

HPC-II Spring 2009 – Project 1

Checkpointing with Parallel I/O

Robert van Engelen

Due date: March 5, 2009

1 Introduction

1.1 HPC Account and Login Information

For this assignment you need an SCS account. The account gives you access to `pamd`. You also need an account on the FSU HPC cluster to run MPI jobs. Contact the instructor if you do not have an SCS account.

To obtain an account on the FSU HPC cluster, do the following:

- If you are an SCS student, then:
 1. Use your favorite web browser to go to <http://www.hpc.fsu.edu/>
 2. Click "Connecting" under "Getting Started" in the left hand menu.
 3. Click "HPC Account Application Details"
 4. Click "Apply" for a owner-based access account
 5. Select "Non-Faculty Account"
 6. Fill in the form:
 - (a) Enter you FSUID
 - (b) Select your unit: SCS
 - (c) Fill in all the rest of the personal information
 7. Click "Request Account"

After receiving your account creation acknowledgment, you can login to the head node from any machine:

```
ssh -Y scs.hpc.fsu.edu
```

You should edit and save your files on this head node. We will also build the project binaries and use "jumpshot" on the head node.

You should run the project binaries as interactive jobs on a single node with four cores. Use the MOAB "msub" command to start an interactive job to the classroom queue:

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

You can then run the MPI binaries interactively. How this is done is explained later.

- If you are not an SCS student, then:

1. Use your favorite web browser to go to <http://www.hpc.fsu.edu/>
2. Click "Connecting" under "Getting Started" in the left hand menu.
3. Click "HPC Account Application Details"
4. Click "Apply" for an general access account
5. Select "Sponsored Account"
6. Fill in the form:
 - (a) Enter your FSUID
 - (b) Choose your faculty sponsor: van Engelen
 - (c) Fill in all the rest of the personal information
7. Click "Request Account"

After receiving your account creation acknowledgment, you can login to the head node from any machine:

```
ssh -Y submit.hpc.fsu.edu
```

You should edit and save your files on this head node. We will also build the project binaries and use "jumpshot" on the head node.

You should run the project binaries as interactive jobs on a single node with four cores. Use the MOAB "msub" command to start an interactive job to the classroom queue:

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

You can then run the MPI binaries interactively. How this is done is explained later.

From the head node you can also check the availability of the scs_classroom queue before starting a job:

```
showq -w class=scs_classroom
```

Then use `msub` to start an interactive job to log on to the execute node via the `scs_queue`.

1.2 Setup

In your HPC account create a directory named "Pr1" and copy the files from:

`http://www.cs.fsu.edu/~engelen/courses/HPC-adv/Pr1.zip`

To do so, use `wget http://www.cs.fsu.edu/%7eengelen/courses/HPC-adv/Pr1.zip` from the head node.

The package bundles the following files:

- `build.sh`: wrapper to build `heatdist`
- `Makefile`: a Makefile to build the project files via `build.sh`
- `heatdist.c`: steady-state heat distribution
- `run.sh`: wrapper to run `heatdist`

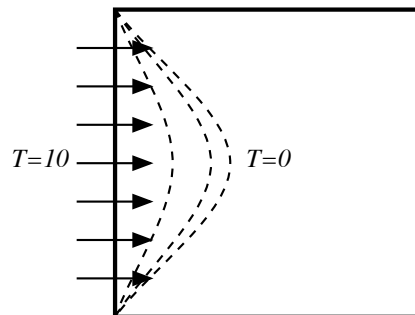
For this project you need to write a report. In your report you should answer the questions and show the code changes you made for this project

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:

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

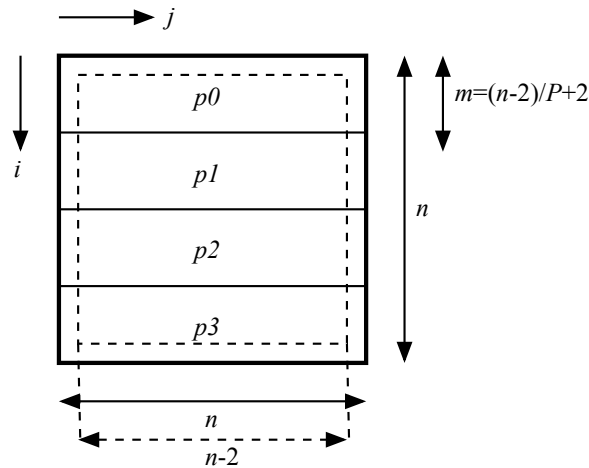
with boundary condition $T = 10$ on the leftmost boundary with $j = 0$. Thus, the heat will radiate to the interior of the box, approximately as illustrated here:



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, 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 $(n - 2) \times (n - 2)$ points and boundary values are stored at the edges that are one point thick (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 $(m - 2) \times (n - 2)$ (rows by columns). 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

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

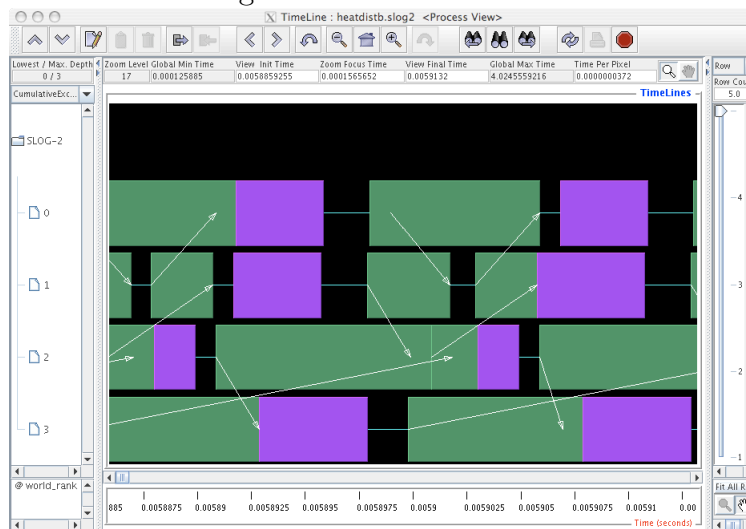
- Open a terminal and `ssh -Y` to the HPC head node

- Run `./build.sh`
- Start an interactive job for 4 cores with `msub -I -l nodes=1:ppn=4 -q scs_classroom`
- Run `./run.sh`

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

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

- On the head node run `./jumpshot.sh`
- Open `heatdist.clog2` by selecting the "Select a new log file" button (lower left corner) and double-clicking the `heatdist.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/recv 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);

/* 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 Checkpointing

In this project we add parallel file I/O to save and retrieve the `hp` temperature field to/from a file using the various MPI parallel I/O mechanisms.

First you should add two command-line options to the program. The first specifies the number n of Jacobi iterations. The second option is a file name. Use `argc` and `argv` to read these options. For example, running

```
heatdist 1000 myfile
```

reads file `myfile` to obtain the initial data for `hp`, executes $n = 1000$ Jacobi iterations, and saves the updated `hp` data to `myfile`.

When the `heatdist` program runs without the file name option, then the initial `hp` field is set as in the current program, n Jacobi iterations should be performed, and the data (with the boundaries) should be saved to file `heatdata`. When the file name option is given, the `heatdist` program should read the file to obtain the initial data, execute n Jacobi iterations, and should then save the results back to the file.

Note that In a real checkpointing scenario, a simulation program continues to run and save the data and program state to a file. In this way, the simulation program can be restarted after failure. In our experiment we only read/write once to file.

The grid size of your experiment should be 1000×1000 interior points:

```
#define MAXSIZE (1000+2)
```

You need to set this macro in your code. This generates more realistic data to experiment with parallel file I/O. For debugging you may want to start with a smaller grid with `MAXSIZE = (80+2)`, which allows for printing of the top part of the solution to screen.

For this I/O experiment, we can pick any number of Jacobi iterations since we don't really care what the solution is. We only care about saving and retrieving data and doing this efficiently. A realistic number of iterations for a 1000×1000 grid exceeds $n \geq 5000$.

The file is written row-major from the top-left to bottom-right row-by-row, *excluding the ghost cells* and *excluding the top and bottom boundaries*. That is, each processor writes a block of $m - 2$ rows and n columns to the file for its local region that excludes the ghost cells and top/bottom boundary cells. The entire region written is $n - 2$ rows by n columns in blocks of $m - 2$ rows by n columns. We simplified our task by saving the left and right boundaries (n columns including the left and right boundary). In this way, we can still read to data back for a different number of processors, since we got rid of the ghost cells with duplicate values.

Experiment with MPI parallel I/O by writing three versions of the `heatdist` code using level 0, level 1, and level 3 MPI I/O access:

- Level 0: each processor uses a single `MPI_File_seek` and `MPI_File_read` (or `write`) call to position the cursor and read/write the data from/to file. Then use MPE and jumpshot to determine the timings of the read file access and write file access. Print a screen dumps of the jumpshot graph for you report.
- Level 1: same as level 0, but now use `MPI_File_seek`, `MPI_File_read_all` (and `write_all`) to read/write collectively. Determine the timings from the jumpshot graphs.
- Level 3: use `MPI_Type_create_subarray` and `MPI_File_set_view` to create a file view. Note that we have a blocked distribution over rows (vertical) and collapsed (degenerate) over columns. Use `MPI_File_read_all` (and `write_all`) to read/write collectively a portion of the file. Determine the timings from the jumpshot graphs. You need to

figure out how to set up 2D arrays for `gsizes`, `lsizes`, and `starts` to initialize the subarray:

```
MPI_Type_create_subarray(2, gsizes, lsizes, starts, MPI_ORDER_C,  
                        MPI_FLOAT, &filetype);  
MPI_Type_commit(&filetype);
```

and then create a view:

```
MPI_File_set_view(fh, 0, MPI_FLOAT, filetype, "native", MPI_INFO_NULL);
```

and open the file to read/write. You need to figure out how to invoke the `MPI_File_read_all` (and `write_all`) with the correct starting point offset into the local `hp` array to read/write to avoid the top/bottom ghost cells. Note: there is no need to create a Cartesian processor grid as was shown on the lecture slides 33-34. Just create the subarray, fileview, and perform read/write.

Show all three code solutions for parallel I/O in your report with the timing results and jumpshot graphs, each for the I/O write and read experiment with levels 0, 1, and 3.

To find more information on the MPI calls, visit:

<http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html>

for MPI parallel I/O:

<http://www.mpi-forum.org/docs/mpi-20-html/node171.htm#Node171>

and for MPI subarray type specifically:

<http://www.mpi-forum.org/docs/mpi-20-html/node221.htm#Node221>

End.