Umeå University
Department of Computing Science

Parallel Programming 7.5 p
5DV152

Exercises, Chapter/Topic 1

Submitted	2017-01-26
Author:	Lorenz Gerber (dv15lgr@cs.umu.se lozger03@student.umu.se)
Instructor:	Lars Karlsson / Mikael Ränner

Contents

1 Introduction

This report is part of the mandatory coursework. It describes the solutions for several chosen exercises from the course book [**?**].

2 1.1 - Formulas for block partitioning

The overwhelming idea is to load balance `p` number of cores with `n` tasks. Here, we use two functions to obtain block partitioning using a `for loop` :

```
for (my_i = my_first_i; my_i < my_last_i; my_i++)
```

The functions `my_first_i` and `my_last_i` are used to set the limits in the loop. Besides `n, i and p` we also need an index for the actual core: `k`. It is understood that indicies `i` and `k` start at `0`. The book text hints to start with the case when `n` is evenly divisible by `p`:

```
my_first_i = k * n / p
my_last_i = (k + 1) * n / p
```

Testing this expression for `n = 10, p = 5, k = {0, 1, ..., 4}` seems to be correct. Now when `n` is not even divisible by `p`, one has to distribute the `n mod p` tasks for example on the first `n mod p` cores:

```
my_first_i = k * n / p + ( k < n mod p ? k : n mod p )
my_last_i = (k + 1) * n / p + (k + 1 < n mod p ? k + 1 : n mod p)
```

Testing this expression for `n = 9, p = 5, k = {0, 1, ..., 4}` gives the correct results.

3 1.2 - Modify 1.1 with non-uniform costs

The calls happen in parallel. It can be still assumed that `k = 0` will get the first call, `k = 1` the second and so on. However, this doesn't really matter as the processing time increases monotonously. Hence the solution in *1.1* will still provide the correct solution.

4 1.3 - Tree-structured global sum

```
divisor = 2
core_diff = 1

while(core_diff < p){
    if(k % divisor == 0)
        receive and add from k + core_diff
    else
        send to k - core_diff
        break

    divisor * 2
    core_diff * 2
}
```

```
if( k == 0 )
    return final result
```

5 1.4 - Alternative algorithm for 1.3

```
counter = 2
bitmask = 1

while ( counter < p ){
    if(k bitwiseXor bitmask > k)
        receive and add from k bitwiseXor bitmask
    else
        send to k bitwiseXor bitmask
        break

    bitwiseLeft bitmask
    counter * 2
}

if( k == 0)
    return final result
```

6 1.5 - Generalization of 1.3 and 1.4

6.1 Generalized form or 1.3

```
divisor = 2
core_diff = 1

while(core_diff < p){
    if(k % divisor == 0)
        if (k + core_diff < p)
            receive and add from k + core_diff
    else
        send to k - core_diff
        break

    divisor * 2
    core_diff * 2
}

if( k == 0 )
    return final result
```

6.2 Generalized form of 1.4

```
counter = 2
bitmask = 1

while ( counter < p ){
    if(k bitwiseXor bitmask > k)
        if (k bitwiseXor bitmask < p)
            receive and add from k bitwiseXor bitmask
    else
        send to k bitwiseXor bitmask
        break

    bitwiseLeft bitmask
    counter * 2
}

if( k == 0)
    return final result
```

7 1.6 - Cost anlaysis of global sum algorithms

The number of receives and additions for core 0 is in the original ‘pseudo-code global sum’ `p-1` and in the tree-structured global sum `ceiling(log2(p))`.