



Ferramentas modernas para programação multithread

Gerson Geraldo H. Cavaleiro
André Rauber Du Bois

Universidade Federal de Pelotas
Centro de Desenvolvimento Tecnológico

XXXIII Jornadas de Atualização em Informática - JAI 2014
28 a 30 de julho de 2014



Contatos

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

{[@inf.ufpel.edu.br">gerson.cavalheiro,dubois](mailto:gerson.cavalheiro,dubois)}

1 Primeiros passos

- Conceitos
- Arquiteturas paralelas
- Sistemas operacionais
- Ferramentas de programação multithread

2 Exposição da concorrência

3 Ferramentas

- Pthreads
- C++11
- OpenMP
- Cilk Plus
- TBB

4 Keep walking

- ... Concorrência e paralelismo
- ... Arquitetura de computadores paralelos
 - ◆ Multiprocessador e multicore
 - ◆ Hierarquia de memória e cache
 - ◆ Instruções de baixo nível
- ... Sistemas operacionais
 - ◆ Processo e processo leve
 - ◆ Processo multithread
 - ◆ Escalonamento
- ... Ferramentas de programação multithread
 - ◆ Thread e tarefa
 - ◆ Modelos de implementação
 - ◆ Apresentação da interface

Concorrência

Expressa a ideia de competição por recursos e também independência temporal. Em um programa concorrente, além de haver competição pelos recursos físicos (processador, memória etc), também existe a competição pelos dados manipulados no programa.

- É uma propriedade da aplicação

Concorrência e paralelismo

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Concorrência

Expressa a ideia de competição por recursos e também independência temporal. Em um programa concorrente, além de haver competição pelos recursos físicos (processador, memória etc), também existe a competição pelos dados manipulados no programa.

- É uma propriedade da aplicação

Paralelismo

Expressa a capacidade de um hardware executar ao mesmo tempo duas ou mais atividades. As atividades de um programa em execução paralela possuem recursos suficientes de hardware, no entanto ainda são sujeitas a competir pelos dados

- Descreve uma característica do hardware

Concorrência e paralelismo

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Concorrência

Expressa a ideia de competição por recursos e também independência temporal. Em um programa concorrente, além de haver competição pelos recursos físicos (processador, memória etc), também existe a competição pelos dados manipulados no programa.

- É uma propriedade da aplicação

Paralelismo

Expressa a capacidade de um hardware executar ao mesmo tempo duas ou mais atividades. As atividades de um programa em execução paralela possuem recursos suficientes de hardware, no entanto ainda são sujeitas a competir pelos dados

- Descreve uma característica do hardware

Normalmente a concorrência de uma aplicação é muito maior que o suporte de hardware disponível

Arquiteturas paralelas

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Possuem mais de uma unidade de execução, que podem estar organizadas de diferentes formas, atenção para arquiteturas **MIMD** (*Multiple Instruction flow, Multiple Data flow*)

Arquiteturas paralelas

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Possuem mais de uma unidade de execução, que podem estar organizadas de diferentes formas, atenção para arquiteturas **MIMD** (*Multiple Instruction flow, Multiple Data flow*) que podem ainda ser classificadas quando a configuração da memória:

- Fracamente acopladas ou multicomputadores (como aglomerados de computadores)
- Fortemente acopladas ou multiprocessadores

Possuem mais de uma unidade de execução, que podem estar organizadas de diferentes formas, atenção para arquiteturas **MIMD** (*Multiple Instruction flow, Multiple Data flow*) que podem ainda ser classificadas quando a configuração da memória:

- Fracamente acopladas ou multicomputadores (como aglomerados de computadores)
- Fortemente acopladas ou multiprocessadores
As quais ainda podem ser classificadas segundo o espaço de endereçamento:
 - ◆ **UMA** (*Uniform Memory access*)
 - ◆ **NUMA** (*Non-Uniform Memory access*)

Considerar que:

- Uma **CPU** (*Central Processor Unit*) possui hierarquia de cache, decodificador de instruções e unidades para operações lógicas e aritméticas
- Um **core**, ou núcleo, consiste em uma CPU
- Um **processador** possui um ou vários *cores* e inclui uma hierarquia de cache, compartilhando ou não níveis entre estes
- Um processador **multicore** possui dois ou mais *cores*
- Um processador é dito **manycore** quando o número de *cores* é na ordem de centenas
- Um **multiprocessador** é visto como uma coleção de CPUs compartilhando um espaço de endereçamento comum

Arquiteturas paralelas fortemente acopladas

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Outras tecnologias:

- **CMP** (*Chip-level MultiProcessing*) é a tecnologia de duplicação de *cores*
- **SMT** (*Simultaneous MultiThreading*) é outra tecnologia, a qual duplica certas unidades de um core (como o hyperthreading da Intel).

Hierarquia de memória e cache

Primeiros passos

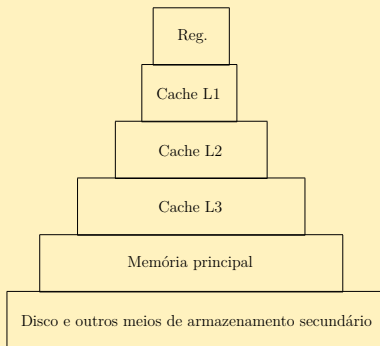
Exposição da concorrência

Ferramentas

Keep walking

Os níveis de cache são internos ao processador, havendo várias configurações:

- Um nível de cache pode especificar caches específicos para Instruções e para dados
- Em um multicore, um determinado nível de cache pode ser privado a um core ou compartilhado entre todos ou entre um grupo de cores



Multiprocessadores e multicores

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Para discussão dos recursos de programação multithread, multiprocessadores e multicores são equivalentes. Em ambos casos é possível que diferentes fluxos de execução de um mesmo programa (threads) compartilhem variáveis alocadas em um espaço de endereçamento compartilhado, assim não há reflexos na discussão do uso dos recursos de programação multithread. Da mesma forma, não é relevante considerar questões sobre a disponibilidade não da tecnologia SMT, a organização do espaço de endereçamento (UMA ou NUMA) ou questões sobre a hierarquia de cache. Estes aspectos, no entanto, devem ser considerados quando questões de desempenho do programa são relevantes. Neste caso, recursos de escalonamento em nível aplicativo devem ser incorporados.

Uma arquitetura com 8 cores

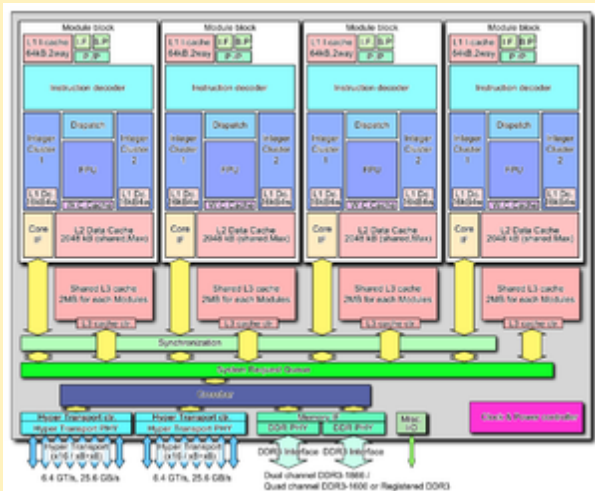
Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Processador
da família
AMD
Bulldozer



Uma arquitetura de um core

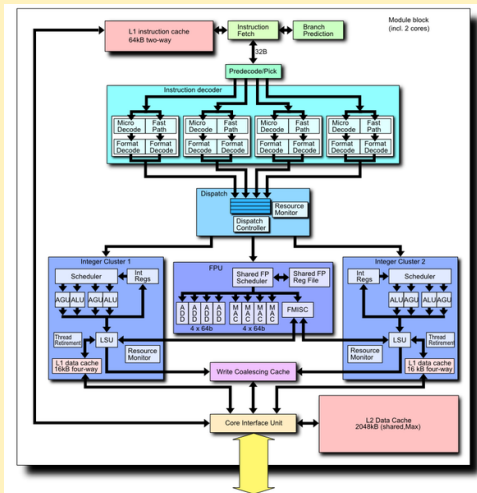
Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Core de processador
da família AMD
Bulldozer



Uma abstração de arquitetura paralela

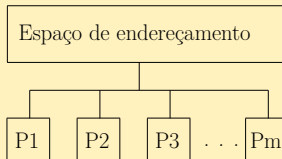
Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Nossa abstração para uma
arquitetura multiprocessada



Instruções de baixo nível

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

- Refletem a arquitetura do hardware
 - ◆ ADD op1, op2
 - ◆ CMP op1, op2
 - ◆ JZ lab
 - ◆ MOV dst, src
- Utilizam o espaço de endereçamento disponível
- O hardware garante coerência de cache nos diferentes cores, mas não previne dois ou mais acessos simultâneos (leituras/escritas concorrentes)

Instruções de baixo nível

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

- Refletem a arquitetura do hardware
 - ◆ ADD op1, op2
 - ◆ CMP op1, op2
 - ◆ JZ lab
 - ◆ MOV dst, src
- Utilizam o espaço de endereçamento disponível
- O hardware garante coerência de cache nos diferentes cores, mas não previne dois ou mais acessos simultâneos (leituras/escritas concorrentes)

Garantia de atomicidade

- LOCK é um prefixo para uma instrução assembly que garante acesso com exclusividade de uma linha de cache a um core

Instruções de baixo nível

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

- Refletem a arquitetura do hardware
 - ◆ ADD op1, op2
 - ◆ CMP op1, op2
 - ◆ JZ lab
 - ◆ MOV dst, src
- Utilizam o espaço de endereçamento disponível
- O hardware garante coerência de cache nos diferentes cores, mas não previne dois ou mais acessos simultâneos (leituras/escritas concorrentes)

Garantia de atomicidade

- LOCK é um prefixo para uma instrução assembly que garante acesso com exclusividade de uma linha de cache a um core

Este prefixo deve ser aplicado à

- INC dta
- XCHG op1, op2
- CMPXCHG dst, old, new

Instruções de baixo nível

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

- Refletem a arquitetura do hardware
 - ◆ ADD op1, op2
 - ◆ CMP op1, op2
 - ◆ JZ lab
 - ◆ MOV dst, src
- Utilizam o espaço de endereçamento disponível
- O hardware garante coerência de cache nos diferentes cores, mas não previne dois ou mais acessos simultâneos (leituras/escritas concorrentes)

Garantia de atomicidade

- LOCK é um prefixo para uma instrução assembly que garante acesso com exclusividade de uma linha de cache a um core

Este prefixo deve ser aplicado à

- INC dta
- XCHG op1, op2
- CMPXCHG dst, old, new
- Disponível nas APIs das linguagens de alto nível

Sistemas operacionais

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Definem uma série de serviços que suportam a execução de diversas unidades de execução em nível aplicativo.

Sua principal função é gerir a utilização de recursos e garantir a continuidade de todas execuções em curso.

Processo

Primeiros passos

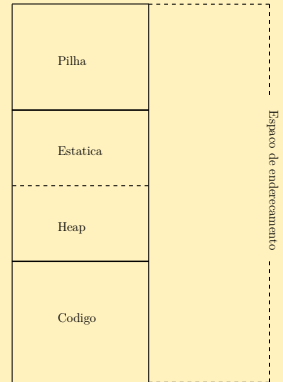
Exposição da concorrência

Ferramentas

Keep walking

Um processo é uma instância de um programa em execução e inclui

- Estado dos registradores (incluindo o PC e o SP)
- Uma pilha
- Espaço de endereçamento (dados/código)
- O Program Control Block (PCB) contém informações associadas a cada processo.
 - Estado (registradores)
 - Escalonamento
 - Memória (swap)
 - E/S (arquivos)



Tipicamente um processo possui um fluxo de execução, iniciado no momento do lançamento do programa

Processo multithread

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Um processo é dito **multithread** caso defina múltiplos fluxos de execução, chamados **threads**

- Um thread é chamado de processo leve, pois sua manipulação é mais *leve* que a de um processo, uma vez que boa parte das informações pertinentes ao processo, as descritas no PCB, são compartilhadas entre os threads, exceto pilha e estado dos registradores

Um programa multithread reflete uma arquitetura multiprocessada/multicore, uma vez que todos os threads criados no contexto de um processo compartilham o mesmo espaço de endereçamento (exceto a pilha)

Durante seu ciclo de vida, um processo ou thread pode se encontrar no estado

- Pronto
- Executando
- Waiting
- Terminado

A operação de escalonamento garante acesso aos fluxos de execução aos recursos de processamento disponíveis. Diferentes políticas podem ser utilizadas (prioridade, round-robin, FIFO etc) fazendo as transições nestes estados

O sistema operacional tem pouca informação sobre as aplicações em execução, suas decisões levam em consideração o uso dos recursos disponíveis.

Três modelos de implementação

- 1×1 : a concorrência descrita na aplicação é mapeada diretamente em um fluxo de execução manipulável pelo sistema operacional
- $n \times 1$: um programa multithread possui apenas um fluxo de execução manipulável pelo sistema operacional. As n atividades criadas durante a execução são escalonadas, em nível aplicativo, sobre este único recurso de execução
- $n \times m$: neste caso, há uma distinção entre os níveis de exploração da concorrência da aplicação e o paralelismo do hardware. As n atividades criadas pelo programa em execução são escalonadas, em nível aplicativo, sobre os m recursos de processamento disponíveis. Estes m recursos de processamento são escalonados pelo sistema operacional

Ferramentas de programação multithread

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

No nosso estudo, os modelos que permitem a exploração efetiva de um hardware paralelo serão considerados

1×1

$n \times m$

- Pthreads
- C++11

- OpenMP
- Cilk Plus
- TBB

No modelo 1×1 todo o escalonamento é realizado em nível do sistema operacional. No modelo $n \times m$, existe também escalonamento realizado em nível aplicativo.

● Em nível aplicativo, o escalonamento pode considerar propriedades do programa em execução e otimizar algum índice de desempenho do programa

Ferramentas de programação multithread

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

1×1

- Pthreads
- C++11

No programa são criados **threads** de forma explícita, os quais consistem em unidades de execução manipuláveis pelo sistema operacional.

Ferramentas de programação multithread

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

$$n \times m$$

- OpenMP
- Cilk Plus
- TBB

Introduzem os conceitos de **tarefa** e **processador virtual** (os termos utilizados variam). As tarefas descrevem a concorrência da aplicação. Os processadores virtuais consistem em threads nativas do sistema operacional. Em nível aplicativo as tarefas são escalonadas sobre os processadores virtuais

Apresentação de Interface

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Definidas em um padrão

- Pthreads
- C++11
- OpenMP

Bibliotecas

- Pthreads
- OpenMP
- TBB

Pré-processamento

- OpenMP

Extensão de linguagem

- OpenMP
- Cilk Plus

Linguagem

- C++11

1 Primeiros passos

- Conceitos
- Arquiteturas paralelas
- Sistemas operacionais
- Ferramentas de programação multithread

2 Exposição da concorrência

3 Ferramentas

- Pthreads
- C++11
- OpenMP
- Cilk Plus
- TBB

4 Keep walking

Modelos de decomposição paralela

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Primeiríssimas abordagens

Taxonomia de Flynn

Aplicada aos modelos de arquiteturas

Fluxos		Instruções	
		Simples	Múltiplos
Dados	Simples	SISD	MISD
	Múltiplos	SIMD	MIMD

Modelos de decomposição paralela

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Primeiríssimas abordagens

Taxonomia de Flynn

Aplicada aos modelos de arquiteturas

Fluxos		Instruções	
		Simples	Múltiplos
Dados	Simples	SISD	MISD
	Múltiplos	SIMD	MIMD

Aplicada aos modelos de arquiteturas

- No contexto de programas: **paralelismo de tarefas** e **paralelismo de dados**

Primeiríssimas abordagens

Paralelismo de tarefa

- MIMD/MPMD e MISD
- Os diferentes fluxos de execução possuem comportamento distinto, havendo necessidade de comunicação entre eles
- Exemplos
 - ◆ Computação dos parâmetros de entrada de uma função
 - ◆ Computação independente de partes de um problema
 - ◆ Agentes autônomos
- O número de tarefas identifica o paralelismo a ser explorado
 - ◆ Número fixo •problema não escalável

Primeiríssimas abordagens

Paralelismo de dados

- SIMD/SPMD
- os diferentes fluxos de execução possuem o mesmo comportamento e atuam sobre conjuntos de dados distintos, não havendo necessidade de comunicação
- Exemplos
 - ◆ Processamento de uma imagem
 - ◆ Fractais, Ray-tracing
 - ◆ Equações diferenciais sobre domínios regulares
- O espaço de dados a ser resolvido determina o tamanho do problema

Visão atual

Especialização dos modelos básicos

Paralelismo de tarefas

- Fluxos independentes
- Fluxo de dados
- Recursivo
- Apply-to-all
- Farm processing
- ...

Modelos de decomposição paralela

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Visão atual

Especialização dos modelos básicos

Paralelismo de tarefas

- Fluxos independentes
- Fluxo de dados
- Recursivo
- Apply-to-all
- Farm processing
- ...

Paralelismo de dados

- Iterativo
- All-to-all
- Scan (prefix-sum)
- Divisão e conquista
- Zip
- ...

Modelos de decomposição paralela

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Visão atual

Especialização dos modelos básicos

Paralelismo de tarefas

- Fluxos independentes
- Fluxo de dados
- Recursivo
- Apply-to-all
- Farm processing
- ...

Paralelismo de dados

- Iterativo
- All-to-all
- Scan (prefix-sum)
- Divisão e conquista
- Zip
- ...

A variedade é extensa, existindo várias propostas de ferramentas que oferecem **esqueletos** para descrição do paralelismo em um nível mais abstrato

Modelos de decomposição paralela

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Visão atual

Especialização dos modelos básicos

Paralelismo de tarefas

- **Fluxos independentes**

- **Fluxo de dados**

- **Recursivo**

- Apply-to-all

- Farm processing

- ...

Alguns destes padrões são relevantes pois são expressos naturalmente nas ferramentas estudadas

Paralelismo de dados

- **Iterativo**

- All-to-all

- Scan (prefix-sum)

- Divisão e conquista

- Zip

- ...

Fluxos independentes

- Como algoritmos Produtor/Consumidor, Mestre/Escravo, Monte Carlo...

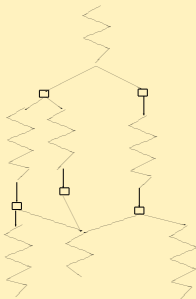


Os fluxos de execução compartilham áreas de dados. As escritas e leituras nesta área permitem a comunicação entre os fluxos

- Além de primitivas para criação e controle do término de fluxos de execução, se fazem necessários mecanismos para sincronizar as operações de leitura e escrita no dado compartilhado

Fluxo de dados

- as atividades comunicam entre si por meio de passagem de parâmetros e retorno de resultados



- Primitivas para criação e controle do término de fluxos de execução são suficientes

Modelos de decomposição paralela

Primeiros passos

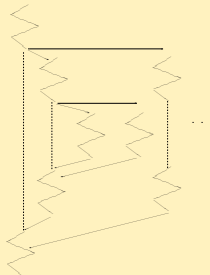
Exposição da concorrência

Recursivo

Ferramentas

Keep walking

- Um fluxo de execução se decompõe em novos fluxos de execução e permanece bloqueado, aguardando o resultado dos fluxos criados para compor agregar os resultados parciais e fornecer a solução global



- Primitivas para criação e controle do término de fluxos de execução são suficientes. A natureza do problema é estruturada e não existe necessidade da identificação individual de cada fluxo de execução

Modelos de decomposição paralela

Primeiros passos

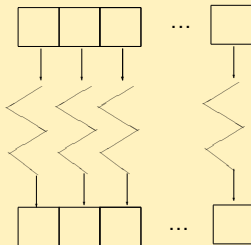
Exposição da concorrência

Ferramentas

Keep walking

Iterativo

- Cada elemento de um espaço de dados deve sofrer uma operação, de forma independente dos demais



- Deve haver recursos para decompor o espaço de dados em unidades de cálculo independentes

Modelos de decomposição paralela

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Iterativo

Um exemplo: computar o quadrado do valor de todas as posições de um vetor

```
for( i = 0 ; i < TAM ; i++ )  
    vetor[i] = vetor[i] * vetor[i];
```

Neste exemplo, cada posição [i] é atualizada com o quadrado do valor original

Modelos de decomposição paralela

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Iterativo

Outro exemplo: fazer com que a posição i de um vetor seja a soma de todos os valores armazenados neste mesmo vetor em todas posições anteriores a i

```
for( i = 1 ; i < TAM ; i++ )  
    vetor[i] = vetor[i] + vetor[i-1];
```

Neste exemplo, a paralelização não é trivial, pois existe uma dependência entre a execução da iteração i com a iteração $i-1$

Iterativo

Na verdade, existem três situações que podem ocasionar problema por dependência

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

1) aux = x[1] x[1] = y[0] y[1] = x[2],aux

2) aux = x[2] x[2] = y[1] y[2] = x[3],aux

3) aux = x[3] x[3] = y[2] y[3] = x[4],aux

4) aux = x[4] x[4] = y[3] y[4] = x[5],aux

Modelos de decomposição paralela

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Iterativo

Na verdade, existem três situações que podem ocasionar problema por dependência

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

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

Dependência verdadeira: o cálculo na iteração i requer um resultado computado na iteração $i-1$

Modelos de decomposição paralela

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Iterativo

Na verdade, existem três situações que podem ocasionar problema por dependência

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

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

Antidependência: o cálculo na iteração i requer um valor na posição $i+1$ que já pode ter sido alterado

Modelos de decomposição paralela

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Iterativo

Na verdade, existem três situações que podem ocasionar problema por dependência

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

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

Dependência de saída: em uma mesma iteração, um dado é atualizado e lido, havendo a possibilidade de conflito entre duas iterações

Tipos de fluxos de execução

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Thread

- Corpo em uma função ou em um método
- Pode ser uma abstração para uma sequência de tarefas
- Pode ser escalonada em nível aplicativo ou sistema

Tanto threads como tarefas podem ser identificados individualmente ou serem anônimos

Tarefa

- Corpo em um bloco de comandos ou em um método
- Constitui em uma unidade de execução escalonável em nível aplicativo

Tipos de fluxos de execução

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Thread ou tarefas identificadas individualmente

- A operação de criação de um novo fluxo de execução retorna um identificador único para o novo fluxo criado
 - Este identificador consiste em um dado, armazenado em uma variável no programa. Como qualquer variável, pode ser passado como parâmetro para qualquer outro fluxo de execução e então ser objeto de sincronização
 - A consequência é que a concorrência de um programa pode ser descrita de forma irregular (não estruturada)
- Um programa irregular é mais difícil de ser escalonado em nível aplicativo, uma vez que o mecanismo de escalonamento não pode aplicar nenhuma heurística que considere como a concorrência do programa irá se desenvolver

Tipos de fluxos de execução

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Thread ou tarefas anônimas

- A operação de criação de um novo fluxo de execução não possui retorno
 - Não raro, a operação de criação nestes casos é capaz de criar diversos fluxos de execução simultaneamente
 - O fluxo de execução que criou um (ou uma coleção de) fluxo é responsável por sincronizar com seu término
 - A consequência é que a concorrência é descrita de forma regular, mais precisamente, aninhada
- É comum impor abstrações de alto nível na interface de ferramentas de programação para promover seu desempenho por heurísticas de execução que considerem a estrutura do programa

Onde o problema se encontra

- Na especificação do corpo das atividades concorrentes
- Nas formas de coordenação entre as atividades
- Nos mecanismos de compartilhamento dos dados

Níveis de descrição da concorrência

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Granularidade

Quantidade de cálculo associada a uma atividade

A concorrência pode ser expressa em termos de:

- instrução
 - ◆ intra-instrução (instruções CISC)
 - ◆ arquiteturas vetoriais
- blocos
- procedimentos/funções
- processos
- aplicações

Níveis de descrição da concorrência

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Granularidade

Quantidade de cálculo associada a uma atividade

A concorrência pode ser expressa em termos de:

- instrução
 - ◆ intra-instrução (instruções CISC)
 - ◆ arquiteturas vetoriais
- **blocos**
- **procedimentos/funções**
- processos
- aplicações

Níveis de descrição da concorrência

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Granularidade

Quantidade de cálculo associada a uma atividade

A concorrência pode ser expressa em termos de:

- instrução
 - ◆ intra-instrução (instruções CISC)
 - ◆ arquiteturas vetoriais
- **blocos**
- **procedimentos/funções**
- processos
- aplicações

Em um mesmo nível, diferentes tamanhos de grão indicam a quantidade de interação do programa com o mecanismo de escalonamento: quanto maior o grão, menor será a interferência do escalonamento e, potencialmente, pior o escalonamento

Controle de evolução das atividades

■ Criação/término das atividades

- ◆ o estado das tarefas criador e criada ou terminada e sincronizada são mutuamente conhecidos

■ Mecanismos de exclusão mútua

- ◆ garantem execução em regime de exclusividade

■ Mecanismos de controle de fluxo

- ◆ permitem cadenciar a evolução da execução

- O aspecto relevante é garantir que as atividades parceiras em uma operação de sincronização tenham mútuo conhecimento sobre seus estados de execução

1 Primeiros passos

- Conceitos
- Arquiteturas paralelas
- Sistemas operacionais
- Ferramentas de programação multithread

2 Exposição da concorrência

3 Ferramentas

- Pthreads
- C++11
- OpenMP
- Cilk Plus
- TBB

4 Keep walking

Pthreads

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

- Conceito de thread
- Criação e sincronização de threads
- Mecanismos de coordenação
- Atributos de threads

Pthreads: Conceito de thread

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Um thread é definido pelo corpo de uma função com a seguinte estrutura:

```
void* foo( void* dta ) {  
    ...  
    ...  
    return ret;  
}
```

- Os parâmetros e o retorno da função executada pelo thread representam endereços de memória (estática ou dinâmica)

Pthreads: Conceito de thread

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Observe a diferença na alocação dos dados retornados pela função foo

```
void* foo( void* dta ) {  
    void *ret=malloc(sizeof(int));Certo  
    ...  
    return ret;  
}
```

```
void* foo( void* dta ) {  
    int ret;Errado  
    ...  
    return &ret;  
}
```

• Os parâmetros e o retorno da função executada pelo thread representam endereços de memória (estática ou dinâmica)

Manipulação dos threads

```
void* foo( void* dta ) {  
    ...  
    return ret;  
}
```

- `return` ou `pthread_exit` para terminar um thread
- `pthread_create` lança a execução de um novo thread
- `pthread_join` sincroniza a execução do thread corrente sobre um thread especificado

Obs.: as funções retornam um código de erro (0 sem erro)

Manipulação dos threads

```
void* foo( void* dta ) {  
    ...  
    return ret;  
}
```

- return ou pthread_exit para terminar um thread
- pthread_create lança a execução de um novo thread
 - ◆ pthread_create(pthread_t &id, pthread_attr_t &atrib, (void*)(*func)(void*), void* dta)
- pthread_join sincroniza a execução do thread corrente sobre um thread especificado
 - ◆ pthread_join(pthread_t &id, void** dta)

Manipulação dos threads

```
void* quadrado( void* dta ) {  
    int* q = (int*)malloc(sizeof(int));  
    *(int*)q = *(int*)dta * *(int*)dta;  
    return q;  
}  
  
int num = 20;  
int main() {  
    pthread_t id;  
    ...  
    pthread_create(&id, NULL, quadrado, &num);  
    ...  
    pthread_join(&id, &ret);  
}
```

- Onde a memória alocada para q deve ser liberada?

- Caso o parâmetro dta refira-se a uma variável anônima, onde ela deveria ser liberada?

Manipulação dos threads

- Submetidos ao escalonamento do sistema operacional
- Autônomos entre si
- Atributos dos threads (NULL utiliza os valores default)
 - ◆ tamanho da pilha
 - ◆ tipo de escalonamento (round-robin, FIFO, OTHER)
 - ◆ prioridade
 - ◆ joinable/detached
 - ◆ afinidade
 - ◆ ...
- Um thread pode suportar apenas um join (joinable)

Pthreads: Conceito de thread

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Manipulação dos threads

Com a primitiva `pthread_once` é possível garantir que um trecho de código será executado uma única vez, independente do número de vezes que for invocado

```
pthread_once_t ctr = PTHREAD_ONCE_INIT;

void Incializacao() { FazAlgumaCoisa(); }

void* CorpoThread(void *dta) {
    pthread_once( &ctr, Incializacao );
}
```

Coordenação entre threads

- `pthread_mutex_t`: permite o controle da execução em regime de exclusão mútua
- `pthread_cond_t`: permite controlar o avanço de threads em função de uma condição dada pelo programa
- As variáveis definidas a partir de tipos de dados definidos na biblioteca Pthreads devem ser utilizadas como **dados opacos**, ou seja, devem ser acessadas exclusivamente pelas primitivas de manipulação oferecidas

Pthreads: Compartilhamento de dados

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Coordenação entre threads

Uma **seção crítica** consiste em um trecho de código que acessa um dado compartilhado

■ pthread_mutex_t

- ◆ pthread_mutex_lock: adquire o direito de executar em regime de exclusão mútua uma seção crítica
- ◆ pthread_mutex_unlock: informa saída de uma seção crítica

int saldo = 313;	
Thread A	Thread B
1. auxA = saldo;	1. auxB = saldo;
2. auxA++;	2. auxB++;
3. saldo = auxA;	3. saldo = auxB;

• Qual o resultado em saldo caso a execução das instruções fosse A.1, A.2, B.1, A.3, B.2, B.3?

Pthreads: Compartilhamento de dados

Primeiros passos

Coordenação entre threads

Keep walking

Uma **seção crítica** consiste em um trecho de código que acessa um dado compartilhado

■ pthread_mutex_t

- ◆ pthread_mutex_lock: adquire o direito de executar em regime de exclusão mútua uma seção crítica
- ◆ pthread_mutex_unlock: informa saída de uma seção crítica

<pre>int saldo = 313; pthread_mutex_t m;</pre>	
Thread A	Thread B
<pre>0. pthread_mutex_lock(&m); 1. auxA = saldo; 2. auxA++; 3. saldo = auxA; 4. pthread_mutex_unlock(&m);</pre>	<pre>0. pthread_mutex_lock(&m); 1. auxB = saldo; 2. auxB++; 3. saldo = auxB; 4. pthread_mutex_unlock(&m);</pre>

- A ordem de execução das seções críticas em A e em B não pode ser determinada

Coordenação entre threads

Uma **condição** é um estado atingido por um dado no contexto do programa de aplicação

■ pthread_cond_t

- ◆ pthread_cond_signal:
- ◆ pthread_cond_broadcast: ambas primitivas informam que uma condição foi satisfeita. Signal ativa um único thread que esteja aguardando a condição, broadcast ativa todos os threads bloqueados na condição
- ◆ pthread_cond_wait: bloqueia o thread de forma a aguardar a sinalização de condição satisfeita

• Diferente do mutex, a variável de condição não possui estado. Na sua manipulação, assume-se que a condição está satisfeita no justo momento em que foi sinalizada, mas nada pode se dizer sobre o estado da condição no instante imediatamente posterior

Pthreads: Compartilhamento de dados

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Coordenação entre threads

■ pthread_cond_t

```
int nbelem = 0;
Buffer b;
pthread_mutex_t m;
pthread_cond_t c;
```

Thread Produtor

```
for(;;) {
    it = ProduzItem();
    pthread_mutex_lock(&m);
    Insere(b,it);
    nbelem++;
    if( nbelem == 1 )
        pthread_cond_signal(&c);
    pthread_mutex_unlock(&m);
}
```

Thread Consumidor

```
for(;;) {
    pthread_mutex_lock(&m);
    while( nbelem == 0 )
        pthread_cond_wait(&c,&m);
    it = Retira(b);
    nbelem- -;
    pthread_mutex_unlock(&m);
    ProcessaItem(it);
}
```

Pthreads: Compartilhamento de dados

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Coordenação entre threads

■ pthread_cond_t

```
int nbelem = 0;
Buffer b;
pthread_mutex_t m;
pthread_cond_t c;
```

Thread Produtor

```
for(;;) {
    it = ProduzItem();
    pthread_mutex_lock(&m);
    Insere(b,it);
    nbelem++;
    if( nbelem == 1 )
        pthread_cond_signal(&c);
    pthread_mutex_unlock(&m);
}
```

Thread Consumidor

```
for(;;) {
    pthread_mutex_lock(&m);
    while( nbelem == 0 )
        pthread_cond_wait(&c,&m);
    it = Retira(b);
    nbelem- -;
    pthread_mutex_unlock(&m);
    ProcessaItem(it);
}
```


Pthreads: É bom saber que...

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

- Chamadas aos serviços de Pthreads implicam em sobrecustos de execução
 - ◆ Uma estratégia comum na implementação de programas é criar um banco de threads responsáveis para execução e fazer com que estes threads compartilhem uma lista de trabalho
- A utilização de mecanismos de contenção reduzem o potencial de execução paralela
 - ◆ `pthread_mutex_trylock` como alternativa ao `lock`
 - ◆ `pthread_mutex_tryjoin` como alternativa ao `join`
- Um mutex pode ser definido como reentrante
 - ◆ Neste caso, se um mesmo thread executar mais de uma operação de lock sobre o mesmo mutex, não haverá bloqueio. Um cuidado a tomar é executar o mesmo número unlocks
- `pthread_once` define uma função que será executada uma única vez, independente do número de vezes que for chamada

Pthreads: Escalonamento

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

- O padrão estabelece que três tipos de escalonamento podem ser selecionados no momento do lançamento de um thread: round-robin, FIFO e OTHER (default)
 - ◆ Todos apoiados nos mecanismos de escalonamento do sistema operacional
- Atributos de criação de threads permitem especificar o tipo de escalonamento que o thread deve receber e sua prioridade
- O tamanho da pilha (dados locais) do thread também pode ser especificado nos atributos
- Não existe uma heurística que procure otimizar algum índice de desempenho
 - ◆ A não ser aquelas no sistema operacional que buscam otimizar a localidade dos threads

Threads estão incluídas na linguagem. Como C++ é uma linguagem orientada a objetos, entende-se que os recursos para manipular threads estejam disponíveis no contexto de classes e objetos

É mais do que uma interface para Pthreads, pois faz parte da linguagem e não há problemas de incompatibilidade na manipulação dos dados

É mais do que uma interface para Pthreads, pois aumenta a capacidade de expressão do programa

O suporte adjacente são bibliotecas nativas do sistema operacional (fatalmente baseadas em Pthreads)

C++11: Corpo de um thread

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

a classe `std::thread` define um contêiner para um thread a ser executado. O corpo do thread é passado como parâmetro de construção de um objeto desta classe

O corpo propriamente dito pode ser expresso por:

- uma função convencional
- um método qualquer de um objeto
- um objeto contendo o operador() redefinido
- uma função lambda

Em todos os casos, o retorno é representado pelo tipo `void`, mas não há restrição quanto ao número e/ou tipo de parâmetros

C++11: Corpo de um thread

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Exemplos

```
void CorpoUm(int n) {  
    foo(n);  
}  
  
class CorpoDois {  
    void operator()(int n){  
        foo(n);  
    }  
};  
  
class CorpoTres {  
    int n;  
public:  
    CorpoTres(int n):n(n){}  
    void bar(){foo(n);}  
};
```

```
... main ... {  
    CorpoDois cp2;  
    CorpoTres cp3(3);  
    std::thread t1(CorpoUm,1);  
    std::thread t2(cp2,2);  
    std::thread t3(&CorpoTres::bar,cp3);  
    std::thread t4([]{foo(4);});  
  
    t1.join(); t2.join(); t3.join();  
    t4.join();  
    return 0;  
}
```

C++11: Corpo de um thread

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Exemplos

```
void CorpoUm(int n) {  
    foo(n);  
}  
  
class CorpoDois {  
    void operator()(int n){  
        foo(n);  
    }  
};  
  
class CorpoTres {  
    int n;  
public:  
    CorpoTres(int n):n(n){}  
    void bar(){foo(n);}  
};
```

```
... main ... {  
    CorpoDois cp2(2);  
    CorpoTres cp3;  
    std::thread t1(CorpoUm,1);  
    std::thread t2(cp2,2);  
    std::thread t3(&CorpoTres::bar,cp3,3);  
    std::thread t4([]{foo(4);});  
  
    t1.join(); t2.join(); t3.join();  
    t4.join();  
    return 0;  
}
```

C++11: Corpo de um thread

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Exemplos

```
void CorpoUm(int n) {  
    foo(n);  
}  
  
class CorpoDois {  
    void operator()(int n){  
        foo(n);  
    }  
};  
  
class CorpoTres {  
    int n;  
public:  
    CorpoTres(int n):n(n){}  
    void bar(){foo(n);}  
};
```

```
... main ... {  
    CorpoDois cp2(2);  
    CorpoTres cp3(3);  
    std::thread t1(CorpoUm,1);  
    std::thread t2(cp2);  
    std::thread t3(&CorpoTres::bar,cp3);  
    std::thread t4([]{foo(4);});  
  
    t1.join(); t2.join(); t3.join();  
    t4.join();  
    return 0;  
}
```

C++11: Corpo de um thread

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Exemplos

```
void CorpoUm(int n) {  
    foo(n);  
}  
  
class CorpoDois {  
    void operator()(int n){  
        foo(n);  
    }  
};  
  
class CorpoTres {  
    int n;  
public:  
    CorpoTres(int n):n(n){}  
    void bar(){foo(n);}  
};
```

```
... main ... {  
    CorpoDois cp2(2);  
    CorpoTres cp3;  
    std::thread t1(CorpoUm,1);  
    std::thread t2(cp2);  
    std::thread t3(&CorpoTres::bar,cp3);  
    std::thread t4([]{foo(4);});  
  
    t1.join(); t2.join(); t3.join();  
    t4.join();  
    return 0;  
}
```


C++11: parâmetros por referência

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Observe que um objeto thread recebe, como parâmetro, o corpo do thread a ser executado. Assim, mesmo que o corpo de uma função identifique uma passagem por referência, a referência recebida será para uma cópia do dado armazenado no objeto thread:

```
void incrementa( int& dta ) { dta++; }  
void main() {  
    int n = 313;  
    std::thread errado(incrementa,n); É passado uma cópia  
    std::cout << n << std::endl;  
    std::thread certo(incrementa,std::ref(n));    std::cout << n <<  
std::endl;  
    return 0;  
}
```

C++11: parâmetros por referência

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Outro exemplo

```
struct Fibo {  
    int n, r;  
    Fibo(const int n) n(n), r(0) {}  
    void operator()() { r = FiboSeq(n); }  
    int getRes() { return r; }  
};  
  
int main() {  
    Fibo f(20);  
    thread tfibo(std::ref(f));  
    tfibo.join();  
    std::cout << "Fibonacci(20) = " << tfibo.getRes();  
    return 0;  
}
```

C++11: Coordenação entre threads

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Mecanismos oferecidos na forma de classes, sendo acessados por meio de objetos

- classe `mutex`
- classe `condition_variable`

As funcionalidades oferecidas, na forma de métodos, são equivalentes aqueles oferecidos em Pthreads

C++11: Coordenação entre threads

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Funcionalidades extras na manipulação de mutexes

- a função auxiliar **std::lock**
 - ◆ recebe vários objetos mutex e retorna apenas quando todos os mutexes forem obtidos
- a classe **std::lock_guard**
 - ◆ quando um thread termina sem que um mutex seja liberado, o objeto é destruído e seu estado indeterminado. Para prevenir tal situação, um objeto desta classe pode ser criado para servir como contêiner de um objeto mutex

C++11: Coordenação entre threads

Funções básicas Funções de manipulação de mutexes Keep walking

```
struct Dados {
    int vA, vB;
    std::mutex m;
    Dados(const Dados& d) { vA = d.vA; vB = d.vB; }
    Dados& operator=(const Dados& d) { vA = d.vA; vB = d.vB;
    return *this;
}
};

void swap( Dados& a, Dados&b ) {
    std::lock(a.m,b.m,a.m,a.m);
    std::lock_guard <std::mutex> guardaA(a.m,std::adopt_lock);
    std::lock_guard <std::mutex> guardaB(b.m,std::adopt_lock);
    Dados temp(a); a = b; b = temp;
}
```

No exemplo, os mutexes já foram adquiridos (adopt_lock), mas poderiam ser solicitados (try_to_lock) ou assumir que eles serão adquiridos na sequência (defer_lock)

C++11: Coordenação entre threads

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Acesso atômico

A classe genérica `std::atomic` permite realizar acesso atômico a variáveis de tipos primitivos de C++ (char, int, ullong...)

```
struct Contador {
    std::atomic<int> cont;
    Contador( int v = 0 ) { cont.store(v); }
    int operator++() { ++cont; return cont.load(); }
    int operator--() { --cont; return cont.load(); }
    int get() { cont.load(); }
};

int main() {
    static Contador c;
    std::cout << c.get() << std::endl << ++c << std::endl
               << ++c << std::endl << --c << std::endl
               << c.get() << std::endl;
    return 0; }
```

C++11: futures e promisses

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Permite construir relações de dependências de dados entre threads

- Um objeto **promessa** é representa uma *dívida*, soldada na quando receber um valor
- Um objeto **futuro** é representa a *cobrança*, soldada na quando receber um valor
- Quando a *dívida* é assumida, é estabelecido canal para o *pagamento*

C++11: futures e promisses

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Permite construir relações de dependências de dados entre threads

```
void produz(std::promise<int>& prom) {
    prom.set_value(313);
}

void consome(std::future<int>& fut) {
    std::cout << "Valor: " << fut.get();
}

int main() {
    std::promise<int> prom;
    std::future<int> fut = prom.get_future();
    std::thread thProd(produz, std::ref(prom));
    std::thread thCons(consome, std::ref(fut));
    thProd.join(); thCons.join();
    return 0;
}
```


C++11: futures e promisses

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Permite construir relações de dependências de dados entre threads

```
void produz(std::promise<int>& prom) {  
    prom.set_value(313);  
}  
  
void consome(std::future<int>& fut) {  
    std::cout << "Valor: " << fut.get();  
}  
  
int main() {  
    std::promise<int> prom;  
    std::future<int> fut = prom.get_future();  
    std::thread thProd(produz, std::ref(prom));  
    std::thread thCons(consome, std::ref(fut));  
    thProd.join(); thCons.join();  
    return 0;  
}
```

C++11: futures e promisses

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Permite construir relações de dependências de dados entre threads

Usar quando o custo para produzir resultados não compensa a criação de inúmeros threads

Ver exemplo mais elaborado

OpenMP

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

- Início dos esforços datam dos anos 1990
- Padrão estabelecido por uma parceria de diversas empresas
 - ◆ Para C/C++ e Fortran
- As versões recentes inovam ao:
 - ◆ explicitar o conceito de tarefa
 - ◆ incluir manipulação atômica de variáveis
- Voltada à programação
- Busca explorar o paralelismo para processamento de alto desempenho
 - ◆ Escalonamento em nível aplicativo
- Modelo $n \times m$
 - ◆ tarefa
 - criadas de forma **implícita** ou **explícita**
 - ◆ thread
 - um processador virtual é chamado de **thread**
 - um grupo de threads compõe um **pool de execução**

Princípio básico

O programador expõe a concorrência de seu programa utilizando as diretivas de paralelização, mas é o ambiente de execução que ativa a execução paralela efetivamente

Simplicidade de programação

Retirando as diretivas de paralelização, o programa torna-se um programa sequencial inteiramente operacional

Portabilidade de desempenho

Variando o número de threads no pool de execução o paralelismo pode ser adequado às capacidades do hardware disponível

OpenMP: interface básica

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

- **Diretivas de paralelização:** expõem o paralelismo da aplicação, podem ser parametrizados com cláusulas
 - ◆ São tratadas em um processo de pré-processamento
- **Primitivas de serviços:** permitem interações do programa com o ambiente de execução
 - ◆ A biblioteca é ligada ao programa
- **Variáveis de ambiente:** definem comportamentos padrões para os programas. São geridas pelo sistema operacional
 - ◆ Os valores padrões são considerados no momento do lançamento do programa

OpenMP: interface básica

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

- **Diretivas de paralelização:** expõem o paralelismo da aplicação, podem ser parametrizados com cláusulas
 - ◆ São tratadas em um processo de pré-processamento
- **Primitivas de serviços:** permitem interações do programa com o ambiente de execução
 - ◆ A biblioteca é ligada ao programa
- **Variáveis de ambiente:** definem comportamentos padrões para os programas. São geridas pelo sistema operacional
 - ◆ Os valores padrões são considerados no momento do lançamento do programa

Nos interessa ver as **diretivas de paralelização**, algumas **primitivas de serviços** serão vistas a medida em que ocorrerem nos exemplos

OpenMP: modelo de execução

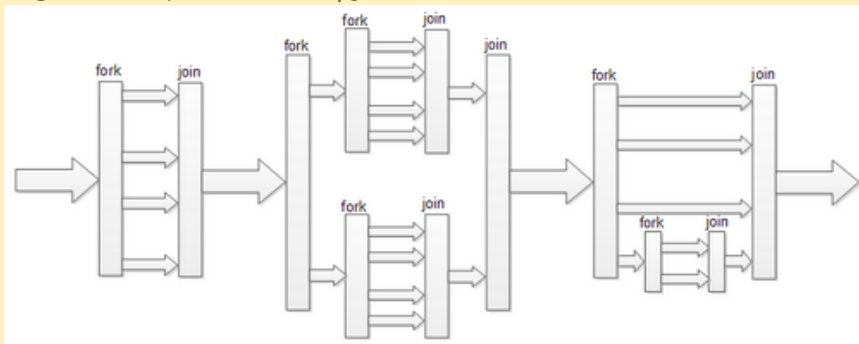
Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Segue um esquema de **fork/join** aninhados



OpenMP: regiões paralelas

Identifica o início de um trecho de código *produtor de tarefas*.

Primeiros passos

Exposição da concorrência

Ferramentas

Keep working

Inicializa o pool de execução e o escalonamento

Diretiva **parallel**

```
...  
#pragma omp parallel [cláusulas]  
  Comando  
...
```


OpenMP: regiões paralelas

Identifica o início de um trecho de código *produtor* de tarefas.

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Inicializa o pool de execução e o escalonamento

Diretiva **parallel**

```
...  
#pragma omp parallel [cláusulas]  
  Comando  
...
```

Os comandos que seguem identificam as tarefas a serem geradas

OpenMP: regiões paralelas

Identifica o início de um trecho de código *produtor de tarefas*.

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Inicializa o pool de execução e o escalonamento

Diretiva **parallel**

```
...  
#pragma omp parallel [cláusulas]  
  Comando  
...
```

Os comandos que seguem identificam as tarefas a serem geradas

- O thread que executa a diretiva parallel é chamado de thread master
- Como a tarefa em execução no thread master é suspensa enquanto as tarefas criadas não forem executadas, o thread master participa da computação das tarefas no pool de execução
- A tarefa que originou a concorrência somente é retomada quando todos os threads criados tiverem terminado

Diretiva **parallel**

```
#pragma omp parallel
{
    printf("%d", omp_get_thread_num());
    if( omp_get_thread_num() == 0 )
        printf("%d", omp_get_num_threads());
}
```

- Neste exemplo as tarefas são criadas de forma implícita
- Tantas tarefas serão criadas quanto for o número de threads no pool de execução

Diretivas **sections** e **section**

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            foo();
        }
        #pragma omp section
        {
            bar();
        }
        ...
    }
}
```

- São criadas tantas tarefas quanto forem as seções especificadas

OpenMP: criação implícita de tarefas

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Diretivas **single** e **master**

```
#pragma omp parallel
{
    printf("%d",
omp_get_thread_num());
    #pragma omp master
        printf("%d",
omp_get_num_threads());
}
```

- É criada uma única tarefa, a ser executada por qualquer thread (single) ou exclusivamente no thread (master)

OpenMP: criação implícita de tarefas

Diretiva **for**

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

```
#pragma omp parallel
#pragma omp for
for( i = 0 ; i < TAM ; i++ )
    vet[i] = vet[i] * vet[i];
```

- O laço é fatiado em *chunks*, cada chunk consistem em uma sequência de iterações
- Algumas restrições à paralelização automática do for
 - ◆ a variável de iteração, o limite superior e o valor inicial devem ser do tipo inteiro com sinal
 - ◆ a variável de iteração pode ser apenas incrementada ou decrementada no final de cada iteração
 - ◆ apenas os operadores relacionais $<$, \leq , $>$ e \geq podem ser utilizados

Diretiva **for**

```
#pragma omp parallel schedule(static,2)
#pragma omp for
for( i = 0 ; i < TAM ; i++ )
    vet[i] = vet[i] * vet[i];
```

- O escalonamento pode ser (informado com uma cláusula)
 - ◆ static (default): indicado quando as iterações possuem o mesmo custo computacional. Todos os chunks são alocados no lançamento aos threads
 - ◆ dynamic: os custos computacionais das iterações são diferentes. Cada thread recebe um chunk para iniciar o processamento, os demais permanecem em uma lista compartilhada de forma a balancear a carga
 - ◆ guided: indicado quando associado à cláusula nowait

OpenMP: criação implícita de tarefas

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Supondo que existam **4 threads** no pool de execução, quantas tarefas são geradas no código abaixo?

```
#pragma omp parallel
{
    printf("A");
    for( i = 0 ; i < 2 ; i++ )
        printf("B");
    #pragma omp for
    for( i = 0 ; i < 4 ; i++ )
        printf("C");
    #pragma omp sections
    {
        #pragma omp section
        printf("D");
        #pragma omp section
        printf("E");
    }
}
```


OpenMP: criação implícita de tarefas

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Supondo que existam **4 threads** no pool de execução, quantas tarefas são geradas no código abaixo?

```
#pragma omp parallel
{
    printf("A");
    for( i = 0 ; i < 2 ; i++ )
        printf("B");
    #pragma omp for
    for( i = 0 ; i < 4 ; i++ )
        printf("C");
    #pragma omp sections
    {
        #pragma omp section
        printf("D");
        #pragma omp section
        printf("E");
    }
}
```

Será impresso, em alguma ordem:

- A A A A
- B B B B B B B B
- C C C C
- D
- E

Portanto, 18 tarefas.

OpenMP: relaxamento das barreiras

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Cláusula **nowait**

- Por padrão, o término do comando associado a uma região paralela representa uma barreira, nenhum thread pode avançar a computação enquanto todas as tarefas definidas não forem completadas
- A cláusula **nowait** relaxa esta condição de barreira, permitindo que um thread avance sobre uma nova fonte de tarefas
- O escalonamento **guided** passa a ser interessante, uma vez que o primeiro thread que chegar a esta fonte de tarefas pegará o maior chunk, adiantando o serviço

OpenMP: Compartilhamento de dados

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Todos os identificadores que compõem o escopo de uma região paralela são acessíveis nas tarefas criadas. Devem ser definidos, por meio de cláusulas, a forma como os dados serão compartilhados entre as tarefas.

- **shared**: as variáveis listadas são compartilhadas entre as tarefas
- **private**: cada tarefa possui uma instância própria de cada identificador listado
- **firstprivate**: cada tarefa possui uma instância própria de cada identificador listado, sendo esta inicializada com o último valor anotado na tarefa geradora
- **lastprivate**: cada tarefa possui uma instância própria de cada identificador listado, sendo que a variável original é atualizada com o valor anotado na instância local à última tarefa executada
- **reduction**: cada tarefa possui uma instância própria dos identificadores listados. Ao final da execução de cada tarefa, uma operação de redução combina os valores armazenados em cada cópia sobre a variável original.

Existem comportamentos padrões para as diferentes diretivas, convém explicitar todas as formas de compartilhamento para evitar consequências indesejadas.

Exemplo **private** e **shared**

```
int quant, num; #pragma omp parallel shared(quant), private(num)
{
    num = omp_get_thread_num();
    printf("%d", num);
    if( num == 0 ) {
        quant = omp_get_num_threads();
        printf("%d", quant);
    }
}
```

OpenMP: Compartilhamento de dados

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Exemplo **reduction**

```
int i, soma = 0;
int vet[TAM];
...
#pragma omp parallel
  #pragma omp for reduction(soma,+), shared(vet), private(i)
    for( i = 0 ; i < TAM ; i++ )
      soma += vet[i];
```

- A redução implica que cada tarefa possua uma instância própria do dado a ser reduzido.
- Cada cópia é inicializada com o valor identidade da operação
- A operação de redução é aplicada sobre cada valor parcial obtido nas cópias sobre o dado original
- **Semântica de escritas concorrentes**

OpenMP: Compartilhamento de dados

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Exemplo **reduction**

```
int i, soma = 0;
int vet[TAM];
...
#pragma omp parallel
  #pragma omp for reduction(soma,+), shared(vet), private(i)
    for( i = 0 ; i < TAM ; i++ )
      soma += vet[i];
```

Operação	Operador	Identidade
Soma	+	0
Subtração	-	0
Multiplicação	*	1
E aritmético	&	xxx
Ou aritmético		0
XOR	^	sin0
E lógico	&&	1
Ou lógico		0

OpenMP: Compartilhamento de dados

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Diretiva **atomic**

- Permite realizar uma operação de acesso a dados, em escrita ou leitura, de forma atômica
- Apenas uma instrução é submetida a esta diretiva
- Caso a escrita seja resultado da avaliação de uma expressão **esta não é avaliada de forma atômica**
- São permitidos acesso de três tipos, read, write, update (default)

Incrementa

```
int cont;  
...  
#pragma omp atomic  
    cont++
```

Decrementa

```
...  
#pragma omp atomic  
    cont = count - 1;
```

Diretiva **critical**

- Delimita uma seção crítica (regime de exclusão mútua)
- Um rótulo permite identificar uma região crítica de interesse

```
void move( Buffer *a, Buffer *b ) {  
    Item* it;  
    #pragma omp critical  
        it = retira(a);  
    #pragma omp critical  
        it = insere(b);  
}  
... main ... {  
    Buffer bufA, bufB;  
    ...  
    move(bufA,bufB);  
    ...  
}
```


OpenMP: Compartilhamento de dados

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Diretiva **critical**

- Delimita uma seção crítica (regime de exclusão mútua)
- Um rótulo permite identificar uma região crítica de interesse

```
void move( Buffer *a, Buffer *b ) {  
    Item* it;  
    #pragma omp critical  
        it = retira(a);  
    #pragma omp critical  
        it = insere(b);  
}  
... main ... {  
    Buffer bufA, bufB;  
    ...  
    move(bufA,bufB);  
    ...  
}
```

Os acessos em retirada e inserção não podem ser executados de forma independente

OpenMP: Compartilhamento de dados

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Diretiva **critical**

- Delimita uma seção crítica (regime de exclusão mútua)
- Um rótulo permite identificar uma região crítica de interesse

```
void move( Buffer *a, Buffer *b ) {
    Item* it;
    #pragma omp critical
        it = retira(a);
    #pragma omp critical
        it = insere(b);
}

... main ... {
    Buffer bufA, bufB;
    ...
    move(bufA,bufB);
    ...
}
```

Os acessos em retirada e inserção não podem ser executados de forma independente

```
void move( Buffer *a, Buffer *b ) {
    Item* it;
    #pragma omp critical(remocao)
        it = retira(a);
    #pragma omp critical(insercao)
        it = insere(b);
}

... main ... {
    Buffer bufA, bufB;
    ...
    move(bufA,bufB);
    ...
}
```

Inserções e remoções podem ocorrer simultaneamente

Diretivas **task** e **taskwait**

- Funcionalidade que expõe a possibilidade de criação de tarefas de forma recursiva
- A barreira não é implícita, deve ser realizada com **taskwait**
- Permite construir uma estrutura de tarefas recursiva
- Tarefas podem ser *tied* ou *untied*. No primeiro caso (default), uma tarefa suspensa somente poderá reiniciar no thread que a lançou

Diferença entre **sections** e **task**

- As tarefas geradas em **sections** são escalonadas em um contexto encapsulado na estrutura **sections** (barreira de sincronização)
- As tarefas geradas por **task** são escalonadas em pontos específicos de escalonamento, com uma estratégia própria

OpenMP: Criação explícita de tarefas

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Diretivas **task** e **taskwait**

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

```
int main() {  
    int r;  
    #pragma omp parallel  
    {  
        #pragma omp single  
        r = Fibo(20);  
    }  
    printf ("fib(20) = %d", r);  
    return 0;  
}
```

OpenMP: Criação explícita de tarefas

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Escalonamento das tarefas criadas explicitamente

- Considera o relacionamento entre as tarefas e a sua natureza recursiva
- Operações de escalonamento ocorrem quando ocorre um ponto de sincronização (implícito ou explícito)
- As ações de escalonamento podem ser (sem prioridade entre elas)
 - ◆ 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 pronta
- O estado da lista de tarefas bloqueadas determina regras de seleção de tarefa
 - ◆ Lista vazia: uma tarefa *tied* pode ser lançada
 - ◆ Lista não vazia: uma tarefa *tied* somente poderá ser lançada se for descendente de todas as tarefas na lista de bloqueadas

Privilegia localidade de referência

- Ferramenta de programação que estende as funcionalidades da linguagem C com primitivas para exposição da concorrência na forma de um programa multithread
 - ◆ Threads anônimas
 - ◆ Programa estruturado (fork/join aninhado)
- Modelo $n \times m$
- Não existe o conceito de memória compartilhada, o que implica que a passagem de parâmetros e o retorno de resultados é o principal mecanismo de comunicação entre os threads
 - ◆ Existem uma abstração para objetos com uma política de redução
- Escalonamento baseado em um estratégia de fluxo de dados (minimização do *caminho crítico*)
- Interface despojada: `cilk_spawn`, `cilk_sync` e `cilk_for`

Exemplo de programa

```
int Fibo( int n ) {
    if( n < 2 ) return n;
    else {
        int x = cilk_spawn Fibo(n-1),
            y = Fibo(n-2);
        cilk_sync;
        return x+y;
    }
}

int main() {
    std::cout << "Fibonacci de 20"<< ": «< Fibo(20) << std::endl;
    return 0;
}
```

Exemplos de programa iterativo

Bom exemplo

```
...  
cilk_for(i = 0 ; i < TAM ; i++)  
    FazAlgumaCoisa(i);
```

Não há restrição quanto a composição do loop. A criação de tarefas se dá na forma de **divisão e conquista**

Mau exemplo

```
...  
for( i = 0 ; i < TAM ; i++ )  
    cilk_for FazAlgumaCoisa(i);
```

Neste contra-exemplo, são criadas TAM tarefas para realizar o cálculo

Cilk Plus: Modelo de execução

- A operação `cilk_spawn` cria um novo thread, no entanto, ao invés de submeter o novo thread ao escalonamento e prosseguir a execução do thread corrente, é iniciada a execução do thread criado e o thread antigo é que é suspenso
- Com apoio de um processo de compilação são gerados **clones lentos** e **clones rápidos** de cada thread
- Introduz os conceitos:
 - ◆ **tarefa**: conjunto de instruções entre duas operações de sincronização (`spawn`, `join`, início e final de thread)
 - ◆ **continuação**: primeira tarefa pronta em um thread suspenso
- Sempre que um thread é suspenso, sua continuação é armazenada em uma lista local
- Sempre que um processador virtual termina a execução de um thread, busca em sua lista local um novo trabalho para ser executado
 - ◆ Caso a lista local esteja vazia, a lista de outro processador virtual é visitada em busca de trabalho

Cilk Plus: Escalonamento

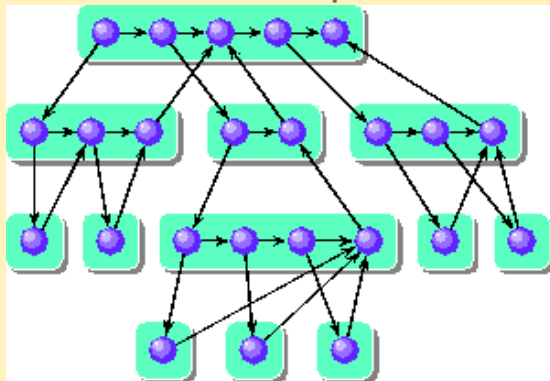
Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Grafo de dependências entre tarefas



- **Work-stealing**, roubo de trabalho
 - ◆ priorizando tarefas no caminho crítico
 - ◆ privilegiando explorar a localidade das tarefas
 - ◆ consciência da natureza recursiva e aninhada da aplicação
- Cada processador mantém uma lista local de tarefas prontas
 - ◆ novas tarefas são inseridas no início da lista local
 - ◆ ao buscar uma nova tarefa para executar em sua própria lista, a prioridade é da que estiver na cabeça da lista (a mais profunda)
- Em caso de um roubo de trabalho, um processador virtual é escolhido aleatoriamente e uma tarefa no final (mais próxima ao início do programa) é migrada
- A estratégia de roubo potencializa os benefícios da operação de migração de tarefa, uma vez que, considerando a natureza recursiva e aninhada do programa, quanto mais antiga a tarefa for, maior será seu potencial de geração de trabalho

Exemplo de exploração do escalonamento

// //

Boa implementação

```
int Fibo( int n ) {  
    if( n < 2 ) return n;  
    else {  
        int x = cilk_spawn Fibo(n-1),  
            y = Fibo(n-2);  
        cilk_sync;  
        return x+y;  
    }  
}
```

Má implementação

```
int Fibo( int n ) {  
    if( n < 2 ) return n;  
    else {  
        int x = cilk_spawn Fibo(n-2),  
            y = Fibo(n-1);  
        cilk_sync;  
        return x+y;  
    }  
}
```

Cilk Plus: Redução de dados

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

- Uma biblioteca fornece um conjunto de classes genéricas para realizar operações de redução de dados
- As classes `reducer_opadd` e `reducer_opmax` estão presentes, podendo novas classes serem desenvolvidas
- O tipo parametrizado a estas classes deve suportar as operações previstas para redução, no caso os operadores `+` e `+=` para `opadd` e o operador `<` para `opmax`

Exemplo de uso de operadores de redução

```
void Fibo(int n,  
cilk::reducer_opadd<int>& r)  
{  
    if (n < 2) r += n;  
    else {  
        cilk_spawn Fibo(n-1,r);  
        Fibo(n-2,r);  
        cilk_sync;  
    }  
}
```

```
int main() {  
    cilk::reducer_opadd<int> r;  
    Fibo(20,r);  
    std::cout << r.get_value() <<  
    std::endl;  
    return 0;  
}
```

TBB: Threading Building Blocks

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

- Conjunto de classes e funções genéricas em C++ para descrever com alto grau de abstração a concorrência de uma aplicação
- Interface baseada em C++, com conceitos herdados da STL (contêiner, iterador e algoritmo)
 - ◆ Dois níveis de abstração, o primeiro oferecendo um conjunto de **esqueletos** na forma de funções genéricas para implementar estruturas concorrentes recorrentes, o segundo expondo a existência de um grafo de dependência entre tarefas e a existência de um mecanismo de escalonamento
 - ◆ A forma usual de programação implica em estruturar tarefas em forks/joins aninhados, no entanto é possível criar tarefas fora desta estrutura (um passo para a irregularidade)
- Modelo $n \times m$
- Escalonamento baseado no esquema de adotado por Cilk Plus

TBB é a encarnação em C++ de Cilk Plus

TBB: Programado com esqueletos

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Esqueletos analisados: **parallel_for** e **parallel_reduce**

- Ambos representam estruturas iterativas
- O espaço de iteração a ser percorrido é dividido em tarefas
- A criação de tarefas é na forma de divisão em conquista

TBB: Programado com esqueletos

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

Esqueletos analisados: **parallel_for** e **parallel_reduce**

- Ambos representam estruturas iterativas
- O espaço de iteração a ser percorrido é dividido em tarefas
- A criação de tarefas é na forma de divisão em conquista

Ao contrário do que ocorrem em OpenMP e Cilk Plus, as iterações dos laços não são mapeadas em tarefas pelo ambiente de forma transparente ao programador. Ao contrário, ao programar os laços o programador deve ter ciência de que ele será executado de forma concorrente

TBB: Programado com esqueletos

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

parallel_for

```
struct Quadrado {
    int *vet;
    const int tam;
    Quadrado(int *v, int t):vet(v),tam(t) {}
    void operator()( const tbb::blocked_range<int>& r) const {
        for( int i = r.begin() ; i != r.end() ; i++ )
            vet[i] *= vet[i];
    }
};

...
int vet[TAM];
inicializa(vet,TAM);
Quadrado quad(vet,TAM);
tbb::parallel_for(tbb::blocked_range<int>(0,100),quad);
```

TBB: Programado com esqueletos

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

parallel_for

```
struct Quadrado {  
    int *vet;  
    const int tam;  
    Quadrado(int *v, int t):vet(v),tam(t) {}  
    void operator()( const tbb::blocked_range<int>& r) const {  
        for( int i = r.begin() ; i != r.end() ; i++ )  
            vet[i] *= vet[i];  
    }  
};  
  
...  
int vet[TAM];  
inicializa(vet,TAM);  
Quadrado quad(vet,TAM);  
tbb::parallel_for(tbb::blocked_range<int>(0,100),quad);
```

O método operator() é constante pois não pode alterar o estado interno do objeto

TBB: Programado com esqueletos

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

parallel_reduce

```
struct Soma {
    int *vet;
    int somaparcial;
    const int tam;
    Soma(int *v, int t):vet(v),tam(t) { somaparcial = 0; }
    void operator()( const tbb::blocked_range<int>& r) {
        for( int i = r.begin() ; i != r.end() ; i++ )
            somaparcial += vet[i];
    }
    void join( const Soma& s) { somaparcial += s.somaparcial; } };
...
int vet[TAM];
inicializa(vet,TAM);
Soma quad(vet,TAM);
tbb::parallel_reduce(tbb::blocked_range<int>(0,100),quad);
```

TBB: Programado com esqueletos

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

parallel_reduce

```
struct Soma {
    int *vet;
    int somaparcial;
    const int tam;
    Soma(int *v, int t):vet(v),tam(t) { somaparcial = 0; }
    void operator()( const tbb::blocked_range<int>& r) {
        for( int i = r.begin() ; i != r.end() ; i++ )
            somaparcial += vet[i];
    }
    void join( const Soma& s) { somaparcial += s.somaparcial; } };
...
int vet[TAM];
inicializa(vet,TAM);
Soma quad(vet,TAM);
tbb::parallel_reduce(tbb::blocked_range<int>(0,100),soma);
```

O operador() altera o estado interno do objeto. O atributo que faz a redução deve ser inicializado com o valor identidade da operação. O método join deve ser implementado com a redução propriamente dita

TBB: Interagindo com o escalonamento

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

A classe task

- Classe abstrata, devendo ser especializado o método `execute`
- Objetos desta classe devem ser construídos com o operador **new** redefinido pela interface
- Há uma diversidade de mecanismos, na forma de métodos, disponíveis para manipular tasks. A opção por um ou outro mecanismo influencia no desempenho final

Ver exemplo

1 Primeiros passos

- Conceitos
- Arquiteturas paralelas
- Sistemas operacionais
- Ferramentas de programação multithread

2 Exposição da concorrência

3 Ferramentas

- Pthreads
- C++11
- OpenMP
- Cilk Plus
- TBB

4 Keep walking

Keep walking

Primeiros passos

Exposição da concorrência

Ferramentas

Keep walking

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
Escalação em nível aplicativo	Não	Não	Sim	Sim	Sim
Decomposição do paralelismo	Explícita	Explícita	Implícita	Implícita	Implícita
Tamanho do executável	Pequeno	Grande	Pequeno	Médio	Grande