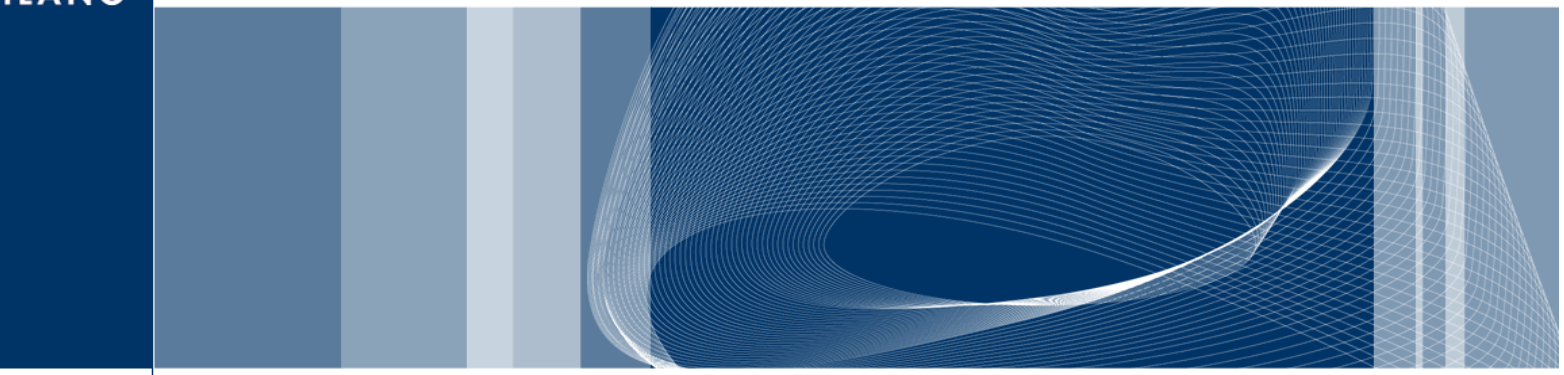
POLITECNICO DI MILANO



Boost.MPI Library

Algoritmi e Calcolo Parallelo

Daniele Loiacono

Riferimenti

- Boost homepage
  http://www.boost.org/

- Tutorial on Boost.MPI
  http://www.boost.org/doc/libs/1_52_0/doc/html/mpi/tutorial.html

- Tutorial on Boost.Serialization
  http://www.boost.org/doc/libs/1_52_0/libs/serialization/doc/

Cosa sono le librerie Boost?

- Il progetto Boost fornisce librerie C++
  - gratuite e open source
  - portabili
  - revisionate da esperti
  - facilmente integrabili con le librerie standard del C++
- Le librerie Boost offrono diverse funzionalità, tra cui:
  - Algoritmi generici
  - Gestione dei grafi
  - Visualizzazione
  - Funzioni matematiche
  - Puntatori e container
  - MPI

Boost.MPI

- Non è un'implementazione alternativa dello standard MPI
- È un interfaccia C++
  - in linea con il moderno stile di sviluppo in C++
  - facilita l'integrazione con il codice C++
  - semplifica l'invio dei messaggi e la gestione di tipi di dato definiti dall'utente
- Al momento fornisce solo alcune funzionalità dello standard MPI
  - Inizializzazione ambiente MPI
  - Gestione comunicatori
  - Comunicazioni punto-punto bloccanti e asincrone
  - Comunicazioni punto-punto non bloccanti
  - Comunicazioni collettive
  - Gestione tipi di dato utente

Elementi Base

Compilare ed eseguire un programma Boost.MPI

- L'esecuzione e la compilazione si basano sugli strumenti forniti dall'implementazione MPI di sistema (e.g., OpenMPI e MPICH)
- Per compilare è inoltre necessario richiedere il linking delle librerie boost MPI e di serializzazione (necessaria per l'invio dei messaggi)
- Compilazione

```
mpic++ –o myprog myprog.c -lboost_mpi-mt -
lboost_serialization-mt
```

  - Occorre linkare le librerie Boost.mpi e Boost.serialization (dove necessaria per l'invio dei messaggi)
- Per eseguire un progama

```
mpirun –np <N> myprog
```

  - Dove `–np <N>` specifica che il programma sarà composto da N processi paralleli

Hello world in Boost.MPI

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

int main(int argc, char* argv[])
{
  mpi::environment env(argc, argv);
  std::cout << "Hello World!" << std::endl;
  return 0;
}
```

Un nuovo Hello World!

```
#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;
  std::cout << "I am process " << world.rank();
  sdt::cout << " of " << world.size() << std::endl;
  return 0;
}
```

Scoprire informazioni sull'ambiente

- La classe `mpi::communicator` permette di istanziare un comunicatore attraverso ogni processo può:
  - sapere quanti processi partecipano a questa computazione
  - localizzarsi all'interno della computazione (cioè scoprire il suo ID)
  - inviare e ricevere dei messaggi
- Il metodo `int size()` restituisce il numero di processi della computazione
- Il metodo `int rank()` restituisce il *rank* (o ID) del processo, che è sempre compreso tra 0 e n-1 (dove n è il numero di processi che partecipano alla computazione)
- Se istanziato con il costruttore di default (senza parametri), il comunicatore comprende *tutti* i processi che partecipano alla computazione (equivale cioè al `MPI_COMM_WORLD`)
- Anche in Boost.MPI è possibile creare un comunicatore che contenga solo un sottoinsieme dei processi coinvolti nella computazione (usando un opportuno costruttore)

Comunicazioni punto-punto

Invio e ricezione di messaggi

- Nella libreria Boost.MPI è possibile inviare e ricevere messaggi attraverso due metodi della classe `mpi::communicator`
  - `template<typename T> void send(int dest, int tag, const T & value)`
  - `template<typename T> status recv(int source, int tag, T & value)`
- I due metodi consentono di inviare e ricevere un messaggio specificando
  - rank del destinatario
  - tag numerica del messaggio
  - dato da inviare
- Il metodo recv ritorna una variabile di tipo `mpi::status` che consente di verificare la sorgente e la tag del messaggio, la presenza di errori e il numero di elementi ricevuti.
- I metodi send a recv forniti in Boost.MPI implementano sono bloccanti e asincroni

Esempio: ping di un carattere

```
#include <iostream>
#include <boost/mpi/environment.hpp>
#include <boost/mpi/communicator.hpp>
namespace mpi = boost::mpi;
int main(int argc, char *argv[])
{
        int numtasks, rank, tag=1; char msg;
        mpi::environment env(argc, argv);
        mpi::communicator world;
        if (world.rank() == 0) {
                std::cin >> msg;
                world.send(1,tag,msg);
                mpi::status s = world.recv(1,tag,msg);
        } else if (world.rank() == 1) {
                mpi::status s = world.recv(0,tag,msg);
                world.send(0,tag,msg);
        }
}
```

Inviare/Ricevere dati in Boost.MPI

- I metodi `send` e `recv` della classe `mpi::communicator` semplificano l'invio e la ricezione dei dati ma richiedono che il tipo di dati inviato sia built-in (int, float, etc.) o **serializzabile** (attraverso la libreria boost)
- La compatibilità dei diversi tipi di dato sono riassunte dal seguente schema

Tipo di dato	Compatibilità con Boost.MPI
Tipi built-in	Piena compatibilità (nessun accorgimento è necessario)
Tipi STL	Usare i tipi ridefiniti in Boost.Serialization Includere la libreria Boost.Serialization nella fase di linking
Array	Utilizzare i metodi: - `template<typename T> void send(int, int, const T *, int)` - `template<typename T> status recv(int, int, T *, int)`
Tipo utente	Rendere il tipo serializzabile (utilizzando Boost.Serialization) Includere la libreria Boost.Serialization nella fase di linking

Esempio: stringhe

```
#include <iostream>
#include <boost/mpi.hpp>
#include <boost/serialization/string.hpp>
namespace mpi = boost::mpi;
int main(int argc, char *argv[])
{
  const int tag=1;
  mpi::environment env(argc, argv);
  mpi::communicator world;
  if (world.rank() == 0) {
      std::string msg = "messaggio";
      world.send(1,tag,msg);
  } else if (world.rank() == 1) {
      std::string msg;
      mpi::status s = world.recv(0,tag,msg);
  }
}
```

Esempio: vector

```
#include <iostream>
#include <boost/mpi.hpp>
#include <boost/serialization/vector.hpp>
namespace mpi = boost::mpi;
int main(int argc, char *argv[])
{
  const int tag=1;
  mpi::environment env(argc, argv);
  mpi::communicator world;
  std::vector<int> v;
  v.push_back(100); v.push_back(200); v.push_back(300);
  if (rank == 0)
      world.send(1,tag,v);
  else if (rank == 1)
      world.recv(0,tag,v);
}
```

Esempio: vector (invio parziale)

```
std::vector<int> v;
…
if (world.rank() == 0) {
    int load = v.size()/world.size();
    int start = load+v.size()%world.size();
    for (int i = 1; i < world.size(); ++i){
      std::vector<int> to_send (v.begin()+start,
                                v.begin()+start+load);
      world.send(i,tag,to_send);
      start+=load;
    }
 }
else {
    std::vector<int> v;
    world.recv(0,tag,v);
}
```

Esempio: invio di un array

```
if (rank == 0) {
  float v[N];
  for (int i = 0; i < N; ++i)
    v[i]=i*0.1f;
  int load = N/world.size();
  int start = load+N%world.size();
  for (int i = 1; i < world.size(); ++i){
    world.send(i,tag,v+start,load);
    start+=load;
  }
}
else {
    int load = N/world.size();
    float v[load];
    world.recv(0,tag,v,load);
}
```

Rendere serializzabile un tipo dati utente

- Per permettere alla libreria Boost.Serialization di serializzare i dati, occorre implementare il seguente metodo privato:

```
template<class Archive>
void serialize(Archive & ar, const unsigned int version)
```

  - dove `Archive` è un tipo definito nelle Boost.Serialization simile ad un input/output datastream
  - `version` permette di mantenere la retro compatibilità con implementazioni precedenti
  - per consentire l'invocazione del metodo, si deve garantire l'accesso `private` alla classe `boost::serialization::access`

Esempio: serializzazione della classe point

```
class point {
public:
        point();
        point(double x, double y);
        double dist2();
        double get_x() {return x;};
        double get_y() {return y;};
private:
        double x;
        double y;
};
```

Esempio: serializzazione della classe point

```
class point {
public:
        point();
        point(double x, double y);
        double dist2();
        double get_x() {return x;};
        double get_y() {return y;};
private:
        friend class boost::serialization::access;
        template<class Archive>
        void serialize(Archive & ar, const unsigned int version){
                ar & x & y;
        };
        double x;
        double y;
};
```

Esempio: serializzazione della classe point

```
class point {
public:
        point();
        point(double x, double y);
        double dist2();
        double get_x() {return x;};
        double get_y() {return y;};
private:
        friend class boost::serialization::access;
        template<class Archive>
        void serialize(Archive & ar, const unsigned int version){
                ar & x & y;
        };
        double x;
        double y;
};
```

L'operatore `&` equivale a `>>` se `Archive` è uno stream di input e a `<<` se è uno stream di output

Esempio: invio di variabili point

```
…
std::vector<point> points;
if (world.rank() == 0)
{
      for (int i = 0; i < N; ++i)
          points.push_back(point());
      world.send(1,tag,points);
}
else if (world.rank()==1)
{
      world.recv(0,tag,points);
      for (int i = 0; i < points.size(); ++i)
      {
        std::cout << points[i].get_x() << ",";
        std::cout << points[i].get_y() << std::endl;
      }
}
```

Rendere serializzabile un tipo dati utente (2)

- Quando non è possibile o consigliabile modificare la classe che si vuole rendere serializzabile, è possibile ottenere lo stesso risultato implementando

```
namespace boost {
  namespace serialization {
    template<class Archive>
    void serialize(Archive & ar, MyClass &c, const unsigned int version)
    {
       // codice necessario per la serializzazione
    }
  }
}
```

- Questa soluzione è possibile soltanto se la classe `MyClass` consente di ricostruire lo stato dell'oggetto con un accesso `public`

Esempio: serializzazione della classe point (2)

```
class point {
public:
        point();
        point(double x, double y);
        double dist2();
        double x;
        double y;
};
namespace boost {
  namespace serialization {
    template<class Archive>
    void serialize(Archive & ar, point &p, const unsigned int version)
    {
       ar & x & y;
    }
  }
}
```

Comunicazioni non bloccanti

Modi di comunicazione

- La Boost.MPI supporta anche le comunicazioni non bloccanti asincrone attraverso i seguenti metodi della classe `mpi::communicator`

```
template<typename T> request isend(int dest, int tag, const T & value)
template<typename T> request irecv(int source, int tag, T & value)
```

  - hanno gli stessi argomenti di `send` e `recv` ma ritornano un oggetto di tipo `mpi::request` che consente di gestire la comunicazione
- Per aspettare il completamento delle comunicazioni non bloccanti è disponibile il metodo `void wait()` di `mpi::request`
- Se si desidera invece attendere il complementamento di più comunicazione bloccanti, è possibile usare la seguente funzione

```
template<typename ForwardIterator>
void wait_all(ForwardIterator first, ForwardIterator last)
```

  - dove `first` and `last` sono il primo e l'ultimo ogggetto all'interno di una sequenza di `mpi::request`

Esempio: comunicazioni non bloccanti

```
…
if (world.rank() == 0) {
    mpi::request reqs[2];
    std::string msg, out_msg = "Hello";
    reqs[0] = world.isend(1, 0, out_msg);
    reqs[1] = world.irecv(1, 1, msg);
    mpi::wait_all(reqs, reqs + 2);
    std::cout << msg << "!" << std::endl;
  } else {
    mpi::request reqs[2];
    std::string msg, out_msg = "world";
    reqs[0] = world.isend(0, 1, out_msg);
    reqs[1] = world.irecv(0, 0, msg);
    mpi::wait_all(reqs, reqs + 2);
    std::cout << msg << ", ";
  }
```

Comunicazioni collettive

Comunicazioni collettive

- Boost.MPI permette anche di utilizzare le comunicazioni collettive definite dallo standard MPI
- A questo scopo, nell'header `mpi/collectives.hpp` sono definite le funzioni
  - `broadcast`
  - `reduce`
  - `scatter`
  - `gather`
- Come nel caso di `send` e `recv`, per ciascuna funzione sono disponibili alcune a varianti (e.g., variante per master, per client, per l'invio/ricezione di array, ecc.).

Broadcast

```
template<typename T>
void broadcast(const communicator & comm, T & value, int root);
```

- `comm` è il communicator su cui avviene la comunicazione
- `value` deve contenere nel processo root il dato da inviare in broadcast e alla fine della broadcast conterrà lo stesso dato in tutti I processi
- `root` è il rank del processo che invia il dato in broadcast
- Il tipo di dato `T`, deve essere reso compatibile con la Boost.MPI

Esempio: broadcast

```
#include <boost/mpi.hpp>
#include <iostream>
#include <string>
#include <boost/serialization/string.hpp>
namespace mpi = boost::mpi;
int main(int argc, char* argv[])
{
  mpi::environment env(argc, argv);
  mpi::communicator world;
  std::string value;
  if (world.rank() == 0) {
    value = "Hello, World!";
  }
  broadcast(world, value, 0);
  std::cout << "Process #" << world.rank() << " says ";
  std::cout << value << std::endl;
  return 0;
}
```

Reduce

```
template<typename T, typename Op>
void reduce(const communicator & comm, const T & in_value,
  T & out_value, Op op, int root);

template<typename T, typename Op>
void reduce(const communicator & comm, const T & in_value,
       Op op, int root);
```

- `comm` è il communicator su cui avviene la comunicazione
- `in_value` è il contributo di ciascun processo alla riduzione
- `out_value` conterrà il risultato della riduzione nel processo root alla fine della comunicazione
- `op` è l'operazione di riduzione desiderata
- `root` è il rank del processo che opera la riduzione finale
- Il tipo di dato `T`, deve essere reso compatibile con la Boost.MPI
- La seconda variante non contiene l'argomento `out_value` ed è quindi utilizzabile nei processi slave

Operazioni di riduzione

MPI	Boost.MPI Equivalent
MPI_BAND	bitwise_and
MPI_BOR	bitwise_or
MPI_BXOR	bitwise_xor
MPI_LAND	std::logical_and
MPI_LOR	std::logical_or
MPI_LXOR	logical_xor
MPI_MAX	maximum
MPI_MAXLOC	unsupported
MPI_MIN	minimum
MPI_MINLOC	unsupported
MPI_PROD	std::multiplies
MPI_SUM	std::plus

- Le operazioni di riduzioni qui riportate sono template e quindi richiedono di specificare il tipo di dato su cui operare
- È inoltre possibile definire nuove operazioni di riduzione (purchè binarie ed associative)

Esempio: reduce

```
#include <boost/mpi.hpp>
#include <iostream>
#include <cstdlib>
namespace mpi = boost::mpi;
int main(int argc, char* argv[])
{
  mpi::environment env(argc, argv);
  mpi::communicator world;
  std::srand(time(0) + world.rank());
  int my_number = std::rand();
  if (world.rank() == 0) {
    int minimum;
    reduce(world, my_number, minimum, mpi::minimum<int>(), 0);
    std::cout << "The minimum value is " << minimum << std::endl;
  } else {
    reduce(world, my_number, mpi::minimum<int>(), 0);
  }
  return 0;
}
```

Scatter

```
template<typename T>
void scatter(const communicator & comm,
  const std::vector< T > in_values, T & out_value, int root);

template<typename T>
void scatter(const communicator & comm, T & out_value, int root);
```

- `comm` è il communicator su cui avviene la comunicazione
- `in_values` è il vettore che contiene gli elementi da inviare ai processi (l'elemento in posizione i-esima verrà inviato al processo con rank i)
- `out_value` conterrà il dato inviato
- `root` è il rank del processo che invia i dati tramite la scatter
- Il tipo di dato `T`, deve essere reso compatibile con la Boost.MPI
- La seconda variante non contiene l'argomento `in_values` ed è quindi utilizzabile nei processi slave

Esempio: scatter

```
#include <boost/mpi.hpp>
#include <iostream>
#include <vector>
#include <cstdlib>
namespace mpi = boost::mpi;
int main(int argc, char* argv[])
{
  mpi::environment env(argc, argv);
  mpi::communicator world;
  int my_number;
  if (world.rank() == 0) {
    std::vector<int> all_numbers;
    std::srand(time(0));
    for (int proc = 0; proc < world.size(); ++proc)
      all_numbers.push_back(std::rand());
    scatter(world, all_numbers, my_number,0);
  } else
      scatter(world, my_number,0);
  std::cout << "P #" << world.rank() << ” got " << my_number << std::endl;
  return 0;
}
```

Gather

```
template<typename T>
void gather(const communicator & comm, const T & in_value,
                        std::vector< T > & out_values, int root);

template<typename T>
  void gather(const communicator & comm, const T & in_value, int root);
```

- `comm` è il communicator su cui avviene la comunicazione
- `out_values` è il vettore che conterrà nel processo root tutti i contributi dei processi coinvolti (l'ordine dei contributi è determinato dal rank dei processi) `op` è l'operazione di riduzione desiderata
- `root` è il rank del raccoglie i dati tramite la gather
- Il tipo di dato `T`, deve essere reso compatibile con la Boost.MPI
- La seconda variante non contiene l'argomento `out_values` ed è quindi utilizzabile nei processi slave

Esempio: gather

```
#include <boost/mpi.hpp>
#include <iostream>
#include <vector>
#include <cstdlib>
namespace mpi = boost::mpi;
int main(int argc, char* argv[])
{
  mpi::environment env(argc, argv);
  mpi::communicator world;
  std::srand(time(0) + world.rank());
  int my_number = std::rand();
  if (world.rank() == 0) {
    std::vector<int> all_numbers;
    gather(world, my_number, all_numbers, 0);
    for (int proc = 0; proc < world.size(); ++proc)
      std::cout << "P #" << proc << ": " << all_numbers[proc] << std::endl;
  } else {
    gather(world, my_number, 0);
  }
  return 0;
}
```