Paralelização de códigos, OpenMP, MPI e GPUs

2 – OpenMP, MPI, GPUs

Paulo Penteado
Northern Arizona University

pp.penteado@gmail.com

Esta apresentação: http://www.ppenteado.net/ast/pp_para_iiwcca_2.pdf
Arquivos do curso: http://www.ppenteado.net/ast/pp_para_iiwcca/

Programa

2 – OpenMP, MPI, GPUs

- OpenMP
  - Características
  - Diretrizes
  - Estruturação
  - Construções
  - Sincronização
- MPI
  - Características
  - Estruturação
  - Formas de comunicação
  - Principais funções
  - Sincronização
- GPUs
  - GPUs x CPUs
  - CUDA
  - OpenCL
  - OpenACC
  - Estruturação
- Algumas referências

Slides em http://www.ppenteado.net/ast/pp_para_iiwcca_2.pdf
Exemplos em http://www.ppenteado.net/ast/pp_para_iiwcca/

pp.penteado@gmail.com

OpenMP - Open Multi-Processing (OMP)

É o padrão melhor estabelecido para programação em paralelo com **memória compartilhada (múltiplos *threads*).**

Antes do OpenMP:

- Não havia um padrão: quem escrevia cada compilador inventava sua forma de criar e usar *threads*.
- Para mudar de linguagem, sistema operacional, computador, ou simplesmente compilador, era necessário aprender uma nova forma, e reimplementar o código de acordo.

OpenMP:

- Engloba uma biblioteca que provê a funcionalidade, e uma API (*Application Programming Interface*) que especifica a forma de a usar.
- **Independente de linguagem, compilador, sistema operacional e hardware.**
- Padrão aberto, e em contínua evolução, decidido pelo *Architecture Review Board* (ARB), que inclui os principais desenvolvedores de hardware e software.
- 1.0 em 1997, 2.0 em 2002, 3.0 em 2008, 3.1 previsto para 2011.

OpenMP - Características

Paralelismo apenas por memória compartilhada:

- **Cada unidade de execução é um *thread*.**
- Todos os *threads* têm acesso a uma mesma memória (global, pública).
- Cada *thread* tem também sua memória independente dos outros (local, privada), para seu uso interno.
- Mais simples de programar que com memória distribuída (ex: MPI).

Com mais freqüência é usado para paralelismo de dados (e tem maior variedade de ferramentas para isso), mas também faz paralelismo de tarefas.

Tradicionalmente suportado por compiladores de C, C++ e Fortran:

- A funcionalidade e a semântica (significado das construções) são as mesmas, dentro dos limites de cada linguagem.
- A sintaxe (forma de expressar) é semelhante (não é idêntica por as linguagens terem sintaxes diferentes).

Não é adequado para paralelização extrínseca.

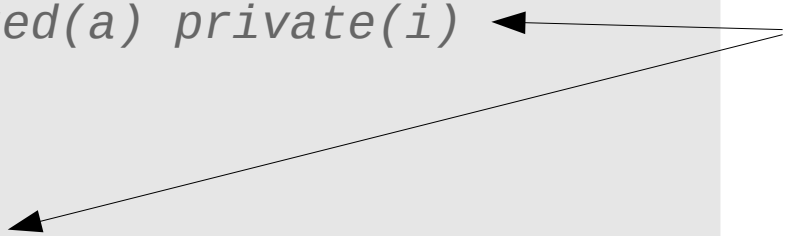
OpenMP - Características

Implementado através de **diretrizes de compilação** (instruções que serão executadas pelo compilador, não pelo programa produzido):

- **Diretrizes são vistas como comentários** por compiladores que não conheçam OMP (ou quando OMP é desabilitado na hora de compilar).
- Mantém o código funcional como programa serial.
- A paralelização de um código serial pode ser feita gradualmente: **não é necessário reestruturar o programa apenas para incluir os usos de OMP** (mas pode ser necessário reestruturar para o tornar paralelizável).

Ex (Fortran):

```
program pp_omp_ex1
!Generic example of the look of an OMP program
implicit none
integer, parameter :: n=1000
integer :: i,a(n)
!$omp parallel do shared(a) private(i)
do i=1,n
  a(i)=i
enddo
!$omp end parallel do
end program pp_omp_ex1
```



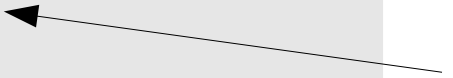
OpenMP - Características

Implementado através de **diretrizes de compilação** (instruções que serão executadas pelo compilador, não pelo programa produzido):

- **Diretrizes são vistas como comentários** por compiladores que não conheçam OMP (ou quando OMP é desabilitado na hora de compilar).
- Mantém o código funcional como programa serial.
- A paralelização de um código serial pode ser feita gradualmente: **não é necessário reestruturar o programa apenas para incluir os usos de OMP** (mas pode ser necessário reestruturar para o tornar paralelizável).

Ex (C/C++):

```
int main(int argc, char *argv[]) {
/*Generic example of the look of an OMP program*/
int n=1000;
int a[n];
#pragma omp parallel for shared(a,n)
for (int i=0; i<n; i++) {
  a[i]=i;
}
return 0;
}
```



OpenMP - Características

Estrutura típica de um programa OMP (Chapman et al. 2007):

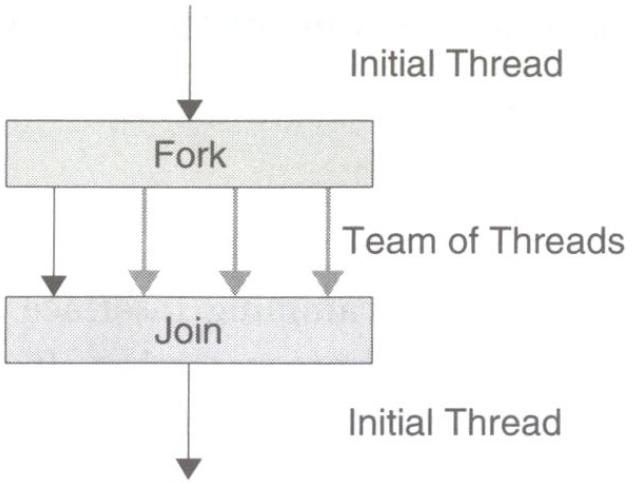


Figure 2.1: **The fork-join programming model supported by OpenMP** – The program starts as a single thread of execution, the initial thread. A team of threads is forked at the beginning of a parallel region and joined at the end.

O *thread master* (inicial, 0) é onde a execução do programa se inicia e termina.

O *master* em algum ponto cria um time (conjunto) de outros *threads*, que vão executar algo em paralelo: ***fork***.

Quando todos os *threads* de um time chegam ao final da região paralela, todos menos o *master* terminam: ***join***.

É possível haver várias regiões paralelas, com seriais entre elas.

OpenMP - Características

Em alguns casos, é possível também ter vários níveis de paralelização (*nesting*):

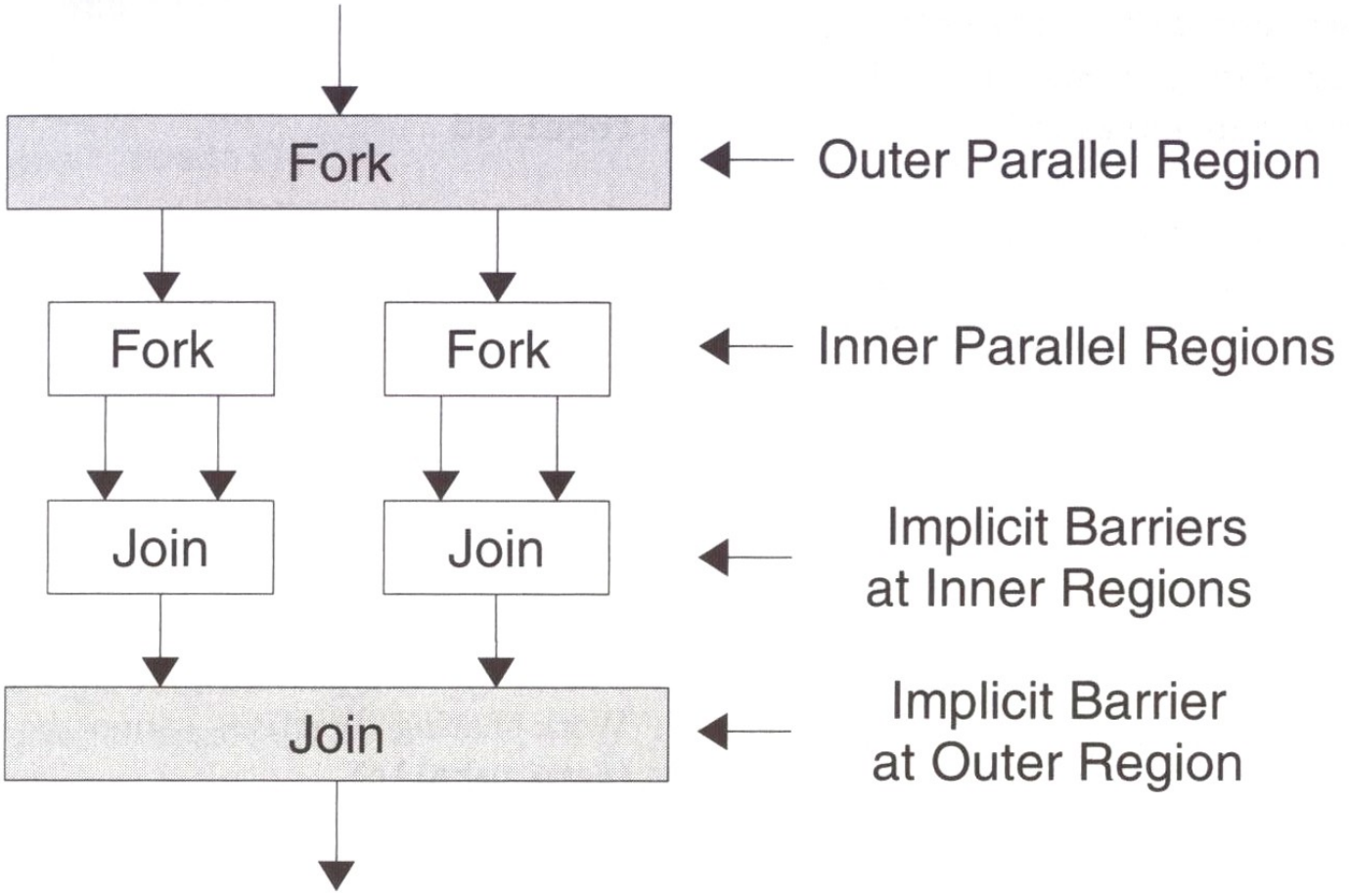


Figure 6.20: **Nested OpenMP parallel regions** – Inner parallel regions introduce implicit synchronization points in addition to the barrier synchronization points at the outer parallel regions.

(Chapman et al. 2007)

OpenMP - Construções

Trechos de código são paralelizados em OMP por **construções** (*constructs*).

Cada construção é especificada por diretrizes OMP (“comentários”), que delimitam a região de código a ser paralelizada, e como ela será paralelizada.

Fora das construções, o código é serial, inalterado pelo OMP.

Construções

- Base para todas as outras, especificando as seções paralelas:
  - `parallel`

- Dividem dados/tarefas entre os *threads* (*worksharing constructs*):
  - `loop`
  - `section`
  - `workshare` (só em Fortran)
  - `single`
  - `task` (só a partir do 3.0)

- Controlam sincronização (de dados e outros recursos):
  - `barrier`
  - `master`
  - `critical`
  - `atomic`
  - `locks`

OpenMP - `parallel`

Delimita uma região paralela.

Sozinha, apenas especifica que a região deve ser executada por todos os *threads*, sem especificar o que será diferente de um *thread* para outro. Ex:

C

```c
#include <stdio.h>
int main() {
/*Example of work replicated over the threads*/
#pragma omp parallel
{
  printf("Doing some stuff\n");
}
return 0;
}
```

C++

```cpp
#include <iostream>
int main() {
/*Example of work replicated over the threads*/
#pragma omp parallel
{
  std::cout<<"Doing some stuff"<<std::endl;
}
}
```

Bloco paralelo: executado em **`OMP_NUM_THREADS`** (variável de sistema) *threads* simultaneamente.

OpenMP - `parallel`

Delimita uma região paralela.

Sozinha, apenas especifica que a região deve ser executada por todos os *threads*, sem especificar o que será diferente de um *thread* para outro. Ex:

Fortran

```
program pp_omp_ex2
!Example of work replicated over the threads
Implicit none
!$omp parallel
write (6,*) 'Doing some stuff'
!$omp end parallel
end program pp_omp_ex2
```

Região paralela: executada em **OMP_NUM_THREADS** (variável de sistema) *threads* simultaneamente.

Quando executado, este programa apenas replica o trabalho, idêntico:

```
[user@computer dir]$ export OMP_NUM_THREADS=1

[user@computer dir]$ ./pp_omp_ex2.out
Doing some stuff

[user@computer dir]$ export OMP_NUM_THREADS=2

[user@computer dir]$ ./pp_omp_ex2.out
Doing some stuff
Doing some stuff
```

OpenMP - `parallel`

Como executar algo útil (diferente entre *threads*) só com `parallel`?

Fazendo uso do número do *thread*. Ex:

Fortran

```
program pp_omp_ex3
!Example of how to use thread numbers
!$ use omp_lib
implicit none
integer :: tid
!$omp parallel
tid=0 !Dummy thread number for the case OMP is not enabled
!$ tid=omp_get_thread_num()
write (6,*) 'Doing some stuff in thread ',tid
!$omp end parallel
end program pp_omp_ex3
```

Sentinelas (*sentinels*):

Linhas iniciadas com **`!$`** só são compiladas por compiladores que sabem OMP, se OMP estiver habilitado. Caso contrário, são só comentários, preservando o código serial.

A função **`omp_get_thread_num()`**, parte da biblioteca OMP, retorna o número do *thread* sendo executado. Pode ser usado para a decisão de o que cada *thread* faz.

Paralelização em OMP usando o índice do *thread* é semelhante a paralelização com CUDA (GPUs Nvidia).

OpenMP - `parallel`

Como executar algo útil (diferente entre *threads*) só com `parallel`?

Fazendo uso do número do *thread*. Ex:

Diretrizes para compilação condicional:

Só com compiladores que sabem OMP, se OMP estiver habilitado, _OPENMP é definido. Caso contrário, estes trechos são ignorados, deixando apenas o código serial.

C++

```
#include <iostream>
#ifdef _OPENMP
  #include<omp.h>
#endif
int main() {
/*Example of how to use thread numbers*/
#pragma omp parallel
{
  int tid=0; /*Dummy thread number for the case OMP is not enabled*/
  #ifdef _OPENMP
    tid=omp_get_thread_num();
  #endif
  std::cout << "Doing some stuff in thread "
   << tid << std::endl;
}
}
```

OpenMP - `parallel`

Como executar algo útil (diferente entre *threads*) só com `parallel`?

Fazendo uso do número do *thread*. Ex:

C

```c
#include <stdio.h>
#ifdef _OPENMP
  #include<omp.h>
#endif
int main() {
/*Example of how to use thread numbers*/
#pragma omp parallel
{
  int tid=0; /*Dummy thread number for the case OMP is not enabled*/
  #ifdef _OPENMP
    tid=omp_get_thread_num();
  #endif
  printf("Doing some stuff"
   " in thread %i\n",tid);
}
return 0;
}
```

Diretrizes para compilação condicional:

Só com compiladores que sabem OMP, se OMP estiver habilitado, _OPENMP é definido. Caso contrário, estes trechos são ignorados, deixando apenas o código serial.

OpenMP - `parallel`

Neste exemplo, ao executar o programa compilado sem OMP:

```
[user@computer dir]$ g++ pp_omp_ex3.cpp -o pp_omp_ex3.out

[user@computer dir]$ ./pp_omp_ex3.out
Doing some stuff in thread 0
```

Com OMP:

```
[user@computer dir]$ g++ pp_omp_ex3.cpp -o pp_omp_ex3.out -fopenmp

[user@computer dir]$ export OMP_NUM_THREADS=1

[user@computer dir]$ ./pp_omp_ex3.out
Doing some stuff in thread 0

[user@computer dir]$ export OMP_NUM_THREADS=2

[user@computer dir]$ ./pp_omp_ex3.out
Doing some stuff in thread 1
Doing some stuff in thread 0
```

A ordem da saída é aleatória: depende de que *thread* por acaso termine primeiro.

OpenMP – *worksharing constructs*

Apenas `parallel` provê controle de muito baixo nível sobre os *threads*.

As construções de divisão de trabalho (*worksharing*) são formas prontas de paralelizar alguns tipos de tarefas:

- `loop` (**do** em Fortran, **`for`** em C/C++):
  Cada thread executa uma fração das iterações do loop.

- `section`
  Cada `section` specifica uma tarefa a ser executada por um thread.

- `workshare` (só em Fortran)
  Especifica que as operações vetoriais contidas na seção devem ser distribuídas entre os *threads*.

- `single`
  Especifica que uma seção do trecho paralelizado deve ser executado por apenas um dos *threads* (importante para problemas de sincronia ou acesso a recursos).

- `task` (só a partir do OMP 3.0)
  Semelhante a `section`, cada uma especifica uma tarefa a ser dada a um *thread*, mas mais elaboradas, incluindo criação dinâmica de *tasks*.

OpenMP – *loop*

do em Fortran, **`for`** em C/C++

Cada *thread* executa uma fração das iterações do *loop*. Ex (Fortran):

```
program pp_omp_ex4
!Example of a parallel loop
!$ use omp_lib
implicit none
integer,parameter :: n=7
integer :: i,a(n),t(n),tid
!$omp parallel shared(a,t) private(i,tid)
tid=0 !Dummy thread number, for the case OMP is not enabled
!$ tid=omp_get_thread_num()
!$omp do
do i=1,n
  a(i)=2*i
  t(i)=tid !Record which thread did which iteration
enddo
!$omp end do
!$omp end parallel
!Show what was produced
write(6,*) 'i,a(i),thread number'
do i=1,n
  write(6,*) i,a(i),t(i)
enddo
end program pp_omp_ex4
```

OpenMP – *loop* (C++)

```
#include <iostream>
#ifdef _OPENMP
  #include<omp.h>
#endif
int main() {
/*Example of a parallel loop*/
int n=7;
int a[n],t[n];
#pragma omp parallel shared(a,t,n)
{
  int tid=0; /*Dummy thread number for the case OMP is not enabled*/
  #ifdef _OPENMP
    tid=omp_get_thread_num();
  #endif
  #pragma omp for
  for (int i=0; i<n; i++) {
    a[i]=2*i;
    t[i]=tid; /*Record which thread did which iteration*/
  }
}
/*Show what was produced*/
std::cout << "i,a(i),thread number" << std::endl;
for (int i=0; i<n; i++) {
  std::cout << i << " " << a[i] << " " << t[i] << std::endl;
}
}
```

OpenMP – *loop* (C)

```c
#include <stdio.h>
#ifdef _OPENMP
  #include<omp.h>
#endif
int main() {
/*Example of a parallel loop*/
int n=7;
int a[n],t[n];
#pragma omp parallel shared(a,t,n)
{
  int tid=0; /*Dummy thread number for the case OMP is not enabled*/
  #ifdef _OPENMP
    tid=omp_get_thread_num();
  #endif
  #pragma omp for
  for (int i=0; i<n; i++) {
    a[i]=2*i;
    t[i]=tid; /*Record which thread did which iteration*/
  }
}
/*Show what was produced*/
printf("i,a(i),thread number\n");
for (int i=0; i<n; i++) {
  printf("%i %i %i \n",i,a[i],t[i]);
}
return 0;
}
```

OpenMP – *loop*

Um resultado possível deste exemplo, quando executado:

```
[user@computer dir]$ gfortran pp_omp_ex4.f90 -o pp_omp_ex4.out

[user@computer dir]$ ./pp_omp_ex4.out
 i,a(i),thread number
           1           2           0
           2           4           0
           3           6           0
           4           8           0
           5          10           0
           6          12           0
           7          14           0

[user@computer dir]$ gfortran pp_omp_ex4.f90 -o pp_omp_ex4.out -fopenmp

[user@computer dir]$ export OMP_NUM_THREADS=2

[user@computer dir]$ ./pp_omp_ex4.out
 i,a(i),thread number
           1           2           0
           2           4           0
           3           6           0
           4           8           0
           5          10           1
           6          12           1
           7          14           1
```

OpenMP – `section`

Cada `section` é executada por um *thread* (paralelismo de tarefas). Ex (Fortran):

```
program pp_omp_ex5
!Example of sections
!$ use omp_lib
implicit none
integer,parameter :: n=7
integer :: i,a(n),ta(n),b(n),tb(n),tid
!$omp parallel shared(a,ta,b,tb) private(i,tid)
!$omp sections
!$omp section
 tid=0 !Dummy thread number, for the case OMP is not enabled
 !$ tid=omp_get_thread_num()
 do i=1,n
  a(i)=2*i
  ta(i)=tid !Record which thread did which iteration
 enddo
!$omp section
 tid=0 !Dummy thread number, for the case OMP is not enabled
 !$ tid=omp_get_thread_num()
 do i=1,n
  b(i)=3*i
  tb(i)=tid !Record which thread did which iteration
 enddo
!$omp end sections
!$omp end parallel
!Show what was produced
write(6,*) 'i,a(i),thread that made a(i),b(i), thread that made b(i)'
do i=1,n
 write(6,*) i,a(i),ta(i),b(i),tb(i)
enddo
end program pp_omp_ex5
```

OpenMP – `section`

Cada `section` é executada por um *thread* (paralelismo de tarefas). Ex (C++):

```
#include <iostream>
#ifdef _OPENMP
  #include<omp.h>
#endif
int main() {
/*Example of sections*/
int n=7;
int a[n],ta[n],b[n],tb[n];
```

(continua na próxima página)

```
#pragma omp parallel shared(a,ta,b,tb,n)
{
  #pragma omp sections
  {
    #pragma omp section
    {
      int tid=0; /*Dummy thread number for the case OMP is not enabled*/
      #ifdef _OPENMP
        tid=omp_get_thread_num();
      #endif
      for (int i=0; i<n; i++) {
        a[i]=2*i;
        ta[i]=tid; /*Record which thread did which iteration*/
      }
    }
    #pragma omp section
    {
      int tid=0; /*Dummy thread number for the case OMP is not enabled*/
      #ifdef _OPENMP
        tid=omp_get_thread_num();
      #endif
      for (int i=0; i<n; i++) {
        b[i]=3*i;
        tb[i]=tid; /*Record which thread did which iteration*/
      }
    }
  }
}
/*Show what was produced*/
std::cout<<"i,a(i),thread that made a(i),b(i), thread that made b(i)"<<std::endl;
for (int i=0; i<n; i++) {
  std::cout<<i<<" "<<a[i]<<" "<<ta[i]<<" "<<b[i]<<" "<<tb[i]<<std::endl;
}
}
```

OpenMP – `section`

Cada `section` é executada por um *thread* (paralelismo de tarefas). Ex (C):

```c
#include <stdio.h>
#ifdef _OPENMP
  #include<omp.h>
#endif
int main() {
/*Example of sections*/
int n=7;
int a[n],ta[n],b[n],tb[n];
```

(continua na próxima página)

```
#pragma omp parallel shared(a,ta,b,tb,n)
{
  #pragma omp sections
  {
    #pragma omp section
    {
      int tid=0; /*Dummy thread number for the case OMP is not enabled*/
      #ifdef _OPENMP
        tid=omp_get_thread_num();
      #endif
      for (int i=0; i<n; i++) {
        a[i]=2*i;
        ta[i]=tid; /*Record which thread did which iteration*/
      }
    }
    #pragma omp section
    {
      int tid=0; /*Dummy thread number for the case OMP is not enabled*/
      #ifdef _OPENMP
        tid=omp_get_thread_num();
      #endif
      for (int i=0; i<n; i++) {
        b[i]=3*i;
        tb[i]=tid; /*Record which thread did which iteration*/
      }
    }
  }
}
/*Show what was produced*/
printf("i,a(i),thread that made a(i),b(i), thread that made b(i)\n");
for (int i=0; i<n; i++) printf("%i %i %i %i %i\n",i,a[i],ta[i],b[i],tb[i]);
return 0;
}
```

OpenMP – `workshare`

Especifica que as operações vetoriais devem ser executadas em paralelo, com os elementos divididos entre os *threads*.

Apenas para Fortran.

Ex:

```
program pp_omp_ex6
!Example of workshare
!$ use omp_lib
implicit none
integer,parameter :: n=7
integer :: i,j,a(n),b(n),c(n)
!$omp parallel shared(a,b,c) private(i,j)
!$omp workshare
forall (i=1:n) a(i)=i
forall (j=1:n) b(j)=2*j
!OMP makes sure that all results are available before the next line
c(1:n)=a(1:n)+b(1:n)
!$omp end workshare
!$omp end parallel
end program pp_omp_ex6
```

OpenMP – cláusulas

Especificam ou modificam aspectos do funcionamento das construções:

Acesso a dados:

- `shared` - Especifica variáveis a serem compartilhadas entre todos os *threads*.
- `private` - Especifica variáveis a serem privadas (locais): uma cópia existe para cada *thread*.
- `lastprivate` - Como `private`, mas mantém o último valor local após a seção paralela.
- `Firstprivate` - Como `private`, mas inicializa a variável com o valor que tinha antes da seção paralela.
- `default` - Determina se por default uma variável é `shared`, `private` ou `none`.

Controle de execução:

- `nowait` - Suprime barreiras implícitas (como ao final de *loops*).
- `schedule` - Determina como iterações de *loops* são divididas entre *threads*.

OpenMP – sincronização

Em algumas situações, é necessário controlar o acesso ou a ordem de operações entre *threads*. Ex. (Fortran):

```
program pp_omp_ex7
!Example of a race condition
!$ use omp_lib
implicit none
integer :: i,n,a
n=1000000
a=0
!$omp parallel shared(a,n) private(i)
!$omp do
do i=1,n
  !When one thread writes to a, the value of a
  !it had read might not be current anymore
  a=a+i
enddo
!$omp end do
!$omp end parallel
!Show what was produced
write(6,*) 'a=',a
end program pp_omp_ex7
```

OpenMP – sincronização

Em algumas situações, é necessário controlar o acesso ou a ordem de operações entre *threads*. Ex. (C++):

```
#include <iostream>
#ifdef _OPENMP
  #include<omp.h>
#endif
int main() {
/*Example of a race condition*/
int n=1000000,a=0;
#pragma omp parallel shared(a,n)
{
  #pragma omp for
  for (int i=0; i<n; i++) {
    /*When one thread writes to a, the
value of a
    it had read might not be current
anymore*/
    a+=i;
  }
}
/*Show what was produced*/
std::cout << "a=" << a << std::endl;
}
```

OpenMP – sincronização

Em algumas situações, é necessário controlar o acesso ou a ordem de operações entre *threads*. Ex. (C):

```c
#include <stdio.h>
#ifdef _OPENMP
  #include<omp.h>
#endif
int main() {
/*Example of a race condition*/
int n=1000000,a=0;
#pragma omp parallel shared(a,n)
{
  #pragma omp for
  for (int i=0; i<n; i++) {
    /*When one thread writes to a, the value of a
    it had read might not be current anymore*/
    a+=i;
  }
}
/*Show what was produced*/
printf("a= %i\n",a);
return 0;
}
```

OpenMP – sincronização

Condição de corrida (*race condition*)
O código do exemplo anterior tem comportamento indeterminado.
O resultado depende de que *thread* por acaso chega antes ao trecho que altera a variável compartilhada entre *threads*.
Exemplos de resultados de o executar:

```
[user@computer dir]$ gcc -std=c99 pp_omp_ex7.c -o pp_omp_ex7.out

[user@computer dir]$ ./pp_omp_ex7.out
 A=  1784293664

[user@computer dir]$ gcc -std=c99 pp_omp_ex7.c -o pp_omp_ex7.out -fopenmp

[user@computer dir]$ export OMP_NUM_THREADS=20

[user@computer dir]$ ./pp_omp_ex7.out
 a=  1340951740

[user@computer dir]$ ./pp_omp_ex7.out
 a=   482130990

[user@computer dir]$ ./pp_omp_ex7.out
 a=  1846303706

[user@computer dir]$ ./pp_omp_ex7.out
 a=  1784293664

[user@computer dir]$ ./pp_omp_ex7.out
 a=   822224612
```

OpenMP – sincronização

Construções para evitar condições de corrida, ou controlar a execução de um trecho do código:

- `barrier` – barreira explícita: execução só continua quando todos os *threads* chegam à barreira.
- `ordered` – força que um trecho seja executado pelos *threads* na ordem do loop.
- `master` – código a ser executado só pelo *thread* master (como semelhante a `single`, mas sem barreiras implícitas).
- `critical` – especifica um trecho que pode ser executado por apenas um thread de cada vez.
- `atomic` – Alternativa mais eficiente (mas mais restrita) a `critical`, especifica uma operação atômica, para impedir que múltiplos *threads* a executem ao mesmo tempo.
- `locks` – Semelhante a semáforos (*semaphores*), define uma variável de trava (`lock`): apenas um *thread* pode estar com a trava de cada vez; os outros têm que esperar até que ela seja liberada. Pode provocar *deadlocks* (adiante).

MPI – Message Passing Interface

O principal padrão para paralelismo em memória distribuída (**múltiplos processos**).

Extrínseco a linguagens / compiladores: **formado apenas por uma biblioteca, especificando a interface (API) de rotinas / funções para comunicação entre os processos.**

É um padrão menos organizado (mais variável) que OpenMP.

Não há memória compartilhada: MPI apenas faz comunicação entre processos, por chamadas de funções / rotinas.

Como padrão é mais simples que OpenMP (apenas rotinas de comunicação, todo o paralelismo é explícito).

Por ser uma ferramenta mais simples (tem menos recursos, e é de mais baixo nível), dá mais trabalho para usar.

Programas têm que ser reescritos (com freqüência, reestruturados) para fazer uso das funções do MPI:

É possível (com diretrizes de preprocessador) fazer o mesmo código funcionar com e sem MPI, mas em geral é trabalhoso.

MPI – Características

Usado tanto para paralelismo de dados como paralelismo de tarefas.

Tradicionalmente implementado para C, C++ e Fortran:

- A funcionalidade e a semântica (significado das construções) são as mesmas, dentro dos limites de cada linguagem.
- A sintaxe (forma de expressar) é semelhante (não é idêntica por as linguagens terem sintaxes diferentes).
- APIs específicas para Fortran, C e C++.

MPI 1 até 1.2. MPI 2 até 2.2. MPI 3 ainda em discussão.

Implementações menos comuns para outras linguagens (ex: Java, Python, IDL).

Implementações comuns (variam no que suportam, e na forma de usar):

- Open MPI (MPI 2, aberta)
- MPICH (MPI 1, aberta)
- MPICH2 (MPI 2, aberta)
- Intel (proprietária)
- PGI (MPI 1 e 2, proprietária)
- SGI MPT (proprietária)

MPI – Características

A criação dos múltiplos processos normalmente é feita externamente:

- Um programa da biblioteca (em geral, **`mpirun`** ou **`mpiexec`**) inicia `N` processos.
- Os processos estão dentro de um mesmo domínio de comunicação (têm o mesmo comunicador, **`MPI_COMM_WORLD`**), cada um com seu número dentro deste domínio.
- Os processos podem ser (com freqüência são) executados em computadores diferentes (em geral, diferentes nós de um *cluster*).
- Normalmente todos os processos são cópias do mesmo programa: cabe a cada processo fazer algo diferente baseado em seu número: SPMD (*Single Program, Multiple Data*), **o que não é mesmo que SIMD** (*Single Instruction, Multiple Data*): cada processo pode fazer coisas diferentes.
- Em alguns casos, é possível que um processo inicie outros (*nesting*).

É comum, mas não necessário, que os processos tenham acesso a um mesmo sistema de arquivos, podendo compartilhar arquivos (é problema deles o fazer: MPI apenas transmite mensagens entre processos).

Processos são agrupados por **comunicadores**: conjuntos de processos, formam domínios sobre os quais comunicação ocorre. Todos sempre fazem parte do comunicador default, **`MPI_COMM_WORLD.`**

Processos são executados por **processadores** (não o mesmo que processadores físicos).

MPI x OpenMP

A estrutura do que acontece entre processos MPI pode ser pensada como um caso particular* do que pode ser feito entre *threads* OpenMP. Seria semelhante a:

- **Paralelização apenas com a construção de mais baixo nível (`parallel`)**: é trabalho de cada unidade (processo em MPI, *thread* em OpenMP) decidir fazer algo diferente, com base em seu número.
- **Todas as variáveis sendo `private`: não há memória compartilhada.**
- Por ser apenas comunicação entre processos, MPI tem grande variedade de funções para comunicar dados de formas diferentes.
- Não há construções de mais alto nível, que automaticamente dividam o trabalho entre os processos.
- **Cabe ao programador estruturar a divisão do trabalho, seja por tarefas, seja por dados.**

*Mas OpenMP não pode usar memória distribuída; neste ponto, MPI não é um caso particular de OpenMP.

MPI x OpenMP

Relembrando o exemplo de `parallel` em OpenMP:

Fortran

```
program pp_omp_ex3
!Example of how to use thread numbers
!$ use omp_lib
implicit none
integer :: tid
!$omp parallel
tid=0 !Dummy thread number for the case OMP is not enabled
!$ tid=omp_get_thread_num()
write (6,*) 'Doing some stuff in thread ',tid
!$omp end parallel
end program pp_omp_ex3
```

Em MPI, de forma semelhante, cada processo conhece o seu número (o equivalente ao número do *thread*, `tid,` acima).

- Cabe ao processo usar seu número para decidir o que fazer (para não repetir a mesma ação de todos os outros).

A maior diferença está na ausência de variáveis compartilhadas:

- Mensagens são usadas para enviar dados entre processos, chamando as funções do MPI.

MPI – exemplo ilustrativo

Descobrir e mostrar onde é executado cada processo. Em Fortran:

```
program pp_mpi_ex1
!Simple general MPI example
use mpi
implicit none
integer :: numtasks,rank,ierr,rc,name_len,name_len_max
integer :: i
!Variables for MPI
character(len=MPI_MAX_PROCESSOR_NAME) :: proc_name
integer :: isendbuf,isendcount,root
integer,allocatable :: irecvbuf(:)
character(1),allocatable :: crecvbuf(:)
!Variables to acumulate data
integer,parameter :: length=100000
real(selected_real_kind(15,307)) :: atmp(length),btmp(length)

!MPI initialization
call mpi_init(ierr)
if (ierr .ne. MPI_SUCCESS) then
  print *,'Error starting MPI program. Terminating.'
  call mpi_abort(MPI_COMM_WORLD,rc,ierr)
end if

!Initialize "global" variables
root=0 !The process that will do the receiving and output
isendcount=1 !How many integers will be gathered

!Get the MPI "locals"
call mpi_comm_rank(MPI_COMM_WORLD,rank,ierr) !The process rank
call mpi_get_processor_name(proc_name,name_len,ierr) !The processor name
```

MPI – exemplo ilustrativo (Fortran, continuação)

```
!Do the initialization, only by the root process
if (rank .eq. root) then
  call mpi_comm_size(MPI_COMM_WORLD,numtasks,ierr)
  allocate(irecvbuf(numtasks*isendcount)) !Make room for gathering the process
ranks
endif

!Do the initialization, only by the root process
if (rank .eq. root) then
  call mpi_comm_size(MPI_COMM_WORLD,numtasks,ierr)
  allocate(irecvbuf(numtasks*isendcount)) !Make room for gathering the process
ranks
endif

!Do "the work" (by all processes)
isendbuf=rank

!Find out the maximum length of the processor name, accross all processes
call
mpi_allreduce(name_len,name_len_max,isendcount,MPI_INTEGER,MPI_MAX,MPI_COMM_WOR
LD,ierr)

!Waste some time, so that the program takes a noticeable time to finish
forall(i=1:length) atmp(i)=(i-1)/((length-1)*1d0)
do i=1,100
  btmp=acos(cos(atmp*acos(-1d0)))
enddo
!End of "the work"
```

MPI – exemplo ilustrativo (Fortran, continuação)

```
!Make room to get the processor names
if (rank .eq. root) then
  allocate(crecvbuf(name_len_max*numtasks))
endif

!Get the results from everybody
call
mpi_gather(isendbuf,isendcount,MPI_INTEGER,irecvbuf,isendcount,MPI_INTEGER,root
,MPI_COMM_WORLD,ierr)
call
mpi_gather(proc_name(1:name_len_max),name_len_max,MPI_CHARACTER,crecvbuf,name_l
en_max,MPI_CHARACTER,root,MPI_COMM_WORLD,ierr)

!Show the results (only the root process does this)
if (rank .eq. root) then
  print *,'I am process number ',rank
  print *,'Running on processor ',proc_name(1:name_len)
  print *,'Number of tasks is ',numtasks
  do i=0,numtasks-1
    print *,'Process rank ',irecvbuf(i+1),' ran on processor
',crecvbuf(i*name_len_max+1:(i+1)*name_len_max)
  enddo
endif

call mpi_finalize(ierr)
end program pp_mpi_ex1
```

MPI – exemplo ilustrativo

Descobrir e mostrar onde é executado cada processo. Em C++:

```
#include <iostream>
#include <mpi.h>
#include <math.h>
#include <string>
int main(int argc, char * argv[]) {
/*Simple general MPI example*/

/*MPI initialization*/
MPI::Init(argc,argv);

/*Initialize "global" variables*/
int root=0; /*The process that will do the receiving and output*/
int isendcount=1; /*How many integers will be gathered*/
/*Get the MPI "locals"*/
int rank=MPI::COMM_WORLD.Get_rank(); /*The process rank*/
char proc_name[MPI::MAX_PROCESSOR_NAME];
int name_len;
MPI::Get_processor_name(proc_name,name_len); /*The processor name*/

/*Do the initialization, only by the root process*/
int numtasks;
int* irecvbuf;
if (rank==root) {
  numtasks=MPI::COMM_WORLD.Get_size();
  irecvbuf=new int[numtasks]; /*Make room for gathering the process ranks*/
}
```

MPI – exemplo ilustrativo – C++ (continuação)

```
/*Do "the work" (by all processes)*/
int isendbuf=rank;
int name_len_max;

/*Find out the maximum length of the processor name, accross all processes*/
MPI::COMM_WORLD.Allreduce(&name_len,&name_len_max,isendcount,MPI::INT,MPI::MAX)
;

/*Waste some time, so that the program takes a noticeable time to finish*/
int length=100000;
double atmp[length],btmp[length];
for (int i=0; i<length; i++) atmp[i]=(i-1)/((length-1)*1e0);
for (int i=0; i<100; i++) btmp[i]=acos(cos(atmp[i]*acos(-1e0)));
/*End of "the work"*/

/*Make room to get the processor names*/
char* crecvbuf;
if (rank==root) crecvbuf=new char[(name_len_max)*numtasks];
```

MPI – exemplo ilustrativo – C++ (continuação)

```
/*Get the results from everybody*/
MPI::COMM_WORLD.Gather(&isendbuf,isendcount,MPI::INT,irecvbuf,isendcount,MPI::INT,root);
MPI::COMM_WORLD.Gather(&proc_name,name_len,MPI::CHAR,crecvbuf,name_len,MPI::CHAR,root);

/*Show the results (only the root process does this)*/
if (rank==root) {
  std::cout<<"I am process number "<<rank<<std::endl;
  std::cout<<"Running on processor "<<proc_name<<std::endl;
  std::cout<<"Number of tasks is "<<numtasks<<std::endl;
  std::string names(crecvbuf);
  for (int i=0; i<numtasks; i++){
    std::cout<<"Process rank "<<irecvbuf[i]<<" ran on processor ";
    std::cout<<names.substr(i*name_len_max,name_len_max)<<std::endl;
  }
}

MPI::Finalize();
}
```

MPI – exemplo ilustrativo

Descobrir e mostrar onde é executado cada processo. Em C:

```c
#include <stdio.h>
#include <mpi.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
int main(int argc, char * argv[]) {
/*Simple general MPI example*/

/*MPI initialization*/
MPI_Init(&argc,&argv);

/*Initialize "global" variables*/
int root=0; /*The process that will do the receiving and output*/
int isendcount=1; /*How many integers will be gathered*/
/*Get the MPI "locals"*/
int rank;
MPI_Comm_rank(MPI_COMM_WORLD,&rank); /*The process rank*/
char proc_name[MPI_MAX_PROCESSOR_NAME];
int name_len;
MPI_Get_processor_name(proc_name,&name_len); /*The processor name*/

/*Do the initialization, only by the root process*/
int numtasks;
int* irecvbuf;
if (rank==root) {
  MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
  irecvbuf=malloc(numtasks*sizeof(int)); /*Make room for gathering the process
ranks*/
}
```

MPI – exemplo ilustrativo – C (continuação)

```c
/*Do "the work" (by all processes)*/
int isendbuf=rank;
int name_len_max;

/*Find out the maximum length of the processor name, accross all processes*/
MPI_Allreduce(&name_len,&name_len_max,isendcount,MPI_INT,MPI_MAX,MPI_COMM_WORLD);

/*Waste some time, so that the program takes a noticeable time to finish*/
int length=100000;
double atmp[length],btmp[length];
for (int i=0; i<length; i++) atmp[i]=(i-1)/((length-1)*1e0);
for (int i=0; i<100; i++) btmp[i]=acos(cos(atmp[i]*acos(-1e0)));
/*End of "the work"*/

/*Make room to get the processor names*/
char* crecvbuf;
if (rank==root) crecvbuf=malloc((name_len_max)*numtasks*sizeof(char));
```

MPI – exemplo ilustrativo – C (continuação)

```c
/*Get the results from everybody*/
MPI_Gather(&isendbuf,isendcount,MPI_INT,irecvbuf,isendcount,MPI_INT,root,MPI_COMM_WORLD);
MPI_Gather(&proc_name,name_len,MPI_CHAR,crecvbuf,name_len,MPI_CHAR,root,MPI_COMM_WORLD);

/*Show the results (only the root process does this)*/
if (rank==root) {
  printf("I am process number %i\n",rank);
  printf("Running on processor %s\n",proc_name);
  printf("Number of tasks is %i\n",numtasks);
  char* name=malloc((name_len_max)*sizeof(char));
  for (int i=0; i<numtasks; i++){
    strncpy(name,crecvbuf+i*(name_len),name_len);
    printf("Process rank %i ran on processor %s\n",irecvbuf[i],name);
  }
}

MPI_Finalize();
return 0;
}
```

MPI – exemplo ilustrativo

Compilação e execução deste exemplo, com o Open MPI (detalhes variam com implementação):

```
penteado@gina-n2:~/mpitest> mpif90 pp_mpi_ex1.f90 -o pp_mpi_ex1.out

penteado@gina-n2:~/mpitest> mpirun -np 4 -machinefile nodefile ./pp_mpi_ex1.out
 I am process number              0
 Running on processor gina-n2
 Number of tasks is               4
 Process rank             0  ran on processor gina-n2
 Process rank             1  ran on processor gina-n1
 Process rank             2  ran on processor gina-n1
 Process rank             3  ran on processor gina-n2
```

A lista de nós a usar (no caso, **gina-n1** e **gina-n2**) vem do arquivo **nodefile**. Se não especificada, em geral todos os processos são executados no próprio computador de onde é chamado o **mpirun** / **mpiexec**.

mpif90 é um *wrapper* fornecido com muitas implementações, que chama o compilador de Fortran com os argumentos necessários para usar MPI. Há, de forma semelhante, **mpicc** (C) e **mpic++** (C++). No caso, equivalente a:

```
[user@computer dir]$ /usr/lib64/openmpi/bin/mpif90 -show
gfortran -I/usr/include/openmpi-x86_64 -pthread -m64
-I/usr/lib64/openmpi/lib -L/usr/lib64/openmpi/lib -lmpi_f90 -lmpi_f77
-lmpi -lopen-rte -lopen-pal -ldl -Wl,--export-dynamic -lnsl -lutil -lm
-ldl
```

MPI – Formas de comunicação

Operação com bloqueio (*blocking operation*):

- O processo só pode continuar para a próxima operação quando esta tiver terminado.

Operação sem bloqueio (*non-blocking operation*):

- O processo pode continuar para a próxima operação antes do fim desta: chegar à linha de código seguinte não significa que esta operação terminou.

De uma forma análoga, global, as comunicações podem ser vistas como:

Sícronas:

- O processo que remetente tem que esperar que o processo destinatário comece a receber os dados.

Assíncronas:

- O processo remetente não precisa esperar o destinatário começar a receber os dados, podendo continuar com a próxima operação. Alcançar a linha seguinte do programa não significa que o destinatário já esteja recebendo os dados.

MPI – Formas de comunicação

Operação com bloqueio (*blocking operation*):

- O processo só pode continuar para a próxima operação quando esta tiver terminado.

Operação sem bloqueio (*non-blocking operation*):

- O processo pode continuar para a próxima operação antes do fim desta: chegar à linha de código seguinte não significa que esta operação terminou.

De uma forma análoga, global, as comunicações podem ser vistas como:

Sícronas: 

- O processo que remetente tem que esperar que o processo destinatário comece a receber os dados.

Assíncronas: 

- O processo remetente não precisa esperar o destinatário começar a receber os dados, podendo continuar com a próxima operação. Alcançar a linha seguinte do programa não significa que o destinatário já esteja recebendo os dados.

MPI – Principais funções

Controle

- **`MPI_Init`** - deve ser sempre a primeira operação, para inicializar o MPI.
- **`MPI_Finalize`** - deve ser sempre a última operação, para encerrar o MPI.
- **`MPI_Abort`** - usada para terminar a execução de todos os processos.
- **`MPI_Barrier`** - barreira explícita: execução só continua quando todos os processos alcançarem a barreira.
- **`MPI_Test`** - verifica se uma comunicação (`Send` / `Receive`) já terminou.
- **`MPI_Wait`** - espera até que uma comunicação (`Send` / `Receive`) termine.

Informações

- **`MPI_Comm_size`** - Informa o número de processos no comunicador.
- **`MPI_Comm_rank`** - Informa o número do processo, dentro do comunicador.
- **`MPI_Get_processor_name`** - Informa o nome do processador executando o processo (costuma ser o nome do computador).

MPI – Principais funções

Comunicação (apenas as funções mais comuns): Mandam blocos de variáveis de um dos tipos do MPI (tipos básicos para inteiros, reais e caracteres), ou de uma estrutura definida com rotinas do MPI.

- **`MPI_Send`** - Manda uma mensagem para um processo específico (bloqueando).
- **`MPI_Recv`** - Recebe uma mensagem, de um processo específco ou qualquer um (bloqueando).
- **`MPI_Sendrecv`** - Combina Send com Recv em uma única função.
- **`MPI_Isend`** - Como Send, mas sem bloquear.
- **`MPI_Irecv`** - Como Recv, mas sem bloquear.
- **`MPI_Bcast`** - Manda a mesma mensagem para todo mundo (bloqueando).
- **`MPI_Reduce`** - Combina um elemento de dados de cada processo, por uma operação binária (ex: adição, max), resultando em um único valor (bloqueando).
- **`MPI_Gather`** - Recolhe um elemento de dados de cada processo (bloqueando).
- **`MPI_Scatter`** - O oposto de `Gather` (bloqueando).

Boost.MPI

A API (interface) MPI padrão para C++ foi classificada como *deprecated* (“obsoleta”) no MPI 2.2, porque era pouco mais que sintaxe pouco melhor para a API em C.

A API Boost.MPI, da biblioteca Boost (C++) é **muito mais conveniente** que a API padrão:

- Objetos fazem todo o trabalho de comunicações e contabilidade de dados, sem ser necessário carregar muitas variáveis para passar muitos argumentos.
- A serialização do Boost se encarrega de empacotar / distribuir os dados, sem que o programador precise criar e interpretar os buffers de baixo nível do MPI.
- **Muito mais fácil mandar variáveis** que sejam mais complicadas que arrays 1D de um dos tipos (simples) do MPI. Especialmente relevante para tipos de tamanhos variáveis (*strings*, em particular).

O mesmo exemplo anterior, usando Boost.MPI:

Boost.MPI

O mesmo exemplo anterior, usando Boost.MPI:

```
#include <boost/mpi.hpp>
#include <boost/serialization/string.hpp>
#include <iostream>
#include <math.h>
#include <string>
int main(int argc, char * argv[]) {
/*Simple general Boost.MPI example*/

/*MPI initialization*/
boost::mpi::environment env(argc,argv);
boost::mpi::communicator world;

/*Initialize "global" variables*/
int root=0; /*The process that will do the receiving and output*/

/*Do "the work" (by all processes)*/
/*Waste some time, so that the program takes a noticeable time to finish*/
int length=100000;
double atmp[length],btmp[length];
for (int i=0; i<length; i++) atmp[i]=(i-1)/((length-1)*1e0);
for (int i=0; i<100; i++) btmp[i]=acos(cos(atmp[i]*acos(-1e0)));
/*End of "the work"*/
```

Pypar

Exemplo similar, usando Pypar:

```
import pypar
p = pypar.rank()
P = pypar.size()
node = pypar.get_processor_name()
print "I am process %d of %d on node %s" %(p, P, node)
pypar.finalize()
```

```
[penteado@javelina mpihello]$ python ./ni.py
Pypar (version 2.1.5) initialised MPI OK with 1 processors
I am process 0 of 1 on node javelina.phy.nau.edu
```

```
[penteado@javelina mpihello]$ mpirun -np 3 python ./ni.py
Pypar (version 2.1.5) initialised MPI OK with 3 processors
I am process 2 of 3 on node javelina.phy.nau.edu
I am process 0 of 3 on node javelina.phy.nau.edu
I am process 1 of 3 on node javelina.phy.nau.edu
```

pp_pypar

Esqueleto para código MPI em Python:

```
class pypar_test(pp_pypar.pp_pypar):

    #Here goes the code to create/read the data to process, put into
one element of work_array per process
    def make_stuff_to_process(self):
        work_array = range(0,7)
        for i in range(len(work_array)):
            work_array[i] = np.arange(0.0, 10.0)
        return work_array

    #Here goes the code that does the processing of each element of
work_array
    def do_stuff(self,work_array):
        # Code below simulates a task running
        time.sleep(np.random.random_integers(low=0, high=5))
        result_array = work_array.copy()
        return result_array


if __name__ == '__main__':
  p=pypar_test()
  p.process_cyc
  print "I got "+str(len(p.results))+" Results"
```

http://www.ppenteado.net/pp_pylib/pp_pypar_test.py

Boost.MPI – ex. (continuação)

```
/*Get the results from everybody*/
std::vector<std::string> proc_names; /*Where the names will go*/
std::vector<int> proc_ranks; /*Where the numbers will go*/
std::string name=boost::mpi::environment::processor_name();
if (world.rank()==root) { /*If root, must provide a place to store the data*/
  gather(world,world.rank(),proc_ranks,root);
  gather(world,name,proc_names,root);
} else { /*If not root, just send the data to the gather*/
  gather(world,world.rank(),root);
  gather(world,name,root);
}

/*Show the results (only the root process does this)*/
if (world.rank()==root) {
  std::cout<<"I am process number "<<world.rank()<<std::endl;
  std::cout<<"Running on processor "<<proc_names[world.rank()]<<std::endl;
  std::cout<<"Number of tasks is "<<world.size()<<std::endl;
  for (int i=0; i<world.size(); i++){
    std::cout<<"Process rank "<<proc_ranks[i]<<" ran on processor ";
    std::cout<<proc_names[i]<<std::endl;
  }
}
}
```

Deadlocks

Tanto com OpenMP como com MPI (muito mais facilmente em MPI) é necessário cuidado para evitar *deadlocks*:

- A unidade (processo / thread) **A** está esperando a unidade **B**, e
- a unidade **B** está esperando a unidade **A**.

Ambas ficam esperando uma a outra, eternamente: o programa trava.

Deadlocks

Tanto com OpenMP como com MPI (muito mais facilmente em MPI) é necessário cuidado para evitar *deadlocks*:

- A unidade (processo / thread) **A** está esperando a unidade **B**, e
- a unidade **B** está esperando a unidade **A**.

Ambas ficam esperando uma a outra, eternamente: o programa trava.

555-0123

Discando 555-3210

555-3210

Discando 555-0123

Ambos ficam esperando, porque estão ligando para um número ocupado.

GPUs

Graphical Processing Units

Criadas para livrar a CPU de processar gráficos em 2D e 3D e vídeos.

Trabalho simples, mas pesado (grande volume de dados, muitos quadros por segundo).

~1990-1995, aceleradores gráficos criados para realizar operações em imagens.

1992 – OpenGL (Open Graphics Library), criado pela SGI.

- Padrão aberto para API (Application Programming Interface) para gerar cenas em 3D.

1997 – OpenGL implementado em GPUs, livrando as CPUs deste trabalho pesado.

Complexidade exigida para cenas em 3D levou ao desenvolvimento de *shaders* programáveis – capazes de executar código arbitrário.

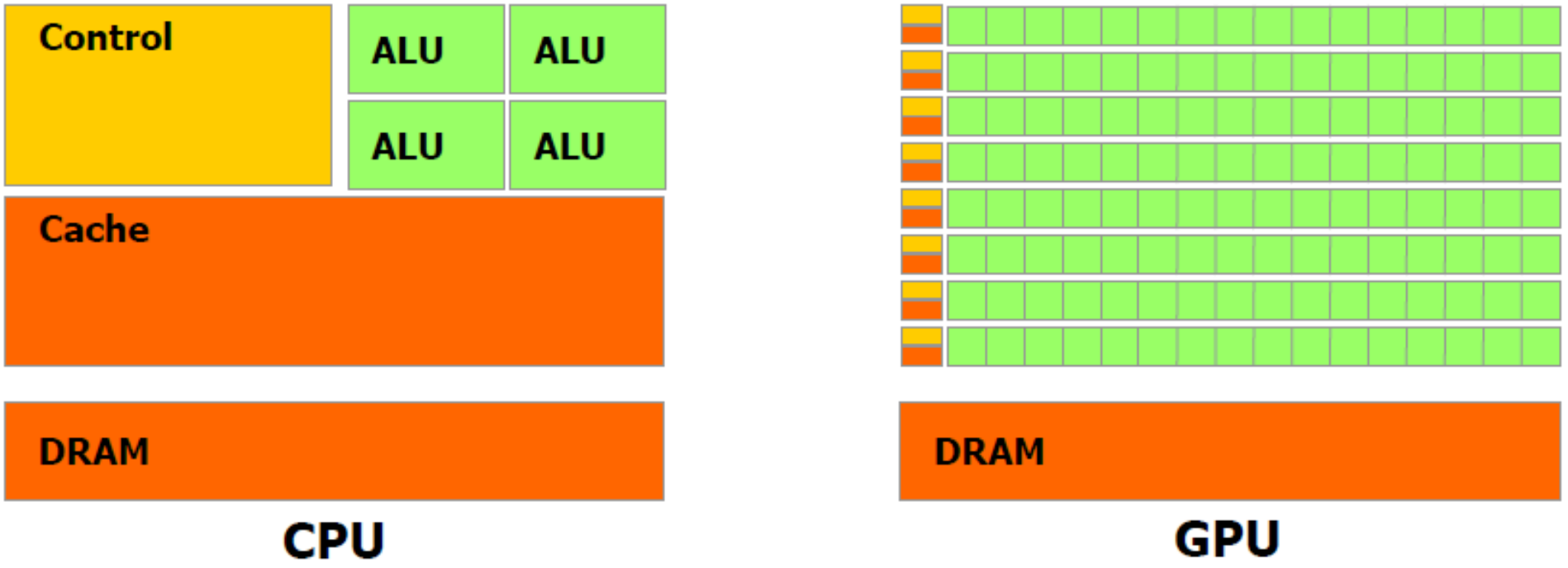
Hoje podem ser 1-2 ordens de magnitude mais rápidas que CPUs de preços semelhantes.

GPUs x CPUs

- Nvidia GeForce GTX série 700 (Kepler):
  $7x10^9$ transistores, 2880 núcleos, ~0.2-5Tflops, ~$700
- Intel Core i7:
  $1.4x10^9$ transistores, 4-8 núcleos (HT), ~0.1-0.2Tflops, ~$300

CPUs têm núcleos mais complexos e mais cache: melhores para operações independentes, acesso aleatório à memória e operações com poucos dados.

GPUs melhores para tarefas simples em grandes *arrays*.



(©NVIDIA)

Plataformas disponíveis

Antigamente, toda a programação para GPUs usava OpenGL:

- Necessário expressar tudo como operações gráficas (ex. operações sobre texturas): difícil, baixo nível e desajeitado, e só em precisão simples.

APIs específicas para GPGPU (*General Purpose Computing on GPUs*):

- CUDA (2007) - Compute Unified Device Architecture
  - Desenvolvida pela NVIDIA, juntamente com o hardware.
  - Proprietária mas gratuita.
  - Mais madura e amigável.

- OpenCL (2008) – Open Computation Language
  - Desenvolvida pelo Khronos Group, semelhante a OpenGL.
  - Mais flexível.

- Direct Compute (2009) - Baseada no DirectX 11
  - Mais primitiva e limitada, menos usada.

- OpenACC (2011)
  - Padrão aberto para diretrizes de compilação (semelhante a OpenMP).
  - Implementado para CUDA no compilador PGI, outros em desenvolvimento.

- BrookGPU (2007): quase ninguém conhece ou usa.

OpenCL

Padrão aberto, desenvolvido pelo Khronos Group.

API independente de hardware – pode ser implementada por CPUs, clusters, GPUs, OpenGL, CUDA.

Programas com OpenCL podem ser executados em qualquer plataforma OpenCL, independente do hardware.

Deliberadamente de baixo nível (OpenCL C, próximo ao hardware, apesar de abstrato).

Ainda não tão evoluído como CUDA, há poucas implementações, bibliotecas e interfaces para mais alto nível.

Operações SIMD semelhantes a OpenMP, agrupando os dados em workgroups. Separa a memória em privada (elemento), local (workgroup), global e do host.

Alguns dispositivos (CPUs) também têm paralelismo de tarefas.

OpenCL- algumas implementações

OpenCL C (padrão aberto)

- Linguagem para programar os *kernels*.
- Subconjunto do C, adicionado de funções para transporte de dados entre memórias e controle dos ramos.

Suportado em todas as GPUs NVIDIA com CUDA.
Suportado pelas GPUs com ATI Stream.

HMPP Workbench – CAPS

- Compilador para programas híbridos em C, Fortran
- Opera com CUDA, OpenCL, e CPUs de múltiplos núcleos.
- Através de compiladores Intel, gcc, gfortran, PGI, Sun, Open64.

Libra – GPU Systems

- Compilador híbrido (C++), para OpenCL, OpenGL, CUDA, CPUs.

Interfaces para Python (Clyther, PyOpenCL), Ruby, .NET.

ViennaCL

- Implementação aberta de BLAS, interface para C++ e MATLAB

CUDA

Desenvolvida desde 2007, em conjunto com o hardware NVIDIA.

Precisão simples e dupla (alguns modelos), inteiros.

Possui memória compartilhada entre os *threads*.

Otimizada para o hardware.

Usada pelas GPUs GeForce (consumidor), Quadro (*workstations*), Tesla (específicas para computação).

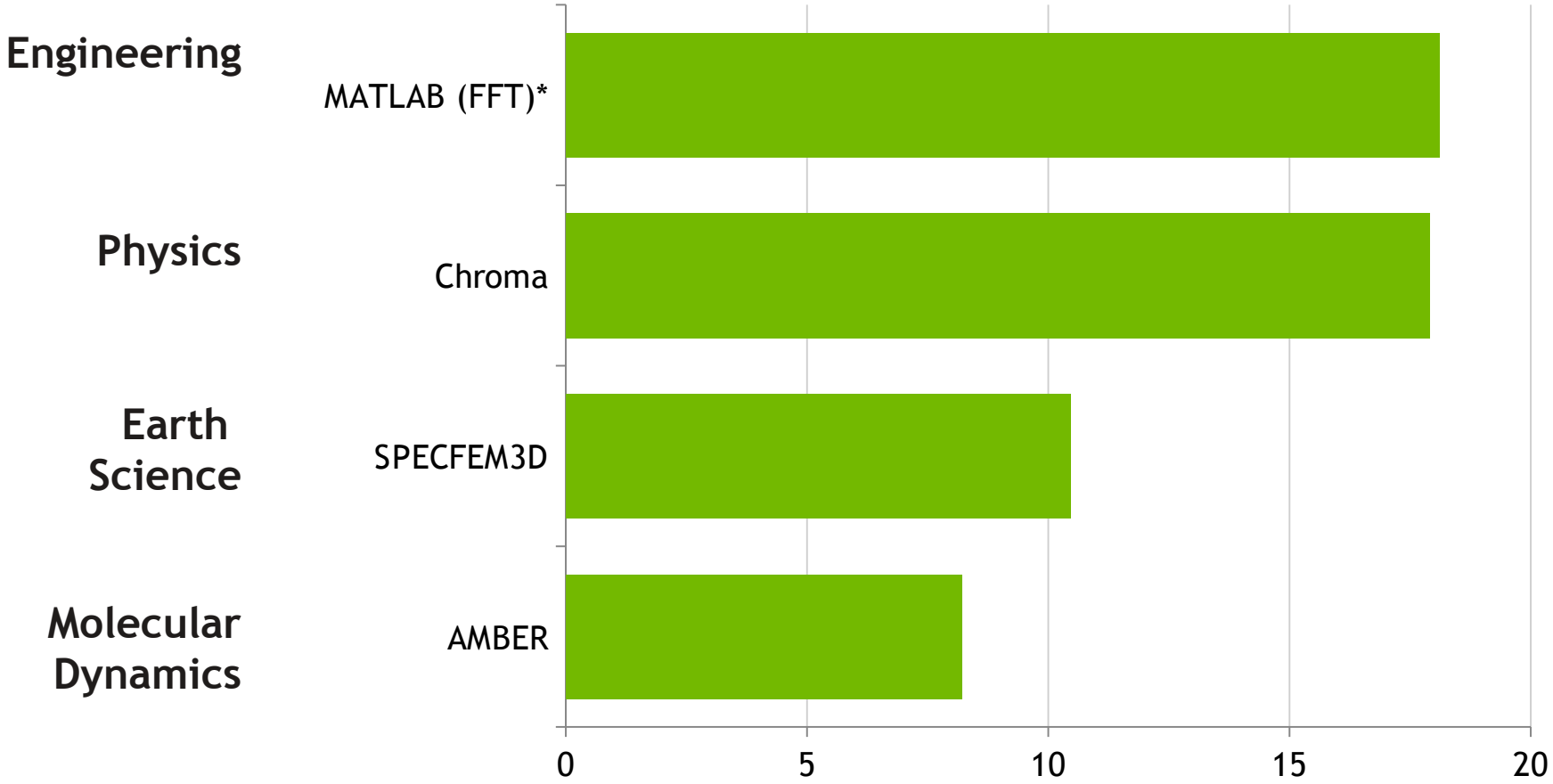
Rápida quando usa a própria memória – gargalo fica na transferência entre CPU e GPU.

Tem a maior variedade de bibliotecas e APIs.

CUDA

Aceleração (*speedup*):

NVIDIA Tesla K20 / Intel Xeon E5 2687w *



* Gráfico preparado pela NVIDIA

CUDA – implementações / bibliotecas

CUDA SDK – NVIDIA (gratuita):

- Compilador (Open64), debugger e profiler para *kernels* (C/C++)
- Suporte a OpenCL
- Bibliotecas BLAS e FFT (C, com interfaces para Fortran).
- IDE NSight (Eclipse / Visual Studio)

CUDA NPP – NVIDIA (gratuita):

- Biblioteca para processamento de imagens (C).

PyCUDA (gratuita)

- Interface aberta para escrever *kernels* com Python+Numpy.

Anaconda Accelerate (gratuito para uso acadêmico)

- Compilador de Python+Numpy para CUDA

Jacket – Accelereyes

- Interface e bibliotecas para MATLAB.

JCUDA, CUDA.NET – Hoopoe (gratuitas)

- Interface e bibliotecas para Java e .NET.

Jcuda, Jcublas, Jcufft (gratuitas)

- Interfaces abertas para CUDA e bibliotecas para Java.

CUDA – implementações / bibliotecas

CULAtools – EM Photonics
- ~LAPACK híbrido, C,C++.

CUDA Fortran PGI
- Compilador para *kernels* escritos em Fortran, C, C++.

GPULib – Tech X Corporation (gratuita para uso acadêmico)
- Interface e biblioteca para operações com *arrays* (IDL, MATLAB).

Fortran CUDA – Hoopoe (gratuito, mas abandonado)
- Compilador para *kernels* em Fortran.
- Biblioteca FFT.

HMPP Workbench – CAPS
- Compilador para programas híbridos C, Fortran
- Opera com CUDA, OpenCL, e CPUs de múltiplos núcleos.
- Através de compiladores Intel, gcc, gfortran, PGI, Sun, Open64.

Test Drive gratuito
- Acesso limitado a *clusters* com Tesla K40.
- http://www.nvidia.com/object/gpu-test-drive.html

CUDA – implementações / bibliotecas

MAGMA – Univ. Tennessee, et al. (gratuita)
- Biblioteca híbrida (CPU+GPU) BLAS e LAPACK (C).

Ecolib – GPU Computing
- BLAS, LAPACK, números aleatórios e VaR (C++).

R+GPU (gratuita)
- Funções de R reimplementadas usando CUDA.

Libra – GPU Systems
- Compilador híbrido (C++), para OpenCL, OpenGL, CUDA, CPUs.

Flagon
- API, CUFFT, CUBLAS abertas para Fortran.

F2C-ACC – NOAA
- Tradutor aberto de Fortran para C+CUDA (ainda não completo).

Thrust
- Biblioteca aberta para funcionalidade semelhante à C++ STL.

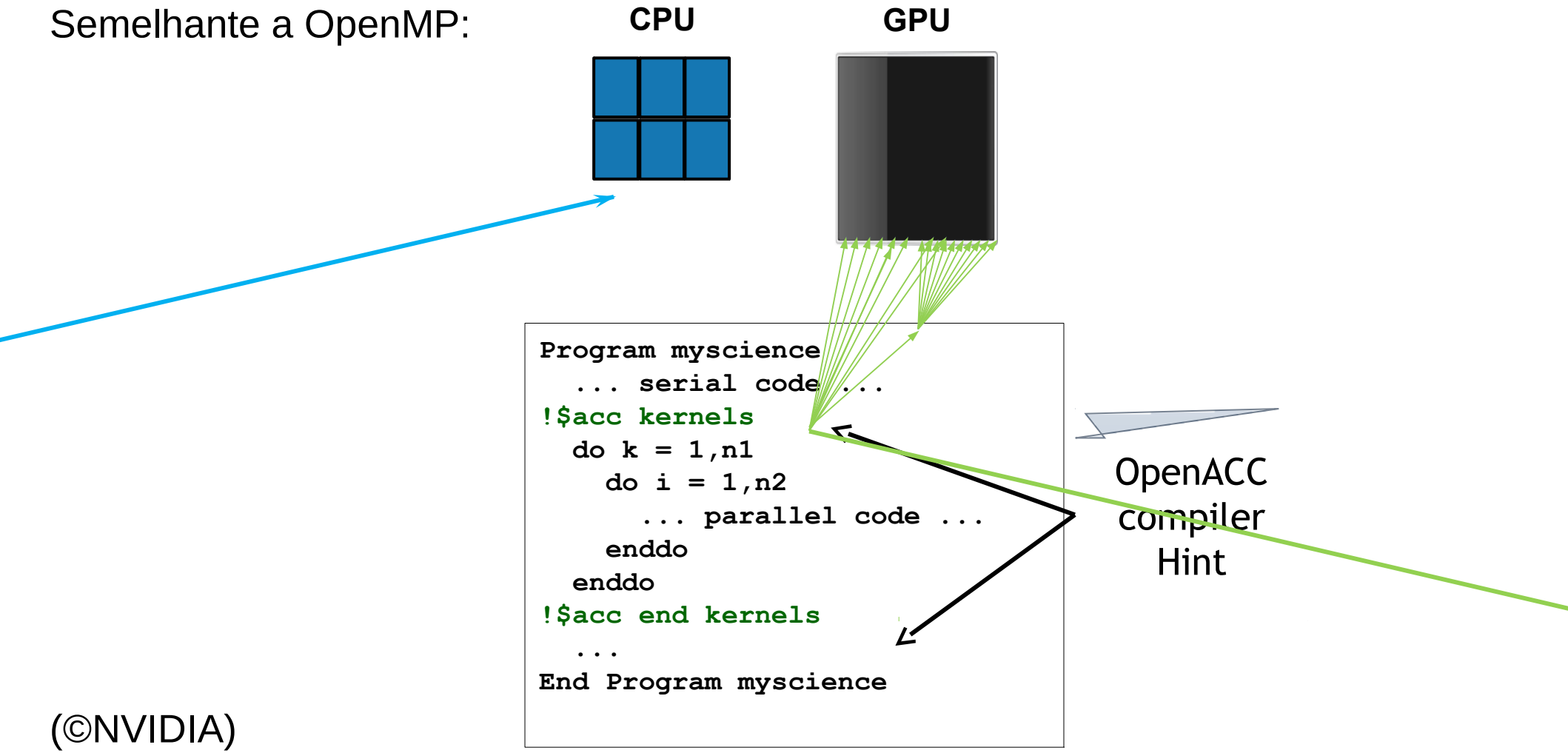
CUDA – OpenACC

Atualmente, exige o compilador PGI.

Outras implementações estão sendo desenvolvidas (gcc inclusive).

Diretrizes de compilação indicam paralelização a ser feita pelo compilador.

Semelhante a OpenMP:



(©NVIDIA)

CUDA – Como escrever um *kernel*?

CUDA C, C++ e Fortran:

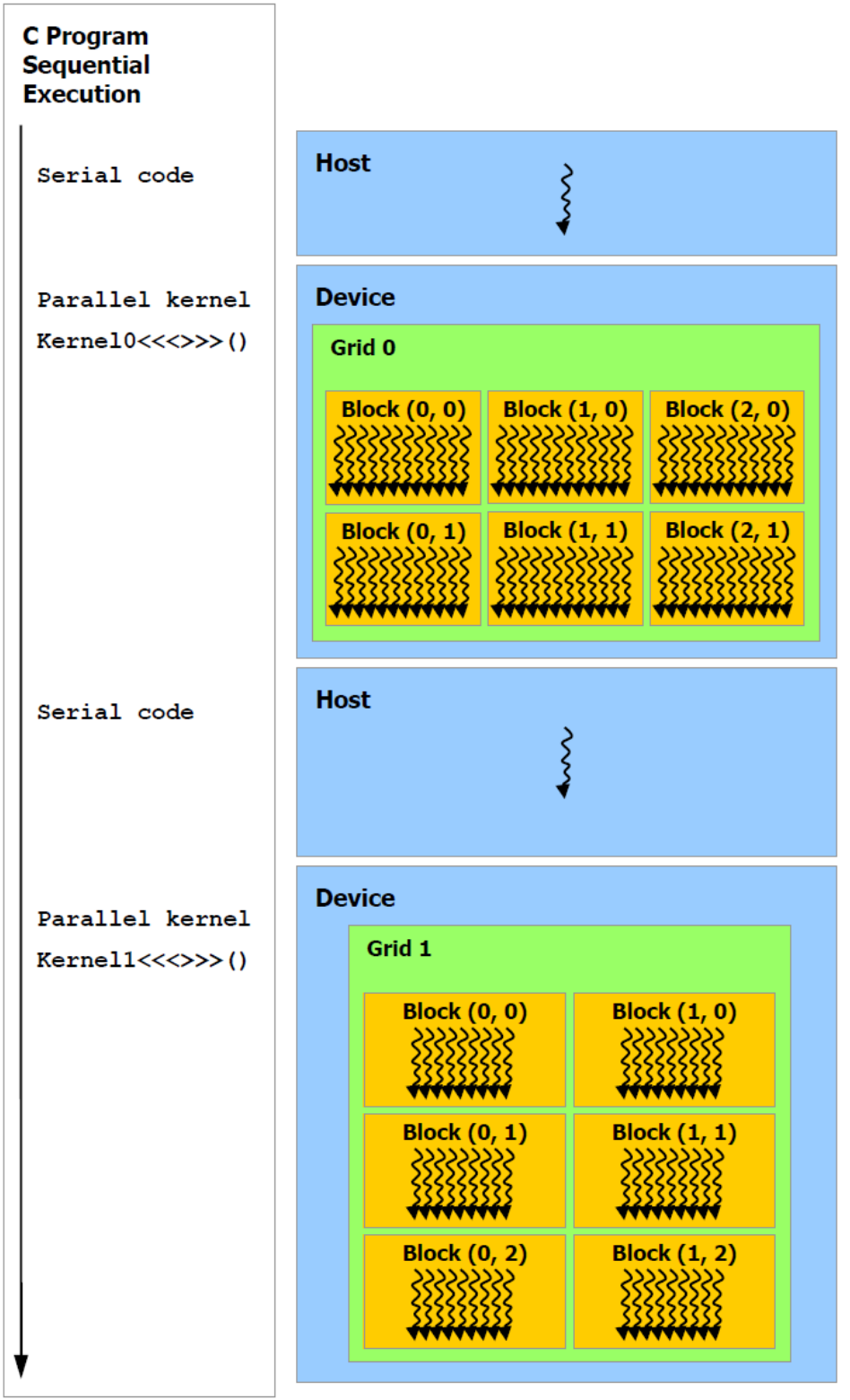
- Há limitações sobre o que o código de um *kernel* pode fazer (altamente dependentes da versão do *hardware*, do *driver* e do compilador).
- As linguagens para escrever os *kernels* funcionam aproximadamente como C, C++, Fortran padrão, mas com restrições.

Principais diferenças ao programar para GPU:

- O programa principal é executado pela CPU. Apenas algumas funções (*kernels*) serão executados pela GPU.
- Não há mistura: uma função é executada inteiramente na CPU (*host*) ou inteiramente na GPU (*device*).
- Não há mistura de memória: CPU e GPU têm RAMs separadas; uma não vê a memória da outra.
- **CUDA faz apenas paralelismo de dados**: o mesmo *kernel* é executado por todos os núcleos da GPU ao mesmo tempo; cada núcleo executa um *thread*.
- Só um programa acessa a GPU de cada vez.

CUDA – Exemplo

Estrutura típica de um programa CUDA



(©NVIDIA)

CUDA – Exemplo (C)

Kernel para adição de dois vetores, C=A+B:

```
// Kernel definition
 __global__ void VecAdd(float* A, float* B, float* C)
{
  int i = threadIdx.x;
  C[i] = A[i] + B[i];
}
```

ThreadIdx identifica cada *thread*, usado aqui para separar que elementos cada um processa. Pode ser 1D, 2D ou 3D.

Threads podem ser agrupados em blocos, 1D ou 2D. Blocos têm memória compartilhada local.

Programa que usa o *kernel*:

```
int main()
{
   ...
// Kernel invocation with N threads
  VecAdd<<<1, N>>>(A, B, C);
}
```

CUDA – Exemplo (C)

```
// Kernel that executes on the CUDA device
__global__ void square_array(float *a, int N)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx<N) a[idx] = a[idx] * a[idx];
}

// main routine that executes on the host
int main(void)
{
  float *a_h, *a_d;  // Pointer to host & device arrays
  const int N = 10;  // Number of elements in arrays
  size_t size = N * sizeof(float);
  a_h = (float *)malloc(size);        // Allocate array on host
  cudaMalloc((void **) &a_d, size);   // Allocate array on device
  // Initialize host array and copy it to CUDA device
  for (int i=0; i<N; i++) a_h[i] = (float)i;
  cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
  // Do calculation on device:
  int block_size = 4;
  int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);
  square_array <<< n_blocks, block_size >>> (a_d, N);
  // Retrieve result from device and store it in host array
  cudaMemcpy(a_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
  // Print results
  for (int i=0; i<N; i++) printf("%d %f\n", i, a_h[i]);
  // Cleanup
  free(a_h); cudaFree(a_d);
}
```

CUDA – Exemplo (Fortran)

Programa que usa o *kernel*:

```
   subroutine mmul( A, B, C )
!
   use cudafor
   real, dimension(:,:) :: A, B, C
   integer :: N, M, L
   real, device, allocatable, dimension(:,:) :: Adev,Bdev,Cdev
   type(dim3) :: dimGrid, dimBlock
!
   N = size(A,1) ; M = size(A,2) ; L = size(B,2)
   allocate( Adev(N,M), Bdev(M,L), Cdev(N,L) )
   Adev = A(1:N,1:M)
   Bdev = B(1:M,1:L)
   dimGrid = dim3( N/16, L/16, 1 )
   dimBlock = dim3( 16, 16, 1 )
   call mmul_kernel<<<dimGrid,dimBlock>>>( Adev,Bdev,Cdev,N,M,L )
   C(1:N,1:M) = Cdev
   deallocate( Adev, Bdev, Cdev )
!
   end subroutine
```

CUDA – Exemplo (Fortran) - *kernel*

```
   attributes(global) subroutine MMUL_KERNEL( A,B,C,N,M,L)
!
   real,device :: A(N,M),B(M,L),C(N,L)
   integer,value :: N,M,L
   integer :: i,j,kb,k,tx,ty
   real,shared :: Ab(16,16), Bb(16,16)
   real :: Cij
!
   tx = threadidx%x ; ty = threadidx%y
   i = (blockidx%x-1) * 16 + tx
   j = (blockidx%y-1) * 16 + ty
   Cij = 0.0
   do kb = 1, M, 16
      ! Fetch one element each into Ab and Bb;  note that 16x16 = 256
      ! threads in this thread-block are fetching separate elements
      ! of Ab and Bb
      Ab(tx,ty) = A(i,kb+ty-1)
      Bb(tx,ty) = B(kb+tx-1,j)
      ! Wait until all elements of Ab and Bb are filled
      call syncthreads()
      do k = 1, 16
         Cij = Cij + Ab(tx,k) * Bb(k,ty)
      enddo
      ! Wait until all threads in the thread-block finish with
      ! this iteration's Ab and Bb
      call syncthreads()
   enddo
   C(i,j) = Cij
!
   end subroutine
```

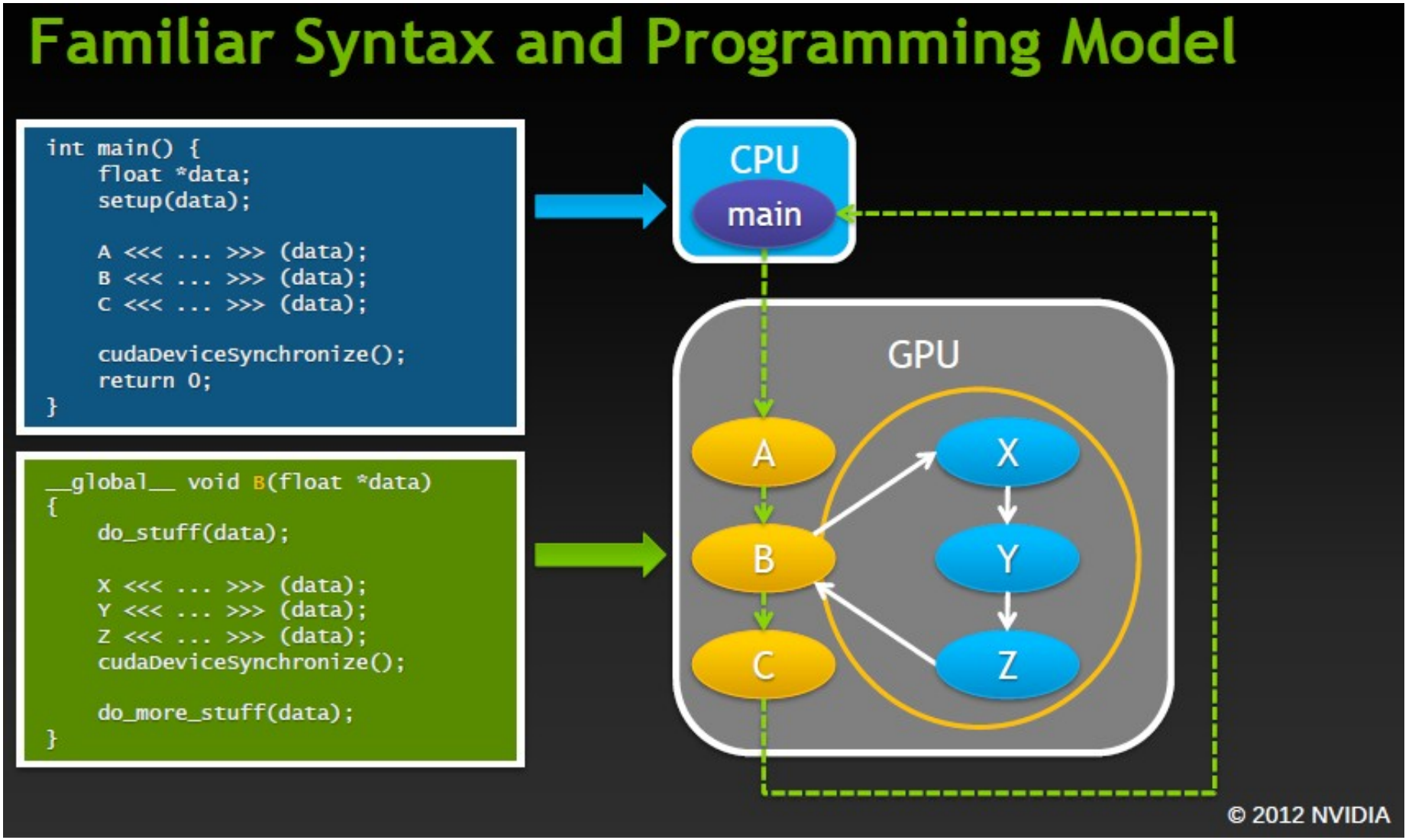
CUDA – Novidades

Evolução muito rápida, diminuindo muito as limitações de programar em CUDA:

Algumas novidades não exigem novas GPUs, apenas troca de *driver* e *toolkit*.

CUDA – Novidades

- CUDA 5.0 (2012):
  - Paralelismo dinâmico: permite que *threads* iniciem outros *threads.*
  - Permite *printf* de dentro dos *threads*.
  - NSight IDE (incluindo *debugger*) para todas as plataformas.



CUDA – Novidades

- CUDA 6.0 (2013):

Memória unificada: dentro do código, parece que CPU e GPU enxergam as mesmas variáveis – a transferência entre memórias é feita automaticamente pelo CUDA.

CPU Code

```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  fread(data, 1, N, fp);

  qsort(data, N, 1, compare);


  use_data(data);

  free(data);
}
```

CUDA 6 Code with Unified Memory

```
void sortfile(FILE *fp, int N) {
  char *data;
  cudaMallocManaged(&data, N);

  fread(data, 1, N, fp);

  qsort<<<...>>>(data,N,1,compare);
  cudaDeviceSynchronize();

  use_data(data);

  cudaFree(data);
}
```

(©NVIDIA)

Algumas referências - CUDA

- *CUDA Training – Apresentações, vídeos, cursos, tutoriais, exemplos*
  https://developer.nvidia.com/cuda-training

- *CUDA Fortran for Scientists and Engineers (2013)*
  Ruetsch e Fatica
  https://developer.nvidia.com/content/cuda-fortran-scientists-and-engineers

- *CUDA by Example: An introduction to general-purpose GPU programming (2011)*
  Sanders e Kandrot
  https://developer.nvidia.com/content/cuda-example-introduction-general-purpose-gpu-programming-0

- *Programming Massively Parallel Processors (2010)*
  Hwu e Kirk
  https://developer.nvidia.com/content/programming-massively-parallel-processors-hands-approach

- *GPU Gems (2007)*
  https://developer.nvidia.com/content/gpu-gems

- *CUDA Zone*
  https://developer.nvidia.com/cuda-zone

- *GPU Test Drive*
  https://developer.nvidia.com/cuda-zone

- *CUDA Tools & Ecosystem*
  http://developer.nvidia.com/cuda-tools-ecosystem

Algumas referências - MPI

- *Message Passing Interface Forum*
  http://www.mpi-forum.org/

- Especificação
  http://www.mpi-forum.org/docs/
  http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf
  http://www.mpi-forum.org/docs/mpi22-report/mpi22-report.htm

- Boost.MPI
  http://www.boost.org/doc/libs/1_46_1/doc/html/mpi.html

- *Programming MPI with C++*
  http://www.rz.rwth-aachen.de/global/show_document.asp?id=aaaaaaaaaabluvs

- *Introduction to MPI-IO*
  (coordenção do acesso ao mesmo arquivo por todos os processos; alternativa à opção comum de apenas um processo acessar o arquivo, e mandar / receber os dados por MPI).
  www.lrde.epita.fr/~ricou/mpi-io.ppt

- Open MPI
  http://www.open-mpi.org/

- MPICH2
  http://www.mcs.anl.gov/research/projects/mpich2/

- Introdução ao MPI (CENAPAD-Unicamp)
  http://www.cenapad.unicamp.br/servicos/treinamentos/mpi.shtml

Algumas referências

Gerais (incluindo OpenMP, MPI e outros)

- *Parallel Programming: for Multicore and Cluster Systems* (2010)
  Rauber e Rünger
  http://www.amazon.com/Parallel-Programming-Multicore-Cluster-Systems/dp/364204817X/

- *An Introduction to Parallel Programming* (2011)
  Peter Pacheco
  http://www.amazon.com/Introduction-Parallel-Programming-Peter-Pacheco/dp/0123742609/

- *An Introduction to Parallel Programming with OpenMP, PThreads and MPI* (2011)
  Robert Cook
  http://www.amazon.com/Introduction-Parallel-Programming-PThreads-ebook/dp/B004I6D6BM/

- *Introduction to High Performance Computing for Scientists and Engineers* (2010)
  Hager e Wellein
  http://www.amazon.com/Introduction-Performance-Computing-Scientists-Computational/dp/143981192X/

Algumas referências - OpenMP

- Site oficial
  http://openmp.org/
- *OpenMP Forum*
  http://openmp.org/forum/
- The Community of OpenMP Users, Researchers, Tool Developers and Providers
  http://www.compunity.org/
- *Using OpenMP: Portable Shared Memory Parallel Programming* (2007, só até OMP 2.5)
  Chapman, Jost, van der Pas
  http://www.amazon.com/Using-OpenMP-Programming-Engineering-Computation/dp/0262533022/
- *C++ and OpenMP*
  http://www.compunity.org/events/pastevents/parco07/parco_cpp_openmp.pdf
- *OpenMP C and C++ Application Program Interface*
  http://www.openmp.org/mp-documents/cspec20.pdf
- *Parallel Program in Fortran 95 Using OpenMP*
  http://www.openmp.org/presentations/miguel/F95_OpenMPv1_v2.pdf
- *Complete Specifications - (May, 2008)*
  http://www.openmp.org/mp-documents/spec30.pdf
- *Version 3.0 Summary Card C/C++*
  http://www.openmp.org/mp-documents/OpenMP3.0-SummarySpec.pdf
- *Version 3.0 Summary Card Fortran*
  http://www.openmp.org/mp-documents/OpenMP3.0-FortranCard.pdf

Algumas referências

Gerais (incluindo OpenMP, MPI e outros)

- *Parallel Programming: for Multicore and Cluster Systems* (2010)
  Rauber e Rünger
  http://www.amazon.com/Parallel-Programming-Multicore-Cluster-Systems/dp/364204817X/

- *An Introduction to Parallel Programming* (2011)
  Peter Pacheco
  http://www.amazon.com/Introduction-Parallel-Programming-Peter-Pacheco/dp/0123742609/

- *An Introduction to Parallel Programming with OpenMP, PThreads and MPI* (2011)
  Robert Cook
  http://www.amazon.com/Introduction-Parallel-Programming-PThreads-ebook/dp/B004I6D6BM/

- *Introduction to High Performance Computing for Scientists and Engineers* (2010)
  Hager e Wellein
  http://www.amazon.com/Introduction-Performance-Computing-Scientists-Computational/dp/143981192X/