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 [1].

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 indices `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

Here the answers for exercise 1.1 have to be modified for the case that each call to `i` takes 2ms seconds longer starting at $i_0 = 2ms$, $i_1 = 4ms$, $i_2 = 6ms$, etc. Hence new formulas for `my_first_i` and `my_last_i` have to be derived.

Here the total elapsed time $N = n^2 + n$ (where `n` is the number of tasks) was used. In a similar way, any intermediate point can be expressed as $I = i^2 + i$ (where `i` is the index of `n`). The average time that every core should be busy, can be described as the total time elapsed divided by number of cores: $M\frac{N}{p}$. Now one can devise an equality for intermediate `i` values: $i^2 + i = k * M$ where `k` is the core index. This is a quadratic equation that can be solved for `i` in $\mathbb{N}$ (hence, results should be rounded to the closest integer value). The obtained values are the breaks between a sequence of tasks.

- For `my_last_i`, the above expression is modified to $i^2 + i = (k+1)M$
- $my\_last\_i(i_q)$ for $q = 0$ is by definition `0`, and for $i_q, q > 0$ it is $my\_last\_i(i_{q-1}) + 1$.

For practical usage, the above formula has to be reformulated to have a single occurrence of `i` on one side of the equation:

$$i = \frac{\sqrt{4kn^2 + 4kn + 4n^2 + 4n + p}\sqrt{p} - p}{2p}$$

4 1.3 - Tree-structured global sum

The aim was to write pseudo code that calculates the tree structured global sum described on page 5. The book hints to use a variable called `divisor` that is initialized with the value `2` and another variable called `core_difference` that is initialized with the value `1`. It was proposed that `divisor` is doubled in each iteration and that `n mod divisor` is used to determine `send = 0` and `receive = 1`. From figure 1.1 in the book, one can see that this rule works. Note especially that core `k = 0` will read in each iteration as `0 mod x = 0` is true for any `x`. Further, the book proposes that `core_diff`, when doubled in each iteration, can be used to describe the difference in value between a core pair that is involved in a 'send/receive' operation. The correctness of this can also be verified in figure 1.1 of the book.

Assuming that `k` is the core index, this allows already to write the main part of the algorithm:

```
if( k % divisor == 0 )
    receive and add from k + core_diff
else
    send to k - core_diff
    break
```

From figure 1.1, it can also be seen that `send` is the last operation a core does. Afterwards, it terminates. This is solved here with a break statement after send. Further, there need to be some control structure to know when to finish. Here it was decided to use a `while` loop with the comparison `core_diff < p`, where `p` is the number of cores in the system. In each iteration `divisor` and `core_diff` are doubled. Moreover, before exit, the core `k = 0` should return the result. Below is the complete code listing that incorporates the the described features:

```
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

Here the aim was to modify the pseudo code from 1.3 using bit-wise operators, basically to obtain `k` indices for `send` and `receive`. In the book, the idea is visually described using a table (page 13, exercise 1.4). The bit-wise `Xor` operator is applied to a binary `bitmask` with the initial value 001_2 and the binary value of each `k`. This will flip the last bit in every `k` value, resulting in two `k`'s exchanging their initial value with each other. This can be exploited to use the core `k` whose value becomes lower by the bit-shift as the `sender` and the other one in the 'flipping pair' as `receiver`. After the first iteration, the `bitmask` is bit-shifted to the left, hence 001_2 becomes 010_2. Now follows the next iteration round with applying `Xor` to all remaining `k`'s.

Here the value of the bitmask itself can be used in a `while` loop to determine the end of the iteration using the comparison: `bitmask =< p`. Also in this version, after the while loop the core `k = 0` needs to return the result.

```
bitmask = 1

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

    bitwiseLeft bitmask
}

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

6 1.5 - Generalization of 1.3 and 1.4

Here it was required to modify 1.3 and 1.4 to a more generalized form that could also handle the case when `p` is not a power of two value. Basically the same code was used just an additional `if` control sequence around the 'receive' statement that handles the case when there is no neighbor in the graph that could send the result. This situation is verified by comparing `k` with `p`.

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

A similar approach is possible to modify the bitwise-operation-global-sum. Instead of `k`, the resulting value of `k bitwiseXor bitmask` is compared to `p` for deciding if a receive operation needs to take place.

```
bitmask = 1

while ( bitmask =< 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
}

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

7 1.6 - Cost analysis of global sum algorithms

In the original ‘global sum’ code core 0 needs to receive results from each core except himself, hence the result is:

$$p-1$$

The tree structured global sum represents an inverted binary tree. The number of additions to core zero can be compared to the height of the binary tree and the number of cores as it’s leafs. The relation between leafs l and tree height h in a *perfect, full* binary tree can be expressed as $l = 2^h$, hence isolating h, the formula can be expressed as $h = log(l)$. In our examples, the number of additions to the zero-th core is discrete, hence the relation can be expressed as:

$$\lceil log_2(p) \rceil$$

References

[1] P.S. Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufman, 2011.