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

Você pode também usar um aplicativo para fazer o login remoto, como o **putty**.

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*

O sistema de arquivos do *cluster* é distinto dos demais sistemas de arquivos, ou seja, arquivos armazenados na sua conta no laboratório da modelagem computacional não são compartilhados no *cluster*. Isso significa que você tem de transferí-los antes de usá-los. Você pode fazer isso via algum software de transferência de arquivos, em geral com uso da interface gráfica, ou através da linha de comando. Nesta seção, ilustraremos o uso da linha de comando, tendo em vista que o mecanismo de transferência de arquivos via interface gráfica pode variar de software para software.

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

Dependendo da ferramenta utilizada para paralelizar o seu software, os mecanismos para compilar e executar as aplicações variam. Vamos, nas próximas seções, detalhar como aplicações desenvolvidas com as bibliotecas MPI, PThreads, OpenMP e CUDA devem ser executadas. O mecanismo para a execução de *jobs* sequenciais é parecida com a descrita para a execução com *PThreads*.

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_usuário), 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/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-ompi
```

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 primeir *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.

```c
// Hello World para Pthread. Source:> helloPthread.c

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int thread_count;

void* Hello(void* rank);

int main(int argc, char* argv[])
{
    long thread;
    pthread_t* thread_handles;

    thread_count = strtol(argv[1], NULL, 10);

    thread_handles = malloc(thread_count * sizeof(pthread_t));

    for(thread = 0; thread < thread_count; ++thread)
    {
        pthread_create(&thread_handles[thread], NULL, Hello, (void*) thread);
    }

    printf("Hello from main thread\n");

    for(thread = 0; thread < thread_count; ++thread)
    {
        pthread_join(thread_handles[thread], NULL);
    }

    free(thread_handles);

    return 0;
}

void* Hello(void* rank)
{
    long my_rank = (long) rank;

    printf("Hello from thread %ld of %d\n", my_rank, thread_count);

    return NULL;
}
```

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*):

```
#!/bin/bash
#$ -pe threaded 16
#$ -N mythreadjob
#$ -cwd
#$ -j y
#$ -S /bin/bash
```

```
/home/verao06/Paralelo/HelloWorld/helloPthread $NSLOTS
```

Submeta o *job* com *qsub*. As saídas estarão no arquivo mythreadjob.oNUMERO_DO_JOB.

No caso de execução de um *job* sequencial, deve-se usar o mesmo pe, mas com um único *core*.

5 OpenMP

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

```
// Programa Hello World do OpenMP. Source:> helloOpenMP.c

#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void Hello(void);

int main(int argc, char* argv[])
{
    int thread_count = strtol(argv[1], NULL, 10);

    #pragma omp parallel num_threads(thread_count)
    Hello();

    return 0;
}

void Hello(void)
{
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();

    printf("MyRank: %d\n", my_rank);

    //sleep(my_rank); // Trabalho pesado !!!

    printf("Hello from thread %ld of %d\n", my_rank, thread_count);
}
```

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):

```
#!/bin/bash
#$ -pe threaded 16
#$ -N mythreadjob
#$ -cwd
#$ -j y
#$ -S /bin/bash
/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.

```
#include <stdio.h>
#include <stdlib.h>

/**
 No arquivo job colocar sempre o numero de thread por bloco
 Usage:> $NVCC -g -G helloCUDA.cu -o helloCUDA
 qsub job
 **/

#define N (2048*2048)                          // Tamanho do vetor
#define THREAD_PER_BLOCK 256                    // Numero de thread disparadas por bloco

// Funcao que sera executada na GPU
// Realiza a soma entre dois vetores
__global__ void add (int *a, int *b, int *c)
{
    // Descobre qual o indice da minha thread
    int index;
    index = threadIdx.x + blockIdx.x*blockDim.x;
    // Realiza a soma nos indices da thread do bloco especificado pela GPU
    c[index] = a[index] + b[index];
}

// Funcao que gera um vetor randomico
void random_ints (int *v)
{
    int i;
    for (i = 0; i < N; i++)
    {
        v[i] = rand() % 100;
    }
}

int main ()
{
    int *a, *b, *c;
    int *d_a, *d_b, *d_c;
    int size;

    // Calcula a quantidade de memoria necessaria para alocacao
    size = N*sizeof(int);

    // Aloca memoria para as copias de a, b, c na GPU (ponteiro duplo eh para passar ↩
        por referencia o vetor)
    cudaMalloc((void**)&d_a,size);
    cudaMalloc((void**)&d_b,size);
    cudaMalloc((void**)&d_c,size);

    // Alocar memoria para os vetores na CPU
    a = (int*)malloc(size); random_ints(a);
    b = (int*)malloc(size); random_ints(b);
    c = (int*)malloc(size);
```

```
52
53      // Copia os vetores da CPU para a GPU
54      cudaMemcpy(d_a,a,size,cudaMemcpyHostToDevice);
55      cudaMemcpy(d_b,b,size,cudaMemcpyHostToDevice);
56      // Agora a GPU recebeu os vetores a, b atraves dos ponteiros d_a, d_b
57
58      // Executa o kernel na GPU para realizar a soma de dois vetores em paralelo
59      add<<<N/THREAD_PER_BLOCK,THREAD_PER_BLOCK>>>(d_a,d_b,d_c);
60
61      // Copia de volta para CPU o resultado da soma que vai estar referenciado por d_c
62      cudaMemcpy(c,d_c,size,cudaMemcpyDeviceToHost);
63
64      bool is_ok = true;
65      int i;
66      for(i = 0; i < N; ++i)
67      {
68          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 çãposio %d\n", i);
83      }
84
85
86      // Libera a memoria alocada pelo programa
87      free(a); free(b); free(c);
88      cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
89
90      return 0;
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
```

```
#$ -pe threaded 10
#$ -N myjob
#$ -cwd
#$ -j y
#$ -S /bin/bash
export LD_LIBRARY_PATH=/share/apps/cuda/lib64:/share/apps/cuda/lib
echo "NHOSTS=$NHOSTS, NSLOTS=$NSLOTS, TMPDIR/machines=$TMPDIR/machines"
cat $TMPDIR/machines
/home/verao06/Paralelo/CUDA/helloCUDA $NSLOTS
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*. Não instale nenhum software ou biblioteca em sua conta antes de consultar esse diretório.

Exemplos de softwares e bibliotecas presentes neste diretório:

Anaconda automake blas FreeFem metis OpenBLAS OpenFOAM petsc R TensorFlow...

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 $3.10.0 - 693.5.2.\text{el}7.\text{x}86\_64$ do *linux*.

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