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 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

In my opinion, this question can be understood in several different ways:

3.1 First attempt

Let there be 10 tasks (`n`) that take `2, 4, ..., 20 ms` to finish, and there are 3 cores (`p`). So one could try to distribute the tasks onto the cores that the standard deviation of total runtimes for all cores is as little as possible. A possible solution for the above described actual setting would be:
i_9 to k_0, i_8 to k_1, i_7 to k_2. Then, i_0 to k_0, i_1 to k_1, i_2 to k_2. And i_6 to k_0, i_5 to k_1, i_4 to k_2. Finally, i_3 to k_0. This somehow works, makes sense and I guess it would be possible to devise a general formula for this.
However, it does not adhere to the actual question: *How would you change your answer to the preceding question if ....* And the preceding question is: *Devise formulas for the functions that calculate 'my_first_i' and 'my_last_i' in the global sum example.* Hence, to make formulas for first `i` and last `i` the tasks should be assigned to the cores in a consecutive sequence. Therefore, second attempt:

3.2 Second attempt

If the tasks have to be in a consecutive order, formulas for `my_first_i` and `my_last_i` have to be derived.

If the tasks shall be processed in order, one can express the total elapsed time for a sequential solution as $N = n^2 + n$ (where `n` is the number of tasks) and simlar at any intermediate point as $I = i^2 + i$ (where `i` is the index of `n`). The average time that every core should be busy, can be expressed 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 occurence of `i` on one side of the equation:

$$i = \frac{\sqrt{4kn^2 + 4kn + 4n^2 + 4n + p}\sqrt{p} - p}{2p} \tag{1}$$

However, this solution seems to me not very meaningful as it produces longer run-times than the solution in the first attempt. Therefore I tried to look again more detailed into how the question is formulated and came up with:

3.3 Third attempt

The question states `i = k` requires `k + 1` times as much time than call `i = 0`. Somehow this does not make sense. Because in this formula, to define the time, there need to be equal indices k as indices i. That would mean that the problem is by definition embarrassingly parallel.

From the wording of the question in the second part *the first **call** takes..., the second **call** requires...* etc., I conclude that the author means that every time the algorithm is executed it will take inherently (in the current case) 2 milliseconds longer to finish. Maybe it should not be `k`, as index for cores, but just an arbitrary index to construct a number series. This brings me to the fourth and last idea:

3.4 Fourth attempt

If every call to the algorithm takes inherently longer, the task's `i` are actually independent of the time it takes to run and one can not start executing the 'long running' `i`'s as described in attempt 1 and 2. In this case, it's obvious that the solution for 1.1 is also valid here.

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 can terminate. 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 to use bit-wise operaters, 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 bitwise `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 eachother. This can be exploited to use the core `k` whose value becomes lower by the bitshift 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 could be maintained with just an additional `if` control sequence around the 'receive' statement that handles the case when there is no neighbour in the graph that could send the result.

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

```
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 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))`.

References

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