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

MÁRCIO DE OLIVEIRA DA SILVA

Controle de granularidade de tarefas em OpenMP

Trabalho de Conclusão apresentado como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação

Prof. Dr. Nicolas Maillard
Orientador

Porto Alegre, dezembro de 2011

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

de Oliveira da Silva, Márcio

Controle de granularidade de tarefas em OpenMP / Márcio de Oliveira da Silva. – Porto Alegre: Graduação em Ciência da Computação da UFRGS, 2011.

66 f.: il.

Trabalho de Conclusão (bacharelado) – Universidade Federal do Rio Grande do Sul. Curso de Bacharelado em Ciência da Computação, Porto Alegre, BR–RS, 2011. Orientador: Nicolas Maillard.

1. UFRGS. 2. Programação concorrente. 3. Programação paralela. 4. Paralelismo de tarefas. 5. Granularidade de tarefas. 6. Openmp. I. Maillard, Nicolas. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitora de Graduação: Profa. Valquiria Link Bassani
Diretor do Instituto de Informática: Prof. Luís da Cunha Lamb
Coordenador do Curso: Prof. Raul Fernando Weber
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS 5

LISTA DE FIGURAS 7

LISTA DE TABELAS 9

1 INTRODUÇÃO 11
1.1 Estrutura do Trabalho 12

2 CONTEXTO CIENTíFICO: COMPUTAÇÃO PARALELA 13
2.1 Arquiteturas Paralelas 13
2.1.1 Arquiteturas com memória compartilhada 14
2.1.2 Arquiteturas com memória distribuída 16
2.2 Programação Paralela 17
2.2.1 Paralelismo de Dados 18
2.2.2 Paralelismo de Laços 18
2.2.3 Paralelismo de Tarefas 18
2.3 APIs com suporte a Paralelismo de Tarefas 18
2.3.1 Cilk 19
2.3.2 Intel® Threading Building Blocks 19
2.3.3 OpenMP 20
2.4 Considerações finais 20

3 OPENMP 23
3.1 O que é OpenMP 23
3.1.1 Histórico 23
3.2 Modelo de execução 24
3.2.1 Exemplos de uso 25
3.3 Paralelismo de tarefas em OpenMP 28

4 GRANULARIDADE DE TAREFAS 31
4.1 Speedup 31
4.2 Balanceamento de carga 31
4.3 Sobrecusto e granularidade de tarefas 32

5 CONTROLE DE GRANULARIDADE DE TAREFAS EM OPENMP . . . 35
5.1 O parâmetro *maxdepth* 35
5.2 Cláusulas auxiliares para a diretiva `task` 35
5.2.1 Tarefa *undeferred* 36

5.2.2 Tarefa *final* 36
5.2.3 Tarefa *merged* 37
5.3 Estratégias de controle de granularidade de tarefas 37
5.3.1 Controle manual 38
5.3.2 Cláusula if 39
5.3.3 Cláusula final 39
5.3.4 Cláusula final + API 39
5.4 Conclusão 41

6 AVALIAÇÃO DAS ESTRATÉGIAS PROPOSTAS 43
6.1 Metodologia de execução dos experimentos 43
6.1.1 Amostragem 43
6.1.2 Algoritmos testados 44
6.1.3 Plataforma experimental 44
6.2 Resultados 46
6.2.1 Quicksort 46
6.2.2 NQueens 51
6.2.3 Fibonacci 56

7 CONCLUSÃO 63

REFERÊNCIAS 65

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
OpenMP ARB	*OpenMP Architecture Review Board*
OpenMP	*Open Multi-Processing*
SMP	*Symmetric Multiprocessor*
UMA	*Uniform Memory Access*
NUMA	*Nonuniform Memory Access*
TBB	*Threading Building Blocks*

LISTA DE FIGURAS

2.1 Arquitetura com memória compartilhada 15
2.2 Arquitetura com memória distribuída 16
2.3 Exemplo de código utilizando **Cilk** 19
2.4 Exemplo de código utilizando **TBB** 20

3.1 Diretiva `omp for` 25
3.2 Exemplo de uso da diretiva `omp for` 25
3.3 Exemplo de uso da cláusula `shared` 26
3.4 Saída do programa com cláusula `shared` 27
3.5 Exemplo de uso da cláusula `private` 27
3.6 Saída do programa com cláusula `private` 27
3.7 Fibonacci com paralelismo de tarefas **OpenMP** 29

4.1 Uma árvore de chamadas recursivas 33

5.1 Exemplo de uso da cláusula `if` 36
5.2 Exemplo de uso da cláusula `final` 37
5.3 Exemplo de uso da cláusula `mergeable` 37
5.4 Controle manual de granularidade de tarefas 38
5.5 Controle de granularidade com cláusula `if` 39
5.6 Controle de granularidade com cláusula `final` 40
5.7 Controle de granularidade com cláusula `final` + API 40

6.1 Quicksort: diferentes profundidades máximas com 2 *threads* 47
6.2 Quicksort: diferentes profundidades máximas com 4 *threads* 47
6.3 Quicksort: diferentes profundidades máximas com 8 *threads* 48
6.4 Quicksort: diferentes profundidades máximas com 16 *threads* 48
6.5 Quicksort: diferentes profundidades máximas com 32 *threads* 49
6.6 Quicksort: *speedup* em diferentes números de *threads* 50
6.7 NQueens: diferentes profundidades máximas com 2 *threads* 51
6.8 NQueens: diferentes profundidades máximas com 4 *threads* 52
6.9 NQueens: diferentes profundidades máximas com 8 *threads* 53
6.10 NQueens: diferentes profundidades máximas com 16 *threads* 53
6.11 NQueens: diferentes profundidades máximas com 32 *threads* 54
6.12 NQueens: *speedup* em diferentes números de *threads* 55
6.13 Fibonacci: diferentes profundidades máximas com 2 *threads* 56
6.14 Fibonacci: diferentes profundidades máximas com 4 *threads* 57
6.15 Fibonacci: diferentes profundidades máximas com 8 *threads* 58

6.16 Fibonacci: diferentes profundidades máximas com 16 *threads* 59
6.17 Fibonacci: diferentes profundidades máximas com 32 *threads* 60
6.18 Fibonacci: *speedup* em diferentes números de *threads* 61

LISTA DE TABELAS

2.1 Multiprocessadores UMA comerciais 15
2.2 Os 3 supercomputadores mais poderosos atualmente 17

1 INTRODUÇÃO

Na ciência da computação, as melhores soluções conhecidas (em relação à complexidade computacional) para muitos problemas, são dadas por algoritmos cujo número de operações não pode ser determinado estaticamente. Isto acontece, por exemplo, em algoritmos que exploram um espaço de busca irregular e imprevisível, tais como os que aparecem em problemas de otimização combinatória. Bem como em algoritmos que utilizam técnicas que visam explorar de forma não exaustiva e irregular um determinado espaço de busca, como programação dinâmica e *backtracking*.

Tais algoritmos possuem a característica de realizar chamadas recursivas que executam tarefas (conjuntos de instruções) sem dependência causal entre si. Portanto, é uma ideia interessante que estas tarefas possam ser processadas de forma paralela, a fim de otimizar o uso dos recursos providos pelos múltiplos elementos de processamento[1] disponíveis nas máquinas atuais.

Devido ao número de operações não poder ser determinado estaticamente, a quantidade de tarefas geradas não é conhecida com antecedência, ocasionando um paralelismo irregular, pois, se considerarmos a criação de tarefas como nodos em uma árvore, não é possível determinar seu balanceamento, largura ou altura.

Com o objetivo de prover meios de expressar este tipo de paralelismo, existem interfaces de programação (APIs) que disponibilizam diretivas para explicitar tarefas paralelas, instruindo o sistema operacional a criar novas unidades de execução cada vez que as tarefas forem acionadas. No entanto, estas ferramentas precisam lidar com o problema causado pela incapacidade de prever o quão profunda e larga será a árvore de tarefas criadas em uma execução.

Quando a granularidade das tarefas se torna muito fina, ou seja, a árvore de tarefas atinge um nível com muitos nodos, o custo imposto ao sistema operacional para criar e manter tantas unidades de execução supera o ganho em desempenho proporcionado pelo paralelismo. Por isso, a necessidade de um mecanismo eficiente para impedir que isto ocorra se apresenta como um desafio para as interfaces de programação que suportam paralelismo de tarefas.

Este trabalho apresenta uma análise de diferentes estratégias de controle de granularidade de tarefas, em programas que fazem uso deste tipo de paralelismo, utilizando uma API para programação paralela chamada **OpenMP**.

[1] Como em (CER 2011), aqui serão adotados os termos **elemento de processamento** para processadores ou núcleos, e **unidade de execução** para representar *threads* ou processos.

1.1 Estrutura do Trabalho

Primeiramente, o texto se propõe a situar o leitor no contexto científico que envolve o assunto, no capítulo 2. Após, no capítulo 3, é feita uma breve apresentação da API **OpenMP**, com histórico, modelo de execução e exemplos de uso. Em seguida, no capítulo 4, são apresentadas questões que envolvem a granularidade de tarefas criadas ao longo da execução de programas, como balanceamento de carga e sobrecusto decorrente do uso deste tipo de paralelismo. Ainda neste capítulo, é apresentada uma métrica comumente utilizada na avaliação de desempenho de programas paralelos, chamada *speedup*. Feito isso, no capítulo 5 são apresentadas as estratégias de controle de granularidade de tarefas em **OpenMP** propostas. No capítulo 6 está um relatório dos experimentos realizados, com descrição da metodologia de execução, resultados obtidos e análises sobre os mesmos. Por fim, no capítulo 7, são feitas as devidas considerações finais acerca do trabalho realizado e dos resultados alcançados, apresentando sugestões para trabalhos futuros.

2 CONTEXTO CIENTÍFICO: COMPUTAÇÃO PARALELA

Este capítulo tratará sobre o contexto científico atual relacionado à computação paralela. Primeiramente, na seção 2.1, serão mostradas as principais arquiteturas de *hardware* empregadas. Em seguida, na seção 2.2, serão apresentados os diferentes tipos de paralelismo encontrados em programas paralelos. Com isso, seguirá, na seção 2.3, um apanhado das APIs de programação paralela com suporte a paralelismo de tarefas mais conhecidas.

2.1 Arquiteturas Paralelas

Um dos objetivos da programação paralela é resolver problemas complexos e sobre grandes quantidades de dados utilizando de forma eficiente os recursos computacionais disponíveis atualmente em sistemas com múltiplos elementos de processamento. No entanto, a popularização dos sistemas com múltiplos elementos de processamento se deu apenas no início do século 21, motivada principalmente pela saturação dos recursos tecnológicos utilizados em monoprocessadores[1], que se tornaram uma barreira para suprir a demanda de processamento sobre volumes de dados cada vez maiores.

Os monoprocessadores tiveram grandes melhorias de desempenho no período de 1986 a 2002, porém, esta evolução esbarrou em limites físicos, como o calor gerado pelo crescente número de transistores necessários para aumentar a taxa de relógio, e o alto consumo de energia elétrica (HEN 2006).

Além do fim comercial da era dos monoprocessadores, outros fatores também contribuíram para a evolução das arquiteturas paralelas, como:

- Interesse crescente em servidores de alto desempenho
- Crescimento de aplicações que manipulam grandes volumes de dados
- Um melhor entendimento de como utilizar sistemas com múltiplos elementos de processamento com eficiência, especialmente em servidores, onde há, naturalmente, um significativo paralelismo de unidades de execução.

Esta ideia de utilizar múltiplas unidades de processamento para aumentar o desempenho e a disponibilidade dos recursos computacionais existe desde quando os primeiros computadores eletrônicos foram construídos. Em 1966, Flynn (FLY 72) propôs um modelo para dividir computadores em categorias bem abrangentes, que ainda é útil, pois estabelece classes de computadores de acordo com sua capacidade de paralelizar instruções e dados. Através deste modelo, foram apresentadas as seguintes categorias:

[1] o termo **monoprocessador** aqui significa "processador com apenas um núcleo de processamento", ou seja, um processador sem capacidade de tratar duas ou mais unidades de execução paralelamente.

- *Single Instruction, Single Data* (SISD) - Categoria onde se enquadram os monoprocessadores. Computadores SISD não são capazes de explorar qualquer tipo de paralelismo temporal, uma vez que apenas uma instrução pode ser processada por ciclo.

- *Single Instruction, Multiple Data* (SIMD) - Uma mesma instrução é executada por múltiplas unidades de processamento em diferentes dados de uma estrutura. Computadores SIMD exploram paralelismo em nível de dados ao aplicar uma mesma operação sobre múltiplos operandos. Por exemplo, em um *array* de números inteiros (múltiplos dados), realizar a operação que incrementa o valor contido em cada posição do *array* em 1, de forma simultânea.

- *Multiple Instruction, Single Data* (MISD) - Até o dia em que este trabalho foi escrito, não existem processadores comerciais deste tipo.

- *Multiple Instruction, Multiple Data* (MIMD) - Cada unidade de processamento busca sua própria instrução e opera sobre seus próprios dados. Computadores MIMD exploram paralelismo em nível de *threads* e de processos, possibilitando que múltiplas unidades de execução sejam processadas paralelamente.

Devido ao objeto de estudo deste trabalho (controle de granularidade de tarefas em **OpenMP**) estar diretamente relacionado a paralelismo em nível de *threads*, a única categoria que nos interessa, dentre as citadas acima, é a MIMD.

Processadores MIMD são divididos em dois grupos, que são determinados de acordo com a organização da memória de trabalho e a maneira como os diferentes elementos de processamento estão interconectados. Estes grupos são classificados como Arquiteturas com Memória Compartilhada e Arquiteturas com Memória Distribuída.

2.1.1 Arquiteturas com memória compartilhada

Para sistemas com um número relativamente pequeno de elementos de processamento, é possível que os mesmos compartilhem o mesmo espaço de endereçamento. Isso torna a implementação de programas paralelos mais simples, pois, diferentemente do que acontece em arquiteturas com memória distribuída, a comunicação entre os elementos de processamento ocorre de forma "natural", através do mesmo espaço de endereçamento. Assim, o compartilhamento de dados entre as unidades de execução de um programa paralelo se torna extremamente rápido.

Devido a esta arquitetura apresentar um espaço de endereçamento único, porém múltiplos elementos de processamento, os processadores são chamados de **multiprocessadores** (DER 2003).

Em relação ao tipo de acesso à memória do sistema, multiprocessadores podem ser classificados como:

- **Acesso à memória uniforme (UMA)**: A memória principal utilizada nestas máquinas é centralizada e encontra-se à mesma distância de todos os elementos de processamento, fazendo com que a latência de acesso à memória seja uniforme. Para satisfazer a demanda de memória de todos os elementos de processamento, de forma a não sobrecarregar o único barramento de comunicação, é necessário que cada um possua uma memória especial, pequena e de acesso rápido, chamada de memória *cache*. Isso faz com que o número de acessos à memória principal seja reduzido. A figura 2.1 ilustra esta arquitetura.

- **Acesso não uniforme à memória (NUMA)**: A memória principal utilizada nestas máquinas é implementada com múltiplos módulos, que são associados um a cada elemento de processamento. O espaço de endereçamento é único, e cada elemento pode endereçar toda a memória do sistema. Se o endereço gerado pelo elemento de processamento encontrar-se no módulo de memória diretamente ligado a ele, dito local, o tempo de acesso será menor que o tempo de acesso a um módulo que está ligado diretamente a outro, dito remoto, que poderá ser acessado apenas através de uma rede de interconexão. Por isso, estas máquinas possuem um acesso não uniforme à memória (DER 2003).

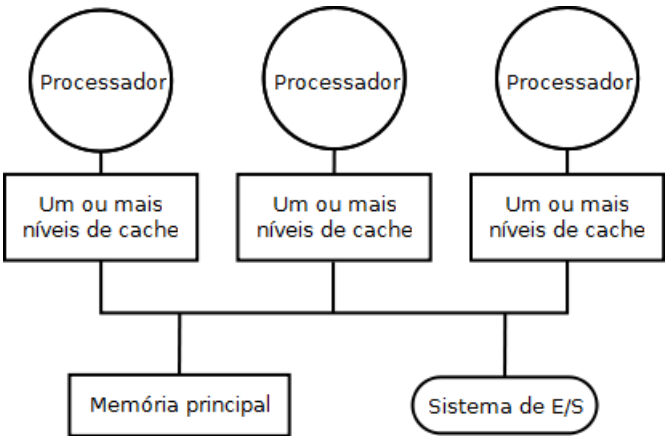

Figura 2.1: Arquitetura com memória compartilhada UMA. (HEN 2006)
Múltiplos subsistemas processador-cache compartilham a mesma memória principal, tipicamente conectados por um ou mais barramentos ou a um *switch*. Os elementos são gerenciados por um único sistema operacional, acessado através de um sistema de E/S.

Multiprocessadores UMA são muito populares nos dias de hoje. Servidores, computadores de uso pessoal (*desktops e notebooks*) e até mesmo dispositivos móveis (*smartphones e tablet PCs*) dispõem desta tecnologia. Com isso, percebe-se um avanço significativo dos processadores comerciais nos últimos anos. Na tabela 2.1 estão alguns exemplos[2]:

Nome	Uso típico	Elementos de processamento	Cache
Intel® Xeon® E7-8870	servidores	10	30 MB
AMD® Opteron 6100	servidores	12	12 MB
Intel® Core i7-2600	*desktop*	4	8 MB
AMD® Phenom II 1090	*desktop*	6	9 MB
ARM® Cortex-A9	dispositivos móveis	2	1 MB

Tabela 2.1: Multiprocessadores UMA comerciais

A principais desvantagens de sistemas com memória compartilhada são as seguintes:

- Pouca escalabilidade de elementos de processamento. Ao inserir novos elementos, o tráfego no barramento de comunicação com a memória principal aumenta geometricamente.

[2]pesquisa feita em 16 de Setembro de 2011, nos sites dos fabricantes.

- Uma vez que o espaço de endereçamento é único, a arquitetura não garante acesso exclusivo à memória principal a um elemento de processamento específico. A responsabilidade de desenvolver mecanismos de sincronização corretos é do desenvolvedor.

2.1.2 Arquiteturas com memória distribuída

Memória distribuída refere-se à localização física da memória. Se a memória é implementada com vários módulos, e cada módulo foi colocado próximo de um processador, então a memória é considerada distribuída (DER 2003). Assim, os elementos de processamento de um sistema com esta arquitetura possuem espaços de endereçamento distintos. Como não é possível o uso de variáveis compartilhadas neste ambiente, a troca de informações é feita por envio de mensagens através de uma rede de interconexão. Por isso, estas máquinas tambén são chamadas de sistemas de troca de mensagens (*message passing systems*) (DER 2003).

Estas características resultam no fato de este tipo de máquina paralela ser construída através da replicação de toda a arquitetura, e não apenas dos elementos de processamento, como nos multiprocessadores. Por isso, são chamadas de **multicomputadores** (DER 2003).

Esta arquitetura é adequada para suportar um número muito grande de elementos de processamento, pois, ao atribuir a cada elemento uma memória de trabalho própria, evita que todos disputem o mesmo recurso físico (memória). Esta disputa, quando excessiva, sobrecarrega o barramento de comunicação entre os processadores e a memória, gerando aumento de latência no acesso e consequente queda de desempenho de todo o sistema. A figura 2.2 ilustra basicamente a arquitetura.

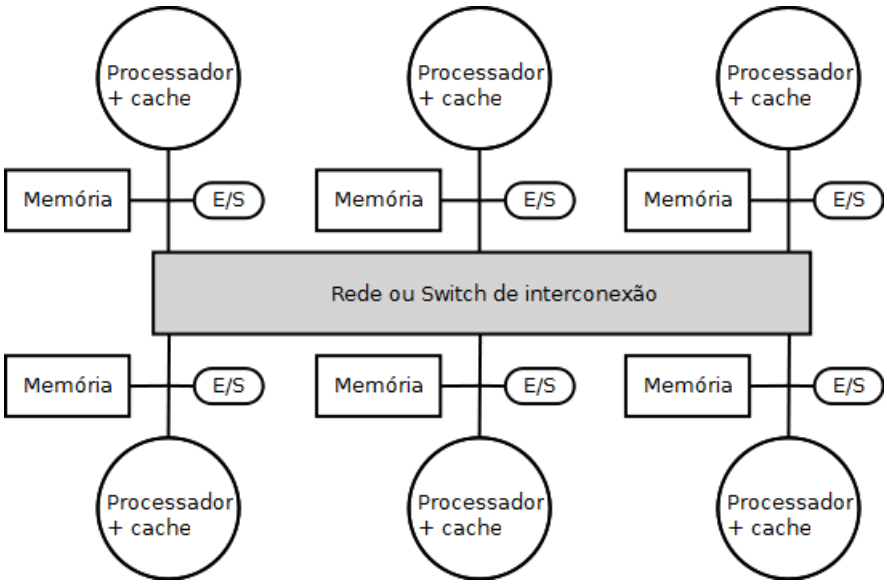

Figura 2.2: Arquitetura com memória distribuída básica. (HEN 2006)
Múltiplos elementos de processamento consistindo em nodos individuais que possuem processador, memória de trabalho, tipicamente um sistema de entrada e saída e uma interface com a rede de interconexão, responsável pela comunicação com os demais nodos.

O grande número de elementos de processamento distribuídos em diferentes locais físicos exige uma rede de interconexão com alta largura de banda, a fim de minimizar

a perda de desempenho decorrente das várias trocas de mensagens feitas durante o processamento de programas paralelos com alta taxa de comunicação entre as unidades de execução.

Outra característica inerente a multicomputadores advém do fato de cada nodo possuir seu próprio relógio físico. Como não é possível garantir de forma eficiente que todos os relógios estão sincronizados, é necessário que os programas paralelos implementem mecanismos adicionais para garantir a ordem temporal dos eventos, comumente chamados de relógios lógicos. Isso gera mais sobrecusto, pois tratam-se de operações adicionais a disputar os recursos de processamento com outras tarefas.

Segundo (HEN 2006), esta arquitetura possui dois grandes benefícios. Primeiramente, é uma maneira de escalar a largura de banda entre os processadores e a memória de trabalho com bom custo-benefício, principalmente quando a maioria dos acessos são locais. Além disso, quando são feitos muitos acessos à memória local, ocorre uma redução significativa na latência de acesso, pois a distância entre cada memória e seu elemento de processamento é pequena e não há concorrência pelo barramento de comunicação.

De acordo com o **TOP500 Project**[3], entidade que agrega informações e estatísticas sobre computadores de alto desempenho construídos ao redor do mundo, também chamados de supercomputadores, os sistemas computacionais com maior desempenho invariavelmente utilizam arquitetura com memória distribuída. Na tabela 2.2, estão alguns dados dos 3 primeiros colocados no ranking[4] da entidade:

Nome	Elementos de processamento	Desempenho (GFlops)	Local
K computer	548352	8162000	Japão
Tianhe-1A	186368	2566000	China
Jaguar	224162	1759000	Estados Unidos

Tabela 2.2: Os 3 supercomputadores com maior desempenho atualmente, de acordo com o **TOP500 Project**.

Nota-se o número massivo de elementos de processamento utilizados por tais sistemas, algo inviável em arquiteturas com memória compartilhada.

2.2 Programação Paralela

Com a evolução e popularização das arquiteturas paralelas, torna-se cada vez mais importante o estudo e a aplicação de técnicas para implementar algoritmos concorrentes sobre tais máquinas, para que os múltiplos elementos de processamento disponíveis sejam utilizados com eficiência.

A programação paralela trata da implementação de algoritmos concorrentes, utilizando recursos que permitem expressar o paralelismo das operações e incluindo mecanismos de sincronização e comunicação entre diferentes unidades de execução.

Programas paralelos podem apresentar diferentes formas de paralelismo, de acordo com a forma com que realizam operações simultâneas. Estes tipos são categorizados como: Paralelismo de Dados, Paralelismo de Laços e Paralelismo de Tarefas.

[3]http://http://www.top500.org

[4]Dados coletados dia 16 de Setembro de 2011, no endereço http://www.top500.org/lists/2011/06

2.2.1 Paralelismo de Dados

O paralelismo de dados acontece quando uma operação é aplicada sobre grandes conjuntos de dados de forma paralela. Neste caso, a concorrência não é especificada na forma de diferentes operações simultâneas, mas na aplicação da mesma operação sobre múltiplos elementos de uma estrutura de dados.

Como exemplo de paralelismo de dados, consideremos uma soma de matrizes. Sabemos que na soma de duas matrizes, os elementos são somados de forma independente em suas respectivas posições. Assim, dadas as matrizes A e B, para obtermos a soma de ambas, armazenando o resultado em uma terceira matriz C, podemos obter cada elemento C_{ij} paralelamente, aplicando a operação "soma" (uma operação) sobre todos os pares de elementos A_{ij}, B_{ij} (múltiplos dados) simultaneamente.

2.2.2 Paralelismo de Laços

O paralelismo de laços é similar ao paralelismo de dados, porém o mesmo se aplica quando queremos paralelizar operações contidas em um laço *for*, ou seja, um programa iterativo onde o número de repetições é conhecido com antecedência. Porém, para que os passos possam ser executados simultaneamente de forma correta, é necessário que não exista dependência causal entre o resultado do processamento de cada iteração, naturalmente.

O uso de paralelismo de laços permite expressar paralelismo de dados de forma mais genérica e escalável. Genérica pois, uma vez um laço *for* é paralelizado (através de uma API para programação paralela com suporte a paralelismo de laços, como **OpenMP**), a implementação se torna independente do número de unidades de execução que serão utilizadas pelo programa. E escalável porque isso faz a implementação ser adaptável ao ambiente onde o programa será executado.

2.2.3 Paralelismo de Tarefas

Paralelismo de tarefas é utilizado para expressar paralelismo de instâncias de conjuntos de instruções (tarefas) em um programa. A concorrência é especificada na forma de tarefas, que executam conjuntos de instruções determinados pelo programa. De maneira que, por exemplo, uma rotina (conjunto de instruções) pode ser executada n vezes, simultaneamente.

Por ter este caráter genérico, este tipo de paralelismo pode ser utilizado para paralelizar tarefas cujo número de execuções não pode ser previsto estaticamente, como acontece em programas recursivos e laços *while*, que implementam algoritmos irregulares. Nestes casos, também pode ser chamado de paralelismo irregular.

Em programas com paralelismo irregular, o número total de tarefas criadas, número de tarefas filhas geradas por uma tarefa, e pontos de sincronização não são conhecidos previamente. Assim, paralelismo irregular incorre em sobrecusto em tempo de execução, o que pode comprometer o desempenho caso as tarefas atinjam granularidade muito fina (AYG 2010). Para evitar que este sobrecusto domine a execução dos programas, é necessário implementar mecanismos de controle de granularidade das tarefas concorrentes.

2.3 APIs com suporte a Paralelismo de Tarefas

O uso de paralelismo de tarefas em arquiteturas com memória compartilhada é cada vez mais frequente (APE 2010), devido a esta abordagem ter se mostrado uma maneira

eficiente de paralelizar algoritmos irregulares.

Com o intuito de fornecer facilidades para o desenvolvimento de programas com paralelismo de tarefas, existem APIs que fornecem meios de definir explicitamente as partes do código que devem ser executadas como tarefas, em sistemas com memória compartilhada. Destas, esta seção apresentará as mais utilizadas e com maior nível de maturidade atualmente. **Cilk**, **Intel® Threading Building Blocks** e **OpenMP**.

Vale lembrar que, além das APIs vistas aqui. também existem APIs para programação paralela em sistemas com memória distribuída. Sendo a **MPI**[5] a mais difundida atualmente.

2.3.1 Cilk

Cilk[6] foi a primeira API a introduzir suporte a paralelismo de tarefas em arquiteturas com memória compartilhada. A API pode ser definida como um pré-compilador para a linguagem C, que atua ao adicionarmos as instruções `cilk`, `spawn` e `sync` para expressar o paralelismo. Caso estas instruções não estejam presentes no código, o mesmo continua válido para qualquer compilador C, característica conhecida como *elision*.

A figura 2.3 mostra um exemplo de uma implementação da versão recursiva do algoritmo que calcula o n-ésimo número da sequência de Fibonacci, onde cada chamada recursiva utiliza a instrução `spawn` para indicar uma nova tarefa **Cilk**. Como as variáveis x e y só poderão ser acessadas após o término das tarefas, é feito o uso da instrução `sync`, que bloqueia a execução da unidade atual até que as tarefas filhas (as que calculam os valores de x e y) tenham terminado.

```
cilk int fib(int n) {
    int x, y;
    if (n < 2) return n;
    x = spawn fib(n - 1);
    y = spawn fib(n - 2);
    sync;
    return x + y;
}
```

Figura 2.3: Exemplo de código utilizando **Cilk**

Cilk criou o princípio *work-first*, empregando um algoritmo de escalonamento de tarefas chamado *workstealing*, eficiente em tempo e ótimo em espaço. (BLU 94; BLU 99)

2.3.2 Intel® Threading Building Blocks

A API **TBB**[7] baseia-se no conceito de tarefas lógicas mapeadas em objetos da linguagem C++. Estas tarefas caracterizam-se por serem mais leves do que *threads*, pois não carregam dados adicionais como pilha de execução e estado de registradores. Esta compactação em termos estruturais possibilita que a execução de tarefas, em relação ao uso puro de *threads*, seja até 18 vezes mais rápida em sistema GNU/Linux, e até centenas de vezes mais rápida em sistemas Windows, ao diminuir o sobrecusto gerado pelo gerenciamento de *threads* (WIL 2008).

[5] http://www.mcs.anl.gov/research/projects/mpi/
[6] http://supertech.csail.mit.edu/cilk/
[7] http://threadingbuildingblocks.org/

Durante a inicialização de um programa que utiliza **TBB**, ocorre a criação de um *pool* de tarefas para cada *thread* que será utilizada. A execução das tarefas existentes no *pool* obedece um escalonamento baseado em *workstealing*, possibilitando um balanceamento de carga dinâmico. Também oferece suporte a decomposição de dados e controle de granularidade automática (ROB 2008).

TBB é uma biblioteca que fornece várias estruturas de dados para C++ para facilitar a programação paralela. Através de tais objetos, é possível expressar paralelismo de dados, laços e tarefas. Um exemplo de implementação de um algoritmo recursivo que calcula o n-ésimo número da sequência de Fibonacci, com paralelismo de tarefas, pode ser visto na figura 2.4[8].

```
int fib(int n) {
    int x, y;
    tbb::task_group g;
    g.run([&]{x = fib(n - 1);});
    g.run([&]{y = fib(n - 2);});
    g.wait();
    return x + y;
}
```

Figura 2.4: Exemplo de código utilizando **TBB**

2.3.3 OpenMP

A norma[9] **OpenMP**[10] é a mais difundida e efetivamente utilizada para programação paralela em sistemas com memória compartilhada atualmente.

OpenMP especifica um conjunto de diretivas para o compilador e rotinas a serem chamadas em tempo de execução, com o objetivo de expressar paralelismo em arquiteturas com memória compartilhada. A norma também define variáveis de ambiente a serem utilizadas pelas rotinas da API, em tempo de execução. As linguagens suportadas são Fortran, C e C++. Sendo que, inicialmente, não existia suporte a paralelismo de tarefas, que foi introduzido apenas em 2008, na versão 3.0.

Uma implementação de **OpenMP** será utilizada durante este trabalho e, por isso, há um capítulo dedicado exclusivamente ao assunto (capítulo 3).

2.4 Considerações finais

A computação paralela, apesar de não se tratar de uma área jovem em relação à história da informática (o modelo de Flynn foi proposto na década de 60), está em franco desenvolvimento. Arquiteturas paralelas com cada vez mais poder de processamento são projetadas. E para fazer um bom uso de tais máquinas, a programação paralela acompanha esta evolução, se apresentando como um tópico de pesquisa interessante.

[8] http://software.intel.com/en-us/articles/using-tasks-instead-of-threads/

[9] O termo "norma" aqui é utilizado para denotar o fato de **OpenMP** ser uma especificação de componentes e funcionalidades a serem implementados em compiladores, e não de uma implementação pronta. O termo "API" será utilizado quando estivermos nos referindo à implementação da norma.

[10] http://www.openmp.org

As APIs para programação paralela possuem papel importante neste cenário, pois trazem formas abstratas de representar diferentes tipos de paralelismo. O capítulo seguinte é dedicado a API **OpenMP**, que será utilizada neste trabalho.

3 OPENMP

Este capítulo trata de maneira mais minuciosa a norma **OpenMP**, cujo suporte a paralelismo de tarefas será a base deste trabalho. A seção 3.1 apresenta a norma, mostrando brevemente como a mesma surgiu e seu estado atual. A seção 3.2 descreve o modelo de execução e mostra exemplos de uso de algumas diretivas. Por fim, na seção 3.3 é apresentado um exemplo de programa que utiliza paralelismo de tarefas.

3.1 O que é OpenMP

OpenMP é uma norma desenvolvida para facilitar a implementação de programas paralelos em ambientes com memória compartilhada. Não se trata de um padrão sancionado oficialmente, mas sim de um acordo realizado entre membros da ARB[1], que compartilham o interesse em obter uma abordagem para programação paralela portável, amigável ao desenvolvedor e eficiente (CHA 2008).

3.1.1 Histórico

No início dos anos 90, os fabricantes de multiprocessadores com memória compartilhada forneciam estensões para a linguagem de programação Fortran, com a intenção de prover aos desenvolvedores meios de paralelizar laços em programas sequenciais. Estas estensões traziam diretivas que delegavam ao compilador a tarefa de paralelizar automaticamente tais laços. Porém, havia muitas implementações divergentes, mas com funcionalidades similares. Não existia um padrão.

A partir disso, em 1994 ocorreu a primeira tentativa de estabelecer um conjunto de estensões padronizado para auxiliar a implementação de programas paralelos em Fortran, o que foi chamado de ANSI X3H5. Este padrão foi amplamente adotado para compilar programas paralelos com a finalidade de serem executados em arquiteturas com memória distribuída, seguindo a popularidade que estas máquinas estavam obtendo na época (BAR 2011).

Assim, foi aberto o caminho para o desenvolvimento de um padrão a ser utilizado por compiladores para arquiteturas com memória compartilhada e, em 1997, foi iniciada a especificação da norma **OpenMP**. O projeto foi conduzido pela ARB, que originalmente incluía os seguintes membros:

- Compaq / Digital

[1] Como consta em http://openmp.org/wp/about-openmp/ (acessado dia 27 de Julho de 2011), a **OpenMP ARB** (ou somente ARB) é a corporação sem fins lucrativos que é dona da marca **OpenMP**, mantém a especificação do **OpenMP** e produz e aprova suas novas versões. A ARB ajuda a organizar e financiar conferências, *workshops* e outros eventos relacionados, a fim de promover o **OpenMP**.

- Hewlett-Packard Company
- Intel Corporation
- International Business Machines (IBM)
- Kuck & Associates, Inc. (KAI)
- Silicon Graphics, Inc.
- Sun Microsystems, Inc.
- U.S. Department of Energy ASCI program

Hoje, a norma **OpenMP** é totalmente funcional e implementada por muitos compiladores comerciais e livres[2] como API para programação paralela em sistemas com memória compartilhada.

A primeira versão da norma (1.0) foi especificada apenas para Fortran. Já no ano seguinte (1998) foram incorporadas diretivas para as linguagens C e C++. Desde então, 8 novas versões foram criadas, trazendo novas diretivas e rotinas, bem como suporte a diferentes tipos de paralelismo, entre eles o paralelismo de tarefas, que foi introduzido na versão 3.0, em 2008. Atualmente, a norma encontra-se na versão 3.1, liberada em Julho de 2011.

3.2 Modelo de execução

Para facilitar o entendimento do restante do texto, abaixo segue uma lista de termos que serão utilizados (definições obtidas em (OPE 2011)):

- **grupo**: Um conjunto de uma ou mais *threads* participantes de uma região `parallel`.
- **região**: Todo o código encontrado durante a execução de uma dada diretiva ou rotina **OpenMP**.
- **região sequencial**: Todo o código encontrado durante a execução de um programa **OpenMP** que não faz parte de uma região paralela, correspondente a uma diretiva `parallel`, ou de uma região de tarefa, correspondente a uma diretiva `task`.
- **região paralela**: Uma região que é executada por um grupo de uma ou mais *threads*.
- **barreira**: Um ponto na execução do programa encontrado por um grupo de *threads*. A partir deste ponto, nenhuma *thread* no grupo pode executar até que todo o grupo tenha atingido a barreira e todas as tarefas explícitas geradas pelo grupo tenham terminado.

OpenMP utiliza o modelo de execução paralela *fork-join* (OPE 2011). Múltiplas unidades de execução realizam tarefas definidas implicitamente ou explicitamente por diretivas **OpenMP**. No caso da implementação utilizada neste trabalho (compilador GCC 4.7), as unidades de execução são *POSIX Threads*[3].

[2] http://openmp.org/wp/openmp-compilers/
[3] https://computing.llnl.gov/ials/pthreads/

Um programa **OpenMP** inicia como uma única unidade de execução, chamada de *thread* inicial. A *thread* inicial executa dentro de uma região sequencial, que é chamada de região da tarefa inicial.

A principal diretiva **OpenMP**, utilizada em todos os programas que fazem uso da API, é a diretiva `parallel`. Ela é responsável por demarcar as regiões do código a serem executadas por múltiplas *threads* (regiões paralelas).

Quando uma *thread* encontra uma diretiva `parallel`, ela cria um grupo incluindo a si mesma e zero ou mais *threads* adicionais (este número é definido pela variável de ambiente `OMP_NUM_THREADS`, ou explicitamente na diretiva `parallel`), tornando-se *master* do grupo. Para cada *thread*, é delegada a execução das instruções contidas na região paralela. No fim da região é criada, implicitamente, uma barreira a partir da qual apenas a *thread master* pode retomar a execução, retornando à região que foi suspensa quando a diretiva `parallel` foi encontrada.

3.2.1 Exemplos de uso

A seguir estão alguns exemplos de uso da norma **OpenMP**. Lembrando que o conteúdo das seções que seguem está longe de contemplar todas as facilidades providas pela norma. O objetivo é mostrar um pouco da sua sintaxe e diretivas e cláusulas geralmente utilizadas.

3.2.1.1 Paralelismo de Laços

A forma mais simples de se paralelizar um programa, utilizando **OpenMP**, é identificar o paralelismo de laços. Para isso, basta que seja utilizada a diretiva `for`, como na figura 3.1.

```
#pragma omp for [cláusula [cláusula ...]]
    for (i=0; i < n; i++) { ... }
```

Figura 3.1: Diretiva `omp for`

Caso não esteja dentro uma região paralela, a diretiva `for` pode ser combinada com a `parallel` para especificar em uma só linha que um grupo de *threads* deverá ser criado para a execução paralela do laço. Além disso, a diretiva aceita o uso de cláusulas adicionais, que influenciam no funcionamento da mesma. No exemplo da figura 3.2, utilizamos a cláusula `schedule`.

```
/* O laço a ser paralelizado deve estar */
/* imediatamente após a declaração da diretiva. */
#pragma omp parallel for schedule(dynamic, 10)
    for (i=0; i < N; i++) {
        c[i] = a[i] + b[i];
    }
```

Figura 3.2: Exemplo de uso da diretiva `omp for`

No início do laço, o grupo de *threads* é criado, com tamanho igual ao valor da variável de ambiente `OMP_NUM_THREADS`. Cada uma das *threads* executa um bloco de iterações do laço. No fim, as *threads* são sincronizadas, e a *master* continua a execução.

A cláusula `schedule` permite que o usuário defina a maneira como as iterações serão agrupadas e distribuídas entre as *threads* do grupo, ou seja, a forma como as iterações serão escalonadas ao longo do grupo de *threads*. No exemplo acima, os argumentos **dynamic** e 10 fazem com que as iterações sejam distribuídas em blocos de tamanho 10, a medida com que são requisitadas pelas *threads*. Cada *thread* executa um bloco de iterações, então requisita outro bloco, até não haver mais blocos a serem distribuídos. Quando nenhum tamanho de bloco é especificado, é assumido o valor 1.

Outros valores possíveis para a cláusula `schedule` são:

- **static**, *tamanho_bloco*: As iterações são divididas em blocos com o tamanho indicado e cada bloco é atribuído a uma *thread* do grupo de forma circular, realizando um escalonamento do tipo *round-robin*.

- **guided**, *tamanho_bloco*: Funciona como o **dynamic**, utilizado no exemplo (distribuição de iterações sob demanda). Porém, o tamanho de cada bloco é proporcional ao número de iterações ainda não distribuídas dividido pelo número de *threads* do grupo, de forma decrescente até chegar ao tamanho *tamanho_bloco*. Além, nenhum bloco pode ter tamanho menor do que o especificado, com exceção do último, que pode conter menos iterações.

A norma não define um tipo de escalonamento padrão a ser utilizado pelo compilador quando a cláusula `schedule` está ausente. Portanto, o mesmo depende da implementação utilizada.

3.2.1.2 Acesso e escopo de variáveis em regiões paralelas

OpenMP provê, como outras APIs para programação paralela, meios para evitar eventuais conflitos de escrita concorrente em regiões de memória. Nas figuras que seguem, estão exemplo de uso de algumas cláusulas com esta finalidade.

```
int t_id = -1;
#pragma omp parallel shared(t_id)
{
    t_id = omp_get_thread_num();
    printf("
      Thread: %d, t_id: %d\n", omp_get_thread_num(), t_id
    );
}
```

Figura 3.3: Exemplo de uso da cláusula `shared`.
A rotina `omp_get_thread_num()` retorna o identificador da *thread* em execução.

A cláusula `shared(t_id)`, da figura 3.3, indica que a região de memória apontada pela variável t_id, declarada fora da região paralela, é compartilhada entre todas as *threads* do grupo. Assim, esta cláusula deve ser utilizada quando sabemos que não haverá escritas concorrentes que causem inconsistência dos dados. O programa em questão apresenta este comportamento errôneo, como podemos ver na saída produzida, exibida na figura 3.4.

```
Thread: 0, t_id: 0
Thread: 2, t_id: 1
Thread: 1, t_id: 1
Thread: 3, t_id: 3
```

Figura 3.4: Saída do programa com cláusula `shared`.
Observe que o valor de t_id impresso na segunda linha é inconsistente com o identificador da *thread*.

Quando for necessário que região paralela possua sua própria versão de uma variável declarada anteriormente, apontando para uma região de memória distinta, utilizamos a cláusula `private`. A figura 3.5 mostra como fica o código.

```
int t_id = -1;
#pragma omp parallel private(t_id)
{
    t_id = omp_get_thread_num();
    printf("
      Thread: %d, t_id: %d\n", omp_get_thread_num(), t_id
    );
}
```

Figura 3.5: Exemplo de uso da cláusula `private`.

Ao utilizarmos a cláusula `private`, a variável dentro da região paralela é inicializada com um valor arbitrário. Portanto, quando for necessário que este valor seja igual ao valor da variável antes da região paralela, deve ser utilizada a cláusula `firstprivate`. Caso utilizássemos esta cláusula no lugar da `private`, na figura 3.5, o valor de t_id seria inicializado como -1 para cada *thread*.

Analogamente, para que o valor da variável depois da região paralela assuma o último valor computado, deve ser utilizada a cláusula `lastprivate`.

Note que, quando uma variável é *shared*, seu valor inicial sempre será igual ao valor antes da região paralela, e o último valor computado também estará na variável depois da região. Pois se trata da mesma região de memória.

Assim, podemos observar que o uso de variáveis *shared* produz menos sobrecusto que variáveis *private*, devido a primeira não necessitar de alocação de novas regiões de memória.

```
Thread: 1, t_id: 1
Thread: 2, t_id: 2
Thread: 0, t_id: 0
Thread: 3, t_id: 3
```

Figura 3.6: Saída do programa com cláusula `private`.
Observe que agora o valor de t_id é consistente com o identificador de cada *thread*.

3.3 Paralelismo de tarefas em OpenMP

A partir da versão 3.0, a diretiva `task` foi incorporada à norma **OpenMP**, oferecendo suporte a paralelismo de tarefas. Desde então, é possível expressar paralelismo em algoritmos irregulares através da norma.

A seguir, estão listados alguns termos que serão utilizados no restante do texto:

- **ambiente de dados**: Todas as variáveis associadas à execução de uma tarefa. O ambiente de dados de uma tarefa é construído a partir do ambiente de dados da tarefa que a gerou, no momento de sua criação.

- **tarefa**: Uma instância específica de código executável e seu ambiente de dados, gerada quando uma *thread* encontra uma diretiva `task`.

- **região de tarefa**: Uma região que consiste em todo o código encontrado durante a execução de uma tarefa.

- **tarefa *tied***: Uma tarefa que, quando suspensa, pode ser retomada apenas pela *thread* que a suspendeu. Tarefas *tied* são geradas por padrão quando utilizamos a diretiva `task`.

- **tarefa *untied***: Uma tarefa que, quando suspensa, pode ser retomada por qualquer *thread* do grupo. Tarefas *untied* são geradas quando utilizamos a cláusula `untied` em conjunto com a diretiva `task`. O uso de tarefas *untied* tem o objetivo de auxiliar o escalonamento de tarefas ao longo das *thread*. No entanto, em casos onde a tarefa faz muitos acessos e operações com variáveis *private*, pode ser mais adequado utilizar tarefas *tied*, pois isso faz com que não seja necessário copiar todo o ambiente de dados de uma tarefa para eventuais novas *threads* que venham a processá-la.

Quando uma *thread* encontra uma diretiva `task`, uma nova tarefa é gerada, e sua execução é atribuída a uma das *threads* do grupo atual. Como a execução da tarefa depende da disponibilidade da *thread* para realizar o trabalho, a execução de uma nova tarefa pode ser imediata ou postergada. *Threads* tem permissão para suspender uma região de tarefa corrente a fim de executar uma tarefa diferente. Porém, é garantido que o término de todas as tarefas de uma região paralela ocorra antes do fim da região, ou seja, a *thread master* só ultrapassa a barreira de fim de região paralela após todas as tarefas terem sido processadas.

A sequência de Fibonacci é um exemplo clássico utilizado em aulas de algoritmos e programação para ensinar recursividade, devido à simplicidade do código necessário para expressar esta equação recorrente. Na figura 3.7, está uma implementação em C de um algoritmo recursivo que calcula o n-ésimo número da sequência, utilizando paralelismo de tarefas **OpenMP**.

Na função `main` é criada a região paralela onde a função `fib` será executada. Porém, como queremos que apenas uma *thread* faça a chamada inicial da função, devemos utilizar a diretiva `single` antes da primeira chamada. A cláusula `nowait` faz com que as demais *threads* do grupo não aguardem o fim da região delimitada pela diretiva `single`, permitindo que as mesmas participem da execução de `fib`, possibilitando o paralelismo das tarefas.

Como podemos observar, cada tarefa irá gerar duas tarefas filhas, para calcular x e y. A diretiva `taskwait` se trata de um ponto de sincronização das tarefas criadas, assim,

```
int main(int argc, char *argv[]) {
    const int n = atoi(argv[1]);
    #pragma omp parallel
    {
        #pragma omp single nowait
            fib(n);
    }
}

int fib(int n) {
    int x, y;
    if (n < 2) return n;
    #pragma omp task untied shared(x)
        x = fib(n - 1);
    #pragma omp task untied shared(y)
        y = fib(n - 2);
    #pragma omp taskwait
    return x + y;
}
```

Figura 3.7: Fibonacci com paralelismo de tarefas **OpenMP**

a instrução "*return x + y;*" será atingida apenas após o término das tarefas filhas, que calcularam os valores de x e y.

4 GRANULARIDADE DE TAREFAS

Neste capítulo serão introduzidos os conceitos de *speedup* e balanceamento de carga no contexto de paralelismo de tarefas, bem como serão descritas causas que levam à necessidade de haver controle de granularidade de tarefas em programas com este tipo de paralelismo.

4.1 Speedup

O *speedup* é uma métrica que representa a razão entre o tempo de execução de um programa paralelo e o tempo de execução de sua versão sequencial. Por isso, trata-se de uma boa medida para avaliarmos quantitativamente a melhoria de desempenho trazida pela versão paralela de um programa em relação à sua versão sequencial. Segue abaixo a fórmula do *speedup*:

$$S_u = T_1/T_u$$

Onde:

- u: É o número de unidades de execução utilizadas pelo programa.
- T_1: É o tempo de execução do programa sequencial.
- T_u: É o tempo de execução da versão paralela executada em um ambiente com u unidades de execução disponíveis.

4.2 Balanceamento de carga

Um fator importante para o desempenho de programas paralelos é o balanceamento da carga de trabalho realizada pelos elementos de processamento ao longo das múltiplas unidades de execução que o programa utiliza. Como em arquiteturas com memória compartilhada o número de elementos de processamento geralmente é menor do que o número de unidades de execução, é necessário que seja feito o escalonamento de tais unidades, a fim de que todas sejam "atendidas" pelo processador.

A mesma relação existe entre o número de tarefas **OpenMP** de um programa com paralelismo de tarefas e o tamanho do grupo de *threads* utilizado pelo mesmo (definido através da variável de ambiente OMP_NUM_THREADS). Como o número de tarefas criadas é maior que o número de *threads*, a API precisa fazer o escalonamento das tarefas para que todas possam ser atribuídas a *threads* e, eventualmente, processadas. Assim, podemos estabelecer uma relação de ordem entre a quantidade de cada elemento:

$$qtd.\ de\ tarefas > qtd.\ de\ threads > qtd.\ de\ elementos\ de\ processamento$$

Podemos dizer que, em geral, elementos de processamento são um recurso escasso em relação ao número de *threads*, que, por sua vez, são um recurso escasso em relação ao número de tarefas criadas.

Neste trabalho não serão feitas análises sobre os possíveis algoritmos de escalonamento de tarefas encontrados em implementações da norma **OpenMP**. Entretanto, pesquisas sobre o assunto podem ser encontradas em outros trabalhos (DUR 2008). Contudo, é importante citar que um escalonamento eficiente (tanto entre *threads* e elementos de processamento quanto entre tarefas e *threads*) deve minimizar o tempo de ociosidade dos recursos escassos, distribuindo-os de forma justa ao longo dos elementos que os necessitam, o que é chamado de balanceamento de carga.

4.3 Sobrecusto e granularidade de tarefas

Como podemos observar, programas com paralelismo de tarefas demandam processamento adicional (sobrecusto resultante do gerenciamento de *threads* e tarefas) em relação as suas versões sequenciais. Este sobrecusto é decorrente do custo imposto ao sistema operacional para gerenciar o grupo de *threads* utilizadas pelo programa, bem como o custo de criação, manutenção e escalonamento das tarefas **OpenMP**. Referente a isso, abaixo seguem os termos que serão utilizados no resto do texto:

- **Sobrecusto de threads**: É o custo imposto ao sistema operacional para gerenciar (criar, escalonar e efetuar trocas de contexto) o grupo de *threads* utilizados pelo programa (cujo tamanho é definido pela variável de ambiente OMP_NUM_THREADS). É inversamente proporcional ao número de elementos de processamento disponíveis, pois quanto mais elementos de processamento o sistema possuir, maior o número de *threads* que poderão ser tratadas simultaneamente. E é diretamente proporcional ao tamanho do grupo de *threads*, pois, quanto mais *threads* existirem, maior será o trabalho do sistema operacional para gerenciá-las.

- **Sobrecusto de tarefas**: Inversamente proporcional à granularidade de tarefas. Ou seja, quanto mais fina a granularidade, maior o sobrecusto, pois maior será o tempo gasto no gerenciamento (criação e escalonamento) das tarefas. Também é inversamente proporcional ao tamanho do grupo de *threads*, pois, quanto mais *threads* disponíveis para processar as tarefas, menos tempo as mesmas ficarão suspensas, esperando uma *thread* atendê-las, ou seja, maior será a "vazão" de tarefas processadas.

- **Granularidade de tarefas**: A granularidade de tarefas é inversamente proporcional ao número de tarefas concorrentes em um dado momento da execução, e diretamente proporcional à complexidade de cada tarefa. Assim, quando temos um número muito grande de tarefas pouco complexas, dizemos que é uma granularidade fina. Quando existe um número pequeno de tarefas complexas, trata-se de uma granularidade grossa. É importante lembrar que uma granularidade muito grossa possui pouco nível de paralelismo, o que pode ocasionar uma subutilização dos elementos de processamento.

O programa que calcula recursivamente o n-ésimo número da sequência de Fibonacci, contido no capítulo 3, é um bom exemplo de algoritmo que apresenta granularidade de tarefas muito fina. Pois tratam-se de tarefas simples (um *if* e uma soma) e de uma árvore de chamadas recursivas que cresce geometricamente, como ilustrado na figura 4.1.

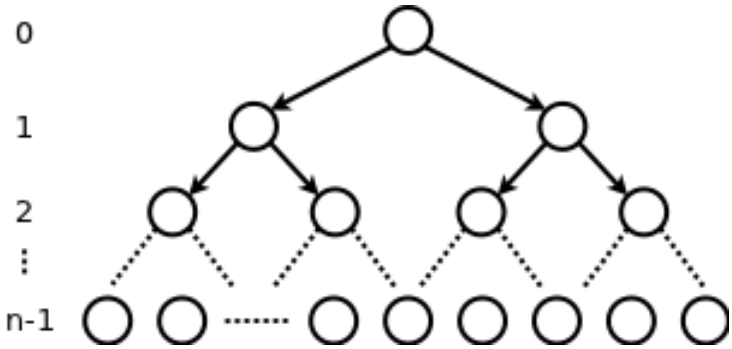

Figura 4.1: Ilustração de uma árvore de chamadas recursivas.
Neste caso, cada nível i da árvore possui 2^i nodos, ou seja, a cada nível da recursão, 2^i tarefas são criadas.

Pelas descrições acima, é possível perceber que existe um compromisso a ser feito entre **sobrecusto de threads** e **sobrecusto de tarefas** (que, por sua vez, está diretamente relacionado à granularidade de tarefas). Quanto mais fina a granularidade de tarefas, maior o nível de paralelismo, porém, maior o **sobrecusto de tarefas**. Ademais, quanto maior o tamanho do grupo de *threads*, menor o **sobrecusto de tarefas**, porém, maior o **sobrecusto de threads**, que varia de acordo com o número de elementos de processamento da arquitetura.

Neste ponto, os itens que podemos controlar programaticamente, com **OpenMP**, são o tamanho do grupo de *threads* e a granularidade máxima de tarefas criadas em tempo de execução.

O controle do tamanho do grupo de *threads* deve levar em consideração o número de elementos de processamento disponíveis no sistema. Caso o **sobrecusto de threads** supere o benefício de tê-las, é necessário diminuir o tamanho do grupo.

O controle de granularidade de tarefas se apresenta como uma maneira de evitar que o *speedup* seja degradado em programas com paralelismo irregular. Pois, um controle eficiente evita uma subutilização do nível de paralelismo (granularidade muito grossa), e não permite um alto nível de **sobrecusto de tarefas** (granularidade muito fina).

5 CONTROLE DE GRANULARIDADE DE TAREFAS EM OPENMP

Neste capítulo serão exibidas as estratégias de controle de granularidade em **OpenMP** que serão avaliadas neste trabalho. Começando por uma seção que explica a ideia central por trás das estratégias, passando pelas cláusulas **OpenMP** que serão utilizadas para controlar a granularidade de tarefas e, por fim, descrevendo o funcionamento das estratégias.

5.1 O parâmetro *maxdepth*

As estratégias de controle de granularidade de tarefas utilizadas neste trabalho possuem em comum a ideia de serializar a execução das tarefas quando a granularidade se tornar muito fina. Para isso, é definido um limiar a partir do qual a criação de novas tarefas concorrentes deve ser evitada. A partir deste limiar, toda a vez que uma *thread* encontra uma diretiva `task`, a tarefa deverá ser executada imediatamente, como em um programa sequencial. Assim, o **sobrecusto de tarefas** é reduzido, pois o grupo de *threads* pode se dedicar a dar vazão às tarefas já criadas, em vez de perder ciclos de processamento para o gerenciamento de novas tarefas **OpenMP**.

Este limiar é definido pelo usuário, indicando qual a profundidade máxima da árvore de chamadas recursivas em que novas tarefas devem ser criadas. A partir de agora, será utilizado o termo *maxdepth* para este parâmetro. Quando a profundidade da árvore de chamadas recursivas for maior ou igual a *maxdepth*, a granularidade é muito fina para que o desempenho seja beneficiado pelo paralelismo de tarefas.

Como se pode concluir, uma escolha ótima de *maxdepth* coincide com o ponto de maior *speedup* do programa, dado a estratégia de controle utilizada e o tamanho do grupo de *threads*. Além, isto também significa que limite para o **sobrecusto de tarefas** foi atingido, evitando que o mesmo prejudique o *speedup*.

5.2 Cláusulas auxiliares para a diretiva `task`

Além da diretiva `task`, a norma **OpenMP** provê cláusulas adicionais que podem afetar a maneira como novas tarefas são criadas e escalonadas. Estas cláusulas recebem, como argumento, uma expressão booleana arbitrária, definida pelo usuário. Toda vez que uma *thread* encontra uma diretiva `task` acompanhada por uma destas cláusulas, sua expressão é avaliada. O valor da expressão irá definir o tipo da tarefa criada. As seções que seguem mostram estes tipos e as cláusulas associadas.

5.2.1 Tarefa *undeferred*

Uma tarefa *undeferred* é uma tarefa para qual a execução não é postergada em relação a região geradora. Ou seja, a região geradora é suspensa até que a execução da tarefa *undeferred* termine. Assim, a tarefa ainda é criada, porém sua execução acontece imediatamente, diminuindo o **sobrecusto de tarefas**, pois uma nova tarefa será criada por esta *thread* apenas após o término da tarefa *undeferred*.

Este tipo de tarefa é criada quando utilizamos a cláusula `if` e sua expressão é avaliada como *false*. A figura 5.1 mostra um exemplo de uso.

```
int fib(int n) {
    int x, y;
    if (n < 2) return n;
    #pragma omp task untied if(n > 10) shared(x, n)
        x = fib(n - 1);
    #pragma omp task untied if(n > 10) shared(y, n)
        y = fib(n - 2);
    #pragma omp taskwait
    return x + y;
}
```

Figura 5.1: Exemplo de uso da cláusula `if`

A semântica da cláusula `if` nos diz, basicamente, "se esta expressão for falsa, crie uma tarefa *undeferred*, caso contrário, crie uma tarefa normal (postergável)".

No caso da figura 5.1, quando n for menor ou igual a 10, tarefas *undeferred* serão criadas, serializando a execução. Porém ainda haverá sobrecusto, pois tarefas *undeferred* também possuem ambiente de dados próprio. Além disso, há o sobrecusto decorrente do tratamento da diretiva e da avaliação da expressão booleana.

5.2.2 Tarefa *final*

Uma tarefa *final* força todas as suas tarefas filhas a também serem *final* e incluídas sequencialmente na região geradora, sendo executadas imediatamente pela *thread* que as encontra.

Este tipo de tarefa é criado quando utilizamos a cláusula `final` e sua expressão é avaliada como *true*. A figura 5.2 mostra um exemplo de uso.

Note que esta expressão possui semântica contrária a da cláusula `if`. Na cláusula `if`, enquanto a expressão é **verdadeira**, tarefas normais são criadas, e quando a expressão é **falsa**, tarefas *undeferred* são geradas. Já na `final`, enquanto a expressão é **falsa**, tarefas normais são criadas, e quando a expressão é **verdadeira**, tarefas *final* são geradas.

Tarefas *final* são semelhantes a tarefas *undeferred*. Ambas fazem com que a tarefa seja imediatamente executada. No entanto, tarefas *final* afetam todas as tarefas filhas, de maneira que todas também se tornam *final*. Assim, não há sobrecusto decorrente da avaliação da expressão booleana em tarefas que já estão em uma região *final*. Além disso, é possível verificar se o programa está em uma região *final*, através da rotina `omp_in_final()`. Possibilitando otimizações no código.

```
int fib(int n) {
    int x, y;
    if (n < 2) return n;
    #pragma omp task untied final(n <= 10) shared(x, n)
        x = fib(n - 1);
    #pragma omp task untied final(n <= 10) shared(y, n)
        y = fib(n - 2);
    #pragma omp taskwait
    return x + y;
}
```

Figura 5.2: Exemplo de uso da cláusula `final`

5.2.3 Tarefa *merged*

Uma tarefa *merged* é uma tarefa cujo ambiente de dados é o mesmo da região geradora. Ou seja, não é necessária a criação de um ambiente de dados próprio para a tarefa. O que, naturalmente, diminui drasticamente o sobrecusto de tarefas.

Apenas tarefas *undeferred* e *final* podem se tornar tarefas *merged*. Para que isto aconteça, é necessário utilizar a cláusula `mergeable` em conjunto com as diretivas `if` ou `final`. Como mostrado na 5.3.

```
int fib(int n) {
    int x, y;
    if (n < 2) return n;
    #pragma omp task untied final(n <= 10) shared(x, n) mergeable
        x = fib(n - 1);
    #pragma omp task untied final(n <= 10) shared(y, n) mergeable
        y = fib(n - 2);
    #pragma omp taskwait
    return x + y;
}
```

Figura 5.3: Exemplo de uso da cláusula `mergeable`

A cláusula `mergeable` deve ser utilizada com cautela, pois existem casos onde o uso da mesma pode produzir resultados indesejados. Por exemplo, quando a tarefa faz operações sobre variáveis *private* cujos identificadores também são utilizados após a tarefa. Nesse caso, o programa apresentará um comportamento para tarefas normais, e outro para tarefas *merged*. Já que tarefas *merged* causarão alterações nas mesmas regiões de memória apontadas pelas variáveis que estão fora da região da tarefa.

5.3 Estratégias de controle de granularidade de tarefas

Como mencionado anteriormente, as estratégias de controle de granularidade de tarefas que serão propostas neste trabalho possuem em comum a ideia de serializar a execução das tarefas quando a granularidade se tornar muito fina. Para isso, utilizamos o parâmetro

maxdepth para limitar a profundidade máxima da árvore de criação de tarefas concorrentes. Já vimos que as cláusulas `if` e `final` podem nos ajudar a realizar este controle, dado que seja criada uma variável para armazenar a profundidade atual, a ser comparada com *maxdepth* na expressão booleana das cláusulas.

Porém, este controle também pode ser realizado manualmente pelo usuário, sem uso de cláusulas adicionais. Chamaremos esta estratégia de "Controle manual".

Assim, a seguir são apresentadas as 4 estratégias propostas, "Controle manual", "Cláusula `if`", "Cláusula `final`" e "Cláusula `final` + API". Esta última é uma estratégia que utiliza a cláusula `final` em conjunto com a rotina `omp_in_final()`, mencionada na seção 5.2.2.

5.3.1 Controle manual

Esta estratégia, como o nome sugere, não utiliza cláusulas auxiliares da norma. Trata-se de um controle feito pelo usuário diretamente com funções da linguagem de programação. Por isso, podemos dizer que é uma estratégia de controle de granularidade que não se baseia puramente em diretivas e rotinas da API.

Nesta estratégia, o usuário deve modificar o código fonte, definindo um caminho sequencial e um paralelo. A figura 5.4 mostra um exemplo que implementa o algoritmo *NQueens* (algumas linhas de código estão sendo omitidas):

```
void nqueens (char *a, int n, int pos, int depth) {
  ...
  for (i = 0; i < n; i++) {
    ...
    if (pos == n - 1)
      ++cnt;
    else {
      b[pos] = i;
      if (depth < max_depth) {
        #pragma omp task untied
          nqueens(b, n, pos + 1, depth + 1);
      } else {
        nqueens(b, n, pos + 1, depth + 1);
      }
    }
  }
}
```

Figura 5.4: Controle manual de granularidade de tarefas

Note que, no caminho sequencial, a diretiva **OpenMP** está ausente. Assim o código que seria executado por uma tarefa (chamada recursiva para `nqueens`), é executado sequencialmente pelo programa.

Esta estratégia apresenta o menor **sobrecusto de tarefas**, pois o controle é realizado diretamente pelo usuário. Não há sobrecusto decorrente da execução de cláusulas adicionais, bem como, no caminho sequencial, há a ausência completa de diretivas **OpenMP**.

5.3.2 Cláusula if

Para que a granularidade de tarefas seja controlada pela cláusula `if`, a expressão booleana passada como argumento deve retornar *false* quando a profundidade da árvore de chamadas recursivas é maior ou igual a *maxdepth*. A figura 5.5, mostra um exemplo, baseado no algoritmo *NQueens*.

```
void nqueens (char *a, int n, int pos, int depth) {
  ...
  for (i = 0; i < n; i++) {
    ...
    if (pos == n - 1)
      ++cnt;
    else {
      b[pos] = i;
      #pragma omp task untied if (depth < maxdepth) mergeable
        nqueens (b, n, pos + 1, depth + 1);
    }
  }
}
```

Figura 5.5: Controle de granularidade de tarefas com cláusula `if`

Nesta estratégia, quando `depth` for maior ou igual a `maxdepth`, tarefas *undeferred* serão geradas. Diminuindo o **sobrecusto de tarefas**.

5.3.3 Cláusula final

Para controlarmos a granularidade de tarefas utilizando a cláusula `final`, a expressão booleana passada como argumento deve retornar *true* quando a profundidade da árvore de chamadas recursivas é maior ou igual a *maxdepth*.

A figura 5.6 contém um exemplo de uso, em uma função que implementa o algoritmo *NQueens*.

O número de linhas de código é idêntico ao da versão com cláusula `if`, porém a expressão da cláusula teve sua operação invertida.

Nesta estratégia, quando `depth` for menor que `maxdepth` as tarefas geradas e suas filhas serão *final*. Evitando parte do **sobrecusto de tarefas** e do sobrecusto gerado pela avaliação das expressões booleanas das tarefas filhas, pois as mesmas já serão *final*.

5.3.4 Cláusula final + API

Esta estratégia envolve o uso da cláusula `final` em conjunto com uma rotina de tempo de execução da norma, chamada `omp_in_final()`. Esta rotina retorna *true* quando estiver sendo chamada dentro de uma região de tarefa *final*, e *false*, caso contrário.

Com o uso desta rotina, é possível definir um caminho sequencial que evita qualquer tipo de sobrecusto adicional por parte da API. Assim, a partir do momento em que a cláusula `final` tem sua expressão avaliada como verdadeira, as próximas chamadas recursivas utilizarão o caminho definido pela rotina `omp_in_final()`.

Neste caso, quando a tarefa que estiver executando for uma tarefa *final*, a rotina `omp_in_final()` retornará *true*. Assim uma nova chamada de *nqueens* é feita sem

```
void nqueens (char *a, int n, int pos, int depth) {
  ...
  for (i = 0; i < n; i++) {
    ...
    if (pos == n - 1)
      ++cnt;
    else {
      b[pos] = i;
      #pragma omp task untied final (depth >= maxdepth) mergeable
        nqueens (b, n, pos + 1, depth + 1);
    }
  }
}
```

Figura 5.6: Controle de granularidade de tarefas com cláusula `final`

```
void nqueens (char *a, int n, int pos, int depth) {
  ...
  for (i = 0; i < n; i++) {
    ...
    if (pos == n - 1)
      ++cnt;
    else {
      b[pos] = i;
      if (omp_in_final()) {
        nqueens (b, n, pos + 1, depth + 1);
      } else {
        #pragma omp task untied final (depth >= max_depth) mergeable
          nqueens (b, n, pos + 1, depth + 1);
      }
    }
  }
}
```

Figura 5.7: Controle de granularidade de tarefas com cláusula `final` + API

que a diretiva `task` seja executada novamente. Eliminando o sobrecusto decorrente desta operação.

O tamanho do código necessário para utilizar esta estratégia é equivalente ao do código utilizado para o controle manual de granularidade. No entanto, esta estratégia utiliza apenas cláusulas e rotinas **OpenMP**.

5.4 Conclusão

Podemos esperar que as estratégias que definem explicitamente caminhos de execução diferentes quando a árvore de chamadas recursivas atinge *maxdepth* apresentarão maior *speedup*, já que o **sobrecusto de tarefas** no caminho sequencial é nulo. As demais estratégias (cláusula `if` e cláusula `final`) apresentam abordagens com apenas um caminho de execução, onde o controle de granularidade é feito totalmente pela API. Assim, podemos esperar que as estratégias produzam *speedups* distintos, na seguinte ordem crescente: cláusula `if` (tarefas *undeferred*), cláusula `final` (tarefas *final*), cláusula `final` + API (tarefas *final* e caminho sequencial definido por rotina da API) e controle manual (caminho sequencial definido através de um `if` da linguagem de programação).

6 AVALIAÇÃO DAS ESTRATÉGIAS PROPOSTAS

Este capítulo contém um relatório de experimentos feitos com as estratégias de controle de granularidade de tarefas **OpenMP** mostradas no capítulo 5. O objetivo é analisar o *speedup* proporcionado por tais abordagens em diferentes cenários. A seção 6.1 descreve como os experimentos foram realizados. Na seção 6.2 estão os resultados obtidos.

6.1 Metodologia de execução dos experimentos

6.1.1 Amostragem

A fim de calcular o *speedup* de cada estratégia, os experimentos consistem em amostras de 30 execuções, com os parâmetros desejados (*maxdepth* e parâmetro de entrada do programa). A cada execução, foi coletado o tempo decorrido, e com ele obtida a média aritmética dos tempos e o seu desvio padrão. Com isso, o *speedup* desta média em relação à média dos tempos de execução da versão sequencial foi calculado.

Para determinar o intervalo de confiança do tempo de execução de cada amostra, foi calculado seu erro de estimação. O erro de estimação representa metade da amplitude do intervalo de confiança, e é obtido através da seguinte relação:

$$e = z(\sigma/\sqrt{n})$$

Onde:

- σ: É o desvio padrão da amostra.
- n: É o número de elementos da amostra.
- z: É a confiança desejada.

Com o erro de estimação dos tempos de execução, o erro de estimação do *speedup* médio de cada amostra foi obtido, através da seguinte equação:

$$e_{speedup} = (speedup_{max} - speedup_{min})/2$$

Onde $speedup_{max}$ é o *speedup* máximo de cada amostra, definido pela equação:

$$speedup_{max} = (t_{seq} + e_{seq})/(t_{amostra} - e_{amostra})$$

Onde:

- t_{seq}: É o tempo de execução médio da versão sequencial do programa.

- e_{seq}: É o erro de estimação do tempo de execução da versão sequencial do programa.
- $t_{amostra}$: É o tempo de excução médio da amostra.
- $e_{amostra}$: É o erro de estimação do tempo de execução da amostra.

Analogamente, o $speedup_{min}$ foi obtido através da equação:

$$speedup_{min} = (t_{seq} - e_{seq})/(t_{amostra} + e_{amostra})$$

6.1.2 Algoritmos testados

Os experimentos foram baseados em 3 algoritmos: Quicksort, NQueens e Fibonacci. Para cada algoritmo foram escritos quatro programas utilizando paralelismo de tarefas, um para cada estratégia, e um programa sequencial. Além disso, também foram feitos testes com uma versão com paralelismo de tarefas que não faz controle de granularidade.

Para fins de praticidade, utilizaremos a nomenclatura *algoritmo-estrategia* para identificá-los. Como segue, por exemplo, para os programas que implementam o algoritmo *NQueens*:

- *nqueens-nocontrol*: Programa NQueens com paralelismo de tarefas, sem controle de granularidade.
- *nqueens-if*: Programa NQueens com paralelismo de tarefas, utilizando a cláusula `if`.
- *nqueens-final*: Programa NQueens com paralelismo de tarefas, utilizando a cláusula `final`.
- *nqueens-finalapi*: Programa NQueens com paralelismo de tarefas, utilizando cláusula `final` + API.
- *nqueens-manual*: Programa NQueens com paralelismo de tarefas, utilizando controle manual.

6.1.3 Plataforma experimental

Os experimentos foram realizados em um nodo da plataforma francesa Grid'5000[1], localizado no site Porto Alegre, Brasil. O nodo possui dois processadores Intel ©Xeon E5310 Quad Core 1.60 GHz, totalizando 8 elementos de processamento, e 16 GB de memória RAM. Todos os códigos foram compilados com o compilador GCC versão 4.7 (*snapshot* 20110917), sem *flags* adicionais além da necessária para o uso da API **OpenMP** (`-fopenmp`). Esta versão *snapshot* foi utilizada porque as versões do GCC anteriores a 4.7 não implementam a última versão da norma **OpenMP** (3.1) e, portanto, não possuem suporte à cláusula *final*. Até a data em que este trabalho foi escrito, não havia sido lançada uma versão final do GCC versão 4.7.

Os experimentos foram realizados com grupos de *threads* de tamanho 2, 4, 8, 16 e 32. Além disso, para analisarmos o efeito de diferentes escolhas de *maxdepth*, foram coletadas amostras pra diferentes valores deste parâmetro, escolhidos de acordo com a característica da árvore de chamadas de cada algoritmo.

Nas seções que seguem, o símbolo N é utilizado para denotar o parâmetro de entrada do programa, enquanto o símbolo n é o argumento de entrada de uma tarefa.

[1] http://www.grid5000.fr/

6.1.3.1 *Quicksort*

O Quicksort é um algoritmo recursivo de ordenamento que utiliza divisão e conquista para ordenar uma *array* de tamanho N. No caso médio, a altura da árvore de chamadas do Quicksort é $log_2(N)$, e cada nível i contém 2^i tarefas concorrentes. Cada tarefa consiste em escolher um elemento do *array* ao acaso, chamado de pivô, criar duas partições do *array*, uma com elementos menores que o pivô e outra com os demais elementos (maiores ou iguais ao pivô) e criar duas novas tarefas, uma para cada partição criada. Quando o *array* de entrada possuir tamanho 1, a recursão termina

A entrada utilizada para os testes com o Quicksort foi um *array* de tamanho 100000000, populado com números inteiros, escolhidos aleatoriamente. Usando as estatísticas do caso médio, a árvore de chamadas recursivas terá altura $log_2(100000000)$, o que nos dá, arredondando para baixo, 26. Assim, foram executados testes com os seguintes valores de *maxdepth*: 10, 15, 20 e 25.

6.1.3.2 *NQueens*

NQueens é o nome dado a um problema computacional que consiste em descobrir quantas maneiras existem de distribuir N rainhas de xadrez ao longo de um tabuleiro com dimensões NxN, de forma que nenhuma rainha esteja ameaçando as outras $N - 1$.

O código do programa NQueens utilizado para os experimentos foi retirado do diretório de testes da *libgomp*[2]. GNU libgomp é a implementação da norma **OpenMP** utilizada pelo compilador GCC. Seu código fonte é parte integrante do pacote que contém o código do compilador.

Esta implementação utiliza *backtracking* para resolver o problema. Com esta estratégia, cada nível da árvore de chamadas corresponde a uma das N linhas do tabuleiro sendo tratada. Cada tarefa consiste em gerar as possíveis configurações do tabuleiro com uma rainha na linha atual e passar essas configurações para as próximas tarefas, que farão o mesmo para a próxima linha do tabuleiro. O crescimento dos galhos da árvore termina quando todas as rainhas foram inseridas (o que significa que uma solução foi encontrada), ou quando é detectada uma situação onde não é possível posicionar uma nova rainha sem que ela ameace as demais. Assim, a altura máxima da árvore será N, atingida pelos nodos que chegam a uma configuração de tabuleiro com todas as rainhas posicionadas.

A entrada utilizada para os testes com NQueens foi 14, o que significa que o programa contará o número de maneiras possíveis de colocar 14 rainhas em um tabuleiro com 14 linhas e 14 colunas, de forma que nenhuma rainha esteja ameaçando as outras. Os valores de *maxdepth* testados foram: 3, 7, 11 e 13.

6.1.3.3 *Fibonacci*

O programa calcula o N-ésimo número da sequência, de modo que cada tarefa constitui-se em criar duas novas tarefas com argumentos $n - 1$ e $n - 2$ e retornar a soma dos resultados. A recursão termina quando $n < 2$.

A árvore de chamadas terá altura $N - 2$, com 2^i tarefas concorrentes em cada nível i. Os testes foram realizados com entrada 40, e os valores de *maxdepth* testados foram: 5, 10 e 15.

Vale notar que este é o algoritmo cujas tarefas possuem a menor complexidade (trata-se de apenas uma soma de inteiros) dentre os experimentados. Assim, é o caso em que o sobrecusto da criação de tarefas **OpenMP** é mais significativo.

[2]http://gcc.gnu.org/onlinedocs/libgomp/

6.2 Resultados

A seguir estão os gráficos com as curvas de *speedup* obtidas durante os experimentos. Para cada algoritmo, há uma seção que exibe as curvas em função da variação no parâmetro *maxdepth*, e uma seção com a curva em função do tamanho do grupo de *threads* utilizado.

6.2.1 Quicksort

6.2.1.1 Profundidade máxima

As figuras 6.1 a 6.5 mostram os resultados dos experimentos com o algoritmo Quicksort utilizando as quatro estratégias de controle de granularidade de tarefas, além da versão que não faz controle de granularidade.

Nos casos com 2 e 4 *threads*, as estratégias que utilizam as cláusulas `if` e `final` não apresentaram *speedup* maior do que a versão sem controle de granularidade. Ou seja, para um número pequeno de *threads*, o sobrecusto gerado pelo processamento necessário para o uso das cláusulas foi maior do que o **sobrecusto de tarefas** gerado pela granularidade fina. Já a partir de 8 *threads*, fica evidente que, sem controle de granularidade de tarefas, o *speedup* é severamente prejudicado, em relação às versões com controle.

As versões que utilizam cláusula `final` + API e controle manual são amplamente superiores em relação às demais. O que indica que o sobrecusto evitado pelo caminho sequencial é significativo em granularidades finas.

Outro ponto a ser comentado é a influência do valor de *maxdepth* sobre os *speedups*. É possível observar que a curva de *speedup* é crescente até o ponto $maxdepth = 20$, quando ocorre uma inflexão da curva. Isso sugere que, a partir deste ponto, a granularidade se torna muito fina, diminuindo o benefício do paralelismo de tarefas.

6.2.1.2 Número de threads

A figura 6.6 mostra a curva de *speedup* de cada estratégia, em função do tamanho do grupo de *threads* utilizado, com valor de *maxdepth* fixado em 20.

Nota-se que, em todos os casos, o ponto de inflexão da curva de *speedup* acontece quando temos um grupo de 8 *threads*. Devido a carga de trabalho exigida pelas tarefas do Quicksort, o **sobrecusto de threads** passa a prejudicar o desempenho a partir do momento em que é utilizado um tamanho de grupo de *threads* maior que a quantidade de elementos de processamento do sistema.

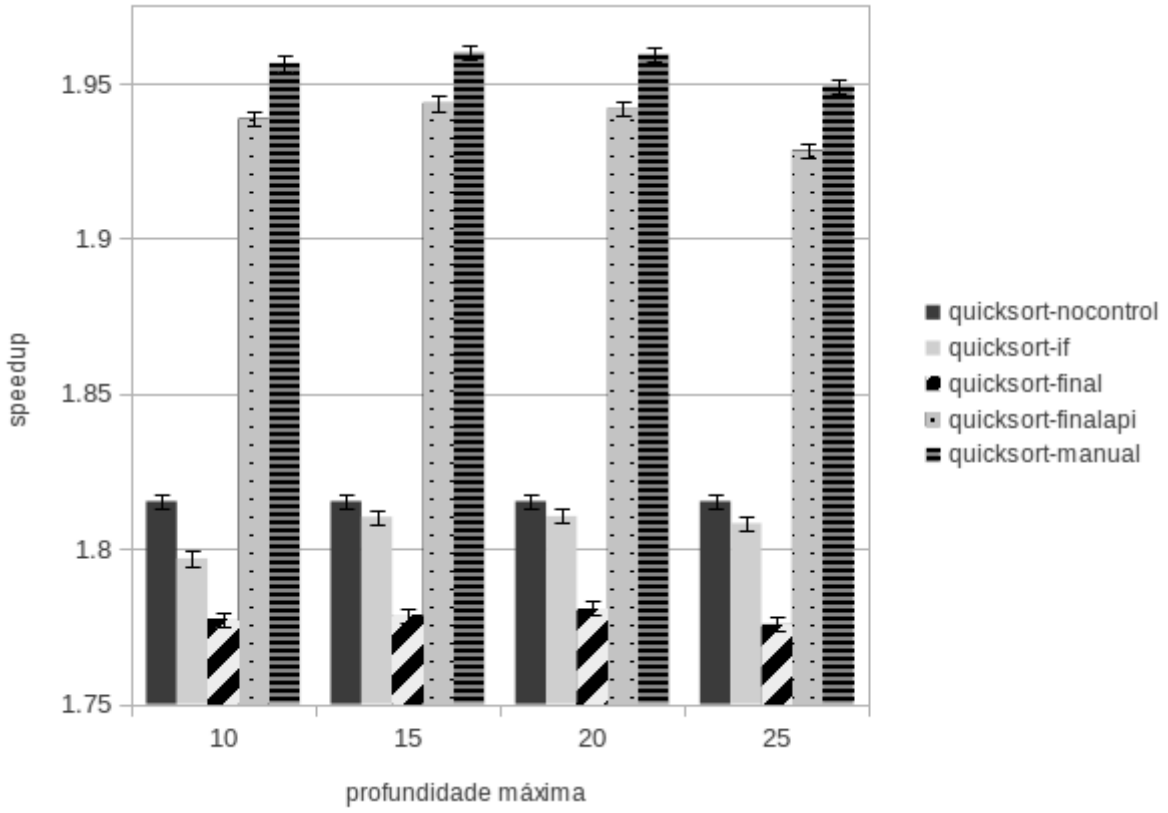

Figura 6.1: Quicksort: speedup de diferentes profundidades máximas com 2 *threads*

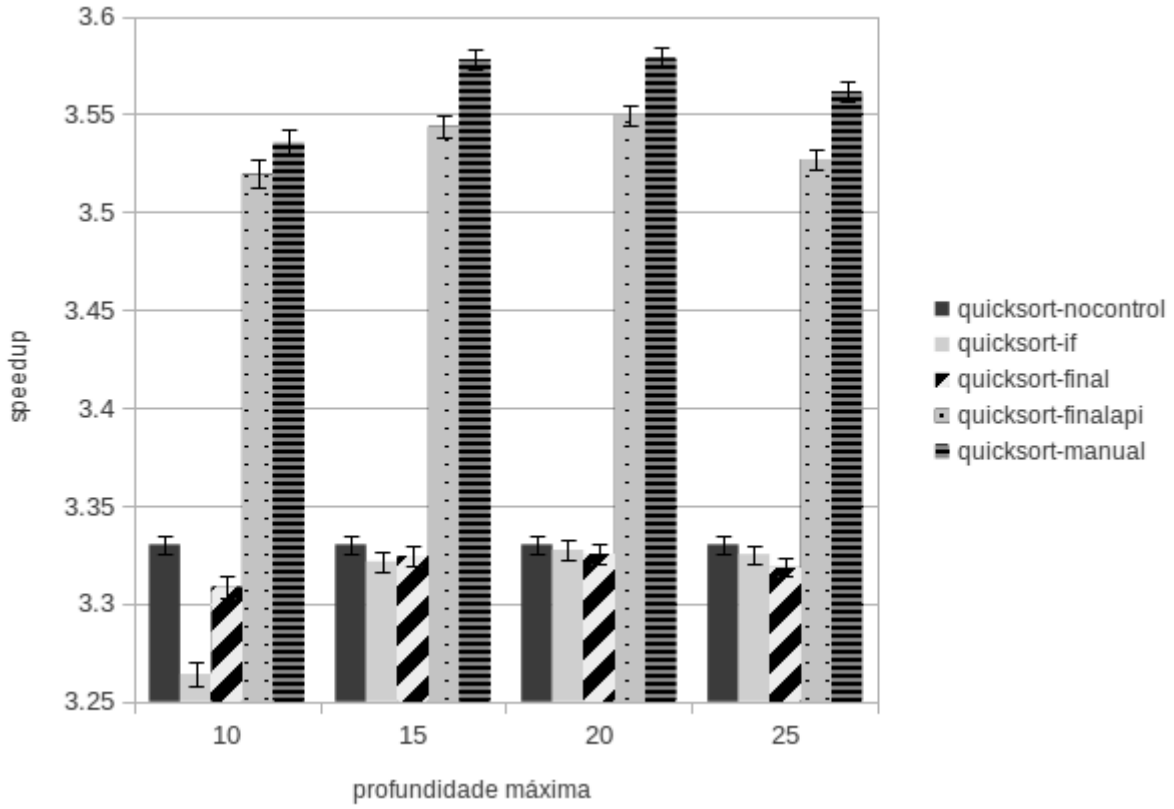

Figura 6.2: Quicksort: speedup de diferentes profundidades máximas com 4 *threads*

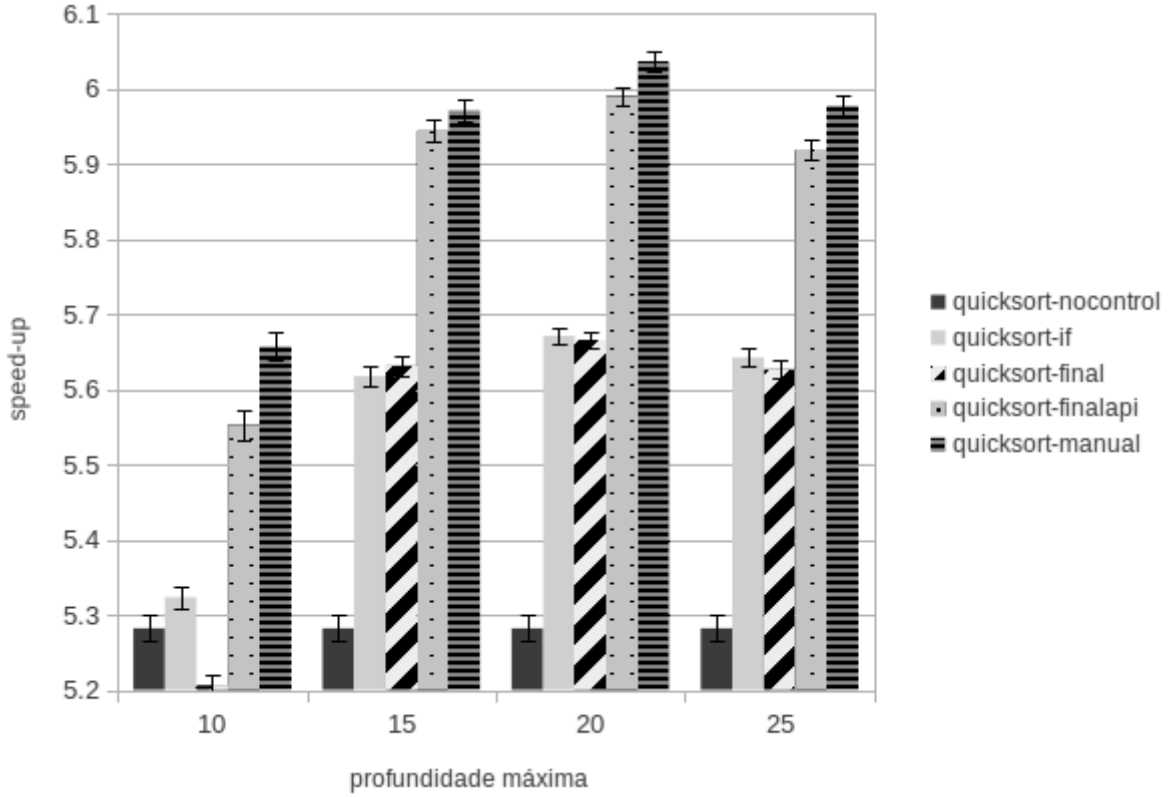

Figura 6.3: Quicksort: speedup de diferentes profundidades máximas com 8 *threads*

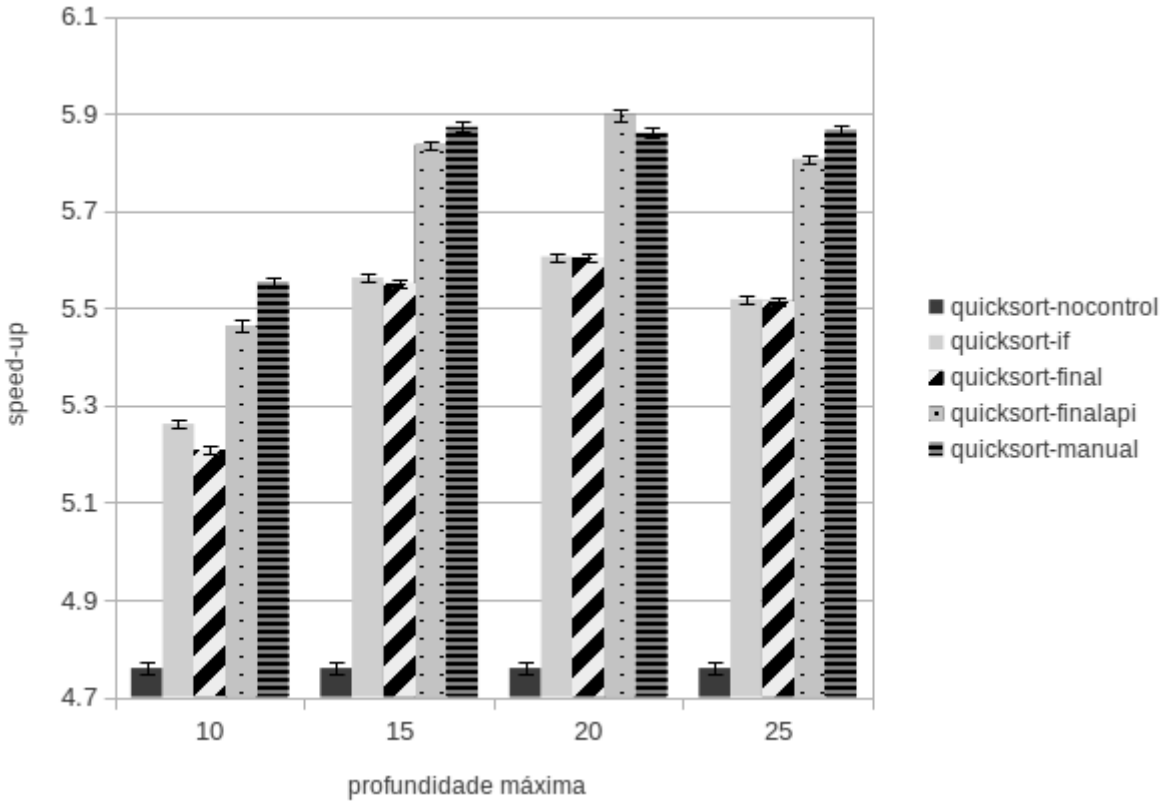

Figura 6.4: Quicksort: speedup de diferentes profundidades máximas com 16 *threads*

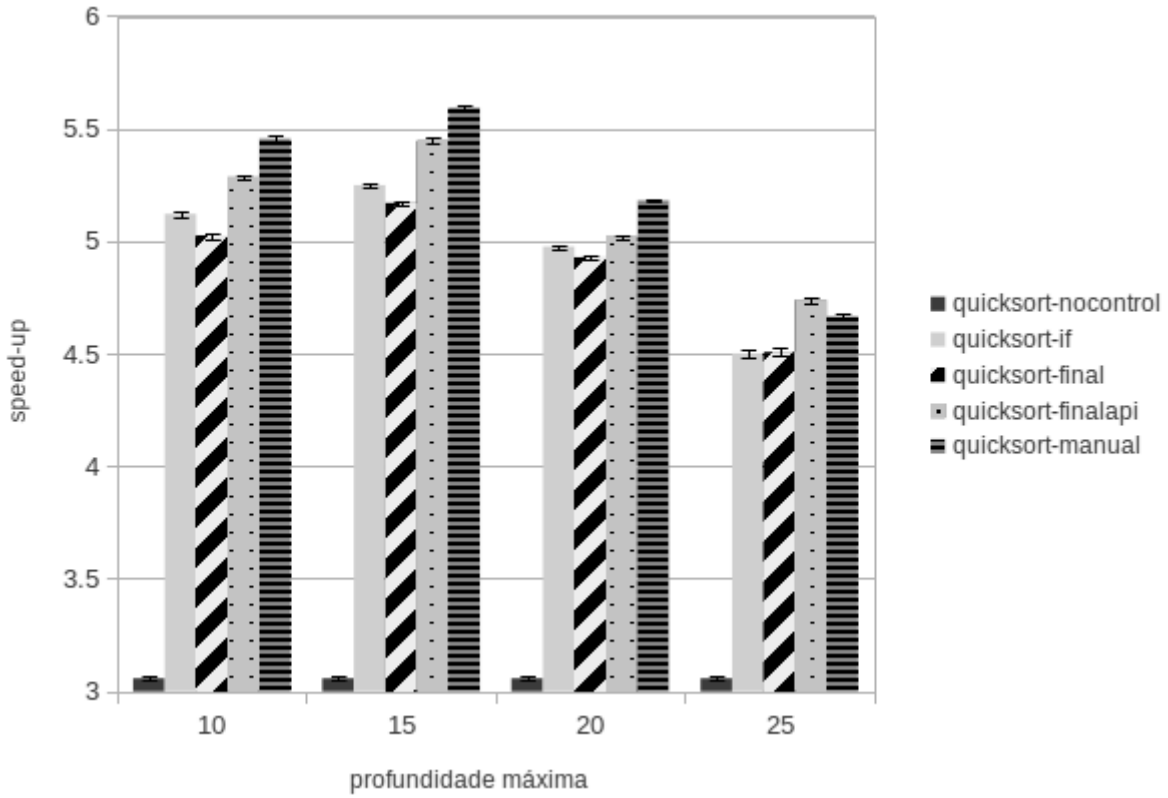

Figura 6.5: Quicksort: speedup de diferentes profundidades máximas com 32 *threads*

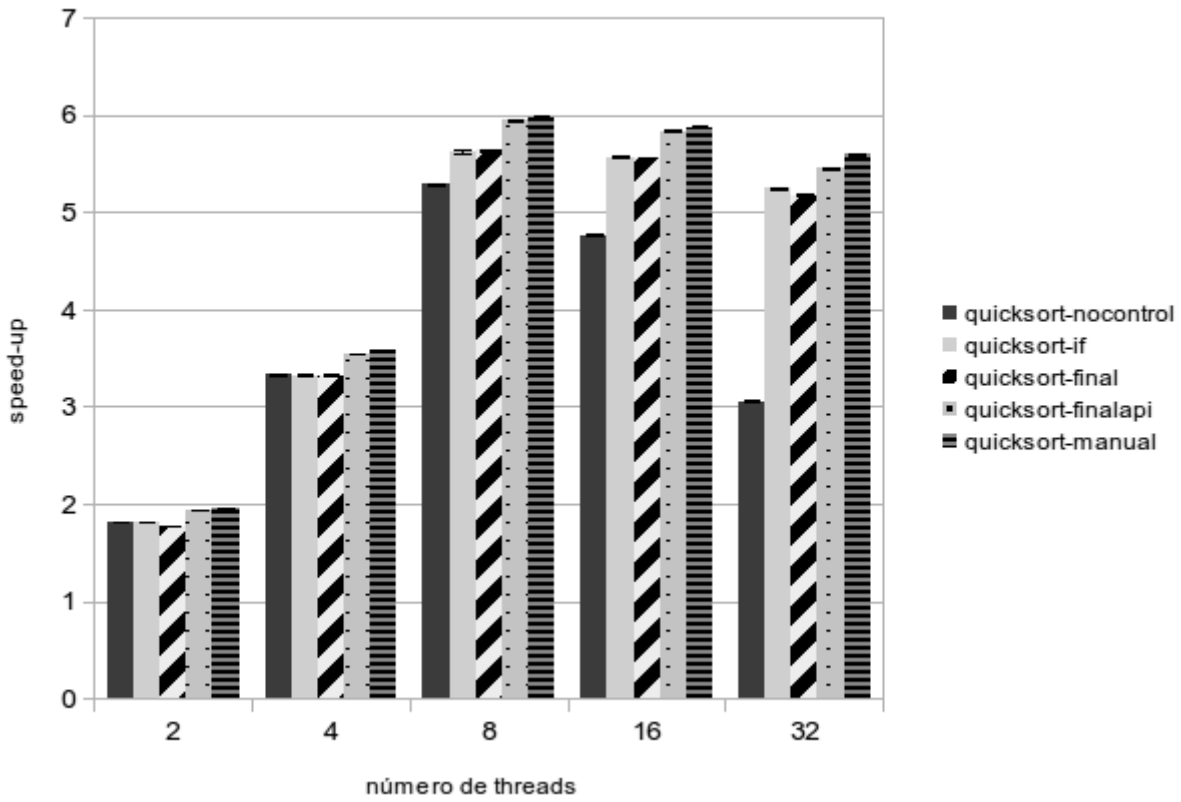

Figura 6.6: Quicksort: *speedup* em diferentes números de *threads*

6.2.2 NQueens

6.2.2.1 Profundidade máxima

As figuras 6.7 a 6.11 mostram os resultados dos experimentos com o algoritmo NQueens utilizando as quatro estratégias de controle de granularidade de tarefas, além da versão que não faz controle de granularidade.

Como esta implementação de NQueens faz *backtracking*, sua árvore de chamadas recursivas possui diferenças estruturais em relação às árvores do Quicksort e do Fibonacci. Trata-se de uma árvore que apresenta muitos nodos logo nos primeiros níveis da recursão. No caso de $N = 14$, no segundo nível da árvore já teremos 156 tarefas, contra 4 em uma árvore binária.

Com isso, verificamos que o controle de granularidade é muito efetivo para o NQueens com paralelismo de tarefas. Ademais, notamos que para valores muito altos de *maxdepth* (próximos da profundidade máxima), o *speedup* é comprometido pelo **sobrecusto de tarefas**.

Apesar de os programas que utilizam cláusula `if` e cláusula `final` terem atingido valores satisfatórios de *speedup*, as versões com cláusula `final` + API e controle manual ainda são superiores, tendo alcançado *speedup* superlinear (*speedup* com valor maior que o número de elementos de processamento).

O motivo pelo qual ocorre *speedup* superlinear, neste caso, é o fato da árvore de chamadas resultante do *backtracking* apresentar uma distribuição de soluções não uniforme (RAO 93).

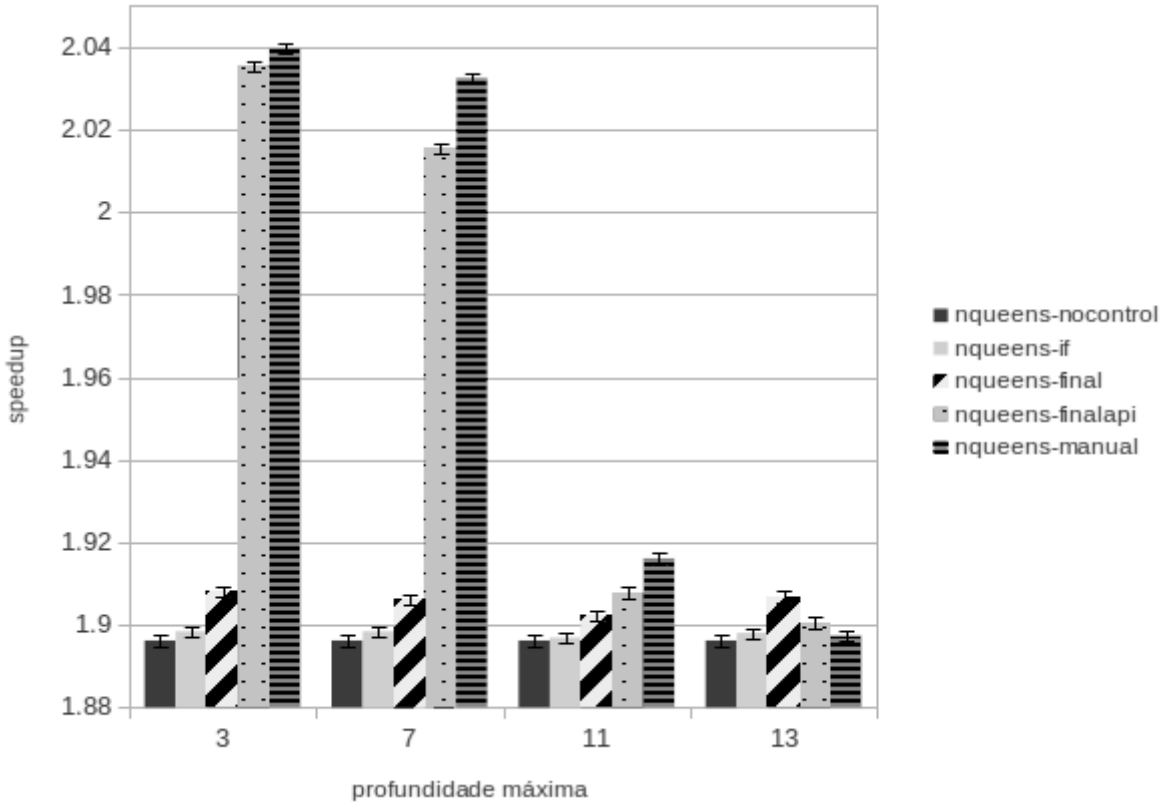

Figura 6.7: NQueens: speedup de diferentes profundidades máximas com 2 *threads*

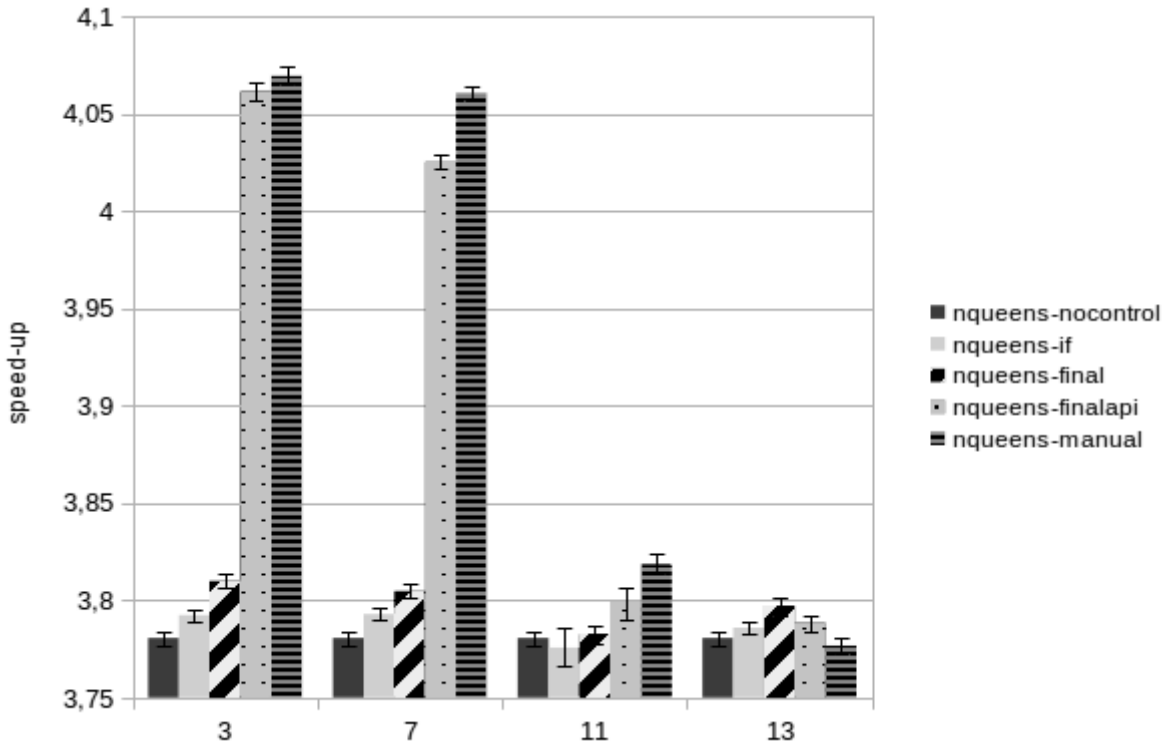

Figura 6.8: NQueens: speedup de diferentes profundidades máximas com 4 *threads*

6.2.2.2 Número de threads

A figura 6.12 mostra a curva de *speedup* de cada estratégia em função do tamanho do grupo de *threads* utilizado, com valor de *maxdepth* fixado em 3.

Nota-se que não há ganho significativo de desempenho após 8 *threads*, pois os valores de *speedup* já estão atingindo o limite do *hardware* utilizado. Neste caso, o que limita o aumento do *speedup* é o número de elementos de processamento disponíveis no sistema (8). Assim, para o NQueens, não seria necessário definir um tamanho de grupo de *threads* maior que a quantidade de elementos de processamento disponíveis.

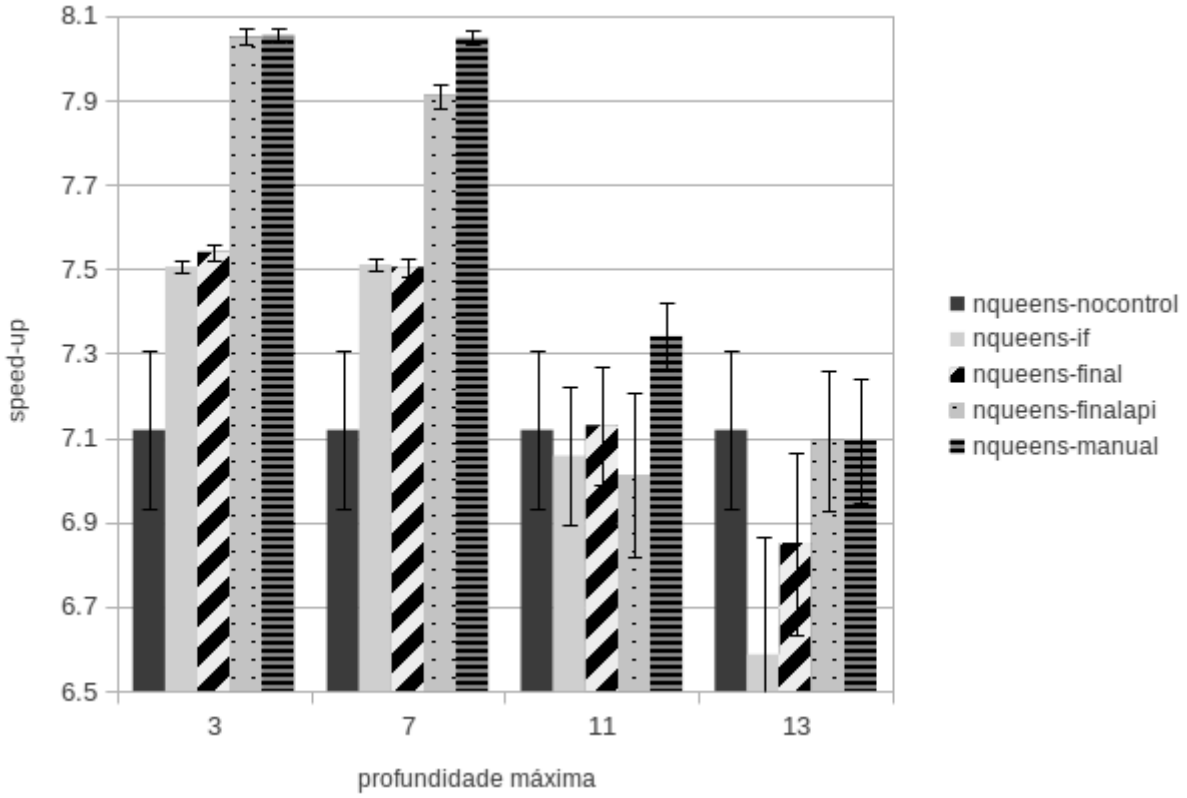

Figura 6.9: NQueens: speedup de diferentes profundidades máximas com 8 *threads*

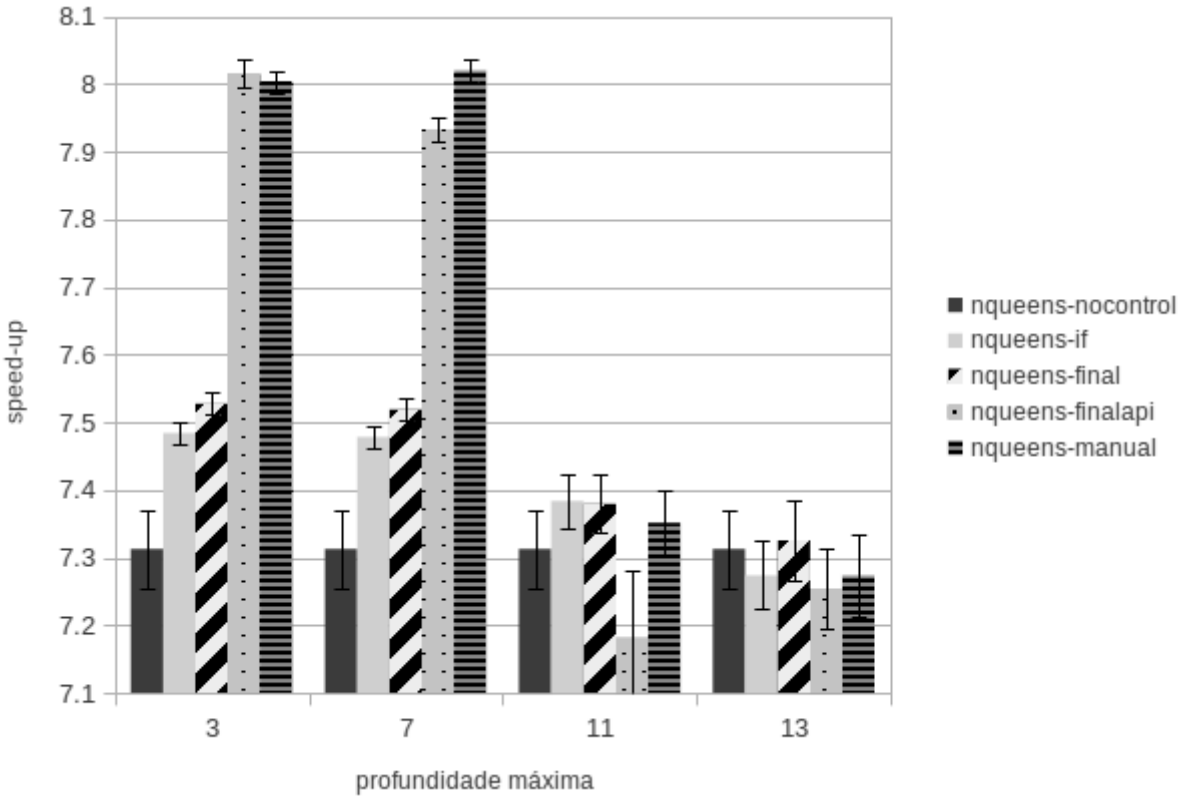

Figura 6.10: NQueens: speedup de diferentes profundidades máximas com 16 *threads*

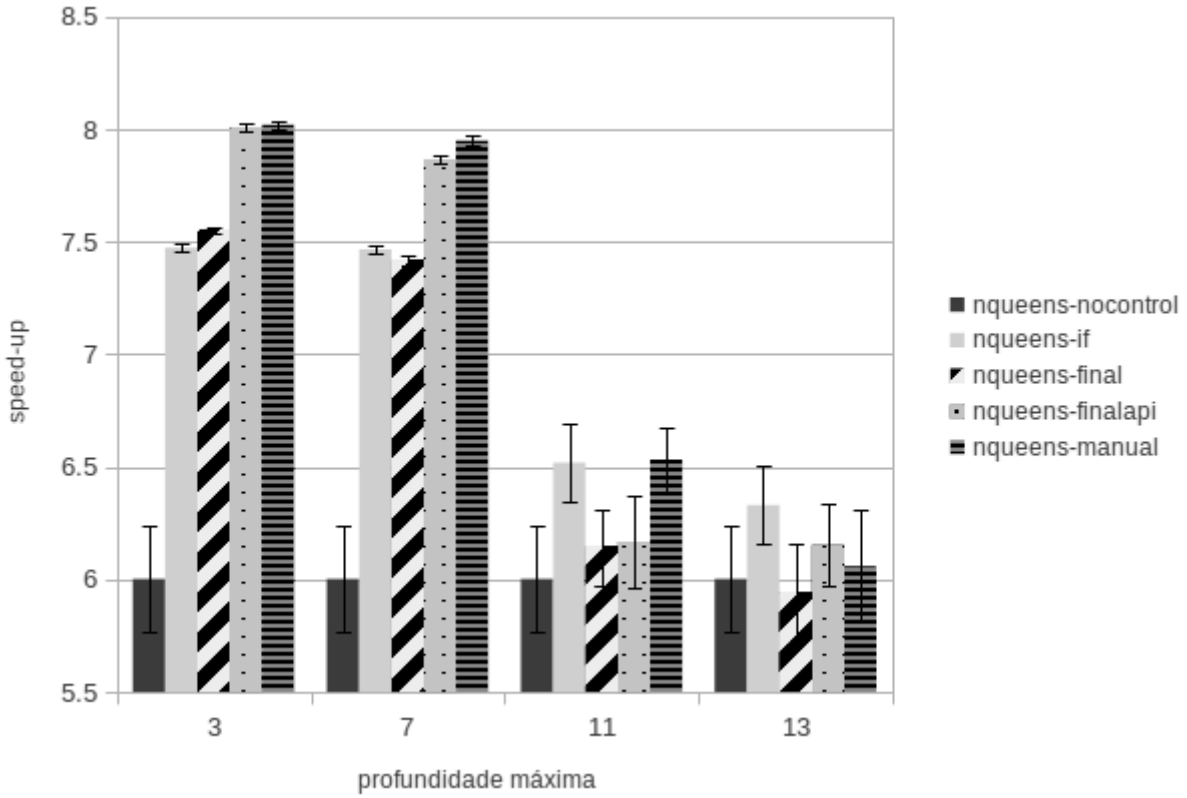

Figura 6.11: NQueens: speedup de diferentes profundidades máximas com 32 *threads*

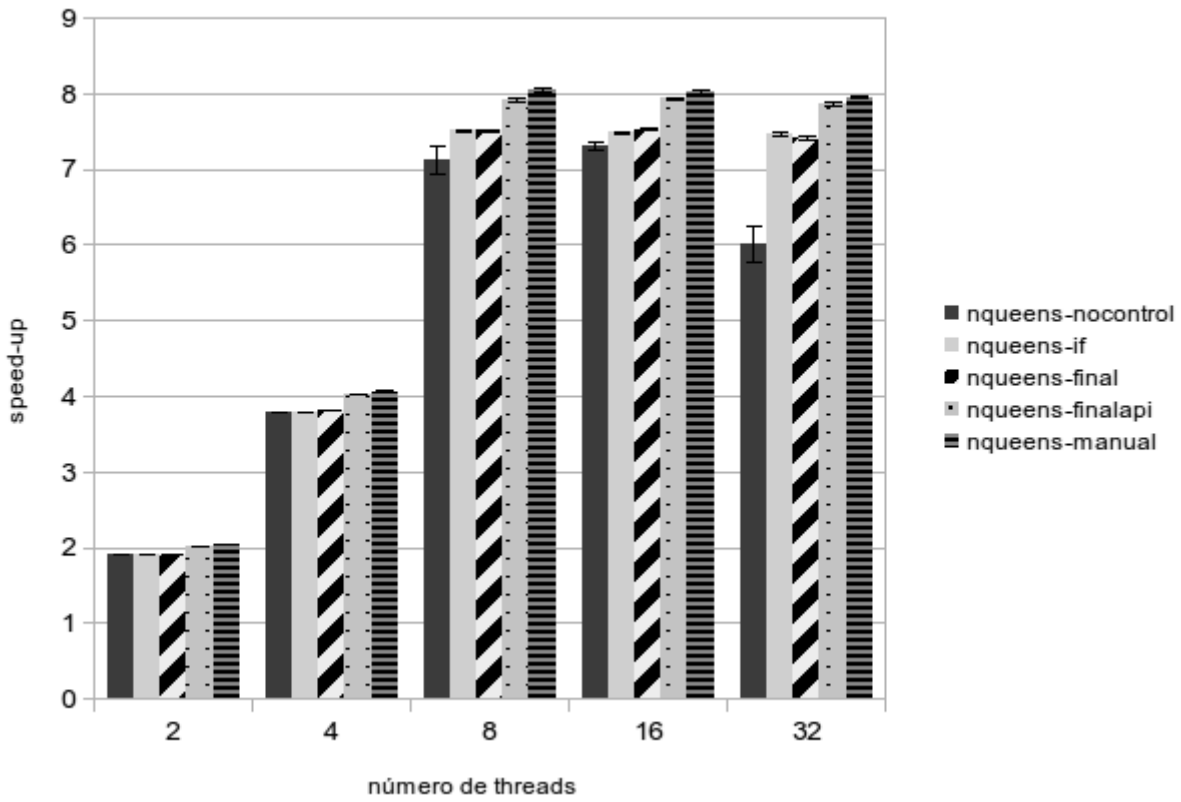

Figura 6.12: NQueens: *speedup* em diferentes números de *threads*

6.2.3 Fibonacci

6.2.3.1 Profundidade máxima

As figuras 6.13 a 6.17 mostram os resultados dos experimentos com o algoritmo Fibonacci utilizando as quatro estratégias de controle de granularidade de tarefas.

No caso do Fibonacci, não estão sendo exibidos os resultados referentes a versão *no-control*, pois, devido à granularidade de tarefas afetar de forma agressiva o *speedup* deste algoritmo, os resultados obtidos para tal versão do programa ficaram muito próximos de zero. Portanto, os mesmos foram omitidos para aumentar a clareza dos gráficos apresentados.

Como dito anteriormente, o algoritmo recursivo que calcula o n-ésimo número da sequência de Fibonacci é, dos algoritmos testados, o que possui tarefas menos complexas. Assim, torna-se ainda mais desafiador o controle de granularidade de tarefas deste programa.

As versões que utilizam cláusula `if` e cláusula `final` se mostraram ineficientes para este algoritmo, devido ao grande peso que o **sobrecusto de tarefas** exerce sobre a execução. Sendo que, para até 8 *threads*, apresentaram *speedup* menor que 1, isto é, foram mais lentas que a versão sequencial. Já as versões com cláusula `final` + API e controle manual se mostraram mais eficientes, porém não atingiram *speedups* próximos aos atingidos nos outros experimentos.

Para uma entrada 40, observamos que, a partir de um valor maior que 15 para *max-depth*, o *speedup* começa a diminuir.

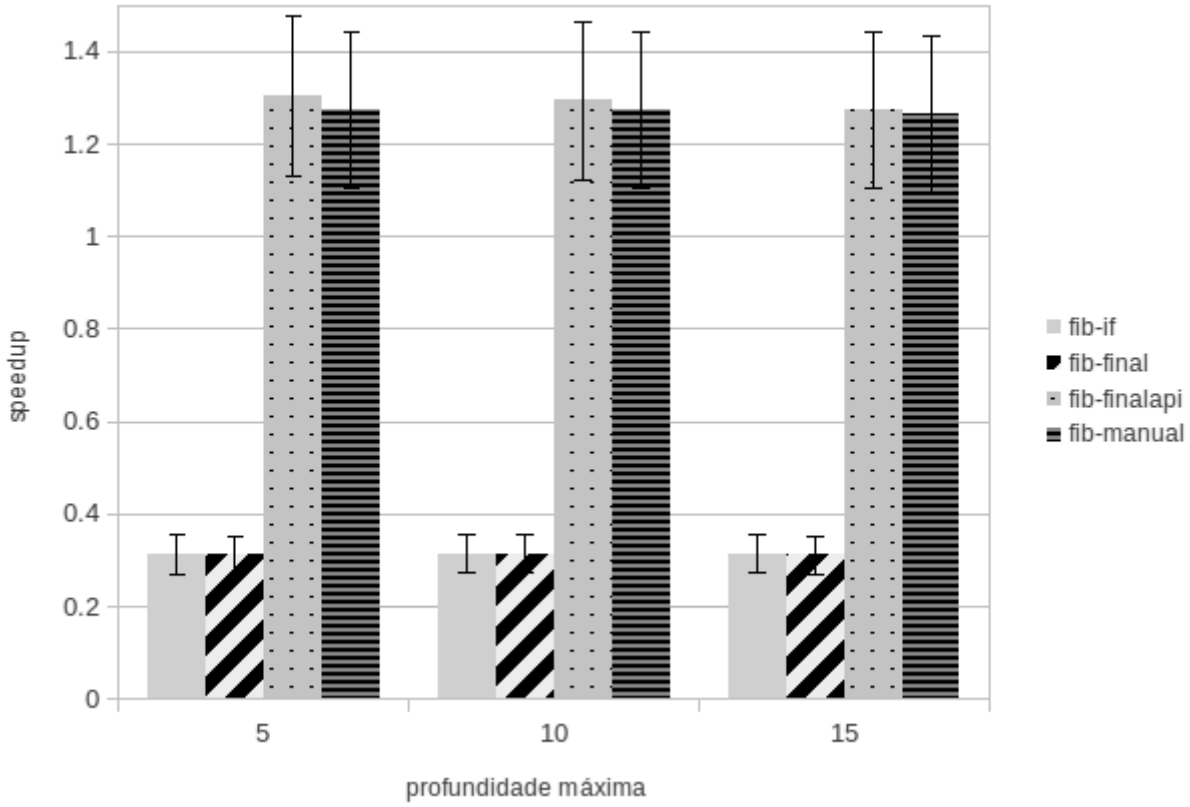

Figura 6.13: Fibonacci: speedup de diferentes profundidades máximas com 2 *threads*

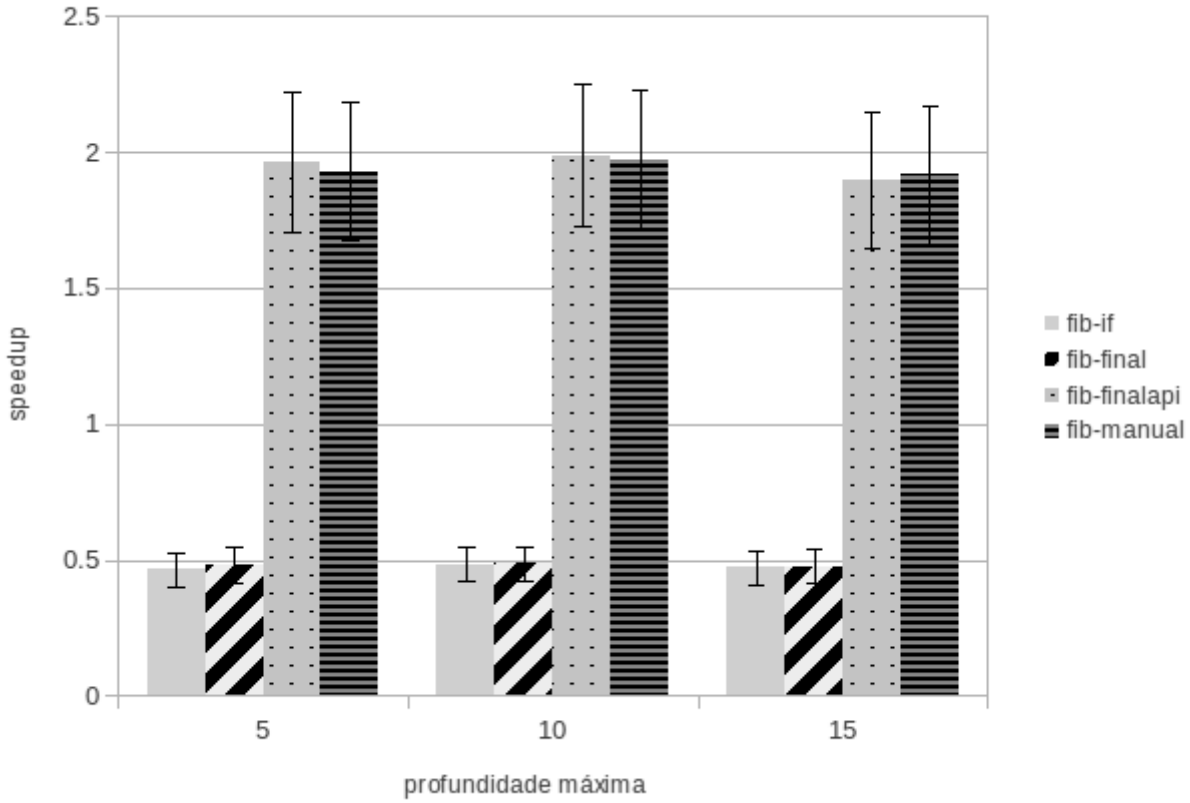

Figura 6.14: Fibonacci: speedup de diferentes profundidades máximas com 4 *threads*

6.2.3.2 Número de threads

A figura 6.18 mostra a curva de *speedup* de cada estratégia em função do tamanho do grupo de *threads* utilizado, com valor de *maxdepth* fixado em 10.

Podemos observar que, apesar de os valores de *speedup* se manterem baixos com poucas *threads*, não há ponto de inflexão na curva, pelo menos no intervalo coletado [2, 32]. Isso significa que, mesmo aumentando o **sobrecusto de threads**, o mesmo não supera o **sobrecusto de tarefas** gerado pela execução do Fibonacci. No entanto, é possível observar que a taxa de aumento de *speedup*, em função do aumento no número de *threads*, começa a diminuir a partir de 16 *threads*, pois $\Delta S_{16-32} < \Delta S_{8-16}$ (onde $\Delta S_{i-j} = |speedup_j - speedup_i|$).

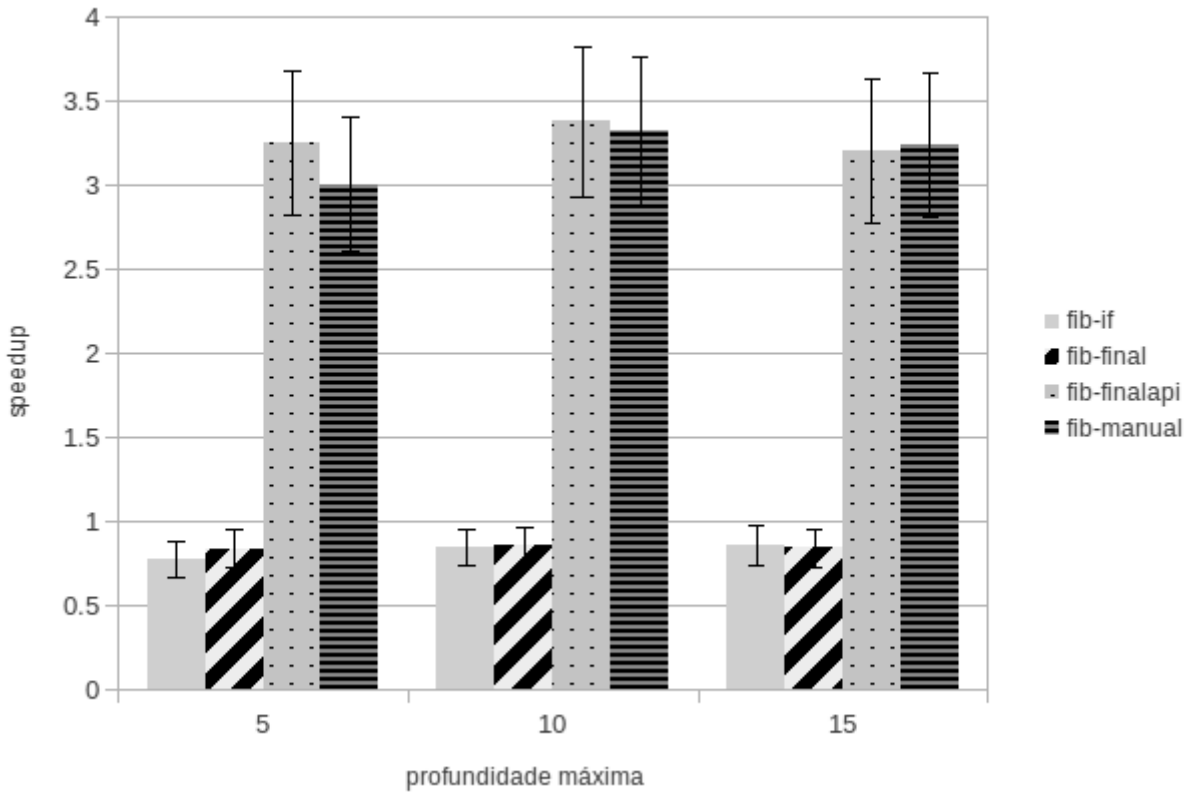

Figura 6.15: Fibonacci: speedup de diferentes profundidades máximas com 8 *threads*

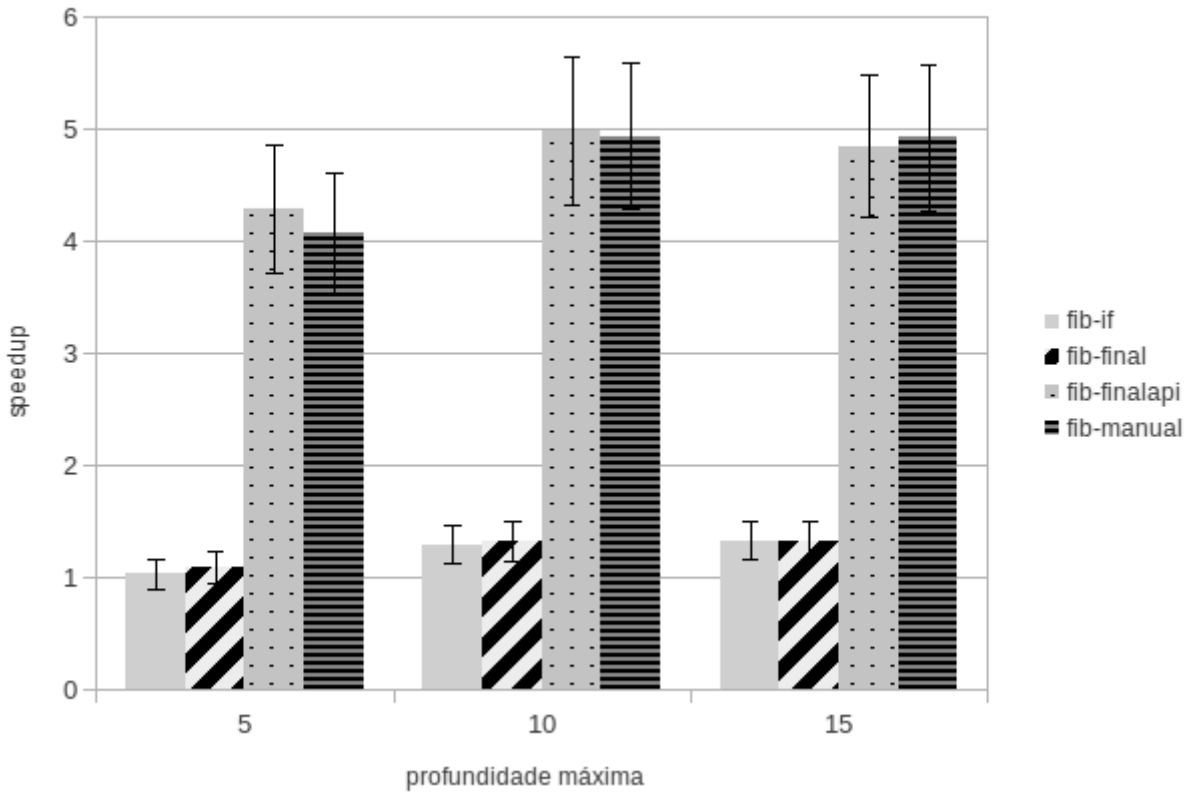

Figura 6.16: Fibonacci: speedup de diferentes profundidades máximas com 16 *threads*

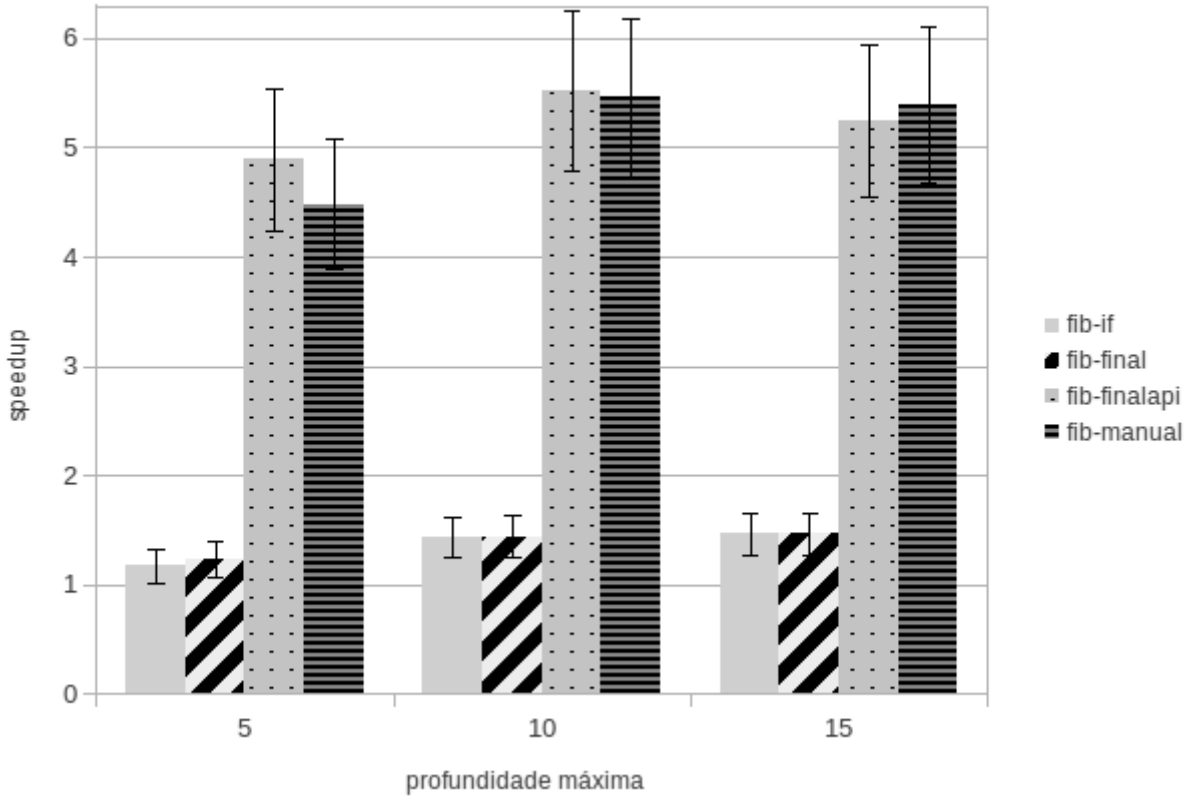

Figura 6.17: Fibonacci: speedup de diferentes profundidades máximas com 32 *threads*

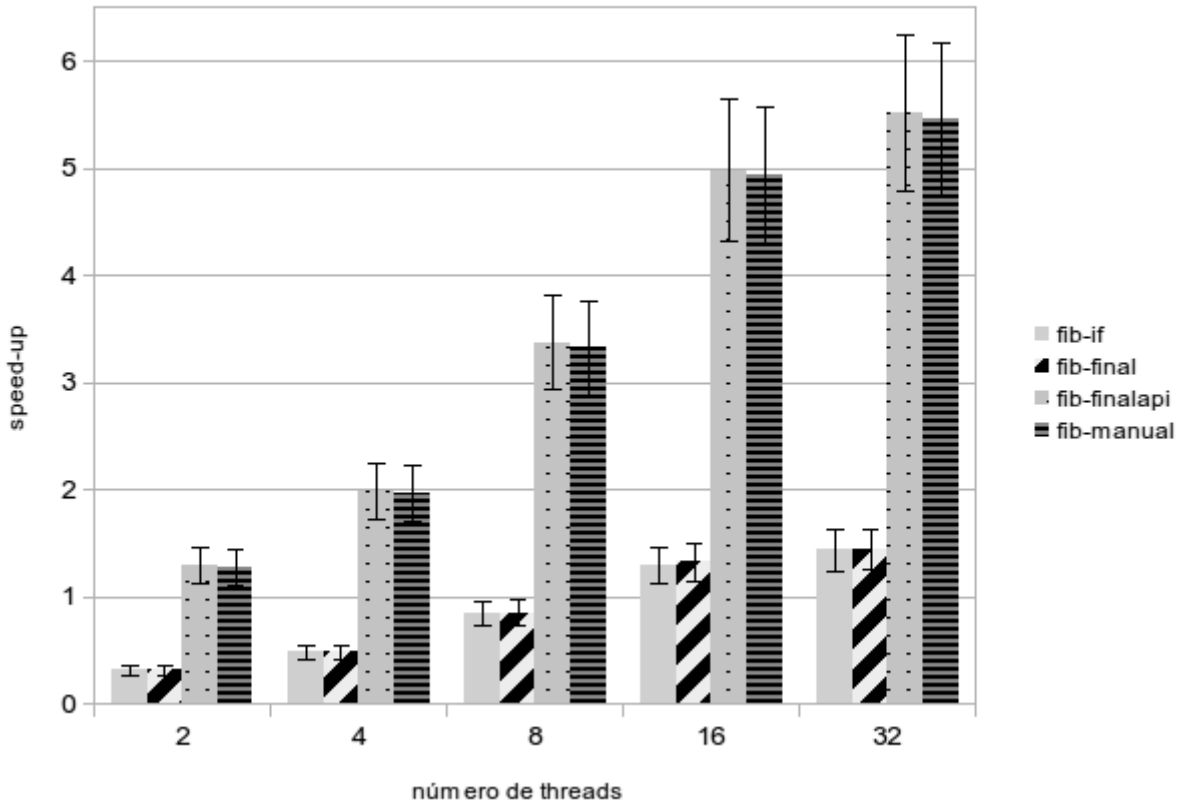

Figura 6.18: Fibonacci: *speedup* em diferentes números de *threads*

7 CONCLUSÃO

Os experimentos realizados com as quatro estratégias de controle de granularidade apresentadas neste trabalho mostram que as estratégias apresentam *speedups* significativos, porém, as estratégias que definem explicitamente um caminho sequencial apresentam grande vantagem.

A norma **OpenMP** evoluiu em relação ao suporte a paralelismo de tarefas, considerando que o mesmo foi introduzido apenas no ano de 2008. No entanto, alguns recursos ainda são muito novos, e será necessário aguardar futuras implementações para que seja feita uma avaliação que possa mensurar com maior grau de confiança os benefícios trazidos. Um exemplo disto, é o uso de `mergeable`, `final` e `omp_in_final()`, inseridos em 2011, na versão 3.1 da norma. Através das análises feitas aqui, foi possível verificar que, no momento, o uso da cláusula `mergeable` faz com que as diretivas `if` e `final` se tornem muito semelhantes em termos de *speedup*, para os algoritmos testados.

Apesar da evolução da API, foi verificada a dificuldade de se obter *speedups* altos em programas com paralelismo de tarefas muito pequenas, caso do Fibonacci. Vale lembrar que a implementação da norma utilizada nas avaliações foi um *snapshot* do compilador GCC versão 4.7. Assim, é possível que estas deficiências sejam amenizadas até que a versão estável seja liberada. No entanto, em um algoritmo irregular, como o *NQueens*, todas as estratégias obtiveram bons resultados.

Enfim, a partir da discussão sobre as causas de sobrecusto em programas com paralelismo de tarefas, e da apresentação de estratégias de controle de granularidade disponíveis em **OpenMP**, o objetivo do trabalho foi alcançado. Ficando, como trabalho futuro, uma pesquisa com maior ênfase sobre a aplicação da diretiva `final` em conjunto com a rotina `omp_in_final`, que se tratam de construções muito recentes da norma. Além disso, seria interessante verificar quais melhorias de desempenho a versão final do compilador GCC versão 4.7 trará em relação à versão em desenvolvimento utilizada aqui. Bem como a aplicação dos experimentos aqui realizados em arquiteturas com quantidades maiores de elementos de processamento.

REFERÊNCIAS

[APE 2010] APEL, B. G.; MAILLARD, N. Rumos para paralelismo de tarefas eficiente em memória distribuída. **Semana Acadêmica do Instituto de Informática. Universidade Federal do Rio Grande do Sul.**, 2010.

[AYG 2010] AYGUADé, E. et al. An extension to improve openmp tasking control. **Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More. 6th International Workshop on OpenMP, IWOMP 2010, Tsukuba, Japan**, 2010.

[BAR 2011] BARNEY, B. **Openmp tutorial**. 2011. https://computing.llnl.gov/tutorials/openMP/: [s.n.], 2011. Acessado dia 18 de Setembro de 2011.

[BLU 99] BLUMOFE, R. D.; LEISERSON, C. E. Scheduling multithreaded computations by work stealing. **ACM**, v.46, p.720–748, 1999.

[BLU 94] BLUMOFE, R.; LEISERSON, C. Scheduling multithreaded computations by work stealing. **Foundations of Computer Science, 35th Annual Symposium.**, p.356–368, 1994.

[CER 2011] CERA, M. C. et al. Providing adaptability to mpi applications through explicit task parallelism. **International Symposium on Computer Architecture and High Performance Computing**, 2011.

[CHA 2008] CHAPMAN, B.; JOST, G.; PAS, R. van der. **Using openmp**: portable shared memory parallel programming. [S.l.]: The MIT Press, 2008.

[DER 2003] LUZZATO, A. (Ed.). **Arquiteturas paralelas**. [S.l.]: Editora Sagra Luzzato, 2003.

[DUR 2008] DURAN, A.; CORBALáN, J.; AYGUADé, E. Evaluation of openmp task scheduling strategies. **IWOMP 2008**, 2008.

[FLY 72] FLYNN, M. J. Some computer organizations and their effectiveness. **IEEE Transaction on Computers**, v.C-21, p.948–960, 1972.

[HEN 2006] HENNESSY, J. L.; PATTERSON, D. A. **Computer architecture**: a quantitative approach. 4.ed. [S.l.]: Morgan Kaufmann, 2006.

[OPE 2011] OPENMP application program interface 3.1 complete specifications. 2011.

[RAO 93] RAO, V. N.; KUMAR, V. On the efficiency of parallel backtracking. **IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS**, v.4, 1993.

[ROB 2008] ROBISON, A.; VOSS, M.; KUKANOV, A. Optimization via reflection on work stealing in tbb. **Parallel and Distributed Processing, IPDPS, IEEE. International Symposium**, p.1–8, 2008.

[WIL 2008] WILLHALM, T.; POPOVICI, N. Putting intel©threading building blocks to work. **IWMSE 08: Proceedings of the 1st international workshop on Multicore software engineering**, p.3–4, 2008.