Parallel Programming Lab, Week 7: An MPJ Task Farm

All our earlier examples of parallel programs followed the pattern of breaking a problem into fixed "chunks" (or *domains*). Each thread or process was assigned one chunk - often at the start of the program - and was responsible for the computations associated with that part of the problem.

A different and more dynamic approach, applicable to some problems, is a *task farm*. In this approach, members of a pool of worker processes or threads grab small parts of a problem, work on those until they are complete, then grab new tasks, until the problem as a whole is solved. This approach is particularly suitable for *embarassingly parallel* problems, and in this lab we will return to one such problem - Mandelbrot Set computation.

Apart from being a useful paradigm in their own right, task farms allow us to explore some new features of message passing in MPI.

A Slow Mandelbrot

There are many problems that are very demanding computationally and also happen to be embarassingly parallel, thus lending themselves well to task farming. One example might be rendering of elaborate scenes in an animated film.

A simple rendering of the Mandelbrot set is embarassingly parallel but less computationally demanding. To illustrate the principles of task farming we artificially increase the work load by making the "cut off" value that determines whether points are within the Mandelbrot Set artificially high. You may remember how in our calculation, the constant CUTOFF determined the number of iterations of the inner loop executed by points within the inner "black" region of the set. Larger values of CUTOFF *are* needed for high resolution plots of the the details of the set, but here we just use this as a device to ramp up the amount of computation, to model more challenging problems.

Try running this new sequential version of Mandelbrot, where we have also added a kind of graphical monitor of the progress of the calculation:

```
import java.awt.* ;

import javax.swing.* ;


public class SlowMandelbrot {


    final static int N = 1024 ;

    final static int CUTOFF = 100000 ;


    static int [] [] set = new int [N] [N] ;


    public static void main(String [] args) throws Exception {


        Display display = new Display() ;


        // Special value for uninitialized pixels (rendered white below)


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

            for(int j = 0 ; j < N ; j++) {

                set [i] [j] = -1 ;

            }

        }

        display.repaint() ;


        // Calculate set


        long startTime = System.currentTimeMillis();
```

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

        double cr = (4.0 * i - 2 * N) / N ;
        double ci = (4.0 * j - 2 * N) / N ;

        double zr = cr, zi = ci ;

        int k = 0 ;
        while (k < CUTOFF && zr * zr + zi * zi < 4.0) {

            // z = c + z * z

            double newr = cr + zr * zr - zi * zi ;
            double newi = ci + 2 * zr * zi ;

            zr = newr ;
            zi = newi ;

            k++ ;
        }

        set [i] [j] = k ;
    }
    display.repaint() ;
}
```

```
        long endTime = System.currentTimeMillis();

        System.out.println("Calculation completed in " +
                                     (endTime - startTime) + " milliseconds");
    }

    static class Display extends JPanel {

        Display() {

            setPreferredSize(new Dimension(N, N)) ;

            JFrame frame = new JFrame("MandelBrot");
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setContentPane(this);
            frame.pack();
            frame.setVisible(true);
        }

        public void paintComponent(Graphics g) {
            for(int i = 0 ; i < N ; i++) {
                for(int j = 0 ; j < N ; j++) {
                    int k = set [i] [j] ;
                    Color c ;

                    if(k == -1) {    // uninitialized
                        c = Color.WHITE ;
                    }
                    else {
                        float level ;
                        if(k < CUTOFF) {
                            level = k < 50 ? (float) k / 50 : 1.0F ;
                        }
                        else {
                            level = 0 ;
                        }
                        c = new Color(level/2, level/2, level) ;    // Blueish
                    }

                    g.setColor(c) ;
                    g.fillRect(i, j, 1, 1) ;
                }
            }
        }
    }
}
```

Make a note of how long this takes. If you are worried about the overhead of the graphics, reduce the frequency of the display (at the moment the display is repainted at the end of each "j-loop").

An MPJ Task Farm

The logic of a basic task farm in MPI has some subtleties. But the basic framework of the program we present can be reused for all kinds of problem - only isolated parts of the code are specific to our Mandelbrot Set calculation.

Before reading the code, it would be useful to look at this week lecture on MPI features.

It discusses in particular use of tags and wildcards in send and receive.

One process - most naturally process 0 - will act as the master process, controlling the farm. The other processes - *P*-1 of them - will work as slaves or workers. The logic of the worker processes is simplest.

The Worker Logic

The description here relates to the full code given below. You might find it convenient to copy that to a separate editor before reading this description.

Each worker first sends a "hello" message to the master, which just alerts the master that this worker is ready to receive work. There doesn't have to be any data in this message (MPI happily allows messages where the number of elements in the data buffer is zero).

In a loop, the worker then receives tasks from the master. In our example the task message only needs to contain the start address of a block of rows of the image that the worker should now compute.

When these have been computed, the rows are sent back to the master in a "result" message. Then the worker goes back to waiting for the next task.

If at any time the worker receives a "goodbye" message from the master instead of a new task, this indicates that the calculation is complete, and the worker shuts down.

The Master Logic

The master is responsible for farming out blocks of the image to free worker processes, and for collating together incoming results ready (in our example) for display.

In its main loop, the master first waits for an incoming request for work from a free worker. In our implementation, a "request" for new work can either take the form of the initial "hello" message from a worker, or it can come in the form of a completed result from an earlier task, implying the worker is ready from another task.

The method invocation that deals with incoming requests:

```
  Status status =
              MPI.COMM_WORLD.Recv(buffer, 0, BUFFER_SIZE, MPI.INT,
                                         MPI.ANY_SOURCE, MPI.ANY_TAG) ;
```

illustrates some interesting features of the MPI Recv operation.

The first point to note is that this message could come from *any* worker process.

The *src* argument takes the special value MPI.ANY_SOURCE to indicate this. The second point is that in this code we are for the first time making serious use of the *tag* feature in MPI. This is a user-defined integer value embedded in a message that indicates the *purpose* of the message. In this case a message coming from a worker can have either of the tags TAG_HELLO or TAG_RESULT. The receive operation accepts messages with either of these tags, as indicated by the special value MPI.ANY.

Another feature we are using here for the first time is the fact that the Recv method actually returns an object of type Status. This object contains metadata about the message just received. In particular it has fields that identify the *actual* source process and the *actual* tag of the message received. These fields are used in our master code. The Status object also contains information on the *actual* number of elements received in the message and copied to buffer. Although we won't use this capability, you can see how this might be useful because in our example BUFFER_SIZE is only an *upper limit* on the number of such elements. "Result" messages will have this many elements, but "hello" messages in fact contain zero data elements.

If the tag in the received message indicates it contains a block of results, these are copied from buffer to appropriate part of the array set. Notice that the *first* element of buffer contains the position in set where these rows belong. The worker must return this information, because the master doesn't keep any record of which task was sent to which worker.

In any case, the master then sends a new task to the worker from which it just received a request - identified by status.source. In our case this message just contains one number - the address on the first row in the next block to be calculated. If all blocks have already been dispatched to workers, the master instead sends a "goodbye" message to the worker.

Notice that nextBlockStart is the address of the next row that is to be sent to workers. The results may arrive back much later, and usually by the time they do this counter will have been incremented as tasks are sent to other workers. This is why the result message needs to contain the corresponding value for the actual task that was processed, so that the results can go in the right place in set.

The condition on the main loop in the master probably needs some comment. The main part of the condition is just

```
    while(numBlocksReceived < NUM_BLOCKS ...)
```

We keep looping until all blocks have come back from workers (*not* the same condition as all blocks having been *sent out* to workers, which was nextBlockStart < N). The second part of the condition:

```
numHellos < numWorkers
```

isn't quite essential, but it deals with the unlikely case that some very slow workers don't send their hello messages until all work has been finished. With this condition the master will at least stay alive long enough to send a goodbye message to such latecomers, shutting them down cleanly.

The Full Code

```
import mpi.* ;


import java.awt.* ;
import javax.swing.* ;


public class MPJSlowMandelbrot {


    final static int N = 1024 ;
    final static int CUTOFF = 100000 ;


    final static int BLOCK_SIZE = 4 ;   // rows in block of work


    final static int NUM_BLOCKS    = N / BLOCK_SIZE ;
    final static int BUFFER_SIZE = 1 + BLOCK_SIZE * N ;


    // tag values
    final static int TAG_HELLO      = 0 ;
    final static int TAG_RESULT    = 1 ;
    final static int TAG_TASK       = 2 ;
final static int TAG_GOODBYE = 3 ;

static int [] [] set ;

public static void main(String [] args) throws Exception {

    MPI.Init(args) ;

    int me = MPI.COMM_WORLD.Rank() ;
    int P = MPI.COMM_WORLD.Size() ;

    int numWorkers = P - 1 ;

    int [] buffer = new int [BUFFER_SIZE] ;

    if(me == 0) {    // master process - sends out work and displays results

        set = new int [N] [N] ;
        Display display = new Display() ;

        for(int i = 0 ; i < N ; i++) {
            for(int j = 0 ; j < N ; j++) {
                set [i] [j] = -1 ;
            }
        }
        display.repaint() ;
        // Calculate set

        long startTime = System.currentTimeMillis();

        int nextBlockStart = 0 ;
        int numHellos = 0 ;
        int numBlocksReceived = 0 ;

        while(numBlocksReceived < NUM_BLOCKS || numHellos <
numWorkers) {

            // Receive hello or results from any worker

            Status status =
                    MPI.COMM_WORLD.Recv(buffer, 0, BUFFER_SIZE,
MPI.INT,
                                        MPI.ANY_SOURCE,
MPI.ANY_TAG) ;

            if(status.tag == TAG_RESULT) {

                // Save returned results to `set' and display

                int resultBlockStart = buffer [0] ;
                for(int i = 0 ; i < BLOCK_SIZE ; i++) {
                    for(int j = 0 ; j < N ; j++) {
                        set [resultBlockStart + i] [j] = buffer [1 + N * i + j] ;
                    }
                }
                numBlocksReceived++ ;
                display.repaint() ;
            }
            else {    // tag is TAG_HELLO
                numHellos++ ;
            }

            // Send next block of work or finish tag to same worker
            if(nextBlockStart < N) {
                buffer [0] = nextBlockStart ;
                MPI.COMM_WORLD.Send(buffer, 0, 1, MPI.INT, status.source,
TAG_TASK) ;
                nextBlockStart += BLOCK_SIZE ;
                System.out.println("Sending work to " + status.source) ;
            }
            else {
                MPI.COMM_WORLD.Send(buffer, 0, 0, MPI.INT, status.source,
TAG_GOODBYE) ;
                System.out.println("Shutting down " + status.source) ;
            }
        }

        long endTime = System.currentTimeMillis();

        System.out.println("Calculation completed in " +
                           (endTime - startTime) + " milliseconds");
    }
    else {    // worker process

        // Send request to master for a first block of work

        MPI.COMM_WORLD.Send(buffer, 0, 0, MPI.INT, 0, TAG_HELLO) ;

        boolean done = false ;
        while(!done) {
            Status status = MPI.COMM_WORLD.Recv(buffer, 0, 1, MPI.INT, 0,
MPI.ANY_TAG) ;

            if(status.tag == TAG_TASK) {
                int blockStart = buffer [0] ;

                for(int i = 0 ; i < BLOCK_SIZE ; i++) {
                    for(int j = 0 ; j < N ; j++) {

                        double cr = (4.0 * (blockStart + i) - 2 * N) / N ;
                        double ci = (4.0 * j - 2 * N) / N ;

                        double zr = cr, zi = ci ;

                        int k = 0 ;
                        while (k < CUTOFF && zr * zr + zi * zi < 4.0) {

                            // z = c + z * z
                            double newr = cr + zr * zr - zi * zi ;
                            double newi = ci + 2 * zr * zi ;

                            zr = newr ;
                            zi = newi ;

                            k++ ;
                        }

                        buffer [1 + N * i + j] = k ;
                    }
                }
                buffer [0] = blockStart ;
                MPI.COMM_WORLD.Send(buffer, 0, BUFFER_SIZE, MPI.INT, 0,
TAG_RESULT) ;
            }
            else {    // tag is TAG_GOODBYE
                done = true ;
            }
        }
    }

    MPI.Finalize() ;
}

static class Display extends JPanel {
```

```
Display() {

    setPreferredSize(new Dimension(N, N)) ;

    JFrame frame = new JFrame("Mandelbrot");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setContentPane(this);
    frame.pack();
    frame.setVisible(true);
}

public void paintComponent(Graphics g) {
    for(int i = 0 ; i < N ; i++) {
        for(int j = 0 ; j < N ; j++) {
            int k = set [i] [j] ;
            Color c ;

            if(k == -1) {   // uninitialized
                c = Color.WHITE ;
            }
            else {
                float level ;
                if(k < CUTOFF) {
                    level = k < 50 ? (float) k / 50 : 1.0F ;
                }
                else {
                        level = 0 ;
                    }
                    c = new Color(level/2, level/2, level) ;    // Blueish
                }

                g.setColor(c) ;
                g.fillRect(i, j, 1, 1) ;
            }
        }
    }
  }
}
```

Experiments and Questions

Review the instructions from the last two weeks about running MPJ programs on the Hadoop cluster. We will again rely on X11 forwarding, so the instructions in week 6 will be especially relevant.

Compile and run the code above. As usual you can start by running the parallel program in multicore mode, but the most of the interest is in cluster mode. In either case, *note the number of workers is* one less than *the* -np *value you specifify to mpjrun*

Explore the limits of parallel speedup you can obtain on this problem by running across multiple nodes of the cluster. Think big - in recent years the most computers a student ran programs like this on was all 70-odd PCs in the LG0.14a/b labs! This year you will be running on the Hadoop cluster, but you still potentially have nine nodes with 12 cores each at your disposal.

Try changing BLOCK_SIZE to see whether that helps.

How would you modify the program to farm out square blocks of the image instead of vertical strips, and would this be an improvement? (There is a sample solution here, which you may also try running.)

"Slow Mandelbrot" Task Farm with Square Blocks

```
import mpi.* ;


import java.awt.* ;

import javax.swing.* ;


public class MPJSquaresMandelbrot {


    final static int N = 1024 ;

    final static int CUTOFF = 100000 ;


    final static int BLOCK_EDGE = 64 ;   // side of square block (should divide N).


    final static int BLOCK_SIZE = BLOCK_EDGE * BLOCK_EDGE ;


    final static int NUM_BLOCKS    = (N * N) / BLOCK_SIZE ;

    final static int BUFFER_SIZE = 2 + BLOCK_SIZE * N ;


    // tag values

    final static int TAG_HELLO      = 0 ;

    final static int TAG_RESULT    = 1 ;

    final static int TAG_TASK        = 2 ;

    final static int TAG_GOODBYE = 3 ;


    static int [] [] set ;
```

```
public static void main(String [] args) throws Exception {

    MPI.Init(args) ;

    int me = MPI.COMM_WORLD.Rank() ;
    int P = MPI.COMM_WORLD.Size() ;

    int numWorkers = P - 1 ;

    int [] buffer = new int [BUFFER_SIZE] ;

    if(me == 0) {    // master process - sends out work and displays results

        set = new int [N] [N] ;
        Display display = new Display() ;

        for(int i = 0 ; i < N ; i++) {
            for(int j = 0 ; j < N ; j++) {
                set [i] [j] = -1 ;
            }
        }
        display.repaint() ;

        // Calculate set

        long startTime = System.currentTimeMillis();
```

```
            int nextBlockStart_i = 0 ;

            int nextBlockStart_j = 0 ;

            int numHellos = 0 ;

            int numBlocksReceived = 0 ;


            while(numBlocksReceived < NUM_BLOCKS || numHellos < numWorkers)
{

                // Receive hello or results from any worker

                Status status =
                        MPI.COMM_WORLD.Recv(buffer, 0, BUFFER_SIZE, MPI.INT,
                                            MPI.ANY_SOURCE,
MPI.ANY_TAG) ;

                if(status.tag == TAG_RESULT) {

                    // Save returned results to `set' and display

                    int resultBlockStart_i = buffer [0] ;

                    int resultBlockStart_j = buffer [1] ;

                    for(int i = 0 ; i < BLOCK_EDGE ; i++) {

                        for(int j = 0 ; j < BLOCK_EDGE ; j++) {

                            set [resultBlockStart_i + i] [resultBlockStart_j + j] =
buffer [2 + BLOCK_EDGE * i + j] ;

                        }

                    }

                    numBlocksReceived++ ;
                    display.repaint() ;
                }
                else {    // tag is TAG_HELLO
                    numHellos++ ;
                }

                // Send next block of work or finish tag to same worker
                if(nextBlockStart_j < N) {
                    buffer [0] = nextBlockStart_i ;
                    buffer [1] = nextBlockStart_j ;
                    MPI.COMM_WORLD.Send(buffer, 0, 2, MPI.INT, status.source,
TAG_TASK) ;
                    nextBlockStart_i += BLOCK_EDGE ;
                    if(nextBlockStart_i >= N) {
                        nextBlockStart_i = 0 ;
                        nextBlockStart_j += BLOCK_EDGE ;
                    }
                    System.out.println("Sending work to " + status.source) ;
                }
                else {
                    MPI.COMM_WORLD.Send(buffer, 0, 0, MPI.INT, status.source,
TAG_GOODBYE) ;
                    System.out.println("Shutting down " + status.source) ;
                }
            }

            long endTime = System.currentTimeMillis();
            System.out.println("Calculation completed in " +
                                   (endTime - startTime) + " milliseconds");
        }
        else {   // worker process

            // Send request to master for a first block of work

            MPI.COMM_WORLD.Send(buffer, 0, 0, MPI.INT, 0, TAG_HELLO) ;

            boolean done = false ;
            while(!done) {
                Status status = MPI.COMM_WORLD.Recv(buffer, 0, 2, MPI.INT, 0,
MPI.ANY_TAG) ;

                if(status.tag == TAG_TASK) {
                    int blockStart_i = buffer [0] ;
                    int blockStart_j = buffer [1] ;

                    for(int i = 0 ; i < BLOCK_EDGE ; i++) {
                        for(int j = 0 ; j < BLOCK_EDGE ; j++) {

                            double cr = (4.0 * (blockStart_i + i) - 2 * N) / N ;
                            double ci = (4.0 * (blockStart_j + j) - 2 * N) / N ;

                            double zr = cr, zi = ci ;
                            int k = 0 ;
                            while (k < CUTOFF && zr * zr + zi * zi < 4.0) {

                                // z = c + z * z

                                double newr = cr + zr * zr - zi * zi ;
                                double newi = ci + 2 * zr * zi ;

                                zr = newr ;
                                zi = newi ;

                                k++ ;
                            }

                            buffer [2 + BLOCK_EDGE * i + j] = k ;
                        }
                    }
                    buffer [0] = blockStart_i ;
                    buffer [1] = blockStart_j ;
                    MPI.COMM_WORLD.Send(buffer, 0, BUFFER_SIZE, MPI.INT, 0,
TAG_RESULT) ;
                }
                else {    // tag is TAG_GOODBYE
                    done = true ;
                }
            }
        }
```

```
        MPI.Finalize() ;
    }

    static class Display extends JPanel {

        Display() {

            setPreferredSize(new Dimension(N, N)) ;

            JFrame frame = new JFrame("Laplace");
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setContentPane(this);
            frame.pack();
            frame.setVisible(true);
        }

        public void paintComponent(Graphics g) {
            for(int i = 0 ; i < N ; i++) {
                for(int j = 0 ; j < N ; j++) {
                    int k = set [i] [j] ;
                    Color c ;

                    if(k == -1) {    // uninitialized
                        c = Color.WHITE ;
                    }
                    else {
                        float level ;
                        if(k < CUTOFF) {
                            level = k < 50 ? (float) k / 50 : 1.0F ;
                        }
                        else {
                            level = 0 ;
                        }
                        c = new Color(level/2, level/2, level) ;    // Blueish
                    }

                    g.setColor(c) ;
                    g.fillRect(i, j, 1, 1) ;
                }
            }
        }
    }
}
```