

# Roteiro para Uso do *Cluster* do Programa de Pós-Graduação em Modelagem Computacional - v3.0

## 1 Logando no *cluster*

Para se logar no *cluster*, abra um terminal no linux e na sequência faça um *ssh* para *cluster.mmc.ufjf.br*. Por exemplo:

```
$ ssh seu_usuario@cluster.mmc.ufjf.br
```

Obs: Se você é aluno do Curso de Verão ou da disciplina de graduação “Programação Paralela”, utilize um dos usuários: *verao01*, *verao02*, ..., *verao10*. Peça ao professor a senha dessas contas.

A máquina na qual você está se logando remotamente é o *front-end* do *cluster*. O *front-end* deve ser usado apenas para a edição e compilação dos códigos. A execução das aplicações deve ser realizada necessariamente nas máquinas escravas, através da submissão de *jobs* para a fila do SGE.

Nas próximas seções serão apresentados exemplos que ilustram como deve ser realizada a compilação e a execução das aplicações no *cluster*.

## 2 Copiando os arquivos para o *cluster*

Crie seus diretórios com o comando *mkdir*. Os arquivos com o código-fonte do seu programa paralelo devem ser copiados da sua máquina local para o *cluster* com o comando *scp*. O comando *scp* requer o caminho completo do arquivo a ser copiado. Para descobrir o caminho completo, pode-se utilizar o comando *pwd* no terminal, tanto na origem (máquina onde se encontra o arquivo a ser copiado), quanto no destino (*front-end* do *cluster*). Execute o comando *scp* com esse padrão:

```
$ scp <caminho_origem> <caminho_destino>
```

Por exemplo, para copiar o arquivo *helloworld.c* da máquina onde o mesmo se encontra para o *cluster*, utiliza-se:

```
$ scp /home/berg/Documentos/ICE/Programacao_Paralela/HelloWorld/helloworld.c  
verao06@cluster.mmc.ufjf.br:/home/verao06/Paralela/HelloWorld
```

## 3 MPI

No *cluster* temos duas implementações distintas do MPI: *mpich* e *openmpi*. As duas podem ser utilizadas para compilar e executar códigos MPI. Deve-se apenas verificar se a mesma implementação utilizada na compilação é também utilizada na execução da aplicação.

Para compilar o código paralelo MPI com *mpich*, utilize o caminho completo para o compilador. Para cada linguagem de programação, um compilador diferente deve ser utilizado:

```
/opt/mpich3/gnu/bin/mpicc (para códigos em C)
```

```
/opt/mpich3/gnu/bin/mpic++ (para códigos em C++)
```

```
/opt/mpich3/gnu/bin/mpif77 (para códigos em Fortran77)
```

**/opt/mpich3/gnu/bin/mpif90** (para códigos em Fortran90)

No caso de *openmpi*, temos:

**/opt/openmpi/bin/mpicc** (para códigos em C)

**/opt/openmpi/bin/mpic++** (para códigos em C++)

**/opt/openmpi/bin/mpif77** (para códigos em Fortran77)

**/opt/openmpi/bin/mpif90** (para códigos em Fortran90)

Caso você queira compilar o arquivo `helloworld.c`:

```
1 // Exemplo do Hello World versao MPI. Source:> helloworld.c
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <mpi.h>
6
7 const int MAX_STRING = 100;
8
9 int main () {
10     int q;
11     char greeting[MAX_STRING];
12     int comm_sz;
13     int my_rank;
14
15     MPI_Init(NULL, NULL);
16     MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
17     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
18
19     if (my_rank != 0) {
20         sprintf(greeting, "Greetings from process %d of %d!", my_rank, comm_sz);
21         MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
22     }
23     else {
24         printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
25         for (q = 1; q < comm_sz; q++) {
26             MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
27             printf("%s\n", greeting);
28         }
29     }
30     MPI_Finalize();
31     return 0;
32 }
```

deve executar o comando:

**`$/opt/openmpi/bin/mpicc helloworld.c -o helloworld`**

para compilá-lo usando a implementação *openmpi* ou

**`$/opt/mpich3/gnu/bin/mpicc helloworld.c -o helloworld`**

para usar a implementação *mpich*.

Por padrão, a implementação *openmpi* é utilizada na compilação dos códigos. Em outras palavras, caso *mpicc* seja digitado diretamente na linha de comando, será invocado o compilador disponibilizado pela implementação *openmpi*. Caso você queira que o compilador *mpich* seja chamado por padrão, edite o seu arquivo *.bash\_profile*, localizado no seu diretório *home* (o diretório que você é direcionado ao se logar, i.e., `/home/seu_usuario`), acrescentando a linha:

**PATH=/opt/mpich3/gnu/bin/:\$PATH**

Após salvar o arquivo *.bash\_profile*, execute o comando:

**\$source ~/.bash\_profile**

Quando você voltar a se logar no cluster, o seu PATH será automaticamente atualizado. Em outras palavras, a execução do comando *source* só é necessária quando for feita uma alteração em uma variável de ambiente (como PATH), e desejamos que essa alteração seja aplicada imediatamente.

Assim, para compilar o arquivo *helloworld.c*, basta fazer:

**\$mpicc helloworld.c -o helloworld**

### 3.1 Executando seu programa

O *cluster*, como já dito, utiliza um sistema de filas (SGE) para a execução das aplicações. As aplicações a serem executadas são conhecidas como *jobs* (tarefas). Para submeter *jobs* para execução na fila, faz-se necessário antes descrever as características das máquinas necessárias para sua execução. Essa descrição pode ser feita tanto pela linha de comando, como com a ajuda de um arquivo com a descrição do *job*. A vantagem do uso do arquivo é que descreve-se as características das máquinas apenas uma vez:

Para o exemplo do *helloworld.c*, considere que o mesmo foi compilado com *mpich* e que desejamos executá-lo usando 10 núcleos:

```
1 #!/bin/bash
2 #$ -pe mpich 10
3 #$ -N myjob
4 #$ -cwd
5 #$ -j y
6 #$ -S /bin/bash
7 MYPROG="/home/verao06/Paralela/HelloWorld/helloworld"
8 MPICH2_ROOT="/opt/mpich3/gnu"
9 echo "NHOSTS=$NHOSTS, NSLOTS=$NSLOTS, TMPDIR=/tmp, machines=$TMPDIR/machines"
10 cat $TMPDIR/machines
11 $MPICH2_ROOT/bin/mpiexec -n $NSLOTS $MYPROG
12 exit 0
```

Dois pontos devem ser destacados. O primeiro diz respeito ao fato que a mesma implementação do MPI (*mpich*) foi usada na compilação e na execução do *job* (observe que foi usado */opt/mpich3/gnu/bin/mpiexec* na execução). O segundo ponto diz respeito ao ambiente paralelo (*parallel environment* - *pe*) usado na execução (na segunda linha, *mpich*). Toda vez que *mpich* for usado na compilação, deve-se escolher o ambiente paralelo *mpich* (ou *mpich\_single* - aloca um processo por máquina, ou *mpich\_rr* - aloca os processos usando um esquema *round-robin*). No caso da implementação *openmpi*, deve-se usar *orte* como *pe* (ou *orte\_single*, ou *orte\_rr*). Neste caso, o *job* é ligeiramente diferente:

```
1 #!/bin/bash
2 #$ -N teste_job          # nome do meu job (opcional)
3 #$ -S /bin/bash         # qual o shell que vou usar (obrigatorio)
4 #$ -pe orte 10          # qual o ambiente a ser usado (orte) e quantos nos eu quero (10)
5 #$ -cwd                # quero que o job remoto execute a partir do diretorio atual (↔
# obrigatorio)
6 #$ -o my_job.out        # nome do arquivo onde as saidas serao escritas (opcional)
7 #$ -e my_job.err        # nome do arquivo onde os erros serao escritos (opcional)
```

```

8 # $ -j y
9
10 /opt/openmpi/bin/mpirun -np $NSLOTS hello-mpi

```

Para submeter o *job*, execute o comando:

**\$ qsub job**

Para acompanhar o estado do seu *job*, use *qstat*:

**\$ qstat** ou **\$ qstat -f**

Os estados reportados são: c: *job* completou sua execução; E: *job* apresentou erro durante execução, e por isso execução não foi finalizada; q: *job* está na fila; w: *job* aguardando para executar; r: *job* está em execução; t: *job* está sendo movido para execução em um computador escravo; s: *job* teve execução suspensa.

Observe que, para cada *job*, um identificador (número do *job*) é associado. Esse identificador pode ser utilizado para coletar maiores informações sobre um *job*, por exemplo, a causa de um erro:

**\$ qstat -j <NUMERO do JOB>**

Se quiser cancelar um *job*, use comando *qdel*:

**\$ qdel <NUMERO do JOB>**

Para ver as características das máquinas escravas do *cluster* (número de cores e quantidade de memória), use comando:

**\$ qhost**

Pode-se submeter um *job* para uma máquina específica (por exemplo, podemos submete-lo para as máquinas com processadores Intel, mais rápidos que os processadores AMD):

**\$ qsub -q all.q@<nome\_da\_maquina> job**

Neste exemplo, as saídas geradas pelo primeiro *job* estarão no arquivo myjob.oNUMERO\_DO\_JOB. Pode-se alterar o nome do *job* ou o diretório onde as saídas geradas são armazenadas.

## 4 Pthread

Considere que o seguinte código será usado para testar submissões com Pthread.

```

1 // Hello World para Pthread. Source:> helloPthread.c
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <pthread.h>
6
7 int thread_count;
8
9 void* Hello(void* rank);
10
11 int main(int argc, char* argv[])
12 {
13     long thread;
14     pthread_t* thread_handles;
15
16     thread_count = strtol(argv[1], NULL, 10);

```

```

17  thread_handles = malloc(thread_count * sizeof(pthread_t));
18
19
20  for(thread = 0; thread < thread_count; ++thread)
21  {
22      pthread_create(&thread_handles[thread], NULL, Hello, (void*) thread);
23  }
24
25  printf("Hello from main thread\n");
26
27  for(thread = 0; thread < thread_count; ++thread)
28  {
29      pthread_join(thread_handles[thread], NULL);
30  }
31
32  free(thread_handles);
33
34  return 0;
35 }
36
37 void* Hello(void* rank)
38 {
39     long my_rank = (long) rank;
40
41     printf("Hello from thread %ld of %d\n", my_rank, thread_count);
42
43     return NULL;
44 }

```

Inicialmente compile seu código com o comando do gcc/g++/fortran (dependendo da linguagem utilizada para implementar a sua aplicação), lembrando-se de adicionar o *flag* para *pthread*.

**\$ gcc meu\_codigo.c -o meu\_executavel -lpthread**

**\$ gcc helloPthread.c -o helloPthread -lpthread**

No arquivo com a descrição do *job*, usamos o pe *threaded* (veja que foram requisitados 16 *cores*, um para cada *thread*):

```

1  #!/bin/bash
2  #$ -pe threaded 16
3  #$ -N mythreadjob
4  #$ -cwd
5  #$ -j y
6  #$ -S /bin/bash
7  /home/verao06/Paralelo/HelloWorld/helloPthread $NSLOTS

```

Submeta o *job* com *qsub*. As saídas estarão no arquivo mythreadjob.oNUMERO\_DO\_JOB.

## 5 OpenMP

Considere que o seguinte código será usado para testar submissões com OpenMP.

```

1  // Programa Hello World do OpenMP. Source:> helloOpenMP.c
2
3  #include <stdio.h>
4  #include <stdlib.h>

```

```

5 #include <omp.h>
6
7 void Hello(void);
8
9 int main(int argc, char* argv[])
10 {
11     int thread_count = strtol(argv[1], NULL, 10);
12
13     #pragma omp parallel num_threads(thread_count)
14     Hello();
15
16     return 0;
17 }
18
19 void Hello(void)
20 {
21     int my_rank = omp_get_thread_num();
22     int thread_count = omp_get_num_threads();
23
24     printf("MyRank: %d\n", my_rank);
25
26     //sleep(my_rank); // Trabalho pesado !!!
27
28     printf("Hello from thread %ld of %d\n", my_rank, thread_count);
29 }

```

Inicialmente compile seu código com o comando do gcc/g++/fortran (dependendo da linguagem utilizada para implementar a sua aplicação), lembrando-se de adicionar o *flag* para openmp.

**\$ gcc meu\_codigo.c -o meu\_executavel -fopenmp**

**\$ gcc helloOpenMP.c -o helloOpenMP -fopenmp**

No arquivo com a descrição do *job*, usamos o pe *threaded* (veja que foram requisitados 16 *cores*, um para cada thread):

```

1 #!/bin/bash
2 #$ -pe threaded 16
3 #$ -N mythreadjob
4 #$ -cwd
5 #$ -j y
6 #$ -S /bin/bash
7 /home/verao06/Paralelo/HelloWorld/helloOpenMP $NSLOTS

```

Submeta o *job* com *qsub*. As saídas estarão no arquivo mythreadjob.oNUMERO\_DO\_JOB.

## 6 CUDA

Considere que o seguinte código será usado para testar submissões com CUDA.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 /**
5  No arquivo job colocar sempre o numero de thread por bloco
6  Usage:> $NVCC -g -G helloCUDA.cu -o helloCUDA
7  qsub job

```

```

8  **/
9
10 #define N (2048*2048)                // Tamanho do vetor
11 #define THREAD_PER_BLOCK 256        // Numero de thread disparadas por bloco
12
13 // Funcao que sera executada na GPU
14 // Realiza a soma entre dois vetores
15 __global__ void add (int *a, int *b, int *c)
16 {
17     // Descobre qual o indice da minha thread
18     int index;
19     index = threadIdx.x + blockIdx.x*blockDim.x;
20     // Realiza a soma nos indices da thread do bloco especificado pela GPU
21     c[index] = a[index] + b[index];
22 }
23
24 // Funcao que gera um vetor randomico
25 void random_ints (int *v)
26 {
27     int i;
28     for (i = 0; i < N; i++)
29     {
30         v[i] = rand() % 100;
31     }
32 }
33
34 int main ()
35 {
36     int *a, *b, *c;
37     int *d_a, *d_b, *d_c;
38     int size;
39
40     // Calcula a quantidade de memoria necessaria para alocao
41     size = N*sizeof(int);
42
43     // Aloca memoria para as copias de a, b, c na GPU (ponteiro duplo eh para passar ↔
44     // por referencia o vetor)
45     cudaMalloc((void**)&d_a, size);
46     cudaMalloc((void**)&d_b, size);
47     cudaMalloc((void**)&d_c, size);
48
49     // Alocar memoria para os vetores na CPU
50     a = (int*)malloc(size); random_ints(a);
51     b = (int*)malloc(size); random_ints(b);
52     c = (int*)malloc(size);
53
54     // Copia os vetores da CPU para a GPU
55     cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
56     cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
57     // Agora a GPU recebeu os vetores a, b atraves dos ponteiros d_a, d_b
58
59     // Executa o kernel na GPU para realizar a soma de dois vetores em paralelo
60     add<<<N/THREAD_PER_BLOCK, THREAD_PER_BLOCK>>>(d_a, d_b, d_c);
61
62     // Copia de volta para CPU o resultado da soma que vai estar referenciado por d_c
63     cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
64
65     bool is_ok = true;
66     int i;
67     for(i = 0; i < N; ++i)
68     {
69         if(c[i] != a[i] + b[i])

```

```

69     {
70         printf("a[%d] + b[%d] != c[%d]. %d + %d != %d\n", i, i, i, a[i], b[i], c[i]);
71         is_ok = false;
72         break;
73     }
74 }
75
76 if(is_ok)
77 {
78     printf("Resultado correto\n");
79 }
80 else
81 {
82     printf("Resultado incorreto na posição %d\n", i);
83 }
84
85 // Libera a memoria alocada pelo programa
86 free(a); free(b); free(c);
87 cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
88
89 return 0;
90 }
91 }

```

Para compilar códigos CUDA é necessário utilizar o compilador da Nvidia, *nvcc*, que se encontra no caminho:

**\$ /opt/cuda\_6.5.14/bin/**

De modo semelhante ao feito com o caminho para o compilador MPI, pode-se incluir na variável *PATH* do arquivo *.bash\_profile* o caminho para o compilador CUDA:

**PATH=/opt/mpich3/gnu/bin/./opt/cuda\_6.5.14/bin/:\$PATH**

Veja que podemos manter o caminho para o compilador *mpicc*, adicionando outro caminho para o compilador CUDA.

A compilação é feita com:

**\$ nvcc -g -G helloCUDA.cu -o helloCUDA**

Na descrição do *job* CUDA teremos:

```

1 #!/bin/bash
2 #$ -pe threaded 10
3 #$ -N myjob
4 #$ -cwd
5 #$ -j y
6 #$ -S /bin/bash
7 export LD_LIBRARY_PATH=/share/apps/cuda/lib64:/share/apps/cuda/lib
8 echo "NHOSTS=$NHOSTS, NSLOTS=$NSLOTS, TMPDIR=/machines=$TMPDIR/machines"
9 cat $TMPDIR/machines
10 /home/verao06/Paralelo/CUDA/helloCUDA $NSLOTS
11 exit 0

```

Submeta o arquivo de *job* da mesma maneira que os anteriores, com o comando *qsub*.



## 7 Outros softwares e bibliotecas

No diretório `/share/apps` estão disponíveis diversas aplicações, que podem ser acessadas a partir de qualquer máquina escrava do *cluster*.

## 8 Descrição das máquinas do *cluster*

Todas as máquinas *compute-0-\** possuem processadores AMD Opteron modelo 6272. As máquinas possuem 64 *cores* (*compute-0-0*) ou 32 *cores* (demais máquinas, *compute-0-1* a *compute-0-12*). Todas as máquinas *compute-1-\** possuem processadores Intel Xeon modelo E5620. Todas possuem 8 *cores* (*compute-1-0* a *compute-1-37*). Nas medições realizadas por um aluno, um *core* Intel é 1,72 vezes mais rápido que um *core* AMD para uma aplicação ponto-flutuante.

Todas as máquinas AMD estão conectadas por rede *Infiniband* e *Gigabit Ethernet*. As máquinas Intel estão conectadas apenas por *Gigabit Ethernet*.

As máquinas *compute-0-\** possuem 2 GPUs Tesla M2075, com exceção da máquina *compute-0-0*, que possui 4 GPUs Tesla M2090. As máquinas *compute-1-0* a *compute-1-7* também possuem GPUs. Temos duas GPUs por máquina: Teslas M2050 (*compute-1-0* e *compute-1-1*) e Tesla C1060 (demais). Em todas as máquinas está instalado o driver versão 340.102.

Em todas as máquinas do *cluster* está instalada a versão 3.10.0 – 693.5.2.el7.x86\_64 do *linux*.

Caso o seu *job* não use CUDA, dê preferência para executá-lo nos nós *compute-1-8* em diante.