Parallel Coursework 2

January 7, 2019

1 How to:

2 Implementation detail

The program begins with the main process parsing the input arguments whilst the other processes wait to recieve them. This was done to ensure that not all of the processes parse the same input arguments.

Then the main process reads the array from a given file into memory. Afterwhich, it begins its computation. When going through the computation for the first time, the main process allocates indexes and row counts for later.

The main process then sends all of the indicies and the data associated to each process which the indicies are assigned to. All of the chunks of data are sent at this point since it is the first time passing through. The main process also copies its allocated chunk from the read memory.

On second passing only the last row is sent to the next process. This is because each process will have 2 extra rows which will be updated each iteration. These will be recieved from the responsible thread. In this case the main thread is resposible for sending it to the next process, and likewise for the next process which will send its first row to the main thread, hence we recieve it.

During the loop, we need to process each chunk. The main thread goes through each element of the chunk and updates the array newRows from the values of oldRows accordingly. It then proceeds to calculate the precision, which is the difference from the previous value. If the precision calculated is larger than our allowed precision then a flag is set detailing this.

The main process then proceeds to collect the precision data from the other processes.

3 Testing

The program, as it is written does not output the result of the computation. This can however be changed by uncommenting the block of code around line 300 and compiling. This will output the matrix to a file. We compared this with our computation for our first parallel coursework giving us the same answers.

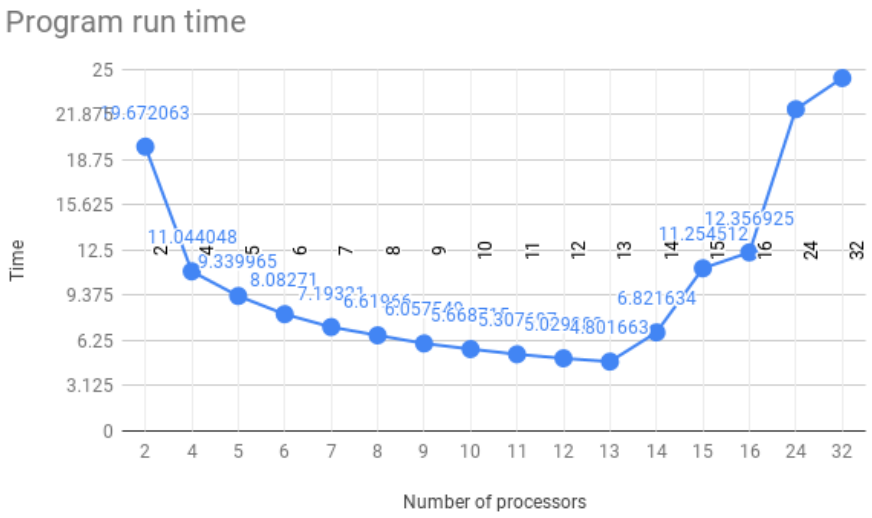
We could never, however, get the program to run on Balena. This is due to an error with slurm: `/cm/local/apps/slurm/var/spool/job1847022 ↪ /slurm_script: 'intel-lic': not a valid identifier`. The author is vastly disapointed with this result.

However against recommendation we run it on the main node in Balena anyway.

4 Speedup

Amdah's law says there will be a limit to speedup for any given number of processors. We ran the following command on the main Balena node: `mpirun ↪ -np 2 ./main -d 1000 -p 10E-4 array3`. Where the `-np` parameter is varied for various sizes of processor, starting with 2. Giving us the following data:

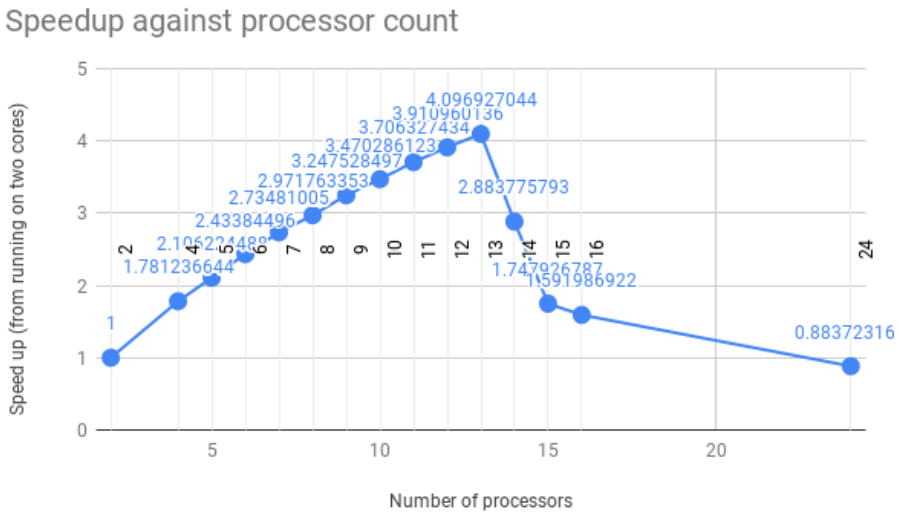
run time.png run time.png



We note that speed up will be proportional to 1/time therefore our greatest speed up is at 13 processors.

We could not get a time for a sequential run but we will pretend that 2 processors is the same as a sequential run and calcaulate the speed up from running on two processors.

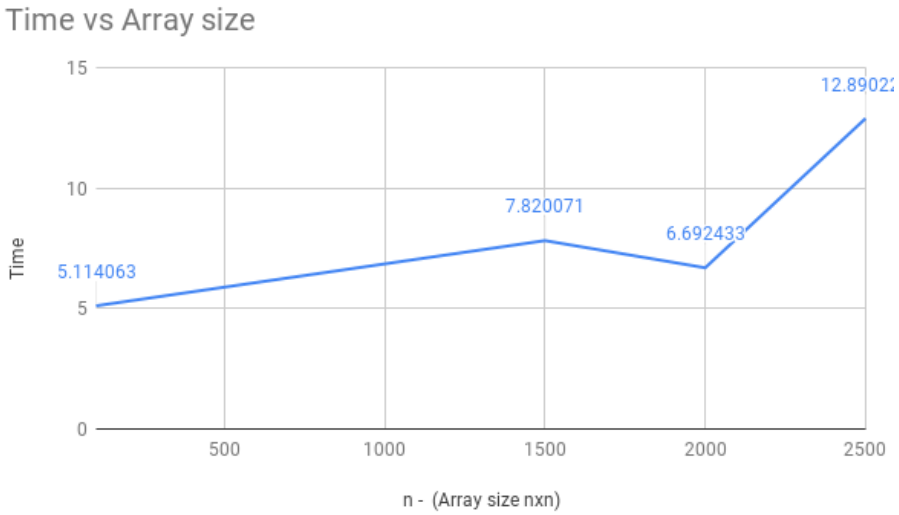
against processor count.png against processor count.png



As seen, we get an optimal speedup at around 8 processors. This satisfies Amdahls law by quite a bit, however is expected with this kind of program.

Using Gustafson's law we will see a better speedup on larger problems. We therefore proceed to collect data about different array sizes. We run the following programs `mpirun -np 13 ./main -d 500 -p 10E-4 array3`, where `-d 500` can be changed for various array sizes.

vs Array size.png vs Array size.png

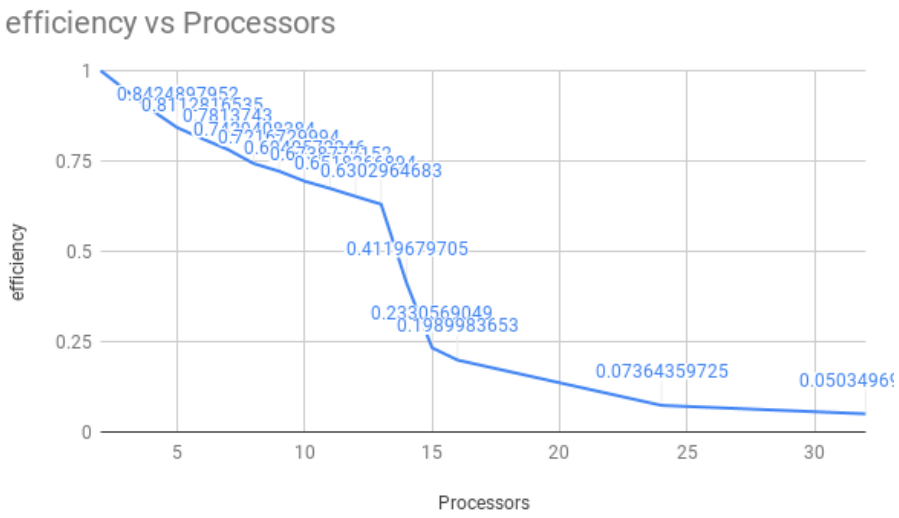


This provides good evidence that scaling the problem to very large matrix

sizes will be in good time. Given the linear regression found in the graph.

5 Efficiency

vs Processors.png vs Processors.png



Efficiency is the same story as speed up with acceptable efficiency before and at 13 precesses but it seems the overhead is too great at that point and the processors are doing a lot of switching. The efficiency will also increase with the problem size.

6 Scalibility

Our implementation has a good scalibility potential. As MPI is an open standard it is widely implemented on many different architectures of cluster. This means it should be very easy to get our program running on a large parallel cluster with many nodes.

7 Comments on development

During the development of the application, the program was plauged with bugs. It is common practice to use a low(ish) level language such as C (or fortran) for performance applications. We believe that this is not optimal. Development is slow and riddled with potholes, debugging is difficult and there is still no real assurance that the program will run correctly.

To reduce overhead the program was written in a compact way, using as little overhead C may provide as possible, such as function calls.

We believe that it is possible to write high-performance applications that are quicker to write, easier to debug and more importantly even prove are correct using much higher-level languages such as Haskell.

We do however not that Haskell compilers are not yet sophisticated enough to out perform C. However in some cases (especially sequential programs) Haskell has proven its strengths against C and run faster. Coupled with the side effects of being strongly typed and verifiable, this is an obvious bonus for intricate parallel applications where mistakes are easier to make. This also means that many more people can write these kinds of parallel applications without having to resort to C.