Gerando códigos concorrentes

Técnicas básicas de projeto de códigos concorrente considerando o paradigma de memória compartilhada

Quatro passos da paralelização

- Análise da versão sequencial
- Projeto e implementação
- Testes de correção
- Análise de desempenho

Análise da versão sequencial

1. Identificar pontos onde se possa implementar concorrência.
2. Definir tipo de decomposição de domínio.
3. Encontrar *hot spots*.
    - Normalmente laços.

(1) Análise de dependências

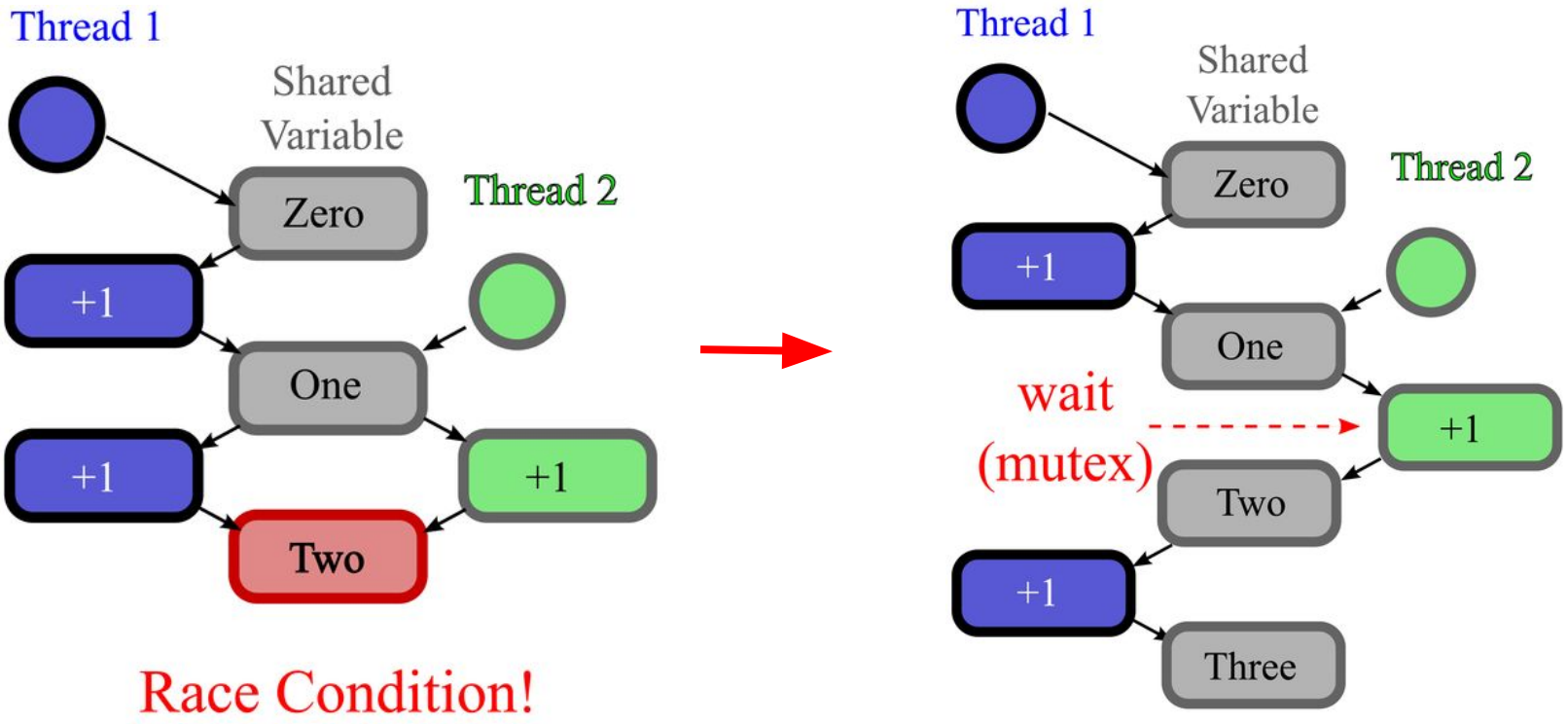
```
for(i=0; i<N; i++) {
    X[i] = Y[i] + Z[i];
    A[i] = Y[i] + delta;

}
```

```
SUM = 0;
for(i=0; i<N; i++) {
    X[i] = Y[i] + Z[i];
    A[i] = X[i] + delta;
    SUM += A[i];
}
```

Algoritmo 2.1 - Exemplo de laços sem e com dependência de dados

- X[i] aparece à esquerda e à direita da atribuição (escrita e leitura).
- SUM sofre operações de leitura e escrita (+=).

(1) Análise de dependências: condições de corrida



http://www.techenablement.com/intel-xeon-phi-optimization-part-1-of-3-multi-threading-and-parallel-reduction/

(2) Decomposição de domínio

- Decomposição funcional
  - Dividir o trabalho entre tipos de tarefas (ou funções) diferentes
- Decomposição de domínio
  - Dividir os dados entre tarefas que executam o mesmo código

(3) Detecção de *hot spots* para melhoria de desempenho por concorrência

- Detecção de trechos mais demorados (“gargalos”)
- Laços são bons candidatos
  - Tarefas repetitivas sobre um conjunto de dados homogêneos (decomposição de domínio)
- Escolhe-se os laços candidatos e efetua-se a medição de tempo.

(3) Detecção de *hot spots*

```
#include <stdio.h>
#include <sys/time.h>
#define N 1000000

int main(void) {
   int k;
   double p = 1, x;
   struct timeval inicio, final;
   long long tmili;

   gettimeofday(&inicio, NULL);
   x = 1.0 + 1.0/N;
   for(k=0; k<N; k++)
      p = p*x;
   gettimeofday(&final, NULL);

   tmili = (int) (1000*(final.tv_sec - inicio.tv_sec) +
           (final.tv_usec - inicio.tv_usec) / 1000);

   printf("PI aproximado: %g\n", p);
   printf("tempo decorrido: %lld ms\n",tmili);
   return 0; }
```

Algoritmo 2.2 - Exemplo de código-fonte com instrumentação para medida de tempo de execução de um trecho específico.

Projeto e implementação

1. Observar plataforma de hardware disponível (arquitetura paralela)
2. Considerar uso de padrões conhecidos de programação concorrente/paralela
3. Realizar testes de correção
4. Realizar uma análise de desempenho

(1) Arquitetura paralela disponível

- Observar plataforma de hardware disponível e definir paradigmas de programação:
  - Memória compartilhada
  - Aceleradores
  - Memória distribuída

Padrões de programação

Padrões de código são soluções genéricas e reutilizáveis para problemas comuns em determinados contextos.

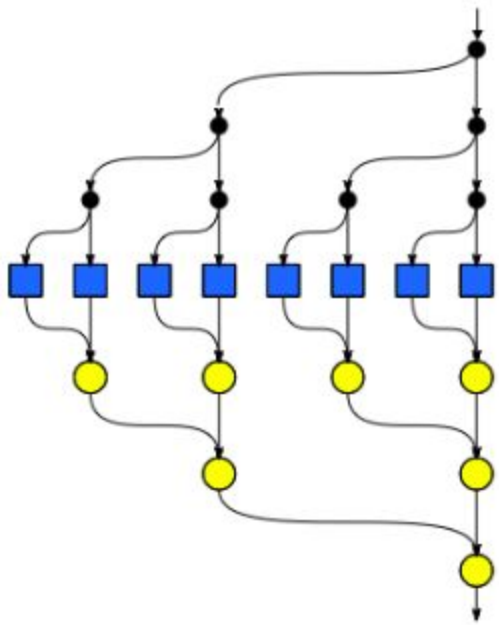
(2) Uso de padrões

- O projeto de código paralelo pode usar algum padrão de código conhecido para este fim
- Exemplo: McCool et al (2012).

(2.1) Padrão fork-join

A operação ***fork*** permite que o fluxo de controle seja dividido em múltiplos fluxos de execução paralelos

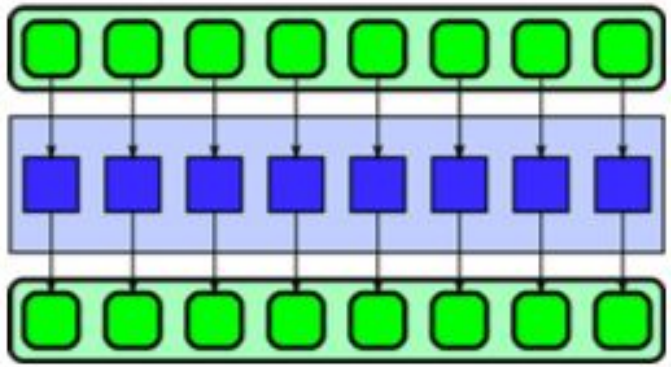
Na operação ***join*** estes fluxos se reunirão novamente no final de suas execuções, quando apenas um deles continuará.



(2.2) Padrão map

Replica uma mesma operação sobre um conjunto de elementos indexados.

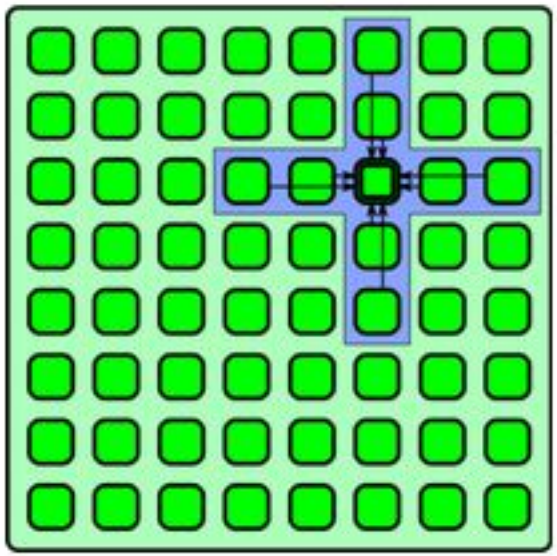
Este padrão se aplica à paralelização de laços, nos casos onde se pode aplicar uma função independente a todo o conjunto de elementos.



(2.3) Padrão stencil

É uma generalização do padrão map, onde a função é aplicada sobre um conjunto de vizinhos.

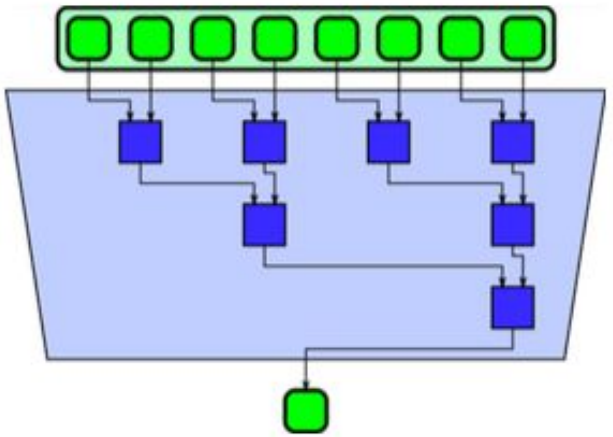
Os vizinhos são definidos a partir de um conjunto *offset* relativo a cada ponto do conjunto.



(2.4) Padrão reduction

Combina todos os elementos de uma coleção em um único elemento a partir de uma função combinadora associativa.

Uma operação muito comum em aplicações numéricas é realizar um somatório ou encontrar o máximo de um conjunto de elementos.



Testes de correção

Atenção a aspectos da execução paralela que se diferencia da sequencial e podem ocasionar erros ou inconsistências de execução.

(3) Testes de correção

- Código *multithreading* é altamente sujeito a erros e não-determinismo
- Atenção a:
  - Erros causados pela alteração concorrente de dados compartilhados,
    - Ex: condições de corrida (*race conditions*).
  - Mau-uso dos mecanismos de controle de acesso à seções críticas (por exemplo, *mutex*)
  - Erros de sincronização entre *threads*.

(3) Testes de correção

- Existem ferramentas de depuração disponíveis
  - Maioria não é gratuita
  - Uso requer certo treinamento
- Teste básico
  - Comparar saída da versão paralela com saída de uma versão sequencial correta
  - Problema: não determinismo (executar várias vezes…)
- Recomenda-se testes sistemáticos principalmente em problemas críticos.