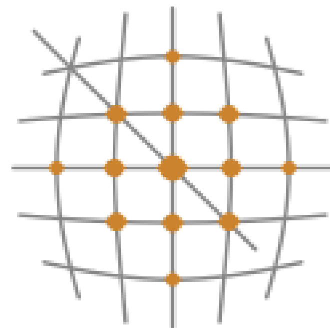
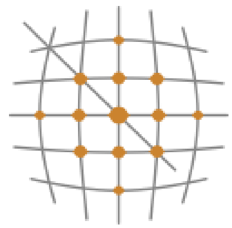
Scientific Computing

Paschalis Korosoglou

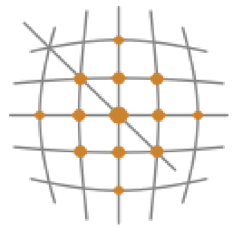


Scientific Computing Center
Aristotle University of Thessaloniki



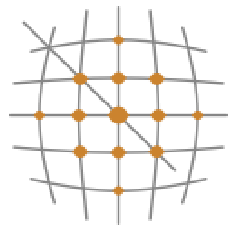
Outline

- Programming for scientific purposes
- Linux (Makefiles, Compilers, Libraries & Tools)
- Parallel programming
  - OpenMP
  - MPI
- Grid, HPC, Cloud etc



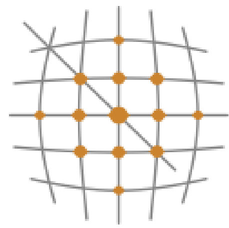
Science goes "*in-silico*"

- Exact solutions are not always possible using current theoretical tools and methods
  - i.e. many problems we have to solve are non-linear
- Numerical integration and simulation technics are providing answers to difficult problems
- The more complex the problem the more demanding the solution will be
  - Better hardware, improved software etc



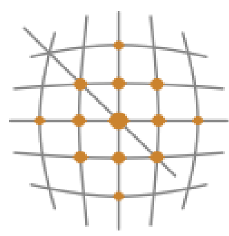
Methods

- Monte-carlo (Map reduce in general)
- Finite differences, finite volumes (structured grids)
- Finite elements (unstructured grids)
- Spectral analysis
- Dense Linear Algebra
- Sparse Linear Algebra
- N-body & particles simulations



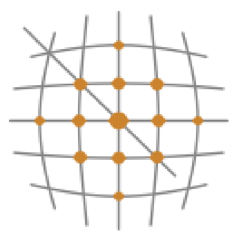
Fields of application

- Astrophysics
- Biology
- Chemistry
- Climate
- Economy
- Engineering
- High energy physics
- Nanotechnology
- Seismology
- Sociology and many many more…



What any scientist will ask oneself

- *"How can I solve an even bigger problem than the one I finally solved last night after struggling for the last three years?"*
- *"I am happy with the solution **but** I want to solve it a couple trillion more times with varying initial conditions"*
- *"Ok, those are valid points. I want to go both ways!"*

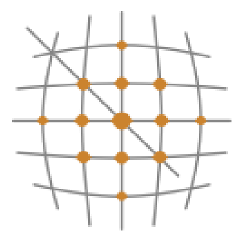


The computer engineer will respond..

- "*Well you can try improving your code. What does profiling tell you? Do you overlap computation with communication? And what about I/O? Is that going to be a bottleneck?*" – Software refactoring

- "*Ok, look. We have a new machine on the way and we expect it to be available in a couple of months. If you can wait until then we can try out your code and hopefully it will work the way you want it to*" – Hardware specs

- "*Have you tried linking it with mkl?*" – Code re-use



Start off serially (take one step at a time)

- But don't get too excited. You are not living in the 90s any more…

"From 2007 to 2011, maximum CPU clock speed (with Turbo Mode enabled) rose from 2.93GHz to 3.9GHz, an increase of 33%. From 1994 to 1998, CPU clock speeds rose by 300%."

Intel CPU Trends
(sources: Intel, Wikipedia, K. Olukotun)

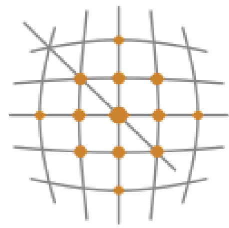
Dual-Core Itanium 2
Pentium 4
Pentium
386

Transistors (000)
Clock Speed (MHz)
Power (W)
Perf/Clock (ILP)

10,000,000
1,000,000
100,000
10,000
1,000
100
10
1
0

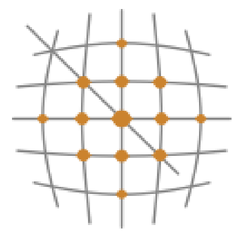
1970 1975 1980 1985 1990 1995 2000 2005 2010

Scientific Computing Center
Aristotle University of Thessaloniki
http://www.grid.auth.gr



Parallel programming

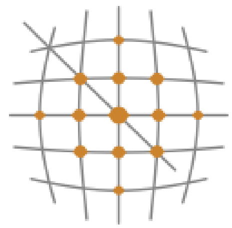
- Parallel programming may overcome the hardware issues but before doing anything parallel make sure that:
  - Your serial code is already optimal! Questions to ask yourself:
    - *Are you using other people's computational and I/O libraries?*
    - *Have you tested with other compilers and, if yes, have you tried various optimization flags?*
    - *What does profiling tell you?*



Why Linux?

- It’s stable, light and well documented
- It does the job
- It supports as many and even more tools for computing

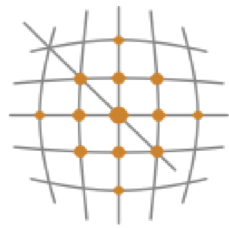
More than 80% of the systems on the Top500 list run Linux



What you will need to login

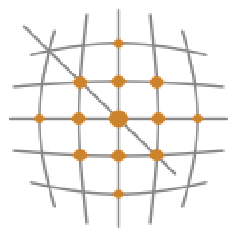
- If you are running Windows download and install
  - Putty
  - Winscp

http://wiki.grid.auth.gr/wiki/bin/view/Groups/ALL/UserInterfacesAtAuth



Setting the ground

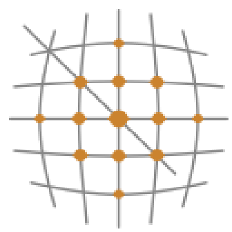
- Anything in Courier usually denotes something you type in a terminal window
- Lines starting with '#' or '$' signs usually denote commands you will have to issue
- The '<','>' marks are used to denote segments you need to change before you type in.
- Stuff in *italic* is usually also stuff you will need to replace



Hands-on!

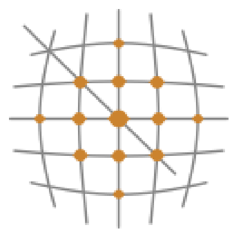
```
$ ssh demoXY@ui.afroditi.hellasgrid.gr

$ cd /mnt/cpg/demo/demoXY

$ svn co http://svn.hellasgrid.gr/svn/…
```

http://goo.gl/PHpyp



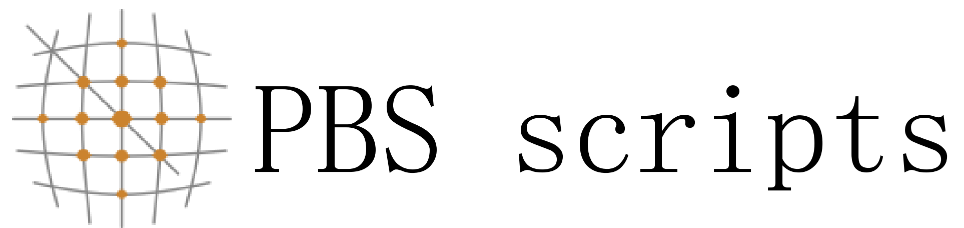
Useful commands

Command	Description
ls -la	List contents of current working directory (the -ls part is optional for long and all listing)
mkdir *directory*	Create a new directory (folder)
cd *directory*	Change the current working directory. Note that cd .. Takes you one folder up.
pwd	Print the current working directory (where - on which path - am I)
hostname	Print the name of the host (where am I logged onto)
whoami	Print my username (who am I)
who	Print a list of everyone logged in right now.



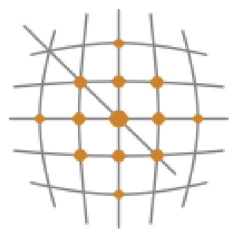
Information discovery

Command	Description
uname -a	Provides information on architecture and kernel version
cat /etc/redhat-release	OS name and version (redhat based systems only)
pbsnodes	Provides a list of available resources, their properties and their current state
qstat	Displays the submitted list of your jobs



PBS scripts

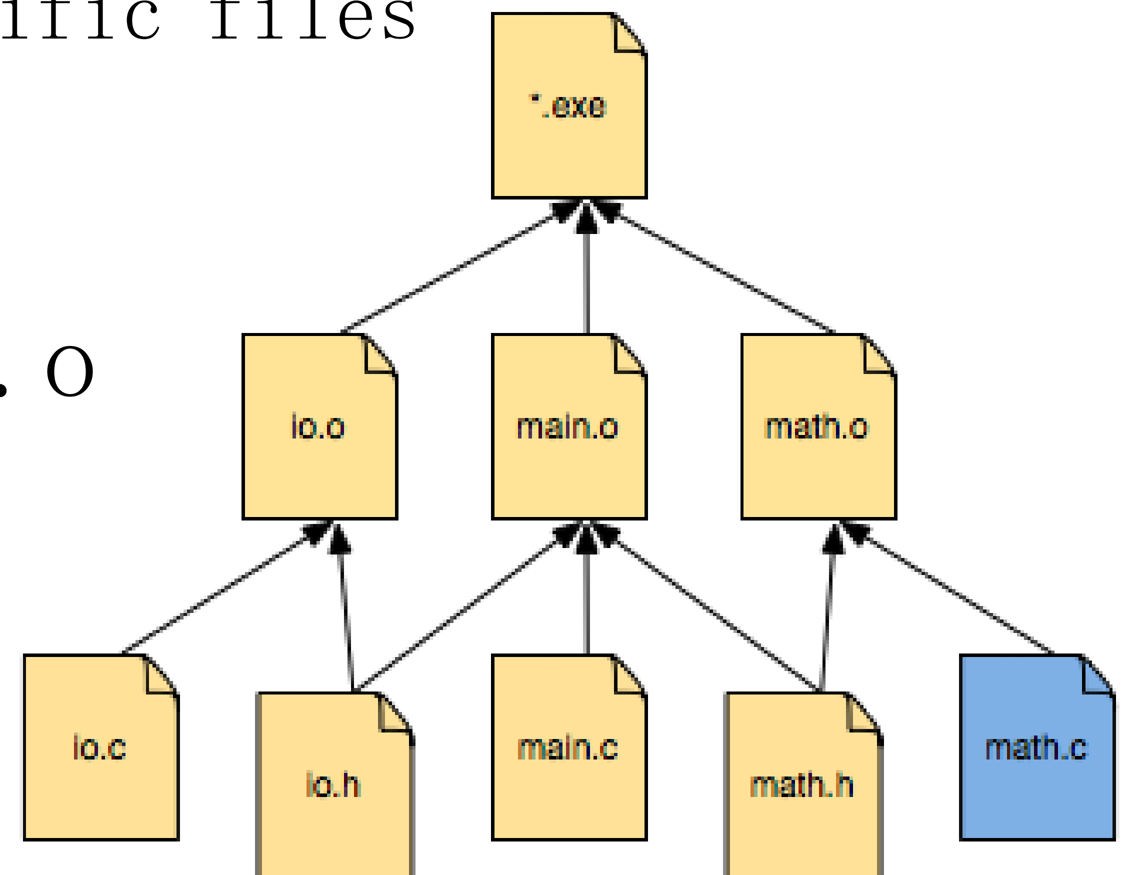
```
#!/bin/bash
#PBS -N hostname
#PBS -q complex
#PBS -l walltime=10:00:00
#PBS -j oe

# this is a comment
/bin/hostname
```

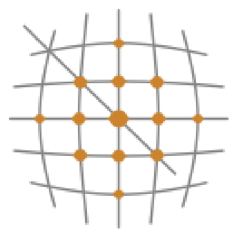


The Makefile

- Define a set of rules to follow
  - Used mainly for source code **compilation**
  - Helpful during development phase when small changes are made to specific files

```
math.o: math.c math.h
    gcc -c math.c -o math.o
```



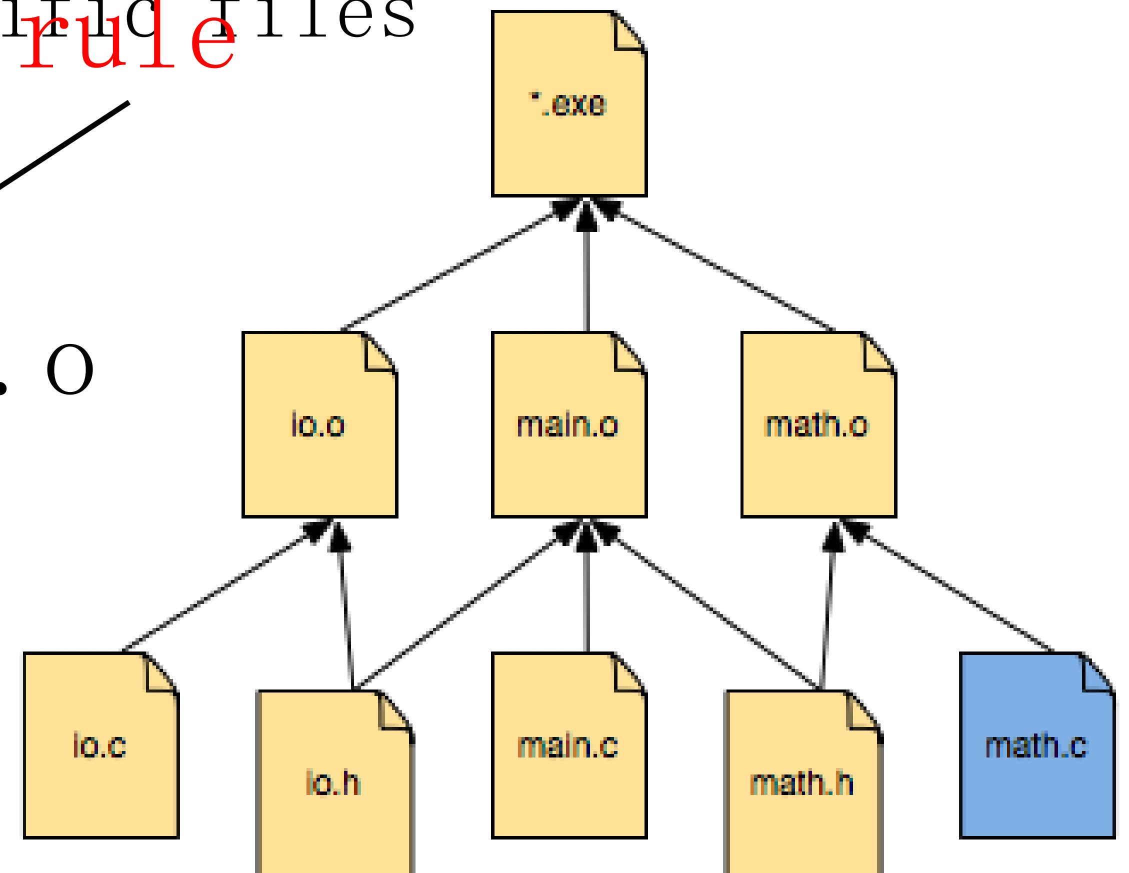


The Makefile

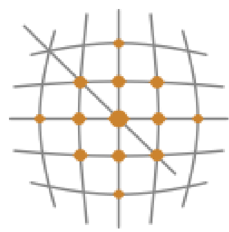
- Define a set of rules to follow
  - Used mainly for source code **compilation**
  - Helpful during development phase when small changes are made to specific files

target dependancies rule

```
math.o: math.c math.h
    gcc -c math.c -o math.o
```



Scientific Computing Center
Aristotle University of Thessaloniki
http://www.grid.auth.gr



The Makefile

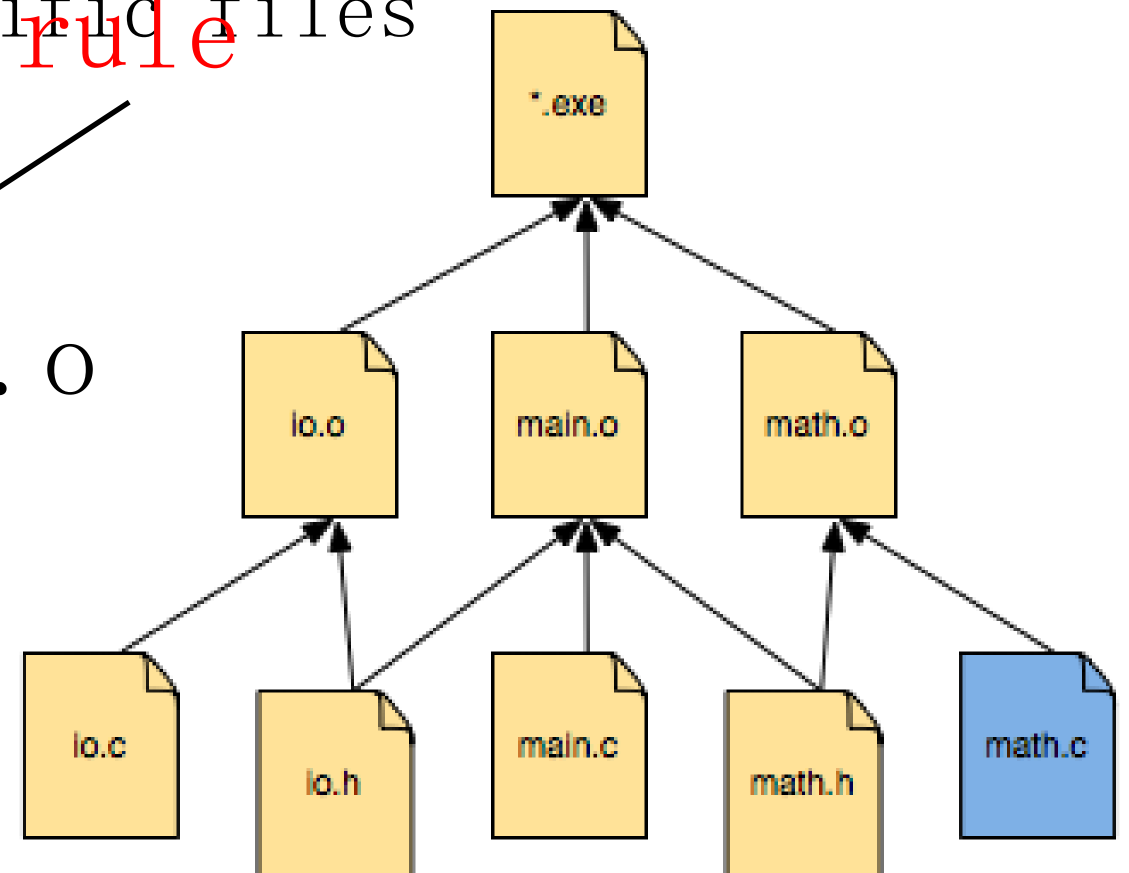
- Define a set of rules to follow
  - Used mainly for source code **compilation**
  - Helpful during development phase when small changes are made to specific files

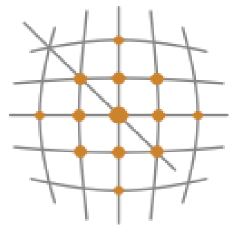
target dependancies rule

```
math.o: math.c math.h
    gcc -c math.c -o math.o
```

or

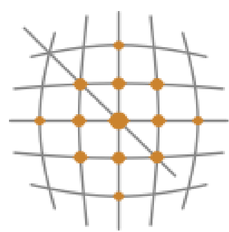
```
math.o: math.c math.h
    gcc -c $< -o $@
```



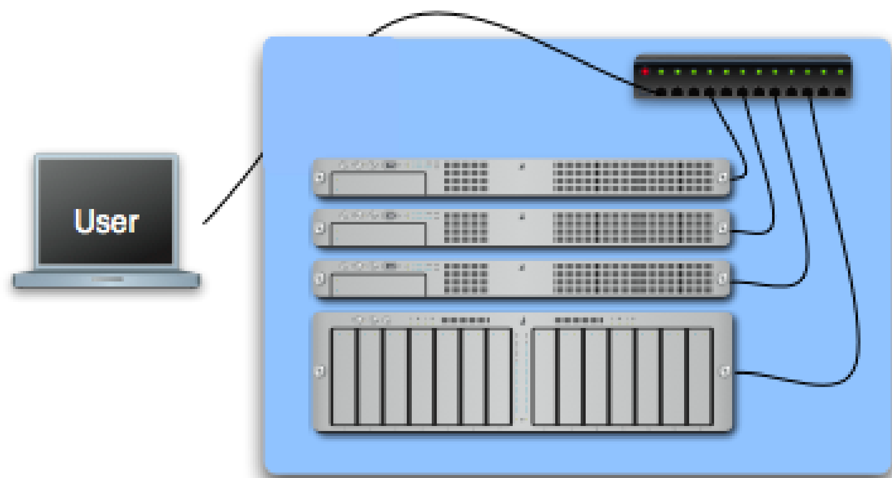


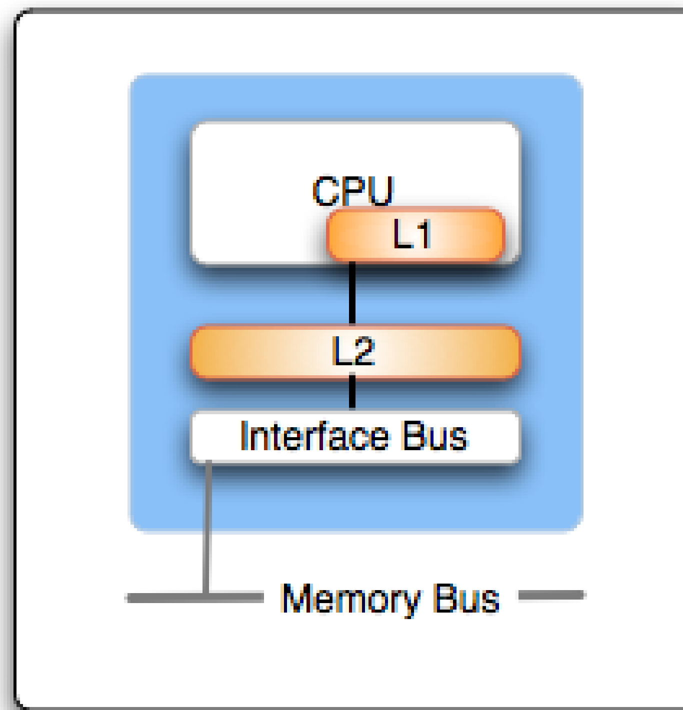
Overview of Parallel computing

- In parallel computing a program spawns several concurrent processes
  - decrease the runtime needed to solve a problem or
  - increase the problem size to be solved
- The original problem is decomposed into tasks that ideally run independently
- Source code development within some parallel programming environment
  - hardware platform
  - nature of the problem
  - performance goals

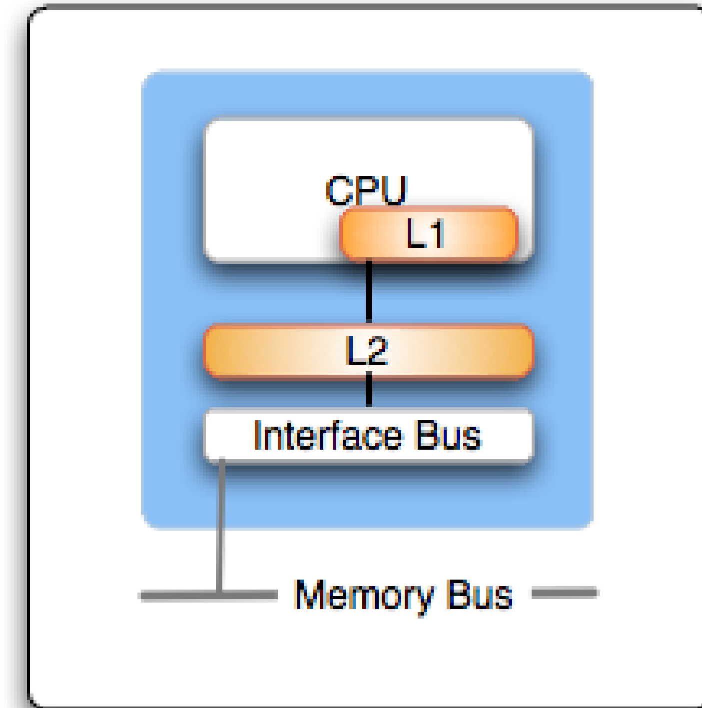


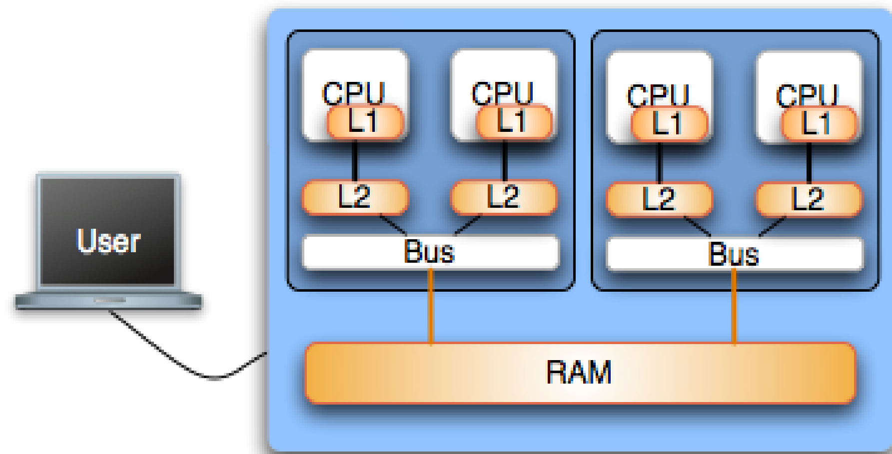
Hardware considerations

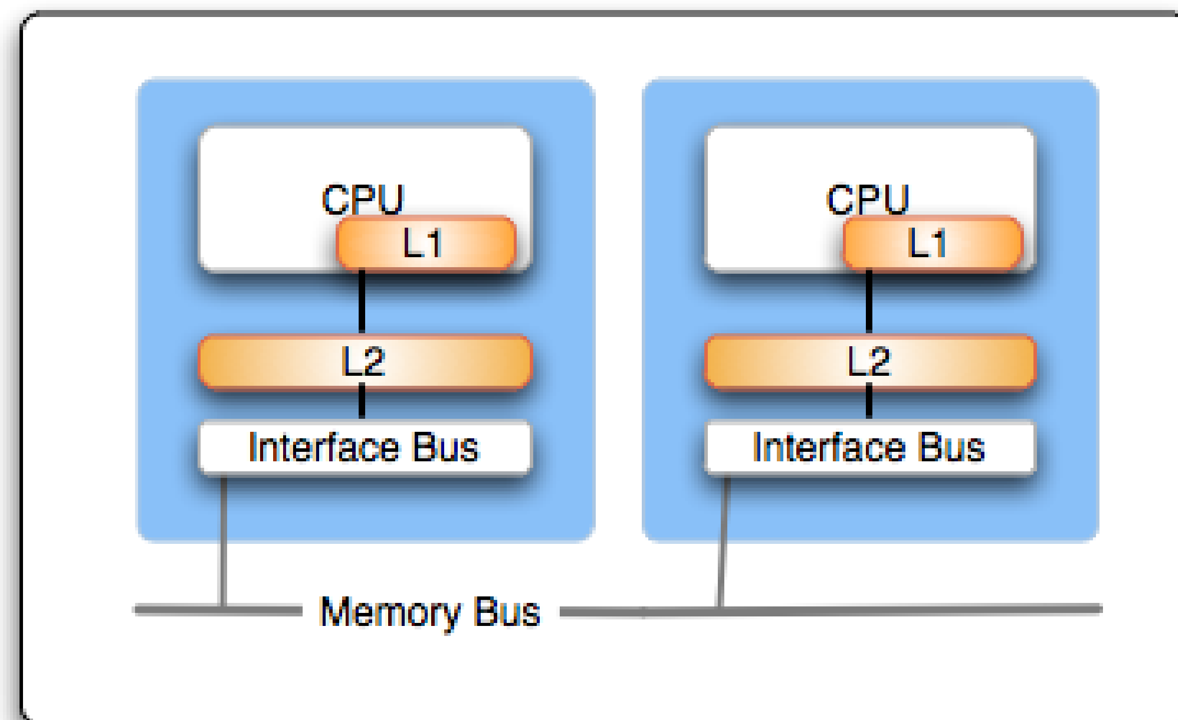




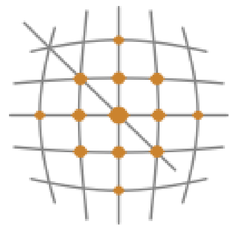
LAN





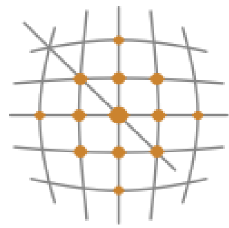


Scientific Computing Center
Aristotle University of Thessaloniki
http://www.grid.auth.gr



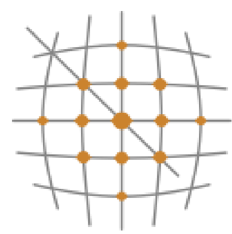
Hardware considerations

- Distributed memory systems
  - Each process (or processor) has unique address space
  - Direct access to another processors memory not allowed
  - Process synchronization occurs implicitly
- Shared memory systems
  - Processors share the same address space
  - knowledge of where data is stored is of no concern to the user
  - Process synchronization is explicit
  - Not scalable



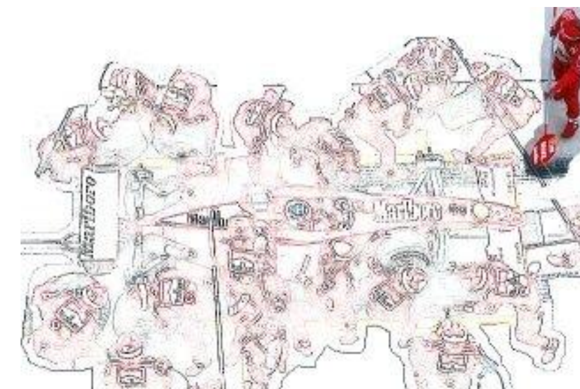
Parallel programming models

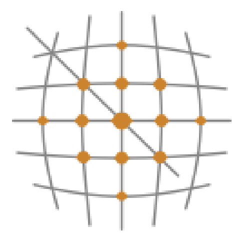
- Distributed memory systems
  - Programmer uses “Message Passing” in order to sync processes and share data among them
  - Message passing libraries
    - MPI
    - PVM
- Shared memory systems
  - Thread based programming approach
  - Compiler directives (i.e. OpenMP)
  - Message passing may also be used



Nature of the problem

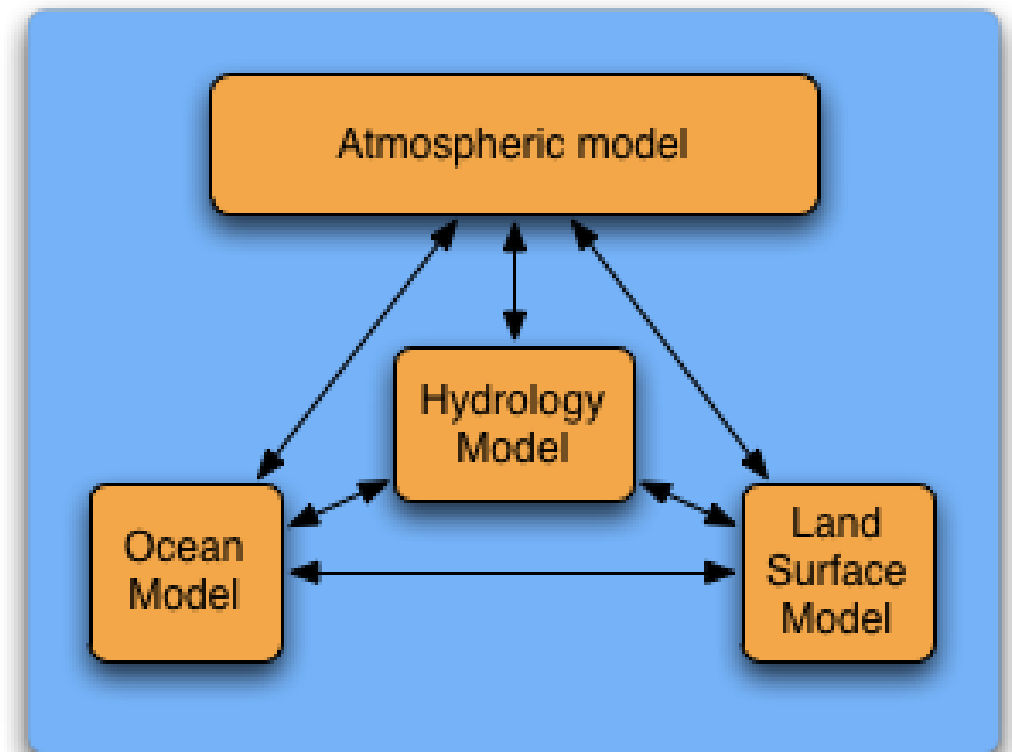
- Parallel computing
  - Embarrassingly parallel (**parametric** studies)
  - Multiple processes concurrently (**Domain** and/or **functional decomposition**)



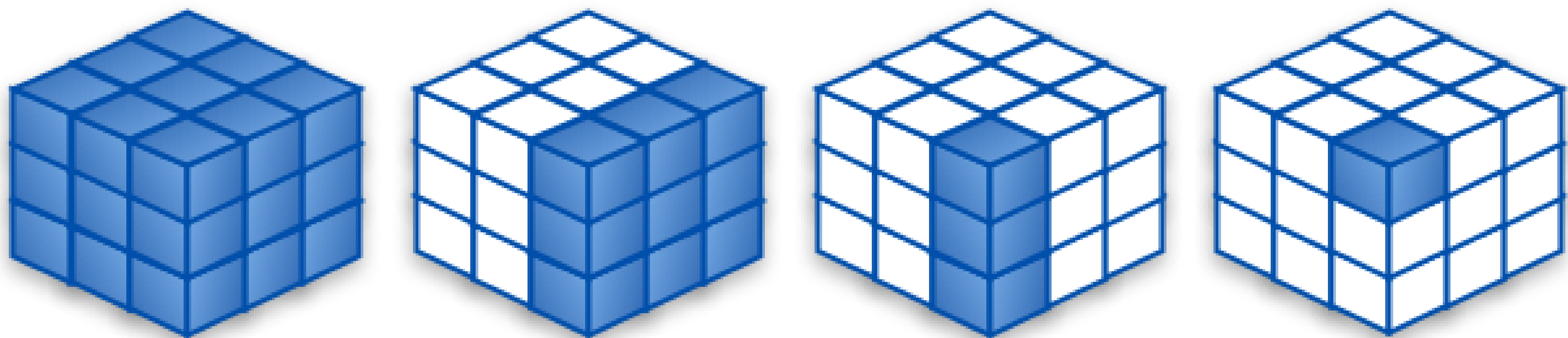


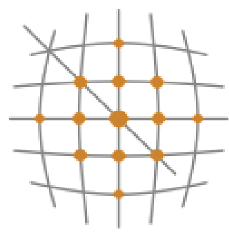
Nature of the problem (examples)

- Functional partitioning



- Domain decomposition





Amdahl’s Law

Amdahl’s Law

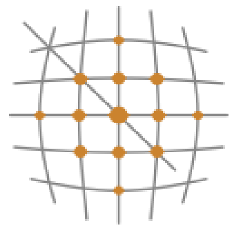
Parallel Portion
50%
75%
90%
95%

Speedup
20.00
18.00
16.00
14.00
12.00
10.00
8.00
6.00
4.00
2.00
0.00

1 2 4 8 16 32 64 128 256 512 1024 2048 4096 8192 16384 32768 65536

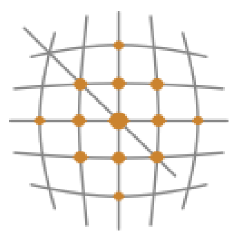
Number of Processors

Amdahl’s law predicts the theoretical maximum speedup when using multiple processors

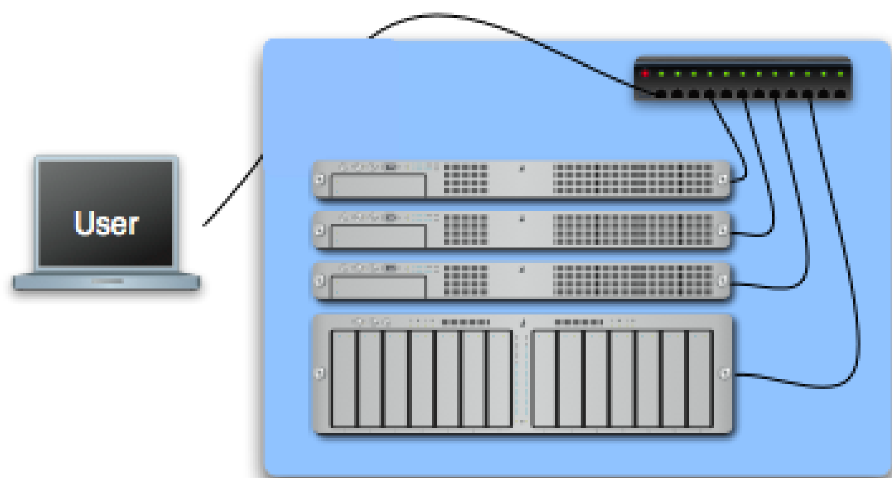


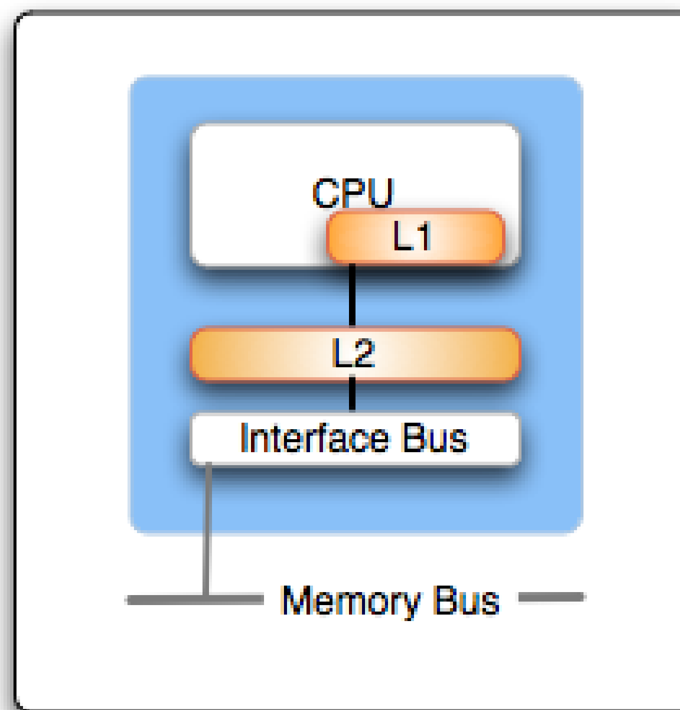
Overview of OpenMP

- Shared vs Distributed memory models
- Why OpenMP
- How OpenMP works
- Basic examples
- How to execute the executable

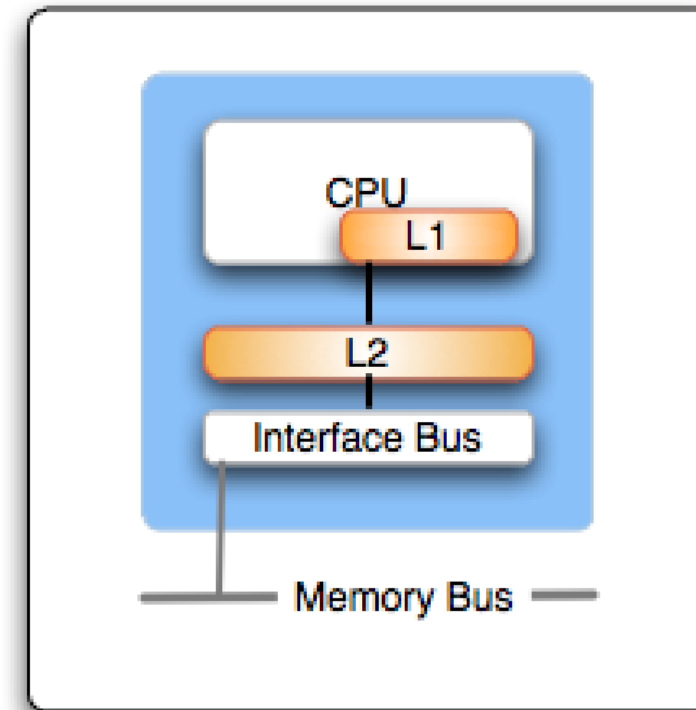


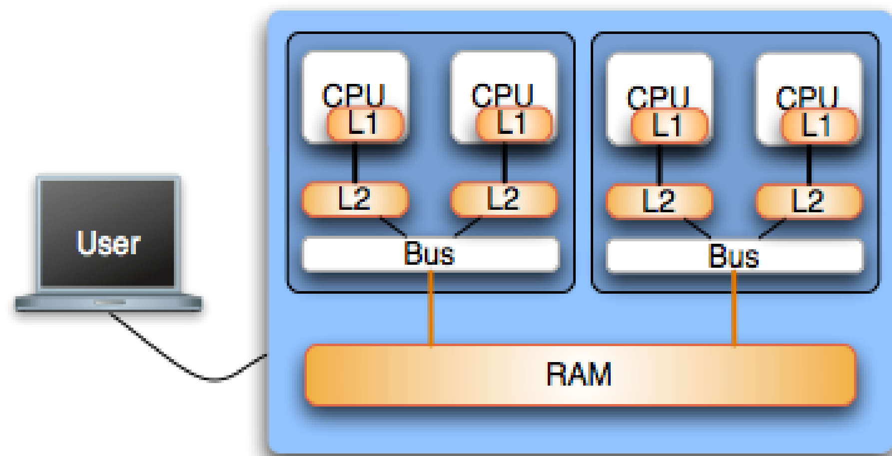
Hardware considerations

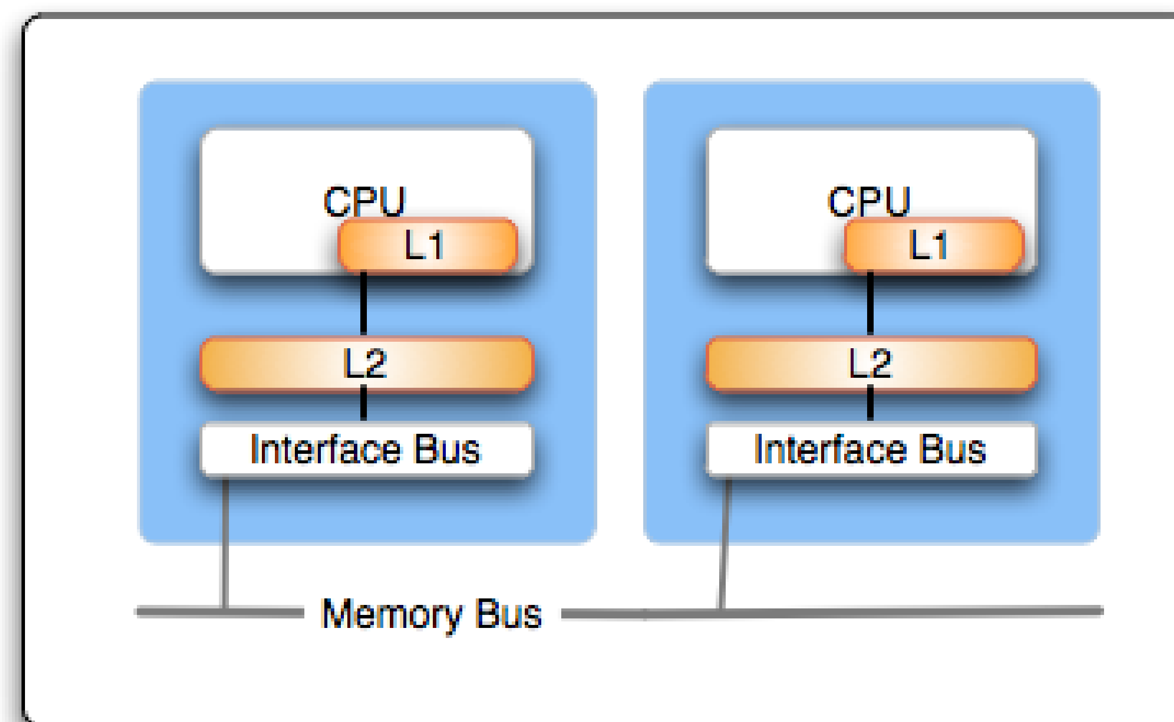




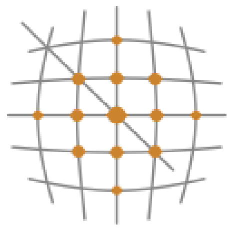
LAN





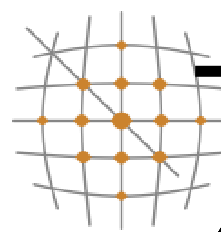


Scientific Computing Center
Aristotle University of Thessaloniki
http://www.grid.auth.gr



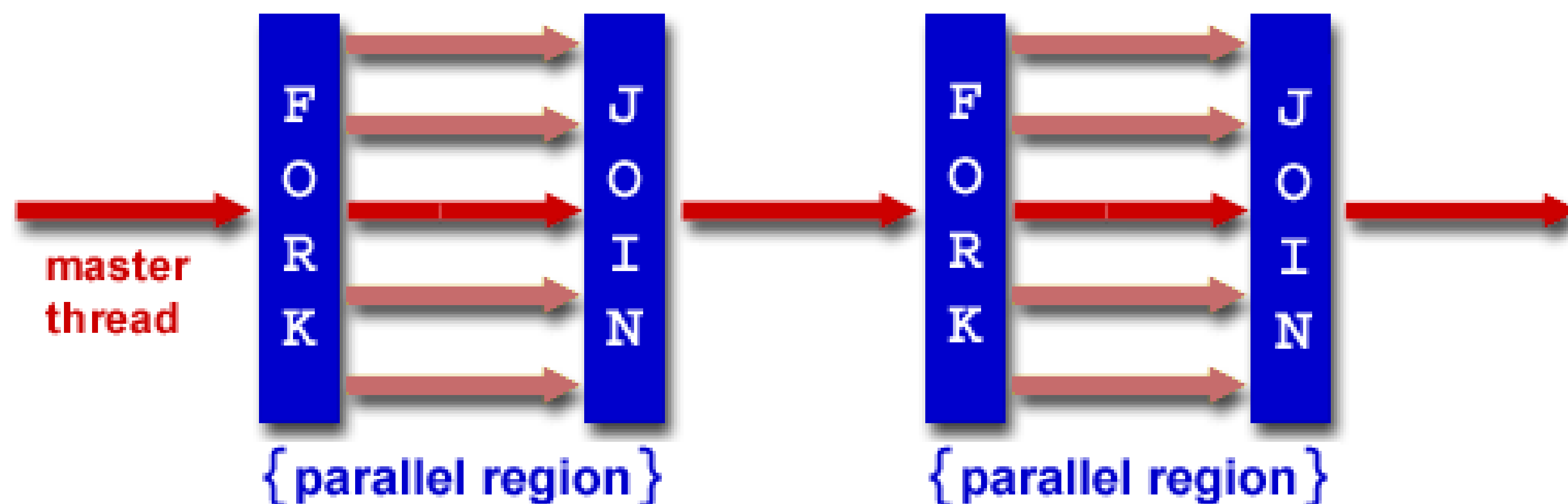
Introduction to OpenMP

- API extension to C/C++ and Fortran languages
  - Most compilers support OpenMP
    - GNU, IBM, Intel, PGI, PathScale, Open64 …
- Extensively used for writing programs for shared memory architectures over the past decade
- Thread (process) communication is implicit and uses variables pointing to shared memory locations; this is in contrast with MPI which uses explicit messages passed among each process

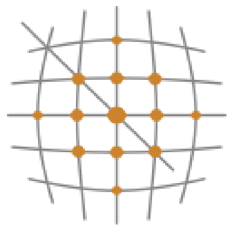


Threaded parallel programming (openMP)

- openMP is based on a fork - join model
  - Master - worker threads

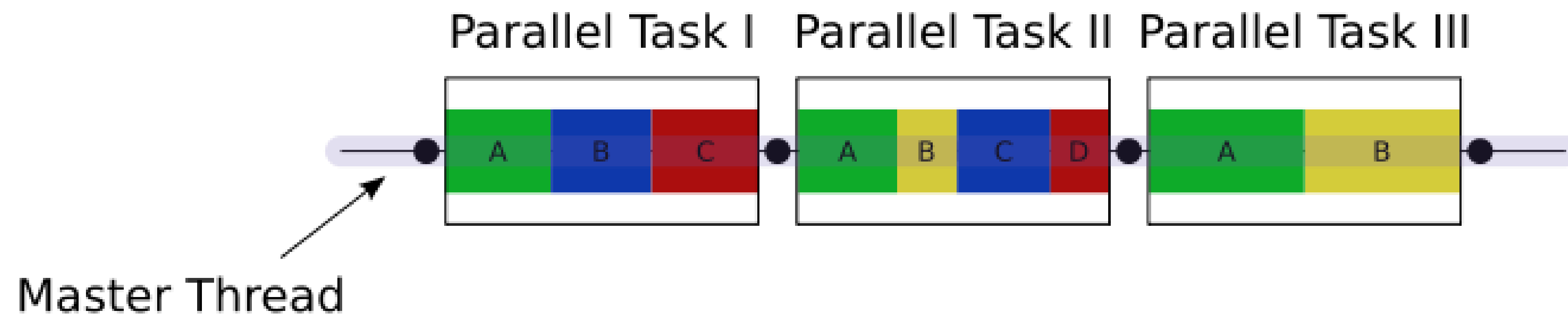


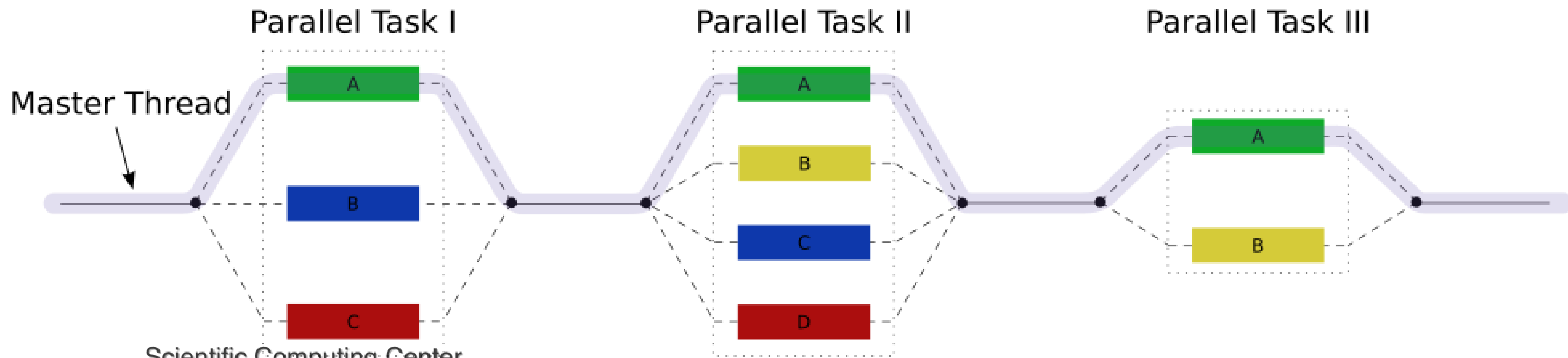
Use of directives and pragmas within source code

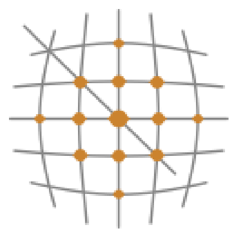


Approach towards parallelism

- From serial to parallel with OpenMP

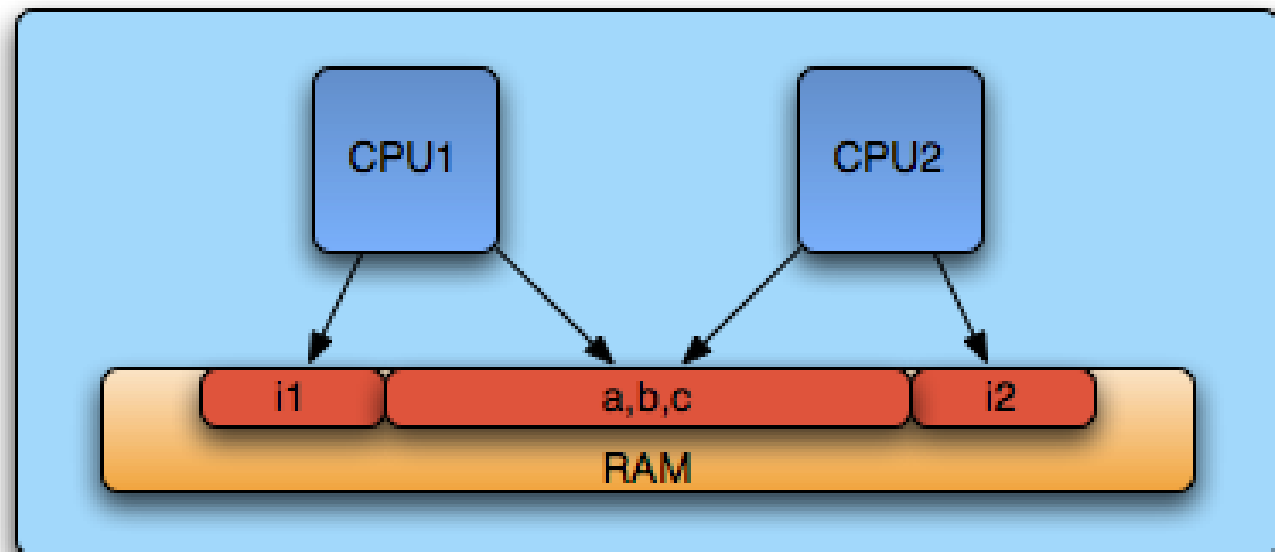


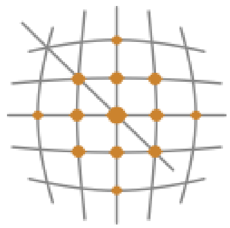




Memory issues

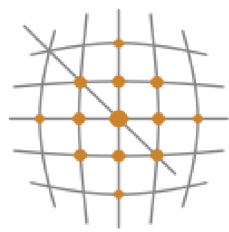
- Threads have access to the same address space
  - Communication is implicit
- Programmer needs to define
  - local data
  - shared data





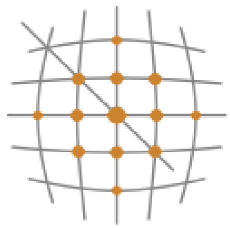
Hello world example

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

using namespace std;

int main(int argc, char* argv[])
{
  #pragma omp parallel
  {
    printf("Hello World! This is thread %d out of %d\n",
               omp_get_thread_num(), omp_get_num_threads());
  }
  return 0;
}
```



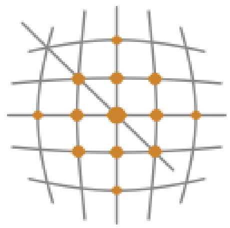
Common API calls

Call	Description
int omp_get_num_threads()	Returns the number of threads in the concurrent team
int omp_get_thread_num()	Returns the id of the thread inside the team
int omp_get_num_procs()	Returns the number of processors in the machine
int omp_get_max_threads()	Returns maximum number of threads that will be used in the next parallel region
double omp_get_wtime()	Returns the number of seconds since a time in the past
bool omp_in_parallel()	1 if in parallel region, 0 otherwise



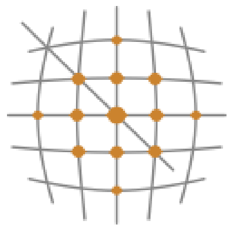
Common API calls example

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

using namespace std;

int main(int argc, char* argv[])
{
  double start = omp_get_wtime();
  if( !omp_in_parallel() )
  {
    printf("Number of processors is: %d\n", omp_get_num_procs());
    printf("Number of max threads is: %d\n", omp_get_max_threads());
  }
  sleep(1);
  double end = omp_get_wtime();
  printf("start = %.16g\nend = %.16g\ndiff = %.16g\n",
             start, end, end - start);
  return 0;
}
```



Data scoping

- For each parallel region the data environment is constructed through a number of clauses
  - shared (variable is common among threads)
  - private (variable inside the construct is a new variable)
  - firstprivate (variable is new but initialized to its original value)
  - default (used to set overall defaults for construct)
  - lastprivate (variable’s last value is copied outside construct)
  - reduction (variable’s value is reduced at the end)



A few examples

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

x = 3
x = 2
x = 3

or

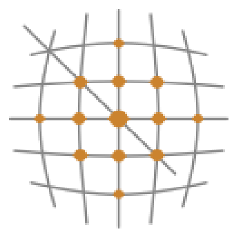
x = 2
x = 3
x = 3

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

Will print anything and then x=1

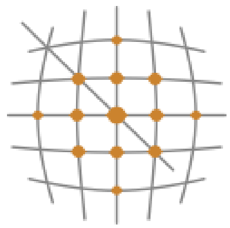
```
int x=1;
#pragma omp parallel firstprivate(x) num_threads(2)
{
   x++;
   printf("%d\n",x);
}
printf("%d\n",x);
```

x = 2
x = 2
x = 1



Synchronization

- OpenMP provides several synchronization mechanisms
  - barrier (synchronizes all threads inside the team)
  - master (only the master thread will execute the block)
  - critical (only one thread at a time will execute)
  - atomic (same as critical but for one memory location)



Synchronization examples

```
int x=1;
#pragma omp parallel num_threads(2)
{
    x++;
    #pragma omp barrier
    foo(x);
}
```

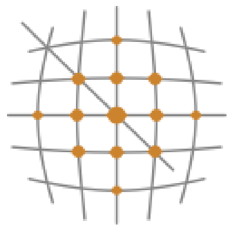
foo(3), foo(3)

```
int x=1;
#pragma omp parallel num_threads(2)
{
    #pragma omp master
    {
        x++;
    }
    foo(x);
}
```

foo(2), foo(2) or foo(1), foo(2)

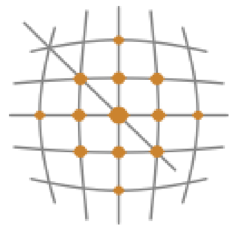
```
int x=1;
#pragma omp parallel num_threads(2)
{
  #pragma omp critical
  {
    x++;
    foo(x);
  }
}
```

foo(2), foo(3)



Data parallelism

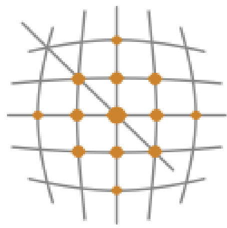
- Worksharing constructs
  - Threads cooperate in doing some work
  - Thread identifiers are not used explicitly
  - Most common use case is loop worksharing
  - Worksharing constructs may not be nested
- DO/for directives are used in order to determine a parallel loop region



The for loop directive

```
#pragma omp for  [clauses]
for (iexpr ; test ; incr)
```

- Where clauses may be
  - private, firstprivate, lastprivate
  - Reduction
  - Schedule
  - Nowait
- Loop iterations must be independent
- Can be merged with parallel constructs
- Default data sharing attribute is shared

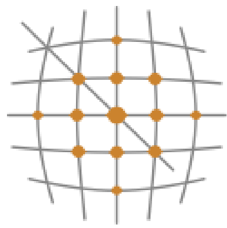


```
int i,j;
#pragma omp parallel
#pragma omp for private(j)
for(i=0; i<N; i++)
{
    for(j=0; j<N; j++)
        m[i][j] = f(i,j);
}
```

j must be declared private explicitly

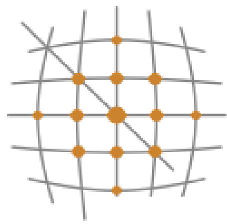
i is privatized automatically

Implicit synchronization point at the end of for loop



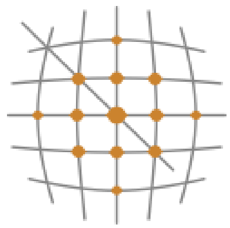
The schedule clause

- Schedule clause may be used to determine the distribution of computational work among threads
    - static, chunk; The loop is equally divided among pieces of size chunk which are evenly distributed among threads in a round robin fashion
    - dynamic, chunk; The loop is equally divided among pieces of size chunk which are distributed for execution dynamically to threads. If no chunk is specified chunk=1
    - guided; similar to dynamic with the variation that chunk size is reduced as threads grab iterations
- Configurable globally via OMP_SCHEDULE
    - i.e. setenv OMP_SCHEDULE "dynamic,4"



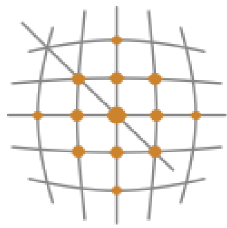
Adding two vectors

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

using namespace std;

int main(int argc, char **argv)
{
  int n = atoi(argv[1]);
  double *x, *y;

  x = new double [n]; for(int i=0; i<n; i++) x[i] = (double) (i+2);
  y = new double [n]; for(int i=0; i<n; i++) y[i] = (double) (i*3);

  double start = omp_get_wtime();
  #pragma omp parallel for
    for (int i=0; i<n; i++) x[i] = x[i] + y[i];
  double end = omp_get_wtime();
  printf("diff = %.16g\n", end - start);

  return 0;
}
```



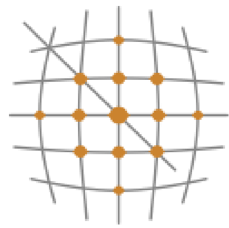
Reduction clause

- Useful in the case one variable's value is accumulated within a loop
- Using the reduction clause
  - A private copy per thread is created and initialized
  - At the end of the region the compiler safely updates the shared variable
  - Operators may be +, *, -, /, &, ^, |, &&, ||



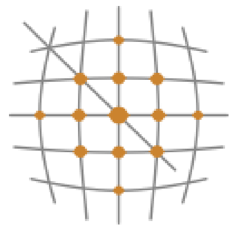
Reduction clause example

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

using namespace std;

int main(int argc, char* argv[])
{
	// declerations
	int i,N=atoi(argv[1]);
	vector <double> A(N);
	double s;
	// calculations
	#pragma omp parallel for shared(A,N) private(i)
	for(i=0; i<N; i++)
	{
		A[i] = pow(cos((double) i),2)/3.0 - 1.0/sqrt((double) (i+1));
	}
	#pragma omp parallel for shared(A,N) private(i) reduction(+:s)
	for(i=0; i<N; i++)
	{
		s += A[i];
	}
	cout << "Total sum is s = "<< s << endl;
	return 0;
}
```



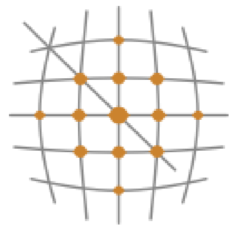
Message Passing Model

- A process may be defined as a program counter and an address space
- Each process may have multiple threads sharing the same address space
- Message Passing is used for communication among processes
  - synchronization
  - data movement between address spaces



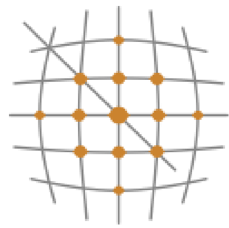
Message Passing Interface

- MPI is a message passing library specification
  - not a language or compiler specification
  - no specific implementation
- Source code portability
  - SMPs
  - clusters
  - heterogenous networks



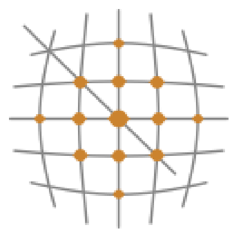
Types of communication

- Initialization, Finalization and Synchronization calls
- Point-to-Point calls
  - data movement
- Collective calls
  - data movement
  - reduction operations
  - synchronization



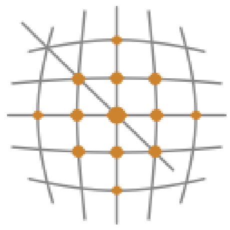
MPI Features

- Point-to-point communication
- Collective communication
- One-sided communication
- Communicators
- User defined datatypes
- Virtual topologies
- MPI-I/O



Basic MPI

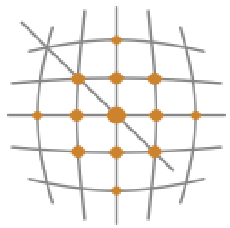
- MPI_Init
- MPI_Comm_size (get number of processes)
- MPI_Comm_rank (gets a rank value assigned to each process)
- MPI_Send (cooperative point-to-point call used to send data to receiver)
- MPI_Recv (cooperative point-to-point call used to receive data from sender)
- MPI_Finalize



Hello World!

```fortran
program hello
include 'mpif.h'
integer rank, size, ierror

call MPI_INIT(ierror)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
print*, 'Hello world! This is process ', rank, ' out of ', size
call MPI_FINALIZE(ierror)
end
```

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

using namespace std;

int main (int argc, char* argv[])
{
  int rank,size;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  MPI_Comm_size( MPI_COMM_WORLD, &size );
  printf( "Hello world! This is process %d out of %d\n", rank, size );
  MPI_Finalize();
  return 0;
}
```

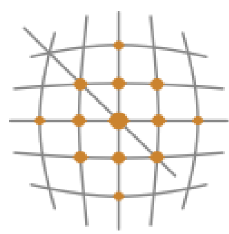


Hello World!

```fortran
program hello
include 'mpif.h'
integer rank, size, ierror

call MPI_INIT(ierror)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
print*, 'Hello world! This is process ', rank, ' out of ', size
call MPI_FINALIZE(ierror)
end
```

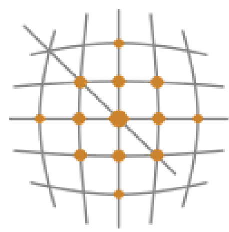
```cpp
#include <iostream>
#include <mpi.h>

using namespace std;

int main (int argc, char* argv[])
{
  int rank,size;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  MPI_Comm_size( MPI_COMM_WORLD, &size );
  printf( "Hello world! This is process %d out of
  MPI_Finalize();
  return 0;
}
```

```fortran
if ( ierror .ne. MPI_SUCCESS ) then
    ... do error handling as desired ...
end if
```



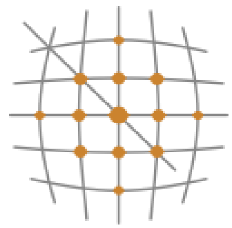
Starting and exiting the MPI environment

- MPI_Init
  - C style: int MPI_Init(int *argc, char ***argv);
    - accepts argc and argv variables (main arguments)
  - F style: MPI_INIT (IERROR)
    - Almost all Fortran MPI library calls have an integer return code
  - Must be the **first** MPI function called in a program
- MPI_Finalize
  - C style: int MPI_Finalize();
  - F style: MPI_FINALIZE (IERROR)



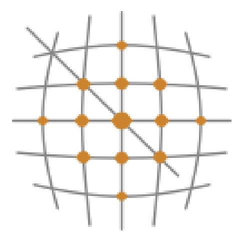
Communicators

- All mpi specific communications take place with respect to a communicator
  - Communicator: A collection of processes and a context
- MPI_COMM_WORLD is the predefined communicator of all processes
- Processes within a communicator are assigned a unique rank value

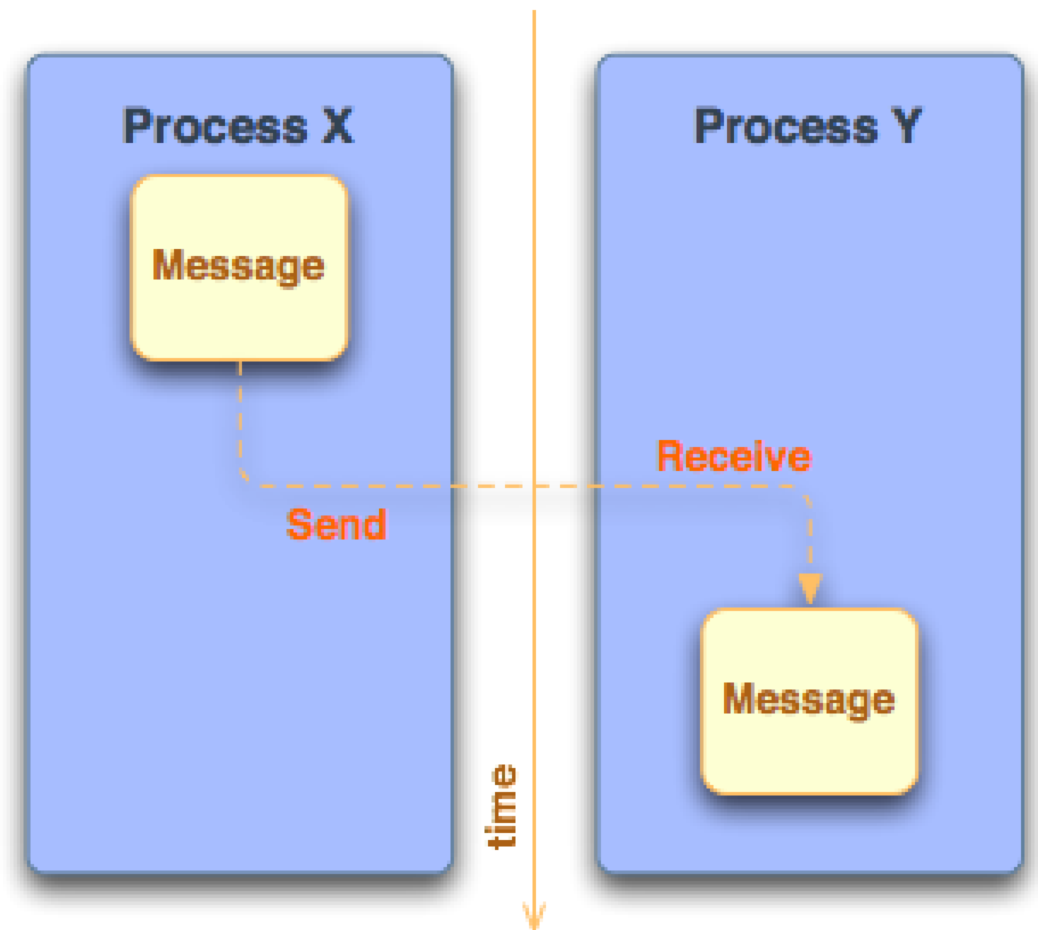


A few basic considerations

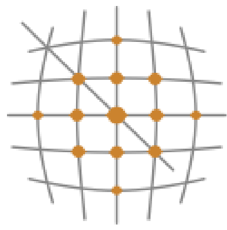
- Q: How many processes are there? A: (N)
  - (C) `MPI_Comm_size( MPI_COMM_WORLD, &size );`
  - (F) `MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr)`
- Q: Which one is which? A: [0,(N-1)]
  - (C) `MPI_Comm_rank( MPI_COMM_WORLD, &rank );`
  - (F) `MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr)`
  - The rank number is between 0 and (size - 1) unique per process



Sending and receiving messages

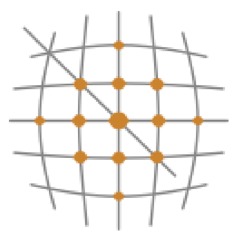


- Questions
  - Where is the data?
  - What type of data?
  - How much data is sent?
  - To whom is the data sent?
  - How does the receiver know which data to collect?



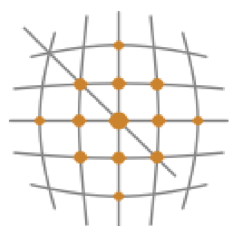
What is contained within a message?

- message data
  - buffer
  - count
  - datatype
- message envelope
  - source/destination rank
  - message tag (tags are used to discriminate among messages)
  - communicator



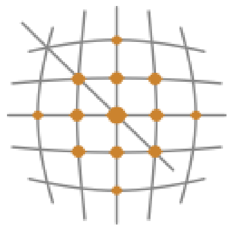
MPI Standard (Blocking) Send/Receive

- Syntax
  - `MPI_Send(void *buffer, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm);`
  - `MPI_Recv(void *buffer, int count, MPI_Datatype type, int src, int tag, MPI_Comm comm, MPI_Status status);`
- Processes are identified using dest/src values and the communicator within the message passing takes place
- Tags are used to deal with multiple messages in an orderly manner
  - MPI_ANY_TAG and MPI_ANY_SOURCE may be used as wildcards on the receiving process



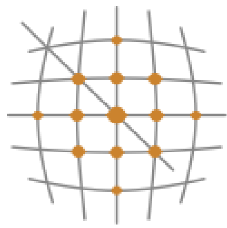
MPI Datatypes

C Data Types		Fortran Data Types	
MPI_CHAR	signed char	MPI_CHARACTER	character(1)
MPI_SHORT	signed short int		
MPI_INT	signed int	MPI_INTEGER	integer
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_REAL	real
MPI_DOUBLE	double	MPI_DOUBLE_PRECISION	double precision
MPI_LONG_DOUBLE	long double		
		MPI_COMPLEX	complex
		MPI_DOUBLE_COMPLEX	double complex
		MPI_LOGICAL	logical
MPI_BYTE	8 binary digits	MPI_BYTE	8 binary digits
MPI_PACKED	data packed or unpacked with MPI_Pack()/ MPI_Unpack	MPI_PACKED	data packed or unpacked with MPI_Pack()/ MPI_Unpack



Getting information about a message

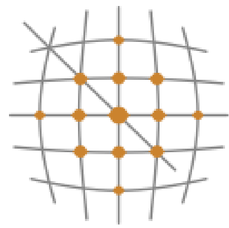
- Information of source and tag is stored in MPI_Status variable
  - status.MPI_SOURCE
  - status.MPI_TAG
- MPI_Get_count can be used to determine how much data of a particular type has been received
  - MPI_Get_count(&status, MPI_Datatype, &count);



Yet another listing (deadlock?)

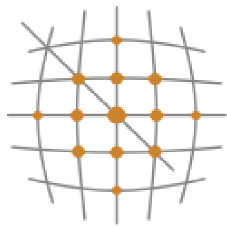
```
Program Output:
from process 1 using tag 158 buf1[N-1] = 8.1
from process 0 using tag 157 buf0[N-1] = 0.9
```

```
double buf0[N], buf1[N];
MPI_Status status;
int tag_of_message, src_of_message;
if( rank == 0 )
{
        for(int i=0; i<N; i++)     buf0[i] = 0.1 * (double) i;
        MPI_Send(buf0, N, MPI_DOUBLE, 1, 157, MPI_COMM_WORLD);
        MPI_Recv(buf1, N, MPI_DOUBLE, 1, 158, MPI_COMM_WORLD, &status);
        tag_of_message = status.MPI_TAG;
        src_of_message = status.MPI_SOURCE;
        cout << "from process " << src_of_message << " using tag " <<
                tag_of_message << " buf1[N-1] = " << buf1[N-1] << endl;
}
else if ( rank == 1 )
{
        for(int i=0; i<N; i++)     buf1[i] = 0.9 * (double) i;
        MPI_Send(buf1, N, MPI_DOUBLE, 0, 158, MPI_COMM_WORLD);
        MPI_Recv(buf0, N, MPI_DOUBLE, 0, 157, MPI_COMM_WORLD, &status);
        tag_of_message = status.MPI_TAG;
        src_of_message = status.MPI_SOURCE;
        cout << "from process " << src_of_message << " using tag " <<
                tag_of_message << " buf0[N-1] = " << buf0[N-1] << endl;
}
```



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)
- MPI uses internal buffers (the envelope) to pack messages, thus short messages do not produce deadlocks
- To avoid deadlocks one either reverses the Send/Receive calls on one end or uses the Non-Blocking calls (MPI_Isend or MPI_Irecv respectively followed by MPI_Wait)

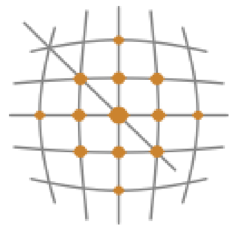


Communication Types

- **Blocking**: If a function performs a blocking operation, then it will not return to the caller until the operation is complete.

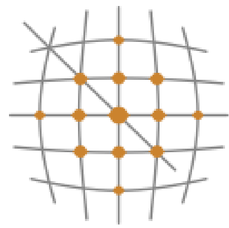
- **Non-Blocking**: If a function performs a non-blocking operation, it will return to the caller as soon as the requested function has been initialized.

- Using non-blocking communication allows for higher program efficiency <u>if calculations can be performed while communication activity is going on</u>. This is referred to as **overlapping computation with communication**



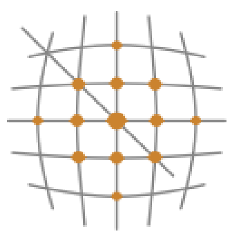
Communication (i.e. Send) Modes

- Standard mode (MPI_Send, MPI_ISend)
  - Send will complete when buffer is available for use
- Synchronous mode (MPI_Ssend, MPI_Issend)
  - The send will complete only until a matching receive has been posted and transfer has started
- Buffered mode (MPI_Bsend, MPI_Ibsend)
  - Send is complete as soon as the user buffer is copied to the system buffer
- Ready mode (MPI_Rsend, MPI_Irsend)
  - send starts only if a matching receive has been posted

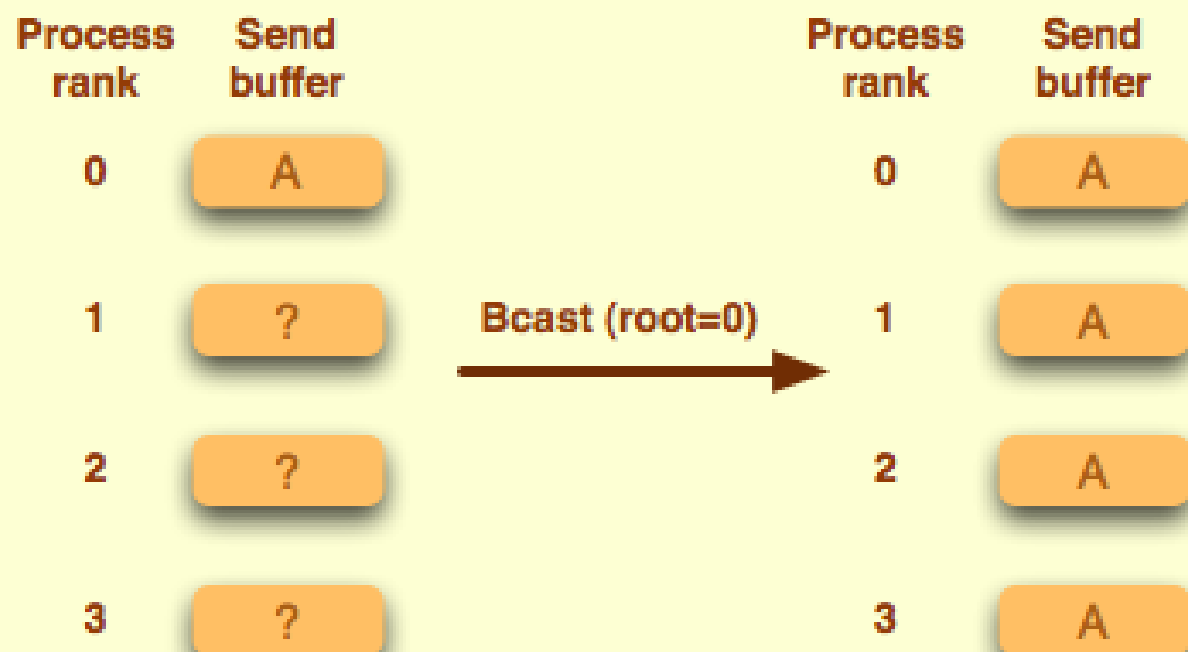


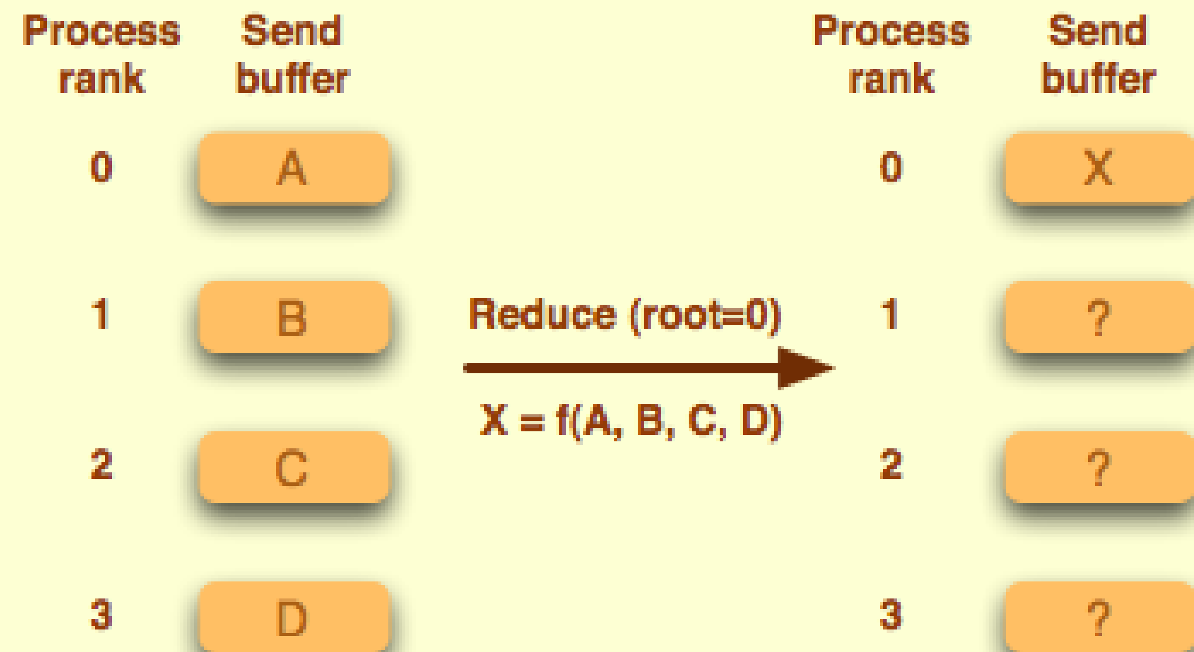
Collective communications

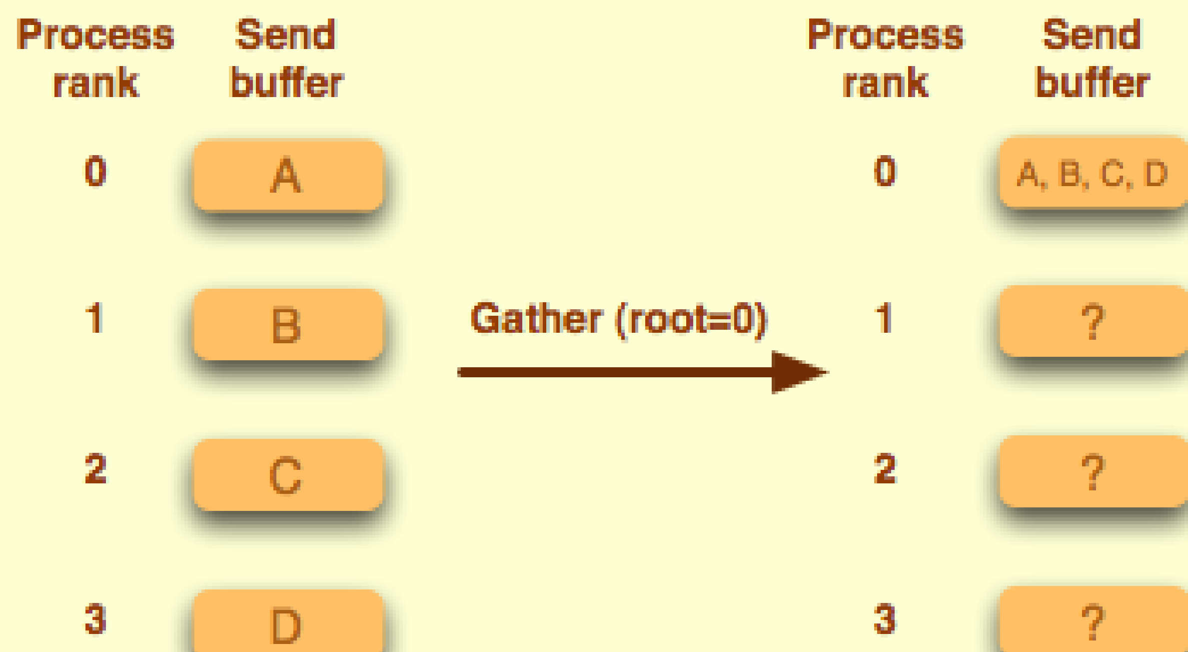
- All processes within the specified communicator participate
- All collective operations are blocking
- All processes must call the collective operation
- No message tags are used
- Three classes of collective communications
  - Data movement
  - Collective computation
  - Synchronization

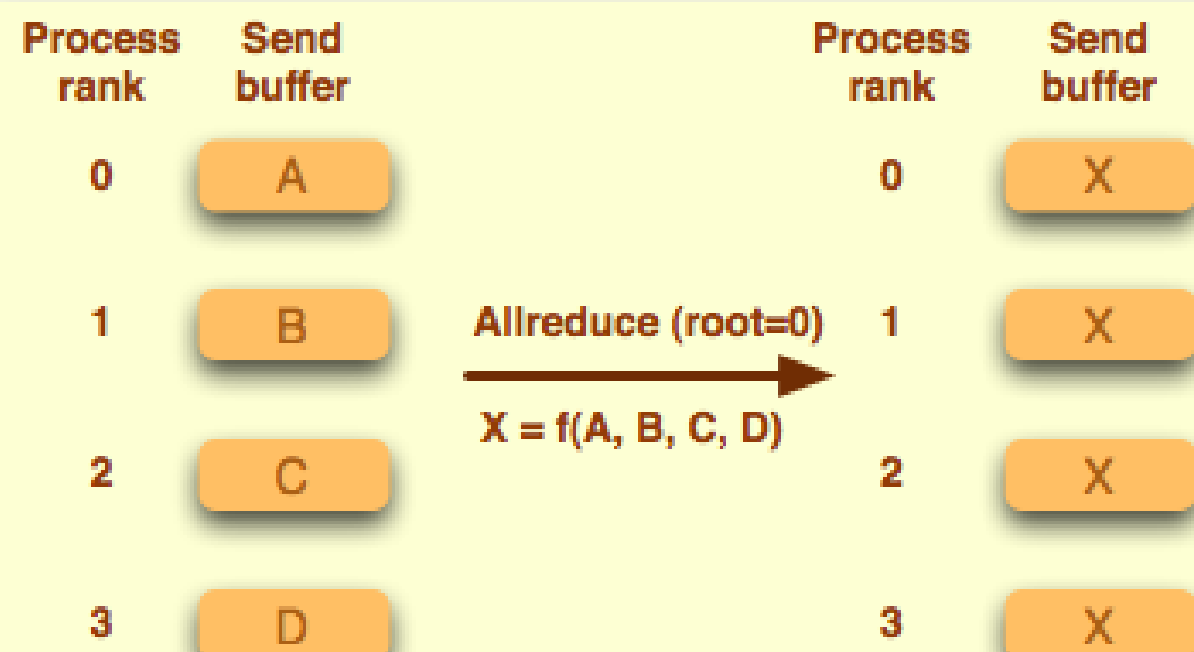


Examples of collective operations



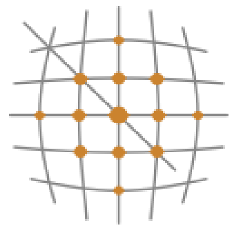






Synchronization

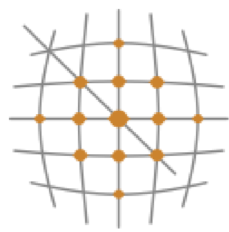
- MPI_Barrier (comm)
- Execution blocks until all processes in comm call it
- Mostly used in highly asynchronous programs



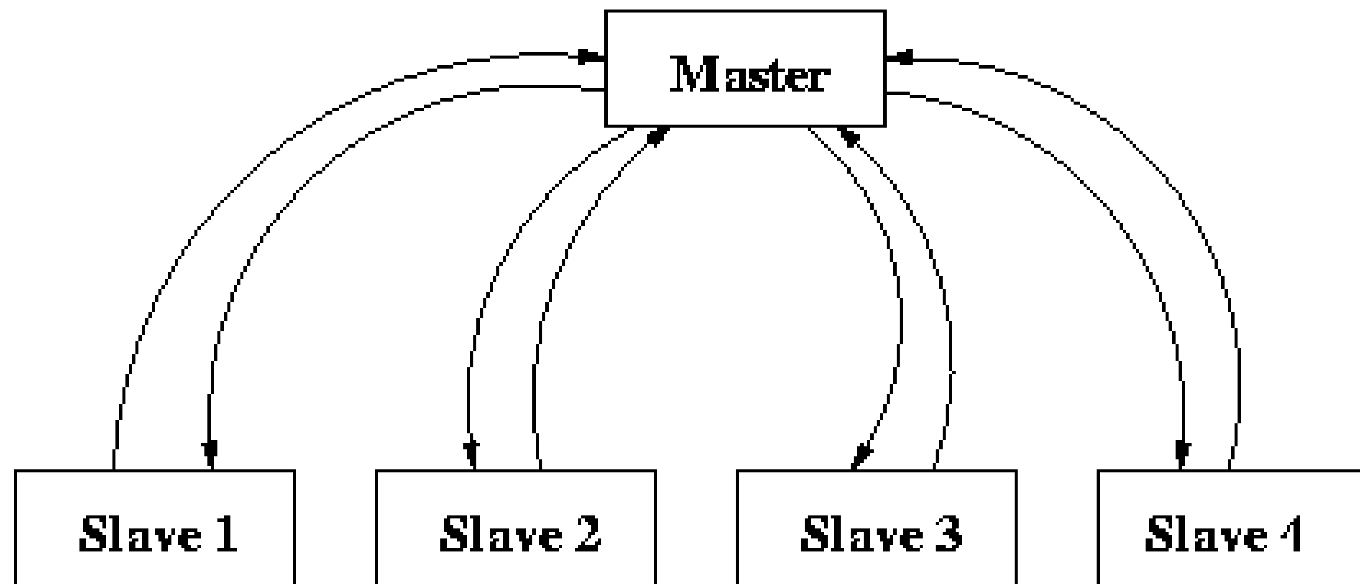
Timing using MPI

- MPI_Wtime returns number of seconds since an arbitrary point in the past

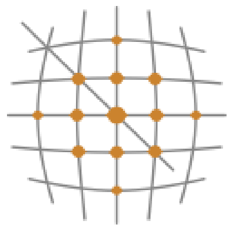
```
double mpi_t0,mpi_t1;
if(rank == 0)
{
        mpi_t0 = MPI_Wtime();
}
sleep(1);
MPI_Barrier( MPI_COMM_WORLD );
if(rank == 0)
{
        mpi_t1 = MPI_Wtime();
        printf("# MPI_time = %f\n", mpi_t1-mpi_t0);
}
```



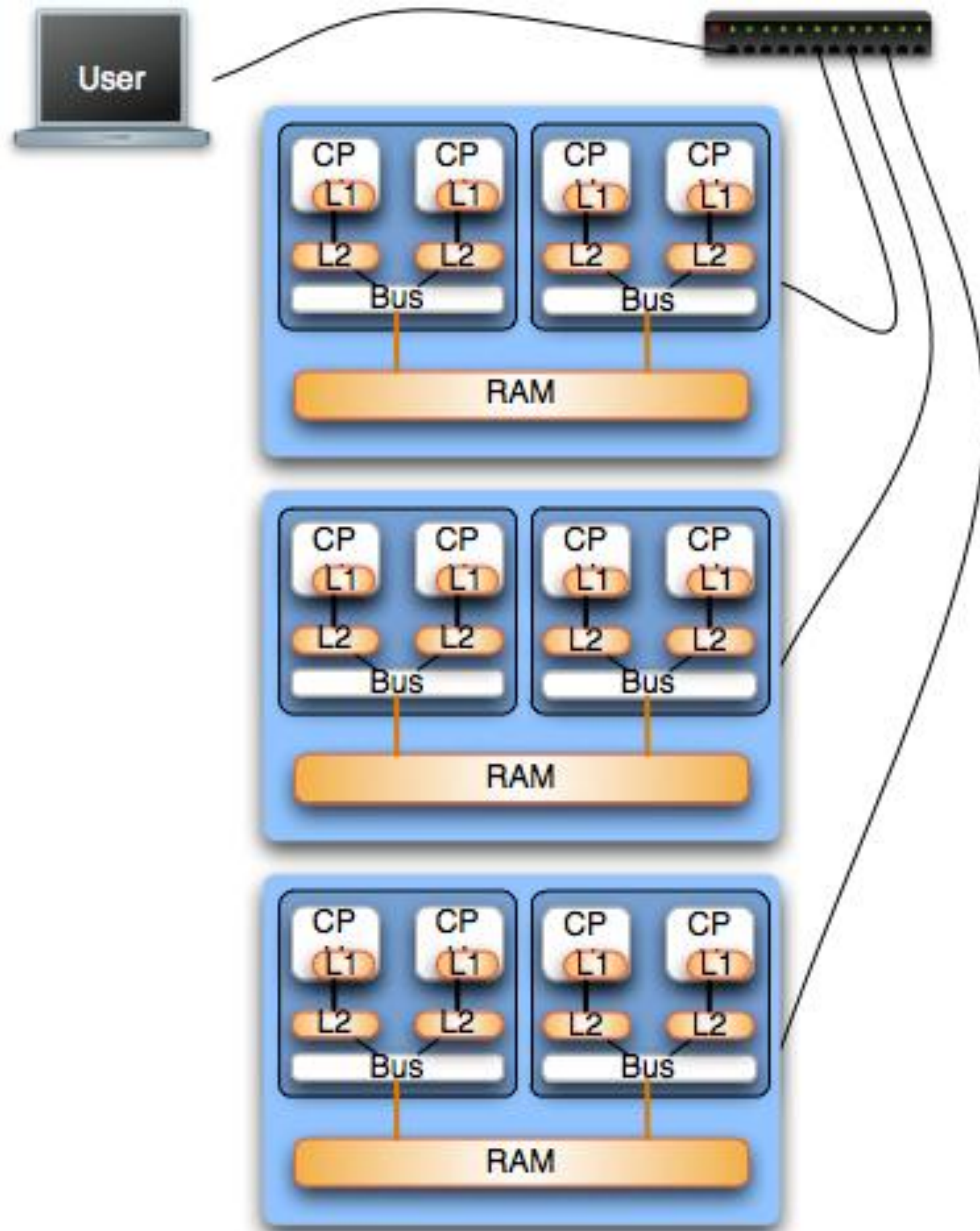
Compile and submit a “mm” job



Master process decomposes matrix a and provides slave processes with input. Each slave process caries ns=nm/sz rows of a and the complete b matrix to carry out computations. Results are sent back to master who prints out timing.



Mixing MPI and OpenMP



- Hybrid architectures
  - Clusters on SMPs
  - HPC Platforms
    - IBM BlueGene (i.e. Jugene)
    - IBM P6 (i.e. Huygens)
- Good starting point
  - Mapping of MPI on nodes (interconnection layer)