HPC Spring 2017 – Final Project 3
2D Steady-State Heat Distribution with MPI

Robert van Engelen

Due date: May 5, 2017

1 Introduction

In this project you will learn how to run, analyze and improve a parallel 2D steady-state problem implemented in MPI.

1.1 Download the Project Files

Open a terminal and execute `ssh -Y youracct@hpc-login.rcc.fsu.edu` to log into the HPC. Next, download the project source:

```
[youracct@hpc-login-35 ~]$ wget http://www.cs.fsu.edu/~engelen/courses/HPC/Pr3.zip
```

The package bundles the following files:

- `build.sh`: shortcut to load modules and build `heatdista` and `heatdistb`
- `Makefile`: a Makefile to build the project
- `heatdista.c`: steady-state heat distribution with asynchronous, non-blocking MPI
- `heatdistb.c`: steady-state heat distribution with blocking MPI
- `runa.sh`: sbatch script for job submission of `heatdista`
- `runb.sh`: sbatch script for job submission of `heatdistb`
- `runa-trace.sh`: sbatch script for job submission of `heatdista` tracing for jumpshot
- `runb-trace.sh`: sbatch script for job submission of `heatdistb` tracing for jumpshot
- `jumpshot.sh`: merges traces and starts jumpshot in a browser

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.

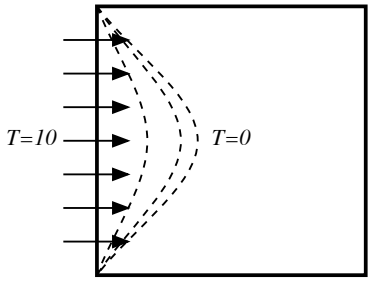
This project is directly related to the material introduced in class, see lecture notes *Message Passing Part III: Examples*.

1.2 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:

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

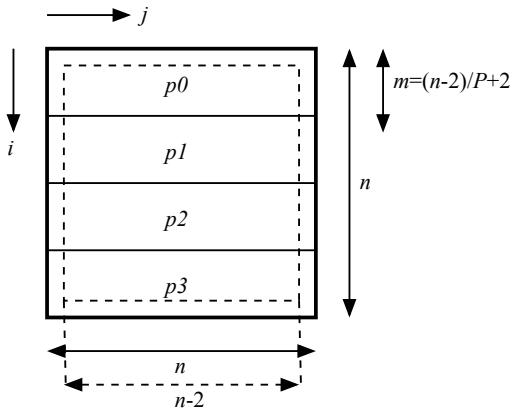
with boundary condition $T = 10$ on the leftmost boundary with $j = 0$. Thus, the heat will radiate to the interior of the box, as illustrated here:



By repeating the Jacobi iterations Eq. (1) for time steps $k = 0, \ldots$ 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 (ghost cells) and 2) a local computation to update the grid.

We partition the domain row-wise in blocks as shown here for $P = 4$ processors:



We use a $n \times n$ grid with $n = 242$ for this project, 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 240×240 points and boundary values are stored at the edges, where $T = 10$ on the left and $T = 0$ everywhere else. These boundary values are not updated in each Jacobi iteration.

Each processor has a local grid region of $m = \frac{n-2}{P} + 2$ rows by n columns, with an interior region of $(m-2) \times (n-2)$ grid points that excludes the edges. The edges are either boundaries with the fixed boundary values for T for $j = 0$ and $j = n - 1$, or halos (ghost cells) with rows $i = 0$ (top) and $i = m - 1$ (bottom). These ghost cells are updated in each iteration via nearest-neighbor communications, except for processor $p = 0$ (at the top) and $p = P - 1$ (at the bottom). See lecture notes *Message Passing Part III*.

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 The MPI Implementation

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

- Open a terminal and execute `ssh -Y youracct@hpc-login.rcc.fsu.edu` to log in.
- Execute:

  ```
  [youracct@hpc-login-35 ~]$ module load intel-openmpi
  [youracct@hpc-login-35 ~]$ module load tau
  [youracct@hpc-login-35 ~]$ make
  ```

  You **MUST load these two modules every time** you log in before compiling and submitting jobs.

- Submit a batch job to run heatdistb on 4 processors (`runb.sh` uses `srun -n 4`):

  ```
  [youracct@hpc-login-35 ~]$ sbatch runb.sh
  ```

When the job is finished, the output is saved to slurm-xx.out that shows the upper part of the temperature field radiating from the left edge of the grid. Enlarge your terminal window to at least 120 columns to view this file.

Four files `profile.x.0.0.0` are saved to your directory. To view the profile, start paraprof:

```
[youracct@hpc-login-35 ~]$ paraprof &
```

In the TAU ParaProf window click on "node 0" to view the profile of the job on node 0. The `update()` function updates the `h[]` with `hnew[]` and exchanges the ghost rows with `MPI_Sendrecv()`. These two operations perform the critical work. Because the number of iterations is limited, the amount of work is initially low compared to `MPI_Init` for example.

In the TAU ParaProf window select menu "Windows" then "Communication Matrix". This shows the number of messages sent and received (or select another metric if desired) between the 4 processors. For example, node 0 sends only to node 1 as we would expect. Each Jacobi iteration requires a send-recv exchange.

While the paraprof display per node gives us an idea of the exclusive execution times of each operation, it does not tell us what happens during message passing. To do this we need to create a trace.

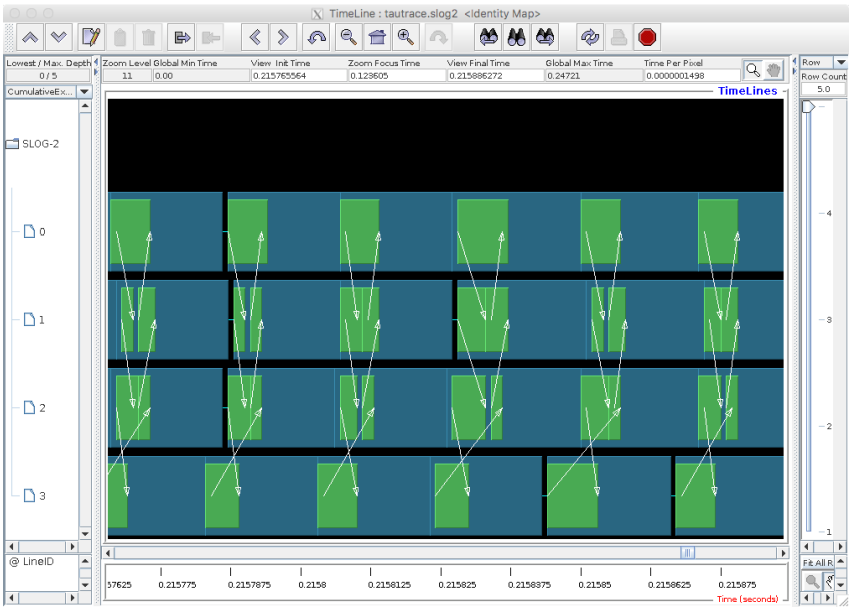
Exit paraprof, then execute:

```
[youracct@hpc-login-35 ~]$ make clean
[youracct@hpc-login-35 ~]$ sbatch runb-trace.sh
```

It is always a good idea to remove the old profiles and traces with `make clean`. When the job is finished this produces several .trc and .edf files that must be merged to create a trace that can be viewed with jumpshot. The `jumpshot.sh` script does this and also launches jumpshot:

```
[youracct@hpc-login-35 ~]$ ./jumpshot.sh
```

In the "Legend" window select the triangle next to "V" and "disable all". Then in the "V" column check `MPI_Sendrecv`, `update`, `message 0` and `message 1`. In the TimeLine window zoom in (by clicking the (+) button repeatedly) to the region on the right side of the timeline until you see the following details (warning: this X11 application will run very slow when run over the network from outside the FSU network):



Each horizontal bar represents the activity of one of the four processors. The dark green rectangles represent inclusive time spent in `update()` and the light green rectangles represent time spent in `MPI_Sendrecv()`.

We see that the `MPI_Sendrecv` calls take about less than half of the total inclusive `update()` time, so the t_{comp} to t_{comm} ratio is slightly greater than 1:1.

A Jacobi iteration is implemented using MPI blocking `MPI_Sendrecv` 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);

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]);

/* 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];
```

Before optimizing communications with asynchronous calls as `heatdista.c` shows, we should make sure to get all the results and add a iteration stopping test.

2.1 Task 1: Improved Printing of the Grid

When viewing the slurm-xx.out file you will notice that only the top part of the grid for $p = 0$ is dumped. To dump the entire grid, we need each processor to print the `hp[]` temperature field of its local internal region of the grid in turn (i.e. without the ghost cells).

To do so, you need to use a send and recv message to/from processors to inform each other in turn that the other finished printing. So, after printing the top part by processor $p = 0$, it must send a message to processor $p = 1$ so that $p = 1$ can print its part, after which it sends a message to processor $p = 2$, and so on. Add code so that all processors except $p = 0$ are waiting on a message before printing their part of the grid, then send a message to the next processor, except for processor $p = P - 1$ that has no successor.

At the end of each print of the `hp[]` local internal grid region to stdout by a process, use `fflush(stdout); usleep(100);`. This allows the output to be flushed from output buffers before the next processor dumps its part of the grid.

Include the code changes in your report.

2.2 Task 2: Adding a Pacheco Stopping Test

We can increase the iterations by increasing `MAXSTEP` to improve the iterative solution by reducing the approximation error. However, better is to add a stopping test.

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

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

where $\epsilon = 0.01$ is given by `TOLERANCE=0.01` in the code. To compute the termination test efficiently in parallel, you need to compute the local errors

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

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

Implement the Pacheco test and stopping criterion using an MPI collective `MPI_Allreduce` to compute the global sum and broadcast it so each processor can determine when to stop. To limit the overhead of this test, the test should only be performed every 100 iterations meaning you should test when `t%100 == 0` in `update()`.

Note: it will take more than 1000 but less than 10000 iterations to obtain a good approximation of the steady-state solution. So we set `MAXSTEP` to 10000.

Include the code changes in your report.

Experiment with 1, 2, 4, and 8 processors by editing the runb-trace.sh script (e.g. change `srun -n 4` to srun -n 8) and submit the job:

```
[youracct@hpc-login-35 ~]$  make clean
[youracct@hpc-login-35 ~]$  sbatch runb-trace.sh
```

The slurm-xx.out should show the grid, number of iterations, and total time taken to compute.

Repeat if necessary to verify the stability of the timings.

For each run, write down the total parallel execution time reported by the program in slurm-xx.out and number of iterations it took to converge. Calculate the relative speedup and add to your table. Your table should have four rows, one for each number of processors 1, 2, 4, and 8 that you tested. Also for each run of 1, 2, 4, and 8 processors, open jumpshot to inspect the trace. Estimate the ratio of t_{comp} to t_{comm} by comparing the timelines of `MPI_Sendrecv` and `update` (where `update` shows the total time spent in that function, including `MPI_Sendrecv`). Write this ratio in your table.

2.3 Task 3: MPI Asynchronous Non-blocking Implementation

The disadvantage of the previous implementation is that communication and computation are not overlapped. By overlapping we can try to 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, and the network use versus shared memory. Performance is also affected by tracing overhead, triggering 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 (due to changes to 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 $i = 2, \ldots, n-2$. 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 1, 2, 4, and 8 processors by editing the runa-trace.sh script (e.g. change `srun -n 4` to srun -n 8) and submit the job:

```
[youracct@hpc-login-35 ~]$  make clean
[youracct@hpc-login-35 ~]$  sbatch runa-trace.sh
```

The slurm-xx.out should show the grid, number of iterations, and total time taken to compute.

For each run, write down the total parallel execution time reported by the program and number of iterations it took to converge. Calculate the relative speedups and add to your table. Your table should have four rows, one for each number of processors 1, 2, 4, and 8 that you tested. Also for each run of 1, 2, 4, and 8 processors, open jumpshot to inspect the traces. Estimate the ratio of t_{comp} versus everything else that MPI does as t_{comm}. To do so, execute after each run:

```
[youracct@hpc-login-35 ~]$ ./jumpshot.sh
```

In the "Legend" window select the triangle next to "V" and "disable all". Then in the "V" column check `MPI_Isend`, `MPI_Irecv`, `update`, `message 3` and `message 4`. In the TimeLine window zoom in (by clicking the (+) button repeatedly) to the region on the right side of the timeline.

2.4 Bonus task +10% to this assignment: Gauss-Seidel Relaxation

Copy `heatdista.c` to a new file `heatdistc.c` to implement your changes. Also change `Makefile` by adding a build step for `heatdistc.c` and create `runc.sh` and `runc-trace.sh` files to submit `heatdistc`.

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

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 four rows, one for each number of processors 1, 2, 4, and 8 that you tested. Also for each run of 1, 2, 4, and 8 processors, open jumpshot to inspect the traces. Estimate the ratio of t_{comp} versus everything else that MPI does as t_{comm}.

- End.