UNIVERSIDADE FEDERAL FLUMINENSE
INSTITUTO DE COMPUTAÇÃO
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Leonardo Gomes Marcello
Leonardo Sant'Anna do Valle Dias

SISTEMA OPERACIONAL LINUX EM ARQUITETURAS MULTICORE

Niterói
2012

Leonardo Gomes Marcello

Leonardo Sant'Anna do Valle Dias

SISTEMA OPERACIONAL LINUX EM ARQUITETURAS MULTICORE

Monografia apresentada ao Departamento de Ciência da Computação da Universidade Federal Fluminense como parte dos requisitos para obtenção do Grau de Bacharel em Ciência da Computação.

Orientadora: Maria Cristina Silva Boeres

Niterói

2012

Leonardo Gomes Marcello

Leonardo Sant'Anna do Valle Dias

SISTEMA OPERACIONAL LINUX EM ARQUITETURAS MULTICORE

Monografia apresentada ao Departamento de Ciência da Computação da Universidade Federal Fluminense como parte dos requisitos para obtenção do Grau de Bacharel em Ciência da Computação.

Aprovado em fevereiro de 2012

BANCA EXAMINADORA

_____

Profa. MARIA CRISTINA SILVA BOERES, Ph.D.
Orientadora
UFF

_____

Profa. ALINE DE PAULA NASCIMENTO, D.SC.
UFF

_____

Prof. JOSÉ VITERBO FILHO, D.SC.
UFF

Niterói

2012

RESUMO

A arquitetura multicore surgiu como uma evolução em arquiteturas de computadores com o objetivo de aumentar o desempenho de execução de aplicações atuais através da exploração de paralelismo de execução. Uma máquina multicore possui mais de um núcleo de processamento e tais sistemas possibilitam a aplicação de técnicas que exploram o paralelismo a nível de threads, aliada às características existentes nas arquiteturas modernas.

Para gerenciar e explorar todos os benefícios de tais arquiteturas, os sistemas operacionais e aplicações foram adaptados e até reprojetados para aumentar o desempenho de execução. Um exemplo de sistema operacional estudado neste projeto é o sistema operacional Linux, que implementa técnicas de gerenciamento e escalonamento de processos com o intuito de maximizar desempenho.

Este trabalho tem como objetivo mostrar os motivadores para a evolução das arquiteturas, quais foram as alternativas desenvolvidas, como o sistema operacional Linux se adequou a essas mudanças e ilustrar, através de testes, como pode se tirar proveito da arquitetura multicore.

Palavras Chave:

arquitetura *multicore*, escalonamento de threads, Linux, multiplicação de matrizes paralela

ABSTRACT

The multicore architecture has emerged as an evolution in computer architectures in order to increase the performance of current applications by exploiting parallel execution. A multicore machine has more than one processing core, and such systems allow the use of techniques that exploit thread level parallelism, combined with the existing features in modern architectures.

To manage and exploit all the benefits of such architectures, operating systems and applications have been adapted and even redesigned to increase the execution performance. An example of operating system studied in this project is the Linux operating system, which implements management techniques and process scheduling in order to maximize performance.

This work aims to show the motivations for architecture evolution, the alternatives developed, how the Linux operating system adapted itself to these changes and illustrate, through some experiments, how to take advantage of the multicore architecture.

Keywords:

multicore architecture, thread scheduling, Linux, parallel matrix multiplication

LISTA DE ACRÔNIMOS

CPU: Central Processing Unit
ILP: Instruction Level Parallelism
SMP: Symmetric Multi-Processing
TLP: Thread-level parallelism
ULA: Unidade Lógica e Aritmética
CMP: Chip Multiprocessor
SMT: Simultaneous Multiprocessing
VLSI: Very Large Scale Integration
ACPI: Advanced Configuration and Power Interface
PPE: Power Processor Element
SPEs: Synergistic Processing Elements
EIB: Element Interconnect Bus
RISC: Reduced Instruction Set Computer
LS: Local Store
SRAM: Static Random Access Memory
CFS: Completely Fair Scheduler
API: Application Programming Interface
POSIX: Portable Operating System Interface

Às nossas famílias e amigos.

AGRADECIMENTOS

À professora Cristina Boeres,

pela sua orientação,

Aos professores do Instituto de

Computação,

pelo conhecimento que nos foi transmitido,

Aos nossos pais e familiares,

pelo apoio e carinho.

SUMÁRIO

LISTA DE FIGURAS

1 INTRODUÇÃO

Cada vez mais as aplicações exigem um processamento elevado, ocasionando demandas por hardwares mais eficientes. Hoje em dia existe uma preocupação constante com a evolução dos hardwares, principalmente como seria essa evolução e até mesmo quais seriam seus limites.

Em 1965, Gordon Earle Moore, cofundador da Intel Corporation, através de uma afirmação em um artigo científico [1], descreveu a tendência da tecnologia de processadores na área de computação, através da Lei de Moore, a qual diz que com o ritmo de evolução da tecnologia observado, é possível duplicar o número de transistores numa mesma área de silício aproximadamente a cada 18 meses. Esta lei foi baseada na observação do avanço da fotolitografia, permitindo a produção de transistores cada vez menores.

Com o passar do tempo, Moore aumentaria sua projeção. Em 1975, afirmou que o número de transistores dobraria a cada 2 anos, como pode ser observado na Figura 1.1, onde a linha azul representa esta nova projeção. Em novembro de 2003, quatro pesquisadores da Intel [2] publicaram um trabalho onde mostravam que a utilização dessa Lei estava com seus dias contados.

Em 2005, Moore afirmou que a sua lei não seria válida para sempre, pois em um determinado momento o limite físico seria alcançado [2]. Os transistores alcançariam o tamanho atômico, fazendo com que os mesmos não pudessem mais ser reduzidos. Afirmou também que teríamos ainda 10 a 20 anos antes que o limite fosse alcançado.

Mais recentemente, com a necessidade de adquirir uma solução viável para o problema do tamanho dos transistores, surgiram algumas soluções e entre elas as arquiteturas multicores. Nessa arquitetura, dois ou mais núcleos são postos em um mesmo chip, onde os núcleos possuem memória cache própria, na maioria das vezes compartilhada, e compartilham memória principal.

Com isto, os Sistemas Operacionais foram especificados para serem compatíveis com a nova realidade, ou seja, preparados para utilizar sabiamente essa nova arquitetura, trazendo benefícios aos seus usuários. Neste trabalho, veremos os motivadores para esta mudança, vantagens tanto em nível de hardware

quanto em nível de software da arquitetura *multicore*. Também iremos ilustrar através de testes como pode ser vantajosa a utilização desta arquitetura.

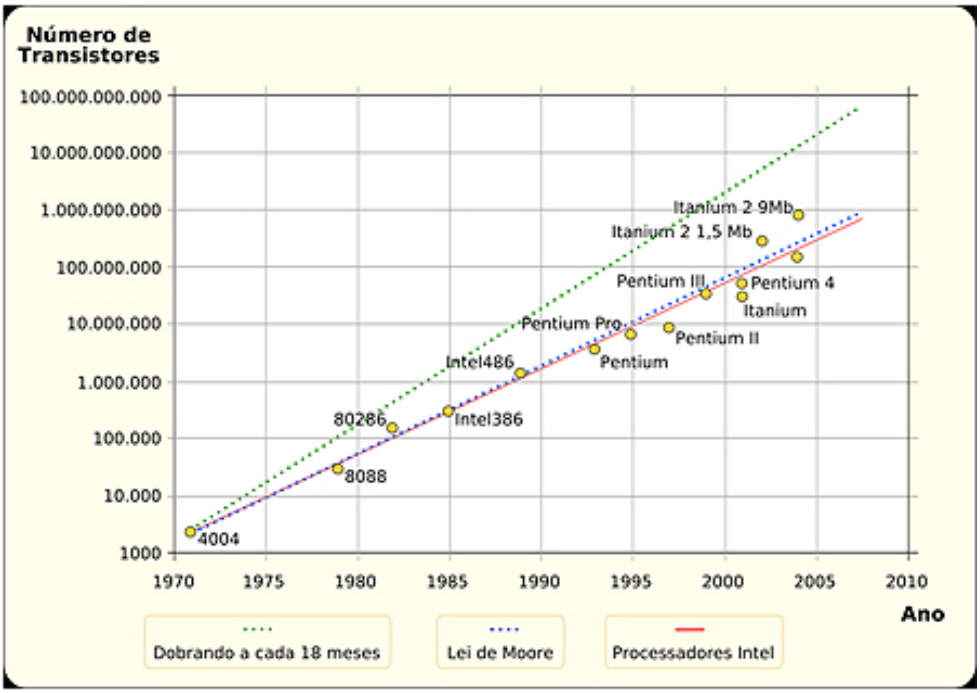

Figura 1.1 Gráfico ilustrando a Lei de Moore. [3]

1.1 OBJETIVO DO TRABALHO

A motivação para o trabalho foi estudar a evolução das arquiteturas de computadores e como elas abordaram a questão do paralelismo e como os sistemas operacionais lidaram com essas mudanças. Também, através de um experimento, medir o ganho de desempenho no processamento de uma aplicação a medida em que a dividimos em partes que podem ser executadas simultaneamente em um computador que utiliza a arquitetura multicore.

1.2 ORGANIZAÇÃO DO TEXTO

O texto deste trabalho se divide da seguinte maneira: o Capítulo 2 possui a explanação das limitações das arquiteturas *single-core*, os motivos que impulsionaram a busca de novas tecnologias e as alternativas encontradas para resolver este problema. No capítulo 3 encontra-se a explicação sobre as arquiteturas

multicore, tanto homogênea quanto heterogênea. O Capítulo 4 apresenta o estudo realizado sobre o funcionamento dos escalonadores do sistema operacional Linux, desde a versão do kernel 2.4 até a versão 2.6.24. O Capítulo 5 é composto pelos dados dos testes realizados, assim como os resultados obtidos. No Capítulo 6 encontramos as avaliações dos resultados obtidos através dos testes realizados.

2 MOTIVAÇÕES PARA MÁQUINAS MULTICORE

Em computação, o desenvolvimento tecnológico deve acompanhar a demandas de processamento que são cada vez maiores. Os processadores *single-core* começaram a encontrar obstáculos como a limitação de velocidade de processamento, fazendo com que o desenvolvimento de uma nova tecnologia fosse necessário. Neste capítulo será descrito quais eram os limitadores da arquitetura *single-core* e entender melhor os motivos que impulsionaram essa mudança.

2.1 LIMITES FÍSICOS

Atualmente, a necessidade de maior poder computacional se torna mais evidente, considerando, por exemplo, aplicações gráficas que requerem um alto poder computacional e jogos cada vez mais sofisticados [4]. Para atender a esta crescente demanda, a estratégia que vinha sendo utilizada era aumentar o número de transistores em um mesmo chip, adquirindo assim processadores com frequências mais elevadas. Contudo, através do trabalho publicado em 2003 por quatro pesquisadores da Intel [5], foi exposto que esta linha de pensamento estava com seus dias contados. Assim, surge a necessidade de uma tecnologia alternativa capaz de suprir as limitações arquiteturais e tecnológicas.

No início da década anterior, os transistores possuíam uma camada de silício de 90nm e porta de 50nm, como pode ser visto na Figura 2.1, sendo que atualmente, a Intel possui transistores com camada de silício de 22nm [6]. Com a largura da porta diminuindo, dreno e fontes do transistor ficam cada vez mais próximos e com a diminuição dos mesmos chegaríamos a um ponto em que o transistor deixaria de ser um dispositivo de processamento confiável. Por exemplo, se a largura da porta chegar à 5nm, a probabilidade da corrente passar sem tensão aplicada é de 50%. O tamanho dos componentes do transistor não é o maior dos problemas. O grande problema seria o número dos mesmos em uma mesma área. Com isso os processadores atingiriam temperaturas maiores e se as mesmas não fossem dissipadas para o ambiente, o chip literalmente derreteria.

Tendo em mente esses problemas e limitações, a comunidade da área chegou à conclusão de que seria necessária uma nova alternativa para suprir as limitações da arquitetura *single-core*.

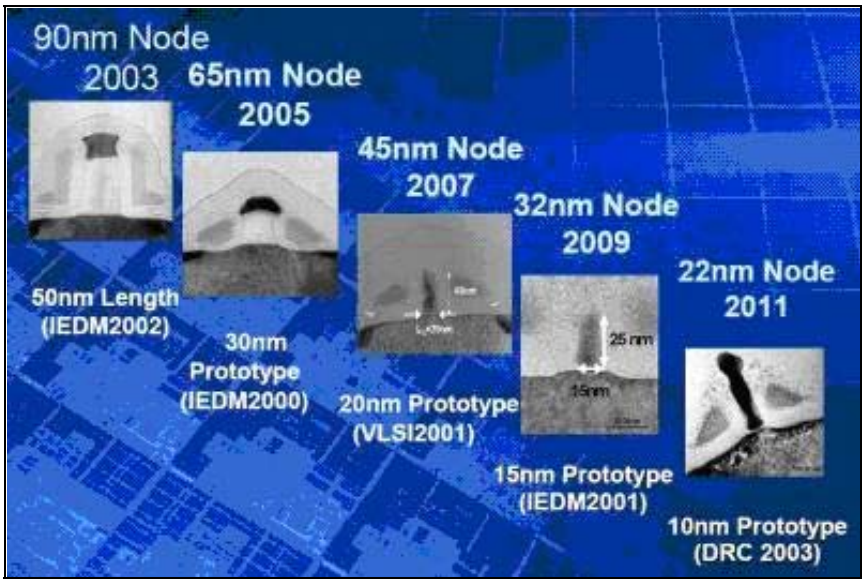

Figura 2.1 Protótipos de processadores Intel. [5]

2.2 TÉCNICAS PRINCIPAIS QUE VISAM MELHORIA DE DESEMPENHO DE EXECUÇÃO

As técnicas abaixo foram desenvolvidas durante a evolução das arquiteturas de processadores, com o objetivo de ultrapassar as limitações encontradas, melhorando a execução de aplicações.

i. *Aumento da memória cache,*
ii. *Múltiplas unidades de processamento*
iii. *Paralelismo à nível de instrução - ILP*
iv. *Arquiteturas superescalares*
v. *Arquiteturas SMP*
vi. *Paralelismo à nível de threads - TLP*
vii. *Hyper-threading*

2.2.1 AUMENTO DA MEMÓRIA CACHE

A memória cache é um tipo de memória rápida utilizada para armazenar ambos dados e códigos utilizados quando executando uma aplicação, evitando assim o acesso frequente à memória principal. Em geral, os processadores possuem até três níveis de cache: L1, L2 e L3, sendo que a primeira possui uma velocidade de acesso mais rápida que as outras. Um problema da memória cache é que seu preço é elevado quando comparado ao da memória principal.

2.2.2 MÚLTIPLAS UNIDADES DE PROCESSAMENTO

Uma das soluções para o problema da limitação do desempenho da arquitetura single-core é a utilização de múltiplas unidades de processamento, dentre as quais tarefas a serem processadas podem ser divididas entre as unidades. Neste caso, distribuir igualmente as tarefas entre os processadores já é um bom modo de se começar e, com o atual desenvolvimento da tecnologia, é possível unir várias unidades processadoras com menor custo no que é chamado multiprocessador. Alguns exemplos de tais máquinas são os supercomputadores K Computer do Japão e Tianhe-1A da China, que são os 2 computadores mais rápidos do mundo atualmente [7].

2.2.3 PARALELISMO EM NÍVEL DE INSTRUÇÃO – ILP

Para aumentar o desempenho de execução em processadores atuais, paralelismo em nível de instrução é explorado em tempo de compilação e execução, fazendo com que operações como leitura e escrita de memória, adição de inteiros, e multiplicações de ponto flutuante sejam executadas em paralelo. No nível de instruções, o paralelismo é explorado por arquiteturas de processadores capazes de executar instruções ou operar dados em paralelo [10].

Os processadores atuais que utilizam tais técnicas são capazes de avaliar em tempo real o código do programa, conseguindo assim determinar qual instrução

poderá ser processada com segurança, independente da ordem de execução ou de outras instruções.

A utilização de ILP envolve técnicas que visam aumentar o número de instruções executadas por ciclo, otimizando a utilização do processador, explorando a aplicação de pipeline. O pipeline se baseia em executar partes de instruções em um mesmo ciclo. Para isto, a instrução é dividida em N passos e, a cada passo, o valor resultante desta etapa é guardado para ser usado na próxima etapa. Estes valores são guardados em registradores chamados de latches [8,9].

Com esta técnica, as instruções podem ser sobrepostas, fazendo com que em cada ciclo do processador um passo de cada uma delas seja executado, evitando que algum dos componentes do processador fique ocioso.

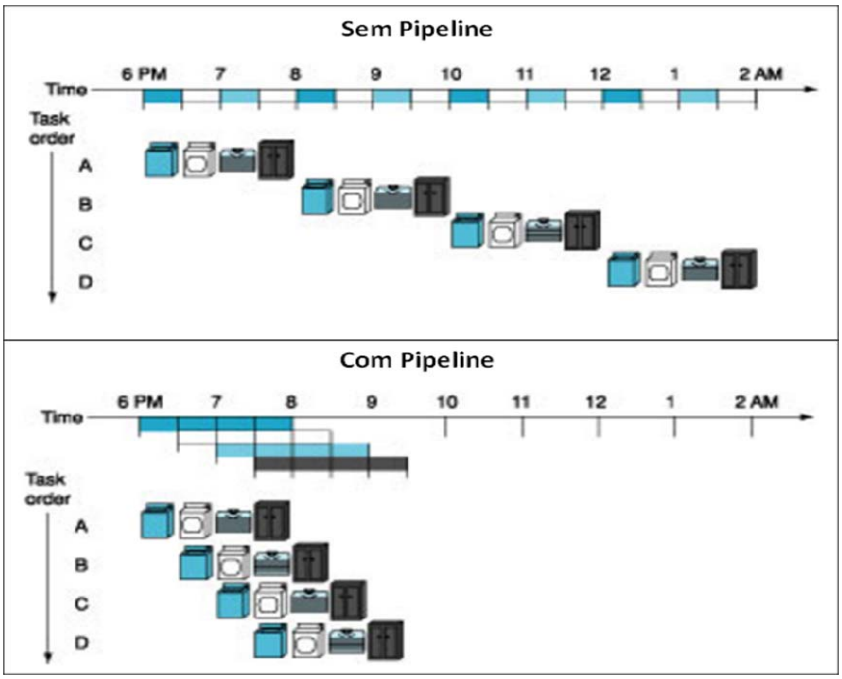

Figura 2.2 Analogia à técnica de pipeline em uma lavanderia

Podemos ver na Figura 2.2 que, quando executamos o processo sem pipeline, os pedidos são executados um a um, ou seja, enquanto o pedido A está sendo atendido, os outros estão aguardando a completude do mesmo. Já na execução com pipeline, utilizam-se eficientemente os recursos, ou seja, nenhum dos recursos da lavanderia fica ocioso, sendo que em determinado momento, um recurso não pode estar alocado para mais de um pedido.

O pipeline pode reduzir a duração do ciclo de um processador, aumentando assim, a taxa de rendimento das instruções, ou seja, o número de instruções executadas em um determinado intervalo de tempo, porém ele aumenta a latência do processador em pelo menos, a soma de todas as latências dos latches. A latência de um pipeline é o tempo que uma única instrução leva para passar por todos os estágios do pipeline [8]. Quanto maior o número de estágios de pipeline, maior a latência, ocasionada pela soma das latências dos latches.

2.2.4 ARQUITETURAS SUPERESCALARES

Em meados da década de 1980 os processadores superescalares começaram a ser projetados. O objetivo dos projetistas era romper a barreira do pipeline de uma única instrução executada por ciclo de clock [10].

Em uma arquitetura superescalar, várias instruções podem ser iniciadas ao mesmo tempo e executadas independentemente, ou seja, diferentemente do pipeline, onde é permitido que diversas instruções sejam executadas ao mesmo tempo, mas elas devem estar obrigatoriamente em estágios diferentes do pipeline. Como vimos anteriormente, utilizando o pipeline conseguimos aumentar o desempenho do processador fazendo com que seus recursos não fiquem ociosos, porém ele aumenta a latência de cada instrução (quanto mais estágios de pipeline, maior o tempo para o término do ciclo) devido à adição de componentes para dar suporte ao pipeline [10].

As arquiteturas superescalares possuem todas as características do pipeline, porém, além disso, as instruções podem estar em um mesmo estágio simultaneamente. Com isso é possível iniciar múltiplas instruções no mesmo ciclo de *clock*, como pode ser observado na figura 2.3, onde as instruções 1 e 2 são executadas simultaneamente. Para conseguir isso, as arquiteturas superescalares detectam quais instruções podem ser executadas em paralelo em tempo de compilação ou de execução.

Uma característica importante das arquiteturas superescalares é a capacidade de disparar processos de inicialização da execução de várias instruções,

inclusive fora de ordem. [11]. Porém, esse recurso requer um alto nível de complexidade de hardware para controle e tratamento de interrupções.

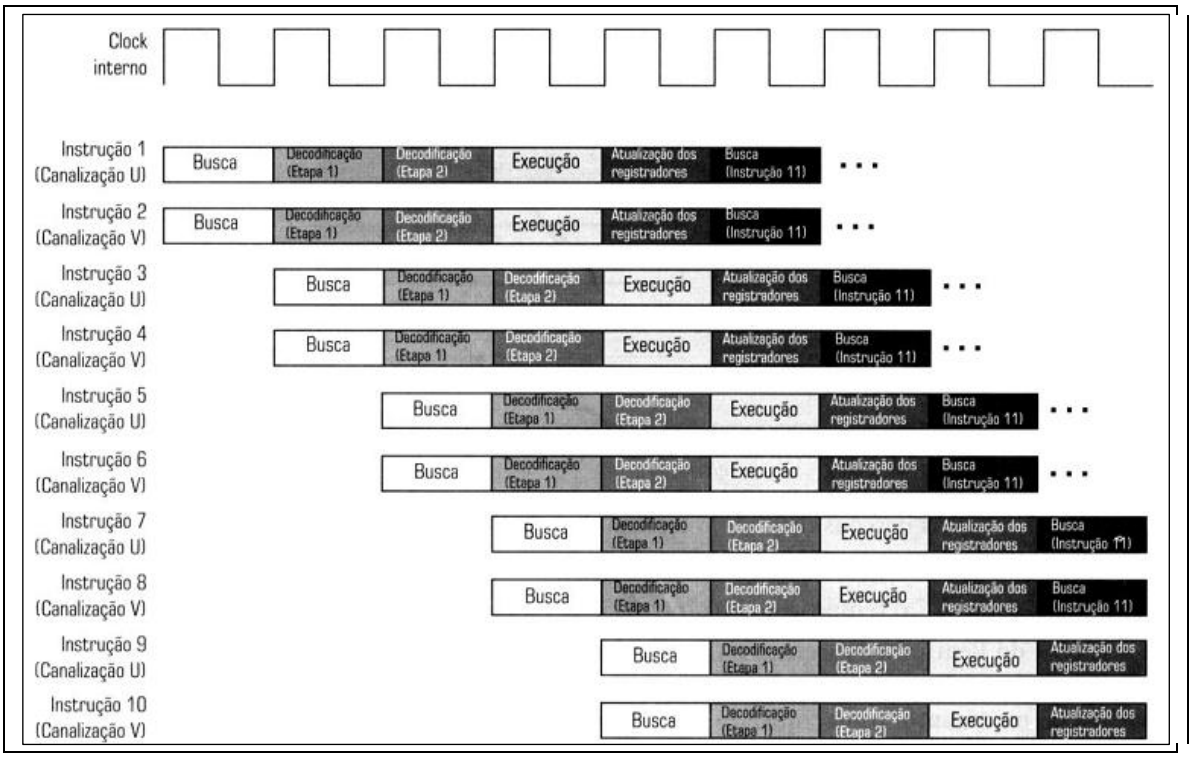

Figura 2.3 Pipeline em Arquiteturas Superescalares

2.2.5 ARQUITETURAS SMP

As arquiteturas SMP oferecem uma solução para integração de duas ou mais unidades de processamento que compartilhem uma área de memória central, com o objetivo de aumentar o desempenho de execução de múltiplos processos. Todos os processadores são completos, com ULA (Unidade Lógica e Aritmética), registradores e cache, e compartilham memória principal.

Nessa arquitetura, os processadores não são dedicados a um determinado tipo de tarefa (não são especializados), ou seja, todos os processadores são capazes de realizar as mesmas funções. Esse tipo de estrutura torna possível para qualquer um dos processadores executarem uma determinada parte de um programa, e ainda, que o computador possa continuar funcionando mesmo se algum dos núcleos de processamento seja danificado, embora perdendo desempenho [13].

Uma outra vantagem é que o sistema pode ter seu desempenho incrementado ao adicionar-se novos processadores.

O gerenciamento dos componentes de um sistema construído sobre uma arquitetura SMP é de responsabilidade do sistema operacional [10], e deve sempre buscar manter todos os seus núcleos de processamento trabalhando o maior tempo possível. Um uso incorreto dessa arquitetura poderia tornar o sistema menos poderoso do que um sistema que contasse com apenas um dos processadores, com mesmo poder de processamento [14].

Além do gerenciamento, é importante que as aplicações estejam preparadas para que possam ser executadas em paralelo, tirando assim maior proveito dos processadores. Para tanto, as aplicações devem ser capazes de se dividir em threads ou em processos.

Como cada processador tem a sua própria memória cache, as arquiteturas SMP devem trazer soluções para que seja mantida a coerência dos dados copiados da memória principal pois pode acontecer de vários processadores buscarem para sua respectiva memória cache cópias do conteúdo de um mesmo endereço e, quando esses valores são atualizados, corre-se o risco de existirem cópias de um mesmo endereço de memória com valores diferentes entre os processadores [15]. Entre as possíveis soluções para esse problema temos: bloquear a cópia de dados para cache em endereços de memória compartilhados para escrita; ou enviar uma mensagem a todos os processadores cada vez que um conteúdo da cache é alterado, invalidando ou atualizando os valores de outras possíveis cópias do mesmo endereço. Ambas as soluções reduzem o desempenho do sistema.

2.2.6 PARALELISMO EM NÍVEL DE THREAD – TLP

Paralelismo em nível de thread é, assim como o paralelismo em nível de instrução, uma outra forma de se identificar threads independentes que podem ser executadas em paralelo. Esse nível de paralelismo pode ser alcançado em arquiteturas que contam com várias unidades de processamento e consiste em dividir uma determinada tarefa em threads que podem ser executadas em paralelo entre elas. [16]

Tanto a arquitetura CMP (Chip Multiprocessor) quanto a técnica SMT Simultaneous Multithreading) permitem a utilização de paralelismo em nível de thread, como pode ser visto na Figura 2.4.

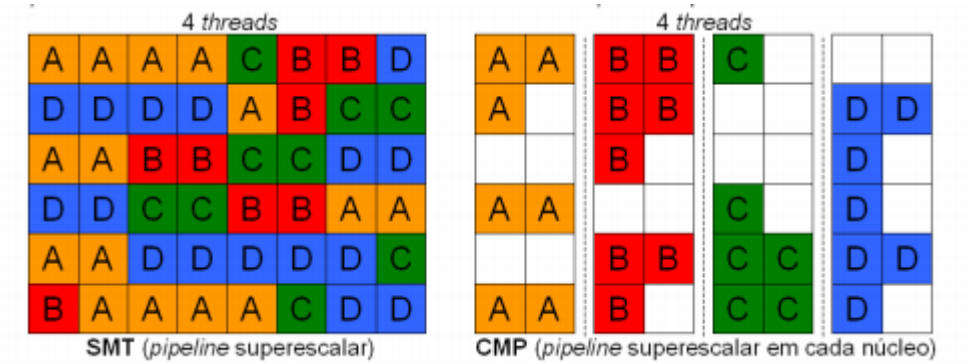

Figura 2.4 SMT x CMP [17]

Na arquitetura CMP, o paralelismo em nível de threads é obtido através da utilização de vários núcleos de processamento, instalados em um único chip, disputando recursos compartilhados como, por exemplo, níveis de memórias cache. Cada um dos núcleos pode possuir pipelines superescalares, fazendo com que além do paralelismo em nível de threads, consiga-se explorar o paralelismo em nível de instrução [18]. Na Figura 2.4 (CMP), cada um dos núcleos executa uma thread e cada thread possui duas instruções sendo executadas paralelamente, sendo assim um exemplo de implementação de CMP utilizando ILP e TLP. Além disso, outras vantagens de ter vários núcleos em um único chip são a economia de energia, espaço e a comunicação mais rápida entre os núcleos [19].

A técnica SMT possibilita vários contextos de execução ativos em um único ciclo de processamento, ou seja, permite executar em um mesmo ciclo várias threads de diferentes processos em um mesmo núcleo de processamento simultaneamente, como pode ser visto na Figura 2.4. Os recursos do processador são melhor aproveitados, pois as tarefas podem compartilhar recursos entre si [20].

Com esse compartilhamento de recursos entre threads em um processador no mesmo ciclo, paralelismo em nível de thread é essencialmente convertido em paralelismo em nível de instrução. Processadores SMT podem explorar as vantagens desses dois tipos de paralelismo, obtendo um significativo ganho de desempenho [16].

2.2.7 HYPER-THREADING

Oficialmente chamada de Hyper-Threading Technology [21], é a implementação de SMT utilizada nos processadores Intel. Esta tecnologia foi desenvolvida visando obter uma melhora de desempenho através do uso do poder excedente do processador para executar várias threads simultâneamente. Esta tecnologia foi inicialmente utilizada nos processadores Xeon, e posteriormente nos processadores Pentium 4.

Esta tecnologia habilita dois processadores lógicos em um processador single-core, replicando, particionando e compartilhando recursos. Ela permite paralelismo entre threads no processador, conseguindo assim um melhor aproveitamento de seus recursos.

Hyper-Threading duplica algumas seções do processador como as que armazenam estado arquitetural, porém não duplicam os principais recursos de execução. Com isso, cada processador é referenciado como um processador lógico, podendo manter dois estados arquiteturais.

Segundo a Intel, em alguns casos esse ganho de performance chega a 30% [9], dependendo da configuração do sistema. Entretanto, dependendo da aplicação, este desempenho pode ser até pior. Em aplicativos não otimizados para multiprocessamento os pontos negativos do Hyper-Threading, ou seja, as colisões e o overhead do multiprocessamento ganham destaque, fazendo com que o sistema fique mais lento do que ficaria com o recurso desabilitado.

Outra desvantagem no uso do hyperthreading é que um sistema operacional que não seja específico para processadores intel não distingue processadores físicos de processadores lógicos, quando delega threads para serem executadas. Com isso, pode acontecer de o escalonador do Sistema Operacional entregar duas threads para um mesmo processador, quando poderia ser feita uma divisão entre dois processadores [22].

2.3 ARQUITETURA MULTICORE

Várias alternativas foram criadas para maximizar o desempenho dos processadores, sendo que, na maioria dos casos, este ganho é gerado através do paralelismo, obtido pelo uso de novas técnicas ou arquiteturas.

A arquitetura multicore (múltiplos núcleos) veio como uma das soluções para suprir as limitações encontradas nas arquiteturas single-core. Diferente do conceito de SMP, onde temos diversos processadores ligados em um barramento, na arquitetura multicore temos os núcleos compartilhados em um único chip. É possível montar uma arquitetura que utilize de SMP para ligar diversos processadores multicore [40].

Nos próximos capítulos, estudaremos a arquitetura multicore com mais detalhes, analisando como o sistema operacional Linux se adaptou a esta tecnologia e, através de testes, mostraremos como utilizá-la eficientemente,

3 ARQUITETURAS MULTICORE

Como pôde ser visto, a principal motivação para o desenvolvimento de processadores multicore deve-se às limitações físicas encontradas para o aumento de desempenho dos processadores, como previsto na Lei de Moore. O aumento de desempenho vinha sendo alcançado devido à aplicação de técnicas que exploram o paralelismo de instruções e através do aumento da freqüência do processador, o que em breve alcançaria um nível onde não seria mais possível seu desenvolvimento, devido ao problema do tamanho dos transistores. Com isso, os processadores multicore se tornaram uma alternativa, oferecendo a oportunidade de executar múltiplas aplicações de tarefas e aumentar desempenho, se realizada de maneira correta.

Uma arquitetura multicore é geralmente um multiprocessamento simétrico (Simultaneous Multiprocessing – SMP) implementado em um único circuito VLSI (Very Large Scale Integration), onde os múltiplos núcleos do processador normalmente compartilham um segundo ou terceiro nível de cache.

O objetivo é melhorar o paralelismo no nível de aplicações ou tarefas, ajudando especialmente as aplicações que não conseguem se beneficiar dos processadores superescalares atuais por não possuírem um bom paralelismo no nível de instruções.

Atualmente, dois tipos de arquiteturas multicore se destacam: homogênea (exemplo: Intel Core 2 Quad) e heterogênea (exemplo: Processador Cell BE). A arquitetura multicore homogênea é uma arquitetura com núcleos homogêneos enfileirados, [12] tendo como exemplo o Intel Core 2 Quad. Já em uma arquitetura heterogênea, os núcleos podem ser diferentes e ter, inclusive, funções específicas. Utilizaremos como exemplo de arquitetura Heterogênea o Processador Cell BE, por ser um dos casos de maior sucesso desta arquitetura [41].

3.1 ARQUITETURAS MULTICORE HOMOGÊNEAS

Arquiteturas multicore homogêneas obrigatoriamente possuem núcleos idênticos uns aos outros, tratados pelo sistema operacional como processadores

distintos, possuindo seus próprios recursos de execução, e na maioria dos casos sua própria cache L1. Além dos recursos exclusivos de cada núcleo, na maioria dos casos existe uma cache L2 ou L3 compartilhada. O gerenciamento de energia é compartilhado entre os núcleos.

O fato de possuir mais de um núcleo possibilita a execução de atividades simultaneamente e por isso, diferentemente do que acontece no pipeline, pode não haver perda de latência na execução.

Como em processadores multicore os núcleos trabalham simultaneamente, a freqüência não precisa ser tão elevada individualmente. Por exemplo, um processador com dois núcleos de 1,5 GHz pode ter um desempenho superior a um processador de único núcleo operando a uma freqüência de 3Ghz (caso a aplicação possa utilizar paralelismo). Assim, um programa que execute operações em paralelo pode tirar proveito da arquitetura multicore. A grande vantagem é que em baixas freqüências tanto consumo de energia como calor dissipado diminuem.

As arquiteturas multicore se beneficiam de técnicas como CMP para possibilitar paralelismo em nível de thread, onde cada um dos núcleos é responsável pela execução de uma thread. Pode-se também incorporar outras técnicas, como utilizar vários núcleos SMT, possiblitando que threads sejam executadas em paralelo em um mesmo núcleo. Um exemplo seria a utilização da técnica de hyper-threading.

Além de poder incorporar paralelismo em nível de thread, a arquitetura multicore permite a utilização de pipelines superescalares, para se obter paralelismo em nível de instrução [15].

3.1.1 GERENCIAMENTO DE ENERGIA

Em arquiteturas multicore, normalmente os núcleos de um mesmo chip vão possuir a mesma voltagem, pois existe apenas um regulador de voltagem presente nas placas-mãe. Tecnologias de gerenciamento de energia do processador são definidas na especificação ACPI (Advanced Configuration and Power Interface), que são interfaces independentes de plataformas para descoberta de hardware, configurações, monitoração e gerenciamento de energia. Estas tecnologias são divididas em categorias ou estados chamados de P-States (Power Performance

States), C-States (Processor Idle Sleep States), S-States (Sleep Stages) e T-States (Throttle Stages).[42].

A ACPI[23] é uma especificação industrial aberta criada pela parceria: Hewlett-Packard, Intel, Microsoft, Phoenix, e Toshiba. Esta estabelece interfaces industriais padronizadas permitindo que configuração e gerência de energia sejam controladas pelo sistema operacional.

C-States são estados utilizados para quando o CPU está desligado ou com seus recursos reduzidos, com a exceção do estado C0, onde o CPU está ativo realizando alguma atividade. Os demais estados (C1 à Cn) são atribuídos quanto o processador está ocioso, onde há um menor consumo de energia e dissipação de calor. Diferentes processadores suportam diferentes números de C-States, onde várias partes do CPU estão desligadas. Geralmente, quanto maior o C-State, mais recursos do CPU estão desativados, reduzindo significativamente o consumo de energia. Porém, pelo fato de possuir mais recursos desativados, é maior o tempo necessário para que o CPU fique ativo.

P-States são estados pré-definidos, sendo combinações de freqüência e voltagem que o processador pode operar quando a CPU está ativa. A CPU utiliza DFS (Dynamic Frequency Scaling) e DVS (Dynamic Voltage Scaling) para implementar os diversos P-States suportados por um processador. DFS e DVS são técnicas que modificam dinamicamente a freqüência e a voltagem do núcleo do processador, baseadas nas condições de operação. Ao reduzir a voltagem do núcleo, diminui-se a dissipação de energia dos transistores da CPU, ocasionando uma redução no consumo de energia.

O P-State do processador é determinado pelo sistema operacional, onde o tempo para realizar trocas de P-States é relativamente pequeno. É de responsabilidade do processador balancear consumo e performance do processador. O balanceamento de consumo/potência é ilustrado na Figura 3.1.

P-State	Frequency	Voltage	Power
P0	1.6 GHz	1.484 V	25 Watts
P1	1.4 GHz	1.420 V	~17 Watts
P2	1.2 GHz	1.276 V	~13 Watts
P3	1.0 GHz	1.164 V	~10 Watts
P4	800 MHz	1.036 V	~8 Watts
P5	600 MHz	0.956 V	6 Watts

Figura 3.1 – P-States no Intel Pentium M 1.6 GHz CPU (Potência x Consumo) [42]

As transições de P-States dos núcleos têm que ocorrer simultaneamente. O gerenciamento destes P-States pode ser controlado por hardware ou software. Com este mecanismo, quando um dos núcleos do chip transitar para um P-State diferente, o chip transita para este estado, garantindo assim um bom desempenho. Se um núcleo estiver 100% ocupado executando uma atividade, esta coordenação irá garantir que os núcleos ociosos do mesmo chip não possam entrar em P-States de baixa performance, resultando em um chip no P-State de maior performance.

Enquanto em um estado C0, a ACPI permite que a performance do processador seja alterada através de transições dos P-States. Utilizando os P-States, pode-se consumir diferentes quantidades de energia provendo performances diferentes no estado C0 (execução). Em geral, quanto maior os P e C-States, menor será o consumo de energia e o calor dissipado.

3.2 ARQUITETURAS MULTICORE HETEROGÊNEAS

Este tipo de arquitetura, os diversos elemento não são necessáriamente similares. Um exemplo de um multicore heteronêgeo é o sistema composto pelo processo Cell BE. Cell é um chip heterogêneo multiprocessado que consiste em um núcleo central denominado PPE (Power Processor Element) e oito co-processadores especializados (SPEs), utilizados para processamento intensivo de dados, como os encontrados em criptografia, mídia e em aplicações específicas. O sistema é integrado por um barramento de alto desempenho (EIB).

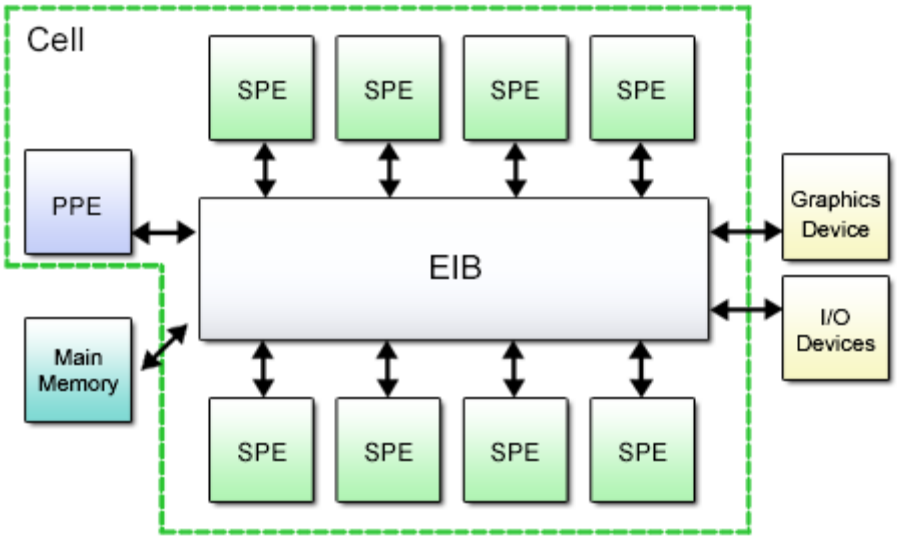

Figura 3.2 Sistema Cell Broadband Engine.

3.2.1 POWER POCESSOR ELEMENT (PPE)

O PPE é formado por um processador de propósito genérico PowerPC RISC de 64 bits capaz de executar até duas threads simultaneamente. Ele possui um cache L1 de 32KB para dados e 32KB para instruções e cache L2 de 512KB para dados e instruções rodando a 3.2 GHz. O PPE é responsável pela direta comunicação com o sistema operacional, divisão de tarefas (para si, e para os SPEs) e alocação de recursos. Ele é otimizado para o processamento intensivo de tarefas [24].

3.2.2 SYNERGISTIC PROCESSOR ELEMENTS (SPEs)

Os SPEs são processadores vetoriais de arquitetura RISC com instruções do tipo SIMD, contendo 128 registradores de 128 bits, quatro unidades de ponto flutuante capazes de realizar um total de 32 bilhões de operações por segundo, e mais quatro unidades de operações sobre inteiros, capazes também de 32 bilhões de operações por segundo, trabalhando com um *clock* de 4GHz de freqüência [25]. Os seus núcleos foram desenvolvidos para suportar programação de alto nível, como por exemplo, C e C++, com instruções específicas relacionadas à manipulação

de conteúdo, relacionados à computação gráfica, áudio, jogos e novos métodos de interação homem-máquina [24].

Ao invés de possuir uma memória cache, cada SPE possui uma SRAM privada de 256Kb embutida para instruções e dados chamada de *Local Store* (LS). O endereçamento na LS é mapeado diretamente na memória principal, e nenhum protocolo de coerência de cache é utilizado. É de responsabilidade do software gerenciar o movimento de dados entre a memória principal e a LS.

3.3 SISTEMA MULTICORE ALVO

Este projeto tem como objetivo estudar e analisar a execução de uma aplicação paralela em um sistema multicore homogêneo. Neste caso, os núcleos de processamento possuem freqüências iguais e mesma disponibilidade de memória. Ainda, o sistema operacional escolhido para o estudo foi o Linux, devido a maior disponibilidade de software e documentação. Mais especificamente, o objetivo deste estudo é saber como o escolanador do Linux tira proveito da disponibilidade de núcleos de processamento e módulos de memória no sistema alvo.

4 O ESCALONADOR DO SISTEMA OPERACIONAL LINUX

Com a evolução das arquiteturas, os processadores passaram a possibilitar multiprocessamento, fazendo com que houvesse uma necessidade dos sistemas operacionais se adaptarem e gerenciarem a execução de diferentes tarefas nos diferentes núcleos. Neste capítulo veremos alguns conceitos envolvidos e como o Sistema Operacional Linux evoluiu para se adaptar a essas modificações.

4.1 ESCALONAMENTO A NÍVEL DE SISTEMA OPERACIONAL

O Kernel de um sistema operacional fornece uma interface para que os processos utilizem os recursos do sistema e gerencia esses recursos com o objetivo de executar os processos com máxima eficiência [26]. As funções normalmente atribuídas ao kernel são:

- Criação, escalonamento e finalização de processos
- Alocação e liberação de memória
- Controle do sistema de arquivos
- Operações de entrada e saída

Em computadores multiprogramados, variados processos podem competir pela CPU ao mesmo tempo. Quando o número de processos submetidos é superior ao número de unidades de processamento, deve ser feita uma escolha de qual processo será executado em que unidade de processamento, em seguida. O escalonador é a parte do Sistema Operacional responsável por realizar esta escolha de acordo com uma determinada política que estabelece prioridades entre processos dependendo dos objetivos a serem alcançados. O algoritmo de escalonamento que implementa uma determinada política deve garantir que cada processo ganhará CPU em algum momento, evitando que o mesmo fique ocioso.

É de responsabilidade do kernel do Sistema Operacional o escalonamento dos processos na CPU, de forma que seja aproveitado o máximo de capacidade de

processamento do computador e que o usuário receba as respostas das suas requisições no menor tempo possível. Além de utilizar o máximo de poder de processamento disponível, o escalonador também pode classificar as tarefas a serem executadas, levando em consideração, por exemplo, a frequência com que os processos são chamados, se o tempo total de processamento necessário para aquele processo é conhecido e ainda o quão prejudicial seria se o processo não for executado naquele instante. Em um sistema multiprocessado, o kernel do Sistema Operacional eventualmente precisará distribuir entre todos os núcleos de processamento esses processos [27].

4.2 POLÍTICA DE ESCALONAMENTO DO LINUX

O algoritmo de escalonamento de processos utilizado pelo sistema operacional é de crucial importância e é um dos aspectos que mais influenciam na eficiência da utilização do conjunto de CPUs de um computador. O Linux é um Sistema Operacional voltado para computadores pessoais e por isso, o algoritmo implementado prioriza processos interativos.

Nas arquiteturas multicore, na maioria dos casos, aplicações que possam ser executadas utilizando paralelismo obterão benefício dos múltiplos núcleos. Neste trabalho será estudado o Kernel do Linux, desde a versão 2.4 até a versão 2.6.24.

4.2.1 KERNEL 2.4

A partir da versão 2.4 do Kernel do Linux, as arquiteturas multicore passaram a ser consideradas em seu desenvolvimento. Cada processo recebe um quantum ou “fatia de tempo” de CPU atribuído ao processo naquele momento. Quando o quantum de um processo termina, o escalonador é acionado e outro processo passa a ser executado [34].

A prioridade de cada processo é ajustada dinamicamente pelo escalonador, visando equalizar o uso do processador entre os processos, ou seja, processos que recentemente ocuparam o processador durante um período de tempo considerado “longo” têm sua prioridade reduzida. De forma análoga, aqueles que estão há muito

tempo sem executar, o que pode acontecer quando muitos processos com prioridades mais altas entram na lista de processos prontos, recebem um aumento na sua prioridade, sendo então beneficiados em novas operações de escalonamento. Esta prioridade é dada por um valor que é chamado de “merecimento” (*Goodness*) [28]. O valor do merecimento é calculado seguindo os seguintes critérios:

- Tarefas que dependiam de uma ação vinda de algum dispositivo de I/O ganham prioridade quando essa ação é efetuada.
- Tarefas que precisam de interação em tempo real ganham prioridade sobre tarefas que não precisam dessa interação.
- Tarefas especificadas pelo usuário como prioritárias, quando o usuário possui esse tipo de permissão.

Os processos inicialmente esperam em uma única fila, como pode ser observado na Figura 4.1. O próximo processo a ser alocado em alguma CPU é o que possui o maior valor de merecimento, que não necessariamente será o primeiro da fila.

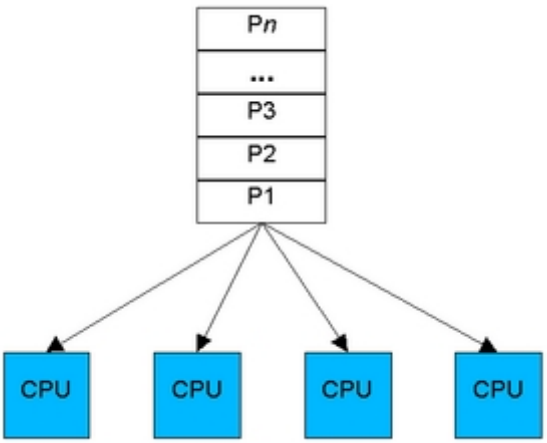

Figura 4.1 Fila de processos O(n)

O tempo entre o início da execução até sua conclusão ou até que retorne à fila de espera é chamado de fatia de tempo ou *quantum*. O cálculo de merecimento dos processos é atualizado a cada período de tempo pré especificado pelo Linux.

Neste momento, o escalonador varre todos os processos atualizando o valor de seus respectivos merecimentos, para assim, escolher o processo com o maior valor de merecimento para entrar em execução.

4.2.2 KERNEL 2.6

O algoritmo do kernel 2.4 faz com que o desempenho do Sistema Operacional passe a não ser tão eficiente à medida que aumentamos o número de processadores, pois, cada vez que um processador é liberado, a fila de processos prontos é percorrida em busca do processo com maior valor de merecimento, ou seja, temos um algoritmo de ordem de complexidade O(n) [35], sendo executado cada vez que um processador é liberado. Além disso, com mais processadores aumenta a chance de um processador eleger um processo que já esteja sendo executado por outro processador, o que obriga uma nova busca pela fila de processos [29]. Como solução de tal problema, um novo algoritmo de escalonamento foi implementado no Kernel 2.6 do Linux, onde não importa o número de processos ativos, o tempo de escalonamento é sempre constante.

Nesse kernel, os processos são divididos em filas de execução separadas por processador. O algoritmo trabalha com uma estrutura denominada vetor de prioridades [36], que manipula um conjunto de 140 filas de processos, como visto na Figura 4.2, onde cada uma dessas filas representa um nível de prioridade. O vetor de prioridades contém também variáveis que controlam o número de tarefas contidas no vetor de prioridades e quais filas contém processos ativos. Com esta organização, a tarefa do escalonador passa a ser simplesmente buscar o primeiro processo na fila de maior prioridade e enviá-lo para execução.

Cada CPU trabalha com dois vetores de prioridades, chamados de ativo e expirado. Quando um processo é selecionado da fila do vetor de prioridades ativo, ganha processamento e caso termine seu quantum antes de ser finalizado é alocado em alguma das filas do vetor de prioridades expirado. Diferentemente do algoritmo do Kernel 2.4, onde o *quantum* e a prioridade de todos os processos eram recalculados no fim de um período de tempo pré-definido, no algoritmo do Kernel 2.6, o *quantum* e a prioridade de cada processo são atualizados no momento em que o processo é alocado no vetor de prioridades expirado.

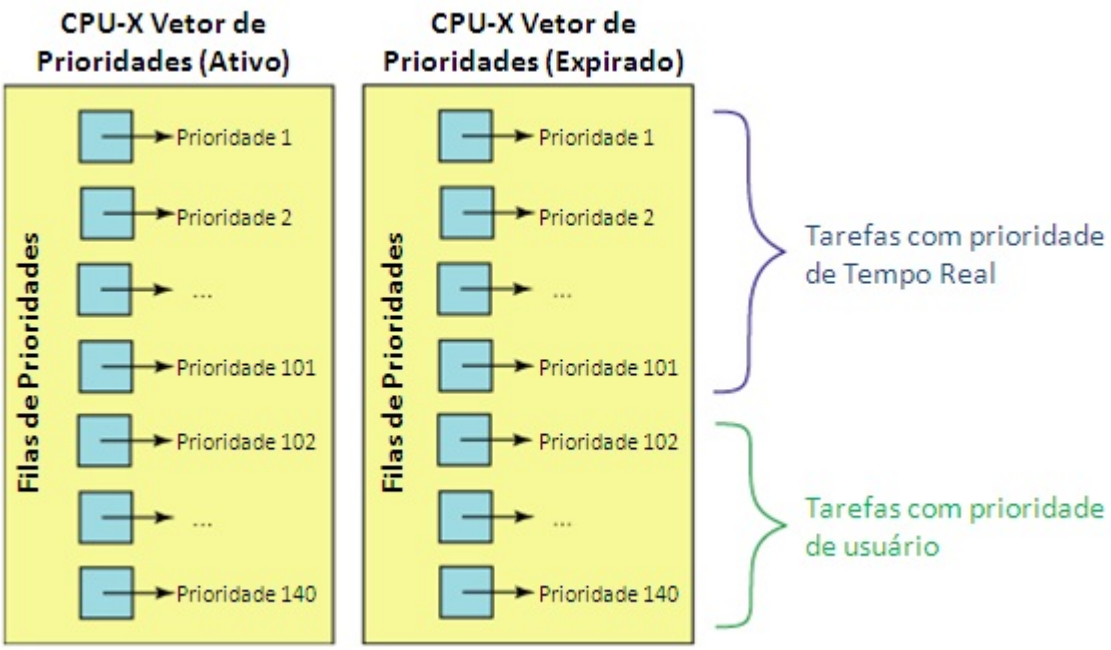

Figura 4.2 Estrutura das filas de execução do escalonador do kernel 2.6

Quando todos os processos se encontram no vetor de prioridades expirado, o escalonador simplesmente troca os ponteiros entre os vetores de prioridades. Com isso, o escalonador passa a considerar o vetor de prioridades expirado como ativo e vice-versa.

De tempos em tempos o escalonador faz um balanceamento de carga entre os processos para evitar que um processador termine todas as tarefas de sua fila enquanto outro esteja com sua fila de execução cheia. O balanceamento de carga também é invocado no momento em que um processador terminou de executar todas as suas tarefas [36].

O balanceador de carga atua a partir de cada um dos núcleos com o objetivo de trazer tarefas para sua fila de execução. Primeiramente ele busca pelo núcleo com maior carga de processos e verifica se este tem 25% a mais de carga que o núcleo atual. Se o núcleo mais carregado atender este critério, o balanceamento de carga é realizado, dando-se prioridade a tarefas alocadas no vetor de prioridades expirado, pois essas tarefas tem menos chance de estarem mapeadas na memória cache do processador de origem. A partir do vetor selecionado, as tarefas de prioridade mais alta, onde é mais importante que o balanceamento de cargas seja realizado, são migradas para o núcleo que executou o balanceamento de carga [36].

4.2.3 ESCALONAMENTO NO KERNEL 2.6.23

A principal característica na versão 2.6.23 é a implementação do algoritmo chamado Completely Fair Scheduler (CFS) para o escalonamento dos processos na CPU. Esse algoritmo substitui o algoritmo desenvolvido na versão 2.6 e traz um novo conceito de escalonamento, que procura fazer com que o processador trabalhe o mais próximo possível do que seria considerado um “processador multitarefas ideal”, ou seja, um processador que possa executar várias tarefas em um mesmo intervalo de tempo, gastando em cada uma dessas tarefas uma parte do seu poder de processamento [31]. Para isso, enquanto uma tarefa aguarda processamento (o que não aconteceria em um processamento ideal), seu tempo de espera é guardado e dividido pelo número de tarefas assinaladas para o processador, afim de que o escalonador tenha a informação de quanto de processamento uma tarefa teria recebido se estivesse trabalhando em um ambiente ideal. A tarefa a qual o processador possui a maior “dívida” é eleita para ser executada.

No algoritmo de escalonamento presente no kernel 2.6, os processos são guardados em filas de execução separadas por processador. O algoritmo CFS mantém essa característica de separar os processos entre as CPUs, porém estes processos, ao invés de serem alocados em filas de execução, são armazenados em uma estrutura chamada de árvore rubro-negra.

A árvore rubro negra é uma árvore de busca binária, que se baseia em regras para inserção e remoção de nós que assegura que os nós estejam sempre balanceados. A estrutura da árvore rubro-negra para organizar os processos que aguardam sua fatia de tempo no processador é adequada, pois mantém seus vértices (que armazenam os descritores dos processos) ordenados em uma árvore balanceada. Sendo balanceada, esta árvore tem menor altura, logo, a busca por processos é O(log n) (altura mínima) [43].

O controle dessa árvore pelo escalonador é realizado utilizando-se como chave para os nós da árvore um valor denominado "Virtual Runtime” [30]. Enquanto uma thread está sendo processada, é esperado que todos os outros processos tenham seu tempo de espera incrementados. Para que o escalonador não precise percorrer todas as tarefas para atualizar seus valores, cada uma das tarefas recebe um virtual runtime, que, no momento de sua execução, é incrementado de acordo

com o tempo ganho de processamento. A tarefa com menor valor de virtual runtime é, portanto, a próxima a ser executada. O nível de prioridade do processo pode desacelerar a quantidade de valor adicionado ao virtual runtime de uma tarefa [31].

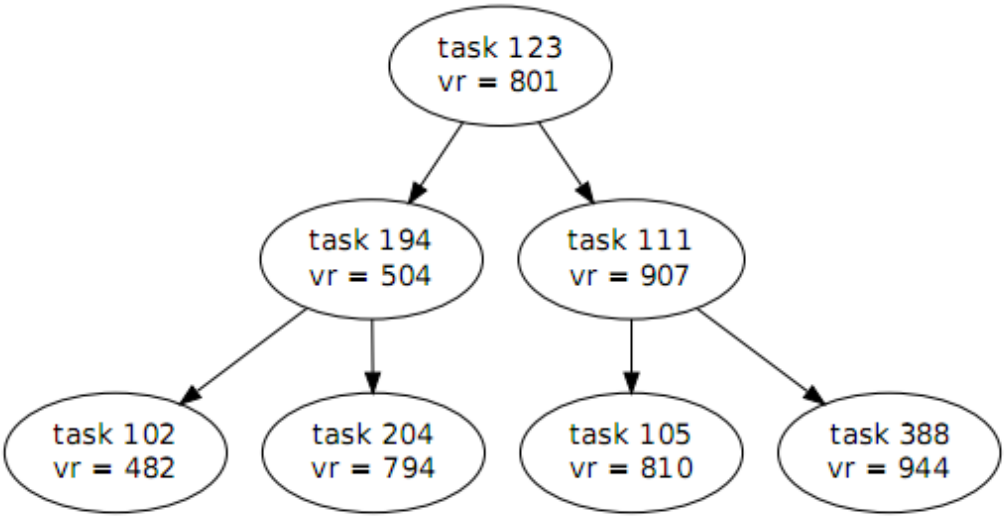

Figura 4.3 Árvore de processos (CFS) [30]

No algoritmo CFS, é guardado o valor que representaria a divisão ideal de CPU por processo até o instante atual, chamado de fair_clock. Por exemplo: se, em um intervalo de tempo x, tivessemos quatro processos na CPU, em um processador ideal, cada um destes deveria receber 25% do poder de processamento da CPU durante o tempo x, como ilustrado na Figura 4.4. Em um processador real, apenas um processo pode ganhar CPU em um determinado momento, sendo assim, este valor é incrementado com x/4, que seria o tempo que cada processo deveria ter ganho de CPU referente à sua cota justa de processamento. Quando uma nova tarefa é criada, o valor do fair_clock é atribuído ao seu virtual runtime, garantindo assim que uma nova tarefa não possua prioridade maior do que uma que estava aguardando processamento [31].

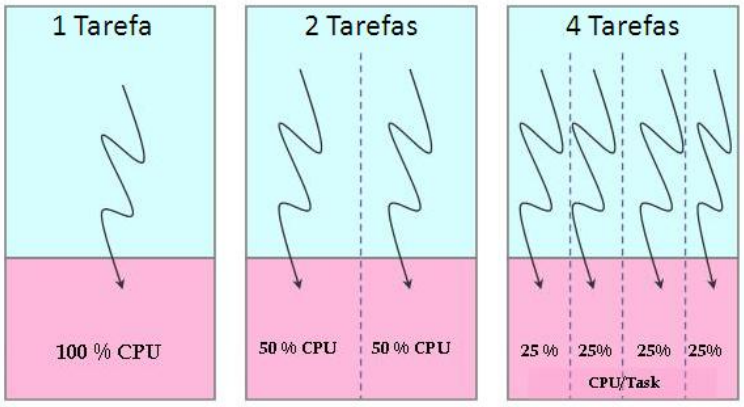

Figura 4.4 Representação de um processador ideal

O CFS não utiliza o conceito de time-slices. Um processo, ao ganhar CPU, permanece até que exista um outro processo com virtual runtime inferior [30]. Para evitar que haja um excesso de trocas de contexto, que poderia ser ocasionado se o processador recebesse vários processos em um periodo de tempo muito curto, é definido um tempo mínimo de processamento, denominado granularidade mínima [33]. O valor padrão desse parâmetro é de 16000000 ns, e pode ser ajustado de acordo com o propósito do sistema operacional (Desktops, Servidores...).

O CFS, por fazer buscas em uma árvore binária pelo próximo processo, possui complexidade O(logn), o que é uma desvantagem em relação ao algoritmo antecessor, porém esse problema é minimizado alocando-se na cache um ponteiro para o nó mais a esquerda, que é o próximo a ser escalonado na maioria dos casos. Comparado com o escalonador da versão 2.6, o CFS tende a obter melhor performance principalmente quando lida com tarefas interativas. [38] Outra vantagem do CFS em relação ao anterior é que o algoritmo da versão linux 2.6 trabalha com calculos complexos de heurísticas construídos com trechos de códigos de dificil manutenção. [29]

4.2.4 LINUX KERNEL VERSÃO 2.6.24

A versão 2.6.24 do Kernel Linux manteve o algoritmo de escalonamento CFS e adicionou a ele um novo conceito de grupos de escalonamento. Antes, o algoritmo somente interpretava o valor de tempo de espera entre processos individualmente [32]. Agora, o algoritmo separa os valores de espera para grupos de processos

levando em consideração grupos de escalonamento. Isso é possível dividindo os processos em grupos denominados entidades de escalonamento, que são vistos como entidades a serem escalonadas, sendo essas um único processo ou não. E cada uma dessas entidades possui uma organização de seus processos que fica encapsulada. Quando o escalonador busca a próxima tarefa, ele verifica se esta é um processo puro ou uma entidade de escalonamento com vários processos. Caso não seja um único processo, ele busca novamente dentro dessa estrutura e assim vai até encontrar o próximo processo. À medida que o processo é executado, os valores são propagados para os processos respeitando a hierarquia de entidades de escalonamento.

Esses grupos de escalonamento são utilizados, por exemplo, para evitar que um usuário malicioso domine toda a CPU ao inserir um número muito grande de processos. Em um caso onde temos 50 processos rodando em uma CPU, todos irão receber 2% de tempo da CPU. Todavia se, desses 50 processos, 48 são de um mesmo usuário, esse tempo deve ser dividido antes de forma que cada usuário possua 50% da CPU, caso os dois usuários estejam requisitando o máximo de processamento.

O usuário administrador pode habilitar ou desabilitar os grupos de escalonamento. O usuário administrador pode habilitar grupos de escalonamento para determinados tipos de tarefas e para diferentes usuários. Além disso, o usuário pode criar grupos de escalonamento customizados, e associar tarefas para esses grupos. Para isso, existe uma pseudopasta “cgroups”, reservada pelo sistema operacional, onde o os grupos são criados como se fossem subpastas de cgroups. Também é possível definir a parcela de processamento cada um desses grupos receberá [39].

Neste capítulo, vimos que os sistemas operacionais trouxeram em suas atualizações soluções para se adaptarem as novas tecnologias. A partir da versão 2.4, o kernel do linux passou a contar com um escalonador preparado para lidar de forma eficiente com diversos núcleos de processamento. As versões seguintes tiveram modificações na lógica de escalonamento baseadas principalmente na necessidade de lidar com um número cada vez maior de núcleos de processamento e na crescente mudança no desenvolvimento de softwares, que explora cada vez mais técnicas de programação paralela.

5 RESULTADOS EXPERIMENTAIS

A partir do estudo realizado, entendemos o que é paralelismo de atividades e como o sistema operacional trata essa situação. O estudo foi focado no Sistema Operacional Linux com Kernel 2.6.24, devido à disponibilidade de informações e melhor possibilidade de entendimento de execução.

O objetivo desse experimento foi avaliar a diferença obtida na execução de uma determinada tarefa em um ambiente multiprocessado à medida que dividimos esta em partes que possam ser executadas paralelamente. Para tal, foi desenvolvida uma aplicação que realiza a multiplicação de duas matrizes e retorna o tempo gasto nessa tarefa. São dados como parâmetros para essa execução o tamanho das matrizes e o número de threads que serão utilizadas.

5.1 PROGRAMA UTILIZADO

A aplicação foi desenvolvida em C, utilizando a API de threads POSIX thread (pthread), que possibilita a criação de threads dinamicamente. Alem disso, usando C, a responsabilidade da alocação de memória e construção de ponteiros fica com o programador ao escrever o código. Com isso, podemos ter mais controle de como queremos que os dados sejam distribuídos entre as threads. No apêndice A deste documento temos o código fonte da aplicação.

Na primeira etapa, a aplicação lê os parâmetros passados e a partir deles, cria os ponteiros e aloca espaço na memória tanto para as matrizes quanto para as threads.

Todas as threads recebem como parâmetro um registro denominado t_argumentos, que contém ponteiros para as matrizes que estão sendo multiplicadas, para a matriz resultado e ainda informações sobre qual é a área de atuação da thread na matriz resultado. A Figura 5.1 mostra a divisão da matriz resultado em uma operação envolvendo duas threads, onde cada uma delas atua em uma área específica da matriz. Em nenhum momento duas threads irão acessar o mesmo elemento da matriz resultado, ou seja, não existe uma seção crítica na

memória alocada por esta aplicação e as threads podem ser executadas de forma independente. Nesse caso, o ganho com paralelismo deve ser ainda maior.

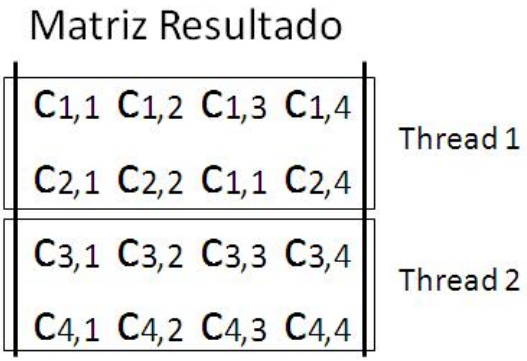

Figura 5.1. Divisão das threads na matriz resultado.

Com todas as matrizes alocadas e todas as threads carregadas com as informações das matrizes, a aplicação armazena em uma variável o tempo inicial da execução. Feito isso, o próximo passo é disparar todas as threads e ao fim da execução armazenar o tempo utilizado. Após isso, a aplicação compara o tempo atual com a variável onde foi guardado o tempo inicial da execução e retorna a diferença em um arquivo texto.

5.2 EXECUÇÃO DOS TESTES

Descreveremos aqui como foram realizados os testes de performance do ambiente multiprocessado, os resultados obtidos e uma análise dos mesmos.

5.2.1 AMBIENTE UTILIZADO

O computador utilizado nos testes possui um processador Intel core2quad Q6600 de 2.40 GHz, com 3 GB de memória RAM e Sistema Operacional Ubuntu 8.04. A versão do kernel presente no Sistema Operacional é a 2.6.24.

5.2.2 INSTÂNCIAS

Os testes foram realizados utilizando matrizes quadradas (n x n) de ordem n = 1000, 2000 e 5000 respectivamente. Para cada uma dessas ordens, foi repetido o calculo utilizando 1, 2, 4, 8, 10 e 20 threads. Esse processo foi realizado três vezes para cada número de threads com a finalidade de possibilitar o cálculo de um tempo médio de execução para cada configuração, com o objetivo de diminuir os riscos de os resultados não serem fidedignos pela utilização do computador por terceiros durante a execução.

Em todos os casos de teste, a divisão entre o número de linhas da matriz resultado e o número de threads é exata. Com isso, todas as threads terão de executar o mesmo número de operações, como por exemplo, para uma multiplicação de matrizes de ordem 2000, utilizando 8 threads, cada uma delas ficará com a incumbência de calcular 250 linhas da matriz resultado.

5.2.3 PASSO-A-PASSO

Selecionamos alguns dos casos de teste para ilustrar como o escalonador do kernel 2.6.24 opera quando um processo cria um determinado número de tarefas que serão compartilhadas entre os quatro núcleos de processamento do computador. A aplicação é basicamente composta por uma thread que é responsável por disparar as threads que realizam o trabalho da multiplicação de matrizes e como o tempo de processamento desta thread é irrisório, apenas o tempo de execução das threads que realizam as operações nas matrizes foi considerado nos testes. A estrutura criada pelo sistema pode ser vista na figura 5.2.

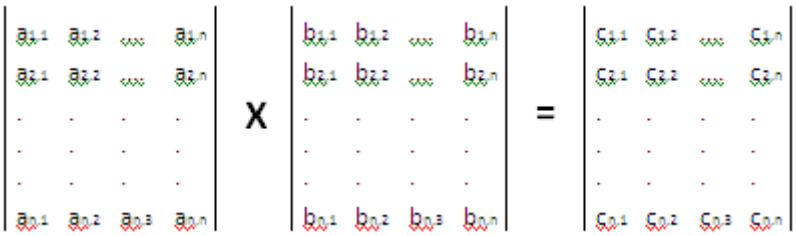

Figura 5.2. Multiplicação de Matrizes

Para uma multiplicação de matrizes utilizando apenas uma thread, esta simplesmente é alocada em uma das quatro árvores de execução de cada um dos processadores e, logo em seguida, ganha espaço no processador e executa até ser concluída, já que não há outra tarefa aguardando processamento. No caso da multiplicação de matrizes utilizando quatro threads, o comportamento acaba sendo similar, pois nesse caso, cada um dos núcleos receberá uma das threads em suas respectivas árvores de escalonamento e o processamento será realizado em paralelo.

Quando o número de threads excede o número de núcleos de processamento, já podemos esperar uma atuação relevante do algoritmo CFS no escalonamento. No caso de uma multiplicação de matrizes utilizando 20 threads, o escalonador realiza o balanceamento de carga [37], dividindo as threads entre os processadores, de forma que cada uma das árvores de cada processador receba 5 threads, onde todas terão seu valor de virtual runtime praticamente igual, já que estão sendo criadas imediatamente uma após a outra [30].

Com isso, será dado à thread eleita o tempo mínimo de execução, definido pelo valor de granularidade configurado pelo administrador do sistema no escalonador, onde o valor padrão é 16.000.000 ns [33]. Enquanto é executada, seu valor de virtual_runtime é incrementado até o momento em que ocorre sua conclusão ou seu virtual_runtime deixa de ser o menor da árvore, sendo que o tempo mínimo de execução é definido pelo valor de granularidade mínima. Caso a thread não seja concluída, ela será realocada na árvore de escalonamento de acordo com seu valor de virtual_runtime. Este processo se repete até que todas as threads sejam concluídas, ocorrendo em todos os núcleos simultaneamente.

5.2.4 RESULTADOS DA EXECUÇÃO

A partir da bateria de testes realizada, chegamos aos resultados vistos na figura 5.3.

Ordem da Matriz	Threads	Tempo 1 (s)	Tempo 2 (s)	Tempo 3 (s)	Tempo Médio (s)
1000	1	6,875297	6,992536	6,973815	6,947216
1000	2	3,546932	3,616637	3,809806	3,657792
1000	4	2,08857	1,926569	2,457343	2,157494
1000	8	1,931144	1,873671	2,115019	1,973278
1000	10	1,870688	1,88621	2,119471	1,958790
1000	20	1,871511	1,81537	2,021089	1,902657
2000	1	69,119733	69,15412	70,617936	69,630596
2000	2	36,654882	35,794317	36,602515	36,350571
2000	4	19,791573	19,938269	20,05696	19,928934
2000	8	19,520435	19,315618	18,870226	19,235426
2000	10	19,199954	19,214161	18,650372	19,021496
2000	20	19,075422	19,267662	18,557921	18,967002
5000	1	1376,737537	1380,688072	1372,916148	1376,780586
5000	2	724,266494	722,206071	710,865986	719,112850
5000	4	401,866376	442,457785	376,220543	406,848235
5000	8	397,096551	439,835341	396,539847	411,157246
5000	10	402,214352	381,897434	399,602501	394,571429
5000	20	401,417073	381,914236	413,020519	398,783943

Figura 5.3. Tabela com os resultados dos testes

Em uma primeira análise dos resultados, o que pode claramente ser percebido é o enorme ganho de desempenho quando dividimos a multiplicação de matrizes por duas e quatro threads, como pode ser visualizado na figura 5.4, onde temos a média dos resultados obtidos por threads. Isso é facilmente explicado pelo fato de que cada thread terá a sua disposição um núcleo de processamento dedicado, se considerarmos apenas a multiplicação das matrizes como processos do computador.

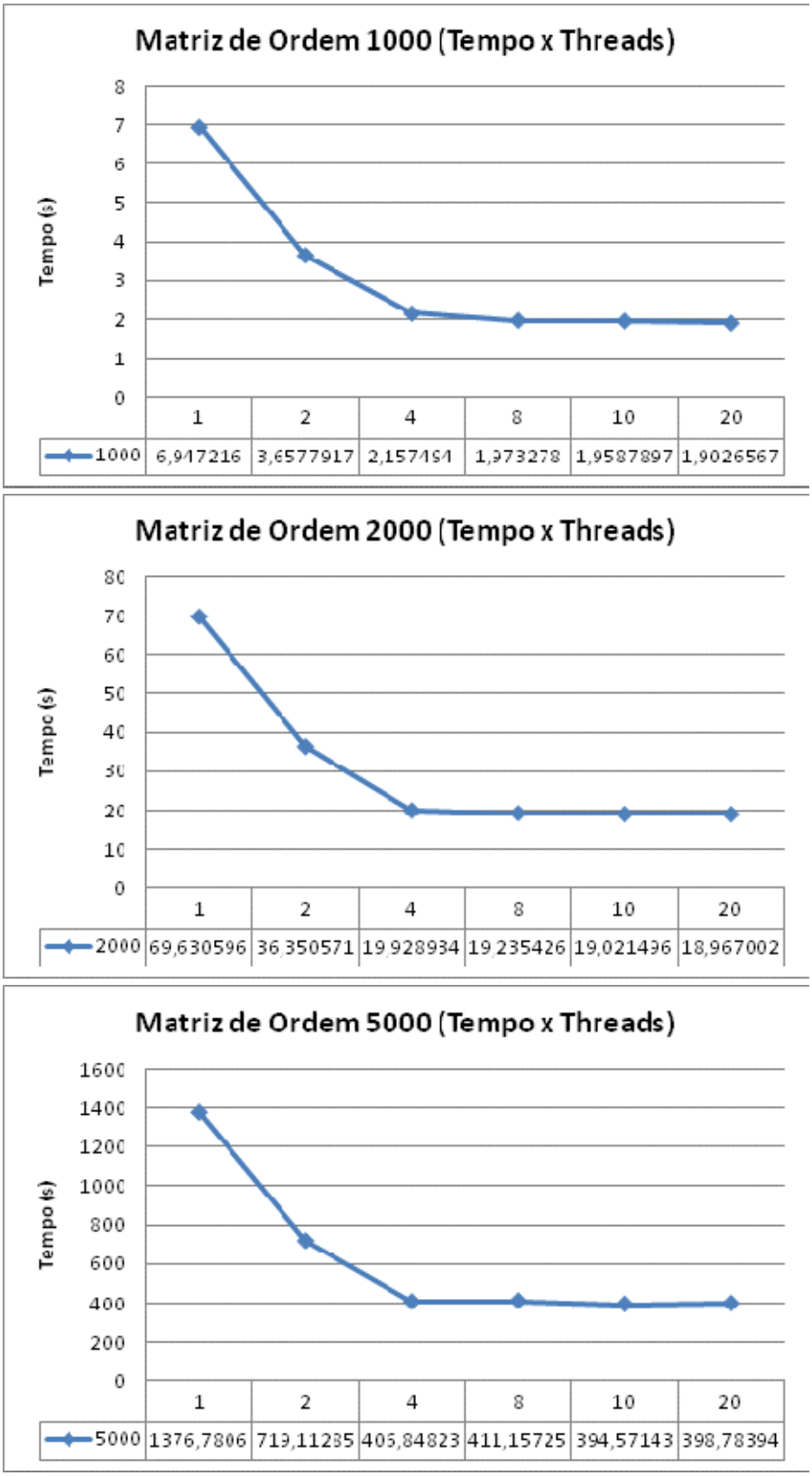

Figura 5.4. Gráfico ilustrativo dos resultados dos testes

A partir de 4 threads, temos os 4 núcleos funcionando durante a execução e podemos notar uma leve melhora no desempenho a partir da execução com 8 threads. Com o aumento do número de threads, o número de trocas de contexto aumenta, porém, como pode ser visto na figura 5.4, os resultados continuam ganhando em desempenho até um número de threads igual a 10.

Este fato ocorre, pois, o número de threads executando em um mesmo núcleo aumenta, fazendo com que processos concorrentes, como por exemplo, rotinas do sistema, ganhem menos tempo de processador.

Como pode ser visto no resultado de 20 threads com uma matriz de ordem 5000, o número de trocas de contexto faz com que aconteça uma leve queda de desempenho, se comparado ao resultado com 10 threads.

6 CONCLUSÃO

O foco deste trabalho é a questão da busca por melhor desempenho no processamento de tarefas. A partir desse estudo podemos observar técnicas que foram utilizadas em arquiteturas de computadores para se obter maior poder de processamento e, entre várias dessas técnicas, a utilização de processamento paralelo é explorada em diversos níveis. Listamos essas técnicas enquanto apresentamos detalhes das arquiteturas SMP e Multicore.

Vimos porque a arquitetura multicore é utilizada como solução para se obter maior eficiência no processamento das tarefas, devido a possiblidade de se obter desempenho de processamento sem necessariamente elevar a frequência do processador, o que resulta em economia de energia e menos calor dissipado. Também vimos questões que devem ser levadas em considerações quando se constroi um escalonador que seja responsável por gerenciar a distribuição de tarefas entre diversos núcleos, através do estudo da evolução do Kernel do Linux a partir da versão 2.4 até a versão 2.6.24.

Com o conhecimento adquirido sobre os escalonadores, montamos um experimento usando a versão 2.6.24 do Kernel do Linux e uma aplicação multithreading de multiplicação de matrizes, para que pudessemos na prática ver o desempenho do algoritmo de escalonamento CFS, principalmente com número de processos maior que a quantidade de núcleos. Analisando os resultados, concluimos que o algoritmo CFS consegue utilizar com eficiência todos os núcleos já que, em um processador com quatro núcleos, quando dividíamos os calculos em quatro threads, o tempo de processamento se reduzia a quase um quarto do tempo de processamento do mesmo número de calculos com apenas uma thread. Vimos ainda que continuavamos a obter ganho de desempenho quando dividiamos o calculo em um número de threads maior que o número de processadores, pois com um número maior de threads, as chances de o escalonador eleger uma thread da aplicação ao invés de alguma rotina do sistema operacional, por exemplo, aumenta. Apenas no teste de multiplicação de matrizes com ordem 5000 utilizando 20 threads

tivemos uma queda de desempenho, ocasionada pelo aumento do número de trocas de contexto.

Tivemos a oportunidade ainda de demonstrar como é importante que, durante o desenvolvimento de aplicações, sejam identificados pontos onde técnicas de programação paralela possam ser aplicadas e os beneficios trazidos por essa prática. Como foi dito, em uma mutiplicação de matrizes, que foi o problema abordado nesse expermento, conseguimos obter até quase quatro vezes melhor desempenho no processamento dos calculos em um computador com quatro núcleos, ao dividir a responsabildade de calcular as linhas da matriz resultado entre eles.

APÊNDICE A

Código fonte da aplicação de multiplicação de matrizes.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#include <sys/time.h>

typedef struct {

      float ** A;
      float ** B;
      float ** C;
      int inicio;
      int fim;
      int dim;
} t_argumentos;
void * multiplica(void * args) {

      t_argumentos * argumentos = (t_argumentos *) args;
      int dim;
      float ** A, ** B, ** C;
      int i, j, k;
      int inicio, fim;

      inicio = argumentos->inicio;
      fim = argumentos->fim;
      dim = argumentos->dim;
      A = argumentos->A;
      B = argumentos->B;
      C = argumentos->C;

      for(i=inicio; i<fim; i++){
            for(j=0; j<dim; j++){
                  C[i][j] = 0;
                  for(k=0; k<dim; k++){
                        C[i][j] += A[i][k] * B[k][j];
                  }
```

```
            }
      }

      return(NULL);
}

int main(int argc, char ** argv) {

      FILE *arquivo;

      if((arquivo = fopen("resultb.txt","w+")) == NULL)
      {
            printf("\nErro ao abrir arquivo");
            return 1; /* informar que o programa terminou com erro */
      }

      int iteracoes = 0;
      int numero_de_matrizes = atoi(argv[1]);

      for(iteracoes = 0; iteracoes < numero_de_matrizes; iteracoes++) {
            int dim;
            float ** A, ** B, ** C;
            int i;
            int n_threads;
            int linhas_por_thread;
            t_argumentos * args;
            pthread_t * threads;
            struct timeval begin, end;


            dim = atoi(argv[iteracoes * 2 + 2]);
            n_threads = atoi(argv[iteracoes * 2 + 3]);
            linhas_por_thread = dim / n_threads;

            A = (float **) malloc(dim * sizeof(float *));
            B = (float **) malloc(dim * sizeof(float *));
            C = (float **) malloc(dim * sizeof(float *));
            args = (t_argumentos *) malloc(n_threads *
sizeof(t_argumentos));
            threads = (pthread_t *) malloc(n_threads * sizeof(pthread_t));
```

```
            for(i=0; i<dim; i++){
                  A[i] = (float *) malloc(dim * sizeof(float));
                  B[i] = (float *) malloc(dim * sizeof(float));
                  C[i] = (float *) malloc(dim * sizeof(float));
            }


            gettimeofday(& begin, NULL);


            for(i = 0; i < n_threads - 1; i++) {

                  args[i].A = A;
                  args[i].B = B;
                  args[i].C = C;
                  args[i].inicio = i * linhas_por_thread;
                  args[i].fim = (i + 1) * linhas_por_thread;
                  args[i].dim = dim;

                  pthread_create(& threads[i], NULL, multiplica, &
args[i]);
            }


            args[i].A = A;
            args[i].B = B;
            args[i].C = C;
            args[i].inicio = i * linhas_por_thread;
            args[i].fim = dim;
            args[i].dim = dim;

            pthread_create(& threads[i], NULL, multiplica, & args[i]);


            for(i = 0; i < n_threads; i++) {

                  pthread_join(threads[i], NULL);
            }

            gettimeofday(& end, NULL);
```

```
            double tempo_gasto;
            tempo_gasto = ( (double) (end.tv_usec - begin.tv_usec) ) /
1.0e6;
            tempo_gasto += ( (double) (end.tv_sec - begin.tv_sec) );
            char texto[100];

            sprintf(texto,"%f",tempo_gasto);

            fprintf(arquivo,"%s%d%s%d%s%s","\ntamanho = ",dim," numero de
threads = ",n_threads," tempo = ", texto);

      }

      fclose(arquivo);

      return(0);
}


      .
```

REFERÊNCIAS BIBLIOGRÀFICAS

[1] Cramming more components onto integrated circuits. Electronics Magazine (1965). ftp://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf

[2] Marcelo D'Emidio, Prof. Doutor. Avaliação da Lei de Moore e proposta de um modelo de previsão alternativo baseado em técnicas de extrapolação de tendências. Future Studies Research Journal, São Paulo, v. 1, n. 2, pp. 03-22, Jul./Dez. 2009.

[3] Super-Sequenciamentos de DNA e a Lei de Moore. http://scienceblogs.com.br/rnam/2011/04/super-sequenciamentos_de_dna_e.php, Último Acesso 12/05/2011.

[4] Bruce Dawson, Software Design Engineer. Coding for Multiple Cores on Xbox and Microsoft Windows. http://msdn.microsoft.com/en-us/library/ee416321.aspx, Último Acesso 30/11/11.

[5] Cardoso, Bruno. Rosa, Sávio. Fernandes, Tiago. Multicore. http://www.ic.unicamp.br/~rodolfo/Cursos/mc722/2s2005/Trabalho/g07-multicore.pdf, Último Acesso 11/07/2011.

[6] Introducing the world's first 3-D transistor ready for high-volume manufacturing. http://www.intel.com/technology/architecture-silicon/22nm/index.htm, Último Acesso 12/05/2011.

[7] Top 500 Supercomputer Sites http://www.top500.org/, Último Acesso 09/01/2012

[8] Nicholas Carter – Arquitetura de Computadores – Coleção Schaum, Bookman, 2003.

[9] Performance Insights to Intel® Hyper-Threading Technology. http://software.intel.com/en-us/articles/performance-insights-to-intel-hyper-threading-technology/, Último acesso 04/07/2011

[10] J. Hennessy and D. Patterson. Computer Architecture - A Quantitative Approach. Morgan Kaufmann, 2003.

[11] Marcelo Fontes Santana – Arquiteturas Superescalares http://www.ic.unicamp.br/~ducatte/mo401/1s2010/T2/100602-t2.pdf

[12] Jussara M. Kofuji, Sergio T. Kofuji, Roberto K. Hiramatsu - Programando Multicore com IBM-full System Simulator – Cell Broadband Engine http://www.sbc.org.br/sbac/2007/cdrom/papers/wscad/minicursos/33149_1.pdf, Último acesso 09/01/2012.

[13] André Luís Fávero - Suporte a Multiprocessadores Simétricos (SMP) em kernel Linux. http://www.inf.ufrgs.br/gppd/disc/cmp134/trabs/T1/041/afavero/smp.pdf

[14] IBM – Documentação do AIX - Symmetrical Multiprocessor concepts and architecture. http://publib.boulder.ibm.com/infocenter/pseries/v5r3/index.jsp?topic=/com.ibm.aix.prftungd/doc/prftungd/smp_concepts_arch.htm

[15] Douglas Camargo Foster - Arquiteturas Multicore. http://www-usr.inf.ufsm.br/~andrea/elc888/artigos/artigo2.pdf

[16] J. Lo, S. Eggers, J. Emer, H. Levy, R. Stamm and D. Tullsen, Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading, In ACM Transactions on Computer Systems, vol. 15, pp. 322-354, 1997. http://www.ecst.csuchico.edu/~juliano/csci620/Papers/jLo1997tlp2ilp.pdf

[17] Henrique C. Freitas, Marco A. Z. Alves, Nicolas B. Maillard, Philippe O. A. Navaux - Ensino de Arquiteturas de Processadores Multi-Core Através de um Sistema de Simulação Completo e da Experiência de um Projeto de Pesquisa.

[18] L. Spracklen, S. G. Abraham, “Chip Multithreading: Opportunities and Challenges”, IEEE International Symposium on High-Performance Computer Architecture, pp. 248-252, February 2005.

[19] D. Nussbaum, A. Fedorova, and C. Small. An Overview of the Sam CMT Simulator Kit. Technical Report. Sun Microsystems, Inc., Mountain View, CA, USA, 2004.

[20] GONÇALVES, R. A. L. et al. A Simulator for SMT Architecture: Evaluating Instruction Cache Topologies, SBAC-PAD, São Pedro, 2000

[21] Deborah T. Marr Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, Michael Upton. Hyper-Threading Technology Architecture and Microarchitecture.

[22] White Paper – Hyper-Threading Multiprocessor System Performance on Server 2003 http://www.tmurgent.com/WhitePapers/WP_HyperThread.pdf

[23] Advanced Configuration & Power Interface Documentation
http://www.acpi.info

[24] José Ricardo de Oliveira Damico - Introdução ao Processador CELL BE
http://dcon.com.br/jd.comment/intro-cell-be-pt_BR.pdf

[25] Douglas José S. Rodrigues - A Arquitetura Cell
http://www.ic.unicamp.br/~ducatte/mo401/1s2006/T2/011104-T.pdf

[26] Tanenbaum, A. S. Sistemas Operacionais Modernos. Traduzido por R. A. L. Gonçalves; L. A. Consularo. 2ª Edição. São Paulo: Pearson Prentice Hall, 2003b.

[27] Paulo Baltarejo Sousa - Implementing a Multiprocessor Linux Scheduler for Real-Time Sporadic Tasks http://dsie.fe.up.pt/site/docs/paulosousa/pbsousa_dsie09_camera_ready.pdf

[28] Rick Lindsley - Kernel Korner - What's New in the 2.6 Scheduler - Linux Journal - 2004 http://www.linuxjournal.com/article/7178

[29] M. Tim Jones - Inside the Linux scheduler - IBM developerWorks - 2006 http://www.ibm.com/developerworks/linux/library/l-scheduler/

[30] Taylor Groves, Jeff Knockel, Eric Schulte - BFS vs. CFS Scheduler Comparison – 2009 http://www.cs.unm.edu/~eschulte/data/bfs-v-cfs_groves-knockel-schulte.pdf

[31] Avinesh Kumar - Multiprocessing with the Completely Fair Scheduler - IBM developerWorks - 2008 http://www.ibm.com/developerworks/linux/library/l-cfs/

[32] Corbet - CFS group scheduling - LWN.net – 2007 http://lwn.net/Articles/240474/

[33] IBM – Linux Information – Adjusting CFS Parameters http://publib.boulder.ibm.com/infocenter/lnxinfo/v3r0m0/index.jsp?topic=%2Fliaai%2Fsaptuning%2Fsaptuningadjust.htm

[34] William Stallings, Operating Systems: Internals and Design Principles, Prentice-Hall, Inc., 2011

[35] Josh Aas - Understanding the Linux 2.6.8.1 CPU Scheduler - 2005 http://joshaas.net/linux/linux_cpu_scheduler.pdf

[36] Robert Love - The Linux Process Scheduler - 2003 https://www.cs.drexel.edu/~wmm24/cs370/resources/Scheduler.pdf

[37] Márcio Augusto de Souza, Omar Andrés Carmona Cortez, Luciano José Senger, Regina Helena Carlucci Santana - Sistema de Monitoração para o Escalonamento de Processos: Estrutura e Métricas de desempenho http://www.unieuro.edu.br/downloads_2005/ruti_01_01_SistemaMonitoracaoEscalonamento.pdf

[38] Henrik Austad - A Survey of Real-Time Scheduling algorithms for the Linux kernel
http://folk.ntnu.no/henrikau/sched/rt_sched_pro.pdf

[39] CFS Scheduler Documentation -
http://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt

[40] Stefanus Du Toit - The difference between multi-core and multi-processing - 2008 http://software.intel.com/en-us/blogs/2008/04/17/the-difference-between-multi-core-and-multi-processing/

[41] IBM – The Cell Project at IBM Research
https://www.research.ibm.com/cell/home.html

[42] Intel and Core i7 (Nehalem) Dynamic Power Management
http://cs466.andersonje.com/public/pm.pdf

[43] J. Szwarcfiter e L. Markeson, Estrutura de Dados e Algoritmos , Editora LTC