

TDT4200 Parallel Computing

Problem Set 2

Stian Hvatum (hvatum)
MTDT

September 6, 2012

Contents

Task 1	Miscellaneous theory	2
a)	SISD, SIMD, MISD & MIMD	2
b)	MPI_Pack / Unpack	2
c)	MPI_Sendrecv	3
d)	Deadlocks	3
Task 2	Amdahl's law and multicore	3
Task 3	Communication in a Cartesian grid	3
a)	Strip partitioning	4
b)	Block partitioning	4
c)	8x2	4
d)	4x4	4
Task 4	Image processing	4
A	Code for Task 2	5

Task 1 Miscellaneous theory

a) SISD, SIMD, MISD & MIMD

SISD

Single instruction single data is the traditional sequential, single threaded program, running one instruction at a time, on a single set of data.

SIMD

Single instruction multiple data is a parallel program where the same instruction gets applied to multiple data(sets), eg. parallel image processing or solving linear equation sets in parallel. This is typical in GPUs and array/vector-based processors.

MISD

Multiple instructions single data is a parallel program where different instructions gets applied to the same dataset. Multithreaded programming can be seen as an example of this, since the same memory space has multiple instructions operating on it. Another example is math where operands are equal but the operations are different.

MIMD

Multiple instructions multiple data is a parallel program where different instructions are working on different data. This is the type of programs that can run on a cluster, since the different processes on the cluster do not actually share memory space, neither share program counter, and may branch independently and differently from each other.

b) MPI_Pack / Unpack

The MPI_Pack and MPI_Unpack functions allow packing uncontinuous data before sending, and restoring it to its original (or other convenient) state after receiving. Normally you can define your own MPI_Datatype to handle this, but MPI allows some flexibility by providing both solutions. MPI_Pack and MPI_Unpack also provide backwards-compatibility with older versions of MPI and provides functions might be valuable for a library writer. MPI_Unpack also waits for the number of elements specified, while MPI_Recv receives as much as is available and then returns.¹

¹<http://www.mpi-forum.org/docs/mpi-11-html/node62.html>

c) MPI_Sendrecv

If two or more MPI-processes are both sending and receiving at some point in the program, we have a scenario where deadlocks may rise. If we use MPI_Sendrecv, we are certain that no deadlocks will occur, while achieving the same communication. Since synchronisation is also possibly better handled than consecutive calls to MPI_Send and MPI_Recv, performance might also be better.²

d) Deadlocks

1 | \ \ TODO

Task 2 Amdahl's law and multicore

If we apply Amdahl's law to a heterogeneous machine where R out of N BCEs are used to build a single large core, while N - R BCEs are used for small cores. In this context, Amdahl's law states that speedup to a program where a fraction, f, can be parallelized, can be stated as

$$\text{Speedup}_{\text{asymmetric}}(f, n, r) = \frac{1}{\frac{1-f}{\text{perf}(r)} + \frac{f}{\text{perf}(r)+n-r}} \quad (1)$$

To find what R that gives the highest speedup, I wrote a small C-program³ that uses a brute force method to find the R-value where Eq. (1) is highest. Source is bundled with this report, compile with -lm.

The output of the program is:

```
1 | Result 0: Max. perf. @ R=678, speedup is 101.716278
2 | Result 1: Max. perf. @ R=835, speedup is 51.025921
3 | Result 2: Max. perf. @ R=748, speedup is 40.259754
4 | Result 3: Max. perf. @ R=875, speedup is 18.041071
```

Where Result 0 is a), 1 is b), 2 is c) and 3 is d).

²http://mpi.deino.net/mpi_functions/MPI_Sendrecv.html

³Appendix A

Task 3 Communication in a Cartesian grid

We have a 1024x1024 grid of bytes and 16 processors in a grid. Communication occurs at 1000 bytes / second with a startup time of 50 ms. Communication is full duplex, and communication is with one other processor at a time, but multiple pairs of processors can communicate at the same time.

The communication time for border exchange will be:

- a) Strip partitioning**
- b) Block partitioning**
- c) 8x2**
- d) 4x4**

Task 4 Image processing

This task is delivered as *hvatum_code.tar.gz*

A Code for Task 2

```
1  /*
2  * Written by Stian Hvatum, 6. sept. 2012
3  *
4  * This file is an implementation of Amdahl's law as an optimization problem.
5  * Parameters are given as stated in Task 2, Problem Set 2, TDT4200 Fall 2012.
6  *
7  */
8
9
10 #include <stdio.h>
11 #include <math.h>
12
13 float speedup(float f,int n,int r,float (*perf)(int))
14 {
15     return (1 / (((1-f)/(*perf)(r)) + f/((*perf)(r)+n-r)));
16 }
17
18 float perfA(int r)
19 {
20     return sqrt(r);
21 }
22
23 float perfB(int r)
24 {
25     return pow(r,1.0f/3.0f);
26 }
27
28 int main()
29 {
30     float res[4];
31     float max[4];
32     int ind[4];
33
34     int i = 0;
35     int j = 0;
36
37     for (i = 0; i < 4; i++)
38     {
39         res[i] = 0;
40         max[i] = 0;
41         ind[i] = 0;
42     }
43
44     for (i = 0; i < 1024; i++)
45     {
46         res[0] = speedup(0.8, 1024, i, &perfA);
47         res[1] = speedup(0.5, 1024, i, &perfA);
48         res[2] = speedup(0.8, 1024, i, &perfB);
49         res[3] = speedup(0.5, 1024, i, &perfB);
50
51         for (j = 0; j < 4; j++)
52         {
53             if (res[j] > max[j])
54             {
55                 max[j] = res[j];
56                 ind[j] = i;
57             }
58         }
59     }
60
61     for (i = 0; i < 4; i++)
62     {
63         printf("Result %d: Max. perf. @ R=%d, speedup is %f\n", i, ind[i], max[i]);
64     }
65 }
```