SPM Project 2023-2024

Salvatore Correnti

September 5, 2024

Contents

Abstract

This report describes the results of the parallelization analysis conducted on a distributed wavefront computation problem.

In particular, we discuss and compare the results of: a shared-memory implementation, using Fastflow, for a single multi-core machine; and a distributed-memory implementation, using MPI, for a cluster of multi-core machines.
The results show good performances in all cases, and we also see that partitioning the matrix in "tiles" significantly improves performance.

1 Parallelization Strategy

In the sequential version, for each diagonal k of the upper-triangular matrix, all elements are calculated sequentially: in this way, we guarantee that at each time we have all elements needed for calculating each item.
Our first approach, stemming from this consideration, was to parallelize the computation of each diagonal across workers: at each step k we split the k-th diagonal across the workers, compute all its items and then update the matrix. For the single multi-core machine we compared the results when using "Block", "Cyclic" and "Block-Cyclic" policies over the diagonals, as defined in the course. After that, we decided to generalize the approach by partitioning the matrix into square *tiles* of a given *tileSize* T, with the possibility of having rectangular tiles at the bottom of each "diagonal" made by these tiles. We then apply all the policies described above to the matrix made up by all the tiles. For each tile, we compute the value for each item by traversing the tile from the left to the right and from the bottom to the top. The idea of tiling is to try to exploit problem locality and cache usage, supported by the fact that we will have all the items on the same row computed one after the other.

For the **Fastflow** solution, we used the `ParallelFor` construct, which automatically distributes the computation of a loop in which all iterations are independent from each other across multiple workers.

For the **MPI** solution, given a fixed number of nodes in the cluster and a number of worker processes within those nodes (and supposing that the number of workers is always greater or equal than the number of nodes), both the master and each worker have their own copy of the matrix with all the items computed up to the current moment. At each step K, we distribute the computation of the K-th diagonal (or "tile-diagonal") items throughout all the workers according to the Block Policy, and after the computation each worker sends to the master node the values it has computed. The master then updates its copy of the matrix with the new values and then sends the

whole K-th diagonal values to each worker, which updates its own copy of the matrix.
This communication strategy has been adopted to reduce the total number of exchanged messages between the nodes: we have in fact that the total amount of exchanged data has the size of $W \times N_K$, where W is the number of workers and N_K is the number of elements in the K-th diagonal, while the total number of exchanged messages is $2W$. If we consider instead a strategy in which each worker sends its computed data both to the master and to all the other workers, the total amount of exchanged data is still $W \times N_K$, but the total number of exchanged messages is W^2.
For simplicity, we kept the usage of **Cyclic** and **Block-Cyclic** policies only for the **Fastflow** solution, since we saw that those different policies do not significantly impact both execution time and parallelization efficiency.

Figure 1: An example of partition of a 9×9 (left) and an 8×8 (right) matrix with a tile size T of 2: gray tiles are included in the first diagonal by definition, but are excluded from any calculation. We did not consider load balancing issues when N is not a multiple of T, but it was not the case of our experiments.

2 Computing Platforms and Experiments Setup

For our experiments, we used both the `spmcluster` and `spmnuma` architectures: the first is a cluster of multi-core machines, while the second is a single `NUMA` multi-core machine. We ran the **Fastflow** implementation on *both* architectures in order to compare their impact on the problem, while the **MPI** implementation was run *only* on `spmcluster` architectures.
For all architectures, we exploited parallelism up to the maximum number of available workers for each single user, which is 32 for `spmnuma` and 16 for `spmcluster`.

We decided to experiment with matrix sizes of 1000, 2000, 4000, 6000 and 8000 for all the experiments, while including also a size of 10000 for the

MPI version on `spmcluster`.

We also defined a quick **checksum** function that is called at the end of each run, which computes a checksum of the computed matrix, as a quick double check for the correctness of the implementation. Checksum is defined over matrix $M \in \mathbb{R}^{N \times N}$ as: $C(M) := \sum_{i=1}^{N} \bigoplus_{j=1}^{N} M_{ij}$

3 Performance Analysis

For all our experiments, we considered **speedup**, **efficiency**, **strong scalability** and **weak scalability** metrics for our performance analyses. In particular, for weak scalability, we considered **total time** needed by the problem according to its time complexity.

We can compute the number of operations needed to fill the matrix as: $T(N) = \sum_{k=1}^{N} (N-k)(2k) = \frac{1}{3}N^3 - \frac{1}{3}N$ We then expect that when scaling $N \to kN$ with $1 \to k$ workers, we will get a total time of $\frac{1}{3}k^2N^3 - \frac{1}{3}N \approx k^2(T(N) + \frac{1}{3}N)$, i.e. a factor of k^2 plus a linear factor on N in total time.

3.1 Fastflow on `spmcluster`

3.1.1 Speedup and Efficiency

(a) Speedup over problem sizes with Block Policy and 1×1 blocks

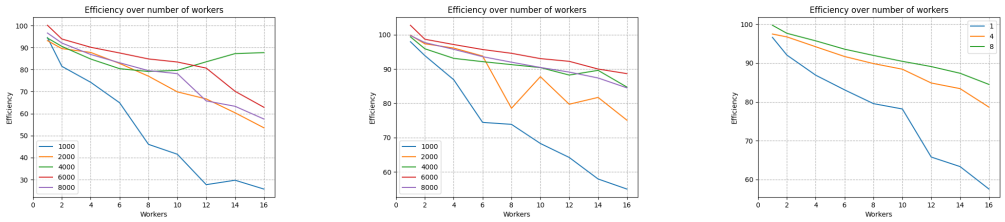
(b) Speedup over problem sizes with Block Policy and 8×8 blocks

(c) Speedup over values for blocks with Block Policy and size of 8000

Figure 2: Speedup values over sizes for left and center and over block sizes with 8000 size for right

From 2, we see that speedup is substantially monotonically increasing with number of workers, except for the case $N = 1000$, for which we see no performance improvements for more than 6 workers. We see an anomalous pattern for $N = 4000$, for which speedup increases faster than any other

value, especially for more than 8 workers. This anomaly does not occur when increasing tile size to 8, and has appeared throughout multiple runs. Besides from this, we see how a tile size of 8 causes a significant improvement in speedup, especially for $N \leq 2000$. Figure 2c shows speedup values for $N = 8000$ for tile sizes of $1, 4, 8$: we see how increasing from 1 to 4 gibrs a significant boost for more than 8 workers, with little improvements for a tile size of 8. Efficiency plots in 3 show how efficiency is rapidly lost when



(a) Efficiency over problem sizes with Block Policy and 1×1 blocks

(b) Efficiency over problem sizes with Block Policy and 8×8 blocks

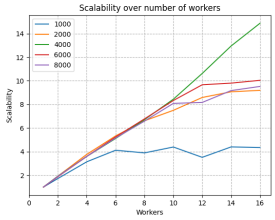
(c) Efficiency over values for blocks with Block Policy and size of 8000

Figure 3: Efficiency values over sizes for left and center and over block sizes with 8000 size for right

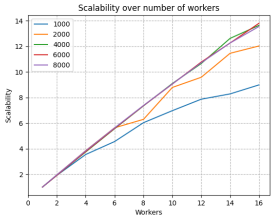
increasing number of workers for $N = 1000$, while it is substantially retained for $N \geq 2000$ (3a). With a tile size of 8, we see in 3b a similar relative pattern between problem sizes, except that efficiency is maintained over **50%** even for $N = 1000$, while for $N \geq 2000$, we see that efficiency keeps around **80-90%** even with 16 workers. Finally, in 3c we see the impact of tile sizes of $1, 4, 8$ on problem with size $N = 8000$. Increasing from 1 to 4 causes efficiency from dropping to $\approx$ **60%** to dropping to $\approx$ **80%**. We even see a superlinear efficiency of **102.7%** for 1 worker and $N = 6000$, corresponding to a difference of ≈ 3 seconds of computing time.

3.1.2 Strong Scalability

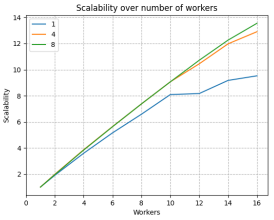
In 4 we see the strong scalability data, respectively for 1×1 tiles in 4a, for 8×8 tiles in 4b and a comparison between all tile sizes for a matrix size of 8000 in 4c. Scalability values are similar to speedup ones, and in particular we see again the anomalous values for $N = 4000$ and the substantial improvements induced by moving from tile size of 1 to 4.



(a) Strong Scalability over problem sizes with Block Policy and 1×1 blocks



(b) Strong Scalability over problem sizes with Block Policy and 8×8 blocks



(c) Strong Scalability over values for blocks with Block Policy and size of 8000

Figure 4: Strong Scalability values over sizes for left and center and over block sizes with 8000 size for right

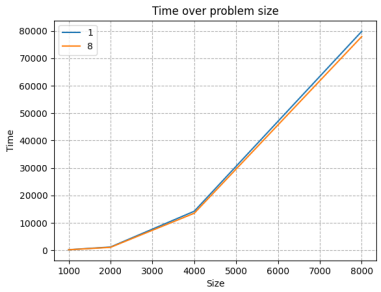
3.1.3 Weak Scalability

In 5 we see both total time and total time divided by W^2 (where W = number of workers) for tile sizes of 1 and 8. We see how a tile size of 8 is effective in reducing total time, and from 5b we see that quadratically scaled total time is more or less linear with N.

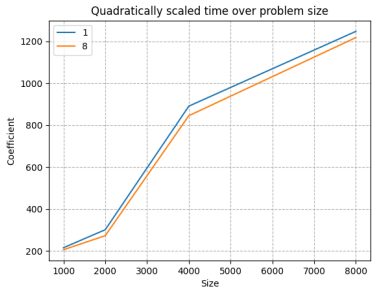
3.2 Fastflow on `spmnuma`

3.2.1 Speedup and Efficiency

From 6, we can see that when increasing problem size, the speedup increases itself, according to the fact that the parallelizable part of the program contains all computational burden of the items of the matrix, and that the total number of items increases quadratically with matrix size.
We can also observe the impact of incrementing block size: in the leftmost figure we see results with a tile size of 1, while in the centered one we see with results a tile size of 8. We notice how for 6b, the quality of the speedups increases for all matrix sizes. To better show the impact of varying tile size, in 6c, we show the results obtained on matrix size of 8000 with tile sizes of 1, 4 and 8. We see how the most impact is when moving from 1 to 4, with a significantly smaller improvement when moving from 4 to 8. In 7 we see the efficiency of the Fastflow implementation: in 7a over sizes with tile size of 1, in 7b over sizes with tile size of 8, and finally in 7c over tile sizes of $1, 4, 8$ with matrix size of 8000.
As we see, the parallel implementation starts to be effective efficiency-wise



(a) Weak Scalability (time) over problem sizes with Block Policy



(b) Weak Scalability (quadratically scaled time) over problem sizes with Block Policy

Figure 5: Strong Scalability values over sizes for left and center and over block sizes with 8000 size for right

from a matrix size of 4000, and we also see how increasing tile size gives substantial improvements. In particular, with 8×8 files, we see that we even gain a **superlinear speedup** for a matrix size of 8000: **108.9%** with **2** workers, **107.2%** with **4** workers, and downward to **102.6%** with **16** workers. This superlinear speedup is probably explained by the fact that calculating a small 8×8 square of adjacent positions one after each other makes a better use of cache memories and also implies less synchronization steps between threads (because there are less iterations in the outermost loop, each one needing synchronization between threads). Intuitively, this tends to be more impactful when matrix size increases.

3.2.2 Strong Scalability

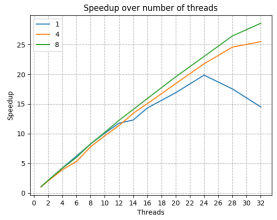
In 8 we see the strong scalability data, respectively for 1×1 tiles in 8a, for 8×8 tiles in 8b and a comparison between all studies tile sizes for a matrix size of 8000 in 8c. Scalability values are similar to what we found for speedup: we see the same almost flat curves for $N = 1000$ and $N = 2000$, with a substantial improvement when increasing tile size from 1 to 8. Similarly for speedup, we see that the increase in tile size from 1 to 4 is responsible for most of the scalability optimizations, while 4 and 8 gives almost identical results, except for > 28 workers.

3.2.3 Weak Scalability

For Weak Scalability, in 9 we see that tile size has almost no impact on relative problem times, and in particular we see in 9b that by dividing total

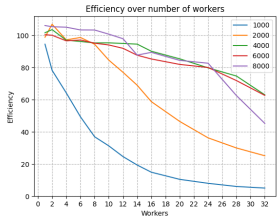
(a) Speedup over problem sizes with Block Policy and 1×1 blocks

(b) Speedup over problem sizes with Block Policy and 8×8 blocks

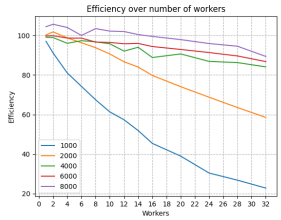


(c) Speedup over values for blocks with Block Policy and size of 8000

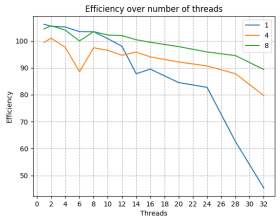
Figure 6: Speedup values over sizes for left and center and over block sizes with 8000 size for right



(a) Efficiency over problem sizes with Block Policy and 1×1 blocks



(b) Efficiency over problem sizes with Block Policy and 8×8 blocks



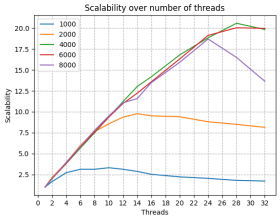
(c) Efficiency over values for blocks with Block Policy and size of 8000

Figure 7: Efficiency values over sizes for left and center and over block sizes with 8000 size for right

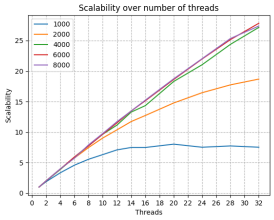
time by W^2, (W = number of workers), we have linear values with N.

3.3 MPI on `spmcluster`

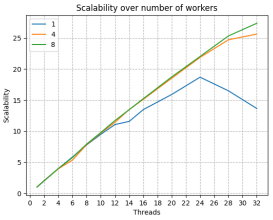
For the `MPI` implementation on **`spmcluster`**, for simplicity we considered the possibility of employing multiple MPI workers over the same node in order to increase total number of workers. In particular, we experimented with $1, 2, 4$ and 8 nodes and with $1, 2, 4, 8, 16$ workers per node, up to a maximum amount of workers of 64. While the overhead caused by the running of multiple MPI processes with their message exchanges over the same cluster node has shown to limit the scalability when having more than 4 workers per node, the results show how it is still possible to gain significant improvements by increasing the number of workers per node up to 4.



(a) Strong Scalability over problem sizes with Block Policy and 1×1 blocks



(b) Strong Scalability over problem sizes with Block Policy and 8×8 blocks



(c) Strong Scalability over values for blocks with Block Policy and size of 8000

Figure 8: Strong Scalability values over sizes for left and center and over block sizes with 8000 size for right

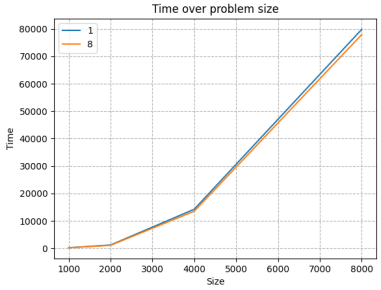
Since in this case we have two "configuration dimensions" (i.e., number of nodes and workers per node), we focus mainly on the (strong) scalability and its relative efficiency, both by fixing the number of nodes and scaling on the number of workers and by fixing the number of workers per node and scaling on the number of nodes. We always took as base for scalability calculations the results with 1 worker and 1 node.

3.4 Scalability with fixed nodes

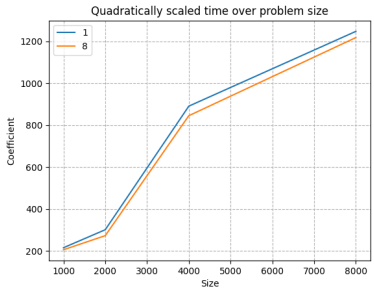
In 10, we see strong scalability values when increasing total number of MPI workers while maintaining a fixed number of nodes. In particular, in the first 3 figures we consider up to 16 workers per node, while in the last one we limit to up to 8 workers per node. Most of the improvements happen with low workers per node (wpn), and in particular with 12 wpn with 1 node, 8 wpn with 2 nodes, 4 wpn with 4 nodes and 2 wpn with 8 nodes, i.e. the maximum efficiency on all sizes is reached when there are ≈ 16 total workers. We see also that in most cases, scalability starts to decline after a certain number of workers, and this is probably due to the computational overhead of synchronizing many MPI processes, several of them on the same nodes. An improvement may be to consider instead multiple threads with shared memory per node, instead of multiple MPI processes.

3.5 Scalability with fixed workers per node

In 11, we see strong scalability values with fixed workers per node (wpn) of $1, 2, 4, 8$ over the total number of available nodes $(1, 2, 4, 8)$, up to 64 total

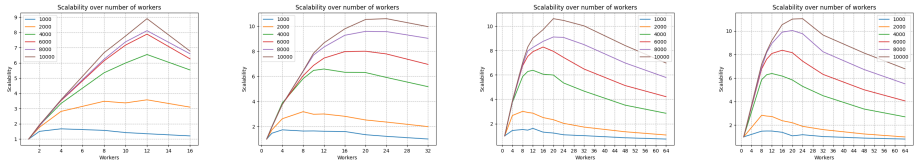


(a) Weak Scalability (time) over problem sizes with Block Policy



(b) Weak Scalability (quadratically scaled time) over problem sizes with Block Policy

Figure 9: Strong Scalability values over sizes for left and center and over block sizes with 8000 size for right, with 1×1 and 8×8 tiles.



(a) Strong Scalability with 1 available node and 8×8 tiles.

(b) Strong Scalability with 2 available nodes and 8×8 tiles.

(c) Strong Scalability with 4 available nodes and 8×8 tiles.

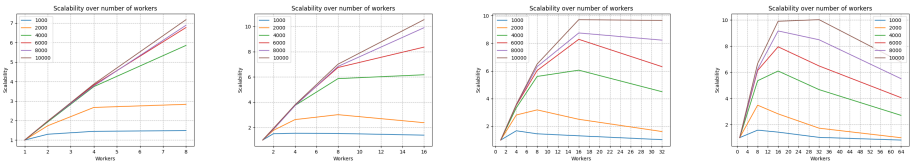
(d) Strong Scalability with 8 available nodes and 8×8 tiles.

Figure 10: Strong Scalability values over sizes with fixed total nodes.

workers. We see that most of the performance gains are for at most 2 wpn, with an efficiency up to $\approx 70\%$ with 2 wpn, 8 nodes and $N = 10000$. As already observed before, we see a stabilization and a subsequent decline in scalability with ≥ 4 wpn and ≥ 4 total nodes. This suggests an optimal balance between overhead cost and increase in efficiency with a total of ≈ 16 workers, in all cases.

3.6 Weak Scalability with fixed workers per node

For Weak Scalability, in 12 we see total time and quadratically scaled total time over sizes from 1000 to 8000 with fixed numbers of workers per node. As we see from 12b, we have a substantially linear increase in time after having



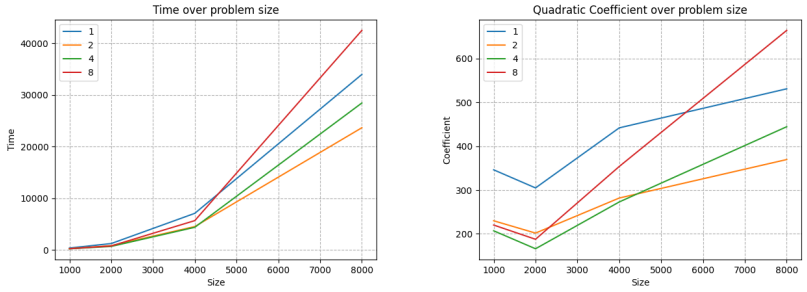
(a) Strong Scalability with 1 worker per node and 8×8 tiles.

(b) Strong Scalability with 2 workers per node and 8×8 tiles.

(c) Strong Scalability with 4 workers per node and 8×8 tiles.

(d) Strong Scalability with 8 workers per node and 8×8 tiles.

Figure 11: Strong Scalability values over sizes with fixed workers per node.



(a) Weak Scalability time comparison over $1, 2, 4, 8$ workers per node and 8×8 tiles.

(b) Weak Scalability quadratic coefficients over $1, 2, 4, 8$ workers per node and 8×8 tiles.

Figure 12: Weak Scalability values over sizes with fixed workers per node and 8×8 tiles.

scaled it quadratically, which agrees with theoretical predictions, indicating both a good fit and a substantial parallelizable part of the problem.

4 Compilation and Test Instructions

The source codes for the **Fastflow** and **MPI** implementations are respectively in `UTWavefrontFF.cpp` and `UTWavefrontMPI.cpp`. We also provided a `Makefile` for compiling and cleaning all sources, and several `Bash` scripts for running all the tests. In particular:

- `make all` and `make cleanall` compile and clean all files. Other options for compiling source codes are described in the `Makefile` itself

- `setup.sh` **must be run** before any tests to download `cereal` and `fastflow` Git repositories
- `ffruns_spmcluster.sh` and `ffruns_spmnuma.sh` take as argument matrix size and run all the tests for Fastflow implementation respectively on `spmcluster` and `spmnuma` `mpiruns.sh` takes as arguments matrix size and number of nodes and runs all the tests for MPI implementation on `spmcluster`

There are also several `Python` scripts we used for cleaning and preparation of the produced data, and for making plots.

5 Conclusions

We have seen the impact of parallelization over the original problem, both through a shared-memory **Fastflow** and a distributed-memory **MPI** implementations. For the first one, we compared the results obtained on `spmcluster` and `spmnuma` architectures, while for the second one, we ran onlt over `spmcluster`.
In all experiments, we saw that the actual advantages of parallelization start to appear with a matrix size $N \geq 2000$, which is reasonable considering that parallelization of the wavefront is limited by the need to compute diagonals one after each other, and parallelization of the dotproducts is difficult (and we did not parallelize over it).
As one would expect, there was no significant difference between (tile) distribution policies, and in particular we kept only the **Block** one for **MPI** experiments. Instead, the usage of tiles of 4×4 and 8×8 sizes revealed to be effective in improving both total computation time and speedup, efficiency and scalability. For $N = 1000$, it helped to improve efficiency on `spmcluster` Fastflow from $\approx 25\%$ ($T = 1$) to $\approx 55\%$ ($T = 8$) with 16 workers, and in several cases we also got **superlinear speedups**. Overall, our **Fastflow** implementation has shown to be quite robust with respect to all possible variables, especially with $N > 2000$.
The **MPI** implementation was limited by the frequent usage of multiple MPI workers on the same node, but overall it showed good strong and weak scalabilities, especially with a maximum of ≈ 16 nodes, corresponding to ≈ 2 workers per node with 8 nodes. It could be investigated whether switching to a multithreaded solution for each node together with MPI for synchronizing all nodes could improve those performances.