Insper

17 - Introdução a MPI

Super Computação 2018/2

Igor Montagner, Luciano Soares

Neste roteiro iremos trabalhar com nossos primeiros programas em MPI.

Parte 0 - instalação e compilação de programas usando MPI

A API original do MPI é em *C*, mas uma versão muito mais prática para *C++* é oferecida pelo pacote *boost-mpi*. Sua instalação é muito simples em sistemas baseados em Debian/Ubuntu:

```
> apt install libboost-mpi-dev
```

A compilação de programas MPI é feita usando um compilador especial chamado `mpicxx` para *C++* e `mpicc` para *C*. Compile o programa *hello.cpp* usando a linha de comando abaixo.

```
> mpicxx hello.cpp -o hello -lboost_mpi
```

A execução de programas MPI também é diferente. Usamos o programa `mpiexec` para rodar nossos programas. `mpiexec` é responsável por criar o número de processos que desejamos, tanto na máquina local como em máquinas remotas na rede.

```
> mpiexec ./hello
```

O número de processos usados é controlado pela flag `-n`. Para executar o programa `hello` com 16 processos usaríamos, por exemplo, o seguinte comando.

```
> mpiexec -n 16 ./hello
```

Se você ver 16 prints com contagens entre 0 e 15 então tudo funcionou corretamente. Se não peça ajuda ao professor.

Parte 1 - envio de mensagens síncronas

O envio e recebimento de mensagens usando `boost-mpi` é muito simples. Cada comunicador possui funções `send` e `recv` que enviam e recebem dados de maneira síncrona. Ou seja, `send` só retorna quando os dados forem enviados de fato e `recv` obviamente só retorna quando os dados forem recebidos.

```
world.send(recipient, tag, data);
world.recv(sender, tag, data_by_ref);
```

Os campos *recipient* e *sender* são inteiros contendo o *rank* do processo destinatário/remetente da mensagem. O campo *tag* é um inteiro que pode ser usado como identificador do tipo da mensagem enviada ou de qual é seu conteúdo. Note que `recv` ós retorna se o processo de rank *recipient* enviar uma mensagem com a mesma *tag*.

Exercício: teste o programa abaixo executando-o com 2 processos. Descreva o que acontece quando ele é executado e explique porque isto ocorre.

```cpp
#include <boost/mpi/environment.hpp>
#include <boost/mpi/communicator.hpp>
#include <iostream>
namespace mpi = boost::mpi;

int main(int argc, char* argv[]) {
    mpi::environment env(argc, argv);
    mpi::communicator world;

    if (world.rank() == 0) {
        int data;
        world.send(1, 1, 100);
        world.recv(1, 0, data);
        std::cout << "Received" << data << " from 1 \n";
    } else {
        int data;
        world.send(0, 1, 200);
        world.recv(0, 0, data);
        std::cout << "Received" << data << " from 0 \n";
    }
    return 0;
}
```

Exercício: conserte o programa acima.

Exercício: execute agora o programa consertado com 3 ou mais processos. Ele ainda funciona? Se não conserte-o e explique as modificações feitas. Se sim aponte quais modificações feitas no item anterior são responsáveis por garantir que o programa funcione com mais processos.

Exercício: crie um programa MPI que, para cada processo i envia uma mensagem com o valor i^2 para o processo $i+1$. O último processo deverá enviar uma mensagem para o processo 0. Ao receber a mensagem seu programa deve imprimir seu conteúdo e o remetente.

Parte 2 - exercícios

Exercício: Crie um programa que encontra o máximo de um vetor usando MPI. Para efeitos didáticos iremos gerar um vetor aleatório no processo de rank 0.

a. Dados processos de ranks *0-(N-1)* (ou seja, N processos), como você organizaria a troca de mensagens entre os processos?

b. Implemente o programa acima usando MPI. Teste-o com diferentes números de processos e tamanhos de entrada, Verifique que os resultados se mantém iguais e se há ganho de desempenho.

Exercício: Neste exercício iremos criar um programa que ordena um array gigante usando passagem de mensagens e 2 processos. Para efeitos didáticos vamos criar um array aleatório no processo de rank 0 e depois enviá-lo para o segundo processo.

a. Supondo que seu programa tenha dois processos com ranks *0* e *1*, crie um diagrama que explica quais mensagens serão trocadas e o quê deverá ser executado por cada processo.

b. Implemente seu programa. Teste-o com diferentes números de processos e tamanhos de entrada, Verifique que os resultados se mantém iguais e se há ganho de desempenho.