ERAD SP 2013

IV Escola Regional de
Alto Desempenho de São Paulo
São Carlos / SP

Minicurso

Programando para múltiplos processadores: Pthreads, OpenMP e MPI

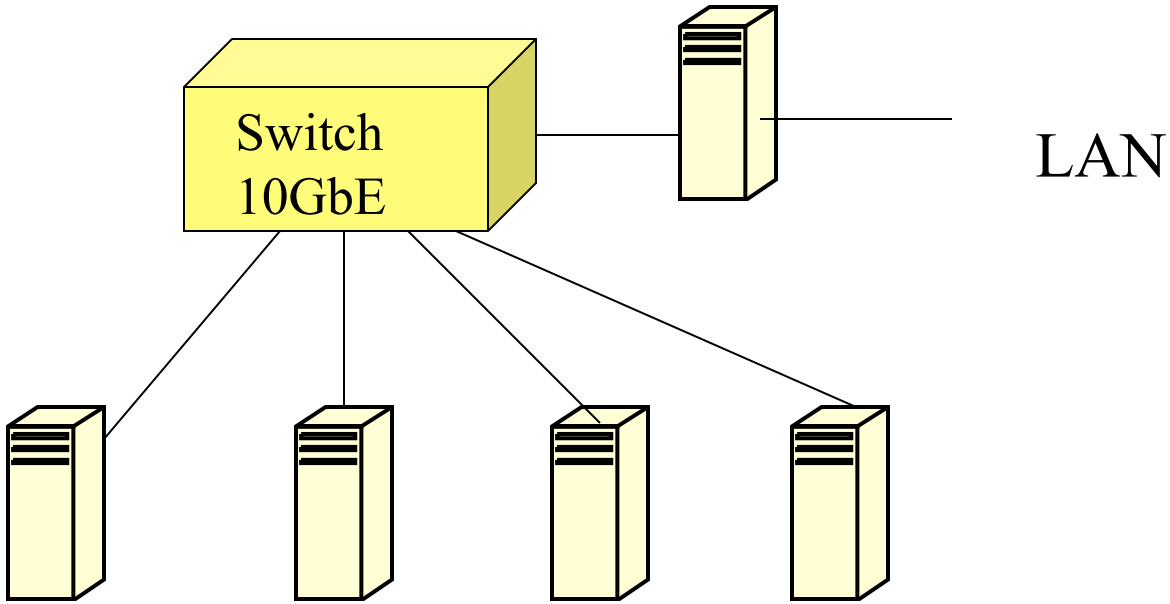
Liria Matsumoto Sato (EP/USP) e
Helio Crestana Guardia (DC/UFSCar)

Sistemas Distribuídos

- Introdução
- Modelos de computação
- Programação utilizando MPI
- Programação Híbrida: MPI e OpenMP
- Considerações sobre desempenho
- Conclusão

Arquiteturas Paralelas

Cluster



Programação em Sistemas Distribuídos

- Bibliotecas:
  - MPI (message passage interface)
  - programação utilizando processos
  - Comunicação entre processos por passagem de mensagem: send e receive
- Programação mais complexa comparada à programação com variáveis compartilhadas

Programação em Sistemas Distribuídos

Sistemas DSM (distributed shared memory): simula memória compartilhada, permitindo a programação no paradigma de variáveis compartilhadas

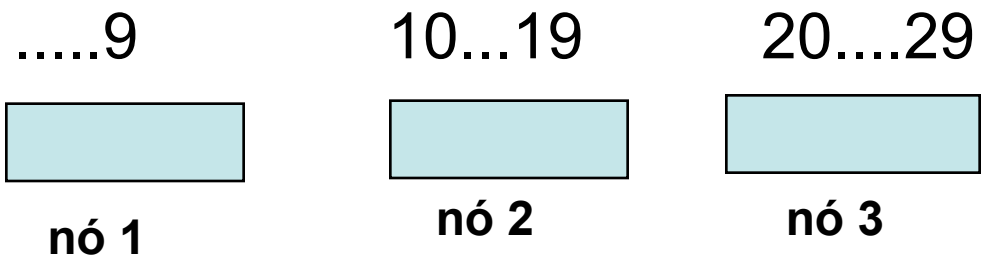
Bibliotecas

- PVM (Parallel Virtual Machine)
- MPI (message passage interface): interface que buscou a padronização
  - programação baseada em processos
  - funções de comunicação entre processos por passagem de mensagem: bloqueantes, não bloqueantes e coletivas
  - Diversas implementações

Modelos de Computação

spmd (single program multiple data)

•todos os nós executam o mesmo programa sobre dados múltiplos

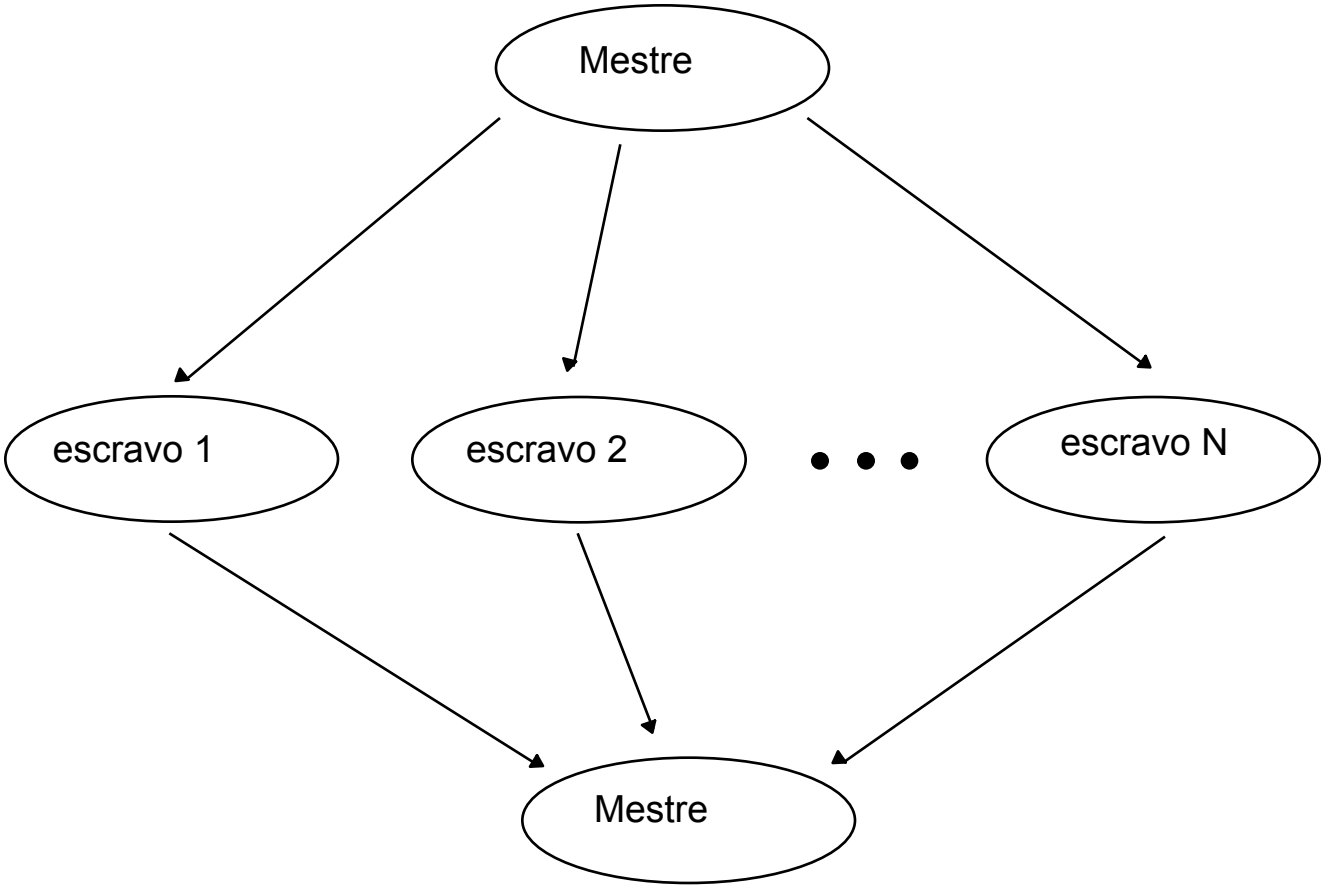


Modelos de Computação

esquema mestre escravo com spmd

- todos executam o mesmo programa
- através de um controle interno ao programa um nó executa a função mestre e os demais são escravos.

Modelos de Computação



MPI

Diversas Implementações

Exemplo:

OPEN MPI

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

OPENMPI

Compilação:

mpiCC –o trivial trivial.c (ou trivial.cpp)

execução: **mpirun -np 2 - -hostfile hosts trivial**

Arquivo contendo especificação dos hosts

(exemplo: hosts)

192.168.10.1

192.168.1.2

192.168.1.3

Computador local: não é necessário especificar hosts

mpirun –np 4 trivial

MPI

- Processos: são representados por um único "rank"(inteiro) e ranks são numerados 0, 1, 2 ..., N-1. (N = total de processos)
- Enter e Exit

MPI_Init(int *argc,char *argv);

MPI_Finalize(void);

- Quem eu sou?

MPI_Comm_rank(MPI_Comm comm,int *rank);

- informa total de processos

MPI_Comm_size(MPI_Comm comm,int *size);

Programa MPI - SPMD

```
#include <mpi.h>
........

main(argc, argv)
int                 argc;
char                        *argv[];
{
    int             size, rank;
    MPI_Status      status;
     .......

 // Initialize MPI.
    MPI_Init(&argc, &argv);
     .........
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     ......
```

Programa MPI - SPMD

```
if (0 == rank)              //  rank = 0
     {
          ..........

     }
    else                    // rank > 0
    {
          ..........

    }

    MPI_Finalize();
    return(0);
}
```

MPI: envio e recepção de mensagens

- Enviando Mensagens

MPI_Send(void *buf,int count,MPI_Datatype dtype,int dest,int tag,MPI_Comm comm);

- Recebendo Mensagens

MPI_Recv(void *buf,int count,MPI_Datatype dtype,int source,int tag,MPI_Comm comm,MPI_Status *status);

status: status.MPI_TAG

status.MPI_SOURCE

Exemplo: enviando mensagens

```
//  rank 0, send a message to rank 1.
    if (0 == rank) {
          for (i=0;i<64;i++)
            buf[i]=i;
       MPI_Send(buf, BUFSIZE, MPI_INT, 1, 11, MPI_COMM_WORLD);
    }

  //  rank 1, receive a message from rank 0.
    else {
          MPI_Recv(buf, BUFSIZE, MPI_INT, 0, 11, MPI_COMM_WORLD,
                                  &status);
            for(i=0;i<64;i++)
           {   printf("%d",buf[i]);
           fflush(stdout);}
            printf("\n");
    }
```

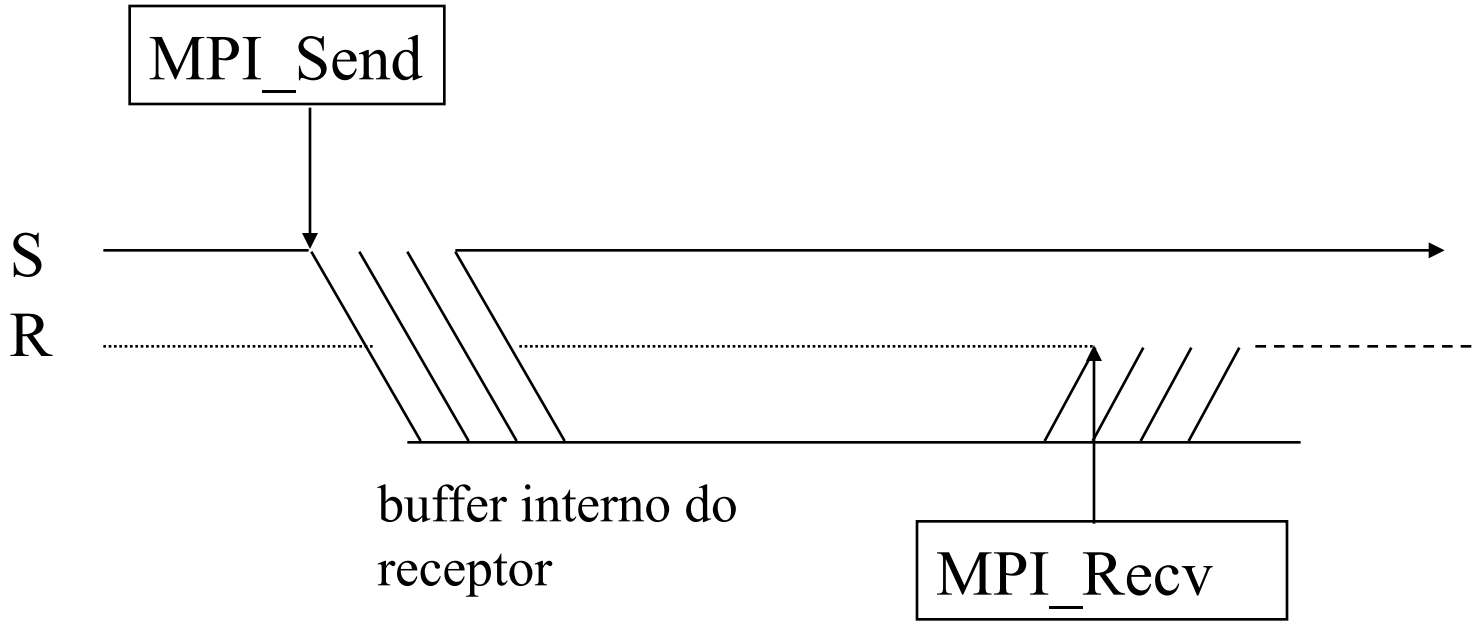
Comunicação

- Comunicação ponto a ponto: uma origem e um destino
  - Envio bloqueante
  - Recepção bloqueante
  - Envio não bloqueante
  - Recepção não bloqueante

- Comunicação Coletiva

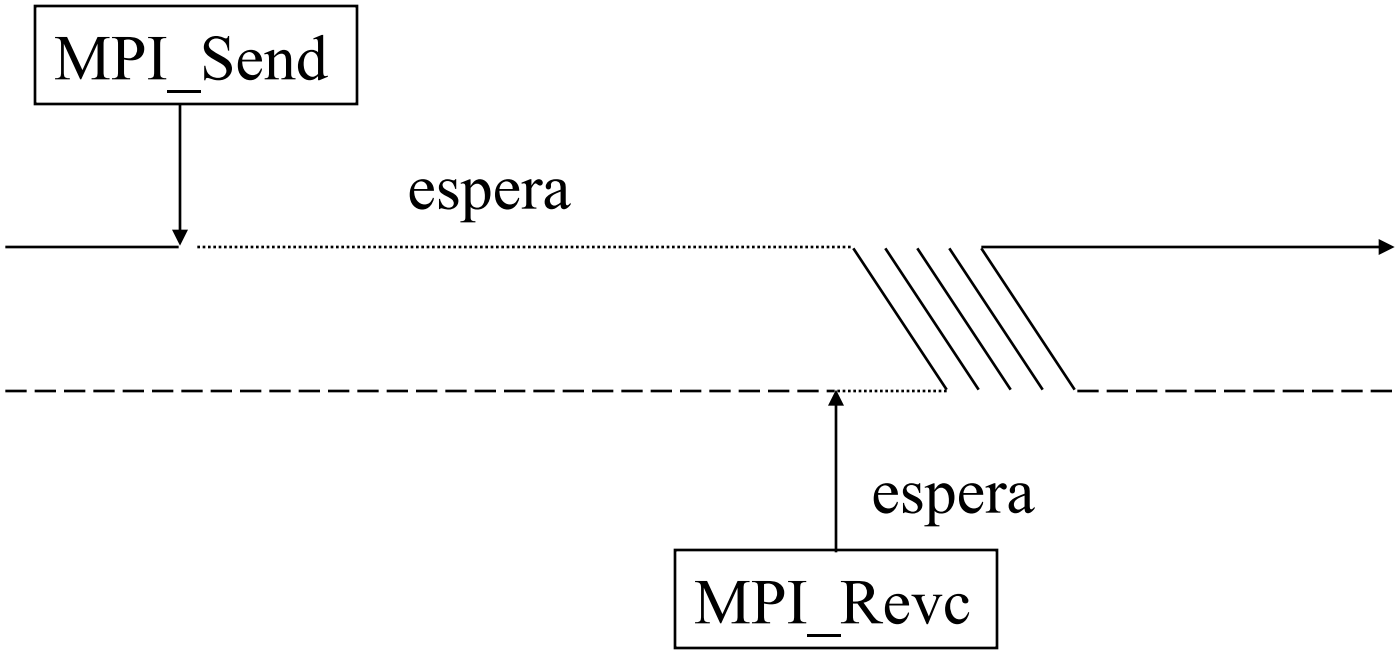
Envio Padrão Bloqueante

- O comportamento do sistema depende do tamanho da mensagem, se é menor ou igual ou maior do que um limite. Este limite é definido pela implementação do sistema e do número de tarefas na aplicação.
- mensagem<= limite



Envio Padrão Bloqueante

- mensagem>limite



Envio Bloqueante

- **Padrão:**

MPI_Send(&buf,n_elementos,MPI_INT,dest, tag, MPI_COMM_WORLD)

- **Blocking Syncronous Send :**

MPI_Ssend (&buf,n_elementos,MPI_INT, dest, tag, MPI_COMM_WORLD)

- tarefa transmissora: envia para a tarefa receptora uma mensagem "**ready to send**".
- tarefa receptora: ao executar uma chamada receive, envia para a tarefa transmissora uma mensagem de "**ready to receive**".
- Os dados são transferidos.

Envio Bloqueante

- **Blocking Ready Send**

MPI_Rsend (&buf,n_elementos,MPI_INT, dest, tag, MPI_COMM_WORLD)

- envia a mensagem na rede.
- Requer que uma notificação de "**ready to receive**" tenha chegado.
- Notificação não recebida: erro.

Envio Bloqueante

- **Blocking Buffered Send**

MPI_Bsend(&buf,n_elementos,MPI_INT,
dest, tag, MPI_COMM_WORLD)

- Aplicação deve prover o **buffer**: array alocado estaticamente ou dinamicamente com malloc. Deve ser incluido bytes do header.
- **MPI_Buffer_attach**(...)
- Dados do buffer de mensagem copiados para buffer do usuário. Execução retorna.
- O dado será copiado do buffer do usuário sobre a rede quando uma notificação de “ready to receive”tiver chegado
- **MPI_Buffer_detach**(...)

Recepção Bloqueante

MPI_Recv(&a,10, MPI_INT, origem, tag, MPI_COMM_WORLD, &status)

- **bloqueante**: receptor fica bloqueado até receber a mensagem.
- **tag**: pode ser MPI_ANY_TAG (qualquer tag)
- **origem**: pode ser MPI_ANY_SOURCE (qualquer origem)
- **status**: 2 campos
  - status.source: origem da mensagem
  - status.tag: tag da mensagem

Programação: soma de 2 vetores

```
main(int argc, char *argv[])
{
    int n,n_nos, rank;
    MPI_Status status;
    int inicio, fim, vetor[1000], soma_parcial, i, soma_total, soma;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &n_nos);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    n=1000;   inicio=rank*(n/n_nos);   fim=inicio+(n/n_nos);
```

```
if (rank==0) {
      for(i=0;i<n;i++)
            vetor[i]=1;
     for (i=1;i<n_nos;i++)
        MPI_Send(&vetor[0]+i*n/n_nos,n/n_nos,MPI_INT,i,10,
        MPI_COMM_WORLD);
 }
// processos com rank diferente de 0
else {
      MPI_Recv(vetor,n/n_nos,MPI_INT,0,10, MPI_COMM_WORLD,
          &status);
}
```

```
// todos os processos
soma_parcial=0;
for(i=0;(i<n/n_nos);i++)
        soma_parcial+=vetor[i];
```

```
 if (rank==0) {
       soma_total=soma_parcial;
       for(i=1;i<n_nos;i++) {
           MPI_Recv(&soma,1,MPI_INT,MPI_ANY_SOURCE,11,
           MPI_COMM_WORLD,&status);
           soma_total+=soma;
       }
      printf("RESULTADO=%d\n",soma_total);
      fflush(stdout);
 } else {
      MPI_Send(&soma_parcial,1,MPI_INT,
      0,11,MPI_COMM_WORLD);
 }

 MPI_Finalize();

 return(0);

}
```

Recepção: outras funções

- **MPI_Probe(in source,int tag,MPI_Comm comm,MPI_Status *status)**

  Sincroniza uma mensagem e retorna informações. Não retorna até que uma mensagem seja sincronizada.

- **MPI_Get_count(MPI_Status *status,MPI_Data type dtype,int *count)**

  Retorna o número de elementos da mensagem recebida.

  uso: quando não se conhece o tamanho da mensagem a ser recebida. Mensagem sincronizada com MPI_Probe.

Funções de comunicação não bloqueantes

- chamadas não bloqueantes retornam imediatamente após o início da comunicação. O programador não sabe se o dado enviado já saiu do buffer de envio ou se o dado a ser recebido já chegou. Então, o programador deve verificar o seu estado antes de usar o buffer.

Comunicação não bloqueante

- **envio**: retorna após colocar o dado no buffer de envio.

MPI_Isend(void *buf,int count,MPI_Datatype dtype,int dest,int tag,MPI_Comm comm,MPI_Request *req)

req: objeto que contém informações sobre a mensagem, por exemplo o estado da mensagem.

Comunicação não bloqueante

- **Recepção**: é dado o início à operação de recepção e retorna.

MPI_Irecv(void *buf, count,MPI_INT, origem, tag, MPI_Comm comm, MPI_Request *req)

- Dados: não devem ser lidos enquanto não estiverem disponíveis

  Verificação: **MPI_Wait** ou **MPI_Test**.

Comunicação não bloqueante

MPI_Wait(MPI_Request *req,MPI_Status *status)

Espera completar a transmissão ou a recepção.

MPI_Test(MPI_Request *req,int *flag,MPI_Status *status)

Retorna em flag a indicação se a transmissão ou recepção foi completada. Se true, o argumento status está preenchido com informação

MPI_Iprobe(int source,int tag,MPI_Comm comm,int *flag,MPI_Status *status)

Seta flag, indicando a presença do casamento da mensagem.

Comunicação não bloqueante

```
if (rank==0) {
 ............

MPI_Isend(&a[0][0]+(k*n*i),k*n,MPI_DOUBLE,i,
10,MPI_COMM_WORLD,&req);
printf("rank 0 apos Send\n");
fflush(stdout);
.......

}
else {
        MPI_Recv(a,k*n,MPI_DOUBLE,
0,10,MPI_COMM_WORLD,&status);

............

}
```

Comunicação coletiva

MPI_Bcast(void *buf,int count,MPI_Datatype dtype,int root,MPI_Comm comm);

- todos processos executam a mesma chamada de função.
- Após a execução da chamada todos os buffers contêm os dados do buffer do processo root.

Comunicação coletiva

MPI_Scatter(void *sendbuf,int sendcount,
MPI_Datatype sendtype,void *recvbuf,
int recvcount,MPI_Datatype recvtype,int root,
MPI_Comm comm);

- todos os N processos do comunicador especificam o mesmo count (número de elementos a serem recebidos).
- O buffer de root: contém sendcount*N elementos do datatype tipo especificado.
- processo root: distribuirá os dados para os processos, incluindo ele próprio.

Comunicação coletiva

MPI_Gather(void *sendbuf,int sendcount,MPI_Datatype sendtype,void *recvbuf,int recvcount,MPI_Datatype recvtype,int root,MPI_Comm comm);

- operação gather: reverso da operação scatter (dados de buffers de todos processos para processo root)

Exemplo: MPI_Scatter

```
if (rank==0) {
        for(i=0;i<n;i++)
          for(j=0;j<n;j++)
            a[i][j]=1;
      }
   MPI_Scatter(a,k*n,MPI_DOUBLE,a,k*n,MPI_DOUBLE,
   0,MPI_COMM_WORLD);            //  rank 0: root
      parcial=0;
      for(i=0;(i<k);i++)
       for(j=0;j<n;j++)
        parcial+=a[i][j];
```

Comunicação coletiva

- *MPI_Reduce(void *sendbuf,void *recvbuf,int count,MPI_Datatype dtype,MPI_Op op,int root,MPI_Comm comm);*

- Elementos dos buffers de envio: combinados par a par para um único elemento correspondente no buffer de recepção do root.

Operaçoes de redução:

- MPI_MAX (maximum), MPI_MIN (minimum), MPI_SUM (sum), MPI_PROD (product), MPI_LAND (logical and), MPI_BAND (bitwise and) MPI_LOR (logical or) MPI_BOR (bitwise or) MPI_LXOR (logical exclusive or) MPI_BXOR (bitwise exclusive or)

Exemplo: MPI_Reduce

```
k=n/n_nos;
MPI_Scatter(a,k*n,MPI_DOUBLE,a,k*n,MPI_DOUBLE,
   0,MPI_COMM_WORLD);  // rank 0: root
parcial=0;
fim=k;
for (i=0;i<fim;i++)
        for (j=0;j<n;j++)
           parcial=parcial+a[i][j];
MPI_Reduce(&parcial,&soma,1, MPI_DOUBLE,
   MPI_SUM, 0, MPI_COMM_WORLD);   // rank 0:  root
```

Programação Híbrida

- OpenMp ou pthreads: multicores
- MPI: cluster com nós com processadores muticores

MPI e OpenMp

- Processos MPI em cada nó
- OpenMP nos códigos a serem executados pelos processos
- Exemplo

```
// Soma de elementos de um vetor
#include <mpi.h>
#include <stdio.h>
#include <omp.h>
#define n 1000000
main(int argc, char *argv[])
{
        int  i,n_nos, rank;
        MPI_Status status;
        double vetor[n];
        double soma,soma_parcial;
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &n_nos);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
if (rank == 0){
        printf("n_nos=%d\n",n_nos);
        fflush(stdout);
        for(i=0;i<n;i++)
          vetor[i]=1;
      }
MPI_Scatter(vetor,n/n_nos,MPI_DOUBLE,vetor,n,/nosMPI_DOUBLE,0,
                    MPI_COMM_WORLD) ;
soma_parcial=0;
omp_set_num_threads(4);
 #pragma omp parallel shared(vetor) private(i) reduction(+:soma_parcial)
 {
     #pragma omp for
       for(i=0;i<n/n_nos;i++) {
          soma_parcial+=vetor[i];
        }
  }
```

```
MPI_Reduce(&soma_parcial,&soma,1,MPI_DOUBLE,MPI_SUM,
0,MPI_COMM_WORLD);
        if (rank==0) {
           printf("RESULTADO=%f\n",soma);
           fflush(stdout);
        }
            MPI_Finalize();
            return(0);
}
```

Para obter um bom desempenho

- Processos com granularidade grossa
- minimize o número de mensagens
- maximize o tamanho de cada mensagem
- use alguma forma de Balanceamento de carga

Fatores que impactam no desempenho

- **Computadores multicore:**
  - acessos a arquivos simultaneamente
  - acessos a memória simultaneamente
- **Clusters:**
  - acessos simultâneos a arquivos armazenados em um sistema de armazenamento compartilhado
  - simultaneidade de comunicação de mensagens
  - Granularidade dos processos: computação envolvida em relação à comunicação

Conclusão

Impacto da computação de alto desempenho nas áreas da computação

- Novas ferramentas de programação
- Metodologias e Ferramentas de desenvolvimento
- Novos algoritmos

Desafios

- Facilitar a programação
- Facilitar o uso das plataformas (multi-core, clusters, grids)
- Disseminar como utilizar e explorar o uso dos recursos destas plataformas

Conclusão

Pesquisas

- construção de grids para processamento paralelo
  - MPI utilizando múltiplos clusters e servidores
- ambientes de programação paralela
  - ideal: mesma linguagens atendendo estas plataformas
- clusters heterogêneos
- uso de outras arquiteturas (GPUs, Xeon-phi)
- paralelização de aplicações

Bibliografia

1. Ben-Ari, M. "Principles of Concurrent and Distributed Programming", Addison-Wesley, Second Edition, 2006.
2. Akthter, S.; Roberts, J. " Multi-Core Programming"; Intel Press, 2006.
3. Grama, A.; Gupta,A.; Karypis, G.; Kumar, V. "Introduction to Parallel Computing"; Addison-Wesley, Second Edition, 2003.
4. Culler, D.E.;Singh, J.P.;Gupta, A. "Parallel Computer Architecture: a hardware/software approach"; Morgan Kaufmann Publishers, Inc. , 1999.
5. Quinn, M.J. “Parallel Programming in C with MPI and OpenMP”; McGraw-Hill – Higher Education, 2004.