

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

LEANDRO ZULIAN GALLINA

**Avaliação de Desempenho do OpenMP
em Arquiteturas Paralelas**

Prof. Nicolas Maillard
Orientador

Porto Alegre, Novembro de 2006

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Reitor: Prof. José Carlos Ferraz Hennemann

Vice-reitor: Prof. Pedro Cezar Dutra Fonseca

Pró-Reitor de Graduação: Prof. Carlos Alexandre Neto

Diretor do Instituto de Informática: Prof. Philippe Olivier Alexandre Navaux

Coordenador do CIC: Prof. Raul Weber

Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

AGRADECIMENTOS

Agradeço especialmente aos meus pais, Luiz Antonio e Marilene, os quais, mesmo à distância, me respaldaram com todo o apoio e carinho necessários durante toda a minha vida, o que me ajudou a concluir o curso de graduação na Universidade Federal do Rio Grande do Sul.

Agradeço ao meu irmão, Laércio, pelo apoio e pela amizade.

Agradeço aos meus colegas de faculdade, em especial a Carlos Loth, Carlos Moreira, Edson Bicca, Felipe Giacomel, Márcio Mello, Peter Elbern, Rafael de Abreu e Roberto Rosenfeld, e também a suas famílias, por terem sido, de alguma forma ou outra, a família que eu não tive em Porto Alegre.

Agradeço aos professores do Instituto de Informática da UFRGS pelo trabalho realizado e pela educação gratuita e de qualidade que me proporcionaram.

Agradeço a todos que, de alguma forma, colaboraram com a conclusão deste curso e a realização deste sonho.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS	6
LISTA DE FIGURAS	8
RESUMO	9
ABSTRACT	10
1 INTRODUÇÃO	11
2 ARQUITETURAS DE COMPUTADORES PARALELAS	13
2.1 Lei de Amdahl	14
2.1.1 Falhas na Lei de Amdahl	15
2.2 Arquiteturas de Processadores Single-Core	15
2.3 Threads Baseadas em Hardware; TMT e SMT	16
2.4 Processadores com Threads de Hardware	17
2.5 Tecnologia Hyper-Threading (HT)	18
2.6 Processadores Multi-core	18
2.7 Perspectivas de Penetração no Mercado	19
3 THREADS	21
3.1 Definição de Thread	21
3.2 Threads a Nível de Usuário	21
3.3 Threads a Nível de Sistema Operacional	22
3.3.1 Mapeamento 1:1	23
3.3.2 Mapeamento N:1	23
3.4 Threads a Nível de Hardware	23
3.5 Criação de Threads	24
3.6 Threads POSIX	24
3.6.1 Criação de Threads com a Biblioteca Pthreads	25
3.6.2 Gerência de Múltiplas Threads	26
3.6.3 Mecanismos de Sincronização	27
3.6.4 Desvantagens da Biblioteca Pthreads	29
4 OPENMP	30
4.1 Disponibilidade do OpenMP em Compiladores	31
4.2 Programação com OpenMP	31
4.3 Exemplo Completo: Cálculo de Pi	32
4.4 Diretivas para trabalhar com variáveis	34

4.5	Diretivas de Distribuição de Tarefas	37
5	AVALIAÇÃO DE DESEMPENHO DO OPENMP	42
5.1	Benchmarks Criados	42
5.2	Benchmarks NAS	45
6	CONCLUSÃO	48
	REFERÊNCIAS	50

LISTA DE ABREVIATURAS E SIGLAS

ALU	Arithmetic and Logic Unit
API	Application Programming Interface
APIC	Advanced Programmable Interrupt Controller
CFD	Computational Fluid Dynamics
CMP	Chip Multi-Processing
CPU	Central Processing Unit
EPIC	Explicitly Parallel Instruction Computing
FSB	Front Side Bus
GCC	Gnu C/C++ Compiler
GOMP	Gnu OpenMP
HT	Hyper-Threading
ICC	Intel C/C++ Compiler
ILP	Instruction Level Parallelism
LLC	Last Level Cache
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
MMX	Multimedia Extensions
NAS	NASA Advanced Supercomputing
NASA	National Aeronautics and Space Administration
NPB	NAS Parallel Benchmarks
OpenMP	Open Multi Processing
PCB	Process Control Block
RAM	Random Access Memory
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
SMP	Symmetric Multi-Processing

SMT	Simultaneous Multi-Threading
SSE	Streaming SIMD Extensions
SSE2	Streaming SIMD Extensions 2
SSE3	Streaming SIMD Extensions 3
TLP	Thread Level Parallelism
TMT	Time Multi-Threading

LISTA DE FIGURAS

Figura 2.1: Evolução dos preços médios do Intel D805 em 2006	19
Figura 2.2: Evolução dos preços médios do Intel Pentium HT em 2006	20
Figura 5.1: Desempenho do OpenMP no cálculo de π	43
Figura 5.2: Desempenho do OpenMP na multiplicação de matrizes quadradas	43
Figura 5.3: Detalhe do desempenho do OpenMP na multiplicação de matrizes quadradas	44
Figura 5.4: Desempenho do OpenMP com as diretivas dynamic e static	44
Figura 5.5: Comparação de desempenho entre GCC e ICC nos benchmarks BT, CG, EP e FT	46
Figura 5.6: Comparação de desempenho entre GCC e ICC nos benchmarks LU, MG e SP	47

RESUMO

A tendência atual da tecnologia de processadores é agregar cada vez mais núcleos em um mesmo processador. Assim, escalabilidade e portabilidade estão se tornando necessidades cada vez maiores nos aplicativos de software. O uso do OpenMP permite que o código seja paralelizado fácil e eficazmente. Este trabalho visa analisar algumas das principais estruturas do OpenMP e executar benchmarks para verificar o ganho de desempenho obtido com seu uso.

Com o propósito de verificar o impacto que a utilização de um compilador específico exerce no desempenho do OpenMP, os benchmarks serão executados nos compiladores ICC da Intel e GCC da Gnu.

Palavras-chave: OpenMP, Arquitetura, Compiladores, Portabilidade, Escalabilidade, Multi-Core.

OpenMP Performance in Parallel Architectures

ABSTRACT

Technology trends point to a future where increasingly more cores will be found in the same processor. This brings a growing need for scalability and portability in software applications. OpenMP allows code to be parallelized easily and effectively. The objective of this paper is to analyze some of the main constructs of OpenMP and run benchmarks on it to study the gain of performance OpenMP can bring.

The choice of a compiler used with OpenMP causes great effect on its performance. To best verify this impact, the benchmarks will be run in two compilers: ICC and GCC.

Keywords: OpenMP, Architecture, Compilers, Portability, Scalability, Multi-Core.

1 INTRODUÇÃO

A tecnologia de semicondutores avançou rapidamente nas últimas décadas. Os processadores se tornaram progressivamente mais rápidos, em um ritmo veloz.

Empresas fabricantes como Intel e AMD têm uma necessidade bem definida: continuar produzindo processadores com performance cada vez melhor. Da maneira com que foi conduzida no último quarto de século, o aumento de desempenho dos processadores com um único núcleo (“single-core”) decorre basicamente de dois fatores: a diminuição do tamanho de seus componentes e o aumento da velocidade do *clock* do processador. Estes componentes físicos estão alcançando um tamanho tão pequeno que em breve eles não poderão mais diminuir de tamanho.

Na medida que este limite físico se aproxima, os fabricantes de chips começaram a se voltar em direção aos processadores com vários núcleos, os multi-core. Ao tomar vantagem do paralelismo existente nos programas, o processador multi-core apresenta ganho em desempenho baseado no número de cores que oferece, e não no tamanho de seus componentes ou relógio do processador. A tendência é que, em um futuro próximo, todos os computadores para uso pessoal estejam dotados de processadores multi-core.

Ao passo que este futuro se aproxima, cabe aos desenvolvedores de software criar programas que sejam escaláveis. Afinal, que utilidade teria um processador com 64 cores para um programa executando naquele computador com apenas uma thread? É muito provável que um programa escrito nas APIs atuais precise ser totalmente modificado no momento em que ele precisar ser adaptado para uma máquina com mais processadores. Isto traz uma dificuldade que impede o usuário de usufruir plenamente dos ganhos de desempenho trazidos pelos processadores multi-core.

Para suprir esta necessidade, foi criado o padrão OpenMP. Através da aplicação de diretivas sobre o código já existente, o usuário do OpenMP não precisa modificar radicalmente sua aplicação para contar com os benefícios de um ambiente multi-processado.

O objetivo deste trabalho é apresentar a arquitetura de computadores com múltiplos processadores, mostrar o OpenMP como solução para criação de aplicações paralelas e avaliar seu desempenho nestas máquinas.

No capítulo 2 são apresentados conceitos básicos de arquiteturas de computadores, preparando o usuário para conhecer as arquiteturas paralelas e a diferença entre computadores multi-core e multi-processados.

No capítulo 3 é apresentado o conceito de threads e algumas das APIs disponibilizadas atualmente ao programador. Também serão conhecidas algumas das desvantagens trazidas.

O quarto capítulo apresenta o OpenMP, com algumas das principais construções

que o programador deve conhecer para criar um aplicativo paralelo.

O quinto capítulo traz uma avaliação de desempenho do OpenMP em diversos benchmarks. Para calcular o impacto causado pela escolha de um compilador com OpenMP, os benchmarks serão executados nos compiladores ICC e GCC para que seu desempenho seja comparado.

2 ARQUITETURAS DE COMPUTADORES PARALELAS

Para obter a execução de software de forma paralela dentro de um sistema de computador, o hardware deve oferecer uma plataforma capaz de executar simultaneamente múltiplas threads.

Em relação ao paralelismo de hardware que disponibilizam, as arquiteturas de computador podem ser classificadas em duas dimensões diferentes. Uma é em relação aos fluxos de instruções que um computador de uma arquitetura pode executar ao mesmo tempo. A outra é em relação aos fluxos de dados que podem ser processados simultaneamente. Assim, existem quatro combinações diferentes sob as quais se encontram todas as arquiteturas de computador existentes, formando aquilo que se chama de Taxonomia de Flynn (1).

A máquina mais simples sob essa taxonomia é a SISD (Single Instruction, Single Data). Trata-se de um computador seqüencial tradicional que não provê nenhum paralelismo a nível de hardware. Apenas um fluxo de dados é processado pelo computador em um determinado momento. Estes computadores estão atualmente obsoletos, mas eram máquinas bastante comuns no início da computação doméstica.

Em seguida existe a máquina MISD (Multiple Instruction, Single Data), que seria um computador onde múltiplas instruções processam um único fluxo de dados ao mesmo tempo. Esta é uma máquina hipotética, que não possuiria muita utilidade se construída: normalmente, múltiplos fluxos de instruções precisam de múltiplos fluxos de dados para ser úteis.

Uma máquina SIMD (Single Instruction, Multiple Data) é aquela em que um único fluxo de instruções pode processar diversos fluxos de dados ao mesmo tempo. A maior parte dos computadores domésticos atuais provê algum tipo de processamento SIMD. O processador Pentium IV da Intel disponibiliza instruções MMX, SSE (Streaming SIMD Extensions), SSE2 e SSE3, capazes de processar múltiplos dados em único ciclo de relógio. A AMD comercializa processadores com instruções SIMD dotados de uma tecnologia chamada 3DNow! Os primeiros computadores com instruções SIMD existentes surgiram em computadores vetoriais como o Cray-1.

Por último encontra-se a máquina MIMD (Multiple Instruction, Multiple Data), onde são executados diversos fluxos de instruções acessando diferentes fluxos de dados ao mesmo tempo. Este é o modelo encontrado em plataformas multi-core atuais.

2.1 Lei de Amdahl

Para medir os benefícios que uma arquitetura multi-core é capaz de oferecer, é necessário algum tipo de métrica que forneça em números o ganho obtido ao executar paralelamente uma determinada aplicação nesta arquitetura. Em outras palavras, obtém-se a razão entre o tempo de execução da aplicação rodando de forma puramente seqüencial e o tempo de execução da mesma aplicação rodando de forma paralela.

A Lei de Amdahl (2) é uma das fórmulas que tenta explicar o ganho obtido em uma aplicação ao adicionar mais cores de processamento. O norueguês Gene Amdahl iniciou suas pesquisas seguindo a idéia de que um trecho do programa é puramente seqüencial, e o desempenho deste trecho não é aumentado quando aumenta o número de threads de execução. Apenas o desempenho do trecho que não é seqüencial pode ser aumentado. Assim, o ganho de performance G é dado pela seguinte fórmula:

$$G = \frac{1}{S + \frac{1-S}{n}}$$

onde S é o tempo gasto executando a parte seqüencial do programa e n é o número de cores de processamento.

Por exemplo, suponha um programa que possui 60% do seu código puramente seqüencial, e os 40% restantes do código podem ser executados por múltiplas threads. Se colocarmos este programa para executar em uma máquina dual-core, aplicando a Lei de Amdahl temos:

$$\frac{1}{0.6 + \frac{0.4}{2}} = \frac{1}{0.8} = 1.25$$

Portanto, o ganho de performance será de 25%.

Se o número de processadores ou cores tende ao infinito, o limite da fórmula de Amdahl tende a:

$$\frac{1}{S}$$

ou seja, o ganho máximo que se pode obter depende da porção do código que não pode ser executada paralelamente. No exemplo acima, onde 60% do código do programa é puramente seqüencial, o limite do ganho será de:

$$\frac{1}{0.6} = 1.67$$

o que significa que o programa executará no máximo com um ganho de 67%, independente do número de cores que forem adicionados.

Observando a Lei de Amdahl, é fácil chegar à conclusão que é mais importante diminuir a parte seqüencial do código do que aumentar o número de cores de processamento.

Não se deve esquecer ainda a ação do overhead causado pelo uso de várias threads em um programa, pois ao usar threads o sistema operacional precisa de tempo para realizar a troca de contexto entre elas e a sincronização de dados. Assim, a Lei de Amdahl fica:

$$G = \frac{1}{S + \frac{1-S}{n} + H(n)}$$

2.1.1 Falhas na Lei de Amdahl

Devido às limitações teóricas previstas na Lei de Amdahl, diversos pesquisadores das décadas de 1970 e 1980 evitaram construir máquinas paralelas massivas, acreditando que o ganho seria limitado.

Contudo, a Lei de Amdahl possui várias pressupostos que não se aplicam no mundo real. Por exemplo, a Lei de Amdahl parte do pressuposto que o melhor algoritmo sequencial (aquele que executa quando o número de processadores $n = 1$) é limitado estritamente pela disponibilidade de ciclos de processador. Na verdade, se cada core do sistema multi-core estiver dotado de sua própria memória cache, uma porção maior de dados do programa será guardada na memória cache do sistema, reduzindo a latência de memória.

Outra falha da Lei de Amdahl é supor que a solução sequencial é a melhor possível. Porém, muitas vezes um problema é resolvido de forma mais eficiente quando usa várias threads.

Além disso, a Lei de Amdahl parte do princípio que um problema continua do mesmo tamanho quando aumenta o número de processadores disponíveis. Mas é freqüente que um problema aumente de tamanho quando possui mais recursos computacionais, de certa forma mantendo constante o tempo de execução da aplicação.

As descobertas acima puderam ser observadas na prática através do trabalho de pesquisadores como John Gustafson, que em 1988 mostrou seu hipercubo de 1024 processadores onde a Lei de Amdahl falhava em prever o ganho de desempenho obtido com a paralelização. Baseado neste trabalho, a seguinte fórmula foi proposta:

$$G = N + (1 - N) * s$$

onde N é o número de processadores e s é a razão entre o tempo gasto na execução da porção sequencial do programa pelo tempo total de execução. Esta fórmula costuma ser chamada de Lei de Gustafson e foi provada como sendo equivalente à Lei de Amdahl.

2.2 Arquiteturas de Processadores Single-Core

Para melhor entender o funcionamento das threads em processadores multi-core, é preciso antes rever os fundamentos da arquitetura de processadores single-core. Conhecer os detalhes de um processador é fundamental para obter melhor performance de algumas aplicações.

Chipset é o conjunto dos chips que ajudam os processadores a interagir com a memória física e outros componentes. Podemos chamar de chip um processador que não necessariamente possui capacidade central de processamento.

O processador é dividido em unidades funcionais, como a Unidade Lógica e Aritmética (ALU), as Control Units e as Prefetch Units (1). As unidades funcionais recebem instruções para executar operações. O conjunto de unidades funcionais forma o microprocessador, também conhecido como Unidade Central de Processamento (CPU). O processo de fabricação de um microprocessador produz um pacote que costuma ser chamado de processador. O processador é encaixado no sistema do computador através de um socket. Para simplicidade, neste trabalho os termos processador e microprocessador serão utilizados referindo-se à mesma entidade física.

Um processador busca instruções de software como entrada em um processo conhecido como *Fetch*. Em seguida, estas instruções são decodificadas para que sejam

entendidas pelo processador, na etapa de *Decode*. Depois de decodificadas, as instruções realizam tarefas específicas (*Execute*), e é produzida uma saída (*Write*). Todas estas operações são realizadas dentro dos blocos funcionais dentro de um processador e todos os estágios de pipeline ocorrem dentro dos limites do processador.

Dentro de um processador fica a memória cache, que se encontra dividida em níveis. A maioria dos processadores atualmente disponíveis no mercado disponibiliza uma cache de nível 1 (L1), menor e mais rápida, e uma cache de nível 2 (L2), que possui um tamanho maior – mas ainda assim, nada comparável ao tamanho da memória física (RAM) do computador. A maioria dos processadores de 32 bits atuais não disponibiliza ainda uma cache de nível 3. No Pentium IV, por exemplo, a cache L2 possui 512 Kb, e não existe cache de nível 3 (L3), a não ser na versão Pentium IV Extreme Edition, que possui cache L3 de 2 MB. Processadores da linha Intel Itanium possuem caches L3 de até 6 MB.

O processador possui uma unidade chamada de Local APIC que lida com as interrupções específicas do processador. Esta unidade não deve ser confundida com a I/O APIC, uma unidade fora do processador que costuma ser encontrada em sistemas baseados em múltiplos processadores.

A Unidade de Interface é o bloco funcional que faz a interface do processador com o barramento de sistema, também conhecido como Front Side Bus (FSB).

O vetor de registradores é o conjunto de registradores presentes em um processador. O número de registradores varia de um processador para outro. O processador Pentium possui apenas oito registradores inteiros, enquanto que processadores Itanium de 64 bits possuem 128 registradores inteiros.

Os recursos de execução incluem a ALU inteira, a execução de ponto flutuante e as unidades de branch. A velocidade com que os diferentes blocos funcionais executam suas operações é variável.

Para processadores superescalares como o Pentium, existem blocos funcionais que existem exclusivamente para o funcionamento dos pipelines. Uma destas unidades é o agendador (scheduler), que determina quando micro-operações estão prontas para executar baseado nas dependências destas micro-operações em recursos como os registradores.

2.3 Threads Baseadas em Hardware; TMT e SMT

Um processador single-core superescalar é capaz de suportar aplicações com múltiplas threads que parecem executar simultaneamente. Isto acontece através de um mecanismo chamado de paralelismo a nível de instruções de máquina (ILP – Instruction Level Parallelism). O processador realiza uma troca de contexto, preservando o estado atual da instrução sendo executada antes de trocar para outra instrução. Para esta operação de troca de contexto valer a pena, ela não deve demorar mais do que alguns ciclos do processador.

O processador mais simples que podemos encontrar em termos de threading é o processador escalar que executa uma única thread de cada vez. Com um processador destes, a possibilidade de executar múltiplas threads é gerenciada pelo sistema operacional, que decide qual processo vai tomar conta do recurso único de hardware existente.

Um processador um pouco mais elaborado oferece compartilhamento de recursos de hardware, e este compartilhamento pode ser grosso (*coarse-grained*) ou fino (*fine-*

grained). No caso do compartilhamento grosso, uma thread tem controle total sobre os recursos do processador por um período de tempo conhecido, o chamado quanta. No caso do compartilhamento fino, a troca de contexto de threads acontece a nível de instrução de hardware. Estes dois tipos de multi-threading são conhecidos como multi-threading a nível de tempo (TMT). Na aplicação sendo executada, o usuário tem a impressão que múltiplas threads são executadas simultaneamente. Porém, não existem recursos de hardware suficientes para que isso aconteça. Esta ilusão é criada pelo escalonador do sistema operacional e pelo agendador do hardware.

No caso de sistemas multi-processados ou com múltiplos processadores simétricos (SMP) com memória compartilhada, o mecanismo de paralelismo a nível de threads (TLP – Thread Level Parallelism) pode ser utilizado executando diversas threads em diversos processadores ao mesmo tempo. O escalonador do sistema operacional continua com a responsabilidade de determinar qual thread executa em qual processador em determinado momento. (Ele já tinha esta tarefa antes, porém agora múltiplos processadores estão disponíveis.) O multi-threading simultâneo (SMT) permite que várias threads entrem em competição pelos recursos existentes, combinando o ILP com o TLP. Este tipo de multi-threading é mais eficiente quando uma aplicação precisa de recursos de hardware complementares ao mesmo tempo. Um processador SMT converte a TLP em ILP, misturando o multi-threading fino com o grosso.

Um processador com dois ou mais cores é chamado de processador CMP (chip multi-processing). Cada core executa threads de hardware de forma independente dos outros, e todos eles acessam uma memória compartilhada para comunicação entre threads. No CMP, múltiplos processadores se encontram em uma mesma placa de silício.

2.4 Processadores com Threads de Hardware

A implementação de threads em hardware evoluiu de forma significativa desde os processadores escalares simples existentes. O primeiro processador superescalar lançado no mercado foi o Intel Pentium em 1993. No ano 2000, foi lançado o primeiro processador com tecnologia Hyper-Threading (HT). No mesmo ano, surgiram no mercado os processadores Itanium de 64 bits, com a arquitetura EPIC. Em 2005, a Intel lançou os primeiros processadores dual-core.

Um processador superescalar é aquele onde encontramos múltiplos pipelines. (Quando um processador possui um único pipeline, ele é denominado de processador escalar.) O primeiro processador superescalar disponível para os consumidores domésticos foi o Pentium em 1993. Esta arquitetura superescalar evoluiu transformando-se na microarquitetura Intel NetBurst, encontrada no processador Pentium IV.

A sigla EPIC significa Explicitly Parallel Instruction Computing. O principal produto hoje no mercado que segue esta linha é o Intel Itanium. Todos os processadores desta arquitetura são de 64 bits. Um código fonte que foi compilado para uma arquitetura superescalar não funciona em um processador da arquitetura EPIC. Isso acontece porque um compilador para a arquitetura EPIC deve compilar o código fonte tendo em vista a geração de código de máquina explicitamente paralelo. Enquanto um processador superescalar determina em tempo de execução quais instruções vão para quais pipelines do processador, em uma máquina EPIC

a maior parte deste trabalho é realizada pelo compilador. Assim, as múltiplas unidades de execução de uma arquitetura EPIC são usadas de forma mais eficiente, porém o compilador é mais complexo. A principal desvantagem desta solução é que o EPIC não é compatível com código de máquina já produzido para arquiteturas superescalares e suas antecessoras.

2.5 Tecnologia Hyper-Threading (HT)

A Tecnologia HT é um mecanismo de hardware onde múltiplas threads de hardware executam em um único ciclo em um processador single-core. Esta tecnologia foi a primeira solução SMT para processadores de uso doméstico da Intel.

Nos processadores HT disponíveis atualmente, duas threads podem executar em único core através do compartilhamento e replicação de alguns recursos do processador. Para o Sistema Operacional, é como se existissem dois processadores, os chamados processadores lógicos, permitindo que o escalonador coloque duas threads diferentes para executar ao mesmo tempo. Porém, existe um único processador físico, que é compartilhado entre as diferentes threads. Um processador com tecnologia HT é quase idêntico a um sem a tecnologia: eles possuem o mesmo número de registradores e ambos contêm apenas um processador físico. Contudo, o tamanho da placa de silício é maior, devido à replicação de recursos do processador.

2.6 Processadores Multi-core

Para entender o que é um processador multi-core, é necessário compreender a diferença entre um core e um processador. Em um processador single-core, o core é a soma de todos os blocos funcionais que participam diretamente da execução de instruções, podendo incluir a cache de último nível (LLC) e o barramento de sistema (FSB).

Se os diversos cores de um processador multi-core compartilharem a cache LLC, não existe problema em sincronizar o conteúdo das caches. Porém, é necessário manter alguma tag que associe cada linha da cache ao core correspondente, ou então dividir a cache entre os cores. Se os cores do processador multi-core compartilharem o FSB, o tráfego deste barramento é diminuído.

O número de cores em um processador multi-core é variável, mas eles são simétricos, apresentando-se em um número que é potência de 2. Os produtos atuais da Intel em multi-core, tratando-se do ano de 2006, possuem no máximo 2 cores, sendo assim denominados dual-cores.

Em maio de 2006, a AMD divulgou que colocará no mercado processadores quad-cores em 2007. Estes processadores seguirão a arquitetura do Athlon 64. Nos processadores dual-core da AMD existentes atualmente, existem dois níveis de memória cache, não compartilhada. Os processadores quad-core que serão lançados em 2007 serão dotados de uma memória cache L3 compartilhada pelos quatro cores. A Intel não possui nenhum produto quad-core planejado para 2007, apenas um chip de codinome “Clovertown” que é na verdade composto de dois processadores dual-core ocupando um único socket (8) (9).

Um multiprocessador representa vários processadores físicos distintos, que ocupam diferentes sockets dentro uma máquina. Um processador multi-core representa vários cores em único processador físico, ocupando um único socket dentro do sis-

tema. É possível que um sistema multiprocessador contenha vários processadores físicos que são multi-core.

2.7 Perspectivas de Penetração no Mercado

A tecnologia de semicondutores se desenvolveu muito rapidamente nas últimas décadas. Porém este ritmo de desenvolvimento parece estar próximo de um limite físico, um ponto a partir do qual os elementos de hardware se tornam tão pequenos que a tecnologia deixa de evoluir com a mesma intensidade ou então nem sequer evolui mais. Isto significa que a performance em processadores single-core tende a atingir um limite.

A solução imaginada por empresas como a Intel é partir para a comercialização em massa de processadores com múltiplos cores. A empresa acredita que a penetração dos multi-cores no mercado de desktop será de mais de 90% ao final de 2007, e de praticamente 100% no mercado de servidores. Estes dados refletem a expectativa da empresa nos mercados dos Estados Unidos, Europa e Japão.

No Brasil, os processadores dual-core demonstravam preços muito proibitivos no primeiro semestre de 2006, e ainda longe de ter a penetração de mercado esperada nos países citados anteriormente.

Durante o segundo semestre de 2006, estes processadores demonstraram uma queda bastante acentuada nos preços. O gráfico da figura 2.1 demonstra a evolução dos preços médios do processador dual-core Intel D805 2,66 GHz.

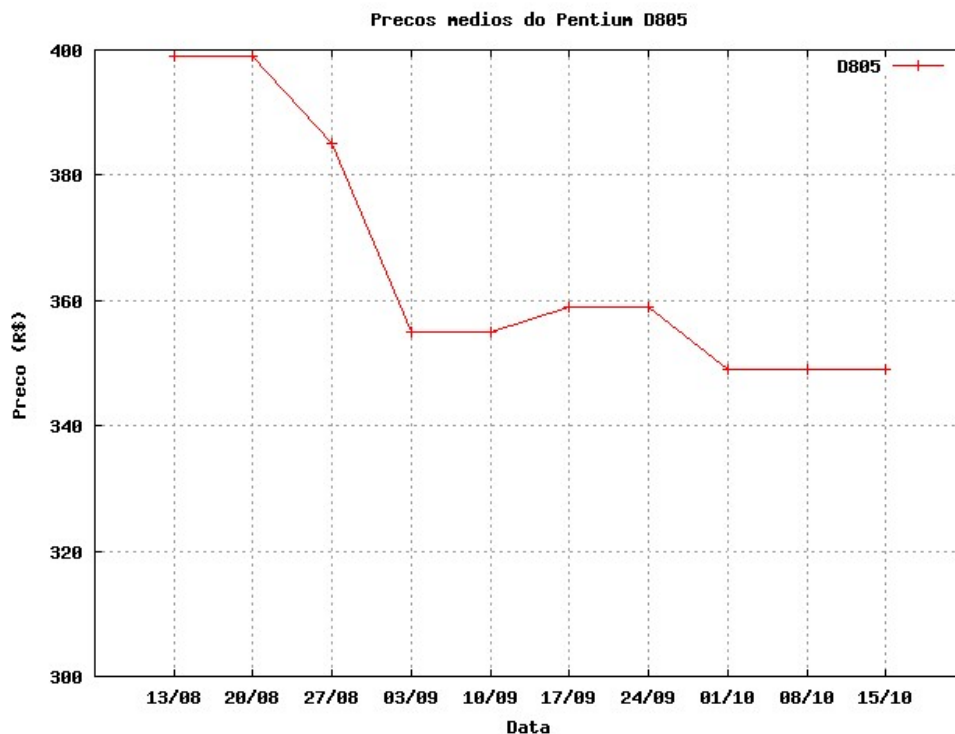


Figura 2.1: Evolução dos preços médios do Intel D805 em 2006

O gráfico da figura 2.2 exibe a evolução dos preços médios dos processadores Intel Pentium HT 3 GHz e Intel Pentium HT 3,2 GHz.

Como se pode observar, os preços tiveram uma queda bastante acentuada durante

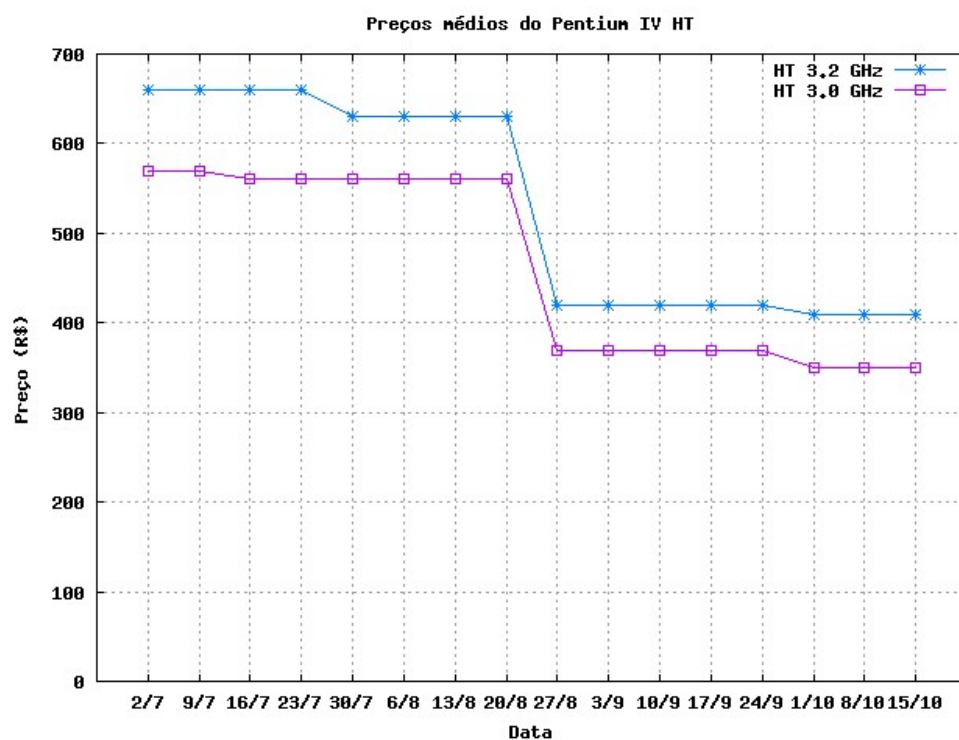


Figura 2.2: Evolução dos preços médios do Intel Pentium HT em 2006

o segundo semestre de 2006, mostrando que a tendência dos processadores multi-core está sendo bem absorvida pelos mercados brasileiros (10).

3 THREADS

Para aproveitar melhor os recursos de multi-threading oferecidos por processadores multi-core, é necessário conhecer os mecanismos de software que permitem ao programador usufruir desta melhora de performance.

Existem diversas APIs de software para a criação de threads. O programador vê as threads neste nível. Como já foi dito, cabe ao Sistema Operacional e ao próprio hardware escalonar o tempo de execução de cada thread.

3.1 Definição de Thread

Uma thread é uma seqüência única de instruções, executada paralelamente a outras seqüências de instruções. Todo programa de computador tem pelo menos uma thread, que é a thread principal ou “main”. Esta thread inicializa o programa e executa suas instruções. O programador pode então usar certas instruções para criar outras threads.

No nível de hardware, uma thread é um caminho de execução independente de outros caminhos de execução de hardware.

O modelo de threads é apresentado em três camadas. A primeira camada é a de threads a nível de usuário, que são as threads criadas e manipuladas pelo software. Abaixo, encontramos a camada de threads a nível de kernel, que são as threads utilizadas pelo Sistema Operacional. Por último está a camada de threads de hardware, que é como as threads são executadas pelos recursos de hardware. Uma thread em um programa normalmente envolve as três camadas: uma thread de software é convertida em uma thread do Sistema Operacional que por sua vez a envia para o hardware executar como uma thread de hardware.

3.2 Threads a Nível de Usuário

As threads a nível de usuário são aquelas criadas e destruídas diretamente pelo programador. É responsabilidade deste alocar recursos, aplicar mecanismos de sincronização e liberar os recursos utilizados por cada uma das threads.

Em aplicações nativas – isto é, aquelas que não executam dentro de um ambiente de execução como Java ou .NET – a criação de uma thread é feita através da chamada de uma função da API do Sistema Operacional. Quando o programa executa, esta função indica ao SO que ele deve criar uma thread a nível de kernel. Então as instruções contidas naquela thread são passadas ao processador para executá-las na thread a nível de hardware.

Em aplicações que executam dentro de um ambiente de execução, existe a camada de abstração da máquina virtual entre as chamadas de função do software e as chamadas oferecidas pelo Sistema Operacional. A máquina virtual não realiza o scheduling das threads criadas no código gerenciado. Em vez disso, as chamadas a funções de criação de threads são convertidas em chamadas à API do Sistema Operacional.

As threads são criadas pelo programador usando APIs como POSIX threads (Pthreads) e OpenMP. APIs como a Pthreads e a API do Windows para threads são APIs de baixo nível, em que o usuário explicitamente especifica a criação e a destruição de threads, dando ao programador mais controle sobre elas. Já o OpenMP abstrai a criação explícita de threads, facilitando seu gerenciamento pelo programador. Normalmente, no OpenMP são necessárias menos linhas de código para dividir o mesmo trabalho em múltiplas threads do que em APIs como Pthreads e Windows threads.

Mais tarde, estas APIs serão abordadas em maior detalhe.

3.3 Threads a Nível de Sistema Operacional

Os Sistemas Operacionais modernos estão divididos em duas camadas. Na camada de usuário, temos as bibliotecas de sistema, as funções que fazem parte da API do Sistema Operacional e as aplicações que são executadas. Na camada do kernel, acontecem as atividades do sistema e que são mantidas ocultas do usuário, como a gerência de memória, controle de operações de entrada e saída etc. (3).

As threads criadas por uma aplicação podem ser convertidas pelo Sistema Operacional em threads a nível de kernel ou threads a nível de usuário. A API Pthreads e o OpenMP trabalham diretamente com threads a nível de kernel. A API do Windows para threads suporta tanto threads a nível de kernel como threads a nível de usuário. Estas threads a nível de usuário do Windows são chamadas de fibras (4). As fibras devem ser manualmente agrupadas pelo programador para que ele especifique um mapeamento entre elas e as threads a nível de kernel que se encontram abaixo. Isto permite um controle maior mas ao mesmo tempo exige um esforço de programação muito maior, o que faz com que as fibras não sejam muito utilizadas.

O Sistema Operacional mantém um pool de threads de nível de kernel para reutilizá-las conforme terminam. Isto é feito para compensar o overhead que existe na criação e na destruição de threads. Este tipo de thread oferece uma performance melhor, e diferentes threads de um mesmo processo podem executar em diferentes cores ou processadores. Cada processo executado dentro do Sistema Operacional recebe um Process Control Block (PCB), que contém dados sobre a prioridade do processo, o espaço de memória reservado ao processo e o identificador do processo. As threads dentro de um processo compartilham o mesmo espaço de memória. O Sistema Operacional mantém uma tabela de todas as threads executando sem fazer distinção sobre que thread pertence a qual processo.

Em um computador com múltiplos processadores, o programador pode especificar que deseja que um determinado processo seja executado por um determinado processador. Este conceito, conhecido como afinidade de processador (2), toma vantagem do fato que alguns remanescentes do processo são mantidos no estado do processador, em particular a cache, desde a última vez que o processo executou, e assim escaloná-lo para executar no mesmo processador pode resultar em um tempo

de execução menor do que se fosse rodar em outro processador. O Sistema Operacional não garante que sempre respeitará a afinidade de processador. Sob certas circunstâncias, um processo pode ser escalonado para executar em outro processador caso o algoritmo detecte que o tempo de execução pode ser menor nestas circunstâncias. Um exemplo seria o caso em que um sistema com dois processadores tem dois processos que possuem afinidade com o mesmo processador. Ao invés de deixar um dos processadores inutilizado, o Sistema Operacional rompe a afinidade de processador e distribui cada processo a um processador diferente.

Um Sistema Operacional aplica um esquema de mapeamento entre processos que pode ser 1:1 ou N:1.

3.3.1 Mapeamento 1:1

No modelo 1:1, é responsabilidade do Sistema Operacional fazer o escalonamento das threads para os processadores. Este modelo também é conhecido como modelo preemptivo. Um Sistema Operacional que implementa este modelo pode realizar uma troca de contexto quando julgar necessário, garantindo assim que cada thread execute por um período, ou quanta, razoável. Também permite ao sistema perceber rapidamente eventos externos como operações de entrada e saída.

Em um determinado momento de execução, as threads podem ser separadas em duas categorias: a daquelas que esperam por operações de entrada e saída (“I/O bound”) e as que estão utilizando a CPU (“CPU bound”). Em sistemas operacionais antigos, os processos ficavam em um estado de “busywait” quando esperavam por operações de entrada e saída, ou seja, mantinham controle da CPU conferindo seguidamente se a operação de I/O desejada tinha sido satisfeita (pressionamento de uma tecla, leitura de dados do disco rígido etc.). Com o advento do modelo preemptivo, esses processos I/O bound puderam ser bloqueados, aguardando a chegada dos dados necessários enquanto outro processo assume a CPU. A chegada dos eventos de entrada e saída gera uma interrupção, que permite ao processo bloqueado retornar à execução.

Alguns dos sistemas operacionais mais usados hoje utilizam o mapeamento 1:1, como Linux, Windows XP e Windows 2000 (6).

3.3.2 Mapeamento N:1

No modelo N:1, é responsabilidade de cada thread executando ceder de forma voluntária o controle da CPU para outras threads através de chamadas de função no software. Este modelo também é conhecido como modelo cooperativo.

Por colocar nas mãos do programador a responsabilidade de ceder o controle da CPU, o modelo cooperativo é considerado inferior ao modelo preemptivo. Uma aplicação mal concebida ou mal intencionada pode causar a instabilidade do sistema a ponto de pará-lo completamente.

Alguns sistemas operacionais antigos utilizavam o mapeamento N:1, como Windows 3.1 e as versões do Mac OS anteriores ao Mac OS X.

3.4 Threads a Nível de Hardware

O hardware executa as instruções das camadas de software. As instruções das threads de software da aplicação passam pelo Sistema Operacional e são encaminhadas para os recursos de hardware.

Para executar múltiplas threads simultaneamente em hardware, era exigido que um computador tivesse múltiplos processadores. Cada thread executava em um processador separado. Com o advento da tecnologia HT, tornou-se possível executar duas ou mais threads em um mesmo processador ao mesmo tempo. Como dito anteriormente, este processador é dotado da duplicação de certos componentes, permitindo o processamento concorrente de múltiplas threads.

Processadores multi-core providenciam dois ou mais cores de execução, permitindo a execução paralela de múltiplas threads em hardware. O número de threads de hardware que podem ser executadas ao mesmo tempo deve ser considerado na hora de construir um software: para estabelecer um verdadeiro paralelismo, o número de threads de software deveria ser igual ao número de threads de hardware. Porém, isto dificilmente se verifica, visto que o número de threads de software costuma ser maior que o número de threads de hardware disponíveis. Um número muito grande de threads de software pode diminuir a performance do programa. É preciso manter um bom balanceamento entre threads de software e hardware para atingir boa performance.

3.5 Criação de Threads

Como já foi dito, um processo pode apresentar mais de uma thread, e cada uma destas threads compartilha do mesmo espaço de endereçamento, apesar de operarem de forma independente.

Cada thread possui seu próprio espaço na pilha. Gerenciar este espaço na pilha é tarefa do Sistema Operacional. Quando uma thread é criada, o Sistema Operacional aloca para ela um espaço na pilha. O tamanho padrão deste espaço varia entre diferentes sistemas operacionais. Em certas ocasiões, é possível que o tamanho padrão do espaço na pilha reservado para uma thread seja pequeno demais para certa aplicação, o que leva o programador a ter que gerenciar o espaço na pilha manualmente. Porém na maioria dos casos isto não é necessário, e apenas desenvolvedores de sistemas operacionais precisam se preocupar com isto.

Criar e destruir threads implica em alocar e desalocar espaço na pilha, respectivamente. Este é um dos motivos pelos quais o Sistema Operacional fornece um *pool* de threads. Assim, diminui o overhead de criação e destruição das threads.

Após sua criação, uma thread pode ocupar um destes quatro estados diferentes (6):

- Pronto
- Executando
- Bloqueado
- Terminado

3.6 Threads POSIX

Como foi dito anteriormente, o programador emprega APIs conhecidas para criar threads em software. O próximo capítulo abordará o OpenMP, uma API portátil criada através de um esforço conjunto de vários órgãos e empresas para facilitar a criação de programas multi-threaded.

Outras APIs para a criação de programas multi-threaded já existem. Para melhor compreender a motivação que levou ao surgimento do OpenMP, é necessário estudar os pontos fortes e pontos fracos destas APIs. Aqui será falado brevemente sobre a API POSIX Threads, ou simplesmente Pthreads.

A biblioteca Pthreads é uma biblioteca portátil de threads criada para providenciar uma interface de programação comum para diferentes sistemas operacionais. Pthreads é a interface de criação de threads padrão no Linux e também usada freqüentemente na maioria das plataformas Unix. Existe uma versão de código aberto disponível também para Windows (11).

O foco das funções padrão da biblioteca Pthreads está na criação, destruição e sincronização de threads. Outras funções como prioridades de threads não fazem parte da API Pthreads padrão, mas podem ser disponibilizadas por implementações específicas da biblioteca.

3.6.1 Criação de Threads com a Biblioteca Pthreads

A criação de threads na biblioteca Pthreads é feita através da função `pthread_create`:

```
pthread_create(&id_da_thread, atributos, nome_da_funcao, parametros);
```

O primeiro argumento é uma variável do tipo `pthread_t` à qual é atribuído o identificador da thread. O segundo argumento especifica os atributos da thread, de forma que usar `NULL` identifica que não há atributos. O terceiro argumento é o nome da função que será executada ao iniciar a thread. A thread executa a função e depois encerra. O quarto argumento é usado para passar parâmetros para a função do terceiro argumento.

O programa a seguir é um exemplo de como usar a função `pthread_create` para criar um programa com diversas threads:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define TOTAL_THREADS 2

void *HelloWorld(void * thread_num)
{
    int id_thread = *((int*) thread_num);
    printf("Ola mundo! Aqui e a thread %d\n", id_thread);
}

int main()
{
    int i;
    int retval;
    pthread_t threads[TOTAL_THREADS];

    for (i = 0; i < TOTAL_THREADS; i++)
    {
```

```

    retval = pthread_create(&threads[i],
                           NULL,
                           HelloWorld,
                           (void*) &i);

    if (retval != 0)
        fprintf(stderr,
            "Ocorreu um erro ao lancar a thread %d\n", i);
}

printf("Fim da thread principal\n");

return 0;
}

```

O resultado da execução é:

```

Ola mundo! Aqui e a thread 0
Ola mundo! Aqui e a thread 1
Fim da thread principal

```

A função `HelloWorld` é passada como um argumento para a função `pthread_create` na `main`. Cada iteração do laço `for` faz a thread principal disparar uma nova thread que executa o código da função `HelloWorld`.

3.6.2 Gerência de Múltiplas Threads

A biblioteca Pthreads um modelo de programação similar ao *fork/join* dos sistemas Unix (5). Os programas iniciam a execução de forma sequencial, executando até encontrar uma região paralela. A thread única que existia separa-se (*fork*) em um grupo de múltiplas threads executando simultaneamente. Quando as threads completam a execução do trecho paralelo, elas se sincronizam e terminam (*join*), deixando novamente uma única thread executando.

A operação *fork* é realizada pela biblioteca Pthreads através da criação de uma thread. A operação *join* é realizada pela função `pthread_join`:

```
pthread_join(id_da_thread, retval);
```

onde `id_da_thread` é uma variável do tipo `pthread_t` que representa o identificador da thread que deve ser esperada. Ao executar esta função, o Sistema Operacional faz com que a thread atual fique bloqueada até que a thread `id_da_thread` termine sua execução. O parâmetro `retval` é um buffer do tipo `void**` onde é colocado o valor de retorno da função.

O exemplo anterior foi modificado de forma que a função `HelloWorld`, após exibir seu cumprimento, dorme por dois segundos e então imprime uma mensagem indicando que terminou sua execução.

```

void *HelloWorld(void * thread_num)
{
    int id_thread = *((int*) thread_num);
    printf("Ola mundo! Aqui é a thread %d\n", id_thread);
}

```

```
sleep(2);
printf("Fim da thread %d\n", id_thread);
}
```

Ao executar o programa será produzida a seguinte saída:

```
Ola mundo! Aqui é a thread 0
Ola mundo! Aqui é a thread 1
Fim da thread principal
```

A função HelloWorld não executa até o final. Em algum momento durante os dois segundos em que as threads disparadas pela função main estão dormindo, o controle da CPU é passado de volta à thread principal. Como não havia sido especificado que a thread principal deveria esperar pelas outras threads, ela continua sua execução e encontra o seu término, encerrando assim o programa.

Este problema é solucionado ao especificar à thread principal que ela deve esperar pelas outras threads para terminar. Isto é feito com a função `pthread_join`. O seguinte trecho de código deve ser inserido antes de terminar a função main:

```
for(i = 0; i < TOTAL_THREADS; i++)
    pthread_join(threads[i], NULL);
```

Ao executar o programa sera produzida a seguinte saída:

```
Ola mundo! Aqui é a thread 0
Ola mundo! Aqui é a thread 1
Fim da thread 0
Fim da thread 1
Fim da thread principal
```

Como se pode ver, desta vez a thread principal esperou corretamente pelo fim das outras threads para encerrar.

3.6.3 Mecanismos de Sincronização

Criar uma thread é uma tarefa relativamente simples, que não exige a maior parte do tempo necessária para se criar uma aplicação multi-threaded. O verdadeiro desafio nestes casos é garantir que em um ambiente do mundo real as threads criadas pelo desenvolvedor ajam de uma maneira pré-determinada, de forma a cumprir seu objetivo, sem que entrem em condições como deadlocks ou corrupção de dados causados por condições de corrida (5).

Antes de partir para os mecanismos fornecidos pela biblioteca Pthreads para garantir a sincronização de threads, é necessário rever alguns conceitos usados para sincronizar acesso concorrente a recursos compartilhados.

Uma *seção crítica* é um bloco de código de que só pode ser acessado por um determinado número de threads simultaneamente.

Um *semáforo* é uma estrutura de dados que limita o acesso de uma seção crítica a um determinado número de threads.

Um *mutex* é um tipo especial de semáforo que permite acesso exclusivo de uma seção crítica a uma única thread.

Em seguida será exemplificado o uso de um mutex na biblioteca Pthreads. A biblioteca emprega os mutexes através das funções `pthread_mutex_lock` e `pthread_mutex_unlock`. As duas funções recebem uma variável do tipo `pthread_mutex_t` como argumento e bloqueiam e desbloqueiam o mutex, respectivamente.

O mutex é útil para resolver problemas clássicos de programação paralela como o do produtor e consumidor. No código abaixo, o único array A deve ser acessado de forma concorrente, porém no máximo uma thread pode ter acesso ao array ao mesmo tempo.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define MAX_SIZE 100

int A[MAX_SIZE];
int pointer = 0;

pthread_mutex_t meu_mutex =
    PTHREAD_MUTEX_INITIALIZER;

int consumir()
{
    int temp = -1;
    pthread_mutex_lock(&meu_mutex);

    if (pointer > 0)
    {
        --pointer;
        temp=A[pointer];
    }

    pthread_mutex_unlock(&meu_mutex);

    return temp;
}

void inserir(int valor)
{
    pthread_mutex_lock(&meu_mutex);

    if (pointer < MAX_SIZE)
    {
        A[pointer] = valor;
        pointer++;
    }

    pthread_mutex_unlock(&meu_mutex);
}
```

A macro `PTHREAD_MUTEX_INITIALIZER` inicializa o tipo de dados `pthread_mutex_t`. Grande parte das vezes, para criar um mutex corretamente basta usar esta macro definida.

3.6.4 Desvantagens da Biblioteca Pthreads

A biblioteca Pthreads não oferece nenhuma função que indique o número de processadores existentes em uma máquina. Isto prejudica a escalabilidade de uma aplicação. Suponha um código que realizasse um cálculo com muitas iterações independentes umas das outras. Se fosse sabido quantos processadores uma máquina disponibiliza, poderiam ser criadas tantas threads quanto fosse o número de processadores, e distribuir as iterações do cálculo uniformemente entre os processadores. A ausência de uma função *portável* que indique o número de processadores – uma determinada implementação de Pthreads poderia apresentar esta função, apesar dela não fazer parte do padrão Pthreads – prejudica a escalabilidade da biblioteca como um todo.

Além disso, de maneira geral, é exigido do programador um esforço muito grande para realizar tarefas simples. Por exemplo, na seção 3.6.2, na página 26, foi necessário expressar explicitamente que era desejado que o programa aguardasse o fim das threads antes de terminar, e para isso foi necessário todo o esforço de programação do join explícito. No próximo capítulo, será visto como o OpenMP permite ao programador criar e gerenciar threads de maneira muito menos trabalhosa.

4 OPENMP

Para quem desenvolve software, seria interessante desfrutar dos ganhos de performance oferecidos por processadores dual-core ou multi-core. No futuro, é provável que os computadores venham a oferecer um número cada maior de cores. Dependendo do jeito que for desenvolvida, uma aplicação com objetivo de executar em um processador dual-core hoje pode não ter muito ganho de performance ao executar em um processador quad-core no próximo ano.

Uma solução para suprir esta necessidade é o OpenMP (7). Definido por um consórcio de fabricantes de hardware e software, juntamente com universidades e membros do governo norte-americano, o OpenMP é um modelo de programação para máquinas multi-threaded com memória distribuída. O uso do OpenMP torna explícito o paralelismo existente em programas.

Projetada para ser uma API escalável e portátil, o OpenMP adiciona uma camada de abstração acima do nível das threads, deixando o programador livre da tarefa de criar, gerenciar e destruir threads.

O padrão OpenMP é definido em Fortran, C e C++, com implementações disponíveis para Windows e Unix. Este trabalho tratará exclusivamente do OpenMP na linguagem C/C++, executado em sistemas Windows e Linux, conforme a disponibilidade das máquinas.

Utilizando as diretivas *#pragma* disponíveis na linguagem, o programador explicita os locais do programa que serão paralelizados. A etapa de pré-compilação do programa converte as diretivas em chamadas para threads comuns, como POSIX Threads por exemplo. O programador não precisa usar as APIs de threads diretamente, isto é feito pelo OpenMP. Isto traz portabilidade aos sistemas desenvolvidos, pois mesmo as APIs de threads mais abrangentes costumam apresentar diferenças em arquiteturas diferentes.

O OpenMP apresenta ainda a possibilidade de determinar junto ao sistema o número de cores ou processadores disponíveis. Assim, um mesmo código em OpenMP pode ser aproveitado em um sistema com um, dois, quatro cores ou mais, sem precisar ser reescrito. Cabe ao compilador adaptar o código existente para criar o maior número de threads possível.

Entre as empresas participantes do consórcio que criou o OpenMP (12) incluem-se a Intel, HP, SGI, IBM, Sun, Compaq, KAI, PGI, PSR, APR, Absoft, ANSYS, Fluent, Oxford Molecular, NAG, DOE, ASCI, Dash e Livermore Software.

4.1 Disponibilidade do OpenMP em Compiladores

Como participante do grupo de empresas por trás do OpenMP, a Intel disponibiliza no ICC, seu compilador para C/C++, amplo suporte ao OpenMP nas arquiteturas de 32 bits e 64 bits (13). O ICC está disponível em versão trial de 30 dias para Windows, e é gratuito para uso não comercial no Linux.

O compilador C++ do Microsoft Visual Studio 2005 traz suporte ao OpenMP (14). O compilador C++ está disponível gratuitamente para Windows, enquanto que o Visual Studio é um produto pago, também disponível somente para Windows.

Existe um projeto open-source chamado GOMP (Gnu OpenMP), de autoria da Gnu, criado com o intuito de oferecer suporte do OpenMP ao compilador GCC (15). A partir da versão 4.2, este suporte deverá ser fornecido junto com o compilador.

4.2 Programação com OpenMP

No capítulo anterior foi abordada a programação com a API Pthreads. Os exemplos apresentados neste capítulo demonstrarão a facilidade de programação do OpenMP frente à Pthreads. Facilidade de programação do OpenMP traduz-se em menor esforço de programação para as empresas, menos horas de treinamento e menos defeitos. Tudo isso leva um produto com preço final mais barato ao usuário final do software.

Assim com a biblioteca Pthreads, O OpenMP utiliza um modelo de programação similar ao *fork/join* dos sistemas Unix (5).

Para usar o OpenMP com C/C++, basta incluir o arquivo `omp.h` e usar diretivas de compilação `#pragma` e chamadas à API do OpenMP. O número de threads de uma região seqüencial é determinado pelo número de processadores ou cores existentes, a menos que especificado o contrário.

Os seguintes exemplos foram compilados usando-se o compilador ICL da Intel na versão 9.1, em sistemas Linux. Abaixo segue um programa de exemplo que exibe “Olá Mundo” na tela.

```
#include <stdio.h>
#include <omp.h>

int main()
{
#pragma omp parallel
{
    printf("Ola Mundo!\n");
}

    return 0;
}
```

Ao executar o código em um sistema single-core, o resultado obtido é o seguinte:

Ola Mundo!

Ao executar em um sistema dual-core, o OpenMP corretamente distribui as threads para os múltiplos processadores:

```
Ola Mundo!
Ola Mundo!
```

Quando o programador quiser especificar o número de threads, ele pode fazer isto chamando a função `omp_set_num_threads`. Esta chamada tem prioridade sobre a determinação do sistema em relação ao número de cores disponíveis, podendo assim criar um número de threads maior que o número de cores.

Para fazer cada thread imprimir na tela sua identificação, é usada a função `omp_get_thread_num`, que retorna o ID da thread.

```
int id;
omp_set_num_threads(2);
#pragma omp parallel private(id)
{
    id = omp_get_thread_num();
    printf("Ola Mundo! Aqui é a thread %d\n", id);
}
```

A diretiva `private` acompanhada da variável `id` significa que cada thread mantém uma cópia local desta variável. O valor desta variável não é mantido fora da região paralela. O resultado da execução é:

```
Ola Mundo! Aqui é a thread 0
Ola Mundo! Aqui é a thread 1
```

4.3 Exemplo Completo: Cálculo de Pi

O seguinte exemplo completo exercitará algumas diretivas do OpenMP. O OpenMP pode ser usado para cálculos matemáticos complexos. Um exemplo pode ser dado ao calcular o valor de π com a seguinte fórmula:

$$\pi = 4 - 4/3 + 4/5 - 4/7 + 4/9 - 4/11 + \dots \quad (4.1)$$

O código seqüencial abaixo realiza o cálculo de π :

```
#include <stdio.h>

#define num_passos 20000000

int main()
{
    double pi = 0;
    int i;

    for (i = 0; i < num_passos; i++)
    {
        pi += 4.0 / (4.0 * i + 1.0);
        pi -= 4.0 / (4.0 * i + 3.0);
    }
}
```

```
    printf("O valor de pi é %f\n", pi);
}
```

Suponha agora que está disponível um sistema multi-core. É fácil perceber que o trecho mais trivial para paralelização é o loop for, pois cada soma poderia ser realizada de forma paralela. Para isto, basta inserir a diretiva `omp parallel for`, e o trecho de código ficaria assim:

```
#pragma omp parallel for
for (i = 0; i < num_passos; i++)
{
    pi += 4.0 / (4.0 * i + 1.0);
    pi -= 4.0 / (4.0 * i + 3.0);
}
```

Porém, dois problemas surgem da solução apresentada. Em um sistema single-core, um compilador razoavelmente inteligente colocaria a variável `pi` em um registrador, sem precisar escrever e ler da memória. Em um sistema multi-core, os registradores não são compartilhados. Para o código funcionar corretamente, seria necessário escrever e ler o valor da variável `pi` na memória a cada iteração. Considerando que o tempo de entrada e saída de dados para a memória é dramaticamente maior que o tempo de processamento, o programa com certeza executaria mais lentamente do que o original (1).

O outro problema ocorre devido à competição que os diversos cores ou processadores estabelecem em relação à variável `pi`. Em um sistema dual-core, por exemplo, poderia ocorrer a seguinte seqüência de execução:

- O core 1 lê a variável `pi` da memória, que está configurada como 0.
- O core 2 lê a variável `pi` da memória, que está configurada como 0.
- O core 1 calcula `pi` com relação ao índice `i=0`, e a variável `pi` agora possui valor 2,667.
- O core 2 calcula `pi` com relação ao índice `i=1`, e a variável `pi` agora possui valor 0,229.
- O core 1 escreve `pi` na memória, com o valor 2,667.
- O core 2 escreve `pi` na memória, com o valor 0,229.

Como se pode ver, a próxima etapa de execução continuará com o valor 0,229 na variável `pi`, e não 2,896, que seria o valor correto. Para resolver estes dois problemas, usamos a diretiva `reduction` do OpenMP. Aplicamos a seguinte modificação no código:

```
#pragma omp parallel for reduction (+:pi)
for (i = 0; i < num_passos; i++)
{
    pi += 4.0 / (4.0 * i + 1.0);
    pi -= 4.0 / (4.0 * i + 3.0);
}
```

Isto faz com que cada thread possua o seu valor próprio para a variável π , e a operação $+$ é aplicada sobre o valor de todas elas ao final do código paralelo. É uma solução simples, eficaz e que não insere bugs no código existente.

4.4 Diretivas para trabalhar com variáveis

Ao trabalhar em um ambiente multi-threaded, é importante para o programador contar com mecanismos de sincronização. Como visto, condições de corrida podem prejudicar a corretude de um programa, levando a resultados incorretos devido ao paralelismo. Além da diretiva `reduction`, o OpenMP disponibiliza várias alternativas para evitar este tipo de erro.

Normalmente, as variáveis em OpenMP são compartilhadas. Em C/C++, as variáveis que estão no escopo onde é iniciado o bloco do código paralelo são compartilhadas. Porém, isso pode causar problemas. O código abaixo é usado para multiplicar matrizes em um programa C/C++ com OpenMP.

```
#define SIZE 2

int i, j, k;
int A[2][2] = { {1, 2}, {3, 4} };
int B[2][2] = { {5, 6}, {7, 8} };
int C[2][2];

#pragma omp parallel for
for (i = 0; i < SIZE; i++)
{
    for (j = 0; j < SIZE; j++)
    {
        int soma = 0;
        for (k = 0; k < SIZE; k++)
        {
            soma += A[i][k] * B[k][j];
        }
        C[i][j] = soma;
    }
}
```

Ao encontrar a diretiva `parallel for`, o OpenMP compartilha todas as variáveis que já estão naquele escopo. Seria normal compartilhar as variáveis A, B e C contendo os valores das matrizes. Porém, os índices dos loops `for` não podem ser compartilhados. O resultado disto seria desastroso. Uma thread interferindo no índice do loop de outra thread causa um problema grave de sincronização.

Para evitar que isso ocorra usa-se a diretiva `private`. As variáveis marcadas como `private` não são compartilhadas entre threads. A diretiva `shared` também existe e serve para explicitar que uma variável deve ser compartilhada entre as threads.

```
#pragma omp parallel for private(i,j,k) shared(A,B,C)
for (i = 0; i < SIZE; i++)
{
```

```

for (j = 0; j < SIZE; j++)
{
    int soma = 0;
    for (k = 0; k < SIZE; k++)
    {
        soma += A[i][k] * B[k][j];
    }
    C[i][j] = soma;
}
}

```

Note que a variável `soma` não é marcada como `private`. Como ela é declarada dentro de um bloco de código paralelo, ela já é privativa para cada thread. Caso esta variável fosse estática (palavra-chave `static` de C), ela não seria privativa para cada thread.

Além disso, o valor de uma variável marcada como `private` não é mantido após o código paralelo. É como se fosse outra variável, que nem ao menos recebe qualquer valor atribuído antes do código paralelo. Ao executar o seguinte código:

```

int x = 3;

#pragma omp parallel private (x)
{
    printf("No começo da thread valor de x é %d\n", x);
    x = 5;
    printf("No fim da thread valor de x é %d\n", x);
}

printf ("O valor final de x é %d\n", x);

```

temos como resultado:

```

No começo da thread valor de x é 0
No fim da thread valor de x é 5
O valor final de x é 3

```

Se for desejado que a variável privativa mantenha o valor que existia antes de iniciar o código paralelo, pode-se usar a diretiva `firstprivate`:

```

int x = 3;

#pragma omp parallel firstprivate (x)
{
    printf("No começo da thread valor de x é %d\n", x);
    x = 5;
    printf("No fim da thread valor de x é %d\n", x);
}

printf ("O valor final de x é %d\n", x);

```

Resultado:

No começo da thread valor de x é 3

No fim da thread valor de x é 5

0 valor final de x é 3

A diretiva `lastprivate`, usada apenas com `parallel for`, especifica que o valor da variável deve ser igual ao valor atribuído a ela durante a última iteração do `for`. Exemplo:

```
int x;

#pragma omp parallel for lastprivate (x)
for (x = 0; x < 10; x++)
{
    // do stuff...
}

printf ("0 valor final de x é %d\n", x);
```

Resultado:

0 valor final de x é 10

Por padrão, as variáveis no OpenMP são `shared`. Na API OpenMP para C/C++, podemos alterar este padrão através da cláusula `default(none)`, que obriga o programador a especificar se quer que as variáveis sejam `shared`, `private` ou outro atributo. Por exemplo, no programa de multiplicação de matrizes, ao adicionar a cláusula `default(none)`:

```
#pragma omp parallel for default(none) private(i, j, k)
for (i = 0; i < SIZE; i++)
{
    for (j = 0; j < SIZE; j++)
    {
        int soma = 0;
        for (k = 0; k < SIZE; k++)
        {
            soma += A[i][k] * B[k][j];
        }
        C[i][j] = soma;
    }
}
```

É recebido um erro de compilação por não especificar o compartilhamento das variáveis A, B e C (os erros abaixo são listados pelo compilador ICL da Intel com OpenMP em ambiente Linux):

```
multi_matrix.c(14): error: "A" must be specified in a variable
list at enclosing OpenMP parallel pragma
```

```
#pragma omp parallel for default(none) private(i, j, k)
^
```

```
multi_matrix.c(14): error: "B" must be specified in a variable
list at enclosing OpenMP parallel pragma
```

```
#pragma omp parallel for default(none) private(i, j, k)
^
```

```
multi_matrix.c(14): error: "C" must be specified in a variable
list at enclosing OpenMP parallel pragma
```

```
#pragma omp parallel for default(none) private(i, j, k)
^
```

Quando compilado no GCC versão 4.2, são produzidos os seguintes erros:

```
private.c: In function 'main':
private.c:22: error: 'A' not specified in enclosing parallel
private.c:14: error: enclosing parallel
private.c:22: error: 'B' not specified in enclosing parallel
private.c:14: error: enclosing parallel
private.c:22: error: 'C' not specified in enclosing parallel
private.c:14: error: enclosing parallel
```

4.5 Diretivas de Distribuição de Tarefas

Por padrão, em um programa OpenMP, o número de threads que executa determinada região paralela de código é decidido dinamicamente. Configurar o número de threads com a chamada da função `omp_set_num_threads` indica o número máximo de threads que serão executadas, mas podem existir menos.

Se desejar, o programador pode usar diretivas específicas para controlar diretamente esta distribuição de tarefas entre as threads.

A diretiva `sections` distribui blocos de código para diferentes threads. O exemplo abaixo faz com que uma thread execute o código da função `executaA()`, outra thread a função `executaB()` e uma terceira thread a função `executaC()`:

```
#pragma omp parallel
#pragma omp sections
{
#pragma omp section
    executaA();
#pragma omp section
    executaB();
#pragma omp section
    executaC();
}
```

A diretiva `barrier` é uma diretiva de sincronização que exige que todas as threads cheguem a um determinado ponto da execução para continuar. O código abaixo:

```
int th_id;

omp_set_num_threads(3);

#pragma omp parallel private(th_id)
{
    th_id = omp_get_thread_num();
    printf("Executando thread %d\n", th_id);
#pragma omp barrier
    printf("Continuando thread %d\n", th_id);
}
```

Produz como resultado:

```
Executando thread 0
Executando thread 1
Executando thread 2
Continuando thread 0
Continuando thread 2
Continuando thread 1
```

A diretiva `master` faz apenas a thread principal executar um trecho de código. As outras threads ignoram este trecho:

```
int th_id;

omp_set_num_threads(3);

#pragma omp parallel private(th_id)
{
    th_id = omp_get_thread_num();
    printf("Iniciando thread %d\n", th_id);
#pragma omp master
    {
        printf("Apenas thread %d aqui\n", th_id);
    }
    printf("Terminando thread %d\n", th_id);
}
```

Resultado:

```
Iniciando thread numero 0
Apenas thread 0 aqui
Iniciando thread numero 2
Iniciando thread numero 1
Terminando thread numero 1
Terminando thread numero 2
Terminando thread numero 0
```

Marcar um bloco de código com a diretiva `single` faz com que apenas uma thread execute aquele código. Ao final do trecho de código, implicitamente existe uma diretiva `barrier` que exige que todas as outras threads cheguem àquele ponto para continuar:

```
int th_id;

omp_set_num_threads(3);

#pragma omp parallel private(th_id)
{
    th_id = omp_get_thread_num();
    printf("Iniciando thread %d\n", th_id);
#pragma omp single
    {
        printf("Apenas thread %d aqui\n", th_id);
    }
    printf("Terminando thread %d\n", th_id);
}
```

Resultado:

```
Iniciando thread 0
Apenas thread 0 aqui
Iniciando thread 1
Iniciando thread 2
Terminando thread 0
Terminando thread 2
Terminando thread 1
```

A diretiva `critical section` determina uma seção crítica do código, onde somente uma thread pode executar aquele código de cada vez. O problema do produtor e consumidor é resolvido em OpenMP com esta diretiva.

```
#define MAX_SIZE 100

int A[MAX_SIZE];
int pointer = 0;

int consumir()
{
    int temp = -1;
#pragma omp critical
    {
        if (pointer > 0)
        {
            --pointer;
            temp = A[pointer];
        }
    }
}
```

```

}
return temp;
}

void inserir(int valor)
{
#pragma omp critical
{
    if (pointer < MAX_SIZE)
    {
        A[pointer] = valor;
        pointer++;
    }
}
}

```

Este código pode ser comparado com o código do mesmo problema resolvido com a biblioteca Pthreads, na página 28, seção 3.6.3. O uso das funções `pthread_mutex_lock` e `pthread_mutex_unlock` é propenso a erros, pois esquecer de desbloquear um mutex após passar pela seção crítica é um erro ao qual qualquer programador está sujeito. Com o OpenMP, o programador especifica um bloco entre chaves. Caso esquecesse de fechar a diretiva `critical`, o código não compilaria. Este tipo de construção facilita a programação com OpenMP e é uma de suas vantagens em relação ao uso da Pthreads.

Alternativamente a `critical`, pode ser usada a diretiva `atomic`, que funciona como uma dica para o compilador usar instruções de hardware para obter a exclusão mútua. O compilador pode escolher por ignorar esta diretiva e compilar como se fosse uma seção crítica comum.

O OpenMP permite ao programador definir como será feita a distribuição de trabalho em um loop `for` através da diretiva `schedule`. Por padrão, as execuções de um loop `for` são divididas entre as threads do sistema de forma igual. Por exemplo, o código abaixo executado em uma máquina dual core teria uma thread executando as iterações de 0 a 49 e outra thread executando as iterações de 50 a 99:

```

#pragma omp parallel for
for (i = 0; i < 100; i++)
    doStuff(i);

```

O programador é livre para distribuir as iterações de outra maneira. Seria possível, por exemplo, que as iterações de 0 a 49 levassem mais tempo para executar e assim a segunda thread seria “desperdiçada” após terminar a sua carga de trabalho. A distribuição desta carga pode ser feita de forma estática, ou seja, é conhecido em tempo de compilação quais iterações ficarão a carga de quais threads, ou de forma dinâmica, quando isto é determinado em tempo de execução.

Por padrão, o OpenMP usa a distribuição estática, que o programador pode explicitar com a diretiva `schedule static`. Adicionalmente, o programador pode dizer qual o número de iterações que cada thread deve executar de forma consecutiva. Por exemplo, no loop abaixo de 1000 iterações, a distribuição `static` com o parâmetro 10 diz ao OpenMP que as primeiras 10 iterações devem ser de uma thread, as

10 iterações seguintes de outra thread, e assim por diante até completar todas as iterações.

```
#pragma omp parallel for schedule (static, 10)
for (m = 0; m < 1000; m++)
    doStuff(m);
```

Se nada for especificado, o número de iterações é dividido de forma igual entre as threads.

Também é possível usar a diretiva `schedule dynamic`, que diz ao OpenMP atribuir as iterações às threads conforme as threads se tornam disponíveis. Aparentemente, esta seria a solução ideal para evitar que uma thread ficasse sem trabalho. Porém, é necessário um overhead a mais de troca de contexto, que pode fazer com que a aplicação torne-se mais lenta. Aqui também o programador pode especificar a quantidade de iterações que deseja executar de cada vez.

As construções do OpenMP mostradas neste capítulo mostram como construir um programa paralelo com o OpenMP. No próximo capítulo, será analisado o desempenho do OpenMP utilizando alguns dos programas exemplificados neste capítulo, além de outros.

5 AVALIAÇÃO DE DESEMPENHO DO OPENMP

As aplicações criadas em OpenMP são independentes de plataforma. Porém, esta portabilidade esconde muitos detalhes ao programador. As diretivas do OpenMP fornecem uma abstração que escondem os detalhes da criação de uma aplicação multi-threaded. Como resultado, o compilador e o ambiente de execução afetam dramaticamente a performance de uma aplicação OpenMP.

Para exemplificar a influência que o compilador exerce sobre a performance de uma aplicação OpenMP, foi criada uma série de benchmarks diferentes executando sobre uma máquina com dois processadores Pentium III de 1266 MHz de clock e 512 KB de cache L2 cada um. Os testes foram executados com dois compiladores diferentes, o compilador da Intel para C/C++ na versão 9.1 e o compilador GCC.

Algumas considerações devem ser feitas sobre o compilador GCC: como foi dito, a implementação do OpenMP no GCC é objetivo do projeto open-source GOMP. Como um projeto em andamento, não é garantido que todas as características do OpenMP estejam implementadas. A versão do GCC usada nestes testes é experimental, visto que a versão 4.2 não havia sido lançada oficialmente até a conclusão deste trabalho. O projeto GOMP disponibiliza a versão mais recente do GCC como somente-leitura em um repositório controlado com a ferramenta Subversion de controle de versão. A versão utilizada aqui é identificada como 4.2.0 20061008.

Alguns programas em OpenMP listados em capítulos anteriores deste trabalho foram empregados como benchmarks para analisar a sua performance. Além disso, benchmarks da NASA Advanced Supercomputing Division foram compilados e executados com os dois compiladores acima citados.

5.1 Benchmarks Criados

O código para cálculo de π foi modificado para executar com o número de passos iniciando em 10^8 e incrementado em 10^8 a cada iteração até 10^9 . Foram criadas duas versões: uma com a diretiva `omp parallel for` do OpenMP e uma sem a diretiva OpenMP. Cada programa foi executado 10 vezes com cada compilador para que fosse obtida uma média de performance. Os resultados obtidos estão ilustrados na figura 5.1 na página 43.

Como se pode ver, o tempo de execução é reduzido pela metade com a diretiva `parallel for` do OpenMP. Além disso, o tempo de execução é menor com o compilador da Intel do que com o compilador GCC, exemplificando a forte influência do compilador no desempenho do OpenMP.

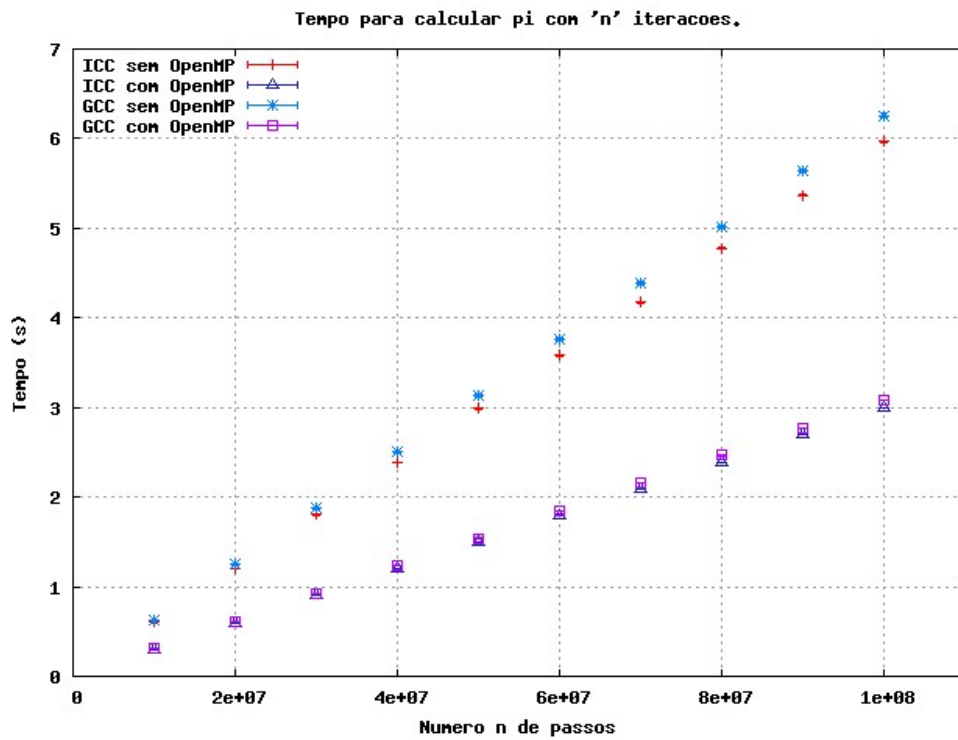


Figura 5.1: Desempenho do OpenMP no cálculo de π

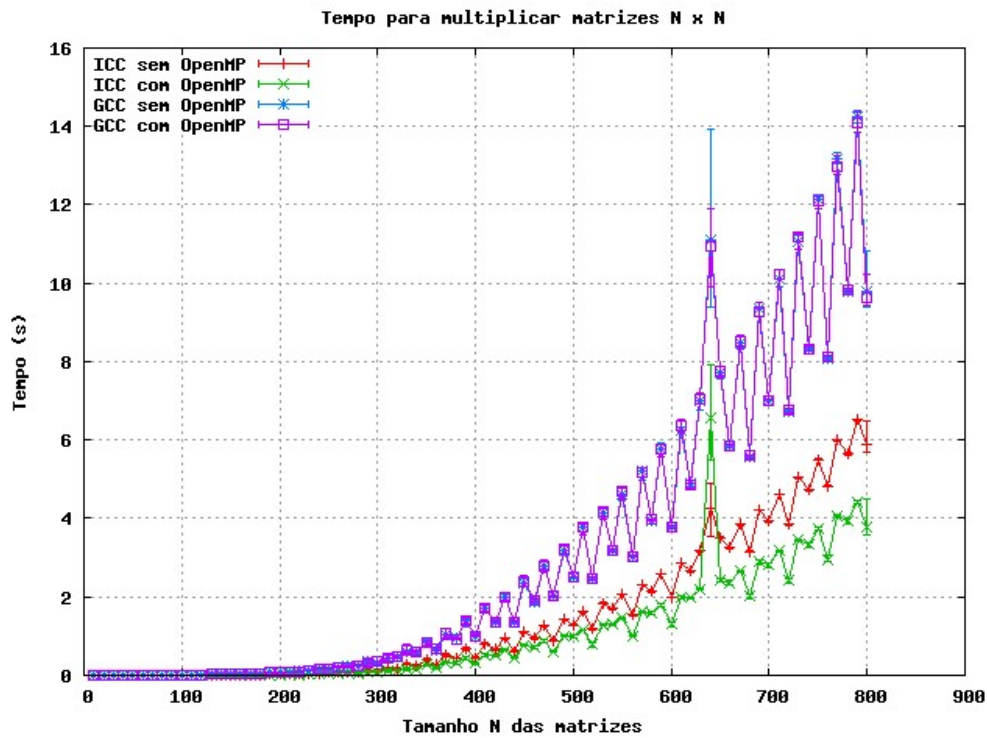


Figura 5.2: Desempenho do OpenMP na multiplicação de matrizes quadradas

Em seguida, foi exercitado o código de multiplicação de matrizes quadradas, com o tamanho das matrizes variando de 2×2 a 800×800 . Novamente, o código foi testado com e sem as diretivas OpenMP em ambos os compiladores. Os resultados podem ser vistos na figura 5.2.

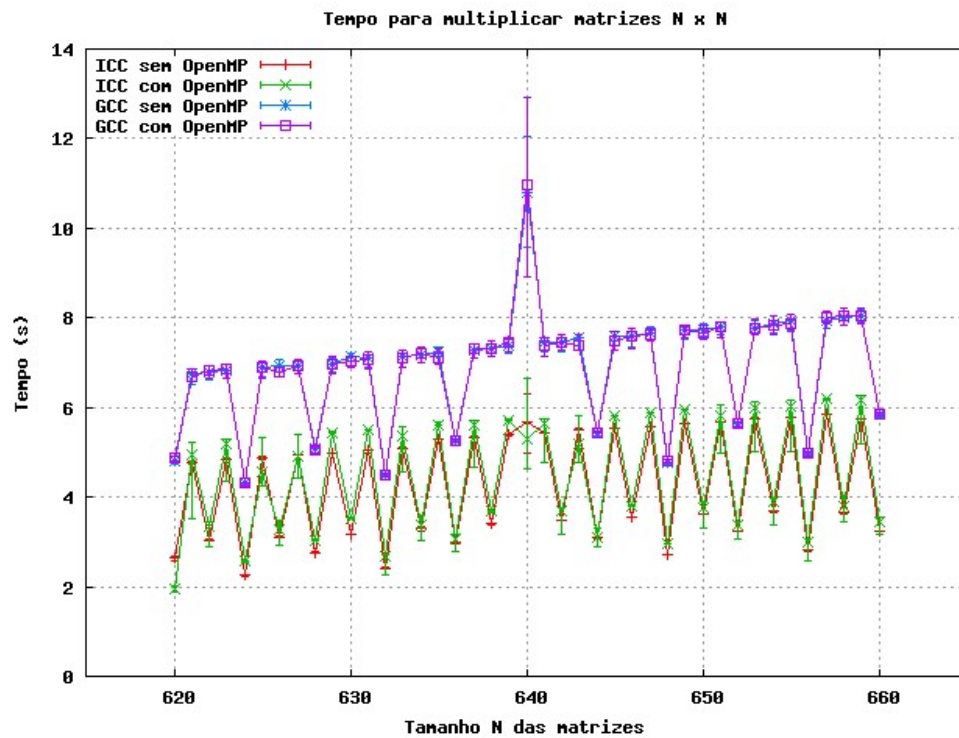


Figura 5.3: Detalhe do desempenho do OpenMP na multiplicação de matrizes quadradas

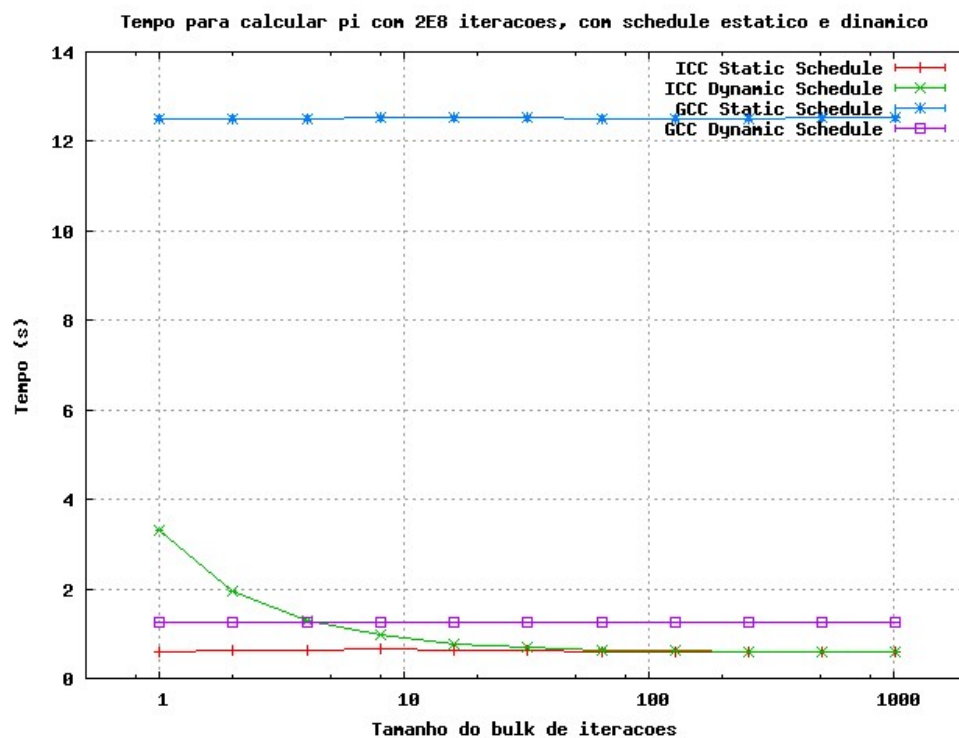


Figura 5.4: Desempenho do OpenMP com as diretivas dynamic e static

Com o compilador da Intel, o uso do OpenMP levou a um tempo de execução menor para a multiplicação de matrizes quadradas. Porém, com o compilador da Gnu, o tempo de execução manteve-se praticamente invariável. Isto acontece porque a implementação da diretiva reduction não está completa na versão do GCC que foi

estudada.

Adicionalmente, pode-se perceber que, quando o tamanho da matriz quadrada está por volta de 640×640 , existe um pico de tempo de execução. Executando os testes com mais detalhe em torno deste valor, foi obtido o gráfico da figura 5.3 na página 44. O gráfico mostra que o tempo de execução fica muito maior quando a matriz tem exatamente 640×640 de tamanho.

As diretivas `schedule static` e `schedule dynamic` também foram testadas quanto ao seu desempenho. O programa que calcula o valor de π com 2×10^8 passos foi compilado para usar as diretivas `static` e `dynamic`, com o número de iterações variando na sequência 1, 2, 4, 8 ... até 1024.

No compilador da Intel, a diretiva `schedule static` apresenta o mesmo desempenho independente do número de iterações que serão usadas em cada vez. Isto ocorre porque estas iterações são escalonadas em tempo de compilação. Por outro lado, a diretiva `schedule dynamic` apresenta péssimo desempenho quando o número de iterações usadas é pequeno. Isto ocorre devido ao overhead de se trocar de contexto a cada vez que as iterações são esgotadas. Conforme o número de iterações cresce, o desempenho tende a ser o mesmo que o da diretiva `static`.

No compilador da Gnu, as duas diretivas mostram um desempenho constante em relação ao número de iterações. Além disso, o tempo de execução da diretiva `static` é muito superior ao da diretiva `dynamic`. Isto sugere que as diretivas `schedule dynamic` e `schedule static` talvez não tenham sido implementadas corretamente pela versão do GCC em estudo.

5.2 Benchmarks NAS

O conjunto *NAS Parallel Benchmarks* (16) foi criado para comparar o desempenho de supercomputadores paralelos. Trata-se de oito programas derivados da dinâmica de fluidos computacional (CFD), sendo cinco kernels (CG, MG, FT, IS, EP) e três aplicativos simulados (BT, SP, LU), criados pela NASA.

A NASA não disponibiliza uma implementação do benchmark NAS com OpenMP. Em vez disso, ela sugere a utilização de benchmarks implementados por terceiros. A versão utilizada aqui foi obtida em (17).

Cada programa do benchmark é disponibilizado em cinco “classes” diferentes, sendo que cada classe representa um conjunto de dados diferente a ser analisado pelo programa. As classes são identificadas pelas letras A, B, C, S e W. O tempo de execução dos programas varia grandemente de acordo com cada classe de testes. Os dados das classes B e C faziam com que os benchmarks demorassem muito tempo para rodar, e por motivos de simplicidade foram omitidos. Neste trabalho estão os resultados da mensuração da execução A, S e W.

Os gráficos das figuras 5.5 e 5.6 representam o tempo de execução dos programas BT, CG, EP, FT, LU, MG e SP. O programa IS foi omitido por executar com tempo inferior a 0,01s em todas as execuções. Cada programa foi compilado e executado com o compilador da Intel e o GCC. Como o tempo de execução de cada programa variava dramaticamente de acordo com a classe escolhida, o desempenho, em cada gráfico, foi normalizado. Para cada uma das classes, o tempo de execução do programa compilado com o GCC foi tomado como base. O tempo de execução do programa compilado com o ICC foi normalizado.

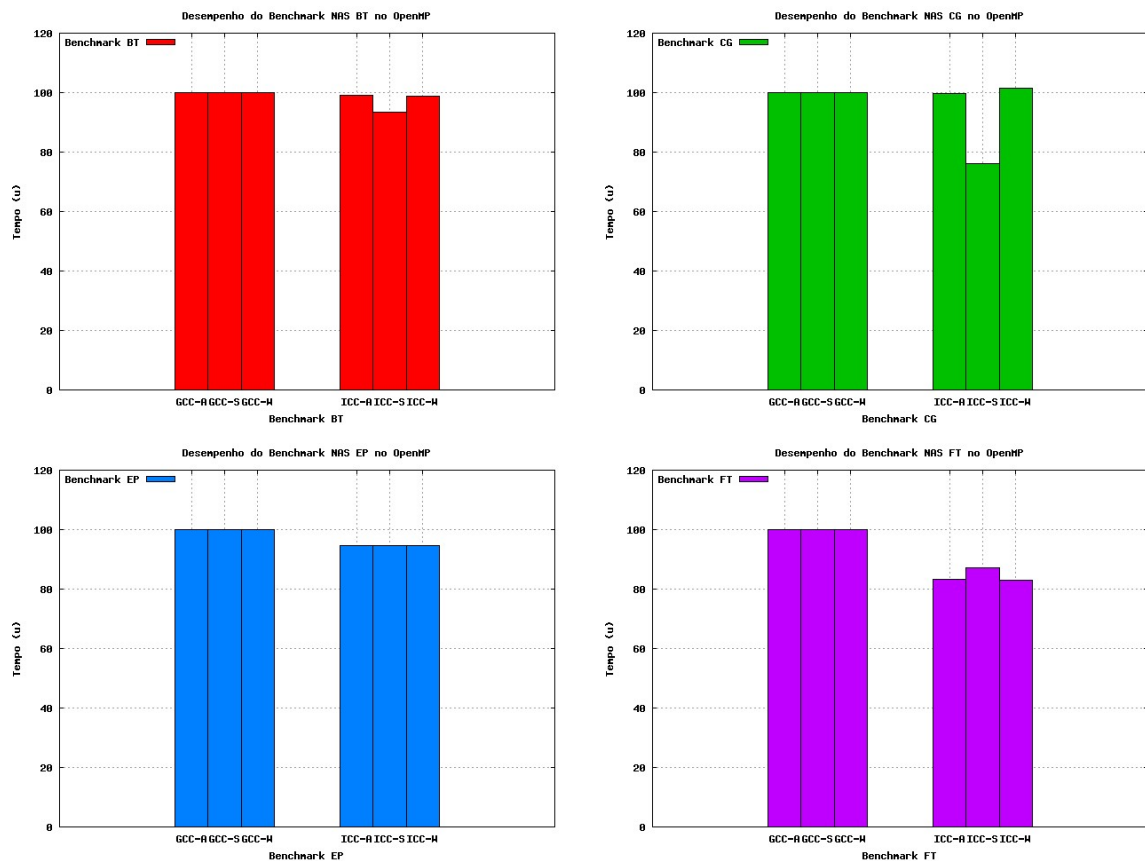


Figura 5.5: Comparação de desempenho entre GCC e ICC nos benchmarks BT, CG, EP e FT

A normalização aplicada nos gráficos torna possível se abstrair o tempo de execução de cada programa e mensurar o que é realmente importante: o tempo de execução do ICC em relação ao GCC, permitindo comparar o desempenho dos dois compiladores.

Analisando os gráficos, observa-se que em alguns benchmarks o desempenho dos dois compiladores é semelhante, enquanto que em outros benchmarks o ICC apresenta performance muito superior ao GCC.

Nos benchmarks BT, EP, LU e MG, o desempenho do OpenMP é muito parecido ao compararmos o ICC e o GCC. De maneira geral, o desempenho do ICC é ligeiramente superior, mas por margens de no máximo 5%.

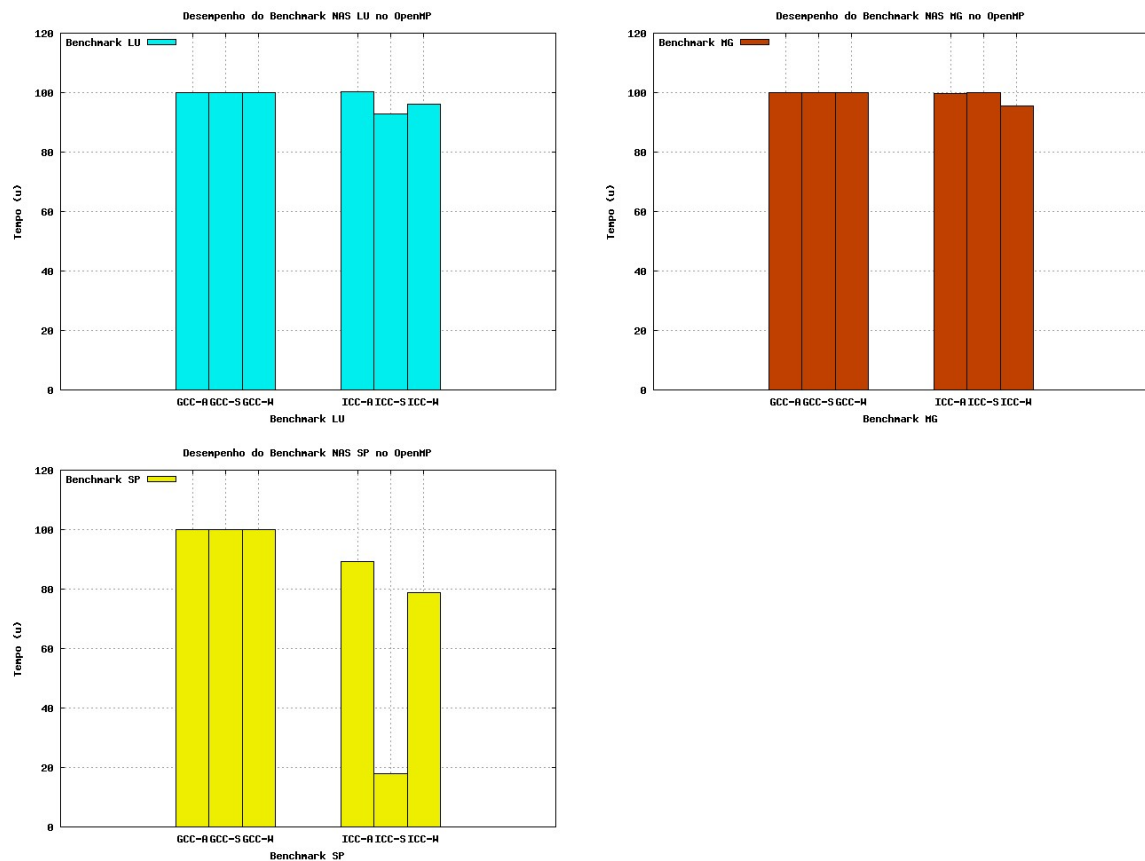


Figura 5.6: Comparação de desempenho entre GCC e ICC nos benchmarks LU, MG e SP

Nos benchmarks FT e SP, o tempo de execução com o compilador ICC é vastamente menor, superando de 10% a 20% o GCC. Na classe S de testes do benchmarks SP, o tempo de execução é 80% menor no ICC.

O benchmark CG mostra um desempenho bastante peculiar. Na classe W, o desempenho é ligeiramente pior no ICC do que no GCC. No entanto, na classe S, o desempenho do ICC supera o GCC em mais de 20%.

Observa-se de maneira geral que o compilador da Intel apresenta um desempenho melhor do que o compilador GCC. Apesar da vantagem em se utilizar o ICC, deve-se destacar o fato de que o GCC constitui um compilador gratuito de código aberto e que a implementação do OpenMP nele é recente e ainda tem o que amadurecer. Dados estes fatos, a diferença de desempenho não é tão grande de forma que se possa desprezar por completo o uso do GCC com o OpenMP. O GCC com OpenMP tem potencial para no futuro apresentar desempenho muito semelhante ao do compilador da Intel.

6 CONCLUSÃO

Neste trabalho foi abordado o OpenMP e o seu desempenho em arquiteturas paralelas. Através de uma contextualização do mundo atual e do corrente estado da tecnologia, compreende-se que o futuro dos processadores encontra-se na comercialização em massa dos multi-cores.

Inicialmente, foi realizada uma apresentação de conceitos básicos da arquitetura de computadores. Os computadores single-core, que até o ano de 2005 reinavam em absoluto no mercado de desktops, aos poucos não encontram mais espaços para melhoria de performance, pois estes processadores estão alcançando um limite físico de seus componentes que impede uma melhora significativa no desempenho.

Partindo do fato que grande parte dos aplicativos produzidos hoje são multi-threaded, é seguro afirmar que recursos de hardware que permitam a execução de múltiplas threads simultaneamente tornem os computadores, em geral, mais rápidos. Na primeira década do século XXI, a Intel lançou no mercado os processadores com tecnologia HT, que através da replicação de alguns recursos do processador permitem ao Sistema Operacional executar duas threads em dois processadores lógicos distintos, mesmo que abaixo deles exista um processador só. Mais tarde, o lançamento de processadores dual-core tornou possível ao usuário doméstico experimentar a verdadeira capacidade de multi-threading dentro do hardware. Com processadores multi-core, é possível executar diversas threads em hardware simultaneamente.

A mudança em direção aos processadores multi-core é permanente. O mercado aponta nesta direção e os preços têm caído consideravelmente, tanto nos Estados Unidos e na Europa como no Brasil.

Conceitos fundamentais de threads também foram abordados por este trabalho. Existem diversas APIs para threading atualmente, e a API Pthreads é uma das mais aceitas no momento. Foram apresentadas algumas características básicas desta API, e foi possível ver que muitos algoritmos de multi-threading simples são difíceis de se programar ao se utilizar a API Pthreads. Dificuldade de programação costuma levar a dificuldade de manutenção e surgimento de *bugs*.

Em seguida, foi apresentado o OpenMP, e algumas das suas estruturas básicas de programação. Através da simples leitura de algumas listagens de código do OpenMP, pode-se ver que a programação é muito mais simples com o OpenMP do que com a Pthreads. Foram listadas neste trabalho algumas implementações em que a comparação entre o OpenMP e a Pthreads mostra que o OpenMP é superior tanto em termos de clareza de código como facilidade de programação, assim como escalabilidade.

Durante a realização deste trabalho, o OpenMP mostrou-se como solução adequada para criação de aplicações paralelas. A facilidade de programação do OpenMP

também o coloca como a melhor solução para adaptar programas existentes de forma a executar paralelamente. Esta técnica pode ser aplicada em programas comerciais de forma a melhorar seu desempenho em arquiteturas paralelas. A facilidade de programação do OpenMP traduz-se em menor esforço de programação para as empresas e produtos com preço final mais barato para o consumidor.

A bibliografia e a documentação do OpenMP costumam apresentá-lo como sendo, em termos de performance, muito dependente da implementação. O uso das diretivas de programação do OpenMP provavelmente esconde alguns problemas de implementação muito capciosos, que exigem um estudo técnico e cuidadoso para serem criadas corretamente. Para medir a influência disto no desempenho, foram realizados benchmarks no OpenMP, usando dois compiladores: o ICC, da Intel, e o GCC, da Gnu.

Nos testes realizados, o compilador da Intel apresentou, em geral, performance superior ao compilador GCC. Antes de descartar o GCC como alternativa viável, não se pode esquecer de dois importantes fatores: em primeiro lugar, a Intel é uma empresa que investe fortemente em seu produto, que é um dos líderes do segmento, e apóia fortemente o OpenMP, tendo sido uma das primeiras a dar suporte a ele em seu compilador. Em segundo lugar, o suporte ao OpenMP embutido no GCC ainda tem caráter experimental, e não está oficialmente integrado à versão de *release* do GCC, pelo menos não na data de composição deste trabalho. Estes dois fatores combinados explicam por que o desempenho foi superior em programas compilados com o ICC do que com o GCC.

Porém, apesar dos fatores apresentados, em alguns benchmarks a performance do compilador Intel não apresentou superioridade tão grande em relação ao GCC. Certamente o compilador da Gnu tem potencial para no futuro apresentar um desempenho melhor, especialmente considerando o quanto recente é a implementação do OpenMP no GCC.

Os experimentos realizados neste trabalho poderiam ser estendidos, futuramente, de forma a englobar máquinas multi-core com dois, quatro ou mais cores. Seria interessante analisar o ganho de performance obtido ao executar os mesmos benchmarks em máquinas muito semelhantes em configuração, exceto pelo processador, variando-o entre um single-core, um dual-core e um quad-core, e observar o ganho em desempenho obtido nestes casos.

REFERÊNCIAS

- [1] PATTERSON, D. A. ; HENNESSY, J. L. *Computer organization and design: the hardware/software interface*. 3rd ed. Amsterdam ; Boston : Elsevier/Morgan Kaufmann, 2005.
- [2] AKHTER, S. ; ROBERTS, J. *Multi-Core Programming: Increasing Performance through Software Multi-threading*. Intel Press, 2006.
- [3] TANENBAUM, A. S. *Sistemas Operacionais Modernos*. 2a. ed. Pearson, 2003.
- [4] PETZOLD, C. *Programming Windows*. 5th ed. Microsoft Press, 1998.
- [5] ANDREWS, G. *Concurrent Programming: Principles and Practice*. The Benjamin Cummings, 1991.
- [6] CARISSIMI, A. S. ; TOSCANI, S. S. ; OLIVEIRA, R. S. *Sistemas Operacionais e Programação Concorrente*. Porto Alegre : Sagra Luzzato, 2003.
- [7] CHANDRA, R. ; DAGUM, L. ; KOHR, D. ; MAYDAN, D. ; McDONALD, J. ; MENON, R. *Parallel Programming in OpenMP*. San Francisco : Morgan Kaufmann Publishers, 2001.
- [8] INFO Online. *AMD anuncia processador com quatro núcleos*. Disponível em: <<http://info.abril.com.br/aberto/infonews/052006/16052006-15.shl>>. Acessado em 18 de maio de 2006.
- [9] CNet News. *New Intel quad-core chips are really double duo-cores*. Disponível em: <http://news.com.com/New+Intel+quad-core+chips+are+really+double+duo-cores/2100-1006_3-6047182.html>. Acessado em 20 de maio de 2006.
- [10] Zero Hora. *Caderno de Informática*. Porto Alegre, 2006.
- [11] Pthreads Win32. *Open Source POSIX Threads for Win32*. Disponível em: <<http://sourceware.org/pthreads-win32/>>. Acessado em 21 de outubro de 2006.
- [12] OpenMP: Simple, Portable, Scalable SMP Programming. *OpenMP*. Disponível em: <<http://www.openmp.org/>>. Acessado em 12 de maio de 2006.
- [13] Intel Corporation. *Intel C/C++ Compiler*. Disponível em: <<http://www.intel.com/>>. Acessado em 12 de maio de 2006.

- [14] Microsoft Visual Studio 2005 Professional Guided Tour. *OpenMP*. Disponível em: <http://msdn.microsoft.com/vstudio/tour/vs2005_guided_tour/VS2005pro/Framework/CPlusOpenMP.htm>. Acessado em 12 de maio de 2006.
- [15] GOMP. *GOMP: An OpenMP Implementation for GCC*. Disponível em: <<http://gcc.gnu.org/projects/gomp/>>. Acessado em 12 de maio de 2006.
- [16] NASA Advanced Supercomputing Division. *NAS Parallel Benchmarks*. Disponível em: <<http://www.nas.nasa.gov/Resources/Software/npb.html>>. Acessado em 13 de outubro de 2006.
- [17] NAS Parallel Benchmarks in OpenMP. *OpenMP C versions of NPB*. Disponível em: <<http://phase.hpcc.jp/Omni/benchmarks/NPB/>>. Acessado em 13 de outubro de 2006.