Arquitectura de Computadores II

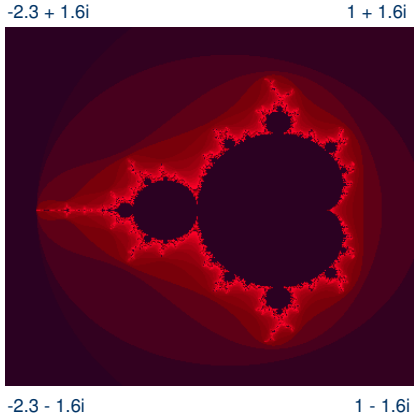
Message Passing Interface (MPI)

2004/2005

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

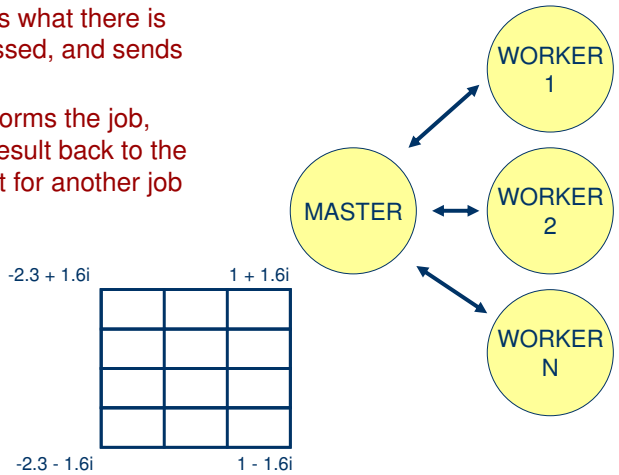
Example of Task Farming

- 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



Task Farming Mandelbrot (DSM – Message Passing Model)

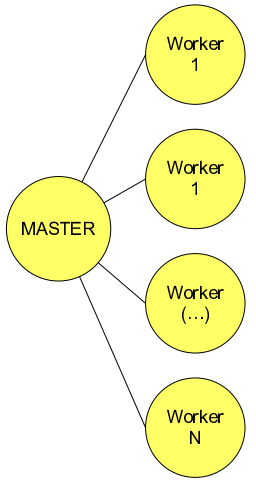
- Each worker asks the master for a job
- The master sees what there is still to be processed, and sends it to the worker
- The worker performs the job, and sends the result back to the master, asking it for another job



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

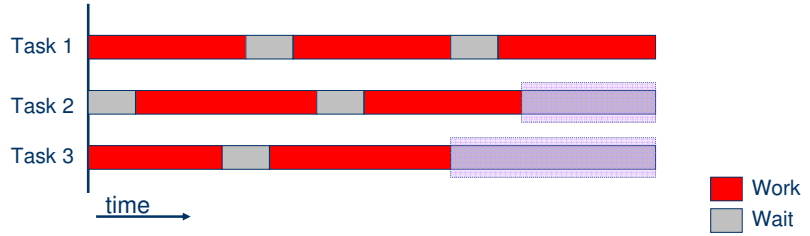
```
send_master(REQUEST_WORK);

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

Final Considerations…

Beware of Amdahl's Law!

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!



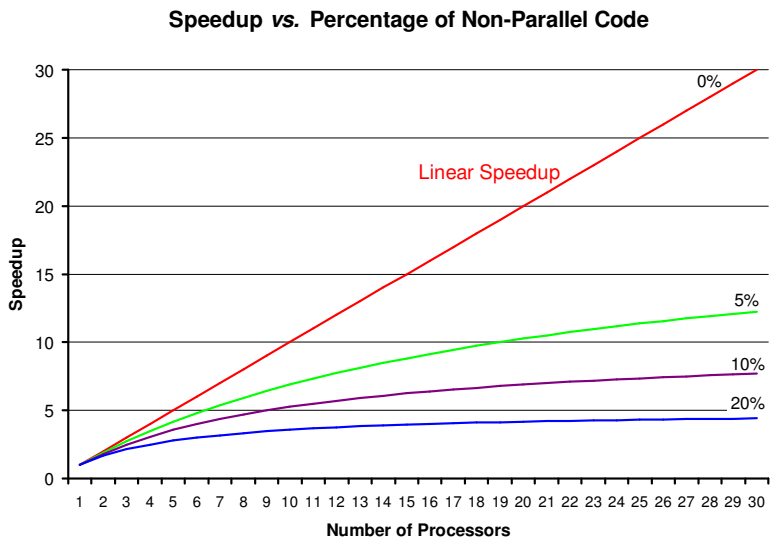
Amdahl's Law

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

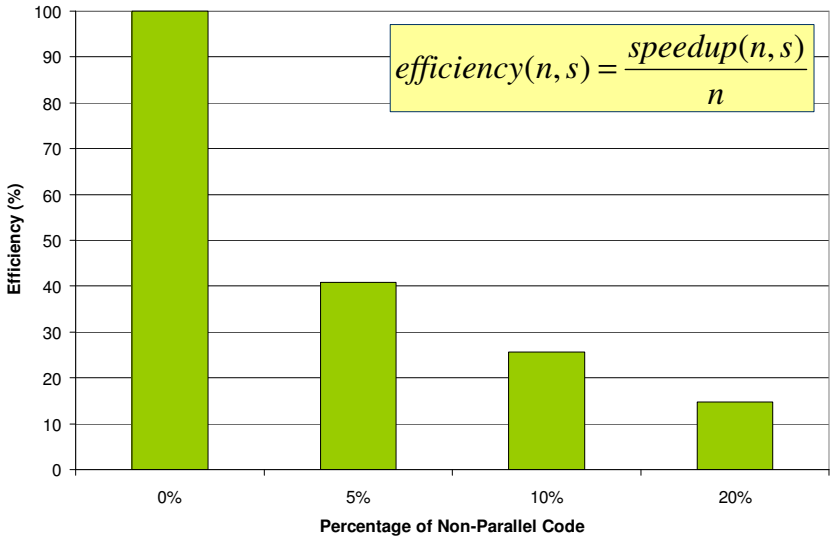
$$speedup(n,s) = \frac{T}{T \cdot s + \frac{T \cdot (1-s)}{n}} = \frac{1}{s + \frac{(1-s)}{n}}$$

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!



Efficiency Using 30 Processors



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

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"

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

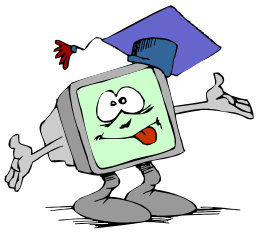
- Sometimes it is advisable to have an extra process, called sink, where the results are sent to

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

That's it!

"Good! Enough of this, show me the API's!"



MPI

- **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

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"

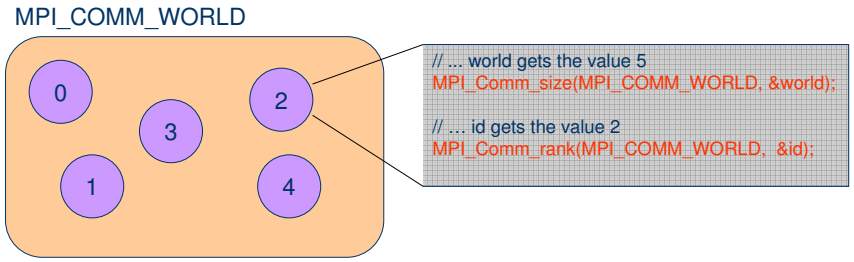
    "Linker Input" = "mpi.lib"
- Running
  - Make sure that the executable is in a shared directory or is available in the same place by all the nodes
  - Make sure you turn off your firewall ☹
  - mpiexec -hosts 2 orion vega \\Mandel\Mandel_MPI.exe
  - mpiexec -machinefile my_cluster -np 2 \\Mandel\Mandel_MPI.exe

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.

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*.



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

Receive a Message

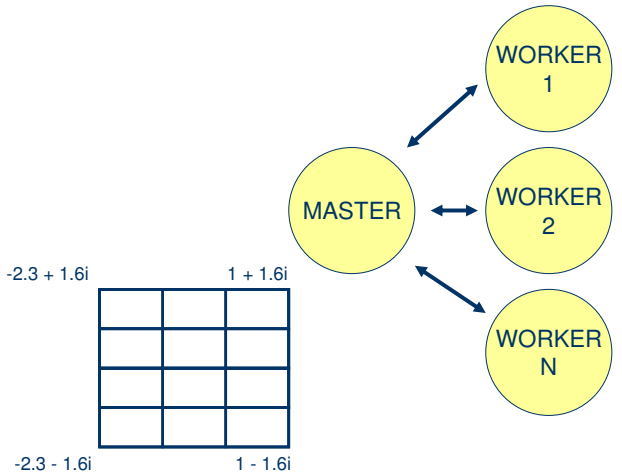
```
int MPI_Recv(void* buffer, int count, MPI_Datatype type,
             int source, int tag, MPI_Comm com, MPI_Status *status);
```

- 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

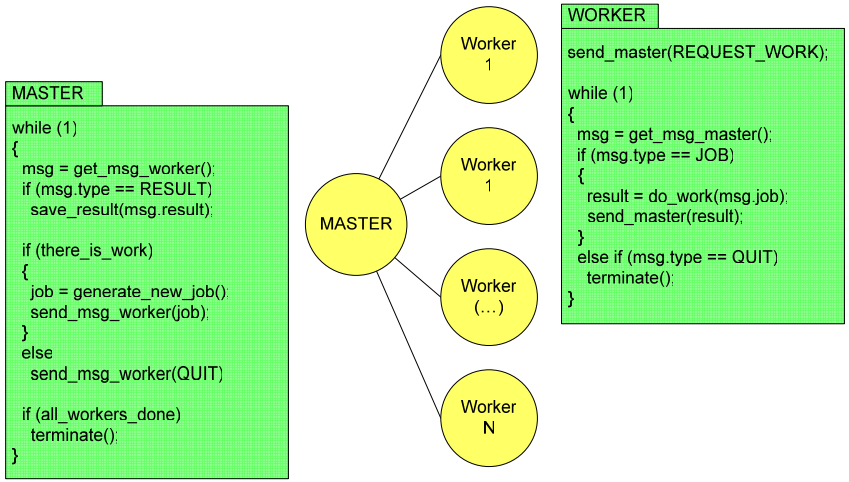
Some Observations

- 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.

An Example of Task Farming



Remember the Task-Farm?



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
```

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;
```

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 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

```
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...
}
```

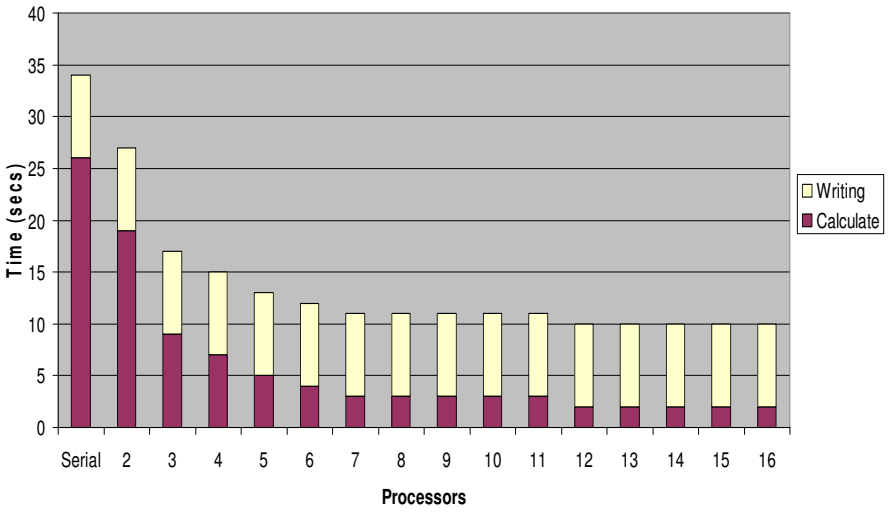
Mandelbrot – The Worker (cont.)

```
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);
}
```

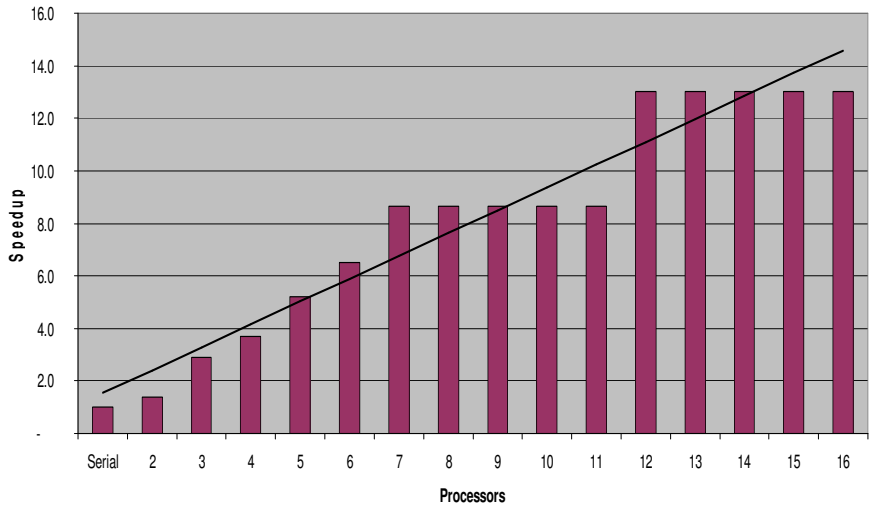
Mandelbrot – Homework

- 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!

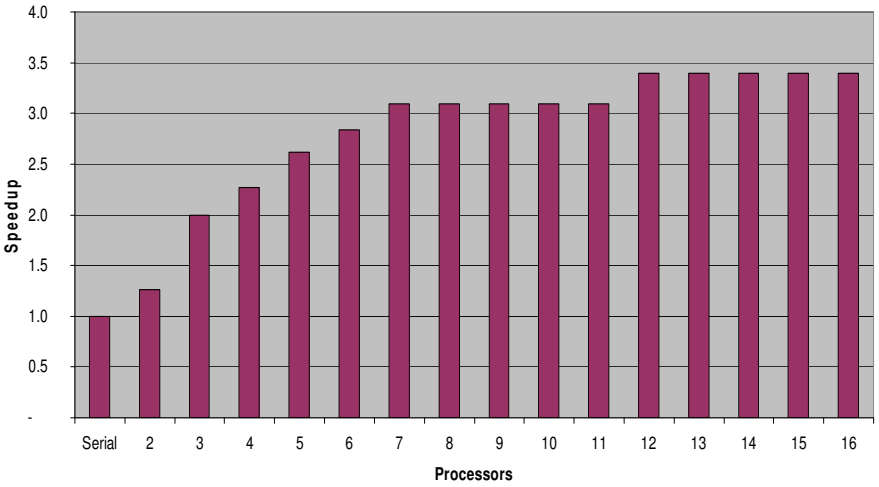
Total Time to Calculate



Speedup (Calculation Part Only)



Speedup (Global/Real)



Amdal's Law in Action!!!!

MPI Routine Types

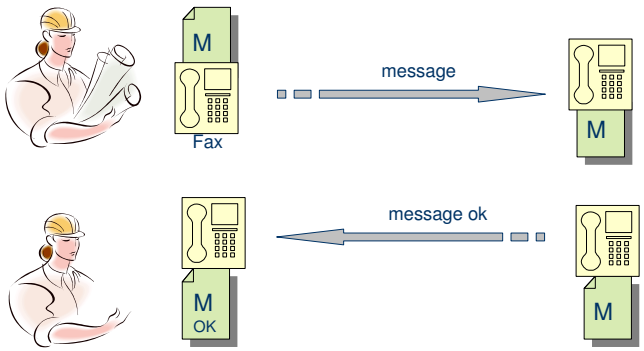
- In MPI there are two major types of communication routines:
  - Point-to-point: where a certain process (*originator*) sends a message to another specific process (*destination*).
    E.g. MPI_Send()/MPI_Recv()
  - Collective: where a certain group of processes performs a certain action collectively.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

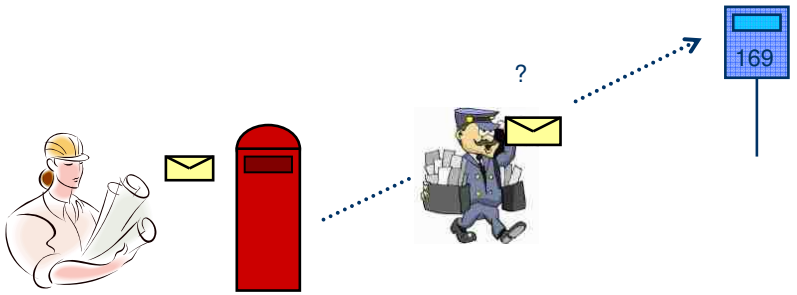
Synchronous Send

- 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.



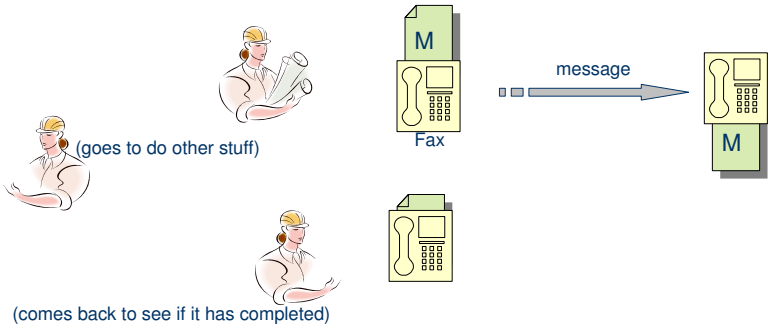
MPI_Send

- 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

- 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



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);
}
```

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

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

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);

(...)
```

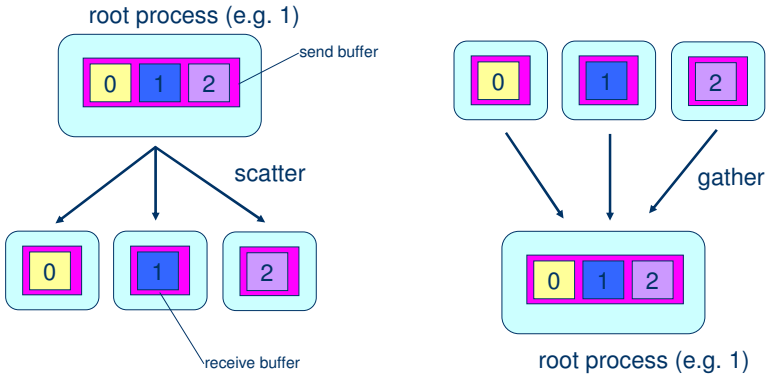
Broadcast

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



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

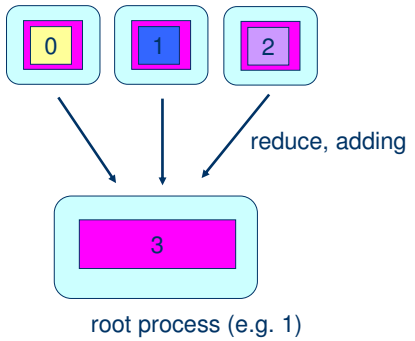
```
int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype
               void *recvbuf, int recvcount, MPI_Datatype recvtype,
               int root, MPI_Comm communicator);
```

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

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);
…
```

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!)

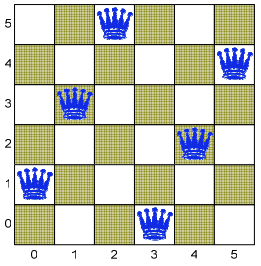
Operation	Meaning
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bitwise and
MPI_LOR	logical or
MPI_BOR	bitwise or
MPI_LXOR	logical xor
MPI_BXOR	bitwise xor
MPI_MAXLOC	maximum and location
MPI_MINLOC	minimum and location

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);

…
```

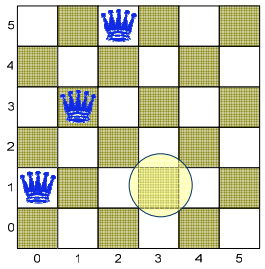
Homework

- 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 backtracking algorithm

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;
}
```

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;
}
```

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

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.)