Grand Central Dispatch

Marcio Rocha dos Santos

Trabalho de conclusão de curso

Orientador: Prof. Dr. Alfredo Goldman vel Lejbman

Co-Orientador: Emilio De Camargo Francesquini

São Paulo, Dezembro de 2010

Sumário

Capítulo 1

Introdução

No passado, o melhor caminho para os fabricantes de chips melhorarem o desempenho dos computadores, foi o de aumentar a velocidade do clock do processador. Mas não demorou muito a esbarrar num conjunto de limitantes físicos. um desses limitantes é a chamada *power wall* ou "barreira de potência", que ocorre devido ao alto tráfego de energia através dos chips, e a perda dessa energia na forma de calor pode até fazer com que o chip alcance a temperatura de fusão do metal que o compõe, arruinando-o completamente. Os sistemas atuais de arrefecimento já estão com certa dificuldade de dissipar esse calor de um modo eficiente e barato.

Entretanto, como a demanda por computadores de alto desempenho continua crescendo, a indústria mudou para chips com múltiplos núcleos e clocks menores, que pode fornecer mais desempenho, consumindo menos energia. Mas para tirar o máximo proveito destes processadores, um novo modo de programar deve acompanhar o desenvolvimento da nova arquitetura. Porém, apesar dos avanços na área de hardware, a computação paralela não tem se desenvolvido rápido o suficiente. A computação enfrenta uma crise com a falta de programadores qualificados nessa área e de ferramentas que auxiliem a paralelização. Mesmo programas atuais que se utilizam de dois ou quatro processadores não são aptos para os *manycores* do futuro com dezenas de núcleos e centenas de threads.

A programação paralela é difícil com a gestão manual de threads. No entanto, os melhores programadores do mundo são pressionados a criar grandes programas multithread em lingua-

gens de baixo nível como C ou C + +, garantindo exclusão mútua, sem ocorrência de deadlock, condição de corrida, e outros perigos inerentes ao uso de múltiplas threads simultaneamente. Aplicações extremamente cuidadosas de bloqueio primitivo se fazem necessárias para evitar queda de desempenho e bugs, ao lidar com dados compartilhados. Portanto nosso problema é: hardware com mais recursos computacionais do que os programadores sabem lidar, na sua maioria completamente ocioso, e ao mesmo tempo o usuário está totalmente impedido de usar tais recursos.

Estudaremos uma solução da Apple, já implementada na versão Snow Leopard do MAC OS X, que promete tornar o trabalho dos desenvolvedores de software mais fácil, utilizando os recursos do hardware ao máximo. Essa solução da Apple se chamada Grand Central Dispatch (GCD), uma biblioteca C introduzida nos níveis mais baixos do sistema operacional.

Capítulo 2

História

A primeira geração de computadores estabeleceu os conceitos básicos de organização dos computadores eletrônicos, e na década de 50 apareceu o modelo computacional que se tornaria a base de todo o desenvolvimento subsequente, chamado "modelo de von Neumann". O modelo de von Neumann é composto basicamente por processador, memória e dispositivos de E/S. O processador executa instruções sequencialmente, de acordo com a ordem ditada por uma unidade de controle.

Figura 1.1 - Arquitetura de von Neumann

O problema dessa arquitetura chama-se gargalo de von Neumann. Quando as arquiteturas eram mais simples, funcionava bem. Porém, foram surgindo necessidades de melhora, tais como: aumentar a velocidade dos dispositivos, o que depende do desenvolvimento tecnológico; executar tarefas em paralelo.

O avanço dos dispositivos foi acontecendo de forma gradual até os anos 80, com o surgimento da filosofia VLSI. (Very-large-scale integration (VLSI) é o processo de criação de circuitos integrados, combinando milhares de transistores em um único chip) Historicamente, a utilização de paralelismo começou pelo hardware. O aumento da velocidade do hardware, por mais rápida que possa ser, sempre esbarra no limite tecnológico de cada época. Dessa forma, desde o início do desenvolvimento dos computadores, percebeu-se que a utilização de paralelismo é essencial. A paralelização proporcionou grandes mudanças nas arquiteturas, que

evoluíram ao longo dos anos com o surgimento de diversas características nas máquinas, tais como:

- em 1953, surgiu o conceito de palavra, levando à manipulação de vários bits em paralelo, ao invés da unidade;
- em 1958, processadores de E/S possibilitaram o paralelismo entre E/S e processamento, tornando a tarefa de E/S mais independente;
- em 1963, como memória também era o gargalo do sistema, surgiram técnicas como cache e interleave (que permite a obtenção de mais de uma palavra na memória, em paralelo);
- em 1970, aparecimento de pipelines, paralelizando a execução de instruções, operações aritméticas, etc;
- no período de 1953 a 1970, o paralelismo existente é transparente ao usuário, influenciando apenas na melhoria do desempenho da máquina, sendo denominado paralelismo de baixo nível.
- em 1975, nasceu o Illiac IV, com 64 processadores, primeira forma em que o paralelismo diferia do de baixo nível. Foi o computador mais rápido até o aparecimento do Cray;
- em 1984, surgimento da filosofia VLSI de projeto de microcomputadores, oferecendo maior facilidade e menor custo no projeto de microprocessadores cada vez mais complexos e menores (computadores pessoais).

A partir do aparecimento dos computadores pessoais e a redução dos custos dos componentes, foi possível colocar dois ou mais processadores em máquinas paralelas. E enfim o surgimento, em 2005, dos processadores multicore no mercado, com o Intel®Pentium®D, deu inicio a era da arquitetura multicore.

Capítulo 3

Conceitos

3.1 Programação paralela

A programação sequencial consiste na execução de várias tarefas, uma após a outra. A programação concorrente é definida como a execução simultânea de várias instruções. Pode também ser formada por várias instruções que foram iniciadas em determinado instante e ainda não finalizaram. A programação concorrente pode ser executada tanto em um só processador, explorando o que se chama de pseudo-paralelismo, quanto em uma arquitetura paralela, com vários processadores. No caso de vários processadores, essa programação passa a incorporar a chamada programação paralela, que caracteriza-se pela execução de várias tarefas em um mesmo instante. Embora relacionada com programação paralela, programação concorrente foca mais na interação entre as tarefas.

Algumas definições de programação paralela: Almasi: "Coleção de elementos de processamento que se comunicam e cooperam entre si e resolvem um problema mais rapidamente". Quinn : "Processamento de informação que enfatiza a manipulação concorrente de dados que pertencem a um ou mais processos que resolvem um único problema". Hwang: "Forma eficiente do processamento de informações com ênfase na exploração de eventos concorrentes no processo computacional"

As vantagens da utilização de programação paralela consistem em: alto desempenho, acabando com o chamado "gargalo de von Neumann". É o objetivo principal da programação

paralela. A divisão de um problema em diversas tarefas pode levar a uma redução considerável no tempo de execução; substituição dos computadores de altíssimo custo. As arquiteturas paralelas apresentam um custo menor que uma arquitetura com um único processador e a mesma capacidade; tolerância a falhas, pois a programação paralela facilita a implementação de mecanismos de tolerância a falhas; modularidade, sendo que um programa longo pode ser subdividido em várias tarefas que executam independentemente e se comunicam entre si.

O uso de programação paralela tem desvantagens como: programação mais complexa, pois acarreta em uma série de problemas inexistentes na programação sequencial, tais como comunicação, sincronismo, balanceamento de carga, etc; sobrecargas introduzidas para suprir fatores como comunicação e sincronismo entre as tarefas em paralelo. Essas sobrecargas devem ser cuidadosamente analisadas, pois impedem que seja alcançado o speedup ideal.

3.2 Speedup e eficiência

uma característica fundamental da computação paralela trata-se do aumento de velocidade de processamento através da utilização do paralelismo. Neste contexto, duas medidas importantes para a verificação da qualidade de algoritmos paralelos são speedup e eficiência. uma definição largamente aceita para speedup é: aumento de velocidade observado quando se executa um determinado processo em p processadores em relação à execução deste processo em 1 processador. Então, tem-se:

$$speedup = \frac{T_1}{T_p}$$

onde,

- T_1 = tempo de execução em 1 processador;
- T_p = tempo de execução em p processadores.

Idealmente, o ganho de speedup deveria tender a p, que seria o seu valor ideal. Porém, três fatores podem ser citados que influenciam essa relação, gerando sobrecargas que diminuem o valor de speedup ideal: sobrecarga da comunicação entre os processadores, partes do código executável estritamente sequenciais e o nível de paralelismo utilizado (em virtude do uso de

granulação inadequada à arquitetura). Outra medida importante é a eficiência, que trata da relação entre o speedup e o número de processadores.

$$Eficiencia = \frac{speedup}{p}$$

No caso ideal ($speedup = p$), a eficiência seria máxima e teria valor 1 (100%).

3.3 Programação concorrente

um programa sequencial é composto por um conjunto de instruções que são executadas sequencialmente, sendo que a execução dessas instruções é denominada um processo. um programa concorrente especifica dois ou mais programas sequenciais que podem ser executados concorrentemente como processos paralelos [1].

A programação concorrente existe para que se forneçam ferramentas para a construção de programas paralelos, de maneira que se consiga melhor desempenho e melhor utilização do hardware paralelo disponível. A computação paralela apresenta vantagens importantes em relação à computação sequencial, como foi exposto acima, e todas essas vantagens podem ser citadas como pontos de incentivo para o uso da programação concorrente.

um programa sequencial é constituído basicamente de um conjunto de construções já bem dominadas pelo programador em geral, como por exemplo, atribuições, comandos de decisão (if... then... else), laços (for... do), entre outras. um programa concorrente, além dessas primitivas básicas, necessita de novas construções que o permitam tratar aspectos decorrentes da execução paralela dos vários processos. A fase de definição da organização das tarefas paralelas é de extrema importância, pois o ganho de desempenho adquirido da paralelização depende fortemente da melhor configuração das tarefas a serem executadas concorrentemente.

Na definição do algoritmo, é necessário um conjunto de ferramentas para que o programador possa representar a concorrência, definindo quais partes do código serão executadas sequencialmente e quais serão paralelas. As construções para definir, ativar e encerrar a execução de tarefas concorrentes são o foco da tecnologia a ser tratada nesta monografia.

Capítulo 4

Grand Cental Dispatch

Com o nome fazendo referência ao Grand Central Terminal, um importante terminal ferroviário e metroviário localizado em Manhattan, Nova Iorque, Grand Central Dispatch ou GCD é um sistema de gerenciamento de pools de threads em nível de sistema operacional, desenvolvido pela Apple e lançado junto ao MAC OS X 10.6 *Snow Leopard*. Combinando um modelo fácil de usar com melhor aproveitamento dos recursos de hardware disponíveis, a idéia central do GCD é transferir a responsabilidade do gerenciamento inteligente de threads para o sistema operacional. As vantagens, ao fazer uso desta tecnologia, são muito relevantes.

Mesmo a aplicação mais bem escrita pode não ter o melhor desempenho possível, pois para obter tal desempenho é necessário ter pleno conhecimento sobre tudo o que acontece no sistema em tempo de execução. E desse principio vem o diferencial do GCD. É muito mais simples para o programador escrever seus aplicativos sem se preocupar com a disponibilidade dos recursos no momento da execução. Os blocos de tarefas serão colocados em filas de execução e gerenciados pelo sistema operacional que tem controle total sobre os eventos do sistema. Sendo assim, o desenvolvedor de software só precisa escrever código compatível com o GCD, tentando paralelizar ao máximo seu código, para tirar o máximo proveito dos núcleos e processadores múltiplos, e confiar no sistema operacional.

A Apple disponibilizou os códigos da biblioteca da API de espaço de usuário (libdispatch) e das alterações realizadas no kernel XNU (o kernel de código aberto da Apple comum ao OS X e ao Darwin), sob os termos de Apache License, Version 2.0. Mas o GCD também depende

de uma extensão da linguagem, ainda não disponível no compilador GCC, para fazer uso dos blocks (estrutura que será abordada mais adiante). E o fato de que GCD é uma biblioteca C significa que ele pode ser usado de todos os derivados da linguagens C suportados no SO: Objective-C, C++ e Objective-C++. O suporte à extensão no compilador é um pré-requisito para os desenvolvedores de aplicativos que queiram tirar proveito do GCD.

O GCD é semelhante a outras tecnologias, como OpenMP ou ao TBB da Intel. Os três trabalham com certo tipo de abstração à criação de threads, permitindo que o desenvolvedor determine partes do código como "tarefas"a serem paralelizadas de alguma forma. Porém, o GCD faz uso dos "blocos", em vez das diretivas de pré-processador do OpenMP ou dos modelos do TBB.

Como dito, GCD é implementado com um conjunto de extensões para a linguagem C, uma nova API e mecanismo de "runtime". O GCD permite programar várias unidades de trabalho usando quatro abstrações:

- Block objects;
- Dispatch queues;
- Synchronization;
- Event sources.

4.1 Block Objects

Block Objects (informalmente, "blocos") é uma extensão da linguagem C, assim como Objective-C e C++, que tornam mais fácil para os programadores definirem unidades independentes de trabalho. Os blocos são semelhantes a ponteiros de função tradicional, porém com algumas características de objeto. As principais diferenças são:

- Os blocos podem ser definidos em linha, como "funções anônimas";
- Blocos fazem cópias das variáveis do escopo pai com permissão somente para leitura, similar as "closures"em outras linguagens.

Esse tipo de funcionalidade é comum em linguagens interpretadas, dinamicamente tipadas, mas não era disponível para programadores C. A Apple publicou as especificações de blocos e suas implementações em código aberto sob a licença MIT, adicionou suporte a blocos através do projeto compiler-rt, GCC 4.2 e clang, e apresentou suas considerações como parte da próxima versão da linguagem C.

4.1.1 Sintaxe

Uma variável block se parece como um ponteiro pra função, exceto que se usa o acento circunflexo em vez de um asterisco.

```
void (^meuBloco)(void);
```

A variável resultante pode ser inicializada, atribuindo a ela um bloco literal com a mesma assinatura (argumentos e tipos de retorno).

```
meuBloco = ^(void){ printf("hello world\n"); };
```

Esta variável pode ser invocada como um ponteiro de função:

```
meuBloco(); // imprime "hello world\n"
```

4.2 Dispatch queues

Dispatch queues (Filas de despacho) é a umas das principais abstrações do GCD. É através das filas que os desenvolvedores programam unidades de trabalho para serem executadas de forma concorrente ou serial. Com a vantagem de ser muito fácil de usar e a promessa de maior eficiência, Dispatch queues podem ser usadas em todas as tarefas que são executadas em threads diferentes.

Uma fila dispatch é a estrutura que serve de interface entre a aplicação e o mecanismo de execução. *Tarefas* (unidades independentes de trabalho) são incluídas e retiradas das filas com operações atômicas que garantem a confiabilidade dos dados. Assim sendo, filas dispatch

são um caminho fácil para executar *tarefas* de forma concorrente ou serial numa aplicação. Todas as filas dispatch são estruturas first-in, first-out (FIFO). Desta forma, a execução de uma *tarefa* adicionada na fila respeita a ordem em que foi adicionada. O GCD já fornece algumas filas dispatch pré-definidas, mas outras filas podem ser criadas pelo programador para propósitos específicos. Vamos ver os tipos de fila dispatch fornecidas pelo GCD e os respectivos funcionamentos.

Serial queues (Filas seriais) ou filas privadas são geralmente usadas para controlar o acesso a recursos específicos como regiões críticas. Cada fila serial tem suas tarefas executadas uma de cada vez e em ordem como já mencionado acima, mas a execução de tarefas em filas seriais distintas podem ser concorrentes.

Concurrent queues (Filas de concorrência) ou filas globais têm suas tarefas executadas simultaneamente em quantidade variada que depende das condições do sistema. A execução das tarefas ocorre em threads distintas gerenciadas pela fila dispatch. O GCD fornece três filas de concorrência ao aplicativo que se diferem apenas pela prioridade, e o programador não pode criar novas filas de concorrência.

Main dispatch queue (Fila principal) se trata da fila serial que tem a execução de suas tarefas na thread principal do aplicativo. Frequentemente utilizada como ponto chave para sincronização das tarefas.

4.3 Synchronization

Quatro mecanismos primários são fornecidos pelo Grand Central Dispatch para melhor controle da finalização das tarefas assíncronas.

- **Synchronous Dispatch** - Muitas vezes é necessário saber quanto um bloco terminou sua execução. Uma forma muito simples é adicionar o bloco na fila:

  ```
  dispatch_sync(uma_fila, ^{ espere_por_mim(); });
  ```

  Todavia, esse recurso requer atenção do processo pai, que fica parado aguardando até que o bloco chamado finalize sua execução.

- **Callbacks**;
- **Groups**;
- **Semaphores**.

4.4 Event sources

Capítulo 5

Atividades realizadas

5.1 Explorando as funcionalidades

O Grand Central Dispatch é uma tecnologia bem recente com poucas fontes de documentação. Por este motivo, suas funcionalidades foram testadas de diversas formas para que seu uso fosse bem entendido. Dessa forma, os benefícios que o GCD trouxe puderam ser melhor explorados.

Fazendo uso de um MacBook white com processador Intel Core 2 Duo 2.26GHz, 2GB de memória rodando MAC OS X 10.6.5, não houve problema algum ao deixar a máquina preparada para programação com a biblioteca *libdispatch*. O computador veio de fábrica com a versão *Snow Leopard* do MAC OS X, que contém suporte nativo ao GCD, foi necessário instalar apenas o pacote de programação Xcode, para que o computador ficasse apto a compilar e executar os programas que fazem uso da biblioteca *libdispatch* e dos blocos.

A primeira ferramenta a ser examinada foi o Block object. Desde as variações de sua sintaxe, acesso a variáveis do escopo pai, suas variáveis de escopo, à execução de suas operações, uma boa abordagem do uso de blocos pôde ser feita. Em seguida, foi a vez das ferramentas de sincronização e bloqueio. Tarefas com tempo de funcionamento variado permitiram examinar ferramentas de sincronização, concorrências entre as tarefas e acesso exclusivo a regiões críticas.

5.2 Testes de desempenho e Resultados obtidos

Após dar uma boa olhada nas facilidades e utilidades abordadas pelas ferramentas disponíveis do Grand Central Dispatch, chegou a hora de avaliar o desempenho das aplicações constituídas com a tecnologia.

O GCD foi feito pela Apple, que é uma empresa privada com fins lucrativos. A maioria das publicações que falam sobre o GCD é da própria Apple, com muitas propagandas, prometendo melhor desempenho dos aplicativos, aproveitamento máximo dos núcleos ou processadores, e tudo respaldado, é claro, pela incrível facilidade de uso. Segundo a própria Apple, o overhead gerado com a criação manual de inúmeras threads é evitado no GCD, pois as tarefas aguardam a disponibilidade de processadores lógicos para serem executadas. As instruções de encapsulamento de tarefas são curtas, portanto eficientes. Incluir ou remover tarefas das filas são operações atômicas disponíveis no hardware, o que assegura a confiabilidade dos dados e a rapidez da operação. Portanto, os testes feitos visavam verificar as reais vantagens, ou desvantagens, do GCD como ferramenta usada pelos programadores para paralelizar seus códigos e criar eficientes aplicativos.

5.2.1 Tarefas crescentes

Os primeiros testes foram feitos com aplicativos que tem seu trabalho facilmente paralelizável. Implementamos um algoritmo para o cálculo do produto de matrizes quadradas, por se tratar da escolha mais natural e intuitiva para os primeiros testes.

A ideia foi deixar que a geração de cada linha da matriz, resultante do produto, fosse gerada por uma tarefa independente. Matrizes do tipo *double* foram geradas dinamicamente com entradas adquiridas de uma função que gerava números aleatórios. No estudo das ferramentas do GCD dei mais atenção a duas formas de lançar várias tarefas em paralelo e aguardar a conclusão das mesmas, usando a chamada dispatch_apply() e dispatch_group_async(). Esse foco duplo é devido ao fato de executar uma série de vezes uma função contendo um "sleep(aleatório)" fazendo uso da chamada dispatch_apply(), o sistema permitia somente duas tarefas em execução por vez, provavelmente devido ao fato de estar sendo executado em um computador com dois núcleos. Já com o uso da chamada dispatch_group_async(), todas as

tarefas foram iniciadas.

Os testes de produtos de matrizes quadradas de dimensões diferentes comparam administração manual de threads, chamadas dispatch_apply() e dispatch_group_async(). Observem trechos de código:

Trechos dos código serão acrescentados nessa área assim que eu conseguir fazer isso no Latex de forma elegantes. Por hora os codigos estarão em : linux.ime.usp.br/ marcio/-mac499/src/

Os resultados não mostraram vantagem do GCD sobre o uso manual de threads, pelo contrário, conforme cresce o tamanho da matriz o GCD vai tendo uma desvantagem clara, frente ao uso manual de threads. Resultados semelhantes foram obtidos ao aplicar testes no problema de encontrar números primos, em que cada tarefa independente procurava primos num intervalo distinto, mas de mesmo tamanho.

A princípio, esse tipo de teste não foi intencional, mas percebemos que esses problemas tinha uma característica comum. No produto de matrizes, o tamanho de cada unidade de trabalho dependia da dimensão da matriz, pois o tamanho da linha é proporcional a dimensão da matriz. Na busca por números primos, o tamanho das unidades de trabalho também acompanharam o escopo, pois quanto mais distante um número está de 1, maior é o numero de comparações que a tarefa deve executar.

5.2.2 Tarefas curtas

O incentivo dos desenvolvedores do GCD é que os programadores compartilhem ao máximo o trabalho a ser realizado por seus aplicativos em tarefas menores, distribua essas tarefas nas filas da melhor maneira possível e deixe o resto por conta do GCD. Dessa forma, teríamos um grande número de tarefas curtas nos aplicativos a serem executadas. Para simular esse comportamento, testamos o desempenho de aplicativos com grandes números de tarefas curtas. O teste que realizamos foi reproduzido com tarefas que executavam um número fixo de operações de soma e produtos de números “reais” gerados aleatoriamente. Assim, o tamanho de cada tarefa não tem relação com o número de tarefas. Observe o trecho de código das tarefas:

Trecho de código aqui.

Resultados de desempenho aqui.

5.2.3 Divisão do trabalho

Muitas vezes, temos que conhecer o hardware em que o software será rodado, para escreve-lo de maneira a obter o melhor desempenho possível. Entretanto, sabemos que não é uma tarefa fácil escrever o software pensando o tempo todo em como será o comportamento do mesmo em um hardware específico. E ainda podemos perder em desempenho ao rodar o aplicativo em hardware diferente. O GCD promete ser uma arma muito poderosa para na solução desse tipo de problema. A administração das threads por conta do S.O. permite programar sem se preocupar com a disponibilidade dos recursos. Nosso objetivo nesse testes é verificar essa afirmação.

No teste anterior, foi possível observar certa vantagem do GCD com inúmeras tarefas. A grande quantidade de tarefas de tamanho fixo nos deu indícios fortes de que o uso manual de Thread pode ter desempenho menor que o GCD com overhead muito grande. Nesse novo teste, o trabalho total a ser realizado pelo aplicativo não cresce com o tamanho de entrada. É muito comum softwares com conjunto de instruções muito estáveis, ou conjunto fixo de instruções. Então, foi estudado o desempenho das tecnologias ao dividir o trabalho a ser realizado pelo aplicativo em tarefas cada vez menores.

Criamos duas matrizes do tipo *double* quadradas, de ordem 512, para calcular uma matriz resltante do produto dessas duas matrizes é necessário 512^2 produtos de vetores. Então o total desses produtos foram distribuidos entre 2 tarefa, depois 8, 32, 128, 256, 512, 2028, 8192, 16384 e 32768 tarefas. Observe os Resultados:

Trecho de código aqui.

Resultados de desempenho aqui.

Capítulo 6

Conclusões

Nos testes efetuados até então, os aplicativos implementados com uso da tecnologia fornecida pelo Grand Central Dispatch obtiveram resultados favoráveis, no que diz respeito ao desempenho dos mesmos, quando dividimos o trabalho a ser realizado pelo aplicativo em várias outras tarefas menores. Algo em seu mecanismo de gerenciamento de threads não o deixa em vantagem quando o trabalho a ser realizado por uma tarefa independente não é tão pequeno quanto se deseja, e isso se mostrou mais claro conforme aumentamos o numero dessas tarefas e seus respectivos tamanhos.

Embora o GCD não resolva o problema de decidir quais as partes do trabalho podem ser realizadas em paralelo, o que ainda é um problema difícil, quando o programados resolve esse problema o GCD tem uma participação muito importante no restante das trabalho. A clareza, legibilidade dos códigos e até facilidade de distribuição das tarefas são vantagens muito relevantes.

O GCD realmente é uma tecnologia preparada para o futuro, em que os computadores terão centenas ou milhares de núcleos em seus processadores e os códigos escritos hoje não precisarão ser refeitos para aproveitar essas novas vantagens. E por se tratar de uma tecnologia disponibilizada à comunidade de software livre, a difusão de seu uso e possíveis aperfeiçoamentos é algo bem real.

Referências Bibliográficas

[1] ANDREWS, G. R., Schineider, F. B., *Concepts and Notations for Concurrent Programming*, ACM Computing Survey, v. 15, no.1, pp. 3-43,1983. 7

Computação Paralela - USP - São Carlos - R.H.C. Santana, M.J. Santana, M.A. Souza, P.S.L. Souza, A.E.T. Piekarski

http://arstechnica.com/apple/reviews/2009/08/mac-os-x-10-6.ars/

http://libdispatch.macosforge.org/

http://lwn.net/Articles/352978/

http://developer.apple.com/technologies/mac/snowleopard/gcd.html

http://developer.apple.com/ GCD /Reference/reference.html