

# Paralelização

## Introdução a vetorização, OpenMP e MPI

### 2 – OpenMP, MPI e escolhas

Paulo Penteado  
IAG / USP

[pp.penteado@gmail.com](mailto:pp.penteado@gmail.com)

Esta apresentação: [http://www.penteado.net/papers/iwcca/pp\\_para\\_1.pdf](http://www.penteado.net/papers/iwcca/pp_para_1.pdf)

Aquivos do curso: <http://www.penteado.net/papers/iwcca/>

Artigo do curso: [http://www.penteado.net/papers/iwcca/iwcca\\_pfp.pdf](http://www.penteado.net/papers/iwcca/iwcca_pfp.pdf)

# Programa

## 1 – Conceitos e vetorização

- Motivação
- Formas de paralelização
  - Paralelismo de dados
  - Paralelismo de tarefas
- Principais arquiteturas paralelas atuais
  - Com recursos compartilhados
  - Independentes
- Paradigmas discutidos neste curso:
  - Vetorização
  - OpenMP
  - MPI
- Vetorização
- Algumas referências

Slides em [http://www.ppenteado.net/ast/pp\\_para\\_1.pdf](http://www.ppenteado.net/ast/pp_para_1.pdf)

## 2 – OpenMP, MPI, escolhas

- OpenMP
- MPI
- Escolha de forma de paralelização
- Algumas referências

Slides em [http://www.ppenteado.net/ast/pp\\_para\\_2.pdf](http://www.ppenteado.net/ast/pp_para_2.pdf)

Exemplos em  
Artigo do curso

[http://www.ppenteado.net/ast/pp\\_para/](http://www.ppenteado.net/ast/pp_para/)  
[http://www.ppenteado.net/ast/pp\\_para/iwcca\\_pfp.pdf](http://www.ppenteado.net/ast/pp_para/iwcca_pfp.pdf)

# 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.
- Geralmente muito 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).

# OpenMP - Características

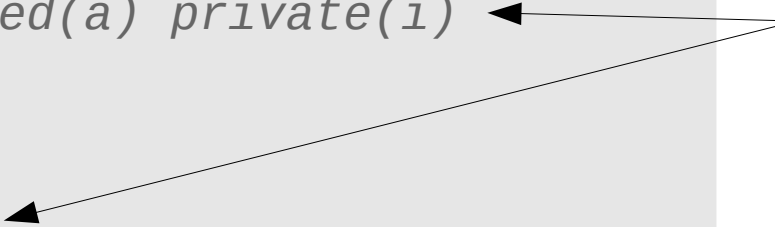
Implementado através de **diretrizes de compilação** (instruções que serão executadas pelo compilador, não pelo programa produzido):

- **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
```

Diretrizes (comentários em Fortran padrão)



# OpenMP - Características

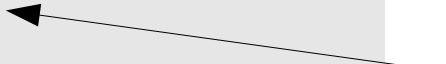
Implementado através de **diretrizes de compilação** (instruções que serão executadas pelo compilador, não pelo programa produzido):

- **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;  
}
```

Diretriz (pragma)



# OpenMP - Características

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

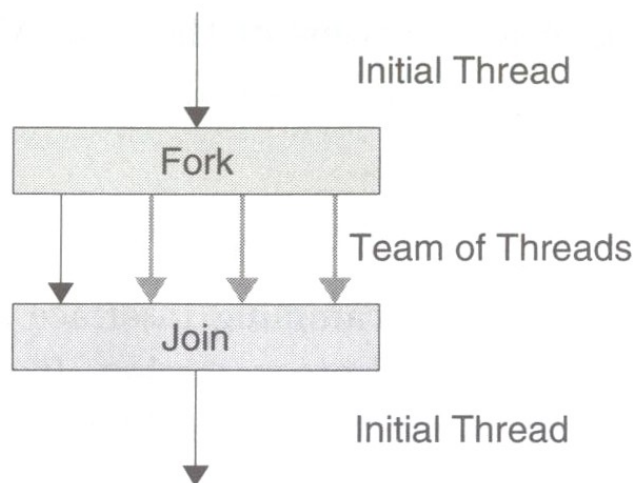


Figure 2.1: **The fork-join programming model supported by OpenMP** – The program starts as a single thread of execution, the initial thread. A team of threads is forked at the beginning of a parallel region and joined at the end.

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*):

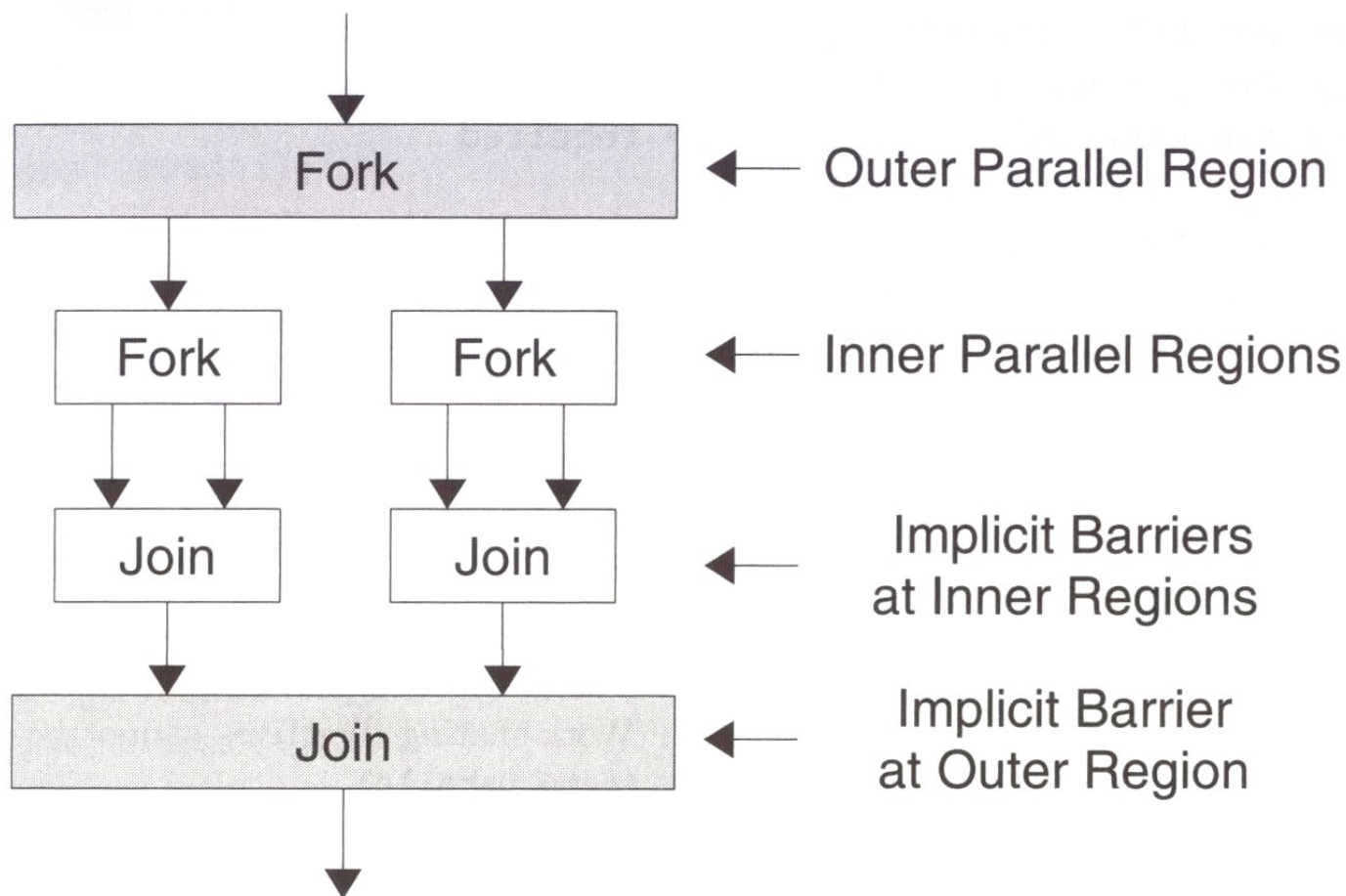


Figure 6.20: **Nested OpenMP parallel regions** – Inner parallel regions introduce implicit synchronization points in addition to the barrier synchronization points at the outer parallel regions.



# 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

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

C++

```
#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:

Sentinelas (*sentinels*):

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
```

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:

```
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;  
    }  
}
```

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

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

Fazendo uso do número do thread. Ex:

```
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` especifica 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)

```
#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):

```
#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=10000000,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):

```
#include <stdio.h>
#ifdef _OPENMP
    #include<omp.h>
#endif
int main() {
    /*Example of a race condition*/
    int n=10000000,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:

```
#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)

```
/*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)

```
/*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:

### **Síncronas:**

- 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.

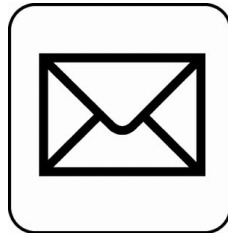
De uma forma análoga, global, as comunicações podem ser vistas como:

**Síncronas:**



- 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 específico 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"*/
```

# 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



555-3210



Discando 555-3210

Discando 555-0123

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



# Escolhas

**Não existe uma “melhor solução”.**

Algumas são mais adequadas a algumas situações.

**Muitos problemas podem combinar vários paradigmas (soluções híbridas).**

- Ex: um programa MPI onde cada processo use OpenMP e vetorização.

**Há também paralelização indireta:**

- O programador usa bibliotecas paralelizadas prontas.
- As bibliotecas podem ter sido implementadas por especialistas (cientistas da computação) para fazer seu trabalho da forma mais eficiente possível.
- Casos comuns de estarem implementados:
  - Álgebra linear
  - FFT
  - Operações de processamento de imagens
  - Algoritmos comuns (ordenação, contêiners, etc.)
  - Problemas científicos comuns: visualização, CFD, vizinhos (KNN), etc.

# Escolhas - comparação

## Vetorização

- Vantagens:
  - Paralelização, se disponível, feita pelo compilador / interpretador.
  - Torna o programa mais organizado e robusto.
  - Permite identificar trechos que podem ser paralelizados de outras formas, explicitamente.
  - Muito mais fácil que as outras formas.
  - Compiladores podem gerar código vetorial para um só núcleo (MMX, SSE, AVX), ou para vários (em vários threads).
- Desvantagens:
  - Em geral limitada a memória compartilhada.
  - Limitada a paralelismo de dados.
  - Só se aplica aos casos mais simples de paralelização.
  - Abrangência muito variável entre linguagens.

# Escolhas - comparação

## OpenMP

- Vantagens:
  - O padrão melhor estabelecido para paralelização.
  - Padrão ainda ativamente desenvolvido.
  - Independente de linguagem, compilador e plataforma.
  - Permite paralelismo de dados e tarefas.
  - Possui tanto construções de baixo nível como de alto nível.
  - Mantém compatibilidade com código serial.
  - Mais fácil que MPI.
- Desvantagens:
  - Limitado a memória compartilhada.
  - Só implementado em Fortran, C, C++.

# Escolhas - comparação

## MPI

- Vantagens:
  - O padrão melhor estabelecido para memória distribuída.
  - Pode ser usado tanto em sistemas de memória compartilhada como de memória distribuída.
  - O padrão mais comum para clusters.
  - Padrão ainda ativamente desenvolvido.
  - Permite paralelismo de dados e tarefas.
  - Implementado em várias linguagens (Fortran, C, C++, Python, Java, IDL, etc.).
- Desvantagens:
  - Mais difícil de usar que OpenMP e vetorização.
  - Sem usar Boost.MPI, desajeitado para transmitir dados mais complexos que arrays de tipos simples.
  - Exige reescrever (possivelmente reestruturar) o código serial.
  - Padrão ainda menos maduro que OpenMP.
  - Ainda varia bastante entre implementações, principalmente na execução (`mpirun`, `mpiexec`).

# 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](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](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 - 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](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=aaaaaaaaaablusv](http://www.rz.rwth-aachen.de/global/show_document.asp?id=aaaaaaaaaablusv)
- *Introduction to MPI-IO*  
(coordenação 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](http://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 e MPI e outros)

- *Parallel Programming: for Multicore and Cluster Systems* (2010)  
Rauber e Rüniger  
<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/>

# Programa

## 1 – Conceitos e vetorização

- Motivação
- Formas de paralelização
  - Paralelismo de dados
  - Paralelismo de tarefas
- Principais arquiteturas paralelas atuais
  - Com recursos compartilhados
  - Independentes
- Paradigmas discutidos neste curso:
  - Vetorização
  - OpenMP
  - MPI
- Vetorização
- Algumas referências

## 2 – OpenMP, MPI, escolhas

- OpenMP
- MPI
- Escolha de forma de paralelização
- Algumas referências

Esta apresentação: [http://www.ppenteadonet/ast/pp\\_para/pp\\_para\\_2.pdf](http://www.ppenteadonet/ast/pp_para/pp_para_2.pdf)

Aquivos do curso: [http://www.ppenteadonet/ast/pp\\_para/](http://www.ppenteadonet/ast/pp_para/)

Artigo do curso: [http://www.ppenteadonet/ast/pp\\_para/iwcca\\_pfp.pdf](http://www.ppenteadonet/ast/pp_para/iwcca_pfp.pdf)

[pp.penteadon@gmail.com](mailto:pp.penteadon@gmail.com)