SSC-0742
PROGRAMAÇÃO CONCORRENTE

Aula 07 – Técnicas de Desenvolvimento de Programas Paralelos – Parte 2

Prof. Jó Ueyama

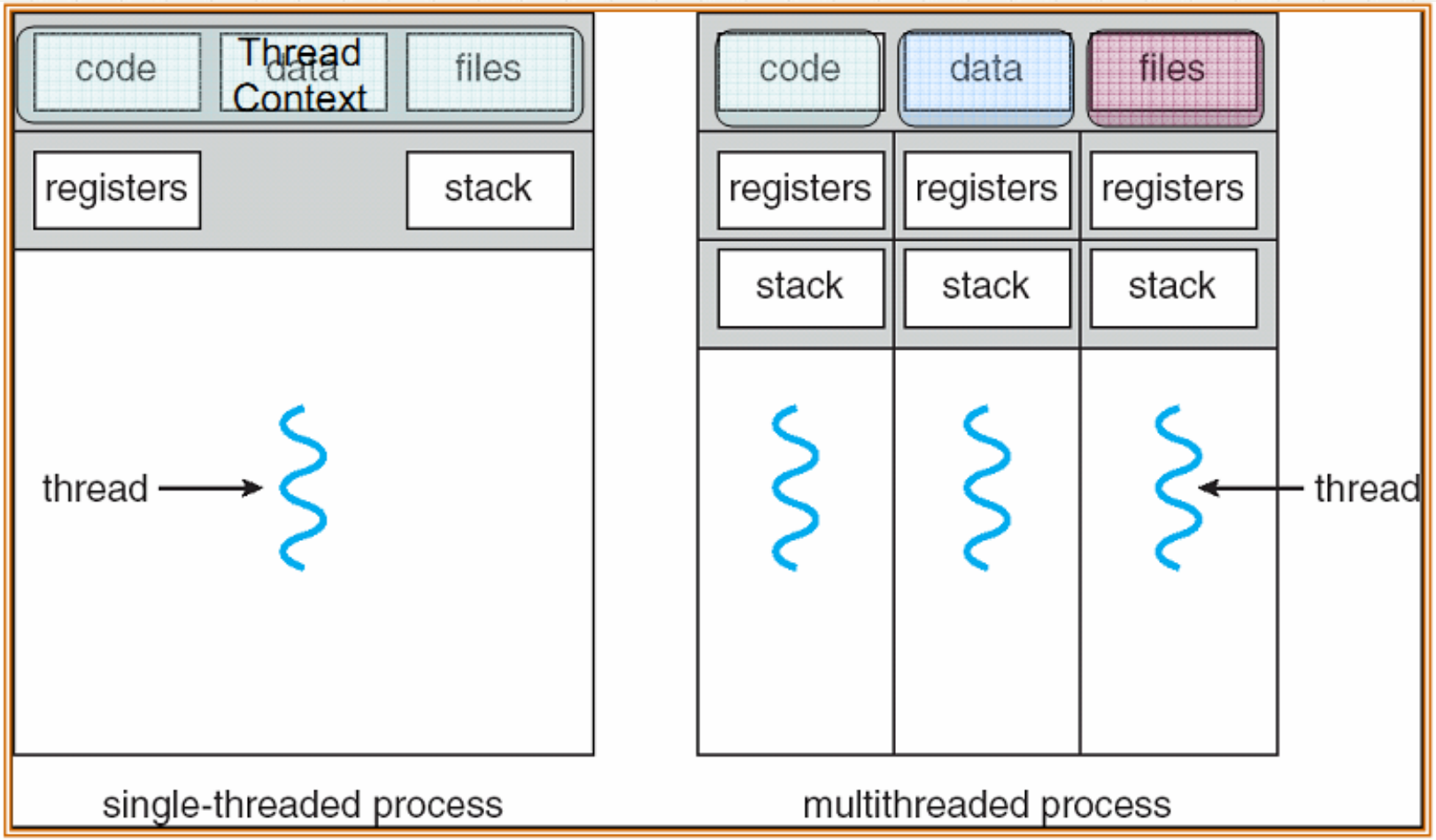
Créditos

Os slides integrantes deste material foram construídos a partir dos conteúdos relacionados às referências bibliográficas descritas neste documento

Visão Geral da Aula de Hoje

1. Threads
2. Sincronização
4. Dependência de Dados
5. Escalonamento
6. Granularidade
7. Exercício e Leitura Recomendada

Threads


code
data
files
Thread Context
registers
stack
thread
code
data
files
registers
registers
registers
stack
stack
stack
thread
single-threaded process
multithreaded process

Threads

- Um processo tradicional possui um fluxo de execução único
- No modelo multithread, em cada processo pode haver diversos fluxos de execução ou threads
- As diversas threads de um processo compartilham o espaço de endereçamento, mas apresentam contextos de execução diferentes

Threads

Exemplo

```
for (row = 0; row < n; row++)
  for (column = 0; column < n; column++)
    matC[row][col] = mult(matA, row, matB, col)
```

Programa Paralelo

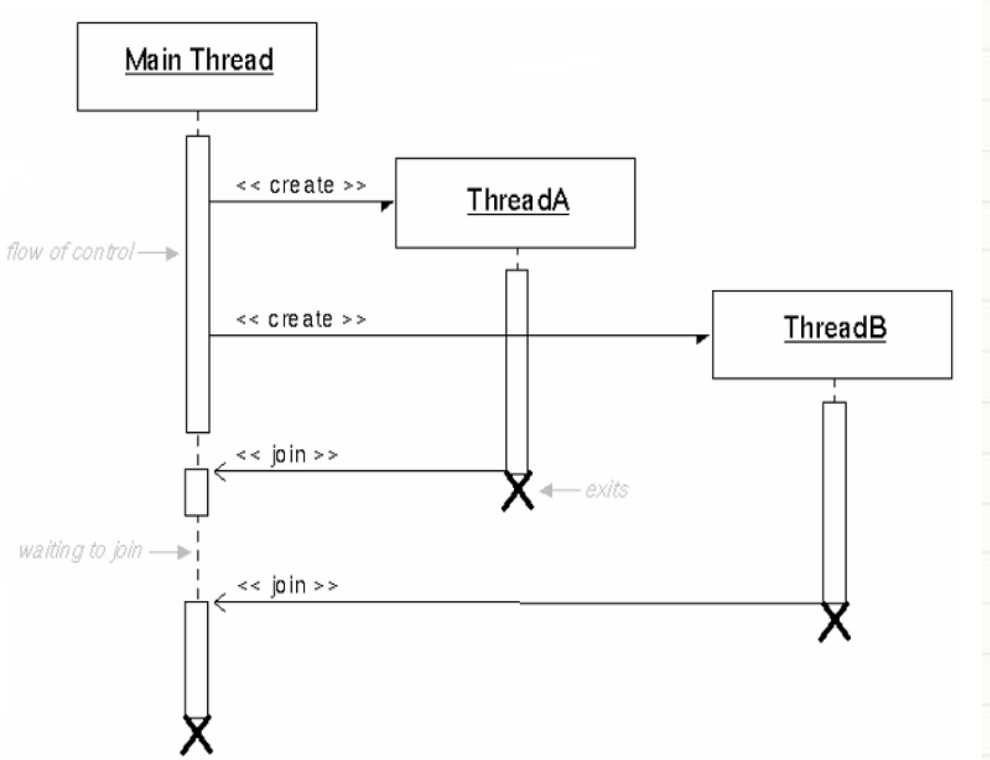
```
for (row = 0; row < n; row++)
  for (column = 0; column < n; column++)
    matC[row][col] = new_thread(mult(matA, row,
                                     matB, col))
```

Threads

- API de Threads POSIX
  - Pthreads (padrão para programação em threads)
  - Conceitos que implementam são independentes de plataformas (usado para programação de outras APIs)
    - Exploram os Wrappers para “resolver” o problema da heterogeneidade de arquiteturas
  - Acessível via C / C++
  - Suportada por SOs open-source

Threads

- Duas funções básicas para definir fluxos concorrentes
- Criação
  - pthread_create
- Término
  - pthread_join



Threads

```
#include <pthread.h>
#include <stdlib.h>
#define MAX_THREADS 16
void *compute_pi (void *);
....
main() {
    ...
    pthread_t p_threads[MAX_THREADS];
    pthread_attr_t attr;
    pthread_attr_init (&attr);
    for (i=0; i< num_threads; i++) {
       hits[i] = i;
       pthread_create(&p_threads[i], &attr, compute_pi,
                                        (void *) &hits[i]);
    }
    for (i=0; i< num_threads; i++) {
       pthread_join(p_threads[i], NULL);
       total_hits += hits[i];
    }
    ...
}
```

SINCRONIZAÇÃO

Sincronização

- O modelo de programação fornece o formalismo necessário para controlar o paralelismo identificado numa aplicação
  - Suporte para a concorrência
  - Suporte para a sincronização
  - Suporte para a comunicação
- Num modelo de programação em memória compartilhada, a comunicação é implicitamente assegurada.
- Assim, o modelo de programação em memória compartilhada foca essencialmente
  - Expressão da concorrência
  - Mecanismos de Sincronização
  - Minimização das interações e overhead

Sincronização

- **Tipos de Sincronização**
  - Barreira
  - Lock/Semáforo
  - Operações de comunicação síncronas

Sincronização

- Tipos de Sincronização
  - **Barreira**
    - Normalmente envolve a maioria ou todas as tarefas de um programa paralelo;
    - Cada tarefa executa o seu trabalho até que a barreira é atingida. Em seguida ele para ou bloqueia;
    - Quando a última tarefa atinge a barreira, todas as tarefas estão sincronizadas. Muitas vezes neste instante, uma seção serial de código é executada. Em outros casos, as tarefas são liberadas para continuar seu trabalho.

Sincronização

- Tipos de Sincronização
  - **Semáforo**
    - Pode envolver qualquer número de tarefas;
    - Geralmente utilizado para proteger o acesso aos dados globais ou uma seção de código. Apenas uma tarefa ao mesmo tempo pode usar o bloqueio/ semáforo;
    - Outras tarefas podem tentar adquirir o bloqueio, mas devem esperar até que a tarefa que possui o bloqueio o libere;
    - Pode ser bloqueante ou não-bloqueante.

Sincronização

- Tipos de Sincronização
  - **Operações de Comunicação Síncrona**
    - Envolve somente as tarefas executando operações de comunicação;
    - Quando uma tarefa executa uma operação de comunicação, alguma forma de coordenação é necessária com outra(s) tarefa(s) participante(s). Exemplo:
      - Antes de uma tarefa poder executar uma operação de envio, deve primeiro receber uma confirmação da tarefa receptora de a recepção está em curso.

Sincronização

- Os modelos de programação em memória compaartilhada podem variar na forma de fornecer:
  - Modelo de concorrência
  - Partilha de dados
  - Suporte da sincronização

Sincronização

- Na maioria das plataformas o modelo de processos permite uma primeira aproximação ao modelo paralelo
- Geralmente com processos:
  - Não é possível partilhar dados em memória de forma simples
  - A sincronização entre processos é por vezes rudimentar
- Assim, a noção de thread, é mais adaptada às necessidades de programação paralela
  - Partilham o espaço de endereçamento (memória, código)
  - São facilmente criadas e destruídas
  - Permitem sincronização sofisticada
  - É o modelo mais adequado à programação em memória compartilhada

Sincronização

- Duas formas de realizar programação baseada em threads:
- Usar diretamente a API fornecida por uma plataforma
  - **Posix Threads**
  - Solaris Threads
  - Java Threads
  - Windows Threads
  - Etc.
- Usar um modelo de mais alto nível em que as directivas de paralelismo são expressas numa extensão da linguagem de programação
  - **OpenMP**

Sincronização

- **Tipos de Sincronização**
  - Barreira
  - Lock/Semáforo
  - Operações de comunicação síncronas

Sincronização

- Tipos de Sincronização
  - **Barreira**
    - Normalmente envolve a maioria ou todas as tarefas de um programa paralelo;
    - Cada tarefa executa o seu trabalho até que a barreira é atingida. Em seguida ele para ou bloqueia;
    - Quando a última tarefa atinge a barreira, todas as tarefas estão sincronizadas. Muitas vezes neste instante, uma seção serial de código é executada. Em outros casos, as tarefas são liberadas para continuar seu trabalho.

Sincronização

- Tipos de Sincronização
  - **Semáforo**
    - Pode envolver qualquer número de tarefas;
    - Geralmente utilizado para proteger o acesso aos dados globais ou uma seção de código. Apenas uma tarefa ao mesmo tempo pode usar o bloqueio/ semáforo;
    - Outras tarefas podem tentar adquirir o bloqueio, mas devem esperar até que a tarefa que possui o bloqueio o libere;
    - Pode ser bloqueante ou não-bloqueante.

Sincronização

- Tipos de Sincronização
  - **Operações de Comunicação Síncrona**
    - Envolve somente as tarefas executando operações de comunicação;
    - Quando uma tarefa executa uma operação de comunicação, alguma forma de coordenação é necessária com outra(s) tarefa(s) participante(s). Exemplo:
      - Antes de uma tarefa poder executar uma operação de envio, deve primeiro receber uma confirmação da tarefa receptora de a recepção está em curso.

Sincronização

- **Em exclusão mútua**
  - Quando várias threads tentam acessar ou modificar o mesmo endereço de memória, os resultados podem ser incoerentes caso não sejam tomadas as precauções para garantir uma ordem determinística

Sincronização

- Em exclusão mútua
  - Ex: Duas threads que acessam a mesma variável

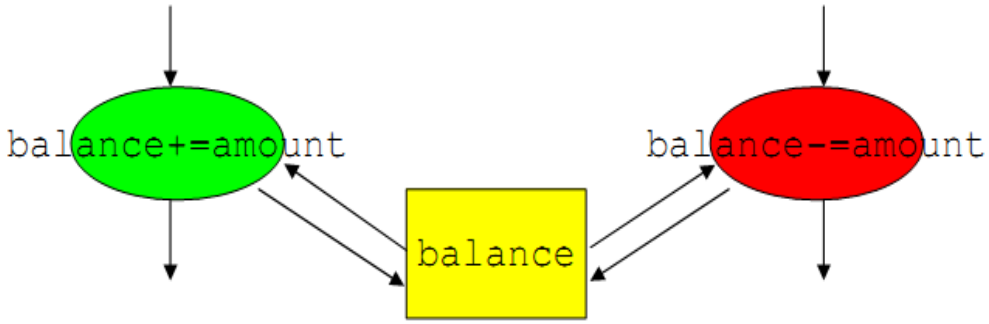
```
shared double balance;
```

$thread_1$:

```
. . .
balance = balance + amount;
```

$thread_2$:

```
. . .
balance = balance - amount;
```



Sincronização

- Se thread1 lê o valor da variável, é interrompida pelo escalonador do sistema e thread2 é executada, quando volta a correr, vai modificar o valor que a thread2 atualizou para um valor que não condiz com a modificação da thread2

Sincronização

- O código no exemplo anterior denomina-se uma seção crítica, e só pode ser executado por uma thread de cada vez
  - Tem de ser serializado
- As seções críticas constituem um problema recorrente de toda a programação paralela
  - Há inúmeros metodologias para a sua resolução

Sincronização

- Nos pthreads, as proteções das secções críticas são realizadas com recurso, como *mutex locks*.
  - Objeto de sincronização que só pode ter dois estados
    - Locked e unlocked
- Num dado instante, só uma thread é capaz de mudar o estado do mutex, tornando-se assim proprietária exclusiva do mesmo
  - Assim, se o mutex estiver associado à secção crítica, é garantido que só uma thread a poderá executar

Sincronização

- Sempre que existe uma seção crítica no código executado threads, estas devem seguir o protocolo de acesso e saída:
- Teste e aquisição do mutex em uma operação atômica

```
while (mutex <=> locked)        <-- Esta operação é indivísivel !
    thread sleep;
// mutex locked
Início secção crítica

Fim secção crítica
mutex_unlock;
// mutex unlocked

resto do código
```

Secção Crítica

Sincronização

- As pthreads fornecem múltiplas primitivas de sincronização, mas a maioria dos casos pode ser resolvido recorrendo a 3 funções:
  - Pthread_mutex_init
  - Pthread_mutex_lock
  - Pthread_mutex_unlock

Sincronização

- Assim o código pode ser escrito:

```
shared double balance;
pthread_mutex_t lock;
pthread_mutex_init(&lock, NULL);
```

$thread_1$:

```
. . .
pthread_mutex_lock(&lock);
balance = balance + amount;
pthread_mutex_unlock(&lock);
. . .
```

$thread_2$:

```
. . .
pthread_mutex_lock(&lock);
balance = balance - amount;
pthread_mutex_unlock(&lock);
. . .
```

Dependência de dados

Dependência de Dados

- Uma dependência existe entre as declarações do programa quando a ordem de execução das instruções afeta os resultados do programa;
- A dependência de dados resulta do uso da mesma localidade de armazenamento por diferentes tarefas;
- Dependências são importantes para a programação paralela porque são um dos principais inibidores de paralelismo.

Dependência de Dados

- **Loop com dependência de dados**

```
DO 500 = START, END
      A(J) = A(J-1) * 2.0
500 CONTINUE
```

- O valor A(J-1) deve ser calculado antes do valor de A(J), pois A(J) apresenta dependência de dados em A(J-1). O paralelismo é inibido;
- Se uma tarefa 2 tem A(J) e a tarefa 1 tem A(J-1), para computar o valor correto de A(J), será preciso:
  - ***Arquitetura de Memória Distribuída*** – A tarefa 2 deve obter o valor de A(J-1) da tarefa1 depois que a tarefa 1 termina a computação;
  - ***Arquitetura de Memória Compartilhada*** – A tarefa 2 deve ler A(J-1) depois da tarefa 1 atualizá-lo.

Dependência de Dados

- **Dependência de dados fora de *Loops***

Tarefa 1	Tarefa 2
X = 2	X = 4
.	.
.	.
Y = X**2	Y = X**3

Dependência de Dados

- **Dependência de dados fora de *Loops***
  - Como no exemplo anterior, o paralelismo é inibido. O valo de Y é dependente:
    - **Arquitetura de Memória Distribuída** – Se e quando o valor de X é comunicado entre as tarefas;
    - **Arquitetura de Memória Compartilhada** – Que tarefa armazena por último o valor de X.
- Apesar de todas as dependências de dados serem importantes para o projeto de programas paralelos, *loops* com dependência de dados são particularmente importantes uma vez que eles são possivelmente o alvo mais comum dos esforços de paralelização.

Dependência de Dados

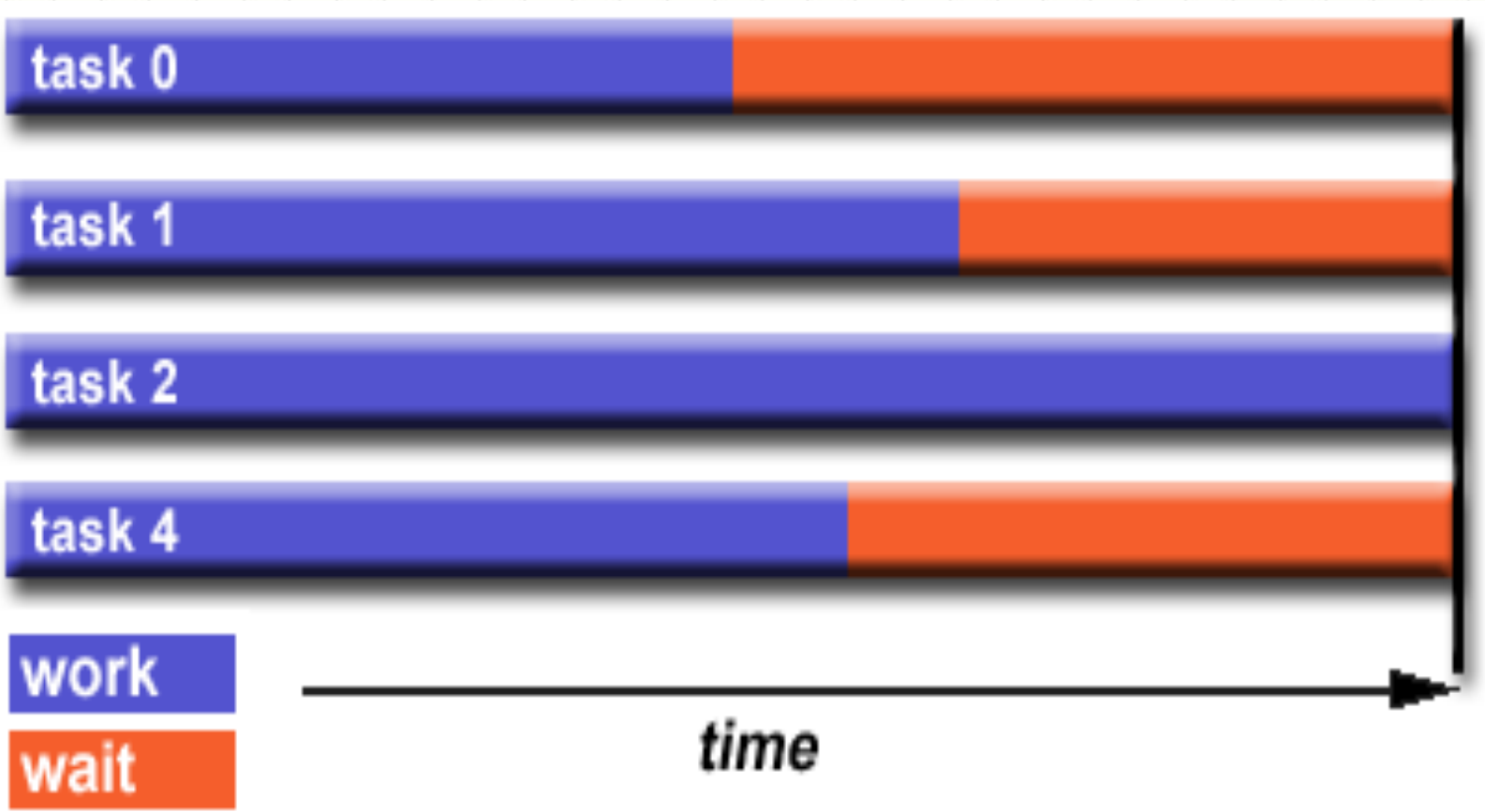
- Como tratar a dependência de dados?
  - Arquitetura de Memória Distribuída
    - Comunicar os dados necessários nos pontos de sincronização;
  - Arquitetura de Memória Compartilhada
    - Sincronizar operações de leitura/escrita entre as tarefas

Balanceamento de Carga

Balanceamento de Carga

- Prática de distribuir trabalho entre tarefas de modo que todas sejam mantidas ocupadas o tempo todo;
  - Por que é importante para programas paralelos?
- Exemplo: Se todas as tarefas estão sujeitas à uma barreira de sincronização, a tarefa mais lenta que irá determinar o desempenho global.

Balanceamento de Carga



Balanceamento de Carga

- Como conseguir balanceamento de Carga?
  - **Particionar igualmente o trabalho que cada tarefa recebe**
    - Para operações com matrizes ou vetores, onde cada tarefa realiza um trabalho semelhante, distribuir uniformemente o conjunto de dados entre as tarefas;
    - Para iterações de *loop* em que o trabalho realizado em cada iteração é similar, distribuir uniformemente as iterações entre as tarefas;
    - Se uma mistura heterogênea de máquinas com diferentes características de desempenho estão sendo utilizadas, utilizar algum tipo de ferramenta de análise de desempenho para detectar os desequilíbrios de carga.

Balanceamento de Carga

- Como conseguir Balanceamento de Carga?
  - **Utilizar atribuição dinâmica de trabalho**
    - Certas classe de problemas resultam em desequilíbrios de carga, mesmo se os dados estejam distribuídos uniformemente entre as tarefas
      - Matrizes esparsas – Algumas tarefas terão dados reais para trabalhar enquanto as outras não;
      - Métodos de Grade Adaptativa – Algumas tarefas podem precisar refinar sua malha, enquanto outras não.
    - Quanto a quantidade de trabalho de cada tarefa é intencionalmente variável ou não pode ser prevista, o uso de um escalonador pode ser útil;
    - Pode ser necessário desenvolver um algoritmo que detecta e trata os desequilíbrios de carga à medida que ocorrem dinamicamente no código.

Balanceamento de Carga

- Balanceamento X Escalonamento
  - Os conceitos de balanceamento de carga e de scheduling são muito próximos e, normalmente, usam-se com o mesmo significado.
- Objetivos
  - Maximizar o desempenho de sistema paralelo, transferindo tarefas de processadores mais sobrecarregados para outros que estejam mais leves

Balanceamento de Carga

- A estratégia de escalonamento envolve:
  - Determinar quais tarefas são executadas em paralelo
  - Em que local elas serão executadas

A decisão pode ser tomada em tempo de execução ou com base em um conhecimento prévio.

Balanceamento de Carga

- A estratégia de escalonamento depende das propridades das tarefas
  - Custos das tarefas (e.g. quantidade de ciclos)
  - Dependência entre as tarefas
  - Localidade (e.g. local e/ou remoto)

Balanceamento de Carga

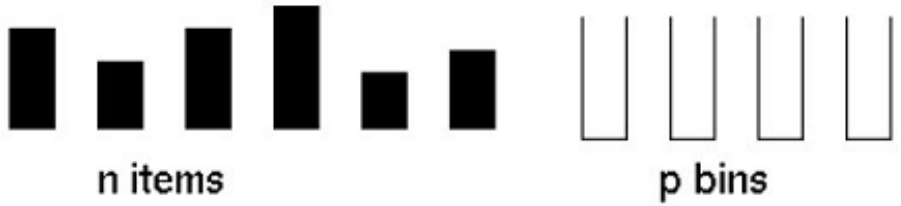
- **Custos das Tarefas**
  - Elas têm o mesmo custo
  - Se não, quando esses custos são conhecidos?
    - Antes da execução, quando a tarefa é criada ou apenas quando termina?
- **Dependências**
  - Elas podem executar em qualquer ordem?
  - Se não, quando são conhecidas as dependências?
    - Antes da execução, quando a tarefa é criada ou apenas quando termina?
- **Localidade**
  - É importante que algumas tarefas executem num mesmo processador (ou próximo deste) para reduzir a comunicação?
  - Quando se conhece a informação sobre comunicação?

Balanceamento de Carga

Escalonamento de um conjunto de tarefas

Easy: The tasks all have equal (unit) cost. branch–free loops

Harder: The tasks have different, but known, times. sparse matrix–vector multiply



Hardest: The task costs unknown until after execution. GCM, circuits, search

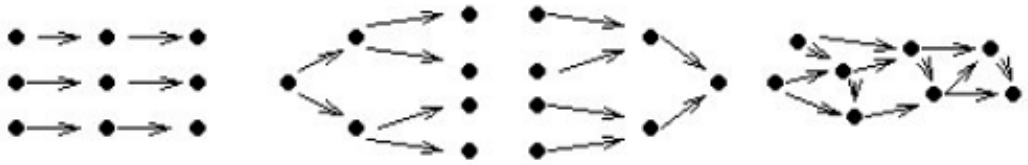
Balanceamento de Carga

Escalonamento de um grafo tarefas

Easy: The tasks can execute in any order.

dependence free loops

Harder: The tasks have a predictable structure.



matrix computations (dense, and some sparse, Cholesky)

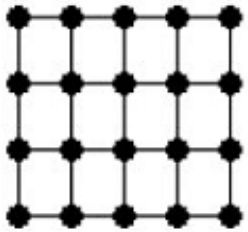
Hardest: The structure changes dynamically (slowly or quickly) search, sparse LU

Balanceamento de Carga

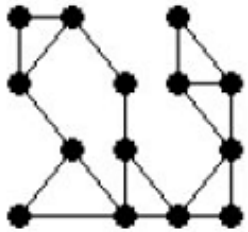
Escalonamento de um conjunto de tarefas

Easy: The tasks, once created, do not communicate. — embarrassingly parallel

Harder: The tasks communicate in a predictable pattern. — PDE solver



regular



irregular

Hardest: The communication pattern is unpredictable. — discrete event simulation

Balanceamento de Carga

- O escalonamento pode ser:
  - Estático
    - Decisões tomadas em tempo de compilação
  - Dinâmico
    - Usa informações do estado da computação em tempo de execução, para tomar decisões

Balanceamento de Carga

Por que escalonamento dinâmico?

Balanceamento de Carga

- Estratégias de Escalonamento Dinâmico
  - Centralizadas
  - Distribuídas

Balanceamento de Carga

- **As decisões de partilha podem ser:**
  - Sender-initiated (ou work distribution)
  - Receiver-initiated (ou work stealing)
- **Como selecionar um worker para realizar as computações?**
  - Round-Robin
  - Polling/Stealing Aleatório
  - Repete o Último

Balanceamento de Carga

- **Estratégias de Partilha**
  - Sender-initiated
  - Receiver-initiated
  - Adaptative I
  - Adaptative II

GRANULARIDADE

Granularidade

- Medida qualitativa da relação de computação para comunicação;
- Períodos de cálculo são normalmente separados por períodos de comunicação ou por eventos de sincronização.
  - I/O Bound
  - CPU Bound

Granularidade

- **Paralelismo de grão fino**
  - Pequenas quantidades de trabalho computacional são realizadas durante comunicação entre processos;
  - Baixa taxa de computação para a comunicação;
  - Facilita o balanceamento de carga;
  - Se a granularidade é muito fina, é possível que a sobrecarga de comunicação e sincronização entre tarefas demore mais do que a computação.
    - Para realizar o *context switch*, por exemplo.

Granularidade

- Paralelismo de grão grosso
  - Grandes quantidades de trabalho computacional são realizados entre os eventos de comunicação/ sincronização;
  - Alta taxa de computação para a comunicação;
  - Mais difícil de balancear a carga eficientemente.

Granularidade

- **Grão Fino x Grão Grosso**
  - A granularidade mais eficiente depende do algoritmo e do ambiente de hardware no qual ele é executado;
  - Na maioria dos casos a sobrecarga associada à comunicação e sincronização é elevada em relação à velocidade de execução. Nessa caso é vantajoso ter granularidade grossa;
  - Paralelismo de grão fino pode ajudar a reduzir sobrecargas devido ao desequilíbrio das cargas.

Importante

- Operações de E/S são consideradas inibidoras do paralelismo;
- Sistemas de E/S paralelos podem não estar disponíveis para todas as plataformas (e.g. DMA);
- Em um ambiente onde todas as tarefas enxergam o mesmo espaço de arquivo, operações de gravação podem resultar na sobrescrita do arquivo;
- Operações de leitura podem ser afetadas
  - Servidor não possui capacidade para lidar com múltiplas operações de leitura ao mesmo tempo.
- E/S que deve ser conduzida por meio da rede de comunicação pode ocasionar sobrecarga.

Importante

- Sistemas de Arquivos Paralelos disponíveis
  - GPFS (General Parallel Filesystem) → IBM AIX
  - Lustre: Para Cluster Linux (SUN)
  - PVFS (Parallel Virtual Filesystem for Linux Clusters)
  - PanFS: Panasas ActiveScale File System for Linux Clusters
  - HP SFS (HP StorageWorks Scalable File Share)

EXERCÍCIO E LEITURA RECOMENDADA

Exercício

- Acessar o Moodle

Leitura Recomendada

- Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar - 2ª ed., Addison Wesley

Bibliografia

- Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar - 2ª ed., Addison Wesley
- Introduction to Parallel Computing
  - https://computing.llnl.gov/tutorials/parallel_comp/
- Programação Paralela e Distribuída
  - http://www.dcc.fc.up.pt/~ricroc/aulas/1011/ppd/apontamentos/load_balancing.pdf
- NetLab – Computação Paralela
  - http://netlab.ulusofona.pt/cp/

Dúvidas

Próxima Aula...

- Avaliação de Desempenho de Programas Paralelos