BRUNDU
FRANCESCO
MATR.438905

SPM REPORT FISHES AND SHARKS

Framework: FastFlow
Target architecture: Intel® Xeon Phi™ coprocessor

SPM Final project

Fishes and Sharks

Table of Contents

1. Specifications 2
2. Project Design 3
3. Finding Concurrency design space 4
4. Algorithm Structure design space 7
5. Supporting Structure design space 10
6. Implementation Mechanisms design space 13
7. Performances 15
8. User manual 20

Bibliography 21

1. SPECIFICATIONS

An initial population of fishes and sharks evolves according to the following set of rules:
1. Fish breeding age starts at 2, shark breeding age starts at 3
2. Fish live up to 10, then they die, sharks live up to 20, then they die
3. An empty cell with >= 4 fish (shark) neighbours and >= 3 of them in breeding age and < 4 shark (fish) neighbours is filled by a new fish (shark) individual with age 1
4. A fish in a cell
    dies if > = 5 neighbours are sharks (Shark food)
    dies if>= 8 neighbours are fishes (overpopulation)
    otherwise its age increases
5. A shark in a cell
    dies if >= 6 neighbours are sharks and =0 neighbours are fishes (starvation)
    dies with 1/32 probability by random causes
    otherwise its age increases

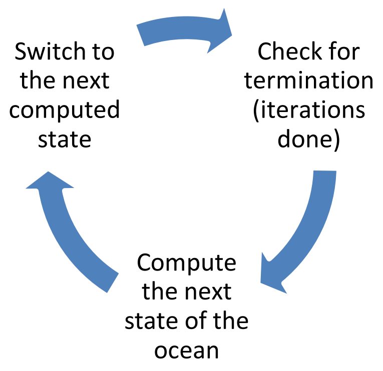
The application computes the evolution of an initial population is a mesh of N*N cells (25% sharks, 50% fishes and 25% empty cells) for a number M of iterations. A library call will be provided to display a matrix of short integer values on the screen, which will run on the host processor where the Xeon PHI co-processor is attached.

Notes:
Ocean modelled as a 2D toroidal grid
Instead of an infinite grid, our grid will be finite with toroidal boundary conditions. This means that cells on the edges of the grid will “wrap around” to connect with the opposite edge of the grid: the northernmost cells are adjacent to the southernmost cells, and the westernmost cells are adjacent to the easternmost cells. Thus, the grid’s shape is like the surface of a torus (hence “toroidal”) or donut.

2. Project Design

Given an initial ocean with fishes and sharks, the algorithm simulates the evolution of the ocean for s time steps.
Each time steps consists of 2 steps:

1. computes the next state of the ocean
2. switch to the computed state



Switch to the next computed state
Check for termination (iterations done)
Compute the next state of the ocean

To guarantee the correctness of the algorithm, regardless the order of the updates over the grid, we use two oceans, the actual and the next one. At the end of the computation the matrices swap their role and another time step is computed in a double buffering fashion.

The infos needed for each cell are: the type (empty, fish or shark) and the age.
The program itself is a time-step loop in which each iterations computes the next state

A first solution would be an AOS representation of the data, and then this can be made more memory efficient with a SOA data structure with the grid modelled as two parallel arrays one for the type and one the age.
This data representation can be optimized collapsing all the infos into one int, only 4 byte per cell needed.
The encoding is 0 = empty, 1 = fish, 11 = fish breed, 100 = shark, 1100 = shark breed; this definition maps the type infos into 4 digits d4 d3 d2 d1, finally the age of a given cell can be put into d6 and d5.
This way we do not only reduce the required space and the memory access to retrieve the data, but also the instructions needed to do the checks for each type of cell. We now need only sum, division and modulo operations. To do count the number of all the kind of elements in the neighbourhood we just sum up the boundary cells and obtain in each digit the count of the associated category.
This can be done because there are at most 8 neighbours.

```
inline int count(Type ** actual, const long i, const long j) {
  return actual[i - 1][j - 1] + actual[i - 1][j] + actual[i - 1][j + 1]
      + actual[i][j - 1] + actual[i][j + 1]
   + actual[i + 1][j - 1] + actual[i + 1][j] + actual[i + 1][j + 1];
}
…
  int neighborhood = count(mesh.contains, i, j);
  int count_fish = (int) neighborhood % 10;
  int count_fish_breeding = (int) (neighborhood / 10) % 10;
  int count_shark = (int) (neighborhood / 100) % 10;
  int count_shark_breeding = (int) (neighborhood / 1000) % 10;
  int age = (mesh.contains[i][j] / 10000);
```

Listing 1.Integer count and decoding.

In the next section we will provide an analysis of the project design based on the parallel patterns methodology described in [1] [2] [3]. The analysis of the experimental results on the target architecture is in 7 and 4 contains the user manual.

3. Finding Concurrency design space

First we consider the patterns in the finding concurrency pattern group.

This is by definition a stencil problem; naturally we can consider the "data decomposition" pattern to model its parallel execution. We will look for concurrency during the computation of the next time step, which has a cost of N^2.

Data/task decomposition

The ocean grid is a large central data structure, in a shared-memory environment, where for each game tick a cell's next state depends only on the current state of its local neighbours, Figure 1.

Thus, we can effectively decompose the evolution algorithm into a set of localized problems. Decomposing the grid to be updated instead of the actual grid, change the problem between having or not independent local writes. In this way the current state grid is read-only during the next state computation, so there are no data dependencies created by the decomposition, the next mesh instead is write-only, but the update is local to the cell. The dependencies between two consecutive iterations have not been changed.

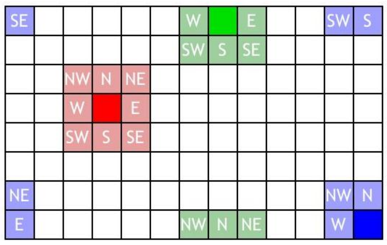


Figure 1. Cell boundary in a toroidal grid.

The grid can be modelled as a 1D or a 2D array, Figure 3, and then it can be broken into chunks (segments, rows or columns blocks) which are associated with a Unit of execution (UE). The basic task becomes the update of such a chunk, Figure 2.

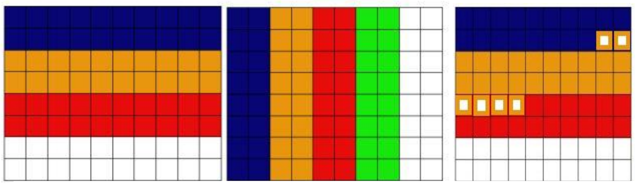


Figure 2. Row block (a) and column block (b) decomposition. Segment decomposition (c) mapped back to matrix form.

The 1D array segment decomposition give the most balanced decomposition, at most one more cell is assigned to some tasks.

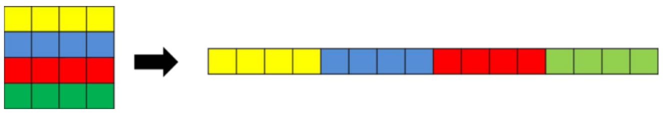


Figure 3.Matrix transposition into 1D array.

The row/column wise decomposition gives one more row/column to some tasks, the time needed for the biggest chunk is a lower bound on the service time. The column wise decomposition sometimes is known also as strip mining, because the column size is a multiple of the cache line size, this should improve locality.

Considering how the data is accessed the row blocks exploit the most prefetching, and the data write are always local and contiguous.

Group task & Order task patterns

"How the tasks are grouped to simplify the management of dependencies"

Temporal dependencies are between tasks of consecutive time steps.

Given the ocean modelled with two grids, resp. read-only and write-only, the tasks into a group are independent each other. We can group together and execute concurrently all the tasks that compute the very next state.

"How to order the groups of tasks"

The dependencies are only between tasks of different time steps, consecutive iterations must be serialized.

Data sharing pattern

"Which data and how it is shared among the tasks"

We are targeting a shared memory system, we can simply access the shared global data structures, and no message passing is needed. Each block of the grid is identified by appropriate indices.

The task local data is the next state grid block, which can be updated independently; each one is associated with the task that updates it.
The neighbourhood of a block in the actual state grid is shared between neighbouring blocks. Knowing that this grid is read-only, no synchronization is needed.
The grids are swapped between two consecutive iterations, the read-only ones becomes the write-only and vice versa.

The random computation has a global shared state; so the random computation will be treated as a shared data access.

Design evaluation pattern

"The decomposition and dependency analysis is good enough?"
"Which kind of performance can we achieve?"

The decomposition is suitable for the Xeon phi, which has 60 cores multithread (4 HW contexts per core), can handle 240 concurrent threads in a ccNUMA memory system. Contexts on the same PE share L1- L2 caches; the phi coprocessor has fast L2 communication among cores. The memory hierarchy of the target machine can favour some scheduling policy. Better memory sharing if the boundary is already on the local memory.

We can have as many tasks of varying size and number as we need, paying attention that the granularity of the block impacts the efficiency, memory overheads and parallel overheads can be of the same order of magnitude of the computation if not enough work to do; the efficiency depends also on the load balance between tasks which is kept minimum.
The tasks are regular and independent tasks are grouped into iterations.
Tasks are also independent of scheduling and extensive data sharing is not a problem within the same address space. The scheduling and load balance are easy to manage.

Sync mechanisms only between consecutive iterations, and balanced works means waiting time minimized by the cost of the barrier.

We maintained a simple data organization and resource management, which is good for code management and debug.

A simple but effective fault tolerance management policy can be the checkpointing, which can be applied to handle HW fault with a restart. A basic solution can be doing a checkpoint after a number of iterations, number based on the average time needed to compute one. The checkpoint includes few information, like the ocean actual state grid and the number of iterations done so far, the restart is simply done starting a new computation with the saved mesh as initial mesh.

4. Algorithm Structure design space

"Which patterns are more appropriate for this problem?"

"Which implementation will provide the best performance?"

"How much overhead is introduced?"

In this design space we decide the abstract algorithm to be used to implement the concurrent activities identified in the previous step.

Appropriate patterns are:

Organize by task, task parallelism;

Organize by data decomposition, geometric decomposition.

For this problem the way data is decomposed and shared among tasks stands out as the major and most productive way to organize the concurrency. We decide to use a "Geometric decomposition" pattern from the "organize by data decomposition" group to implement the computation of the next state. Each computation basically has a cost of N^2 memory references and of N^2 integer/random computations.

Choosing the alg struct pattern we consider the constraints of the target platform such as the low clock rate with respect to the host (Xeon), and the fact that the best performances are achieved with vectorization and simd like algorithms.
In our case we cannot take advantage of vectorization because we compute a point sum over arrays not a new array and a "reduce"-like alg would be much more complex to handle.

The system can handle 240 UEs, of which until 59 can have a dedicated PE (1 PE for the OS), then until 4 HW contexts per core. In a ccNUMA memory bound application the more adjacent the data memory location the least the access time, there is an overhead penalty if we go through main memory or communication between far cores.

Geometric decomposition

We have a sequence of iterations, each composed by a group of independent tasks defined over a region of the ocean grid, identified with the corresponding limit indices; we share the read only actual grid information among the neighbourhood. Depending on the target platform (distributed/shared memory) we can decompose accordingly the mesh in regions.

```
for (long it = 0; it < iterations; it++)    {
    for (long i = 0; i < grid.n; i++)
         for (long j = 0; i < grid.n; j++)
        update (i, j, actual, next);
    swapGrids();
}
```

Listing 2.Algorithm structure.

The majority of the elements can be updated using only the data from within the chunk, while boundary elements need neighbouring chunks.
The most heavy operation in a cell update is the rand(), which needs a critical section, associated to the shark death, a chunk with a lot of sharks are more computationally intensive than the others.
With a good probability the sharks should be equally distributed through the ocean, but if they condensate in some regions, the concurrent computations get serialized.
The management of memory accesses and UE management and synchronization is a key factor in the design, if the time spent by a UE computing the updates of the assigned block is smaller or of the same order of magnitude of the parallel overhead, there isn't any performance improvement. Care should be taken also if the memory access time is smaller of the time spent computing the retrieved elements, this will imply a computation time bounded by the memory access time.

Shared memory requires that the data into cache is used many times before new data replaces it, using large blocks of contiguous loop iterations, like in the row block decomposition, increases the chance that multiple values fetched in a cache line will be utilized and subsequent loop iteration are likely to find at least some data in cache.

The ideal working set (WS) would be to have the two assigned blocks (the actual and the next) into cache, so that for every iteration I have all the data I need locally. Actually this is possible only for very small blocks. Considering for each cell we need 4 byte (integer) and that the core cache has 32KB for the data, to be shared also among the 4 contexts, but let's consider we are using only one context per core.

# cells	WS size (Byte)	L1 cache (32K) fit?	L2 cache (512K) fit?
4,000 (N≈63)	2* 16 KB (next+actual)	Yes	Yes
64,000 (N≈252)	2* 256 KB	No	Yes
Any bigger		No	No

As can be seen the whole WS gets big quickly, unfortunately this algorithm goes through the entire block without much local or temporal reuse, the most we can achieve is to exploit the little temporal locality present.
During the computation of the ith row and expect to already have the rows i and i-1, this way in the inner loop we need to retrieve only one new row at a time, otherwise we need to retrieve three rows per row update.
A cache line contains up to 16 consecutive row cells.
Considering a cell^{i,j} update we use four cache lines, one for the write and read the three lines relative to the three rows. Careful in the memory data layout and alignment must be taken because false sharing is one of the most important sources of performance degradation and this happens if we have two consecutive rows of two different worker into the same cache line.

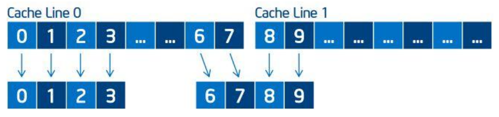


Figure 4. A row split into some cache lines.

Our regular memory access pattern naturally exploits prefetching.

For what has been said so far, the column block version does not fully exploit the prefetching using only a limited number of columns, the 1D segment block decomposition is balanced, but it does not prevent false sharing, splitting the same cache line content with two workers.
The aligned rows version is less balanced than the segment one, but prevents that different rows share a cache line, aligning each row begin to the cache lines, avoiding false sharing among workers.

An improvement of memory spatial locality is achieved with the shadow copy/halo border and near chunks to near UEs block mapping.
The shallow boundary/halo prevents that reading the boundary of a row needs a cache fault, in this case I already have in the current cache line the boundary point, an exception can be made for the last-column halo cell, which in an unlucky case resides in a new cache line, but given the access pattern can be prefetched accordingly. Without the halo, for each cell boundary checks and % ops are needed and for big enough rows one cache fault just to retrieve the boundary point.
The copy of the grid border let us manage the toroidal grid as a squared one, Figure 5, so the data access pattern can be more easily inferred and optimized by the compiler, we took out the modulo from the cell index computation. This also eliminate for each first element/row and each last element/row update the cache faults to read the corresponding border element, they will be already in the current cache line or at least prefetched.
With the 2D array grid, the first row and the last one are only shallow copies (we simply share the pointers), we can write them only once.
The writes remain local to the task block because who updates the border cells updates also the copies.

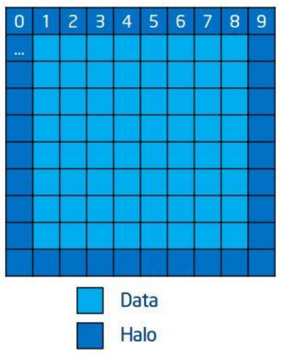


Figure 5. Grid with the border replicated into the halo cells.

Algorithm re-evaluation: the proposed algorithm is well supported by the target architecture and the target programming environment.

5. Supporting Structure design space

"Describe the software constructs or structures that support the expression of the parallel algorithm"

We start considering more concrete implementation aspects by exploring the "supporting structures" design space. The target programming environment is Fast Flow/C++.

As far as the "program structures" pattern group is concerned, we should look for patterns modeling the different concurrent activities individuated so far.
The number of workers used, the parallelism degree, will determine the performances of our application. When considering the implementation of each epoch, according to the "data decomposition" pattern, we have two choices that are worth being explored, the "SPMD" and the "loop parallelism" patterns.

Instead while considering the "data structures" pattern group we will pick up the "distributed array" for the grid management and the "shared data" for the random computation.

Loop parallelism

This pattern addresses the problem of transforming a serial program whose runtime is dominated by compute-intensive loops into a parallel program (sequential equivalent) where the different iterations of the loop are executed in parallel.

The overall program structure for this application is composed by a sequence of nested loops. We will apply the loop parallelism to the first inner for acheaving a row block decomposition or collapsing the two inner for to obtain a segment block decomposition.

```
for (long it = 0; it < iterations; it++)    {
    for (long i = 0; i < grid.n; i++)
         for (long j = 0; i < grid.n; j++)
        update (i, j, actual, next);
    swapGrid();
}
```

Listing 3. Algorithm structure.

The Fast flow parallel_for abstraction models the loop parallelism. This is implemented with a farm composed by an emitter and wrapped around workers.
Fast flow parallel for can statically split the loop iteration space equally and contiguously between nw workers, manages the implicit barrier before the swap and the worker threads behaviour among different iterations, Figure 6.
The split of the loop iteration space can be dynamic (round robin) or static; we will use the static partitioning which compute at the very beginning the partitioning and so do not use an emitter thread to send the tasks to the workers.

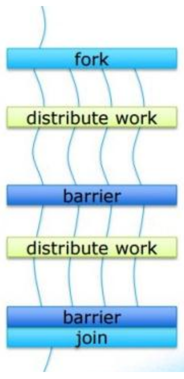


Figure 6. Parallel for unfolding.

When a large number of threads are involved, as in the case of the phi, the overhead incurred by placing

the thread creation and destruction inside the loop over the time steps would be prohibitive, the parallel_for does a fork at the first call and then simply distribute the work at consecutive calls, a call terminates as soon as the last worker end its task on the barrier.
Using an active thread wait between consecutive iteration (spinWait) and an active barrier synchronization (spinBarrier) can reduce the parallel overhead / serial fraction among the iterations.
The swap operation is protected out of the two update loop by the barrier.

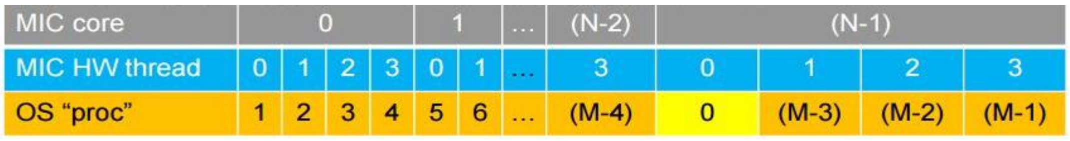
This approach is effective if the compute times for the loops are large enough to compensate for parallel loop overhead.

```
// step = 1 grain = 0 for static sched
// row block
  for (unsigned it = 0; it < iterations; it++) {
      pf.parallel_for(0, grid.n, step, grain, FbyRow);
      swapGrid();
 }

// row block with halo
  for (unsigned it = 0; it < iterations; it++)    {
      pf.parallel_for (1, grid.n - 1, step, grain, funCompute);
      swapGrid ();
 }

// segment block
for (unsigned it = 0; it < iterations; it++) {
      // explicit loop collapse, n = N*N
      pf.parallel_for(0, n, step, grain, FbySegment);
      swapGrid ();
 }
```

Listing 4. Loop parallelization with FastFlow according to the given decompositions.

The parallel for assume a SMP memory architecture, unfortunately we use a NUMA, but we try to overcome this assumption accordingly distributing and scheduling the tasks for the given target architecture.

The Intel® Xeon Phi™ coprocessor has N cores, each with 4 hardware thread contexts, for a total of M=4*N threads; The OS maps "procs" to the M hardware threads:



MIC core	0				1		…	(N-2)	(N-1)			
MIC HW thread	0	1	2	3	0	1	…	3	0	1	2	3
OS "proc"	1	2	3	4	5	6	…	(M-4)	0	(M-3)	(M-2)	(M-1)

The OS runs on proc 0, which lives on core N-1. A Rule of thumb is to avoid using OS procs 0, (M-3), (M-2), and (M-1) to avoid contention with the OS, only less than 2% resources unused (1/#cores).
We assign threads to core N-1 only when more than 236 workers are used. The maximum numbers of workers in a static scheduling configuration that can be run in parallel are 239, counting the OS. This can change if we use a dynamic scheduling with an emitter thread.
In the performance analysis we will look for 59, 118, 177 and 236 as key number of UE, because there we

change from k threads per core to k+1, and we can notice the performance impact of sharing the resources.

For the scheduling fastflow expose threadMapper::istance()->setMappingList (thread_mapping_string).
To use our scheduling policy we also set the icc flag –DNO_DEFAULT_MAPPING, otherwise workers are assigned to incremental contexts.
We define two main static scheduling policies:

- The first one assigns context "1, 5, 9…233, 2, 6… 236, 237, 238, 239, 0" noting the last four contexts. To use this policy MIC_MAP must be defined in the main.
- The second one which, based on the number of workers, assigns consecutive grid blocks to consecutive contexts, in a threads-per-core-balanced way. This minimizes the data distance into L1-L2 caches, combined with a first touch allocation. E.g. 61 workers will be assigned, in order, to the contexts "1, 2, 5, 6, 9, 13, 17, 21... 233" instead of "1, 5, 9, 13… 233, 2, 6". To use this policy MIC_MAP2 must be defined in the main.

"First touch" is a common NUMA page-replacement algorithms, the PE first referencing a region of memory will have the page holding that memory assigned to it. A common technique to exploit this strategy is to initialize data in parallel loops using the same loop schedule as will be used later in the computations.

```
// "touch first" initialization, favors data near the thread local caches
  pf.parallel_for_idx(0, grid.n, step, grain,
        [&](const long start, const long end, const long thid) {
           initGrid(in_grid, grid, start, end);
  });
```

Listing 5. First touch initialization with parallel_for, chosen a scheduling policy and a data layout.

At this point a model for the service time or the completion time can be formalized, given the number of workers P, and the size of the N*N grid.
As the parallel overhead (po) in the service time, we model the time needed to distribute the work and the time to synchronize on the barrier. The parallel for is built upon a farm with feedback, the cost model would be

$$Tservice(N,p) = \max\{Temitter, Tcompute(N,p)\}$$
$$Tcompute(N,p) = Tcellupdate * \frac{N*N}{p}$$

Given a two stage pipeline implementation, the emitter time estimates also the time for the barrier synch.
Using the static scheduling configuration without the emitter thread, we model the overhead to access the substituting data structure. This time is not anymore a two stage pipeline but only one with a collapsed cost.
In the static scheduling without emitter, the service time would be

$$Tservice(N,p) = Tcompute(N,p) + Tpo(p)$$
$$Tpo(p) = T_{parfor} * p$$

Where T_{parfor} is an average estimate of the management time per added worker.

The parallel overhead introduced is linear in the number of UEs and the inherent serial fraction needed for the management of the worker is repeated for each iteration. Amdahl law and its requirement to minimize a program's serial fraction often mean that loop-based approaches are only effective for systems with a smaller number of PEs.
The completion time over s time steps can be modeled as:

$$Ttot(N, s, p) = s\ (Tcompute(N, p) + Tpo(p))$$

Shared data & distributed array patterns

The random computation is done with the drand48() (about 28ns) which is a lot faster than the rand(); these function btw use for the computation a shared global state data structure, so in a concurrent application their call must be into a critical section, we manage it with a mutex.
This limits the scalability, the fraction of runtime spent during inherently serial work limits how many workers can be used (Amdahl law), the more the workers the more the serialized work summed by the introduced parallel overhead. We will observe this fact in the efficiency also.

The sequential equivalence of the program is maintained but cannot be checked with the results because of the probabilistic choices taken during the computation.

Given the decomposition done so far the management of the grids reduces to the correctness of the barrier and the subsequent "centralized" swap.

6. Implementation Mechanisms design space

Here we recall some specifics related with the used mechanisms that can be found in the project code, the code itself is well documented, so we do not spend a lot of time here.
The UE management is done with FastFlow parallel_for, parallel_for_idx.
The communication is via shared memory.
The synchronization with the barrier managed by the parallel_for and the critical sections are done with the std::mutex.
The data alignment is done allocating the data with mm_malloc(…, 64) where 64 byte is the cache line size for the data alignment. The free must be done with mm_free().

When using the aligned variable "var" the compiler should be informed with `__assume_aligned(var, 64)` or declare a loop to be aligned: `#pragma vector aligned`.

The `drand48()` is very efficient, an average of 28ns but thread not safe. The critical sections are minimized taking advantage of the lazy evaluation of the C++ if construct. This brings a lot of performance improvement.

```
// Lazy evalutation of rand()
if (died || dieprob())
```

Listing 6. Short circuit evaluation.

A single Xeon Phi core is slower than a Xeon core due to lower clock frequency, smaller caches and lack of sophisticated features such as out-of-order execution and branch prediction.

Xeon Phi can only perform memory reads/writes on 64-byte aligned data. Any unaligned data will be fetched and stored by performing a masked unpack or pack operation on the first and last unaligned bytes. This may cause performance degradation, especially if the data to be operated on is small in size and mostly unaligned. In the following, we list a few of the most common ways to let the compiler to align the data or to assume that the data has been aligned. The phi has also L2 hardware prefetch.

SIMD and vectorization cannot be used, because we compute a point value instead of a vector value.
Prefetching can be inhibited with the icc flag `-no-prefetch`.
Let the compiler know there are no loop carried dependencies, but only affect compiler heuristics with `#pragma IVDEP`.
Let the compiler know the loop should be vectorized, but only affect compiler heuristics with `#pragma VECTOR`.
Force the compiler to vectorize a loop, independent of heuristics with `#pragma SIMD` or `#pragma vector always`.
Function inlining and is also been used to improve the sequential code performances.

7. Performances

The comparisons among performance predicted with the performance models and the one observed during the experiments

“. . . speedup should be measured by scaling the problem to the number of processors, not by fixing the problem size” Gustafson-Barsis’ Law.

For the project also other variations have been developed, but the ones presented here are the most significant for the performance comparison.

The general methodology adopted to obtain more accurate measures is to separate the first iterations of the parallel_for, eliminating so the time of the threads generation and the data assessment in the caches.
The observed service time is averaged over some iterations. Because of the variability of the system and memory workload and the sequence of the critical sections, it is better to do also independent observations to eliminate the outliers. The more time steps we go through the better the service time approximation.
The parallel for in the first call, just create the threads, for a good evaluation of the performances it is recommended to not include the first iterations, where the threads are created and the cache starts to fulfill itself.

Now we measure the parallel overhead introduced by the parallel_for in different configurations, Figure 7, so we can estimate with the cost model the expected behavior of the program, with the mic mapping.

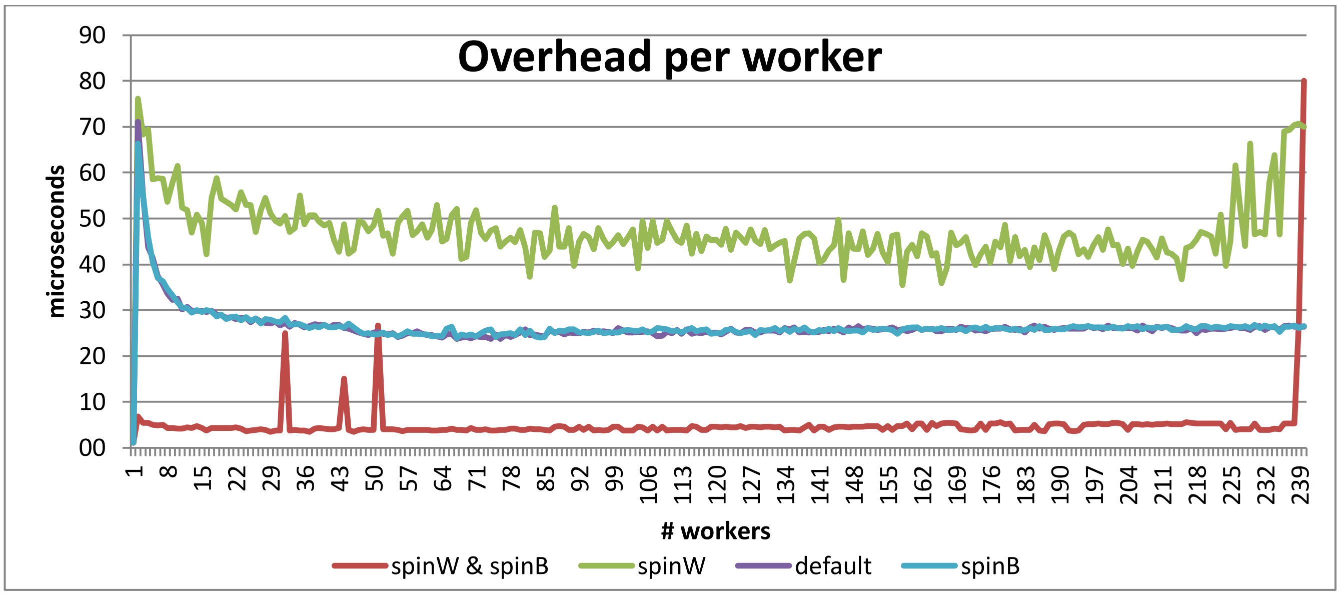


Figure 7. Comparison among the different parfor configurations.

We choose the configuration with spinWait and spinBarrier which gives the least overhead per worker, of the four configurations the ones with spinWait give the most variability, these configurations are more influenced by the system workload than the others, this can be observed also in the variance of the observations. Independently of the configuration if we use only 1 worker the overhead is 1 μsec, if we use more worker than supported UEs the overhead increases a lot, due to the serialization of not concurrent threads. This means that if we approximate the sequential time with the time with one worker we add only one microsecond per iteration to the total computation. To be noted that the spinW&spinB can have a lot of outliers, this will imply some peaks in the performance measures.

Now we measure the sequential time of the implementation of some proposed decomposition, varying the size of the grid, so we can see how much the memory layout and data access pattern impact the performances.

We compute the average time to update a cell over Niter time steps and an N*N grid using nw workers as:

$$Tcell = \mathrm{nw} * \frac{Ttot}{niter * N * N} \ (for\ overhead\ comparison)$$

The plotted time is an average over independent runs.

First we show the update computation time per cell, for some of the data representation and decomposition previously described.

The comparisons are among:

- 1D array grid layout
  - segment wise decomposition
  - row wise decomposition
- 2D array grid layout
  - row wise decomposition with rows aligned to cache line with shallow boundary

In this sequential comparison, Figure 8, there are no false sharing effects and memory movements or contention among workers. The row aligned version as expected is the best performing, this is given by the less instruction needed to manage this decomposition. The row decomposition have less instruction to do and win with the segment decomposition, which also as already described have more The padding to make the row aligned to cache line, does not bring any improvements in the sequential tests.

As the grid size grows the WS gets bigger than the caches, but the regular and contiguous array access pattern is prefetching friendly, which mask the memory access time. As the ocean size gets bigger the average computation time per cell remains constant. The best performing as expected is the 2D version with aligned row and boundary replication.

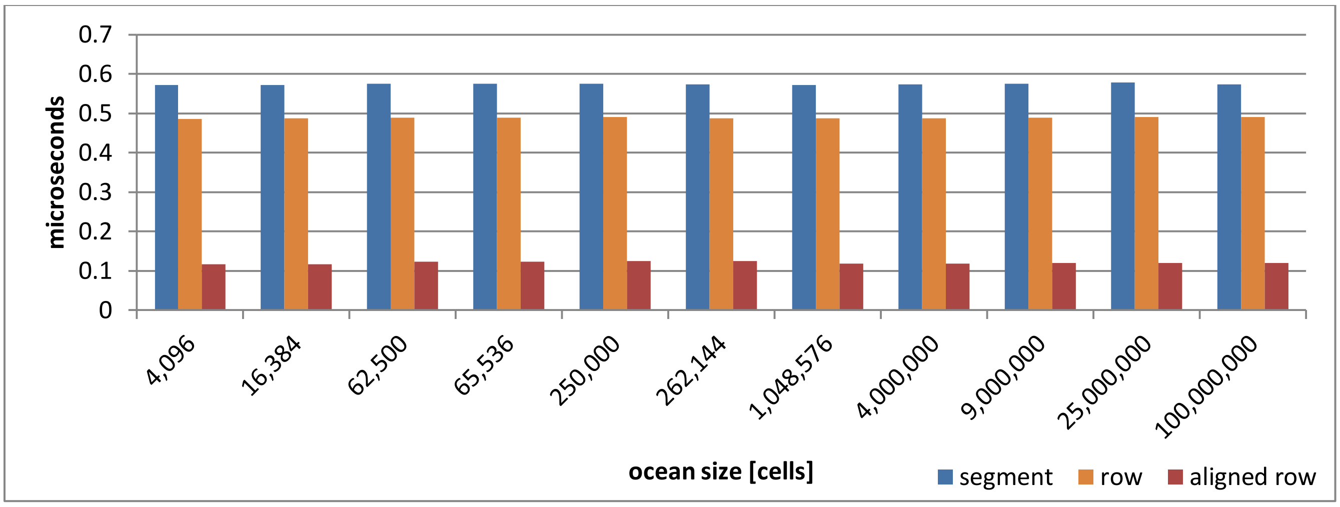


Figure 8. Cell update sequential time comparison changing the size of the problem.

We chose to further analyse the performances of the aligned row version, doing a comparison of the overhead introduced by using more workers and the different scheduling policy. The comparison is done using 59 118 177 and 236 workers which as seen respectively use from one to four threads per core. The scheduling policy are the default, the mic_map and the mic_map2, Figure 9.
As expected as we increase the number of threads increases also the parallel overhead and the memory access pattern penalty. As we increase the grid size the grain of the computation increases and the impact of the parallel overhead is reduced, but increases also the size of the memory to transfer from one core to the other, this explains why the mic_map2 scheduling which assigns near workers to the same or near cores have better performances.

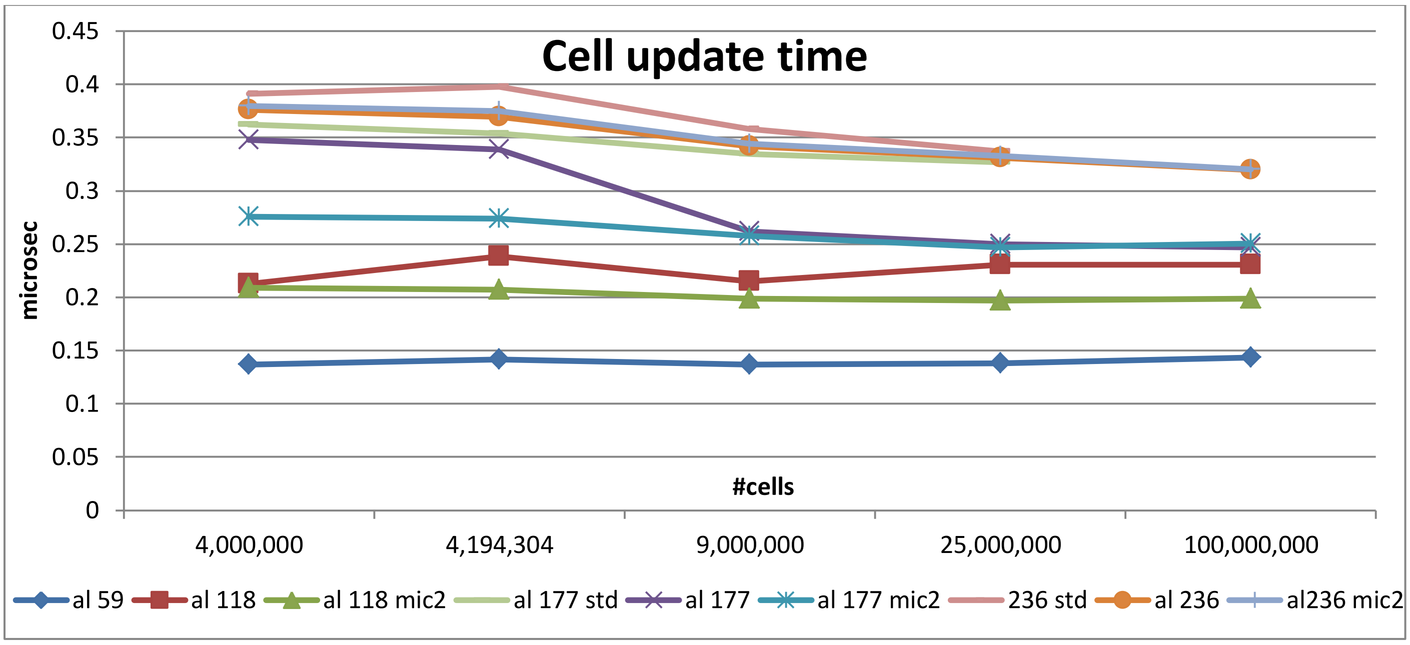


Figure 9. Cell update time comparison varying the grid size among different scheduling and #workers.al stands for aligned row, std for standard scheduling and otherwise mic scheduling.

Figure 10 and Figure 11 contain the speedup and efficiency varying the number of workers for the aligned row decomposition. We can observe the improvement of having 1, 2, 3 or 4 threads concurrently running on a core, respectively less than 59, 118, 177 workers, in the speedup/efficiency charts we can appreciate the effect of using one or more threads per core.

In the efficiency chart we find plateaus and analogously a constant improvement in the speedup, showing that the penalty of using one more UE are somewhat constant. This happens because there is less computational power for each thread in a core and more concurrent memory accesses conflict between the local working sets, noting that the more concurrent accesses to the caches the more cycles it needs to get the data.

Lets see why when we go from k thread per core to k+1 we have a decrease of the efficiency and speedup, this is observable in the "limit" points. The improvement given by this added UE does not compensate the introduced overhead; indeed some more UEs are needed to compensate that overhead. The overhead is given by the concurrent local resource access and share (L1 and L2 caches and processing power). The other jumps are given by variations on the system workload, independent runs shows that for a given configuration the time fluctuate more the more threads are involved, as already seen in the fast flow par for monitoring. If we use a more stable configuration of the parallel for, as without spinWait, the fluctuations decrease, but the mesh grain should be made proportional to the introduced overhead.

Load imbalance also impacts the performances, the work distribution is not optimal, some threads are loaded while others idle, and the slowest thread determines total speedup.

Also the critical section degrade the performances, locks prohibit threads to concurrently enter the code regions, effective serial execution.

Comparing the speedup and the efficiency given by grid of different size we can see that the growing pattern is the same, but with a little more stability and less susceptible to the parallel overhead variations.

The size of the compared meshes is chosen simply to make the computation time bigger enough than the parallel overhead.

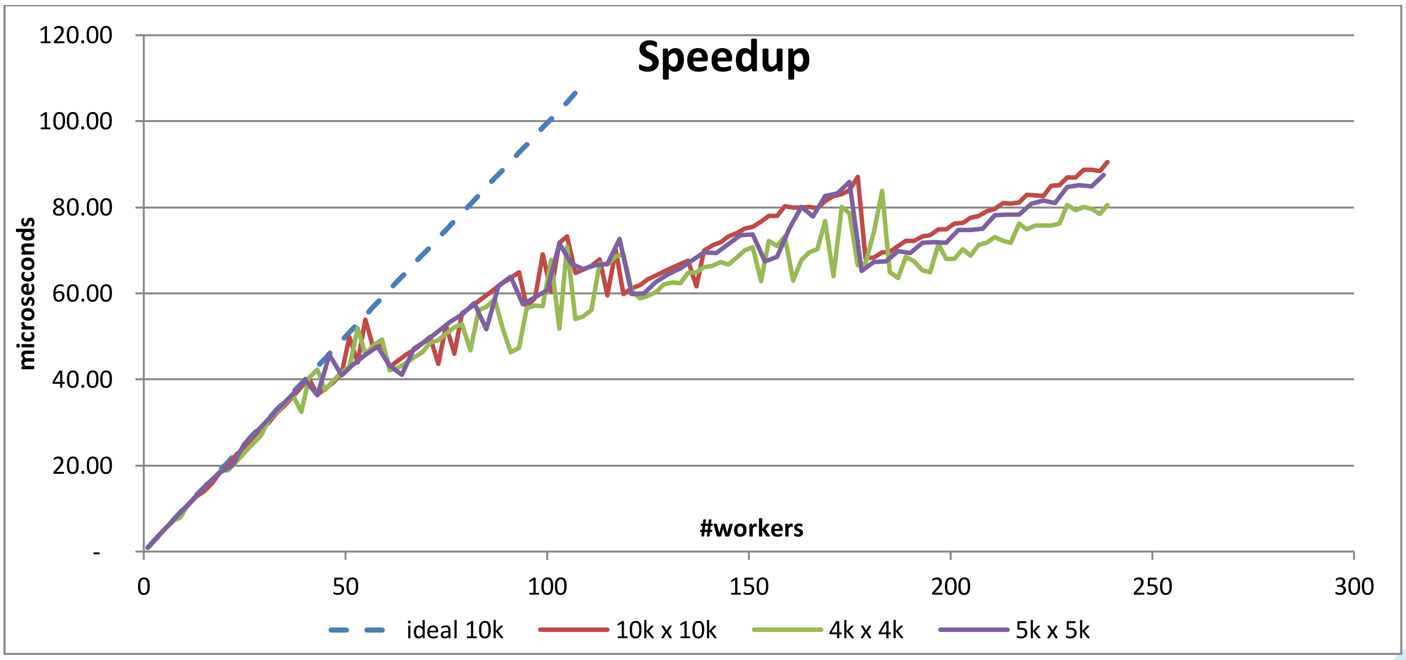


Figure 10. Speedup varying the #workers respect the one worker completion time.

During the experiments we measured 45% efficiency/improvement by adding a 3rd thread to a core and

35% adding the 4th thread; while before 50 workers we are over 95%, and then it starts decreasing as the memory system gets saturated and as the probability of a serialized critical section grows, going to 70% as we start using multi-threading.

The unlucky peaks in the charts are leaved to better appreciate the general behaviour and some unlucky runs caused by the worker synchronization and the fluctuations in the system workload, same worker configurations in distinct run show this.

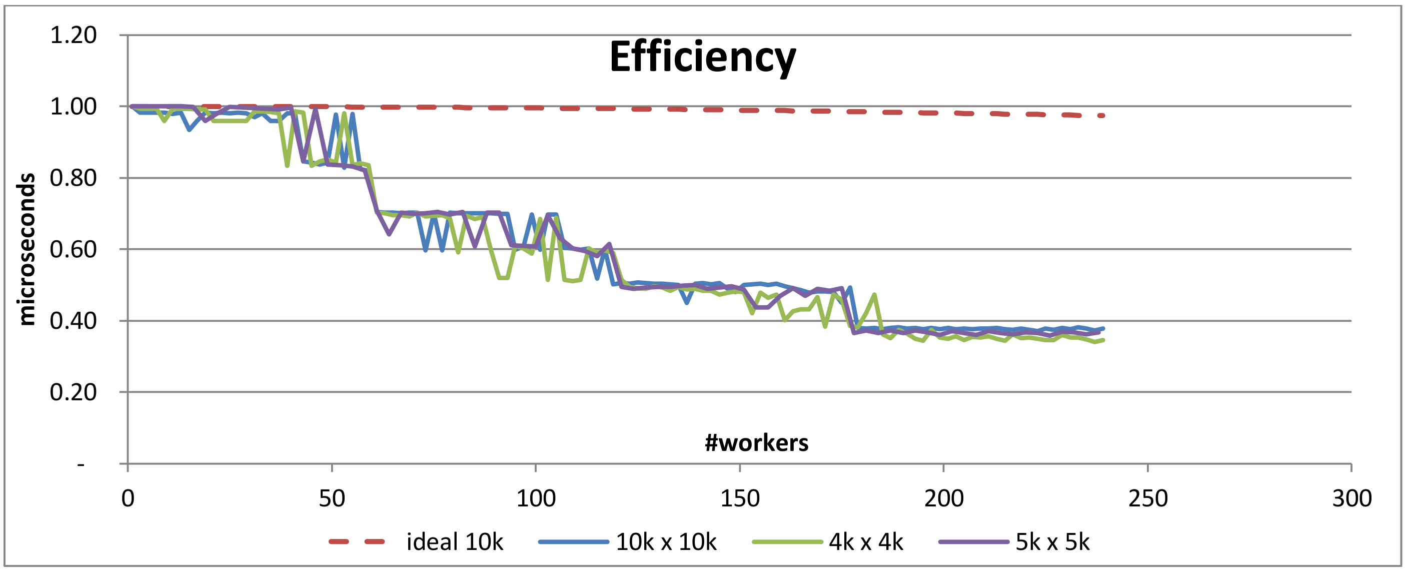


Figure 11. Efficiency varying the #workers respect the one worker completion time.

Concluding that an ideal speedup and efficiency is obtained until the workers are allocated to a new core, then the speedup and the efficiency degrade linearly with the number of threads per core, when multithreading is used. This happens because of the application use of the memory resources and the intrinsic serial section made of mutually exclusive random computations.

8. USER MANUAL

How the code may be compiled and run

Copy the files fsf6b.cpp; fsf5coll.cpp; initGrid.cpp; Makefile; gridBuilder.cpp; runtests.sh into the home or another folder of the host system.
Specify into the Makefile variable `FF_HOME` the path of the FastFlow library folder (it is not included in the .rar).
To build sequential version of the application use `#define SEQUENTIAL` into fsf6b.cpp.
To print the output times in microseconds use `#define MICROSEC` otherwise milliseconds will be used.
To run the application with the mic mapping use `#define MIC_MAP or #define MIC_MAP2` otherwise the default scheduling will be used.
To run the fast flow parallel overhead test use `#define FFTEST`.
Execute Make all.
To create an initialized NxN grid run the gridBuilder program:
`./gridBuilder <N> <new-file-name>`.
e.g. `./gridBuilder 5000 5k.m`

Copy the initialized grid file(s) and the program file (fsf6b or fsf5coll) into the mic via scp.
Application (./fsf6b or ./fsf5coll) usage:

```
./fsf6 –m <grid-path> –i <iterations> –w <num-workers>
-i number of time step iterations to do
-m initialized grid file name (and path if needed)
-w number of workers
```

And as output will appear: `N'\t'iterations'\t'num_workers'\t'Tcomplete"\t"tcellupdate`
If FFTEST is defined: `N'\t'iterations'\t'num_workers'\t'Tcomplete"\t"TperWorker"\t"Tforkperworker`

If you run the program with the wrong parameters the help will appear.

Alternatively you can use the parametric script "runtests.sh" to build the application, copy the file into the mic, run the application and create a log file with the values printed by the program in a (tab separated) format.
The tests needs to be modified (in the for loop set) based on the grid files built and available on the mic. The number of worker configurations to tests should also be modified accordingly (in the for loop set). There is also the possibility to modify the for-cycle relative to the iterations; in this way multiple runs can be made with various iterations.
Testing script usage:

```
./runtests.sh -p <program-name> [-c <coprocessor-name>]
-p program name
-c coprocessor name (default mic1)
```

e.g. `./runtests –p fsf6b –c mic1`

BIBLIOGRAPHY

[1] Timothy G. Mattson, *Patterns for Parallel Programming*.

[2] Michael McCool, Arch D. Robison, and James Reinders, *Structured Parallel Programming (Patterns for Efficient Computation)*.

[3] M. Danelutto, *SPM course notes*.

[4] Intel. Intel. [Online]. https://software.intel.com/en-us/articles/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-1-optimization

[5] PRACE. PRACE (Best Practice Guide – Intel Xeon Phi). [Online]. http://www.prace-ri.eu/best-practice-guide-intel-xeon-phi-html/#id-1.13.5