

HPC Fall 2007 – Project 3

2D Steady-State Heat Distribution Problem with MPI

Robert van Engelen

Due date: December 14, 2007

1 Introduction

1.1 Account and Login Information

For this assignment you need an SCS account. The account gives you access to `pamd` and the `phoenix` cluster. Contact the instructor if you do not have an SCS account. To login to `phoenix` from outside SCS you must first `ssh` to `pamd` (using `ssh name@pamd.scs.fsu.edu`) and then `ssh` to `phoenix`. Use `ssh -Y` to enable X11 forwarding.

1.2 Setup

Note that when you login to `phoenix` your home directory is mounted as `fshome`. If you don't have a `.cshrc` file in `phoenix` home, then copy your `fshome/.cshrc` to your `phoenix` home.

In your `.cshrc` on `phoenix` add the following lines:

```
# OpenMPI with GCC, use missing man pages from MPICH
setenv OPENMPIHOME /opt/openmpi
setenv MANPATH ${MANPATH}:${OPENMPIHOME}/man:/usr/local/mpich2/1.0.4p1/man
setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:/usr/local/openmpi/gcc/lib:/opt/openmpi/gcc/lib
set path = ( $OPENMPIHOME/bin /opt/openmpi/gcc/bin $path )
```

1.3 Download

Next, download the project source code from

<http://www.cs.fsu.edu/~engelen/courses/HPC/Pr3.zip>

The package bundles the following files:

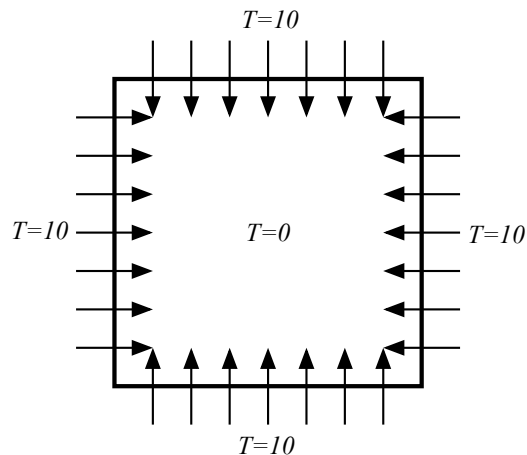
- `Makefile`: a standard Makefile to build the project.
- `heatdist.c`: steady-state heat distribution Jacobi iterations in MPI
- `run.sh`: wrapper to run `mpirun`
- `sge.sh`: wrapper to run SGE script
- `heatdist.sh`: script to run SGE job (used by `sge.sh`)

1.4 Overview

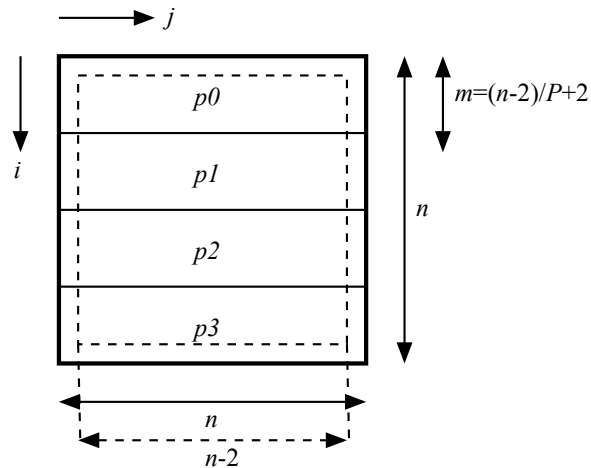
The steady-state heat distribution problem consists of an $n \times n$ discrete grid with temperature field h . Given boundary conditions $T = 10$, we solve h iteratively starting with $h = 0$ using the Jacobi iteration:

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

Starting with $h_{i,j}^t = 0$ for step $t = 0$, this (slowly) converges to the exact solution $h_{i,j}^t = T = 10$ with $t \rightarrow \infty$ for all points, given the boundary condition $T = 10$ surrounding the $n \times n$ square region as shown below:



The parallelization strategy consists of applying a partitioning strategy to decompose the domain row-wise in blocks:



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 along the edges (the 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 fixed boundary values, for $j = 0$ and $j = n - 1$, or “ghost cells” (also referred to as “halos”) for 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^t at iteration t 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.

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.

The implemented code overlaps computation with communication using 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);
}
/* 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 `heatdist.c` for more details.

1.5 Getting Started

To get started with this assignment and try out the package content, follow these steps:

- Open a terminal and `ssh -Y` to `phoenix` (via `pamd.scs.fsu.edu`)
- Run `make` to build `heatdist`
- Run `sh run.sh` which solves the steady-state heat distribution problem using 4 tasks on the single `phoenix` head node with a 82×82 grid using Jacobi iterations.

The run produces output that shows the upper region of the temperature field residing on processor `p0` coded as 0-9 values. Note that the the value 9 represents the solution $T = 10$.

The run also produces a log file `heatdist.clog2` with MPI statistics. You can inspect the log file as follows:

- Start `jumpshot heatdist.clog2` in an xterm.
- Answer YES to convert the CLOG-2 file to SLOG-2 format.
- Select CONVERT button.
- After conversion select the OK button.
- View the timeline by zooming in and out with the cursor. Since there are 5000 iterations that can take up to two seconds to compute, you need to go to a 0.1 second scale to view the actual MPI operations and messages. The blue rectangles represent computation (with communication overlap that requires an `MPI_Wait` and `MPI_Waitall` to receive the ghost cell values needed to compute the outer interior points). Always use `make clean` to cleanup the log files.

2 Your Assignment

1. 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.
2. Run the code with $P = 1$, $P = 2$, $P = 4$, and $P = 8$ on the `phoenix` head node using script `run.sh` (edit the script to change the process count). The `phoenix` head node is a 2 dual-core Xeon processor. For each run with $P = 1$, $P = 2$, $P = 4$, and $P = 8$ write down the running time (as displayed by the program) and determine from a couple of snapshots of the jumpshot profile the approximate average computation and

communication ratio (assume that anything that is not computation is communication overhead). Note that the blue rectangle without any inner nested rectangles represents pure computation (study the MPE event calls in the code!).

3. The current code runs 5000 iterations, which is more than enough to converge. Add the Pacheco termination criterion 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 criterion 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 $\sum_{p=0}^{P-1} e_p \leq \epsilon^2$. Each processor should “know” that the termination criterion is met, since they all should terminate in the same iteration t . Run the code with $P = 1$, $P = 2$, $P = 4$, and $P = 8$ on the `phoenix` head node with `run.sh` and compare the total running times to the running times you recorded in the previous question. Note that the tolerance allows the innermost region of the solution to be smaller than $T = 10$. Hint: you need a global reduction and broadcast, or a combination of the two.

4. Reduce the overhead of the termination criterion check by checking for convergence periodically once in 100 iterations. Compare the running times to the running times you recorded in the previous question. Explain why periodic termination checking might be good enough.
5. In this part we will run the updated code (with periodic termination check) with $P = 4$ and $P = 8$ on the `phoenix` cluster using script `sge.sh`. Edit `heatdist.sh` to adjust the number of processors. Before you can run the code, copy `sge.sh`, `heatdist.sh`, and `heatdist` binary to the `phoenix` home dir to run `sge.sh`. Use `qstat` to check the job status. Use `qdel` with the job id to remove the job when you notice a problem. When all goes well the output is saved in `heatdist.sh.oX` with X the job id. The logs are saved in `heatdist.clog2`. Write down the total running time and determine from a couple of jumpshot profile snapshots the approximate average computation and communication ratio (anything that is not computation is communication overhead).
6. 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`. Determine for $P = 1$, $P = 2$, $P = 4$, and $P = 8$ on the `phoenix` head node (`run.sh`) how many iterations it takes to converge using this relaxation scheme.

End.