

HPC Fall 2010 – Final Project 3

2D Steady-State Heat Distribution with MPI

Robert van Engelen

Due date: December 10, 2010

1 Introduction

1.1 HPC Account Setup and Login Procedure

Same as in Project 1.

1.2 Download the Project Files

Next, download the project source:

```
[yourname@submit ~]$ wget http://www.cs.fsu.edu/~engelen/courses/HPC/Pr3.zip
```

The package bundles the following files:

- `build.sh`: wrapper to build `heatdista` and `heatdistb`
- `Makefile`: a Makefile to build the project files via `build.sh`
- `heatdista.c`: steady-state heat distribution with asynchronous non-blocking MPI calls
- `heatdistb.c`: steady-state heat distribution with blocking MPI calls
- `jumpshot.sh`: starts jumpshot MPE log browser
- `runa.sh`: wrapper to run `heatdista`
- `runb.sh`: wrapper to run `heatdistb`

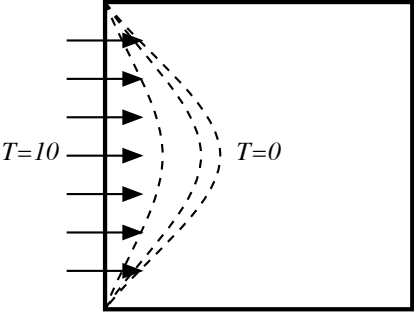
For this project you need to write a report. In your report you should answer the questions and show the code changes you made for this project.

1.3 Overview

The steady-state heat distribution problem consists of an $n \times n$ discrete grid with temperature field h . Given boundary conditions $T = 10$, we solve h iteratively starting with $h = 0$ for all interior points and then improve the solution using Jacobi iterations:

$$h_{i,j}^{k+1} = \frac{1}{4}(h_{i-1,j}^k + h_{i+1,j}^k + h_{i,j-1}^k + h_{i,j+1}^k) \tag{1}$$

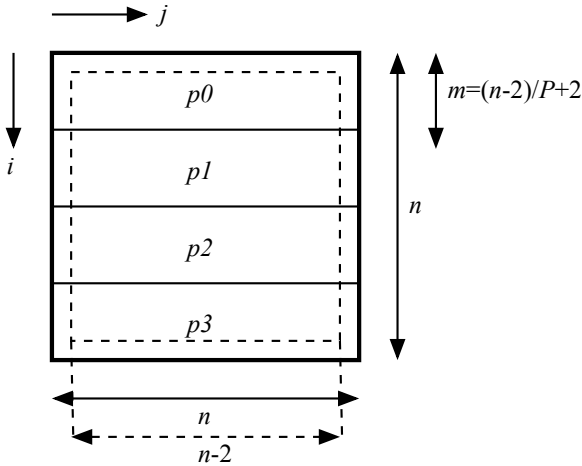
with boundary condition $T = 10$ on the leftmost boundary with $j = 0$. Thus, the heat will radiate to the interior of the box, approximately as illustrated here:



By repeating the Jacobi iterations Eq. (1) for time steps $k = 0, \dots$ until convergence we obtain the steady-state solution.

We can apply domain decomposition to parallelize the Jacobi iterations. Each iteration consists of two phases: 1) a message passing exchange to update the halos (aka. ghost cells) and 2) a local computation to update the grid.

We partition the domain in row-wise in blocks as follows (assuming $P=4$ processors):



In this project we use a $n \times n$ grid with $n = 82$, where the boundary values are positioned along the edges of the grid ($i = 0, i = n - 1, j = 0$, and $j = n - 1$), thus the interior region consists of 80×80 points and boundary values are stored at the edges (obviously these boundary values are not updated in each Jacobi iteration).

Each processor has $m = \frac{n-2}{P} + 2$ grid points, with interior region $(m - 2) \times (m - 2)$. The edges are either boundaries with the fixed boundary values, for $j = 0$ and $j = n - 1$, or halos (ghost cells) for the local rows $i = 0$ and $i = m - 1$. These ghost cells are updated in each iteration via nearest-neighbor communications, except for processor $p = 0$ (top) and $p = P - 1$ (bottom).

On each processor the interior points of field h^k at iteration k are updated as follows:

```
for (i = 1; i < m-1; i++)
  for (j = 1; j < n-1; j++)
    hnew[n*i+j] = 0.25*(hp[n*(i-1)+j]+hp[n*(i+1)+j]+hp[n*i+j-1]+hp[n*i+j+1]);
```

Note that the grid is mapped to one-dimensional arrays `hp` and `hnew` using a row-major layout, where the $h_{i,j}$ point corresponds to element `hp[n*i+j]`. This makes it easy to communicate ghost cell rows, because of the row-wise layout.

The above code assumes that the values of `hp` at ghost cell rows $i = 0$ and $i = m - 1$ were copied from the `hp` values from the processors above and below, respectively. Note that the values of `hp` at $j = 0$ and $j = n - 1$ are fixed boundary values.

2 MPI Implementation

To get started with this assignment, let's try out the package content:

- Open a terminal and `ssh -Y submit.hpc.fsu.edu` to log into the HPC submit node (SC student use `sc.hpc.fsu.edu`)
- Run `./build.sh`
- Start an interactive job for 4 cores on a compute node with `msub -I -l nodes=1:ppn=4 -q sc_classroom`
- Run `./runb.sh` on the compute node

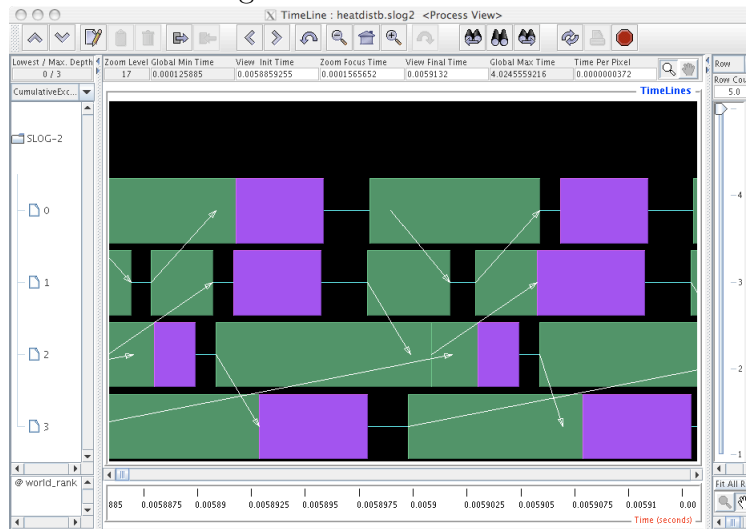
The run produces output that shows the upper region of the temperature field residing on processor $p = 0$.

IMPORTANT: As of this time of writing, the `sc_classroom` compute node `hpc-5-2` does not have an IB card installed so the MPI runs fail with a bunch of errors, which should be fixed soon. If the `hpc-5-2` node still does not work via the `sc_classroom` queue, you should use the `backfill` queue instead. Just use `msub -I -l nodes=1:ppn=4 -q backfill`.

Because we use MPE, the run also produces a log file `heatdistb.clog2` with MPI statistics. Inspect the log as follows:

- Go back to the submit node and run `./jumpshot.sh`

- Open `heatdistb.clog2` by selecting the "Select a new log file" button (lower left corner) and double-clicking the `heatdistb.clog2` file in the file list.
- Answer YES to convert the CLOG-2 file to SLOG-2 format.
- Select the CONVERT button.
- When converted select the OK button.
- View the timeline by zooming in by clicking the (+) button repeatedly. Alternatively, you can also hold and drag the cursor horizontally to select a region to zoom in. You will need to go to a 0.00015 zoom focus time to see the MPI operations more clearly. The view should be something like:



Each horizontal line represents the activity of a processor. The purple rectangles represent computation time. The dark green rectangles represent send-recv time. Arcs denote message exchanges. From the graph we can see that the send/rcv calls are blocking and take a long time compared to computation. There is no computation-communication overlap.

A Jacobi iteration is implemented using MPI blocking send-recv calls to exchange the ghost cells as follows:

```

if (p > 0 && p < P-1)
{
    MPI_Sendrecv(&hp[n*(m-2)], n, MPI_FLOAT, p+1, 0,
                &hp[n*0], n, MPI_FLOAT, p-1, 0, MPI_COMM_WORLD, &status);
    MPI_Sendrecv(&hp[n*1], n, MPI_FLOAT, p-1, 1,
                &hp[n*(m-1)], n, MPI_FLOAT, p+1, 1, MPI_COMM_WORLD, &status);
}
else if (p == 0 && p < P-1)
    MPI_Sendrecv(&hp[n*(m-2)], n, MPI_FLOAT, p+1, 0,

```

```

        &hp[n*(m-1)], n, MPI_FLOAT, p+1, 1, MPI_COMM_WORLD, &status);
else if (p > 0 && p == P-1)
    MPI_Sendrecv(&hp[n*1],      n, MPI_FLOAT, p-1, 1,
                &hp[n*0],      n, MPI_FLOAT, p-1, 0, MPI_COMM_WORLD, &status);

/* Start of computation */
MPE_Log_event(event1a, 0, NULL);

for (i = 1; i < m-1; i++)
    for (j = 1; j < n-1; j++)
        hnew[n*i+j] = 0.25f*(hp[n*(i-1)+j]+hp[n*(i+1)+j]+hp[n*i+j-1]+hp[n*i+j+1]);

/* End of computation */
MPE_Log_event(event1b, 0, NULL);

/* Check convergence every 100 iterations (Pacheco) */
/* NOT IMPLEMENTED: set stop = 1 to terminate */

/* Copy new */
for (i = 1; i < m-1; i++)
    for (j = 1; j < n-1; j++)
        hp[n*i+j] = hnew[n*i+j];

```

The MPI calls are automatically logged by MPE. To log the computation, we inserted the `MPE_Log_event` calls as shown above.

2.1 Improved Printing of the Grid

Only the top part of the grid for $p = 0$ is dumped. To dump the entire grid, we need each processor to print the temperature field on its local grid in turn (without the ghost cells).

To do so, you need to use send-recv message passing calls between processors to inform each other in turn that the other finished printing. So, after printing the top part by processor $p = 0$, it sends a message to $p = 1$ so that this next processor can print its part, and so on.

At the end of each print dump done by a processor, use `flush(stdout)`; and `sleep(1)`; to allow the screen to be updated before the next processor dumps the grid.

Remember to edit and build your code on the submit node and run the code interactively with 4 cores on the hpc-x-y node using "msub" to start an interactive session on the classroom sub-cluster:

```
msub -I -l nodes=1:ppn=4 -q sc_classroom
```

then

```
./runb.sh
```

2.2 Adding a Pacheco Stopping Test

As you can see from the dumps, the temperature field does not look right. It is not updated to the steady-state solution we might expect. We can increase the iterations by increasing `MAXSTEP` to improve the solution. However, better is to add a stopping test.

We add a Pacheco termination test to the code, where the iterations should stop when

$$\sum_{i=1}^{n-2} \sum_{j=1}^{n-2} (h_{i,j}^t - h_{i,j}^{t+1})^2 \leq \epsilon^2$$

where $\epsilon = 0.1$ is given by `TOLERANCE=0.1` in the code. To compute the termination test efficiently in parallel, you need to compute the local errors

$$e_p = \sum_{i=1}^{m-2} \sum_{j=1}^{n-2} (\text{hp}_{i,j} - \text{hnew}_{i,j})^2$$

and then check that $\sum_{p=0}^{P-1} e_p \leq \epsilon^2$. Only when the sum of the local e_p is less than ϵ^2 we should stop! Convergence testing is global.

Because testing is global, each processor should also “know” that the termination criterion is met, since they all should terminate in the same iteration. Figure out how to share the termination test results among all processors.

Implement the Pacheco test. To limit the overhead of testing, the test should only be performed every 100 iterations.

Note: it will take more than 1000 but less than 5000 iterations to obtain a good approximation of the steady-state solution. Thus, set `MAXSTEP` to 5000.

Experiment with your stopping test using 1, 2, and 4 cores on a single node, by using the “`msub`” command from the submit node to reserve 4 cores:

```
msub -I -l nodes=1:ppn=4 -q sc_classroom
```

then edit the `runb.sh` file to specify the number of processes with option `-np`:

```
mpirun -np 4 $PWD/heatdistb
```

Now (re)run with `./runb.sh` on the `hpc-x-y` node. Repeat if necessary to verify the stability of the timings.

For each run, write down the total parallel execution time reported by the program and number of iterations it took to converge (can they or should they be different at all?). Calculate the relative speedups and parallel efficiency and add them to your table. Your table should have three rows, one for each number of processors 1, 2, and 4 that you tested.

Also for each run of 1, 2, and 4 cores, open `jumpshot` to inspect the execution timings in the Jacobi iterations. Estimate the ratio of t_{comp} to t_{comm} by comparing the timelines of the send-recv and computation. Write the ratio in your table. How does this ratio relate to the parallel efficiency?

2.3 MPI Asynchronous Non-blocking Implementation

The disadvantage of the previous implementation is that communication and computation are not overlapped. By overlapping we can hide communication latencies and hopefully improve the performance of an MPI application. Whether we obtain a speedup depends on the efficiency of the non-blocking implementation in the MPI library. Performance is also affected by the logging with MPE, since we need more MPI calls to use asynchronous non-blocking operations, thereby triggering more logging events! However, it is generally good practice to use non-blocking calls to implement communication/computation overlap as long as the cost of neither one increases (e.g. by changing the algorithms).

To implement the communication/computation overlap, we will use the `MPI_Isend` and `MPI_Irecv` calls. This is accomplished by updating the ghost cells while the “inner” interior rows are computed where the inner interior rows are $i = 2, \dots, n - 2$. The outer interior rows are then updated when the ghost cell values arrive:

```
if (p < P-1)
{
    MPI_Isend(&hp[n*(m-2)], n, MPI_DOUBLE, p+1, dntag, MPI_COMM_WORLD, &sndreq[0]);
    MPI_Irecv(&hp[n*(m-1)], n, MPI_DOUBLE, p+1, uptag, MPI_COMM_WORLD, &rcvreq[0]);
}
if (p > 0)
{
    MPI_Isend(&hp[n*1], n, MPI_DOUBLE, p-1, uptag, MPI_COMM_WORLD, &sndreq[1]);
    MPI_Irecv(&hp[n*0], n, MPI_DOUBLE, p-1, dntag, MPI_COMM_WORLD, &rcvreq[1]);
}
/* Compute inner interior rows */
for (i = 2; i < m-2; i++)
    for (j = 1; j < n-1; j++)
        hnew[n*i+j] = 0.25*(hp[n*(i-1)+j]+hp[n*(i+1)+j]+hp[n*i+j-1]+hp[n*i+j+1]);
/* Wait on receives */
if (P > 1)
{
    if (p == 0)
        MPI_Wait(&rcvreq[0], stat);
    else if (p == P-1)
        MPI_Wait(&rcvreq[1], stat);
    else
        MPI_Waitall(2, rcvreq, stat);
}
/* Compute outer interior rows */
for (j = 1; j < n-1; j++)
{
    hnew[n*1+j] = 0.25*(hp[n*0+j] +hp[n*2+j] +hp[n*1+j-1] +hp[n*1+j+1]);
    hnew[n*(m-2)+j] = 0.25*(hp[n*(m-3)+j]+hp[n*(m-1)+j]+hp[n*(m-2)+j-1]+hp[n*(m-2)+j+1]);
}
/* Wait on sends to release buffer */
if (P > 1)
{
```

```

    if (p == 0)
        MPI_Wait(&sndreq[0], stat);
    else if (p == P-1)
        MPI_Wait(&sndreq[1], stat);
    else
        MPI_Waitall(2, sndreq, stat);
}
/* Check convergence every 100 iterations (Pacheco) */
/* NOT IMPLEMENTED: set stop = 1 to terminate */
/* Copy new */
for (i = 1; i < m-1; i++)
    for (j = 1; j < n-1; j++)
        hp[n*i+j] = hnew[n*i+j];

```

See also the lecture notes and the code `heatdista.c` for more details.

Study the code. Explain why we need to wait for the `MPI_Irecv` and `MPI_Isend` to complete at the two specific points in the code.

Add the same Pacheco test stopping code in `heatdista.c` as you did for `heatdistb.c`.

Experiment with the asynchronous non-blocking `heatdista` code using 1, 2, and 4 cores on a single node, by using the "msub" command from the submit node to reserve 4 cores:

```
msub -I -l nodes=1:ppn=4 -q sc_classroom
```

Edit the `runa.sh` file to specify the number of processes with option `-np`:

```
mpirun -np 4 $PWD/heatdista
```

Now (re)run with `./runa.sh` on the `hpc-x-y` node. Repeat if necessary to verify the stability of the timings.

For each run, write down the total parallel execution time reported by the program and number of iterations it took to converge (can they or should they be different at all?). Calculate the relative speedups and parallel efficiency and add them to your table. Your table should have three rows, one for each number of processors 1, 2, and 4 that you tested.

Also for each run of 1, 2, and 4 cores, open jumpshot to inspect the execution timings in the Jacobi iterations. Estimate the ratio of t_{comp} versus everything else that MPI does (all other blocks that are NOT nested inside the purple Computation block). Note that t_{comp} now consist of two parts in each iteration. Write the ratio in your table. How does this ratio relate to the parallel efficiency? Note: to improve the readability of the Jumpshot view, deselect the `MPI_Wait` and `MPI_Waitall` and refresh the view.

2.4 Gauss-Seidel Relaxation

Use Gauss-Seidel relaxation by updating `hp` instead of `hnew`. You cannot use `MPI_Isend` reliably any longer, since the data will be changed while the send is in progress. Use a

local-blocking send instead but keep the non-blocking `MPI_Irecv`.

Put your changes in a new file `heatdistc.c`. Also change `Makefile` by adding a build step for `heatdistc.c` and create a `runc.sh` file to run MPI for `heatdistc`.

Experiment with `heatdistc` using 1, 2, and 4 cores on a single node, by using the "msub" command from the submit node to reserve 4 cores:

```
msub -I -l nodes=1:ppn=4 -q sc_classroom
```

Edit the `runc.sh` file to specify the number of processes with option `-np`:

```
mpirun -np 4 $PWD/heatdistc
```

Now (re)run with `./runc.sh` on the `hpc-x-y` node. Repeat if necessary to verify the stability of the timings.

For each run, write down the total parallel execution time reported by the program and number of iterations it took to converge (can they or should they be different at all?). Calculate the relative speedups and parallel efficiency and add them to your table. Your table should have three rows, one for each number of processors 1, 2, and 4 that you tested.

Also for each run of 1, 2, and 4 cores, open jumpshot to inspect the execution timings in the relaxed Jacobi iterations. Estimate the ratio of t_{comp} versus everything else that MPI does (all other blocks that are NOT nested inside the purple Computation block). How does this ratio relate to the parallel efficiency?

End.