NORWEGIAN UNIVERSITY OF SCIENCE AND TECHNOLOGY
TDT4200 – PARALLEL COMPUTING

Problem Set 1: MPI Intro

Per Magnus Veierland
permve@stud.ntnu.no

September 3, 2015

PART 1: THEORY

PROBLEM 1: GENERAL THEORY

a) **Write a table explaining Flynn's Taxonomy.**

Flynn's taxonomy is a popular system used to categorize and describe computer architectures according to the number of instruction streams and data streams the system can operate on simultaneously.

- *Single instruction, single data-stream (SISD)* refers to a classical and commonly used computer architecture in which there is a single stream of instructions being executed by the processor on a single stream of data fetched from memory. Conceptually this means that the processor has a single control unit and a single Arithmetic Logic Unit (ALU).
- *Multiple instruction-stream, single data-stream (MISD)* refers to a computer architecture which conceptually has multiple control units and a single ALU, i.e. it performs multiple instructions on the same set of data. This is an uncommon and little used configuration, but can be used in systems built for fault-tolerance where instructions can be computed redundantly and the results compared against each other to verify correctness.

- *Single instruction-stream, multiple data-stream (SIMD)* refers to a computer architecture which conceptually has a single control unit and multiple ALUs. This means that the processor can execute single instructions which operate on multiple sets of data and can be found in almost all desktop processors with instruction sets additions such as Streaming SIMD Extensions (SSE).
- *Multiple instruction-stream, multiple data-stream (MIMD)* refers to a computer architecture which conceptually has multiple control units and multiple ALUs. This refers to full hardware parallelism where multiple instruction streams are executed concurrently with separate data streams. The two common types of MIMD systems are *shared-memory systems* in which multiple processor cores shares a common memory, and *distributed-memory systems* where multiple pairs of processors with individual memories communicate via some interconnect.

i) **Explain how, where, and why MPI fits into Flynn's Taxonomy, and why if not.**

The Message Passing Interface (MPI) library enables coordination of- and communication between separate programs running on separate data on separate computing nodes. In terms of Flynn's taxonomy this would make MPI a MIMD architecture assuming that the underlying system hardware is able to operate on multiple instruction- and data-stream simultaneously.

b) **Shared Memory**

i) **Explain how and why (and why if not), MPI fits with Shared Memory systems.**

MPI is not primarily used in pure shared-memory architectures as there are easier ways to parallelize programs in such systems via primitives and data exchange through shared memory. However when writing programs which must work in shared-memory and distributed-memory systems, or when writing programs which combine both architectures MPI is a natural tool which allows abstracting these differences and treating everything as processing nodes connected via message passing.

Alternatively it is also possible to treat each shared-memory architecture system as one MPI processing node which runs with efficient shared-memory constructs internally and uses message passing to communicate with other MPI processing nodes.

c) **Distributed Memory**

i) **Explain how and why (and why if not), MPI fits with Distributed Memory systems.**

MPI is primarily used in distributed-memory architectures which are linked by some interconnect such as Ethernet. This architecture suits MPI well since the hardware artifacts such as switches and how physical nodes are distributed is not important to the programmer, only that there are processing nodes which can communicate via message passing. A distributed-memory system suits MPI programming well since systems can easily be scaled up by adding new processing nodes without any change to the program.

PROBLEM 2: CODE THEORY

a) **Does your implementation have any communicational bottlenecks?**

i) **Briefly outline the communicational bottleneck(s), if any.**

When implemented naively with one master node and $P-1$ slave nodes each slave node will compute its result and send it to the master through a single send and a single receive. This results in the master having to perform $P-1$ sequential receive calls to receive the results from all slaves. This bottleneck will only get worse as the number of nodes increases and in addition to increasing the time it takes to compute the final result it wastes CPU time by having slave programs stall while attempting to perform a blocking send to the master node.

ii) **Is it possible to reduce the bottleneck, and if so how?**

This communicational bottleneck can be reduced by using tree-structured communication where the results are collected in branches towards the master through intermediary processing nodes.

1) **If so, outline an algorithm that will reduce the bottleneck.**

With a branching factor of 2, processing nodes with an ID which is not a factor of 2 could communicate their result to the nearest lower processing node which has an ID which is a factor of 2. This could be repeated and the summed results from nodes with an ID factorable by 2 could be passed onto processing nodes which are factorable by 4 etc., until all results has reached the master node. Different branching factors can also be used. This reduces the communication bottleneck greatly by reducing the number of receives the master node must perform to $\log(P)$ and makes it scale much better with an increase in the number of processing nodes.

b) **Does your implementation have any computational bottlenecks?**

i) **Briefly outline the computational bottleneck(s), if any.**

The computation task of the problem set is to compute the sum of a finite series of reciprocals of logarithms. The main components in computing this sum is a) a for-loop iterating the index, b) taking the logarithm of the index, c) a division calculating the reciprocal of the logarithm, d) adding the reciprocal of the logarithm to the sum. Among the elements of the total computation, benchmarking using Valgrind shows that more than 90% of the naive program is spent on calculating logarithms which makes it the bottleneck of the computation.

ii) **Is it possible to reduce the bottleneck(s)? If so, outline an algorithm/method that will reduce the bottleneck. A one line formula, if found, is acceptable (this is a hard question).**

There are several ways to significantly speed up the computation. Below are some of them listed.

- Since computing logarithms is expensive, reusing the results of computing them can provide great benefits by reducing the number of logarithm computations. The product rule of logarithms states that $\log_b(x * y) = \log_b(x) + \log_b(y)$. This rule means that by having the logarithms of the factors of a number, the logarithm of the number itself can be computed cheaply through additions. For any even number X it is then sufficient to calculate the logarithm of $X/2$ and sum it with the logarithm of two, and likewise for increasing factors. An elegant way to compute as few logarithms as possible would be to use the Sieve of Eratosthenes (normally used to find prime numbers). With a sieve table logarithms of factors could be used optimally to find the logarithm of any number. An efficient way to use this technique while also splitting the program to run on several computing nodes was not found during the time available for this problem set.

  A simpler algorithm was devised to reuse logarithms between a number X and its double $2X$ which will always share the factor 2. Starting with a range of numbers $[0, N]$ the range is split into 3: $[0, \frac{N}{4})$, $[\frac{N}{4}, \frac{N}{2})$, $[\frac{N}{2}, N)$. When calculating the logarithms for the numbers in the second range, the logarithms for all even numbers in the third range can be found at the same time since they will always share the factor two with the numbers in the second range. This algorithm can be applied recursively to the first range.

  It can easily be seen that this algorithm will reduce the number of computed logarithms in the third range by a factor of 2. Since it is applied recursively to the first range it can be shown that the number of logarithmic computations saved is given by:

$$\sum_{i=1}^{\infty} \left(\frac{1}{4}\right)^n = \frac{1}{3}$$

  This shows that reusing the logarithms for numbers sharing factors of two using this simple algorithm will reduce the total number of logarithms computed by $\frac{1}{3}$. Further improvements can be gained by reusing other common factors such as 3 and 5, but there may be limits to the benefits of reusing further factors as it also complicates the implementation in ways which can cause worsen performance.

  Modifying the naive implementation to reuse logarithms for factors of two improved the running time from 3345391595 to 2364673326 nanoseconds which is an improvement of 29.32%.

- Computing natural logarithms is more expensive than computing logarithms with base 2. It is shown below that calculating the binary logarithm of e divided by the binary logarithm of a number gives the same result as taking the reciprocal of the natural logarithm of the number. The average cost of computing one natural logarithm is 36 ns and the average cost of computing one binary logarithm is 28 ns. Since the value of $\log_2(e)$ can be precomputed once the change from computing natural logarithms to binary logarithms provides a speedup. When changing the naive implementation to use binary logarithms instead of natural logarithms the program went from spending 90.94% of its running time on the logarithm function to spending 82.85% of its running time on the logarithm function.

$$\frac{1}{\log_e(x)} = \frac{\log_e(e)}{\log_e(x)} = \frac{\log_2(e)}{\log_2(x)}$$

NB! The original measurements showing a difference between the cost of computing natural and binary logarithms were only tested on one libc implementation on my laptop. After retesting the same benchmark with a different implementation on Vilje I found that the average cost of computing one natural logarithm is 24.8 ns while the cost of computing one binary logarithm is 25.9 ns. This result is left as a reminder of the importance of benchmarking on the relevant architecture, and that because some implementations might have different costs for different logarithm bases it is important to benchmark the difference since converting between bases can be cheap.

- The for-loop can be unrolled manually for an improvement in running time without complicating the program significantly. Unrolling the loop such that 8 terms are calculated per iteration instead of 1 reduced the running time of the naive implementation from 3408874911 to 3370192981 nanoseconds with $start = 2$ and $stop = 100000002$ which is an improvement of 1.13%. After changing the naive implementation to use a binary logarithm instead of using a natural logarithm the program spend less time computing the logarithm and more time in the main program loop. Applying both the change from natural to binary logarithms and unrolling the loop by 8 reduced the running time from 3435538199 to 2945741739 nanoseconds which is an improvement of 14.26%. Further small improvements could likely be made by spending more time tuning the unrolling.

PART 2: CODE

PROBLEM 1: MPI INTRO

a) **In the MPI copy `computeMPI.c`, parallelize the serial program from `computeSerial.c` using MPI.**

I've provided a C++ implementation based on *Boost.MPI* in the file `computeMPI.cpp`.

b) **Implement a "make all" Makefile-rule which compiles and executes corresponding MPI file/executable in the given Makefile.**

I've provided a `CMakeLists.txt` file which can compile the program using:

```
$ cmake . && make
```

c) **Analyze your implementation and report the following:**

i) **The amount of operations performed by your program as a function of $O(n)$, $n = stop - start$, and the amount of operations performed per process P_i in your program with $i \in [1, 10]$.**

The amount of logarithms computed for the program is given as:

$TotalAmountOfOperations = \frac{2N}{3}$

The amount of logarithms computed per node is given as:

$TotalAmountOfOperationsPerNode = \frac{2N}{3P}$

ii) **The amount of MPI operations performed by your program as a function of $O(P)$, $P = NumberOfProcesses$.**

Due to a limited amount of time for this problem set I originally implemented the communication naively without a tree-structure and did not have time to improve the implementation.

The program is split into one master process communicating with a set of slave processes. Each slave process will perform its computation based on the command line inputs and send one result to the master process using `MPI_Send`. The number of slave processes is $P - 1$.

The master process will perform its own computation based on the command line inputs and then receive one result from each slave process using `MPI_Recv`.

In total, the number of MPI operations performed by the program will be two times the number of slave processes; since each `MPI_Send` from each slave process must be matched by one `MPI_Recv` by the master process.

$$\mathit{NumberOfMpiOperations} = 2 \times \mathit{NumberOfSlaveProcesses} = 2 \times (P - 1) = 2P - 2$$

iii) **The average amount of MPI operations performed per process P_i in your program with $O(P)$, $P = NumberOfProcesses$.**

$\mathit{AverageNumberOfMpiOperations} = \frac{2P-2}{P}$

iv) **The maximum number of MPI operations performed by any process P_i as a function of $O(P)$, $P = NumberOfProcesses$.**

The maximum number of MPI operations performed by any process will be by the master node which performs one `MPI_Recv` per slave node.

$\mathit{MaximumNumberOfMpiOperationsAnyProcess} = P - 1$

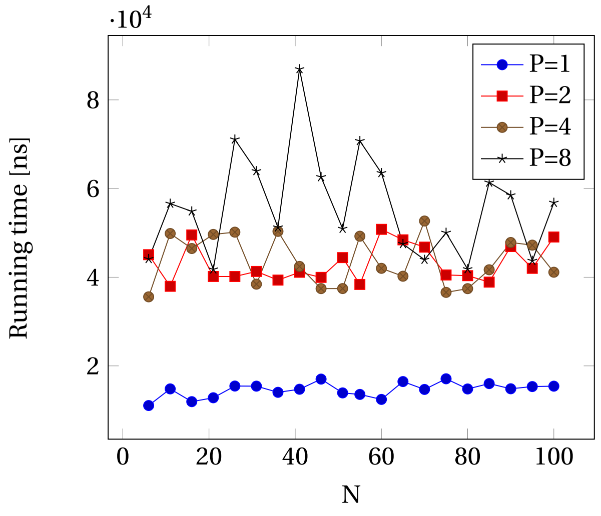
v) **The difference in run-time with $P \in [1, 2, 4, 8]$ and $n = [2 \rightarrow 100, 2 \rightarrow 10^6, 2 \rightarrow 10^9]$ when run on a machine in ITS015.**

Make a graph for each value of P, showing the run-time for different increments of n. Each range of n should be split into 20 steps of equal increments.

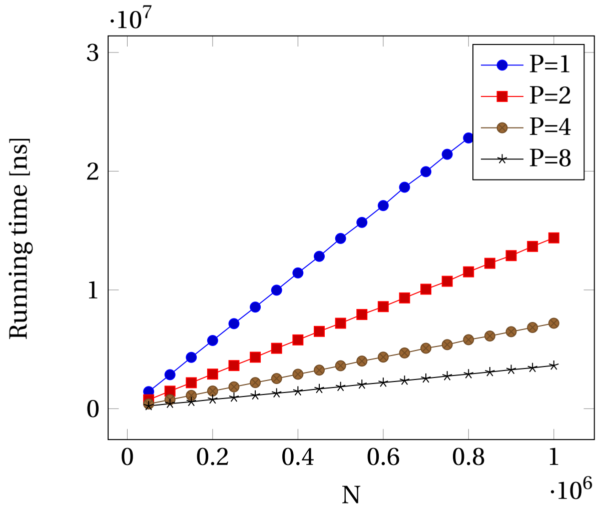
Since the ITS015 lab machines did not have Boost installed at the time I did the following runs on Vilje with the following modules loaded:

- `mpt/2.06`
- `cmake/2.8.11.2`
- `intelcomp/14.0.1`
- `boost/1.55.0`
- `gcc/4.8.2`

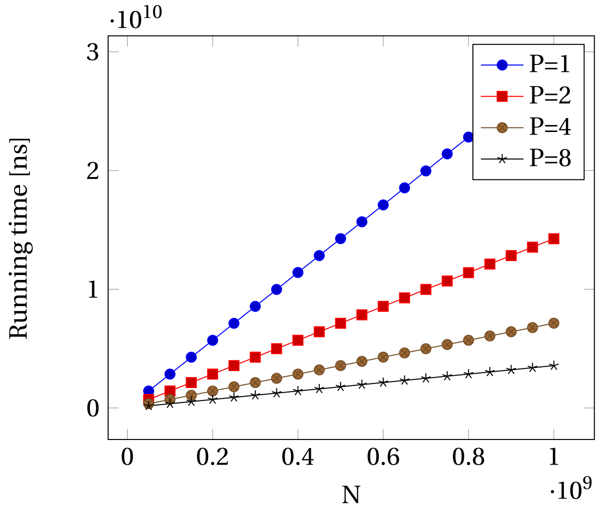
NB: Note that the GCC module is needed as the Intel compiler depends on GCC for the C++ standard library.



·10^4
8
6
4
2
Running time [ns]
0
20
40
60
80
100
N
P=1
P=2
P=4
P=8



·10^7
3
2
1
0
Running time [ns]
0
0.2
0.4
0.6
0.8
1
·10^6
N
P=1
P=2
P=4
P=8



·10^10
3
2
1
0
Running time [ns]
0
0.2
0.4
0.6
0.8
1
·10^9
N
P=1
P=2
P=4
P=8