SPM Project Report
Jacobi (AJ)

Gianmaria di Rito

August 2022

1 Process leading to the final implementation

1.1 Sequential version and consideration on the problem

Jacobi's method is an iterative method that attempts to give a solution to the linear system Ax = b in which for each iteration a more accurate approximation of the vector x is calculated. This method can be transposed in code like this:

```
for(#iterations){
    for(i=0, i < n, i++){ //compute x_i
        for(j=0, j < n, j++){
            sum += (A[i][j] * xprev[j]);
        }
        x[i] = (1 / A[i][i]) * (b[i] - sum);
    }
}
```

For the purpose of implementing a sequential solution as close to optimal as possible, two iterations of the method can be performed for each step of the main loop to perform a kind of forced unroll of the for that might bring some improvement in terms of time. The same optimization will probably also be done by the compiler with the **-O3** flag.

Starting from this basis, first, let us try to identify the types of patterns that could help us in paralleling Jacobi's method. The first consideration is to figure out whether we are facing a problem that can be solved with Stream Parallel Patterns or Data Parallel Patterns. Since we have no data stream and the data needed for computation are all available at the same time, it is easy to see that we should choose patterns of the Data Parallel type. In particular, in this group, in my opinion the pattern that best represents the problem we face is the **Stencil** since features such as the various iterations and the fact that each of these, depends on the previous one with each x_i value depending not only on its neighbors in the array but on the entire previous array.The next

step then is to figure out how to decompose the data needed for each iteration so the matrix **A**, the vector **xprev** of the previous solution, and the vector **b**. Looking at the computation to be performed we can see that, the computation of each element x_i of the new solution, depends on:

- the row i of the matrix **A**,
- the whole previous solution **xprev**,
- the single element b_i of the vector **b**.

During an iteration, each element x_i of the solution vector **x** can be computed independently by a worker who will not need to synchronize in any way with any other worker to read the shared data and to write the new element of the solution vector to the location i.

The only synchronization between the different workers will be there only at the end of an iteration of the method, since each solution depends on the previous one.

Moreover, Being a data parallel pattern type problem we know that the computation will depend:

- on the time required to split the data in the initial collection
- on the time for the actual computation
- on the time for merging the data into the resulting collection

the first and third, however, easily become negligible assigning static subsets of **x** to the various workers and by working directly on the pointers of the solution vector **x**.

Looking instead, at the type of computations to be performed, being always the same, will help an easy division of the workload over the different workers. In the end being the data needed for computation of a fixed size there will be no need for a resource allocation in terms of workers used, of a dynamic type.

1.2 different alternatives considered

Considering the above, the solutions I explored relative to implementation with native C++ threads and then with FastFlow are the following.

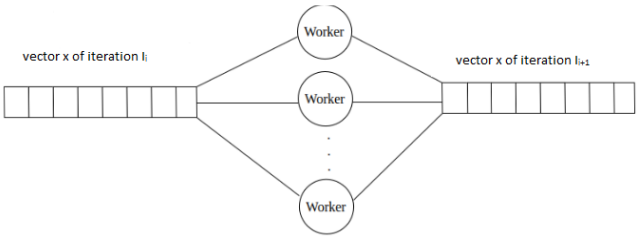
1. In the first implementation related to the native C++ threads, I split the computation of the iterations of the solution vector **x**, going to assign the different elements x_i to workers using a static split policy of type **cyclic** that will go to assign in almost equal parts the workload on them. Each thread then, will perform all iterations of Jacobi's method on a small subset of the vector **x** during which, it is going to write the next approximation of these elements on the shared vector **x**. This solution may be acceptable since as the computations are similar from the point of view of time spent, no overhead should arise due to waiting for the different threads to synchronize at the end of each iteration.

2. In the second, I tried to discard split policies of a static type by opting for **auto scheduling**. In this solution, indeed, the computation of an element x_i of the solution vector for a given iteration is seen as a task. All tasks will fill a queue from which, different workers can pop as soon as they have finished the computation of the previous one. Of course, access to the queue to remove and to add elements are done in mutual exclusion so as not to allow two workers to work on the same x_i element or create inconsistency regarding the elements in the queue. If the queue is empty, the workers wait for it to be refilled, for example, with the element computations of the next iteration. To implement synchronization, at the end of each iteration, I have taken advantage of the **packaged_task** with its **get()** method. This, will make sure that the solution vector **x** of the current iteration is complete before starting with the execution of the next one. I tried to consider such a solution since it should have ensured a minimum waste of time spent on synchronization at the end of the iteration since the vector **x** is filled in the shortest possible time by the workers themselves requesting the tasks as soon as the previous one is finished.

3. Turning to the implementations concerning FastFlow, recalling that the problem we considered has properties that can be solved with Patterns of type Data Parallel. These patterns in FastFLow are implemented through ParallelFor/ParallelForReduce. The most suitable for the problem we are addressing is the ParallelFor that I used to parallelize the loop that cycles over the elements of the solution vector **x**. This loop is fine for the purpose since each iteration in it is independent from the others by concerning only the single element x_i. I decided, moreover, to try to assay both possible versions of this pattern:

   - the static version that can be found via the command **ff::parallel_for()**
   - the version that uses the object of type **ParallelFor**. On this object we can then call the same function as the static version.

   Obviously, the **parallel_for()** function requires parameters that can go into modifying its performance such as, for example, the use of dynamic or static scheduling. The choice regarding the latter will be discussed in the next section, which concerns the evaluations made to reach the chosen alternatives.



The image shows the general functioning of the solutions exhibited above. In FastFlow, the structure of ParallelFor is similar to this but with the addition of a scheduler that distributes the elements of the solution vector **x** at time t_i to the workers.

Next, looking at the computation to be performed for each x_i element:

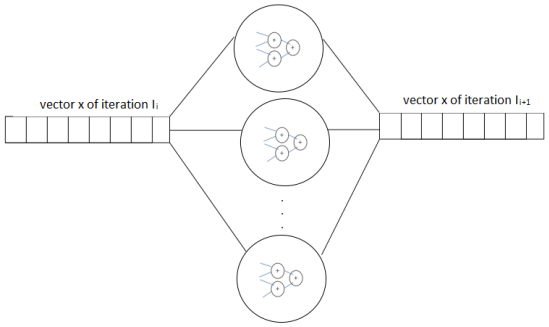
```
for(j=0, j < n, j++){
    sum += (A[i][j] * xprev[j]);
}
x[i] = (1 / A[i][i]) * (b[i] - sum);
```

I thought of including, instead of the workers that are going to perform this code sequentially, an additional pattern of type Data Parallel that is the Reduce. This is because the multiplications between the elements of the row i of the matrix **A** and the elements of the solution vector of the previous iteration **xprev** that are added consecutively in the variable **sum** depict exactly the pattern **Reduce**, more specifically the pattern **MAP/Transform+Reduce**.

For the purpose of trying to implement this additional parallelization inside the "task" function to be passed to threads, I included 3 ways of usage:

1. **sequential** that is the default one that does not add any additional parallelization to perform summation,
2. **parallel exploiting GrPPi** which implements the Map+Reduce pattern via the function **map_reduce()**. This takes:
   - the execution mode that we want to be parallel with native C++ threads.
   - the parameters for scrolling the row i of **A** and the vector **xprev**
   - the two lambdas that will take care of the map/transform step (the multiplication) and the reduce step (the sum with the variable **sum**)
3. **parallel exploiting FastFlow** via the object of type **ParallelForReduce** on which I call the function **parallel_reduce()** that takes almost equal parameters compared to the function of the previous mode.

the general structure of the solution with the addition of the Reduce pattern for each worker becomes this:

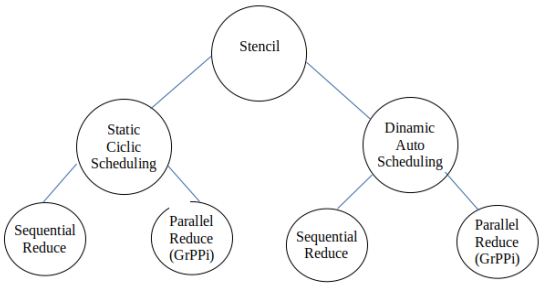


1.3 Evaluations used to pick up the final alternative

In this section, I will explore the various solutions by running tests in order to understand how they perform in relation to the sequential one and among themselves.

1.3.1 Implementation with native C++ threads

In summary, the alternatives considered for the implementation with native C++ threads are encapsulated by the various paths in this tree:



To figure out which solution was best, I explored the paths in this tree with the aim of finding the best one in terms of Latency. To begin the evaluation I did some tests regarding the time spent of the purely sequential solution so as to have a comparison data for the parallel solutions. Out of 10 runs the result is as follows:

Solution	N(matrix size)	Iterations	Average (usec)	Single Iter(usec)
Sequential	1000	1000	1907749	1908

Trying the two types of scheduling, on 10 executions with 8 as number of workers, I expected an almost equal result between the two solutions and close to the execution time of the sequential solution divided by the number of workers. Instead, the results came out as follows:

Solution	N(matrix size)	Iterations	Average (usec)	Single Iter(usec)
Stencil + static sched.	1000	1000	13247121	13247
Stencil + dinamic sched.	1000	1000	380362	380

The solution with Stencil + Static Scheduling met expectations, with the single iteration latency averaging 381 $\approx$ 238 = 1908/8 microseconds of the ideal

parallel time. The solution with Stencil + dynamic scheduling, on the other hand, did very bad, with latency per iteration equal to $13247 \gg 238 = 1908/8$ microseconds. In fact, the latter wastes time by two orders of magnitude more than the sequential solution and the one with Stencil + static scheduling. The reason for this overhead I think can be attributed to the mutually exclusive access on the queue of tasks from which the workers draw and the emitter inserts which probably leads to a significant waste of time compared to the time needed for the single task computation. And this overhead is not justified by any improvement in terms of task distribution to the workers compared to the sequential solution. This is because, since the computations are all similar in terms of time, even assigning them statically there will be no loss of time due to synchronization of the various threads at the end of each iteration. Continuing down the solution tree, discarding the dynamically scheduled branch, we try to add the additional parallelism given by reduce. Using a size (N), for vectors, equal to 1000 as in the previous proofs the theory tells us that each sequential computation of an element x_i should waste time equal to 1000*t$\oplus$. With the parallelization given by the reduce instead, the time is supposed to come to (1000*t$\oplus$ / nw) + t_{split} + t_{merge} per element x_i where **nw** is the degree of parallelism used for each reduce. So in general this would bring the latency for the entire execution to:

$$\frac{1000000 * t_{\oplus}}{nw * mw} + \frac{1000 * t_{split}}{mw} + \frac{1000 * t_{merge}}{mw}$$

While the overall latency for the entire execution without the contribution of the reduce would be this:

$$\frac{1000000 * t_{\oplus}}{mw}$$

where **mw** is the degree of parallelism assigned to the Stencil. Is clear immediately that the result with the addition of the reduce will be better if and only if the terms for the time to split and the time to merge the reduces is negligible in relation to the improvement brought by the contribution of **nw** in the denominator of the main term of the formula. We will also have to consider the overhead given by the creation and consecutive join of all those threads used by the reduce for each computed sum. Continuing with the tests performed again over 10 iterations, these are the results:

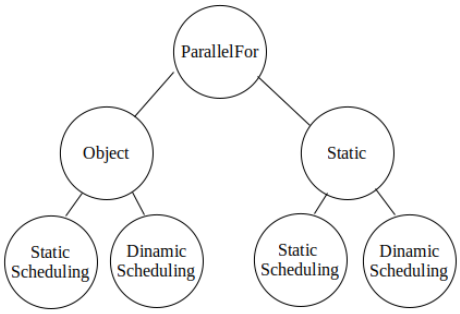
Solution	N(matrix size)	Iterations	Average (usec)	Single Iter(usec)	mw	nw
seq. reduce	1000	1000	380362	380	8	1
par. reduce	1000	1000	72391766	72391	8	4

As we can see, unfortunately, the use of reduce only leads to a terrible performance degradation, about 7 times higher even than the dynamic scheduling

solution. This is probably due to the little improvement brought by the additional degree of parallelization compared to the too much overhead due to handling additional threads and split and merge times. For this reason I discard this type of solution that includes reduce. In the end, therefore, what I consider my final solution relative to native C++ threads is **Stencil + static scheduling + sequential reduce**.

1.4 Implementation with FastFlow

Turning to the implementation with **FastFlow**, from previous experience I decided to discard the solution that included **ParallelFor + ParallelForReduce** since I would probably end up in the same situation of the reduce in the previous case. Based on this consideration the solution tree visited is this:



We begin considering the first two alternatives:

- **parallelFor as object** that will allow us to use spin-wait thus a more efficient lock mechanism,
- **the static call of the parallel_for** method offered by the ff class that will allow us to avoid the initialization that occurs in the alternative with object, but does not use spin-wait.

For each of the two alternatives, the option of using dynamic and static scheduling is then available. Let us see how the 4 different options behave on 10 executions with 8 workers:

Solution	N(matrix size)	Iterations	Average (usec)	Single Iter(usec)
PFObj + static sched.	1000	1000	339891	340
PFObj + dinamic sched.	1000	1000	341011,8	341
PFStatic + static sched.	1000	1000	5869875	5870
PFStatic + dinamic sched.	1000	1000	10220462	10220

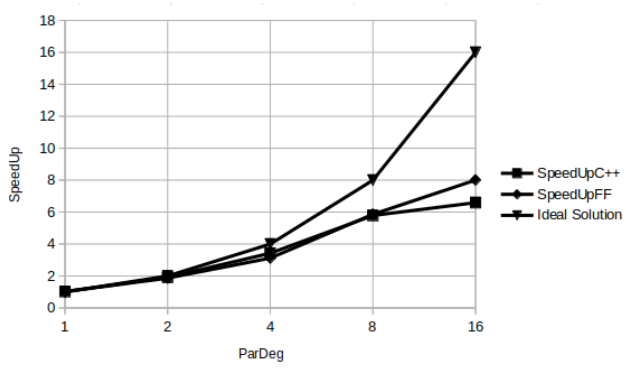
Is clear from the results that the solution using ParalelFor as an object shows far higher performance than the static call of the ff class. This is probably in part due to the optimizations brought by the more efficient lock mechanism but not only. Indeed, using PFObj without spin-lock the performance degrades by an order of magnitude with the single iteration taking 1124 microseconds which however is still less than the 5870 of the static call. The difference between static and dynamic scheduling, on the other hand, is negligible regards the **PFObj** version while it results in a degradation of about two times in the performance of the **PFStatic** solution. My choice for the implementation regarding FastFlow therefore, falls in the alternative **ParallelForObj + static scheduling**.

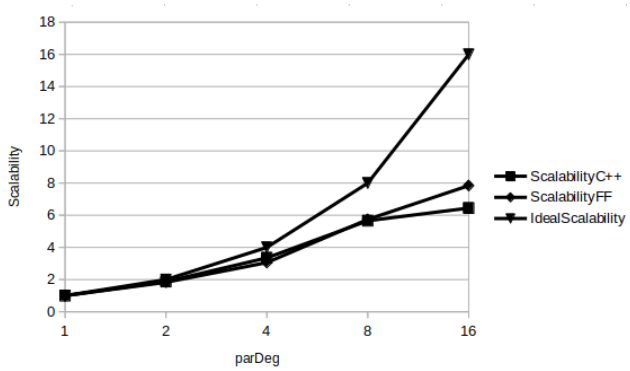
2 Differences between achieved performance figures and the ideal/predicted ones

Having chosen my two alternatives for both the implementation with native C++ threads and with FastFLow, let us compare the results had by them in relation to the ideal theoretical results. The various metrics I will examine will be **SpeedUp**, **Scalability** which gave very similar results and **Efficiency**.

SpeedUp is used to see the improvement in execution speed relative to the degree of parallelism used, while, **Scalability** is the ability of algorithm to effectively utilize an increasing number of cores. The results we get are these:

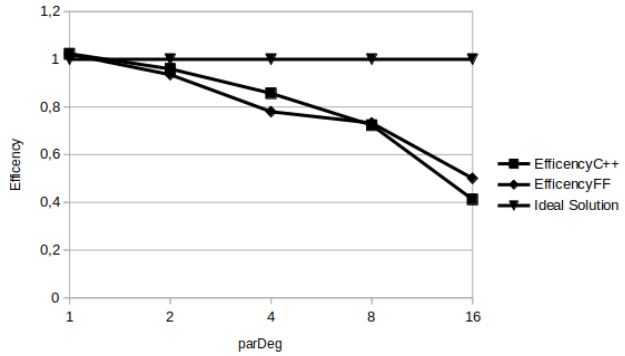
SpeedUpC++	SpeedUpFF	ScalabilityC++	ScalabilityFF	Ideal Solution
1,02	1,02	1	1	1
1,92	1,87	1,876	1,83	2
3,43	3,12	3,35	3,05	4
5,80	5,86	5,67	5,74	8
6,60	8,01	6,44	7,84	16





The images show how both solutions approach the optimum up to the parallelism degree of 4 workers. Then as is physiological, they begin to move away from the ideal performance at some point, with the solution using FastFlow performing better than the one with native C++ threads in the last test with degree of parallelism of 16 workers. The causes of this performance degradation is surely due to some kind of overhead. The first type of overhead that I think affects performance is that due to **thread setup** (creation and the consequent join). As a matter of fact, it only makes sense to increase the number of threads used if the time gained in execution is much greater than the time spent on thread setup.

EfficiencyC++	EfficiencyFF	Ideal Solution
1,02	1,02	1
0,96	0,93	1
0,86	0,78	1
0,72	0,73	1
0,41	0,50	1



As we can see from this graph on efficiency, as parallelism increases the efficiency of parallelism itself degrades. This means that as parallelism increases,

the overhead brought by managing threads is always more significant than the actual improvement brought by adding these threads to the computation. In contrast, as the input data size increases, the overhead due to thread creation has less impact on the overall computation since the latter will be spread over the computation of more elements.

Further overhead could be due to **sub-system memory management**. In fact facing a problem where different workers share a resource (the solution vector x), we might have **false sharing** problems where the **cache coherence protocol** of the respective cores is launched repeatedly but unnecessarily since for each worker, in a certain iteration, it is not necessary to become aware of the updates of the solution vector **x** brought by the other workers during the same iteration. The only update needed for each core's cache is the one related to the state of the solution vector **x** at the end of each iteration. This problem, if present, could have been solved by padding inserting empty positions between the different elements of the solution vector **x**.

3 Instructions to re-compile and run the code

First of all you have to unzip the file **SPM_jacobi(AJ)_diRitoGianmaria.zip**. This folder includes the executable **jacobi.cpp**, his makefile and the **utimer.cpp**. In **jacobi.cpp** there are all the alternatives I considered to achieve my best solutions.
To compile it you need to run the command `make jacobi` in the folder.

Since I used both **FastFLow** and **GrPPi** in the code, the `-I` flags are also inserted in the makefile to tell the compiler where the respective directories are. So in case you want to run the makefile somewhere other than the remote virtual machine you will need to change the paths of these flags.

Once the .cpp has been compiled, it will be enough to run the executable
`./jacobi <N> <nw> <k> <seed> <max>`
where:

- `<N>` is the size of **A**, **x**, and **b**,
- `<nw>` is the degree of parallelism that you want to assign to **Stencil**,
- `<k>` is the maximum number of **iterations** to be performed,
- `<seed>` the seed for creating the random values of **A** and **b**,
- `<max>` is the maximum possible value inside **A** and **b**.