COPPE/UFRJ

UM ESTUDO DO POTENCIAL DE REDES INTRACHIP PARA O ESCALONAMENTO DE PROCESSOS EM ARQUITETURAS MULTICORE

Alexandre Borges Gonçalves

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientadores: Felipe Maia Galvão França
Valmir Carneiro Barbosa

Rio de Janeiro
Setembro de 2010

UM ESTUDO DO POTENCIAL DE REDES INTRACHIP PARA O ESCALONAMENTO DE PROCESSOS EM ARQUITETURAS MULTICORE

Alexandre Borges Gonçalves

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) 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.

Examinada por:

Prof. Felipe Maia Galvão França, Ph.D.

Prof. Valmir Carneiro Barbosa, Ph.D.

Prof. Claudio Luis de Amorim, Ph.D.

Prof. Luiza de Macedo Mourelle, Ph.D.

RIO DE JANEIRO, RJ – BRASIL
SETEMBRO DE 2010

Gonçalves, Alexandre Borges

Um Estudo do Potencial de Redes Intrachip para o Escalonamento de Processos em Arquiteturas Multicore/Alexandre Borges Gonçalves. – Rio de Janeiro: UFRJ/COPPE, 2010.

XI, 108 p.: il.; 29, 7cm.

Orientadores: Felipe Maia Galvão França

Valmir Carneiro Barbosa

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2010.

Referências Bibliográficas: p. 102 – 105.

1. Arquitetura de Computadores. 2. Escalonamento de Processos. 3. Rede Intrachip. 4. Arquiteturas Multicores. I. França, Felipe Maia Galvão *et al.*. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

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

UM ESTUDO DO POTENCIAL DE REDES INTRACHIP PARA O ESCALONAMENTO DE PROCESSOS EM ARQUITETURAS MULTICORE

Alexandre Borges Gonçalves

Setembro/2010

Orientadores: Felipe Maia Galvão França
Valmir Carneiro Barbosa

Programa: Engenharia de Sistemas e Computação

Neste trabalho propusemos um mecanismo para escalonamento de processos em uma rede *intrachip*. Esse esquema foi simulado por meio de uma linguagem de projeto de sistemas e propriedades como balanceamento de carga e escalabilidade foram analisadas. Adicionalmente, contrastamos os resultados obtidos por simulação com aqueles esperados de modelos markovianos de filas. Finalmente, realizamos uma comparação entre o desempenho do escalonamento de processos no sistema operacional Linux e o alcançado pela rede de escalonamento.

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

A STUDY OF POTENTIAL OF INTRACHIP NETWORKS FOR SCHEDULING PROCESS IN MULTICORE ARCHITECTURES

Alexandre Borges Gonçalves

September/2010

Advisors: Felipe Maia Galvão França
Valmir Carneiro Barbosa

Department: Systems Engineering and Computer Science

In this work we proposed a mechanism for process scheduling in a chip network. This scheme was simulated by means of a systems design language and properties like load balance and scalability were analyzed. Additionally, we contrasted results obtained by simulation with those expected from markovian models of queues. Finally, we made a comparison between process scheduling performance on Linux operating system and that which was achieved by the scheduling network.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Listagens

Capítulo 1

Introdução

Em 1965, Gordon Moore, co-fundador da Intel, propôs em um artigo publicado na *Eletronics Magazine* que a complexidade do número de componentes por circuito integrado cresceria por um fator de dois ao ano [1]. Essa previsão, denominada "Lei de Moore" por Carver Mead, professor da Cal Tech, foi mais tarde revista por Moore, que ajustou o prazo requerido para o aumento exponencial no número de transistores por chip para dois anos [2, 3]. Moore, entretanto, nunca afirmou que o número de componentes dobraria a cada 18 meses, como uma versão imprecisa de sua lei declarava [2–4].

Por quatro décadas, a lei de Moore permitiu que processadores menores, mais confiáveis, mais baratos e com uma frequência de operação maior que a de seus antecessores fossem construídos [5, 6]. Durante esse período, a estratégia da indústria consistiu em produzir processadores de núcleo único com um desempenho melhor que o da geração anterior. A evolução do processador se traduzia, basicamente, em um melhor desempenho na execução de aplicações existentes e no suporte a novas aplicações [7]. Em particular, o desempenho de fluxo um execução (*thread*) individual melhorava significativamente em função do aumento de funcionalidade e do incremento na frequência de operação nessa nova geração de processadores.

Entretanto, devido a algumas limitações físicas, em especial restrições na capacidade de remoção de calor, a tendência de aumento tanto da velocidade quanto da densidade do componentes eletrônicos parece ter chegado a um fim [8].

A fim de manter seu modelo de negócio, a indústria voltou-se a fabricação de processadores de vários núcleos, cujo objetivo é sustentar um aumento de desempenho similar ao obtido com a abordagem anterior. Patterson, sugere uma "Nova Lei de Moore", que prevê: O dobro de núcleos, com frequência de operação aproximada, por *chip*, a cada geração de tecnologia [9].

Nessa nova estratégia, a melhora de desempenho é obtida pela execução paralela de aplicações [7]. Em particular, uma aplicação individual só apresenta melhora de desempenho entre duas gerações de processadores se esta pode ser decomposta em

dois ou mais fluxos que executam em núcleos distintos de forma paralela.

O novo compromisso assumido pela indústria indica que o número de núcleos nos processadores comerciais deve aumentar consideravelmente nos próximos anos. Algumas previsões sugerem 64 núcleos por volta de 2015 [9]. De fato, em 2009 a Intel a revelou um *chip* experimental com 48 núcleos, apenas dois anos após apresentar um protótipo, ainda que não plenamente funcional, com 80 núcleos [10].

Esse aumento contínuo no número de núcleos imporá uma demanda crescente por processamento paralelo, que se refletirá, principalmente, na ampliação do número de processos em execução no sistema fazendo com que o escalonamento de processos e a troca de contexto tornem-se pontos ainda mais sensíveis ao desempenho do sistema. A conclusão é que deveremos assistir uma maior pressão pela eficiência no escalonamento de processos nos próximos anos.

Neste trabalho, propomos um mecanismo de escalonamento de processos baseado em *hardware* com intuito de adicionar paralelismo ao escalonamento de processos, realizar balanceamento de carga e liberar o processador para tarefas mais relevantes.

1.1 Objetivos e Contribuições

Nosso objetivo principal é apresentar um mecanismo de escalonamento de processos simples e eficiente, a que denominamos rede de escalonamento *intrachip*.

Como será demonstrado, esse modelo induz de forma intrínseca ao balanceamento de carga entre processadores. Além disso, ele é escalável, aumentando o número médio de processos no sistema em uma proporção menor que um incremento realizado no número de processadores para atender uma demanda maior de processos.

Mostraremos ainda, que o desempenho conseguido por esse mecanismos é muito próximo ao esperado de um modelo M/M/m, com a vantagem de possuir diversos pontos de acesso. Essa característica deverá se tornar cada vez mais importante para sistemas *multicore*, à medida que o número de núcleos de processamento aumenta.

Adicionalmente, através de uma comparação com um sistema operacional real, mostraremos que o escalonamento via *hardware* proposto, que pode ser realizado em paralelo e é menos sujeito a contenção, apresenta um potencial de melhoria que não pode ser explorado pelo escalonamento via *software*.

1.2 Estrutura do Documento

Este dissertação está dividida em seis capítulos. No capítulo 2 abordaremos trabalhos correlacionados e seus objetivos. No capítulo 3 apresentaremos conceitos

relacionados ao escalonamento de processos de forma geral, bem como particularidades do escalonamento *multicore* e discutiremos a evolução do escalonamento de processos no Linux. No capítulo 4, descreveremos a motivação para o mecanismos de escalonamento *intrachip*, sua organização, funcionamento e alguns detalhes de implementação. No capítulo 5, demonstraremos por meio de gráficos o comportamento da rede de escalonamento e as implicações de seu uso. Adicionalmente, vamos contrastar os resultados obtidos com aqueles esperados de modelos markovianos de filas e faremos uma breve comparação com o desempenho alcançado pelo sistema operacional Linux. Finalmente, no capítulo 6, apresentaremos nossas conclusões e propostas para trabalhos futuros.

Capítulo 2

Trabalhos Correlacionados

Neste capítulo apresentaremos alguns trabalhos relacionados ao objeto de estudo dessa dissertação, isto é, o escalonamento em uma rede *intrachip*.

2.1 Redes-em-*Chip*

Network-on-Chip, ou simplesmente *NOC*, é uma tecnologia emergente que permite conectar componentes individuais em um *chip* [11]. A ideia por trás das Redes-em-*Chip* é aplicar a teoria de redes na comunicação desses elementos. Uma utilização óbvia para redes-em-*chip* é a conexão de diversos núcleos de processamentos em um processador, em substituição aos barramentos convencionais que não escalam com o incremento no número de núcleos [11, 12].

Como descreveremos em detalhes no capítulo 4, nossa proposta para escalonamento baseado em *hardware* requer um conjunto de *chips* de escalonamento, interligados segundo uma topologia arbitrária, formando, o que denominamos, uma rede de escalonamento. Entenda-se *chip* de escalonamento como uma unidade que é independente em aspectos lógicos, mas não necessariamente em termos físicos.

Os *chips* de escalonamento e as unidades de processamentos descritas neste trabalho, ainda que de forma muito incipiente, compõem uma rede-em-*chip* e, embora métodos, protocolos e topologias de redes não seja o assunto principal desta dissertação eles ainda permanecem subjacentes ao tema central. Dessa forma, iremos abordar nessa seção alguns trabalhos relacionados a este tópico.

2.1.1 David Atienza

David Atienza et. al [12] descrevem os benefícios de redes-em-*chip* no estado da arte usando um completo fluxo para síntese de rede-em-*chip*, e uma detalhada análise de escalabilidade de diferentes implementações de redes-em-*chip* para nós com tecnologia na escala de nanômetros. Este trabalho, propõe blocos básicos para construção

de arquiteturas de redes-em-*chip*, avalia questões relativas a possíveis topologias, aborda o projeto de redes-em-*chip* livres de *deadlock*, discute consumo de energia e, apresenta experimentos e casos de estudo.

2.1.2 Paul Gratz

Paul Gratz et. al [13] discutem os compromissos assumidos no projeto de uma rede-em-*chip mesh*, 4x10, bidimensional, com quatro canais virtuais utilizando roteamento *wormhole*. Examinam em particular, porque a área e a complexidade comprometem a latência. A rede é avaliada através de cargas sintéticas e realísticas com a utilização de *benchmarks*. Adicionalmente, investiga-se o efeito da largura de banda do *link* e tamanho *buffer* do roteador no desempenho global.

2.1.3 Erno Salminen

Erno Salminen et. al [14] discorrem sobre o estado da arte no campo de avaliação e comparação de redes-em-*chip*. Seu estudo propõe-se a identificar as principais abordagens, como as redes são atualmente avaliadas, e mostrar que aspectos foram cobertos e quais necessitam maior esforço de pesquisa. Finalmente, ele realiza comparações de resultados de várias origens distintas por meio de um conjunto de diretrizes bem definidas.

2.1.4 Edson Ifarraguirre

Edson Ifarraguirre [15] aborda a modelagem, descrição e validação de redes *intrachip* no nível de transação. Este trabalho propõe uma organização para níveis superiores de abstração de projeto a ser empregada durante a modelagem de sistemas digitais complexos. Partindo desta proposta, resultados iniciais da comparação entre modelagens sistêmicas e a modelagem *RTL (Register Transfer Level)* tradicional são apresentados, a partir de um estudo de caso implementado em SystemC e VHDL. O estudo indica a separação de computação e comunicação como forma de lidar com a complexidade. Finalmente, ele propõe a modelagem abstrata de uma rede de comunicação *intrachip* parametrizável denominada Hermes, usando o nível de abstração de transação e empregando a linguagem SystemC.

2.2 Escalonamento Baseado em *Hardware*

Em anos recentes, o escalonamento via *hardware* tem sido objeto de estudo de diversos autores, ainda que com enfoques ligeiramente diferentes. Nessa seção vamos abordar alguns deles.

2.2.1 André Nácul

André Nácul et. al [16] propuseram um sistema operacional em tempo real baseado em *hardware* que implementa a camada de sistema operacional em uma arquitetura de multiprocessamento simétrico com dois núcleos de processamento. Nesse sistema, a comunicação entre tarefas é realizada por meio de uma *API (Application Program Interface)* dedicada e o sistema se encarrega das requisições de comunicação das aplicações e também implementa o algoritmo de escalonamento de tarefas.

Os autores sugerem que essa implementação é capaz de explorar a característica de migração de tarefas suportada por uma arquitetura *SMP* muito mais eficientemente que um sistema operacional em tempo real baseado em *software*, chegando a superar o desempenho atingível com um particionamento de tarefas estático ótimo. Eles também propõem um mecanismo de bloqueio *(lock)* por *hardware* a fim de implementar comunicação em memória compartilhada.

A principal diferença dessa abordagem em relação ao escalonamento *intrachip*, proposto nessa dissertação, é que a primeira move todo sistema operacional para o *hardware*, enquanto a segunda transfere apenas o escalonador de processos, mantendo as demais tarefas sob responsabilidade de um sistema operacional baseado em *software*. Além disso, a implementação em questão limitou-se a apenas dois núcleos de processamento, enquanto o modelo descrito pela rede de escalonamento *intrachip*, por ser simulado, é mais genérico e aceita um número variável de núcleos.

2.2.2 Pramote Kuacharoen

Pramote Kuacharoen et. al [17] descrevem uma arquitetura de escalonador baseado em *hardware* que minimiza o tempo de processador gasto pelo escalonador e processamento de *ticks* de tempo. O escalonador em *hardware* provê três disciplinas de escalonamento: baseada em prioridade, índice monotônico e tempo de morte mais cedo.

A ideia principal é reduzir o custo adicional *(overhead)* migrando serviços do *kernel* como escalonamento, processamento de *ticks* de tempo e manipulação de interrupção para o hardware.

Foram implementados um escalonador baseado em *hardware* através de *VHDL* e um sistema operacional em tempo real escrito em C. Os elementos descritos anteriormente foram, então, movidos para o escalonador em *hardware*, eliminando assim o custo adicional desses serviços.

Nessa implementação, para uma carga de trabalho de 32 processos e um *tick* de tempo de 1 ms, apenas 0.21% do tempo de processador era desperdiçado. Entretanto, se a resolução do *tick* mudava para 10 μs, 21.16% do tempo de processador era comprometido.

Capítulo 3

Escalonamento *Multicore*

Em um sistema computacional, programas que executam atividades de usuários e tarefas de sistema são, comumente, mapeados em entidades denominadas processos. Em geral, há mais processos ativos na memória do sistema do que processadores disponíveis para sua execução. A atividade de decidir como e quando os processos serão atribuídos a um processador para execução é denominada escalonamento e será abordada neste capítulo.

3.1 Processos e fluxos de execução

De um ponto de vista informal, processo é um programa em execução [18]. Entretanto, processos são de natureza dinâmica e possuem um estado operacional associado a sua execução. Além disso, processos podem estar relacionados a recursos do sistema, como arquivos abertos e conexões de redes. Um programa, por sua vez, é apenas uma entidade passiva que reside em uma unidade de armazenamento secundária [18, 19].

O estado operacional de um processo é caracterizado pela condição de suas estruturas internas em um dado instante e, é comumente conhecido como contexto. O contexto de um processo inclui seu espaço de endereçamento, o conteúdo da pilha, o conteúdo dos registradores, o ponteiro de instrução e assim por diante [19]. O contexto do processo é privado, significando que um processo não pode alterar elementos no contexto de outro, a menos que isto seja permitido explicitamente.

Adicionalmente, processos possuem um estado que descreve sua situação sob a perspectiva do sistema. Esse estado é relativo à condição de execução do processo e restringe de maneira implícita os eventos e operações aos quais o processo está sujeito em sua ocorrência. Os estados externos que um processo pode assumir são: novo (*new*) – durante sua criação, pronto (*ready*) – enquanto aguarda por execução, em execução (*running*) – ao ser atribuído a um processador, em espera (*waiting*) – se aguarda por uma operação de entrada e saída ou evento e, terminado (*terminated*)

– ao concluir sua execução [18].

A relação entre programas e processos não é, necessariamente, unívoca. Em alguns casos, um programa pode originar a diversos processos [18]. Por sua vez, processos são compostos de fluxos ou linhas de execução denominados threads. Podemos definir um thread como um fluxo de controle independente (sequência de instruções) dentro de um processo [20]. Um processo pode ser composto de um (*single threaded*) ou vários fluxos de execução (*multi threaded*).

Em contraste com processos em um sistema, fluxos de execução em um mesmo processo compartilham seu contexto. Isto significa que memória e outros recursos são comuns a todos os fluxos de execução pertencentes ao processo [18]. Dessa forma, um fluxo de execução pode, por exemplo, alterar diretamente o valor de uma variável que é utilizada por outro fluxo de execução.

Para propósitos desse texto não faremos distinção entre processos e fluxos de execução. De fato, alguns sistemas operacionais implementam fluxos de execução como processos separados que compartilham espaço de endereçamento, páginas de memória física, arquivos abertos, e assim por diante.

Essencialmente, existem dois tipos principais de operações que um processo pode realizar - computação e entrada e saída. Computação resume-se basicamente a realização de cálculos no processador, enquanto operações de entrada e saída possibilitam a interação com dispositivos internos e externos ao sistema. Leituras e escritas em um disco magnético, entradas recebidas por teclado e o envio de informações a um monitor são exemplos de operações de entrada e saída. Processos que fazem uso intensivo de computação são conhecidos como *CPU-Bound*, enquanto processos que fazem uso predominante de operações de entrada e saída são denominados *I/O-Bound* [18].

3.2 Escalonamento e Compartilhamento de Tempo

Dispositivos de entrada e saída são comparativamente muito mais lentos que o processador. Em um computador pessoal, por exemplo, o processador é aproximadamente um milhão dez vezes mais rápido que o disco magnético. Assim, sem nenhuma medida adicional, na ocorrência de uma operação de entrada e saída, o processador seria forçado a permanecer ocioso, aguardando o término da operação a fim de prosseguir com a execução do processo.

Para contornar o problema mencionado, sistemas computacionais, em geral, implementam um mecanismo conhecido como multiprogramação. Com a multiprogramação sempre que o processador precisa aguardar o termino de uma operação ou a

liberação de um recurso para um processo, outro processo, se possível, é selecionado para execução. O número de processos ativos na memória é conhecido como grau de multiprogramação [18, 21].

Como uma aproximação, supondo que um processo na memória tenha probabilidade p de realizar uma operação de entrada e saída, e que essa probabilidade seja uniforme, com n processos na memória a utilização do processador seria dada por $U = 1 - p^n$. Segue-se que a utilização do processador aumenta à medida que o grau de multiprogramação do sistema aumenta. Consequentemente, o tempo total de processamento diminui. Deve-se observar, contudo, que o grau de multiprogramação do sistema é limitado pela quantidade de memória disponível.

Sistemas mais atuais vão além da multiprogramação implementando o compartilhamento de tempo (*time sharing*) ou multitarefa (*multitasking*) [18, 21]. O objetivo desse mecanismo é transmitir a impressão de que vários processos são executados simultaneamente. Nele, o processador é alocado a cada processo por um curto período de tempo conhecido como *quantum* ou *time slice*. Quando o processo exaure a fatia de tempo que lhe cabe, outro é selecionado pra execução e assim por diante.

Em um ambiente multitarefa, sempre que o sistema necessita remover um processo do controle do processador o estado operacional do processo deve ser armazenado para que este futuramente possa retomar sua execução do ponto onde foi interrompido. O ato de remover um processo do controle do processador preservando seu estado operacional é conhecido como troca de contexto (*Context Switch*) ou troca de processo (*Process Switch*) [19].

A distinção entre ambos reside no fato de que na troca de Processo o processo em execução é completamente removido e substituído por outro, ao passo que na troca de Contexto o estado do processo é armazenado enquanto sua execução é suspensa temporariamente para a realização de alguma tarefa do sistema, como o tratamento de uma interrupção de hardware ou atendimento a uma chamada de sistema. Observe que uma troca de processo realiza operações adicionais além das requeridas por uma troca de contexto [19].

Se o sistema permite que um processo no controle do processador execute até que este bloqueie ou ceda voluntariamente o processador, ele é dito cooperativo. Em um sistema cooperativo um processo permanece em execução até que termine, realize uma operação de entrada e saída ou libere o processador espontaneamente. Se uma troca de processo pode acontecer sob outras circunstâncias, que não dependem do processo em execução, como na ocorrência de uma interrupção de hardware, o sistema é denominado preemptivo. Um sistema preemptivo é mais confiável no sentido de que um processo não pode monopolizar um processador indefinidamente [18].

Sempre que o processador está ocioso ou uma troca de processos ocorre, um

novo processo deve ser selecionado para execução. O responsável por esta seleção é o escalonador da UCP (*CPU Scheduler*). O escalonador é um componente do sistema operacional cuja tarefa é selecionar um processo ativo na memória, alocá-lo ao processador e iniciar sua execução. O critério empregado na seleção do processo a ser executado é conhecido como política de escalonamento (*Scheduling Policy*).

3.3 Políticas de Escalonamento

Como mencionamos, sempre que um processo precisa ser substituído, a política de escalonamento é utilizada. Em outras palavras, é a política de escalonamento que decide quando e como um processo será substituído. O escalonador é a agente do sistema responsável por aplicar a política.

Usualmente, um conjunto de medidas é utilizado para avaliar uma política de escalonamento. O percentual médio de tempo que o processador é utilizado (*CPU utilization*). O número médio de processos terminados por unidade de tempo (*throughput*). O intervalo médio de tempo entre o início e o término da execução um processo (*turnaround*). O tempo médio que um processo aguarda execução na fila de prontos (*wait time*). O tempo médio que um processo demora a responder uma requisição (*response time*) e assim por diante [18, 22].

Os objetivos na escolha de uma política de escalonamento, geralmente, são maximizar a utilização do processador e a produtividade e, minimizar o tempo de processamento, o tempo de espera e o tempo de resposta. Além disso, uma política de escalonamento tenta aplicar equidade, isto é, tenta garantir que cada processo receba uma fatia justa de tempo do processador e que a execução de nenhum processo seja postergada indefinidamente (*starvation*). Nesta seção abordaremos algumas das principais políticas de escalonamento.

3.3.1 Primeiro a Chegar, Primeiro a ser Servido

A política de escalonamento mais elementar é a PCPS (FCFS) - Primeiro a Chegar, Primeiro a ser Servido (*First Come, First Served*).

Ela comumente é implementada como uma fila onde os processos são organizados na ordem de chegada. Processos que chegam ingressam no final da fila. Sempre que o processador fica ocioso, o primeiro processo na fila, se houver, é selecionado para execução e removido da fila. Uma vez selecionado, o processo ganha controle do processador e o mantém até que sua execução termine, realize uma operação de entrada e saída ou abdique explicitamente desse controle. Os demais processos permanecem na fila aguardando até que o processador esteja disponível novamente [18, 22].

A política PCPS não é preemptiva, pois o processo, uma vez selecionado, permanece em execução até que termine ou libere o processador . Como a política PCPS não aplica nenhum tipo de priorização, nenhum processo é abandonado, isto é, cada processo é eventualmente executado. Em outras palavras, não ocorre *starvation* quando essa política é aplicada.

Entretanto, pelo fato de que um processo pode reter o processador por muito tempo, o tempo de processamento, o tempo de espera e o tempo de resposta podem ser altos. A política PCPS também é conhecida como PEPS (*FIFO*) - Primeiro a Entrar, Primeiro a Sair (*First-in First-out*).

3.3.2 Menor Primeiro

Na política PCPS, um processo executa até que sua fase de uso do processador termine, seja pela conclusão do programa, pelo uso de uma operação de entrada e saída ou pela renuncia espontânea do processo ao processador.

Uma abordagem diferente seria escolher o processo com menor previsão de duração da próxima fase de uso do processador. Essa política é conhecida como Menor Primeiro (*Shortest Job First*) [18, 22].

A política Menor Primeiro é ótima em termos de retardo. Ela diminui significativamente o tempo médio de espera e consequentemente o tempo de processamento dos processos. Entretanto, o tempo de resposta do sistema pode não ser apropriado para uso em sistemas interativos.

O problema da política MP é prever o tempo de duração da próxima fase de uso de processador dos processos. Muitas vezes uma previsão baseada no histórico passado do processo é utilizado como uma estimativa. Alguns autores [18, 22] sugerem $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$, onde τ_n e t_n representam respectivamente a previsão e a duração da última fase de processador, τ_{n+1} é a previsão de duração da próxima fase. O papel da constante α é ajustar a contribuição do passado mais e menos recente na estimativa.

A política MP pode ser preemptiva, se interrompe o processo em execução caso surja um processo com menor previsão de duração da próxima fase de uso do processador na fila de prontos.

3.3.3 Alocação Por Prioridade

A política de alocação por prioridade associa uma prioridade, normalmente um valor numérico, a cada processo. Quando o processador fica ocioso, o processo de maior prioridade é escolhido para execução [18, 22].

A política MP é um tipo de política de alocação por prioridade em que a prioridade do processo está associada à duração de sua próxima fase de uso do processador.

Quanto maior esta duração, menor a prioridade do processo.

A prioridade de um processo pode ser definida estaticamente, isto é, fixada no início de sua execução, ou ser dinâmica, sendo ajustada conforme seu comportamento no decorrer da dela.

Diversos critérios podem ser usados para estabelecer a prioridade do processo. Tais critérios estão normalmente relacionados com as características do sistema computacional. Alguns exemplos são o uso de recursos do sistema, quantidade de entrada e saída versus computação, a interatividade e assim por diante.

A política de alocação por prioridade pode ser preemptiva ou não, dependendo da ação tomada quando um processo mais prioritário ingressa na fila de prontos.

Políticas de alocação por prioridade sofrem de abandono de processos, pois processos mais prioritários podem fazer com que outros menos prioritários não sejam executados. A técnica conhecida como envelhecimento (*aging*) tenta endereçar esse problema. A ideia por trás do envelhecimento é aumentar gradualmente a prioridade dos processos que não foram executados à medida que o tempo passa. Dessa forma, todo processo é eventualmente executado.

3.3.4 Escalonamento Garantido

Uma alternativa seria fazer com que a política de escalonamento assuma compromissos realistas sobre a execução dos processos e tente cumpri-los. Por exemplo, supondo que há n processos ativos na memória, uma abordagem seria garantir que cada processo receba aproximadamente $1/n$ ciclos do processador. Esse tipo de política é conhecido como escalonamento garantido (*guaranteed scheduling*) [22].

Numa troca de processo, o escalonamento garantido calcula a razão entre o tempo real de processador e o tempo atribuído ao processo. Este último é o tempo desde sua criação dividido por n. Um valor menor que 1 indica que o tempo de execução foi inferior ao que deveria. Um valor maior que 1 denota excesso no tempo de execução. O processo com menor razão é, então escolhido para execução.

Dessa forma o escalonamento garantido tenta equilibrar os tempos de execução dos processos ativos, portanto não ocorre abandono.

O escalonamento garantido pode ser preemptivo, se a razão entre os tempos dos processo for calculada periodicamente e o processo em execução puder ser substituído por outro que apresente uma razão menor.

3.3.5 Escalonamento por Sorteio

No escalonamento por sorteio (*lottery scheduling*) os diversos processos do sistema recebem bilhetes de loteria que representam os recursos do sistema, como tempo de processador, por exemplo. Sempre que uma decisão de atribuição de recurso precisa

ocorrer, um bilhete de loteria é sorteado aleatoriamente, o processo que detém o bilhete recebe o recurso [22].

Essa política permite que processos sejam tratados de maneiras distintas, por fornecer a alguns um número maior de bilhetes relacionados a um determinado recurso. Além disso, ela é muito flexível pois uma mudança na distribuição dos bilhetes é refletida automaticamente na próxima decisão de alocação. Em virtude disso, ela pode ser usada para implementar complexos esquemas de alocação de recursos.

Em um contexto cooperativo, por exemplo, um processo poderia ceder seus bilhetes a outro, a fim de que este aumente suas chance de ganhar acesso a um recurso necessário a realização da tarefa. Esses bilhetes seriam devolvidos à *posteriori*.

Em se tratando de escalonamento do processador, se o sistema realiza n sorteios por segundo, então cada bilhete daria direito a aproximadamente $1/n$ segundos de processador. Dessa forma, aplicada ao processador a política é preemptiva.

O escalonamento por sorteio apresenta uma equidade em relação ao processo que é proporcional ao número de bilhetes que este possui. Em virtude disso, um processo com um número de bilhetes maior que zero será eventualmente executado, em outras palavras, não ocorre abandono de processos.

Uma das maiores dificuldades no escalonamento por sorteio é determinar como distribuir corretamente aos processos os bilhetes relacionados aos diversos recursos do sistema.

3.3.6 Alocação Circular

A política de alocação circular (*round robin*) organiza os processos em uma fila de forma semelhante à política PCPS, mas adiciona preempção.

Nesse esquema de alocação, uma pequeno intervalo de tempo denominado *quantum* é atribuído ao processo. Uma vez em execução, o processo pode terminar, realizar uma operação de entrada e saída, liberar espontaneamente o processador ou ter seu *quantum* expirado e ser interrompido com o auxílio de um temporizador do sistema. Na primeira situação o processo é removido do sistema, nas demais ele é inserido no final da fila de execução. Em todos os casos, ocorre uma troca de contexto e outro processo, se houver, é selecionado para execução [18, 22].

O tamanho do *quantum* varia com o sistema operacional, mas valores comuns estão entre dezenas e centenas de milissegundos. Se o *quantum* é muito grande a política de alocação circular tende a comportar-se como a política PCPS. Quando o *quantum* é muito pequeno a política de alocação circular é denominada compartilhamento de processador.

Um tamanho de *quantum* menor favorece a interatividade do sistema pois pro-

gramas que interagem com usuários estarão sendo executados mais frequentemente. Entretanto, se o *quantum* for muito pequeno o desempenho do sistema pode degradar em virtude de trocas de contextos excessivas. O ideal, portanto, é que *quantum* seja bem maior que o tempo gasto em uma troca de contexto, mas ainda assim pequeno o suficiente.

O desempenho da política de alocação circular está intimamente ligada ao tamanho do *quantum*. Embora um *quantum* menor favoreça o tempo de resposta, ele implica em um numero maior de trocas de contexto aumentando o tempo médio de espera e o tempo médio de processamento. Por outro lado, aumentar o tamanho do *quantum* não diminui o tempo de processamento de forma proporcional, pois parte significativa dos processos podem já estar esgotando sua fase de processador com um *quantum* menor.

3.3.7 Alocação em Tempo Real

Sistemas de tempo real impõem restrições relacionadas ao tempo de execução de tarefas críticas que o sistema deve suportar [18, 22].

Essencialmente, há dois tipos de sistemas de tempo real: sistemas de tempo real com restrições rígidas (*hard real time*) e sistemas de tempo real flexíveis (*soft real time*).

Em um sistema de tempo real com restrições rígidas uma tarefa (ou processo) crítica do sistema deve ser executada dentro de um limite acordado de tempo. Normalmente, a não execução de uma tarefa crítica no prazo definido pode ser caótica e desencadear consequências indesejáveis. Exemplos de sistemas desse tipo seriam o sistema de controle de um reator nuclear e o piloto automático de uma aeronave. Em virtude de suas características, sistemas de tempo real com restrições rígidas são de propósito específico, e sua implementação não é indicada em sistemas de propósito geral, como computadores pessoais, onde o uso de memória virtual, armazenamento secundário e outros mecanismos tornam impossível fornecer garantias de tempo.

Sistemas de tempo real flexíveis, por sua vez, são menos rigorosos e, em geral, funcionam impondo a priorização de processos críticos sobre os demais processos do sistema. Nesses sistemas, embora processos críticos tenham precedência sobre outros, não há garantias estritas sobre sua execução e um atraso eventual é tolerável. Processamento de imagem e som são comumente suportados por uma política de tempo real flexível implementada em um sistema de propósito geral.

Algoritmos de escalonamento em tempo real podem ser estáticos ou dinâmicos. Algoritmos estáticos tomam suas decisões de escalonamento antes do sistema começar a executar. Algoritmos dinâmicos decidem em tempo de execução. Um algoritmo de escalonamento em tempo real dinâmico clássico é o algoritmo de índice

monotônico (*rate monotonic scheduling*) proposto por Liu e Layland em 1973 [22].

3.3.8 Alocação com Várias Filas

Algumas estratégias de escalonamento organizam os processos que aguardam execução no sistema em diversas filas. Usualmente, cada fila agrupa processos com prioridades semelhantes [18, 22].

Esta organização permite que sejam empregadas políticas de alocação diferentes em cada fila. Por exemplo, uma fila mais prioritária poderia usar PCPS enquanto uma fila menos prioritária adotaria a alocação circular.

Processos em uma fila mais prioritárias, comumente, exercem precedência sobre processos em uma fila menos prioritária. Como há uma priorização, pode haver abandono de processos. Esse problema é, normalmente, tratado com uso de envelhecimento e a migração de processos de filas menos prioritárias para filas mais prioritárias.

Várias questões precisam ser endereçadas com esse tipo de abordagem. A quantidade de tempo de processador destinada a cada fila. A classificação de um processo no início e no decorrer de sua execução. A estratégia de preempção entre as filas. O mecanismo de envelhecimento e migração. O tamanho do *quantum* em cada fila. E assim por diante.

3.4 Arquiteturas *Multicores* & Escalonamento

Durante os 40 anos em que a Lei de Moore vigorou sem alterações [7, 9], o aumento exponencial na capacidade de integração [1] ditou o comportamento da indústria. Processadores de núcleo único com uma quantidade cada vez maior de componentes integrados e uma frequência de operação crescente sucederam-se no mercado [7, 23]. Essa taxa de crescimento proporcionou um avanço contínuo no poder computacional dos processadores, mas ao mesmo tempo aproximou rapidamente a tecnologia dos limites impostos pela Física [8].

Em face dessa dificuldade, os fabricantes de processadores voltaram-se para outras maneiras de melhorar o desempenho a fim de manter o progresso que alimentou a indústria.

Em 2005, a Intel lançou seu primeiro processador para *desktops* com dois núcleos completos operando à mesma velocidade, o Pentium D [24]. Por essa época a indústria havia alterado seu roteiro (*roadmap*) [9] e processadores de vários núcleos tornaram-se a corrente principal (*main stream*) na estratégia dos fabricantes [25].

Há quase um consenso de que arquiteturas *multicore* tornar-se-ão cada vez mais importantes no futuro próximo e que o número de núcleos no processadores comer-

ciais deverá aumentar com o passar do tempo [9, 23].

Processadores *multicores* são compostos de vários núcleos que buscam suas próprias instruções e operam sobre seu próprios dados, seguindo a classificação MIMD (*multiple instruction, multiple data*) da taxonomia de Flynn [23]. Em outras palavras, cada núcleo funciona como um processador isolado. A melhora de desempenho é, então, conseguida com cada núcleo executando um fluxo de execução diferente [7]. A partir de agora usaremos os termos núcleo e processador de forma intercambiável, assim como os termos *multicore* e multiprocessador.

Multiprocessadores são categorizados de acordo com sua organização de memória. Em arquiteturas UMA (*Uniform Memory Access*) existe apenas um memória centralizada compartilhada por meio de um barramento que pode ser acessada ao mesmo custo por todos os núcleos. Essa é a organização predominante nos multiprocessadores atuais e seu principal inconveniente é a contenção pelo acesso ao barramento. Em virtude disso, arquiteturas UMA são mais apropriadas ao uso com um número não muito grande de núcleos [23].

Em arquiteturas NUMA (*Non-Uniform Memory Access*), o espaço de endereçamento é compartilhado, mas a memória e os núcleos são distribuídos em nós. Isto significa que o tempo de acesso a uma palavra de memória pode ser diferente, dependendo do nó onde ela se encontra. Além disso, nós podem ser agrupados criando uma hierarquia com vários níveis de acesso à memória. Como a memória é distribuída, aumentando a largura de banda, esse tipo de organização é mais indicada para uso com um número maior de núcleos. Esse tipo de arquitetura é também conhecida como DSM (*Distributed Shared Memory*) [23, 26, 27].

Adicionalmente, os multiprocessadores são classificados em função das atividades que os processadores exercem em relação aos demais. Se todos os processadores no sistema realizam as mesmas tarefas, diz-se que o sistema realiza multiprocessamento simétrico (*symmetric multiprocessing*) [27, 28]. Se, entretanto, tarefas diferentes são atribuídas aos processadores, o multiprocessador realiza multiprocessamento assimétrico (*asymmetric multiprocessing*) [28].

O multiprocessamento simétrico é geralmente obtido através da execução de uma única instância de sistema operacional em todos os processadores [18]. Nesse caso, as estruturas de dados do sistema são, usualmente, compartilhadas e mecanismos de exclusão mútua são utilizados no controle de acesso a fim de garantir integridade dessas estruturas.

No multiprocessamento assimétrico alguns processadores desempenham tarefas específicas do sistemas enquanto outros podem ser de uso geral. Por exemplo, um processador pode controlar todos os demais, criando um relacionamento mestre-escravos [18]. Uma alternativa seria distribuir tarefas importantes do sistema entre os processadores. Um processador poderia ser responsável pelo escalonamento, outro

pelas operações de entrada e saída e assim por diante.

Por suas características, sistemas de multiprocessamento simétrico são, normalmente, implementados em arquiteturas UMA, sistemas de multiprocessamento assimétrico, por suas vez, são mais comuns em arquiteturas NUMA. Essa relação, entretanto, não é estrita nem obrigatória.

3.4.1 Escalonamento em Multiprocessadores

Em um sistema com vários processadores o escalonamento de processos torna-se ainda mais complexo. Vamos assumir por simplicidade que todos os processadores que executam processos do sistema são idênticos.

Uma alternativa de escalonamento seria manter uma fila de processos prontos para cada processador. Dessa forma, sempre que o processador estivesse ocioso ele escolheria um processo da fila para execução, segundo uma política como as descritas anteriormente. Esse método, no entanto, tem um inconveniente: um processador pode ficar ocioso enquanto há processos aguardando execução na fila de outro processador [18]. Este problema é conhecido como desbalanceamento de carga (*load imbalance*).

Desbalanceamento de carga é um situação onde a carga de trabalho do sistema não está, o tanto quanto possível, igualmente distribuída entre os processadores [29]. Deve-se notar contudo, que o desbalanceamento não ocorre simplesmente porque a carga de trabalho dos processadores difere. Por exemplo, supondo um sistema com quatro processadores e cinco processos ativos na memória, algum processador deverá necessariamente executar dois processos. No caso anterior, apesar da carga de um dos processadores ser diferente não ocorre desbalanceamento, pois não há uma forma melhor de alocar os processos. Em outras palavras, só há desbalanceamento de carga se existe uma divisão de trabalho mais justa entre os processadores.

De uma maneira mais formal, seja um multiprocessador com n processadores ou núcleos, $N(p_i)$ o número de processos ou fluxos de execução no processador p_i, $0 \leq i < n$. Se $max(N(p_i))$ e $min(N(p_i))$ representam respectivamente o maior e o menor número de processos em um processador. Então ocorre desbalanceamento se, $(max(N(p_i)) - min(N(p_i))) > 1$ [29].

O principal impacto do desbalanceamento de carga é que ele aumenta o tempo total de processamento e diminui a taxa de utilização dos processadores, degradando, portanto, o desempenho [29].

Um meio de lidar com o desbalanceamento é realizar a migração de processos ou fluxos de execução de um processador para outro a fim de equilibrar suas cargas de trabalho [26, 27]. Entretanto, em um sistema com arquitetura NUMA a transferência de processos entre quaisquer processadores pode não ser desejada, uma vez que a

hierarquia de acesso a memória impõe custos distintos para a migração de processos [26, 27].

Outra abordagem para realizar o escalonamento de processos em um multiprocessador é manter todos os processos uma única fila compartilhada por todos os processadores [18]. Dessa forma um processo pode ser executado em qualquer processador disponível. Pode inclusive realizar etapas distintas de sua execução em processadores diferentes.

Esse método tem a favor o fato de não sofrer de desbalanceamento, já que com apenas uma fila o sistema está implicitamente balanceado [26]. A principal desvantagem, é que apenas um processador pode acessar as estruturas de dados compartilhadas do sistema em um dado momento. Se dois processadores desejam acessar simultaneamente a fila de processos prontos para execução, um processador obterá acesso enquanto o outro deverá aguardar até que o primeiro termine suas atividades e que o acesso a fila seja liberado. À medida que o número de processadores aumenta essa contenção pode deteriorar o desempenho do sistema. Essa abordagem não é adequada para sistemas NUMA, pois nessa arquitetura os processos deveriam sempre que possível ser mantidos em seu nó de criação.

3.5 Escalonamento no Linux

Nessa seção vamos abordar a evolução do mecanismo de escalonamento do Linux. Entretanto, como muitas características se mantiveram em comum entre várias versões, não vamos explicar detalhadamente o funcionamento de cada versão, mas sim o que a distinguia das demais. Nosso objetivo aqui é destacar as diferenças marcantes entre elas.

3.5.1 Escalonamento em Versões Iniciais

O escalonador do Linux 1.2 era muito simples. Ele usava uma fila circular para o gerenciamento de processos prontos para execução. Processos eram então selecionados segundo a política de alocação circular (*round robin*) [30].

A versão 2.2 introduziu a ideia de classes de escalonamento, permitindo políticas de escalonamento para processos em tempo real, processos não preemptíveis e processos convencionais. Essa versão também adicionou suporte a multiprocessamento simétrico [30].

3.5.2 Escalonamento na Versão 2.4

Na versão 2.4, o Linux possuía apenas uma fila circular duplamente encadeada de processos compartilhada por todos os processadores. Quando um processador

estava ocioso, ele recebia um processo da fila [26, 31]. Essa abordagem era fácil de implementar, fácil de depurar e resultava em um balanceamento de carga natural [26, 32].

Nessa implementação uma taxa de bondade (*goodness rate*) era atribuída a cada processo. Essa taxa era determinada em função do *quantum* restante, da afinidade com o processador, da prioridade e da classe de escalonamento do processo. Assim, quando um processador ficava disponível o escalonador percorria lista e selecionava o processo com maior taxa de bondade [32].

Entretanto, essa abordagem apesar de simples apresentava problemas, pois à medida que o número de processadores aumentava a fila de processos tornava-se um gargalo [26].

Como a fila era usada por todos os processadores e tinha que ser reexaminada a cada decisão de escalonamento, à proporção que o sistema se tornava mais ocupado, a lista de processos crescia e o tempo de busca aumentava devido a sua complexidade linear [30]. Além disso, os demais processadores desejando obter acesso a fila tinham que esperar mais tempo até que a fila de processos estivesse disponível.

Assim, com o sistema mais ocupado, o escalonador consumia incrementalmente mais ciclos de processador, chegando ao ponto em que mais tempo era gasto escalonando processos do que os executando. Dessa forma, o acesso a fila de processos tornou-se um ponto de contenção tanto em sistemas ocupados quanto em sistemas com quatro ou mais processadores. Como resultado, decidir que processo executar poderia requerer um longo tempo [32].

3.5.3 Escalonamento em Versões Iniciais do *kernel* 2.6

Com o advento do *kernel* 2.6 um novo escalonador que trazia várias melhorias e aprimoramentos em relação ao anterior, foi introduzido.

Como em versões anteriores, os processos eram divididos em diferentes classes de escalonamento: processos em tempo real *FIFO*, processos em tempo real *Round Robin* e processos convencionais de tempo compartilhado.

Processos convencionais possuíam dois tipos de prioridade: a estática e a dinâmica. A prioridade estática consistia em um valor entre 100 e 139, com valores menores indicando maior prioridade, e era utilizada basicamente para determinar o tamanho do *quantum* do processo, conforme a seguinte fórmula [27]:

$$quantum = \begin{cases} (140 - \textit{prioridade estática}) \times 20, & \textit{se prioridade estática} < 120 \\ (140 - \textit{prioridade estática}) \times 5, & \textit{se prioridade estática} \geq 120 \end{cases}$$

A prioridade dinâmica, por sua vez, era utilizada pelo escalonador na seleção de processos e estava relacionada diretamente com a prioridade estática pela formula

[27]:

$$prioridade\ din\hat{a}mica = max(100,\ min(prioridade\ est\acute{a}tica - bonus + 5,\ 139))$$

De forma análoga, a prioridade dinâmica do processo também consistia em um valor no intervalo de 100 a 139. De fato, a prioridade dinâmica era fundamentalmente a prioridade estática ajustada ao comportamento passado do processo. O termo *bonus* representava um valor entre 0 e 10 que dependia diretamente do tempo de permanência médio do processo no estado suspenso (*average sleep time*)[27]. Assim, quanto mais tempo suspenso, maior o incremento na prioridade do processo.

Processos eram preemptíveis, isto é, quando um processo assumia o estado pronto, o *kernel* verificava se sua prioridade dinâmica era maior que a do processo em execução. Em caso positivo, o processo em execução era interrompido e o escalonador era invocado a fim de selecionar outro processo para execução. Além disso, um processo podia ser substituído por outro quando seu *quantum* expirasse [27].

Adicionalmente, processos convencionais também eram classificados em interativos ou em lote. Um processo era dito interativo se a seguinte expressão fosse satisfeita, e classificado como em lote, caso contrário [27]:

$$prioridade\ din\hat{a}mica \leq 3 \times prioridade\ est\acute{a}tica \div 4 + 28$$

Processos em tempo real, estavam associados com uma prioridade de tempo real que consistia em um valor entre 0 e 99 [33]. Nessa classe de processos, a prioridade estática servia apenas para definir o tamanho do *quantum* e, uma alteração nessa prioridade não afetava a prioridade de tempo real. Além disso, a prioridade estática era irrelevante para processos de tempo real *FIFO* já que estes não possuem um *quantum* de tempo associado a sua execução.

A prioridade estática e a prioridade de tempo real podiam ser alteradas respectivamente por meio de chamadas as funções `nice()`, `setpriority()` e `sched_setparam()` [27].

Nesse esquema de escalonamento, em vez de uma fila única compartilhada, cada processador possuía dois conjuntos com 140 filas processos. Cada fila estava associada com uma prioridade admitida pelo sistema. O primeiro conjunto de filas destinava-se ao processos ativos e o outro aos processos expirados [32].

Inicialmente, um processo convencional em condições de executar era posto em uma das filas de processos ativos. À medida que o processo era executado, seu *quantum* era decrementado e, eventualmente se esgotava. Então, o *quantum* era recalculado e o processo era movido para uma das filas de processos expirados. Contudo, em se tratando de processos interativos as coisas eram um pouco mais complicadas. Quando um processo interativo expirava, seu *quantum* era redefinido

e o processo era mantido na fila de processos ativos. Processos interativos eram movidos para uma fila de processos expirados apenas se o processo expirado mais antigo houvesse esperado por tempo demais ou se um processo expirado tivesse prioridade estática maior que o processo interativo. Processos em tempo real, por sua vez, eram sempre considerados ativos [27].

Essa organização proporcionava algumas vantagens. Em primeiro lugar, como existia um *bitmap* indicando que filas não estavam vazias, a busca pelo processo mais prioritário do sistema realizada em tempo constante, isto é *O(1)*. A única atividade requerida era descobrir a fila de processos ativos não vazia mais prioritária e selecionar o primeiro processo da fila [32].

Em segundo lugar, se um processo tivesse sua prioridade alterada, seria possível movê-lo em tempo constante para outra fila que representasse sua nova condição [32].

Além disso, quando o conjunto de processos ativos ficava vazio, uma simples troca de ponteiros permitia que os conjuntos de processos permutassem, fazendo que o conjunto de processos expirados passasse a ser visto como o conjunto de processos ativos e vice-versa [27, 32].

Como mencionamos, uma desvantagem da abordagem com várias filas é que ela está sujeita a desbalanceamento de carga. A fim de lidar com esse inconveniente o *kernel* implementava um mecanismo de balanceamento de carga baseado em domínios de escalonamento.

Domínios de escalonamento permitem ao *kernel* estar ciente da topologia do sistema e realizar o balanceamento de carga de uma maneira eficiente [27]. Essencialmente, um domínio de escalonamento é um conjunto de processadores cujas cargas de trabalho deveriam ser mantidas balanceadas pelo *kernel* [27]. Normalmente, domínios de escalonamento são organizados em hierarquias com domínios superiores incluindo domínios filhos os quais representam um subconjunto de processadores.

Neste *kernel*, o domínio de escalonamento de nível inferior continha todos os processadores virtuais em um processador físico, sendo denominado domínio de processador (*CPU domain*). Um segundo nível continha todos os processadores em um mesmo nó de um sistema NUMA ou todos os processadores em um sistema SMP e era chamado domínio físico (*physical domain*). O terceiro nível, domínio de nó (*node domain*), continha todos os processadores em um sistema NUMA [34, 35].

Os domínios de escalonamentos eram particionados em grupos que representavam um subconjunto dos elementos contidos no domínio. Por exemplo, em um domínio de processador um grupo era um conjunto de processadores virtuais.

O balanceamento de carga era, então, realizado entre grupos de um domínio de escalonamento, isto, processos movidos entre processadores de grupos pertencentes a um mesmo domínio de escalonamento [27].

Essa organização permitia que políticas distintas fossem utilizadas em cada nível da hierarquia [35]. No domínio de processador as tentativas de balanceamento ocorriam frequentemente, mesmo quando o desbalanceamento entre processadores virtuais era pequeno [27]. Essa prática era possível porque processadores virtuais compartilham sua *cache*. No domínio físico, as tentativas de balanceamento eram menos frequentes [27], pois o custo a transferência de um processo nesse nível é maior. No domínio de nó, o escalonador devia analisar cuidadosamente o impacto da migração de um processo para outro nó, pois o custo poderia ser proibitivo [35].

3.5.4 Escalonamento a partir da Versão 2.6.23

A partir da versão 2.6.23, o *completely fair scheduler* foi introduzido. Esse escalonador utiliza árvores rubro-negras em vez de filas no gerenciamento de processos.

A principal ideia por trás do CFS é manter o balanceamento (equidade) no fornecimento de tempo de processador aos processos. Segundo seus desenvolvedores, 80% do projeto do CFS pode ser resumido em uma única sentença: "O CFS basicamente modela processamento multitarefa ideal em hardware real". Com multiprocessamento ideal, se há dois processos rodando, cada um receberia 50% de poder computacional [36].

Para determinar o balanceamento, o CFS mantém armazena a quantia de tempo fornecida a um processo que é chamado tempo de execução virtual [30]. Na prática, o tempo de execução virtual é o tempo de execução real normalizado em relação ao número de processos em execução [36].

Os processos prontos para execução são armazenados em um árvore rubro-negra ordenada por seu tempo de execução virtual. Processos que executaram menos tempo são, portanto, armazenados no lado esquerdo da árvore enquanto processos com maior tempo de execução virtual permanecem ao lado direito [30].

A fim de aplicar equidade, o escalonador seleciona o nó mais à esquerda na árvore para execução. O tempo desta execução é adicionado ao tempo de execução virtual do processo e ele é, então, reinserido na árvore. Dessa forma, à medida que os processos do lado esquerdo ganham acesso ao processador eles começam a migrar para o lado direito. Portanto, eventualmente todo processo é executado, ou seja, não há abandono de processos [30].

A implementação do CFS é destinada principalmente a processos não classificados como tempo real [37]. O escalonamento em tempo real permanece basicamente igual ao que era realizado nos primeiros *kernels* 2.6. Entretanto, ele utiliza apenas 100 filas de processos, uma para cada prioridade em tempo real, e não utiliza um conjunto de processos expirados [36].

Nos demais tipos de processo, as prioridades são utilizadas como um fator de

decaimento para o tempo que o processo é permitido executar. Processos menos prioritários têm um fator de decaimento maior, processos mais prioritários têm um fator de decaimento menor. Assim processos menos prioritários irão obter menos tempo de processador que processos mais prioritários.

Observe que em virtude do que foi descrito até aqui, o *quantum* dos processos é definido dinamicamente.

Além disso, o CFS introduz o conceito de escalonamento em grupo (*group scheduling*). A ideia por trás do escalonamento em grupo é que dado um processo que dá origem a outros, todos esses processos compartilham seu tempo virtual. Em outras palavras, em vez de cada processo pertencente a um grupo ser tratado de forma individual com seu próprio tempo virtual, o tempo virtual pertence ao grupo. Assim processos individuais recebem o mesmo tempo de escalonamento que o grupo, introduzindo um novo grau de equidade [30].

O CFS também introduz a ideia de classes de escalonamento, isto é, cada processo pertence a uma classe de escalonamento que determina como o processo será tratado. Uma classe de escalonamento define um conjunto de funções comum (uma interface), que determinam o comportamento do escalonador [30].

Classes de escalonamento tornam-se ainda mais interessantes em conjunto com domínios de escalonamento. Domínios de escalonamento permitem agrupar um ou mais processadores hierarquicamente para propósitos de balanceamento de carga e segregação. Um ou mais processadores podem compartilhar políticas de escalonamento (e balanceamento de carga entre eles) ou implementar políticas de escalonamento independentes a fim de segregar processos [30].

Capítulo 4

Escalonamento em Rede *Intrachip*

Desde sua invenção computadores passaram por mudanças substanciais em sua arquitetura. À medida que esta se modernizava diversas melhorias e aprimoramentos foram introduzidos. Em sua maior parte, esses melhoramentos tinham como objetivo remover responsabilidades do processador, diminuir tempo de processamento e aumentar paralelismo. Neste capítulo, iremos propor um mecanismo de escalonamento *intrachip* como forma de liberar o processador para tarefas mais relevantes, diminuir o tempo de escolha de um processo e consequentemente melhorar o desempenho.

4.1 Motivação para o Escalonamento *Intrachip*

Computadores modernos, em quase sua totalidade, evoluíram de um modelo conhecido como Arquitetura de Von Neumann (*Von Neumann Architecture*) [38, 39]. O modelo de Neumann consistia de uma memória capaz de armazenar tanto instruções quanto dados, uma unidade responsável por realizar operações lógicas e aritméticas sobre dados e, uma unidade de controle apta a interpretar uma instrução e selecionar cursos alternativos de execução baseados nos resultados das operações anteriores. As unidades lógica/aritmética (ULA) e de controle (UC) juntas formavam a unidade central de processamento (UCP). A UCP estava ligada à memória e a dispositivos de entrada e saída através de barramentos por onde dados e instruções podiam trafegar [40].

O modelo de Neumann às vezes é referenciado como Arquitetura de Princeton (*Princeton Architecture*) ou ainda, esquema de programa armazenado (*Stored Program Scheme*) [23, 38, 39]. Essa diversidade de denominações existe porque muitos historiadores acreditam que muito crédito é dado a Neumann em detrimento de outros pesquisadores, que também colaboraram na concepção da arquitetura [23, 39]. Alguns atribuem a Turing o conceito de programa armazenado [41, 42], outros afirmam que Eckert e Mauchly, que trabalharam no desenvolvimento do ENIAC ao qual Neumann se juntou mais tarde, também já haviam discutido essa abordagem [43].

Nesse texto optamos por não fazer distinção entre essas denominações.

À despeito das controvérsias, o esquema de programa armazenado tornou-se cada vez mais popular e influenciou a maior parte dos computadores projetados após sua elaboração.

Nessa arquitetura, durante a execução de um programa a UCP realiza repetidos ciclos de busca – recuperação de uma instrução da memória, decodificação – identificação da instrução e, execução – realização da operação associada a instrução [44]. Em alguns casos, a execução da instrução envolve o recebimento ou envio de dados para a memória. Em outros, pode requerer a interação com um dispositivo de entrada e saída.

A Arquitetura de Von Neumann original modelava basicamente um computador SISD da taxonomia de Flynn. Nela, uma instrução era executada após a outra, em uma sequência, com palavras de memória trafegando individualmente pelo barramento [44–46].

Entretanto, embora o modelo fosse considerado inovador e tenha sido muito bem sucedido e influente, ele não era perfeito. A principal contribuição do Modelo de Neumman, foi a exploração do conceito de programa armazenado e o detalhamento de como um computador de programa armazenado processa a informação.

O grande inconveniente dessa arquitetura é que ela era estritamente serial, isto é, podia realizar apenas uma operação de cada vez. Essa característica fundamental do modelo, apesar de torná-lo mais simples e elegante, era muito restritivo e com o tempo mostrou-se um fator limitante.

Por exemplo, como o modelo propunha uma separação entre memória e UCP, muito tempo era gasto transferindo instruções e dados entre ambos. À medida que o tamanho médio da memória e a disparidade na frequência de operação desta em relação a UCP aumentavam, esse problema tornava-se cada vez mais um limitador de desempenho. Como consequência direta, vários ciclos de processamento eram desperdiçados pela UCP durante uma transferência de dados com a memória. Esse problema foi denominado gargalo de Von Neumann (*Von Neumann Bottleneck*) por Backus em 1977 [47].

Em face dessas limitações, diversas alterações foram propostas e implementadas visando atenuar ou eliminar deficiências intrínsecas ao modelo.

A exploração dos princípios de localidade temporal e espacial conduziu a concepção de memórias cache, cujo objetivo era ser uma área de armazenamento temporária, menor e mais rápida para dados e instruções provenientes da memória principal. Com essa organização, sempre que a UCP precisa acessar uma palavra de memória, ela faz uma solicitação a cache. Se a palavra de memória já se encontra na cache ela é entregue diretamente a UCP. Em caso negativo, um bloco de memória contendo a palavra requerida é copiado para a cache, de onde a UCP pode recuperá-la. A ideia

é manter na cache, palavras de memória que foram utilizadas recentemente (localidade temporal) ou que lhes são adjacentes (localidade espacial) e que, portanto, tem uma boa probabilidade de serem referenciadas em um futuro próximo [23].

Adicionalmente, o uso de duas memórias cache separadas, uma para dados e outras para instruções, permitiu a UCP realizar uma busca de instrução em paralelo com uma leitura/escrita de dados. A utilização de memórias separadas para dados e instruções foi utilizada inicialmente na Arquitetura de Harvard, cujo nome constitui uma oposição a Arquitetura de Princeton, que possuía uma única memória. Além disso, memórias cache foram organizadas em diversos níveis, com diferentes granularidades, com o objetivo de ocultar ainda mais latência no acesso a memória principal [23].

O advento do mecanismo de interrupção por *hardware* tornou possível sobrepor computação e operações de entrada e saída. Com esse esquema, um dispositivo pode ser configurado pela UCP a fim de realizar uma operação requerida. A UCP pode, então, executar instruções (de outros programas) paralelamente. Quando a operação termina, o dispositivo aciona uma interrupção por hardware sinalizando à UCP sua conclusão. Nesse momento, o programa inicial poderia ser retomado [18]. Por sua vez, a introdução do DMA (*direct memory access*) permitiu aos dispositivos se comunicarem diretamente com a memória sem intervenção da UCP [18].

Arquiteturas vetoriais (SIMD) adicionaram o uso de instruções que operam sobre vários elementos de dados. A ideia por trás desse conceito é diminuir a latência no acesso a memória principal operando com blocos [vetores] de dados em lugar de palavras individuais. Se os elementos do vetor são adjacentes, eles podem ser trazidos de uma única vez para uma cache de memória, como forma de atenuar o custo de acesso à memória principal [23].

A introdução do mecanismo de *pipeline* permitiu que várias instruções pudessem estar em execução simultaneamente. Com o pipeline as operações realizada pelas unidade central de processamento (busca, decodificação, execução, acesso a memória) foram divididas em estágios. Nesse esquema, o avanço de uma instrução por esses estágios indica o progresso de sua execução. Assim, com o avanço de uma instrução, o estágio pode ser usado na execução de outra instrução, desde que possíveis dependências e problemas sejam tratados ou inexistentes. Como resultado, várias instruções podem executar ao mesmo tempo, em estágios de execução diferente [23].

A utilização de sistemas multiprocessadores/*multicores* (MIMD) adicionou a possibilidade de execução de vários processos simultaneamente. Embora sistemas de tempo compartilhado com um processador possam transmitir a sensação de simultaneidade, essa experiência é baseada na execução alternada de vários processos durante um curto intervalo de tempo, isto é, não melhora a execução paralela de instruções [23]. Sistemas multiprocessadores, por sua vez, melhoram o desempenho

do sistema permitindo a execução paralela de instruções de diversos processos.

A adição de núcleos de processamento tem o potencial de aumentar o paralelismo na execução de instruções, mas também reforça a complexidade do sistema, podendo dar origem a outras complicações. Em um sistema de memória compartilhada centralizada, por exemplo, a competição pelo barramento pode resultar em uma latência maior no acesso à memória. A fim de contornar com esse problema, surgiram as arquiteturas de memória distribuídas (NUMA). Nessa organização, a memória é distribuída entre os processadores a fim de diminuir a contenção e aumentar a largura de banda [23].

É fácil verificar que em todas as situações anteriores os objetivos das modificações eram ocultar a latência ou adicionar paralelismo. Em ambos os casos a intenção subjacente é aumentar o número de instruções executadas por unidade de tempo. Em outras palavras, o papel do processador é executar instruções de programas a ele submetidos.

Nossa opinião é que o processador deveria se ater a execução de instruções que colaborem para o progresso dos programas executados no sistema. Tarefas secundárias, sem impacto direto na execução desses programas deveriam ser relegadas a circuitos e dispositivos especializados com os quais o processador pudesse interagir, se necessário, em momentos apropriados.

O escalonamento de processos se encaixa perfeitamente nesse contexto. Sua essência é permitir uma utilização mais racional do processador e realizar uma divisão mais justa do acesso a ele. Apesar disso, o escalonamento em si não contribui diretamente para o progresso na execução de qualquer programa, ele apenas define como esse progresso se dará.

Embora o escalonamento traga benefícios diretos por adicionar graus de justiça na execução dos processos e diminuir o tempo total de processamento por meio de multiprogramação/multitarefa, ele também adiciona um custo indireto relacionado a manipulação de estruturas de dados e a seleção de processos. Além disso, como vimos no capítulo anterior, se o sistema operacional utiliza uma estrutura de dados única para os processos, à medida que o número de núcleos aumenta, a contenção no acesso a estrutura torna-se um gargalo para o sistema. Se por outro lado, o sistema operacional utiliza estruturas de dados independentes para cada processador, há o custo de realizar a migração de processos e o custo da contenção no acesso a estrutura de dados entre um subconjunto de processadores durante as operações de migração.

Nesse sentido, o escalonamento diminui o desempenho em potencial do sistema uma vez que vários ciclos de execução são utilizados pelo sistema operacional realizando o escalonamento propriamente dito. Dessa forma, tempo de processamento que deveria ser gasto com a execução de instruções úteis ao desenvolvimento das

atividades do programa é gasto com uma tarefa de suporte.

Na próxima seção iremos descrever um modelo de escalonamento baseado em uma rede de *chips* cujo objetivo principal é liberar o processador e consequentemente o sistema operacional dessa atribuição.

4.2 Organização da Rede de Escalonamento *Intrachip*

Como mencionamos, o escalonamento de processo é um ponto de melhoria em potencial em sistemas computacionais. A ideia principal é remover essa responsabilidade do sistema operacional, permitindo que um hardware especializado realize essa tarefa. Assim, as atividades do processador se concentrariam em sua função principal, a execução de instruções dos diversos processos ativos na memória.

Uma maneira de implementar essa modificação é introduzindo uma rede de *chips* cuja função primordial é realizar o escalonamento e balanceamento de carga.

Nesse modelo, há um *chip* de escalonamento para cada processador no sistema, com o qual ele está conectado. O processador pode enviar e solicitar processos a seu *chip*, conforme necessário. O *chip* mantém uma fila onde ele armazena os processos que estão sob sua responsabilidade.

A rede de *chips* possui uma topologia particular cujo papel é definir como seus componentes estão interligados. Dessa forma, cada *chip* pode estar conectado com até outros $n-1$ *chips*, onde n é número de processadores. Essas conexões permitem a um *chip* enviar processos a determinados vizinhos e receber destes a quantidade de processos que eles detém, assim como, receber processos e enviar o tamanho de sua fila interna para outros vizinhos.

A figura 4.1 mostra a estrutura de interna de um *chip* de escalonamento. A fila no centro da figura representa o conjunto de processos controlados pelo *chip*. Na parte superior, o triângulo representa uma porta de saída enquanto o hexágono representa um canal de entrada, ambos para o processador. O conjunto de hexágonos na parte superior direita representa um canal de entrada para processos. O triângulo na parte inferior direita representa uma porta de saída para envio de informações sobre o número de processos no chip. O triângulo na parte superior esquerda representa uma porta de saída para envio de processos. E finalmente, o hexágono na parte inferior esquerda é um canal de entrada para informações sobre número de processos de algum *chip* vizinho.

Nesse esquema, o número de portas e canais varia de acordo com a topologia. Por exemplo, se a topologia prevê que cada *chip* de escalonamento esteja ligado a outros dois, então as portas e canais para *chips* vizinhos existirão em pares. Além disso, os

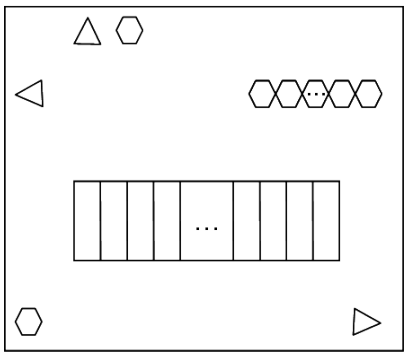

Figura 4.1: A estrutura de um *chip* de escalonamento

canais possuem um tamanho, que indica o número elementos que eles podem receber em um dado instante. Canais de processos podem receber até n, $n \geq 0$, processos vindos de um vizinho. Canais de tamanho de fila recebem apenas uma informação por vez.

A figura 4.2 exibe uma rede de *chips* com quatro elementos conectados um a um. A porta de saída de processos de cada *chip* está conectada com o canal de entrada de processos de seu vizinho à esquerda. A porta de saída de informação sobre o número de processos está conectada ao canal correspondente de seu vizinho à direita. Todos os *chips* que compõem a rede possuem a mesma estrutura.

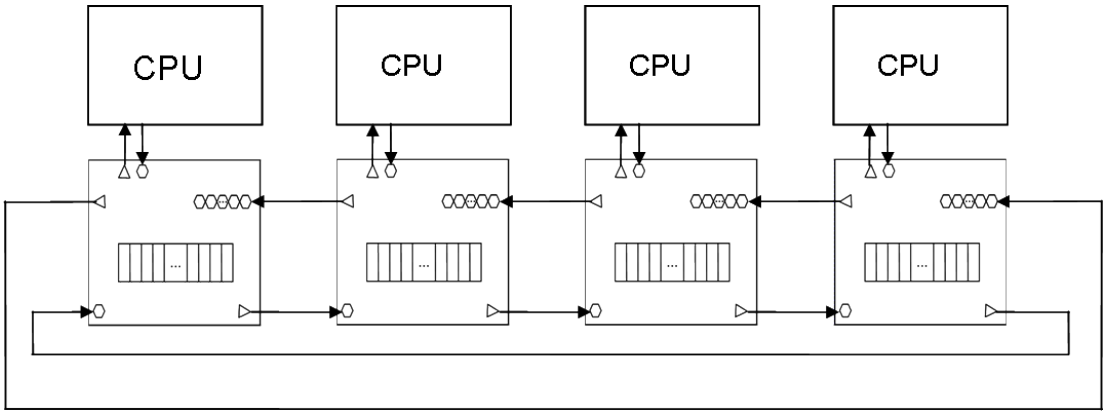

Figura 4.2: Rede com chips de escalonamento conectados individualmente

A figura 4.3 mostra uma rede de escalonamento com quatro *chips* conectados dois a dois. Os processadores foram omitidos por simplicidade. Nessa figura cada *chip* está conectado a dois vizinhos à esquerda, para onde pode enviar processos e de onde recebe informações sobre o número de processos naqueles vizinhos. As conexões para envio de processos utilizam linhas contínuas, enquanto as conexões para envio de informações sobre número de processos estão pontilhadas. Observe que a cada conexão para envio de processos corresponde uma conexão para envio

de informações sobre número de processos, ligando os mesmos dois *chips* mas em sentido contrário, exatamente como na figura anterior.

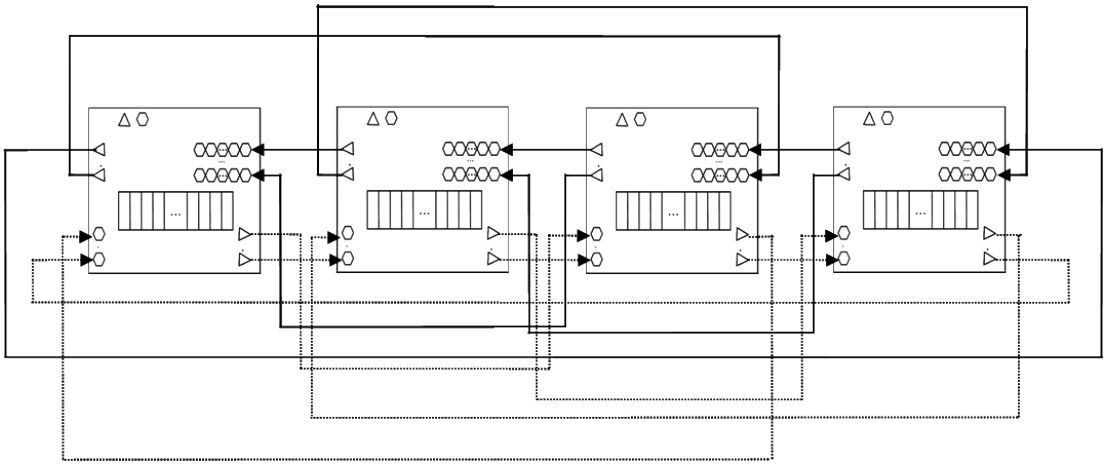

Figura 4.3: Rede com chips de escalonamento duplamente conectados

Topologias arbitrárias podem ser constituídas de forma semelhante, desde que a simetria na ligação entre os *chips* seja mantida. Além disso, como cada *chip* está conectado a apenas um único processador, vários pontos de entrada para rede estão disponíveis no sistema, diminuindo potencialmente o retardo e a contenção na troca de processos.

4.3 Funcionamento da Rede de Escalonamento *Intrachip*

No decorrer de sua operação um *chip* de escalonamento realiza um ciclo contínuo de recepção e envio de processos cujos objetivos principais são fornecer processos para execução e realizar balanceamento de carga.

Em uma situação típica, um novo processo seria criado em um processador que já está alocado. Essa situação ocorre na vida real, por exemplo, quando um programa é iniciado por um comando em um *shell*, ou quando um processo cria processos filhos para realização de tarefas específicas. Como o processador estaria ocupado executando o processo original, isto é, o *shell* ou processo pai, o processo recém-criado seria enviado ao *chip* de escalonamento associado ao processador. Eventualmente, o processo em execução esgotaria sua fatia de tempo e seria também enviado ao *chip* de escalonamento.

Sempre que o processador fica ocioso ele faz uma solicitação ao *chip* com o qual está associado. Essa situação pode ocorrer tanto pelo término quanto pelo esgotamento do *quantum* do processo ora em execução. Na etapa seguinte, o *chip*

responde com o envio do primeiro processo em sua fila interna, que então passará a ser executado pelo processador. Entenda-se por etapa um intervalo de tempo muito pequeno na qual um *chip* pode realizar uma operação, tal qual um ciclo de processador.

Quando um *chip* recebe um processo, este é inserido em sua fila de processos. Esta fila pode estar organizada de acordo com alguma política, como *FIFO* utilizar mecanismos de prioridade. Em uma política *FIFO* um processo recém-chegado ingressaria sempre no final da fila, em uma política por prioridades o processo poderia ingressar em qualquer posição da fila.

Em um intervalo de tempo regular cada *chip* de escalonamento envia o tamanho de sua fila interna de processos a um conjunto de chips vizinhos. Convencionamos na seção anterior que estes seriam os vizinhos à direita do *chip*. Os vizinhos que recebem o tamanho da fila de um dado *chip* são aqueles que potencialmente, em virtude da topologia, podem lhe enviar processos. Como todo *chip* está conectado a pelo menos um vizinho, em cada passo todos o os *chips* recebem informações sobre o tamanho das filas dos vizinhos aos quais eles podem enviar processos, isto é, seus vizinhos à esquerda.

No intervalo de tempo seguinte, com o objetivo de manter o balanceamento de carga, cada *chip* através da aplicação de uma política previamente determinada, seleciona um conjunto de vizinhos como prováveis destinos de processos. Então, com o uso das informações sobre as filas desses vizinhos, os processos são enviados àqueles cujo tamanho das filas sejam inferiores ao do *chip*. Esse último passo previne um envio desnecessário de processos.

Para propósitos desse trabalho quatro políticas foram escolhidas a fim de realizar a seleção inicial dos destinos para os processos, a saber, as políticas alternada, inversamente proporcional, menor e randômica.

A política alternada realiza a seleção variando a escolha entre todos os possíveis destinos. Assim todos os destinos são escolhidos sequencialmente, um após o outro, eventualmente retornando ao primeiro, de onde o ciclo se repete.

Na política randômica uma faixa de números é divida em intervalos de tamanhos iguais, onde cada intervalo corresponde a um vizinho do *chip*. Então, a escolha de um número aleatório, que recai em um desses intervalos determina o destino do processo.

A política inversamente proporcional funciona de forma similar a randômica, mas cada intervalo possui tamanho inversamente proporcional ao tamanho da fila do *chip* vizinho. Dessa forma, *chips* com mais processos tem menos chances de serem escolhidos como destino.

Com a política menor, *chips* com menos processos tem prioridade sobre aqueles com mais processos. Portanto, os destinos são escolhidos em função de vetor cujo

tamanho das filas dos vizinhos estão ordenadas em ordem crescente.

Cabe ressaltar que de acordo com a topologia, um *chip* pode enviar processos a vários vizinhos em um dado instante de tempo. Além disso, em função do tamanho do canal de entrada de processos, um *chip* pode receber diversos processos de uma mesma origem. Em outras palavras, a política pode operar sobre grupos, em certas ocasiões selecionando um destino mais de uma vez.

Até o momento afirmamos que o processador e o *chip* trocam processos entre si, mas nada foi dito sobre a maneira como isto ocorre. Em um sistema como um computador pessoal, por exemplo, a troca de processos poderia consistir no envio e recebimento de ponteiros para o bloco de controle do processo (*process control block*). A partir dessa informação, com o uso de uma estrutura conhecida, o *chip* poderia obter todas as demais informações que fossem necessárias, tais como prioridade de um processo e o tamanho de seu *quantum*. Uma alternativa seria fornecer ao *chip* um conjunto essencial de informações sobre o processo.

Anteriormente, mencionamos que a organização da fila de processos em um *chip* poderia adotar uma política específica. Entretanto, a escolha de uma determinada organização de fila em conjunto com decisões específicas de balanceamento pode resultar em situações adversas ao progresso dos processos.

Em particular, a escolha de uma política por prioridades com processos sendo organizados em ordem decrescente de prioridade na fila de processos, juntamente com a opção de enviar processos menos prioritários durante uma etapa de balanceamento, pode conduzir ao abandono de processos. Note que em um sistema ocupado, com as características descritas, processos menos prioritários poderiam vagar pela rede sem nunca obterem acesso ao processador.

Um modo de resolver esse problema seria manter duas informações de prioridade para cada processo: a prioridade real e uma prioridade ajustada em função do número de migrações impostas ao processo. A primeira seria estática, a segunda, dinâmica. O escalonamento seria realizado então em função da prioridade ajustada do processo.

Com esse esquema, no momento de criação de um processo, ambas as prioridades seriam definidas com o mesmo valor. Então, toda vez que o processo sofresse uma migração, a prioridade ajustada seria incrementada. Finalmente, quando o processo fosse selecionado para execução sua prioridade ajustada seria definida novamente com o mesmo valor da prioridade real. Esse mecanismo possui semelhanças com o envelhecimento de processos, mas é baseado apenas no número de migrações sofridas pelo processo.

De forma similar, poderia haver retenção na execução de processos mesmo se o processo mais prioritário na fila do *chip*, fosse enviado. Observe que a execução desse processo poderia ser postergada em função de uma constante migração entre

chips, já que o processo poderia não permanecer tempo suficiente na fila de um *chip* a fim de ser selecionado para execução.

Uma maneira de lidar com esse inconveniente seria limitar o número de migrações que um processo poderia sofrer. Contudo, por si só esta medida não endereça outras questões como por exemplo, o progresso da execução de processos menos prioritários. Isto implica que mecanismos adicionais de decisão baseados na prioridade dos processos necessitariam ser implementados.

4.4 Implementação da Rede de Escalonamento *Intrachip*

Nessa seção vamos descrever os detalhes de implementação da rede de escalonamento proposta anteriormente. Salientamos que a rede não foi implementada em *hardware* mas em vez disso, sua arquitetura e operação foi simulada por *software* com objetivo de verificar suas propriedades e aplicação.

4.4.1 *Software* de Simulação

SystemC é uma linguagem de projeto de sistemas baseada em C++ que inclui conceitos tanto de *hardware* quanto de *software*. Seu objetivo é permitir a engenheiros projetarem tanto *hardware* quanto *software* que existiriam em um sistema final em um alto nível de abstração. Este alto nível de abstração fornece a equipe de projeto um entendimento prévio fundamental dos complexos detalhes e interações do sistema como um todo e permite melhores compromissos do sistema, verificação melhorada e antecipada, e ganho de produtividade através do reuso de modelos de sistemas anteriores como especificações executáveis [48].

Estritamente falando SystemC não é uma linguagem, mas uma biblioteca de classes em uma linguagem bem estabelecida, C++. O projeto de um sistema então se restringe a criação de um conjuntos de arquivos com códigos fontes e cabeçalhos contendo as especificações requeridas. Estes arquivos são compilados e ligados a biblioteca de classes do SystemC. O resultado é um arquivo executável que contém um *kernel* de simulação e a funcionalidade projetada [48].

SystemC fornece muitos mecanismos orientados a *hardware* que não estão normalmente disponíveis em uma linguagem de *software* mas são necessários para modelar *hardware*. Por exemplo, através de um modelo de tempo, SystemC fornece mecanismos para que o tempo atual de simulação seja obtido e implementa atrasos de tempo específicos. Ela também provê uma variedade de tipos de dados que são requeridos por *hardware* digital que não são fornecidos como tipos nativos em C++. Além disso, SystemC permite que o projeto seja dividido hierarquicamente a fim de

gerenciar sua complexidade e provê meios para modelar a comunicação entre esse componentes. A linguagem também inclui mecanismos de concorrência, ainda que cooperativa, entre os elementos de um sistema [48].

SystemC utiliza o conceito de módulos para representar componentes. Estes componentes podem representar *hardware*, *software*, ou qualquer entidade física. Os componentes podem ser simples ou complexos e, embora independentes, podem ser conectados em uma hierarquia. Por exemplo, um registrador poderia ser mapeado em um módulo e a conexão de vários desses componentes originaria um banco de registradores.

Em SystemC, funções (ou métodos) pertencentes a um módulo podem ser registrados como processos junto ao *kernel* de simulação. Esse ação faz com que essas funções sejam executadas pelo *kernel* em momentos apropriados.

Em uma situação típica de simulação, vários processos (registrados pelos diversos componentes do sistema) podem requerer execução em um dado instante de tempo. Isso faz com que o *kernel* de simulação execute cada processo e atualize suas saídas. A atualização de saídas pode tornar necessária a execução de algum processo, incluindo o próprio, naquele mesmo instante. Nesse caso, o tempo de simulação não é incrementado e a etapa de execução de processos é retomada. Eventualmente, quando nenhum processo adicional necessita ser executado naquele momento, o tempo na simulação é avançado. Se nenhum processo precisa ser executado no futuro, a simulação termina [48]. A realização da fase de execução seguida pela fase de atualização é um conceito importante do SystemC, conhecido como ciclo delta (*delta-cycle*).

Componentes individuais (módulos) podem ser conectados por meio de canais e portas. Um canal é um mecanismo que permite a comunicação entre módulos por meio de uma interface bem definida. Um canal está associado a uma estrutura de dados subjacente que é capaz de manipular informação em um formato específico. Por sua vez, uma porta é basicamente um ponteiro para um canal, isto é, um meio de acessá-lo a partir de um modulo distinto. A tabela 4.1 mostra alguns dos métodos que compõem a interface de um canal do tipo *FIFO*. Portas implementam um conjunto de interfaces por meio das quais interagem com canais. A tabela 4.2 exibe alguns dos métodos que compõem a interface de saída para canais *FIFO*.

Método	**Descrição**
`T read()`	Lê um elemento do tipo T a partir do canal
`void write(const T&)`	Escreve (armazena) um elemento do tipo T no canal
`int num_available()`	Obtém o número de elementos armazenados no canal
`int num_free()`	Obtém o número de elementos que ainda podem ser armazenados no canal

Tabela 4.1: Alguns métodos de canais *FIFO*

Método	Descrição
`void write(const T&)`	Escreve (armazena) um elemento do tipo T no canal apontado pela porta associada
`bool nb_write(const T&)`	Escreve (armazena) sem bloquear um elemento do tipo T no canal apontado pela porta associada
`int num_free()`	Obtém o número de elementos que ainda podem ser armazenados no canal apontado pela porta associada

Tabela 4.2: Alguns métodos da interface de saída para canais *FIFO*

4.4.2 Detalhes Essenciais de Implementação

A implementação do modelo concebido nas seções anteriores resultou em um conjunto de arquivos fontes com aproximadamente 3000 linhas de código. A tabela 4.3 mostra os principais arquivos que compõe o sistema de simulação para o modelo. Alguns arquivos secundários e auxiliares foram omitidos por simplicidade. Além disso, mesmo os arquivos citados não tiveram toda sua funcionalidade explicitada. Faz-se necessário detalhar apenas características essenciais do funcionamento dos componentes mais importantes.

O código contido em `main.cpp` responde por diversas tarefas fundamentais realizadas pelo sistema. Em primeiro lugar, o tratamento dos parâmetros de entrada fornecidos pelo usuário é realizado nesse trecho de código com auxílio da classe `Parameter`. Diversos parâmetros foram definidos tais como tempo de simulação, tempo médio de nascimento de processos, tempo médio de execução de processos, topologia, política de envio de processos, tempo de *quantum*, unidade de tempo de operação do *chip*, diretório de trabalho e assim por diante.

Além disso, praticamente todo dispositivo existente durante a simulação é instanciado durante a execução de `main`. Adicionalmente `main` responde pela conexão da topologia escolhida, pelo estabelecimento dos mecanismos de rastreamento de informações sobre os processos, pelo início da simulação propriamente dita e pela geração de gráficos e algumas estatísticas no final desta.

A tabela 4.4 descreve a função de cada parâmetro aceito pelo sistema de simulação. Note que alguns parâmetros aceitam até n argumentos, onde n é o número de *chips*/processadores informado no parâmetro `-count`. Nesses casos, quando o número de argumentos informados é menor que n, o valor do último argumento fornecido torna-se *default* para todos os omitidos.

O parâmetro `-topology` merece atenção especial. O argumento desse parâmetro é um conjunto de números entre 1 e $n - 1$, onde cada valor representa um vizinho do *chip*. Por exemplo, uma topologia $T = \{1, 2, 4\}$ indicaria que o *chip* estaria conectado ao primeiro, segundo e quarto *chip* vizinho.

Arquivo	Descrição
`chip.cpp`	Implementa operações realizadas pelos *chips* de escalonamento.
`cpu.cpp`	Simula operações sobe responsabilidade dos processadores.
`main.cpp`	Conecta a topologia requisitada e inicia simulação.
`graphic.c`	Cria gráficos relativos a simulação através do Gnuplot.
`nocparm.c`	Salva e recupera parâmetros de simulação.
`mergesort.c`	Implementa o algoritmo de ordenação utilizado com a política.
`parameter.cpp`	Fornece mecanismos que simplificam a obtenção de parâmetros do usuário.
`policy.cpp`	Implementa as classes de políticas através de herança e polimorfismo.
`process.cpp`	Descreve uma estrutura simplificada de processo e operações pertinentes.
`timedist.c`	Permite a utilização de um arquivo externo como distribuição dos tempos de nascimento e execução dos processos.
`timetrace.cpp`	Implementa um mecanismo de rastreamento de medidas de tempo associadas aos processos.
`util.c`	Fornece um conjunto de funções de uso comum.

Tabela 4.3: Principais arquivos do sistema de simulação

Parâmetro	Argumentos	Descrição
`-chiponly`		Simula apenas *chips*
`-count`	*número* (n)	Número de *chips*/processadores na simulação
`-ctu`	*quantidade*	Quantidade de unidades de tempo na qual os *chips* operam
`-dir`	*diretório*	Diretório de trabalho
`-dist`	$arquivo_1 ... arquivo_n$	Nome de arquivos contendo distribuições de tempos a serem usados em lugar de `-mbt` e `-met`
`-mbt`	$media_1 ... media_n$	Tempo médio de nascimento dos processos
`-met`	$media_1 ... media_n$	Tempo médio de execução dos processos
`-policy`	*política*	Política a ser empregada na escolha de destinos para o envio de processos
`-processes`	$quantidade_1 ... quantidade_n$	Número inicial de processos em um *chip*
`-sa`	*quantidade*	Número de processos a serem enviados em cada etapa de balanceamento
`-time`	*tempo*	Tempo de simulação.
`-topology`	*topologia*	Descrição da topologia de conexão.
`-ts`	*quantum*	Tamanho do *quantum*

Tabela 4.4: Parâmetros do sistema de simulação

Algoritmo de Operação do *Chip*

Como mencionado anteriormente, *chips* operam em um intervalo de tempo que é definido através de um parâmetro da simulação. Por falta, o sistema assume que o valor desse parâmetro equivale a uma unidade de tempo. Cada vez que este intervalo de tempo ocorre o *chip* executa o algoritmo mostrado na listagem 4.1:

```
1  Enquanto o tempo de simulação não houver expirado.
2
3    /* Leitura de Processos vindos dos vizinhos à direita */
4    Para cada canal de entrada de processos cep_i associado ao chip vizinho
         i.
5      Enquanto cep_i possuir processos.
6        Obtenha o processo p_j de cep_i.
7        Insira p_j  na fila de processos do chip.
8      FimEnquanto.
9    FimPara.
10
11   /* Leitura de Processos/Requisições vindos da cpu */
12   Enquanto o canal de entrada da cpu possuir requisições ou processos.
13     Obtenha a requisição ou processo rp_x.
14     Se rp_x é uma requisição.
15       Se o chip possuir processos.
16         Obtenha p_j da fila de processos do chip.
17         Envie p_j pela porta de saída para cpu.
18       Senão.
19         Envie uma negação pela porta de saída para cpu.
20       FimSe.
21     Senão.    /* rp_x é um processo */
22       Insira rp_x na fila de processos do chip.
23     FimSe.
24   FimEnquanto.
25
26   /* Leitura do tamanho da fila dos vizinhos à esquerda */
27   Para cada canal de entrada de informação cei_k sobre número de
         processos associado ao chip vizinho k.
28     Obtenha o número de processos de cei_k.
29     Armazene no vetor o  número de processos do vizinho k.
30   FimPara.
31
32   /* Escolha dos chips destinos de processos, políticas como menor e
         inversamente proporcional dependem do número de processos no
         vizinho. */
33   Aplique a política para escolher destinos de envio de processos
         considerando, se necessário, a informação do vetor de número de
         processos.
34
```

```
35    /* Envio de processos aos vizinhos à esquerda. */
36    Para cada chip vizinho de destino d_y escolhido.
37      Se o número de processos em d_y é menor que o número de processos no
            chip.
38        Se o chip possuir processos.
39          Obtenha p_j da fila de processos do chip.
40          Envie p_j pela porta de saída de processos para o chip d_y.
41          Aumente o número de processo de d_y no vetor em uma unidade.
42        FimSe.
43      FimSe.
44    FimPara.
45
46    /* Escrita do tamanho da fila para os vizinhos à direita */
47    Para cada porta de saída de informação psi_i sobre número de processos.
48      Envie o número de processos do chip por psi_i.
49    Fim Para.
50
51    Aguarde até a próxima etapa. /* Em geral uma unidade de tempo */
52
53  FimEnquanto.
```

Listagem 4.1: Algoritmo de Operação do Chip

O algoritmo de operação do *chip* realiza basicamente os seguintes passos:

1. Recebe processos de seus vizinhos à direita (linhas 3-9)
2. Recebe processo e responde requisições do processador (linhas 11-24)
3. Recebe o tamanho da fila de seus vizinhos à esquerda (linhas 26-30)
4. Aplica a política selecionada a fim determinar destinos para o envio de processos (linhas 32-33)
5. Envia um processo a cada vizinho escolhido (se ele possui menos processos que o próprio *chip*) (linhas 35-44)
6. Envia o tamanho de sua fila para os vizinhos à direita (linhas 46-49)
7. Aguarda até o próximo intervalo de tempo (linha 51)

Os *chips* funcionam, dessa forma, num ciclo contínuo de trocas de processos que só termina com o final da simulação. Portanto, durante todo este período os *chips* buscam um balanceamento entre suas cargas. Observe que a política aplicada na escolha de destino de processo é uma das descritas na seção 4.3, entretanto a escolha do processo a ser enviado tanto ao processador quanto a um vizinho obedece a política *FIFO*. Essa opção evita os problemas potenciais de abandono de processos

mencionado previamente e, conquanto não seja a mais adequada em termos de interatividade e processamento em tempo real, é útil na avaliação de tempo médio de resposta e tamanho médio de fila.

Algoritmo de Operação do Processador

O código que implementa o comportamento dos processadores funciona de maneira semelhante, mas tem suas peculiaridades. A listagem 4.2 apresenta o algoritmo de operação do processador.:

```
1  processo_em_execucao = NENHUM.
2  tempo_a_transcorrer = 0.
3  tempo_morte = INFINITO.
4  tempo_chegada = INFINITO.
5  tempo_quantum = INFINITO.
6  tempo_nascimento = tempo_proximo_nascimento( tempo_medio_nascimento ).
7  requisicao_pendente = FALSO.
8
9  Enquanto o tempo de simulação não expirar.
10   /* Cria processos quando um instante de nascimento pré-determinado
        chega */
11   Se tempo_nascimento = 0.
12     Crie o processo p_j.
13     tempo_execucao = tempo_proxima_execucao( tempo_medio_execucao ).
14     p_j.definir_tempo_execucao( tempo_execucao ).
15     Envie p_j ao chip.
16     tempo_nascimento = tempo_proximo_nascimento( tempo_medio_nascimento
           ).
17   FimSe.
18
19   /* Destrói processos quando este exaure todos os seus quantums, isto
        é, quando o processo exaure todo seu tempo de execução */
20   Se tempo_morte = 0 e processo_em_execucao <> NENHUM.
21     Destrua processo_em_execucao.
22     processo_em_execucao = NENHUM.
23     tempo_morte = INFINITO.
24     tempo_chegada = INFINITO.
25     tempo_quantum = INFINITO.
26   FimSe.
27
28   /* Libera o processo, isto é, remove o processo do processador e o
        envia ao chip */
29   Se tempo_quantum = 0  processo_em_execucao <> NENHUM.
30     tempo_execucao = processo_em_execucao.obter_tempo_execucao( ).
31     tempo_execucao = tempo_execucao - parametros.obter_quantum( ).
32     processo_em_execucao.definir_tempo_execucao( tempo_execucao ).
33     Envie processo_em_execucao ao chip.
```

```
34      processo_em_execucao = NENHUM.
35      tempo_morte = INFINITO.
36      tempo_chegada = INFINITO.
37      tempo_quantum = INFINITO.
38    FimSe.
39
40    /* Adquire um processo, solicitado ao chip previamente */
41    Se processo_em_execucao = NENHUM  e tempo_chegada = 0 e
          requisicao_pendente = VERDADEIRO.
42      Se o canal de entrada de processos do processador (cep) possuir um
            processo ou negação.
43        Obtenha o processo ou negação pn_j de cep.
44        requisicao_pendente = FALSO.
45        tempo_chegada = INFINITO.
46        Se  pn_j  é um processo.
47          tempo_morte =  p_nj  .obter_tempo_execucao( ).
48          tempo_quantum = parametros.obter_quantum( ).
49          processo_em_execucao = pn_j.
50        FimSe.
51      Senão.
52        tempo_morte = INFINITO.
53        tempo_quantum = INFINITO.
54        processo_em_execucao = NENHUM.
55        tempo_chegada = 1.
56      FimSe.
57    FimSe.
58
59    /* Solicita um processo ao chip */
60    Se processo_em_execucao = NENHUM  e tempo_chegada = INFINITO e
          requisicao_pendente = FALSO.
61      tempo_chegada = 2.
62      requisicao_pendente = VERDADEIRO.
63      Envie uma requisição de processo ao chip.
64    FimSe.
65
66    /* Determina o intervalo de tempo até o próximo evento */
67    tempo_a_transcorrer = MINIMO( tempo_chegada,  tempo_morte,
          tempo_quantum, tempo_nascimento ).
68
69    /* Aguarda até o instante de tempo apropriado */
70    Aguarde tempo_a_transcorrer unidades de tempo.
71
72    /* Atualiza o tempo até os próximos eventos */
73    tempo_morte = tempo_morte - tempo_a_transcorrer.
74    tempo_chegada = tempo_chegada - tempo_a_transcorrer.
75    tempo_quantum = tempo_quantum - tempo_a_transcorrer.
76    tempo_nascimento = tempo_nascimento  - tempo_a_transcorrer.
```

```
77
78 FimEnquanto.
```

Listagem 4.2: Algoritmo de Operação do Processador

Resumidamente, o algoritmo de operação do processador realiza os seguintes passos:

1. Se for o momento apropriado, cria um novo processo (com um tempo de execução associado) a ser enviado ao chip e determina o momento do próximo nascimento (linhas 10-17)

2. Se o processo em execução esgotou o tempo associado a ele, termina o processo (linhas 19-26)

3. Se o processo em execução exauriu seu *quantum*, decrementa o tempo de execução associado e envia o processo para o *chip* (linhas 28-38)

4. Se necessário, recebe um processo solicitado ao *chip*, atribui-lhe um *quantum* e inicia sua execução (linhas 40-57)

5. Se necessário, solicita um processo ao *chip* caso nenhum esteja em execução (linhas 59-64)

6. Aguarda até o próximo evento (nascimento, término, chegada em potencial ou esgotamento de *quantum* de processo) (linhas 66-76)

Como pode ser observado, a granularidade na passagem de tempo é maior no processador do que no *chip*. Enquanto o *chip* opera em um intervalo fixo de tempo, o processador avança no tempo entre ocorrências de eventos relevante a seu funcionamento.

Deve-se notar, contudo, que nesse modelo só há um processo em execução no processador em um dado instante de tempo. Além disso, o tempo de execução do processo alocado ao processador é decrementado à medida que o tempo passa, permitindo que esse processo eventualmente termine.

Metodologia de Obtenção dos Tempos de Nascimento e Execução

A criação de processos no processador obedece a distribuição de probabilidade exponencial, e é realizada pela aplicação do método da transformada inversa. Este método permite que valores na imagem de uma função de distribuição cumulativa (*cumulative distribution function*) sejam mapeados nos valores correspondentes no domínio da função. Dada uma variável aleatória X, sua função de distribuição cumulativa é definida como a probabilidade da variável assumir um valor menor ou igual a x, em termos formais $F_X(x) = P[X \leq x]$.

A distribuição exponencial é comumente utilizada para modelar o intervalo de tempo entre eventos em um sistema. Ela é frequentemente empregada para modelar chegadas de pacotes em uma rede, ingresso de processos em um sistema, tempo entre falha de componentes e assim por diante. Sua função de distribuição cumulativa é dada pela seguinte formula:

$$F(x) = \begin{cases} 1 - e^{-\lambda x}, & se\ 0 \leq x < \infty \\ 0, & caso\ contrário \end{cases}$$

O termo λ representa a taxa média de ocorrência do evento sendo modelado. É fácil perceber que a imagem desta função está restrita ao intervalo $0 \leq x \leq 1$ por tratar-se de uma distribuição de probabilidade. Então, pela aplicação do método da transformada inversa, obtemos:

$$U = F(x)$$

$$U = 1 - e^{-\lambda x}$$

$$e^{-\lambda x} = 1 - U$$

$$-\lambda x = \ln(1 - U)$$

$$x = -\frac{1}{\lambda} \ln(1 - U)$$

Dessa forma, sendo então $0 \leq U \leq 1$, um número sorteado aleatoriamente, podemos mapeá-lo no intervalo de tempo, o termo x da expressão, na qual ocorrerá o próximo evento modelado.

Portanto, com o uso desse método podemos gerar aleatoriamente intervalos de tempo que determinam o momento de nascimentos de novos processos, bem como o tempo de execução associado a eles, que em ambos os casos obedecem a distribuição exponencial.

4.5 Rede de Escalonamento *Intrachip* em Operação

Nesta seção apresentaremos por meio de figuras o comportamento da rede de escalonamento durante sua operação. As figuras 4.4-4.19 demonstram passo a passo uma situação de execução hipotética em uma rede de escalonamento. Cada figura exibe o estado do conjunto de *chips* em um dado instante de tempo e uma breve descrição sobre a situação de alguns elementos chaves naquela etapa.

A rede adota a mesma topologia descrita na figura 4.2. Entretanto, visando simplificar a demonstração, realizamos algumas alterações na estrutura empregada

nesta seção. Em particular, os dois canais em cada processador e as duas portas nos *chips* que se conectam, não representam de fato dois componentes, mas sim uma porta e um canal com capacidade dupla, isto é, que podem receber e enviar até dois processos por etapa. Além disso, a capacidade fila interna dos *chips* foi limitada a oito processos.

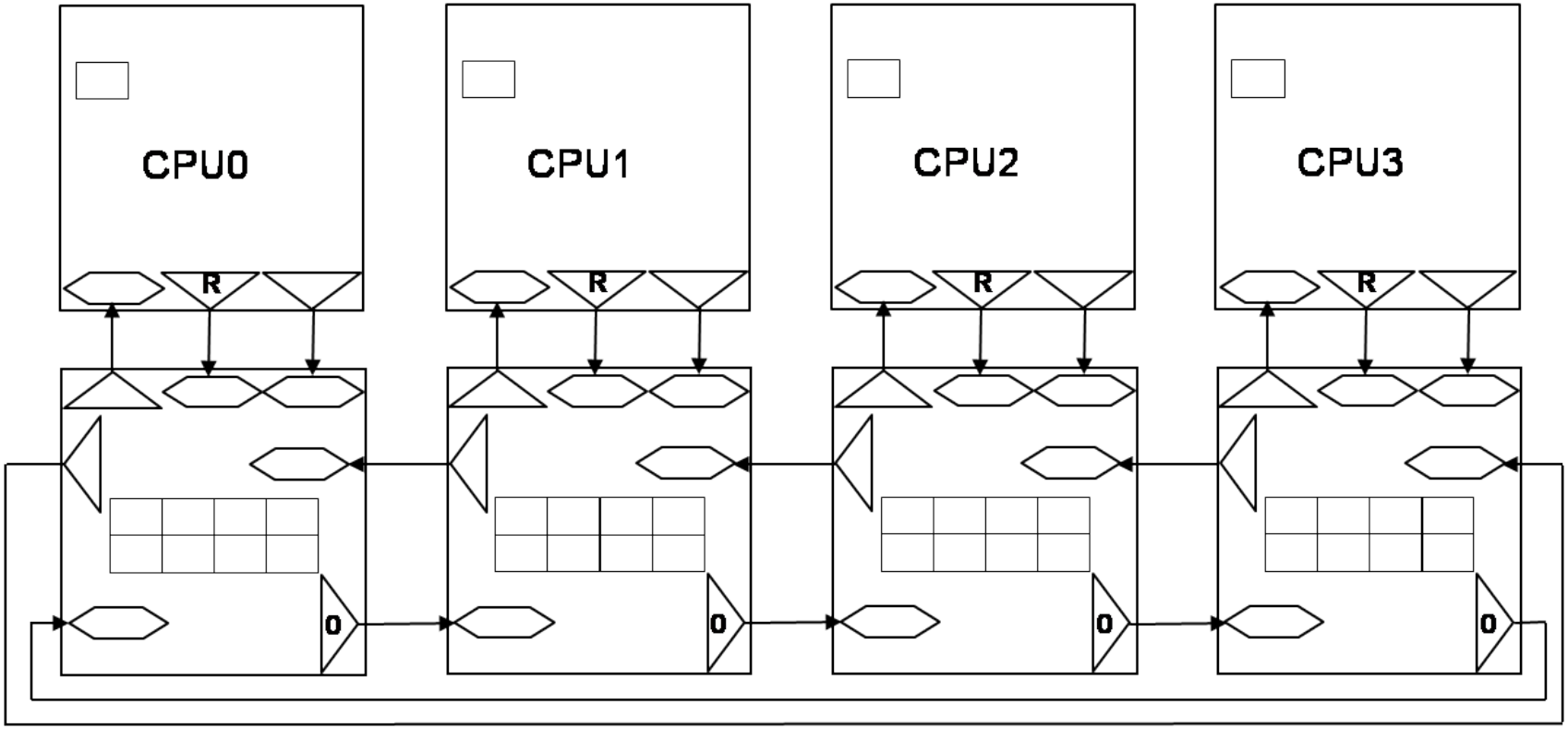

Figura 4.4: Rede de Escalonamento em Operação: Tempo 0

Tempo 0: As filas de processos de todos os processadores e *chips* de escalonamento estão vazias. Como não há processos em execução nos processadores, cada processador faz uma solicitação de processo a seu *chip* associado. Além disso, cada *chip* informa seu número de processos a seu vizinho à direita.

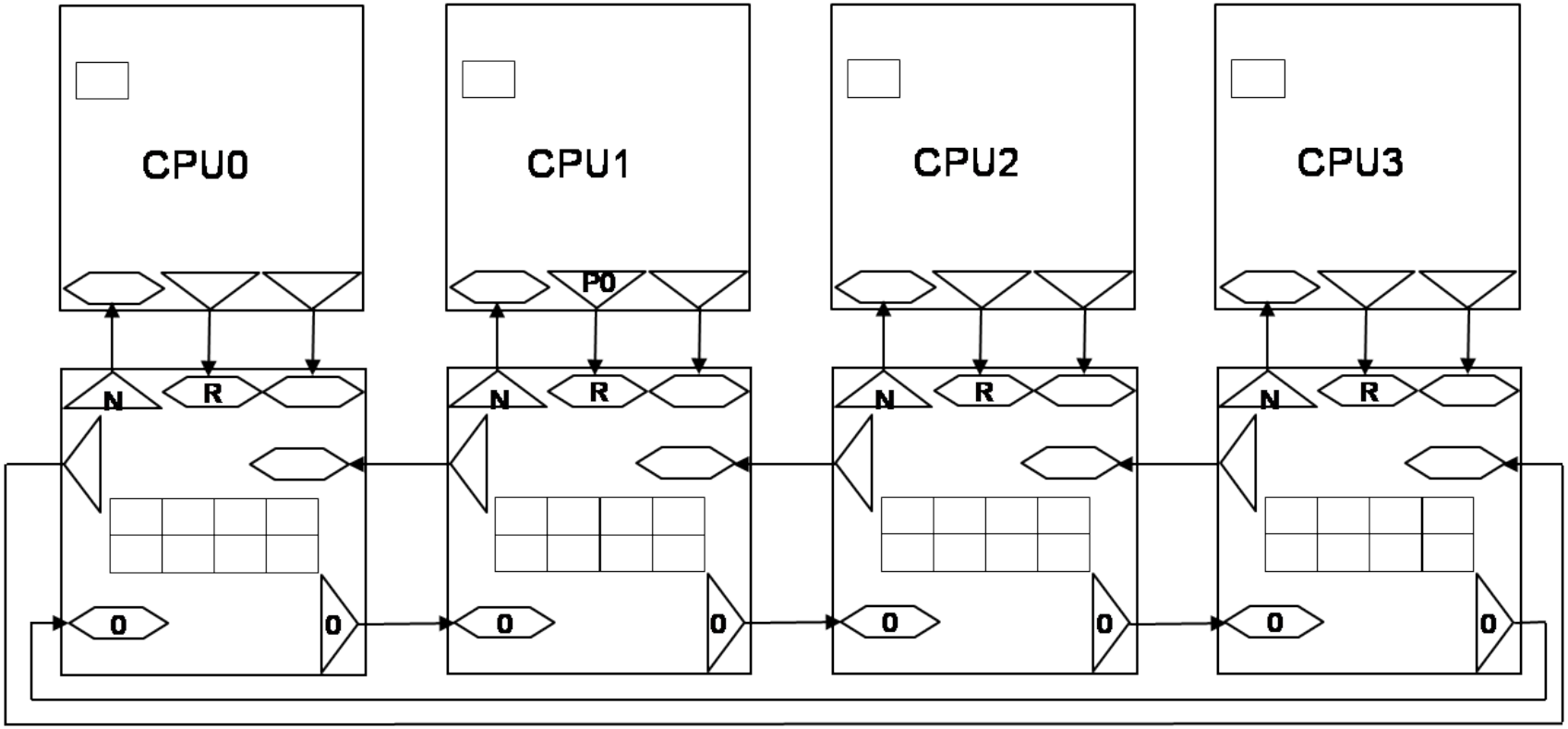

Figura 4.5: Rede de Escalonamento em Operação: Tempo 1

Tempo 1: Cada *chip* de escalonamento recebe um pedido de processo, mas como não há processos sob sua administração, ele responde a seu processador com uma negação. Ainda nesta etapa, o processo 0 é criado no processador 1 e, enviado ao *chip* 1. A informação sobre número de processos enviada na etapa anterior é recebida por cada *vizinho*. Cada *chip* envia novamente seu número de processos atual.

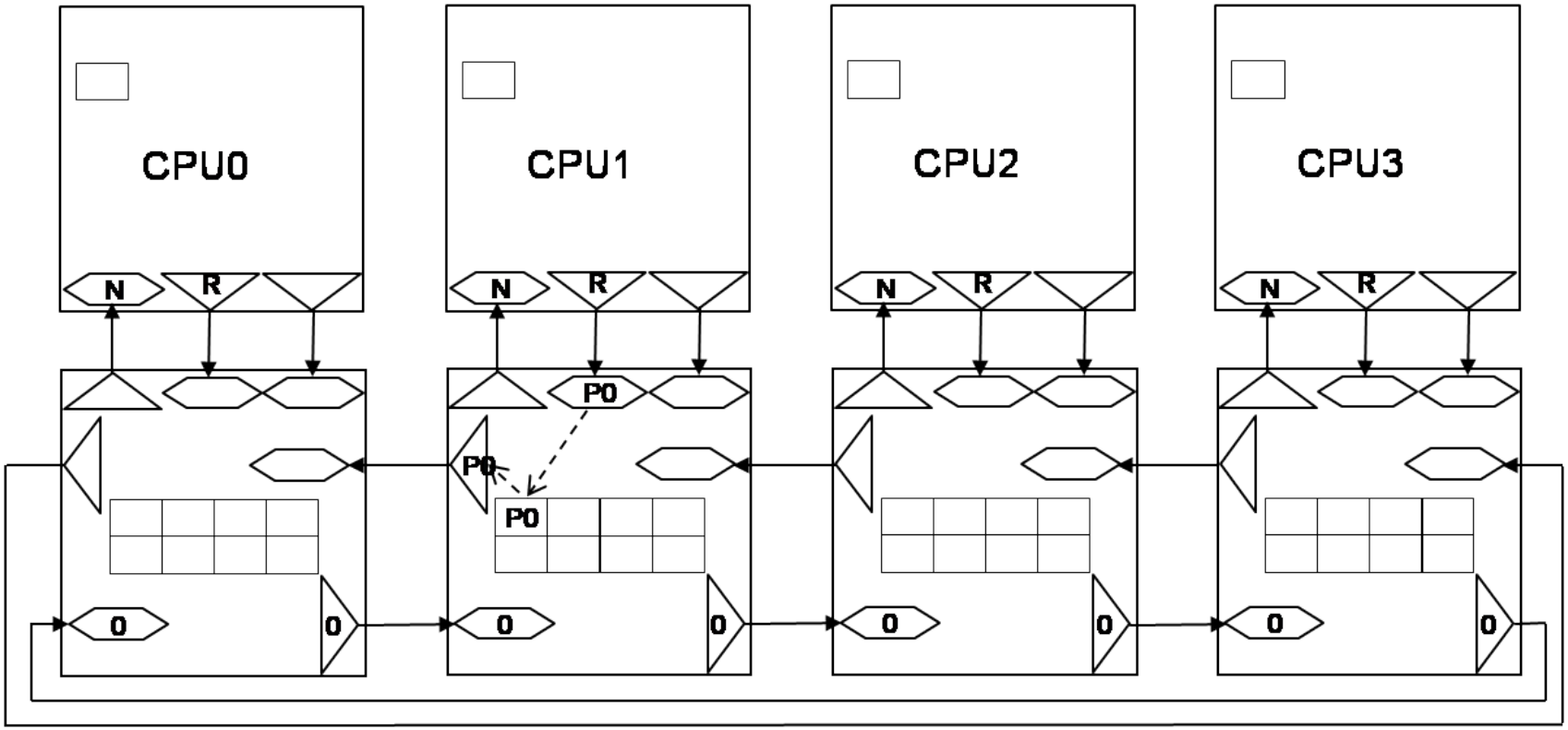

Figura 4.6: Rede de Escalonamento em Operação: Tempo 2

Tempo 2: Cada processador recebe uma negação a seu pedido de processo e responde com um novo pedido. Os *chips* recebem o número de processos em seu vizinho, enviado no passo anterior. O processo que chega ao *chip* 1, é movido para sua fila de processos, e então, enviado ao *chip* 0, pois o número de processos deste (0) é menor que o número de processos daquele (1). Como usual, cada chip envia seu número de processos.

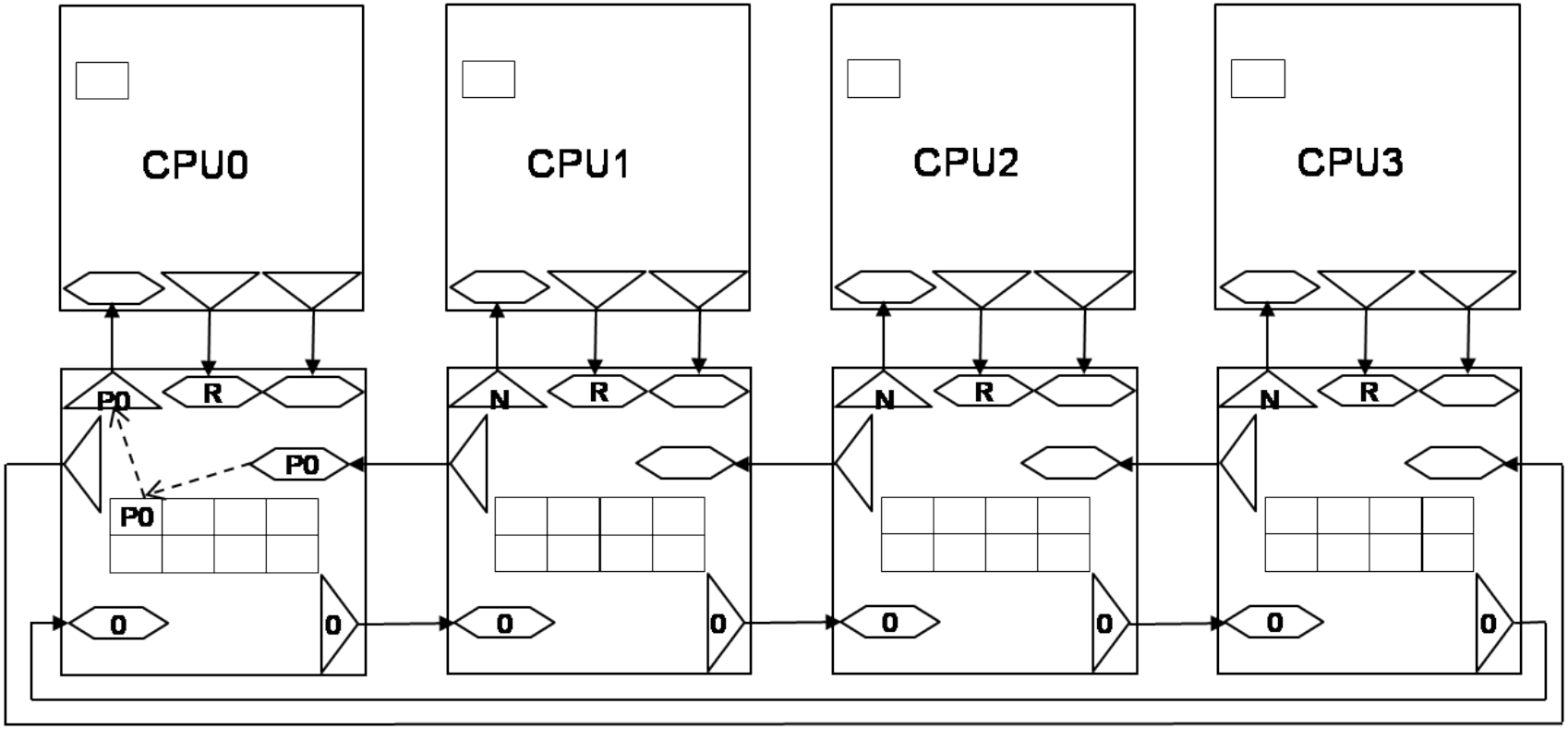

Figura 4.7: Rede de Escalonamento em Operação: Tempo 3

Tempo 3: As solicitações de processos feitas no passo anterior chegam aos *chips*. Todos os *chips*, com exceção do *chip* 0, respondem a seu processador associado com uma negação. O *chip* 0, que acabou de receber o processo 0, responde a seu processador com este processo. Informações sobre número de processos continuam a ser enviadas.

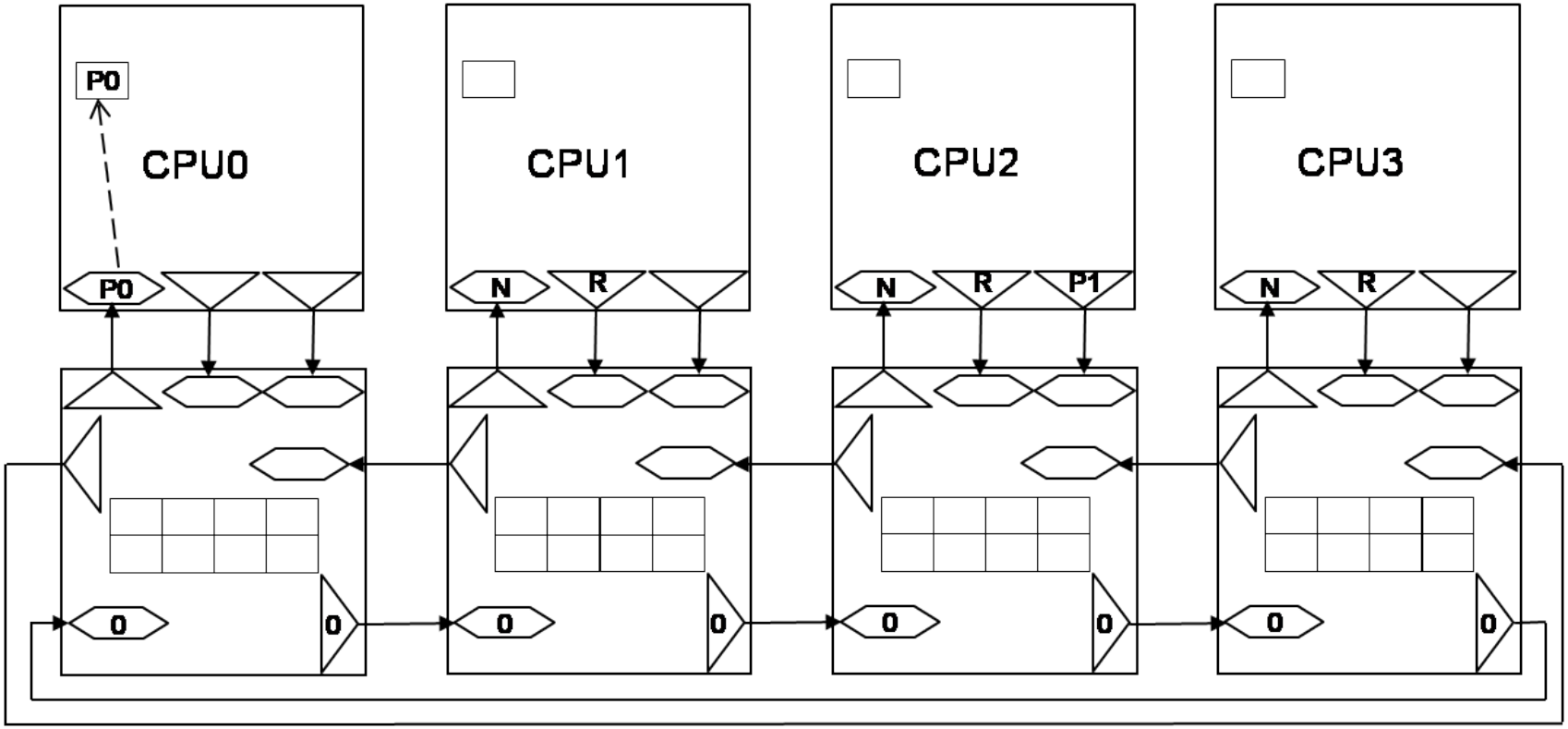

Figura 4.8: Rede de Escalonamento em Operação: Tempo 4

Tempo 4: Os processadores, exceto o processador 0, fazem um novo pedido de processo em virtude da negação recebida. O processador 0 recebe e põe em execução o processo vindo do *chip* 0. O processo 1 surge no processador 2 e, é enviado ao *chip* 2. A informação sobre número de processos em cada *chip* é reenviada.

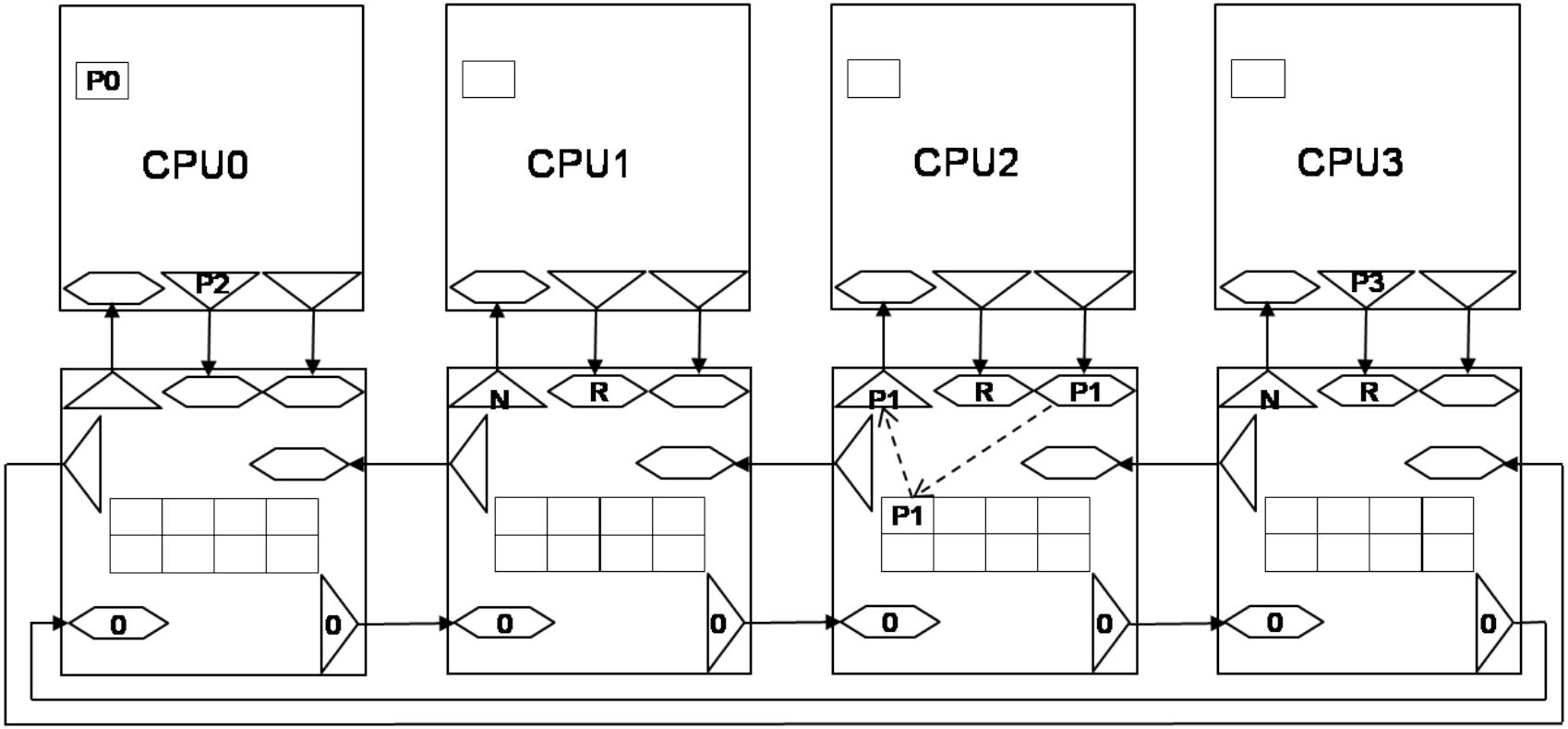

Figura 4.9: Rede de Escalonamento em Operação: Tempo 5

Tempo 5: Os *chips* 1 e 3 respondem com uma negação ao pedido de seus processadores. O processador 0 segue executando o processo 0. O *chip* 2 responde ao processador 2 com o processo 1. Os processos 2 e 3 surgem respectivamente nos processadores 0 e 3 e, são enviados aos *chips* associados.

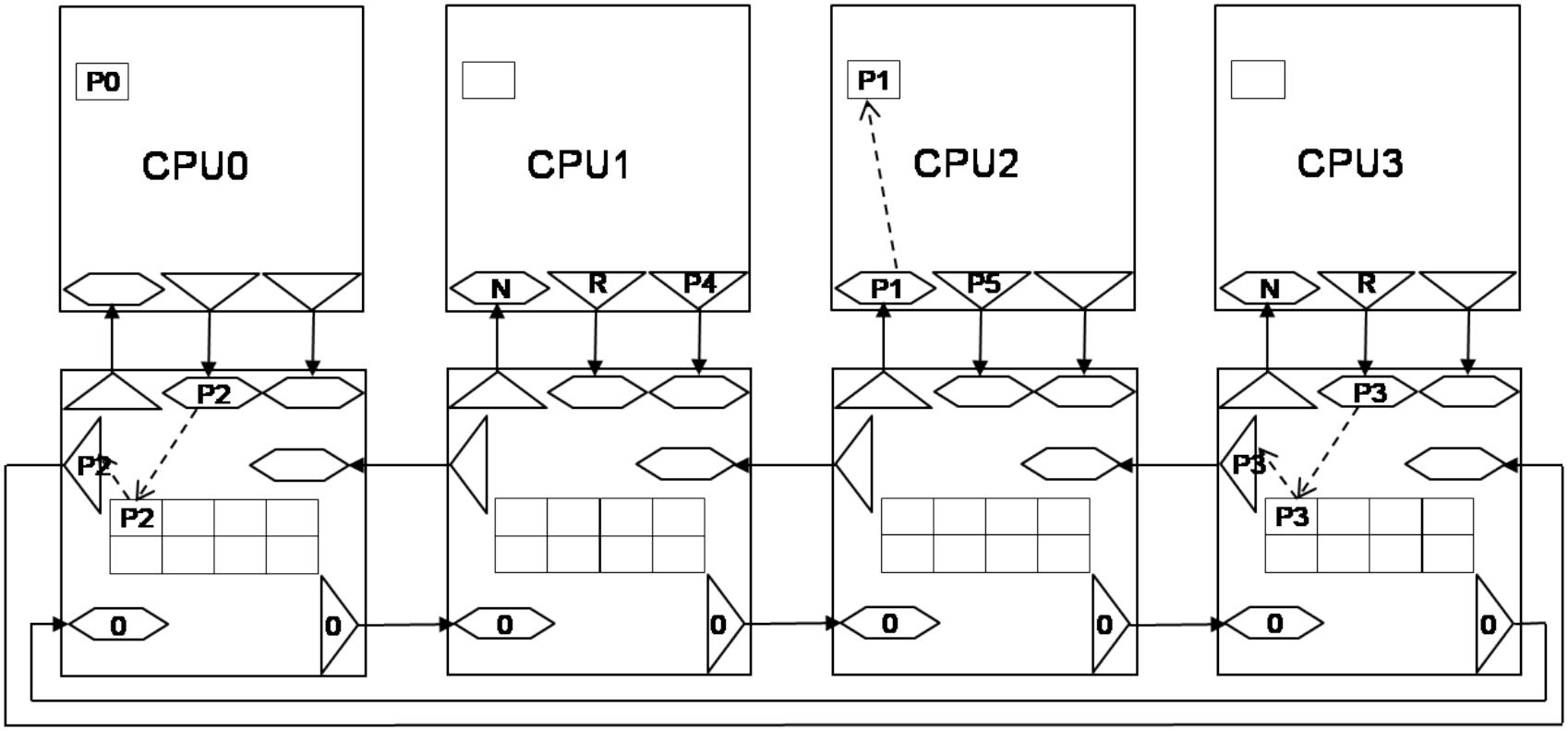

Figura 4.10: Rede de Escalonamento em Operação: Tempo 6

Tempo 6: O processador 0 segue executando o processo 0. O processadores 1 e 3 recebem a negação de seus *chips* associados e refazem seus pedidos de processo. O processador 2 recebe o processo 1 e o põe em execução. Os processos 4 e 5 surgem, respectivamente, nos processadores 1 e 2 e são enviados ao *chips* correspondentes. Os *chips* 0 e 3 recebem respectivamente os processos 2 e 3, repassando-os a seus *chips* vizinhos, pois o número de processos destes é menor.

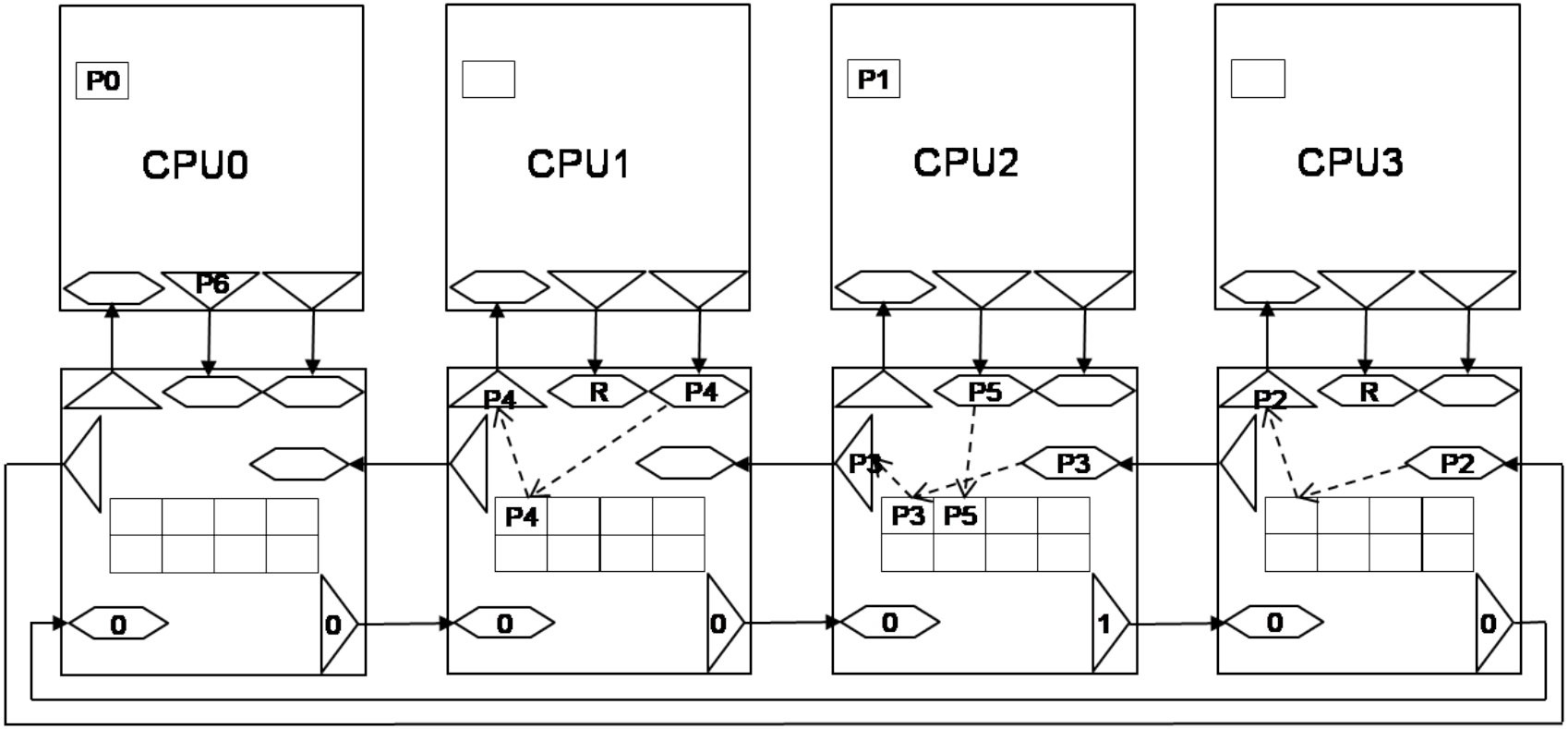

Figura 4.11: Rede de Escalonamento em Operação: Tempo 7

Tempo 7: Os processadores 0 e 2 continuam executando os processos que lhes foram atribuídos. O *chip* 1 responde a seu processador com o processo 4. O *chip* 2 envia o processo 3 ao *chip* 1 porque seu número de processos (2) é maior que o número de processos deste (0). O *chip* 3 responde a seu processador com o processo 2. O processo 6 surge no processador 0.

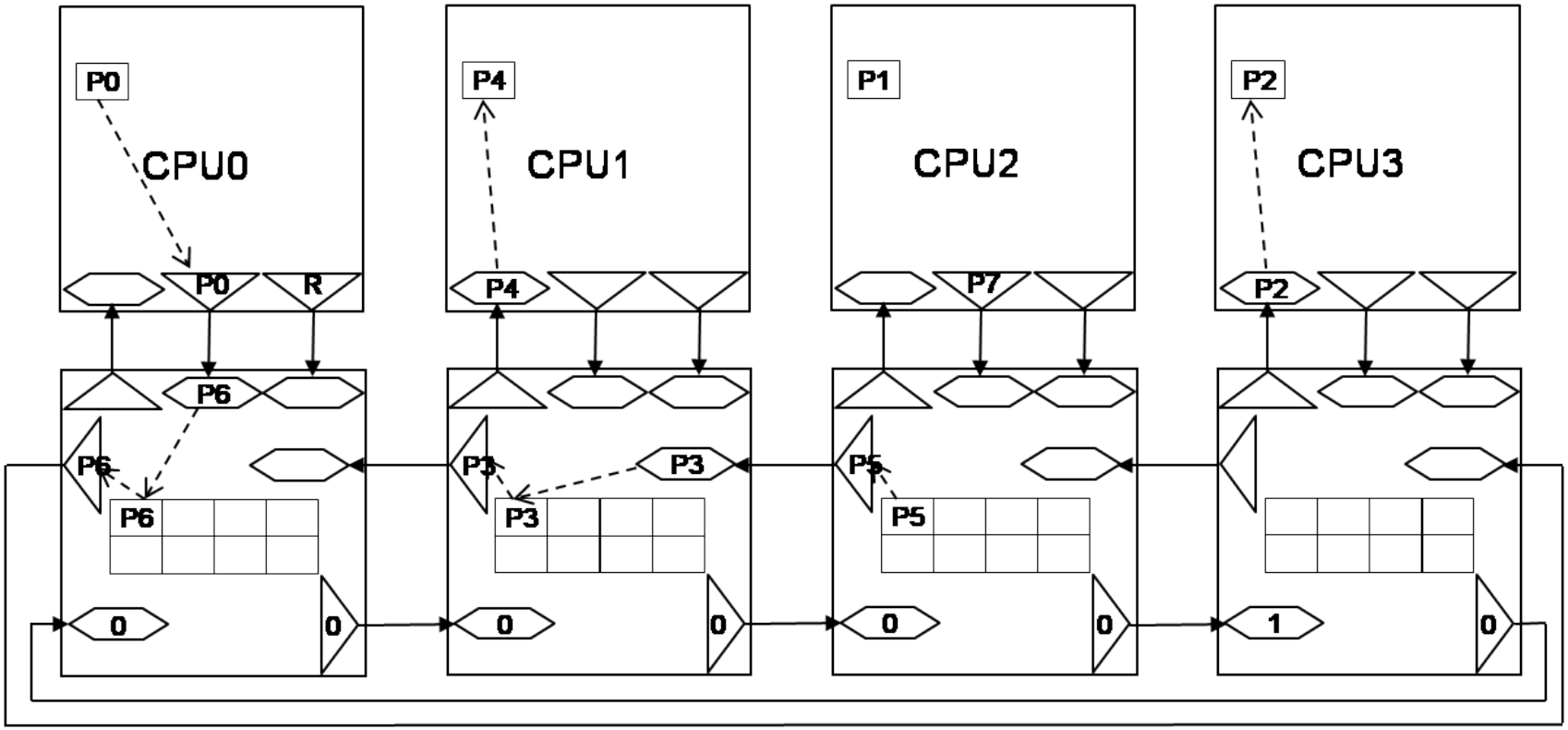

Figura 4.12: Rede de Escalonamento em Operação: Tempo 8

Tempo 8: O processo 0 que executava no processador 0 esgota seu *quantum* e é enviado ao *chip* 0. Os *chips* 0, 1 e 2 encaminham processos a seus vizinhos porque o número de processos informado por estes no passo anterior (0) é menor que o número de processos naqueles *chips* (1). Os processadores 1 e 3 recebem processos que foram solicitados a seus *chips* associados. O processo 7 surge no processador 2.

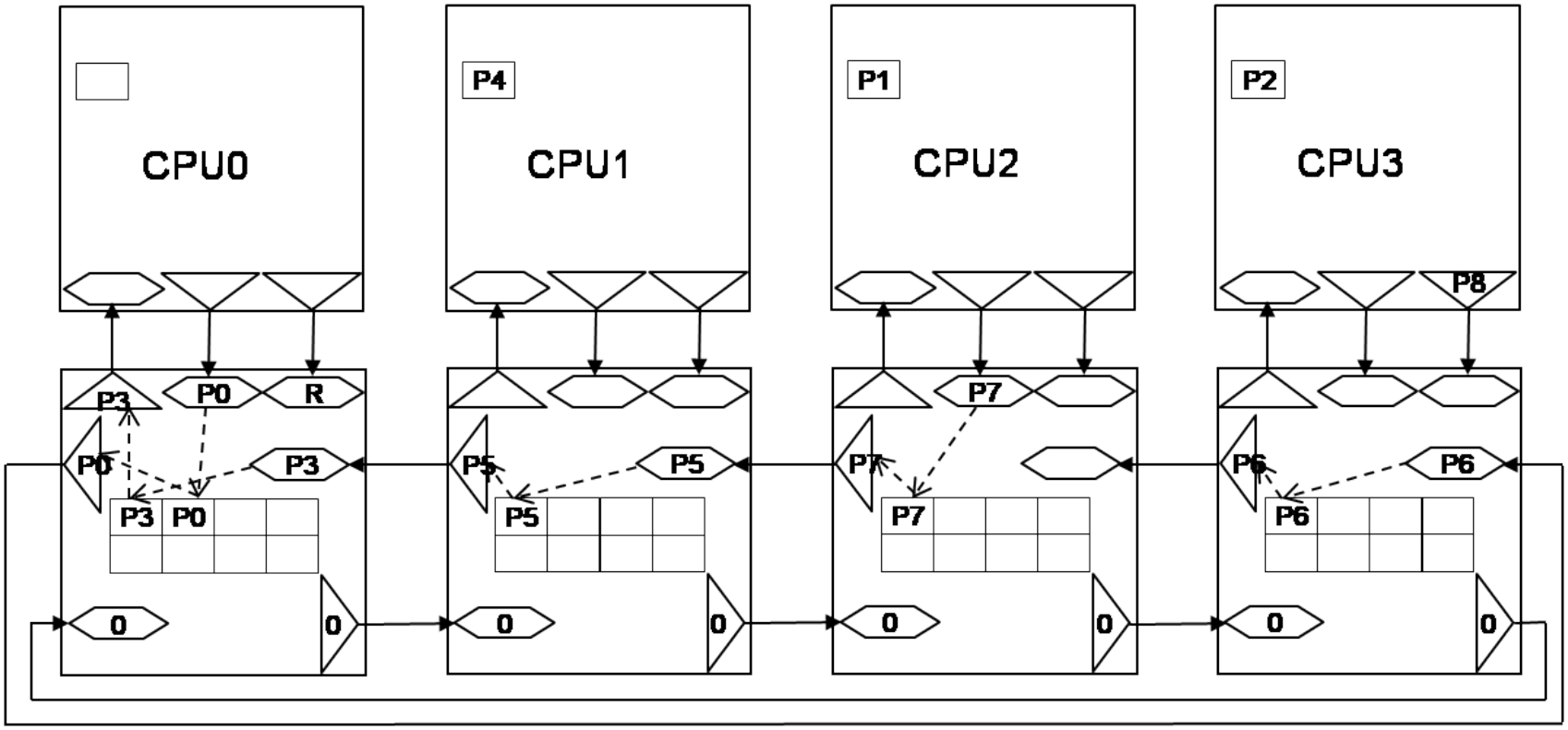

Figura 4.13: Rede de Escalonamento em Operação: Tempo 9

Tempo 9: O *chip* 0 envia o processo 3 ao processador 0. Os demais processadores seguem executando os processos sob seu controle. Cada *chip* envia um processo a seu vizinho, pois o número de processos informado por estes no passo anterior (0) é menor que o número de processos nos *chips* correspondentes (1). O processo 8 surge no processador 3.

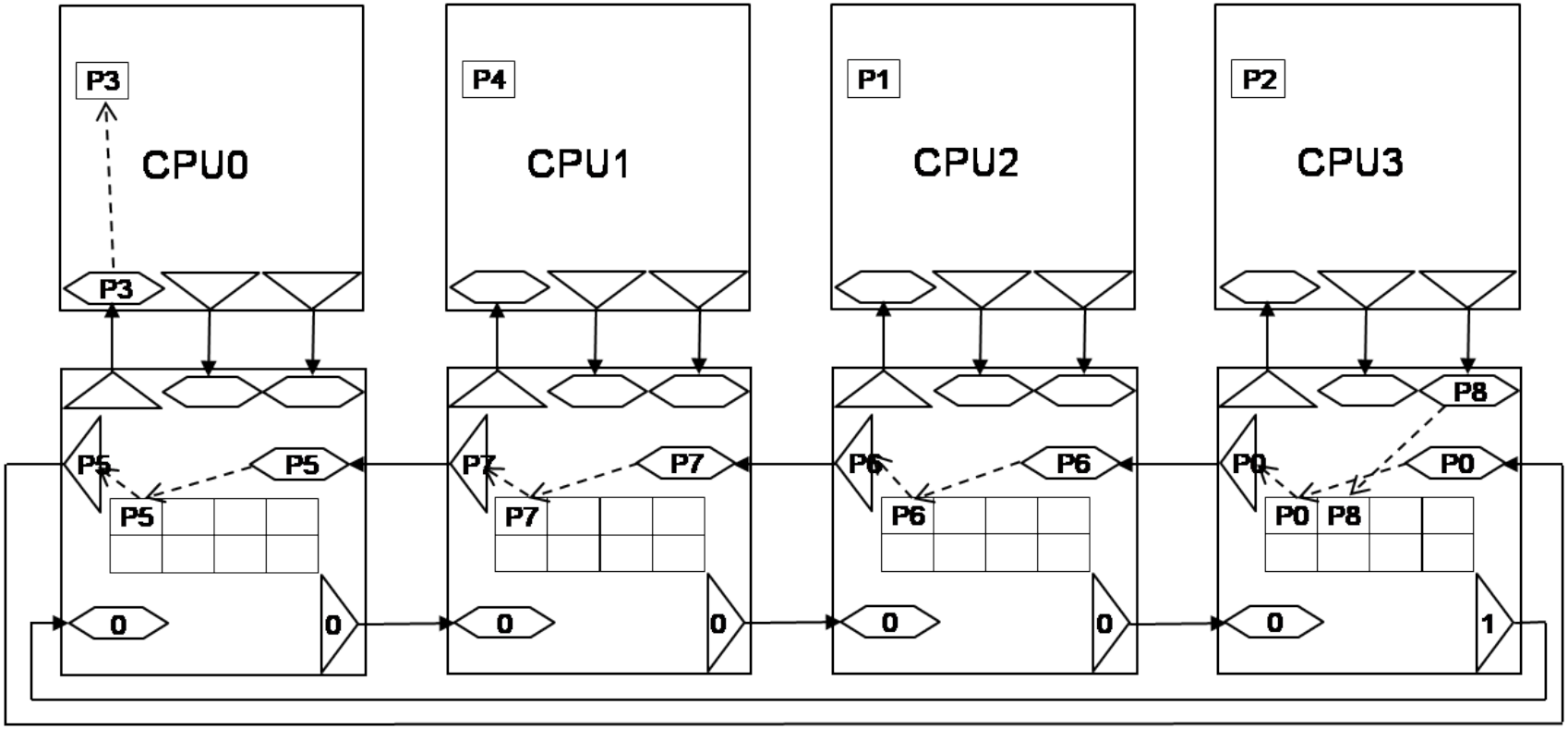

Figura 4.14: Rede de Escalonamento em Operação: Tempo 10

Tempo 10: O processador 0 recebe o processo 3. Os demais processadores seguem executando os processos sob sua administração. Cada *chip* envia um processo a seu vizinho pois o número de processos deste no passo anterior é zero. O processo 8 permanece na fila de processos do *chip* 3, cujo tamanho passa a ser 1.

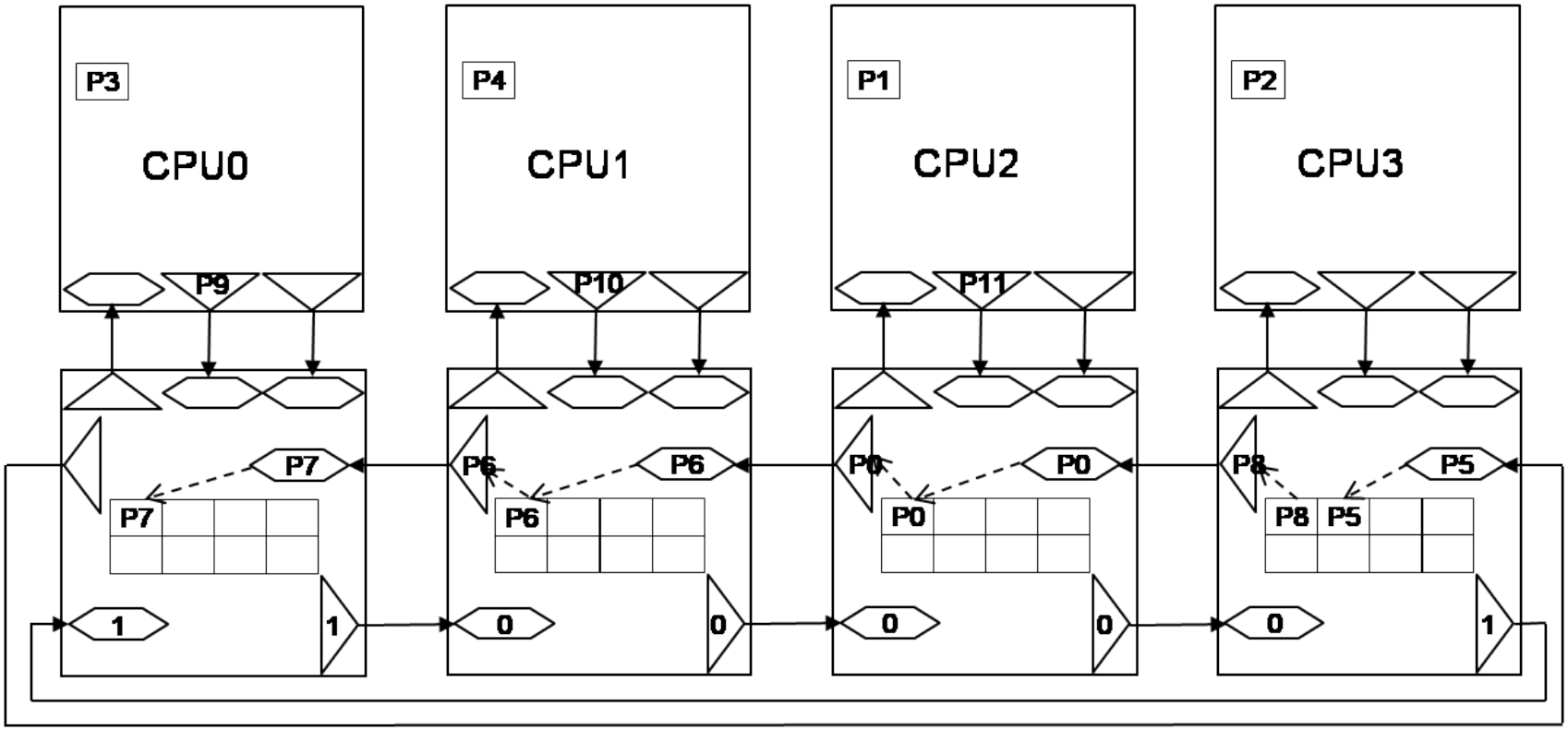

Figura 4.15: Rede de Escalonamento em Operação: Tempo 11

Tempo 11: Cada processador segue executando o processo que lhe foi atribuído. Os processos 9, 10 e 11 surgem nos processadores 0, 1 e 2, respectivamente. Cada *chip*, exceto o *chip* 0, envia um processo a seu vizinho, pois o número de processo neste, no final da etapa anterior, é menor que o número de processos no *chip* no momento da decisão. O *chip* 3 continua possuindo um processo a mais que os demais *chips*.

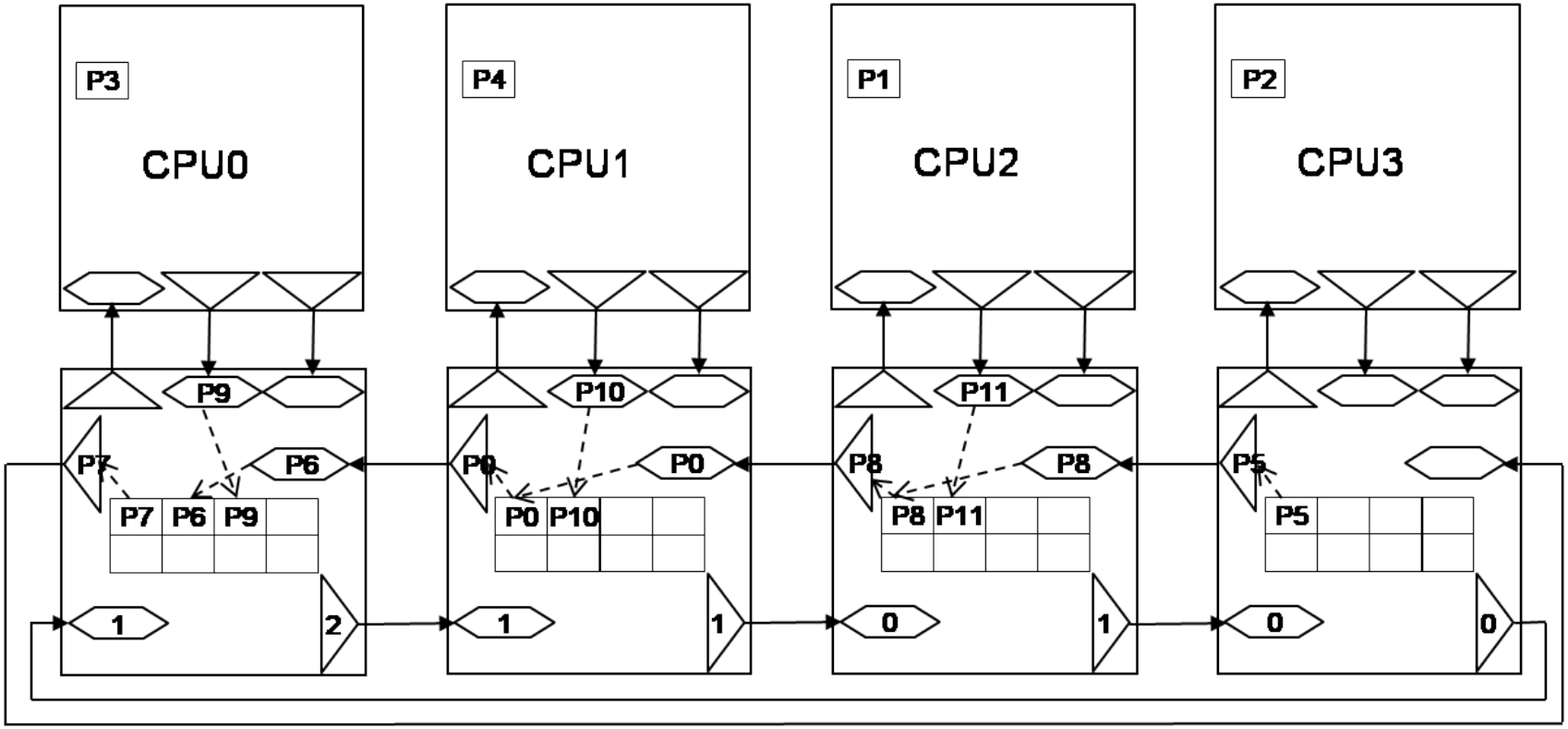

Figura 4.16: Rede de Escalonamento em Operação: Tempo 12

Tempo 12: Cada processador segue executando o processo que lhe foi atribuído. o *chip* 0 envia um processo ao *chip* 3 pois este havia informado possuir apenas 1 processo no passo anterior, enquanto o *chip* 0 possuía 3 processos no momento da decisão. Os demais *chips* tomam decisões semelhantes.

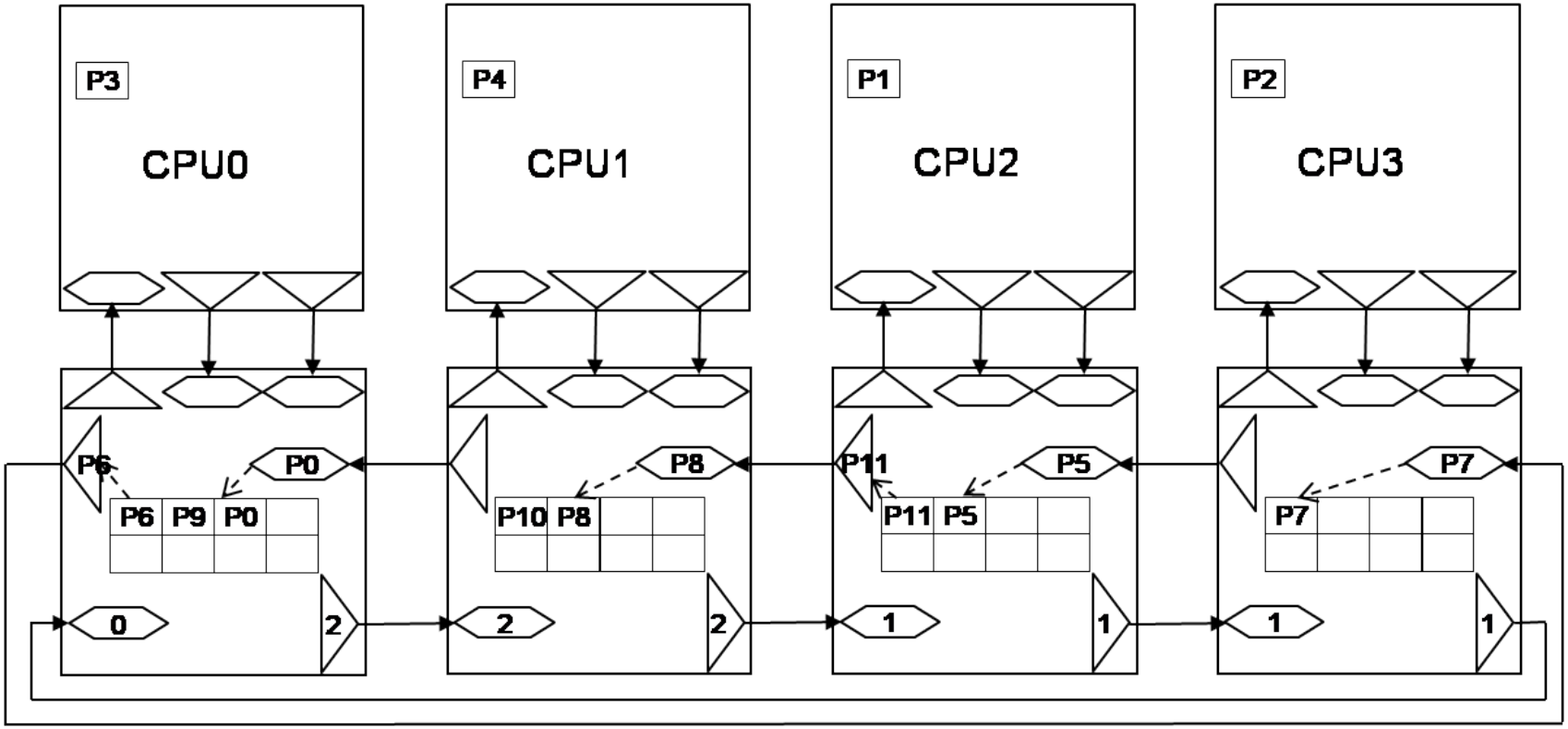

Figura 4.17: Rede de Escalonamento em Operação: Tempo 13

Tempo 13: A execução de processo continua nos processadores. Apenas os *chips* 0 e 2 decidem enviar processos a seus vizinhos em virtude de seu número de processos.

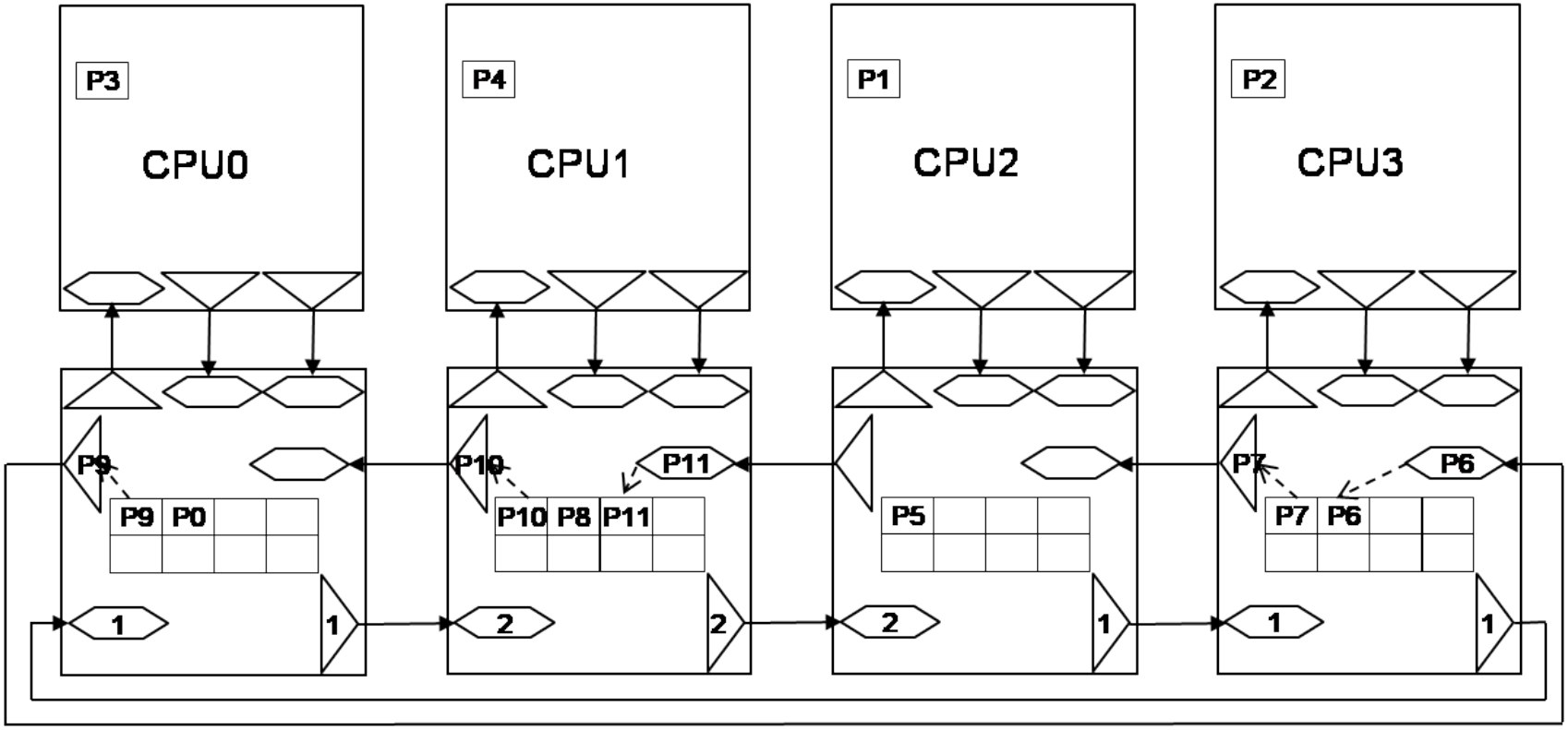

Figura 4.18: Rede de Escalonamento em Operação: Tempo 14

Tempo 14: Os processadores seguem executando seus processos. Todos os *chips*, com exceção do *chip* 2, enviam processos a seus vizinhos.

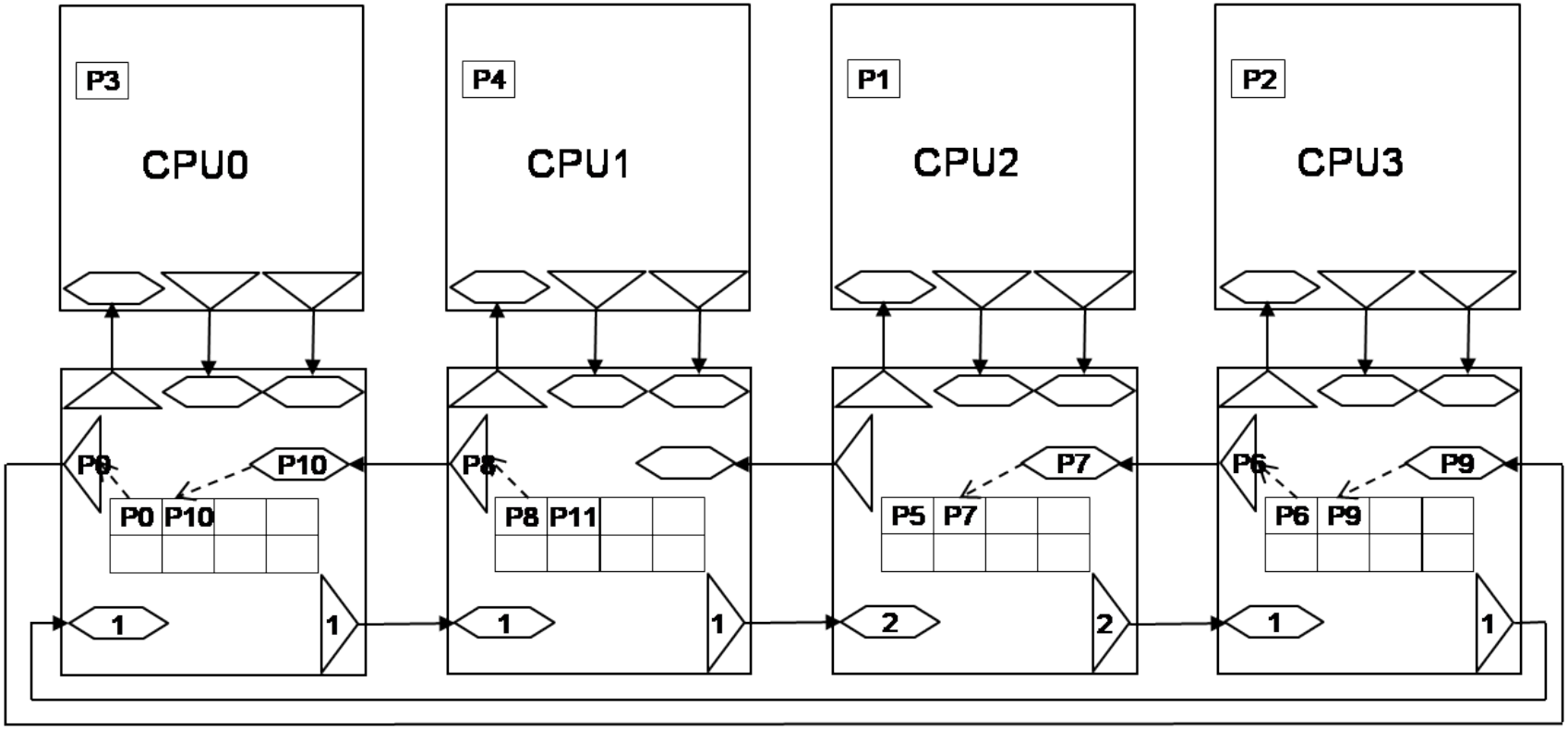

Figura 4.19: Rede de Escalonamento em Operação: Tempo 15

Tempo 15: Os processadores seguem executando seus processos. Novamente, todos os *chips*, com exceção do *chip* 2, enviam processos a seus vizinhos.

Capítulo 5

Resultados & Medidas de Interesse

Neste capítulo vamos apresentar os resultados obtidos através de simulação da arquitetura delineada no capítulo anterior. Em princípio vamos nos ater ao comportamento da rede de escalonamento em virtude de características inerentes ao modelo. A seguir, iremos contrastar o desempenho da rede de escalonamento com modelos markovianos de filas. E finalmente, faremos uma rápida comparação entre o mecanismo *intrachip* e o escalonamento realizado no sistema operacional Linux.

5.1 Implicações do Escalonamento *Intrachip*

Vamos descrever nesse momento as consequências do uso de uma rede de escalonamento de forma geral. A arquitetura emprega será a concebida no capítulo anterior, isto é, uma rede de *chips* de escalonamento conectados entre si por uma topologia específica, com cada *chip* conectado a um processador mestre. Conforme necessário, iremos descrever em cada caso particular a topologia adotada e os parâmetros de simulação relevantes.

5.1.1 Comportamento Elementar do Modelo

A figura 5.1 apresenta um gráfico com o tamanho das filas de oito chips à medida que o tempo de simulação avança. Cada *chip* está conectado a apenas seu vizinho (mais próximo) à esquerda, de forma similar ao que ocorre na figura 4.2. A política alternada foi empregada na simulação. Os tempos médios de nascimento e execução dos processo foram definidos respectivamente em 600 e 590 unidades de tempo. O intervalo de operação dos *chips*, o *quantum* dos processos e o tempo de simulação foram definidos respectivamente em uma, cem e cem mil unidades de tempo, que convencionamos aqui serem milissegundos (ms).

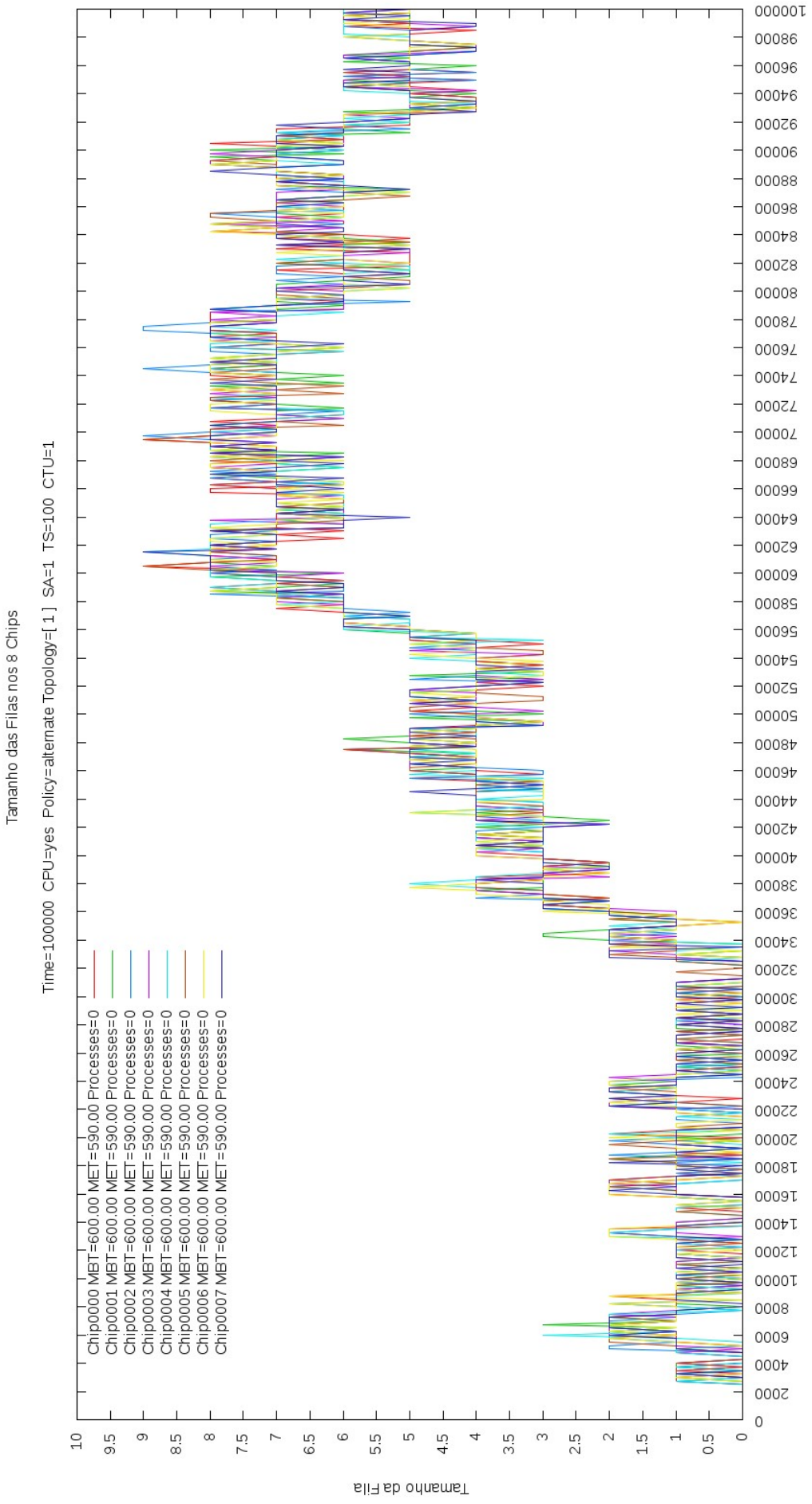

Figura 5.1: Comportamento Elementar da Rede de Escalonamento

Como pode se observado na figura, os tempos médios de nascimento e de execução dos processos são idênticos em todos os processadores. Note que o efeito causado pela operação dos *chips* conforme o algoritmo descrito na seção 4.4.2 é que as filas de processos destes tendem a manter tamanhos semelhantes. A variação existente nos tamanhos de fila deve-se ao ingresso e saída de processos do sistema, bem como a troca de processos com *chips* vizinhos. Entretanto, apesar disso, as filas de processo permanecem concentradas em um intervalo de tamanho razoavelmente delimitado, onde a diferença entre esses tamanhos é normalmente uma ou duas unidades.

A figura 5.2 mostra a distribuição dos tempos de vida para uma execução com oito processadores/*chips* com política alternada e tempos médios de nascimento e execução de respectivamente 600 e 500 unidades de tempo. Como a figura revela, a maior parte dos processos termina em períodos de tempo relativamente curtos. Observe que a grande maioria leva até no máximo 1000 unidades de tempo para executar. Alguns poucos processos levam um tempo superior.

5.1.2 O Efeito de Variação na Política

Na seção anterior examinamos através de um exemplo simples o comportamento geral da rede de escalonamento. Nessa seção vamos abordar o efeito exercido sobre o modelo pelas políticas de seleção de processos descritas na seção 4.3. Note que em função do passo 5 no algoritmo de operação do *chip*, não faz sentido simular variação de política para uma topologia onde os *chips* estejam conectados por uma ligação única. Nesse caso, a existência de apenas um destino de processos e a aplicação daquela restrição fazem com que todas as decisões tomadas pelos chips sejam idênticas independentemente da política empregada. Cabe ressaltar que os tempos médios de nascimento e execução empregados foram respectivamente de 600 e 590 unidades de tempo. O tempo de simulação foi equivalente a 100.000 unidades de tempo.

As figuras 5.3-5.6 apresentam o comportamento de cada política com uma topologia duplamente conectada. Enquanto as figuras 5.7-5.10 mostram o comportamento das políticas quando quatro conexões são permitidas.

Em ambos os casos, a política alternada demonstra o comportamento mais estável, enquanto a política menor revela a maior instabilidade. As políticas randômica e inversamente proporcional apresentam resultados intermediários.

A menor instabilidade da política alternada deve-se ao fato de que esta balanceia melhor o envio de processos. De fato, essa política nunca tenta enviar processos a um mesmo destino de forma consecutiva, havendo outros destinos. Nessa política, o revezamento entre os destinos causa um envio "tardio"de processos, que tem o efeito de fornecer mais tempo para que o estado do *chip*, isto é, seu número de processos

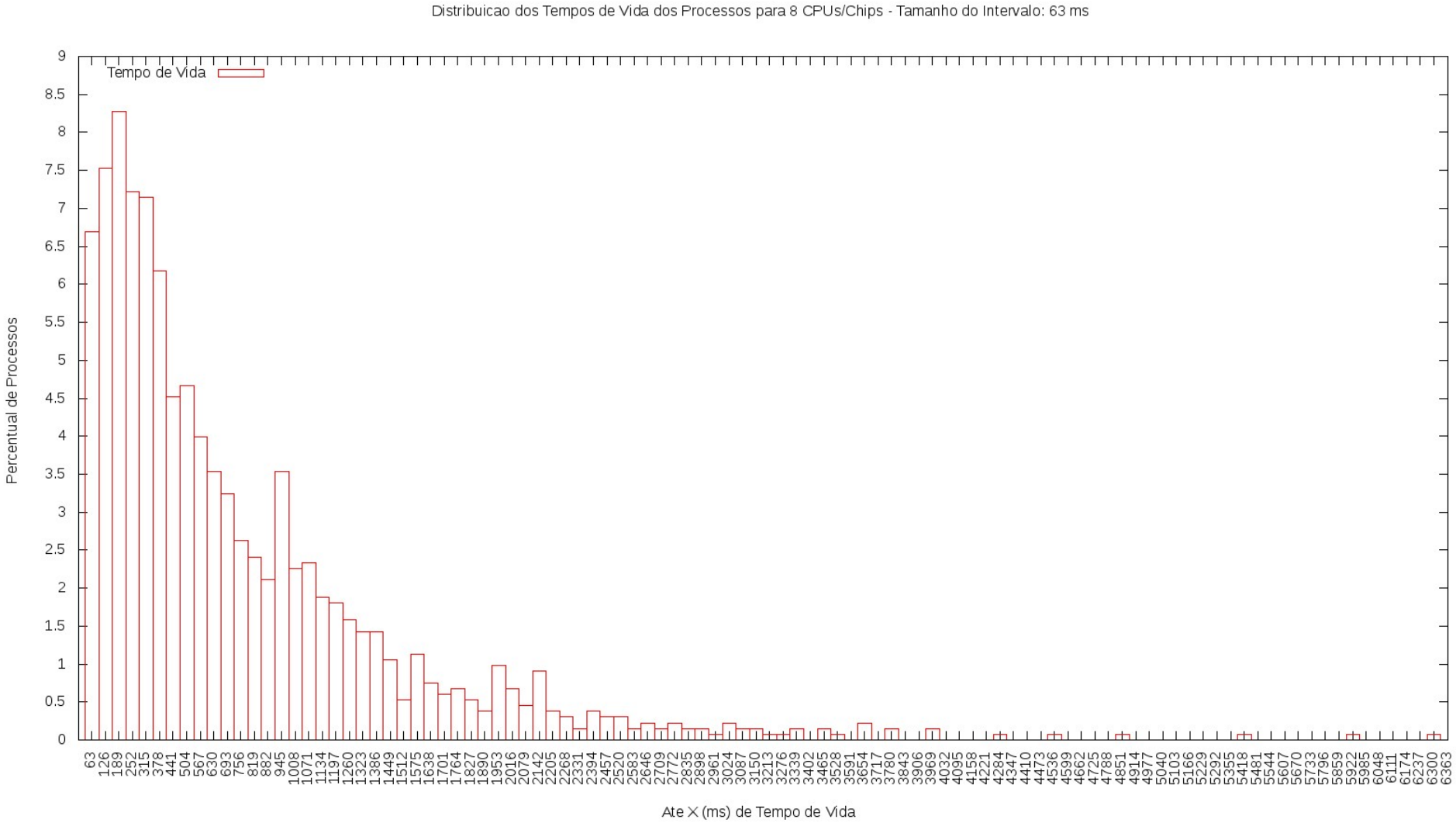

Figura 5.2: Distribuição dos Tempos de Vida (*Turnaround*) para Uma Execução

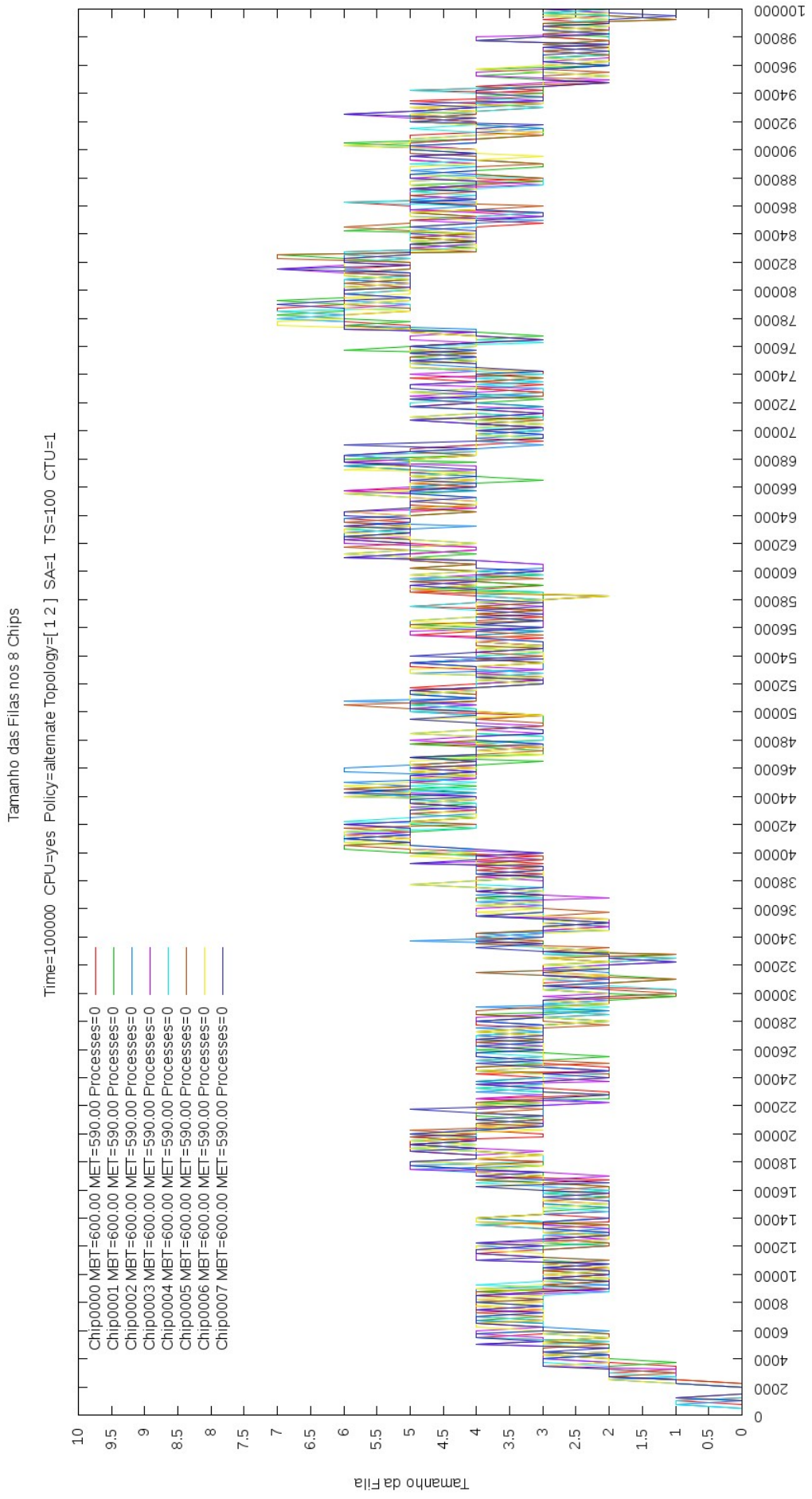

Figura 5.3: Comportamento de Topologia Duplamente Conectada com Política Alternada

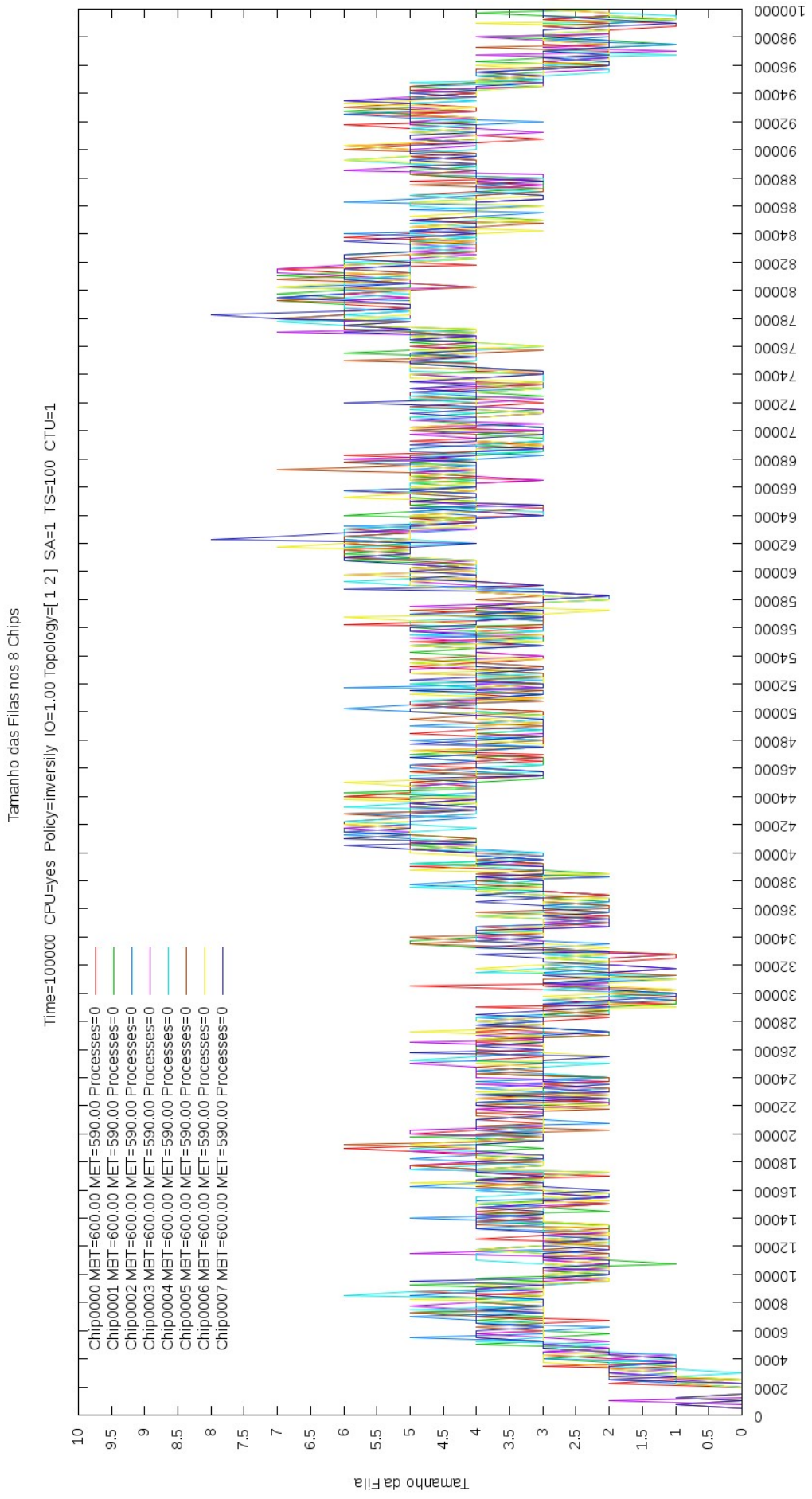

Figura 5.4: Comportamento de Topologia Duplamente Conectada com Política Inversamente ...

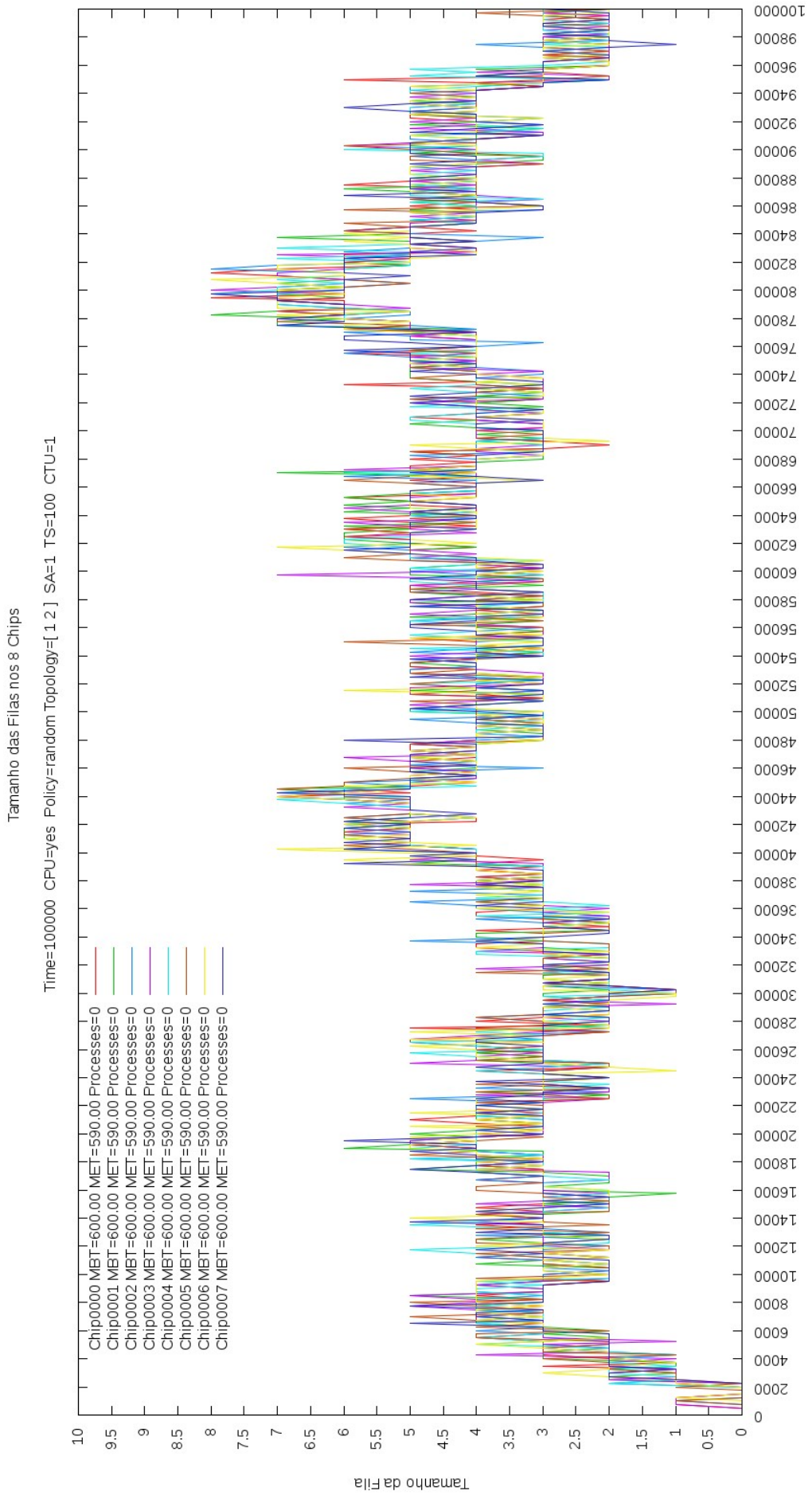

Figura 5.5: Comportamento de Topologia Duplamente Conectada com Política Randômica

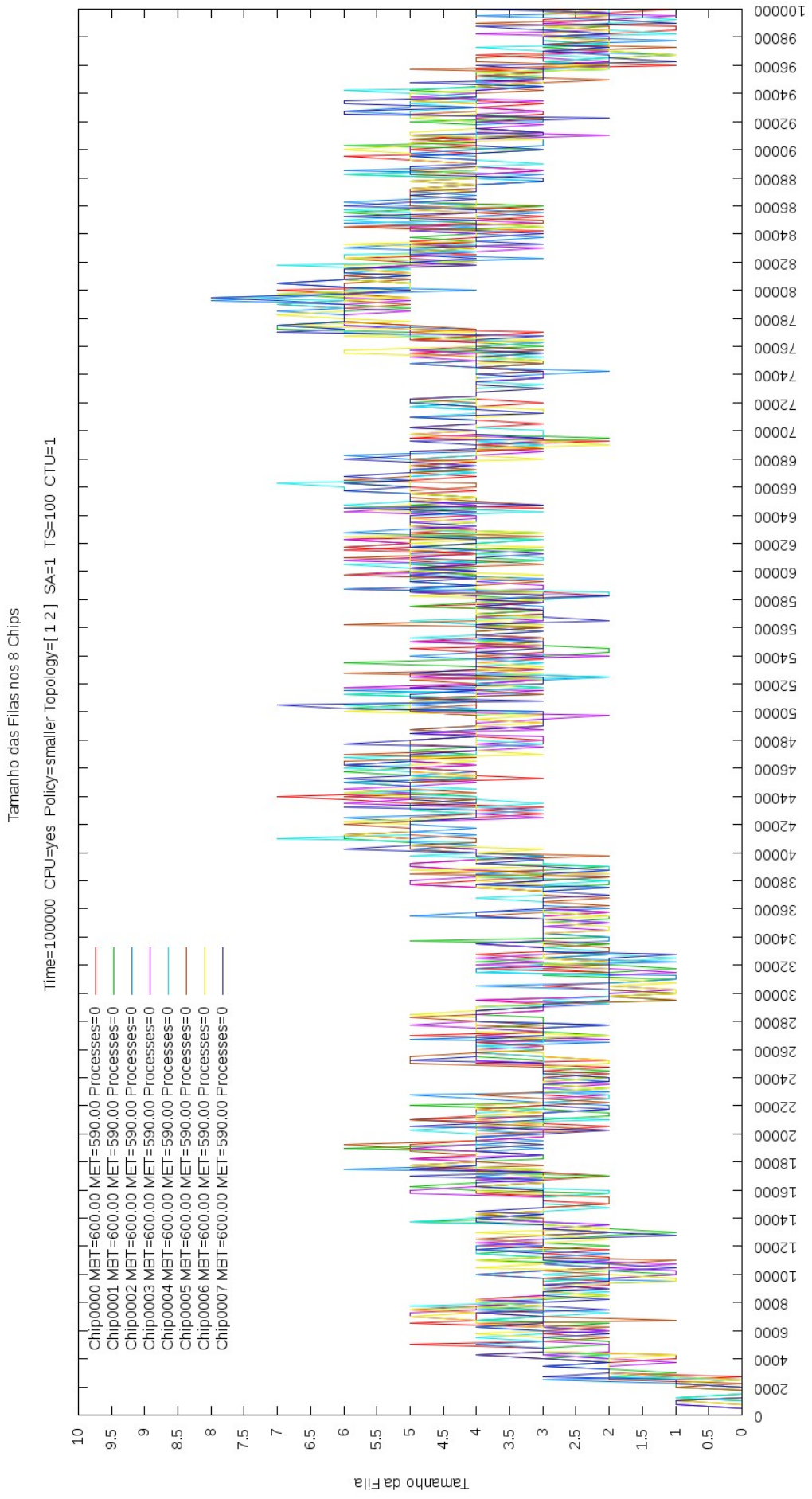

Figura 5.6: Comportamento de Topologia Duplamente Conectada com Política Menor

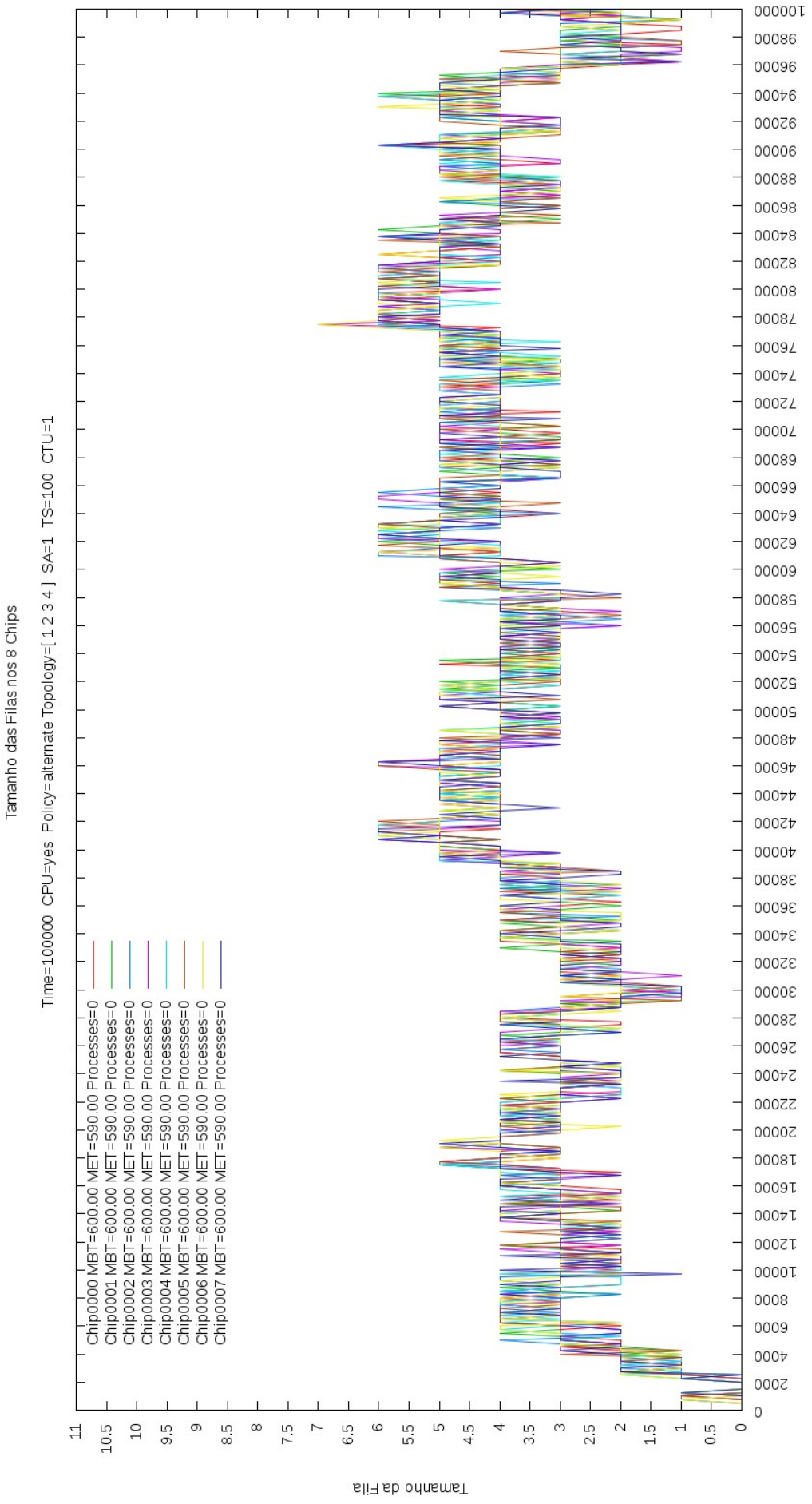

Figura 5.7: Comportamento de Topologia de Quatro Conexões com Política Alternada

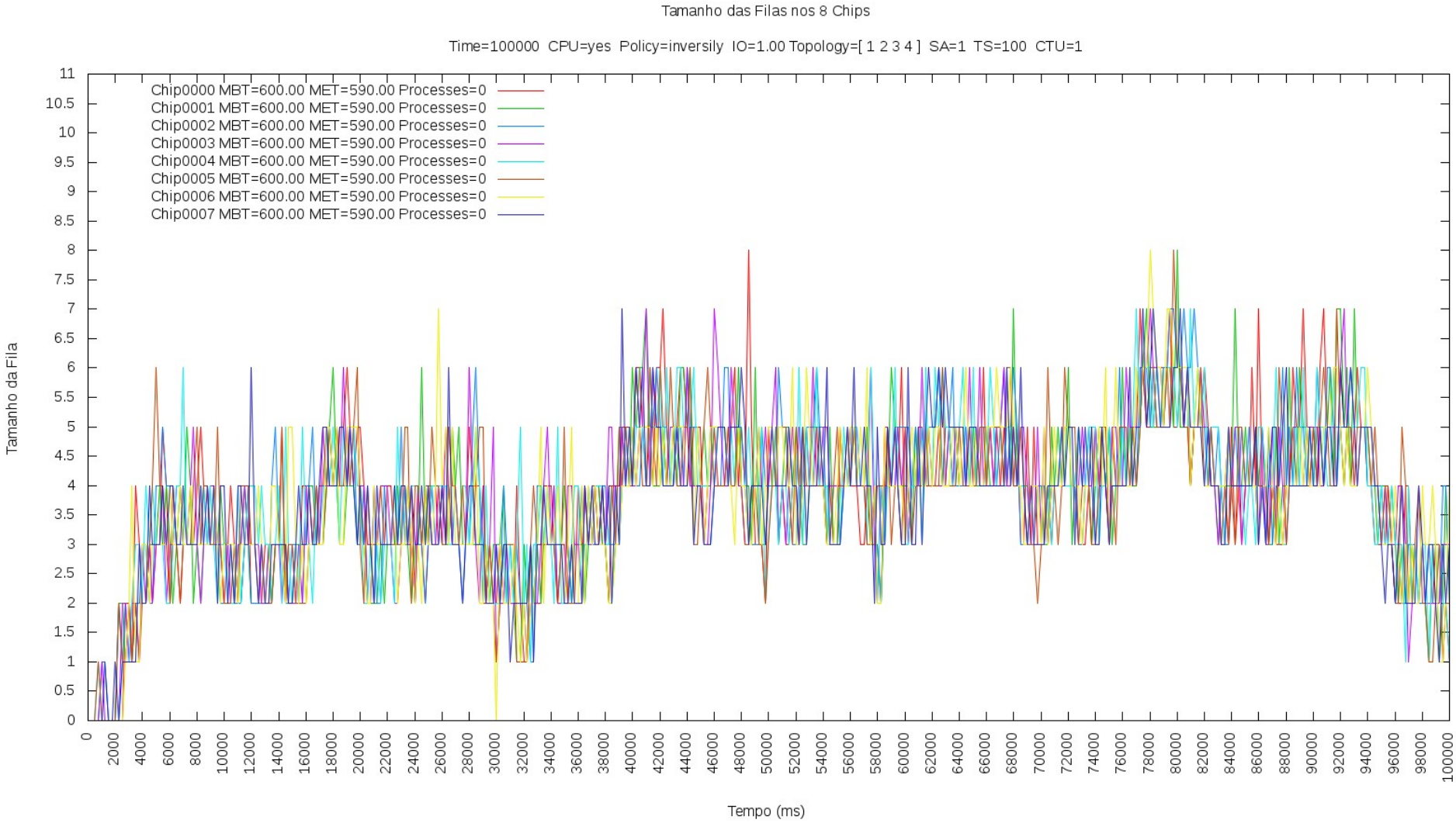

Figura 5.8: Comportamento de Topologia de Quatro Conexões com Política Inversamente Proporcional

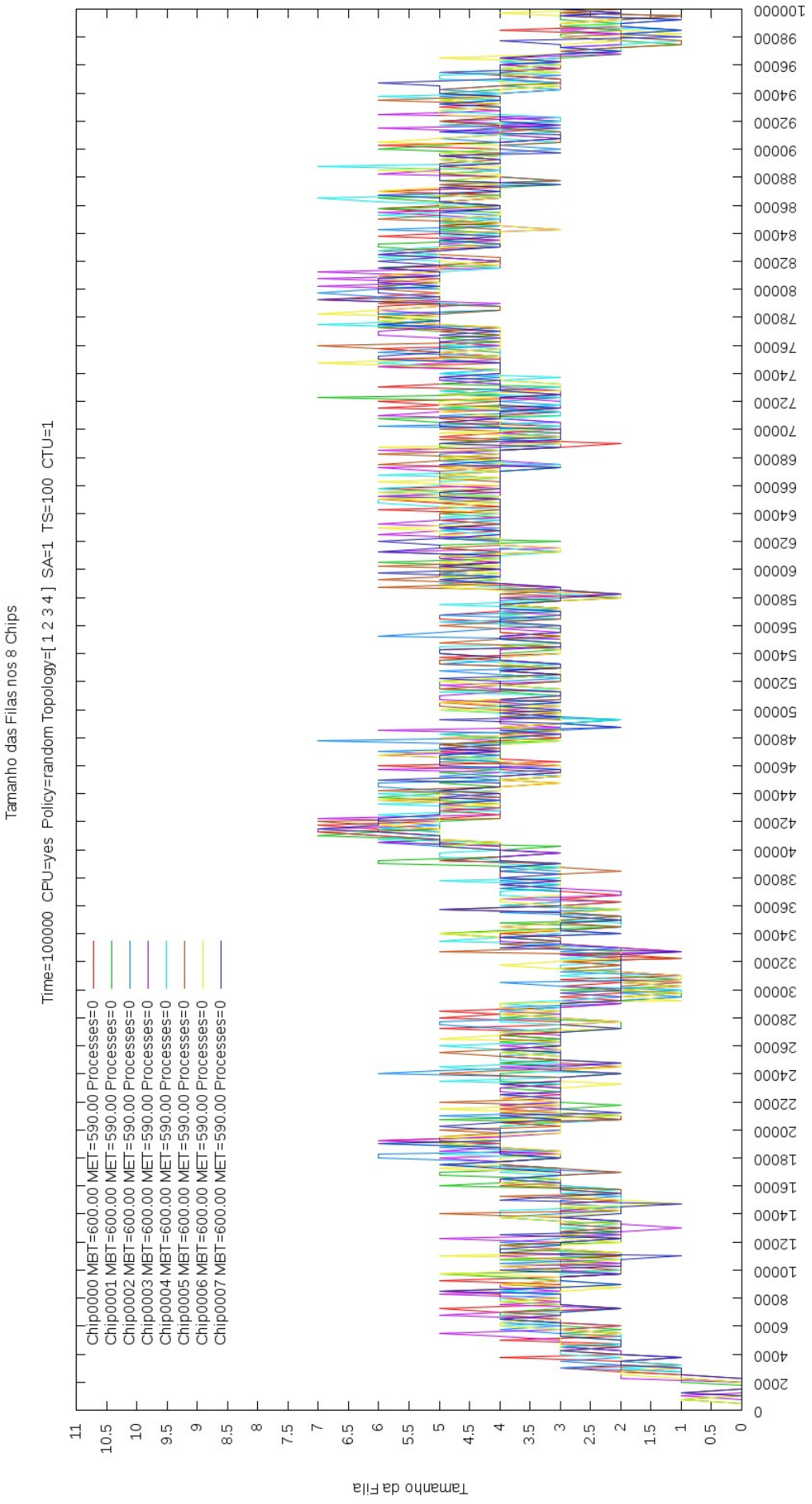

Figura 5.9: Comportamento de Topologia de Quatro Conexões com Política Randômica

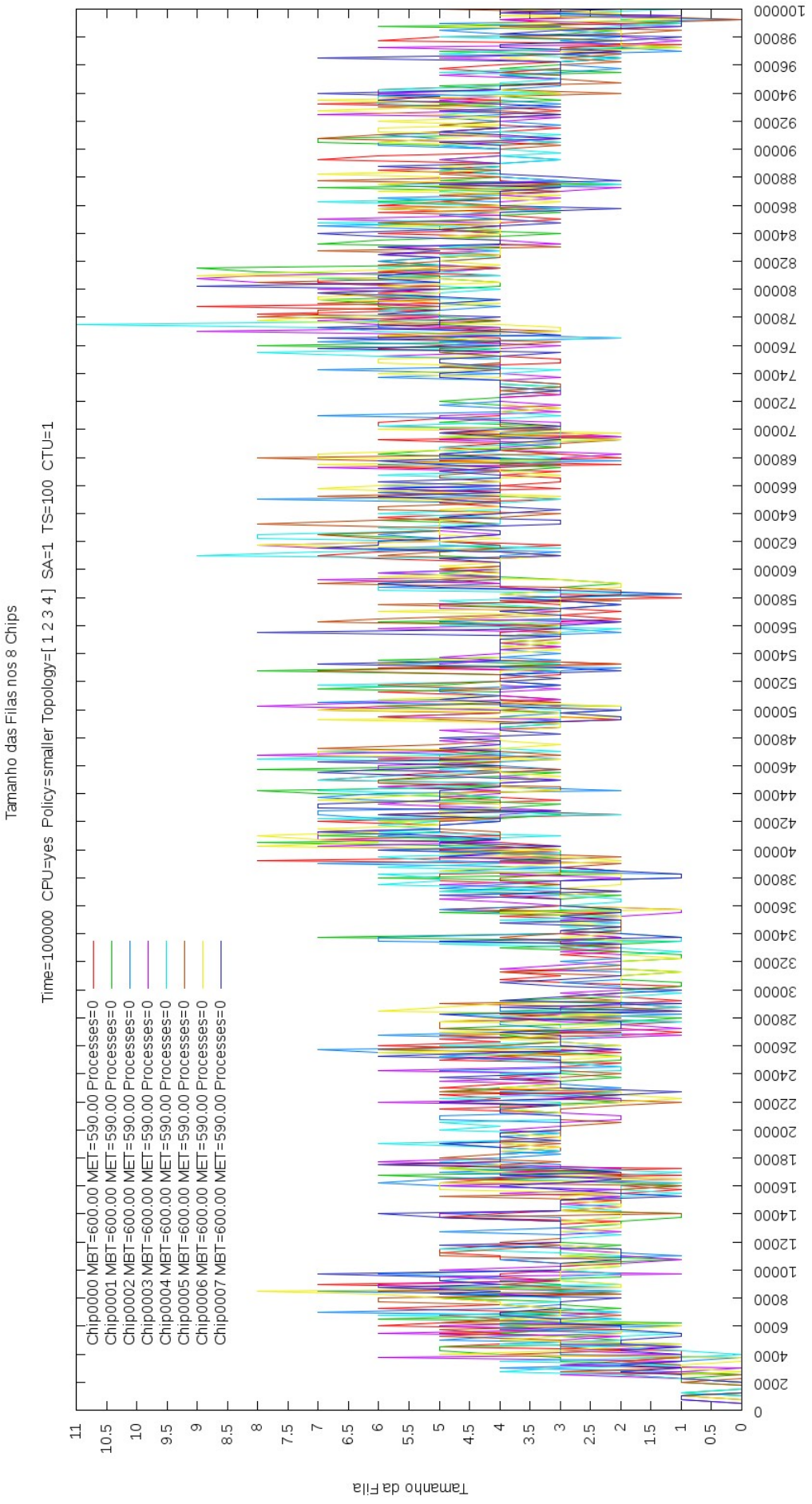

Figura 5.10: Comportamento de Topologia de Quatro Conexões com Política Menor

se estabilize. A consequência é uma menor variabilidade em virtude de trocas mais conscientes.

Por outro lado, como a essência da política menor consiste na escolha do vizinho com menor número de processos, o efeito inverso ocorre. Perceba que dado um *chip* com um menor número de processos, a política menor tende a concentrar neste *chip* a decisão sobre o destino dos processos enviados por seus vizinhos. Esta possibilidade ocorre com menor frequência nas demais políticas, em especial com as políticas alternada e randômica.

Em princípio, um modelo mais estável é preferível, já que uma maior instabilidade pode se refletir em trocas desnecessárias de processos que não adicionam nenhuma melhoria real ao modelo.

5.1.3 A influência de Modificações na Topologia

Nesta seção vamos avaliar o impacto causado pela introdução de modificações na topologia da rede de escalonamento. Em particular, vamos nos ater as consequências da ampliação do número de conexões. As figuras 5.11-5.14 demonstram o efeito de aumentar o número de conexões quando a política alternada se mantém. Enquanto as figuras 5.15-5.18 exibem o resultado de ampliá-las fixando a política menor. Os parâmetros de simulação permanecem idênticos ao caso anterior.

O resultado visível é que o aumento no número de conexões para uma política bem comportada, como a política alternada, não apresenta nenhum efeito particular, com a variabilidade no número de processos dos *chips* permanecendo similar. Para políticas mais temerárias, a ampliação no número de conexões implica em uma maior variabilidade no número de processos.

5.1.4 Balanceamento de Carga na Rede

Nosso objetivo nessa seção é verificar se a rede de escalonamento mantém o balanceamento de carga entre processadores. Para atingir esse propósito vamos simular algumas situações que podem ocorrer em um sistema real.

Carga de Trabalho Inicial Desbalanceada

Em primeiro lugar vamos simular um cenário hipotético na qual um conjunto de processadores possuem uma carga de trabalho desbalanceada no início da operação do sistema. A figura 5.19 simula uma situação na qual quatro processadores possuem uma carga inicial de 100, 200, 300 e 400 processos respectivamente. Os tempos médios de nascimento e execução de processos são 600 ms e 590 ms. A simulação foi realizada durante 10.000 ms.

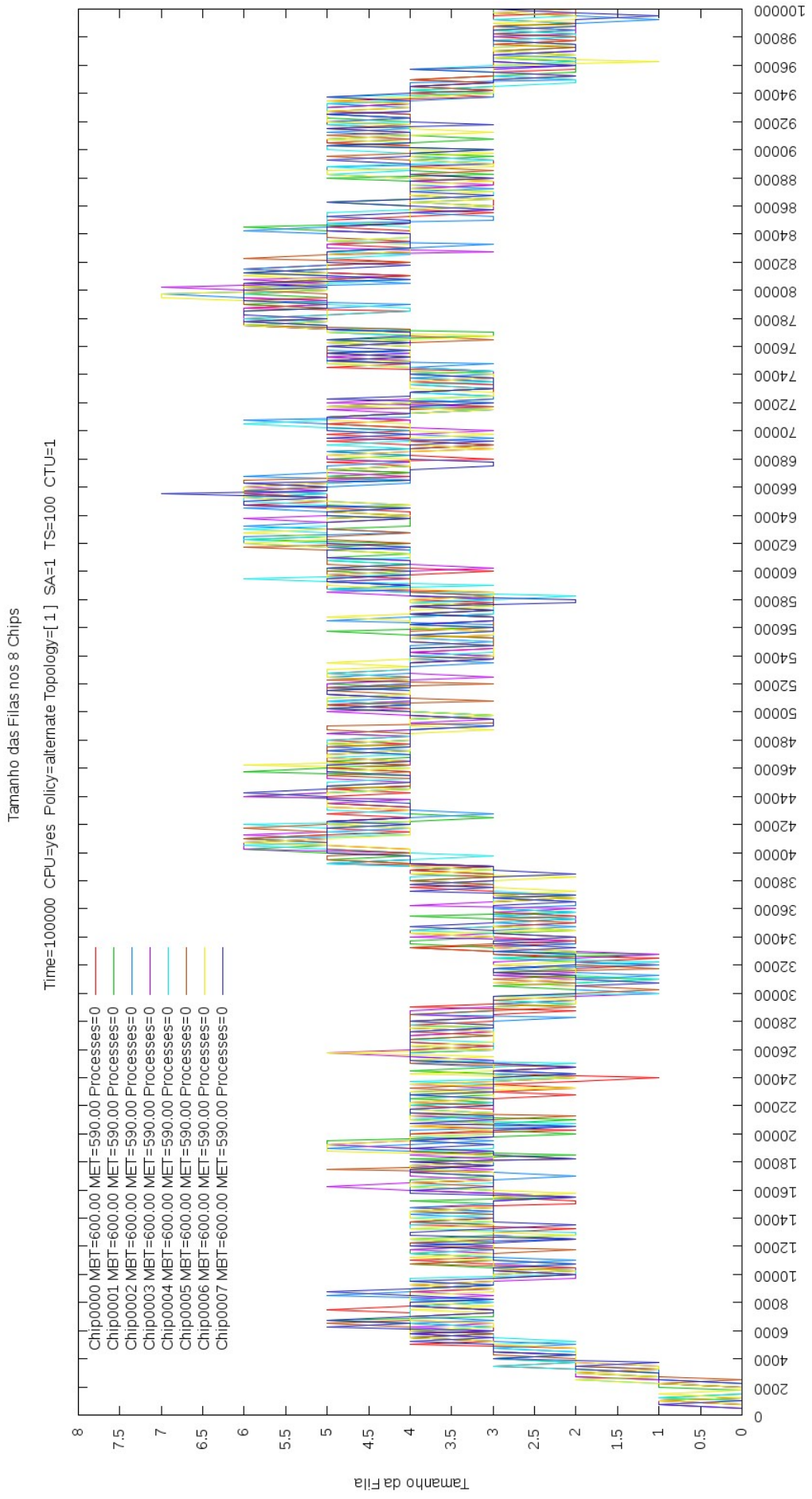

Figura 5.11: Comportamento de Topologia com Conexão Simples para Política Alternada

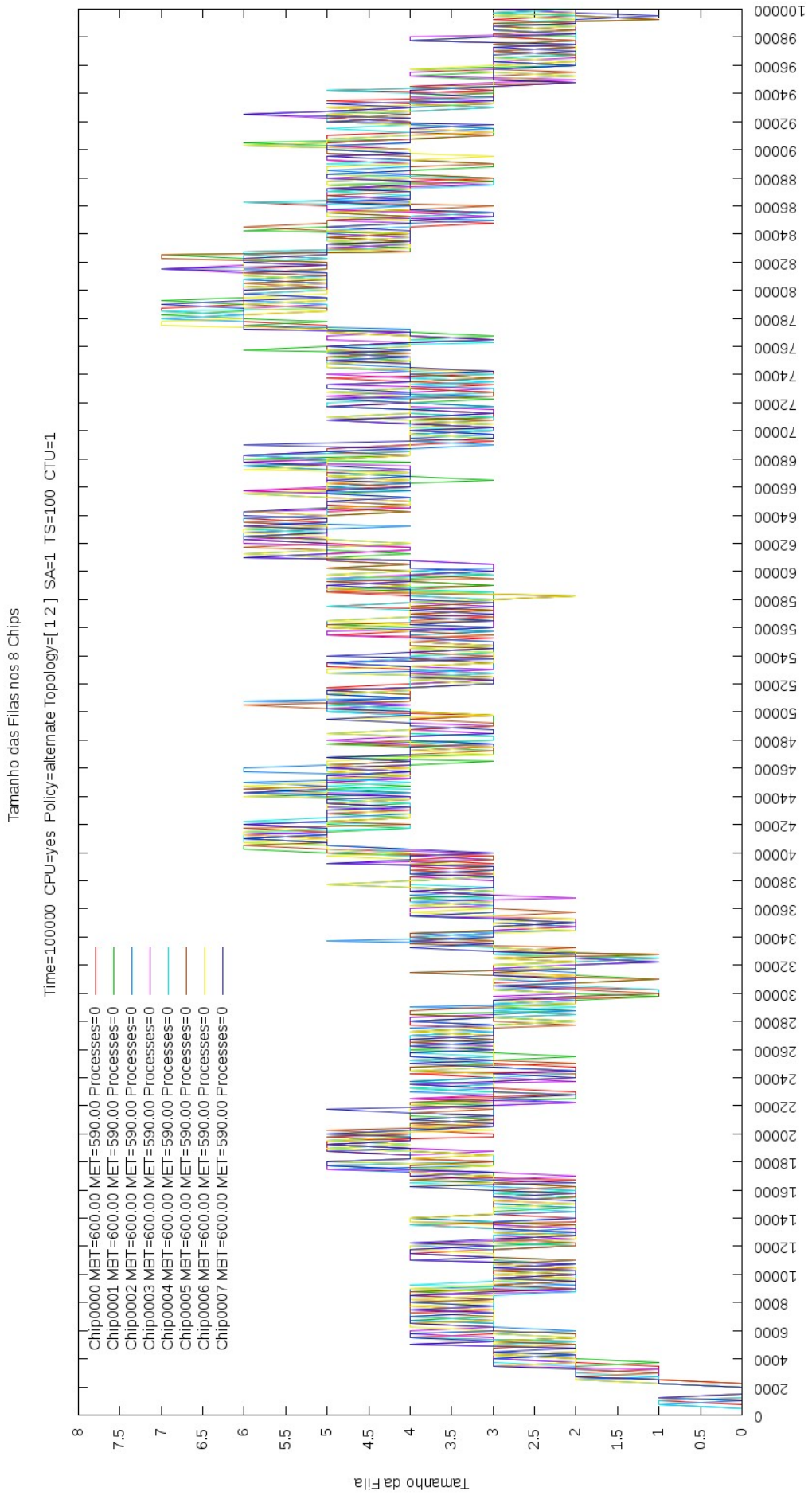

Figura 5.12: Comportamento de Topologia com Conexão Dupla para Política Alternada

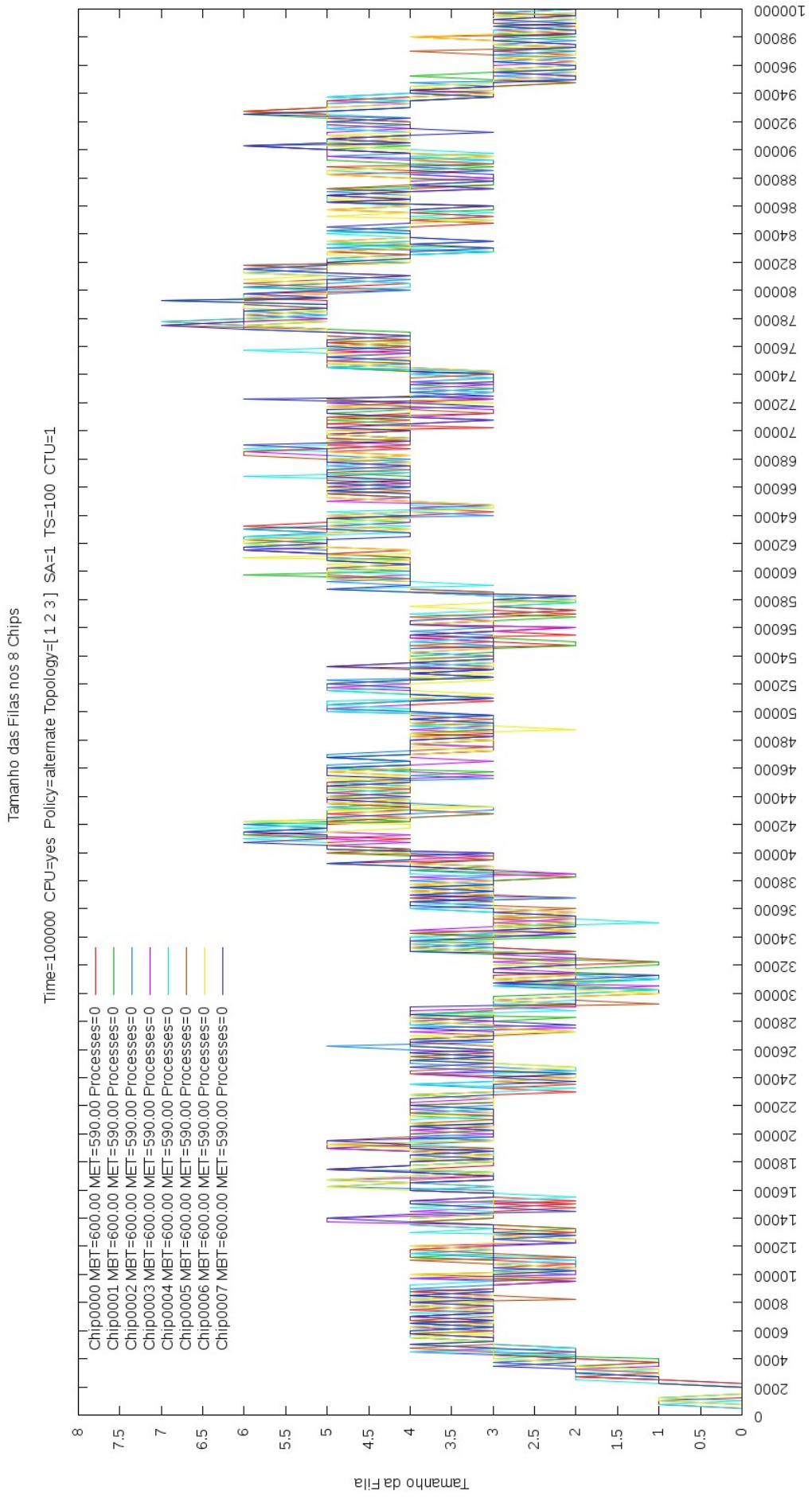

Figura 5.13: Comportamento de Topologia com Conexão Tripla para Política Alternada

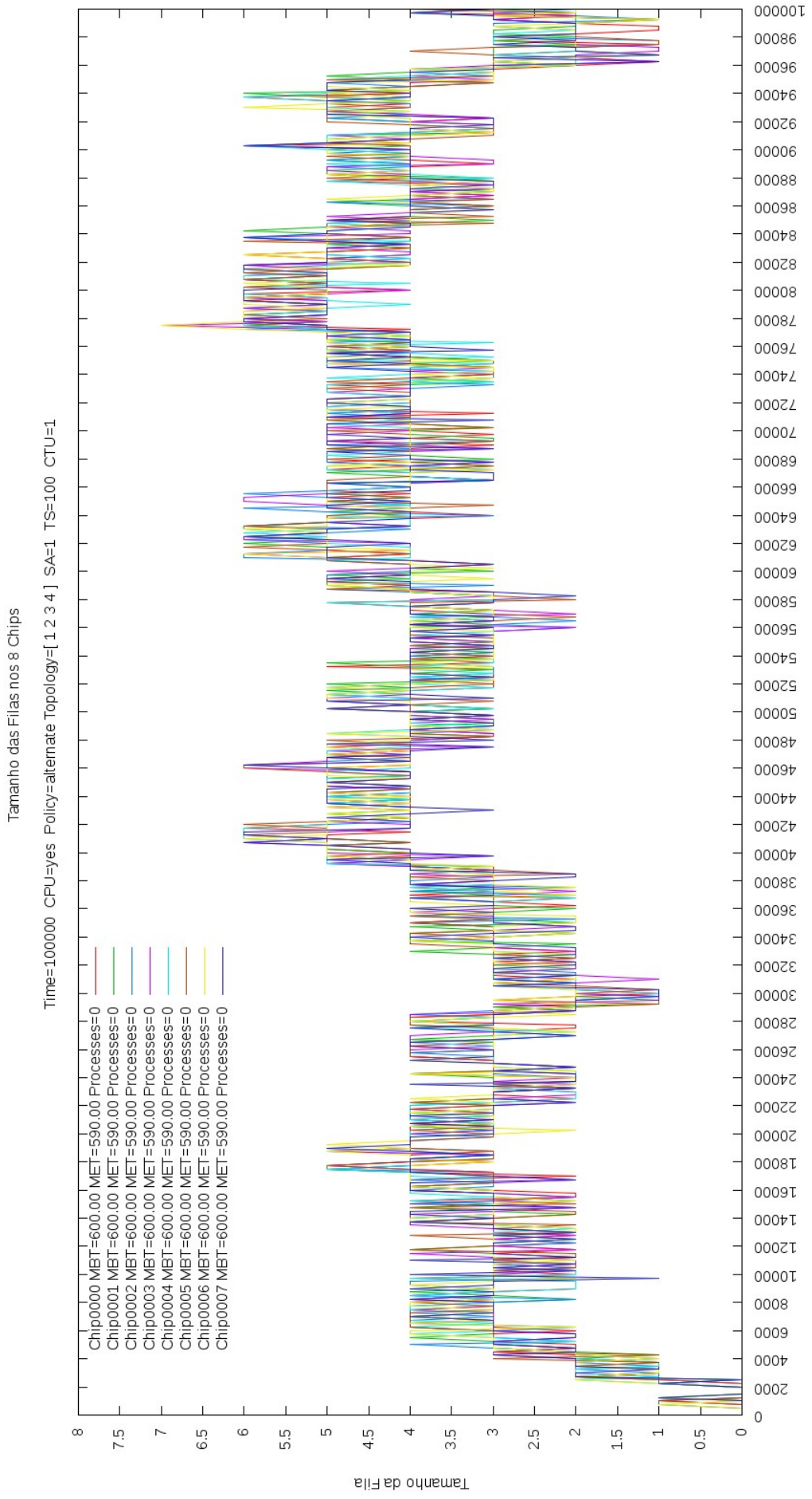

Figura 5.14: Comportamento de Topologia com Conexão Quádrupla para Política Alternada

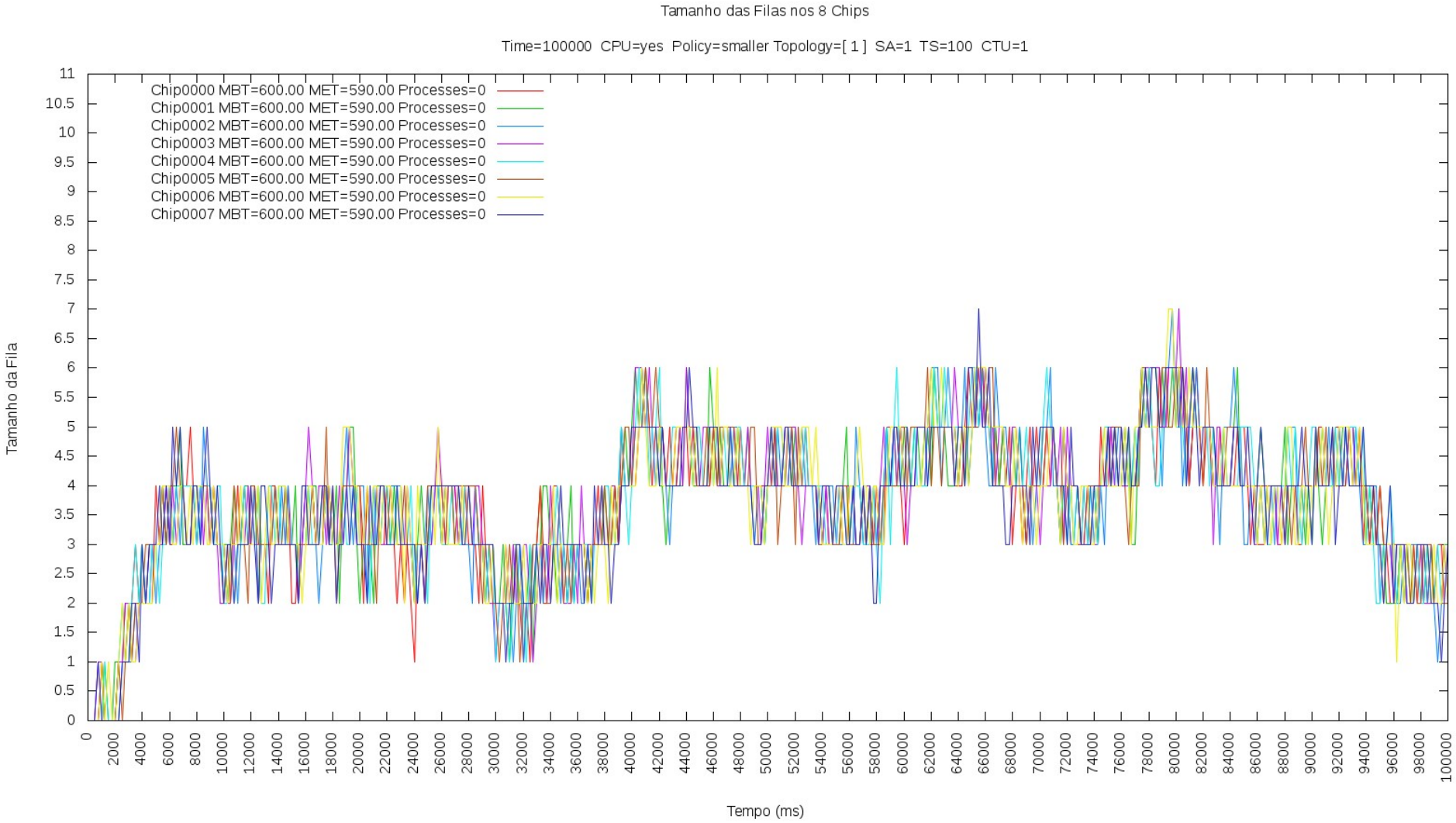

Figura 5.15: Comportamento de Topologia com Conexão Simples para Política Menor

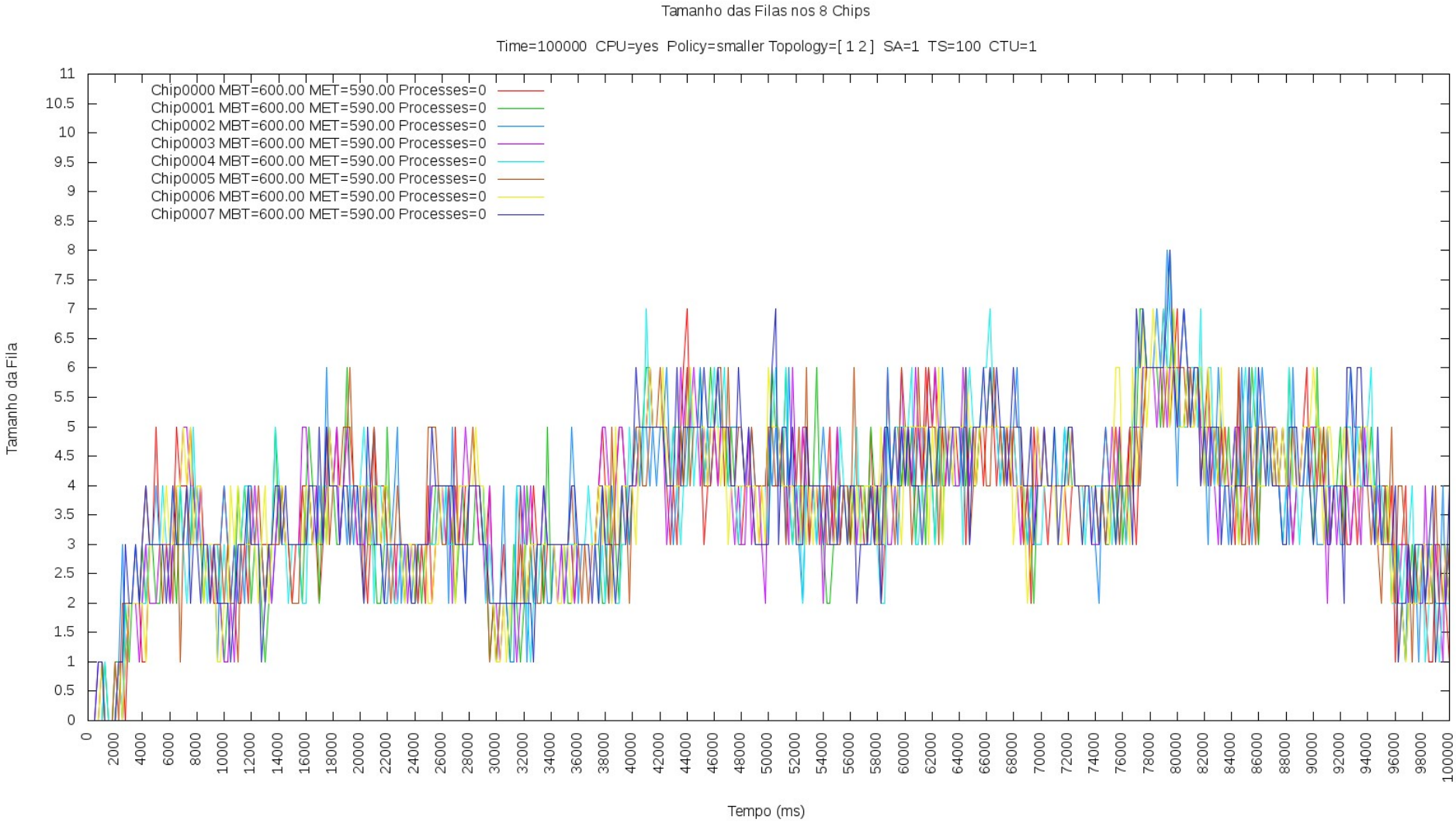

Figura 5.16: Comportamento de Topologia com Conexão Dupla para Política Menor

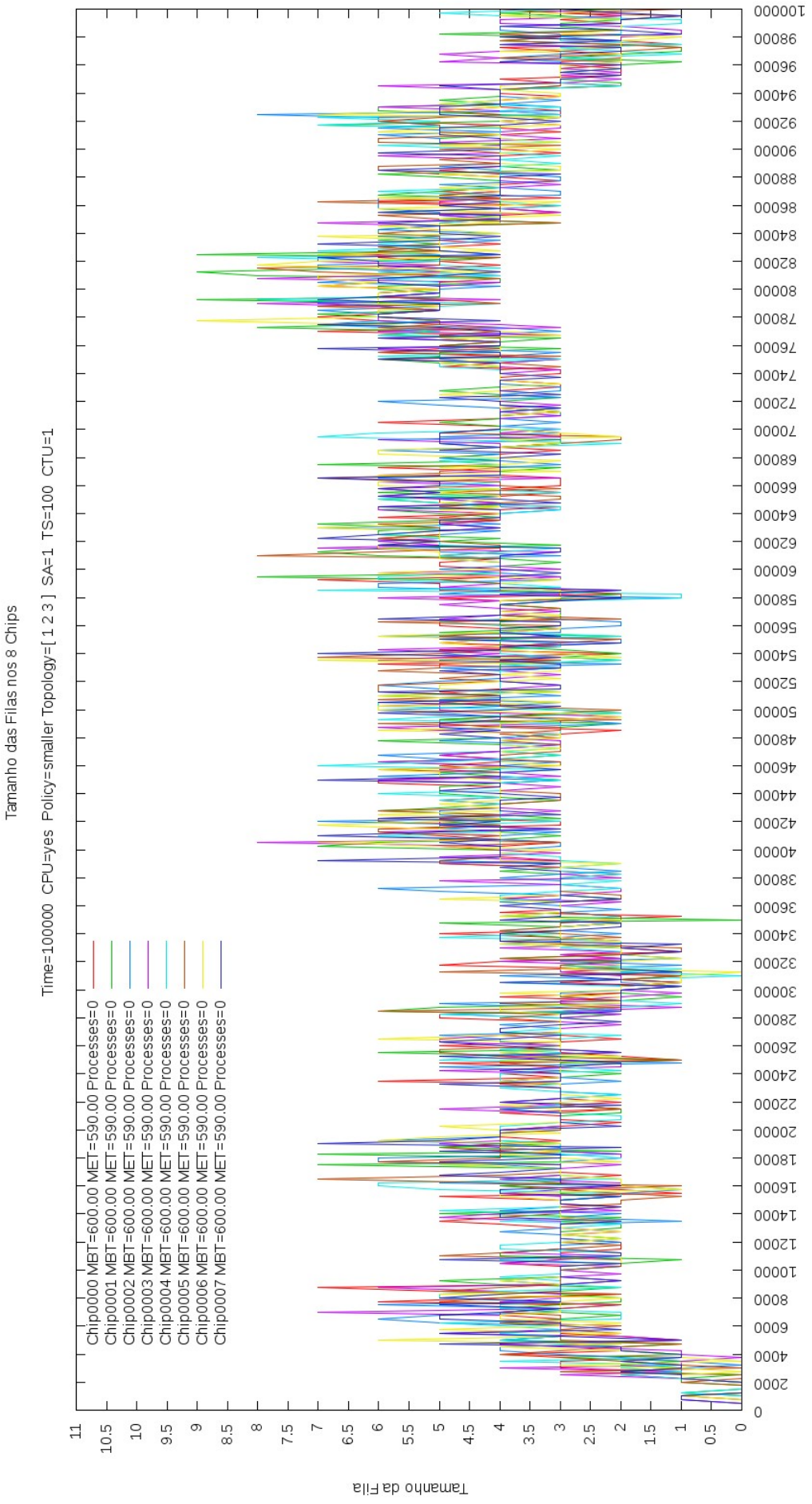

Figura 5.17: Comportamento de Topologia com Conexão Tripla para Política Menor

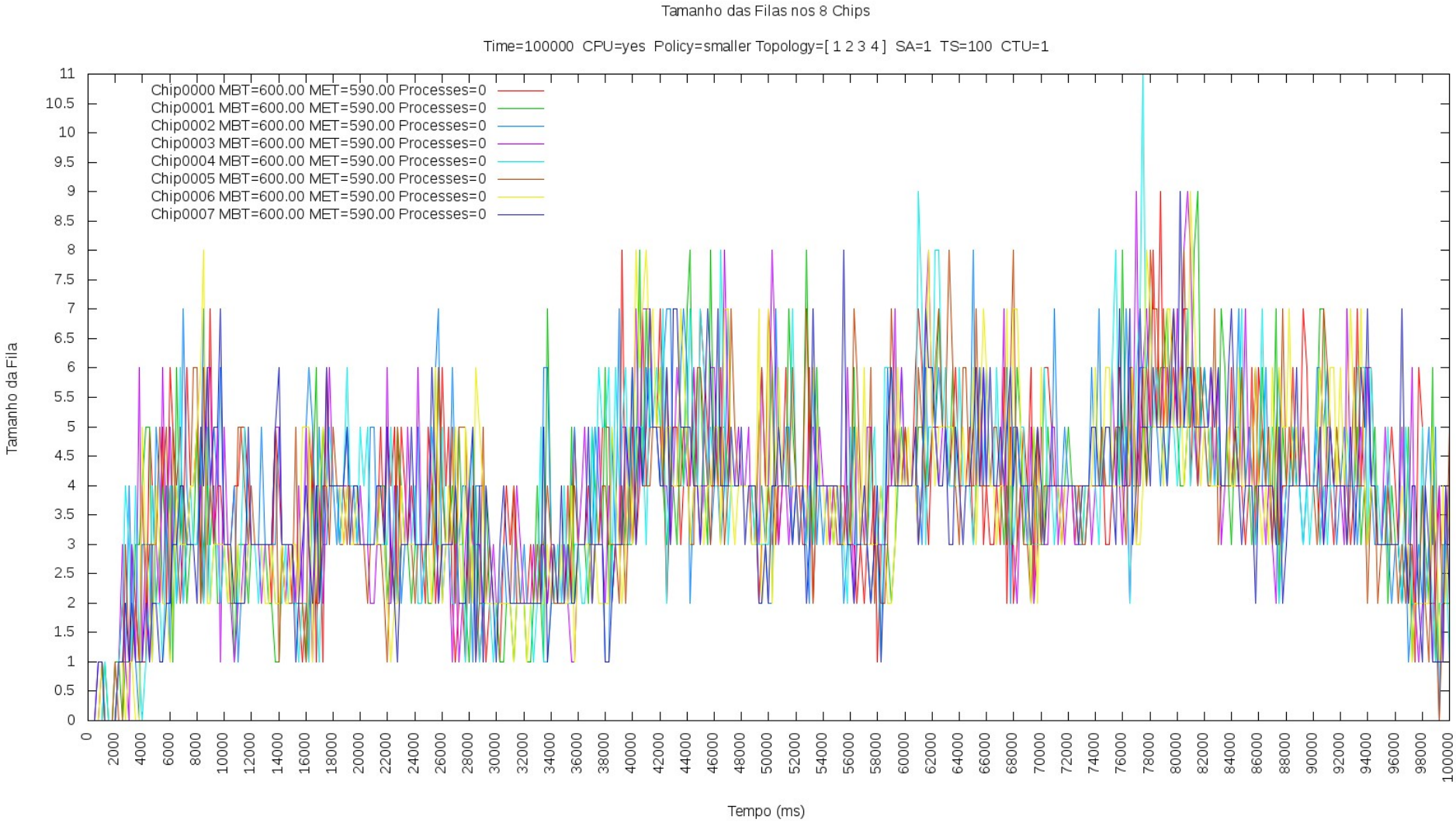

Figura 5.18: Comportamento de Topologia com Conexão Quádrupla para Política Menor

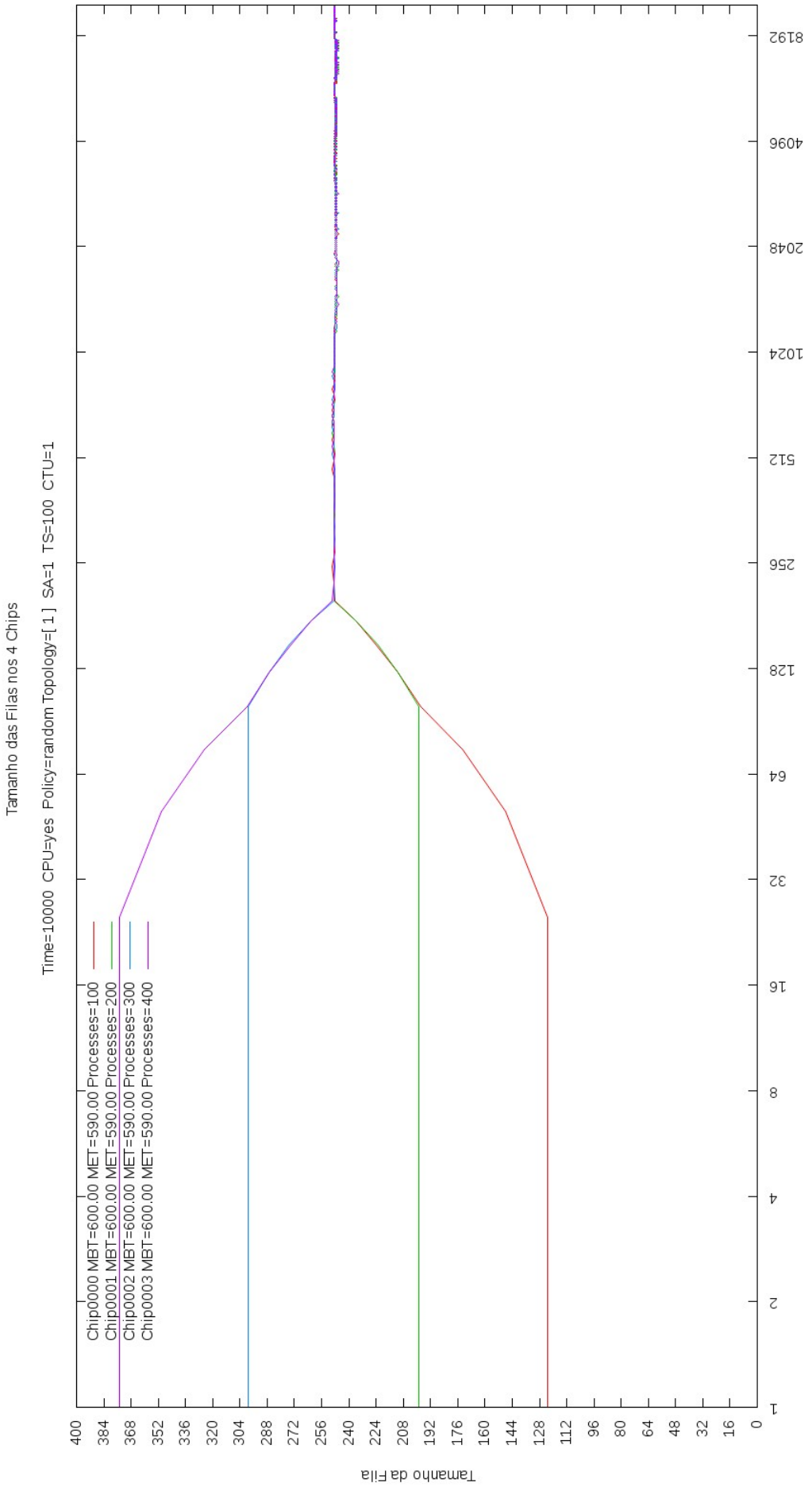

Figura 5.19: Comportamento da Rede com Carga Inicial Desbalanceada

A figura utiliza escala logarítmica a fim de facilitar a visualização dos dados. Como pode ser inferido diretamente, a utilização da rede de escalonamento favorece o balanceamento de carga fazendo com que as filas de processo convirjam um tamanho semelhante.

Taxa de Nascimento de Processos Distintas

A figura 5.20 simula um cenário no qual quatro processadores possuem tempos médios de ingresso de processos diferentes, cujos valores são respectivamente 300, 400, 500 e 600 ms e um tempo médio de execução de processos de 590 ms. Note que nos três primeiros processadores a taxa média de ingresso de processos é maior que a taxa média de execução, originando um desequilíbrio nestes nós. O tempo de simulação empregado foi equivalente a 50.000 ms. Novamente, como pode ser observado, os tamanhos de fila convergem para uma faixa de valores próximos.

Taxa de Nascimento de Processos Intensiva

A figura 5.21 esboça o comportamento da rede de escalonamento em um situação na qual a taxa média de nascimento é maior que a taxa média de execução de processos em todos os nós. Em um sistema real, essa situação tenderia a degradar o desempenho do sistema caso não houvesse uma distribuição igualitária da carga de trabalho. Nesse cenário, um processo chega a cada 500 ms em média, mas seu tempo médio de execução é de 600 ms. O resultado da simulação durante 50.000 ms é semelhante aos obtidos anteriormente.

5.1.5 Escalamento de Desempenho da Rede

Vamos abordar aqui como a rede de escalonamento pode manter seu desempenho quando a demanda, isto é, o número de processos que chegam ao sistema, aumenta. A figura 5.22 apresenta o comportamento da rede quando os tempos médios de nascimento (600 ms) e execução (500 ms) de processos são mantidos e novos nós são adicionados. A numeração na legenda indica o número de processadores/*chips* na topologia. A topologia de cada uma das configurações empregou conexão simples. Para cada número de nós foram realizadas 25 simulações, o gráfico resultante demonstra como o número médio de processos em cada configuração variou com o tempo.

Observe que à medida que novos processadores, que dão origem a mais processos, são adicionados, o número médio de processos no sistema tende a aumentar, como esperado. Entretanto, o aumento no número médio de processos não é proporcional ao aumento no número de nós, sendo inferior a isto, na realidade. Esse fato pode ser verificado pelo exame rápido das médias do número médio de processos em cada

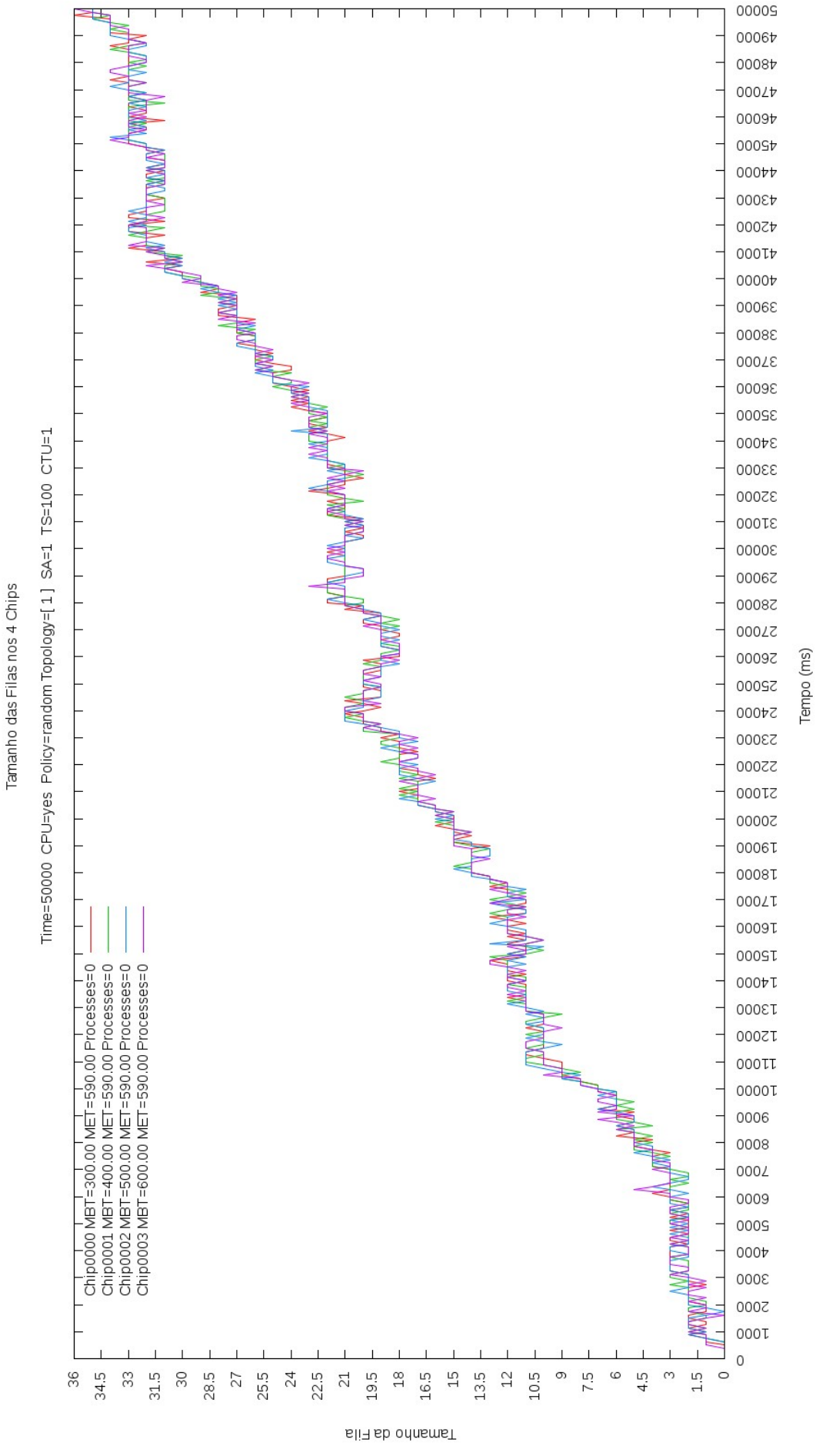

Figura 5.20: Comportamento da Rede com Taxas de Nascimento Distintas

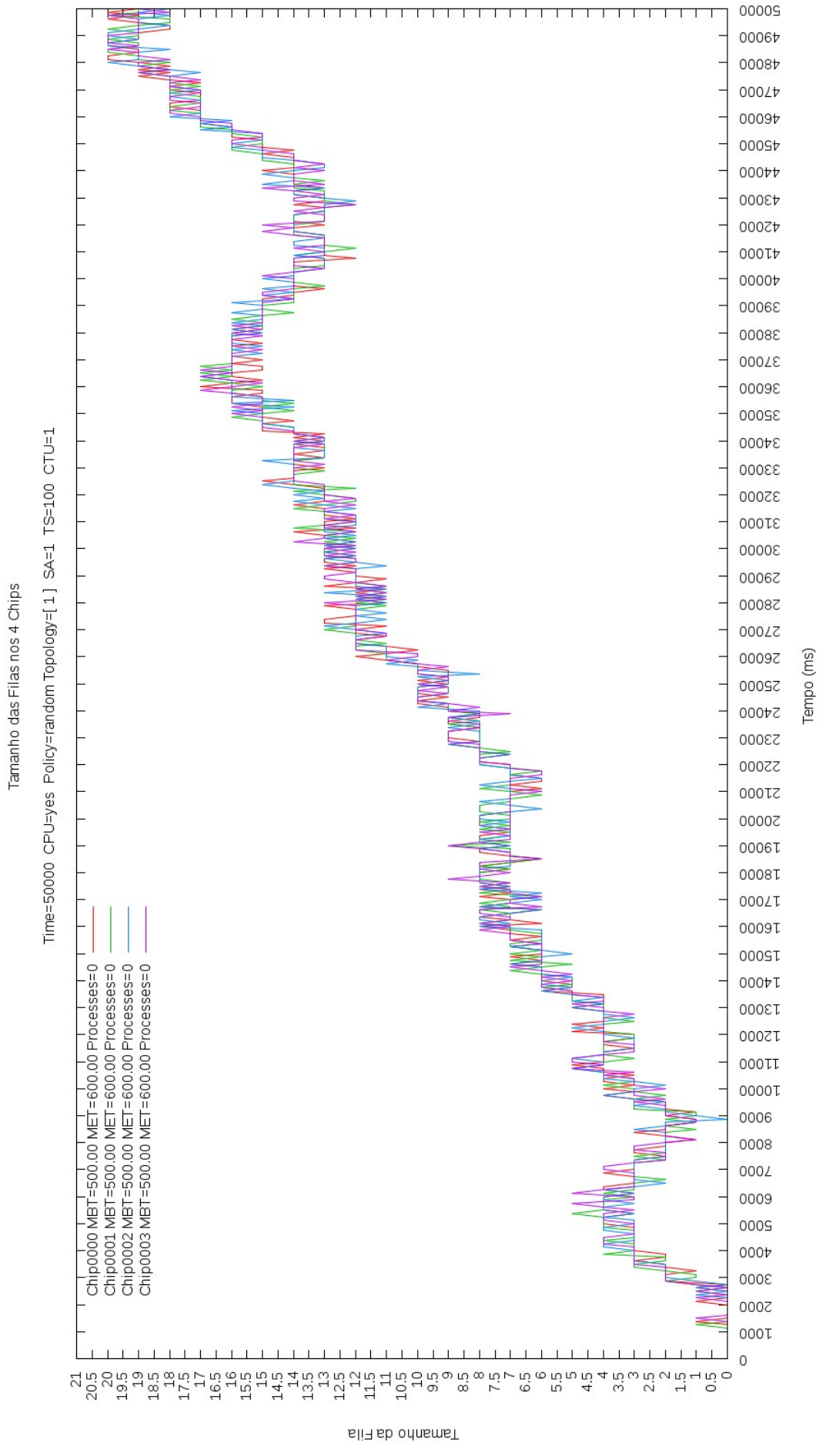

Figura 5.21: Comportamento da Rede com Taxa de Nascimento Intensiva

configuração contidos na tabela 5.1. Em síntese, o modelo sustenta um comportamento semelhante quando a razão entre o ingresso de processos e numero de nós se mantém.

Número de Nós	Número Médio de Processos
4	6.98
8	11.46
12	14.04
16	17.52

Tabela 5.1: Número Médio de Processos no Sistema em Função do Escalamento da Rede

5.1.6 Comportamento Médio da Rede de Escalonamento

Nesta seção vamos avaliar o comportamento médio da rede de escalonamento em função de diferentes políticas e topologias. Os gráficos apresentados nesta seção reproduzem o tamanho médio das filas de 25 simulações para cada política em uma dada topologia. Os tempos médios de nascimento e execução são respectivamente 600 ms e 500 ms.

As figuras 5.23-5.25 ilustram o comportamento médio da rede com diferentes políticas e topologias com duas, quatro e seis conexões. Como pode ser inferido da figura, o aumento no número de conexões não melhora ou piora o desempenho da rede de escalonamento para uma dada política em um sentido estrito. Há melhoras e pioras para todas as políticas à medida que a número de ligações varia.

A conclusão é que uma política em particular poder ser mais adequada a determinada topologia. A tabela 5.2 consolida as informações mostradas nas figuras em um média do número médio de processos no sistema para cada política/topologia.

Política	1 Conexão	2 Conexões	4 Conexões	6 Conexões
Alternada	11.23	10.17	9.51	9.89
Inversamente	10.71	9.89	10.01	11.20
Menor	10.75	11.16	10.80	11.00
Randômica	10.79	9.34	9.66	9.97

Tabela 5.2: Número Médio de Processos no Sistema com Oito Nós em Função da Variação de Políticas e Topologias

No caso anterior, embora não explicitado, assumimos que as distribuições dos tempos de nascimento e execução dos processos podiam variar entre simulações distintas. As figuras 5.26-5.28 apresentam as mesmas situações mostradas nas figuras 5.23-5.25 quando as distribuições dos tempos de nascimento e execução de processos se mantêm fixas, ainda que resguardando as mesmas taxas do exemplo anterior.

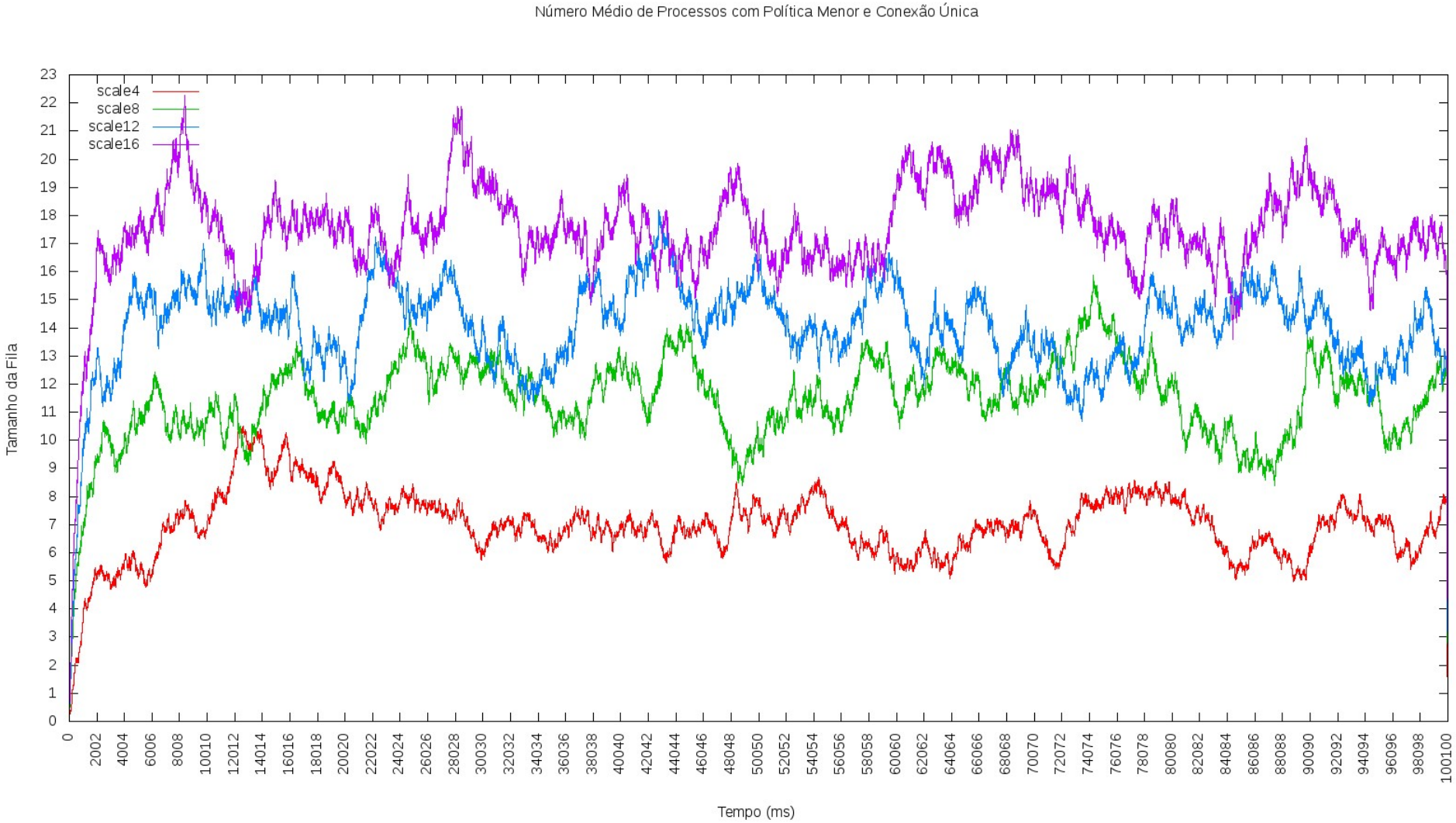

Figura 5.22: Escalamento da Rede pela Adição de Novos Nós

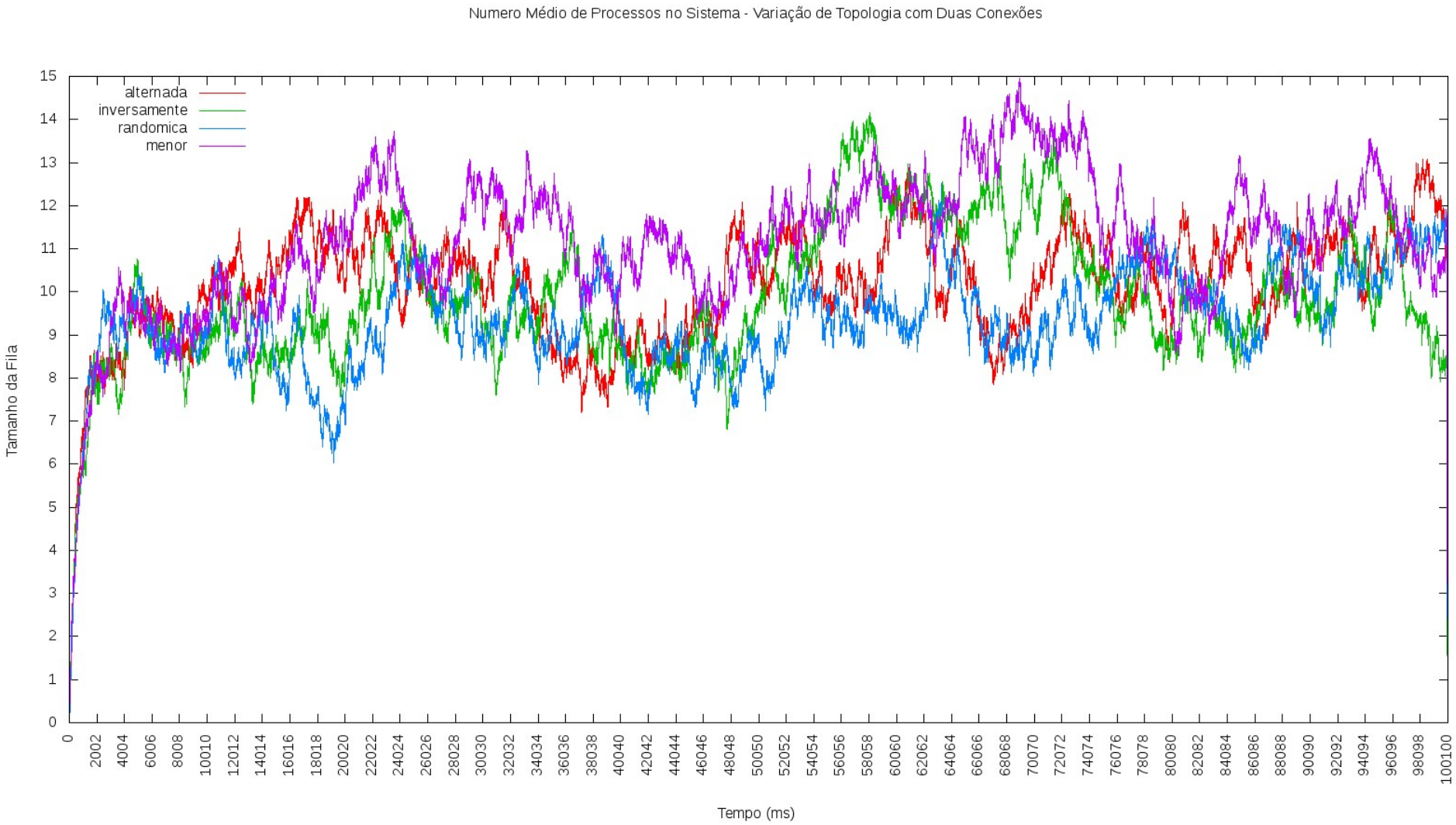

Figura 5.23: Número Médio de Processo no Sistema por Políticas com Duas Conexões

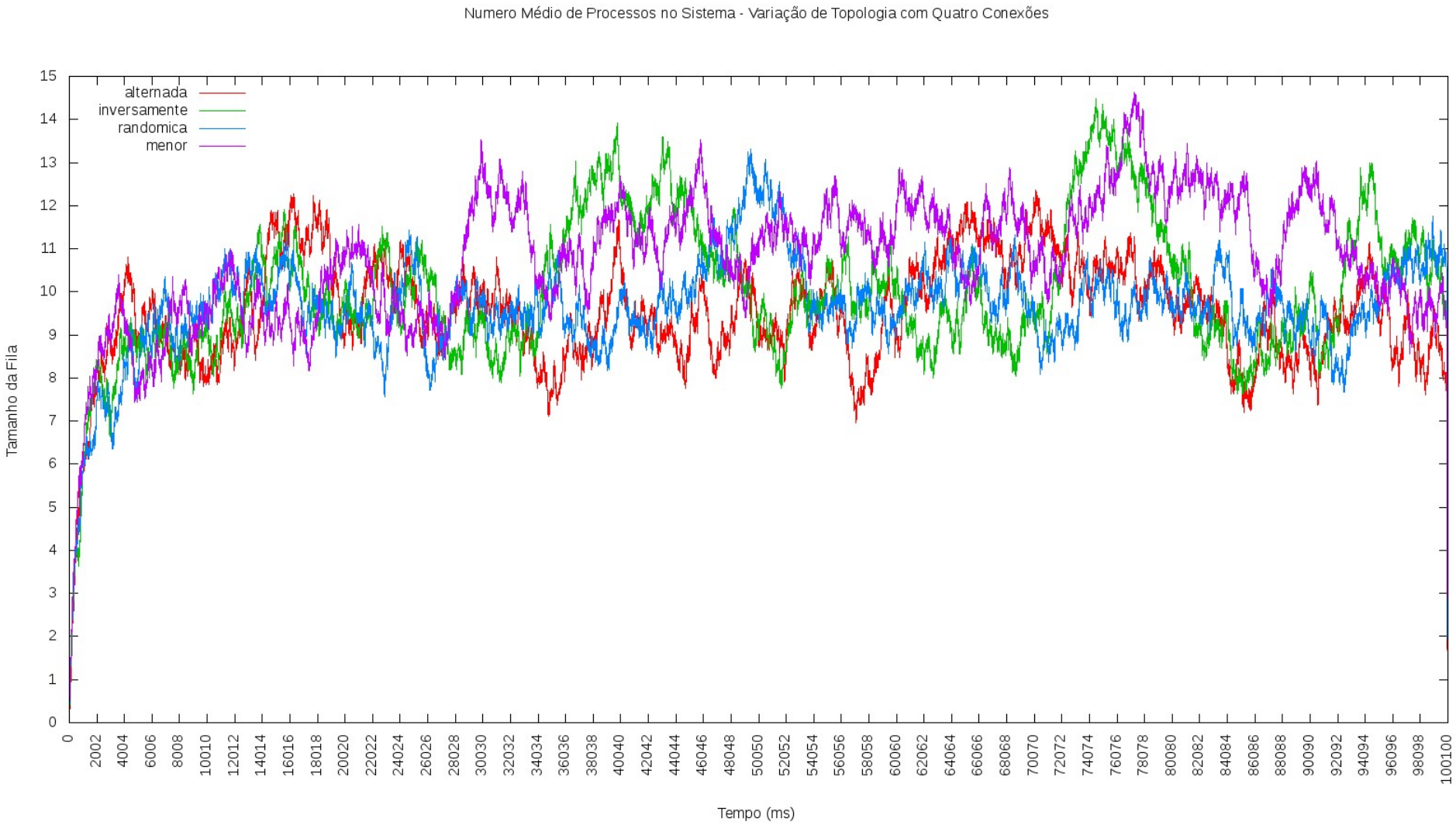

Figura 5.24: Número Médio de Processo no Sistema por Políticas com Quatro Conexões

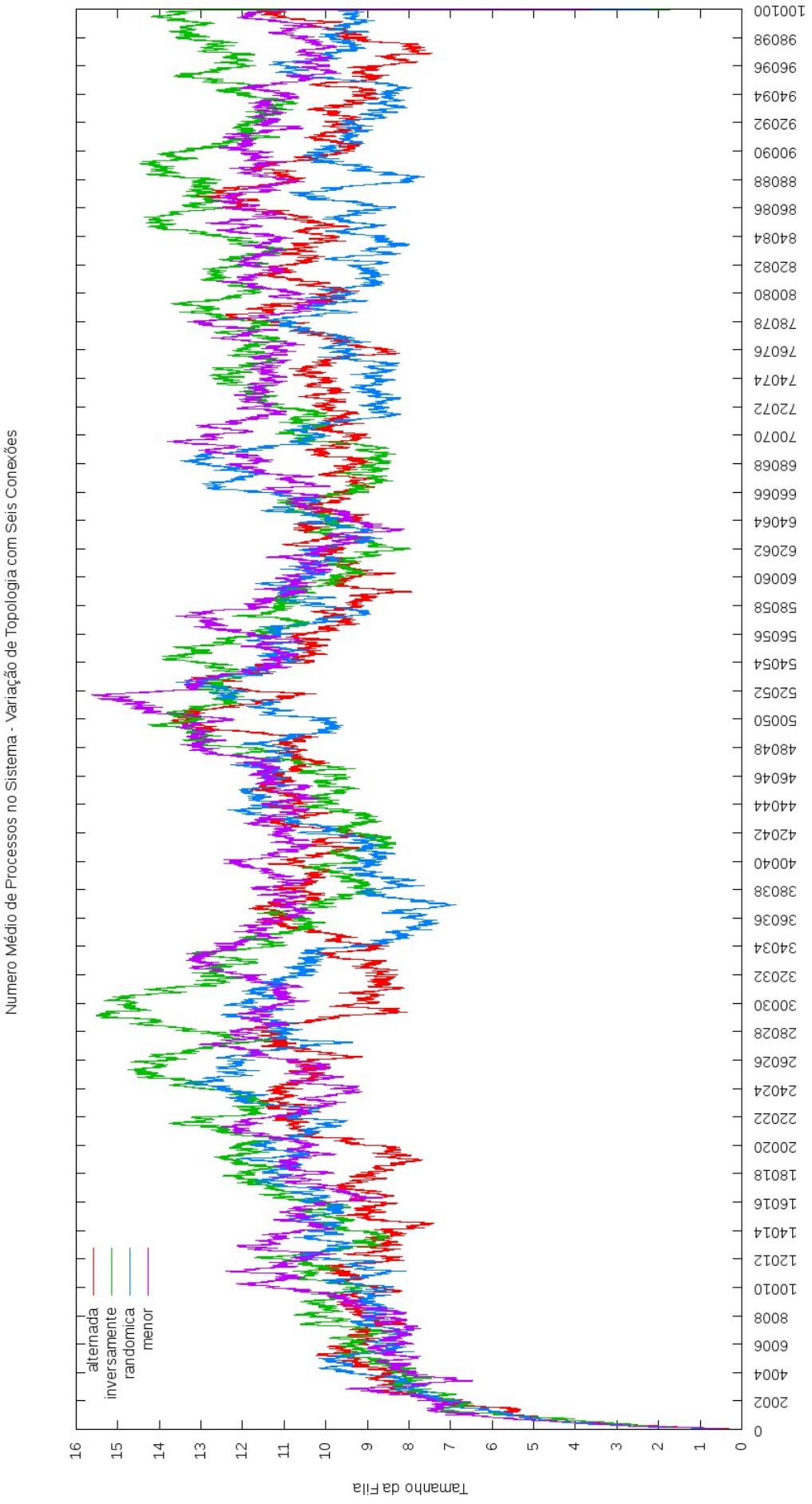

Figura 5.25: Número Médio de Processo no Sistema por Políticas com Seis Conexões

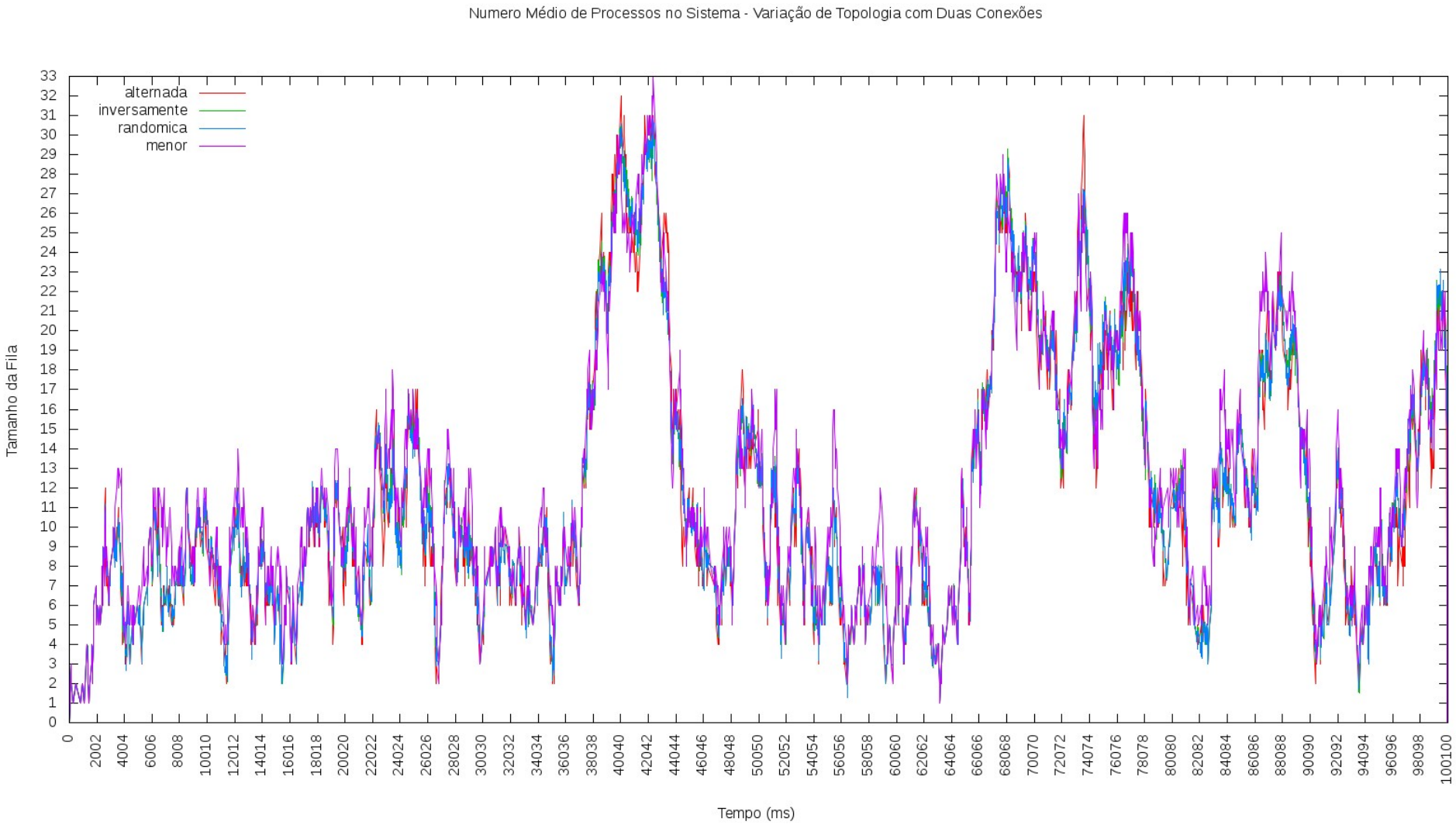

Figura 5.26: Número Médio de Processo no Sistema por Políticas com Duas Conexões e Distribuições de Tempo Fixas

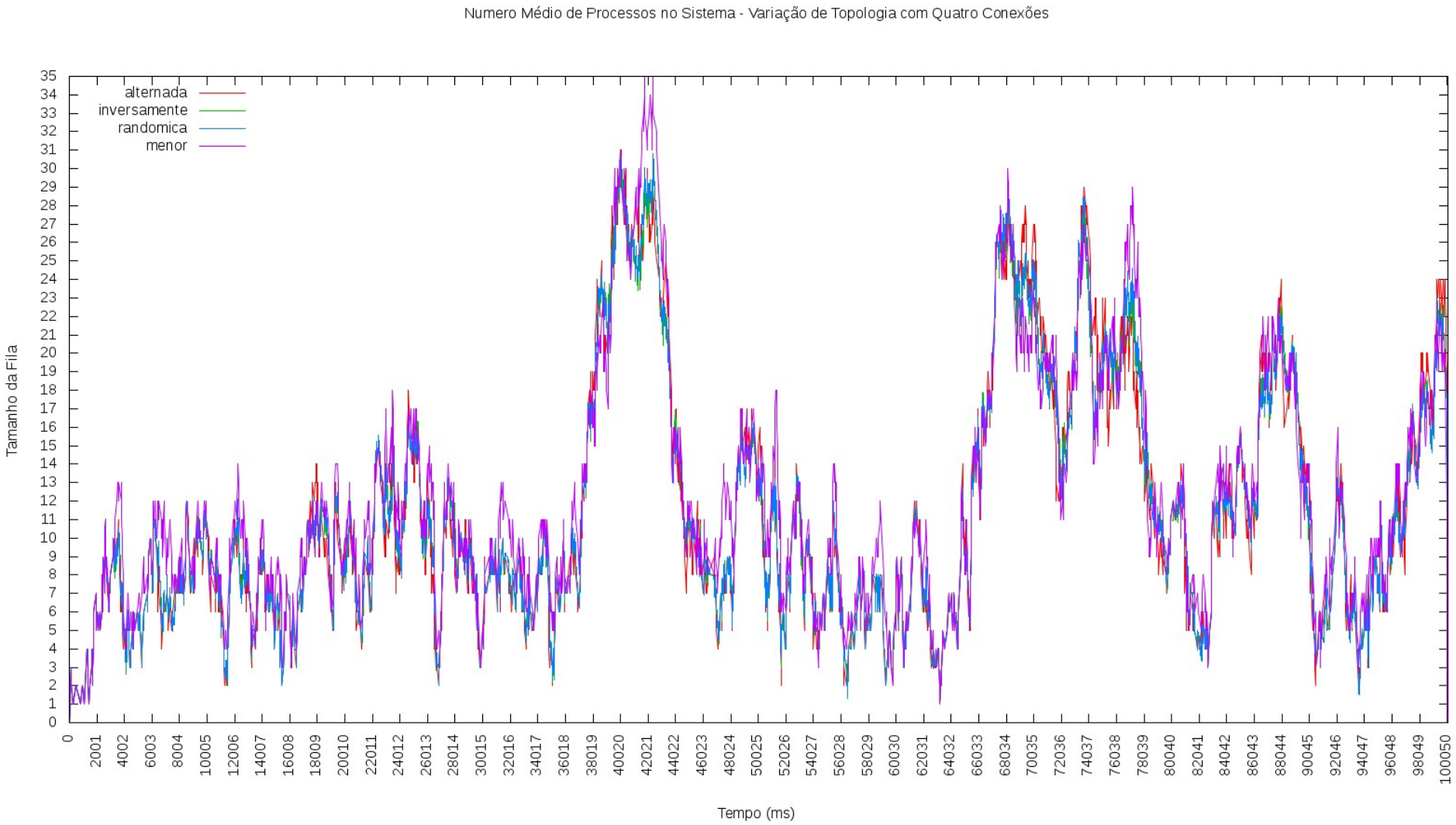

Figura 5.27: Número Médio de Processo no Sistema por Políticas com Quatro Conexões e Distribuições de Tempo Fixas

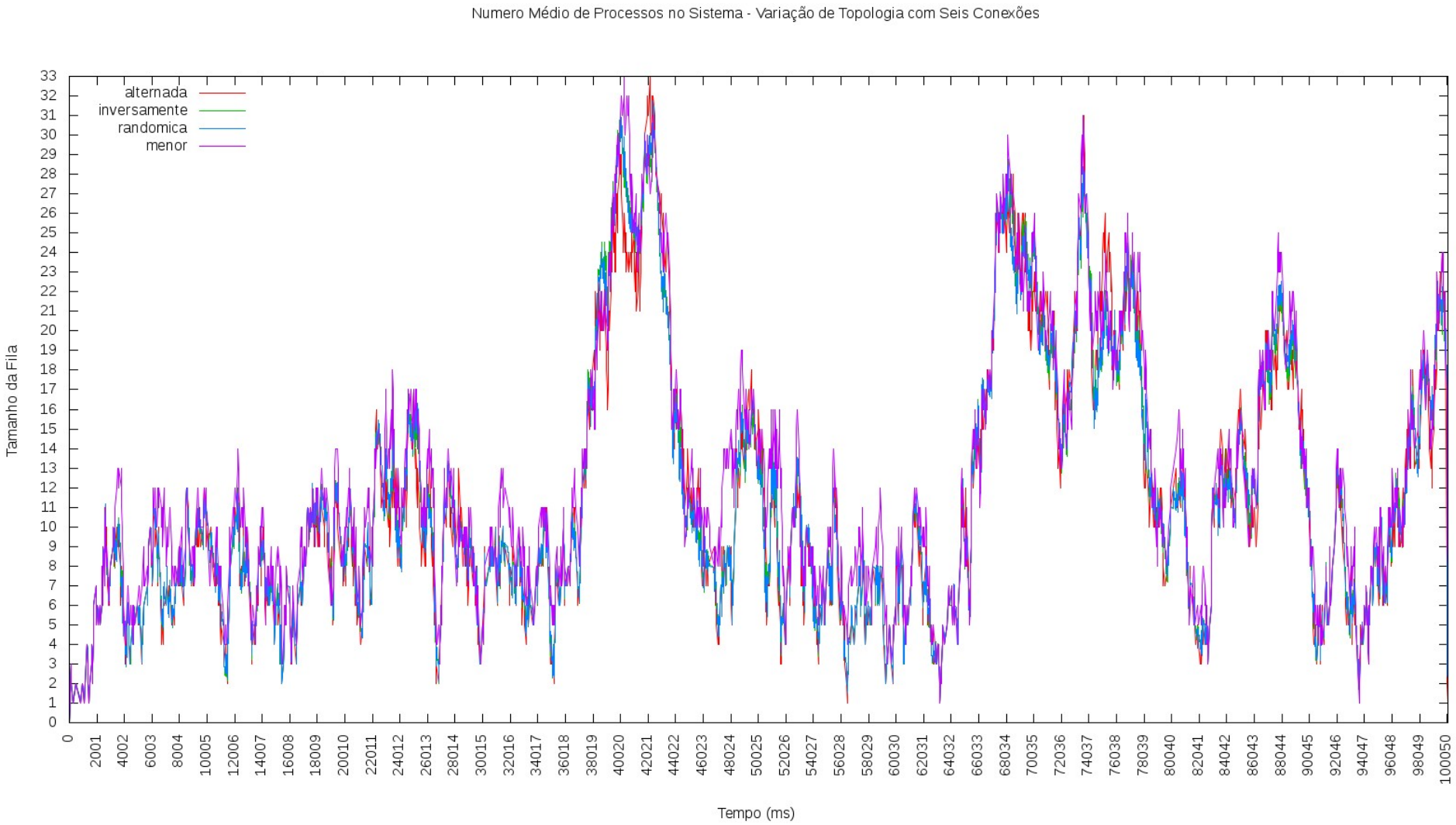

Figura 5.28: Número Médio de Processo no Sistema por Políticas com Seis Conexões e Distribuições de Tempo Fixas

Os resultados dessas novas rodadas de simulações não diferem daqueles obtidos no cenário anterior no que diz respeito a um comportamento geral observável em função do aumento no número de conexões. Entretanto, quando as distribuições de tempo são fixas as políticas apresentam um resultado muito mais homogêneo embora ainda subsista variação. A tabela 5.3 consolida os resultados apresentados.

Política	**1 Conexão**	**2 Conexões**	**4 Conexões**	**6 Conexões**
Alternada	12.14	11.20	11.30	11.28
Inversamente	12.14	11.28	11.27	11.31
Menor	12.14	11.98	12.09	12.29
Randômica	12.14	11.27	11.30	11.31

Tabela 5.3: Número Médio de Processos no Sistema com Oito Nós em Função da Variação de Políticas e Topologias e Distribuições de Tempo Fixas

5.2 Rede de Escalonamento & Modelos Markovianos de Filas

Um processo estocástico é uma família de variáveis aleatórias $\{X(t) \mid t \in T\}$, definida sobre um espaço de probabilidade, indexada por um parâmetro t, onde t varia sobre um conjunto de índices T. Os valores assumidos pela variável aleatória X(t) são chamados estados, e o conjunto de todos os possíveis valores compõem o espaço de estado do processo [49]. Um processo de Markov é um processo estocástico cujo comportamento dinâmico é tal que a distribuição de probabilidade para seu desenvolvimento futuro depende somente do estado presente e não de como o processo chegou neste estado [49]. Se assumirmos que o espaço de estados é discreto, então o processo de Markov é conhecido como cadeia de Markov. Cadeias de Markov são utilizadas em diversas áreas de conhecimento a fim de modelar processos probabilísticos.

5.2.1 Parâmetros de Comparação

Muitas medidas podem ser obtidas com uso de cadeias de Markov. Entretanto, para os propósitos deste trabalho, vamos nos ater a apenas duas: o número médio de elementos no sistema (E[N]) e o tempo médio de resposta (E[R]), que é o tempo médio gasto por esses elementos desde o ingresso até a saída do sistema. O tempo médio de resposta equivale ao *turnaround* definido na seção 3.3.

Como os resultados obtidos pelo uso de cadeias de Markov são baseados na concepção de que o sistema atinge um estado estacionário, convencionamos que o

tempo de simulação será de 1.000.000 de unidades de tempo, a fim de aproximar os resultados da simulação dos obtidos pelo modelo markoviano.

O tempo médio de nascimento (chegada) e execução (serviço) corresponderão respectivamente a 600 e 500 unidades de tempo. Dessa forma, suas respectivas taxas serão $\lambda = \frac{1}{600}$ e $\mu = \frac{1}{500}$. As figuras 5.29-5.31 mostram os esquemas dos modelos de fila cujos resultados serão comparados aos obtidos pela rede de escalonamento.

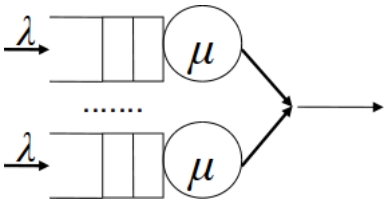

Figura 5.29: 8 filas M/M/1 com taxa de chegada λ e taxa de serviço μ

Figura 5.30: 1 filas M/M/1 com taxa de chegada 8λ e taxa de serviço 8μ

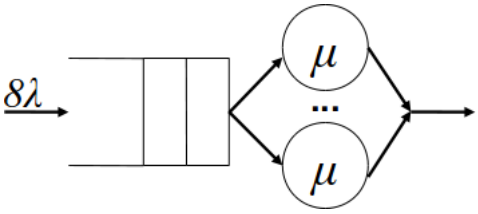

Figura 5.31: 1 filas M/M/8 com taxa de chegada 8λ e taxa de serviço μ

5.2.2 Medidas de Interesse

Para uma fila M/M/1, $E[N] = \frac{\lambda}{\mu - \lambda}$. No caso de uma fila M/M/m, $E[N] = m\rho + \rho\frac{(m\rho)^m}{m!}\frac{\pi_0}{(1-\rho)^2}$, onde $\pi_0 = [\sum_{k=0}^{m-1}\frac{(m\rho)^k}{k!} + \frac{(m\rho)^m}{m!}\frac{1}{1-p}]^{-1}$ e $\rho = \frac{\lambda}{m\mu}$.

A fim de obtermos E[R] utilizamos o resultado de Little, isto é, $E[R] = \frac{E[N]}{\lambda}$. O resultado de Little é particularmente útil porque seu uso não depende da disciplina de escalonamento e nem da distribuição dos tempos de chegada dos processos [18]. Além disso, ele também não depende da distribuição dos tempos de execução dos processos, do número de processadores e do número de filas no sistema.

A tabela 5.4 exibe o número médio de elementos no sistema e o tempo médio de resposta para os modelos de filas mencionados pela utilização dos resultados da cadeia de Markov para filas M/M/1 e M/M/m.

É possível verificar pela tabela que pior desempenho foi apresentado pelo es-

	E[N]	E[R]
8 filas M/M/1	5.00 (por fila)	3000.00 (por fila)
1 fila M/M/1	5.00	375.00
1 fila M/M/8	9.33	699.75

Tabela 5.4: Medidas de Interesse para Modelos Markovianos

quema da figura 5.29 com um tamanho médio de fila igual 5.00 por fila, isto é, 40 no sistema como um todo.

O melhor desempenho foi alcançado pelo esquema da figura 5.30, cujo tamanho médio da fila é 5.00. Entretanto, essa implementação é a mais dispendiosa de todas pois exige uma fila com capacidade oito vezes maior que o esquema anterior e requer uma taxa de serviço igualmente superior.

O esquema da figura 5.31 apresenta bons resultados, possuindo um tamanho de médio fila igual a 9.33, mas exige uma fila de capacidade 8 vezes maior que uma fila M/M/1.

A tabela 5.5, por sua vez, exibe os tamanhos médios de fila medidos para as mesmas configurações da tabela 5.2 durante um período de 1.000.000 de unidades de tempo. Tal qual o exemplo anterior, a variação nos resultados não permite qualquer conclusão definitiva. Entretanto a tabela permite concluir que o resultado médio esperado da rede de escalonamento é muito próximo ao obtido pelo modelo de fila M/M/8 da figura 5.31.

Política	**1 Conexão**	**2 Conexões**	**4 Conexões**	**6 Conexões**
Alternada	11.08	10.03	10.33	10.13
Inversamente	11.26	10.52	10.16	10.47
Menor	11.38	11.21	11.17	11.05
Randômica	10.95	10.32	10.36	10.29

Tabela 5.5: Número Médio de Processos no Sistema com Oito Nós em Função da Variação de Políticas e Topologias (1.000.000 ms)

Observe, entretanto, que os modelos markovianos de fila não prevêem dispêndio de tempo na troca de processos. Em outras palavras, para estes modelos o tempo gasto em uma decisão de escalonamento é inexistente. Esse fato não é verdadeiro no que diz respeito ao modelo *intrachip*, onde duas unidades de tempo são consumidas desde a solicitação até a recepção de um processo pelo processador.

5.2.3 Considerando a Troca de Processos no Modelo de Markov

No apêndice A determinamos o valor esperado para a variável aleatória Y, que representa o tempo médio gasto com execução e trocas de processos com um *quantum*

fixo:

$$E[Y] \approx \frac{1}{\mu} + t \times \left[e^{\mu \times quantum} - 1 \right] \times \frac{e^{-\mu \times quantum}}{(1 - e^{-\mu \times quantum})^2}$$

Sendo μ nesse equação a taxa média de execução de processos, podemos calcular o tempo médio de execução para a variável aleatória Y, considerando a taxa média de execução dada no início dessa seção, isto é, $\mu = \frac{1}{500}$, um tempo de *quantum*=100 e o tempo de troca de processo t=2 unidades de tempo:

$$E[Y] = 500 + 11 = 511$$

Portanto, ao considerar o tempo gasto com trocas de processos, o tempo médio de execução aumenta de 500 para 511 unidades de tempo. A tabela 5.6 mostra os resultados apresentados na tabela 5.4 ajustados para considerar o tempo de troca de processos.

	E[N]	E[R]
8 filas M/M/1	5.74 (por fila)	3444.00 (por fila)
1 fila M/M/1	5.74	430.50
1 fila M/M/8	10.12	759.00

Tabela 5.6: Medidas de Interesse para Modelos Markovianos com Ajuste

Com esse ajuste os resultados obtidos pela rede de escalonamento, que são mostrados na tabela 5.5, se aproximam ainda mais do comportamento de uma fila M/M/m. De fato, em alguns casos a diferença é sutil.

5.3 Uma Breve Comparação com o Linux

Como um recurso adicional realizamos a rápida comparação entre o desempenho obtido pela rede de escalonamento e o desempenho do escalonamento de processos no Linux. Obviamente essa comparação não é perfeita porque é impossível simular todas as interações a que um sistema operacional real está sujeito durante a execução de processos. Dessa forma, fatores tais como mecanismos de interrupção, memória virtual, *swapping* e outros não serão considerados nesse momento.

Para efeito de comparação, nós submetemos o sistema operacional a uma carga de trabalho artificial composta de processos cujas distribuições de tempos de nascimento e execução eram idênticas as utilizadas na simulação.

Conquanto não plenamente possível, buscamos ao máximo fornecer condições justas de comparação. Para tanto, os processos criados durante a simulação receberam uma prioridade de tempo real igual a 98, a segunda maior do sistema. Além

disso, eles foram classificados como processos em tempo real *round robin* com a intenção de reproduzir o comportamento simulado. Adicionalmente, outro processo em tempo real foi classificado como *FIFO* e recebeu a maior prioridade do sistema. Este processo tornou-se responsável pela criação dos primeiros, de acordo com uma distribuição de tempos de nascimento específica.

Para atenuar ainda mais possíveis interferências, o processo criador invocava `nanosleep()` sempre que detectava um longo período de espera até a próxima criação de processo. Por sua vez, os programas trabalhadores permaneciam em um laço até que seu consumo de processador atingisse o tempo estipulado pela distribuição de tempos de execução.

Para realizar o teste, utilizamos a distribuição Micro Core Linux. Esta distribuição é uma versão de Linux em modo texto que ocupa apenas 6MB de imagem e foi escolhida com o objetivo de amenizar influências negativas no teste.

Considerando que mesmo após realizar estas medidas o teste ainda estaria sujeito a efeitos adversos provenientes de outros processos e atividades do sistema operacional, julgamos por bem criar um parâmetro que permitisse atenuar o tempo de execução de um processo no sistema como forma de equilibrar possíveis disparidades.

A tabela 5.7 mostra o número médio de processo no Linux durante a execução da carga de trabalho artificial. A coluna percentual, indica que fração do tempo de execução simulado foi de fato atribuído a carga de trabalho. A segunda coluna exibe o número médio de processos, obtido a partir de 10 execuções independentes, respeitado o percentual associado. Na simulação, a mesma carga de trabalho produz um número médio de processos igual a 8.07, portanto apenas quando o tempo de execução dos processos submetidos ao Linux corresponde a 60% do tempo de execução da carga simulada, o desempenho do Linux é superior.

Percentual	**Número médio de Processos**
0.60	6.83
0.65	9.00
0.70	14.69
0.75	20.69
0.80	33.19
0.85	47.26
0.90	76.77
0.95	105.72
1.00	133.18

Tabela 5.7: Número Médio de Processo no Linux para a Carga de Trabalho Artificial

Esse resultado, entretanto, não é absoluto, pois como mencionamos anteriormente o Linux tem que lidar com outras questões que não apenas o escalonamento propriamente dito. Apesar disso, ele indica que ainda há espaço para aprimoramento

dos mecanismos de escalonamento nos computadores modernos.

Entre os motivos que levaram a rede de escalonamento a apresentar um desempenho superior podemos citar:

1. Com a utilização da rede, o escalonamento de processos pode ocorrer em paralelo com a execução de outros processos. Dessa forma, o processador pode iniciar a execução de um novo processo quase que instantaneamente, em vez de manipular estruturas de dados inserindo e removendo processos a fim de prosseguir.

2. Como o Linux utiliza o modelo de várias filas de execução, pode ocorrer desbalanceamento de carga entre os processadores. Este desbalanceamento é tratado pelo mecanismo de balanceamento de carga do sistema operacional e depende da execução de uma *thread* de migração em cada processador. A *thread* de migração é, então, ativada em situações específicas como a execução de um novo processo. Além disso, periodicamente o Linux verifica se o processador está balanceado em relação a seu domínio de escalonamento e, quando necessário, realiza o balanceamento. Todas essas ações adicionam um *overhead* ao escalonamento de processos que afeta negativamente o desempenho do sistema.

3. Finalmente, sempre que é necessário transferir um processo da fila de execução de um processador para a fila de execução de outro, é preciso adquirir o bloqueio (*lock*) dessas filas a fim de garantir exclusão mútua. Esta situação pode impedir que um processador obtenha um novo processo para execução, obstando assim seu progresso.

Capítulo 6

Conclusões

O objetivo desse capítulo é sintetizar as principais ideias abordadas neste trabalho e indicar possíveis pontos de melhorias.

6.1 Sumário

Neste trabalho, propusemos um mecanismo de escalonamento baseado em *hardware*. Nosso objetivo principal era observar seu comportamento e avaliar seu desempenho no que diz respeito escalonamento processos e balanceamento de carga.

Através de comparações entre resultados obtidos por simulação e dados analíticos, mostramos que o comportamento do modelo no que diz respeito as filas de processos é similar ao modelo de filas M/M/m.

Resultados empíricos sugerem que o mecanismo de escalonamento é capaz de escalar através da adição de novos nós no sistema. Além disso, aumentar a carga de processos de forma proporcional ao aumento no número de nós, não aumenta na mesma proporção o número médio de processos no sistema.

Comparações realizadas com o desempenho da execução de processos no sistema operacional Linux indicam que ainda há espaço para melhorias na eficiência do escalonador enquanto uma atividade de suporte.

Entre as principais contribuições do modelo está o fato de que como cada *chip* interage com apenas um processador a contenção no acesso aos processos mantidos por essa estrutura é muito pequena. Em um modelo M/M/m os processadores teriam de acessar a fila um de cada vez, em exclusão mútua, degradando o desempenho do sistema. Esse fato pode não ser tão relevante com 2 ou 4 núcleos de processamento, mas certamente o seria se centenas fossem utilizados.

6.2 Trabalhos Futuros

Uma proposta de trabalho futuro seria implementar o escalonamento *intrachip* em *hardware* ou em um mecanismo de simulação que permita reproduzir melhor o comportamento do sistema computacional e suas interações a fim de realizar uma comparação mais precisa.

Embora tenha sido abordado ligeiramente neste trabalho, o escalonamento de processos baseado em prioridades e suas implicações não foram estudados em profundidade, ficando como sugestão para um trabalho futuro.

Toda análise realizada nesta dissertação baseou-se na premissa de que o custo para troca de processos na rede era constante, isso não é verdadeiro para arquiteturas NUMA, onde o custo de migração de processos entre nós distintos pode variar. Deixamos essa vertente como sugestão para trabalhos futuros.

Este trabalho procurou investigar as vantagens em potencial do escalonamento *intrachip*. Entretanto, existem alguns pontos de melhoria nos conceitos explorados. Por exemplo, no modelo que implementado, a comunicação entre os *chips* ocorria em intervalos de tempo pequenos e regulares. Muitas vezes essa comunicação era desnecessária pois o estado de um *chip* não havia se alterado desde o instante de tempo anterior. Uma maneira simples de lidar com esse problema é realizar comunicação apenas quando o estado do *chip* for alterado.

Outra questão a ser abordada é que algumas políticas apresentadas nesse trabalho eram muito temerárias, à medida que o número de conexões aumenta, no que diz respeito ao envio de processos aos vizinhos. Este fato pode ser resolvido criando uma hierarquia de vizinhos, onde alguns são mais prioritários que outros em termos de recebimento de processos. Vamos apresentar a seguir dois métodos que tentam endereçar essa questão, embora não tenham sido verificados por simulação.

O primeiro, exige que a diferença entre o número de processo do *chip* para seus vizinhos seja progressivamente maior a fim de realizar o envio de processo. O primeiro vizinho necessitaria apenas possuir um processo a menos. Para o segundo vizinho seriam necessários dois processos a menos, e assim por diante. A variável i representa o índice de conexão do vizinho escolhido pela política de envio de processos.

```
1 Se a diferença entre o número de processos do chip e seu vizinho_i é maior
      ou igual a i e o chip possuir mais de um processo.
2   Envie um processo ao vizinho_i.
3 FimSe.
```

Listagem 6.1: Método de Priorização de Vizinhança por Diferenças

O próximo método é semelhante ao anterior, mas exige que o número de passos

para que o vizinho possa receber um processo seja progressivamente maior. Assim o primeiro vizinho receberia um processo no passo atual, isto é, imediatamente. O segundo vizinho receberia um processo do *chip* apenas se permanecesse com um número menor de processos por dois passos, e assim por diante.

```
1 Se, nos i passos anteriores, a diferença entre o número de processos do
      chip e seu vizinho_i é maior ou igual a um  e  o chip possuir mais de
      um processo.
2     Envie um processo ao vizinho_i.
3 FimSe.
```

Listagem 6.2: Método de Priorização de Vizinhança por Passos

Referências Bibliográficas

[1] MORE, G. E. "Cramming more Components onto Integrated Circuits", *Eletronics Magazine*, v. 38, n. 8, pp. 114–117, abr 1965.

[2] TUOMI, I. "The Lives and The Death of Moore's Law". , out 2002. Disponível em: <http://www.meaningprocessing.com/personalPages/tuomi/articles/TheLivesAndTheDeathOfMoore.pdf>.

[3] RUMELT, R. P. "Gordon Moore's Law". , mai 2003. Disponível em: <http://www.anderson.ucla.edu/faculty/dick.rumelt/Docs/Cases/MooresLaw.pdf>.

[4] *Moore's Law: An Intel Perspective*. Intel, 2005.

[5] GELSINGER, P. "Moore's Law - The Genius Lives On", *IEEE Solid-State Circuits Society Newsletter*, v. 20, n. 3, pp. 18–20, set 2006.

[6] LIDDLE, D. E. "The Wider Impact of Moore's Law", *IEEE Solid-State Circuits Society Newsletter*, v. 20, n. 3, pp. 28–30, set 2006.

[7] SNIR, M. "Making parallel programming synonymous with programming". In: *Hot Chips – A Symposium on High Performance Chips*, n. 21, 2009.

[8] ZHIRNOV, V. V., CAVIN, R. K., HUTCHBY, J. A., et al. "Limits to Binary Logic Switch Scaling—A Gedanken Model". In: *Proceedings of the IEEE*, v. 91, pp. 1934–1939, nov 2003.

[9] PATTERSON, D. A. "Overview of the UC Berkeley Par Lab". In: *Hot Chips – A Symposium on High Performance Chips*, n. 21, 2009.

[10] GAUDIN, S. "Intel Shows Green 48-Core Chip". , dez 2009. Disponível em: <http://www.pcworld.com/article/183589/intel_shows_green_48core_chip.html>.

[11] "Third International Workshop on Network on Chip Architectures: General Information". , dez 2010. Disponível em: <http://nocarc.diit.unict.it/>.

[12] ATIENZA, D., ANGIOLINI, F., MURALI, S., et al. "Network-on-Chip Design and Synthesis Outlook", *Integration: The VLSI Journal*, v. 41, pp. 340–359, 2008.

[13] GRATZ, P., KIM, C., MCDONALD, R., et al. "Implementation and Evaluation of On-Chip Network Architectures". In: *IEEE International Conference on Computer Design*, pp. 477–484, out 2006.

[14] SALMINEN, E., KULMALA, A., HÄMÄLÄINEN, T. D. "On network-on-chip comparison". In: *10th Euromicro Conference On Digital Design Architectures, Methods and Tools*, pp. 503–510, ago 2007.

[15] MORENO, E. I. *Modelagem, Descrição e Validação de Redes Intrachip no Nível de Transação*. Tese de Mestrado, Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre, 2004.

[16] NÁCUL, A. C., REGAZZONI, F., LAJOLO, M. "Hardware Scheduling Support in SMP Architectures". In: *Design, Automation and Test in Europe*, pp. 642–647, abr 2007.

[17] KUACHAROEN, P., SHALAN, M. A., III, V. J. M. "A Configurable Hardware Scheduler for Real-Time Systems". In: *in Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, pp. 96–101. CSREA Press, 2003.

[18] SILBERCHATZ, A., GALVIN, P. B. *Sistemas Operacionais: Conceitos*. 5ª ed. São Paulo, SP, Brasil, Prentice Hall, 2000.

[19] FLOWER, G. "Processes on Linux and Windows NT". , dez 1997. Disponível em: <`http://linuxgazette.net/issue23/flower/page1.html`>.

[20] *Multithreaded Programming Guide*. Sun Microsystems, 2008.

[21] KAWAGUCHI, K. *A multithreaded software model for backpropagation neural network applications*. Tese de Mestrado, University of Texas em El Paso, El Paso, Texas, USA, 2000.

[22] TANENBAUM, A. S., WOODHULL, A. S. *Sistemas Operacionais: Projeto e Implementação*. 2ª ed. Porto Alegre, RS, Brasil, Bookman, 2000.

[23] HENNESSY, J. L., PATTERSON, D. A. *Arquitetura de Computadores: Uma Abordagem Quantitativa*. 3ª ed. Rio de Janeiro, RJ, Brasil, Campus, 2003.

[24] *The Evolution Of a Revolution: Explore The Intel Technology Innovations That Have Changed The World*. Intel, 2007.

[25] BAUTISTA, J. "Tera-scale Computing and Interconnect Challenges – 3D Stacking Considerations". In: *Hot Chips – A Symposium on High Performance Chips*, n. 20, 2008.

[26] CORRÊA, M., ZORZO, A., SCHEER, R. "Operating System Multilevel Load Balancing". In: *ACM Symposium On Applied Computing*, pp. 1467–1471, Dijon, França, abr 2006.

[27] BOVET, P. D., CESATI, M. *Understanding the Linux Kernel*. 3ª ed. Sebastopol, CA, USA, O'Reilly, 2005.

[28] SALZMAN, P. J., BURIAN, M., POMERANTZ, O. *The Linux Kernel Module Programming Guide*, 2007.

[29] WELCH, B. J. "Impact of Load Imbalance on Processors with Hyper-Threading Technology". , out 2008. Disponível em: <http://software.intel.com/en-us/articles/impact-of-load-imbalance-on-processors-with-hyper-threading-technology/>.

[30] JONES, M. T. "Inside the Linux 2.6 Completely Fair Scheduler", *IBM Developer Works*, 2009.

[31] BOVET, P. D., CESATI, M. *Understanding the Linux Kernel*. 1ª ed. Sebastopol, CA, USA, O'Reilly, 2000.

[32] LINDSLEY, R. "What's New in the 2.6 Scheduler?" , mar 2004. Disponível em: <http://www.linuxjournal.com/article/7178>.

[33] LOVE, R. *Linux Kernel Development*. 2ª ed. Indianapolis, IN, USA, Novell Press, 2005.

[34] S.SIDDHA, V. PALLIPADI, A. M. "Chip Multi Processing Aware Linux Kernel Scheduler". In: *Proceedings of the Linux Symposium*, pp. 329–339, Ottawa, ON, Canada, jul 2006.

[35] BLIGH, M. J., DOBSON, M., HART, D. "Linux on NUMA Systems". In: *Proceedings of the Linux Symposium*, pp. 89–101, Ottawa, ON, Canada, Jul 2004.

[36] "CFS Scheduler". , mai 2010. Disponível em: <http://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt>.

[37] PABLA, C. S. "Completely Fair Scheduler". , ago 2009. Disponível em: <http://www.linuxjournal.com/magazine/completely-fair-scheduler>.

[38] REED, D. *A Balanced Introduction to Computer Science*. 1ª ed. Nova Jersey, NJ, USA, Prentice Hall, 2004.

[39] EIGENMANN, R., LILJA, D. J. "Von Neumman Computers". , jan 1998. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.78.4336&rep=rep1&type=pdf>.

[40] LAIRD, A. "Von Neumman Architecture". , jan 2009. Disponível em: <http://www.alexlaird.net/projects/collegiate/springsemester2009/files/topicpaper3-thevonneumannbottleneck.pdf>.

[41] COPELAND, J. "A Brief History of Computing". , jun 2000. Disponível em: <http://www.alanturing.net/turing_archive/pages/Reference%20Articles/BriefHistofComp.html#ACE>.

[42] TURING, A. M. "On Computable Numbers, With an Application to the Entscheidungsproblem". In: *Proceedings of the Londom Mathematical Society*, pp. 230–265, London, UK, mai 1936.

[43] LUKOFF, B. *From Dits to Bits: A Personal History of the Eletronic Computer*. 1ª ed. Porland, OR, USA, Robotic Press, 1979.

[44] DEVLIN, K. "John Von Neumann: The Father of the Modern Computer". , dez 2003. Disponível em: <http://www.maa.org/devlin/devlin_12_03.html>.

[45] SONG, Y. *The Implementation of Sequential and Parallel Algorithms for Saving Almost Block Bidiagonal Systems*. Tese de Mestrado, The University of British Columbia, Vancouver, Canada, 1993.

[46] MÜLLER, B., HAHN, M. "Parallel Processing - The Example of Automatic Relative Orientation". In: *Proceedings of XVII Congress of International Society for Photogrammetry and Remote Sense*, pp. 623–630, Washington, DC, USA, ago 1992.

[47] DIJKSTRA, E. W. "A Review of the 1977 Turing Award Lecture by John Backus". , jul 2006. Disponível em: <http://userweb.cs.utexas.edu/~EWD/transcriptions/EWD06xx/EWD692.html>.

[48] BLACK, D. C., DONAVAN, J. *SystemC: From The Ground Up*. 1ª ed. Boston, MA, USA, Kluwer Academic Publishers, 2004.

[49] TRIVEDI, K. S. *Probability and Statistics with Reliability Queuing, and Computer Science Applications*. 2ª ed. New York, NY, USA, Wiley-Interscience, 2002.

Apêndice A

Adicionando o Custo da Troca de Processos do Escalonamento com *Quantum* Fixo ao Modelo de Markov

Neste momento, vamos adicionar o custo imposto pela troca de processos ao modelo de Markov. Para tanto vamos considerar o tempo gasto entre troca de processos como fazendo parte do tempo de execução do processo que chega ao processador.

Seja X uma variável aleatória exponencial que descreve o tempo de execução dos processos. Podemos definir a variável aleatória Y, que descreve o tempo gasto com execução e trocas dos processos, da seguinte maneira:

$$Y = X + t \times \left\lceil \frac{X}{quantum} \right\rceil$$

Observe que o termo $\left\lceil \frac{X}{quantum} \right\rceil$ descreve o número de trocas a que um processo com tempo de execução x está sujeito. A constante t descreve o tempo gasto em uma troca de processo.

Como Y é uma função de X podemos calcular seu valor esperado aplicando $E[Y] = \int_{-\infty}^{\infty} Y(x) \times f_x(x)\, dx$. Essa integral pode, então, ser dividida em duas partes :

$$E[Y] = \int_{-\infty}^{\infty} x\mu e^{-\mu x}\, dx + \int_{-\infty}^{\infty} t \left\lceil \frac{x}{quantum} \right\rceil \mu e^{-\mu x}\, dx$$

A primeira integral corresponde a expressão do valor esperado para uma variável exponencial, cujo resultado é $\frac{1}{\mu}$.

A segunda integral é mais complicada pelo fato de possuir uma função teto em

sua formação. Observe entretanto que o valor do termo $\lceil \frac{x}{quantum} \rceil$ só varia entre intervalos. Por exemplo. supondo que o *quantum* seja definido em 100 unidades de tempo, o valor da expressão para $x = 0, 1, 2, ..., 99$ é idêntica, assim como para $x = 100, 101, 102, , ..., 199$. Portanto, dentro de um intervalo com o tamanho do *quantum* o valor da expressão é constante.

Podemos então calcular a integral para um intervalo arbitrário da seguinte forma:

$$\int_{(x-1)\times quantum}^{x\times quantum} k\mu e^{-\mu x}\, dx$$

$$k\int_{(x-1)\times quantum}^{x\times quantum} \mu e^{-\mu x}\, dx$$

$$k\left[-e^{-\mu x}\Big|_{(x-1)\times quantum}^{x\times quantum}\right]$$

$$k\left[e^{-\mu(x-1)\times quantum} - e^{-\mu x\times quantum}\right]$$

$$k\left[e^{-\mu x\times quantum}\times\left[e^{\mu\times quantum} - 1\right]\right]$$

Observe que em função da mudança nos limites de integração, dentro do intervalo $\{(x-1)\times quantum, x\times quantum\}$ o valor de k é igual a x. Dessa forma podemos então aproximar o valor da segunda integral.

$$\int_{-\infty}^{\infty} t\left\lceil\frac{x}{quantum}\right\rceil \mu e^{-\mu x}\, dx \approx t\times\sum_{x=1}^{\infty} x\left[e^{-\mu x\times quantum}\times\left[e^{\mu\times quantum} - 1\right]\right]$$

$$\int_{-\infty}^{\infty} t\left\lceil\frac{x}{quantum}\right\rceil \mu e^{-\mu x}\, dx \approx t\times\left[e^{\mu\times quantum} - 1\right]\sum_{x=1}^{\infty} x e^{-\mu x\times quantum}$$

$$\int_{-\infty}^{\infty} t\left\lceil\frac{x}{quantum}\right\rceil \mu e^{-\mu x}\, dx \approx t\times\left[e^{\mu\times quantum} - 1\right]\sum_{x=1}^{\infty} x(e^{-\mu\times quantum})^x$$

Sabendo que $\sum_{x=1}^{\infty} xk^x = \frac{k}{(1-k)^2}$, obtemos:

$$\int_{-\infty}^{\infty} t\left\lceil\frac{x}{quantum}\right\rceil \mu e^{-\mu x}\, dx \approx t\times\left[e^{\mu\times quantum} - 1\right]\times\frac{e^{-\mu\times quantum}}{(1 - e^{-\mu\times quantum})^2}$$

Dessa forma:

$$E[Y] = \int_{-\infty}^{\infty} x\mu e^{-\mu x}\, dx + \int_{-\infty}^{\infty} t\left\lceil\frac{x}{quantum}\right\rceil \mu e^{-\mu x}\, dx$$

$$E[Y] \approx \frac{1}{\mu} + t \times \left[e^{\mu \times quantum} - 1 \right] \times \frac{e^{-\mu \times quantum}}{(1 - e^{-\mu \times quantum})^2}$$