Massachusetts Institute of Technology | Handout 24
6.892: Theory of Parallel Systems | April 8, 1997
Professor Charles E. Leiserson

Lab 3 — Active Messages

This assignment is due in class on *Tuesday, April 22, 1997*.

The purpose of this programming assignment is to develop an understanding of asynchronous algorithms involving active messages. You will improve existing code that draws pictures of the so-called Mandelbrot set. Your Mandelbrot program will be written in C, and it will run on the `xolas` cluster. The assignment explores issues such as deadlock prevention, request-reply protocols, and termination protocols.

1 The Mandelbrot Set

The Mandelbrot set [1], shown in Figure 1, is the set of all complex numbers c such that if one repeatedly iterates $z \leftarrow z^2 + c$, where initially $z = 0$, the magnitude of z eventually becomes arbitrarily large. The Mandelbrot set is a celebrated example of *fractal*. A fractal is a rough or fragmented geometric shape that can be subdivided in parts, each of which is (at least approximately) a reduced-size copy of the whole. We shall compute a fractal picture of the Mandelbrot set for a rectangular region, where each pixel in the region corresponds to a complex number.

The basic algorithm to compute the Mandelbrot set is the following. For a given complex number c, we initialize $z = 0$. Then, we repeat the computation $z = z^2 + c$ up to a given limit of N times. During the execution of the loop, if the magnitude of z gets too large, we exit the loop. Specifically, we exit if $|z| > 4$, since if $|z|$ gets this big, one can prove that $|z| \to \infty$. If we finish all N iterations of the loop, the point is probably inside the Mandelbrot set. If we exit, the point is outside.

To create a fractal picture for the Mandelbrot set, we shall color every pixel c in a given rectangular region according to how many iterations were completed. The larger the iteration limit N, the sharper the detail in our picture, but also the longer the computation.

2 Parallel Algorithm for Mandelbrot Set

In the asynchronous parallel algorithm to compute Mandelbrot set in a region, each processor works on computing its own portion of the region. Since we do not know ahead of time just how much computation will actually be required to render any subregion, assigning the processors equal-sized regions to compute may lead to a load imbalance in which some processors finish before others. Rather than simply letting processors that finish early sit idle, we use a technique called *work stealing* to balance the work load dynamically and keep processors busy helping to render the image. Recall that work stealing is the same technique that Cilk uses.

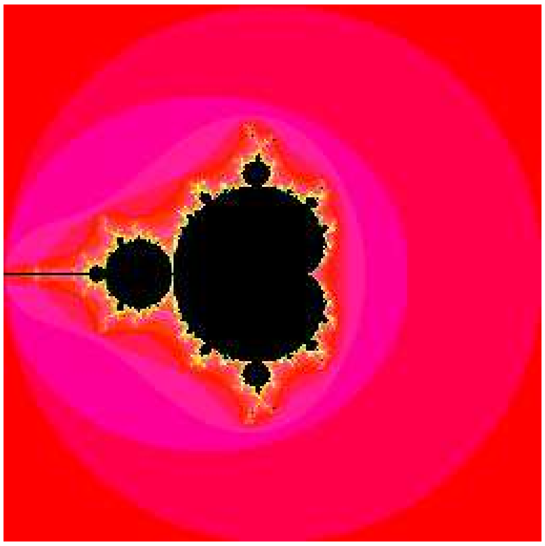

Figure 1: The Mandelbrot set.

Work stealing operates according to the following protocol. When a processor A finishes its local work, it sends a request message to another randomly chosen processor B, asking for more work. Processor A then waits for B to send a reply message. Upon receiving A's request, B attempts to divide its work so that it can give some to A. Processor B replies to A either by indicating that it has no extra work or by sending a description of the work. This process is iterated until all work is completed. This work-stealing protocol is shown in Figure 2.

For the Mandelbrot set problem, the work to be performed by each processor is computing the Mandelbrot set values for a rectangular region of points. When a message requesting work arrives at a processor, the processor splits the region that remains to be computed in half. It keeps one of the two halves for itself to continue working on, and sends the other half in a reply message to the requesting processor. Initially, processor 0's workload is the entire region. The other processors begin with no work and immediately send request messages for work to randomly chosen processors. We'll discuss more details of the algorithm after a brief review of active message passing.

3 Active Messages

An active message is a type of "remote procedure call." An active message contains not only data, but a pointer to a procedure called a *handler*. In an active-message program, the main thread of control of the program is separated from its message-handling threads. When a processor receives a message (by interrupt or polling), the handler for that message is automatically invoked and the data is passed as arguments to the handler.

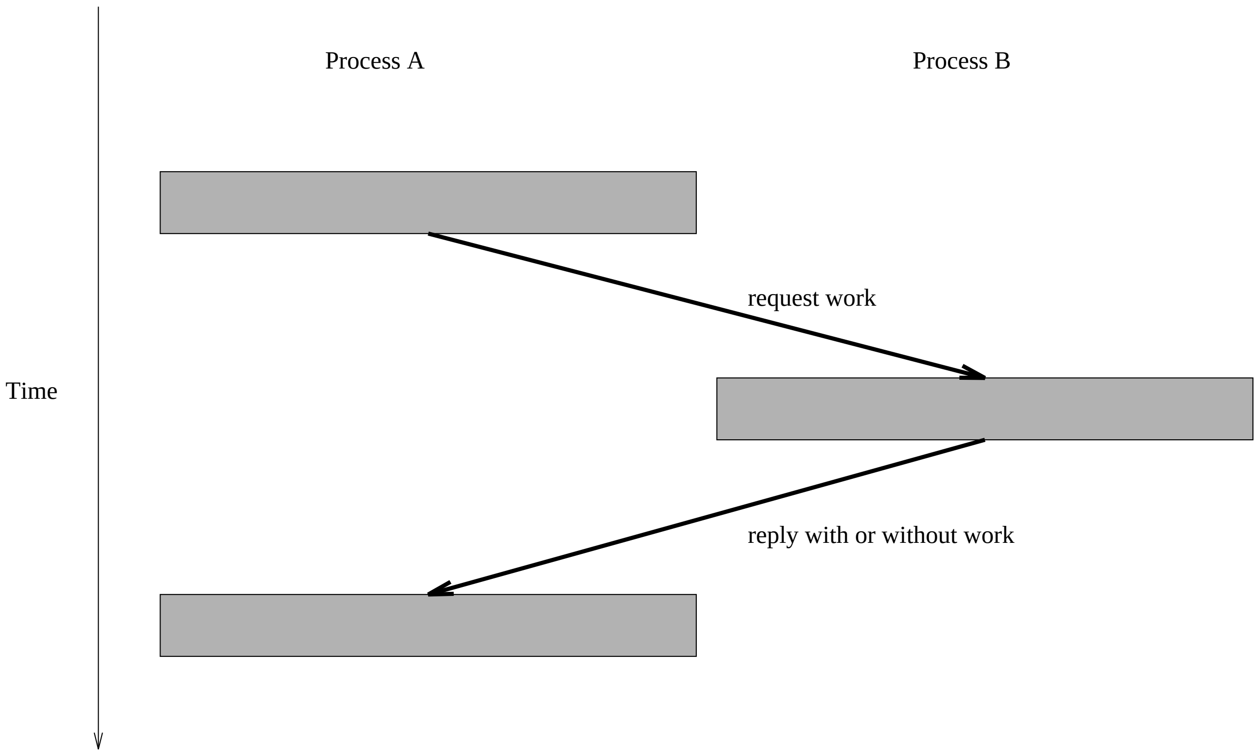

Figure 2: The protocol for stealing work.

handler	data1	data2	data3	data4
word0	word1	word2	word3	word4

Figure 3: The contents of an active message with four data words.

We have built an active-message layer for the `xolas` using the Message Passing Interface (MPI) [2, 3] communication library. (The implementation is included as Appendix A.) This strategy reverses the more frequent situation in which a communication library is built on top of active messages. In each active message, the first word of the message is a pointer to the message handler. The rest of the message is data. For example, Figure 3 shows an active message with four 32-bit words of data. When this message is received, the handler is invoked with its four parameters: *data1, data2, data3, and data4*.

Our active-message layer does not use interrupts. When an active message arrives at a processor, the processor must poll to cause the handler in the message to be invoked. The main thread must poll the network periodically for incoming messages. This can be done with the function `Poll_am()`. When invoked, `Poll_am` tests whether there are pending incoming messages. If so, messages are received and the appropriate handler is invoked with

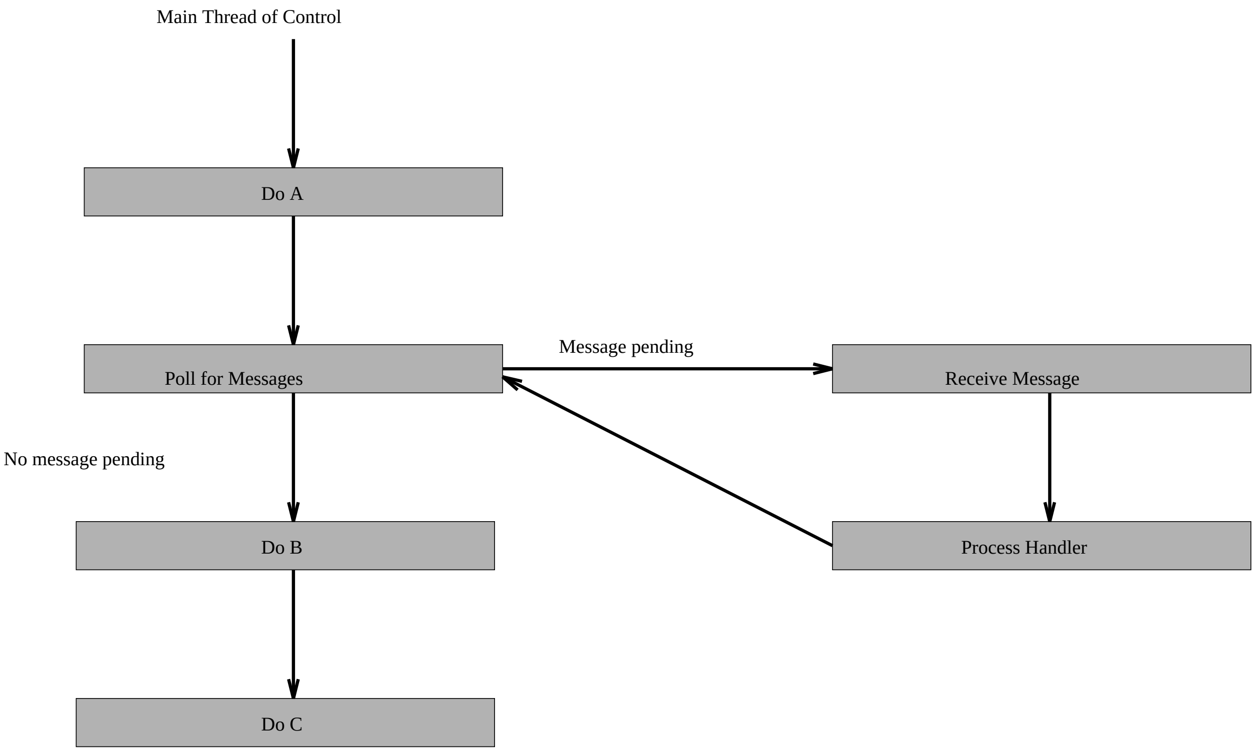

Figure 4: A state diagram showing how polling by the main thread causes handlers to be executed.

the appropriate arguments from the message. After the handler returns, `Poll_am` continues to poll the network, repeating this process, until there are no pending messages (see Figure 4).

There is no linguistic support for active messages: only library functions are provided. Consequently, the user must use the proper form to send an active message depending on the way that the handler will be invoked. For $X = 0, 1, 2, 3, 4$, the functions `Send_am_`X send X integer words of data as an active message:

```
Send_am_X ( int dest_pn,
            void (*handler)(),
            int data1,
            ⋮
            int dataX)
```

The parameters to the functions are as follows:

dest_pn is the destination processor number of the message;

handler is a pointer to the handler function. This function is passed with a pointer *buf* pointing to the received active message stripped off the first word. (Note that the first word of the active message is the handler itself). The prototype of *handler* is

```
void handler(void *buf)
```

data1, ..., dataX are the arguments passed to the handler.

In addition, for $X = 0, 1, 2, 3, 4$, the functions `Send_am_Xp` can be used to send long messages by providing a pointer and length to the data, along with X additional integer arguments:

```
Send_am_Xp ( int dest_pn,
             void (*handler)(),
             int data1,
             ⋮
             int dataX,
             char *p,
             int len)
```

The parameters to the functions are as follows:

dest_pn is the destination processor number of the message;

handler is a pointer to the handler function. This function is passed with a pointer *buf* pointing to the received active message stripped off the first word. The prototype of *handler* is

```
void handler(void *buf)
```

data1, ..., dataX are arguments passed to the handler.

p is a pointer to the character array;

len is the number of the characters in the array;

This function is "blocking," in that control does not return to the caller until it is safe to change the character array. While the caller is blocked, however, handlers on the processor can continue to execute, because the function polls.

When an active message sent with one of the `Send_am_Xp` functions arrives at a processor, the handler is invoked. Within the handler, the pointer argument points to a character array in the buffer of our implementation layer. If the handler polls, this buffer might be modified by the reception of another active message. Consequently, when a handler polls, it should already have made use of all the data in the buffer, copying it to internal storage, if necessary. Such a precaution is not necessary for the integer arguments, because they are passed by value.

To illustrate the use of these active-message functions, let's examine a simple program that implements a request-reply protocol. In this program, processor 0 sends a request message to all other nodes. Upon receiving all replies, it prints the number of processors and the total amount of time to send all requests and receive all replies. Observe the simplicity of the main thread of control (function `main`).

```
#include <stdio.h>
#include <sys/time.h>
#include "am.h"

int done = 0;                 /* Becomes 1 if node received request */
int messages_outstanding = 0; /* Counts number of replies required  */

void ping();
void pong();

int main(int argc, char *argv[])
{
  int i, my_rank, nprocs;
  struct timeval start_wtime, stop_wtime;   /* Records elapsed time */

  Get_environment(argc, argv, &my_rank, &nprocs);
          /* Get the ID of the processor & the number of processors */

  if (my_rank == 0) {
    gettimeofday(&start_wtime, NULL);

    /* Send a request message to each processor                     */
    for (i = 1; i < nprocs; i++) {
      messages_outstanding++;
      Send_am_1(i, ping, my_rank);
    }

    /* Poll the network until all reply messages are received       */
    while (messages_outstanding > 0)
      Poll_am();

    gettimeofday(&stop_wtime, NULL);
    printf("On a partition of %d nodes, elapsed = %lg\n", nprocs,
           (stop_wtime.tv_sec - start_wtime.tv_sec) +
           (stop_wtime.tv_usec - start_wtime.tv_usec) * 1E-6);
  }
  else {
    while (!done)
      Poll_am();
  }
}
```

```
void ping(int *return_node)       /* Request handler                 */
{
  Send_am_0(*return_node, pong);
  done = 1;
}

void pong()                       /* Reply handler                   */
{
  messages_outstanding--;
}
```

4 Pseudocode for Mandelbrot

Let us now return to the Mandelbrot problem. A program `mandelbrot.c` has already been provided for you to compute the Mandelbrot set using the active message layer. Your assignment will be to improve the program. The Mandelbrot program uses auxiliary functions in the file `gui.c` to display the picture on an X window. The program is able to receive key or mouse input from a user in order to determine what region of the Mandelbrot set to display. Specifically, the user can resize the window or select a rectangular area of the window and have that area blown up, zoomed out, expanded, or shrunk. In addition, the user can terminate the program. You can find the program in `/usr/local/6.892/lab-3/mandelbrot.c` on `xolas`.

The compute routine (`compute_mandelbrot`) for the main thread of the Mandelbrot set program is described in the following pseudocode:

`compute_mandelbrot`

While the user doesn't terminate the program
  If there is no local work left to do
  Then attempt to steal work from another processor
  Else compute the Mandelbrot set on next line of the local region
  End If
  Poll for incoming messages
End While

During the main thread of the program, either a processor is doing work (by computing the Mandelbrot set) or it has no work and is trying to steal some from another processor. At the end of each iteration, the processor checks the network for request messages (those requesting work).

The pseudocode for the `steal_work` routine is as follows:

`steal_work`

Randomly select a processor to send request message to
Set flag to indicate that reply has not been received

Send the request message to the processor
While reply has not been received
Poll the network for incoming messages
End While

The function that sends the request message polls the network and invokes a handler if there is an incoming message. This is because other messages may arrive while the processor polls waiting for the reply message, and they must be dealt with (otherwise deadlock occurs).

The next two functions are the message handlers. The function `request_work_handler` handles request messages for work, and the function `reply_with_or_without_work_handler` handles the reply.

`request_work_handler`
Attempt to divide local work
If there is no work to give to requesting processor
Then send a reply indicating there is no work
Else construct reply message so as to indicate what work should be done
Send reply message with work
End If

Upon sending the reply in `request_work_handler`, the processor polls the network for messages. Since this processor may have an outstanding request message, it may invoke the reply handler from this routine.

`reply_with_or_without_work_handler`
If reply contains work
Then Setup work to do (if there is any)
Indicate to other processors that not all local work is done
End If

5 Running Your Program

To run the Mandelbrot program, you must first log in to `xolas` and set up the necessary environment for programming: copy the directory `/usr/local/6.892/lab-3` into your directory, and append the file `dot.login` to your `.login`. This procedure will enable you to use MPI.

In the lab directory you will find the source code `mandelbrot.c`, `gui.c`, `mandelbrot.h`, and a makefile.

1. You may built the executable `mandelbrot` by typing

```
make
```

Once two programs compile and link successfully, you are ready to run it:

```
mpirun mandelbrot
```

2. After you've gotten `mandelbrot` up and running, you can give commands to its user interface. Commands are as follows:

   resize the window This is generally done by grabbing a handle on the upper-right corner of the window and stretching the window to be the desired shape.

   zoom in By "clicking and dragging," you can select a rectangular region of your window and have that area of the screen blown up. To zoom in, move the mouse to one corner of the region. Press and hold down a mouse button. Drag the mouse to the diagonally opposite corner of the region. Release the mouse button. At this point, a new window is created which has the shape of the region you selected, but which has the same area as the original window. The contents of the original window that lie within the selected rectangle expand to fill the new window.

   zoom out Typing "`z`" in the window "zooms out". The current image is redrawn in the center 1/9th of the window, while the remainder of the window is filled with the surrounding area.

   expand Typing "`e`" expands the window by doubling the width and height of the current window.

   shrink Typing "`s`" shrinks the window by halving the width and height of the current window.

   quit Typing "`q`" in the window quits the program. (Implemented, but improperly.)

   exit Typing "`x`" in the window completes the current Mandelbrot set then exits the program. (Unimplemented.)

6 The Programming Assignment

The programming assignment is broken into five parts. With each part, you will be required to think more deeply about the protocol issues involving active messages. (Part 5 requires no implementation, only thinking.)

Part 1: Implementing work stealing

The function `steal_work` is called by `compute_mandelbrot`. The pseudocode for `steal_work` is given in Section 4, but the source code for `steal_work` is not provided. Write your own version of `steal_work`. The template for that function is provided in `mandelbrot.c`.

Since `steal_work` needs to choose a random victim, you may find the C library function `random()` to be of some use. This function returns a pseudorandom number in the range from 0 to $2^{31}-1$. You also may find it helpful to know that we have defined a global variable `nprocs` that contains the number of processors actively performing the computation. These processors have ID's (ranks in MPI terminology) ranging from 0 to `nprocs` -1. The value `nprocs` may be less than the number of processors in the machine, which allows us to run on fewer than the maximum number of processors.

Once you complete the function `steal_work`, an idle processor should be able to steal work from busy processors. The program runs by default on 1 processor. To change this to n, type the following in your UNIX shell:

```
mpirun -np n mandelbrot
```

`mpirun` will actually distribute your job across the machines on the `xolas` cluster. Since the active messages library runs on top of MPI that runs on top of TCP that runs on top of IP that runs of top ..., it is unlikely that you get any speedup. This lab deals with correctness issues, and not with speed. (Keep in mind that active messages are intended to run on the bare metal, but have no way to do that on `xolas` with the hardware at our disposal).

Part 2: Writing proper reply handlers

The following protocol is often used in message passing:

1. Processor B sends a message to processor A containing data.
2. Processor A receives the message (by interrupt or polling the network) and decides what to do with the received data.

The difficulty with this protocol is that it may be the case that B is sending many different kinds of data, each of which should be operated on by A in a different way. This is against the spirit of active messages: A decides what to do with the data when B perfectly knows how A should handle it.

The function `request_work_handler` handles request messages sent to a processor B by another processor A (see Figure 5). Whether or not B has work to give to A, `request_work_handler` sends a reply which invokes the handler `reply_with_or_without_work_handler` on A. In turn, `reply_with_or_without_work_handler` looks at its second parameter to determine whether work was sent in the reply message. This is precisely the unnatural protocol just described. Since B knows whether work is being sent, B should specify the appropriate reply handler in the reply message, and not encode this information a data word.

Use the active-message layer to create two reply handlers replacing `reply_with_or_without_work_handler`. One handler should handle replies containing work, and the other should handle replies which do not, as is shown in Figure 6. The procedure `request_work_handler` should then specify the appropriate handler in its `Send_am_`X calls.

Part 3: Quitting gracefully

Currently, the program may hang when the user keys in `q`. This ungraceful behavior arises, because once the root processor receives the termination signal from the user, it simply sends

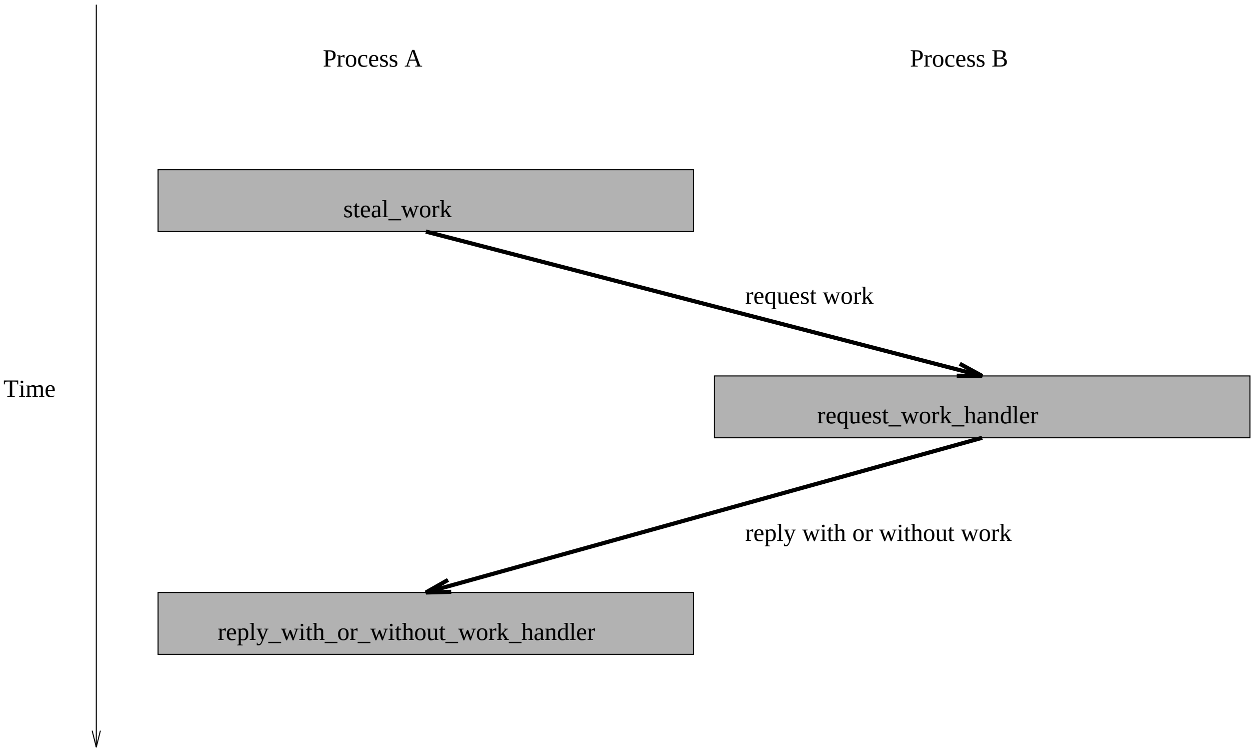

Figure 5: The appropriate action is chosen by the requesting processor.

the termination message to the remaining processors and then quits. If other processors are in the midst of a steal transaction, i.e., waiting for reply of a steal request, then those processors will hang, because they will never receive a steal reply from a processor that has already quit. Modify the program `mandelbrot.c` so that your program quits elegantly.

Part 4: Termination detection

When the user keys in `x`, the current Mandelbrot set should be completely computed, and then the program should terminate. This functionality is not yet implemented. Your job is to implement it.

In order to check for termination (i.e., to determine whether the entire region has been computed), the `reply_with_work_handler` should first be modified to send an acknowledgment message to the victim indicating that the work has been received.

A processor should terminate when when the following two conditions hold:

T1 All processors have indicated that they have finished computing their regions.

T2 The processor has received as many acknowledgments as it has sent replies containing work.

To check termination condition T1, one easy solution is for the root processor to record the number of lines that have been rendered. If all the lines in the window have been

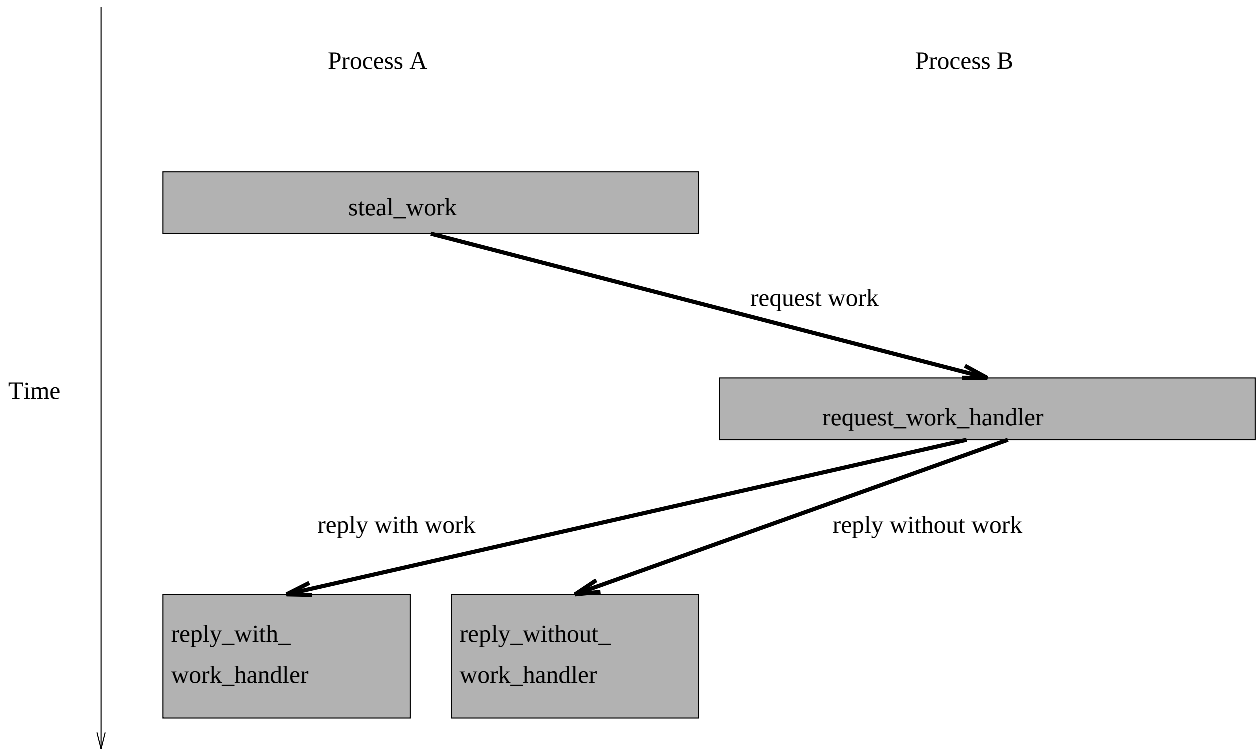

Figure 6: The appropriate action is chosen by the requesting processor.

rendered, then all processors complete their work. This solution relies on the fact that the root processor controls the display, and is therefore in a position to know when the computation is done. We would like you to implement a more robust solution that does not rely on any one processor having this global knowledge of the state of the computation. The robust solution will require you to program an **asynchronous global OR barrier** to run concurrently with the rest of the Mandelbrot program.

The robust termination algorithm works as follows:

Step 1: Each processor contributes a zero value to the barrier if it has done no work since the last barrier synchronization. Otherwise, it contributes a nonzero value to the barrier.

Step 2: After the processor contributes the value, it continues to do work or steal, repeatedly until the processor receives the result of the global OR.

Step 3: If the global OR result is zero, then the processor exits if the termination condition T2 is also met. Otherwise, the processor repeats the algorithm to start a new round of barrier synchronization.

The reason the termination algorithm works is as follows. If all processors contribute zero to the barrier, then all processors have been idle since the last barrier synchronization. Therefore, the termination condition T1 is satisfied. If one processor contributes a nonzero value, however, then that processor may still have some work, and the termination condition T1 is not met.

Use the active message functions to implement an asynchronous global OR barrier, and implement the above termination protocol. Why is it insufficient in Step 1 of the algorithm for a processor to contribute a zero value to the barrier if it *currently* has no work?

Part 5: Stack space

A request handler is allowed to call `Send_am_`X to reply. But since `Send_am_`X polls the network, a new request message may be discovered that causes another request handler invocation. As a consequence, an indefinite number of handler invocations may occur, possibly overflowing a finite-sized call stack.

In the program, the `request_work` handler calls `Send_am_`X to send the acknowledgment message. The call to `Send_am_`X polls the network. Can this polling result in an infinite recursion of handler invocations? If so, describe the situation. If not, give a worst-case scenario, and calculate (asymptotically) how many levels deep this recursion can go. Suppose that we allow processors to make arbitrarily many requests even before getting any replies. Now, can an infinite recursion of handlers occur? Explain as before.

Part 6: Better termination detection

Your implementation of Part 4 just quits when the current image has been computed. This is annoying, because you do not get a chance to see the image. Modify the program so that, instead of quitting, it asks the user whether it wants to quit or continue. The user must interact with processor 0, which in turns tells the other processors what to do.

A Appendix: Implementation of Active Messages

Our implementation of `Send_am_`X, `Send_am_`X`p` and `Poll_am` uses following MPI subroutines:

- `MPI_Issend`. The prototype for the subroutine is

  ```
  int MPI_Issend(void *buf, int count, MPI_Datatype datatype, int dest,
                 int tag, MPI_Comm comm, MPI_Request *request).
  ```

  `MPI_Issend` is *synchronous*, *nonblocking* send. It completes as soon as the sender "knows" the transfer of data will complete, but before the the receiver "knows" the transfer will complete. This subroutine sends the buffer pointed to by `buf` to processor `dest`. The buffer is assumed to contain an array of `count` objects of type `datatype`. `Tag` is used by the receiving processor, which can choose to receive messages with specific tags. `Comm` should be set to `MPI_COMM_WORLD`, `request` is returned from the call to be used for detect the completion of a nonblocking send.

- `MPI_Test`. The prototype for the subroutine is

  ```
  int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
  ```

  A call to `MPI_Test` returns `flag` as `TRUE` if the operation identified by `request` is completed. In such a case, the status object is set to contain information on the completed operation. The call returns `flag` as `FALSE`, otherwise.

- `MPI_Recv`. The prototype for the subroutine is

  ```
  int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,
               int tag, MPI_Comm comm, MPI_Status *status).
  ```

  The subroutine receives one message into the buffer pointed to by `buf`. The message must be from processor `source`, unless `source` is `MPI_ANY_SOURCE`, in which case a message from any processor can be received. The message must carry the specified `tag`, unless `tag` is `MPI_ANY_TAG`, in which case a message with any tag can be received. The meaning of `count` and `datatype` is identical to their meaning in `MPI_Send`. `Comm` should be set to `MPI_COMM_WORLD`. `Status` contains two fields, `MPI_SOURCE` and `MPI_TAG` that store the source and tag of the message. This may be useful if `MPI_ANY_SOURCE` or `MPI_ANY_TAG` are specified. You may retrieve them by using the contents of `status.MPI_SOURCE` and `status.MPI_TAG`.

- `MPI_Iprobe`. This subroutine checks whether a message is available. If you call `MPI_Recv` and no appropriate message is available, your program will block until one arrives (on the processor that issued the call). The prototype for the subroutine is

  ```
  int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag,
                 MPI_Status *status).
  ```

  The subroutine sets `flag` to `TRUE` if a message with appropriate tag and source is available. If it is, then `status` has the same information about the waiting message as it would if that message were received. The `status` returned from `MPI_Iprobe` also contains information of the length of the message received. However, this information is not directly available as a field of the status variable and a call to `MPI_Get_count` is required to "decode" this information.

- `MPI_Get_count`. The prototype for the subroutine is

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

  This subroutine stores the number of entries (each of type `datatype`) received in `count`.

The MPI code for `Send_am_2` and `Poll_am` is as follows:

```
long send_buf[MAX_BUFFER_LEN];
long recv_buf[MAX_BUFFER_LEN];

#define OFFSET(handler) ((int) handler - (int) send_buf)
#define ADDRESS(disp)   (disp + (int) send_buf)

void Send_am_2(int dest_pn, void (*handler)(), int data1, int data2)
{ int flag;
  MPI_Status status;

  MPI_Test(&request, &flag, &status);
  while (flag == FALSE) {
    Poll_am();
    MPI_Test(&request, &flag, &status);
  }
  send_buf[0] = OFFSET(handler);
  send_buf[1] = data1;
  send_buf[2] = data2;
  MPI_Issend(&(send_buf[0]), 3*sizeof(int), MPI_BYTE, dest_pn, 0,
             MPI_COMM_WORLD, &request);
}

void Poll_am()
{ int flag, count;
  MPI_Status status;

  for (;;) {
    MPI_Iprobe(MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &flag, &status);
    if (flag) {
      MPI_Get_count(&status, MPI_BYTE, &count);
      MPI_Recv(&(recv_buf[0]), count, MPI_BYTE, status.MPI_SOURCE, 0,
               MPI_COMM_WORLD, &status);
      (*(void (*)(void *))ADDRESS(recv_buf[0]))(recv_buf+1);
    }
    else
      break;
  }
}
```

Two buffers `send_buf` and `recv_buf` are used for sending and receiving active messages. The sender converts the absolute address of `handler` into the relative address (offset related to `send_buf`), while the receiver converts the latter back to the absolute address. Therefore, even if `handler` is allocated with different (absolute) addresses on different processors, the receiver could get the correct handler address from the relative address.

It is important to know that `Send_am_`X also polls for messages. In particular, if it attempts to send a message when the network busy, or when the system message buffer is full it polls the network. If the send were to block without polling until the network was free, deadlock could result. By polling whenever the network is busy, active messages satisfy the "consumption assumption" that the network will eventually deliver all messages as long as the user promises to eventually remove all delivered messages from the network.

B Acknowledgements

The code for this lab and most of this handout has been written by Mingdong Feng of the National University of Singapore, while teaching with Charles Leiserson the class CS413: Introduction to Parallel Systems, Spring 1996. The original copyright notices are still in the code. The code has been slightly modified to suit `xolas`' idiosyncrasies and my own stylistic paranoias.

References

[1] R. Devaney and L. Keen, eds., *Chaos and Fractals: The Mathematics Behind the Computer Graphics*, American Mathematical Society, Providence, RI, 1989.

[2] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra, *MPI: The Complete Reference*, The MIT Press, 1995.

[3] Message Passing Interface, at `http://www.mcs.anl.gov/mpi/`.