Insper

20 - Comunicação Assíncrona

Super Computação 2018/2

Igor Montagner, Luciano Soares

Até o momento trabalhamos com as funções `world.send(..)` e `world.recv(..)` para comunicação **síncrona** entre processos. Ou seja, estas funções só retornam quando os dados foram efetivamente enviados/recebidos. Por vezes é interessante usar comunicação **assíncrona**, ou seja, gostaríamos de indicar que um dado deve ser enviado/recebido antes de precisar de sua confirmação de envio/recebimento. É necessário, portanto, existir um mecanismo que permita verificar se uma mensagem já foi enviada ou recebida. Neste roteiro iremos

Comunicação assíncrona é usada em duas situações:

1. A ordem de recebimento/envio das mensagens não importa;
2. Desejamos iniciar o recebimento de uma mensagem grande enquanto realizamos outra tarefa.

Parte 1 - chamada `irecv` e `isend`

As chamadas `irecv` e `isend` são versões assíncronas da passagem de mensagens e retornam um objeto do tipo `boost::mpi::request`. Objetos deste tipo podem esperar a mensagem ser efetivada com o método `wait` e testar se já foram efetivados com `test`. Veja abaixo um exemplo de recebimento assíncrono de mensagem. Note que o código só bloqueia quando chamamos `r.wait()`.

```
#include <boost/mpi.hpp>
#include <boost/serialization/string.hpp>
#include <string>
#include <iostream>
#include <ctime>

int main(int argc, char *argv[]) {
    boost::mpi::environment env{argc, argv};
    boost::mpi::communicator world;
    if (world.rank() == 0) {
        std::string s;
        auto r = world.irecv(1, 0, s);
        r.wait();
        std::cout << s << "\n";
    }
    else if (world.rank() == 1) {
        sleep(3);
        std::string s = "Hello async!";
        world.send(0, 0, s);
    }
}
```

Exercício: O código abaixo recebe as mensagens de 1 e 2 de maneira assíncrona, mas não funciona muito diferente de um código síncrono. Modifique-o para usar `boost::mpi::wait_any` (link docs) e mostrar as mensagens na ordem de chegada (ou seja, primeiro a do rank 2 e depois a do rank 1).

```
#include <boost/mpi.hpp>
#include <boost/serialization/string.hpp>
#include <string>
#include <iostream>
#include <vector>
#include <ctime>

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

    if (world.rank() == 0) {
        std::string s[2];
        auto r1 = world.irecv(1, 10, s[0]);
        auto r2 = world.irecv(2, 20, s[1]);

        r1.wait();
        std::cout << s[0] << "\n" << std::endl;

        r2.wait();
        std::cout << s[1] << '\n';
    } else if (world.rank() == 1) {
        std::string s = "Hello, world!";
        sleep(2);
        world.send(0, 10, s);
        std::cout << "Fim rank 1 " << std::endl;
    } else if (world.rank() == 2) {
        std::string s = "Hello, moon!";
        sleep(1);
        world.send(0, 20, s);
        std::cout << "Fim rank 2 " << std::endl;
    }
}
```

Dicas:

1. Você pode criar um vetor de `boost::mpi::request` para passar para `wait_any`;
2. `wait_any` espera até que **exatamente** um request seja completado. Logo, você precisará chamar esta função mais de uma vez.
3.

Exercício: Na aula 17 fizemos um programa que encontra o máximo de um vetor de maneira distribuída. Modifique-o para que:

1. o envio dos vetores seja assíncrono;
2. o recebimento dos máximos parciais seja assíncrono;
3. seu programa deverá esperar todos os recebimentos usando a função `boost::mpi::wait_all` (link docs).