OpenMP / OpenMPI Tutorial

PACS 12/2012

OpenMP

Hello world

Program

```
#include <iostream>

int main (int argc, char* argv[])
{
  std::cout << "Hello World!" << std::endl;

#pragma omp parallel
  {
    std::cout << "Hello World! (This time in parallel!!)" << std::endl;
  } // end of parallel section

  return 0;
}
```

Hello world

Program

```
#include <iostream>
```

No OMP includes

```
int main (int argc, char* argv[])
{
  std::cout << "Hello World!" << std::endl;

#pragma omp parallel
  {
    std::cout << "Hello World! (This time in parallel!!)" << std::endl;
  } // end of parallel section

  return 0;
}
```

Hello world

Program

```
#include <iostream>

int main (int argc, char* argv[])
{
  std::cout << "Hello World!" << std::endl;

#pragma omp parallel
  {
    std::cout << "Hello World! (This time in parallel!!)" << std::endl;
  } // end of parallel section

  return 0;
}
```

Preprocessor directive

Hello world

Compile

```
~$ g++ -fopenmp omp_hello_world.cpp -o omp_hellp_world
```

Hello world

Compile

~$ g++ -fopenmp omp_hello_world.cpp -o omp_hellp_world flag to enable omp

Hello world

Run

~$ ./omp_hellp_world

Hello world

Run

~$ ./omp_hellp_world

no additional command

Control parallelization

Environment variable

```
~$ export OMP_NUM_THREADS=1; ./omp_hello_world
~$ export OMP_NUM_THREADS=2; ./omp_hello_world
~$ export OMP_NUM_THREADS=3; ./omp_hello_world
...
```

...

Get run-time info

Use OpenMP library

```
#include <iostream>
#include <omp.h>

int main (int argc, char* argv[])
{
  int nthreads, tid;
#pragma omp parallel private(tid)
  {
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);
    if (tid == 0)
      {
        nthreads = omp_get_num_threads();
        printf("Number of threads = %d\n",
nthreads);
      }
  }
}
```

Looks very easy, but ...

```
#pragma omp parallel default(none) private(i,k,s) \
 shared(n,m,a,b,c,d,dr)
{
#pragma omp for
for (i=0; i<m; i++) {
int max_val = 0;
s=0 ;
for (k=0; k<i; k++)
  s += a[k]*b[k];

c[i] = s;
dr = c[i];
c[i] = 3*s - c[i];
if (max_val < c[i])
  max_val = c[i];

d[i] = c[i] - dr;
}

} /*-- End of parallel region --*/
```

Looks very easy, but ...

```
#pragma omp parallel default(none) private(i,k,s) \
 shared(n,m,a,b,c,d,dr)
{
#pragma omp for
for (i=0; i<m; i++) {
int max_val = 0;
s=0 ;
for (k=0; k<i; k++)
  s += a[k]*b[k];

c[i] = s;
dr = c[i];
c[i] = 3*s - c[i];
if (max_val < c[i])
  max_val = c[i];

d[i] = c[i] - dr;
}

} /*-- End of parallel region --*/
```

Data Race!!!

Looks very easy, but ...

```
 int parallel_checksum = 0;
#pragma omp parallel shared (a, b, nt)
  {
#pragma omp master
    {
      nt =  omp_get_num_threads ();
      std::cout << "number of threads: " << nt;
    }
#pragma omp for
    for (i=0; i < N-1; i++)
      {
      a[i] = a[i+1] + b[i];
      }

  }
```

Looks very easy, but ...

```
 int parallel_checksum = 0;
#pragma omp parallel shared (a, b, nt)
  {
#pragma omp master
    {
      nt =  omp_get_num_threads ();
      std::cout << "number of threads: " << nt;
    }
#pragma omp for
    for (i=0; i < N-1; i++)
      {
      a[i] = a[i+1] + b[i];
      }

  }
```

Data Race!!!

OpenMPI

Hello world

Program

```
#include <iostream>
#include <cmath>
#include <mpi.h>
using namespace std;

int main(int argc, char ** argv){
  int mynode, totalnodes;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &totalnodes);
  MPI_Comm_rank(MPI_COMM_WORLD, &mynode);

  cout << "Hello world from processor " << mynode << " of " <<
totalnodes << endl;

  MPI_Finalize();
}
```

Hello world

Program

```
#include <iostream>
#include <cmath>
#include <mpi.h>
using namespace std;

int main(int argc, char ** argv){
  int mynode, totalnodes;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &totalnodes);
  MPI_Comm_rank(MPI_COMM_WORLD, &mynode);

  cout << "Hello world from processor " << mynode << " of " <<
totalnodes << endl;

  MPI_Finalize();
}
```

include mpi.h

Hello world

Program

```
#include <iostream>
#include <cmath>
#include <mpi.h>
using namespace std;

int main(int argc, char ** argv){
  int mynode, totalnodes;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &totalnodes);
  MPI_Comm_rank(MPI_COMM_WORLD, &mynode);

  cout << "Hello world from processor " << mynode << " of " <<
totalnodes << endl;

  MPI_Finalize();
}
```

MPI library functions

Hello world

Setup

- Add OpenMPI executables and lbraries the environment

~$ module load openmpi
~$

Compile

~$ mpicxx mpi_hello_world.cpp -o mpi_hello_world
~$ export CXX=mpicxx

Hello world

Setup

- Add OpenMPI executables and lbraries the environment

~$ module load openmpi
~$

needed on all nodes

Compile

~$ mpicxx mpi_hello_world.cpp -o mpi_hello_world
~$ export CXX=mpicxx

Hello world

Setup

- Add OpenMPI executables and lbraries the environment

~$ module load openmpi
~$

Compile

~$ mpicxx mpi_hello_world.cpp -o mpi_hello_world
~$ export CXX=mpicxx

Hello world

Setup

- Add OpenMPI executables and lbraries the environment

~$ module load openmpi
~$

Compile

wrapper script

~$ mpicxx mpi_hello_world.cpp -o mpi_hello_world
~$ export CXX=mpicxx

Basic MPI commands

- MPI_Init
- MPI_Finalize

```
MPI_Init (&argc,&argv)
MPI_Finalize ()
```

- MPI_Comm_size
- MPI_Comm_rank

```
MPI_Comm_size (comm,&size)
MPI_Comm_rank (comm,&rank)
```

- MPI_Send
- MPI_Recv

```
MPI_Send(buffer,count,type,
         dest,tag,comm)
MPI_Recv(buffer,count,type,
         source,tag,comm,status)
```

- MPI_Bcast
- MPI_Reduce

```
MPI_Bcast (&buffer,count,datatype,root,comm)
MPI_Reduce (&sendbuf,&recvbuf,count,
            datatype,op,root,comm)
```

Basic MPI data types

- **`MPI_CHAR`** signed char

- **`MPI_SHORT`** signed short int
- **`MPI_INT`** signed int
- **`MPI_LONG`** signed long int

- **`MPI_UNSIGNED_CHAR`** unsigned char
- **`MPI_UNSIGNED_SHORT`** unsigned short int
- **`MPI_UNSIGNED`** unsigned int
- **`MPI_UNSIGNED_LONG`** unsigned long int

- **`MPI_FLOAT`** float
- **`MPI_DOUBLE`** double
- **`MPI_LONG_DOUBLE`** long double

Example

Compute $$\pi = \int_0^1 \frac{4}{1+x^2}$$

```
#include "mpi.h"
#include "stdio.h"
#include <math.h>

int main(argc,argv)
int argc;
char *argv[];
{
    int done = 0, n, myid, numprocs, i;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
```

Example

```
while (!done)
{
  if (myid == 0) {
      printf("Enter the number of intervals: (0 quits) ");
      scanf("%d",&n);
  }
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
if (n == 0) break;

h   = 1.0 / (double) n;
sum = 0.0;
for (i = myid + 1; i <= n; i += numprocs) {
    x = h * ((double)i - 0.5);
    sum += 4.0 / (1.0 + x*x);
}
```

Example

```
mypi = h * sum;

MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
        MPI_COMM_WORLD);

    if (myid == 0)
        printf("pi is approximately %.16f, Error is %.16f\n",
            pi, fabs(pi - PI25DT));
    }
    MPI_Finalize();
    return 0;
}
```

MPI in C++ (deprecated)

- Most C++ programs use the plain C interface of MPI, (e.g. Karnidiakis book!)
- In addition to that, C++ programs may use the C++ bindings • Simple example:

MPI_Init function:

C: int MPI_Init(int* argc, char*** argv)

C++: void MPI::Init(int& argc, char**& argv)

void MPI::Init()

- Similar: MPI_Finalize function

  C: int MPI_Finalize()

  C++: void MPI::Finalize()

- The functions are defined within the namespace MPI
- Arguments are declared with references instead of pointers

MPI in C++

- Most MPI functions are methods of MPI C++ classes
- MPI class names are derived from the language neutral MPI types by dropping the MPI_ prefix and scoping the type within the MPI namespace: MPI_DATATYPE becomes MPI::Datatype

- The following is an excerpt of the C++ classes of MPI-1:

```
namespace MPI {
  class Comm {...};
  class Errhandler {...};
  class Intracomm : public Comm {...};
  class Exception {...};
  class Graphcomm : public Intracomm {...};
  class Op {...};
  class Datatype {...};
  class Status {...};
};
```

MPI in C++

- Most constants are of type const int:
  MPI::ANY_SOURCE, MPI::ANY_TAG, ...
  MPI::MAX_PROCESSOR_NAME, ...

- The elementary datatypes are const types, const MPI::Datatype:
  MPI::CHAR, MPI::INT, MPI::DOUBLE, ...
  MPI::INTEGER, MPI::REAL, ...

- The predefined communicators are of type MPI::Intracomm:
  MPI::COMM_WORLD, MPI::COMM_SELF

MPI in C++

- Collective operators are of type const MPI::Op:
  MPI::MAX, MPI::SUM, ...

- Communication functions are typically virtual const:
  MPI_Send: void Comm::Send(const void* buf, int count,
  const Datatype& datatype,
  int dest, int tag) const

- The same holds for the collective communication functions:
  MPI_Barrier: void Intracomm::Barrier() const

- If a function has just one argument that is intended to be an output and is not a status object, that argument is dropped and the function returns that value:
  int MPI::Comm::Get_rank()

MPI C++ Example

```
#include "mpi.h"
#include <iostream>

int main(int argc, char* argv[])
{
  MPI::Init(argc, argv); MPI::COMM_WORLD.Set_errhandler(MPI::ERRORS_THROW_EXCEPTIONS);
  try{
    int rank = MPI::COMM_WORLD.Get_rank();
    std::cout << "I am " << rank << std::endl;
   } catch (MPI::Exception e) {
    std::cout << "MPI ERROR: "
              << e.Get_error_code()
              << " - " << e.Get_error_string()
              << std::endl;
   }
}

MPI::Finalize();
return0;
}
```

Available resources

Le risorse HPC – definizione

La definizione di risorsa HPC risiede principalmente in caratteristiche non intuitive

- nodi multiprocessore

- interconnessione nodo-memoria

- interconnessione nodo-nodo

- presenza di coprocessori (es. GPU o MIC)

- I/O dedicato

- high availability

...

... e non nella velocità del singolo core di calcolo! (che comunque non è mai indifferente)

Le risorse HPC@MOX: quali sono

Idra: **16 nodi, 4x2 core/nodo Intel Xeon 5560 (speedmark 5539)**
24 GB RAM per nodo x14 + 32 GB RAM x1 + 96 GB RAM x1
dischi locali ai nodi /scratch
disco condiviso /scratch/idra

Cerbero: **4 nodi, 6 core/nodo Intel i7 3930 (speedmark 12093)**
16 GB RAM per nodo
dischi locali ai nodi /scratch
disco condiviso /scratch/idra

Gigat: **5 nodi, 8x4 core/nodo Intel Xeon E5 4610 v2 (speedmark 12807)**
256 GB RAM per nodo

Le risorse HPC@MOX: organizzazione delle code

Torque (clone di PBS) : software per la gestione delle code

Coda = risorsa hpc + disponibilità (ossia quanti nodi posso allocarmi, quanti job posso lanciare, per quanto tempo posso utilizzarli)

Code attive:

Nome	# nodi	# max nodi (nodes)	# max ore (walltime)	# max job
Gigat	5	2	24	1
idra	15	15	24	2
idralong	8	4	-	1
cerbero	4	1	-	2

Scheduling

```
#!/bin/bash
#PBS -S /bin/bash

# Numero di nodi, di core, massimo walltime, e coda richiesta
#PBS -l nodes=1:ppn=1,walltime=00:05:00  -q cerbero

# Nome del job
#PBS -N myjob

# Regirige lo STDOUT su un file e ci unisce anche lo STDERR
#PBS -o out-qsub.txt
#PBS -j oe
#PBS -e err-qsub.txt

# Il job verrà lanciato dalla stessa directory da cui eseguo il comando qsub
cd ${PBS_O_WORKDIR}

# Impostiamo lo STDOUT riportando questo script
cat qsub_script.sub

# Comandi preliminari
hostname
ulimit -s unlimited # make sure we can put big arrays on the stack
date

# Definizione di variabili d'ambiente necessarie al mio job
export LD_LIBRARY_PATH=/u/software/Repository/opt64/any/lib
export PATH=/usr/local/mpich2_1.2.1p1/bin:/bin:/usr/bin:/usr/sbin:/usr/bin/X11:/sbin:/usr/local/bin:.

#-------------------------------------------------------------------#
# Comando di lancio del mio job
time ../../bin/tg_2D_veio

#-------------------------------------------------------------------#

date
```

Le risorse HPC@MOX: comandi Torque

Per sottomettere un job:

qsub *nome_script_di_lancio*

Per controllare lo stato di un job:

qstat

Per cancellare un job in coda o in esecuzione

qdel *PID_del_job*

/scratch è il disco locale (sul nodo e sul front-end)

/scratch/*nome_nodo* è il disco remoto (sul nodo e sul front-end)