TDT4200 Parallel Computing
Problem Set 2

Stian Hvatum (hvatum)
MTDT

September 7, 2012

Contents

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. parallell image processing or solving linear equation sets in parallell. 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 programing can be sees as an example of this, since the same memory space has multiple instructions operating on it. An other 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 does not acctually share memory space, neither share program counter, and may branch independently and different from each other.

b) MPI_Pack / Unpack

The MIP_Pack and MPI_Unpack functions allows packing uncontinous data before sending, and restoring it to its original (or other convinient) state after receiving. Normally you can define your own MPI_Datatype to handle this, but MPI allows some flexibilty by providing both solutions. MPI_Pack and MPI_Unpack does also provide backwards-compitability with older versions of MPI and provides functions might valueable for a library writer. MPI_Unpack also waits for the number of elemets 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 syncronisation is also possibly better handled than consecutive calls to MPI_Send and MPI_Recv, performance might also be better.[2]

d) Deadlocks

```
\\ TODO
```

Task 2 Amdahl‘s law and multicore

If we apply Amdahl’s law to a hetrogenous 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

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

To find what R that gives the highest speedup, I wrote a small C-program[3] 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:

```
Result 0: Max. perf. @ R=678, speedup is 101.716278
Result 1: Max. perf. @ R=835, speedup is 51.025921
Result 2: Max. perf. @ R=748, speedup is 40.259754
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).

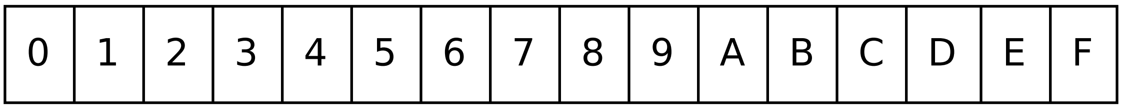
[2] http://mpi.deino.net/mpi_functions/MPI_Sendrecv.html
[3] Appendix A

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, which is 1b/ms 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

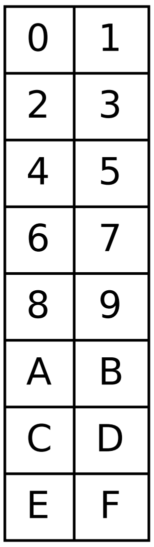


When we have a strip partitioning, each processor will "own" 1024/16 = 64 columns and 1024/1 = 1024 rows with bytes. This means that no communication is needed columnwise, and we have two neighbours that we must transfer our edges to.

What	Duration	#	Sum
Startup	50 ms	2	100 ms
Transfer	1024 ms	2	2048 ms
Sum			2148 ms

b) 8x2

In a 8x2 grid, we must have communication both ways. Each processor must communicate twice vertically and once horizontally. The byte array split across this processor grid gives us 1024/8 = 128 bytes on each row and 1024/2 = 512 bytes at each column. Each processor must send 512 bytes either east or west, and 128 bytes north AND south.



What	Duration	#	Sum
Startup	50 ms	3	150 ms
Transfer	512 ms	1	512 ms
Transfer	128 ms	2	256 ms
Sum			918 ms

c) 4x4

In a 4x4 grid, we have to communicate twice in both vertical and horizontal direction. We have 1024/4 = 256 bytes on both rows and columns, and therefor needs to send and trancieve 256 bytes to 4 neighbours.

What	Duration	#	Sum
Startup	50 ms	4	200 ms
Transfer	256 ms	4	1024 ms
Sum			1224 ms

0	1	2	3
4	5	6	7
8	9	A	B
C	D	E	F

Task 4 Image processing

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

A Code for Task 2

```c
/*
 * Written by Stian Hvatum, 6. sept. 2012
 *
 * This file is an implementation of Amdahl's law as an optimization problem.
 *↩
 Parameters are given as statet in Task 2, Problem Set 2, TDT4200 Fall 2012.
 *
 */


#include <stdio.h>
#include <math.h>

float speedup(float f,int n,int r,float (*perf)(int))
{
    return (1 / (((1-f)/(*perf)(r)) + f/((*perf)(r)+n-r)));
}

float perfA(int r)
{
    return sqrt(r);
}

float perfB(int r)
{
    return pow(r,1.0f/3.0f);
}

int main()
{
    float res[4];
    float max[4];
    int ind[4];

    int i = 0;
    int j = 0;

    for (i = 0; i < 4; i++)
    {
    res[i] = 0;
    max[i] = 0;
    ind[i] = 0;
    }

    for (i = 0; i < 1024; i++)
    {
      res[0] = speedup(0.8, 1024, i, &perfA);
      res[1] = speedup(0.5, 1024, i, &perfA);
      res[2] = speedup(0.8, 1024, i, &perfB);
      res[3] = speedup(0.5, 1024, i, &perfB);

      for (j = 0; j < 4; j++)
      {
    if (res[j] > max[j])
    {
        max[j] = res[j];
        ind[j] = i;
    }
      }
    }

    for (i = 0; i < 4; i++)
    {
    printf↩
("Result %d: Max. perf. @ R=%d, speedup is %f\n", i, ind[i], max[i]);
    }
}
```