CSEP 524 – Parallel Computation University of Washington

Lecture 6: Parallel Language Survey

Michael Ringenburg
Spring 2015

Reminder

- The class on Tuesday, May 19 has been rescheduled to Thursday, May 21.
  - Same time (6:30pm), same place (CSE 305, MS building 99)

Rest of Quarter

- Today
  - More parallel languages
- May 12
  - Intro to Data Analytics Frameworks; Hadoop
  - Brad Chamberlain finishes guest lecture: Data Parallelism in Chapel
- May 21 (moved from 5/19)
  - Data Analytics in Spark
  - First few presentations
- May 26, June 2
  - Rest of project presentations

Plan for today

- Survey three parallel programming environment, including two of the most widely used
  - MPI: Message Passing Interface
  - OpenMP: Open Multi-Processing
  - Coarray C++: Cray PGAS language (introduce ideas of Coarray Fortran in C++)

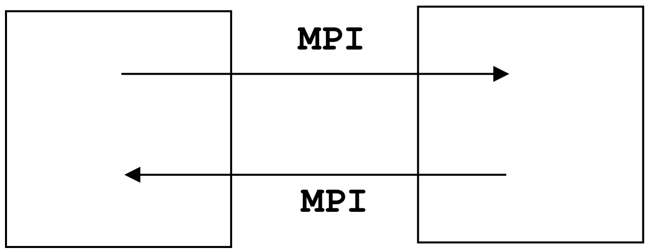
Introduction to MPI

Rajeev Thakur
Argonne National Laboratory

(excerpted and condensed by Brad Chamberlain for CSEP524, Winter 2013)
(further edits by Michael Ringenburg for CSEP524, Spring 2015

The Message-Passing Model

- A *process* is (traditionally) a program counter and address space.
- Processes may have multiple *threads* (program counters and associated stacks) sharing a single address space. MPI is for communication among processes, which have separate address spaces.
- Interprocess communication consists of
  - synchronization
  - movement of data from one process's address space to another's.



What is MPI?

- A *message-passing library specification*
  - extended message-passing model
  - not a language or compiler specification
  - not a specific implementation or product
- For parallel computers, clusters, and heterogeneous networks
- Full-featured
- Designed to provide access to advanced parallel hardware for
  - end users
  - library writers
  - tool developers

MPI Implementations

- MPI is available on all platforms – from laptops to clusters to the largest supercomputers in the world
- Currently, two prominent open-source implementations
  - MPICH2 from Argonne
    - www.mcs.anl.gov/mpich2
  - Open MPI
    - www.open-mpi.org
- Many vendor implementations (many derived from MPICH2)
  - IBM, Cray, Intel, Microsoft, Myricom, SGI, HP, etc
- MVAPICH2 from Ohio State Univ. for InfiniBand
  - http://mvapich.cse.ohio-state.edu/

MPI Resources

- The Standard itself:
  - At http://www.mpi-forum.org
    - All MPI official releases. Latest version is MPI 3.0
    - Download pdf versions
- Online Resources
  - http://www.mcs.anl.gov/mpi
    - pointers to lots of stuff, including other talks and tutorials, a FAQ, other MPI pages
  - Tutorials: http://www.mcs.anl.gov/mpi/learning.html
  - Google search will give you many more leads

Reasons for Using MPI

- **Standardization** - MPI is the only message passing library which can be considered a standard. It is supported on virtually all HPC platforms. Practically, it has replaced all previous message passing libraries.
- **Portability** - There is no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard.
- **Performance Opportunities** - Vendor implementations should be able to exploit native hardware features to optimize performance.
- **Functionality** – Rich set of features
- **Availability** - A variety of implementations are available, both vendor and public domain.

Hello World (C)

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

int main(int argc, char *argv[]) {
    int rank, size;
    // Initialize MPI - many implementations strip
    // mpirun related args, giving "clean" argc/argv
    MPI_Init( &argc, &argv );
    // MPI_COMM_WORLD: All process communicator
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    // Clean up MPI (data structs, etc) - must be same
    // thread as MPI_Init
    MPI_Finalize();
    return 0;
}
```

Some Basic Concepts

- Processes can be collected into *groups*.
- Each message is sent in a *context*, and must be received in the same context.
- A group and context together form a *communicator*.
- A process is identified by its *rank* in the group associated with a communicator.
- There is a default communicator whose group contains all initial processes, called **`MPI_COMM_WORLD`**.

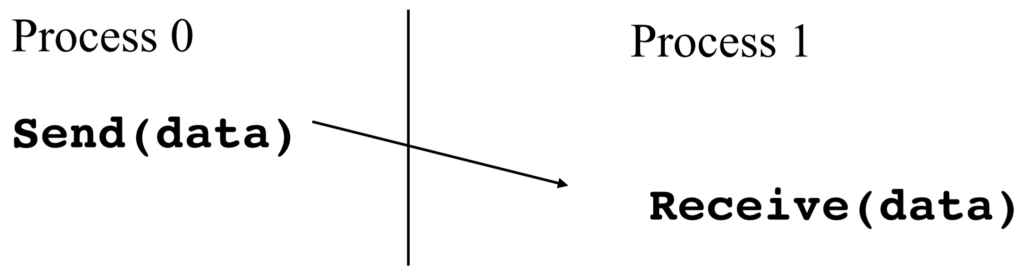
Compiling and Running

- mpicc -o hello hello.c
  - (or mpif77 for Fortran 77, mpif90 for Fortran 90, mpicxx for C++)
  - mpicc etc are scripts provided by the MPI implementation that call the local compiler (e.g., gcc) with the right include paths and link with the right libraries

- mpirun –np 8 hello (or: mpiexec –n 8 hello)
  - Will run 8 processes with the hello executable
  - Further control available to specify location of these processes via a "hosts" file

MPI Basic Send/Receive

- We need to fill in the details in



- Things that need specifying:
  - How will "data" be described?
  - How will processes be identified?
  - How will the receiver recognize/screen messages?
  - What will it mean for these operations to complete?

MPI Datatypes

- The data in a message to be sent or received is described by a triple (address, count, datatype), where
- An MPI *datatype* is recursively defined as:
  - predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE_PRECISION)
  - a contiguous array of MPI datatypes (e.g., a row of C array)
  - a strided block of datatypes (e.g., column of C array)
  - an indexed array of blocks of datatypes (arbitrary pieces of array)
  - an arbitrary structure of datatypes (e.g., a struct)

MPI Tags

- Messages are sent with an accompanying user-defined integer *tag*, to assist the receiving process in identifying the message.
- Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying **`MPI_ANY_TAG`** as the tag in a receive.

MPI Basic (Blocking) Send

MPI_SEND (start, count, datatype, dest, tag, comm)

- The message buffer is described by (**`start, count, datatype`**).
- The target process is specified by **`dest`**, which is the rank of the target process in the communicator specified by **`comm`**.
- When this function returns, the data has been delivered to the system and the buffer can be reused. The message may not have been received by the target process.

MPI Basic (Blocking) Receive

MPI_RECV(start, count, datatype, source, tag, comm, status)

- Waits until a matching (on **`source`** and **`tag`**) message is received from the system, and the buffer can be used.
- **`source`** is the rank in communicator specified by **`comm`**, or **`MPI_ANY_SOURCE`**.
- **`tag`** is a specific tag to match against or **`MPI_ANY_TAG`**
- **`status`** contains further information
- receiving fewer than **`count`** occurrences of **`datatype`** is OK, but receiving more is an error.

Send/Recv example: Passing token around ring

```
int rank, size, tok;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
if (rank != 0) {
  MPI_Recv(&tok, 1, MPI_INT, rank - 1, 0,
           MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  printf("P %d received %d from P %d\n", rank, tok, rank-1);
} else {
  tok = -1; // Proc 0 sets the token's value
}
MPI_Send(&tok, 1, MPI_INT, (rank+1)%size, 0, MPI_COMM_WORLD);

// Now process 0 can receive from the last process.
if (rank == 0) {
  MPI_Recv(&tok, 1, MPI_INT, size - 1, 0,
           MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  printf("P %d received %d from P %d\n", rank, tok, size-1);
}
```

Status Object

- The status object is used after completion of a receive to find the actual length, source, and tag of a message
- Status object is MPI-defined type and provides information about:
  - The source process for the message (status.source)
  - The message tag (status.tag)
- The number of elements received is given by:

int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)

status return status of receive operation (Status)
datatype datatype of each receive buffer element (handle)
count number of received elements (integer)(OUT)

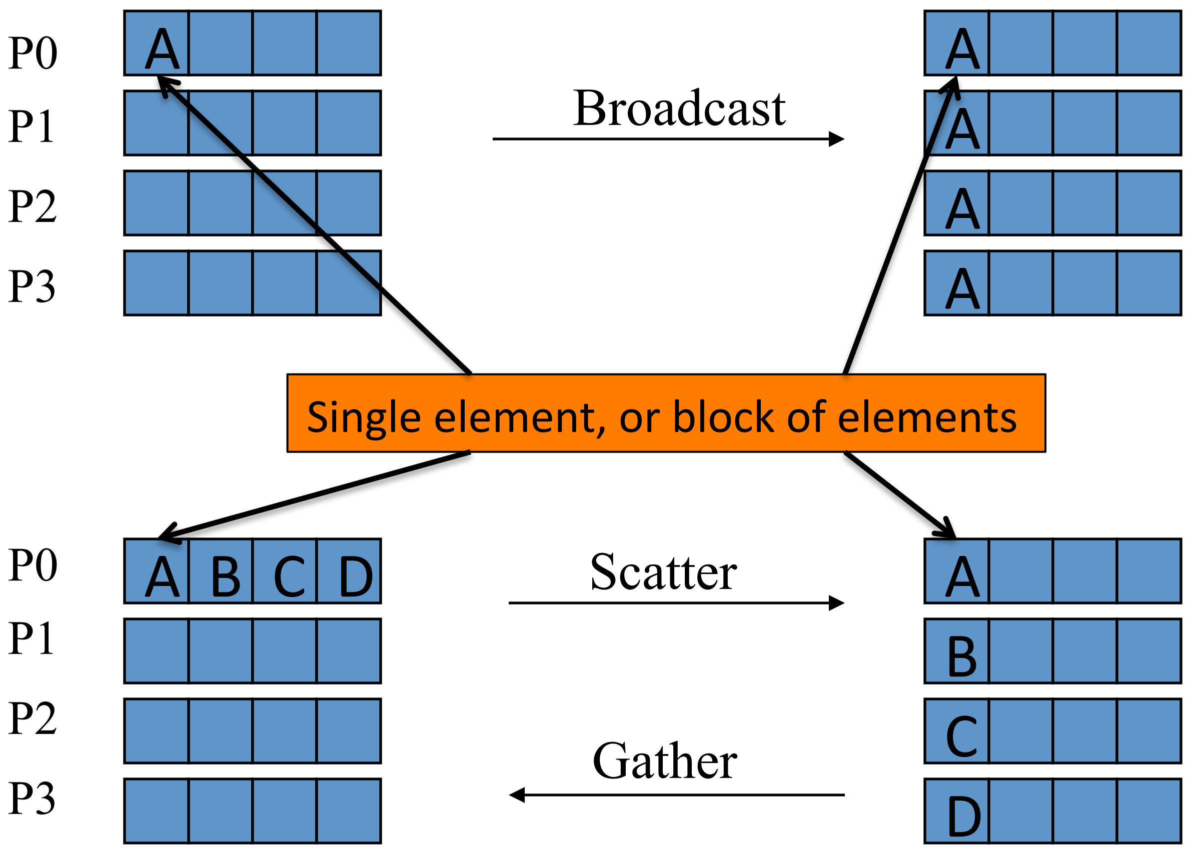
MPI is Simple

- Many parallel programs can be written using just these six functions, only two of which are non-trivial:
  - **`MPI_INIT`** `- initialize the MPI library (must be the first routine called)`
  - **`MPI_COMM_SIZE`** `- get the size of a communicator`
  - **`MPI_COMM_RANK`** `- get the rank of the calling process in the communicator`
  - **`MPI_SEND`** `- send a message to another process`
  - **`MPI_RECV`** `- send a message to another process`
  - **`MPI_FINALIZE`** `- clean up all MPI state (must be the last MPI function called by a process)`
- For performance, however, you need to use other MPI features

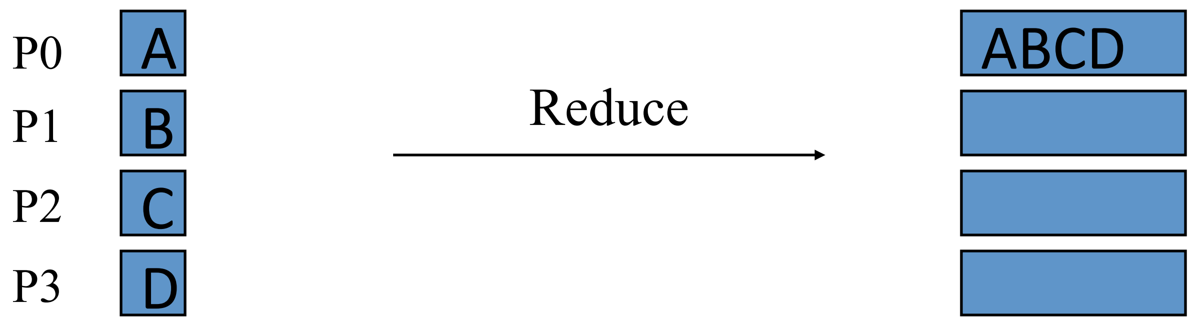
Introduction to Collective Operations in MPI

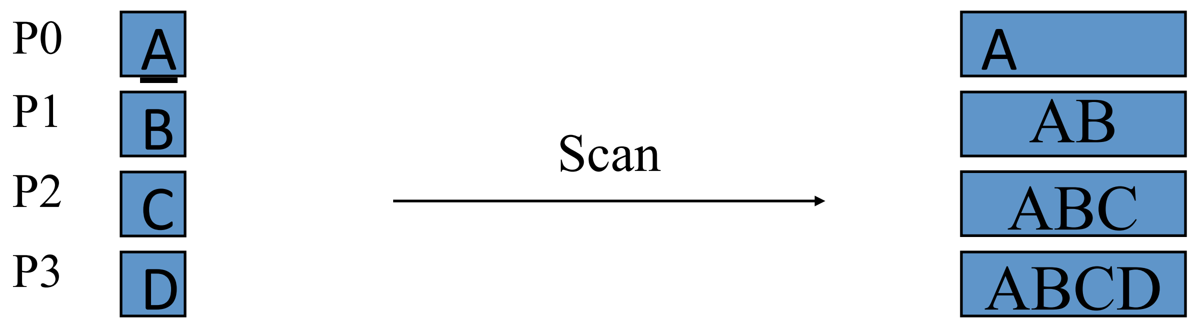
- Collective operations are called by all processes in a communicator.
- **`MPI_BCAST`** distributes data from one process (the root) to all others in a communicator.
- **`MPI_REDUCE`** combines data from all processes in communicator and returns it to one process.
- In many numerical algorithms, **`SEND/RECEIVE`** can be replaced by **`BCAST/REDUCE`**, improving both simplicity and efficiency.

Collective Data Movement



Collective Computation





MPI Built-in Reduce/Scan Computation Operations

• **`MPI_Max`**	Maximum
• **`MPI_Min`**	Minimum
• **`MPI_Prod`**	Product
• **`MPI_Sum`**	Sum
• **`MPI_Land`**	Logical and
• **`MPI_Lor`**	Logical or
• **`MPI_Lxor`**	Logical exclusive or
• **`MPI_Band`**	Binary and
• **`MPI_Bor`**	Binary or
• **`MPI_Bxor`**	Binary exclusive or
• **`MPI_Maxloc`**	Maximum and location
• **`MPI_Minloc`**	Minimum and location

- Can also create custom operations

Example of Collectives: PI in C (1/2)

```
#include "mpi.h"
#include <math.h>
int main(int argc, char *argv[])
{
   int done = 0, n, myid, numprocs, i, rc;
   double PI25DT = 3.141592653589793238462643;
   double mypi, pi, width, sum, x, a;
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);
   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;
```

input/output data

root process

Example of Collectives: PI in C (2/2)

```
    // Estimate pieces of integral of 4/(1 + x^2) from 0 to 1
    width = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
      x = width * ((double)i - 0.5);
      sum += 4.0 / (1.0 + x*x);
    }
    mypi = width * sum;
    // sum pieces
    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;
}
```

input location

output data

operation

root process

Blocking Communication

- In Blocking communication.
  - **MPI_SEND** does not complete until buffer is empty (available for reuse).
  - **MPI_RECV** does not complete until buffer is full (available for use)
- A process sending data will be blocked until data in the send buffer is emptied
- A process receiving data will be blocked until the receive buffer is filled
- Completion of communication generally depends on the message size and the system buffer size
- Blocking communication is simple to use but can be prone to deadlocks

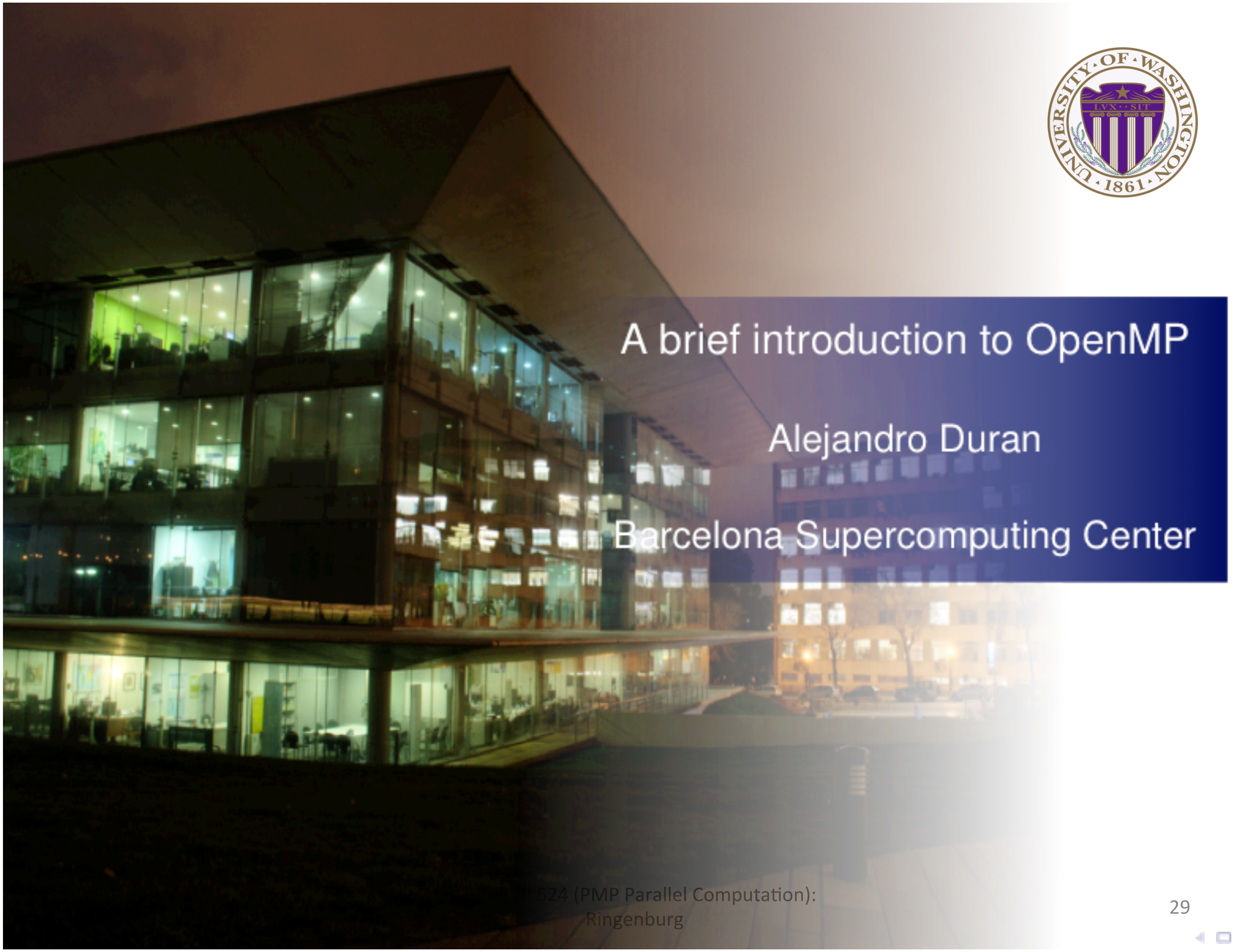
May deadlock →

```
If (my_proc.eq.0) Then
  Call mpi_send(...)
  Call mpi_recv(…)
Else
   Call mpi_send(…) ← UNLESS you reverse send/recv
   Call mpi_recv(….)
Endif
```

Non-Blocking Communication

- Non-blocking (asynchronous) operations return (immediately) ‘ ‘request handles” that can be waited on and queried
  - MPI_ISEND(start, count, datatype, dest, tag, comm, request)
  - MPI_IRECV(start, count, datatype, src, tag, comm, request)
  - MPI_WAIT(request, status)

- Non-blocking operations allow overlapping computation and communication.
- One can also test without waiting using **MPI_TEST**
  - **MPI_TEST(request, flag, status)**
- Anywhere you use **MPI_Send** or **MPI_Recv**, you can use the pair of **MPI_Isend/MPI_Wait** or **MPI_Irecv/MPI_Wait**
- Combinations of blocking and non-blocking sends/receives can be used to synchronize execution instead of barriers (**MPI_Barrier**)



A brief introduction to OpenMP

Alejandro Duran

Barcelona Supercomputing Center

What is OpenMP?

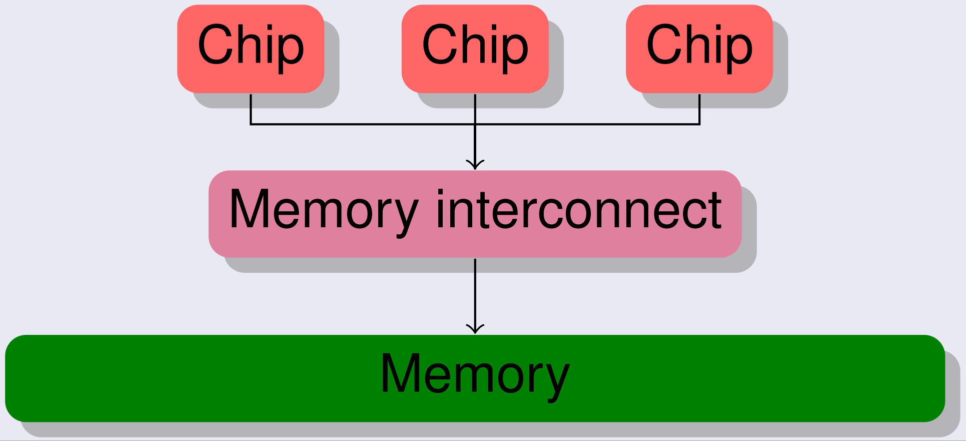
OpenMP 4.0 came out in 2013

- It's an API extension to the C, C++ and Fortran languages to write parallel programs for shared memory machines
  - Current version is 3.1 (June 2010)
  - Supported by most compiler vendors
    - Intel,IBM,PGI,Oracle,Cray,Fujitsu,HP,GCC,...
  - Natural fit for multicores as it was designed for SMPs
- Maintained by the Architecture Review Board (ARB), a consortium of industry and academia

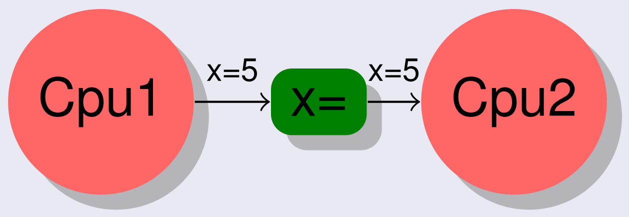
http://www.openmp.org

Target machines

Shared Multiprocessors



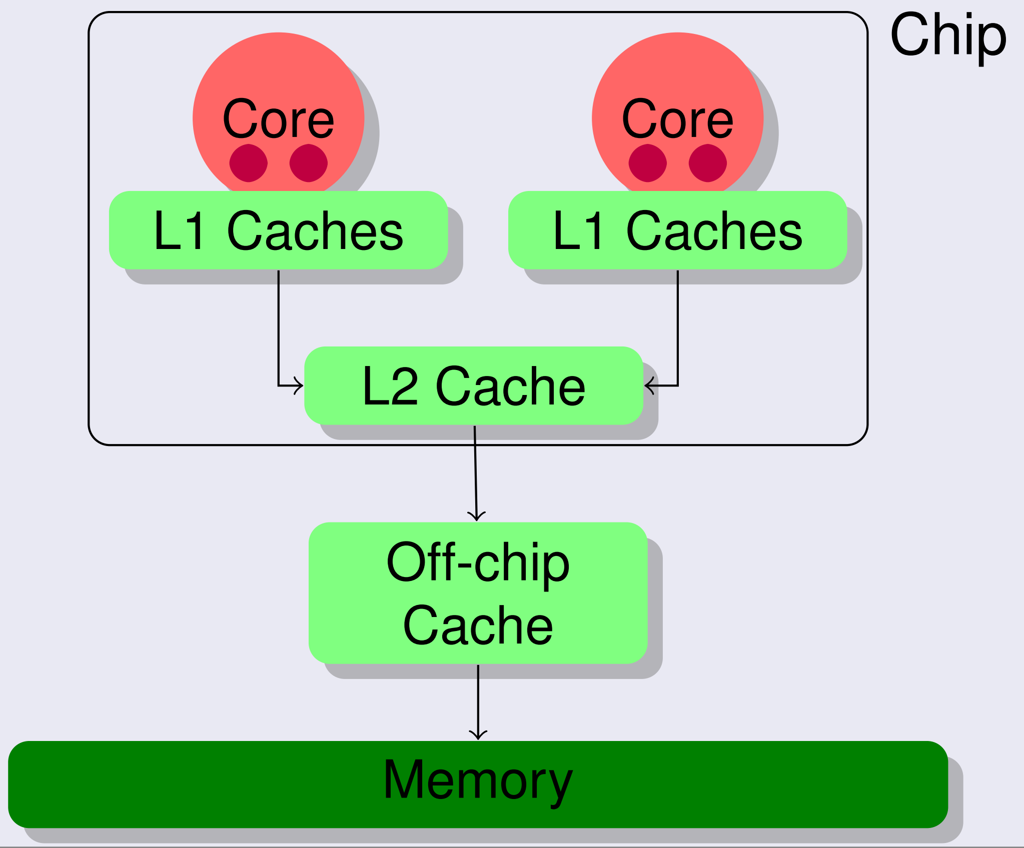
Shared memory



- Memory is shared across different processors
- Communication and synchronization happen implicitely through shared memory

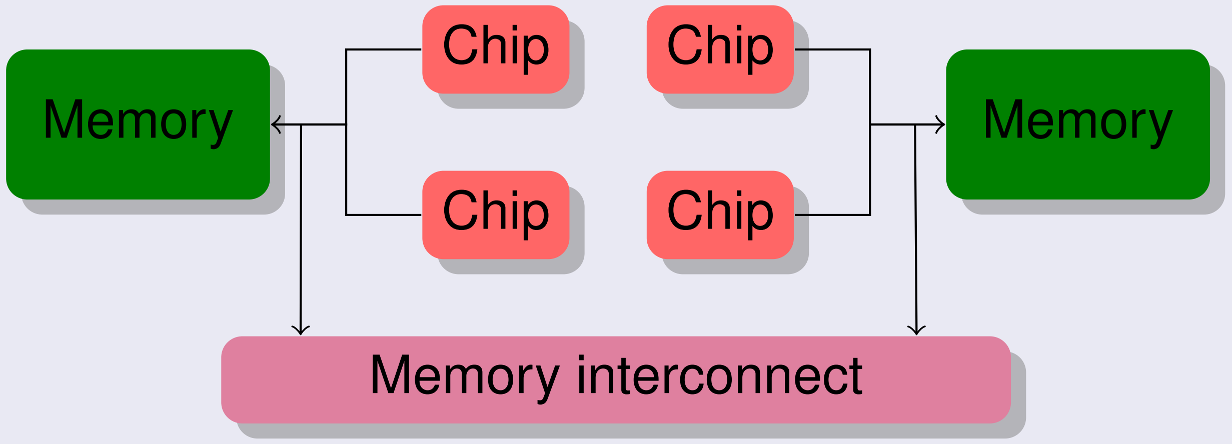
Including...

Multicores/SMTs



More commonly

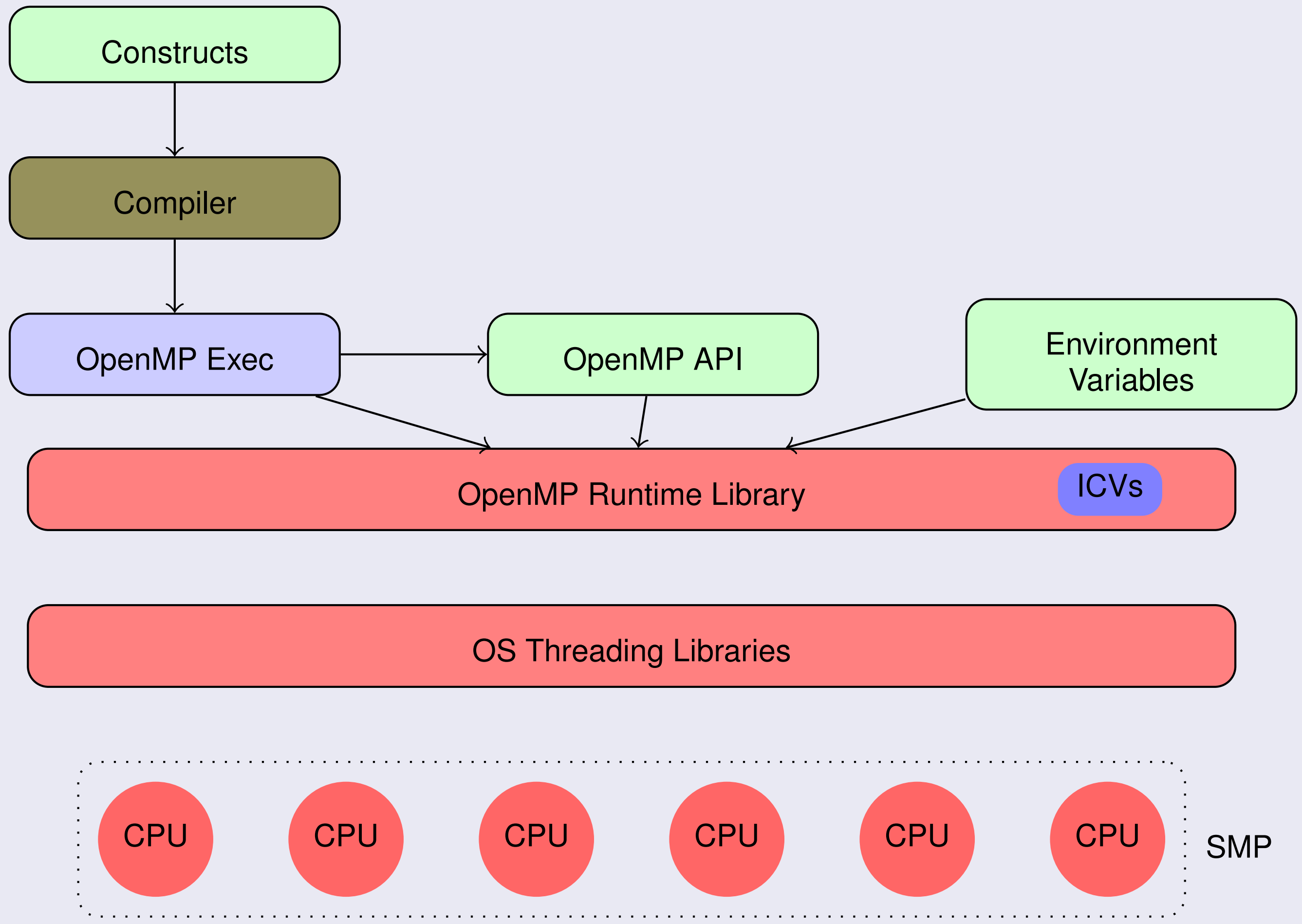
NUMA



- Access to memory addresses is not uniform
- Memory migration and locality are very important

OpenMP at a glance

OpenMP components



OpenMP directives syntax

In Fortran

Through a specially formatted comment:

```
sentinel construct [clauses]
```

where sentinel is one of:

- `!$OMP` or `C$OMP` or `*$OMP` in fixed format
- `!$OMP` in free format

In C/C++

Through a compiler directive:

```
#pragma omp construct [clauses]
```

- OpenMP syntax is ignored if the compiler does not recognize OpenMP

Hello world!

Example

```
int id;
char *message = "Hello␣world!";

#pragma omp parallel private(id)
{
  id = omp_get_thread_num();
  printf("Thread␣%d␣says:␣%s\n", id, message);
}
```

Hello world!

Example

```
int id;
char *message = "Hello␣world!";

#pragma omp parallel private(id)
{
  id = omp_get_thread_num();
  printf("Thread␣%d␣says:␣%s\n", id, message);
}
```

Creates a parallel region of **OMP_NUM_THREADS**

All threads execute the same code

Hello world!

Example

```
int id;
char *message = "Hello␣world!";

#pragma omp parallel private(id)
{
   id = omp_get_thread_num();
   printf("Thread␣%d␣says:␣%s\n", id, message);
}
```

id is private to each thread

Each thread gets its id in the team

Hello world!

Example

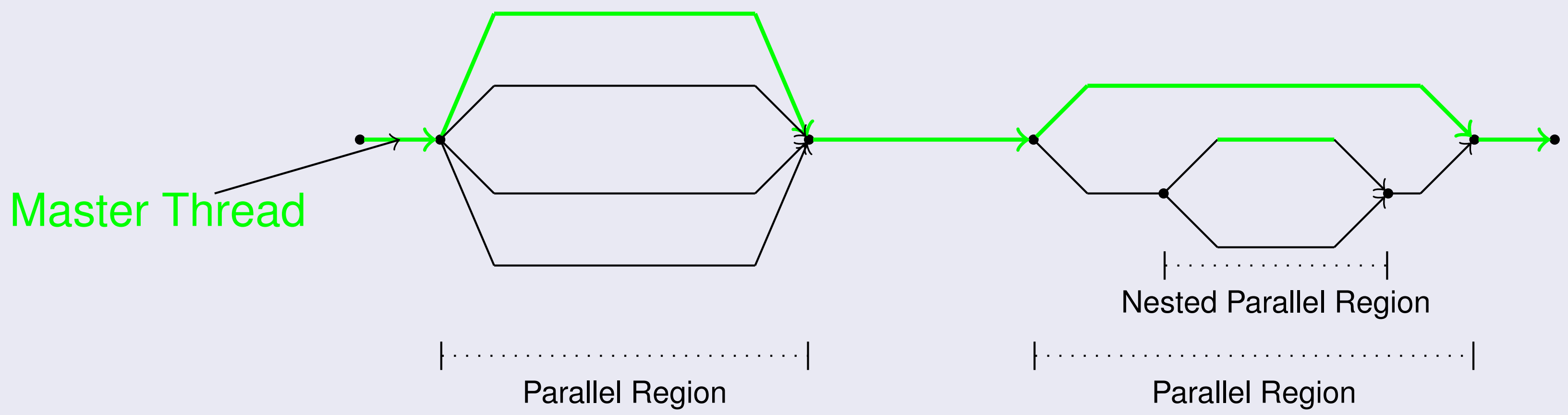
```
int id;
char *message = "Hello␣world!";

#pragma omp parallel private(id)
{
  id = omp_get_thread_num();
  printf("Thread␣%d␣says:␣%s\n", id, message);
}
```

message is shared among all threads

Execution model

Fork-join model

- OpenMP uses a fork-join model
  - The master thread spawns a team of threads that joins at the end of the parallel region
  - Threads in the same team can collaborate to do work



Memory model

- OpenMP defines a weak relaxed memory model
  - Threads can see different values for the same variable
  - Memory consistency is only guaranteed at specific points
    - syncronization constructs, parallelism creation points, . . .
  - Luckily, the default points are usually enough
- Variables can have shared or private visibility for each thread

Data environment

When creating a new parallel region (and in other cases) a new data environment needs to be constructed for the threads. This is defined by means of clauses in the construct:

- **`shared`**
- **`private`**
- **`firstprivate`**
- **`default`**
- **`threadprivate`** ← Not a clause!
- ...

Data-sharing attributes

Shared

When a variable is marked as `shared` all threads see the same variable

- Not necessarily the same value
- Usually need some kind of synchronization to update them correctly

Private

When a variable is marked as `private`, the variable inside the construct is a new variable of the same type with an undefined value.

- Can be accessed without any kind of synchronization

Data-sharing attributes

Firstprivate

When a variable is marked as **`firstprivate`**, the variable inside the construct is a new variable of the same type but it is initialized to the original variable value.

- In a parallel construct this means all threads have a different variable with the same initial value
- Can be accessed without any kind of synchronization

Data-sharing attributes

Example

```
int x=1,y=1,z=1;
#pragma omp parallel shared(x) private(y) firstprivate(z) \
                     num_threads(2)
{
   x++; y++; z++;
   printf("%d\n",x);
   printf("%d\n",y);
   printf("%d\n",z);
}
```

Data-sharing attributes

Example

```
int x=1,y=1,z=1;
#pragma omp parallel shared(x) private(y) fi
                     num_threads(2)
{
   x++; y++; z++;
   printf("%d\n",x);
   printf("%d\n",y);
   printf("%d\n",z);
}
```

The parallel region will have only two threads

Data-sharing attributes

Example

```
int x=1,y=1,z=1;
#pragma omp parallel shared(x) private(y) firstprivate(z) \
                     num_threads(2)
{
   x++; y++; z++;
   printf("%d\n",x);
   printf("%d\n",y);
   printf("%d\n",z);
}
```

Prints 2 or 3. Unsafe update!

Data-sharing attributes

Example

```
int x=1,y=1,z=1;
#pragma omp parallel shared(x) private(y) firstprivate(z) \
                     num_threads(2)
{
   x++; y++; z++;
   printf("%d\n",x);
   printf("%d\n",y);
   printf("%d\n",z);
}
```

Prints any number

Data-sharing attributes

Example

```
int x=1,y=1,z=1;
#pragma omp parallel shared(x) private(y) firstprivate(z) \
                     num_threads(2)
{
   x++; y++; z++;
   printf("%d\n",x);
   printf("%d\n",y);
   printf("%d\n",z);
}
```

Prints 2

Why synchronization?

Mechanisms

Threads need to synchronize to impose some ordering in the sequence of actions of the threads. OpenMP provides different synchronization mechanisms:

- **`barrier`**
- **`critical`**
- **`atomic`**
- **`taskwait`**
- low-level locks

Barrier

Example

```
#pragma omp parallel
{
    foo();
#pragma omp barrier
    bar();
}
```

Forces all foo occurrences too happen before all bar occurrences

Critical construct

Example

```
int x=1;
#pragma omp parallel num_threads(2)
{
#pragma omp critical
    x++;
}
printf("%d\n",x);
```

Critical construct

Example

```
int x=1;
#pragma omp parallel num_threads(2)
{
#pragma omp criti
    x++;
}
printf("%d\n",x);
```

Only one thread at a time here

Critical construct

Example

```
int x=1;
#pragma omp parallel num_threads(2)
{
#pragma omp criti
    x++;
}
printf("%d\n",x);
```

Only one thread at a time here

Prints 3!

Atomic construct

Example

```
int x=1;
#pragma omp parallel num_threads(2)
{
#pragma omp atomic
    x++;
}
printf("%d\n",x);
```

$\leftarrow$ Specially supported by hardware primitives

Atomic construct

Example

```
int x=1;
#pragma omp parallel num_threads(2)
{
#pragma omp atomic
    x++;
}
printf("%d\n",x);
```

Prints 3!

Worksharings

Worksharing constructs divide the execution of a code region among the threads of a team

- Threads cooperate to do some work
- Better way to split work than using thread-ids

In OpenMP, there are four worksharing constructs:

- loop worksharing
- single
- section
- workshare

Restriction: worksharings cannot be nested

The for construct

Example

```
void foo (int *m, int N, int M)
{
  int i;
  #pragma omp parallel
  #pragma omp for private(j)
  for ( i = 0; i < N; i++ )
     for ( j = 0; j < M; j++ )
        m[i][j] = 0;
}
```

The for construct

Example

```
void foo (int *m, int N, int M)
{
  int i;
  #pragma omp parallel
  #pragma omp for ~~private~~(j)
  for ( i = 0; i < N; i++ )
     for ( j = 0; j < M; j++ )
        m[i][j] = 0;
}
```

New created threads cooperate to execute all the iterations of the loop

The for construct

Example

```
void foo (int *m, int N, int M)
{
  int i;
  #pragma omp parallel
  #pragma omp for private(j)
  for ( i = 0; i < N; i++ )
     for ( j = 0; j < M; j++ )
        m[i][j] = 0;
}
```

Loop iterations must be independent

The for construct

Example

```
void foo (int *m, int N, int M)
{
  int i;
  #pragma omp parallel
  #pragma omp for
  for ( i = 0; i
     for ( j = 0;
        m[i][j] = 0;
}
```

The *i* variable is automatically privatized

The for construct

Example

```
void foo (int *m, int N, int M)
{
  int i;
  #pragma omp parallel
  #pragma omp for private(j)
  for ( i = 0; i < N;
     for ( j = 0; j
        m[i][j] = 0;
}
```

Must be explicitly privatized

The reduction clause

Example

```
int vector_sum (int n, int v[n])
{
   int i, sum = 0;
   #pragma omp parallel for

      for ( i = 0; i < n; i++
         sum += v[i];

   return sum;
}
```

Common pattern. All threads accumulate to a shared variable

The reduction clause

Example

```
int vector_sum (int n, int v[n])
{
   int i, sum = 0;
   #pragma omp parallel for reduction(+:sum)

      for ( i = 0; i < n; i++ )
         sum += v[i];

   return sum;
}
```

Efficiently solved with the **reduction** clause

The reduction clause

Example

```
int vector_sum (int n, int v[n])
{
   int i, sum = 0;
   #pragma
  ←
      for (
         sum
  ←
   return su
}
```

Private copy initialized here to the identity value

Shared variable updated here with the partial values of each thread

The single construct

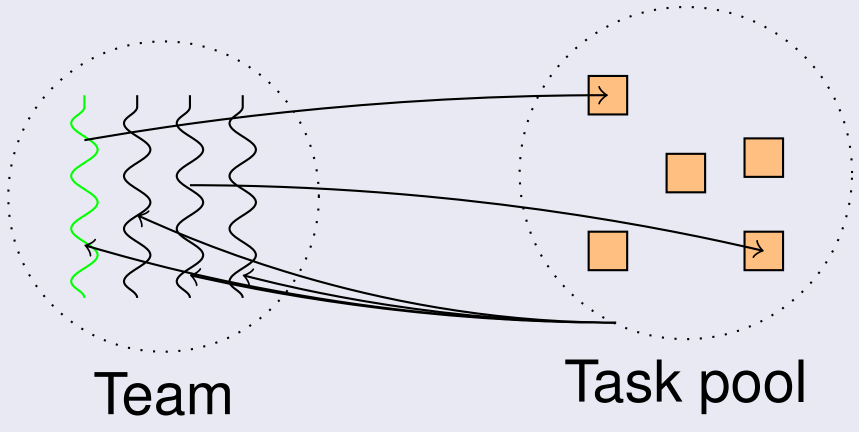
Example

```
int main (int argc, char **argv )
{
    #pragma omp parallel
    {
       #pragma omp single
       {
          printf("Hello␣world!\n");
       }
    }
}
```

This program outputs just one “Hello world”

Task parallelism in OpenMP

Task parallelism model



- Parallelism is extracted from “several” pieces of code
- Allows to parallelize very unstructured parallelism
  - Unbounded loops, recursive functions, ...

What is a task in OpenMP ?

- Tasks are work units whose execution may be deferred
    - they can also be executed immediately
- Tasks are composed of:
    - code to execute
    - a data environment
        - Initialized at creation time
    - internal control variables (ICVs)
- Threads of the team cooperate to execute them

When are task created?

- `Parallel` regions create tasks
  - One implicit task is created and assigned to each thread
    - So all task-concepts have sense inside the parallel region
- Each thread that encounters a `task` construct
  - Packages the code and data
  - Creates a new explicit task

List traversal

Example

```
void traverse_list ( List l )
{
  Element e;
  for ( e = l->first; e ; e = e->next )
     #pragma omp task
       process(e);
}
```

e is **firstprivate**

Taskwait

Example

```
void traverse_list ( List l )
{
  Element e;
  for ( e = l->first; e ; e = e->next )
     #pragma omp task
       process(e);

  #pragma omp taskwait  <——  Suspends current task until all children are completed

}
```

Taskwait

Example

```
void traverse_list ( List l )
{
  Element e;
  for ( e = l->first; e ; e = e->next )
     #pragma omp task
       process(e);

  #pragma omp taskwait

}
```

Now we need some threads to execute the tasks

List traversal

Completing the picture

Example

```
List l

#pragma omp parallel
    traverse_list(l);
```

List traversal

Completing the picture

Example

```
List l

#pragma omp parallel
   traverse_list(l);
```

This will generate multiple traversals

List traversal

Completing the picture

Example

```
List l

#pragma omp parallel
   traverse_list(l);
```

We need a way to have a single thread execute traverse_list

List traversal

Completing the picture

Example

```
List l

#pragma omp parallel
#pragma omp single
   traverse_list(l);
```

List traversal

Completing the picture

Example

```
List l

#pragma omp parallel
#pragma omp single
   traverse_list(l);
```

One thread creates the tasks of the traversal

List traversal

Completing the picture

Example

```
List l

#pragma omp parallel
#pragma omp single
   traverse_list(l);
```

All threads cooperate to execute them

Coarray C++

Troy Johnson (Cray)
Presented by David Henty (EPCC)

HPC Application Trends

- C++ has become an important HPC language
- HPC apps combine base language(s) with parallel model(s)
  - Language: Fortran, C, C++
  - Model: MPI, PGAS, OpenMP, OpenACC
- PGAS models have performance and productivity benefits over traditional MPI
  - Examples: UPC, Fortran coarrays, SHMEM
- Language-based PGAS models permit static type checking
- Problem: No language-based PGAS option for C++
  - Mixing UPC and C++ requires non-portable type-punning tricks that circumvent type checking

Making C++ a PGAS Language

- Desirable to bring Fortran coarray or UPC model to C++
  - Something entirely new is less familiar to programmers
- C++ is evolving more via its template library than by its syntax
  - More templates added by C++11, fewer syntax changes
  - Trend expected to continue with later standard revisions
- Either coarray or UPC features could be added with templates
  - Adding a coarray template is easier
- Coarrays were preferred because
  - Can borrow ideas from an ISO language standard: Fortran
  - Coarrays force programmer to consider locality more, which can permit greater performance

Coarray C++ “Hello World”

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

using namespace coarray_cpp;

int main( int argc, char* argv[ ] )
{
  std::cout << “Hello from image “
            << this_image() << “ of “
            << num_images() << std::endl;
  return 0;
}
```

```
➢ CC –o hello hello.cpp
➢ aprun –n4 ./hello
Hello from image 0 of 4
Hello from image 1 of 4
Hello from image 2 of 4
Hello from image 3 of 4
```

Type System

- General coarray template and specializations
  - template <typename T> class coarray;
  - template <typename T, size_t S> class coarray<T[S]>;
  - template <typename T> class coarray<T[ ]>;
- Examples
  - coarray<int> i; // scalar coarray – one i on each image
  - coarray<int[10][20]> x; // statically-sized array per image
  - coarray<int[][20]> y(n); // dynamically-sized array per image
- Local access: x[1][2] = 0; // write to this_image()'s memory
- Remote access: x(5)[1][2] = 0; // write to image 5's memory

Copointers

- Coreferences have an address() member function that returns a copointer
- Does not change its target image when incremented
- Acts as an iterator if used with standard algorithms
- Example:

```
#include <algorithm>
coarray<int[100]> x;
coptr<int> begin = x(2)[0].address();
coptr<int> end = x(2)[100].address();
std::fill( begin, end, 42 );
```

Coarrays of pointers

- A coarray allocates the same amount of memory on each image
  - Can be wasteful
- Solution is a coarray of pointer type
- Example

```
coarray<int*> p;
p = new int[this_image() * 10];
// initialize data here
sync_all(); // ensure all images have allocated and initialized
int y = p(3)[4]; // accesses p[4] on image 3
```

Image Synchronization and Atomics

- sync_all is one form of image synchronization
- coevents and comutexes are other forms
- coatomics are modeled after C++11 atomics
- Example

```
coarray< coatomic<long> > x(0L); // initialize to 0
size_t n = num_images();
for ( size_t i = 0; i < n; ++i ) {
   x(i) += this_image(); // atomic add
}
sync_all();
assert( x == ( n * ( n – 1 ) / 2 ) );
```

Conclusion

- Cray introduced Coarray C++ to combine the so far separate industry trends of using C++ and PGAS for HPC applications
- Extending C++ via templates did not require compiler modifications
  - Non-Cray compilers can be used to compile Coarray C++ programs on Cray systems
  - Implementation by other HPC vendors is possible
- Extension via templates integrates closely with type system to enable static type checking

Discussion

- We’ve seen lots of languages the last couple weeks, between lectures and readings.
- What are your thoughts/impressions?