Universidade De São Paulo
Instituto de Matemática e Estatística
Departamento de Ciência da Computação

MAC0499 Trabalho de formatura Supervisionado

Simulação de Threads em Ambiente Multi-core

Marcelo Kenji Matsuzaka
kenji.matsuzaka@gmail.com
Márcio Guedes Hasegawa
mghasegawa@gmail.com

São Paulo
27 de fevereiro de 2008

Universidade De São Paulo
Instituto de Matemática e Estatística
Departamento de Ciência da Computação

Marcelo Kenji Matsuzaka
kenji.matsuzaka@gmail.com
Márcio Guedes Hasegawa
mghasegawa@gmail.com

Simulação de Threads em Ambiente Multi-core

MAC0499 Trabalho de formatura Supervisionado do Departamento de Ciência da Computação da Universidade De São Paulo para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: *Marco Dimas Gubitoso*

São Paulo
27 de fevereiro de 2008

Resumo

O objetivo deste projeto é realizar a análise de threads em um ambiente que possua mais de um núcleo. Foram feitos estudos sobre esse tipo de ambiente e sobre threads.

O estudo deste novo ambiente foi muito importante, pois ajudou a compreender como a biblioteca deveria funcionar e o que ela deveria fazer. Foram estudados a origem, a arquitetura e o funcionamento destes tipos de processadores, em especial do processador CELL [1].

Ao estudar threads e suas implementações foi possível, entre outras coisas, escolher qual a linguagem de programação que seria utilizada. Um ponto importante deste estudo é que a implementação de threads deveria ter funções, que além de manipulá-las, pudessem obter algumas informações do kernel.

Com a ajuda desta biblioteca o usuário terá a possibilidade de avaliar se houve um ganho significativo ao executar um programa em um ambiente com mais de um núcleo. Vale lembrar que os resultados são obtidos através de uma simulação, portanto, estes são uma aproximação do desempenho real.

Palavras-chave: Multi-core, Cell, Thread, Pthread

[1] O projeto iniciou-se com o objetivo de entender e executar programas neste processador. Porém, o prazo para a obtenção de uma máquina com esta configuração expirou, fazendo com que o estudo se ampliasse para processadores *multi-core* em geral

Sumário

1 Introdução **1**

2 Multi-core **4**

2.1 Definições e detalhes de algumas arquiteturas 6

2.1.1 Processador duplo 6

2.1.2 Hyper-Threading 6

2.1.3 Dual core e Multi core 8

3 Cell **10**

3.1 Arquitetura 12

3.1.1 Power Processor Element 12

3.1.2 Synergistic Processor Elements 13

3.1.3 Element Interconnect Bus 14

3.2 Aplicações 15

3.2.1 PlayStation 3 15

3.2.2 RoadRunner 16

3.2.3 Decodificação de sinais digitais 16

4 Thread **17**

4.1 Definição 18

4.2 Threads vs Processos 18

4.3 POSIX Threads 19

5 Biblioteca **20**

5.1 Necessidade 21

5.2 Desenvolvimento 21

5.2.1 Marcação de tempos 21

5.2.2 Indentificação de threads e mutexes 22

5.3 Utilização 22

6 Interpretador **23**
6.1 Leitor 24
6.1.1 Linguagem 24
6.1.2 Blocos de código 24
6.2 Algoritmo 25
6.2.1 Política de escalonamento 25
6.3 Saída Gráfica - *Applet* 25
6.3.1 Desenvolvimento 25

7 Exemplo **27**
7.1 Execução 28
7.2 Interpretação dos dados 30
7.3 Utilização da ferramenta 31
7.4 Análise dos resultados 33

8 Parte Subjetiva **34**

CAPÍTULO 1

Introdução

Em 1965, Gordon Earle Moore, co-fundador da *Intel Corporation*, fez uma afirmação em um *paper* científico [1] que descrevia a tendência da evolução dos *hardwares* na área da computação. Segundo Moore, a quantidade de transistores em um circuito integrado dobraria aproximadamente a cada ano. Esta afirmação ficou conhecida como a Lei de Moore.

Ela foi baseada na observação do avanço da fotolitografia, que permitia a fabricação de transistores cada vez menores, fazendo com que sua quantidade dobrasse a cada ano dentro de um circuito integrado.

Esta lei não é aplicada apenas na quantidade de transistores, mas também pode se referir à capacidade de outros dispositivos eletrônicos, como por exemplo: o poder de processamento, capacidade da memória e até na resolução de câmeras digitais.

A seguir, um gráfico que representa a Lei de Moore para a quantidade de transistores em um circuito integrado:

Em 1975, Moore alterou a sua projeção. Agora, a quantidade de transistores em um circuito integrado dobraria a cada 2 anos.

Em Abril de 2005, Moore afirmou em uma entrevista que a sua lei não será válida para sempre, pois chegará um momento em que o limite físico será alcançado. O tamanho dos transistores alncançará o nível atômico e então, não poderão ser mais reduzidos. Ainda nessa entrevista, Moore disse que tínhamos ainda 10 a 20 anos antes que esse limite fosse alcançado.

Alguns fatores que Moore se baseou para chegar às suas afirmações:

Veremos a seguir que, apesar do limite físico ainda não ter sido alcançado, a evolução dos transistores encontrou uma nova barreira. Porém, o poder de processamento continua em ascensão.

[1] Esta afirmação de Moore pode ser encontrada na publicação *Cramming more components onto integrated circuits*, na *Electronics Magazine* do dia 19 de Abril de 1965.

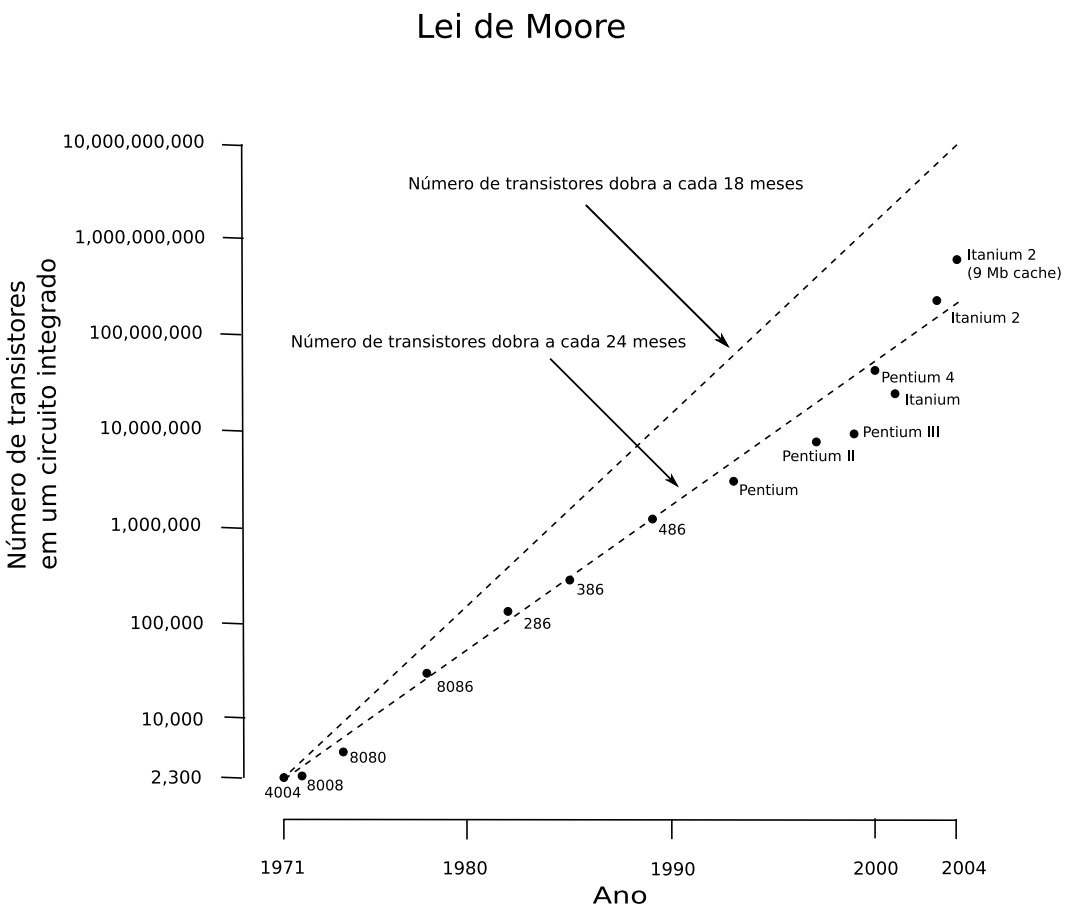


Figura 1.1 Gráfico demonstrando a Lei de Moore

Microprocessador	Ano de fabricação	Transistores
4004	1971	2,300
8008	1972	2,500
8080	1974	4,500
8086	1978	29,000
Intel286	1982	134,000
processador Intel386™	1985	275,000
processador Intel486™	1989	1,200,000
processador Intel® Pentium®	1993	3,100,000
processador Intel® Pentium® II	1997	7,500,000
processador Intel® Pentium® III	1999	9,500,000
processador Intel® Pentium® 4	2000	42,000,000
processador Itanium®	2001	25,000,000
processador Itanium® 2	2003	220,000,000
processador Itanium® 2 (9Mb cache)	2004	592,000,000

Figura 1.2 Tabela referente ao gráfico

CAPÍTULO 2

Multi-core

Como visto, a Lei de Moore dizia que a quantidade de transistores em um circuito integrado aumentaria exponencialmente a cada ano. Este número está diretamente ligado à eficiência, ao poder de processamento e logicamente ao custo do chip.

Em geral, com a diminuição do tamanho dos transistores, a mudança de um estado para outro torna-se cada vez mais rápida. Assim, é possivel que o *clock* dos processadores seja aumentado em grandes proporções, fazendo com que os aplicativos existentes sejam rodados em um tempo cada vez menor.

Nos últimos anos surgiram alguns obstáculos que impediram a continuação do desenvolvimento dos processadores até então convencionais, os *single-core*. O consumo de energia e o custo para dissipar o calor gerado, tornaram a evolução destes processadores praticamente impraticável.

Portanto, antes do limite físico, o limite econômico dificultou a evolução dos processadores. Felizmente, isto não foi suficiente para impedir o seu desenvolvimento, já que os arquitetos de processadores encontraram outras maneiras de incrementar a sua eficiência.

A seguir algumas das principais estratégias adotadas para se obter um maior desempenho dos processadores:

- Maior formato dos dados

  Esta inovação tornou o processamento de dados maiores mais rápido, possibilitando o processador de endereçar um maior volume de memória. Esses avanços foram cruciais para aumentar a eficiência e capacidade do escalonamento do processador.

- Paralelismo do nível de instrução, *Instruction level parallelism - ILP*

  Um processador que utiliza o *ILP* é capaz de avaliar dinamicamente o código de um programa, conseguindo determinar qual instrução poderá ser processada, com segurança, independentemente das outras ou fora da ordem.

  Para não perder a produtividade, enquanto uma determinada instrução está aguardando um dado, o processador executa uma outra instrução independete nesse tempo. Esta es-

trategia tem sido cada vez mais importante, já que a velocidade do processador ultrapasou a da memória.

Os desenvolvedores de processadores têm investido muito para otimizar esse tipo de estratégia para reduzir o tempo de espera devido à memória. Entretanto é uma tarefa muito complexa e, atualmente, tem consumo considerável de energia, além de gerar muito calor.

- Cache maior
  A memória *cache* é utilizada para armazenar dados próximo ao processador, frequentemente encontrada dentro do mesmo chip. Este tipo de memória é muito mais rápida que as demais, reduzindo consideravelmente o tempo de espera de um processador.

  O cache consome menos encergia do que os outros circuitos lógicos, portanto, aumentar a quantidade de cache é uma maneira de incrementar a eficiência e a produtividade de um processador sem causar um aumento proporcional do gasto de energia. Uma das principais desvantagens do cache é o seu custo elevado, o que torna comercialmente inviável um processador com uma grande quantidade dele.

- *Hyper Threading*
  Essa estratégia foi a que teve um maior destaque e divulgação, principalmente porque apareceram processadores incluindo o seu nome neles. Essa estrategia baseia-se em aproveitar o poder de processamento excedente, de um processador, para executar uma outra *thread*, como veremos melhor a seguir.

2.1 Definições e detalhes de algumas arquiteturas

2.1.1 Processador duplo

Um tradicional processador duplo possui, fisicamente, dois processadores no mesmo *chassis*. Estes dois processadores são geralmente encontrados na placa mãe, mas ocasionalmente também podem ser localizados em diferentes placas. Um sistema operacional que utiliza esse tipo de processador é capaz de agendar a execução de dois processos separados, ou de duas *threads* de um processo, simultaneamente.

2.1.2 Hyper-Threading

Oficialmente chamado de Hyper-Threading Technology, marca registrada da *Intel Corporation* para a implementação da tecnologia que suporta mais de uma *thread* na microarquitetura *pentium 4*. Inicialmente utilizada nos processadores *Xeon*, para servidores, e posteriormente nos processadores *pentium 4*.

Como já dito anteriormente, essa tecnologia aproveita a capacidade de processamento ao máximo de um processador, pois utiliza a parte ociosa do processador para executar uma nova *thread*. A tecnologia *Hyper-Threading* duplica certas seções do processador, aquelas que armazenam o estado arquitetural, porém não duplicam os principais recursos da execução. Isto faz com que os sistemas operacionais tratem processadores que utilizam essa tecnologia como se fossem dois processadores. Sendo cada processador referenciado como um procesador lógico, que pode manter dois estados arquiteturais diferentes, permitindo separar os Controladores Interruptos de Programação Avançada, *Advanced Programmable Interrupt Controlers - APIC* [1].

Os recursos compartilhados incluem ítens como *cache*, registradores e unidades de execução, que permitem executar dois programas separados ou duas *threads* simultaneamente. Alguns requisitos necessários para se ter um sistema com a tecnologia *HT* são: possuir um processador com essa tecnologia, um sistema operacional que a suporte e por fim uma *BIOS* que permita habilitar e desabilitar a *Hyper-Threading Technology*.

Existe a possibilidade de se ter dois processadores *HT* na mesma máquina, permitindo a execução de quatro programas ou *threads* simultaneamente.

As vantagens de se ter um processador com a tecnologia *HT* são: maior suporte a códigos

[1] O *APIC* possibilita o multi-processador de ter um gerenciamento interrupto e incorpora tanto a distribuição interrupta simétrica dinâmica como a estática.

multi-thread, possibilidade de ser executar *threads* simultaneamente e melhora no tempo de reação e resposta.

As primeiras implementações utilizavam cerca de 5% a mais da zona morta, *die* [1] *area*, em relação aos processadores sem o *hyper-threading*. Sendo que era estimado que essa porcentagem atingisse entre 15 a 30%. Segundo a *Intel*, os processadores *HT* possuem uma velocidade até 30% maior, quando comparados a processadores *single core* de mesmo *clock*. Porém esse aumento na perfomance varia muito com a aplicação que é utilizada, para alguns programas a perfomance chega a ser inferior.

A seguir algumas aplicações dos processadores com a tecnologia *HT* no cotidiano:

- *HT* em computadores da área comercial
  A tecnologia permite a usuários de *desktop* obter uma maior perfomance em ambientes de múltiplas tarefas. Assim, eles podem executar uma maior demanda de aplicação, mantendo o sistema estável. Departamentos da área da Tecnologia da Informação, podem utilizar um maior volume de serviços de suporte, tornando os seus ambientes mais seguros, eficientes e com uma maior facilidade de gerenciamento.

- *HT* na área de jogos e vídeos
  A tecnologia *Hyper-Threading* combinada com um processador *dual core*, que veremos a seguir, permite que os usuários possam jogar os mais atuais jogos com um grande nível de detalhamento. Já os usuários de vídeo, podem realizar trabalhos cada vez mais complexos e com um maior desempenho, realizando tarefas simultãneas como a criação de um vídeo enquanto o anti-vírus faz a sua varredura.

- *HT* em servidores
  Com a tecnologia *HT*, é possivel executar *threads* em paralelo, tendo cada processador em um servidor. Principalmente utilizado pelas empresas de aplicações *web* e de servidores.

[1] Die é uma palavra utilizada na área de computação, que significa: área quadrada de silicone e que possui o circuito integrado no seu interior. Também é utilizada como sinônimo para *chip*.

2.1.3 Dual core e Multi core

Quando um processador é referenciado como *dual core*, significa que o *chip* do circuito integrado possui dois núcleos, no seu interior. Normalmente, isto significa que dois processadores idênticos foram construídos lado a lado no mesmo circuito integrado, pode-se encontrar processadores que possuam os núcleos empilhados, ou seja, um em cima do outro.

Os núcleos são auto suficientes, portanto cada um dos núcleos possui seu próprio recurso, como estado arquitetural, registradores e unidades de execução. Em relação à memória *cache*, os núcleos a compartilham, porém em alguns processadores, existem camadas que somente podem ser acessadas por um deles, reservas feitas exclusivamente para um dos núcleos.

Núcleos em um procesasdor *dual core*, comunicam-se diretamente dentro do circuito integrado, assim, não é necessário um barramento de comunicação entre eles. Esta é uma das principais diferenças, que tornam os processadores de dois núcleos ou mais, superiores aos computadores com dois processadores separados, *dual processor*.

Os processadores *dual core* podem possuir a tecnologia *Hyper-Threading*, permitindo que apenas um circuito integrado possua dois núcleos, possuindo a capacidade de quatro processadores lógicos. Assim, este processador pode executar quatro programas ou *threads* simultaneamente.

Processadores *multi core* são uma extensão dos processadores *dual core*, dentro dele podem haver dois ou mais núcleos [1]. Assim, a tendência é aumentar cada vez mais a quantidade de núcleos dentro de um circuito integrado. Supondo que o número de transistores permaneçam fixos, pode-se afirmar que a quantidade de núcleos em um processador seguirá a Lei de Moore. Caso esta tendência não siga a Lei de Moore, certamente a quantidade de núcleos em um processador aumentará significativamente nas futuras gerações de processadores.

Ninguém sabe ao certo qual é a quantidade de núcleos que otimizam o desempenho de um processador, pois este número pode variar, conforme apareçam programas otimizados para um maior número de núcleos. Entretanto, um programa feito para ser executado por um processador *single core* ou um que possua alguns núcleos, certamente não aproveitará ao máximo a capacidade de um processador com muitos núcleos. Portanto, há a necessidade de que não apenas os processadores evoluam, mas que os programas acompanhem esta evolução. Por exemplo, os futuros sistemas operacionais deverão ter a capacidade de reconhecer múltiplos processadores e possuir um mecanismo que delegue processos separados, ou *threads*, para os

[1] Cada núcleo possui uma implementação que otimiza: execuções em grande escala, *pipelinig* [2] e múltiplas *threads*.

processadores físicos e lógicos existentes.

Por possuírem vários núcleos trabalhando simultaneamente, os desenvolvedores dos processadores *multi core* diminuiram a frequência de cada núcleo. Assim, o consumo de energia e a quantidade de calor gerado diminuem. Para programas feitos para a tecnologia *multi core* o desempenho não é afetado, porém para programas que utilizam apenas um núcleo há um pequeno decréscimo de desempenho, pois a frequência dos núcleos foi alterada. Mas no geral, temos um acréscimo na eficiência destes tipos de processadores.

Basicamente, um processador *multi core* é similar à um servidor com múltiplos processadores, entretanto os recursos para a computação em paralelo encontram-se no mesmo *chip*. Uma outra característica importante é comunicação entre núcleos, que é mais rapida. Estima-se que o desempenho deste tipo de processador, em relação a um *single core*, seja pelo menos 40% a 80% superior.

CAPÍTULO 3

Cell

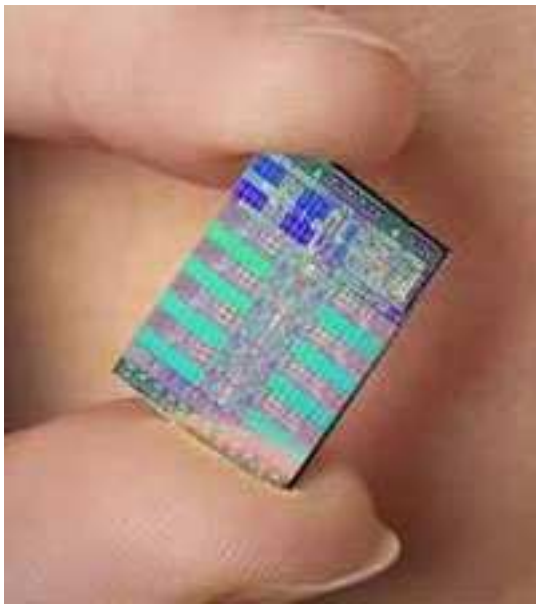


Figura 3.1 Processador Cell. Fonte: Érick Luiz Wutke Ribeiro

Neste capítulo, iremos desenvolver o assunto anterior de forma mais específica e aprofundada, fazendo um estudo de caso de arquitetura multi-core, o projeto Cell.

A idéia inicial deste projeto nasceu em 1999, quando o homem conhecido como o *"Pai do PlayStation"*, Ken Kutaragi, teve a idéia de criar um processador para a próxima geração de seu console que se comportasse como células em um organismo.

A arquitetura de microprocessador denominada *Cell Broadband Engine Architecture* foi desenvolvida conjuntamente pela aliança criada pelas companhias Sony, Toshiba e IBM, conhecida como *STI*. Esta aliança surgiu da experiência prévia, muito bem sucedida, no desenvolvimento do PlayStation 2, no qual a Sony e a Toshiba participaram efetivamente. A inclusão da IBM aconteceu pois este processador não seria de aplicação exclusiva no console da Sony, mas sim de propósitos gerais, então era necessária uma empresa que possuísse conhecimento no desenvolvimento de circuitos.

"Though sold as a game console, what will in fact enter the home is a Cell-based computer." - Ken Kutaragi

O custo da criação da arquitetura e sua primeira implementação girou em torno de 400 milhões de dólares segundo a IBM e durou cerca de 4 anos, iniciados no primeiro trimestre de 2001.

Essa longa extensão de tempo se deu devido às diferentes necessidades que as empresas tinham deste produto, o consenso é que era preciso criar um processador de alto desempenho que não possuísse grandes custos de produção (fabricação) e operação (gasto de energia).

O processador Cell então ficou como uma ponte entre os processadores convencionais e processadores especializados de alto desempenho, como por exemplo processadores gráficos de placas de vídeo, e, assim como estes, tem performance potencial muito grande. De acordo com a IBM, o Cell executa alguns programas 10 vezes mais rápido que processadores convencionais, isso pode parecer absurdo porém a *GPGPU.org* (General-Purpose Computation Using Graphics Hardware) possui estudos que comprovam que algumas placas de vídeo atuais alcançam e até superam este desempenho.

Seu desempenho operando em 4GHz é notável como podemos ver a seguir:

- 256 GFLOPS(*) utilizando ponto flutuante de precisão simples
- 256 GOPS(**) utilizando inteiros
- 25 GFLOPS utilizando ponto flutuante de precisão dupla

(*)GFLOPS: Bilhão de operações de ponto flutuante por segundo
(**)GOPS: Bilhão de operações por segundo

Podemos perceber que o Cell é otimizado para cálculos de ponto flutuante de precisão simples, sendo capaz de realizar cálculos de precisão dupla também, porém há uma perda de desempenho de uma ordem de grandeza.

3.1 Arquitetura

Dado o foco deste trabalho ser ambientes multi-core, será dada ênfase à parte da arquitetura que inclui os núcleos do processador e sua comunicação básica e não todos os elementos que a compõe.

Basicamente, um processador Cell possui um núcleo central denominado *Power Processor Element* (PPE) e múltiplos núcleos cooperativos, os *Synergistic Processing Elements* (SPEs), conectados através de um barramento de alto desempenho, o *Element Interconnect Bus* (EIB).

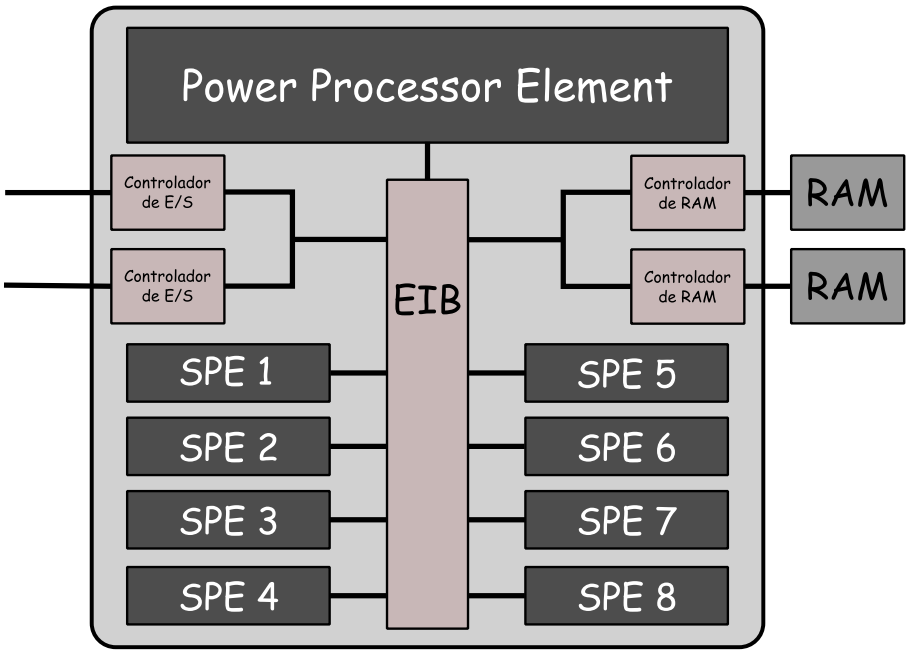


Figura 3.2 Arquitetura do processador Cell

3.1.1 Power Processor Element

O PPE é um núcleo de arquitetura POWER convencional de 64 bits e 512 Kb de cache que gerencia as tarefas que são passadas aos SPEs, ele executa o sistema operacional e os aplicativos, porém tarefas específicas que demandam de alto poder computacional são então repassadas.

Ele ainda possui tecnologia semelhante a tecnologia Hyper-Threading, permitindo que execute duas tarefas paralelamente.

Para tornar sua arquitetura o mais simples possível, foi utilizada a filosofia RISC (Reduced Instruction Set Computer) que favorece um conjunto de instruções reduzido em tamanho e complexidade.

Operando de maneira diferente dos processadores mais modernos convencionais que executam o maior número de instruções possível, executando-as seguindo o padrão *Out-of-Order* (fora de ordem), a aposta da IBM para o processador Cell foi na simplicidade e no alto desempenho, aumentando o clock do processador e mesmo assim gastando menos energia por sua simplicidade.

Apesar deste padrão fazer com que o processador utilize os ciclos que seriam perdidos por alguma espera por dados, ele torna o processador mais complexo, necessitando de uma quantidade grande de circuitos adicionais e demandando mais energia

Este design porém pode ser o ponto fraco do processador pois algumas aplicações mal implementadas podem gerar a perda de muitos ciclos e a mal/não utilização dos SPEs, ou seja, apesar do ganho de desempenho e economia de energia, é mais difícil programar para este processador.

3.1.2 Synergistic Processor Elements

Os SPEs são núcleos de processamento de filosofia RISC, assim como o PPE, porém são organizados como SIMD (Single Instruction, Multiple Data) de 128 bits, esta organização o torna um processador vetorial, que difere do escalar por ser capaz de realizar operações matemáticas em múltiplos dados simultaneamente. Portanto, para utilizar toda a capacidade de um SPE, todo o programa a ser executado tem que ser vetorizado.

Apesar de existirem compiladores com a função de "vetorizar"código, esta é uma funcionalidade ainda pouco explorada e não apresenta grandes resultados ainda, por isso o trabalho de se construir código totalmente otimizado para processadores vetoriais ainda é dos programadores, adicionando mais um empecilho ao desenvolvimento de aplicações para o Cell.

Uma das grandes diferenças entre os SPEs e núcleos de processadores comuns é a utilização de *"local stores"* (memórias locais) de 256 Kb no lugar de *caches*. Essa troca foi feita para que a arquitetura do processador ficasse mais simples, pois quando há *cache* para cada núcleo e dois ou mais núcleos estão utilizando os mesmos dados, a exclusão mútua do acesso e da escrita na memória têm que ser feita em *hardware*.

Como existe a comunicação entre os *"local stores"* através do EIB, o processador simplesmente executa uma tarefa em um SPE, passa o resultado para outro SPE que então executa a sua tarefa e só no final o resultado volta a memória do computador, ou seja, ainda é economizado um passo de cópia de memória ao cache.

3.1.3 Element Interconnect Bus

O EIB é o barramento de comunicação interna do Cell que conecta os elementos que o compõe. Seu poder de transferência é fantástico, atingindo teóricos 384 Gb/s através de três comunicações simultâneas, apesar da IBM afirmar que somente dois terços desse potencial pode ser atingido na prática. Seguindo regras simples, ele é otimizando para a transição de dados grandes. Realizar uma grande quantidade de pequenas operações faz com que seu desempenho seja prejudicado.

3.2 Aplicações

3.2.1 PlayStation 3

Figura 3.3 Console Sony PlayStation 3

O terceiro console de video-game da Sony foi a grande aplicação inicial do Cell. O investimento em sua produção foi alto, criando o console de hardware mais poderoso da atual geração de video-games, que inclui ainda o Nintendo Wii e o Microsoft Xbox 360.

No PlayStation 3, o processador Cell utilizado vem com 7 SPEs ativos ao invés de 8, para não se perderem processadores no controle de qualidade inicial, com seu clock é ajustado para 3,2 GHz, um pouco abaixo de seu potencial para garantir total estabilidade e grande vida útil.

Apesar de sua superioridade, o console está sendo classificado como um fracasso pois seu antecessor, o PlayStation 2, era líder disparado de mercado e seu sucessor é o que menos vende de sua geração. Este insucesso se deve à grande espectativa que foi criada em torno de sua capacidade e seus pífios os resultados posteriores.

Os desenvolvedores de jogos vêm tendo grande dificuldade em utilizar corretamente o Cell em suas "aplicações", tornando seu desempenho semelhante aos demais por um custo de hardware maior.

3.2.2 RoadRunner

Em fase de desenvolvimento, o supercomputador da IBM RoadRunner utilizará processadores Cell com algumas alterações, tornando-o otimizado a cálculos de ponto flutuante de dupla precisão, associados a processadores de servidor da AMD.

Estimasse que ele possuiá mais de 32 mil processadores e será o primeiro computador a ultrapassar a marca de 1 petaflop (quadrilhão de operações de ponto flutuante por segundo), atingindo cerca de 1,6 petaflops.

Este supercomputador será utilizado pelo Departamento de Energia dos Estados Unidos da América para simular o comportamento de material nuclear ao longo do tempo, certificando a segurança e a confiabilidade do arsenal nuclear americano criado durante o período da guerra fria.

3.2.3 Decodificação de sinais digitais

A Toshiba tem planos de criar decodificadores de sinais digitais de televisão que utilizem o Cell. A decodificação destes sinais demanda de um grande poder de processamento em tempo real que este processador pode prover.

O Cell pode operar de modo que os SPEs sejam encadeados em um fluxo, fazendo com que cada núcleo tenha uma função específica, como por exemplo decodificações de vídeo e audio, processamento de brilho e contraste e correção de erros no sinal, tudo sendo processado paralelamente, tendo como resultado a transmissão de sinal de alta definição em tempo real.

CAPÍTULO 4

Thread

Neste capítulo, iremos desenvolver o tema *thread*. Definição, propósito e funcionamento serão abordados. Depois falaremos da implementação que foi utilizada no nosso projeto, as Pthreads

4.1 Definição

Thread é um conjunto de instruções, que define uma tarefa, que pode ser executado simultaneamente ao processo que o criou. Muitas *threads* podem ser executadas paralelamente, sejam elas pertencentes ao mesmo processo ou a processos diferentes.

Em processadores single-core, essa concorrência é simulada através do escalonamento dos recursos do processador. Já em processadores multi-core há execução paralela real, embora também exista o escalonamento. Ele é tão veloz que passa ao usuário a sensação real de paralelismo nas execuções das threads e dos processos, mesmo quando ele não existe.

4.2 Threads vs Processos

A grande diferença entre *threads* e processos é que diferentes *threads* de um processo podem compartilhar recursos enquanto processos não.

Processos possuem na memória todos os recursos para serem executados independentemente, guardando informações de estado, conjunto de instruções a serem executadas, entre outras coisas. Além disso, só podem comunicar-se através de mecanismos do sistema especialmente desenvolvidos para este fim, sendo muito mais custosos que os mecanismos utilizados pelas threads e que serão explicados a seguir.

As *Threads* podem compartilhar recursos diretamente, necessitando apenas de ferramentas que auxiliem o programador a garantir a exclusão mútua para que não ocorram perdas de dados ou *deadlocks* (impasses entre threads que impedem a continuidade de suas execuções), a mais utilizada dentre estas é o semáforo.

A criação de processos também é mais custosa que a criação de *threads*, veja nas tabelas alguns tempos de criação obtidos através de um programa simples que media o tempo gasto em um número pré-definido de pthread_create() e de fork() em milissegundos:

Threads	1.000	5.000	10.000
Real Time	57	188	375
User Time	16	64	144
System Time	20	68	148

Processos	1.000	5.000	10.000
Real Time	593	3.176	10.972
User Time	12	92	208
System Time	248	3.064	10.677

4.3 POSIX Threads

POSIX (*Portable Operating System Interface*) é o nome de um conjunto de padrões que definem a API (*Application Programming Interface*) dos softwares compatíveis com sistemas Unix. POSIX Threads é o padrão referente a criação e manipulação de threads, bibliotecas que implementam este padrão são comumente chamadas de PThreads.

Para a realização do nosso projeto, utilizamos a biblioteca pthread.h para Linux, implementada em C e presente na GNU C Library.

Iremos listar as chamadas da biblioteca que utilizamos para realizar o nosso projeto:

Chamadas de manipulação:

- pthread_create()
  Função que faz uma thread executar um método passado.
- pthread_join()
  Função que esperara pelo término da thread passada.
- pthread_exit()
  Função que termina a thread passada.

Chamadas de sincronização:

- pthread_mutex_init()
- pthread_mutex_destroy()
- pthread_mutex_lock()
- pthread_mutex_unlock()

CAPÍTULO 5

Biblioteca

A biblioteca tem como funcionalidade marcar os tempos dos processos que a utilizam para que possam ser analisados posteriormente pelo interpretador.

5.1 Necessidade

Como o usuário não tem controle sobre como o sistema operacional escalona as threads de um processo, a criação da biblioteca é muito importante para que se possa fazer a análise dos processos corretamente, com seus tempos reais. Sem uma ferramenta como esta, os dados colhidos poderiam se distanciar da realidade, levando a uma análise inútil de dados potencialmente aleatórios.

5.2 Desenvolvimento

Os métodos da biblioteca foram criados para serem o mais semelhante possível da biblioteca usual de *threads*, a maioria deles possui os mesmos parâmetros e não é possível diferenciá-los pela simples utilização. Ela foi criada dessa forma para que fosse portável e não fossem necessárias grandes alterações nos códigos fonte das aplicações a serem analisadas.

Basicamente, as chamadas de métodos anotam no arquivo de saída suas características temporais, como por exemplo: os tempos de criação e finalização de uma *thread*, quando houve a tentativa de aquisição de um mutex e quando realmente ele o adquiriu e ainda quando ele foi liberado.

5.2.1 Marcação de tempos

Para a marcação de tempos reais, foi utilizada a rotina *times()* definida na biblioteca <sys/times.h> do linux. A chamada dessa função devolve uma estrutura que armazena o tempo de usuário e o tempo de sistema do processo que a requisitou. Por tempo de usuário entende-se o tempo que a *thread* permaneceu em execução no processador e por tempo de sistema, todo o tempo restante gasto com chamadas de sistema.

Um dos grandes problemas encontrados durante esta fase do desenvolvimento foi como seria marcado o tempo final de uma thread. Marcá-lo com uma chamada, como foi feito para a marcação dos tempos de aquisição e liberação de *mutexes* não seria possível, pois apenas uma chamada de médoto é passada ao método de criação de thread; marcá-lo com uma chamada feita pelo usuário no código de seu programa seria uma forma deselegante e não seria simples de modificar programas prontos para a utilização da biblioteca; uma outra forma seria fazer o processo esperar pela execução e então marcá-lo, porém esta forma acabaria com a

funcionalidade da biblioteca pois as *threads* não seriam nunca executadas paralelamente.

A solução encontrada para a resolução deste problema foi a criação de um método oculto na biblioteca que marca os seus próprios tempos inicial e final e, em meio a esta marcação, executa um método passado como parâmetro. Quando o usuário chama o método de criação de *thread* da biblioteca, o método que ele passa como parâmetro para ser executado pela *thread* é na realidade repassado como parâmetro a essa função e então é criada uma *thread* de sistema para executá-la. Como é a *thread* que faz então as marcações de tempo necessárias, o programa pode continuar sendo executado mantendo as características originais do mesmo quanto ao paralelismo das *threads* do processo.

Para os métodos que envolvem chamadas relacionadas a *mutexes*, a simples marcação do tempo da chamada é satisfatória para a análise que será realizada, pois o usuário normalmente já faz essas chamadas no momento em que elas devem ter seus tempos marcados.

5.2.2 Indentificação de threads e mutexes

A identificação das *threads* é simples, o ponteiro de *thread* utilizado como parâmetro para as funções é um identificador do tipo *unsigned long*, então existe o campo adequado, no entanto os *mutexes* não possuem um campo identificador para os mesmos. A solução mais simples encontrada foi utilizar o endereço de memória onde o *mutex* está alocado para identificá-lo, isso solucionou todos os problemas de identificação das estruturas.

5.3 Utilização

Para utilizar a biblioteca, basta incluir seu cabeçalho e utilizar seus métodos de forma intuitiva, seguindo basicamente os mesmos passos utilizados com a biblioteca *pthread*. Um cuidado imprescindível a se ter é de chamar no início do programa a rotina initializeVariables que recebe como parâmetro o nome do arquivo de saída onde estarão as informações após a execução do processo, este método cria este arquivo e aloca um *mutex* que será utilizado internamente para garantir que o arquivo não possua acessos de escrita simultâneos.

Ao fim do código, basta incluir uma chamada para a rotina destroyVariables que fecha o *stream* do arquivo e destrói seu mutex.

CAPÍTULO 6

Interpretador

O interpretador de arquivo é a base da avaliação de um programa, através dele será possível verificar como é a execução e testar como possíveis alterações na ordem das instruções do código podem otimizar um algoritmo.

6.1 Leitor

6.1.1 Linguagem

Após a geração do arquivo de saída, havia a necessidade da criação de um leitor que recuperasse as informações e as armazenasse em uma estrutura. Esta estrutura será explicada posteriormente, porém podemos afirmar que é a partir dela que a análise do processo do usuário será realizada.

Assim, a linguagem escolhida foi JAVA, pois ela possui muitas estruturas já implementadas e isso facilitaria muito o trabalho.

6.1.2 Blocos de código

A partir do arquivo de saída, temos as informações necessárias sobre como avaliar o desempenho do programa executado. Cada linha deste arquivo representa um evento ocorrido (p.ex., criação de uma *thread* ou a liberação de um *mutex*).

Na estrutura, estes eventos ocorridos são considerados como blocos de eventos. Um bloco necessariamente possui um dos seguintes tipos: CODE, MUTEX e WAIT. As partes de algoritmo situadas em regiões críticas, ou seja, que não podem ser executadas concorrentemente a outras de mesma característica, foram denominadas blocos de mutex pois é essa estrutura que garante essa proteção. Partes denominadas blocos de código são sequencias contínuas de instruções que não podem ter sua ordem alterada e portanto formam uma única estrutura. Blocos de espera representam o tempo que uma thread fica ociosa aguardando a aquisição de um mutex que já está sendo utilizado.

Assim, ao final da captura das informações do arquivo de saída, haverá uma lista de representantes de *threads* com os seus respectivos blocos de eventos.

6.2 Algoritmo

Após a leitura do arquivo e a criação dos blocos de código e mutex, podemos verificar a existência de blocos de espera. Pode-se afirmar que uma maneira de otimizar um processo é minimizar a quantidade de blocos de espera e suas durações, fazendo com que as threads fiquem improdutivas o menor tempo possível.

Com este conceito em mente, a ferramenta permite ao usuário, alternar as ordens de blocos de código e de mutex, buscando uma execução do processo que minimize a existência dos blocos de espera. Neste ponto, não há o devido controle sobre a necessidade de uma ordem pré-definida de execução de alguns blocos, fica a critério do usuário, utilizar as alterações de ordem de blocos, sem que ele altere o comportamento de seu algoritmo. A ferramenta se encarrega de analisar a ordem dos blocos ordenados pelo usuário e criar os blocos de espera necessários para que não haja, na simulação, o erro de se executar regiões críticas concorrentemente.

6.2.1 Política de escalonamento

A prioridade para a obtenção de um mutex no algoritmo é sempre da thread que o requeriu antes, ou seja, foi utilizada a política FIFO (First In, First Out) para garantir a justiça no escalonamento do mutex. Esta pode não ser a política de escalonamento que faz com que todas as threads encerrem suas execuções no menor tempo possível, mas é o suficiente para a contagem do tempo perdido nos blocos de espera.

6.3 Saída Gráfica - *Applet*

Uma das principais necessidades do nosso projeto era possibilitar o usuário de fazer uma análise técnica de maneira fácil do desempenho de seu programa. Assim, a solução escolhida foi criar um *applet*, que abre automaticamente ao executar o interpretador da biblioteca.

6.3.1 Desenvolvimento

Através da linguagem de programação JAVA é possível gerar, de forma rápida e intuitiva, vários tipos de gráficos (p.ex., gráfico de "pizza"ou de barra). Uma boa maneira de representar threads e o seu tempo de execução é através do gráfico de barras. Portanto, a fim de se produzir

um gráfico de fácil compreensão, cada barra representa uma thread com seus blocos de códigos. O tipo de cada bloco é o fator que os diferenciam na barra[1]. É necessário explicar que, ao fim de cada bloco, haverá um número que representa o momento em que o ele termina. Além disso, haverá também um número no topo da barra de cada *thread* mostrando a duração da *thread*, ou seja, tempo final - tempo inicial.

Assim que executado, a ferramenta criará o gráfico. Porém, para cada alteração na ordem dos blocos, realizada pelo usuário, o *applet* deverá ser executado novamente para que um novo gráfico seja gerado.

Em uma etapa futura, existe o propósito de se criar um gráfico dinâmico, aonde o usuário selecionará um determinado bloco e haverá a possibilidade de escolher se ele deve subir ou descer. Entretanto, até esta etapa ser atingida, a implementação de um algoritmo que avalie não somente blocos de códigos independentes, mas blocos que possuem uma relação deverá estar concluída.

[1]Verde é o bloco de código escolhido pelo usuário; Vermelho é o tempo de ociosidade; Amarelo é o tempo de retenção do *mutex*

CAPÍTULO 7

Exemplo

Os conceitos explicados anteriormente ficarão mais claros com a dissecação detalhada de um exemplo.

7.1 Execução

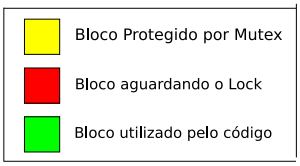
O exemplo a seguir mostra como o usuário pode influenciar no desempenho de um processo. Para testar a ferramenta, foi implementado um programa-teste. Após executado o programa, foi obtido o seguinte arquivo de saída:

5;3076717456;0;10
2;3076717456;134520796;10;7;10;7
5;3085110160;0;16
3;3076717456;134520796;18;19
2;3085110160;134520796;16;19;18;19
5;3076717456;18;29
3;3085110160;134520796;32;45
2;3076717456;134520796;29;45;32;45
5;3068324752;74;86
5;3085110160;32;99
1;3085110160;0;0;99;149
3;3076717456;134520796;103;149
1;3076717456;0;0;103;149
2;3068324752;134520796;86;134;103;149
3;3068324752;134520796;106;149
5;3068324752;106;111
1;3068324752;74;112;111;149

A partir do arquivo de saída gerado pela ferramenta, montamos a simulação da execução do processo. A imagem a seguir mostra como foi a execução do processo exemplo sem a interferência do usuário:

A seguir veremos uma análise detalhada do gráfico.

103
103
32
29
18
10
99
99
32
18
16
37
111
106
103
86
Thread3076717456
Thread3085110160
Thread3068324752
Thread Tool Applet


Bloco Protegido por Mutex
Bloco aguardando o Lock
Bloco utilizado pelo código

7.2 Interpretação dos dados

Veremos agora uma interpretação suscinta a partir da figura 1.

Primeiramente note que foram criadas três *threads*, cada uma com o seu respectivo *id* (3076717456, 3085110160 e 3068324752).

- *Thread* 307671745:

  Pode-se notar que esta *thread* foi iniciada no tempo 0 e possui 5 blocos. Inicialmente a *thread* executa um trecho de código durante um tempo de 10 ms(milissegundos). A seguir, é requisitado e obtido o *lock* do *mutex*, ou seja, não houve tempo de espera. Após 8 ms, este bloco do *mutex* foi finalizado, dando inicio a um outro bloco de código. Depois de terminado este ultimo bloco, é feita uma outra requisição do *lock*, porém desta vez a *thread* teve que aguardar 3 ms para conseguí-la. E por fim, o bloco *mutex* acabou após 71 ms.

  A *thread* teve uma duração de 103 ms.

- *Thread* 3085110160:

  Semelhantemente à primeira *thread*, esta também teve seu início no tempo 0, porém possui 4 blocos de código. O primeiro bloco, tipo *CODE*, teve uma duração de 16 ms. Em seguida foi requisitado o *lock*, entretanto sua aquisição se deu depois de 2 ms. Após a sua utilização, o *lock* é liberado depois de 14 ms. A *thread* acaba com um bloco de código, que foi executado em 67 ms.

  A *thread* possui uma duração de 99 ms.

- *Thread* 3068324752:

  Ao contrário das duas primeiras *threads*, esta teve seu início no tempo 74. O primeiro dos 4 blocos é executado durante 12 ms, este é do tipo *CODE*. Em seguida é requisitado o *lock*, porém passam-se 17 ms para a sua obtenção. Após 3 ms o *lock* é liberado e executa-se o último bloco.

  Esta *thread* é finalizada no tempo 111, entretando sua duração é de 37ms.

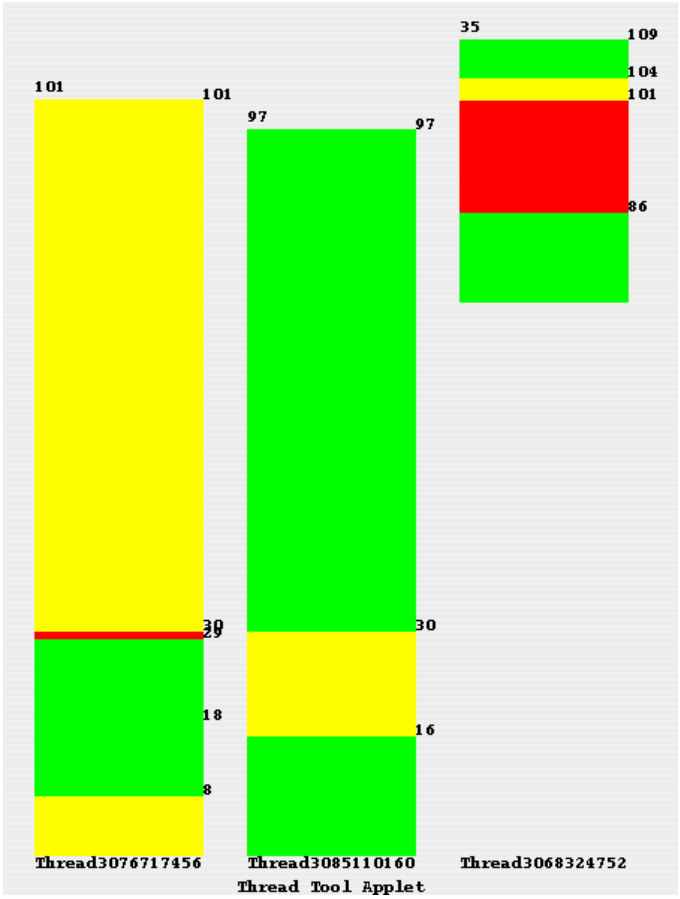
A seguir veremos algumas maneiras de se conseguir um melhor tempo de execução das *threads*.

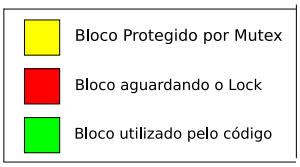
7.3 Utilização da ferramenta

Existem algumas maneiras de se obter um melhor desempenho do programa (p.ex., aproveitar o tempo ocioso da *thread* para executar um bloco de código, ou fazer com que a chamada do *lock* seja adiantada para um momento em que ninguém a utilize).

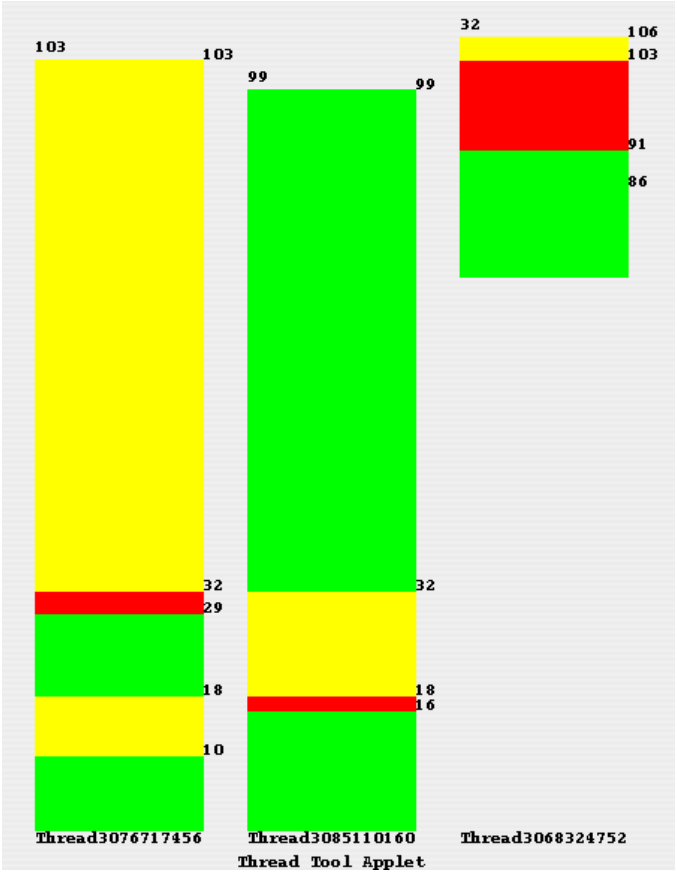
Neste exemplo podemos aplicar pelo menos duas modificações na figura 1:

1. Na *thread* 307671745 podemos trocar o primeiro bloco de código pelo primeiro bloco do *mutex*. Este método fará com que a chamada do *lock* seja adiantada, podendo diminuir o tempo de espera de outras *threads*. Veja a figura do gráfico após a mudança:





2. Na *thread* 3068324752 podemos colocar o último bloco de código para ser executado no momento em que a *thread* encontra-se ociosa. Veja a figura do gráfico após esta mudança:



É válido dizer que estas não são as únicas mudanças possíveis. Este exemplo tem o intuito apenas de demonstrar a ferramenta.

7.4 Análise dos resultados

Após a primeira modificação, podemos notar que todas as *threads* tiveram o seu tempo de duração diminuído significadamente. Cada uma das *threads* ganhou 2 ms no total de sua execução.

Note que inicialmente, a execução perdia 22 ms em espera. Depois da primeira mudança este tempo caiu para 16 ms, cerca de 27% mais eficiente.

Na segunda mudança, as *threads* 307671745 e 3085110160 não ganharam nenhum tempo. Porém a última *thread* ganhou 5 ms no seu tempo total de execução. Consequentemente o tempo total perdido pelo programa caiu de 22 ms para 17 ms, cerca de 23% mais eficiente.

Fazendo essas alterações no programa real, a primeira alteração resultou num ganho de desempenho de 3 ms no total de sua execução, enquanto a segunda alteração resultou num ganho de 4 ms. Mostrando que neste caso, conseguimos obter um resultado bem aproximado do real com a nossa avaliação.

Vale ressaltar que o nosso intuito é de criar uma ferramenta que possibilite tal análise. O quão eficiente o desempenho do programa do usuário pode ser melhorado fica a cargo de quem a utiliza.

CAPÍTULO 8

Parte Subjetiva

Neste ano de 2007 optei em fazer o meu trabalho de formatura com o professor Marco Dimas Gubitoso, professor no qual cursei as matérias de Laboratório de Programação I e II e programação Paralela.

O tema inicialmente sugerido pelo professor era estudar, realizar testes e modificar a maneira de como era realizada o escalonamento no processador CELL. Assim, no primeiro semestre de 2007, juntamente com os alunos Márcio Guedes Hasegawa (IME-USP) e Bruno Silva (POLI-USP) nos dedicamos na compreensão deste novo processador multi core. Estudamos a sua arquitetura e principalmente como ele faz uso dos sete processadores efetivos do seu interior.

Após realizado um estudo inicial, o que nos deu uma certa familiaridade, começamos a pesquisar uma maneira de manipularmos *threads* ou processos. Neste momento, final do primeiro semestre, descobrimos que não seria possível conseguir um computador que possuísse o processador CELL. Isto gerou uma certa decepção, mas a situação se tornou um pouco pior assim que ficamos sabendo que o aluno Bruno havia se desanimado e achou melhor sair do grupo.

O professor Marco Dimas Gubitoso nos deu uma outra idéia, que apesar de não aproveitar totalmente o estudo feito no primeiro semestre, seria uma boa alternativa para darmos sequência ao nosso projeto. Ao invés de realizar testes em um processador em específico, criaríamos um simulador de ambiente com a quantidade de núcleos desejada. Assim o usuário final saberia avaliar o quão eficiente seria executar o seu programa em um ambiente com um número de processadores estipulado. Então, no segundo semestre de 2007 nos dedicamos para criar este simulador. Inicialmente fizemos teste com *PThreads*, e em seguida fomos incrementando nossa aplicação a fim de concretizar o nosso projeto.

Após o ano letivo de 2007, nosso orientador sugeriu uma outra alteração no projeto. Visando aproveitar o que ja estava implementado, deveríamos criar uma ferramenta que permitiria ao usuário analisar o desempenho de uma função. Assim, nas férias nos focamos em criar um arquivo com as informações das *threads* criadas pela função do usuário, o leitor deste arquivo,

e a implementação da ferramenta, juntamente com uma saída gráfica.

Pretendemos dar continuidade neste projeto a fim de atingirmos um nível em que a ferramenta seja capaz de permutar a sequência de blocos dinamicamente, respeitando as possíveis dependencias existentes. Desta forma, proporcionará uma maior interatividade e flexibilidade em seu uso.

O professor Marco Dimas Gubitoso pretende utilizar esta ferramenta em suas aulas de MAC0431 - Introdução à Computação Paralela e Distribuída.

A seguir, as matérias que me ajudaram a desenvolver o projeto:

- MAC0122 - Princípios de Desenvolvimento de Algoritmos
- MAC0211 - Laboratório de Programação I
- MAC0242 - Laboratório de Programação II
- MAC0412 - Organização de Computadores
- MAC0431 - Introdução à Computação Paralela e Distribuída
- MAC0438 - Programação concorrente

Referências Bibliográficas

[1] The cell project at IBM research.
`http://www.research.ibm.com/cell/`.

[2] Intel multi-core technology.
`http://www.intel.com/multi-core/`.

[3] pthread.h API.
`http://opengroup.org/onlinepubs/007908799/xsh/pthread.h.html`.