Laboratory Work in Programming of Parallel Computers, TDDE65

Sebastian Litzinger, Sajad Khosravi

Based in part on earlier work by:

August Ernstsson, Lu Li, Usman Dastgeer, Fredrik Berntsson, and Mikhail Chalabine

April 13, 2026

1 Introduction

The purpose of the laboratory work is to get "hands-on" experience in programming parallel computers. You will implement programs to solve your assignment on the different architectures using communication primitives that are characteristic of the specific paradigm on each computer. The work should be done in groups of two (preferred) or three students. Every participant should clearly understand the assignment, work with the implementation, and understand the algorithms and solutions. Some additional information regarding the programming exercises can be found on the course homepage[1].

Please also refer to the course webpage for important **dates and deadlines**.

2 Assignments

Table 1 describes what you are supposed to do.

Table 1: *Required assignments for TDDE65 course.*

Lab No.	Lab Name	Sections
1 a	Image filter (`pthreads`)	4, 4.4.2
1 b	Image filter (`MPI`)	4, 4.4.1
2	Heat Equation (`OpenMP`)	5
4	Miniproject: Particles	6

[1] `www.ida.liu.se/~TDDE65`

2.1 Demonstration

For lab 1, 2, and the miniproject, you should demonstrate each lab assignment to your laboratory assistant. During demonstration, each group member should be able to explain all of their contributions. Show your source code and explain your implementation. Some things to think about when preparing your presentation:

- **Figures** are always a huge help when trying to express your algorithm, task partitioning, communication flow, etc.
- The **PCAM** method can help you conceptualize and express your approach.
- Provide **graphs** of performance measurements. Show how the execution time scales with different problem size, and even more importantly, different number of cores/ranks. Make sure to use more than one full node (32 ranks) for one data point in the MPI assignments.

2.2 Report

The miniproject has, in addition to the usual demonstration, a written report requirement. Express your approach to solving the problem in the usual report format, and include performance evaluation results and subsequent discussion. Include figures to illustrate communication patterns.

One chapter for each of the two tools should also be included. Discuss your experiences with the tools and relate back to the performance results. How do the tools help you find bugs and identify performance bottlenecks?

A report outline will be available on the course website soon.

3 Working on Sigma, NSC's supercomputer

The Sigma parallel computer cluster at NSC is available for the laboratory assignments.

3.1 Organizational issues

To do laboratory assignments, you need to secure an account on Sigma. The procedure for getting an account on NSC machines (including Sigma) takes some steps. Please follow the instructions given on
`https://www.ida.liu.se/~TDDE65/nsc.shtml`.

As soon as you have your account information you can login to the NSC computer using ssh: `ssh <username>@sigma.nsc.liu.se`. However, using the ThinLinc client is recommended for longer sessions.

More information about Sigma, including its architecture, available software, resource management system (SLURM), etc. can be found at NSC's web pages at `https://www.nsc.liu.se`. See especially `https://www.nsc.liu.se/systems/sigma/`.

3.2 Using Sigma, SLURM, etc.

For information specifically about the workflow of Sigma building, running, and debugging, see lecture slides by Frank Bramkamp and the "Quick Reference for Sigma" document, both available on the course webpage.

3.3 Version control

Git is available on the NSC system by default (no modules needed). We recommend using Git for the course, as it provides both version tracking features, facilitates moving source code between your computer and Sigma, and is great for collaboration. Just make sure to keep your Git repository private among the group members. You can use the LiU GitLab service [2] for free. Note, however, that the assistants do not usually have time to provide support with Git or GitLab.

[2] `http://gitlab.liu.se`

4 Image Filter

The assignment consists of implementing two simple image transformation algorithms. The transformations are from an input image to an output image. There is source code available on the course webpage that provides serial implementations of the two transformations described below.

Use as much as you like of the given serial code. However, observe that the code is not cache optimized and that the structure of the code is not intended to have any similarity to suitable structures for parallel implementations (this does not mean that it necessarily has a bad structure for a parallel implementation).

4.1 Averaging Filter

The first transformation makes the image "blurry". The algorithm works as follows:

> The value for a pixel (x, y) in the output image is the normalized weighted sum of all the pixels in a rectangle in the input image centred around (x, y).

In other words, the pixel in the output image is an average of the rectangle-shaped neighbourhood of the corresponding pixel in the input image. To be more precise:

$$c_o(x_0, y_0) = \frac{\sum_{x=x_0-r}^{x_0+r} \sum_{y=y_0-r}^{y_0+r} w(x - x_0, y - y_0) c_i(x, y)}{\sum_{x=x_0-r}^{x_0+r} \sum_{y=y_0-r}^{y_0+r} w(x - x_0, y - y_0)}$$

Where

- $c_i(x, y)$ is the color of the pixel (x, y) in the input image;
- $c_o(x_0, y_0)$ is the color of the pixel (x_0, y_0) in the output image;
- r - size of the averaging rectangle
- $w(x - x_0, y - y_0)$ the weights distribution function.

The weight function used in the lab outputs coefficients of Gaussian (normal) distribution.

Your implementation should work with different sizes of the rectangle. Therefore, it is recommended that the function implementing this transformation take parameters specifying the size.

4.2 Thresholding Filter

The second transformation is a thresholding filter. The filter computes the average intensity of the whole input image and uses this value to threshold the image. The result is an image containing only black and white pixels. White for those pixels in the input image that are lighter than the threshold and black for those pixels in the input image that are darker than the threshold.

4.3 Image Format

There are some images available on the course webpage with the source files. These images are saved in the *ppm* format, the binary version. The serial code examples contain code to read ppm files and write ppm and pgm files. Your implementations should work with the images `im1.ppm`, `im2.ppm`, `im3.ppm` and `im4.ppm`. They are of different sizes but you can assume that they all are at most 3000×3000 pixels.

4.4 Platform

You have to implement the same filter functionality using two different programming models on the same platform:

4.4.1 Sigma and MPI

In MPI, the communication can be implemented in different ways. Choose a suitable method and motivate the choice in the report you hand in. You may use C or C++ for the lab.

Try to minimize the communication, and avoid overallocating memory, such as allocating the entire data set on each rank. If you do this, you have to clearly motivate why this is the best approach. It is allowed to allocate the entire input and output data set on *one* rank (e.g. rank 0), such as for reading and writing to disk. File system reads do not need to be parallelized and shouldn't be included in time measurements.

Use `MPI_Wtime` to measure the execution time.

4.4.2 Sigma and pthreads

In the beginning it might be easier to work on your local workstation - the pthreads library is portable. Later, when the algorithms works, move the program to Sigma. Verify that it works and that you can utilise up to 32 cores. You can use C or C++ for the lab.

Use `clock_gettime` to measure the execution time.

5 Stationary Heat Conduction Using OpenMP

In this assignment you will solve a stationary heat conduction problem on a shared memory computer (a single compute node on `sigma.nsc.liu.se`), using OpenMP. A serial code for solving the problem is given in the file `laplsolv.c`, which is found on the course home page[3], and should be used as a starting point for your implementation. (Optionally, this lab can be solved in Fortran, based on the serial code in `laplsolv.f90`. The lab assistants can, however, only give limited assistance for Fortran.) Your final parallel program should produce exactly the same results as the serial code does. Do not necessarily assume that the given files are optimized for performance or memory usage; you are encouraged to improve performance in any way you can.

5.1 Description of the problem and the numerical method

The purpose of this section is to explain briefly the numerical details of the code that you are going to use in this excercise. It is not necessary to understand all parts of this discussion.

The problem we are going to solve is the following: Find the stationary temperature distribution in the square $[0,1]\times[0,1]$, if the temperature at the boundary is specified as in Figure 1. The stationary temperature is described by the differential equation

$$\frac{\partial^2 T}{\partial x^2}+\frac{\partial^2 T}{\partial y^2}=0, \qquad 0<x,y<1.$$

We introduce an equidistant grid $\{(x_i,y_j)\}_{i,j=0}^{N+1}$, as seen in Figure 2, and discretize the differential equation using finite differences. Thus the differential equation is replaced by a system of linear equations,

$$-4T_{i,j}+T_{i+1,j}+T_{i-1,j}+T_{i,j-1}+T_{i,j+1}=0, \quad 1\le i,j\le N-1, \tag{1}$$

where $T_{i,j}=T(x_i,y_j)$. The number of unknowns is N^2, and if a large number of grid points is used the problem will be too large to solve using direct methods, e.g. Gaussian Elimination. Instead the problem is solved iteratively.

Let $T_{i,j}^k$ be the approximate temperature for grid point (x_i,y_j) at the kth iteration. The next iterate $T_{i,j}^{k+1}$ is computed by

$$T_{i,j}^{k+1}=(T_{i+1,j}^k+T_{i-1,j}^k+T_{i,j-1}^k+T_{i,j+1}^k)/4, \qquad 1\le i,j\le N-1.$$

Thus the new approximation of the temperature at grid point (x_i,y_j) is obtained by taking the avarage of the values at the neighbouring gridpoints. This particular iterative method is known as the Jacobi method, see Section 8.1.

5.2 Description of the code

The serial program is available in the course home page (see previous page for link).

[3] `www.ida.liu.se/~TDDE65/labs`

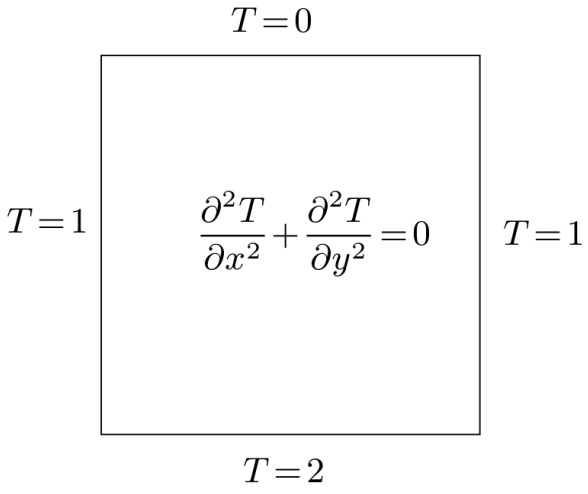

Figure 1: Boundary conditions.

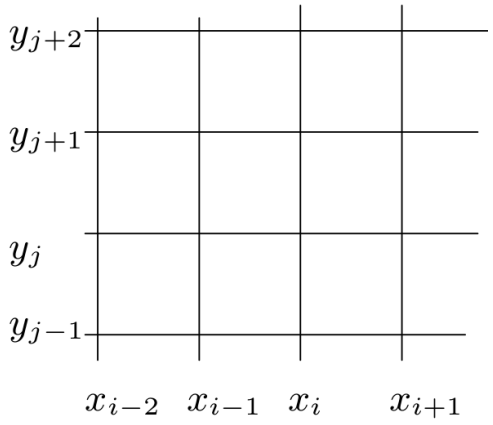

Figure 2: The computational grid.

The temperature data are stored in an $(n+2)\times(n+2)$ matrix,

$$T=\begin{pmatrix} T_{0,0} & T_{0,1} & \dots & T_{0,n} & T_{0,n+1} \\ T_{1,0} & T_{1,1} & \dots & T_{1,n} & T_{1,n+1} \\ \vdots & \vdots & & \vdots & \vdots \\ T_{n,0} & T_{n,1} & \dots & T_{n,n} & T_{n,n+1} \\ T_{n+1,0} & T_{n+1,1} & \dots & T_{n+1,n} & T_{n+1,n+1} \end{pmatrix},$$

where, as previously, $T_{i,j}$ denotes the temperature at grid point (x_i, y_j). Note that only the middle part of the matrix contains unknowns since the temperatures at the boundary of the square are known. The boundary data are explicitly set at the beginning of the computation.

Suppose that the matrix `T` contain the values $\{T^k_{i,j}\}$, i.e. the approximate solution at the kth iteration. The next iterate, i.e. the values $\{T^{k+1}_{i,j}\}$, can be computed by performing the steps of the following pseudocode[4]:

[4]Actually it is Fortran, which has a concise array notation.

```
tmp1 = T(1:n, 0)
do j = 1, n
   tmp2 = T(1:n, j)
   T(1:n, j) = (tmp1 + T(1:n, j+1) + T(0:n-1, j)+T(2:n+1, j)) / 4.0
   tmp1 = tmp2
end do
```

The temporary vectors, `tmp1` and `tmp2`, are necessary since the old values in column j are needed for computing the new values in the $(j+1)$th column.

The assignment is to parallelize the provided serial code using OpenMP. When parallelizing, please consider the following:

1. You should only use at most $\mathcal{O}(N)$ additional memory where N is one side of the square[5].

2. Do not run any computations with more than 100×100 grid points before you are convinced that your code works.

3. Make sure that your parallel code produce exactly the same results as the serial code.

4. The error estimate that is used as a stopping rule for the iteration is a bit tricky to parallelize.

Use the compiler option `-fopenmp` when you compile your program. The number of processors is set by executing the command `export OMP_NUM_THREADS=p`, where `p` is the number of procerssors. On the NSC system, the `ompsalloc` command handles this for you when you specify the corresponding resource specification flags.

[5]This means that you *cannot* create, for example, a copy of the T matrix as it would result in $O(N^2)$ extra memory usage.

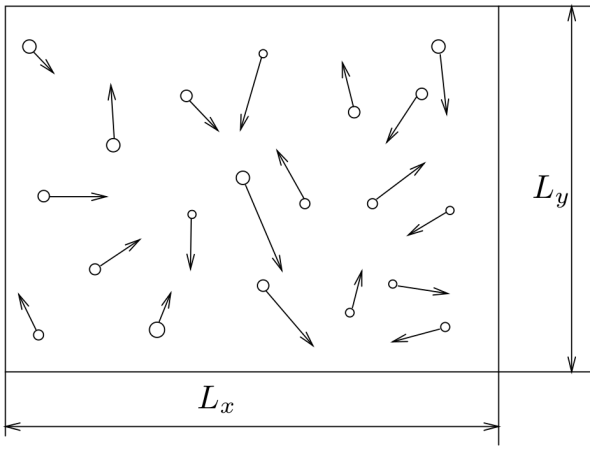

Figure 3: Gas simulation by rigid bodies.

6 Miniproject: Particle Simulation

In this assignment we will do a particle simulation and verify the gas law $pV = nRT$. The particles are hard with a radius 1 and all collisions will be regarded as perfectly elastic (with the walls and other particles) and no friction is present in the box. The box will be a 2 dimensional rectangle (the collisions will be easier to handle). Until a collision occur the particles will travel straight (no external forces) and if a collision occur the momentum and energy is conserved, by the elastic collision. From the following relationships the velocity after the collision can be found, m_1, m_2 is the mass of the particle and $\hat{v}_{(x,y)}$ is the velocity before the collision and $v_{(x,y)}$ after, see Figure 4. The law of conservation of the momentum is, after a suitable rotation of the coordinate system,

$$m_1 v_{1,x} + m_2 v_{2,x} = m_1 \hat{v}_{1,x} + m_2 \hat{v}_{2,x}$$

and the kinetic energy

$$m_1(v_{1,x}^2 + v_{1,y}^2) + m_2(v_{2,x}^2 + v_{2,y}^2) = m_1(\hat{v}_{1,x}^2 + v_{1,y}^2) + m_2(\hat{v}_{2,x}^2 + v_{2,y}^2).$$

The coordinate system is rotated so the tangent to the collision point is vertical so there is no change in the velocity in the y-direction.

When a particle hits wall the particle will bounce back with negative velocity normal to the surface.

With this simulation one can simulate the notion of pressure, the bouncing particles will exhibit a pressure on the walls each time they hit. Each time a particle hits a wall a momentum of $2mv_{x,y}$ will be absorbed by the wall. If we sum all collisions by a wall during t second and divide this by the circumference of the box and t, we will obtain the (two dimensional) pressure in the box. You will use this pressure to verify the pressure law $pV = nRT$, p pressure, V volume (here area), n number of moles (number of particles), R magic constant and T is the temperature, in our case the volume will be instead area.

6.1 Implementation

Write the framework for the simulation of small particles in a rectangular box. Choose and motivate a good distribution of the particles between the processors, implement the

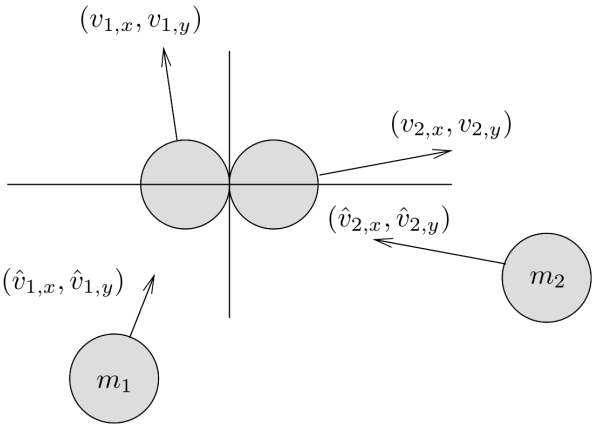

Figure 4: Interaction between two particles.

communication between the processors using MPI, count the pressure. You may use either Fortran or C/C++ as the implementation language.

6.1.1 Data types

The particles in the provided functions are represented by the following struct

```
struct part_cord {float x; float y; float vx; float vy;}
typedef struct cord cord_t;
```

x, y is the position, v_x, v_y the velocity. The walls can be represented by

```
struct cord {float x0; float x1; float y0; float y1;}
typedef struct cord cord_t;
```

These datatypes are defined in the file `coordinate.h`. The particles can be stored on each processor in a fix array.

6.1.2 Functions provided

The interaction between the particles is provided by the following routines in the file `physics.c`;

```
float collide (pcord_t *p1,pcord_t *p2)
interact (pcord_t *p1,pcord_t *p2,float t)
float wall_collide (pcord_t *p, cord_t wall)
feuler (pcord_t *a, float time)
```

The routine `collide` returns -1 if there will be no collision this time step, otherwise it will return when the collision occurs. This will then be used as one of input parameter to the routine `interact`. The routine `interact` moves two particles involved in the collision. Do **not** move these particles again. `wall_collide` checks if a particle has exceeded the

boundary and returns a momentum. Use this momentum to calculate the pressure. The routine `feuler` moves the a particle.

6.1.3 Important implementation issues

The files necessary for the implementation are available on the course home page[6]. Download the files to your working directory on Sigma. Some simplifications to the model can be made. If the particles are small compared to the box and the time step is short, the possibility that a particle will collide with more than one other particle is statistically very small, so if a particle hits another, we can update both and ignore them until the next time step (this is done in the procedure `interact`). Observe that this can be implemented without some sort of update flag array!. Depending on how the particles are distributed over the processors some simplifications of the communication can be done, *motivate each simplification done!*

Also note that if you make such simplifications, when doing the performance and speedup evaluation you should compare to a sequential program *with the same simplifications made*. We are interested in speedup numbers stemming from parallelization, not from approximations or algorithmic reformulations.

- Each time-step must be 1 time unit long.
- The initial velocity should be less then 50. Use the random number generator to generate the absolute velocity and a starting angle. (r=rand()·max_vel; θ=rand()·2π; $v_x = r\cos(\theta)$; $v_y = r\sin(\theta)$)
- Typical numbers for the simulation; number of particles = 10000 × number of processors, and area of the box = $10^4 \cdot 10^4$.
- The pressure can be found at the end of the simulation by dividing the total momentum from the routine `wall_collide` with the number of time-steps and the length of the circumference of the box.
- Think about pros and cons of data structure (arrays, linked list etc.) that you use to represent particles.
- Avoid unnecessary communication by sending all particles at once.

[6] `www.ida.liu.se/~TDDE65/labs`

6.2 A short summary of the structure of the program

- Initiate particles
- Main loop: for each time-step do
  - for all particles do
    - Check for collisions.
    - Move particles that has not collided with another.
    - Check for wall interaction and add the momentum.
  - Communicate if needed.
- Calculate pressure.

6.3 Questions

Before the demonstration, you should measure and note down the following:

1. Explain your choice of distribution of the particles over the processors. Is there an optimal relation between the distribution and the geometry of the domain? Measure this by counting particles passed between processors in each time step.
2. Verify the gas law $pV = nRT$ by changing the number of particles (n) and size of the box (V) and then measure the pressure.

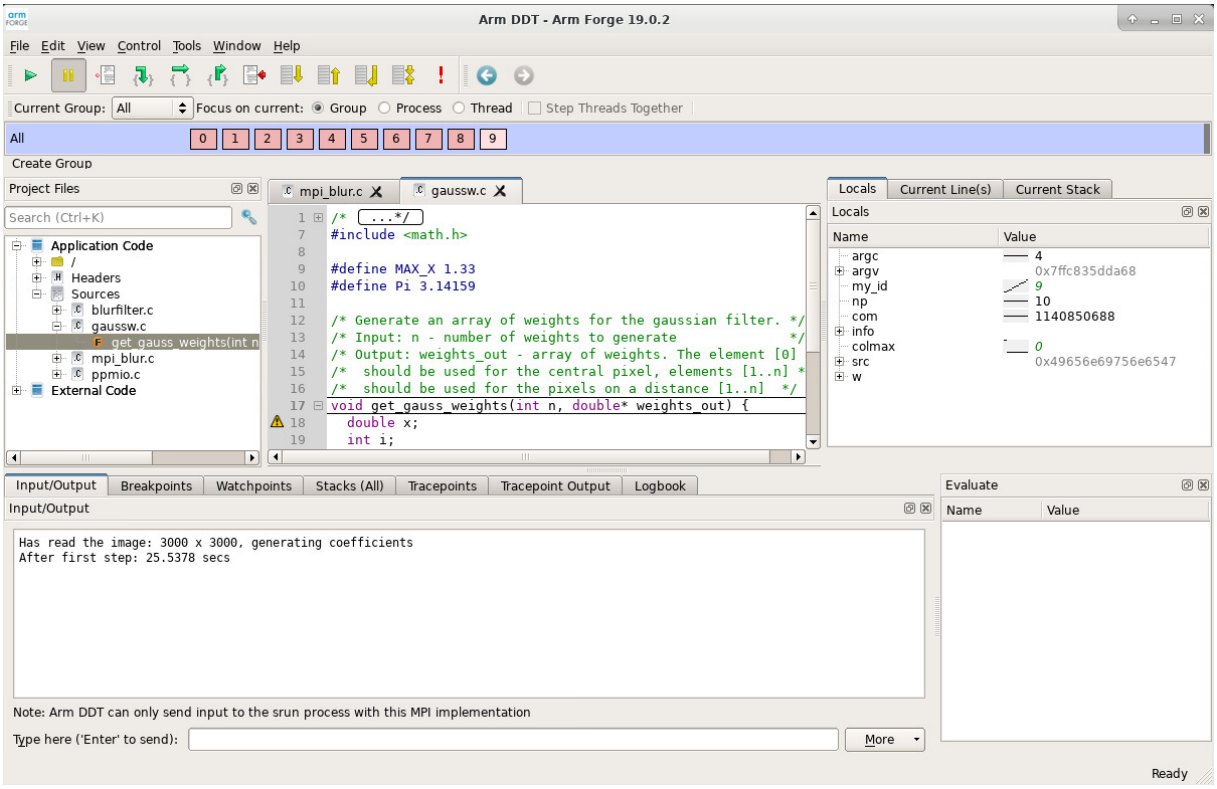

Figure 5: DDT main view, showing 10 ranks running an MPI program.

7 Tools

During this lab you will try two different tools: a parallel debugger, *DDT*, and a tracing tool, *Intel Trace Analyzer and Collector*, ITAC.

Note: External documentation for these tools often show different, or even wrong, ways to invoke them (for NSC/this course). Always look at the NSC slides and website documentation first.

7.1 Debugging with DDT

DDT is a powerful, interactive parallel debugger and works with both MPI and OpenMP. The debugger shows in-line source code which can be expanded or collapsed, and offers standard debugging features such as breakpoints and watchpoints also for parallel programs. The variable inspector can, for instance, show the contents of the same variables on all ranks simultaneously.

Instructions for how to run DDT can be found in the NSC slides and in the quick reference manual for the course. See the DDT documentation on the NSC web page (`https://www.nsc.liu.se/software/installed/tetralith/allinea-DDT/`) and the DDT user manual at `http://content.allinea.com/downloads/userguide-forge.pdf` for more information about the program.

Your miniproject report should contain a short review of the features that you have used. Compare DDT with other debuggers that you used before. Comment on ease of use, manageability for parallel application debugging, etc.

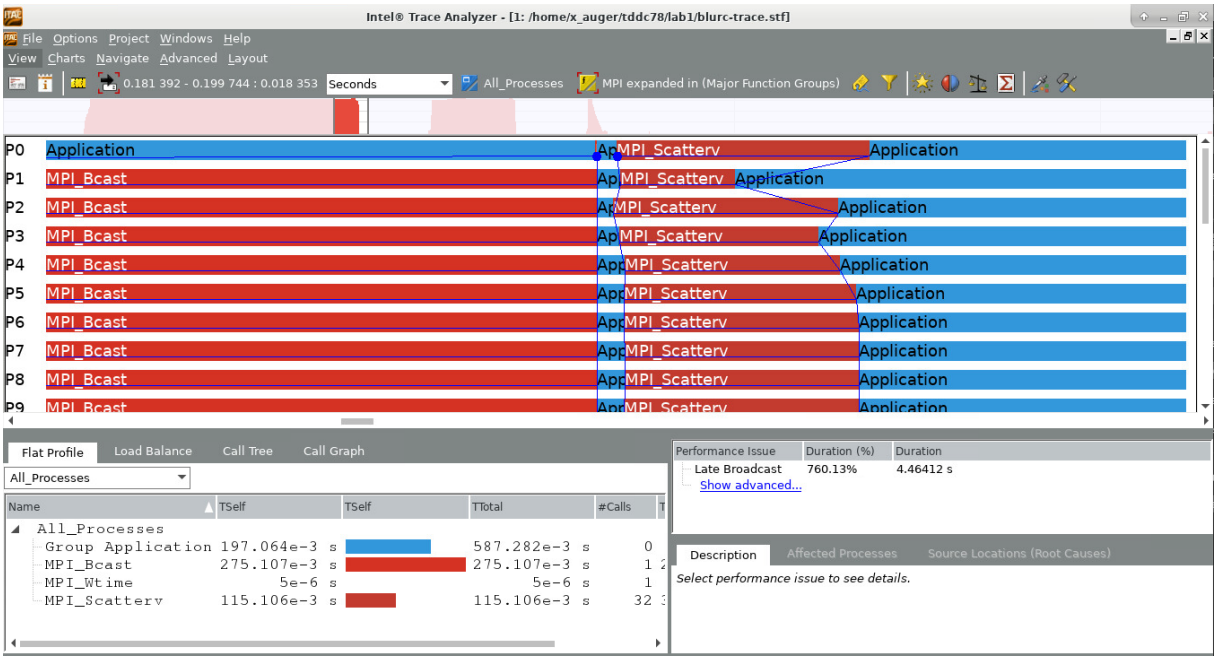

Figure 6: ITAC main view, showing 32 ranks running an MPI program.

7.2 Tracing with ITAC

ITAC is an interactive visualization tool designed to analyze and debug parallel programs, in particular message passing programs using the MPI interface. ITAC is installed on Sigma and all the necessary environment variables are set after following the instructions given in the NSC slides.

Unfortunately, ITAC is not fully publicly advertised on Sigma yet and detailed documentation is still missing. Explore the interface and see what kinds of visualization views are available.

OBS: Your task is to trace the particle simulation lab and present screenshots along with appropriate description/explanation of the traces in your report. [7]

Take screenshots of a few different visualization views that help you identify performance bottlenecks or other interesting information and include in the report.

[7] You may alternatively present traces from another lab as long as they are not overly trivial.

8 Appendix

8.1 The Jacobi method

Usually linear systems that originate from the discretization of partial differential equations are too large to be solved by direct methods. Instead iterative methods are used. In this section we discuss the Jacobi method, which is one of the simplest (and least efficient) iterative methods for solving linear systems of equations. For simplicity we restrict the discussion to 3×3 matrices.

Consider the linear system $Ax=b$, where

$$A=\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}, \quad b=\begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}.$$

The starting point of the Jacobi method (and other iterative methods) is the splitting of the matrix A into two parts, $A=N-M$, representing the diagonal and off–diagonal elements respectively, i.e.

$$N=\begin{pmatrix} a_{11} & 0 & 0 \\ 0 & a_{22} & 0 \\ 0 & 0 & a_{33} \end{pmatrix}, \text{ and, } M=\begin{pmatrix} 0 & -a_{12} & -a_{13} \\ -a_{21} & 0 & -a_{23} \\ -a_{31} & -a_{32} & 0 \end{pmatrix}.$$

Using this notation, the original system of equations, $Ax=b$, can be written as

$$Nx=Mx+b, \quad \text{or,} \quad x=N^{-1}Mx+N^{-1}b.$$

The Jacobi method is based on the above formula. Given an approximation x^k of the solution we compute a (hopefully) better approximation x^{k+1} by

$$x^{k+1}=N^{-1}Mx^k+N^{-1}b.$$

The Jacobi method is very inefficient and should not be used for real–life problems. However, it is as difficult to parallelize as the more sophisticated iterative methods.