Capítulo 6

Desenvolvimento de Aplicações Baseadas em Tarefas com OpenMP Tasks

Lucas Leandro Nesi, Marcelo Cogo Miletto,
Vinícius Garcia Pinto, Lucas Mello Schnorr
Instituto de Informática, Universidade Federal do Rio Grande do Sul
Porto Alegre, Brasil

Resumo

O minicurso enquadra-se no contexto de programação paralela utilizando diretivas de programação para facilitar o desenvolvimento de aplicações. O paradigma orientado a tarefas aporta facilidades na programação paralela porque transfere para um runtime muitas responsabilidades que seriam anteriormente realizadas pelo programador nos paradigmas tradicionais. Por exemplo, é responsabilidade do runtime escalonar as tarefas nas unidades de processamento, gerenciar a memória e o balanceamento de carga. Para isso, basta o programador definir as tarefas e suas dependências de dados. Neste minicurso, será apresentado o paradigma de programação paralela orientado a tarefas, e como construir programas com diretivas de programação utilizando tarefas OpenMP. O minicurso será conduzido de forma prática, com exemplos e exercícios de programas básicos e naturalmente adaptáveis ao paradigma orientado a tarefa. Serão empregados exemplos de aplicações como Mergesort, suavização de Gauss-Seidel e fatoração Cholesky. Por fim, abordaremos rapidamente as ferramentas e métodos de como analisar o desempenho destes programas.

6.1. Introdução

O panorama do Processamento de Alto Desempenho (PAD) passou por uma mudança de paradigma nos últimos anos. A inerente estagnação no aumento da frequência do processador levou à adoção de outras maneiras de atender à necessidade cada vez maior de poder de computação das aplicações paralelas. Atualmente, as plataformas de PAD envolvem nós com processadores equipados com múltiplos *cores* aprimorados com várias placas aceleradoras.

Essa mudança de paradigma no *hardware* revela limitações nas ferramentas tradicionais para programação de aplicações paralelas para PAD. Programar com eficiência essas máquinas, com desempenho portável e escalável, mantém-se desafiador. Por exemplo, o uso de modelos de programação explícitos, tais como a interface de programação MPI e versões antigas do OpenMP, exigem se preocupar com o balanceamento de carga, equilíbrio entre comunicações e computação. Frequentemente, tais preocupações imputadas ao desenvolvedor tornam a aplicação fortemente acoplada a uma plataforma alvo. Por consequência, esse modelo de programação explícito se torna inviável considerando as plataformas para PAD evoluem rapidamente ao longo do tempo.

Enquanto o paradigma de programação paralela tradicional depende de abstrações de baixo nível, como fluxos de execução e sincronizações explícitas, o modelo baseado em tarefas descreve a aplicação paralela com tarefas sequenciais dependentes. As sincronizações explícitas são substituídas por dependências de tarefas que podem ser, em vários casos, inferidas automaticamente do acesso aos dados pelo próprio ambiente de execução. Outra evidência é que semanticamente alguns algoritmos são melhor expressados em tarefas. Um exemplo são algoritmos do tipo "dividir para conquistar", como o *merge sort*, onde uma implementação recursiva com tarefas é natural. Existem outros benefícios da programação baseada em tarefas [Rico et al. 2019], tais como dependências finas entre tarefas no lugar barreiras de sincronização, e o uso de algoritmos de escalonamento sofisticados. O modelo baseado em tarefas é implementado por vários modelos de programação: OpenMP 5 [OpenMP 2020], OmpSs [Duran et al. 2011], Parsec [Bosilca et al. 2012], StarPU [Augonnet et al. 2011], entre outras. A crescente disponibilidade de ferramentas para programar e executar aplicações baseados em tarefas em plataformas híbridas demostra a crescente importância do paradigma baseado em tarefas.

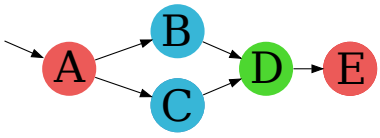

Figura 6.1. Um grafo com três tipos de tarefas (vermelhas, azuis e verdes), sendo que as dependências de dados são representadas pelas arestas.

A interface de programação OpenMP [OpenMP 2020] suporta a programação paralela em máquinas com memória compartilhada, tipicamente um nó computacional equipado com múltiplos processadores *multicore*. Desde as especificações 3.0 e 4.0, OpenMP especifica diretivas para a programação com tarefas. Com estas diretivas, o programador especifica a submissão de tarefas e a relação de dependência entre elas. Um grafo é frequentemente utilizado para ilustrar a aplicação como um todo, como representado na Figura 6.1. Nesta figura, o grafo representa uma aplicação com cinco tarefas (nós identificados de A até E) de três tipos (cores). Tarefas de diferentes tipos implementam funcionalidades diversas, mas que cooperam entre si através das dependências de dados (arestas). Cada tarefa é composta por um trecho de código, frequentemente sequencial, que implementa uma funcionalidade baseada em seus dados de entrada, e gerando na saída um outro conjunto de dados processados. Após a compilação, o ambiente de execução escalonará a tarefa A para execução, que uma vez terminada permitirá o escalonamento para execução das tarefas B e C, e assim por diante.

Este minicurso aborda a forma de se construir programas paralelos com diretivas de programação utilizando tarefas OpenMP. O minicurso traz exemplos e exercícios de programas básicos e naturalmente adaptáveis ao paradigma de tarefas. O texto do minicurso está organizado da seguinte forma. A Seção 6.2 apresenta as diretivas OpenMP relacionadas à criação do grafo de tarefas e geração das dependências. A Seção 6.3 apresenta exemplos de aplicações desenvolvidas em OpenMP de maneira a ilustrar boas práticas de programação. A Seção 6.4 apresenta um método para avaliar o desempenho das aplicações utilizando técnicas de rastreamento da execução das tarefas. Enfim, a Seção 6.5 apresenta um sumário e recomendações de como dar continuidade no estudo sobre o assunto. O repositório[1] contém códigos, apresentação e outros materiais deste minicurso.

6.2. Programação Paralela com Tarefas em OpenMP

A utilização do OpenMP Tasks segue inicialmente os mesmos princípios da utilização de outras diretivas do OpenMP. Anotações de código com `#pragma omp` são inseridas no código em lugares estratégicos. Um compilador que suporta OpenMP (como `gcc`, `clang`, `icc`) é utilizado para compilar o código com as `flags` adequadas[2]. O gerenciamento de *threads* é realizado pelo OpenMP em regiões paralelas anotadas com a diretiva `#pragma omp parallel`. Usualmente precisamos declarar regiões que devem ser executadas por uma única *thread* dentro de regiões `parallel`, para isso, usamos `#pragma omp single` nestes blocos. O OpenMP também dispõe de uma interface de funções, no cabeçalho `omp.h`, para gerenciar e acessar dados disponíveis. Um exemplo dessas funções, utilizada para se obter o identificador da *thread* executando o fluxo de execução, é `omp_get_thread_num`. O exemplo de código abaixo mostra a estrutura básica de um programa OpenMP.

```
1  #include <stdio.h>
2  #include <omp.h>
3  int main(){
4    #pragma omp parallel
5    {//OpenMP inicia várias threads
6      printf("Bloco paralelo executado por:%d\n", omp_get_thread_num());
7      #pragma omp single
8      {// Este Bloco é executado unicamente por uma única thread
9        printf("Bloco único executado por:%d\n", omp_get_thread_num());
10     }
11   }
12 }
```

Para a compilação deste programa utilizando o `gcc`, pode-se utilizar:

```
gcc estrutura_openmp.c -fopenmp
```

O controle do acesso de variáveis em regiões paralelas com tasks é igual a outras diretivas tradicionais como `parallel for`, sendo dada pelas anotações `shared`, `private`, `firstprivate` e `lastprivate`. Por exemplo, em um laço anotado com `parallel for`, onde deseja-se que a variável `vec_c` seja compartilhada por todas as

[1] `https://gitlab.com/lnesi/companion-minicurso-openmp-tasks`

[2] As flags de compilação variam por compilador: -fopenmp (gcc e clang), -qopenmp (icc).

threads, e a variável `vec_p` seja privada (instanciada, lida e modificada apenas dentro de cada *thread*) utiliza-se o código abaixo:

```
1  int vec_c = 5;
2  int vec_p;
3  #pragma omp parallel for shared(vec_c) private(vec_p)
4  for(int i=0; i<10; i++){
5     // vec_c é compartilhada por todas as threads
6     // Existe uma vec_p para cada thread
7  }
```

Depois de relembrar os conceitos básicos do OpenMP, podemos iniciar com as diretivas orientadas a tarefas. Começamos pela principal diretiva para definição de uma tarefa. Ela é dada por `#pragma omp task` imediatamente antes de um bloco de código (que pode ser uma chamada de função) que fará ofício de tarefa, tal como ilustrado neste exemplo:

```
1  void minha_tarefa(char* s){
2      printf("%s ", s);
3  }
4  int main(){
5    #pragma omp parallel
6    {//OpenMP inicia várias threads
7      #pragma omp single
8      {// Este Bloco é executado unicamente por uma thread
9         #pragma omp task
10        minha_tarefa("Olá");
11        //#pragma omp taskwait
12        #pragma omp task
13        minha_tarefa("Mundo");
14     }
15   }
16 }
```

Cada invocação da função `minha_tarefa` será uma tarefa que pode ser executada por qualquer *thread* do grupo de *threads* do OpenMP. Como não existem dependências entre elas, e elas podem ser executadas paralelamente, não existe garantia que a tarefa que imprime "Olá" será executada antes que a tarefa que imprime "Mundo". Isso pode ser verificado executando o programa algumas vezes. Em algumas situações o resultado será "Olá Mundo" enquanto outras "Mundo Olá". Para garantir a execução de "Olá" antes de "Mundo", podemos adicionar a diretiva `#pragma omp taskwait` entre a submissão de ambas as tarefas, na linha 11. Isso significa que quando a *thread* que esta submetendo as tarefas chegar no `taskwait` ela irá esperar pela conclusão de todas as tarefas submetidas até então. Isso garante que antes da submissão da tarefa "Mundo" a tarefa "Olá" já acabou. Entretanto, neste caso, não existe paralelismo. A Figura 6.2 apresenta o DAG desta simples aplicação.

Pode-se lançar várias tarefas com argumentos diferentes que são escalonadas pelo OpenMP. No exemplo abaixo, limitamos o número de *threads* para cinco utilizando `num_threads(5)`. Uma *thread* vai executar o `for` submetendo 10 tarefas com argumentos diferentes de zero à nove. A tarefa é simples, ela recebe `i` como argumento, e dorme `i` segundos. A ordem de execução é estocástica, mas o escalonamento será dado pela ordem de submissão. A

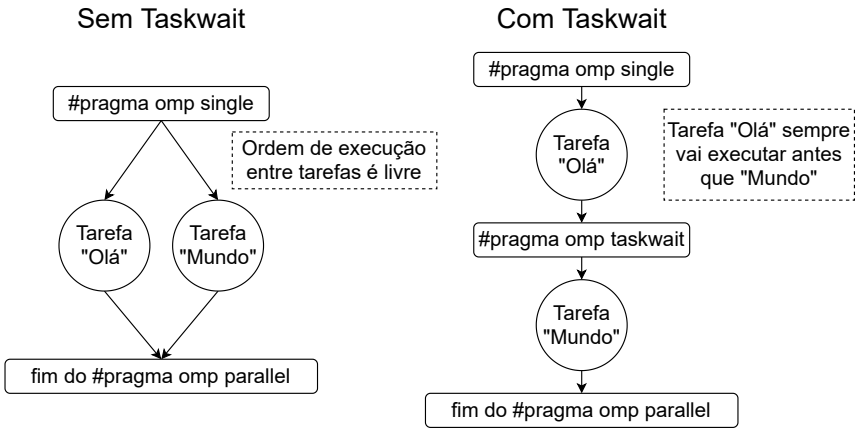

Figura 6.2. DAG para os códigos de Olá Mundo sem e com `taskwait`.

Figura 6.3 mostra uma provável execução do programa abaixo. Esta provável execução leva 13 segundos, onde apenas uma *thread* levou mais tempo que as demais. A situação ideal seria quando a tarefa de tempo 9 fosse executada junto com a tarefa 0, a tarefa 8 com a tarefa 1, a tarefa 7 com a tarefa 2, a tarefa 6 com a tarefa 3, e a tarefa 5 com a tarefa 4. A situação com tempo 13 acontece porque a tarefa mais custosa foi escalonada por último, e não será executada com a tarefa 0. Para tentar melhorar o desempenho pode-se dar dicas ao OpenMP em como escalonar melhor as tarefas utilizando prioridades. Isto é uma responsabilidade do programador, já que o OpenMP não tem como saber previamente o tempo de duração das tarefas ou outras situações.

```
1  void task(int i){
2     sleep(i);
3  }
4  int main(){
5    #pragma omp parallel num_threads(5)
6    {//OpenMP inicia várias threads
7      #pragma omp single
8      {// Este Bloco é executado unicamente por uma thread
9         for(int i = 0; i < 10; i++){
10           #pragma omp task
11           task(i);//Each task will sleep diferently
12        }
13     }
14   }
15 }
```

A utilização de prioridades nas tarefas do OpenMP se dá pela adição da cláusula `priority(valor)` em um `#pragma omp task`, onde o parâmetro valor é a prioridade da tarefa. Tarefas com prioridades maiores serão escalonadas primeiro. As prioridades no OpenMP assumem valores de zero até um valor máximo determinado pela variável de ambiente `OMP_MAX_TASK_PRIORITY`. Entretanto, o padrão desta variável de ambiente é zero. Desta forma, para qualquer utilização de prioridades em um programa com tarefas OpenMP, deve-se definir esta variável de ambiente.

O fragmento de código abaixo apresenta uma primeira variação do código anterior

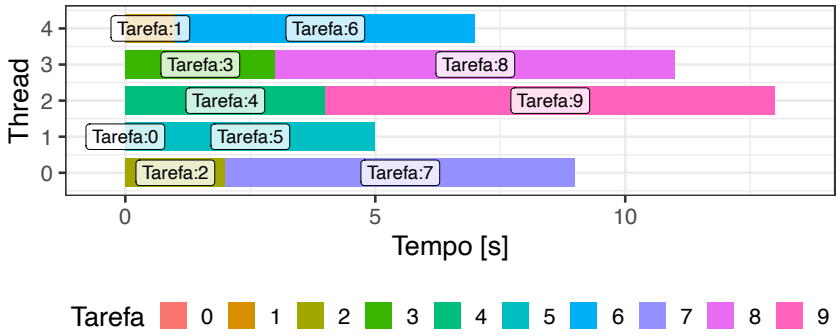

Figura 6.3. Uma provável execução do programa sem prioridades.

adicionando prioridades equivalentes ao tempo de execução das tarefas. Assim, escalonando as tarefas mais custosas primeiro, podemos preencher melhor o tempo. Já que com cinco *threads*, as tarefas 9, 8, 7, 6, 5 seriam escalonadas em um primeiro momento, e quando a primeira tarefa acabar (5), a tarefa 4 seria escalonada para a mesma *thread*. A Figura 6.4 ilustra uma provável execução desta variação. A execução não aconteceu da forma como se imaginava, apesar de reduzir o tempo de execução em um segundo. Isso aconteceu porque o escalonamento das tarefas acontece assim que a tarefa é submetida. Então como as tarefas 0 à 4 são submetidas primeiro, e existem recursos livres, elas são imediatamente escalonadas, quebrando assim a ordem que desejávamos estabelecer. Neste caso, para ficar na ordem desejada pode-se alterar a ordem de submissão ou alterar as prioridades.

```
1    #pragma omp single
2    {// Este Bloco é executado unicamente por uma thread
3       for(int i = 0; i < 10; i++){
4          int prio = i;
5          #pragma omp task priority(prio)
6          task(i);//Each task will sleep diferently
7       }
8    }
```

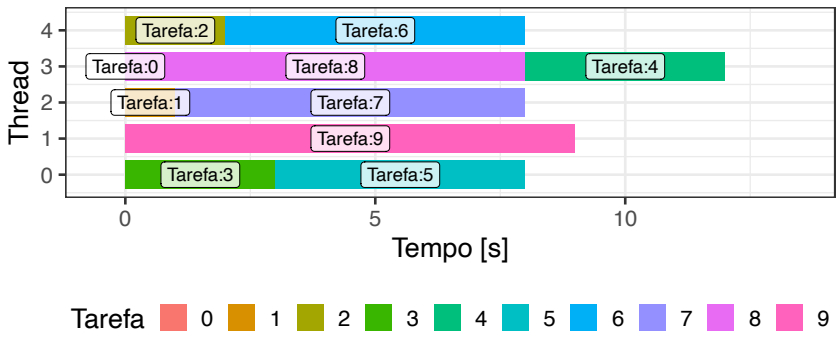

Figura 6.4. Provável execução do programa com prioridades igual ao tempo de execução.

As prioridades que auxiliam o OpenMP para este programa ter o melhor tempo de execução estão no fragmento de código abaixo. Considerando a ordem de submissão, a execução das tarefas 0 à 4 primeiro é inevitável. Desta forma, pode-se aplicar a prioridade máxima para elas, e após isso segue a prioridade do exemplo anterior. Quando a tarefa 0 termina, a tarefa 9 é escalonada para a mesma *thread* e assim por diante. A execução desta versão está na Figura 6.5. Esta versão atinge o menor tempo de execução possível de nove segundos. A Figura 6.6 mostra o DAG desta aplicação com a última versão das prioridades.

```
1    #pragma omp single
2    {// Este Bloco é executado unicamente por uma thread
3       for(int i = 0; i < 10; i++){
4          int prio = i < 5 ? 10 : i;
5          #pragma omp task priority(prio)
6          task(i);//Each task will sleep diferently
7       }
8    }
```

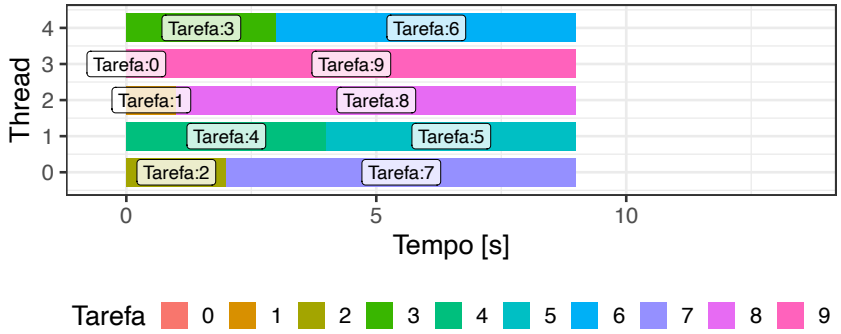

Figura 6.5. Provável execução do programa com novas prioridades.

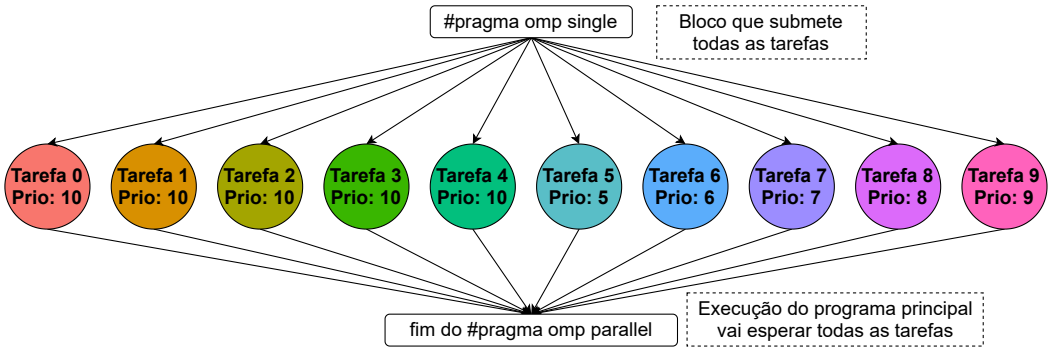

Figura 6.6. DAG da aplicação com 10 tarefas independentes de tempos diferentes.

OpenMP permite lançar tarefas dentro de tarefas. As novas tarefas podem começar sem necessariamente a tarefa que as submeteu terminar. Entretanto, a tarefa que as submete pode esperar por todas as tarefas, chamadas neste contexto de subtarefas, que ela submeteu com `taskwait`. Isto vai gerar a suspensão dela até que todas as subtarefas

estejam terminadas. Deste modo, a tarefa será dividida em etapas que serão executados em momentos distintos. O exemplo do segmento de código abaixo possui uma tarefa (`tarefa_complexa`) que submete outras tarefas (`tarefa_simples`), com e sem `taskwait` dentro da `tarefa_complexa`. O padrão do OpenMP é deixar apenas a *thread* que executou a tarefa antes da suspensão continuar sua execução. Entretanto, podemos utilizar a opção `untied`. Desta maneira, quando uma tarefa que entrou em suspensão voltar a ser executada, qualquer *thread* poderá fazê-lo. Tarefas podem anotar pontos de suspensão com `taskyield`, onde o OpenMP vai decidir se continua a tarefa ou a suspende em favor de outra tarefa (com prioridade maior por exemplo). A Figura 6.7 mostra o DAG do programa, com e sem o `taskwait` da linha 9.

```
1  void tarefa_simples(){
2     sleep(1);
3  }
4  void tarefa_complexa(){
5    #pragma omp task
6    tarefa_simples();
7    #pragma omp task
8    tarefa_simples();
9    //#pragma omp taskwait
10 }
11 int main(){
12   #pragma omp parallel
13   {//OpenMP inicia várias threads
14     #pragma omp single
15     {// Este Bloco é executado unicamente por uma thread
16        #pragma omp task //untied
17        tarefa_complexa();
18        #pragma omp task
19        tarefa_simples();
20     }
21   }
22 }
```

Tarefas podem ter dependências de dados, isto é, uma tarefa pode necessitar de um dado computado por outra tarefa e gerar dados que serão utilizados por outras tarefas. O conjunto de dependências entre tarefas formam a estrutura do DAG e guiam a ordem de execução das tarefas em aplicações mais complexas. Para informar uma dependência de uma variável (de entrada ou saída) em uma tarefa, deve-se utilizar a cláusula `depend(modo: variável)`. Primeiramente informa-se o modo: `in` para variáveis de entrada, `out` para variáveis de saída ou `inout` para ambos os casos. Em seguida temos `:` (dois pontos) e a lista de variáveis. Para vetores, uma das notações possíveis para explicitar a dependência de apenas algumas posições é a notação `vetor[inicio:tamanho]`. A especificação do OpenMP obriga o uso de declarações de posição nos vetores iguais ou disjuntas. Por exemplo, vamos assumir um vetor `A` de tamanho total 4 e uma tarefa que escreve em todas as posições deste vetor, seguido por uma outra tarefa que depende apenas das posições 2 e 3. A notação para a primeira tarefa deverá ser `depend(out: A[0:2], A[2:2])`. A razão disso é que a segunda tarefa deverá ter a notação `depend(in: A[2:2])`. Ou seja, não pode-se utilizar `A[0:4]` na primeira tarefa já que o conjunto de posições não é igual nem disjunto a `A[0:2]`.

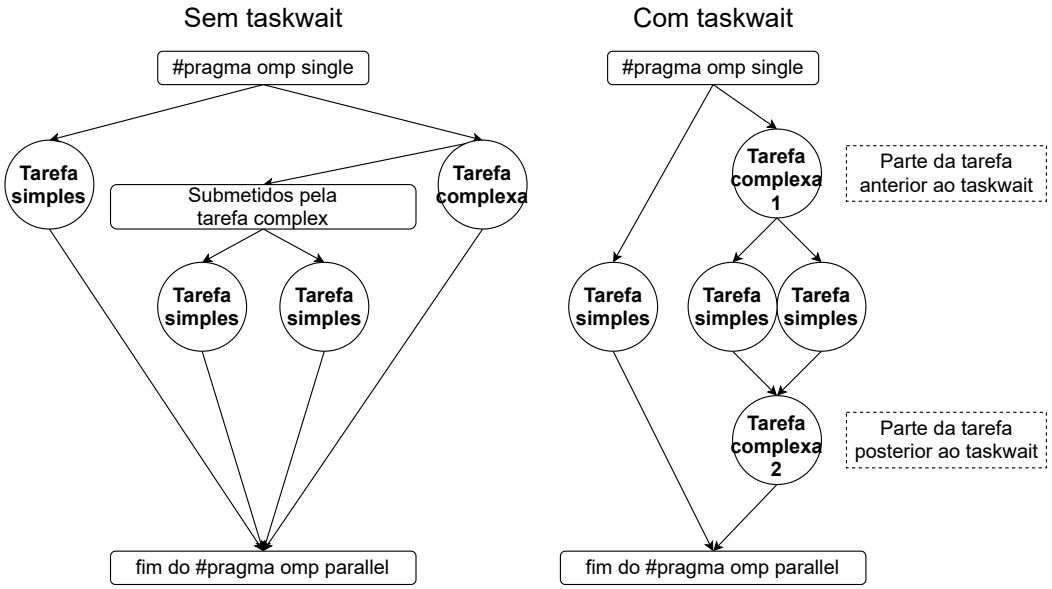

Figura 6.7. DAG da aplicação (tarefa_complexa) sem e com taskwait.

O exemplo do código abaixo emprega tarefas para demonstrar a utilização de dependências para o cálculo da seguinte equação utilizando o vetor `A`:

$$R = A[0] \times A[1] + A[2] \times A[3] \tag{1}$$

Neste caso, sabemos que as operações $A[0] \times A[1]$ e $A[2] \times A[3]$ podem acontecer paralelamente. Define-se cada uma delas como uma tarefa utilizando as variáveis temporárias c e d. Ainda, o exemplo tem uma tarefa de inicialização que define os valores do vetor A. A última tarefa realiza então a operação final $R = c + d$. A tarefa de inicialização tem como dependência todo o vetor. As tarefas de multiplicação tem como dependência duas células do vetor (posições [0,1] e [2,3] respectivamente). Como precisamos definir dependências iguais ou disjuntas, as dependências da tarefa de inicialização são `A[0:2]` e `A[2:2]` como saida (`out`). As dependências da tarefa $c = A[0] \times A[1]$ são `in: A[0:2]` e `out: c`, gerando a dependência com a tarefa anterior. A tarefa $d = A[2] \times A[3]$ tem dependências `in: A[2:2]` e `out: d`, causando novamente a dependência com a tarefa de inicialização. Estas dependências permitem a execução em paralelo das tarefas de multiplicação. Por último, a tarefa $R = c + d$ tem as dependências `in: c, d`, causando a espera de ambas as tarefas de multiplicação. A Figura 6.8 mostra o DAG deste exemplo.

```
1  int main(){
2    int A[4], c=0, d=0, result=0;
3    #pragma omp parallel
4    {
5      #pragma omp single
6      {
7         #pragma omp task depend(inout: A[0:2], A[2:2])
8         {
9            A[0]=1; A[1]=1; A[2]=2; A[3]=3;
```

```
10          }
11          #pragma omp task depend(in: A[0:2]) depend(out: c)
12          {
13             c = A[0] * A[1];
14          }
15          #pragma omp task depend(in: A[2:2]) depend(out: d)
16          {
17             d = A[2] * A[3];
18          }
19          #pragma omp task depend(in: c, d)
20          {
21             result = c + d;
22          }
23       }
24    }
25    printf("Resultado:%d\n", result);
26 }
```

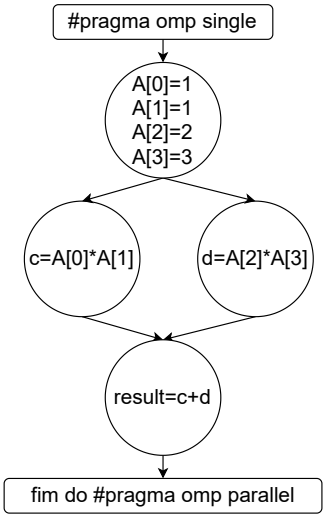

Figura 6.8. DAG da aplicação de exemplo da cláusula depend.

6.3. Exemplos de Aplicações com tarefas OpenMP

Três exemplos são utilizados para ilustrar o emprego das diretivas OpenMP orientadas a tarefas: *Merge Sort*, suavização de Gauss-Seidel e fatoração Cholesky.

6.3.1. Ordenação por mistura (*MergeSort*)

Algoritmos recursivos no modelo divisão e conquista como a ordenação por mistura (*MergeSort*) são diretamente adaptáveis ao modelo baseado em tarefas. As duas chamadas recursivas utilizadas para tratar os subproblemas (linhas 5 e 7 do código abaixo) podem ser transformadas em duas tarefas independentes com a adição das diretivas (`#pragma omp task`). Estas duas tarefas podem ser executadas concorrentemente em qualquer ordem visto que não há dependências de dados entre elas, i.e., ambas acessam partes distintas do vetor.

A adição da diretiva `#pragma omp task` altera o comportamento do código sequencial original (i.e., sem as diretivas OpenMP) pois a execução das funções `merge_sort` das linhas 5 e 7 passa a ser assíncrona. Dessa forma, não há garantias de que as chamadas à `merge_sort` imediatamente anteriores já tenham retornado ao executarmos a chamada à função `merge` na linha 9. Para garantir o correto funcionamento do algoritmo é necessário então impor um ponto de sincronização antes da execução desta função. Esta sincronização é obtida com a diretiva `#pragma omp taskwait` na linha 8, forçando o avanço da execução somente após o fim de todas as tarefas criadas anteriormente.

```
1  void merge_sort(int vetor[], int tam) {
2    int metade = tam / 2;
3    if (tam > 1) {
4      #pragma omp task
5      merge_sort(vetor, metade);
6      #pragma omp task
7      merge_sort(vetor + metade, tam - metade);
8      #pragma omp taskwait
9      merge(vetor, tam);
10   }
11   return;
12 }
```

Ao paralelizar algoritmos recursivos, como o *MergeSort*, é hábito estabelecer limiares para evitar a criação de novas tarefas que tratem de problemas demasiadamente pequenos. No exemplo a seguir, é utilizado o condicionante `if` junto à diretiva `#pragma omp task` para evitar que novas tarefas sejam criadas quando o subproblema for menor ou igual a `MIN_PAR`. Já a adição da cláusula `untied` permite que partes diferentes da uma mesma tarefa sejam executadas por *threads* diferentes. No caso do *MergeSort*, por exemplo, a *thread* 0 poderia executar a tarefa desde início até atingir o `taskwait`, colocando-a em espera. Quando a sincronização fosse concluída (i.e., no final das tarefas criadas anteriormente) a continuação da tarefa e a subsequente chamada a função `merge` poderiam ser executadas pela *thread* 1. Caso a cláusula `untied` seja omitida, o comportamento padrão `tied` é assumido e a continuação da tarefa teria que ser executada na mesma *thread* que a iniciou. Isso pode limitar o paralelismo se tal *thread* estiver ocupada executando outras tarefas. Outras otimizações ainda seriam possíveis neste código, como o emprego de outros algoritmos de ordenação para limitar a recursividade. Entretanto, tais otimizações são independentes do OpenMP e podem ser aplicadas já no algoritmo sequencial, estando portanto fora do escopo deste texto.

```
1  void merge_sort(int vetor[], int tam) {
2    int metade = tam / 2;
3    if (tam > 1) {
4      #pragma omp task if(metade > MIN_PAR) untied
5      merge_sort(vetor, metade);
6      #pragma omp task if(metade > MIN_PAR) untied
7      merge_sort(vetor + metade, tam - metade);
8      #pragma omp taskwait
9      merge(vetor, tam);
10   }
11   return;
12 }
```

6.3.2. Suavização de Gauss-Seidel

Algumas estratégias de se obter desempenho em aplicações consistem em alterar a forma com a qual os laços do programa são percorridos. Técnicas de otimização de laços aninhados [McKinley et al. 1996] transformam os laços a fim de se explorar melhor a hierarquia de cache através da localidade dos dados como nas técnicas de `loop tiling` e `loop interchange`, ou permitem paralelizá-los como a técnica de `loop skewing`. No entanto, a implementação de algumas dessas técnicas, como a de `loop skewing`, não parece óbvia em um primeiro olhar. Também, sua implementação pode ser complexa, reduzindo a legibilidade do código portanto mais sujeita à erros. A programação baseada em tarefas simplifica esse tipo de problema.

O algoritmo de suavização de Gauss-Seidel representa um passo essencial para os métodos `multigrid` [Flannery et al. 1992], suavizando os erros de alta frequência e acelerando a convergência dos resultados. No entanto, tal algoritmo de suavização é considerado de difícil paralelização, pois o cálculo de um valor na posição i, j na iteração k de uma matriz, conforme pode ser visto no código a seguir, depende dos valores previamente calculados para as posições vizinhas à esquerda $(i-1, j)$ e acima $(i, j-1)$ pela iteração k, e dos valores atuais abaixo $(i+1, j)$ e à direita $(i, j+1)$ calculados na iteração $k-1$. A Figura 6.9 ilustra graficamente as dependências para o cálculo de um ponto i,j.

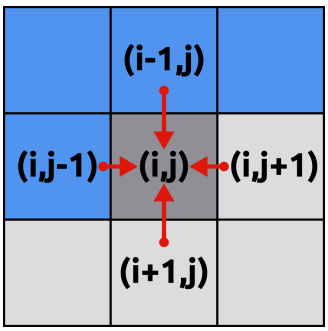

Figura 6.9. Dependências para o cálculo de um ponto i, j na suavização de Gauss-Seidel. Escrevemos em i, j, e lemos dos pontos vizinhos. Os pontos em azul já foram computados pela iteração atual, e os em cinza representam os valores calculados na iteração prévia ou os valores iniciais.

O comportamento da execução sequencial e as suas dependências são ilustradas na Figura 6.10, onde mostramos um exemplo com duas iterações do algoritmo, a primeira em azul e a segunda em amarelo. O percorrimento da matriz por linhas e colunas geram dependências que não permitem uma paralelização direta do algoritmo, tanto dos seus laços internos quanto o laço externo que controla o número de iterações do processo na matriz. Mesmo que algumas dependências sejam liberadas logo no começo, como após a execução da posição 1 que libera a posição 2 e 4, e estas posições liberam a posição 1 da segunda iteração em amarelo, expressar o paralelismo entre iterações parece bastante complexo.

Como mencionado anteriormente, existem algumas técnicas para permitir a exploração do paralelismo em casos específicos como a técnica de `loop skewing`. Usando essa técnica é possível parallelizar o laço interno de atualização de todas as posições de

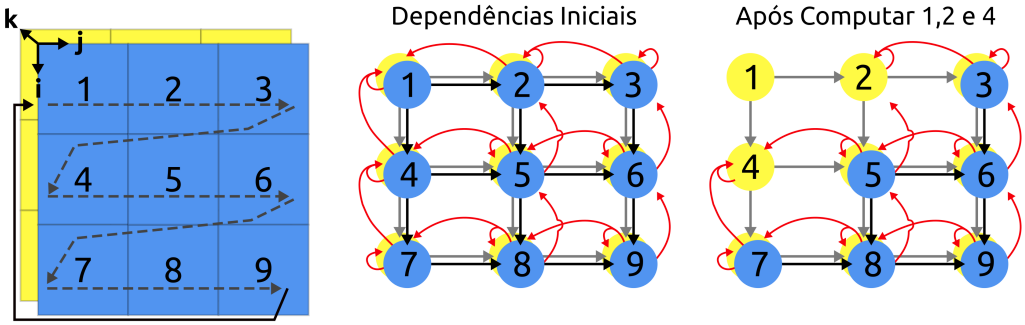

Figura 6.10. Ordem da execução sequencial e as dependências de dados entre as posições da matriz e duas iterações. A primeira iteração é representada em azul e a segunda em amarelo. Dependências entre a primeira e segunda iteração são representadas em vermelho.

uma diagonal percorrendo a matriz nesse sentido. Isso cria um nível de paralelismo dentro das diagonais que pode ser expressado usando diretivas clássicas como o `parallel for`, ilustrado pela Figura 6.11. O nível de paralelismo aumenta e diminui de acordo com a quantidade de elementos de uma diagonal, sendo bem limitado no início e final de cada iteração. Além disso, também é necessária uma sincronização após a computação de cada diagonal, e esta técnica não permite sobrebor as duas iterações (amarela e azul) de uma forma simples, o que também limita o paralelismo. O código para esta técnica é representado a seguir e mostra como pode ser complexo expressar o percorrimento da matriz diagonalmente. Isso torna a a programação mais difícil, menos eficiente, e mais propensa a erros, devido a complexidade das dependências impostas pelo problema.

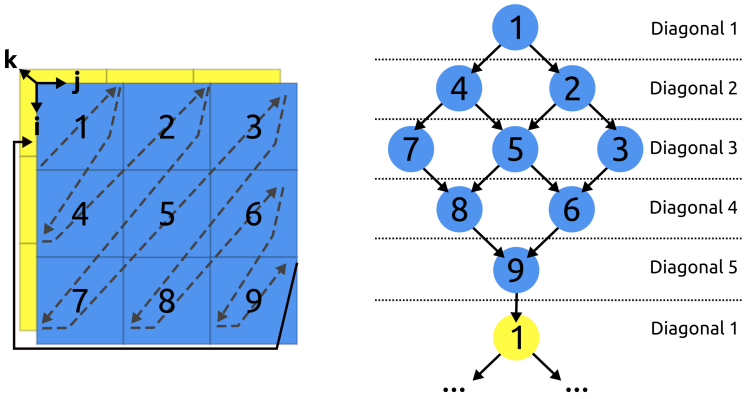

Figura 6.11. Método de Gauss-seidel usando a técnica de *loop skewing*.

```
1  void gauss_seidel_skewed(double **A, double* b, int N, int Niter)
2  {
3    double  h, h2;
4    h = 1.0/(N-1);
5    h2 = h*h;
6
7    for(int k=0; k<Niter; ++k) {
8      // primeiras N diagonais (paralelismo aumentando)
```

```
9     for (int diagonal=1; diagonal <= N; ++diagonal) {
10      #pragma omp parallel for shared(A, b, h2)
11      for (int j=1; j<=diagonal; ++j) {
12        int diag = diagonal+1-j;
13        // Fórmula para suavização de Gauss-Seidel em 2D
14        A[diag][jj] = 0.25 * ( A[diag+1][j] + // vizinho abaixo
15                               A[diag-1][j] + // vizinho acima
16                               A[diag][j+1] + // vizinho direita
17                               A[diag][j-1] - // vizinho esquerda
18                               h2*b[diag]
19                             );
20      } // sincronização entre diagonais
21    }
22
23    // resto das diagonais (paralelismo diminuindo)
24    for (int diagonal=1; diagonal <= N-1; ++diagonal) {
25      #pragma omp parallel for shared(A, b, h2)
26      for (int j=1; j<=N-diagonal; ++j) {
27        int jj = diagonal+j;
28        int diag = N+1-j;
29        // Fórmula para suavização de Gauss-Seidel em 2D
30        A[diag][jj] = 0.25 * ( A[diag+1][jj] + // vizinho abaixo
31                               A[diag-1][jj] + // vizinho acima
32                               A[diag][jj+1] + // vizinho direita
33                               A[diag][jj-1] - // vizinho esquerda
34                               h2*b[diag]
35                             );
36      } // sincronização entre diagonais
37    }
38  }
39 }
```

Dada a maior complexidade de se paralelizar este problema usando técnicas clássicas como a paralelização direta de laços, mostraremos como a programação baseada em tarefas pode ser bastante útil tanto em termos de expressividade do paralelismo, quanto desempenho comparado à solução anterior. O código abaixo já utiliza a otimização de `loop tiling`, percorrendo a matriz em blocos menores no mesmo sentido pela Figura 6.10. Para termos a execução baseada em tarefas, usamos exatamente o mesmo código da versão sequential com as seguintes modificações: 1) adiciona-se as diretivas das linhas 7 e 9 para termos uma região paralela executada por uma única *thread* (aquela que submeterá as tarefas); e 2) define-se a criação de uma tarefa por bloco da matriz usando a construção de tarefas do OpenMP nas linhas 15 a 19, especificando as dependências das posições vizinhas com a cláusula `depend`. A complexidade das dependências do problema é totalmente gerenciada pelo sistema de runtime. Isto simplificou muito a programação neste caso pois a paralelização é feita com uma mínima alteração no programa original, e ainda nos permite explorar o paralelismo em um nível maior, incluindo a execução de múltiplas iterações em paralelo.

```
1  void gauss_seidel_blocos(double **A, double* b, int N, int BS, int
       Niter) {
2    int NB = N / TS; // Número de blocos a partir do Block Size (BS)
3    double  h, h2;
4    h = 1.0/(N-1);
```

```
5   h2 = h*h;
6
7   #pragma omp parallel
8   {
9     #pragma omp single
10    {
11      for(int k=0; k<Niter; ++k) { // iterações de Gauss-Seidel
12        for (int ii=1; ii<N-BS; ii+=BS) { // laços sobre os blocos
13          for (int jj=1; jj<N-BS; jj+=BS) {
14            // cria uma tarefa para cada bloco da matriz
15            #pragma omp task depend(out: A[ii:BS][jj:BS]) depend(in: A[
    ii+BS:1][jj:BS], A[ii-1:BS][jj:BS], A[ii:BS][jj-1:BS], A[ii:BS][jj+
    BS:1]) firstprivate(ii, jj)
16            {
17              for(int i=ii; i<ii+BS; ++i) { // laços sobre um bloco
18                for(int j=jj; j<jj+BS; ++j) {
19                  // Fórmula para suavização de Gauss-Seidel em 2D
20                  A[i][j] = 0.25 * ( A[i+1][j] + // vizinho abaixo
21                                     A[i-1][j] + // vizinho acima
22                                     A[i][j+1] + // vizinho direita
23                                     A[i][j-1] - // vizinho esquerda
24                                     h2*b[i]
25                                   );
26                }
27              }
28            }
29          }
30        }
31      }
32    }
33  }
34  }
```

6.3.3. Fatoração de Cholesky

Algoritmos para fatoração de matrizes permitem decompor uma matriz no produto de duas outras matrizes. A fatoração serve de base para outros algoritmos, auxiliando, por exemplo, na resolução de sistemas de equações lineares. Nesta seção, abordamos o algoritmo de Cholesky para fatorar uma matriz simétrica positiva-definida A em uma matriz triangular L e sua transposta L^T ($A = LL^T$).

Dentre as várias maneiras possíveis de se implementar esta fatoração, escolhemos aqui um algoritmo em blocos [Buttari et al. 2009, Agullo et al. 2010]. Tal algoritmo tem por base quatro operações do padrão BLAS: `potrf`, `trsm`, `syrk` e `gemm`. Estas quatro operações são aplicadas em blocos (submatrizes) da matriz original. Para paralelizar esta aplicação, criaremos uma nova tarefa para executar cada uma das operações BLAS citadas anteriormente, conforme ilustrado nas linhas 13, 17, 23, 29 do código abaixo. Note que utilizamos a anotação `firstprivate` para os ponteiros `Akk`, `Aik`, `Aii`, `Aij` e `Ajk`, uma vez que cada tarefa trabalha sobre blocos diferentes da matriz. Cada operação acessa os blocos da matriz em modos de leitura, escrita ou leitura/escrita. A operação `gemm`, por exemplo, acessa três blocos A, B, C. Os blocos A e B são acessados em modo leitura enquanto o bloco C é acessado em modo leitura/escrita. Por meio da diretiva `depend`

podemos indicar como cada tarefa acessa cada bloco da matriz. Utilizando a sequência de desenrolamento dos laços e o modo de acesso em cada bloco da matriz, é possível construir um grafo de dependências (DAG). A Figura 6.12 ilustra um DAG para esta aplicação quando a matriz *A* é particionada em quatro blocos por linha/coluna.

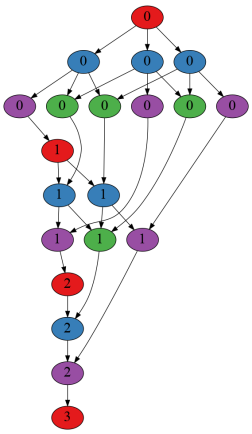

Figura 6.12. DAG da fatoração de cholesky por blocos quando número de blocos por linha/coluna (N) é 4. As cores dos nós representam o tipo da tarefa. Vermelho indica tarefas `potrf`, azul tarefas `trsm`, roxo tarefas `syrk` e verde tarefas `gemm`.

```
1  void cholesky(double *A, int OrdemMatriz, int OrdemBloco){
2    int i, j, k;
3    int NumBlocos = OrdemMatriz / OrdemBloco;
4    int TamBloco = OrdemBloco*OrdemBloco;
5    double *Akk, *Aik, *Aii, *Aij, *Ajk;
6
7    #pragma omp parallel
8    {
9      #pragma omp single
10     {
11       for(k = 0; k < NumBlocos; k++) {
12         Akk = &ELEM_BLOCO(A, OrdemBloco, NumBlocos, k, k);
13         #pragma omp task depend(inout: Akk[0:TamBloco]) firstprivate(
     Akk)
14         potrf(Akk, OrdemBloco);
15         for(i = k+1; i < NumBlocos; i++) {
16           Aik = &ELEM_BLOCO(A, OrdemBloco, NumBlocos, i, k);
17           #pragma omp task depend(in: Akk[0:TamBloco]) depend(inout:
     Aik[0:TamBloco]) firstprivate(Akk, Aik)
18           trsm(Akk, Aik, OrdemBloco);
19         }
20         for(i = k+1; i < NumBlocos; i++) {
21           Aii = &ELEM_BLOCO(A, OrdemBloco, NumBlocos, i, i);
22           Aik = &ELEM_BLOCO(A, OrdemBloco, NumBlocos, i, k);
23           #pragma omp task depend(in: Aik[0:TamBloco]) depend(inout:
     Aii[0:TamBloco]) firstprivate(Aii, Aik)
24           syrk(Aik, Aii, OrdemBloco);
25           for(j = k+1; j < i; j++) {
26             Aij = &ELEM_BLOCO(A, OrdemBloco, NumBlocos, i, j);
27             Aik = &ELEM_BLOCO(A, OrdemBloco, NumBlocos, i, k);
```

```
28              Ajk = &ELEM_BLOCO(A, OrdemBloco, NumBlocos, j, k);
29              #pragma omp task depend(in: Aik[0:TamBloco]) depend(in: Ajk
    [0:TamBloco]) depend(inout: Aij[0:TamBloco]) firstprivate(Aij, Aik,
     Ajk)
30              gemm(Aik, Ajk, Aij, OrdemBloco);
31            }
32          }
33        }
34      }
35    }
36  }
```

Analisando o grafo da Figura 6.12, percebemos que alguns tipos de tarefas mais arestas de dependências que outros. O número de arestas que saem de um dado nó indica quantas outras tarefas dependem da execução desta. Tarefas do tipo `syrk` (em roxo) ou `gemm` (em verde) sempre liberam a execução de apenas uma outra tarefa enquanto tarefas do tipo `potrf` (em vermelho) liberam $(NumBlocos - k - 1)$ tarefas `trsm` (em azul), onde *NumBlocos* é o número de blocos por linha/coluna da matriz original e k é iteração do laço mais externo do algoritmo por blocos. A execução de cada uma das tarefas `trsm` por sua vez, habilita a execução de até $(NumBlocos - k - 1)$ tarefas dos tipos `syrk` e `gemm`. Logo, se priorizarmos a execução das tarefas `potrf` sobre tarefas `gemm` ou `syrk` habilitaremos mais rapidamente um maior número de tarefas, expondo maior nível de paralelismo ao escalonador.

Na Seção 6.2, nós vimos que prioridades é um bom meio para repassar dicas ao *runtime* do OpenMP sobre quais tarefas escalonar primeiro. No código abaixo, adicionamos prioridades a cada uma das tarefas do algoritmo seguindo a lógica discutida acima. A fórmula utilizada no cálculo destes valores de prioridade para cada tipo de tarefa é livremente inspirada naquela utilizado pela biblioteca de álgebra linear Chameleon [Agullo et al. 2010].

```
1  void cholesky(double *A, int OrdemMatriz, int OrdemBloco){
2    int i, j, k, prio;
3    int NumBlocos = OrdemMatriz / OrdemBloco, TamBloco = OrdemBloco*
      OrdemBloco;
4    double *Akk, *Aik, *Aii, *Aij, *Ajk;
5
6    #pragma omp parallel
7    {
8      #pragma omp single
9      {
10       for(k = 0; k < NumBlocos; k++) {
11         Akk = &ELEM_BLOCO(A, OrdemBloco, NumBlocos, k, k);
12         prio = MAX_PRIO - 2*NumBlocos - 2*k;
13         #pragma omp task depend(inout: Akk[0:TamBloco]) firstprivate(
     Akk) priority(prio)
14         potrf(Akk, OrdemBloco);
15         for(i = k+1; i < NumBlocos; i++) {
16           Aik = &ELEM_BLOCO(A, OrdemBloco, NumBlocos, i, k);
17           prio = MAX_PRIO -  2*NumBlocos - 2*k - i;
18           #pragma omp task depend(in: Akk[0:TamBloco]) depend(inout:
     Aik[0:TamBloco]) firstprivate(Akk, Aik) priority(prio)
19           trsm(Akk, Aik, OrdemBloco);
```

```
20          }
21          for(i = k+1; i < NumBlocos; i++) {
22            Aii = &ELEM_BLOCO(A, OrdemBloco, NumBlocos, i, i);
23            Aik = &ELEM_BLOCO(A, OrdemBloco, NumBlocos, i, k);
24            prio = MAX_PRIO - 2*NumBlocos - 2*k - i;
25            #pragma omp task depend(in: Aik[0:TamBloco]) depend(inout:
    Aii[0:TamBloco]) firstprivate(Aii, Aik) priority(prio)
26            syrk(Aik, Aii, OrdemBloco);
27            for(j = k+1; j < i; j++) {
28              Aij = &ELEM_BLOCO(A, OrdemBloco, NumBlocos, i, j);
29              Aik = &ELEM_BLOCO(A, OrdemBloco, NumBlocos, i, k);
30              Ajk = &ELEM_BLOCO(A, OrdemBloco, NumBlocos, j, k);
31              prio = MAX_PRIO - 2*NumBlocos - 2*k - i - j;
32              #pragma omp task depend(in: Aik[0:TamBloco]) depend(in: Ajk
    [0:TamBloco]) depend(inout: Aij[0:TamBloco]) firstprivate(Aij, Aik,
     Ajk) priority(prio)
33              gemm(Aik, Ajk, Aij, OrdemBloco);
34            }
35          }
36        }
37      }
38    }
39  }
```

6.4. Análise de Desempenho de Tarefas OpenMP

O desenvolvimento de aplicações paralelas está ligado ao constante uso de técnicas de análise de desempenho. Nesse sentido, uma das funcionalidades associada às especificações do OpenMP é a chamada *OpenMP Tools Interface* (OMPT). Estas especificações atualmente ainda não são suportadas pelo *runtime* OpenMP do compilador GCC `libgomp`, mas é suportada por outros compiladores tal como o `LLVM/clang` com o runtime `libomp`. OMPT objetiva especificar uma interface de *callbacks* para rastrear o comportamento de aplicações OpenMP. A vantagem dessa API é que ela roda no mesmo espaço de endereçamento que o *runtime* OpenMP, facilitando o acesso a diversas informações sobre o estado do *runtime*.

Os *callbacks* permitem acessar dados de uma *thread* tais como seu estado atual (ex: *Idle*, *Work*, *Barrier wait*, *Overhead*, etc). Estes *callbacks* também nos permitem interceptar eventos, algo útil para a análise de programas baseados em tarefas. Destacamos dois: a criação de tarefas (`ompt_callback_task_create`) e ações de escalonamento sobre as tarefas (`ompt_callback_task_schedule`), definidos no código abaixo. Ambos são associados aos eventos correspondentes na função `ompt_initialize` de inicialização. Além disso, nossa ferramenta deve implementar o método `ompt_start_tool` que realizará a sua inicialização. Com a implementação dos *callbacks* podemos especificar o que a nossa ferramenta fará quando uma tarefa for criada ou um ponto de escalonamento for atingido.

O código abaixo mostra a definição do comportamento da ferramenta para o *callback* do ponto de escalonamento. A variável `prior_task_status` define o estado da tarefa `first_task_data` que chegou no ponto de escalonamento. O último parâmetro, `second_task_data`, contém dados da tarefa que irá assumir a execução após o

ponto de escalonamento. Neste exemplo, apenas escrevemos na tela o identificador da *thread* que estava executando aquela tarefa, o identificador da tarefa, o tipo de evento que ocorreu e uma marca de tempo de quando o evento ocorreu. A ferramenta é finalizada quando a função `ompt_finalize` é chamada.

```
#include <stdio.h>
#include <ompt.h>
#include <omp.h>
double INIT_TIME;
unsigned int TASK_ID;
// Define o comportamento para este callback
static void on_ompt_callback_task_schedule(ompt_data_t *first_task_data
    , ompt_task_status_t prior_task_status, ompt_data_t *
    second_task_data){
  switch (prior_task_status) {
    case ompt_task_complete:
      printf("%d %ld tarefa_terminou %lf\n", omp_get_thread_num(),
    first_task_data->value, omp_get_wtime()-INIT_TIME);
      break;
    case ompt_task_switch: // task second_task_data começou
      printf("%d %ld tarefa_iniciou %lf\n", omp_get_thread_num(),
    second_task_data->value, omp_get_wtime()-INIT_TIME);
      break;
    default:
      break;
  }
}
static void on_ompt_callback_task_create( ompt_data_t *parent_task_data
    ,
      const ompt_frame_t *parent_frame, ompt_data_t* new_task_data,
      int flag, int has_dependences, const void *codeptr_ra)
{
  new_task_data->value = TASK_ID++;
  printf("%d %ld tarefa_criada %lf\n", omp_get_thread_num(),
    new_task_data->value, omp_get_wtime()-INIT_TIME);
}
int ompt_initialize(ompt_function_lookup_t lookup, ompt_data_t* data) {
  printf("%d 0 OMPT_LIB_INICIO %lf\n", omp_get_thread_num(),
    omp_get_wtime()-INIT_TIME);
  // Registra os callbacks que queremos usar
  ompt_set_callback_t ompt_set_callback = (ompt_set_callback_t) lookup(
    "ompt_set_callback");
  ompt_set_callback(ompt_callback_task_schedule, (ompt_callback_t)
    on_ompt_callback_task_schedule);
  ompt_set_callback(ompt_callback_task_create, (ompt_callback_t)
    on_ompt_callback_task_create);
  return 1; //successo
}
void ompt_finalize(ompt_data_t* data) {
  printf("%d 0 OMPT_LIB_FIM %lf\n", omp_get_thread_num(), omp_get_wtime
    ()-INIT_TIME);
}
ompt_start_tool_result_t* ompt_start_tool( unsigned int omp_version,
    const char *runtime_version) {
  INIT_TIME = omp_get_wtime();
```

```
  TASK_ID = 0;
  static ompt_start_tool_result_t ompt_start_tool_result = {&
    ompt_initialize, &ompt_finalize};
  return &ompt_start_tool_result;
}
```

A ferramenta de rastreamento acima pode ser compilada separadamente, criando uma biblioteca que podemos linkar a qualquer programa OpenMP que desejarmos rastrear como exemplificado no processo de compilação a seguir, ou também pode ser adicionada diretamente ao programa. Caso queiramos compilar o código usando o `gcc` devemos especificar o caminho para um *runtime* que suporte as especificações do OMPT, ou podemos simplemente compilar a aplicação usando o compilador `clang`, que usa o runtime `libomp`.

```
# criamos nossa biblioteca para rastrear códigos OpenMP usando OMPT
clang-10 -fopenmp minha_ferramenta_ompt.c -shared -fPIC -o libompt.so
# compilamos a nossa aplicação OpenMP
clang-10 -o GaussSeidel Gauss-Seidel.c -fopenmp -O3
# especificamos a variavél de ambiente OMP_TOOL_LIBRARIES com a nossa
   biblioteca para gerar o rastro da execução
env OMP_TOOL_LIBRARIES=$(pwd)/libompt.so ./GaussSeidel < input
```

Com isto, podemos gerar rastros da execução de diferentes aplicações, o que enriquece o processo de análise de desempenho, permitindo compreender melhor o comportamento de uma aplicação. Como exemplo, a Figura 6.13 representa a execução para a aplicação da suavização de Gauss-Seidel apresentada anteriormente. Podemos ver como o paralelismo de tarefas permite explorar bem os recursos mesmo em um cenário onde temos um problema com dependências bastante complexas.

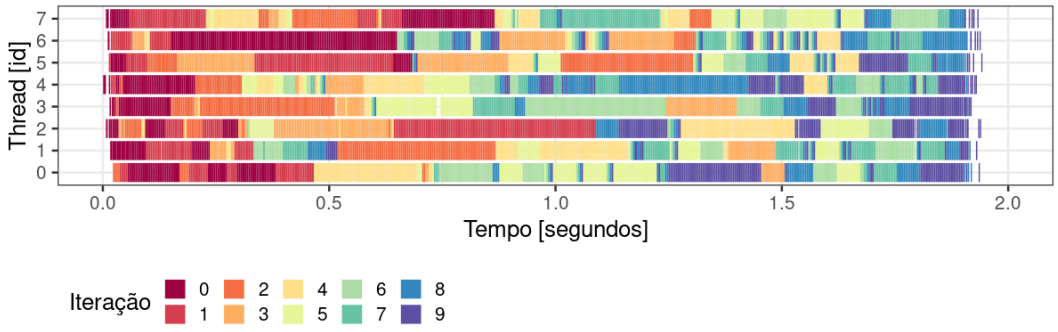

Figura 6.13. Rastro da execução paralela de diferentes iterações para o método de Gauss Seidel usando o paralelismo em tarefas.

6.5. Conclusão

Este minicurso abordou os conceitos básicos fundamentais para se construir programas paralelos com diretivas de programação utilizando tarefas OpenMP. O enfoque no grafo de tarefas (DAG) permite o estabelecimento de dependência finas entre as tarefas permitindo a construção de programas cujo grão de paralelismo pode facilmente ser explorado

para criar uma grande quantidade de tarefas capazes de ocupar todas as *threads* de execução. Esperamos que com os exemplos fornecidos os leitores possam incorporar este paradigma de programação, claramente mais fácil tendo em vista que uma boa parte do trabalho (balanceamento de carga, escalonamento) é relegado ao ambiente de execução. Os conceitos apresentados aqui já permitem começar a programação com tarefas. Referenciamos o leitor à especificação OpenMP [OpenMP 2020], que porta uma enorme quantidade de diretivas auxiliares para tratar casos mais específicos.

6.6. Agradecimentos

Este trabalho foi realizado com o apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) - Finance Code 001, do Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) com a bolsa 141971/2020-7 para o primeiro autor, 131347/2019-5 para o segundo autor, e dos projetos: FAPERGS ReDaS (19/711-6), MultiGPU (16/354-8) e GreenCloud (16/488-9), do projeto CNPq 447311/2014-0, do projeto CAPES/Brafitec 182/15 e CAPES/Cofecub 899/18, e com apoio do projeto Petrobras (2018/00263-5).

Referências

[Agullo et al. 2010] Agullo, E. et al. (2010). Faster, cheaper, better – a hybridization methodology to develop linear algebra software for GPUs. In mei W. Hwu, W., editor, *GPU Computing Gems*, volume 2. Morgan Kaufmann. páginas

[Augonnet et al. 2011] Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. (2011). StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Conc. and Comp.: Pract. and Exp.*, 23(2). páginas

[Bosilca et al. 2012] Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., and Dongarra, J. (2012). DAGuE: A generic distributed {DAG} engine for high performance computing. *Parallel Computing*, 38(1–2):37 – 51. Extensions for Next-Generation Parallel Programming Models. páginas

[Buttari et al. 2009] Buttari, A., Langou, J., Kurzak, J., and Dongarra, J. (2009). A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53. páginas

[Duran et al. 2011] Duran, A., Ayguadé, E., Badia, R. M., Labarta, J., Martinell, L., Martorell, X., and Planas, J. (2011). OmpSs: a proposal for programming heterogeneous multi-core architectures. *Par. Proc. Letters*, 21(02). páginas

[Flannery et al. 1992] Flannery, B. P., Press, W. H., Teukolsky, S. A., and Vetterling, W. (1992). Numerical recipes in c. *Press Syndicate of the University of Cambridge, New York*, 24(78):36. páginas

[McKinley et al. 1996] McKinley, K. S., Carr, S., and Tseng, C.-W. (1996). Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(4):424–453. páginas

[OpenMP 2020] OpenMP (2020). OpenMP application program interface version 5.1. Disponível em: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf. páginas

[Rico et al. 2019] Rico, A., Sánchez Barrera, I., Joao, J. A., Randall, J., Casas, M., and Moretó, M. (2019). On the benefits of tasking with openmp. In Fan, X., de Supinski, B. R., Sinnen, O., and Giacaman, N., editors, *OpenMP: Conquering the Full Hardware Spectrum*, pages 217–230, Cham. Springer International Publishing. páginas