Capítulo 2

Ferramentas modernas para programação multithread

Gerson Geraldo H. Cavalheiro, André Rauber Du Bois

Abstract

With the rise of multiprocessor architectures, parallel programming has grown in importance in the software development scene. For this kind of architecture there many different options of parallel programming tools. However, when choosing the right tool for solving a problem, one must consider its expressive power as well as its execution strategy. In this Chapter, some of the most used multithreaded tools are presented: Pthreads, C++11, OpenMP, Cilk Plus and TBB. The presentation is given in terms of the development facilities provided, featuring abstractions for describing parallelism and execution strategy. Special attention is given to the features that differentiate each tool. The objective is to provide programmers with knowledge to choose the right tool for the right context.

Resumo

O uso de técnicas de programação paralela cresceu em importância no cenário de desenvolvimento de software com a popularização das arquiteturas multiprocessadas. Para este tipo de arquitetura, são várias as opções de ferramentas de programação disponíveis. No entanto, a escolha entre uma ou outra para implementar uma determinada aplicação deve considerar tanto seu poder de expressão como sua estratégia de execução. Neste capítulo são apresentadas algumas das mais utilizadas ferramentas de programação multithread: Pthreads, C++11, OpenMP, Cilk Plus e TBB. Esta apresentação se dá em termos das facilidades de desenvolvimento que oferecem, caracterizando suas abstrações para descrição de paralelismo e estratégia de execução. Destaque é dado para os recursos que diferenciam as ferramenta entre si. O objetivo do texto é o de permitir o programador diferenciar as ferramentas apresentadas em termos de suas vocações e facilidades oferecidos.

2.1. Introdução

O uso de recursos de programação paralela era tido, até pouco tempo, pela maioria dos programadores como uma característica desejável, porém não essencial, em seus programas. Esta posição tem mudado com a grande oferta de hardware paralelo desde o lançamento da tecnologia de processadores multicore. Atualmente esta tecnologia é praticamente onipresente em sistemas computacionais de qualquer porte e o usuário final entende que dispõe de recursos de processamento paralelo, frustrando-se quando seu investimento, em hardware, não é explorado de forma efetiva pelo software que ele necessita. A consequência é a oferta de diversas ferramentas de programação multithread, permitindo que os programadores selecionem aquela mais apta à implementação de seu problema considerando as características de programação e de execução por elas oferecidos. Portanto, a questão que se coloca é a de conhecer estas ferramentas para optar pela mais indicada. Se faz necessário, então, capacitar os programadores com habilidades para explorar de forma efetiva este tipo de arquitetura ([Vaj11, AR06, CdS07]).

Em relação à interface de programação, os recursos de programação disponíveis nas ferramentas multithread modernas são pouco comparáveis aqueles oferecidos pelo tradicional Pthreads. E isto é verificado já na concepção destas ferramentas. Enquanto que a especificação de Pthreads, tal como um padrão, foi necessária para compatibilizar produtos de diferentes fornecedores, as novas ferramentas foram projetadas considerando critérios de expressividade e eficiência na descrição da concorrência de aplicações. Outra diferença marcante é que usualmente a apresentação de Pthreads inicia por contrastar *threads* com *processos*. Já nas novas abordagens, embora o termo thread ainda seja empregado, é comum se referir às unidades de trabalho como *tarefas*, sendo relevante sua comparação com outros recursos de abstração em programas, como *comandos compostos*/*blocos* e *procedimentos*/*funções*. Nestas novas abordagens, os diferenciais se apresentam em termos de vocação de cada ferramenta para representar adequadamente um determinado conjunto de estruturas de relacionamentos entre tarefas. Esta vocação é resultado do projeto da sua interface de programação ter sido realizado tendo em vista o modelo de execução adotado.

O modelo de execução, por sua vez, reflete a estratégia de escalonamento adotada pela ferramenta. Este escalonamento é realizado em nível aplicativo, ou seja, pelo próprio suporte de execução. Nesta abordagem, a estratégia de escalonamento aplica um conjunto de hipóteses sobre as tarefas geradas pelo programa para determinar o mapeamento destas sobre os recursos de processamento disponíveis. Como o objetivo é reduzir o tempo total de execução, o escalonador será melhor sucedido quando a concorrência de um determinado programa refletir estas hipóteses.

Conhecendo as diferentes interfaces de programação e estratégias de execução, o programador poderá selecionar a ferramenta de programação mais adequada para implementação de sua aplicação. Na sequência são apresentadas quatro ferramentas de programação multithread: C++11, OpenMP, Cilk Plus e TBB. A interface destas ferramentas é apresentada em termos de seus recursos de programação. A intenção não é abordar de forma exaustiva todas as funcionalidades destas ferramentas, mas sim discutir o uso dos recursos que permitem construir programas básicos e daqueles que diferenciam as ferramentas entre si. Outro aspecto discutido é o escalonamento por elas

adotado, de forma que o leitor compreenda as premissas assumidas nas diferentes abordagens e suas consequências no desempenho de execução e na especificação da interface de programação. Inicialmente, alguns fundamentos da programação concorrente são apresentados para uniformizar a terminologia utilizada.

2.2. Fundamentos para programação concorrente

A diferença entre programação concorrente e programação paralela é bastante discutida na literatura [SMR10, Bre09, NBF96]. Enquanto que a programação concorrente é voltada para aspectos ligados à descrição de atividades concorrentes de uma aplicação, a programação paralela está ligada a exploração eficiente do hardware disponível. Neste sentido, a expressão mais apropriada neste texto é *programação concorrente*, uma vez que as modernas ferramentas de programação multithread oferecem recursos para descrever a concorrência da aplicação, delegando a um núcleo de escalonamento a responsabilidade de explorar efetivamente os recursos de processamento disponíveis.

Na sequência desta seção são apresentadas as arquiteturas multiprocessadas e os principais conceitos associados à programação multithread utilizando como exemplo Pthreads. A seção termina apresentando modelos de implementação de ambientes de execução multithread.

2.2.1. Arquiteturas multiprocessadoras

Uma arquitetura multiprocessada possui dois ou mais processadores compartilhando o uso de um espaço de endereçamento comum (representação esquemática na Figura 2.1). Este espaço de endereçamento não está, necessariamente, contido em um módulo único de memória, podendo estar distribuído entre diferentes módulos. Classificam-se assim os multiprocessadores como UMA (*uniform memory access*) e NUMA *non-uniform memory access*), respectivamente. Em ambos os casos, o acesso à memória se dá por operações do tipo `load` e `store` convencionais e as posições físicas de memória possuem o mesmo endereço, independente do processador que o acessa. A diferença no tempo de acesso de cada processador a qualquer endereço será constante no caso do espaço de endereçamento estar contido em um módulo único de memória. No caso em que o espaço de endereçamento está distribuído entre diversos módulos, a situação que se apresenta é de que os tempos de acesso podem variar conforme o processador e o endereço de memória acessado.

Neste tipo de arquitetura, a escalabilidade, representada pelo incremento no número de processadores, é limitada pela capacidade do meio de transferência de dados entre os processadores e a memória. Seja pelo uso de um barramento comum, no caso das arquiteturas UMA, seja pelo uso de uma rede de comunicação provida em arquiteturas NUMA. Assim, o efeito ”gargalo de execução”refletido pelo acesso à memória no modelo von Neumann é ainda mais pronunciado. No caso das arquiteturas NUMA, a própria configuração em rede para acesso aos módulos de memória oferece diferentes caminhos atenuando o problema. Outra estratégia amplamente utilizada é a exploração de diferentes níveis de cache, compartilhados, ou não, entre os processadores. Observe-se que uma arquitetura multiprocessada é dita simétrica (SMP, *symetric multiprocessor*), quando o espaço de endereçamento e o(s) caminho(s) para acessá-lo são compartilhados e o con-

trole é realizado por um sistema operacional único.

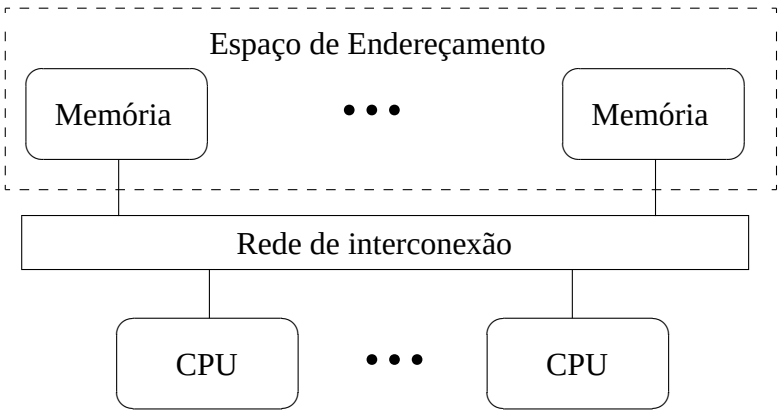

Figura 2.1. Estrutura de um multiprocessador com memória compartilhada.

A tecnologia multicore oferece a incorporação de dois ou mais núcleos de processamento completos em um único chip. Portanto, trata-se de um multiprocessador, havendo instâncias de arquiteturas UMA e NUMA. O uso de processadores multicores tem permitido a construção de diferentes opções de computadores multiprocessados a custos relativamente baixos, considerando o desempenho que oferecem. Outra tecnologia que merece ser citada neste contexto é a SMT (*simultaneous multi-threading*), popularmente conhecida como *hiperthreading*). Esta tecnologia é adotada em diversos processadores multicore, podendo ser habilitada ou não via software. Esta tecnologia incorpora no hardware estruturas que permitem a manipulação eficiente de dois fluxos de execução (threads) simultaneamente, reduzindo o overhead de trocas de contexto entre eles.

2.2.2. A multiprogramação leve

No nível do sistema operacional, um processo é contido em um espaço de endereçamento lógico próprio e seu estado de execução mantido em um bloco de controle de processo. A manipulação deste processo é custosa para o sistema operacional, pois envolve a manipulação de uma extensa porção de memória quando o escalonamento é realizado. Caso uma aplicação concorrente seja descrita em termos de processos que colaboram entre si, os custos envolvidos são ainda maiores, pois operações que envolvam dois ou mais processos devem ser moderadas pelo sistema operacional, embutindo sobrecustos extras. Assim, o uso de múltiplos fluxos de execução, denominados threads ou processos leves, em único processo desonera o sistema operacional tornando a execução mais eficiente. Estes threads compartilham praticamente toda a descrição do processo, em particular as áreas de memória para os dados estáticos e os alocados dinamicamente. Para cada thread é mantida, com exclusividade, uma pilha própria para os registros de ativação de funções e o estado dos registradores. A comunicação entre os threads também é mais eficiente, pois pode ser realizada com simples acessos, em leitura e escrita, a dados nas áreas de memória compartilhadas.

As primitivas básicas na programação multithread permitem a criação de novos fluxos de execução (threads) e a sincronização entre threads. Também considera-se que um thread tem um tempo de vida efêmero, associado ao tempo necessário para executar a

sequência de instruções para ele descrita. Assim, o número de threads em execução não é constante em um determinado programa.

Ao utilizar estas primitivas o programador deve atentar a alguns aspectos não considerados na programação sequencial. Um destes aspectos é a granulosidade. Um programa é dito de granularidade grossa quando a quantidade de cálculo executada entre cada operação de sincronização for grande, e fina quando houver pouco cálculo entre as sincronizações. Observar a granularidade implica em atentar para os efeitos da sobrecarga de execução gerada pelas sincronizações frequentes. Outros aspectos a considerar estão relacionados à má codificação dos algoritmos considerando a dimensão de execução concorrente do programa. Entre os erros mais comuns encontramos a postergação indefinida (*deadlock*) e as condições de corrida. A situação de *deadlock* se apresenta quando um thread indica a necessidade de que uma condição de sincronização seja satisfeita mas, no algoritmo, existe a possibilidade de que nenhum outro thread satisfaça esta condição. Neste caso, fatalmente o programa não terminará de forma correta, uma vez que um thread permanecerá bloqueado indefinitivamente. As condições de corrida, por sua vez, ocorrem quando dois ou mais threads não realizam a sincronização necessária para coordenar o acesso a um dado compartilhado. Neste caso, pode ocorrer inconsistência no valor armazenado, uma vez que este pode estar corrompido por escritas simultâneas.

2.2.3. Escalonamento de programas multithread

Tipicamente as ferramentas de programação multithread são implementadas em um de três modelos: $N \times 1$, 1×1 ou $N \times M$. O modelo $N \times 1$ oferece o recurso de descrever a concorrência da aplicação em termos de threads escalonadas, em nível usuário, em uma única unidade de escalonamento do sistema operacional. Esta unidade de escalonamento é que é de fato mapeada sobre processador da arquitetura. Desta forma, portanto, não existe paralelismo real na execução dos threads gerados pelo programa. Já no modelo 1×1, cada thread criado pelo programa consiste em uma unidade de escalonamento manipulada pelo sistema operacional, havendo desta forma execução paralela das atividades geradas pelo programa. No entanto a possibilidade de que a concorrência da aplicação seja maior que o paralelismo suportado pelo hardware faz com que o sobre-custo de manipulação destes threads não justifique a execução paralela. O modelo $N \times M$ oferece um meio termo, onde os N threads gerados pelo programa são escalonados, em nível usuário, sobre M unidades de escalonamento manipuladas pelo sistema operacional.

As vantagens do modelo $N \times M$ estão relacionadas a separação entre a descrição da concorrência da aplicação e a exploração do paralelismo disponível. Neste caso, estratégias de escalonamento são aplicadas, em nível aplicativo, para gerir a execução dos threads gerados pelo programa de forma a otimizar algum índice de desempenho, tipicamente o tempo de execução. Não raro estas estratégias de escalonamento impõem restrições sobre a interface de programação, como limitar os mecanismos de sincronização oferecidos. Nas ferramentas que implementam este modelo, as N atividades concorrentes geradas pelo programa são referenciadas como threads usuário, tarefas, fibras ou filamentos. As M unidades de escalonamento recebem denominações como thread sistema, processador virtual ou simplesmente thread. Para homogeneizar estes texto, os termos utilizados são tarefas e processadores virtuais.

2.3. Pthreads

Pthreads (POSIX threads [NBF96]) é um exemplo clássico para programação multithread e define um padrão de interface para a linguagem C. Este padrão foi proposto oficialmente em 1995, com o objetivo de padronizar a interface e permitir a colaboração entre produtos de diferentes fornecedores. Pthreads são fornecidas na forma de uma biblioteca pré-compilada, ligada aos arquivos objetos da aplicação para compor o executável.

2.3.1. Interface de programação

Em Pthreads o código dos threads é apresentado na forma de uma função C convencional. Como não existe relação entre os dados manipulados pelo programa de aplicação e aqueles manipulados na biblioteca, a função que contém o código a ser executado por um thread está restrita a receber um único parâmetro de entrada, representando um endereço de memória do tipo `void *`, e a retornar também um endereço de memória do tipo `void *`. A compatibilização dos tipos de dados manipulados pelo programa com o tipo `void` comunicado com a biblioteca Pthreads deve ser feita por meio do uso do recurso de coerção explícita de tipo (*cast*). Importante frisar que os ponteiros citados devem corresponder a endereços nas memórias compartilhadas entre os threads, nunca aos dados presentes na pilha privada a cada thread, pois o tempo de vida da pilha é limitado pelo tempo de vida do thread ao qual ela pertence. Quando o thread termina sua execução, a pilha correspondente é liberada e se, eventualmente, endereços para dados nesta pilha foram comunicados a outros threads não haverá nenhum mecanismo para garantir sua consistência.

Na Figura 2.2 é apresentado um exemplo de programa Pthreads. A função `Fibo` calcula, de modo sequencial e recursivo, o valor de uma posição na série de Fibonacci. O exemplo expõe o baixo acoplamento dos dados entre a biblioteca Pthreads e a linguagem hospedeira, havendo a necessidade de explicitar tanto a alocação dos dados como conversões de tipo.

As primitivas `pthread_create` e `pthread_join` oferecem recursos para criação e sincronização entre threads. Além do endereço da função que deve ser executada pelo thread a ser criado e do endereço de memória que contém os dados de entrada para esta função, a primitiva de criação recebe o endereço de memória de duas instâncias de tipos de dados opacos fornecidos por Pthreads: o de uma variável `pthread_t`, como saída, que conterá o identificar único do thread criado e outro de uma variável `pthread_attr_t`, como entrada, que contém atributos desejados, como tamanho da pilha e prioridade, para execução do novo thread. Estes tipos de dados são ditos *opacos* pois os campos que eles possuem não devem ser acessados diretamente, apenas por meio de primitivas de manipulação previstas pela interface – utiliza-los de forma direta compromete a portabilidade do programa. O identificador único armazenado em `pthread_t` nomeia o thread e permite identifica-lo em uma operação de sincronização feita por `pthread_join`. Os atributos caracterizam propriedades desejadas para execução do thread, como tipo e prioridade de escalonamento e tamanho da pilha. Quando o controle da execução retorna para a primeira instrução subsequente à criação de um thread, deve-se assumir que o thread foi criado e que a execução do thread corrente com o novo thread se dá de forma concorrente.

```
1  #include <pthread.h>
2  #include <malloc.h>
3  #include <stdio.h>
4
5  int n = 20;
6
7  int FiboSeq( int n ) {
8   return (n<2)?n:FiboSeq(n-1)+FiboSeq(n-2);
9  }
10 void* Fibo(void* dta) {
11  int* r = (int*)malloc(sizeof(int));
12  *r = FiboSeq(*(int*)dta);
13  return r;
14 }
15 int main(int argv, char** argc) {
16  pthread_t tid;
17  int *r;
18
19  pthread_create(&tid,NULL,Fibo,(void*)&n);
20  pthread_join(tid,(void**)&r);
21  printf("Fibonacci(20) = %d\n", *r);
22  free(r);
23  return 0;
24 }
```

Figura 2.2. Exemplo de criação e sincronização de threads em Pthreads.

A primitiva de sincronização `pthread_join`, além do identificador do thread a ser sincronizado, recebe como parâmetro o endereço de memória de um ponteiro para uma área de memória do tipo `void`. Este ponteiro será atualizado com o endereço de memória retornado, o qual deverá conter o dado retornado pelo thread sincronizado. Na semântica da operação *join* de Pthreads, quando o controle de execução retornar para a primeira instrução após o `pthread_join`, o thread sincronizado está terminado. Duas variantes são introduzidas em implementações de Pthreads, sendo estas não portáveis, para aumentar o potencial de paralelismo na sincronização entre threads: `pthread_tryjoin_np` e `pthread_timedjoin_np`. Caso o valor retornado por estas primitivas seja 0 (zero), significa que de fato o thread indicado terminou. Caso contrário, como estas primitivas são não bloqueantes, o thread indicado não foi terminado. A diferença entre estas duas primitivas é que *tryjoin* realiza uma tentativa de uma sincronização retornando imediatamente, enquanto que *timedjoin* aguarda que o thread termine em um intervalo de tempo indicado por um parâmetro extra.

Outro cuidado a ser tomado em Pthreads é em relação às condições de corrida, onde dois ou mais threads podem acessar simultaneamente a mesma posição de memória. Nomeia-se seção crítica o trecho de código que acessa uma área de dados compartilhada.

Dois ou mais threads acessando a mesma região crítica devem ser coordenados para acessar os dados em regime de exclusão mútua. O recurso oferecido em Pthreads, também como um tipo de dado opaco, é o `pthread_mutex_t`, o qual possui duas operações `pthread_mutex_lock` e `pthread_mutex_unlock`. Cada instância deste tipo de dado está, em um determinado instante de tempo, em um de dois estados, adquirido ou livre. O estado *adquirido* de um mutex significa que um thread detém o direito ao seu uso. A aquisição de um mutex se dá com a primitiva *lock* e a liberação pela primitiva *unlock*. A semântica associada é de que na instrução que segue a invocação a um *lock*, o mutex está alocado ao thread corrente e a seção crítica pode ser executada. Qualquer outra invocação à primitiva *lock* enquanto o mutex estiver adquirido resulta no bloqueio do thread que fez a tentativa de aquisição. No momento em que um mutex for liberado, um dentre os eventuais threads que estão bloqueados no aguardo do thread adquirirá o mutex e seguirá execução.

Na Figura 2.3 é apresentado um exemplo de utilização de mutex para controlar o acesso de seções críticas a dados compartilhados. O algoritmo possui dois threads, um que incrementa e outro que decrementa um contador compartilhado. Neste exemplo, tanto o dado compartilhado (a variável `contador`) como o mutex `m` estão definidos na área estática do programa. Note que o mutex `m` não tem nenhuma associação sintática com o dado que ele protege. A lógica está inserida na estratégia de controle proposta pelo programador.

```
1  int contador = 0;
2  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3
4  // Trecho em um thread
5  pthread_mutex_lock(m);
6  contador++;
7  pthread_mutex_unlock(m);
8
9  // Trecho em outro thread
10 pthread_mutex_lock(m);
11 contador--;
12 pthread_mutex_unlock(m);
```

Figura 2.3. Exemplo coordenação de seções críticas com uso de mutex em Pthreads.

Existem alternativas que aumentam a concorrência na coordenação de threads por mutex com as primitivas `pthread_trylock` e `pthread_timedlock` a exemplo do que ocorre com a primitiva `pthread_join`. Nestes casos, o valor 0 (zero) em retorno significa que o mutex foi adquirido. Outro recurso interessante no uso de mutex em Pthreads é possibilidade de indicar que o mutex poderá ser sujeito a múltiplas aquisições por um mesmo thread, por exemplo em um algoritmo recursivo. O programador deve explicitar, quando da inicialização do mutex, que esta possibilidade existe e tem, em contra-partida, a possibilidade de construir um código mais enxuto. No entanto o programador deve se

certificar que a operação *unlock* seja invocada o mesmo número de vezes que a operação *lock* o foi.

Outra forma de coordenar a execução de threads em Pthreads se dá pelo uso de variáveis de condição, oferecidas pelo tipo `pthread_cond_t`. Diferente do mutex, uma variável de condição não possui estado associado. Serve como um canal de comunicação de um sinal entre um thread emissor e um ou mais threads receptores. Quando um sinal é emitido, entende-se que uma determinada condição da aplicação foi satisfeita. Threads ao verificarem que uma condição não está satisfeita solicitam sua suspensão de forma explícita invocando o serviço *wait*. O thread permanecerá, então, bloqueado até que receba a sinalização de que a condição está satisfeita. Este sinal é enviado pelo thread que produziu o estado verdadeiro para a condição por meio de invocação à primitiva *signal* ou a primitiva *broadcast* – a diferença entre estas duas primitivas é que um *broadcast* comunica todos os threads que estejam suspensos na condição que a condição está atendida, enquanto *signal* sinaliza apenas um thread. A condição propriamente dita consiste em um dado compartilhado entre os threads, cujo valor indica se a condição está de fato ou não satisfeita. Observe-se que tanto a variável de condição como o dado que contém o valor a ser testado são dados compartilhados entre os threads, devendo estes ser acessados em regime de exclusão mútua (associado a um mutex, portanto).

O programador deve estar atento ao fato de que o teste de validade da condição deve ser feito sobre o dado da aplicação. Assim, não basta a um thread bloqueado em uma condição receber um sinal indicando que a condição, em um determinado instante, esteve satisfeita, um novo teste deve ser realizado para garantir que a condição ainda é verdadeira.

Um exemplo de uso de variável de condição é apresentado na Figura 2.4. Nesta figura existem dois fragmentos de código `A` e `B`. Threads produtores executam o fragmento `A`, produzindo itens e armazenando em um buffer. Threads consumidores executam o fragmento `B`. A condição é de que pelo menos um item esteja presente no buffer para que o consumidor possa prosseguir. Observe no código que um dos parâmetros de `pthread_cond_wait` é o mutex associado à condição. Durante a execução da primitiva *wait* o mutex é liberado para que outros threads tenham acesso ao dado compartilhado e à variável de condição e, assim que o sinal for recebido, o thread entra na disputa pela aquisição do mutex novamente. Como o thread irá, então, entrar na disputa pelo mutex `m`, não há garantias que ele irá consegui-lo de imediato. Assim há a necessidade de testar novamente a condição – devido a isto a operação *wait* está associada a um laço, e não a um teste simples.

Outro recurso de Pthreads é poder realizar múltiplas chamadas a uma mesma função garantindo que ela será executada, de fato, uma única vez, quando da primeira chamada. Este recurso é bastante útil para realizar um processo de inicialização, ou finalização, requerido para um conjunto de threads. A primitiva `pthread_once` recebe como parâmetro o endereço de uma estrutura de controle, do tipo opaco `pthread_once_t` e de uma função do tipo `void` sem parâmetros. Um exemplo típico pode ser a abertura de um arquivo manipulado pelos threads ou a inicialização de uma estrutura de dados compartilhada.

```
1  int contador = 0;
2  list<Item> buffer;
3  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
4  pthread_cond_t c;
5
6  // Fragmento A: Produtores
7  while( true ) {
8   item = ProduzItem();
9   pthread_mutex_lock(m);
10  buffer.push_back(item);
11  contador++;
12  if( contador == 1 ) pthread_mutex_signal(c);
13  pthread_mutex_unlock(m);
14 }
15
16 // Fragmento B: Consumidores
17 while( true ) {
18  pthread_mutex_lock(m);
19  while(contador == 0) pthread_mutex_wait(&c,&m);
20  item = buffer.pop_back();
21  contador--;
22  pthread_mutex_unlock(m);
23 }
```

Figura 2.4. Utilização de variável de condição em um algoritmo simples de compartilhamento de buffer.

2.3.2. Estratégia de execução

O padrão Pthreads não especifica qual modelo de threads, $N \times 1$, 1×1 ou $N \times M$, deve ser utilizado para sua implementação. Nos sistemas modernos, os dois últimos são os mais empregados, como a Native POSIX Thread Library em Linux (NPTL) e em Solaris (*light-weight process*). Cada thread criado é independente, executa de forma sequencial o código da função informada e segue as regras de visibilidade de escopo definida pela linguagem, possuindo pilha de dados própria, cujo tamanho pode ser configurado como um atributo no momento de sua criação. Uma vez lançado e antes de seu término, o thread pode estar em um destes estados: pronto para executar, em execução ou suspenso, refletindo uma política de escalonamento visando compartilhamento de recursos. O thread estará suspenso quando, em execução, solicitar uma operação bloqueante, que o faz aguardar que a operação seja satisfeita. Uma vez satisfeita a operação, o thread volta para o estado pronto. Também retornará ao estado pronto o thread que, durante a execução, for preemptado pelo escalonador. O término de um thread se dá por iniciativa própria, quando o código de sua função chega ao fim e executa a instrução `return`. Um thread também pode ser interrompido em resposta à invocação externa de `pthread_cancel`, mas neste caso, uma eventual sincronização por *join* resultará no retorno de um código de erro específico indicando que o thread foi cancelado.

Como as implementações de Pthreads são realizadas, em geral, em baixo nível, as estratégias de escalonamento estão apoiadas em mecanismos empregados pelo sistema operacional nativo quando implementadas. Assim, questões sobre otimização do uso de recursos (tempo de CPU, hierarquia de cache, consumo de energia, entre outros) não são diretamente tratadas por Pthreads. Ao criar um thread, o programador pode definir como atributo qual estratégia base de escalonamento que deverá ser utilizada para escaloná-lo, podendo ser `SCHED_OTHER`, `SCHED_FIFO` e `SCHED_RR`.

`SCHED_OTHER` é o escalonamento *default*, onde os threads são escalonados por uma estratégia round robin sem prioridade, distribuindo entre os threads fatias de tempo no acesso ao processador em função do conjunto de threads em execução. Outras duas estratégias, disponíveis apenas no modo super-usuário, permitem o escalonamento em modo tempo real, sendo a prioridade, outro atributo que pode ser informado na criação do thread, levada em consideração: `SCHED_FIFO` e `SCHED_RR`. Na estratégia FIFO, os threads são executados na ordem em que são recebidos, considerando a prioridade especificada. Neste caso, um thread em execução somente perderá o processador quando (*i*) terminar sua execução, (*ii*) for lançado um thread com prioridade mais alta, ou (*iii*) executar uma chamada à primitiva `pthread_yield`, que força sua liberação do processador. Na estratégia RR o escalonamento executa round robin entre threads de mesma prioridade.

Importante notar que os threads em Pthreads são *nomeados*, isto é, são identificados individualmente. Isto possibilita que em qualquer ponto do programa, um thread possa sincronizar (realizar *join*) com outro thread caso possua seu identificador. Isto tem por efeito a criação de um relacionamento irregular entre os threads, não sendo possível prever as dependências entre estes. Deve-se observar que, embora possam existir diversos threads executando o código de uma mesma função, um thread suporta apenas uma operação de sincronização (*join*) e um segundo *join* sobre o mesmo identificador de thread ocasiona um erro em tempo de execução.

2.4. C++11

A especificação de 2011 de C++ [Bre09, Com14] inclui, na interface desta linguagem, recursos para manipular threads. Sendo C++ uma linguagem orientada a objetos, os recursos para multithreading estão disponibilizados na forma de classes e objetos. A definição atual reflete justamente o amadurecimento de bibliotecas, como Boost e ACE, no desenvolvimento destes recursos sob a forma de bibliotecas de classes C++. Desta forma, a interface para programação multithread em C++ estende, consideravelmente, o poder de expressão em relação ao convencional uso de uma biblioteca Pthreads em um programa C++. Como exemplos deste potencial, está disponível o uso de mecanismo de tratamento de exceção e é possível parametrizar a entrada de funções sobre threads com tipos de dados utilizados pelo programa. Outro atrativo de C++11 é incorporar na própria linguagem recursos para operações atômicas, abstraindo do programador a necessidade de conhecer características da arquitetura para realizar tais operações.

Por ser um padrão altamente difundido, os compiladores C++ disponíveis no mercado, de forma livre ou comercial, já incorporam facilidades de programação multithread. Este atributo de C++11 o torna bastante competitivo como ferramenta de programação.

2.4.1. Interface de programação

A interface de programação de C++11 conta com uma classe `thread` no espaço de nomes padrão (`std`). Uma instância desta classe é criada passando, ao construtor como parâmetro, o código a ser executado pelo novo thread. O thread é criado durante o processo de construção do objeto `thread` e pode ser escalonado imediatamente. O método `join` do objeto `thread` permite sincronizar o thread corrente com o término do thread associado ao objeto sobre o qual a sincronização está sendo efetuada. Um objeto pode ser associado a um thread uma única vez e o método `join` pode ser invocado no máximo uma única vez.

O corpo de um thread, ou seja, a sequência de instruções a ser executada, pode ser apresentado de diferentes formas: como uma função convencional C, como um método ou como uma função lambda. Caso o código esteja contido em um método, deve ser indicado o endereço do método e o contexto onde ele deve executar, ou seja, qual o objeto. Caso não seja informado endereço de método, é assumido o operador parênteses (`operator()`). Utilizando uma função lambda significa que o programador não possui o interesse em fazer uma função ou definir uma nova classe para o corpo do thread, listando o código a ser executado na própria criação do objeto thread. Em todos estes casos, a função ou o método poderá receber um número não limitado de parâmetros, que serão recebidos por cópia, de qualquer tipo definido no programa, primitivos ou não, e deverá ser do tipo `void`.

Caso seja necessário retorno de resultados, alguns mecanismos estão disponíveis. É possível explicitar, utilizando as funções auxiliares `std::ref` e `std::cref` a passagem de parâmetros por referência ou referência constante. Note que o objeto `thread` recebe seus parâmetros sempre em cópia. Portanto, mesmo que a função/método que executará o corpo do thread receba parâmetros em referência, estes serão referências às cópias mantidas pelo objeto `thread`. Então, surge a necessidade de utilizar as referidas funções auxiliares.

A Figura 2.5 ilustra a criação de um objeto `thread` tendo como corpo do thread o `operador()` definido no tipo `Fibonacci`.

Um cuidado a ser tomado é que, neste caso, pelo menos dois threads terão acesso compartilhado à mesma área de memória simultaneamente. Outra forma de obter resultados da execução de threads é pelo uso do recurso de variáveis futuras, lançando os threads de forma assíncrona. Neste caso, a função a ser executada por um thread deve possuir um tipo de retorno. O recurso a ser utilizado é o provido pela função auxiliar `std::async`, a qual se responsabiliza por executar o serviço solicitado. O retorno desta função é um objeto da classe `std::future<T>`, onde `T` é do tipo retornado pela função/método a ser executado pelo thread. A sincronização é realizada com os métodos bloqueantes `wait` ou `get` do objeto `future<T>` retornado, onde `get` retorna o valor esperado.

Também é possível comunicar dados de forma coordenada entre threads combinando o uso de objetos `std::future<T>` com instâncias da classe `std::promise<T>`. O uso deste recurso permite, dentre outras possibilidades, que um thread retorne dados antes de seu término. Dado os custos de criação de threads, pode não ser interessante criar um thread específico para cada resultado a ser produzido, se

```
 1 struct Fibo {
 2  const int n, r;
 3  Fibo(const int n_) n(n_), r(0) {}
 4  void operator()() { r = FiboSeq(n); }
 5  int getRes() { return r; }
 6 };
 7
 8 int main() {
 9  Fibo f(20);
10  thread tfibo(std::ref(f));
11  ...
12  tfibo.join();
13  std::cout << "Fibonacci(20) = " << tfibo.getRes();
14  return 0;
15 }
```

Figura 2.5. Exemplo de criação e sincronização de threads em C++11.

tornando o thread produtor um contêiner de tarefas independentes. Este mecanismo deve ser visto como uma alternativas às soluções envolvendo seções críticas e/ou variáveis de condição, diminuindo a complexidade do código final.

Um exemplo de uso de objetos *future* e *promises* é apresentado na Figura 2.6, onde um thread executando a função `prod` é responsável pela produção de itens aguardados, como promessas, por um segundo thread. Observe que a cada objeto `promise` é associado um objeto `future`. O uso de estruturas de dados do tipo `list` da STL visa tornar o exemplo mais realista, indicando que o número de iterações entre os threads não necessita ser definido de forma estática.

Outros recursos de sincronização em C++11 permitem coordenar a execução de threads considerando o acesso a seções críticas e variáveis de condição de forma similar aqueles oferecidos por Pthreads (Seção 2.3.1). Estes recursos são oferecidos pelas classes `mutex` e `condition_variable`, respectivamente. Da mesma forma que em Pthreads, variações sobre o mecanismo básico são oferecidos, como mutex recursivo e sincronizações (*unlock* e *wait*) temporizados ou não bloqueantes. Uma funcionalidade adicional em C++11 é a possibilidade da operação *lock* sobre diversos objetos mutex por uma função auxiliar `std::lock`. Nesta opção podem ser passados dois ou mais mutexes por parâmetro, de forma que ou todos os mutexes são obtidos ou nenhum, sendo liberados eventuais mutexes obtidos.

Um aspecto a ser tratado é o caso onde um thread é terminado, normalmente ou por exceção, quando detém um mutex. Nesta situação, o estado do mutex é indefinido. C++11 apresenta como alternativa o uso de objetos das classes `lock_guard`. Ao ser criado um objeto desta classe, um objeto `std::mutex` deve ser passado como parâmetro. É garantido que no término do escopo envolvente ao objeto `lock_guard` o mutex será liberado. O mutex passado como parâmetro pode estar bloqueado ou não, sendo que um

```
1  void produz(std::promise<int>& prom) {
2   prom.set_value(313);
3  }
4  void consome(std::future<int>& fut) {
5    std::cout << "Valor: " << fut.get();
6  }
7  int main() {
8    std::promise<int> prom;
9    std::future<int> fut = prom.get_future();
10   std::thread thProd(produz, std::ref(prom));
11   std::thread thCons(consome, std::ref(fut));
12
13   thProd.join();
14   thCons.join();
15   return 0;
16 }
```

Figura 2.6. Coordenação entre threads utilizando objetos `future` e `promise`.

atributo adicional à criação informa como este mutex deve ser manipulado.

A Figura 2.7 ilustra o uso mecanismo de guarda com um mutex já obtido pelo thread corrente, situação informada pelo rótulo `adopt_lock`. O exemplo apresenta a troca consistente do estado interno de dois objetos. Neste exemplo ainda é ilustrado o uso da função auxiliar `lock` para obter os dois mutexes envolvidos na troca.

Caso exista a necessidade apenas de compartilhar dados entre threads, ainda é fornecida a classe genérica `atomic<T>`, onde `T` é um tipo primitivo de C++11 (como `int`, `bool` e `double`). Na interface de objetos desta classe são oferecidos métodos que executam de forma atômica sobre o estado interno do objeto, como os operadores `=`, `++` e `--` e os métodos `load`, `store` e `fetch_add` que fazem, a leitura, a escrita e um valor passado como argumento com o estado interno do objeto atômico.

A classe `atomic<T>` se apresenta como uma facilidade a mais, uma vez que estes recursos são presentes nas arquiteturas modernas na forma de instruções de máquina e disponíveis tanto na forma de bibliotecas como incorporadas em outras linguagens que suportam concorrência, como em C#. Na Figura 2.8 é apresentado um exemplo de um contador de instâncias ativas de um mesmo thread.

2.4.2. Estratégia de execução

O padrão C++11 não é acompanhado de um ambiente de execução. Sua implementação é realizada, usualmente, sobre os recursos de multiprogramação leve nativos dos diferentes sistemas operacionais, o que significa, em consequência, que é implementado sobre Pthreads, como é o caso dos compiladores GNU C++ e ICPC (da Intel). Os threads são criados e destruídos dinamicamente, não sendo possível, via interface C++11, alterar qualquer atributo do thread, como estratégia e prioridade de escalonamento ou tamanho da pilha de

```
1  struct Dados {
2   int vA, vB;
3   std::mutex m;
4   Dados(const Dados& d) { vA = d.vA; vB = d.vB; }
5   Dados& operator=(const Dados& d) {
6    vA = d.vA; vB = d.vB;
7    return *this;
8   }
9  };
10
11 void swap( Dados& a, Dados&b ) {
12  std::lock(a.m,b.m,a.m,a.m);
13  std::lock_guard<std::mutex> guardaA(a.m,std::adopt_lock);
14  std::lock_guard<std::mutex> guardaB(b.m,std::adopt_lock);
15
16  Dados temp(a);
17  a = b;
18  b = temp;
19 }
```

Figura 2.7. Exemplo de utilização do mecanismo de guarda.

dados. No entanto, a sua interface prevê o método `native_handle` na classe `thread` que permite acesso ao descritor de baixo nível do thread (`pthread_t` no caso de Pthreads) e com ele alterar alguns atributos de um thread já criado, como tipo e prioridade de escalonamento (mas não o tamanho da pilha).

Também a exemplo de Pthreads, a mesma sequência de instruções pode ser executada por diversos threads independentes, mas cada thread pode ser sincronizado uma única vez (*join*). Em contrapartida, é possível trocar a responsabilidade dos threads entre dois objetos `thread`. Isto quer dizer que o fluxo de execução propriamente dito pode suportar apenas uma operação de sincronização, mas o método `join` pode ser chamado sobre o mesmo objeto diversas vezes, conquanto que ele seja associado a diferentes manipuladores de threads.

2.5. OpenMP

OpenMP [CJP07, ARB14] define uma interface de programação multithread compatível com C/C++ e Fortran – o conteúdo neste texto assume o uso de OpenMP em programas C. Uma rede de esforços coletivos, gerenciada no contexto do OpenMP Architecture Review Board (ARB), tem a missão de manter esta interface. As primeiras versões de OpenMP datam de 1997 e 1998, para Fortran e C, respectivamente. A versão 3.0 inova ao permitir a criação de tarefas (*tasks*) de forma explícita e a versão mais atual, OpenMP 4.0, inclui manipulação atômica de dados e facilidades para acessar outros dispositivos (como GPUs).

```
1 struct Contador {
2  std::atomic<int> cont;
3  Contador( int v = 0 ) { cont.store(v); }
4  int operator++() { ++cont; return cont.load(); }
5  int operator--() { --cont; return cont.load();  }
6  int get() { cont.load(); }
7 };
```

Figura 2.8. Controle de um contador de forma atômica.

Importante ressaltar que o projeto de OpenMP sempre considerou seu uso para processamento paralelo, onde foi destacada a importância de questões relacionadas ao desempenho. Desta forma, OpenMP se tornou a primeira ferramenta de programação paralela com mecanismos de escalonamento em nível aplicativo operando em um modelo de execução $N \times M$ com ampla aceitação no mercado de desenvolvimento de software para arquiteturas multiprocessadas.

Na terminologia OpenMP, os M componentes do modelo são designados como *threads*, caracterizando processadores virtuais que dão suporte à execução paralela em um time de execução. As N atividades paralelas geradas pelo programa em execução recebem a denominação de tarefas, as quais podem ser geradas de forma explícita ou implícita.

2.5.1. Interface de programação

A interface é composta por três grupos de elementos: *i*) diretivas de pré-processamento, (*ii*) biblioteca de serviços, e (*iii*) variáveis de ambiente. Sendo que as duas primeiras compõem parte do programa e a última é gerida pelo sistema operacional. As diretivas de pré-compilação descrevem a concorrência e a sincronização entre as atividades concorrentes do programa e são tratadas durante o processo de geração do código. Os serviços de biblioteca permitem acessar e alterar as configurações do ambiente de execução dinamicamente. As variáveis de ambiente definem os comportamentos padrões assumidos na execução de todos programas OpenMP em uma determinada máquina.

Os nomes das variáveis de ambientes e dos serviços de biblioteca (em C/C++) possuem, respectivamente, o seguinte padrão de formação: `OMP_XXX` e `omp_XXX`, onde `XXX` identifica o uso da variável e a finalidade do serviço. Por exemplo, as variáveis `OMP_NUM_THREADS` e `OMP_STACKSIZE` definem os padrões para o número de threads no time de execução e o tamanho da pilha de cada thread que compõe o time de execução. Já os serviços `omp_get_num_threads` e `omp_set_num_threads` permitem acessar e alterar o número de threads no time de execução.

Embora as variáveis de ambientes e, em especial, os serviços de biblioteca sejam bastante úteis no desenvolvimento de programas OpenMP, o restante desta seção dedica-se a discutir as diretivas de pré-processamento e suas cláusulas, uma vez que estas são mais representativas do estilo de programação fornecido em OpenMP.

As diretivas OpenMP, em C/C++ possuem o seguinte formato: `#pragma omp <diretiva> [<lista de cláusulas>]`. Nesta sintaxe, `#pragma omp` consiste em uma sentinela que informa ao pré-processador que uma diretiva OpenMP está sendo apresentada, `diretiva` indica a ação desejada e `lista de cláusulas`, opcionais, são parametrizações para a diretiva. A especificação de uma diretiva pode seguir um comando simples (uma única instrução) ou composto (um bloco de comandos) ou uma sentinela para outra diretiva OpenMP. Observe no entanto que este comando, assim como o início do bloco, não pode estar na mesma linha da diretiva, mas sim na seguinte.

A primeira diretiva a ser considerada é a `parallel`. Esta diretiva constrói uma região paralela, tendo a função de inicializar o time de execução, criando seus threads e inicializando o processo de escalonamento. O thread onde esta diretiva é executada é denominado mestre. Este thread, junto com os demais threads criados, compõem o time de execução das tarefas geradas. Caso um comando, simples ou composto, se apresente na sequência, o número de tarefas criadas será igual ao número de threads no time, sendo que cada tarefa executará uma instância deste comando. Caso outra diretiva OpenMP siga à `parallel`, o resultado é a exploração de diferentes formas de concorrência e sincronização entre tarefas geradas pela aplicação. Este é o caso do uso das diretivas de compartilhamento de trabalho `for`, `sections` e `single`, as quais criam, de forma implícita, tarefas que serão executadas de forma concorrente sobre os threads do time.

A diretiva `for` permite expressar a concorrência de aplicações em termos de paralelismo de dados. Esta diretiva deve ser seguida de um comando *for* ordinário de C/C++. A paralelização consiste em dividir o espaço de dados percorrido no laço em tarefas independentes, atribuindo a cada tarefa a responsabilidade de executar um grupo (consecutivo) de iterações, chamados *chunks* ou partições, deste laço. Para que este particionamento seja possível, a variável de iteração, seu valor inicial e o limite superior devem ser valores inteiros e sem sinal, assim como, no final de cada iteração, a variável pode ser, apenas, incrementada ou decrementada e o término do laço é detectado por um dos seguintes operadores relacionais `>`, `>=`, `<` e `<=`. O bloco de comandos a ser executado em um `for` possui a mesma estrutura de um bloco de comandos em um programa C/C++ convencional, à exceção que a variável de iteração não pode ter seu valor alterado.

Embora a paralelização com a diretiva `for` possa parecer simples, deve-se atentar para que não sejam introduzidas dependências de dados entre instruções executadas em diferentes iterações. Estas dependências podem ser classificadas como: verdadeira (ou de fluxo), de saída ou de antidependência. Considere duas iteração i e j, com $j>i$, de um laço. A dependência verdadeira ocorre quando um dado é atualizado (escrito) na iteração i e utilizado (lido) na iteração j. A antidependência ocorre quando um dado é lido na iteração i e atualizado em j. A dependência de saída é aquela em que um dado é escrito e acessado em uma mesma iteração, podendo ocorrer intercalação da execução de iterações perdendo a integridade do dado.

O problema de dependência pode ser facilmente entendido quando se supõe que um laço determina uma ordem de execução das iterações, no entanto, como a execução é concorrente, nada pode-se afirmar em relação à ordem em que as iterações i e j irão ocorrer de fato. É possível, portanto, que o laço j ocorra em um instante de tempo anterior ao laço i, gerando uma dependência verdadeira quando um dado é utilizado em j sem ter

sido ainda produzido, ou fazendo com que o dado perpetuado em uma escrita seja aquele produzido em *i*, em uma dependência de saída, ou ainda que um dado acessado em *i* já tenha sofrido uma modificação esperada em *j*.

A Figura 2.9 ilustra os três tipos de dependência em um único laço. A figura apresenta também como se dá o acesso aos dados em cada iteração, supondo que os vetores envolvidos possuem tamanho `tam==5`. Neste laço, a dependência verdadeira é observada quando na iteração *i*, os vetores são acessados em leitura na posição *i-1*, sendo possível que a iteração *i-1* ainda não tenha sido executada. A antidependência ocorre quando em uma iteração *i* é acessado em leitura uma posição *i-1* dos vetores. A dependência de saída está na escrita e acesso à variável `aux` na mesma iteração.

```
1 for( x[0] = y[0] = 0; i = 1 ; i < 5 ; i++  {
2    aux=x[i];  x[i]=foo(y[i-1]);  y[i]=bar(x[i-1],aux);
3 }

   1) aux = x[1]   x[1] = y[0]   y[1] = x[0].aux
   2) aux = x[2]   x[2] = y[1]   y[2] = x[1].aux
   3) aux = x[3]   x[3] = y[2]   y[3] = x[2].aux
   4) aux = x[4]   x[4] = y[3]   y[4] = x[3].aux
```

Figura 2.9. Representação de diferentes tipos de dependências entre iterações na paralelização de um laço.

A diretiva `sections` permite descrever o paralelismo de tarefas de uma aplicação, sendo que o bloco de comando que segue esta diretiva deve conter diretivas `section` (no singular), delimitando o código de cada tarefa. A diretiva `single` também é tida como geradora de tarefa, no entanto, gera apenas uma tarefa, que será executada por um dos threads do time.

Um exemplo de código com as diretivas `parallel`, `sections` e `for` é apresentado na Figura 2.10. Neste código, a mensagem `Executando no thread ...` seguida do número do thread que executou a tarefa será impressa tantas vezes quantos forem os threads alocados ao time. Já as mensagens referentes a primeira e a segunda seção serão impressas uma única vez, cada. As mensagens referentes às iterações do laço serão impressas dez vezes, que é o comprimento deste laço.

O programador pode, de forma explícita criar tarefas pela diretiva `task`, seguida de um comando, simples ou composto, que contém o código a ser executado. Diferente das diretivas citadas anteriormente, o término do bloco de comandos que define uma tarefa não consiste em uma barreira implícita. Caso uma tarefa necessite sincronizar o término de (todas) tarefas que tenham sido criadas no seu contexto, uma chamada a diretiva `taskwait` deve ser incluída. O conceito de tarefa explícita foi introduzido em OpenMP para viabilizar a descrição de aplicações com natureza recursiva, havendo o conceito de precedência, explorado pelo núcleo de escalonamento.

O exemplo na Figura 2.11 apresenta o cálculo recursivo de Fibonacci com tarefas explícitas. Neste código é especificada uma região paralela na qual uma única tarefa é

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main() {
5  int i;
6
7  #pragma omp parallel
8  {
9   printf("Executando no thread %d\n",omp_get_thread_num());
10  #pragma omp sections
11  {
12   #pragma omp section
13    printf("Executando no thread %d\n",omp_get_thread_num());
14   #pragma omp section
15    printf("Executando no thread %d\n",omp_get_thread_num());
16  }
17  #pragma omp for
18   for( i = 0 ; i < 10 ; i++ )
19     printf("Executando no thread %d\n",omp_get_thread_num());
20 }
21 return 0;
22 }
```

Figura 2.10. Exemplo de criação de tarefas implícitas em OpenMP.

criada de forma implícita. Desta primeira tarefa tarefas são criadas de forma explícita, em uma estratégia recursiva, para obter o resultado desejado.

Em relação ao compartilhamento de dados, importante observar que as regras de escopo da linguagem hospedeira, C/C++, continuam válidos. Ou seja, todos os identificadores visíveis no comando que compõe o corpo de uma tarefa continuarão visíveis em todas as instâncias desta tarefa. Para controlar o acesso a estes identificadores, certas regras são adotadas. A regra geral diz que todos os identificadores visíveis permanecerão visíveis nas tarefas e irão fazer referência à mesma instância do dado, de forma compartilhada. Este comportamento pode ser alterado com o uso das cláusulas `private`, `firstprivate`, `lastprivate` ou `shared`, sempre seguido de uma lista de identificadores. Enquanto o uso da cláusula `shared` apenas permite explicitar o mesmo comportamento padrão descrito, as demais forçam a criação de instâncias próprias a cada tarefa dos identificadores listados. No caso de identificadores `firstprivate`, cada réplica é inicializada com o último valor anotado antes da geração das tarefas, um valor indeterminado é assumido nos demais casos. O dado original somente é atualizado no final da execução das tarefas no caso de dados `lastprivate`, sendo que o valor assumido é aquele observado na execução da última tarefa que possui uma réplica do identificador. Nada impede que um identificador seja declarado como `first` e `lastprivate` ao mesmo tempo. No caso de tarefas explícitas, as variáveis são compartilhadas apenas

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int Fibo(int n) {
5    if (n<2) return n;
6    else {
7      int x, y;
8      #pragma omp task shared(x)
9       x=Fibo(n-1);
10     #pragma omp task shared(y)
11      y=Fibo(n-2);
12     #pragma omp taskwait
13     return x+y;
14   }
15 }
16
17 int main() {
18   int r;
19   #pragma omp parallel
20   {
21    #pragma omp single
22      r = Fibo(20);
23   }
24   printf ("fib(20) = %d\n", r);
25   return 0;
26 }
```

Figura 2.11. Fibonacci recursivo em OpenMP.

quando forem determinadas como `shared` na região paralela envolvente. Nas demais situações, é assumido `firstprivate`. Regras mais específicas, considerando níveis de aninhamento e heranças de identificadores entre estes níveis, definem outros comportamentos. O mais seguro é explicitar sempre as cláusulas de compartilhamento conforme o desejado.

Outra cláusula de compartilhamento, o `reduction`, permite compartilhar dados em uma estratégia de escritas concorrentes. Dados deste tipo são associados a uma operação associativa, como soma, multiplicação ou OU-lógico, e cada tarefa possui uma réplica da variável inicializada com o valor identidade da operação associada, 0 (zero), 1 (um) ou 0b (zero binário) no caso das operações citadas. Cada tarefa manipula sua cópia do dado como o desejar, sendo que o resultado parcial computado sobre cada réplica em cada tarefa é submetido à operação associada sobre o dado original.

Como apoio complementar, é possível definir uma lista de identificadores cujos threads do time terão instâncias próprias. Estas instâncias possuem valor inicial indefinido. A diretiva a ser usada é `threadprivate`.

A interface de programação de OpenMP definida para a linguagem C é operacional em C++. No entanto, ela não foi definida para esta linguagem. Assim, o uso de instâncias de classes (objetos) não possui total suporte. Não é possível, por exemplo, aplicar a operação de redução sobre um objeto, mas é possível indicá-los com `private`, `first` e `lastprivate` e `shared`. No entanto, cabe verificar qual o comportamento dado à construção das cópias e suas respectivas destruções no compilador utilizado. A especificação atual de OpenMP também comporta o uso de iteradores em contêiners STL no `parallel for`.

Embora os recursos de compartilhamento de dados já apresentados permitam diversos mecanismos de comunicação entre as tarefas, quatro diretivas permitem especificar outros mecanismos de sincronização: `barrier`, `master`, `atomic` e `critical`. Enquanto as duas primeiras regulam o fluxo de execução, criando barreiras de forma explícita, as duas últimas permitem acessar dados em regime de exclusão mútua. `barrier` constrói uma barreira que somente será superada quando todas as tarefas atingirem este ponto. `master` define um comportamento similar ao `single`, considerando que o comando associado será executado, necessariamente, sobre o thread mestre.

A diretiva `atomic` é seguida, necessariamente, de um comando simples que deve ter a forma de uma atribuição por um operador binário (como `+=` ou `-=`) ou incremento ou decremento de uma variável. Apenas a atribuição é realizada de forma atômica, se eventualmente alguma expressão for avaliada, não há nenhum mecanismo de sincronização associado.

A diretiva `critical` oferece a possibilidade de obter execuções de trechos de código em regime de exclusão mútua. Uma chamada a esta diretiva pode ter um rótulo associado, assim a sincronização dos respectivos trechos de código se dará de acordo com o rótulo ao qual se referem.

2.5.2. Estratégia de execução

O modelo de execução de OpenMP é do tipo $N \times M$, como citado anteriormente. O número M de threads no time de execução é, por padrão, igual ao número de CPUs disponíveis na arquitetura, embora pode ser alterado em tempo de execução. Uma forma de alterar é explicitar o número desejado de threads como parâmetro à invocação à primitiva de biblioteca `omp_set_num_threads`. Outra forma é realizando o controle do paralelismo aninhado por meio das primitivas `omp_set_nested` e `omp_set_dynamic`. Caso estas primitivas sejam invocadas com parâmetros habilitando o paralelismo aninhado (qualquer valor diferente de zero passado como parâmetro), a cada nova região paralela será criado um novo time de execução que não irá se relacionar com os demais times. Ou seja, não há compartilhamento de trabalho entre os times de execução.

O modelo de programa proposto por OpenMP explora uma estratégia de execução do tipo *fork*/*join* aninhado. Ou seja, pontos de sincronização implícitos envolvem cada uma das diretivas de paralelismo (exceto a diretiva `task`, cujo caso é discutido adiante). Assim sendo, todas as tarefas são lançadas ao mesmo tempo e o final do bloco que as delimita consiste em uma barreira na qual é aguardado o término de todas as tarefas lançadas. Uma possibilidade é introduzir uma cláusula `nowait`, que relaxa a condição de bloqueio. Assim, quando a cláusula `nowait` é utilizada, na medida em que os threads

detectam que não há mais tarefas a serem executadas naquela região paralela, eles podem avançar sobre a próxima. No que diz respeito ao escalonamento das tarefas geradas em um `for`, observa-se que é possível selecionar, por meio de uma cláusula `schedule` a estratégia desejada dentro das disponíveis, estática, dinâmica ou guiada, e o número de iterações que irão compor cada partição.

Por padrão, o escalonamento de tarefas geradas em um `for` é realizado de forma estática, assumindo tamanho de partição com uma (1) iteração. Assim, são geradas tantas tarefas quanto forem o número de iterações indicado, sendo que estas tarefas são distribuídas de forma circular entre os threads do time. Um número maior de partições permite aumentar a granularidade de execução, compondo tarefas maiores. O escalonamento estático é apropriado quando o custo estimado do cálculo em cada iteração é constante. No caso dos custos serem irregulares, convém utilizar a estratégia dinâmica, onde, inicialmente, cada thread recebe uma tarefa a ser executada e as demais permanecem em uma lista compartilhada aguardando para serem consumidas. A ideia é que novas tarefas sejam consumidas uma a uma pelos threads, aumentando a probabilidade de distribuir igualmente a carga de trabalho gerada. Novamente, o número de iterações informado para a partição regula a granularidade de execução. A estratégia guiada é semelhante à dinâmica, com uma diferença: as partições possuem número de iterações diferentes entre si, sendo que na lista de trabalho, a primeira tarefa é a que contém o maior grupo de iterações, sendo reduzido até que a última possua o número de iterações. Este mecanismo de escalonamento torna-se interessante quando a diretiva `for` segue a outra diretiva de paralelização onde foi informada a cláusula `nowait`. A consequência, neste caso, é que fatalmente um thread irá iniciar a execução destas tarefas antes dos demais, podendo então adiantar o serviço antecipando computações.

O esquema de escalonamento de tarefas geradas explicitamente (`task`) possui diferenças significativas ao escalonamento das tarefas geradas pelas demais diretivas, uma vez que considera o relacionamento de dependência entre tarefas e sua natureza recursiva. As ações de escalonamento podem ser disparadas no contexto de um thread quando este criar ou terminar a execução de uma tarefa ou atingir um ponto de sincronização (implícito ou explícito). As ações de escalonamento podem ser, sem nenhum nível de prioridade especificado: iniciar a execução de uma tarefa ligada ou não ao time corrente que esteja pronta, retomar a execução de uma tarefa que estava bloqueada e que esteja novamente a continuar a executar. Embora não seja definida prioridade entre as ações de escalonamento, o estado da lista de tarefas bloqueadas sobre um determinado thread determina regras a serem seguidas. Se esta lista estiver vazia, uma tarefa amarrada poderá ser lançada. Se esta lista contiver pelo menos uma tarefa, uma nova tarefa amarrada somente poderá ser lançada se for descendente de todas as tarefas na lista de bloqueadas.

2.6. Cilk Plus

O início da história da linguagem Cilk Plus ([Cor14b]) data de 1994 quando do desenvolvimento de Cilk no Massachusetts Institute of Technology (MIT) e dando origem a Cilk Arts, Inc., onde foi desenvolvido Cilk+, compatível com os compiladores GNU GCC e Microsoft C. Esta empresa foi adquirida pela Intel em 2009, sendo lançada a linguagem Cilk Plus, compatível com C++. Quando comparado a outras ferramentas de programação multithread, a interface de programação disponibilizada por Cilk Plus é bastante enxuta.

De fato o programador conta com um número limitado de recursos mas, em compensação, dispõe de um núcleo de execução eficiente e especializado na execução de programas cujo paralelismo possui natureza aninhada e recursiva.

Outra característica desta linguagem é seu caracter determinista, sendo que, se retiradas as palavras reservadas de Cilk Plus, será obtido o programa sequencial equivalente. Este determinismo reflete a ausência de construtores de manipulação de seção crítica e o modelo de programação baseado em fluxo de dados.

2.6.1. Interface de programação

O programador conta com três palavras chave para explicitar a concorrência da aplicação: `cilk_spawn`, `cilk_sync` e `cilk_for`. A operação *spawn* recebe, como parâmetro, uma função e os parâmetros de entrada desta função. A semântica associada é que a função corrente e a invocada por meio da primitiva *spawn* poderão executar de forma concorrente na forma de dois threads independentes. Não há limite no número de threads que podem ser lançados por uma função, no entanto, os threads não são identificados individualmente. Caso a função necessite sincronizar sua execução com o término de todos os threads que tenha criado, tipicamente para recuperar dados de saída, uma invocação à primitiva `cilk_sync` deve ser realizada. Esta primitiva não possui parâmetros e é garantido que a função continuará a executar somente quando todos os threads por ela criados tenham terminado. Esta característica determina o comportamento de criação e sincronização aninhados de threads.

A Figura 2.12 apresenta uma estratégia recursiva para o cálculo de Fibonacci em Cilk Plus. Enquanto não atinge a raiz da solução, são criados threads para computar determinadas posições desta série, sendo que, para cada posição, é necessário obter o valor das duas posições imediatamente anteriores. Na solução apresentada, o parâmetro `n` é passado por cópia e o resultado das parcelas anteriores somente estará disponível após uma etapa de sincronização.

```
1
2  int Fibo( int n ) {
3   if( n < 2 ) return n;
4   else {
5    int x = cilk_spawn Fibo(n-1),
6        y = Fibo(n-2);
7    cilk_sync;
8    return x+y;
9   }
10 }
```

Figura 2.12. Implementação em Cilk do algoritmo de Fibonacci.

Com as primitivas *spawn* e *sync*, uma vasta gama de aplicações podem ser descritas. No entanto, o recurso de paralelização de um laço pode ser realizado, de forma especializada, pela primitiva `cilk_for`. De forma equivalente à OpenMP, o laço é par-

ticionado em grupos de iterações e atribuído a threads independentes, e se colocam as mesmas questões de elaboração do corpo da iteração de forma a evitar dependências de entrada e saída e de antidependência. A característica do *for* paralelo de Cilk Plus é que o particionamento ocorre na forma de divisão e conquista, sendo espaço de iterações dividido ao meio, cada metade dividida novamente, ao meio e sucessivamente até obter grupos unitários.

Observe que, como não existem impedimentos na linguagem nativa, as regras de escopo e de passagem de parâmetros continuam válidas para os threads criados. No entanto, como não existe nenhuma semântica que pode ser atribuída aos identificadores visíveis (como em OpenMP) e também nenhum construtor para controle de acesso a seções críticas, é responsabilidade do programador atentar para sincronizar o acesso a posições de memória compartilhada com uso dos recursos de criação e sincronização de threads. Alternativamente, é oferecida a possibilidade de utilizar hiper-objetos, os quais permitem acessos em escrita concorrente a dados em uma semântica de redução.

A interface de programação de Cilk Plus oferece uma biblioteca de classes com diversas opções de redução de dados. Entre os recursos disponibilizados, é possível manter uma lista compartilhada, manipular strings e arquivos e realizar algumas operações de cálculo como a redução de soma. Estas classes são nomeadas como `reducer_XXX`, onde XXX caracteriza o tipo de serviço desejado, e estão disponíveis no espaço de nomes `cilk`. Algumas destas classes, como a classe `reducer_opadd` e `reducer_opmax` são classes genéricas (*templates*). Nestes casos, os tipos parametrizados devem conter as operações previstas pelo operador de redução. No caso dos exemplos, o tipo de dado deve fornecer o operador ”+” associativo (+=) e ”<” (menor que), respectivamente. Ainda é possível criar novas classes para objetos de redução conforme necessidades específicas.

Um exemplo de operação de redução é apresentado na Figura 2.13. Este exemplo consiste em uma reimplementação do cálculo de Fibonacci, onde o resultado computado em cada thread é acumulado em um objeto de redução compartilhado.

2.6.2. Estratégia de execução

Cilk Plus possui um núcleo de escalonamento implementado no modelo $N \times M$ onde a concorrência da aplicação, descrita em termos de threads usuário, é mapeada, de forma eficiente, sobre um conjunto de processadores virtuais, implementados por threads sistema. O número de processadores virtuais em Cilk Plus é, por padrão, igual ao número de CPUs disponíveis na arquitetura, podendo ser alterado por meio de uma variável de ambiente (`CILK_NWORKERS` ou pela invocação a primitiva `__cilkrts_set_param`. Se a opção for a de alterar o número de processadores virtuais invocando a primitiva citada, observar que esta alteração pode ser realizada apenas antes da primeira invocação à `cilk_spawn` ou à `cilk_for`.

O escalonamento de Cilk Plus introduz os conceito de *tarefa* e *continuação*. Uma tarefa consiste em um trecho de código executado em um thread entre duas operações que necessitem intervenção do escalonador aplicativo. Estes trechos correspondem aos segmentos de código (*i*) entre o início da execução de um thread e a chamada à criação de um novo thread, (*ii*) entre duas invocações a *spawn*, (*iii*) entre uma chamada *spawn* e uma chamada *sync*, e (*iv*) entre um *spawn* ou um *sync* e o término do thread. Desta forma,

```
1  void Fibo(int n, cilk::reducer_opadd<int>& r) {
2   if (n < 2) r += n;
3   else {
4    cilk_spawn Fibo(n-1,r);
5    Fibo(n-2,r);
6    cilk_sync;
7   }
8  }
9
10 int main() {
11  cilk::reducer_opadd<int> r;
12  Fibo(20,r);
13  std::cout << r.get_value() << std::endl;
14  return 0;
15 }
```

Figura 2.13. Implementação em Cilk do algoritmo de Fibonacci utilizando redução nos dados.

um thread consiste em um contêiner de tarefas. Uma continuação corresponde ao estado adquirido pelo thread que executou a criação de um thread na tarefa que sucede à criação de um novo thread.

Como exemplo de composição de tarefas, Na Figura 2.12, os threads criados com o parâmetro `n<2` geram uma única tarefa. Caso contrário, uma tarefa é definida do início da função `Fibo` até a criação do thread para `Fibo(n-1)`. A segunda tarefa no contexto deste thread é responsável pelo cálculo, em sequencial, de `Fibo(n-2)`, até a invocação da sincronização. A terceira e última, tarefa gerada neste thread corresponde ao trecho de código entre a sincronização e o término do thread.

A técnica de escalonamento empregada é um algoritmo de lista ([Gra69, BJK+95]) aplicado de forma dinâmica. Esta lista contém a descrição das tarefas/threads aptos a serem executados. Na implementação realizada, a lista é mantida de forma distribuída, cada processador virtual possui uma lista própria, de forma que threads criados em um determinado processador virtual são armazenados no segmento de lista gerada por este processador virtual. Também de sua lista local são consumidos threads para execução. Um processador virtual somente acessará o segmento de lista mantido em outro processador virtual (procedimento de roubo de trabalho) quando sua própria lista estiver vazia.

Em cada segmento de lista, novos threads são inseridos em um extremo (cabeça), de modo que no outro extremo (cauda) encontram-se os threads mais antigos. Estes threads mais antigos são aqueles mais próximos ao início do programa. No escalonamento, a prioridade é dada para a execução dos threads mais novos, de forma a privilegiar a localidade de referência dos threads e buscar encerrar um nível de recursão do programa. No caso de um roubo de trabalho, ao contrário, a estratégia procura maximizar a operação

de escalonamento realizada, migrando maior quantidade de trabalho possível. Assim, o roubo de trabalho prioriza retirar um thread da cauda de um processador virtual randomicamente escolhido.

A estratégia de escalonamento ainda é complementada por uma heurística que procura minimizar o sobrecusto de execução da maior sequência de tarefas, muitas vezes referenciado como *caminho crítico*, geradas pelo programa. Assim, levando em conta que o programa possui uma natureza aninhada e recursiva e que o escalonamento local privilegia execução em profundidade, quando da criação de um novo thread, o processador virtual passa a executar este novo thread, sendo que a continuação do thread corrente é armazenada na lista de threads.

O trecho de código apresentado na Figura 2.12 reflete a estratégia de escalonamento descrita. Como o cálculo da posição $n-1$ de Fibonacci gera maior carga de trabalho do que a posição $n-2$, o thread que calcula `Fibo(n-1)` está no caminho crítico. Caso o thread fosse criado para o cálculo da posição $n-2$, o programa continuaria correto, embora os recursos de escalonamento não estejam sendo utilizados de forma eficiente.

2.7. Threading Building Blocks

Threading Building Blocks [Cor14a], conhecida pela sigla TBB, consiste em uma biblioteca de classes e funções genéricas (templates) em C++ utilizadas para descrever, com alto grau de abstração, a concorrência de uma aplicação em termos de tarefas concorrentes. Nesta biblioteca, conceitos herdados da STL, como contêineres, iteradores e algoritmos, são amplamente utilizados e o programador dispõe ainda de recursos para criar tarefas utilizando funções lambda.

A interface de programação possui dois níveis de abstração. Em um deles, esqueletos representando padrões recorrentes de paralelismo em aplicações são disponibilizados na forma de funções genéricas. Em um segundo nível, é exposta a presença de um grafo de precedência entre as tarefas, devendo a concorrência e as relações de precedência entre tarefas ser descritas na forma de um grafo por meio da interface fornecida.

O escalonamento provido por esta ferramenta é similar ao oferecido por Cilk, sendo baseado em roubo de tarefas e implementado no modelo $N \times M$. Difere por permitir a presença, no grafo de precedência, de tarefas que não definam predecessores. Assim, ao contrário de Cilk, o grafo de dependências de TBB pode conter vários nós de entrada.

2.7.1. Interface de programação

Um programa TBB em execução gera tarefas. Estas tarefas são organizadas na forma de um grafo de precedência que permite manter a consistência dos parâmetros de entrada e das saídas de cada tarefa. No entanto, TBB conta ainda com recursos de barreiras por mutex para controle ao acesso a seções críticas de forma concorrente entre tarefas. A utilização de mutex obriga o programador tomar certos cuidados, principalmente quando utilizar aninhamento na geração de tarefas, para evitar que as precedências entre tarefas conflitem com a sincronização empregando mecanismos do tipo *lock* e *unlock*. As tarefas em si podem ser criadas de forma explícita ou implícita. Tarefas criadas de forma explícita fazem uso da interface provida pela classe `task` do espaço de nomes de TBB

e de uma série de serviços para construção de um grafo de precedência entre as tarefas. A criação implícita de tarefas são expressas na forma de algoritmos que implementam esqueletos de paralelismo na forma de funções auxiliares também definidas no espaço de nomes de TBB. O uso destes esqueletos abstrai os mecanismos de construção das tarefas e manutenção do grafo.

Independente de terem sido criadas da forma implícita ou explícita, cada tarefa pertence a um grupo de tarefas (`task_group_context`). Existe um grupo previamente criado, ao qual a primeira tarefa criada irá pertencer caso nada ao contrário seja informado. Grupos de tarefas podem ser criados dinamicamente e a criação de uma nova tarefa pode explicitar qual grupo de tarefas irá recebê-la. Caso nenhum grupo seja informado, por padrão a nova tarefa pertencerá ao mesmo grupo da tarefa criadora. Poder manipular grupos de tarefas permite uma classificação das tarefas em um programa, podendo reger as tarefas de forma coletiva, como atribuir prioridade e cancelar a execução atingindo todas as tarefas de um dado grupo.

Dentre os esqueletos de paralelismo disponibilizados por TBB, as funções auxiliares `parallel_for` e `parallel_reduce` exemplificam a criação de tarefas em laços. A ideia é percorrer uma lista de valores, de forma iterativa, e, a cada iteração, aplicar um determinado cálculo. No caso da redução em paralelo ainda é possível reduzir para um resultado único resultados parciais computados a cada iteração.

De forma semelhante ao *for* paralelo de OpenMP, o `parallel_for` particiona o espaço de iteração em *chunks*. Como entrada, esta função requer além do espaço de iteração, pelo menos um objeto já instanciado que contenha o código a ser executado para computação de cada partição gerada (tarefa). Outros dois parâmetros opcionais podem informar a política de particionamento a ser adotada e o grupo ao qual as tarefas irão pertencer.

O espaço de iteração deve ser informado por um objeto de iteração específico. TBB oferece, em sua biblioteca, uma abstração de objeto para iteração na classe `blocked_range` (versões para percorrer espaços de dados em duas e três dimensões também são disponibilizados) genérica para o tipo de dado a ser utilizado na iteração. A construção de um objeto deste tipo requer que o primeiro e o último elemento do espaço de iteração seja informado e, opcionalmente, qual o maior grão de trabalho a ser atribuído a cada tarefa.

O corpo da tarefa deve ser definido em um objeto de uma classe que redefina, de forma pública, o operador parênteses (`operator()`). Este método deve ser do tipo `void` e deve receber como parâmetro de entrada um objeto do mesmo tipo do utilizado na definição do espaço de iteração do comando `parallel_for` a ele aplicado. Isto porque, diferente do que ocorre em OpenMP, o corpo da tarefa é, ele próprio, uma iteração, recebendo como parâmetros as posições inicial e final de seu espaço de iteração. Observe que o objeto propriamente dito é copiado para as várias instâncias de tarefas geradas. Assim, acessos em escritas no estado interno estão sujeitas a conflitos. Para evitar tal situação, o `operator()` deve ainda ser constante.

As política de particionamento, em número de três, são discutidas na próxima subseção. Discutindo a interface de programação cabe salientar que o espaço de

iteração é particionado de forma recursiva e que nem sempre o tamanho do passo informado é o de fato aplicado. Ou seja, se o tamanho do grão informado na construção do objeto `blocked_range` for de fato relevante no algoritmo, a política `simple_partitioner` deve ser utilizada.

O esqueleto `parallel_reduce` possui uma estrutura similar ao *for* paralelo, diferendo por necessitar que a classe que especifica o corpo da tarefa implemente, além do `operator()`, dois métodos construtores e um método `join`. Os métodos construtores devem poder ser executados de forma concorrente com o corpo da tarefa e devem atentar para inicializar o dado a ser reduzido com o valor identidade da operação a ser realizada, por exemplo, com o valor 0 (zero) ou 1 (um) caso a operação seja uma adição ou uma multiplicação. São necessários dois métodos construtores, um para o construtor default (sem parâmetros) e outro sendo um construtor de cópia, recebendo uma referência a um objeto do mesmo tipo e um segundo parâmetro figurativo, cujo uso é necessário apenas no contexto do ambiente de execução para diferenciar o construtor chamado. Outra diferença é que o `operator()` não é, no caso da redução, um método constante, pois ele deve modificar o estado interno do objeto, pois o valor local será acumulado no estado interno e reduzido para o objeto original.

A função auxiliar que implementa o `parallel_reduce` é sobrecarregada de forma a receber o corpo do thread na forma de uma função. Neste caso, outro parâmetro deve especificar como a combinação de dois valores deve ser realizada e a própria função `parallel_reduce` retorna o valor da redução.

Um exemplo de uso dos esqueletos `parallel_for` e `parallel_reduce` é apresentado na Figura 2.14. Neste exemplo, um vetor de caracteres é inicializado, randomicamente, com 20 caracteres maiúsculos de forma paralela. Também de forma paralela, é contabilizado o número de caracteres `'A'` contidos neste vetor. A contabilização utiliza uma estratégia de redução.

A forma explícita de criação de tarefas requer que uma classe estenda a classe `task` e implemente o método `execute` sem parâmetros, definido como virtual puro na classe `task`, com o corpo da tarefa. O retorno deste método ser do tipo `task *`, cujo retorno influi no escalonamento realizado: caso o retorno seja o endereço de uma tarefa, esta tarefa será priorizada para execução; caso seja o valor `NULL`, caberá ao escalonamento selecionar uma tarefa para ser executada. O estado interno do objeto tarefa pode ser utilizado para comunicar dados entre a tarefa e sua tarefa criadora. O método `execute` não é, portanto, constante.

Embora exista uma única classe para descrever tarefas, no momento em que uma tarefa é criada, esta pode ser descendente da tarefa atual ou consistir em uma tarefa que descreva outra seção do grafo de precedência entre tarefas disjunto da tarefa criadora. No primeiro caso, tem-se uma tarefa *filha*, em que a relação de precedência entre a tarefa criadora e a criada está definida. No segundo caso, a tarefa criada representa a origem (raiz) de um novo segmento do grafo. Para especificar o tipo de tarefa criada, a interface de TBB redefine o operador `new` para receber como parâmetro um objeto da classe `allocate_child()` ou `allocate_root` para cada um dos casos possíveis.

Os métodos `spawn`, `spawn_and_wait_for_all` e `spawn_root_and_wait`

```
1  #include <stdlib.h>
2  #include "tbb/parallel_for.h"
3  #include "tbb/parallel_reduce.h"
4  #include "tbb/blocked_range.h"
5
6  struct Texto {
7    char *palavra;
8    const int tam;
9    Texto(const int t):tam(t) { palavra = new char(tam); }
10   void operator()(const tbb::blocked_range<int>& r) const {
11     for( int i=r.begin(); i!=r.end(); ++i )
12       palavra[i] = (int)'A'+random()%27;
13   }
14 };
15 struct Conta {
16   const Texto& frase;
17   const char caracter;
18   int quantos;
19   Conta(const Texto& t, const char& c) :
20           frase(t), caracter(c), quantos(0) {}
21   Conta(Conta& c, tbb::split) :
22           frase(c.frase), caracter(c.caracter), quantos(0) {}
23   void operator()(const tbb::blocked_range<int>& r) {
24     for(int i=r.begin(); i!=r.end(); ++i )
25       if( frase.palavra[i] == caracter) quantos++;
26   }
27   void join( const Conta& c) { quantos += c.quantos; }
28 };
29 int main() {
30   Texto sentenca(20);
31   Conta c(sentenca,'A');
32   tbb::parallel_for(tbb::blocked_range<int>(0, 20), sentenca);
33   tbb::parallel_reduce(tbb::blocked_range<int>(0, 20), c);
34   // O atributo quanto do objeto c possui o resultado
35   return 0;
36 }
```

Figura 2.14. Utilização de esqueletos TBB para laços concorrentes.

são serviços da classe `task` que permitem a criação de novas tarefas. Estes métodos recebem um objeto ou uma lista de objetos `task` para os quais serão criadas as tarefas correspondentes. Os dois primeiros criam tarefas ordinárias, ou seja, tarefas sucessoras (filhas), enquanto a última cria tarefas sem predecessor. Os métodos `spawn` e `spawn_root_and_wait` são estáticos, podendo ser chamados, portanto, de um trecho de código não pertencente a um objeto `task`. Outro detalhe é que os métodos `spawn_and_wait_for_all` e `spawn_root_and_wait` garantem que a tarefa criada será executada sobre o fluxo de execução corrente.

Além dos métodos de criação que incluem uma operação de sincronização (*wait*), o método `wait_for_all` também sincroniza a tarefa corrente com o término de todas as tarefas que ela tenha criado. Outra importante característica de TBB é que o trecho de código a ser executado após a invocação do serviço `spawn_root_and_wait` é considerado uma *continuação* da tarefa que executou esta invocação. Para que esta continuação seja executada, todos os predecessores devem ter sido sincronizados, e isto inclui a tarefa descrita no código anterior a esta invocação. O número de predecessores deve ser explicitado invocando o método `set_ref_count`, passando como parâmetro um inteiro que indica quantas tarefas filhas diretas da tarefa corrente foram criadas adicionado de uma unidade, para que o serviço de sincronização contemple a sincronização da tarefa corrente.

A manipulação explícita do grafo de precedência de TBB pela interface oferecida pela classe `task` é apresentada na Figura 2.15. Nesta implementação, na função `main`, é criada uma tarefa *raiz* a partir da qual é inicializado o cálculo de forma recursiva. Para cada tarefa gerada, enquanto o valor `n` não atingir a condição de término da recursão, duas tarefas filhas são criadas para calcular as posições `n-1` e `n-2`, uma pelo serviço `spawn` e outra pelo serviço `spawn_and_wait_for_all`. Como foram criadas duas tarefas filhas, somando a tarefa corrente, são três as tarefas que precedem a continuação da tarefa corrente, isto é, o código após a sincronização. A invocação a `set_ref_count` deve preceder à primeira criação de tarefa, sob pena de má operação das ações de escalonamento.

Outro serviço na interface de um objeto `task` é o método `set_affinity`. Este método também provê meios para que o programador informe ao escalonamento que a tarefa deve executar sobre um processador virtual específico. Tipicamente este recurso é explorado quando o programador verifica a possibilidade de explorar a localidade de dados da tarefa criada.

Outras duas formas de criação de tarefas são exemplificadas na Figura 2.16. Em uma das alternativas, a função auxiliar `parallel_invoke` pode receber de dois até dez argumentos representando corpos de tarefas, que serão executados de forma paralela, sendo que esta função somente retorna após o término das tarefas criadas. A segunda alternativa faz uso de um grupo de tarefas. Neste caso, um grupo é criado e as tarefas são lançadas no contexto deste grupo pela invocação do método `run` do objeto `task_group`. A sincronização das tarefas do grupo é realizada de forma explícita. O corpo das tarefas, nestes casos, pode ser um objeto que implemente o operador `operator()`, ou uma função lambda. A implementação apresentada ilustra estes dois casos, sendo que a implementação com função lambda permite alterar a variável `x`, uma vez que a recebe por

```
1  #include <iostream>
2  #include <tbb/task.h>
3
4  struct Fibo : public tbb::task {
5    const long n;
6    long* const soma;
7    Fibo(long n_, long* s_) : n(n_), soma(s_) {}
8    tbb::task* execute() {
9      if(n<2) *soma = n;
10     else {
11       long x, y;
12       Fibo& a = *new(tbb::task::allocate_child()) Fibo(n-1,&x);
13       Fibo& b = *new(tbb::task::allocate_child()) Fibo(n-2,&y);
14       set_ref_count(3);
15       spawn(b);
16       spawn_and_wait_for_all(a);
17       *soma = x+y;
18     }
19     return NULL;
20   }
21 };
22
23 int main() {
24   long soma = 0;
25   Fibo& a = *new(tbb::task::allocate_root()) Fibo(20,&soma);
26   tbb::task::spawn_root_and_wait(a);
27   // soma possui o resultado
28   return 0;
29 }
```

Figura 2.15. Exemplo da criação de tarefas explícitas em TBB.

referência.

```
1  #include <tbb/task_group.h>
2  #include <tbb/parallel_invoke.h>
3
4  struct CorpoUm { void operator()()const{ /* ... */ } };
5
6  int main() {
7   tbb::task_group g;
8   CorpoUm c;
9   int x = 2;
10
11  g.run(c); g.run([&]{ x = x*x;}); g.wait();
12  tbb::parallel_invoke(c,[&]{x = x*x;});
13  // x possui o valor 16
14  return 0;
15 }
```

Figura 2.16. Formas alternativas para criação de tarefas em TBB.

Outro característica da interface de TBB é a existência de recursos de programação compatíveis com a especificação C++11, como a classe `thread` e alguns recursos para manipulação de seções críticas, como primitivas de sincronização por mutex e variáveis de condição. Deve-se observar, no entanto, que estes recursos são oferecidos em TBB no espaço de nomes `std` e não no espaço de nomes `tbb`, como era de se esperar. A razão de utilizar o espaço `std` é permitir que no processo de ligação do programa as implementações de TBB sejam diferenciadas de outras eventuais implementações especificadas.

2.7.2. Estratégia de execução

O modelo de execução de TBB está baseado em uma estratégia $N \times M$, onde o número M de processadores virtuais padrão é o número de CPUs disponíveis na arquitetura. Este número pode ser alterado pela criação de um objeto da classe `task_scheduler_init`, antes da criação de qualquer tarefa, passando como parâmetro de criação um inteiro representando o número desejado de processadores virtuais.

O esquema básico de escalonamento em TBB segue a mesma estratégia básica implementada em Cilk Plus, onde uma lista de tarefas global é mantida de forma distribuída entre processadores virtuais. Cada processador virtual insere tarefas nesta lista e dela também retira trabalho a ser realizado. Localmente os trabalhos criados há menos tempo são priorizados. No caso de não haver mais trabalho na lista local, o processador busca trabalho em outro local. Este outro local de busca de trabalho é que difere a estratégia de TBB daquela implementada em Cilk Plus.

Foi apresentado anteriormente que a criação explícita de tarefas se dá invocando o método estático `spawn` da classe `task`. Quando a criação é realizada desta forma,

a nova tarefa é de fato criada nesta lista local. No entanto, outro método estático desta classe, o método `enqueue`, o qual recebe um objeto da classe `task` insere a tarefa em uma lista única compartilhada por todos os processadores virtuais. A diferença dos escalonamentos de TBB e Cilk Plus é exatamente a presença desta lista, a qual é consultada prioritariamente em relação ao roubo de trabalho no final da lista local a outro processador virtual.

Assim, quando do término da execução de uma tarefa, a próxima tarefa a ser executada será selecionada em função de uma sequência de decisões. A primeira que resultar verdadeira, indicará a tarefa a ser executada.

1. Ao término da execução de uma tarefa, o método `execute` pode retornar o endereço de uma outra tarefa, normalmente uma tarefa filha criada em seu contexto. Se este for o caso, esta tarefa retornada será executada.
2. Ao terminar uma tarefa, uma sucessora desta tarefa será executada caso a tarefa terminada tenha sido o último predecessor concluído.
3. Caso exista pelo menos uma tarefa na lista local, a última tarefa inserida.
4. Caso exista, uma tarefa que defina afinidade para o processador virtual.
5. Caso exista, uma tarefa próxima do início da lista de tarefas compartilhada.
6. Caso exista, a tarefa mais antiga na lista local de outro processador virtual randomicamente escolhido, podendo diversos processadores virtuais serem visitados antes que o roubo possa ser efetivado.

Observe que as mesmas características de Cilk Plus são encontradas nas condições 3 e 6, privilegiando, respectivamente, priorizar execução em profundidade das tarefas e valorizar o resultado da migração de tarefas em um algoritmo recursivo. Os demais passos consistem em alterações ao comportamento do escalonamento básico. Importante salientar que estas alterações encontram-se refletidas na interface de programação de TBB. O programador não precisa, evidentemente, utilizá-los. No entanto, abrir mão destes recursos pode limitar o desempenho que pode ser obtido com esta ferramenta.

2.8. Considerações finais

Nesta seção final é apresentada uma discussão sobre o uso das ferramentas tratadas neste capítulo. Após uma discussão sobre aspectos gerais, algumas características são pontuadas e confrontadas as abordagens adotadas entre as diferentes ferramentas. Ao final, são apresentados os códigos obtidos nas diferentes ferramentas de uma mesma aplicação.

2.8.1. Aspectos gerais

A popularização das arquiteturas multicore popularizou o acesso a arquiteturas paralelas. Hoje, devido aos preços competitivos, encontramos sistemas computacionais multiprocessados oferecendo suporte computacional nos mais variados setores de aplicação. Atualmente, ainda utilizamos a expressão *programação paralela* (ou *concorrente*) quando

queremos ressaltar uma característica do código desenvolvido em que os recursos de paralelismo da arquitetura foram efetivamente explorados. No entanto, a curto prazo, pode-se crer que passaremos a utilizar apenas o termo *programação* subentendendo o uso de estratégias de paralelismo e/ou concorrência, em função da onipresença de tais arquiteturas.

Uma prática a ser adotada é escolher a linguagem/ferramenta em função de suas características e adequação ao problema a ser implementado considerando critérios de desempenho. Nestes termos, uma ferramenta será mais adaptada para um determinado problema na medida em que disponibilizar construtores adequados para representar a concorrência expressa pela aplicação. A seleção da ferramenta em função de sua interface deve ser criteriosa, uma vez que a programação multithread, ela própria, já embute questões que facilmente podem incorrer em erros de implementação [HH08]. Da mesma forma, os critérios de desempenho, tipicamente refletidos no tempo de execução, devem satisfazer as necessidades do usuário final sobre o recurso de hardware que ele dispõe. No entanto, a realização da implementação propriamente dita não será bem sucedida caso os recursos oferecidos pela ferramenta escolhida não forem utilizados de forma adequada.

Neste texto foram apresentadas quatro ferramentas modernas para programação multithread, C++11, OpenMP, Cilk Plus e TBB em termos de suas interfaces de programação e estratégias de execução. Também dedicou-se certo espaço para apresentar Pthreads. Pthreads não é exatamente uma ferramenta multithread moderna, mas sua importância está relacionada a três aspectos principais. Primeiro, trata-se de um padrão largamente adotado e presente em praticamente todos sistemas computacionais atuais. Segundo, os recursos de programação que disponibiliza não são especializados em um determinado padrão de paralelismo, o que permite um vasto campo de aplicação. Terceiro, a maturidade desta ferramenta permite crer que existem implementações com níveis de desempenho altamente satisfatórios. Nesta lista ainda poderia ser incluído o fato de que boa parte das ferramentas de programação multithread de mais alto nível utilizam-se de Pthreads em suas implementações.

Os aspectos negativos em relação ao uso de Pthreads na escrita de programas multithread está no baixo nível de acoplamento [Boe05]) deste padrão com sua linguagem hospedeira (a linguagem C), na carência de abstrações de mais alto nível para programação paralela e na ausência de estratégias de escalonamento em nível aplicativo que ofereçam melhores índices de desempenho. Embora, de uma forma ou de outra, as ferramentas modernas busquem atender estes requisitos, não deve-se deixar de notar que o propósito de Pthreads não foi o de disponibilizar uma interface de programação para o processamento de alto desempenho, mas sim uma ferramenta padrão para compatibilizar aplicações de diferentes fornecedores.

Das ferramentas efetivamente modernas, OpenMP é a que está presente no mercado há mais tempo. Fruto do esforço combinado de representantes de várias indústrias, este padrão tem por foco uma interface de programação multithread para o processamento de alto desempenho. Comparando OpenMP com Pthreads, duas grandes diferenças de abordagens se destacam. A primeira diferença a ser notada é a granulosidade das atividades concorrentes geradas pelo programa. Enquanto em Pthreads a granulosidade é definida em termos de funções para as quais são instanciados threads, em OpenMP são blocos de instruções que definem tarefas concorrentes. A segunda diferença está na distinção da

descrição da concorrência da aplicação do paralelismo suportado pelo hardware. Isto é garantido por um esquema de execução $N \times M$ suportado por um mecanismo de escalonamento aplicativo.

A interface de programação de OpenMP é relativamente simples, com poucos construtores e atende boa parte dos problemas passiveis de paralelização. Um aspecto que este padrão ainda pode evoluir é na sua aproximação com C++, pois atualmente existe compatibilidade, mas OpenMP não contempla de forma integral todas as nuances da programação orientada a objetos nesta linguagem. Em relação ao uso, embora possa ser utilizado para descrever problemas com paralelismo de tarefas, a vocação de OpenMP continua representada pela extração do paralelismo de dados de forma automática. Com a possibilidade de criação explícita de tarefas nas novas versões deste padrão, no entanto, a capacidade de expressão do paralelismo de aplicações recursivas foi estendido. Em relação à interface, cabem cuidados com as regras adotadas para os acessos a dados em regiões paralelas aninhadas, pois a forma de acesso default (compartilhada ou privada), pode variar, sendo indicado explicitá-las sempre.

O escalonamento aplicativo de OpenMP é especializado no compartilhamento de carga gerado por regiões paralelas e em situações de paralelismo recursivo e aninhado de tarefas. Alguns recursos são disponibilizados na interface de programação para permitir interação com o escalonador, como relaxar a barreira final de uma região paralela (`nowait`) ou especificar qual tipo de escalonamento (estático, dinâmico ou guiado) deve ser utilizado no escalonamento das partições de um laço.

Com C++11 temos uma situação diferente daquelas observadas com Pthreads e OpenMP: os threads estão embutidos, como em Java [OW04], na linguagem de programação. Como consequência, todo arcabouço de recursos para programação orientada a objetos está disponível. Sem dúvida este aspecto é bastante relevante e positivo para o uso de C++11. No entanto, conta negativamente considerando a curva de aprendizagem da ferramenta. Neste caso, a capacitação no uso das facilidades para programação multithread em C++11 requer que o programador domine a linguagem propriamente dita, de forma a poder separar o que é C++ e o que é programação multithread em C++.

No que tange a recursos de escalonamento, estes não existem em nível aplicativo. Com uso de promessas e variáveis futuras é possível, no entanto, criar o conceito de tarefas no nível do programa e criar um modelo de execução do tipo $N \times M$. No entanto, na ausência de uma padronização para tal esquema de execução, a solução seria integrada ao algoritmo da aplicação.

Na proposta obtida com Cilk Plus existe uma integração que pode ser classificada de moderada com C++. Os tipos de dados definidos em termos de classes pelo programador podem ser utilizados em conjunto com a interface que este ambiente propõe e apenas conhecimentos de recursos elementares de C++, como classes genéricas e sobrecarga de operadores, se fazem necessários. Este aspecto, aliás, pode ser reflexo da concisão da sua própria interface. Cilk Plus permite a construção de programas que possuam uma estrutura de execução aninhada e recursiva, embora uma primitiva para paralelização de laços tenha sido adicionada. Este ambiente não pode, portanto, ser considerado como uma ferramenta de uso genérico, mas sim, altamente especializada.

Por conta desta especialização, o núcleo de escalonamento de Cilk Plus permite atingir excelentes índices de desempenho. Sendo realizado em nível aplicativo, a estratégia busca otimizar a execução da maior sequência de tarefas do programa. Para tanto uma heurística é estabelecida e seguida em tempo de execução, cabendo ao programador descrever a concorrência de seu programa de forma a refletir as premissas que o algoritmo de escalonamento adota.

Em TBB a interface de programação se caracteriza por explorar extensamente os recursos da linguagem C++, talvez mesmo mais que o necessário para explorar o uso de threads em C++11, e expor a estratégia de escalonamento no nível do programa. Por estas duas características, dificilmente TBB seria uma boa escolha para uma primeira ferramenta para programação multithread. A não ser se o programador estiver disposto a apenas utilizar os algoritmos de paralelismo implementado na forma de funções auxiliares. Por outro lado, um programador experiente, em C++ e em programação concorrente, encontrará nesta ferramenta uma vasta gama de recursos para explorar.

Em termos de escalonamento, TBB adota a mesma estratégia de Cilk Plus para escalonar um grafo de precedência de tarefas em termos da otimização da execução do caminho crítico. No entanto, estende este algoritmo para suportar estruturas de concorrência de programas que não sigam o modelo de *fork*/*join* aninhados. O custo desta extensão é que a representação do grafo de precedência é exposta ao programador. Cabe ao programador inserir, junto ao código do seu algoritmo, indicações de como proceder na manipulação de tarefas com vistas a influenciar nas decisões de escalonamento realizadas.

Como exposto, as opções para o programador são variadas, embora alguns aspectos emergentes à programação multithread possam ser observados pelo estudo realizado. Um primeiro aspecto é a aproximação das ferramentas para programação multithread com a linguagem hospedeira. Nos casos apresentados, existe uma clara tendência a absorver princípios da programação orientada a objetos. Se por um lado este aspecto é positivo por permitir um maior nível de abstração, permitindo, portanto, uma maior facilidade para o desenvolvimento de grandes sistemas, por outro lado força a uma curva de aprendizado mais lenta, pois habilidades avançadas de programação no paradigma orientado a objetos podem ser até mais custosas de desenvolver do que aquelas necessárias para a programação concorrente. Também pode ser observada uma tendência ao provimento de recursos para explorar paralelismo iterativo (laços paralelos) e criação de tarefas aninhadas com suporte de escalonamento especializado. Um terceiro aspecto é o modelo de execução $N \times M$ no qual uma estratégia de escalonamento executada em nível aplicativo é capaz de observar características do programa em execução para obter índices de desempenho de execução.

2.8.2. Detalhamento de características

Na Tabela 2.1 são contrastadas algumas características das ferramentas de programação discutidas neste texto. Nesta tabela, as primeiras cinco linhas dizem respeito a interface de programação. As demais, ao suporte de execução.

O primeiro ponto abordado é relacionado a complexidade da interface de programação oferecida. Neste quesito, OpenMP e Cilk Plus são as ferramentas que oferecem as interfaces de programação mais simples. No caso de OpenMP, as diretivas de

paralelização e sincronização consistem em anotações no código, que não alteram o algoritmo da aplicação. Em Cilk Plus, o reduzido número de primitivas é o principal aspecto da baixa complexidade de sua interface. A média complexidade de Pthreads é atribuída ao fato de que esta ferramenta é oferecida na forma de biblioteca, não sendo trivial a compatibilização dos tipos de dados entre esta biblioteca e o programa de aplicação. Também é considerada de complexidade média a interface de C++11, pois requer do programador conhecimentos avançados dos recursos de C++11 para utilização de todo o potencial da interface para programação multithread deste padrão. A alta complexidade da interface de TBB advém não apenas da necessidade de utilizar fortemente os recursos de programação orientada a objetos em C++, mas também da necessidade de interagir com o mecanismo de escalonamento para promover execução eficiente do programa.

Em relação a aderência a linguagem base, é considerado que Pthreads e OpenMP foram desenvolvidas para a linguagem C e as demais para C++. Neste quesito, OpenMP reflete melhor os conceitos de C na manipulação de tipos de dados do que Pthreads, uma vez existe a compatibilização dos tipos de dados passados como parâmetros entre as atividades concorrentes. Já em Cilk Plus, o uso de recursos de orientação a objetos não está totalmente disponível para a interface multithread, sendo explorado em pontos específicos, como nos tipos de dados de redução. Estes aspectos são também considerados para avaliação do suporte à orientação a objetos destas ferramentas.

No que diz respeito a abstrações de alto nível, a referência são a presença de algoritmos (esqueletos [Col04]) definindo comportamentos concorrentes recorrente em aplicações paralelas. OpenMP e Cilk Plus possuem laços paralelos e a possibilidade de realizar operações de redução de dados. Já TBB oferece uma biblioteca de algoritmos de esqueletos para auxiliar no desenvolvimento de programas.

Tabela 2.1. Comparativos entre as ferramentas.

Características	Ferramentas				
	Pthreads	C++11	OpenMP	Cilk Plus	TBB
Complexidade da interface	Média	Média	Baixa	Baixa	Alta
Aderência a linguagem base	Baixa	Alta	Alta	Média	Alta
Suporte à orientação a objetos	Não	Sim	Não	Não	Sim
Abstrações de alto nível	Não	Não	Algum	Algum	Sim
Descrição da concorrência	Explícita	Explícita	Explícita	Explícita	Explícita
Escalonamento em nível aplicativo	Não	Não	Sim	Sim	Sim
Decomposição do paralelismo	Explícita	Explícita	Implícita	Implícita	Implícita
Tamanho do executável	Pequeno	Grande	Pequeno	Médio	Grande

Uma responsabilidade delegada ao programador em todas as ferramentas é a de descrever a concorrência da aplicação. As ferramentas apresentadas não são capazes de inferir quais são as atividades que podem ser, potencialmente, executadas de forma paralela a partir de um programa. Por outro lado, OpenMP, Cilk Plus e TBB realizam escalonamento em nível aplicativo, que tem por responsabilidade realizar o mapeamento da concorrência da aplicação no paralelismo disponibilizado pelo hardware. A proposta desta estratégia é a de desonerar o programador do encargo de realizar o controle do

uso dos recursos de processamento disponíveis, implementando alguma heurística para exploração eficiente do hardware.

Por fim, em [R.14] cita o tamanho do código executável obtido a partir das diferentes ferramentas. Os menores códigos são os providos por Pthreads e OpenMP. Segundo este autor, isto deve-se ao fato de que o código, particularmente em OpenMP, é bastante próximo ao sequencial e que chamadas a serviços externos suprem demandas da execução (chamadas de sistema). No outro extremo, C++11 e TBB geram códigos extensos, em parte resultado do uso dos recursos de programação genérica (uso de *templates*), parte do fato de que poucas chamadas a serviços externos são realizadas, sendo implementado no próprio código os serviços necessários. Cilk Plus gera um código de tamanho médio nesta comparação, pois, a exemplo de OpenMP, o código do programa é muito próximo ao sequencial, mas inclui o uso de tipos genéricos.

2.8.3. Comparativo de programas

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <pthread.h>
 4 typedef struct { int begin, end, local; } args_t;
 5 void* Worker(void* arguments) {
 6   args_t* args = (args_t*)arguments;
 7   for(int i = args->begin; i < args->end; ++i)
 8     args->local += i;
 9   return(NULL);
10 }
11 int main() {
12   int sum = 0, num_threads = 5, step = 100 / num_threads;
13   pthread_t threads[num_threads];
14   args_t arguments[num_threads];
15   for(int i = 0, k = 0; i < 100; i+=step, k++) {
16     arguments[k].begin = i;
17     arguments[k].end = i + step;
18     arguments[k].local = 0;
19     pthread_create(&threads[k], NULL, Worker, &arguments[k]);
20   }
21   for(int i = 0; i < num_threads; ++i) {
22     pthread_join(threads[i], NULL);
23     sum += arguments[i].local;
24   }
25   printf("%d\n", sum);
26   return 0;
27 }
```

Figura 2.17. Somatório dos inteiros no intervalo [0;100) em Pthreads.

Os códigos apresentados na sequencia são versões daqueles encontrados em

```
 1 #include <iostream>
 2 #include <thread>
 3 #include <atomic>
 4 int main() {
 5   std::atomic_int sum = 0;
 6   int num_threads = 5, step = 100 / num_threads;
 7   std::thread threads[num_threads];
 8   for(int i = 0, k = 0; i < 100; i+=step)
 9     threads[k++] = std::thread(
10                          [&sum](int begin, int end)->void {
11                             int local_sum = 0;
12                             for(int i = begin; i < end; ++i)
13                               local_sum += i;
14                             sum += local_sum;
15                          },
16                          i, i + step
17                        );
18   for(int i = 0; i < num_threads; ++i)
19     threads[i].join();
20   std::cout << sum.load() << std::endl;
21   return 0;
22 }
```

Figura 2.18. Somatório dos inteiros no intervalo [0;100) em C++11.

[R.14]. Eles são reproduzidos aqui para permitir visualizar as diferentes soluções providas pelas diferentes ferramentas para uma mesma aplicação. No caso, a aplicação consiste em realizar a soma de todos os valores inteiros no intervalo entre 0 (zero) e 100.

A Figura 2.17 apresenta a primeira implementação, realizada em Pthreads. Como Pthreads não possui nenhuma abstração para descrição da concorrência da aplicação, o espaço de dados a ser percorrido (o intervalo de valores inteiros entre 0 e 100) é segmentado, considerando um número fixo de threads: cinco. Assim, o primeiro thread criado é responsável por acumular os resultados da soma dos valores entre 0 e 19, o segundo os valores entre 20 e 39 e assim por diante. Ao final, o valor final é obtido acumulando os resultados obtidos pela execução de cada thread.

A versão em C++11 deste problema encontra-se na Figura 2.18. De forma semelhante à versão em Pthreads, também foi necessário criar um número pré-definido de threads (cinco) para computar o somatório de diferentes segmentos do espaço de dados. No caso da implementação em C++11 optou-se por apresentar o corpo do thread na forma de uma função lamba. A diferença nesta solução em relação à implementação com Pthreads é que tipos atômicos estão disponíveis e uma variável deste tipo foi utilizada para realizar o acumulo de dados parciais. Note que na implementação realizada cada thread realiza o acesso ao dado atômico uma única vez, utilizando uma variável local para o somatório parcial. Desta forma reduz-se o acesso intensivo ao dado compartilhado.

```
1  #include <stdio.h>
2  #include <omp.h>
3  int main() {
4    int i, sum = 0;
5    #pragma omp parallel for reduction(+: sum)
6    for(i = 0; i < 100; i++)
7      sum += i;
8    printf("%d\n", sum);
9    return 0;
10 }
```

Figura 2.19. Somatório dos inteiros no intervalo [0;100) em OpenMP.

```
1  #include <iostream>
2  #include <cilk/cilk.h>
3  #include <cilk/reducer_opadd.h>
4  int main() {
5    cilk::reducer_opadd<int> sum;
6    _Cilk_for(int i = 0; i < 100; i++)
7      sum += i;
8    std::cout << sum.get_value() << std::endl;
9    return 0;
10 }
```

Figura 2.20. Somatório dos inteiros no intervalo [0;100) em CilkPlus.

Com o recurso de laços paralelos, se apresentam as soluções em OpenMP e Cilk Plus, respectivamente nas figuras 2.19 e 2.20. A grande diferença destas implementações para as precedentes é uso do recurso de laço paralelo para criação de tarefas. Os dois códigos apresentados possuem a mesma estrutura, diferindo quando Cilk Plus oferece o recurso de variável de redução na interface de um tipo de dado genérico.

A solução em TBB (Figura 2.21) também faz uso do recurso de laço paralelo. A solução apresentada utiliza funções lambda tanto para o corpo do thread (terceiro parâmetro), como para a operação de redução no quarto parâmetro. Na comparação desta implementação com as realizadas para OpenMP e Cilk Plus, observa-se que o corpo da tarefa de TBB contempla a execução de um laço, enquanto que nas outras duas ferramentas este laço é abstraído pelo próprio ambiente de execução.

A compilação dos códigos listados foi efetuada com o compilador Intel C++ Compiler. A Tabela 2.2 apresenta as linhas de comando para geração do executável para cada ferramenta. Para geração dos executáveis com Pthreads e C++11 faz-se necessário ligar o código do programa com a biblioteca `pthread`. A versão TBB necessita ligação com a

```
1  #include <iostream>
2  #include "tbb/tbb.h"
3  int main() {
4    int sum = tbb::parallel_reduce(
5                  tbb::blocked_range<int>(0,100),0,
6                  [](const tbb::blocked_range<int>& r,
7                  int local)->int {
8                    for(int i = r.begin(); i != r.end(); i++)
9                      local += i;
10                   return local;
11                 },
12                 [](int x, int y)-> int { return x+y; }
13             );
14   std::cout << sum << std::endl;
15   return 0;
16 }
```

Figura 2.21. Somatório dos inteiros no intervalo [0;100) em TBB.

Tabela 2.2. Linhas de comando para geração dos executáveis.

Ferramenta	Linha de comando para compilação
Pthreads	icpc SomatorioPthreads.c -lpthread
C++11	icpc -std=c++11 SomatorioC++11.cpp -lpthread
OpenMP	icpc -openmp SomatorioOpenMP.c
Cilk Plus	icpc SomatorioCilkPlus.cpp
TBB	icpc -std=c++11 SomatorioTBB.cpp -ltbb

biblioteca `tbb`. O compilador utilizado já possui os recursos para compilar Cilk Plus, para o qual é totalmente compatível. As versões do programa em C++11, por utilizar recursos específicos deste padrão, como a própria classe `thread`, necessita informar que o padrão a ser utilizado para a compilação é o especificado em C++11. Esta informação também é necessária para compilar o programa em TBB, uma vez que nesta versão são utilizadas funções lambda, as quais foram integradas a linguagem C++ pelo padrão C++11.

Referências

[AR06] Shameen Akhter and Jason Roberts, *Multi-core programming: increasing performance through software multi-threading*, Hillsboro, Or. Intel Press, 2006.

[ARB14] OpenMP ARB, *OpenMP application program interface*, `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`, July, 2013. Acesso em 18 de maio de 2014, Version 4.0.

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E.

Leiserson, Keith H. Randall, and Yuli Zhou, *Cilk: An efficient multithreaded runtime system*, Journal of Parallel and Distributed Computing, 1995, pp. 207–216.

[Boe05] Hans-J. Boehm, *Threads cannot be implemented as a library*, ACM/SIGPLAN Conf. on Programming Language Design and Implementation, ACM Press, 2005, pp. 261–268.

[Bre09] Clay P. Breshears, *The art of concurrency: A thread monkey's guide to writing parallel applications*, O'Reilly, 2009.

[CdS07] Gerson Geraldo H. Cavalheiro e Rafael Ramos dos Santos, *Multiprogramação leve em arquiteturas multi-core*, cap. 7, pp. 327–379, PUC-Rio, 2007.

[CJP07] Barbara Chapman, Gabriele Jost, and Ruud van der Pas, *Using OpenMP: Portable shared memory parallel programming (scientific and engineering computation)*, The MIT Press, 2007.

[Col04] Murray Cole, *Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming*, Parallel Computing **30** (2004), no. 3, 389 – 406.

[Com14] The C++ Standard Committee, *Working draft, standard for programming language C++*, `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf`, Acesso em 18 de maio de 2014.

[Cor14a] Intel Corporation, *Intel Threading Building Blocks*, `http://www.threadingbluildingblocks.org`, Acesso em 18 de maio de 2014, Homepage.

[Cor14b] _____, *Introducing Intel® Cilk™ Plus: Extensions to simplify task and data parallelism*, `http://www.cilkplus.org/cilk-plus-tutorial`, Acesso em 18 de maio de 2014, Cilk Plus Tutorial.

[Gra69] R. L. Graham, *Bounds on multiprocessing timing anomalies*, SIAM Journal on Applied Mathematics **17** (1969), no. 2, 416–429.

[HH08] Cameron Hughes and Tracey Hughes, *Professional multicore programming: Design and implementation for C++ developers*, Wrox Press Ltd., Birmingham, UK, 2008.

[NBF96] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell, *Pthreads programming: a POSIX standard for better multiprocessing*, O'Reilly, 1996.

[OW04] Scott Oaks and Henry Wong, *Java threads*, 3nd ed., O'Reilly, 2004.

[R.14] Florian R., *Choosing the right threading framework*, `https://software.intel.com/en-us/articles/choosing-the-right-threading-framework`, Acesso em 18 de maio de 2014, Intel Developer Zone.

[SMR10] Matthew J. Sottile, Timothy G. Mattson, and Craig E. Rasmussen, *Introduction to concurrency in programming languages*, Chapman & Hall/CRC Press, Boca Raton, 2010.

[Vaj11] András Vajda, *Programming many-core chips*, Springer, 2011.