Programação Paralela: das *threads* aos FPGAs

Programação Paralela com OpenMP

Prof. Ricardo Menotti
menotti@ufscar.br

Prof. Maurício Acconcia Dias
macccdias@gmail.com

Prof. Hélio Crestana Guardia
helio.guardia@ufscar.br

Departamento de Computação
Universidade Federal de São Carlos

Atualizado em: 1 de junho de 2020

Computação de Alto Desempenho

O que é: uso de (múltiplos) recursos computacionais, maximizando suas capacidades.

Pra quê?

- Resolver problemas computacionais **mais rapidamente**.
- Resolver novos problemas computacionais, anteriormente intratáveis:
  - Demoravam demais
  - Envolviam manipulação de conjuntos de dados não tratáveis (ou não tratáveis em tempo viável).

Como medir "alto desempenho"?

- Tempo de resposta: *speedup* (Ts/Tp)
- Eficiência: *spedup* / processador
- Eficiência energética: Mflops / Watt

Computação de Alto Desempenho

Técnicas:

- Replicação de elementos funcionais no processador: pipelines, unidades vetoriais, ULAs, ...
- Criação de unidades de processamento especializado
- Replicação de processadores:
  - Múltiplos processadores no modelo SIMD ou SIMT (GPUs)
  - **Múltiplas CPUs (ou *cores*): MIMD**

Processamento paralelo com multiprocessadores

[*Sem virtualização ou containers, sob um único SO*]

Atividades "escalonáveis":

- Processos:
  - Espaços de endereçamento separado
  - Comunicação via mecanismos de IPC (pode incluir SHM)
- ***Threads***:
  - Compartilham espaço de endereçamento do processo
  - Comunicação usando **memória compartilhada**

Programação com *Threads*

- Suporte em múltiplas plataformas e linguagens, com diferentes abstrações
- No nível mais baixo, provido pelo SO, há primitivas para criar *threads*
- API tradicional: Posix Threads (*pthreads*)
    - Programa **precisa ser paralelizado**
        - **Decomposição funcional**: quais funções são independentes e podem virar *threads*
        - **Decomposição de dados**: como **dividir estrutura de dados** para que diferentes partes (linhas, colunas, …) possam ser manipuladas por múltiplas instâncias (*threads*) de uma função

```
void * produz (void *arg)
{
  ...
}
...
void * mult_linha (void *arg)
{
  ...
}
...
```

```
...
pthread_t prod, cons, vet_tid[N];
...
pthread_create(&prod, NULL, produz, (void *)vezes);
...
for(int i=0; i < num_linhas; i++ )
   pthread_create(&vet_tid[i], NULL, mult_linha, (void *)i);
...
pthread_join(prod, (void **)&status);
...
```

Programação com OpenMP

- **OpenMP: especificações** padronizadas (extensões de linguagem C/C++ e Fortran) para paralelização de código.
  - Depende do compilador ter suporte para esses recursos!
- **Objetivo**: paralelizar o programa automaticamente, a partir de **dicas** do **programador** sobre quais trechos podem ser paralelizados.
- **Mecanismos**: extensões de linguagem
  - Pragmas inseridas no código
  - Funções de uma API
  - Variáveis de ambiente

Programação com OpenMP

Modelo de programação:

- Criação de **time de *threads*** para executar um bloco de código paralelo.
- Time de *threads* pode **replicar** o código do bloco paralelo, ou **dividi-lo** entre as *threads*.
- *Threads* também podem ser criadas dinamicamente.
- Também é possível definir como variáveis do programa serão **compartilhadas** ou **replicadas** entre as *threads*.
- É possível executar *loops* usando operações SIMD do processador (*não tratado neste curso*).
- Usar dispositivos, como GPUs, para processamento SIMT (*não tratado neste curso*).

Como é feita a paralelização?

- Programador **insere marcas no código** que **indicam** explicitamente ao compilador quais regiões podem ser **paralelizadas**.
- Esforço de paralelização de um programa com OpenMP resume-se, em geral, à **identificação do paralelismo** e não à **reprogramação** do código para implementar o paralelismo desejado.

Programação com OpenMP

Como usar?

- **Diretivas**: #pragma omp ...
- **Funções da API**: #include <omp.h> … omp_set…(); … omp_get…()
- **Variáveis de ambiente**: export OMP_NUM_THREADS=...

Como compilar?

$ gcc prog.c [-o prog] [-Wall] -fopenmp

$ man gcc

*/**-fopenmp***

*Enable handling of OpenMP directives "#pragma omp" in C/C++ and "!$omp" in Fortran. When **-fopenmp** is specified, the compiler generates parallel code according to the OpenMP Application Program Interface v4.5 <https://www.openmp.org >. This option implies **-pthread**, and thus is only supported on targets that have support for **-pthread**.*

Programação com OpenMP

[*Como é que se brinca disso?*]

Ex: coliru

```
...
int main ()
{
  ...   // Código serial, executado por apenas 1 thread, como usual
  ...
  // Uso da diretiva parallel para criar uma região paralela:
  #pragma omp parallel num_threads(4)
  {
    // Seção paralela, executada por todas as threads do time
    printf("Hello, world!\n");
  }
  // Ao fim do bloco de código da região paralela, a thread master espera pela conclusão das demais
  // Apenas thread master (aquela que encontrou a região paralela e criou o time) prossegue execução
  printf("Goodbye\n");
  ...
}
```

Programação com OpenMP

Ex: coliru

```
#include <omp.h>   // necessário apenas se formos usar funções da API omp
...
int main ()
{
  ...
  // Início de seção paralela: geração das threads do time
  #pragma omp parallel
  {
   ...
    // Seção paralela, executada por todas as threads do time
    printf("Esta é a thread %d de um time de %d.\n", omp_get_thread_num(), omp_get_num_threads());
   ...
  }
  // Apenas master thread prossegue execução após o bloco paralelo
  ...
}
```

Programação com OpenMP

Sobre o uso de diretivas (via *pragma*) e a diretiva *parallel*.

#pragma omp parallel [clause[[,]clause]...] new-line
  structured-block

Cláusulas (*clauses*) incluem:

- if (scalar-expression)
- private (variable-list)
- firstprivate (variable-list)
- default (shared | none)
- copyin (variable-list)
- reduction (operator: variable-list)
- num_threads (integer-expression)
- proc_bind (master | close | spread)
- allocate ([allocator :] list)

Programação com OpenMP

Paralelismo com divisão de trabalho: ***work-sharing***

Além de permitir a **replicação** da execução de trechos de código, OpenMP possui construções para **dividir** as execuções desses trechos.

Construções do tipo *work-sharing* não geram novas *threads*, mas **se aplicam às *threads* do time** associado à região paralela atual.

- ***for*** : dividem as iterações de um comando *for* entre as *threads* do time. Representam o paralelismo de dados.
- ***sections*** : dividem o trabalho em regiões explicitamente definidas. Cada seção é executada por uma *thread*. Pode ser usada para representar o paralelismo funcional.
- ***single*** : serializa um trecho de código, que é executado por apenas 1 das *threads* do time.

Programação com OpenMP: *work-sharing*

Single

```
int main()
{
   int a;
   ...
   #pragma omp parallel shared(a)
  {
     ...                              // Todas as threads do time executam essa parte
     #pragma omp single      // Só uma thread do time vai executar esse bloco
     {
       a=(int)random();
     }                  // Uma barreira é usada aqui, se a cláusula nowait não for especificada. Só quando a última thread do time chegar aqui,
                        //     normalmente a que executou o código do single, é que todas prosseguem na execução do resto do código paralelo.
     ...      // Todas as threads do time executam essa parte
   }          // fim da região paralela
   ...                  // Só a master thread prossegue
}
```

Programação com OpenMP: *work-sharing*

For: dividindo as iterações de um loop

Diretiva ***for*** especifica que as iterações do *loop* abaixo devem ser executadas em paralelo, **dividindo-as** entre *threads* do time.

```
#pragma omp for [clause ...] newline
  for (...; ... ; … ) { }
```

For canônico: quantas iterações? Quais?

```
for ( index = start;
      index {<,<=,>=,>} end;
     { indx++, ++indx, indx--, --indx, indx+=inc, index -= inc, indx= indx+inc, indx=inc+indx, indx=indx-inc} )
```

Programação com OpenMP: *work-sharing*

For: dividindo as iterações de um *loop*

#pragma omp **for** [clause ...] newline

Cláusulas:

schedule (type [,chunk]) , ordered, private (list), first private (list), last private (list), shared (list), reduction (operator: list), collapse (n), **nowait**

- ***schedule***: determina como as iterações do loop são divididas entre *threads* do time.
  - ***static***: iterações divididas em bloco de tamanho *chunk*.
  - ***dynamic***: iterações divididas em blocos de tamanho *chunk* e alocadas dinamicamente entre as *threads*, à medida que terminam as iterações atribuídas anteriormente.
  - ***guided***: número de iterações atribuído em cada rodada é calculado em função das iterações restantes divididas pelo número de *threads*, sendo o resultado decrescido de *chunk*.
  - ***runtime***: atribuição é realizada em tempo de execução, de acordo com variável de ambiente OMP_SCHEDULE.
- ***nowait***: se usada, indica que *threads* não devem ser sincronizadas no fim do *loop* paralelo.
- ***ordered***: indica que as iterações do *loop* devem ser executadas em sequência como se executadas por programa serial.

Programação com OpenMP: for

[Para testar com coliru]

```
int i, id, num_it, vet[MAX];
...
#pragma omp parallel private(id)  // Não precisa declarar i como privada; isso é feito automaticamente pelo programador
{
  // Todas as threads do time executam esse trecho do bloco de código de maneira replicada
  id = omp_get_thread_num();
  ...
  // Construção for divide as iterações entre as threads do time
  // Variável de controle do for (i) é feita privada para cada thread, automaticamente pelo compilador!
  // Diretiva for deve aparecer dentro de uma região paralela
  #pragma omp for        // O único comando permitido na linha abaixo da diretiva for é um for :-)
  for (i=0; i < num_it; i++) {
     printf("Thread %d tratando iteração %d\n", id, i);
     vet[i] = 2 * i;
  }
  // todas as threads replicam esse trecho de código, fora do for, mas dentro da região paralela
 ...
} // fim da região paralela
```

Programação com OpenMP: for

Forma compacta de declaração do *paralell for*

Quando o paralelismo desejado no programa é apenas para divisão das iterações de um comando *for*, é possível usar a declaração compacta da diretiva ***for***:

```
...
 #pragma omp parallel for     // O único comando permitido nesta forma de declaração paralela é o for
 for (i=0; i < NUM; i++) {
   // printf("Thread %d tratando iteração %d\n", omp_get_thread_num(), i);
   ...
 }
...
```

Programação com OpenMP: for

Controlando a divisão das iterações

Por padrão, divisão das iterações do for paralelo é feita em bloco: cada *thread* será encarregada de 1 / N das iterações.

Vale lembrar que cada *thread* num time tem um **número lógico** que vai de 0 (para a *thread master*) a N-1.

Assim, *thread **n*** vai executar iterações n * (1/N) .. (n+1) * (1/N) -1

O compilador está atento, contudo, e consegue tratar os casos em que essa divisão não é exata! Neste caso, as "**resto da divisão inteira**" primeiras *threads* recebem uma iteração a mais cada uma.

O controle do particionamento pode ser feito de três formas:

- **Cláusula *schedule*** na primitiva *for* ;
- Usando a função omp_set_schedule(omp_sched_tkind, intchunk_size); e
- Via variável de ambiente OMP_SCHEDULE

[Para testar com coliru]

Programação com OpenMP: *work-sharing*

Sections: dividindo trechos de código

- Usada dentro de região paralela, diretiva ***sections*** permite especificar seções de código que devem ser divididas entre as *threads* do time.
- Cada ***section*** é executada apenas uma vez, por qualquer uma das *threads* no time.
- Pode haver *threads* executando mais de uma seções, e é possível que alguma *thread* não tenha seção para executar.
- Barreira é inserida automaticamente ao final das *sections*, exceto se a cláusula *nowait* for especificada.

```
#pragma omg sections [clause ...]  newline
{
 [#pragma omp section newline]
   bloco_de_código
 [#pragma omp section newline]
   bloco_de_código
 ...
}
```

Programação com OpenMP: *sections*

```
...
#pragma omp parallel private(... )
{
   ... // Todas as threads do time executam esse trecho
   #pragma omp sections
   {
     #pragma omp section
     {
       faça_isso();
     }
     #pragma omp section
     {
       faça_aquilo();
     }
     #pragma omp section
     {
       faça_ainda_outra_coisa();
     }
   }   // Fim das seções
   ... // Todas as threads do time executam esse trecho
}    // Fim da região paralela
```

Programação com OpenMP

Tasks: criando tarefas sob demanda

O modelo tradicional de programação com *threads* em **OpenMP** trata da criação de **regiões paralelas**, executadas por times de *threads*, e da eventual divisão de trabalho (*worksharing*) da região paralela entre as *threads* do time.

Nesses casos, observa-se que:

- O número de *threads* de uma região é pré-definido na criação dessa região paralela, e pode ser consultado pelas *threads* com a chamada *omp_get_num_threads()*;
- Cada *thread* pode saber qual é seu identificador lógico, obtido com a chamada *omp_get_thread_num()*;
- Programador preocupa-se principalmente com a **divisão de carga** entre as tarefas:
    - Programador pode definir como será a divisão de iterações de um *loop* (*schedule*);
    - Programador pode tomar decisões sobre o que executar em cada *thread*, em função do número lógico de cada uma no time.

Além de permitir a criação de tarefas implícitas, associadas às regiões paralelas, OpenMP permite a **criação de tarefas sob demanda**, dinamicamente, de maneira circunstancial ou recursiva, sem saber a priori quantas tarefas serão necessárias. Isso é feito com a diretiva ***task***.

Na programação com tarefas (*tasks*), programador concentra-se em **como particionar o código** em blocos, que podem ser executados em paralelo, ou seja, em quais trechos de código podem ser transformados em tarefas independentes.

Nesse modelo, cabe ao sistema em tempo de execução determinar como serão o escalonamento e a execução das tarefas criadas.

Programação com OpenMP: tasks

task construct

A diretiva ***task*** é usada **dentro de uma região paralela** e define uma tarefa específica.

Essa tarefa pode ser executada pela *thread* que encontrar essa diretiva, ou deferida para execução por qualquer outra *thread* no time de *threads* corrente.

Quando uma *task* é criada, se houver alguma *thread* ociosa no time da região paralela atual, a *task* pode ser executada imediatamente. Caso contrário, fica a critério do sistema em tempo de execução de OpenMP determinar quando esta *task* será executada.

#pragma omp task [clause ...] newline
structured_block

clauses: if (scalar expression), untied, default (shared | none), private (list), firstprivate (list), shared (list), final (scalar-expression), mergeable, depend(dependence-type : list), priority(priority-value)

Programação com OpenMP: tasks

```
int calc(int start, int finish)
{
   int i, dif;   int sum = 0, sum1, sum2;

   if (finish-start <= MIN_BLK) {    //calcula
      for (i=start; i < finish; i++)
         sum++;
   } else{                    // divide, criando novas tasks
      dif = finish - start;

     #pragma omp task shared(sum1)
     sum1 = calc (start, start + dif / 2);

     #pragma omp task shared(sum2)
     sum2 = calc (finish - dif / 2, finish);

     #pragma omp taskwait     // espera tasks terminarem
     sum = sum1 + sum2;
   }
   return sum;
}
```

```
int main()
{
  int sum;

  #pragma omp parallel     // cria um time de threads
  {
   // Diretiva single faz com que apenas uma thread crie as tasks
   // As demais threads do time vão ficar à disposição para execução das tasks
   #pragma omp single
     sum = calc (0, NUM);
  }

  printf("Soma: %d\n", sum);

  return 0;
}
```

[Para testar no coliru]

Programação com OpenMP: variáveis

```
int global;
...

void f()
{
  int val;     // local, alocada na pilha de cada  thread que chamar esta função...
  ...
}

 int main()
 {
   int i, num, sum=0;
   ...
   #pragma omp parallel  private( ... )
   {
     ...
     if (global > ...) ;    // acesso de  leitura à variável global
     ...
     num = ...;             // acesso de  escrita na variável num, compartilhada pelas  threads do time
     ...
     #pragma omp for  private( ...  )
     for ( i = ...; i < global ; ... ) {    //  leitura de global,  alteração de i: compartilhada?
       ...
       sum = sum + ...;     // alteração de sum. Qual é o  valor inicial das cópias? O que fazer com cópias ao fim do loop?
     }
   }
   ...
   printf("Soma: %d\n", sum);
 }
```

Programação com OpenMP: variáveis

- **Visibilidade das variáveis:** quais variáveis são visíveis pelas threads?
- **Tipo de acesso às variáveis compartilhadas:** leitura e escrita X compartilhadas e privadas (***private***(), ***shared***())
- **Exclusão mútua e operação atômica:** como prover exclusão mútua a variáveis compartilhadas? (***critical***, ***atomic***)
- **Variáveis privadas:** atribuição inicial e propagação de valores (***firstprivate***, ***copyin***)
- **Variáveis privadas:** retorno do valor da variável associada à última iteração (***lastprivate***, ***reduction***)

Programação com OpenMP: variáveis de ambiente

Variáveis de ambiente permitem alterar aspectos da execução de aplicações OpenMP, sem que seja preciso recompilar o código!

- OMP_NUM_THREADS: determina o número máximo de *threads* para uso durante a execução.
- OMP_SCHEDULE: usada na diretiva parallel for, determina como as iterações do loop são escalonadas aos processadores.
- ...
- OMP_NESTED: habilita o uso de paralelismo aninhado.
- OMP_STACKSIZE: controla o tamanho da pilha para threads criadas (non-Master).
- OMP_MAX_ACTIVE_LEVELS: controla o número máximo de regiões paralelas ativas aninhadas.
- OMP_THREAD_LIMIT: ajusta o número de *threads* a serem usadas ao todo no programa OpenMP.
- ...

Programação com OpenMP: conclusões (1/2)

- **Modelo de programação**:
  - Criação de **time de *threads*** para executar um bloco de código paralelo.
  - Time de *threads* pode **replicar** o código do bloco paralelo, ou **dividi-lo** entre as *threads*.
  - *Threads* também podem ser criadas dinamicamente.
  - Também é possível **compartilhar** ou **replicar** variáveis entre as *threads*.
- **Como é feita a paralelização?**
  - Programador, em geral, **insere marcas no código** que **indicam** explicitamente ao compilador quais regiões podem ser **paralelizadas**.
  - Esforço de paralelização de um programa com OpenMP resume-se, em geral, à **identificação do paralelismo** e não à **reprogramação** do código para implementar o paralelismo desejado.

Programação com OpenMP: conclusões (2/2)

- **Paralelização sempre vale a pena?**
  - Quais são as sobrecargas com o paralelismo?
  - Lembrar da Lei de Amdahl: trecho sequencial limita potencial máximo de ganho com a paralelização.
- **Qual é o grau de paralelismo adequado?**
  - Varia em função do número de processadores
  - Varia em função dos tamanhos dos blocos de código paralelos

PS: pelo menos fica mais fácil paralelizar e experimentar com OpenMP :-)