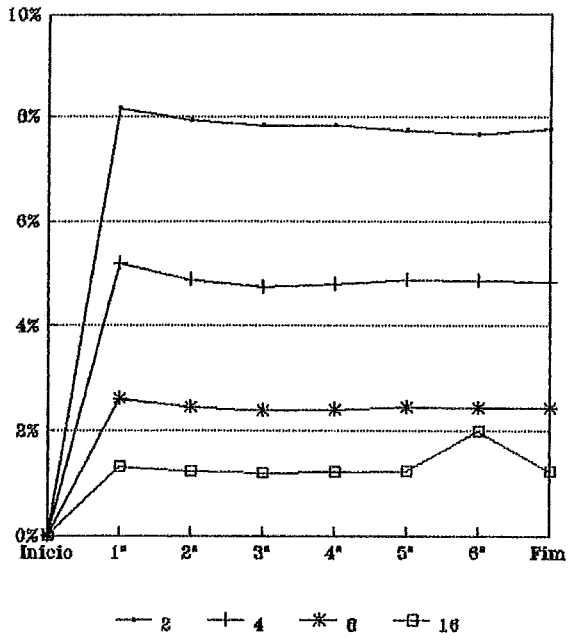
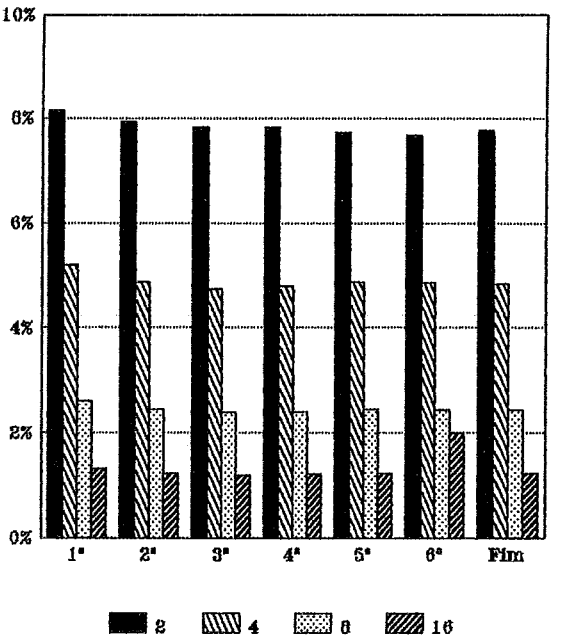


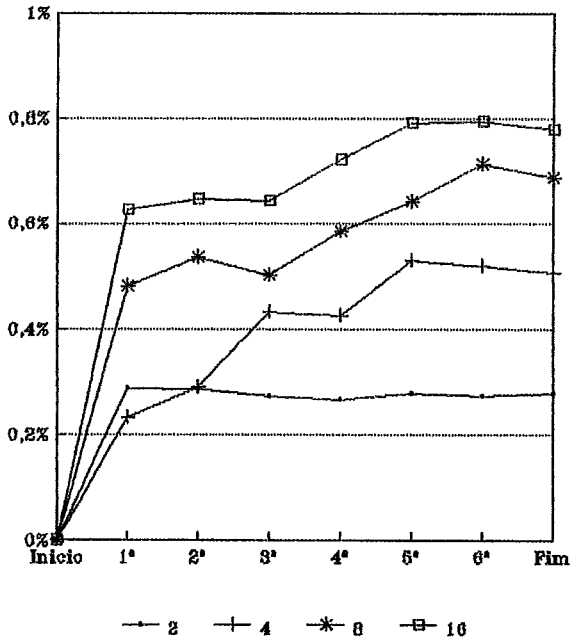
Gráfico E.39: Percentual de Processamento Útil PROLOG
Valores Individuais de cada Processador na Simulação com 4 Processadores
— Coloração —

Backup





Kabu-Wake



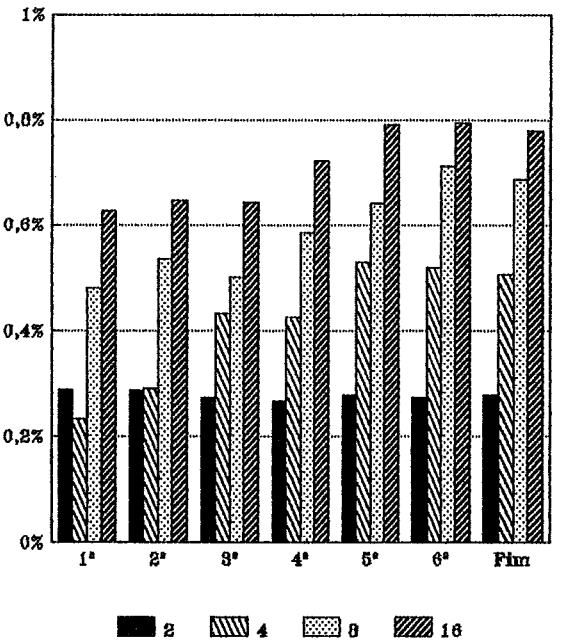
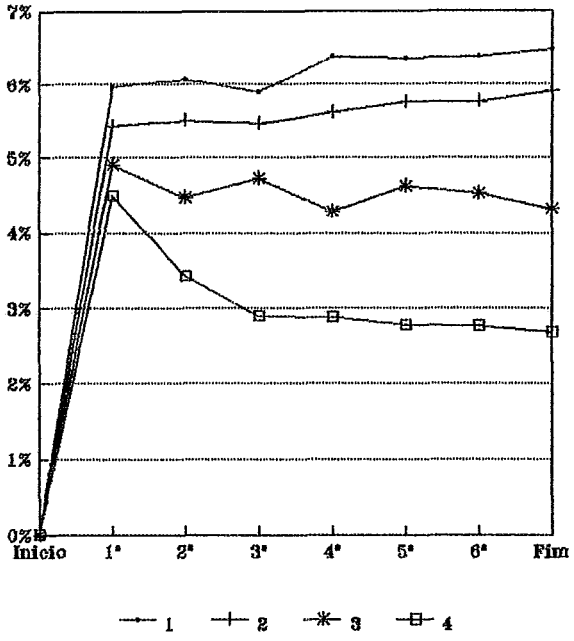
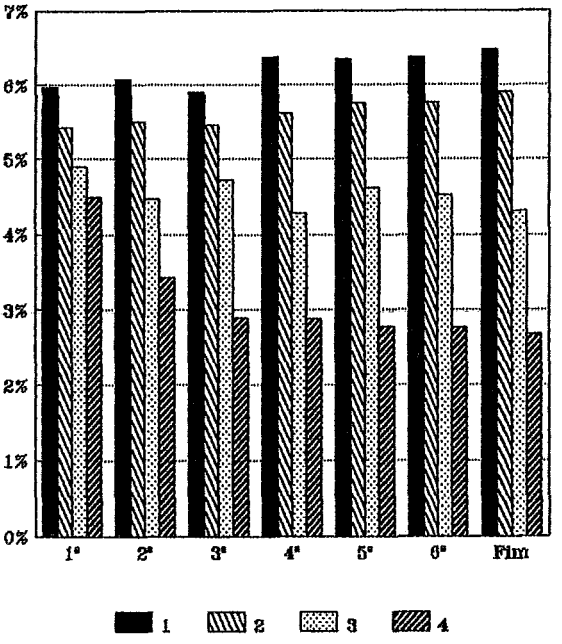


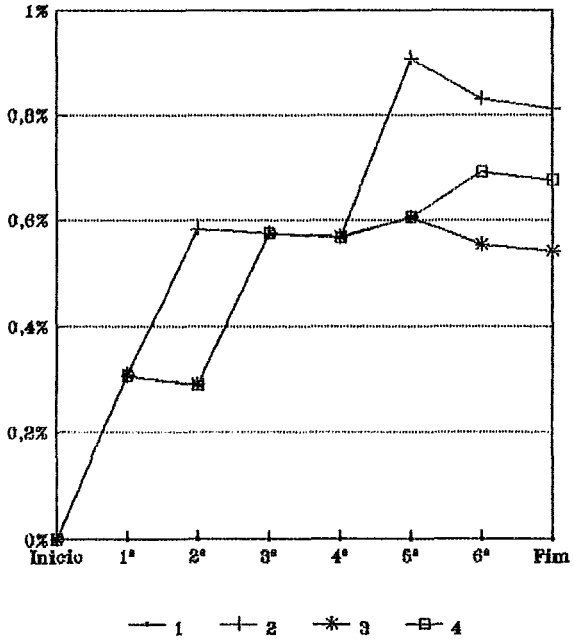
Gráfico E.40: Percentual de Processamento de Comunicações
Valores Médios de cada Simulação com Diferentes Número de Processadores
— Coloração —

Backup





Kabu-Wake



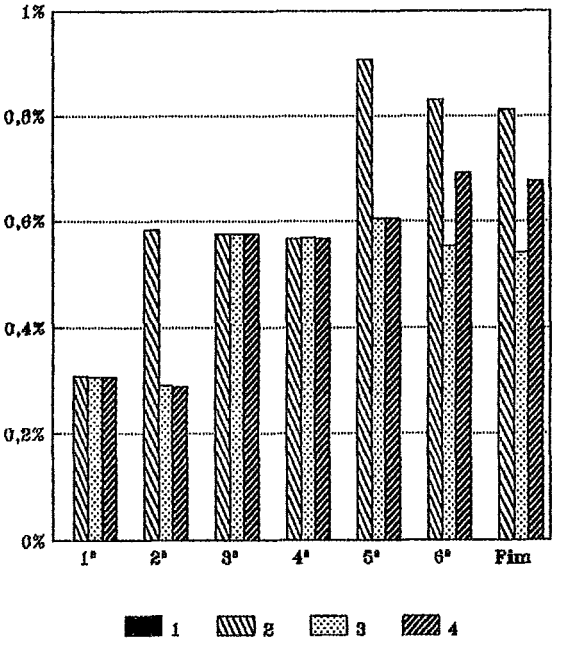
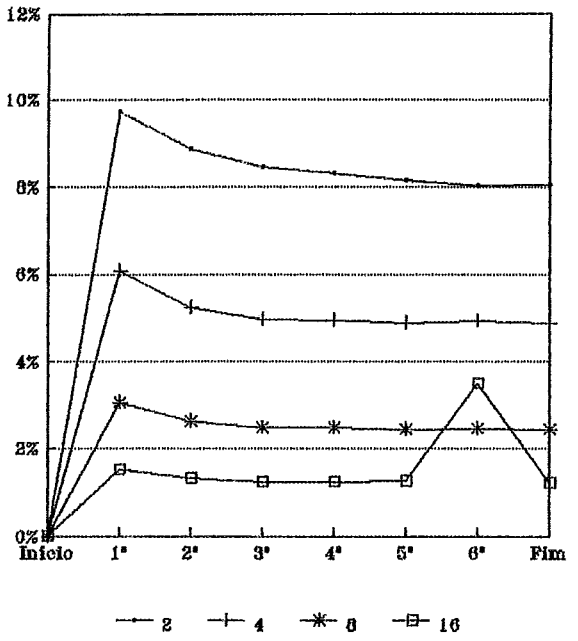
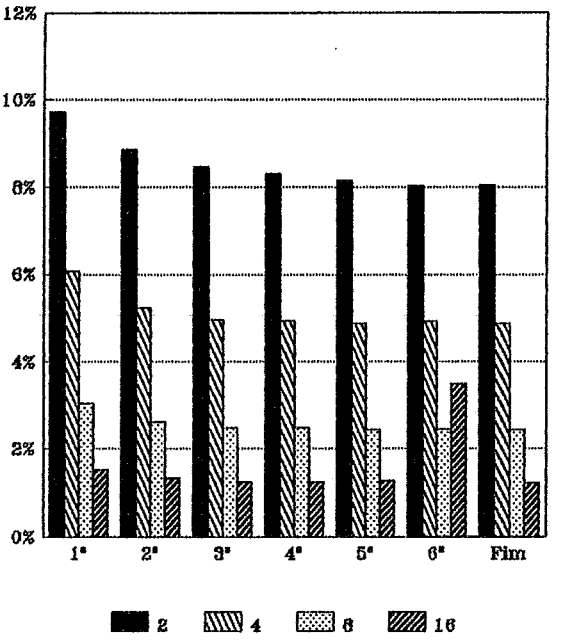


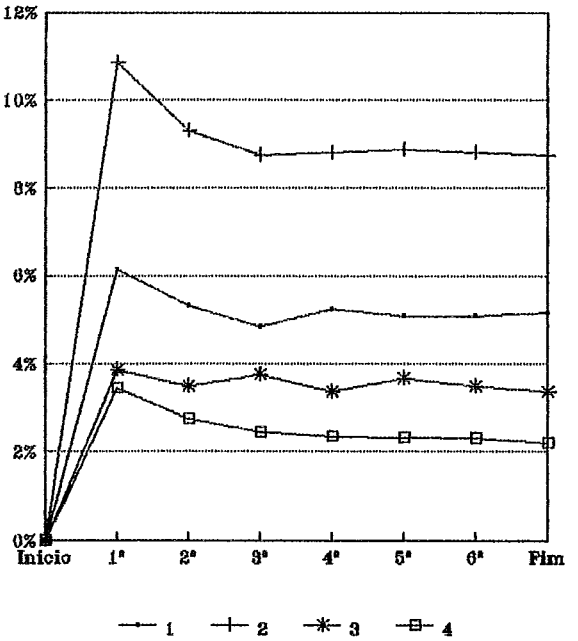
Gráfico E.41: Percentual de Processamento de Comunicações
Valores Individuais de cada Processador na Simulação com 4 Processadores
— Coloração —

Valores Médios de cada Simulação com Diferentes Número de Processadores





Valores Individuais de cada Processador na Simulação com 4 Processadores



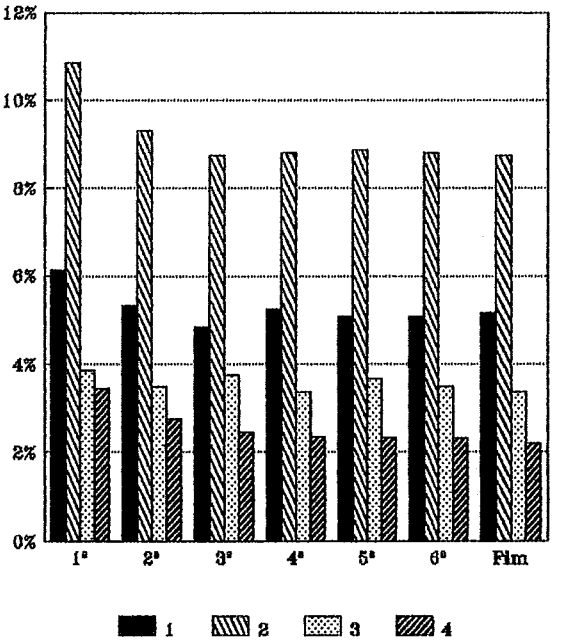
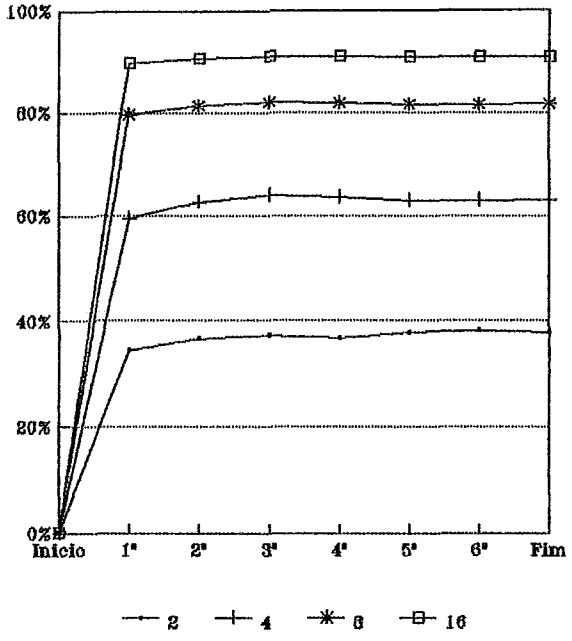
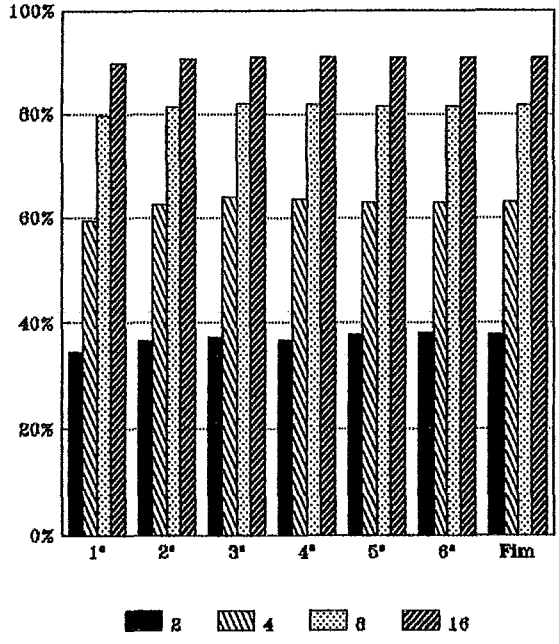


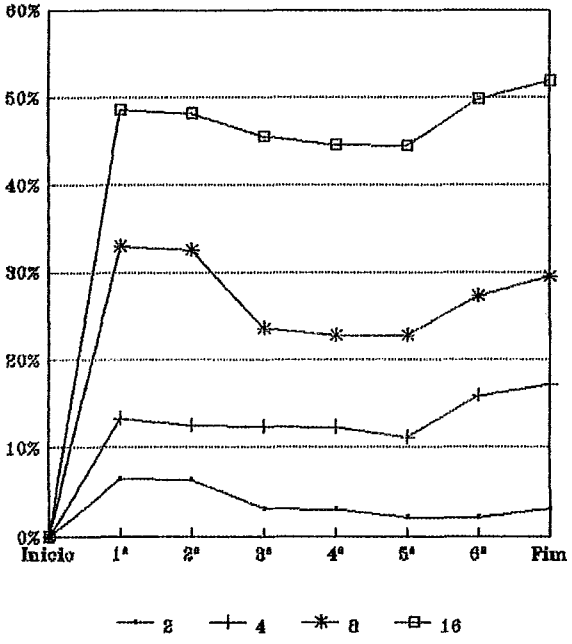
Gráfico E.42: Percentual de Processamento de Manutenção de Processos
Backup
— Coloração —

Backup





Kabu-Wake



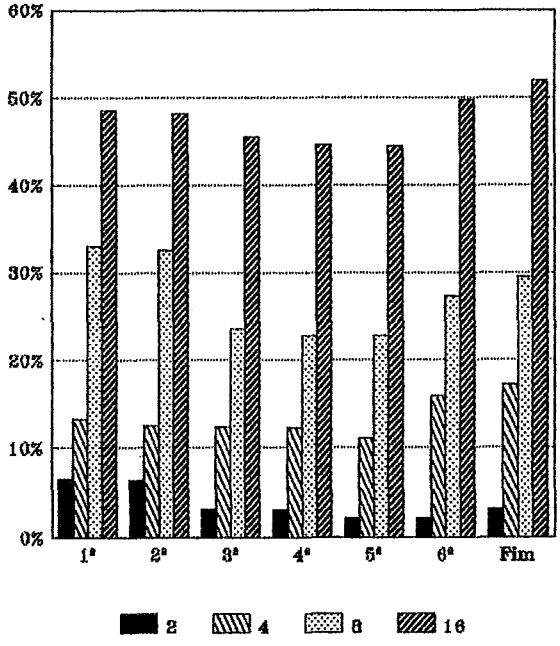
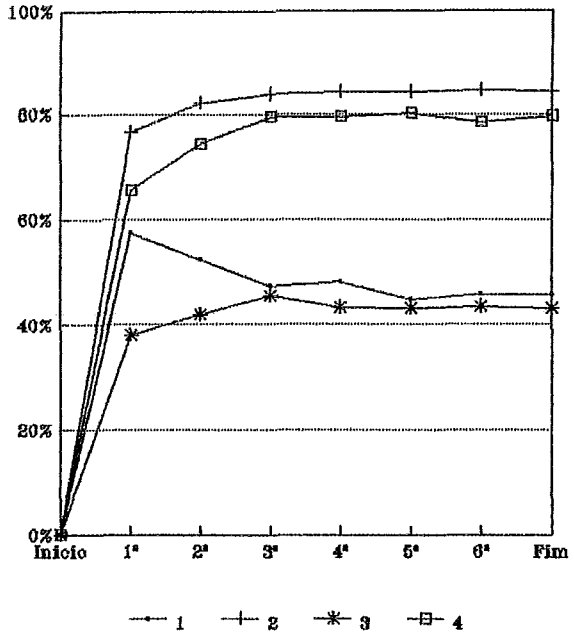
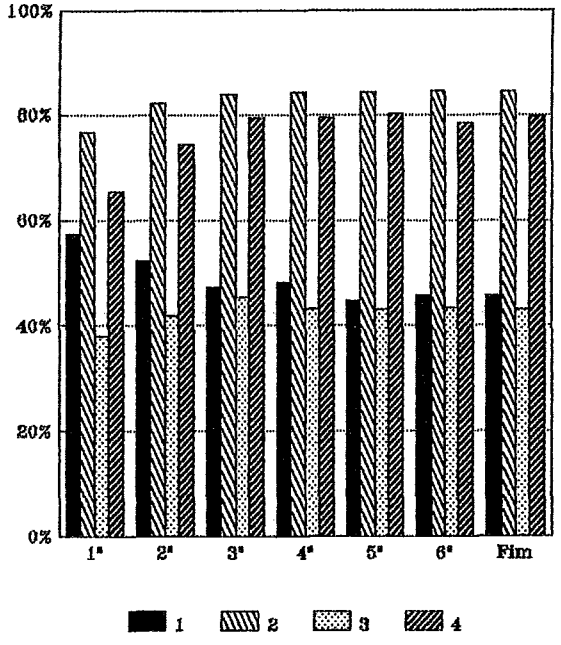


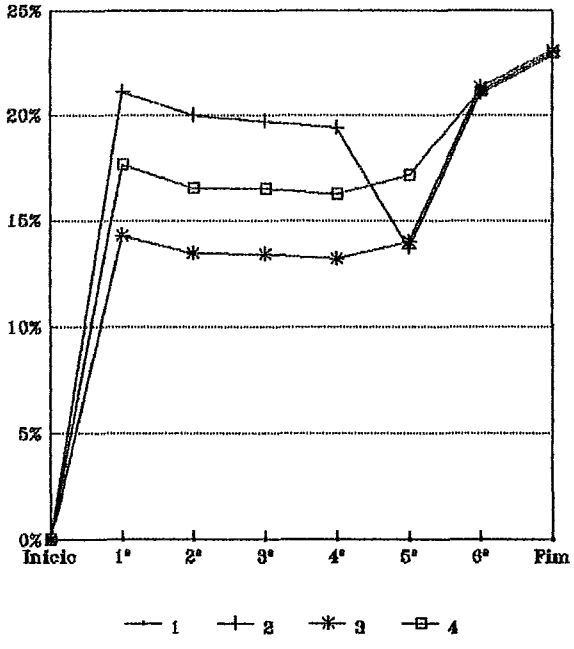
Gráfico E.43: Percentual de Ociosidade dos Processadores
Valores Médios de cada Simulação com Diferentes Número de Processadores
— Coloração —

Backup





Kabu-Wake



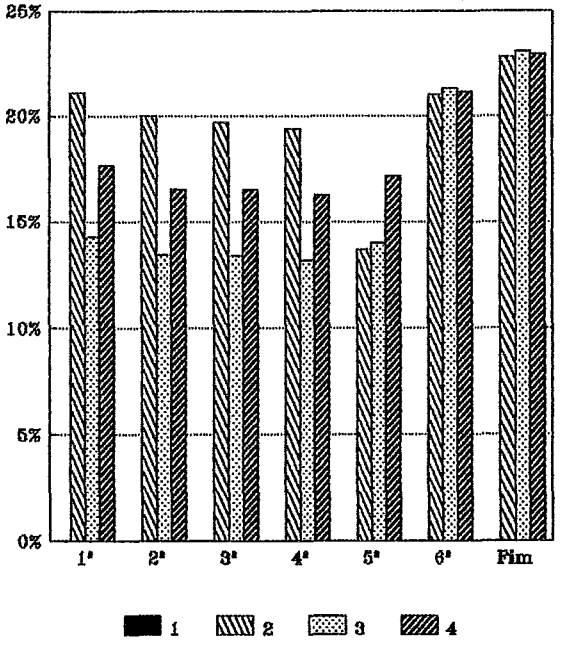


Gráfico E.44: Percentual de Ociosidade dos Processadores
Valores Individuais de cada Processador na Simulação com 4 Processadores
— Coloração —

Referências Bibliográficas

[Aho 86] A. V. Aho, R. Sethi e J. Ullman "COMPILERS – PRINCIPLES, TECHNIQUES AND TOOLS". Addison Wesley, 1986.

[Arity 86] Arity Corporation – CSA Press, 1986.

[Bianchini 89] Ricardo G. Bianchini "EXECUÇÃO PARALELA DE PROGRAMAS LÓGICOS". *Projeto Final de Graduação*, UFRJ, 1989.

[Bianchini 90] Ricardo G. Bianchini "UM NOVO MODELO DE EXECUÇÃO PARALELA DE PROGRAMAS LÓGICOS". *Tese de Mestrado*, COPPE / UFRJ, 1990.

[Casanova 87] A. M. Casanova, F. A. C. Giorno e A. L. Furtado "PROGRAMAÇÃO EM LÓGICA E LINGUAGEM PROLOG". Edgard Blucher, 1987.

[Cohen 85] J. Cohen "DESCRIBING PROLOG BY ITS INTERPRETATION AND COMPILATION". CACM – v. 28, nº 12, Dec 1985.

[Conery 87] John S. Conery "PARALLEL EXECUTION OF LOGIC PROGRAMS". Kluwer Academic Publishers, 1987.

[DeGroot 84] D. DeGroot "RESTRICTED AND PARALLELISM. *Proceedings of the International Conference on Fifth Generation Computer Systems*, (Tokio, Japão), 1984, pp. 471–478.

[Dutra 88a] Inês de Castro Dutra "IMPLEMENTAÇÃO DE UMA MÁQUINA VIRTUAL PROLOG – TRADUÇÃO E EXECUÇÃO DE PROGRAMAS". *Tese de Mestrado*, COPPE / UFRJ, 1988.

[Dutra 88b] Inês de C. Dutra, Ricardo G. Bianchini, Leila M. Eizirik e Cláudio L. de Amorim "EM DIREÇÃO A UMA ESTAÇÃO PROLOG DE ALTO DESEMPENHO - TRADUÇÃO DE PROGRAMAS". *V Simpósio Brasileiro de Inteligência Artificial.* Natal, RN, Outubro de 1988.

[Fox 88] Geoffrey C. Fox (et ali) "SOLVING PROBLEMS ON CONCURRENT PROCESSORS" VOL I. Prentice Hall, 1988.

[Furukawa 82] K. Furukawa, K. Nitta e Y. Matsumoto "PROLOG INTERPRETER BASED ON CONCURRENT PROGRAMMING". *Proceedings of the First International Logic Conference,* (Faculté des Sciences de Luminy, Marseille, France, Sept.), 1982, pp. 38–44.

[Gregory 87] Steve Gregory "PARALLEL LOGIC PROGRAMMING IN PARLOG". Addison–Wesley, 1987.

[Hermenegildo 86] Manuel V. Hermenegildo "AN ABSTRACT MACHINE FOR RESTRICTED AND-PARALLEL EXECUTION OF LOGIC PROGRAMS". *Proceedings of the* 3^{rd} *Int'l. Conference on Logic Programming.* Springer-Verlag, 1986, pp. 25–39.

[Hogger 84] C. J. Hogger "INTRODUCTION TO LOGIC PROGRAMMING". Academic Press, 1984.

[Jacquet 87] J. M. Jacquet "A GUIDED TOUR THROUGH PARALLELISM IN LOGIC PROGRAMMING", National Fund for Scientific Reserch, Institut d'Informatique – F.N.D.P., 1987.

[Kowaltowski 83] Tomasz Kowaltowski "IMPLEMENTAÇÃO DE LINGUAGENS DE PROGRAMAÇÃO". Guanabara Dois, 1983.

[Lima 86] P. M. V. Lima "CONSIDERAÇÕES SOBRE UM SISTEMA PROLOG PARA AMBIENTE DE MULTIPROCESSAMENTO". *VI Congresso da Sociedade Brasileira de Computação,* Anais (Volume I), Recife – Olinda, Jul 1986.

[Lima 87] P. M. V. Lima "IMPLEMENTAÇÃO DE COMPILADORES PROLOG". *Tese de Mestrado,* COPPE/UFRJ, 1987.

[Lipovski 85] G. J. Lipovski e Manuel V. Hermenegildo "B-LOG: A BRANCH AND BOUND METHODOLOGY FOR THE PARALLEL EXECUTION OF LOGIC PROGRAMS". *Proceedings of the 1985 International Conference on Parallel Processing*, Agosto de 1985.

[Peterson 85] James L. Peterson, Abraham Silberschatz "OPERATING SYSTEM CONCETPTS". Addison Wesley, 1985.

[Stone 87] Harold S. Stone "HIGH-PERFORMANCE COMPUTER ARCHITECTURES". Addison Wesley, 1987.

[Sohma 85] Yukio Sohma, K. Saroh, K. Kumon, H. Masuzawa e A. Itashiki "A NEW PARALLEL INFERENCE BASED ON SEQUENTIAL PROCESSING". *Proceedings of the IFIP TC 10 Working Conference on Fifth Generation Computer Architetures 85*, (July 15-18), 1985, pp. 3–14.

[Warren 77] David h. D. Warren "IMPLEMENTING PROLOG - COMPILING LOGIC PROGRAMS". Vols 1,2, DAI Research Reports nºs 39, 40, Department of Artificial Intelligence, University of Edinburgh, May 1977.

[Warren 80] David H. D. Warren "LOGIC PROGRAMMING AND COMPILER WRITING". Software - Pratice and Experience, v 10, pp. 97 - 125, 1980.

[Warren 83] David H. D. Warren "AN ABSTRACT PROLOG INSTRUCTION SET". Techinical Note 309, SRI International, AI Center, Computer Science and Technology Division, 1983.

[Wise 87] Michael J. Wise "PROLOG MULTIPROCESSORS". Prentice Hall, 1987.