Chapter 47

A Style Guide to Project Submissions

Here are some guidelines for how to submit assignments and projects. As a general rule, consider programming as an experimental science, and your writeup as a report on some tests you have done: explain the problem you're addressing, your strategy, your results.

47.1 General approach

As a general rule, consider programming as an experimental science, and your writeup as a report on some tests you have done: explain the problem you're addressing, your strategy, your results.

Turn in a writeup in pdf form (Word and text documents are not acceptable) that was generated from a text processing program such (preferably) LaTeX. For a tutorial, see Tutorials book, section-15.

47.2 Style

Your report should obey the rules of proper English.

- Observing correct spelling and grammar goes without saying.
- Use full sentences.
- Try to avoid verbiage that is disparaging or otherwise inadvisable. The *Google developer documentation style guide* [12] is a great resource.

47.3 Structure of your writeup

47.3.1 Write as if it's an article

Consider this project writeup an opportunity to practice writing a scientific article.

Start with the obvious stuff.

- Your writeup should have a title. Not 'Project' or 'parallel programming', but something like 'Parallelization of Chronosynclastic Enfundibula in MPI'.
- Author and contact information. This differs per publication. Here it is: your name, EID, TACC username, and email.
- Introductory section that is high level: what is the problem, what did you do, what did you find.
- Conclusion: what do your findings mean, what are limitations, opportunities for future extensions.
- Bibliography.

TLDR

-
- Descriptive title. Not just 'Project'.
- Intro and conclusion.

47.3.2 Consider your audience

An article is written for a specific audience: a journal, a conference, or in this case: your instructors. So don't go into details that mean nothing to your audience, and try giving them what they find interesting.

In other words: give enough background on your application, but not too much. You're not writing for your thesis supervisor, you're writing to interested outsiders to your field.

On the other hand, your instructors know everything about parallelism. So don't show a differential equation and say 'and I made this parallel with OpenMP'. Go into detail how you translated your problem into something computational, and then show relevant bits of code.

That does not mean that turning in the code is sufficient, nor code plus sample output. Write an article.

TLDR Equal billing to

- science
- computation
- code

47.3.3 Observe, measure, hypothesize, deduce

Treat your project as if it is a scientific investigation of some phenomenon. Formulate hypotheses as to what you expect to observe, report on your observations, and draw conclusions.

Quite often your program will display unexpected behaviour. It is important to note this, and hypothesize what the reason might be for your observed behaviour.

In most applications of computing machinery we care about the efficiency with which we find the solution. Thus, make sure that you do measurements. In general, make observations that allow you to judge whether your program behaves the way you would expect it to.

TLDR Not just numbers: explain them.

47.3.4 Reporting

Your report should include both code snippets and graphs.

Screenshots of code snippets are not acceptable. Use at least a verbatim/monospace mode in your text processor, but better, use the LaTeX `listings` package or equivalent.

Graphs can be generated any number of ways. Kudos if you can figure out the LaTeX `tikz` package, but Matlab, gnuplot, Excel, or Google Sheets are all acceptable. Again: no screenshots.

For parallel runs you can, but are not required to, use TAU plots; see Tutorials book, section-18.

TLDR Graphs are better than tables are better than words.

47.3.5 Repository organization

If you submit your work through a repository have your pdf file at the top level; organize your sources in clearly named subdirectories. Object files and binaries should not be in a repository since they are dependent on hardware and things like compilers.

47.4 The parallel part

The parallelization part is the most important of your writeup. So don't write 3 pages about your application and 1 about the parallel code. Discuss in detail:

- What is the parallel structure of your problem? Relate the code structure to the application structure.
- Did you use MPI or OpenMP? Why?
- What kind of parallelism did you use? Mostly MPI collectives or point-to-point operations? OpenMP loop parallelism or tasks? Why?

TLDR Go into detail on parallelism!

47.4.1 Performance

The most important reason for parallel programming is to achieve faster performance.

To measure the efficiency of your code, run for a range of processor/core counts. Do you aim for strong or weak scaling?

Make sure your problem is large enough to overcome the overhead of parallelization. Can you run several problem sizes at a given core/process count?

How much speedup are you expecting for your application, given its structure? How much are you getting? If your speedup is low, reason where the problem might lie.

47.4.2 Running your code

A single run doesn't prove anything. For a good report, you need to run your code for more than one input dataset (if available) and in more than one processor configuration. When you choose problem a size, bear the following factors in mind.

- Parallelism introduces overhead, hundreds of cycles for an OpenMP barrier, or a few microseconds for an MPI message. So the amount of work in a parallel region, or between messages, probably needs to be in the thousands of operations.
- Various things in the system introduce random fluctuations, timings that are too small may be meaningless. As a rule of thumb, a timed sections needs to take at least on the order of a second.
- There are various startup phenomena in a parallel code, such as OpenMP thread creation, or allocation of dynamic memory. Make sure you only time the relevant bit of your code; if in doubt, put a loop around it to take multiple measurements, and use an average time.

When you run a code in parallel, beware that on clusters the behaviour of a parallel code will always be different between one node and multiple nodes. On a single node the MPI implementation is likely optimized to use the shared memory. This means that results obtained from a single node run will be unrepresentative. In fact, in timing and scaling tests you will often see a drop in (relative) performance going from one node to two. Therefore you need to run your code in a variety of scenarios, using more than one node.

TLDR

- Large enough parallelism
- Large enough problem

47.4.3 Graphs

In parallel programming, speedup and scaling are the test of how good your work is. So it's up to you to report this as well as you can.

The numbers are not the point of a graph: the point is the conclusions you can draw from them. Thus, if you do a scaling analysis, a graph reporting runtimes should make this point visible. In particular, do not use a linear time axis, as curved graphs are hard to read. Try to find a way to compare your results to a straight line, such as constant time, or linearly increasing speedup.

It is up to you to decide what quantity to report. This may depend on your application.

Use enough data points! Writing a short script to run your program multiple times takes very little time.

47.5 Remarks

47.5.1 Parallel performance or the lack thereof

In a perfect world, the performance of your code should grow with the number of available resources. If your program shows disappointing performance, consider the following.

Synchronizing OpenMP threads at the end of a parallel region takes maybe a few hundred cycles. This means that the amount of work in that region should be considerably more.

If your OpenMP program stops scaling at a certain core count, consider affinity settings; section 25.1.

MPI messages takes a couple of microseconds. Again, this implies that the amount of work between messages needs to be large enough.

47.5.2 Code formatting

Included code snippets should be readable. At a minimum you could indent the code correctly in an editor before you include it in a `verbatim` environment. (Screenshots of your terminal window are a decidedly suboptimal solution.) But it's better to use the `listing` package which formats your code, include syntax coloring. For instance,

```
\lstset{language=C++} % or Fortran or so
\begin{lstlisting}
for (int i=0; i<N; i++)
  s += 1;
\end{lstlisting}
```

With output:

```
for (int i=0; i<N; i++)
  s += 1;
```