Introduction to Parallel Programming using MPI and OpenMP

May/2005

Paulo Marques
Dependable Systems Group
University of Coimbra, Portugal
pmarques@dei.uc.pt

Motivation

- We are interested in the last approach:
  - Add more processors!
    (We don't care about being *too* smart or spending *too much* $$$ in bigger faster machines!)
- Why?
  - It may no be feasible to find better algorithms
  - Normally, faster, bigger machines are very expensive
  - There are lots of computers available in any institution (especially at night)
  - There are computer centers from where you can buy parallel machine time
  - Adding more processors enables you not only to run things faster, but to run bigger problems

Motivation

- I have a program that takes 7 days to execute, which is far too long for practical use. How do I make it run in 1 day?

a) Work smarter!
(i.e. find better algorithms)

b) Work faster!
(i.e. buy a faster processor/memory/machine)

c) Work harder!
(i.e. add more processors!!!)

Motivation

- "Adding more processors enables you not only to run things faster, but to run bigger problems"?!
  - "*9 women cannot have a baby in 1 month, but they can have 9 babies in 9 months*"
  - This is (informally) called the Gustafson-Barsis law
  - What this means is that enormous amounts of parallelism may become available when we are trying to run bigger problems. Although each individual task may not run faster, more problems are solved in the same amount of time.

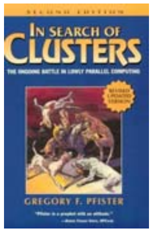
Outline

- Machine Architectures (Briefly)
- Middleware for Parallel Processing
  - Message Passing
  - (Distributed) Shared Memory
  - Parallel Languages
- Types of Parallel Applications
  - Parametric
  - Functional Decomposition and Pipelining
  - Data Parallel (regular/non-regular; task-farming/grid)
- Message Passing: Programming with MPI
- Shared Memory: Programming with OpenMP
- Introduction to Grid Computing

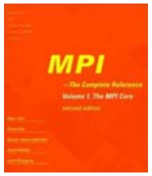
Bibliography – Tutorials

- From the Lawrence Livermore National Laboratory (LLNL): (http://www.llnl.gov/computing/training/index.html)
  - Introduction to Parallel Computing
  - Message Passing Interface (MPI)
  - OpenMP
- From the Edinburgh Parallel Computing Center (EPCC): (http://www.epcc.ed.ac.uk/computing/training/document_archive/)
  - Decomposing the Potentially Parallel Course
  - Writing Message-Passing Parallel Programs with MPI
- From Intel Corporation & Purdue University: (http://www.openmp.org/presentations/index.cgi?sc99_tutorial)
  - OpenMP: An API for Writing Portable SMP Application Software, a tutorial presented at Super Computing'99 by Tim Mattson & Rudolf Eigenmann

Bibliography – Book & Standards



- In Search of Clusters
  Gregory F. Pfister, 2nd ed., January 1998



- MPI, The Complete Reference, Vol. 1: The MPI Core
  Marc Snir et. al., 2nd ed., September 1998
  (http://www.mpi-forum.org/docs/docs.html)
- OpenMP C/C++ Application Programming Interface
  Version 2.0, March 2002.
  (http://www.openmp.org/specs/)

Resources

- General
  - The top 500 fastest computers in the world:
    http://www.top500.org
  - The LINPACK benchmark:
    http://www.netlib.org/benchmark/hpl/
  - The LINPACK benchmark for Windows Clusters
    http://winhpl.dei.uc.pt/
- MPI
  - The MPI forum:
    http://www.mpi-forum.org
  - CriticalSoftware WMPI:
    http://www.criticalsoftware.com/hpc/
  - MPICH MPI:
    http://www-unix.mcs.anl.gov/mpi/mpich/
- OpenMP
  - OpenMP forum:
    http://www.openmp.org
  - General Compilers:
    http://www.openmp.org/index.cgi?resources
  - Intel Compiler:
    http://www.intel.com/software/products/compilers/

Disclaimer

- A great deal of the material found in this course is based on the tutorials, resources and documentation provided in the links listed before. All these materials are freely available on the web, and the copyrights belong to their respective owners.
  - So, don't be surprised if you see some of the diagrams here! ☺

The material presented in this course is largely based on a summer course taught at the University of Coimbra in 2003. At that time, more material was covered. The slides from that course are available at:
http://eden.dei.uc.pt/~pmarques/courses.html

Introduction to Parallel Programming and Grid Computing

Getting Started:

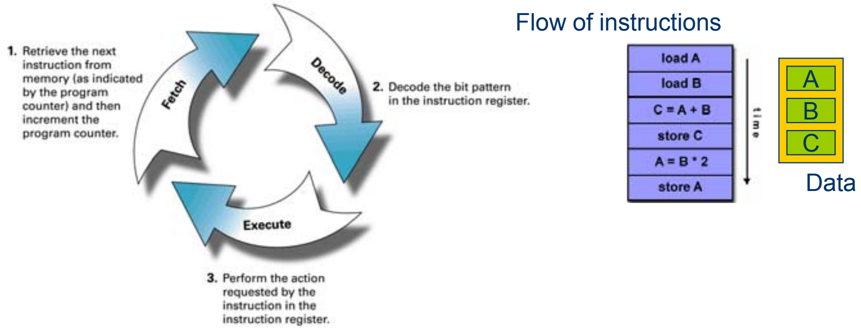
- Machine Architectures

May/2005

Paulo Marques
Dependable Systems Group
University of Coimbra, Portugal
pmarques@dei.uc.pt

von Neumann Architecture

- Based on the fetch-decode-execute cycle
- The computer executes a single sequence of instructions that act on data. Both program and data are stored in memory.


1. Retrieve the next instruction from memory (as indicated by the program counter) and then increment the program counter.
Fetch
Decode
2. Decode the bit pattern in the instruction register.
Execute
3. Perform the action requested by the instruction in the instruction register.
Flow of instructions
load A
load B
C = A + B
store C
A = B * 2
store A
time
A
B
C
Data

Flynn's Taxonomy

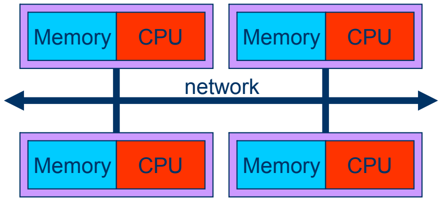
- Classifies computers according to…
  - The number of execution flows
  - The number of data flows

	Number of data flows	
Number of execution flows	SISD *Single-Instruction* *Single-Data*	SIMD *Single-Instruction* *Multiple-Data*
	MISD *Multiple-Instruction* *Single-Data*	MIMD *Multiple-Instruction* *Multiple-Data*

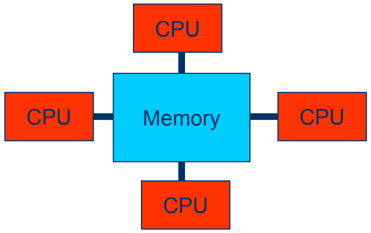
Types of Parallel Machines

- Programs act on data.
- Quite important:
  How do processors access each others' data?

Message Passing Model



Shared Memory Model

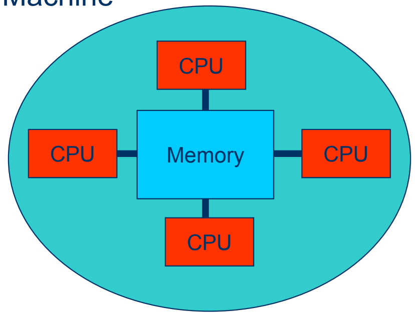


Shared Memory

- Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all memory as a global address space
- Multiple processors can operate independently but share the same memory resources
- Changes in a memory location made by one processor are visible to all other processors
- Shared memory machines can be divided into two main classes based upon memory access times: **UMA** and **NUMA**

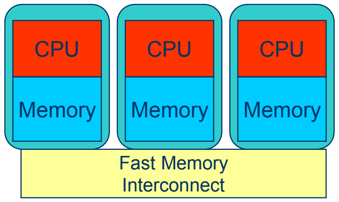
Shared Memory (2)

Single 4-processor Machine



UMA: Uniform Memory Access

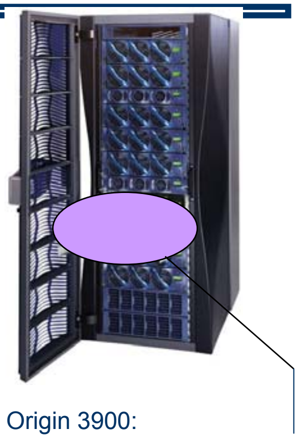
A 3-processor NUMA Machine



NUMA: Non-Uniform Memory Access

UMA and NUMA

The Power MAC G5 features 2 PowerPC 970/G5 processors that share a common central memory (up to 8Gbyte)



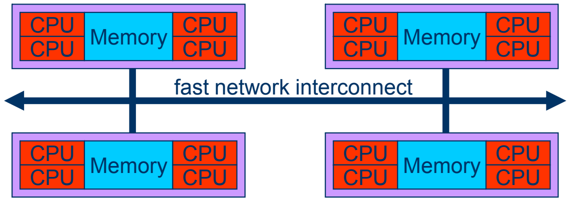
SGI Origin 3900:
- 16 R14000A processors per brick, each brick with 32GBytes of RAM.
- 12.8GB/s aggregated memory bw
(Scales up to 512 processors and 1TByte of memory)

Distributed Memory (DM)

- Processors have their own local memory.
- Memory addresses in one processor do not map to another processor (no global address space)
- Because each processor has its own local memory, cache coherency does not apply
- Requires a communication network to connect inter-processor memory
- When a processor needs access to data in another processor, it is usually the task of the programmer to explicitly define how and when data is communicated.
- Synchronization between tasks is the programmer's responsibility
- Very scalable
- Cost effective: use of off-the-shelf processors and networking
- Slower than UMA and NUMA machines

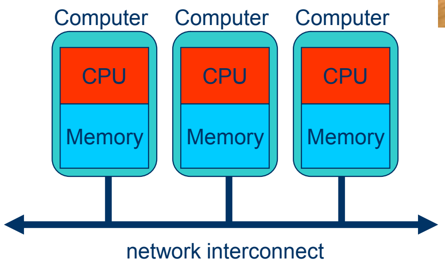
Hybrid Architectures

- Today, most systems are an hybrid featuring shared distributed memory.
  - Each node has several processors that share a central memory
  - A fast switch interconnects the several nodes
  - In some cases the interconnect allows for the mapping of memory among nodes; in most cases it gives a message passing interface



Distributed Memory

TITAN@DEI, a PC cluster interconnected by FastEthernet



ASCI White at the Lawrence Livermore National Laboratory

- Each node is an IBM POWER3 375 MHz NH-2 16-way SMP (i.e. 16 processors/node)
- Each node has 16GB of memory
- A total of 512 nodes, interconnected by a 2GB/sec network node-to-node
- The 512 nodes feature a total of 8192 processors, having a total of 8192 GB of memory
- It currently operates at 13.8 TFLOPS

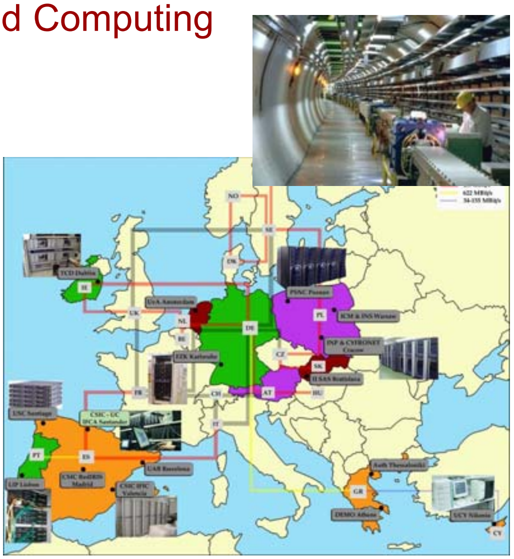
Summary

Architecture	CC-UMA	CC-NUMA	Distributed/ Hybrid
Examples	- SMPs - Sun Vexx - SGI Challenge - IBM Power3	- SGI Origin - HP Exemplar - IBM Power4	- Cray T3E - IBM SP2
Programming	- MPI - Threads - OpenMP - Shmem	- MPI - Threads - OpenMP - Shmem	- MPI
Scalability	<10 processors	<1000 processors	~1000 processors
Draw Backs	- Limited mem bw - Hard to scale	- New architecture - Point-to-point communication	- Costly system administration - Programming is hard to develop and maintain
Software Availability	- Great	- Great	- Limited

So, how do I program these things?

What about GRID computing?

- GRID Computing =
  Large Scale Distributed Computing
- Many Incarnations...
  - Computing Grid
  - Data Grid
  - Access Grid
  - *MyFavoriteThingGrid*

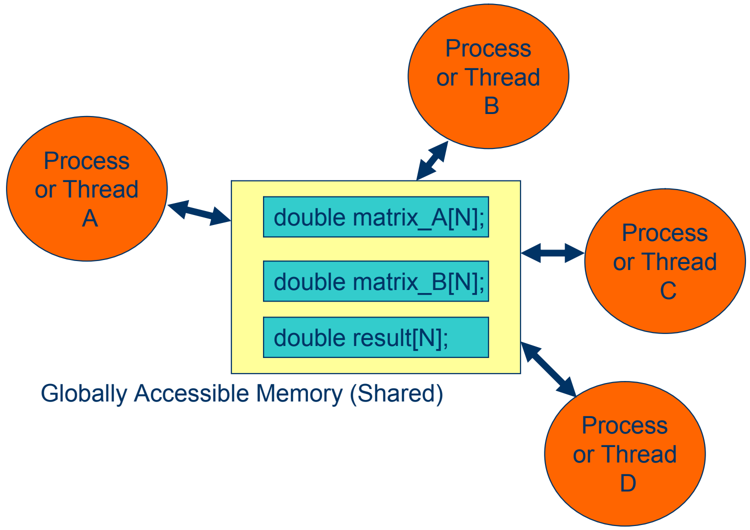


(CERN's Computational/Data Grid)

The main programming models…

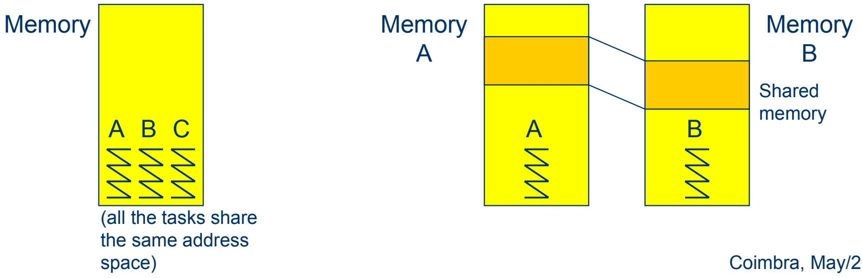
- A programming model abstracts the programmer from the hardware implementation
- The programmer sees the whole machine as a big virtual computer which runs several tasks at the same time
- The main models in current use are:
  - Shared Memory
  - Message Passing
  - Data parallel / Parallel Programming Languages
- Note that this classification is not all inclusive. There are hybrid approaches and some of the models overlap (e.g. data parallel with shared memory/message passing)

Shared Memory Model



Shared Memory Modes

- There are two major shared memory models:
  - All tasks have access to all the address space (typical in UMA machines running several threads)
  - Each task has its address space. Most of the address space is private. A certain zone is visible across all tasks. (typical in DSM machines running different processes)



Shared Memory Model

- Independently of the hardware, each program sees a global address space
- Several tasks execute at the same time and read and write from/to the same virtual memory
- Locks and semaphores may be used to control access to the shared memory
- An advantage of this model is that there is no notion of data “ownership”. Thus, there is no need to explicitly specify the communication of data between tasks.
- Program development can often be simplified
- An important disadvantage is that it becomes more difficult to understand and manage data locality. Performance can be seriously affected.

Shared Memory Model – Closely Coupled Implementations

- On shared memory platforms, the compiler translates user program variables into global memory addresses
- Typically a *thread model* is used for developing the applications
  - POSIX Threads
  - OpenMP
- There are also some parallel programming languages that offer a global memory model, although data and tasks are distributed
- For DSM machines, no standard exists, although there are some proprietary implementations

Shared Memory – Thread Model

- A single process can have multiple threads of execution
- Each thread can be scheduled on a different processor, taking advantage of the hardware
- All threads share the same address space
- From a programming perspective, thread implementations commonly comprise:
  - A library of subroutines that are called from within parallel code
  - A set of compiler directives imbedded in either serial or parallel source code
- Unrelated standardization efforts have resulted in two very different implementations of threads: ***POSIX Threads*** and ***OpenMP***

OpenMP

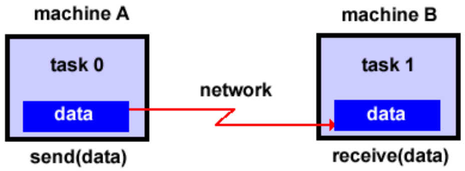
- Compiler directive based; can use serial code
- Jointly defined and endorsed by a group of major computer hardware and software vendors. The OpenMP Fortran API was released October 28, 1997. The C/C++ API was released in late 1998
- Portable / multi-platform, including Unix and Windows NT platforms
- Available in C/C++ and Fortran implementations
- Can be very easy and simple to use - provides for “incremental parallelism”
- No free compilers available

POSIX Threads

- Library based; requires parallel coding
- Specified by the IEEE POSIX 1003.1c standard (1995), also known as PThreads
- C Language
- Most hardware vendors / Operating Systems now offer PThreads
- Very explicit parallelism; requires significant programmer attention to detail

Message Passing Model

- The programmer must send and receive messages explicitly



Message Passing Model

- A set of tasks that use their own local memory during computation.
- Tasks exchange data through communications by sending and receiving messages
  - Multiple tasks can reside on the same physical machine as well as across an arbitrary number of machines.
- Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation.

MPI – The Message Passing Interface

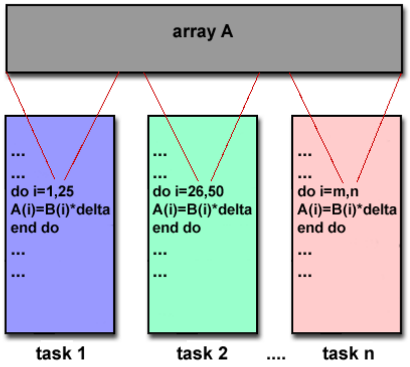
- Part 1 of the **Message Passing Interface (MPI)**, the core, was released in 1994.
  - Part 2 (MPI-2), the extensions, was released in 1996.
  - Freely available on the web: http://www.mpi-forum.org/docs/docs.html
- MPI is now the "de facto" industry standard for message passing
  - Nevertheless, most systems do not implement the full specification. Especially MPI-2
- For shared memory architectures, MPI implementations usually don't use a network for task communications
  - Typically a set of devices is provided. Some for network communication, some for shared memory. In most cases, they can coexist.

Message Passing Implementations

- Message Passing is generally implemented as libraries which the programmer calls
- A variety of message passing libraries have been available since the 1980s
  - These implementations differed substantially from each other making it difficult for programmers to develop portable applications
- In 1992, the MPI Forum was formed with the primary goal of establishing a standard interface for message passing implementations

Data Parallel Model

- Typically a set of tasks performs the same operations on different parts of a big array



```
...
...
do i=1,25
A(i)=B(i)*delta
end do
...
...
```

```
...
...
do i=26,50
A(i)=B(i)*delta
end do
...
...
```

```
...
...
do i=m,n
A(i)=B(i)*delta
end do
...
...
```

Data Parallel Model

- The data parallel model demonstrates the following characteristics:
  - Most of the parallel work focuses on performing operations on a data set
  - The data set is organized into a common structure, such as an array or cube
  - A set of tasks works collectively on the same data structure, however, each task works on a different partition of the same data structure
  - Tasks perform the same operation on their partition of work, for example, “add 4 to every array element”
- On shared memory architectures, all tasks may have access to the data structure through global memory.
- On distributed memory architectures the data structure is split up and resides as "chunks" in the local memory of each task

Summary

- Middleware for parallel programming:
  - Shared memory: all the tasks (threads or processes) see a global address space. They read and write directly from memory and synchronize explicitly.
  - Message passing: the tasks have private memory. For exchanging information, they send and receive data through a network. There is always a *send()* and *receive()* primitive.
  - Data parallel: the tasks work on different parts of a big array. Typically accomplished by using a parallel compiler which allows data distribution to be specified.

Data Parallel Programming

- Typically accomplished by writing a program with data parallel constructs
  - calls to a data parallel subroutine library
  - compiler directives
- In most cases, parallel compilers are used:
  - **High Performance Fortran (HPF):** Extensions to Fortran 90 to support data parallel programming
  - **Compiler Directives:** Allow the programmer to specify the distribution and alignment of data. Fortran implementations are available for most common parallel platforms
- DM implementations have the compiler convert the program into calls to a message passing library to distribute the data to all the processes.
  - All message passing is done invisibly to the programmer

Introduction to Parallel Programming and Grid Computing

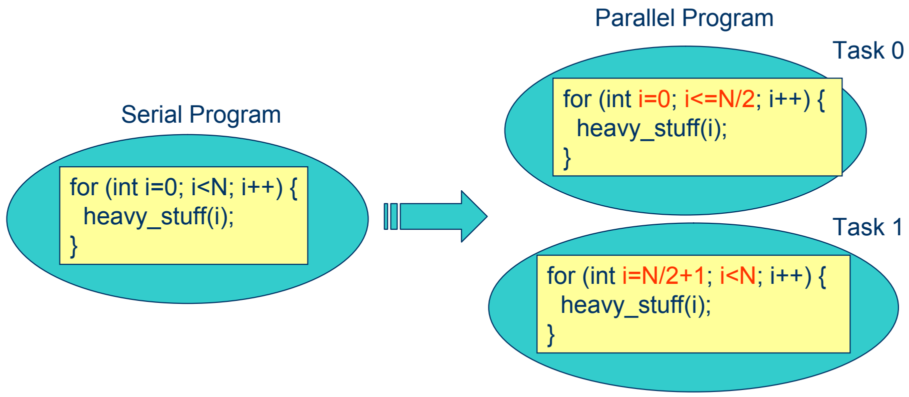
Dividing up the Work:
- Decomposing the Potential Parallel

May/2005

Paulo Marques
Dependable Systems Group
University of Coimbra, Portugal
pmarques@dei.uc.pt

Decomposition – Idea

- One of the first steps in designing a parallel program is to break the problem into discrete “chunks” of work that can be distributed to multiple tasks
  - Decomposition or Partitioning



Serial Program

```
for (int i=0; i<N; i++) {
  heavy_stuff(i);
}
```

Parallel Program

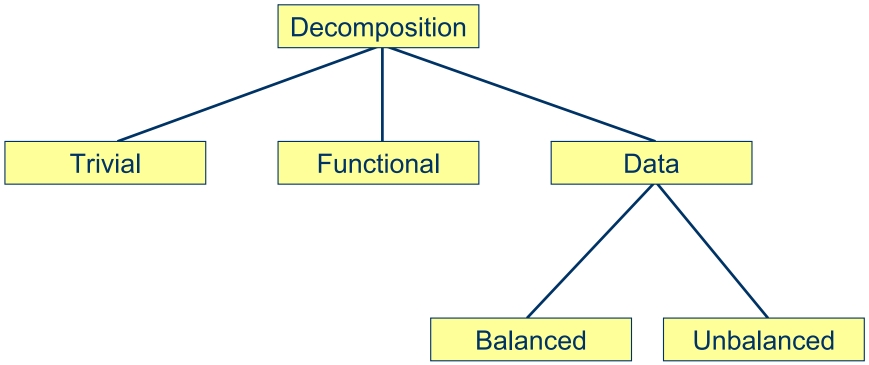
Task 0

```
for (int i=0; i<=N/2; i++) {
  heavy_stuff(i);
}
```

Task 1

```
for (int i=N/2+1; i<N; i++) {
  heavy_stuff(i);
}
```

Decomposition Models



Decomposition

Trivial

Functional

Data

Balanced

Unbalanced

Decomposition

- In practice, the source code and executable is the same for all processes
  - So, how do I get asymmetry?

- There’s always a primitive that returns the number of the process, and one that returns the size of the world (total processes):
  - whoami()
  - totalprocesses()

```
…

int main() {
 int lower, upper;

 if (whoami() == 0) {
   lower   = 0;
   higher  = N/2+1;
 } else {
   lower = N/2 + 1;
   higher = N;
 }

for (int i=lower; i<higher; i++) {
 heavy_stuff(i);
}
```

Trivial Decomposition

- Also called ***embarrassingly parallel*** because it’s so simple
- The whole program consists in independent actions that can be performed separately
  - Example:
    “Calculate all the square roots from 1 to 10.000”

- You can either modify the code, introducing asymmetry, making each task compute its part or you can use tools that parametrically run the code on different nodes.
  - The parameters are introduced in the command line (e.g. Condor)

- Linear speedups (and sometimes super-linear) are typical

In the previous example…

```
for (int i=0; i<N; i++) {
  heavy_stuff(i);
}
```

This is embarrassingly parallel! (if *heavy_stuff()* has no side effects)

You can either:

a) add code to make it asymmetric

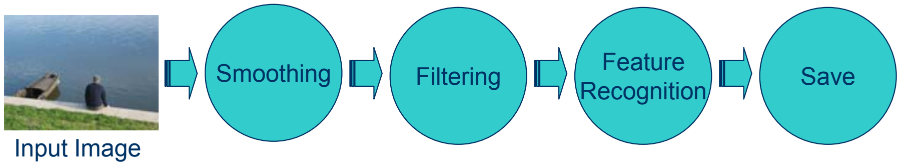
b) parameterize the program so that limits are read from the command line

... USE CONDOR!!!

http://www.cs.wisc.edu/condor/

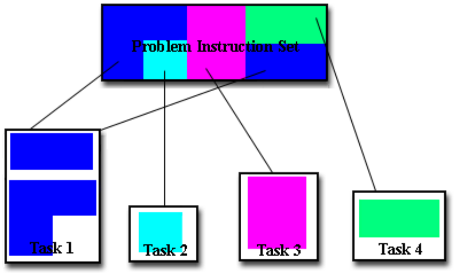
Functional Decomposition - Pipelining

- A simple form of functional decomposition is Pipelining
- Each input passes through each of the subprograms in a given order.
- Parallelism is introduced by having several inputs moving through the pipeline simultaneously
- For an n-depth pipeline, an n speedup is theoretically possible
- It depends on the slowest step of the pipeline



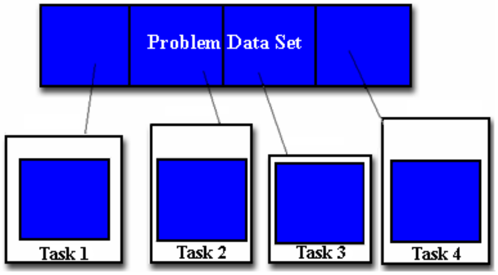
Functional Decomposition

- The focus is on the computation that is to be performed rather than on the data manipulated by the computation
- The problem is decomposed according to the work that must be done. Each task then performs a portion of the overall work.

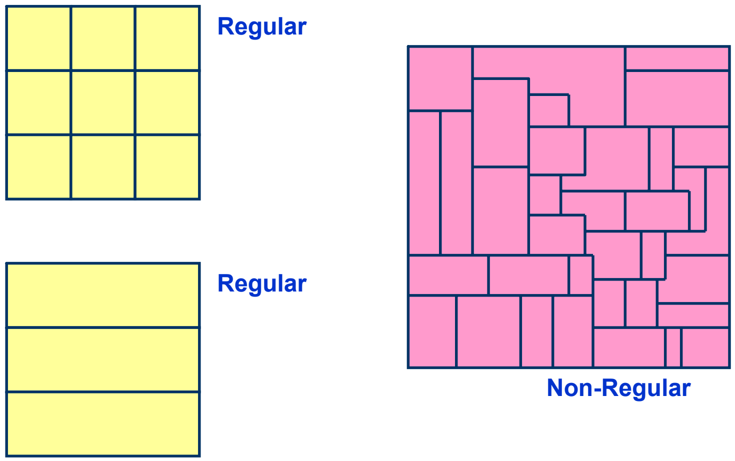


Data Decomposition / Domain Decomposition

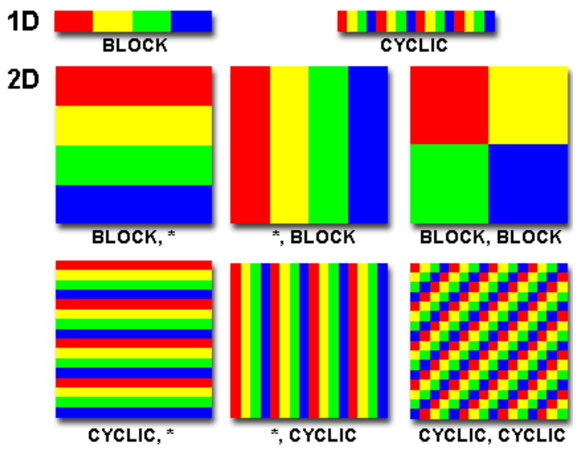
- The decomposition is focused on data
- The data is divided among processes.
- Each task then works on its portion of the data, exchanging messages if necessary



Regular and Non-regular Decomposition



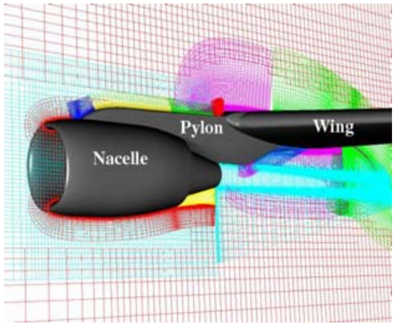
Even Regularly, there are many different ways of decomposing domains

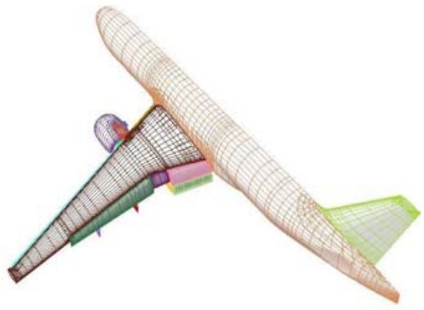


Each color represents a different processor/task

Why not always use regular?

- Different parts of the domain may take very different times to calculate/simulate





Matrix Multiplication

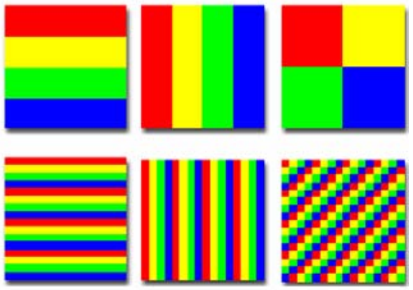
```
double A[N][N], B[N][N], C[N][N];

for (int i=0; i<N; i++) {
  for (int j=0; j<N; j++) {
    total = 0.0;
    for (int k=0; k<N; k++)
       total+= A[i][k]*B[k][j];

    C[i][j] = total;
  }
}
```

As you can see, in matrix multiplication, all the results in C[N][N] are independent from each other!

Quiz – Matrix Multiplication

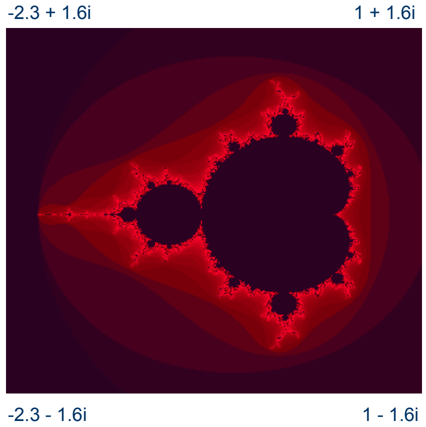
- How should the task of computing matrix C be divided among 4 processors?
  - Assume you are using a shared memory model, matrixes A and B are copied to local memory. C is globally shared. The used language: C/C++

Matrix C



Example of Task Farming

- Mandelbrot Set
  - There is a complex plane with a certain number of points.
  - For each point C, the following formula is calculated: $z \leftarrow z^2+C$, where z is initially 0.
  - If after a certain number of iterations, $|z|>=4$, the sequence is divergent, and it is colored in a way that represents how fast it is going towards infinity
  - If $|z|<4$, it will eventually converge to 0, so it is colored black

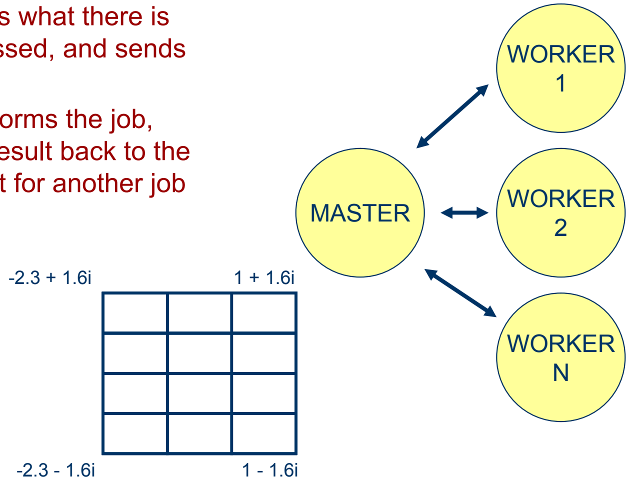


Data Decomposition

- When considering data decomposition, two different types of execution models are typical: task farming and step lock execution (in grid)
- In Task Farming, there is a master that sends out jobs which are processed by workers. The workers send back the results.
- In Step Lock Execution, there is a data grid in which a certain operation is performed on each point. The grid is divided among the tasks. In each iteration, a new value of each data point is calculated

Task Farming Mandelbrot (DSM – Message Passing Model)

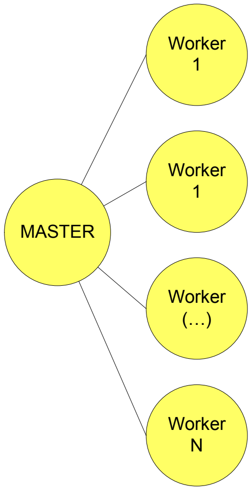
- Each worker asks the master for a job
- The master sees what there is still to be processed, and sends it to the worker
- The worker performs the job, and sends the result back to the master, asking it for another job



Basic Task Farm Algorithm

MASTER

```
while (1)
{
  msg = get_msg_worker();
  if (msg.type == RESULT)
    save_result(msg.result);

  if (there_is_work)
  {
    job = generate_new_job();
    send_msg_worker(job);
  }
  else
    send_msg_worker(QUIT)

  if (all_workers_done)
    terminate();
}
```



WORKER

```
send_master(REQUEST_WORK);

while (1)
{
  msg = get_msg_master();
  if (msg.type == JOB)
  {
    result = do_work(msg.job);
    send_master(result);
  }
  else if (msg.type == QUIT)
    terminate();
}
```

Load Balancing Task Farms

- Workers request tasks from the source when they require more work, i.e. task farms are intrinsically load balanced
- Also, load balancing is dynamic, *i.e.*, tasks are assigned to workers as they become free. They are not allocated in advance
- The problem is ensuring all workers finish at the same time
- Also, for all this to be true, the granularity of the tasks must be adequate
  - Large tasks → Poor load balancing
  - Small tasks → Too much communication

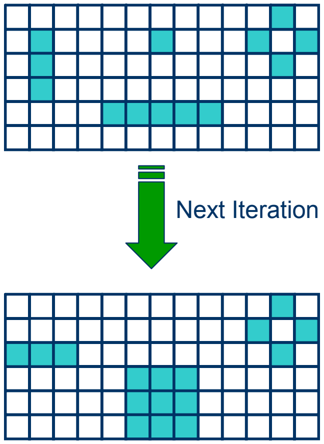
Improved Task Farming

- With the basic task farm, workers are idle while waiting for another task
- We can increase the throughput of the farm by buffering tasks on workers
- Initially the master sends workers two tasks: one is buffered and the other is worked on
- Upon completion, a worker sends the result to the master and immediately starts working on the buffered task
- A new task received from the master is put into the buffer by the worker
- Normally, this requires a multi-threaded implementation

- Sometimes it is advisable to have an extra process, called sink, where the results are sent to

Step Lock Execution

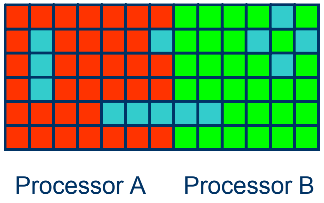
- The Game of Life
  - There is a grid of NxM slots. Each slot may either be empty or contain a cell
  - Each cell with one or no neighbors dies
  - Each cell with four or more neighbors dies
  - Each cell with two or three neighbors survives
  - If a slot has three neighbors with cells, a new cell is born



(Check out: http://www.bitstorm.org/gameoflife/)

The Game of Life

- Each processor is assigned a part of the board.
- It computes its section and when necessary exchanges boundary information with its neighbors
- At the end of each cycle, there is a *barrier* operation

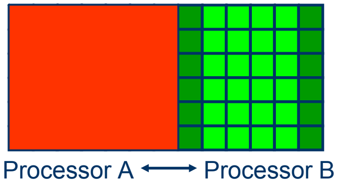


Step Lock Execution on Grids

- When decomposing a domain, the grids do not need to have the same size. But they must take approximately the same amount of time to compute
  - It's a question of load balancing
- At each iteration, a process is responsible for updating its local section of the global data block. Two types of data are involved:
  - Local data which the process is responsible for updating
  - Data which other processors are responsible for updating

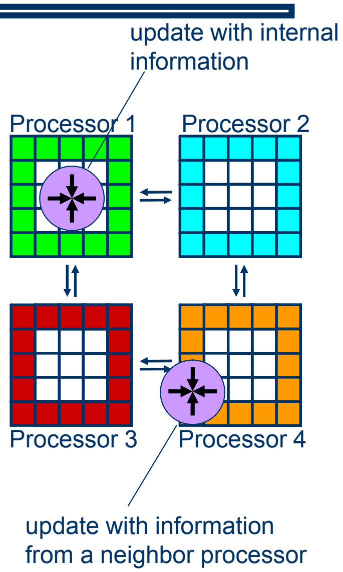
The Game of Life

- On a Shared Memory Architecture
  - Quite simple, just compute the assigned slots and perform a barrier at the end of each cycle
- On a DM Architecture / Message Passing
  - Send the boundary information to the adjacent processors
  - Collect boundary information from the adjacent processors
  - Compute the next iteration



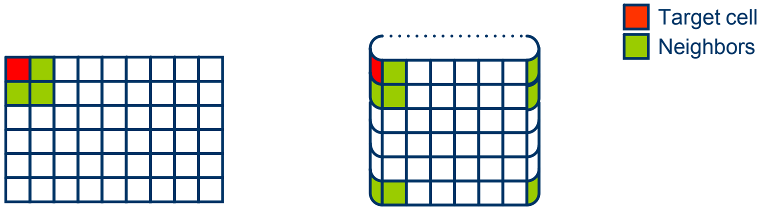
Boundary Swapping

- As seen in the game of life, each process owns a *halo* of data elements shadowed from neighboring processes
- At each iteration, processes send copies of their edge regions to adjacent processes to keep their halo regions up to date
  - This is known as *boundary swapping*


update with internal information
Processor 1
Processor 2
Processor 3
Processor 4
update with information from a neighbor processor

Boundary Conditions

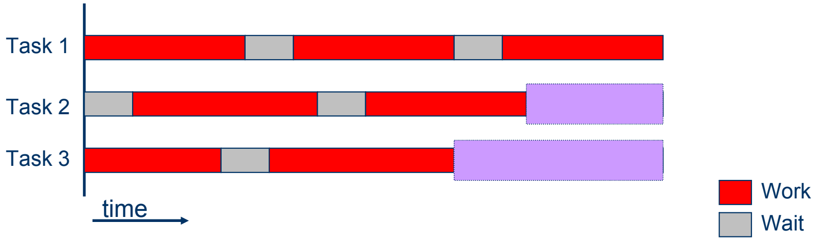
- Boundary swaps at edges of the global grid need to take into account the *boundary conditions* of the global grid
- Two common types of boundary conditions
  - Periodic boundaries
  - Finite (or static) boundaries



Finite Boundary   Periodic Boundary

Load Balancing

- Load balancing is always a factor to consider when developing a parallel application.
  - Too big granularity → Poor load balancing
  - Too small granularity → Too much communication
- The ratio computation/communication is of crucial importance!



Final Considerations…

Beware of Amdahl's Law!

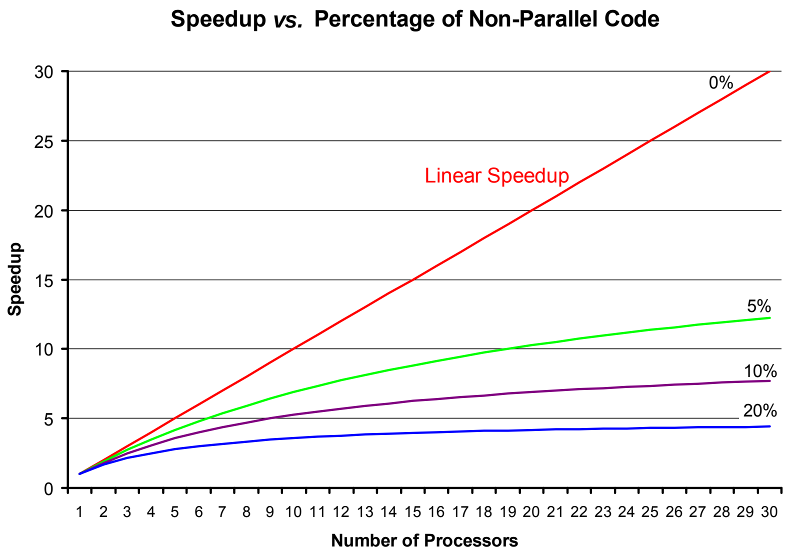
Amdahl's Law

- The speedup depends on the amount of code that cannot be parallelized:

$$speedup(n,s) = \frac{T}{T\cdot s + \frac{T\cdot(1-s)}{n}} = \boxed{\frac{1}{s + \frac{(1-s)}{n}}}$$

n: number of processors
s: percentage of code that cannot be made parallel
T: time it takes to run the code serially

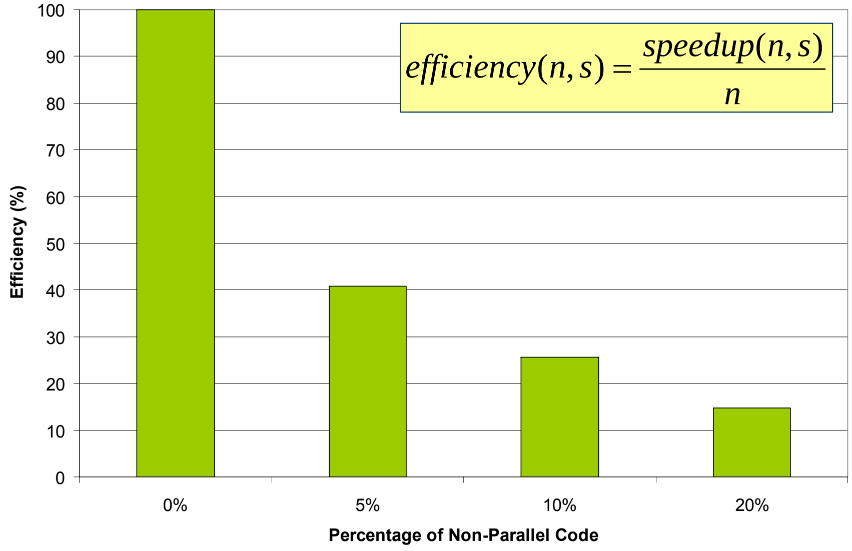
Amdahl's Law – The Bad News!



What Is That s Anyway?

- *Three slides ago…*
  - **"s: percentage of code that cannot be made parallel"**
- Actually, it's worse than that. Actually it's the percentage of time that cannot be executed in parallel. It can be:
  - Time spent communicating
  - Time spent waiting for/sending jobs
  - Time spent waiting for the completion of other processes
  - Time spent calling the middleware for parallel programming
- Remember…
  - if s is even as small as 0.05, the maximum speedup is only 20

Efficiency Using 30 Processors



$$efficiency(n,s) = \frac{speedup(n,s)}{n}$$

Maximum Speedup

$$speedup(n,s) = \frac{1}{s + \frac{(1-s)}{n}}$$

If you have ∞ processors this will be 0, so the maximum possible speedup is 1/s

non-parallel (s)	maximum speedup
0%	∞ (linear speedup)
5%	20
10%	10
20%	5
25%	4

On the Positive Side…

- You can run bigger problems
- You can run several simultaneous jobs (you have more parallelism available)
  - *Gustafson-Barsis with no equations*: *“9 women cannot have a baby in 1 month, but they can have 9 babies in 9 months”*

Introduction to Parallel Programming and Grid Computing

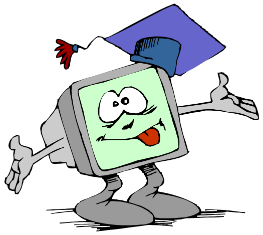
– Distributed Memory –
MPI: Message Passing Interface

May/2005

Paulo Marques
Dependable Systems Group
University of Coimbra, Portugal
pmarques@dei.uc.pt

That’s it!

“Good! Enough of this, show me the API’s!”



Recap

- **MPI** = **M**essage **P**assing **I**nterface
- MPI is a ***specification*** for the developers and users of message passing libraries. It is not a particular library.
- It is defined for C, C++ and Fortran
- Reasons for Using MPI
  - Standardization
  - Portability
  - Performance Opportunities
  - Large Functionality
  - Availability

Programming Model

- Message Passing, thought for distributed memory architectures
- MPI is also commonly used to implement (*behind the scenes*) some shared memory models, such as Data Parallel, on distributed memory architectures
- All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs.
- In MPI-1 the number of tasks is static. New tasks cannot be dynamically spawned during runtime. MPI-2 addresses this.

Hello World (hello_mpi.c)

```
(...)

#include <mpi.h>
#define BUF_SIZE            80

int main(int argc, char* argv[])
{
  int  id;                    // The rank of this process
  int  n_processes;           // The size of the world
  char buffer[BUF_SIZE];      // A message buffer

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  MPI_Comm_size(MPI_COMM_WORLD, &n_processes);

  (...)
```

Hello World (cont.)

```
(...)

 if (id == 0)  {
   printf("I'm process 0, The master of the world!\n");

   // Receive a message from all other processes
   for (int i=1; i<=n_processes-1; i++) {
     MPI_Recv(buffer, BUF_SIZE, MPI_CHAR, MPI_ANY_SOURCE,
                   MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
     printf("%s", buffer);
   }
 }
 else {
   // Send a message to process 0
   sprintf(buffer, "Hello, I'm process %d\n", id);
   MPI_Send(buffer, strlen(buffer)+1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
 }

 MPI_Finalize();
 return 0;
}
```

Compiling & Running the Example (MPICH2-UNIX)

- Compiling

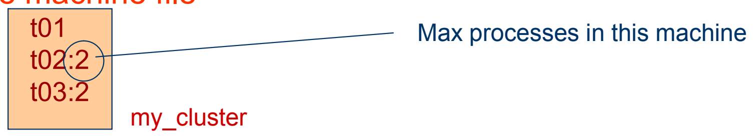
```
[pmarques@ingrid ~/best] mpicc hello_mpi.c -o hello_mpi
```

- Running

```
[pmarques@ingrid ~/best] mpiexec -np 10 hello_mpi

I'm process 0, The master of the world!
Hello, I'm process 1
Hello, I'm process 3
Hello, I'm process 2
Hello, I'm process 5
Hello, I'm process 6
Hello, I'm process 7
Hello, I'm process 9
Hello, I'm process 8
Hello, I'm process 4
```

mpiexec

- Note that mpiexec launches n copies of the program. Each copy executes independently of each other, having its private variable (id).
- By default, mpiexec chooses the machines from a global configuration file: machines.ARCH
  - (normally /usr/lib/mpich/share/machines.LINUX)
- You can specify your own machine file:
  - mpiexec -machinefile my_cluster -np 2 hello_mpi
- A simple machine file



```
t01
t02:2
t03:2
```

Max processes in this machine

my_cluster

Back to the Example

- MPI_Init(&argc, &argv);
  - Must be the first function to be called prior to any MPI functionality
  - argc and argv are passed so that the runtime can extract parameters from the command line
  - Arguments are not guaranteed to be passed to all the programs!
- MPI_Finalize();
  - Must be the last function to be called. No MPI calls can be made after this point.

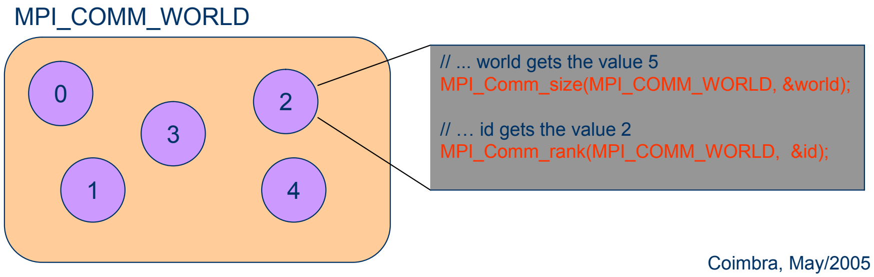
Compiling and Running in Windows

- Compiling
  - Using VisualStudio, include the proper MPI header files and library files

    “C/C++ → Additional Include Directories” =
    “C:\Program Files\MPICH2\include”
    “Linker → Additional Library Directories” =
    “C:\Program Files\MPICH2\lib”
    “Linker → Input” =
    “mpi.lib”
- Running
  - Make sure that the executable is in a shared directory or is available in the same place by all the nodes
  - Make sure you turn off your firewall ☹
  - mpiexec -hosts 2 orion vega \\Mandel\Mandel_MPI.exe
  - mpiexec -machinefile my_cluster -np 2 \\Mandel\Mandel_MPI.exe

COMM_WORLD

- A community of communicating processes is called a *communicator*.
- In MPI it is possible to specify different communicators that represent different sub-communities of processes.
- The default communicator, which allows all processes to exchange messages among themselves, is called MPI_COMM_WORLD.
- Inside a communicator, each process is attributed a number called *rank*.



```
// ... world gets the value 5
MPI_Comm_size(MPI_COMM_WORLD, &world);

// ... id gets the value 2
MPI_Comm_rank(MPI_COMM_WORLD,  &id);
```

Sending a Message

```
int MPI_Send(void* buffer, int count, MPI_Datatype type,
                        int dest, int tag, MPI_Comm comunicator);
```

- Sends a message to another process
  - *buffer* → the message buffer to send
  - *count* → the number of elements in the buffer
  - *type* → the type of the individual elements in the buffer
    - MPI_CHAR, MPI_BYTE, MPI_SHORT, MPI_INT, MPI_LONG, MPI_UNSIGNED_CHAR, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG,MPI_FLOAT, MPI_DOUBLE, or user-defined
  - *dest* → the rank of the destination process
  - *tag* → a number used to differentiate messages
  - *communicator* → the communicator being used
    - MPI_COMM_WORLD, or user-defined

Some Observations

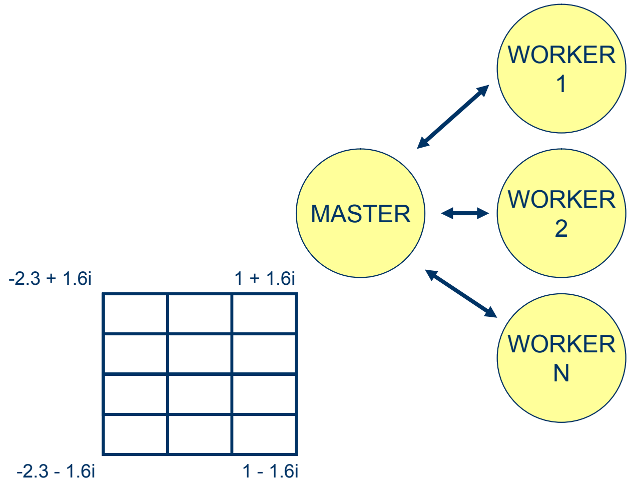
- These simple six routines can get you very far:
  - MPI_Init()
  - MPI_Comm_size()
  - MPI_Comm_rank()
  - MPI_Send()
  - MPI_Recv()
  - MPI_Finalize()
- Even so, the functionality available in MPI is much more powerful and complete. We will see that soon.
  - Nevertheless, we will only cover a small part of MPI-1.
- Typically, the routines return an error code. MPI_SUCCESS indicates everything went ok
- Don't forget to include <mpi.h> and use mpicc and mpirun.
- Don't assume you have the program arguments in any process except on the first.
- Don't assume you have I/O except on the first.

Receive a Message

```
int MPI_Recv(void* buffer, int count, MPI_Datatype type,
                        int source, int tag, MPI_Comm com, MPI_Status *status);
```

- Receives a message from another process
  - *buffer* → the message buffer to send
  - *count* → the number of elements in the buffer
  - *type* → the type of the individual elements in the buffer (must match what is being sent)
  - *source* → the rank of the process from which the message is being sent
    - can be MPI_ANY_SOURCE
  - *tag* → a number used to differentiate messages
    - can be MPI_ANY_TAG
  - *communicator* → the communicator being used
    - MPI_COMM_WORLD, or used-defined
  - *status* → extended information about the message or error
    - can be MPI_STATUS_IGNORE

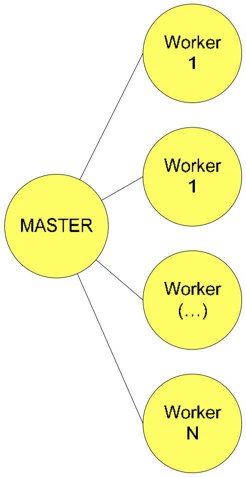
An Example of Task Farming



Remember the Task-Farm?

MASTER

```
while (1)
{
 msg = get_msg_worker();
 if (msg.type == RESULT)
  save_result(msg.result);

 if (there_is_work)
 {
  job = generate_new_job();
  send_msg_worker(job);
 }
 else
  send_msg_worker(QUIT)

 if (all_workers_done)
  terminate();
}
```



WORKER

```
send_master(REQUEST_WORK);

while (1)
{
 msg = get_msg_master();
 if (msg.type == JOB)
 {
  result = do_work(msg.job);
  send_master(result);
 }
 else if (msg.type == QUIT)
  terminate();
}
```

Mandel Task Farm, job/result structs

```
typedef struct
{
  bool work;          // 1-indicates a valid work,  0-to quit
  int i, j;           // the job to perform
} job;

typedef struct
{
  bool dummy;         // 1 indicates an initial request for work
  int rank;           // who is reporting the result
  int i, j;           // position of the result
  unsigned char img[GRID_WIDTH*GRID_HEIGTH];
} result;
```

Simple Task Farm of Mandel (mandel_mpi.c)

```
#define WIDTH           1600
#define HEIGHT          1200
#define MAXIT           100

#define XMIN            -2.3
#define YMIN            -1.6
#define XMAX            +1.0
#define YMAX            +1.6

#define X_TASKS         8                 // Use an 8x8 grid
#define Y_TASKS         8

#define GRID_WIDTH      WIDTH/X_TASKS     // Size of each grid
#define GRID_HEIGTH     HEIGHT/Y_TASKS    //   in pixels

int rank;                                 // Rank of each process
int n_proc;                               // Number of processes

unsigned char* img;                       // Where the image will
                                          //    be stored at the
                                          //    master
```

Mandelbrot – The main()

```
int main(int argc, char* argv[])
{
  MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD, &n_proc);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == 0)
    master();
  else
    worker();

  MPI_Finalize();

  return 0;
}
```

Mandelbrot – The Master

```
void master()
{
  img = (unsigned char*) malloc(sizeof(unsigned char)*WIDTH*HEIGHT);
  if (img == NULL)
  {
    MPI_Abort(MPI_COMM_WORLD, 0);
    return;
  }

  ////////////////////////////////

  // Number of workers still running
  int workers = n_proc - 1;

  // A job to send
  job a_job;
  a_job.i    = 0;
  a_job.j    = 0;
  a_job.work = true;

  // The result being received
  result res;

  // Continues on the next slide...
}
```

Mandelbrot – The Worker

```
void worker() {
  // The width and height of a grid, in real numbers
  double real_width  = (XMAX - XMIN)/X_TASKS;
  double real_height = (YMAX - YMIN)/Y_TASKS;

  // The job that is received
  job a_job;

  // The result that is sent
  result res;

  // Continues on the next slide...
}
```

Mandelbrot – The Master (cont.)

```
  while (true) {
    MPI_Recv(&res, sizeof(result), MPI_BYTE, MPI_ANY_SOURCE,
             MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    if (!res.dummy)
      save_result(&res);

    if (a_job.i < Y_TASKS) {
      MPI_Send(&a_job, sizeof(job), MPI_BYTE, res.rank, 0, MPI_COMM_WORLD);
      ++a_job.j;
      if (a_job.j == X_TASKS) {
        a_job.j = 0;
        ++a_job.i;
      }
    } else {
      a_job.work = false;
      MPI_Send(&a_job, sizeof(job), MPI_BYTE, res.rank, 0, MPI_COMM_WORLD);
      --workers;
    }

    if (workers == 0)
      break;
  }

  write_ppm(img, WIDTH, HEIGHT, MAXIT, "mandel.ppm");
```

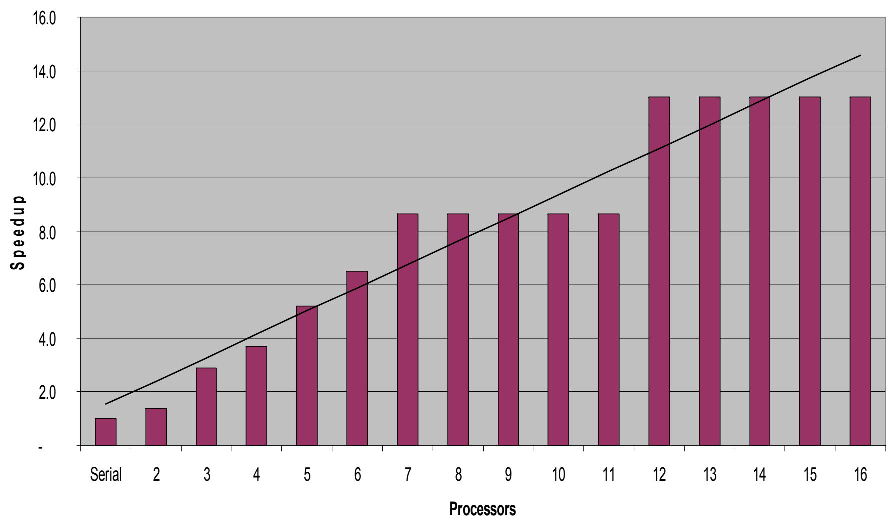
Mandelbrot – The Worker (cont.)

```
  res.dummy = true;
  res.rank  = rank;

  MPI_Send(&res, sizeof(result), MPI_BYTE, 0, 0, MPI_COMM_WORLD);

  while (true)
  {
    MPI_Recv(&a_job, sizeof(job), MPI_BYTE, 0, MPI_ANY_TAG,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    if (a_job.work == false)
      break;

    double xmin = XMIN + real_width*a_job.j;
    double ymin = YMIN + real_height*a_job.i;
    double xmax = xmin + real_width;
    double ymax = ymin + real_height;

    mandel(res.img, GRID_WIDTH, GRID_HEIGTH, MAXIT,
           xmin, ymin, xmax, ymax);

    res.i     = a_job.i;
    res.j     = a_job.j;
    res.dummy = false;
    MPI_Send(&res, sizeof(result), MPI_BYTE, 0, 0, MPI_COMM_WORLD);
  }
```

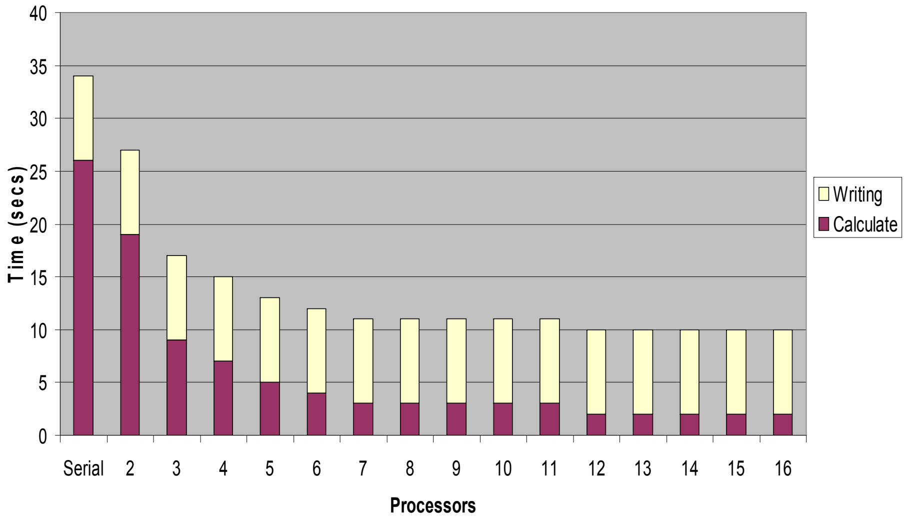
Mandelbrot – Homework

- Although task farming is very simple to implement, if you time the code on the cluster, its performance will be less than desirable…
  - Why?
- How could you improve the performance?
- Can you implement those changes and actually achieve a good speedup?
- For this particular case, can you derive a formula that relates computation, communication, size of matrixes and jobs, and the actual speedup that is possible to obtain?
- How would you change the program so that the values are not hard-coded in global constants?
  - Do it!

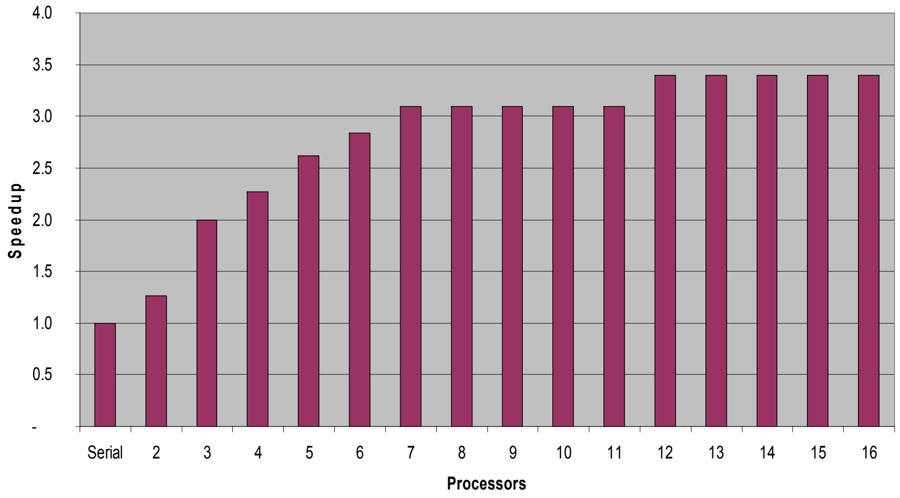
Speedup (Calculation Part Only)



Total Time to Calculate



Speedup (Global/Real)



Amdal’s Law in Action!!!!

MPI Routine Types

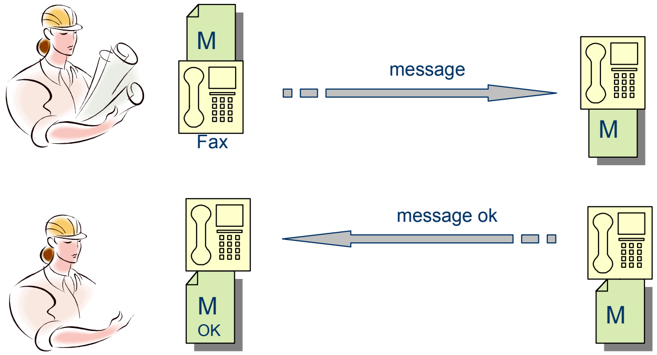
- In MPI there are two major types of communication routines:
  - Point-to-point: where a certain process (*originator*) sends a message to another specific process (*destination*).
    E.g. MPI_Send()/MPI_Recv()
  - Collective: where a certain group of processes performs a certain action collectively.
    E.g. MPI_ Bcast()

Types of Point-to-Point Operations

- There are different types of send and receive routines that are used for different purposes.
- They can be:
  - Synchronous
  - Buffered
  - Ready (*not covered in this course*)
- At the same time, the routines can be
  - Blocking / Non-Blocking
- You can combine a certain type of send with a different type of receive

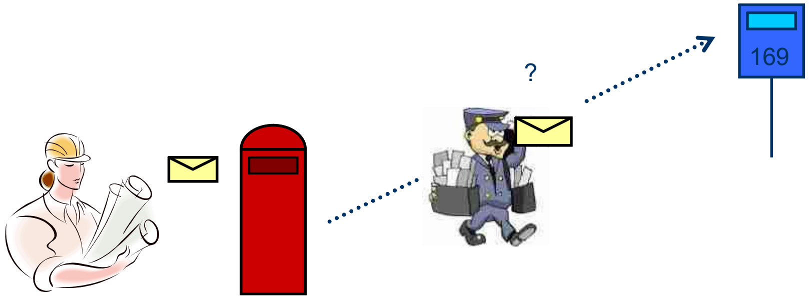
Synchronous Send

- Provide information about the completion (reception) of the message
  - If the routine returns OK, then the message has been delivered to the application at the destination
  - MPI_Ssend() is a *synchronous blocking send*, it only returns if the message has been delivered or an error has been detected



Buffered Send

- The "send" returns independently of whether the message has been delivered to the other side or not
  - The data in the send buffer is copied to another buffer freeing the application layer. The buffer can then be reused by it. MPI_Bsend() returns after the data has been copied.
  - There is no indication of when the message has arrived.



MPI_Send

- MPI_Send() is the standard routine for sending messages.
- It can either be synchronous or buffered. (It's implementation-dependent!)

- Blocking operations
  - Only return when the operation has completed.
  - If MPI_Send() is implemented has buffered-send, it blocks until it is possible to reuse the message buffer. That does not mean that the message has been delivered, only that it is safely on its way!

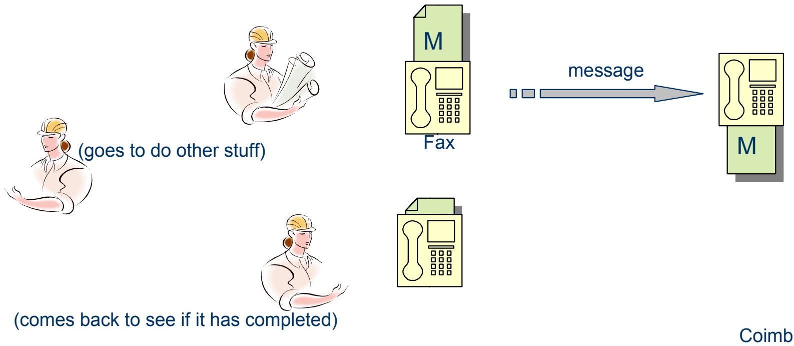
Non-Blocking Operations

- Note that the non-blocking operations return a handler that is used at a later time to see if it has completed…

```
MPI_Request request;
…
MPI_Isend (&buf, count, datatype, dest, tag, comm, &request);

…
while (!done) {
  do_stuff();
  …
  MPI_Test(&request, &done, MPI_STATUS_IGNORE);
}
```

Non-Blocking Operations

- Return straight away and allow the program to continue to perform other work
- At a later time, the program can use a test routine to see if the operation has already completed.
- The buffers can only be reused upon completion



Summary

	Routine	Description
Blocking	**MPI_Send**	**Blocking send, can be synchronous or buffered. Returns when the message is safely on its way.**
	MPI_Recv	**Blocking receive. Blocks until a message has been received and put on the provided buffer.**
	MPI_Ssend	Synchronous send. Returns when the message has been delivered at the destination.
	MPI_Bsend	Buffered send. Returns when the message has been copied to a buffer and the application buffer can be reused.
Non-Blocking (or Immediate)	**MPI_Isend**	**Basic immediate send. Completion (Test/Wait) will tell if the message is safely on its way.**
	MPI_Irecv	**Immediate receive. Starts the reception of a message. Completion (Test/Wait) will tell when the message is correctly on the application buffer.**
	MPI_Issend	Synchronous immediate send. Completion (Test/Wait) will tell if the message has been delivered at the destination.
	MPI_Ibsend	Buffered immediate send. Completion (Test/Wait) will tell if the message has been correctly buffered (thus, it's on its way) and the application buffer can be reused.
	MPI_Test	**Tests if a certain immediate operation has completed.**
	MPI_Wait	**Waits until a certain immediate operation completes.**

Collective Operations

- Different types of collective operations:
  - **Synchronization**: processes wait until all members of the group have reached the synchronization point
  - **Data Movement**: broadcast, scatter/gather, all-to-all
  - **Collective Computation** (reductions): one member of the group collects data from the other members and performs an operation (min, max, add, multiply, etc.) on that data
- We will only see some of these

Barrier

int MPI_Barrier (MPI_Comm communicator);

- Blocks the calling process until all processes have also called MPI_Barrier()
  - It’s a global synchronization point!

```
(...)

// Blocks until all processes have reached this point
MPI_Barrier(MPI_COMM_WORLD);

(...)
```

About Collective Operations

- Collective operations are blocking
- Collective communication routines do not take tag arguments
- Can only be used with MPI predefined data types
- Collective operations within subsets of processes are accomplished by first partitioning the subsets into new groups and then attaching the new groups to new communicators
  - Not discussed here

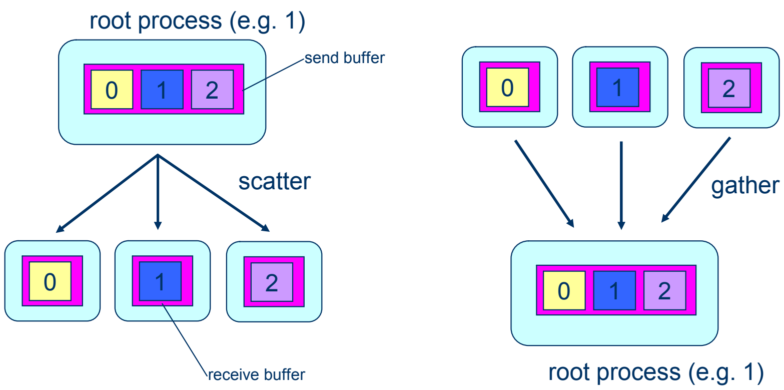
Broadcast

int MPI_Bcast(void* buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

Sends a message from process with rank *root* to all other processes in the same communicator. Note that all other processes must invoke MPI_Bcast() with the same root.

```
int main(int argc, char* argv[])
{
  int  id, max_tolerance;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);

  if (id == 0)
    scanf(“%d”, &max_tolerance);

  MPI_Bcast(&max_tolerance, 1, MPI_INT,
            0, MPI_COMM_WORLD);

  // After this point, all processes
  // have the same max_tolerance variable
}
```

Scatter/Gather



Example – Perform a scatter operation on the rows of an array

```
float sendbuf[SIZE][SIZE] = {
  {1.0, 2.0, 3.0, 4.0},
  {5.0, 6.0, 7.0, 8.0},
  {9.0, 10.0, 11.0, 12.0},
  {13.0, 14.0, 15.0, 16.0}
};

float recvbuf[SIZE];
…

assert(n_proc == SIZE);

// After this line, each process (assuming 4) will have a
// row of the matrix in recvbuf. This includes the root process (0 in this case)
MPI_Scatter(sendbuf, SIZE, MPI_FLOAT, recvbuf, SIZE,
                MPI_FLOAT, 0, MPI_COMM_WORLD);
…
```

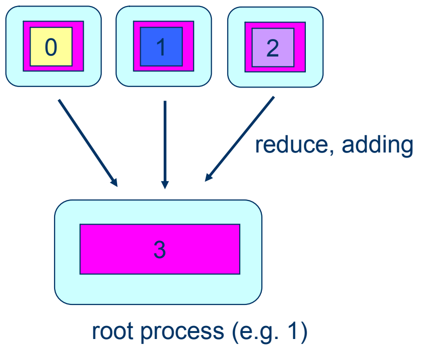
Scatter/Gather

- Used to send or receive data to/from all processes in a group
  - E.g. Initially distribute a set of different tasks among processes, or gather results.
  - Some of the parameters are only valid at the sender or at the receiver

```
int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype
               void *recvbuf, int recvcount, MPI_Datatype recvtype,
               int root, MPI_Comm communicator);
```

```
int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype
                void *recvbuf, int recvcount, MPI_Datatype recvtype,
                int root, MPI_Comm communicator);
```

Reduce



Reduce

```
int MPI_Reduce(void* sendbuf, void* recvbuf, int count,
               MPI_Datatype  datatype, MPI_Op op,
               int root, MPI_Comm comm)
```

Performs a certain operation (op) on all the data in the sendbuf of the processes, putting the result on the recvbuf of the root process. (The operation MPI_Allreduce() is similar but broadcasts the result to all processes!)

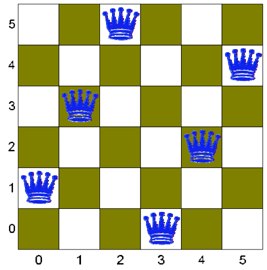
Operation	Meaning
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bitwise and
MPI_LOR	logical or
MPI_BOR	bitwise or
MPI_LXOR	logical xor
MPI_BXOR	bitwise xor
MPI_MAXLOC	maximum and location
MPI_MINLOC	minimum and location

Example – Find the Minimum Value in a Group of Processes, making it available to all processes

```
int local_min;          // Minimum value that each process has found so far
int global_min;         // Global value found in all processes


…

// After this line, each process will have global_min with the minimum value
// among min_path
MPI_Allreduce(&local_min, &global_min, 1, MPI_INT, MPI_MIN,
              MPI_COMM_WORLD);

…
```

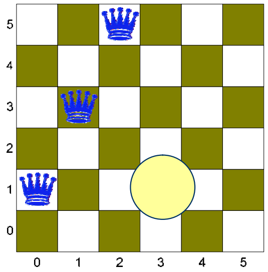
Homework

- Objective: To implement a parallel version of the N-Queens problem using task-farming



- A solution for a 6x6 board is represented by the vector v = *{1,3,5,0,2,4}*
- Each entry ***v[i]*** of the vector represents the column at which the queen is at line ***i***

N-Queens



- Imagine that you have a solution vector up until the position **i-1** (i=3):
  v = {1,3,5,?,?,?}
- Then, placing a queen at column ***i***, line ***v[i]***, is possible if and only if:
  - **for all positions j, 0<=j<i:**
    v[j] ≠ v[i] (not in the same line)
    v[j] ≠ v[i] - (i-j) (not in the same diagonal 1)
    v[j] ≠ v[i] + (i-j) (not in the same diagonal 2)
- This corresponds to a standard <u>backtracking algorithm</u>

NQueens – Serial Version

```
int n_queens(int size) {
  int board[MAX_SIZE];
  return place_queen(0, board, size);
}

int place_queen(int column, int board[], int size) {
  int solutions = 0;

  for (int i=0; i<size; i++) {
    board[column] = i;

    bool is_sol = true;
    for (int j=column-1; j>=0; j--) {
      if ((board[column] == board[j])               ||
          (board[column] == board[j] - (column-j))  ||
          (board[column] == board[j] + (column-j)))
      {
        is_sol = false;
        break;
      }
    }
    if (is_sol) {
      if (column == size-1)
        ++solutions;
      else
        solutions += place_queen(column+1, board, size);
    }
  }

  return solutions;
}
```

Attention!!!

- Your parallel version must return the same results than the serial one.

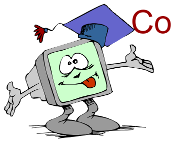
N	Solutions
1	1
2	0
3	0
4	2
5	10
6	4
7	40
8	92
9	352
10	724
11	2680
12	14200
13	73712
14	365596
15	2279184
16	14772512

NQueens – Serial version slightly modified

```
int n_queens(int size)
{
  // The board
  int board[MAX_SIZE];

  // Total solutions for this level
  int solutions = 0;

  // Try to place a queen in each line of the <level> column
  for (int a=0; a<size; a++)
  {
    for (int b=0; b<size; b++)
    {

      if ((a==b) || (a==b-1) || (a==b+1))
        continue;

      // Place queens
      board[0] = a;
      board[1] = b;

      // Check the rest
      solutions += place_queen(2, board, size);
    }
  }

  return solutions;
}
```

This Ends Our Crash-Course on MPI

- What have we seen?
  - Ranking operations (MPI_Comm_rank/MPI_Comm_size/…)
  - Point-to-point communication (MPI_Send/MPI_Recv/…)
  - Task-farming in MPI
  - Collective Operations for Synchronization (MPI_Barrier), Data Movement (MPI_Bcast/MPI_Gather/…), and Computation (MPI_Reduce/MPI_Allreduce)
- What haven't we covered
  - Derived data types
  - Groups and Virtual Topologies
  - MPI-2 (Dynamic process creation, One-Sided Communications, Parallel-IO, etc.)



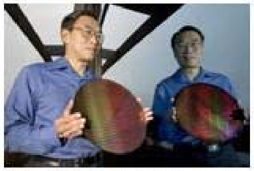
Introduction to Parallel Programming and Grid Computing

– Shared Memory –
OpenMP

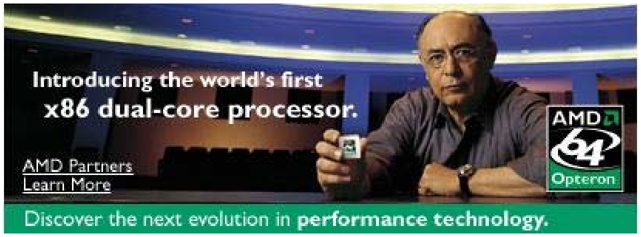
May/2005

Paulo Marques
Dependable Systems Group
University of Coimbra, Portugal
pmarques@dei.uc.pt

Why should I care about this?



Intel® Dual-Core Processors
Find out about Intel's multi-core strategy, and access resources for optimizing your transition to next-generation platforms.


Introducing the world's first
x86 dual-core processor.
AMD Partners
Learn More
AMD 64 Opteron
Discover the next evolution in performance technology.

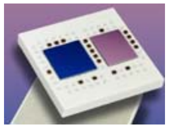
Intel® processor with Hyper-Threading (HT) Technology¹.

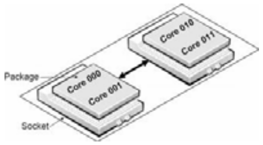
OpenMP – Recap

- Shared Memory Model with Threads
  - Typically used on UMA or NUMA machines
- Compiler directive based
  - Can use serial code
  - Even so, it is necessary to use some library routines
- Jointly defined and endorsed by a group of major computer hardware and software vendors.
- Portable / multi-platform, including Unix and Windows NT platforms
- Available in C/C++ and Fortran implementations
- Can be very easy and simple to use - provides for "incremental parallelism"

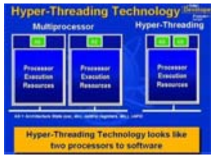
Why should I care about this?

- Multicore Processors are comming





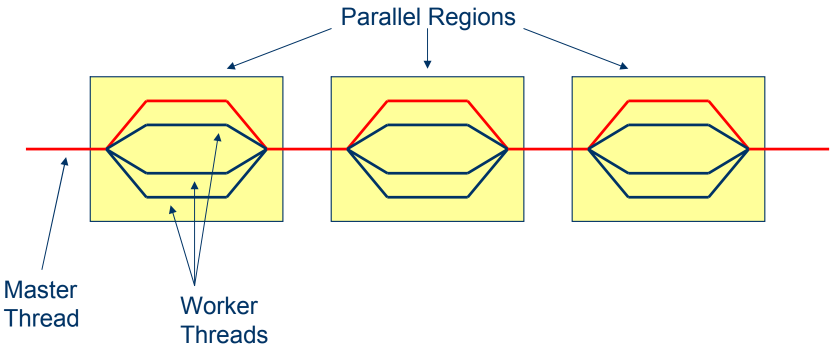
- *Simultaneous Multithreading Processors*
  (e.g. Intel Xeon 3GHz)


Hyper-Threading Technology
Multiprocessor
Hyper-Threading
Processor Execution Resources
Processor Execution Resources
Processor Execution Resources
Hyper-Threading Technology looks like two processors to software

- SMP Machines
  (e.g. Dual AMD Opteron)

OpenMP Programming Model

- Fork-Join Parallelism
  - Master thread creates a number of working threads as needed
  - Typically used on heavy-duty for cycles: parallel loops
  - A sequential program naturally evolves into a parallel one



OpenMP – Some Considerations

- OpenMP is a shared-memory model
- By default, all threads share the same variables
  - Beware: This can lead to race conditions
- There are primitives for synchronizing threads
  - Beware: Synchronization is quite expensive
  - An alternative is to duplicate data, avoiding variable sharing between threads. That can, nevertheless, also be expensive. There are also primitives for this.

How is OpenMP Typically Used?

- OpenMP is mostly used to parallelize time consuming loops
  - The iterations of each “for” loop are divided among different threads

```
void main()
{
  double result[SIZE];


  for (int i=0; i<SIZE; i++)
  {
    huge_computation(i, result[i]);
  }
}
```

```
void main()
{
  double result[SIZE];

  #pragma omp parallel for
  for (int i=0; i<SIZE; i++)
  {
    huge_computation(i, result[i]);
  }
}
```

Syntax

- OpenMP constructs can either be:
  - Compiler directives
  - Library function calls
- Compiler directives apply to structured blocks
  - A *structured block* consists of a block of code with only one entry point at the top and an exit point at the bottom.
  - The only branch statement allowed in a structured block is an exit() call.
- In C/C++, the compiler directives take the form:

  #pragma omp construct [clause [clause] …]

Simple “Hello World” program (hello_omp.c)

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

#define NUM_THREADS     4

int main()
{
  omp_set_num_threads(NUM_THREADS);

  #pragma omp parallel
  {
    int id = omp_get_thread_num();
    printf("Hello, I'm thread %d\n", id);
  }

  return 0;
}
```

OpenMP function call

OpenMP compiler directive

Actually...

- In the last example, you will probably see something like...

```
[pmarques@ingrid ~/best] ./hello_omp
Hello, I'm thrHello, I'm thread 1Hello, I'm thread 3
Segmentation Fault – Core dump
```

- You must be careful about what you call in parallel. The correct version of the previous program would be (hello_omp2.c):

```
...
#pragma omp parallel
{
  int id = omp_get_thread_num();

  #pragma omp critical
  printf("Hello, I'm thread %d\n", id);
}
...
```

Compiling & Running the Example

- Compiling

```
[pmarques@ingrid ~/best] icc -openmp hello_omp.c -o hello_omp
```

- Running

```
[pmarques@ingrid ~/best] ./hello_omp

Hello, I'm thread 0
Hello, I'm thread 1
Hello, I'm thread 3
Hello, I'm thread 2
```

Note: In Windows, you would will have to compile in the following way: icl /Qopenmp hello_omp.c

Constructs

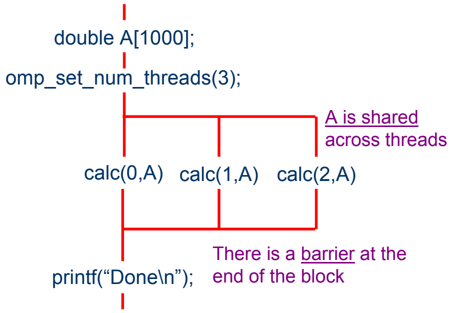
- OpenMP’s constructs fall into 5 categories
  - Parallel regions
  - Work-sharing
  - Data environment
  - Synchronization
  - Runtime functions/Environment variables

- We are going to see a good deal of them!

Parallel Regions

- Threads are created using the #pragma omp parallel directive
  - omp_set_num_threads() sets the number of threads for the next region
  - omp_thread_num() gets the number of the current thread
  - Variables are shared across threads

```
double A[1000];

omp_set_num_threads(3);

#pragma omp parallel
{
   int id = omp_get_thread_num();
   calc(id, A);
}

printf("Done\n");
```



A is shared across threads

There is a barrier at the end of the block

Work-Sharing Constructs – for

- The work-sharing construct "for", divides the iterations of a loop among several threads
- Shorthand for parallel fors:

  #pragma omp parallel for

```
#pragma omp parallel
{
  …

  #pragma omp parallel for
  for (int i=0; i<N; i++) {
    heavy_stuff(i);
  }
}
```

By default there is a barrier at the end of the "omp for". It can be turned off with the "nowait" clause. (Use with care!)

An Important Detail…

- Normally, you specify a number of threads that correspond to the number of processors you have
- OpenMP lets you use two running modes:
  - Dynamic (the default):
    - The number of threads in a parallel region can vary from one parallel region to another
    - Setting the maximum number of threads only sets the maximum: you can get less!
  - Static:
    - The number of threads is always the same and specified by the programmer

And Now, Something Interesting!

```
// Matrix Multiplication (1024x1024)

#include <stdio.h>
#include <time.h>

#define N         1024

double A[N][N], B[N][N], C[N][N];

int main()
{
  time_t t1, t2;
  time(&t1);

  for (int i=0; i<N; i++) {
    for (int j=0; j<N; j++) {
      double total = 0.0;
      for (int k=0; k<N; k++)
        total+= A[i][k]*B[k][j];

      C[i][j] = total;
    }
  }

  time(&t2);
  printf("Multiplying %dx%d matrixes took %d sec\n", N, N, t2-t1);
}
```

Matrix Multiplication, OpenMP version (mult_omp.c)

```
#include <omp.h>

...

int main()
{
  time_t t1, t2;
  time(&t1);

  omp_set_num_threads(omp_get_num_proc());

  #pragma omp parallel for
  for (int i=0; i<N; i++) {
    for (int j=0; j<N; j++) {
      double total = 0.0;
      for (int k=0; k<N; k++)
        total+= A[i][k]*B[k][j];

      C[i][j] = total;
    }
  }

  time(&t2);
  printf(“Multiplying %dx%d matrixes took %d sec\n”, N, N, t2-t1);
}
```

omp parallel for

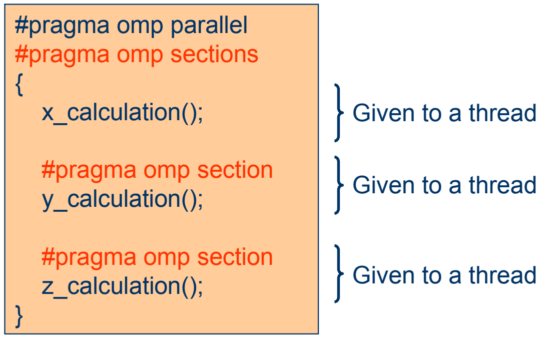
- You can add a schedule clause:
  #pragma omp parallel for schedule(…)
- schedule(static [,chunk])
  - Each thread is given out a certain number of blocks of size *chunk*
- schedule(dynamic [,chunk])
  - Each thread grabs *chunk* iterations from a queue as soon as it finishes its current work
- schedule(guided [,chunk])
  - Each thread dynamically gets blocks of iterations. The size starts out by being big and shrinks down to *chunk* size
- schedule(runtime)
  - Any of the above, as specified in the OMP_SCHEDULE environment variable

Results

- *Ingrid*, a dual-Xeon P4 2GHz machine with support for Hyper-Threading
  - In practice, it thinks it has 4 processors!
- Serial version: 63 seconds!
- Parallel version: 8 seconds!
- This is a 7.8x speedup on a dual processor machine!!!
  - Not bad for 3 lines of code!
  - Can you explain this astonishing result?
    (If you think this question is silly, either you know too much or too little.)

Sections Work-Sharing Construct

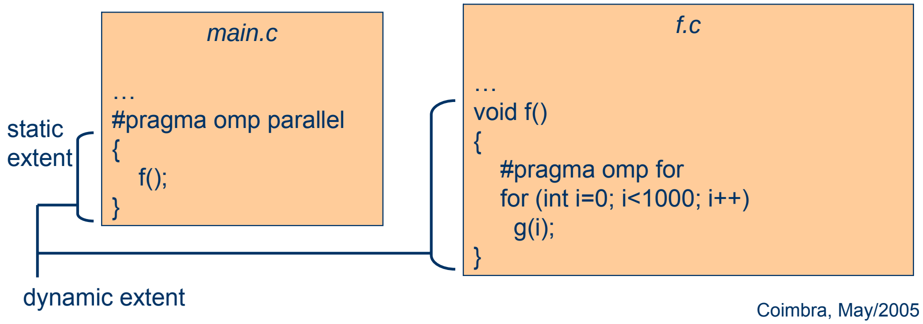
- sections give a different structured block to each thread
- By default there is a barrier at the end of the construct. As in the “for” construct, you can turn it off by specifying the “nowait” flag



```
#pragma omp parallel
#pragma omp sections
{
   x_calculation();

   #pragma omp section
   y_calculation();

   #pragma omp section
   z_calculation();
}
```

Scope of OpenMP Constructs

- In OpenMP there is the notion of *Lexical (or static) extent* and *Dynamic extent*
  - Lexical Extent: corresponds to the static region being defined by the omp pragma
  - Dynamic Extent: corresponds to what is seen at runtime. I.e. the dynamic extent can be larger than the lexical



main.c

```
…
#pragma omp parallel
{
    f();
}
```

f.c

```
…
void f()
{
    #pragma omp for
    for (int i=0; i<1000; i++)
      g(i);
}
```

Changing Storage Attributes

- It is possible to change the way threads see each variable within a parallel block
- For this, the following attributes are used:
  - shared (*the default*)
  - private
  - firstprivate
  - lastprivate
  - threadprivate
- Note that shared, private, firstprivate and threadprivate can apply to any parallel block, while lastprivate only applies to parallel fors

Data Environment Constructs

- Default Storage
  - Shared Memory Model: most variables are shared by default
- Global variables are SHARED among threads
- Not everything is shared:
  - Stack variables in routines called in parallel regions (these are thread private)
  - Automatic variables within a statement block (which are also private)

private clause

- *private(var)* creates a local copy of *var* for each thread
  - The value of var is not initialized
  - the private copy of var is in no way connected with the original variable. (It's like it's a new variable with the same name!)

```
int total;
total = 0;

#pragma omp parallel for private(total)
for (int i=0; i<10; i++)
  total += i;

printf("%d\n", total);
```

Each thread gets a copy of total. But the initial value of total is unknown

The value of total is unknown at this point

A note on usage…

- Note that private (and firstprivate, which we will see next), are mostly used for internal variables needed for a computation
- Example: calculate all the square roots from 1 to 10000 (N) in parallel

```
double result[N];
double aux;

#pragma omp parallel for private(aux)
for (int i=0; i<N; i++)
{
  aux = sqrt(i);
  result[i] = aux;
}
```

Right!

```
double result[N];
double aux;

#pragma omp parallel for
for (int i=0; i<N; i++)
{
  aux = sqrt(i);
  result[i] = aux;
}
```

Wrong!

firstprivate clause

- It's the same as private, but the initial value of the variable is passed to each thread

```
int total;
total = 0;

#pragma omp parallel for firstprivate(total)
for (int i=0; i<10; i++)
  total += i;

printf("%d\n", total);
```

Each thread gets a copy of total, with the initial value of 0

The value of total is not valid at this point (same as private).

lastprivate clause

- Same as private, but passes the value of the last iteration of a parallel for to the variable of the same name it represents
- Note that it is the value <u>of the last iteration</u>
- In a parallel for, it's possible to specify firstprivate and lastprivate

```
int total;
total = 0;

#pragma omp parallel for firstprivate(total) lastprivate(total)
for (int i=0; i<10; i++)
  total += i;

printf("%d\n", total);
```

Tricky: what's the value printed out?

Certainly not what we want…

A Little Reminder

```
int a, b, c;

a = b = c = 1;

#pragma omp parallel firstprivate(a) private(b)
{
  …
}
```

- <u>c</u> is shared by all threads
- In the parallel block, each thread has a copy of <u>a</u> and <u>b</u>
  - <u>a</u> is 1 when each thread starts
  - the value of <u>b</u> is not initialized when each thread starts
- After the block, the values of <u>a</u> and <u>b</u> are undefined

threadprivate clause

- Makes global data (global variables) an integral part of each thread
- This is different from making them private
  - With private, global variables are masked
  - With threadprivate, each thread gets a persistent copy of a global variable that survives across parallel blocks
- Threadprivate variables can be initialized with the “copyin” clause

reduction clause

- A very important clause
- Allows values of a parallel block to be combined after the block
  - reduction(operator:list)
  - The variables in *list* must appear in the enclosing parallel region
- Inside a parallel or work-sharing construct:
  - A local copy of each variable in *list* is made, and initialized according to the operator being specified (e.g. 0 for +, 1 for *, etc.)
  - Local copies of the variable are reduced to a single global copy at the end of the construct

Example of reduction

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

int main()
{
  int total = 0;
  int i;

  #pragma omp parallel for reduction(+:total)
  for (i=0; i<10; i++)
    total+=i;

  printf("total=%d\n", total);

  return 0;
}
```

Always prints out 45!

Synchronization

- OpenMP supports the following constructs for synchronization
  - atomic
  - critical section
  - barrier
  - flush *(not covered here)*
  - ordered *(not covered here)*
  - (single)
  - (master)

} not exclusively synchronization

Atomic

- It’s a type of critical section – only one thread can execute at a time
- It can only be used for guaranteeing correct memory updates
- The statement (memory update) must have the form: var*op*= expr, where op is + * - / & ^ | << or >>, or ++var, --var, var++, var--.

```
int x, b;

x = 0;

#pragma omp parallel private(b)
{
  b = calculate(i);

  #pragma omp atomic
  x+= b;
}
```

Barrier

- Each thread waits until all threads have arrived at the barrier

```
int A[1000];

#pragma omp parallel
{
  big_calculation_1(A);

  #pragma omp barrier

  big_calculation_2(A);
}
```

Critical Section

- Only one thread at a time can enter a critical section.
- Critical sections can be named:
  - #pragma omp critical (*name*)

```
int x, b;

#pragma omp parallel private(b)
{
  b = calculate(i);

  #pragma omp critical
  {
    x+= b;
    write_to_file(b);
  }
}
```

Single

- The single construct is used to implement a block that is executed only by one thread.
- All other threads wait (barrier) until that one finishes that section.

```
#pragma omp parallel
{
  perform_calculationA(omp_get_thread_num());

  #pragma omp single
  {
    printf(“Now at the middle of the calculation…\n”);
  }

 perform_calculationB(omp_get_thread_num());
}
```

Master

- The master construct denotes a structured block that is only executed by the master thread. The other threads just ignore it. (no implied barriers)

```
#pragma omp parallel
{
  #pragma omp master
  {
    printf(“Now starting main calculation…\n”);
  }

  perform_calculation(omp_get_thread_num());
}
```

Fixing the Number of Threads in a Program

a) Turn off dynamic mode

b) Set the number of threads

```
int main()
{
  omp_set_dynamic(false);
  omp_set_num_threads(omp_get_num_procs());

  …
}
```

OpenMP Library Routines

- Locks
  - omp_init_lock(), omp_set_lock(), omp_unset_lock(), omp_tst_lock()
- Runtime
  - Check/Modify number of threads
    - omp_set_num_threads(), omp_get_num_threads(), omp_get_thread_num(), omp_get_max_threads()
  - Turn on/off dynamic mode
    - omp_get_dynamic(), omp_set_dynamic()
  - Number of processors in the system
    - omp_get_num_procs()

Environment Variables

- Some variables control the way OpenMP works by default
  - OMP_SHEDULE → parameters for schedule(dynamic)
  - OMP_NUM_THREADS → default number of threads
  - OMP_DYNAMIC → (TRUE/FALSE) use of dynamic mode
  - OMP_NESTED → (TRUE/FALSE) whether parallel regions can be nested

And Now, a Little Example…

- Our familiar N-Queens Problem
  - Source code slightly changed so that the first two levels are directly checked using a nested “for” loop (this saves a lot of trouble)

Parallel Version of N-Queens (nqueens_omp.c)

```
int n_queens(int size)
{
  // The board
  int board[MAX_SIZE];

  // Total solutions for this level
  int solutions = 0;

  #pragma omp parallel for reduction(+:solutions) private(board)
  // Try to place a queen in each line of the <level> column
  for (int a=0; a<size; a++)
  {
    for (int b=0; b<size; b++)
    {
      // The first two queens cannot be in the same line or diagonal
      if ((a==b) || (a==b-1) || (a==b+1))
        continue;

      // Place queens
      board[0] = a;
      board[1] = b;

      // Check the rest
      solutions += place_queen(2, board, size);
    }
  }

  return solutions;
}
```

Serial Version of N-Queens (nqueens.c)

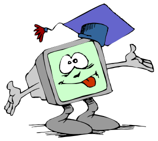
```
int n_queens(int size)
{
  // The board
  int board[MAX_SIZE];

  // Total solutions for this level
  int solutions = 0;

  // Try to place a queen in each line of the <level> column
  for (int a=0; a<size; a++)
  {
    for (int b=0; b<size; b++)
    {

      if ((a==b) || (a==b-1) || (a==b+1))
        continue;

      // Place queens
      board[0] = a;
      board[1] = b;

      // Check the rest
      solutions += place_queen(2, board, size);
    }
  }

  return solutions;
}
```

Results on Ingrid

```
[pmarques@ingrid ~/best] ./nqueens
Running NQueens size=14
Queens 14: total solutions: 365596
It took 12 sec
[pmarques@ingrid ~/best] ./nqueens_omp
Running NQueens size=14
Using 4 threads
Queens 14: total solutions: 365596
It took 5 sec
```

- A 2.4 speedup for a dual-processor with hyper-threading
- Although very good, why isn’t this result as good as matrix-mult with OMP?

And This Concludes Our Crash Course on OpenMP!



IMPORTANT NOTICE