Student's Guide to 1TD070 62007[1]
Version 0.9

Roman Iakymchuk

March 19, 2024

[1]Courtesy of Mikael Rännar, Lars Karlsson and Jerry Eriksson

Contents

Chapter 1

Introduction

In this course you will be introduced to *parallel and distributed programming*: the science and art of constructing and evaluating high-performance parallel algorithms and software. The term *parallel computing* refers to simultaneous execution on multiple processor cores for the primary purpose of speeding up a single computation. This term should not be confused with *concurrent computing* and *distributed computing*. The former (concurrent computing) has to do with computations that *appear* to run at the same time but might in actuality be time-shared on a single core. The primary purpose of concurrent computing is to provide a convenient abstraction rather than to gain performance. The latter (distributed computing) deals chiefly with physically distributed and often loosely coupled computations such as web applications and distributed databases. However, *distributed programming* refers to a branch of parallel programming and is focused on programming multiple nodes connected with a high speed interconnect forming a cluster or a supercomputer.

The *primary purpose* of parallel computing is to gain *performance*, i.e., to increase the *rate of execution* (the speed at which work is being performed). For example, if we are rendering an animated movie we would like to increase the number of *frames per second* that our rendering program can sustain. On the other hand, simply completing the same type of rendering jobs that we could do before but in less time is probably not our main goal. Instead we would like to use the time that we have bought ourselves by increasing the performance to *increase the quality* of the animation we can render in the same amount of time. Rather than rendering in an hour what used to take us a week we could add more visual and physical effects and obtain a better quality movie out of one week's worth of computing.

One faces a number of significant *challenges* when trying to transform a sequential computation into a parallel one. First of all, the computation must be re-imagined not as an ordered sequence of operations but as a *partially ordered set of tasks*. The partial ordering captures dependencies between tasks of the form "make sure to complete A before you start with B" while still leaving enough leeway to express such things as "B and C can be done in any order or even simultaneously". Deciding on a *task decomposition* is crucial and one of the most creative aspects of parallel programming. But a task decomposition by itself is not enough; there are more challenges to address. Perhaps the most significant of these challenges is how to *schedule* the tasks on the available processor cores such that the overall execution time

is minimized. We would like to avoid *(parallel) overhead* caused by such things as processor cores being *idle* (without anything to compute), cores *communicating* and *synchronizing* with each other, and cores *redundantly recomputing* some intermediate quantities.

Once we have grasped the purpose of parallel computing and its primary challenges, we are ready to begin to understand the *concepts and ideas* that have been developed within the parallel computing community. These ideas help us better think about how to approach the challenges in any given application. In short, as we progress through the course we will become better and better at *thinking like a parallel programmer*, which you might say is the primary goal of this course.

Last but not least, we should learn the basics of some *technologies* that we can use to turn our ideas into real executable parallel programs. These technologies come in different shapes and forms. For example, we have *parallel programming languages and language extensions*, *parallel programming APIs and libraries*, *parallel debuggers*, and *parallel profilers/tracers*, just to name a few. We will in this course study some of the most representative and commonly used technologies to give you an idea of the significant diversity that exists in the technological landscape today.

We hope that you are ready to dive head first into this course that we have designed for you. It will certainly not be easy and there will be ups and downs. But if you persevere you are bound to learn some useful ideas and new ways of thinking.

Best wishes from your teacher,
Roman Iakymchuk

1.1 Literature

We will be using the following textbook:

> **An Introduction to Parallel Programming**
> Peter S. Pacheco
> ISBN 9780123742605
> 2011

In addition you may be asked to read a couple of scientific articles.

1.2 Teachers

- **Roman Iakymchuk**
  Course responsible teacher
  Office: Hus 10, 106195
  Email: `roman.iakymchuk@it.uu.se`

Chapter 2

Teaching and learning strategies

When we design a course we start from the goals of the course. We then decide how we will measure your progress towards these goals. Once these things have been determined, we need to decide how to guide you towards fulfilling the goals. This requires us to imagine what you will have to do in order to learn the subject well enough to pass the examinations with excellent results. We call this the *learning strategy*: the strategy that we intend for you to use during the course. Finally, we decide what we, as teachers, should do (within our limited resources) to most effectively support your learning. We call this the *teaching strategy*: the strategy that we will use to support your learning. The teaching and learning strategies go hand in hand; they are designed to be used together.

2.1 Learning strategy

Take a look at the five vertical blue rectangles in Figure 2.1. These correspond to the first five chapters of the textbook. We call them *topics*. The first topic focuses on the purpose of parallel computing and introduces the subject's most central concepts and ideas. The second topic studies the interface between software and hardware so that we better understand how our software interacts with the hardware. The remaining topics focus on some of the most mature and widely used parallel programming models/APIs in use today.

Since the author of the textbook has chosen to structure his book in this way, his perspective on the subject is what you will assimilate as you work your way through the book. His perspective can be characterized as *technologically centered*. An *application centered* perspective, focusing on how to apply parallel computing in applications, would be an equally valid approach. Four different application areas are shown in Figure 2.1 as horizontal green rectangles. As you will see in Section 2.2 below, our teaching strategy aims to support your work with the topics as well as complement the book by helping you obtain also an application centered perspective.

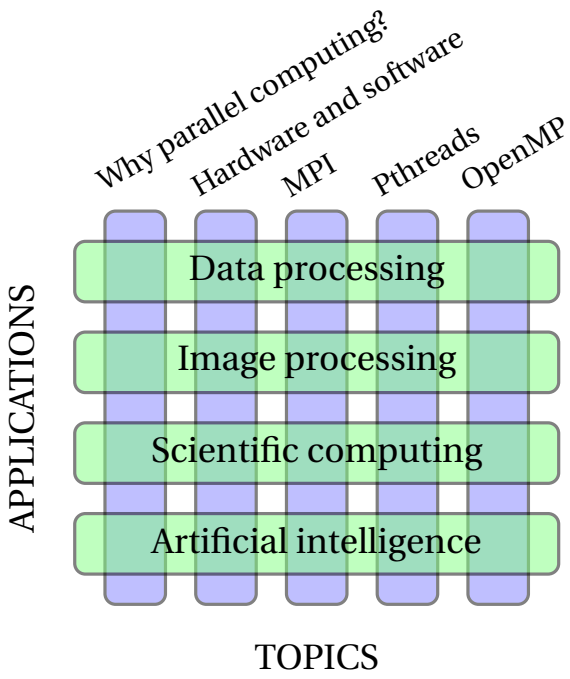

Figure 2.1: Two different perspectives on the subject. Vertical (blue): Chapters/topics in the textbook. Horizontal (green): Application areas studied in class.

2.1.1 Activities

We intend for you to learn primarily by engaging in these types of activities:

Reading the textbook, articles, and technical manuals You should read all of the textbook and the scientific articles assigned during the course. You will also find it useful to read selected parts of the technical manuals for the parallel programming APIs and tools you will use.

Working with exercises For each topic there is a small number of mandatory exercises (most of which have been adapted from the textbook) that you must complete and submit written solutions to us.

Identifying difficult concepts and problems Make it a habit to take a break once a day to identify the concepts and problems that you find most difficult to understand or deal with. Write down what you find and ask your teachers for advice on how to move past the current obstacles.

Participating in discussions of topics Shortly after each topic exercise deadline there is a scheduled activity for discussions on that topic (we strive to cover all). There we will discuss solutions to the exercises and discuss any other matters related to the topic. Come prepared to make sure you get some answers to your lingering questions.

Participating in lectures and demonstrations During the lectures and demonstrations we will model how to think like a parallel programmer. The way you learn from this is by following along with the thought process. However, it is very easy to get lost in this type of group activity since the pace and the presentation is adapted to the group rather than each individual. You should strive to be active and never hesitate to frequently ask for clarifications and elaborations. Do not allow yourself to become lost!

2.2 Teaching strategy

Our teaching strategy is designed to support the learning strategy outlined above.

2.2.1 Activities

We will support your learning primarily by these types of activities:

Regular lectures The textbook, articles, and technical manuals are your primary sources of information. We have scheduled a small number of regular lectures on narrow topics that students in the past have struggled to sufficiently understand by themselves. In this course we will *not* "cover all the material" during lectures. At this point in your studies we assume that you are fully capable of acquiring basic knowledge by reading.

Discussions related to topics Shortly after each topic exercise deadline there is a scheduled discussion on that topic. The purpose of these sessions is to discuss solutions to the exercises and provide an opportunity to get answers to any lingering questions you might have. Therefore it is crucial for your own benefit that you come prepared and contribute to the discussion.

Demonstrations of thinking in applications Working with the exercises will help you get a basic understanding of the fundamentals and technical details. What is not so easy to get from reading is a higher-level perspective that allows one to understand how to think in the context of particular applications. There are some scheduled sessions in which we will demonstrate how to think like a parallel programmer in various applications (see Section **??**). These sessions give you a chance to lift your level of understanding from basic ideas to a more complete and usable comprehension of the subject. The better prepared you are (by reading and working with exercises), the more you will get out of these demonstrations.

Support via email and slack Besides the scheduled activities we will be available via email and slack. In case needed, we (teacher and two TAs) can schedule (drop an email before) face-to-face meetings. Do not hesitate to contact us for any reason whatsoever. We are here to help you.

Chapter 3

Topics / assignments (under development)

This chapter contains three sections; one for each assignment. Each assignment corresponds to one or two chapters in the textbook. For each assignment we suggest what to read and specify the exercises which are mandatory to individually (or in pairs) solve and submit written solutions and the code in a zip or a tarball to us (see Chapter 5 for deadlines). Reading and working with the exercises is how you will spend the majority of your time. *NB: if working in pairs, write a precise and concise statement as the last section, explaining contributions of each student.*

3.1 Assignment 1

3.1.1 Assigned reading

- Refresh your knowledge of (and skill in using) the C programming language. Fortran is also fine.
- Refresh your skills in using the Linux command line interface, including secure remote terminal access and secure file transfer. This is needed to work on UPPMAX
- Read Chapters 1 and 2 in the book.

3.1.2 Assigned exercises

1. **Formulas for block partitioning** (adapted from 1.1)
   Suppose that you are to compute the sum of an array of length n using p processing units. You partition the array into p blocks of near uniform size. Construct formulas for the *first* and *last* indices of block number $k = 0, 1, \ldots, p-1$. Hint: first consider the special case when $p \mid n$ (n is divisible by p).
2. **Tree-structured global sum** (adapted from 1.3)
   Suppose that you have p processing units, each of which has a number. The goal is to compute their global sum using the tree-structured algorithm sketched in Figure 1.1 in

the textbook. The final result should end up on processing unit 0. Construct an algorithm for processing unit $k = 0, 1, \ldots, p-1$ using the communication primitives `send` and `receive`. Note: your solution must work for *any* number of processing units (not only powers of 2, but start with powers of 2).

Tasks:

- construct an algorithm and explain it
- implement your solution
- test on random numbers and different array sizes on different number of processes

3. **Cost analysis of tree-structured global sum algorithm** (adapted from 1.6)
Count the number of *receives* and *additions* that processing unit 0 performs in the algorithm constructed in the previous exercise. Express your solution on the form

$$T(p) = ?r + ?a,$$

where r and a are machine-specific parameters for the time it takes to do one receive and one addition, respectively. Your job is to determine what should go in the places marked with '?'.

4. (optional) **Hardware multithreading and caches** (adapted from 2.8)
Caches can improve the effective memory bandwidth and latency by serving some memory accesses from fast cache memory. Hardware multithreading can reduce the negative impact of memory latency by switching to another (hardware) thread while the current thread waits for a memory access. But running many threads on a hardware multithreaded core might degrade the effectiveness of the cache. Figure out why. Hint: what happens with the cache while a thread is waiting for a memory access?

5. (optional) **False sharing** (adapted from 4.17)
Let A be a matrix of size $m \times n$ that is stored column-wise in an array `A` with a configurable column stride $s \geq m$. More precisely, the matrix element in (zero-based) row i and column j is stored in array index $i + js$. Suppose that we have p threads and wish to partition the matrix into p row blocks (one for each thread) with near uniform size. However, we are willing to compromise a little bit on the uniformity of the sizes if we can eliminate the possibility of false sharing. Describe how to choose the stride s and compute the sizes of the p row blocks such that false sharing is completely eliminated.

6. **Speedup and efficiency** (adapted from 2.16)
Suppose that the execution times of a sequential and parallel program (solving the same problem) are given by the functions

$$T_s(n) = n^2, \qquad T_p(n,p) = \frac{n^2}{p} + \log_2(p).$$

Produce two tables: one for speedup and one for efficiency. Label the columns by $n = 10, 20, 40, 80, 160, 320$ and the rows by $p = 1, 2, 4, 8, 16, 32, 64, 128$. Look down any particular column (i.e., keep n fixed). What happens to the speedup and the efficiency? Do you think this makes sense given that the problem size is fixed and the number of processing units increases? Now look across any particular row (i.e., keep p fixed). What happens now to the speedup and efficiency? Do you think this makes sense given that the number of processing units is fixed and the problem size increases?

7. **Scalability** (adapted from 2.19)
   Suppose that the sequential and parallel execution times are given by the functions

$$T_s(n) = n, \qquad T_p(n,p) = \frac{n}{p} + \log_2(p).$$

   If we increase p by a factor of $k > 1$, then by which factor do we have to increase n to maintain the same efficiency?

3.2 Assignment 2

3.2.1 Assigned reading

- Read Chapter 3 in the book.
- Hint: To get started with MPI, implement and test some of the codes given in Chapter 3 from the book.

3.2.2 Assigned exercises

1. **Array distributions** (adapted from 3.6)
   Let x be an array with 14 entries distributed over $p = 4$ processes. For each of the following array distributions, show how the array elements are mapped to processing units:
   (a) Block distribution.
   (a) Cyclic distribution.
   (a) Block-cyclic distribution with block size $b = 2$.

2. **Tree-structured algorithms for scatter and gather** (adapted from 3.8)
   Suppose that $p = 8$ and $n = 16$.
   (a) Draw a diagram that shows a tree-structured algorithm for `MPI_Scatter` when process 0 needs to scatter an array with n entries.
   (a) Draw a similar diagram but for `MPI_Gather`.

3. **Vector scaling and dot product** (adapted from 3.9)
   Write an MPI program containing parallel versions of the following functions:

```
void vector_scale(int n, double scale, double x[])
{
  for (int i = 0; i < n; i++)
    x[i] *= scale;
}

double dot_product(int n, double x[], double y[])
{
  double s = 0;
  for (int i = 0; i < n; i++)
    s += x[i] * y[i];
  return s;
}
```

The arrays should be distributed using a block distribution. You may assume that n is divisible by p. For each function, follow this algorithm:

(a) Read the function's inputs from standard input into process 0.

(b) Broadcast (the scalars) and scatter (the arrays) from process 0 as appropriate.

(c) Perform computations in parallel.

(d) Gather results on process 0.

(e) Print out the results to standard output from process 0.

4. **Matrix-vector multiplication** Write an MPI program that implements matrix-vector multiplication ($y = Ax$) using a block-column distribution of the matrix. You can have processor 0 read the matrix and simply use a loop to distribute it among the processors (and the same for the vector, which then will have a block(row) distribution). The result vector y should be distributed the same way as x. In the end the program should gather the result vector on process 0 and print it. Assume that the matrix is square of order n and that n is evenly divisible by `comm_sz`.

3.3 Assignment 3

Chapter 4

Examination

The examination consists of three parts, to be solved individually:

1. three sets of assignments,
2. one project

You must pass all two parts to pass the course. Your grade will be determined based on points accumulated on the project.

Please be advised that this course consists of a single module unlike some other courses which have two or even more modules. In other words, you will not receive any of the course credits until you have passed all parts of the examination.

4.1 Exercises

Written solutions to the three sets of assignments/ exercises must be submitted as PDFs no later than their respective deadlines as defined in Section 5.1. Acceptable solutions to *all* exercises is a requirement to pass the course. Errors/ mistakes that hint at misconceptions will be treated more harshly than other kinds of errors, since the purpose of grading the exercises is to test your current level of understanding and correct misconceptions as early as possible. Since you have taken a number of university courses already, we can demand clear and precise writing. Anything less will require a revision before passing.

4.2 Project

The final two/ three weeks of the course are set aside mostly for work on a mandatory project. The purpose of the project is to show that you have an ability to use what you have learned in this course to tackle a more substantial problem.

The project will be revealed towards the end of April and will be individual. Your task will be to implement a parallel algorithm for the assigned application using MPI and experimentally evaluate the performance of the resulting program.

You submit your source code as a ZIP archive together with a report in PDF format. The report should contain the following major components:

1. Statement of the problem.
2. Description of the sequential algorithm.
3. Description of the parallel algorithm.
4. Description of the experimental setup.
5. Summary and discussion of the experimental results.

The experiments should focus on these scenarios:

- A large problem of fixed size solved with an increasing number of cores.
- A large number of cores being used to solve increasingly large problems.

The experimental results should be presented in terms of parallel speedup. The report must present and analyze at least one trace obtained with the Paraver tool.

Chapter 5

Schedule

The full schedule is available on Studium (and in TimeEdit), see the Schedule menu item on the left. I will not repeat it here, because if there are any changes there will only be one place to update.

5.1 Lectures, deadlines, and exams

The content of all lectures can be found on Studium. All deadlines can also be found on Studium under Important Dates.
Please take note of the deadlines!

Appendix A

Accessing UPPMAX system

There is a document on Studium regarding Access to UPPMAX. We would like also to provide few useful links

- Getting started: `https://www.uppmax.uu.se/support/getting-started/`
- We will be using Snowy: `https://www.uppmax.uu.se/support/user-guides/snowy-user-guide`
- Multiple User Guides: `https://www.uppmax.uu.se/support/user-guides/`

A.1 Computer access

Project

A project is allocated a certain run-time on the clusters. All users at UPPMAX belong to at least one project and all jobs run on UPPMAX are associated with a project and the run-time is deducted from the total allocated CPU hours.

You belong to the project **uppmax2024-2-9** and it should be used in all submit files. Note that if you omit the project number in the submit file, then your job will receive a very low priority (and possibly never execute)—so don't forget it.

Log in

You need an SSH client to log in (e.g., `ssh` or PuTTY). The following instructions cover only Linux systems using `ssh` for remote access. If you use some other system or client, then you need to consult its documentation yourself.

Sign in to Snowy using the command:

```
ssh <your user name>@rackham.uppmax.uu.se
```

and then enter your default password. If you entered the correct username and password, then you should now be signed in. If you enter the wrong password three times in a row, then your account will be temporarily frozen and you will have to contact one of the teachers to have it unlocked again, so please be careful when you write your password! If you want to open graphical displays, you need to enable X11 Forwarding:

```
ssh -X <your user name>@rackham.uppmax.uu.se
```

Change password

The first thing you must do is to change your password using the command

```
passwd
```

Failure to do so during your first login may result in your account being locked.

A.2 File systems

For this course you do not have to consider the different type of file systems available at HPC2N. You can just use your home directory and its subdirectories. My advise is that you create one, or more, subdirectory(s) and change the access rights so that only you have access.

Secure file transfer

You can use `scp` for file transfer to/from the file systems at UPPMAX. Use these commands on your local computer:

```
# Copy to cluster
scp file username@rackham.uppmax.uu.se:file

# Copy to local machine
scp username@rackham.uppmax.uu.se:file /tmp
```

A.3 Modules

To set up your environment for using a particular (set of) software package(s), you can use the modules that are provided centrally.

On Snowy, interacting with the modules is done via Lmod, using the `module` command or the handy shortcut command `ml`.

The modules are installed in a hierarchical layout. This means that some modules are only available after loading a specific compiler and/or MPI version.

UPPMAX will from time to time make changes to the various packages—these will be transparent to you if you use `module` to manage your environment.

Loading and unloading modules

The module system have lots of commands for many things, including listing (which modules are currently in your environment), loading (making modules available to your environment), and unloading (deleting them from your environment).

An example of trying to use the Intel compiler without loading, and then loading and unloading:

```
b-an01 $ ml

Currently Loaded Modules:
  1) snicenvironment  (S) 2) systemdefault  (S)

  Where:
    S:  Module is Sticky, requires --force to unload or purge

b-an01 $ mpicc f.c                          #Can not find the compiler
-bash: mpicc: command not found
b-an01 $ ml foss
b-an01 $ ml

Currently Loaded Modules:
  1) snicenvironment (S)   9) libxml2/2.9.10     17) OpenMPI/4.1.1
  2) systemdefault   (S)  10) libpciaccess/0.16  18) OpenBLAS/0.3.18
  3) GCCcore/11.2.0       11) hwloc/2.5.0        19) FlexiBLAS/3.0.4
  4) zlib/1.2.11          12) OpenSSL/1.1        20) FFTW/3.3.10
  5) binutils/2.37        13) libevent/2.1.12    21) ScaLAPACK/2.1.0-fb
  6) GCC/11.2.0           14) UCX/1.11.2         22) foss/2021b
  7) numactl/2.0.14       15) libfabric/1.13.2
  8) XZ/5.2.5             16) PMIx/4.1.0

  Where:
   S:  Module is Sticky, requires --force to unload or purge

b-an01 $ mpicc f.c                          #Now the compiler is found
b-an01 $ ml -foss                           #Remove the compiler module
b-an01 $ ml

Currently Loaded Modules:
  1) snicenvironment  (S) 2) systemdefault  (S)

  Where:
    S:  Module is Sticky, requires --force to unload or purge
```

```
b-an01 $ mpicc f.c                          #Compiler unavailable again
-bash: mpicc: command not found
```

See more information on the HPC2N web, for example:
`https://www.uppmax.uu.se/support/user-guides/snowy-user-guide/`

A.4 Batch system

In general, jobs are run with a batch or queuing system. In order to start the job, a *job script* will usually be utilized. In short, it is a list of commands to the batch system, telling it things like: how many nodes to use, how many processors, how much memory, how long to run, program name, any input data, etc. You also include commands that would otherwise be entered at the command line, like changing directories. When the job has finished running, it will have produced a number of files, like output data, perhaps error messages, etc.

Since the jobs are queued in the system and will run when they are able to get a free spot, it is not generally a good idea to run programs that require any kind of user interaction, like some graphical programs.

Snowy run SLURM as batch scheduling system. SLURM has a system resource manager (allocates and enforces limits on nodes, processors, memory, etc.), and a job scheduler (handles job scheduling policies).

There is a set of batch system commands available to users for managing their jobs. The following is a list of commands useful to end-users:

sbatch <submit_file> submits a job to the batch system

... -M snowy always use this cluster specification to get information about Snowy

squeue shows the current job queue (grouped by running and then pending jobs)

squeue -u <username> shows only the jobs for <username>

scontrol show job <jobid> shows detailed information about a specific job

scancel <jobid> deletes a job from the queue

scancel -u <username> cancel all your jobs (running or pending)

Running jobs

If you have a small test job it is possible to run it on the login node just as you would on any other computer, but any larger and/or serious job (including when you want to time it) should be run using the batch system.

Job scripts

A set of computing tasks submitted to a batch system is called a job. To create a new job script (also called a submit script or a submit file) you need to:

- open a new file in a text editor (e.g., nano, emacs or vim)
- write a job script including batch system directives
- remember to load any modules needed (see links about modules above)
- save the file and submit it to the batch system queue using command sbatch

Here we will demonstrate the usage of the batch system directives on the following simple submit file example. Directives to the SLURM batch system is preceeded with #SBATCH. Comments are preceeded by #.

```
#!/bin/bash
# The name of the account you are running in, mandatory.
#SBATCH -A <account>
# Asking for two nodes
#SBATCH -N 2
# and two tasks
#SBATCH -n 2
# and two cores per task
#SBATCH -c 2
# Request 5 minutes of runtime for the job
#SBATCH -time=00:05:00
# Set the names for the error and output files
#SBATCH -error=job.#SBATCH -output=job.
mpirun ./my_program
```

- `-A` specifies the SNAC project ID, normally formatted as UPPMAXXXXX-YY-ZZ (**mandatory**).
- `-N` number of nodes. If this is not given, enough will be allocated to fullfill the requirements of -n and/or -c.
- `-n` specifies requested number of tasks. The default is one task.
- `-c` specifies requested number of cpus (actually cores) per task. Request that `n` cpus be allocated per task. This can be useful if the job is multi-threaded and requires more than one core per task for optimal performance. The default is one core per task.
- `--time=` is the real time (as opposed to the processor time) that should be reserved for the job. It is beneficial to set this value as accurately as possible since smaller jobs are more likely to fit into slots of unused space faster. Do not set it to small because then your job will be interrupted when the time runs out.

- `--output=` and `--error=` specify paths to the standard output and standard error files (the default is that both standard out and error are combined into `slurm-<jobid>.out` in the directory you are running in)
- By default your job's working directory is the directory you start the job in. If you want to change this, you can do a `cd` before the `mpirun` command.
- Note that if your program does not use MPI (just OpenMP/PThreads) you start your program without `srun/mpirun`. (i.e., the last line would look something like: `./my_program`). You should also just use one task, but with several cores per task.

More information about submit files and how to design them can be found on the UPPMAX web under *Systems and Support*, e.g.:
`https://www.uppmax.uu.se/support/user-guides/slurm-user-guide/`

A.5 Typical workflow

A typical workflow when using UPPMAX is like this:

1. Develop a parallel program somehow and somewhere.
2. Transfer the files to your home directory on UPPMAX, e.g., using the `scp` command. (Not needed if the development is done on UPPMAX.)
3. Compile, link, and debug your program on the login node. (Remember to first load necessary modules.)
4. Create a submit file for your job.
5. Submit a job to the batch system

Appendix B

Hello World

This appendix contains instructions on how to compile and run parallel programs written using MPI. Instructions are both for computers in the labs and for the cluster Snowy at UPPMAX.

B.1 MPI

B.1.1 Source code

Type the following code into a file named `hello-mpi.c`.

hello–mpi.c

```
1  #include <mpi.h>
2  #include <stdio.h>
3
4  int main(int argc, char *argv[])
5  {
6      MPI_Init(&argc, &argv);
7      int num_procs;
8      MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
9      int my_id;
10     MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
11     printf("Hello from process %d within a communicator of %d\n",
12            my_id, num_procs);
13     MPI_Finalize();
14 }
```

Line 1 includes the MPI library header. The call to `MPI_Init` on line 6 should be the first executable statement in the `main` function. Note how the command line arguments are passed to the function in order for the MPI library to process MPI-specific command line arguments. The call on line 8 gets the number of processes in the default communicator. The call on line 10 extracts the ID of the current process. Processes are numbered from zero. The call to `MPI_Finalize` shuts down MPI and no more MPI calls are allowed after this call.

B.1.2 Compilation

MPI programs are compiled using a compiler wrapper called `mpicc`. The purpose of this wrapper is to add appropriate flags and arguments to the compiler and linker. Under the hood your regular compiler is called and you can add arguments that your compiler understands. Compile your program like this:

```
mpicc -o hello-mpi hello-mpi.c
```

B.1.3 Execution

Since MPI programs are designed to be run on many separate computers (each with its own operating system), you can understand that executing an MPI program is not as trivial as running an ordinary sequential or multithreaded program. At the very least, the MPI program must be told which computer(s) to run on and how many processes to spawn. Use one or more of the following ways to run your MPI programs.

Running on CS lab computers

You can also run MPI programs on lab computers. The example below will run four processes on the same computer. In the manual for `mpirun` you can see how to run on more than one physical host machine.

```
scratchy$ mpicc -o hello-mpi hello-mpi.c
scratchy$ mpirun -n 4 ./hello-mpi
Hello from process 1 within a communicator of 4
Hello from process 2 within a communicator of 4
Hello from process 3 within a communicator of 4
Hello from process 0 within a communicator of 4
scratchy$
```

Running on Snowy

Before running and/or compiling a program on Snowy you have to load the right libraries and compilers. There are several possible combinations, but a common choise is to use the command `module add foss`. (You only have to do this once in each session.)

You can use the command `mpirun` to run an MPI program directly on the login node. Use the option `-n` to specify the number of copies of the program (MPI tasks) you want to run.

Note! You are **not** supposed to run large (many tasks) or long jobs on the login node (use the batch system) since the login node is shared by all users. Timing of jobs should also be done through the batch system.

```
b-an01 [~/PB1-22]$ ml foss
b-an01 [~/PB1-22]$ mpicc -o hello-mpi hello-mpi.c
b-an01 [~/PB1-22]$ interactive -A uppmax2024-2-9 -M snowy
-p node -n 4 -t 00:10:00
b-an01 [~/PB1-22]$ srun mpirun -n 4 ./hello-mpi
Hello from process 0 within a communicator of 4
Hello from process 2 within a communicator of 4
Hello from process 3 within a communicator of 4
Hello from process 1 within a communicator of 4
b-an01 [~/PB1-22]$
```

Running in batch mode on UPPMAX

Create the submit script using a text editor and submit it to the batch system. When the job is finished, you find the program output in the output-file. See sample session below.

```
b-an01 [~/PB1-22]$ ml foss
b-an01 [~/PB1-22]$ mpicc -o hello-mpi hello-mpi.c
b-an01 [~/PB1-22]$ cat submit-mpi
#!/bin/bash
#SBATCH -A hpc2n2022-002
#SBATCH -n 4
#SBATCH -time=00:05:00
#SBATCH -error=job.%J.err
#SBATCH -output=job.%J.out

ml foss
srun ./hello-mpi

b-an01 [~/PB1-22]$ sbatch -M snowy submit-mpi
Submitted batch job 41484
b-an01 [~/PB1-22]$ squeue -u mr
JOBID   PARTITION      NAME   USER  ST  TIME  NODES...
41484      single  submit-m     mr   R  0:01      1...
b-an01 [~/PB1-22]$ ls -l job*
-rw-r-r- 1 mr folk   0 Jan 21 13:21 job.41484.err
-rw-r-r- 1 mr folk 236 Jan 21 13:21 job.41484.out
b-an01 [~/PB1-22]$ cat job.41484.out
Hello from process 1 within a communicator of 4 (0.000004)
Hello from process 0 within a communicator of 4 (0.000009)
Hello from process 3 within a communicator of 4 (0.000005)
Hello from process 2 within a communicator of 4 (0.000003)
b-an01 [~/PB1-22]$
```