SHYFEM Parallelization:

An innovative task approach for coastal environment FEM software

Supervisor:

Carlo Cavazzoni

Co-Supervisor:

Stefano Salon

Georg Umgiesser

Candidate:

Eric Pascolo

To Adele
my little,little,little baby!

Contents

1 Introduction **1**
1.1 Shyfem and its use . . . 1
1.2 Hardware and software stack . . . 3
1.3 New approch for shared memory computing torward exascale . . . 6

2 Parallelization and Restoration **9**
2.1 Software Analysis . . . 9
2.2 First Parallelization . . . 11
2.2.1 Subroutine parallelized . . . 11
2.2.2 Benchmark . . . 12
2.2.3 Analysis . . . 14
2.3 Task Parallelization . . . 18
2.3.1 OpenMP task . . . 18
2.3.2 New parallel design . . . 19
2.3.3 Hydrodinamic task . . . 20
2.3.4 Scalar task . . . 21
2.4 Another level of parallelism . . . 22
2.4.1 Refactoring of conz3d . . . 22
2.4.2 New data distribution . . . 23
2.4.3 Nested Task . . . 24
2.5 Debugging . . . 28

3 Conclusion **31**
3.1 Results . . . 31
3.2 Future developments . . . 33

A Code **35**
A.1 First parallelization . . . 35
A.1.1 sp256v . . . 35
A.1.2 conz3d . . . 35
A.1.3 Simple program . . . 36
A.2 OpenMP Task . . . 37
A.2.1 Hydrodynamic task . . . 37
A.2.2 Scalar Task . . . 37
A.3 Modern conz3d . . . 40
A.4 conz3d_omp . . . 40

Chapter 1

Introduction

SHYFEM is a finite element hydrodynamic code [5] written by Georg Umgiesser in the 80s to model Venice lagoon for his master thesis; its development has been continued by CNR-ISMAR group. It is one of the few opensource codes for coastal areas that use a finite element approach. *SHYFEM* is a very important resource because it is focused on coastal areas and can be coupled with other software in order to increase the simulation accuracy in such areas. Coastal areas are strategic because many human activities are here concentrated. This means that a software that produces an accurate representation in coastal areas may also advantage socio-economical activities. *SHYFEM* has been already and successfully applied to several coastal and lagoon environments; for example, it is used to produce tidal forecasts in the Venice lagoon[5] and other lagoons in the Mediterranean sea[6]. It is also used in the Danube Delta[2] and to estimate its effects on the Black Sea, and in Malta to produce coastal forecasts. The main goal of this work is to obtain a new version of SHYFEM that may be faster, parallel, capable to use efficiently modern hardware, and easily coupled with other software.

1.1 Shyfem and its use

SHYFEM [1] is a 3d finite element hydrodynamic model, that includes a wind wave model and with Eulerian and Lagrangian active tracers transport and diffusion simulator. The code is hosted on github:

`https://github.com/SHYFEM-model/`

SHYFEM uses a semi-implicit algorithm for integration in time. It is not completely unstructured grid in all dimensions, because it uses 2D finite element model (FEM) grid in the horizontal and simulates z direction with repetitions of the surface grid. It is possible to simulate friction, which is computed introducing the Strickler formulation, so that the friction coefficient varies with water depth. With *SHYFEM* it is possible to simulate turbulences through the model GOTM, that

[1] www.ismar.cnr.it/shyfem

is state of the art in this field. Another model coupled with hydrodynamic part is wind wave model that allows the computation of generation, propagation and dissipation processes in both coastal areas and open ocean.
In *SHYFEM* it is possible to specify both open and close boundary conditions; the open boundary are prescribed in accordance with the Dirichlet condition while the close boundary have normal velocity set to zero and tangential velocity as free parameter: in this way the software can solve transport variables systems explicitly without solving any linear system.
Another feature often used consists on an Eulerian and a Lagrangian transport module that simulates the advection and the diffusion of active tracers in the waters. The two approaches allow to simulate the trajectory and the reaction of both the dissolved chemicals and the dispersed tracers. This part of software can be coupled with reactors, in example EUTRO,WASP and BFM.

The software is distributed in various directories, but the part of our interest is in the fem3d directory. The main file of *SHYFEM* is shyfem.f and it uses now a FORTRAN module, but when this work started it used a common block. The compilation is possible through a Makefile that reads the configuration parameters in Rules.make file. In Rules.make user can choose:

- FORTRAN and C compilers : users can choose INTEL or GNU
- to activate OpenMP parallelization
- which solver to use for matrix solution (gauss, sparskit, pardiso)
- to activate or not the turbolence model GOTM
- the type of coupled ecosystem model
- debug and optimization flag options

Test case

In order to explain how the software runs, we take in consideration the paper [2]. This is the first study done about a circulation system near Romanian coast. In this case the use of FEM model increases the spatial resolution and gives a better representation of the coastline. This is one of the most important cases of success of *SHYFEM* . The mesh used as input is reported in figure 1.1: *SHYFEM* can use grids with varying resolution: in this case the resolution gradually decreases as it gets further from coast. The input is not a three dimensional mesh but the surface is a two dimensional mesh and under water the sea is subdivided in levels with the same shape of surface, in other words each element is like a big prism having triangular base, and the base and its vertices are defined by the surface mesh and the height (or number of levels), depending on bathymetry of basin. This is a very good set up to explain the work done, since we can observe the computational step and output. *SHYFEM* can calculate:

- Hydrodinamic fields, for example current speed field

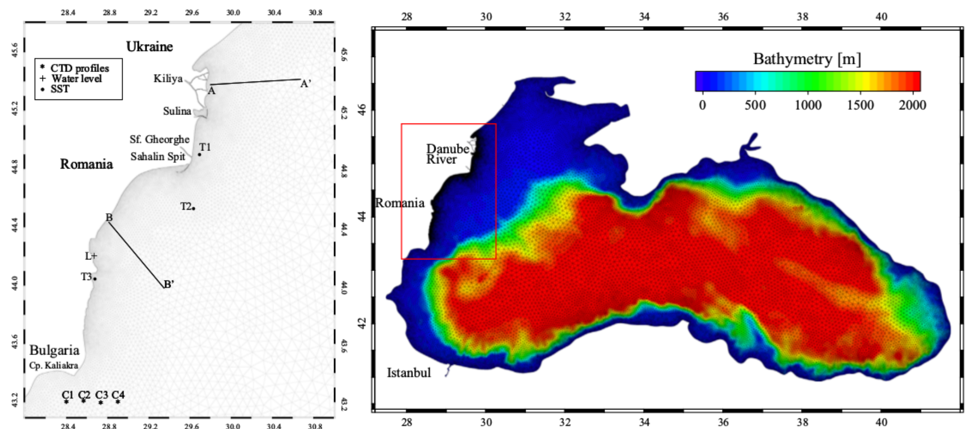


Figure 1.1: Black Sea, on the left the input mesh, on the right the bathymetry map (adapted from [2])

- Temperature and Salinity profiles (from here referred to as TS)
- Biogeochemical tracer (from here referred to as tracer)

We can image the software is split at each time step in two parts: the first is the *hydrodinamic part*, that it calculates the current field on each element; and the second one is the *scalar part*. The second one uses the fields calculated by the hydrodinamic part as input, and compute for each mesh node temperature, salinity and biogeochemical tracer (from now on, all of these will be referred to as "tracer", because the computation is the same). In the *SHYFEM* code, the computation of T, S and tracer uses the same subroutine, so we can consider all of them as *scalar* type. In figure 1.2 we can see the current field plotted and the salinity plotted as color-map; *SHYFEM* computed a complete three dimensional output, and users can obtain both horizontal (upper figures) and vertical (bottom figures) information about current fields and the salinity profile. Figure 1.3 shows a case study which allows to better understand which geographical part of the Danube Delta gives the largest contribution in each year period, it is possible to do this kind of calculation with the tracer part. With SHYFEM users can simulate the diffusion of tracers released in a certain point of the mesh. In this case the authors released three tracers in different outsources: one in Kiliya (blu trace), one in Sulina (red) and one in Sf.Gheorghe (green). Figure 1.3 shows that in spring the major contribute to Black Sea is due to Kiliya outsource.

1.2 Hardware and software stack

To develop the new version of *SHYFEM* we use an HPC Cluster named PICO, it is located in CINECA, the main Italian supercomputing center. In this section we want to describe the cluster and software stack used for development.
PICO HPC Cluster is Linux Infiniband cluster composed by 74 nodes not homogeneous. Each node is design to do a specific task, however in general PICO

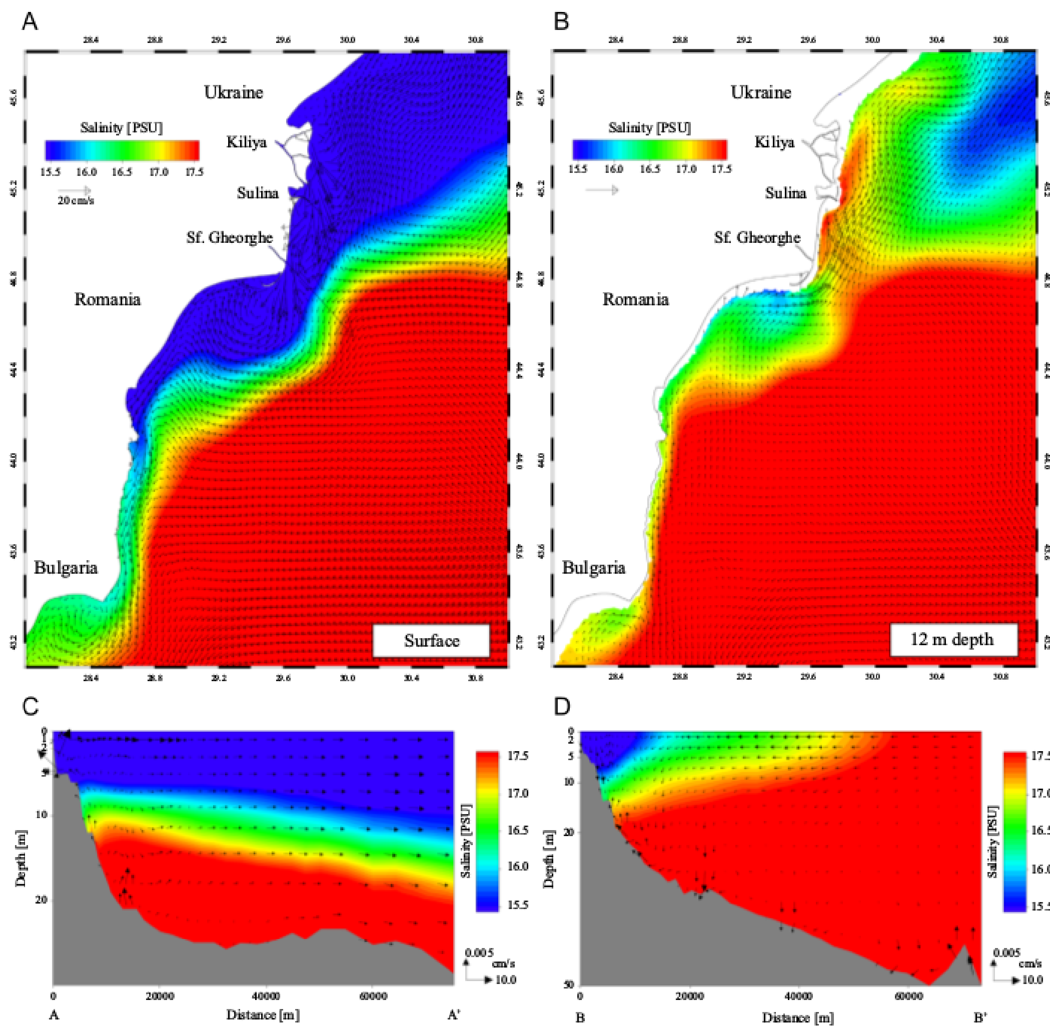


Figure 1.2: Black sea current speed and direction fileds. In color salinity is plotted . In figure A the values are referred to the surface, in figure B the values are referred at 12 m depth, in figure C and D two vertical profiles are plotted , at AA' BB' (see segment in figure 1.1)

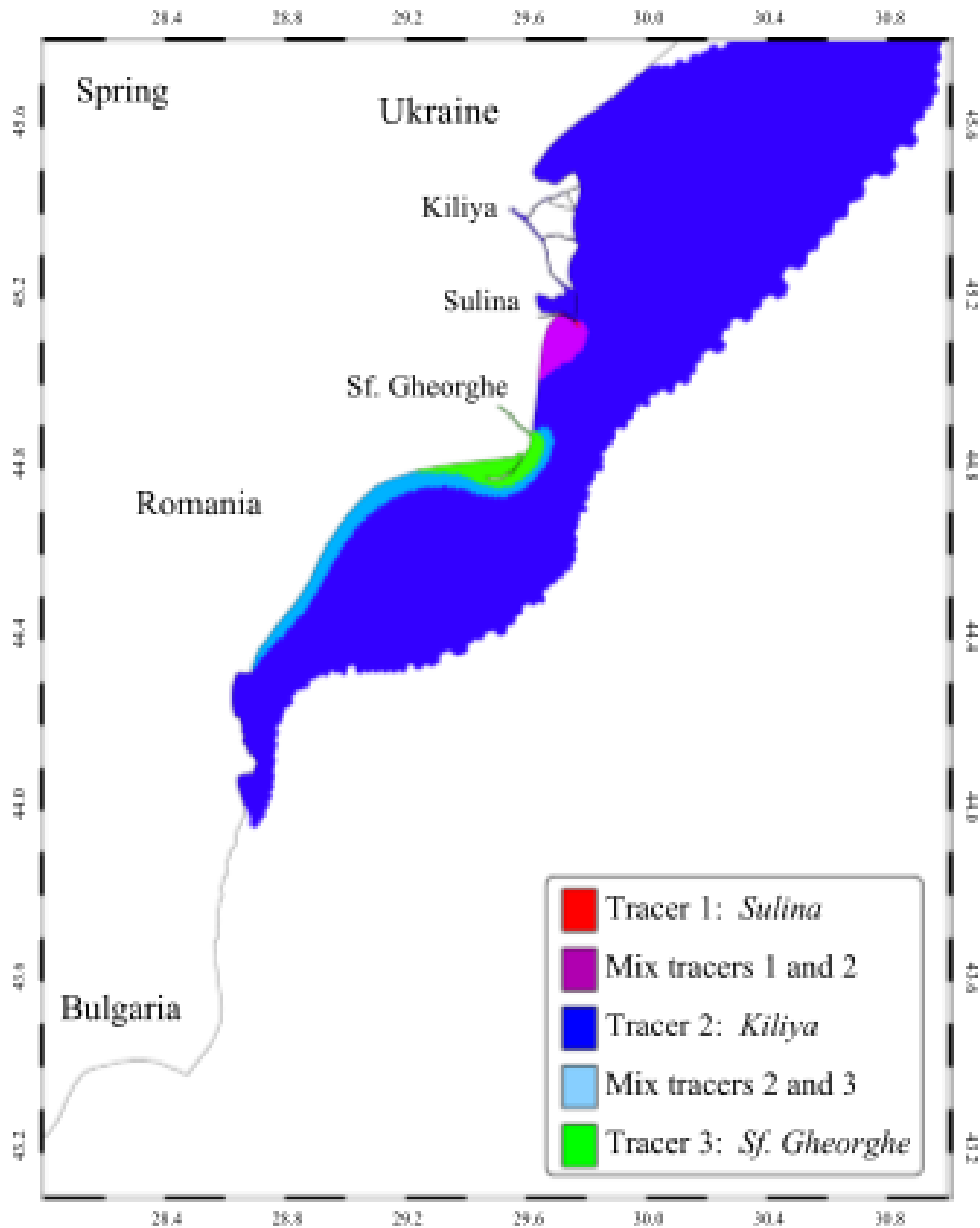


Figure 1.3: This picture shows a study which allows to appreciate and better distinguish the different contributions of water outlets to Danube Delta. This kind of estimation can be done with the tracer part.

is mostly used on High performance computing and data analytics and software development . PICO is composed by:

- Login nodes :
  2 x (2 x Intel Xeon 10 Core E5-2670v2 2.50 GHz, 128 GB mem)
- Compute nodes :
  51 x (2 x Intel Xeon 10 Core E5-2670v2 2.50 GHz, 128 GB mem),
  14 nodes is reserved to OpenStack
- Visualization node :
  2 x (20 core, 128 GB mem, 2 GPU Nvidia K40)
  2 x (20 core, 512 GB mem, 1 GPU Nvidia K6000s)
- BigInsights nodes :
  4 x (16 core, 64 GB mem, 32TB local disks)
- Big memory nodes :
  1 x (32 core, 512 GB mem)
  1 x (40 core, 1024 GB mem)

The compute and visualization nodes contain 2 Intel Xeon ten-core processors (E5-2670 v2) with a clock rate of 2.5 GHz capable of 8 FPO per cycle and Intel hyper-threading technology is disabled. All the nodes are interconnected through a Melanox Infiniband FDR 56 Gb/s network , allowing for a low latency/high bandwidth interconnection. PICO storage system is based on IBM GSS technology and allows to reach a speed of 40 GB/s. The results of this work have been obtained on the compute nodes of PICO, except for the performance analysis obtained by Intel VTune XE , whose GUI is installed only on the Visualization node. The performance analysis is obtained from Intel VTune 16, SHYFEM was built using the compiler from Intel Composer XE-2016 (version 16.0.0).

1.3 New approch for shared memory computing torward exascale

The next generation of High performance computers will possibly reach one exaFLOP: the upcoming era is thus called "Exascale era"[4]. In order to reach 1 exaFLOP of peak performance the new machines are probably going to have millions of CPU cores, 1 peta byte of RAM overall and 1 ExaByte/s of aggregated memory bandwidth; however, there is a constraint on the energy consumed in order to maintain the economic sustainability of the total cost of ownership. If we take an optimized and parallelized code (MPI+OpenMP, for example) and we run it on an exascale cluster, we probably will obtain a very bad performance since the bandwidth is not sufficient. Furthermore, it is not possible to increase the bandwidth due to a constraint in energy consumption.
Given such premise, the HPC vendors, and more generally the HPC community must adopt new parallelization techniques in order to minimize the use of band-

width and therefore increase the scalability of the code. A promising paradigm that offers solutions, addressing the reduction of synchronization point in the application and latency-hiding, is supported by the task based parallelism, that is MPMD[2] approach. When the software encounters a specific directive, it spawns a bunch of work named task; this kind of parallelism is a MPMD , because tasks contain both instruction and data so thus they have a big footprint in memory but reduces notably the synchronization point, because it is in general loosely coupled. This kind of parallelism is called unstructured parallelism because, if the programmer does not specify, the tasks are uncorrelated among them and the temporal order of execution is not important. Task approach reduces also the sleep time and the unbalance among cores due to the use of dynamic scheduling of computational work : this mechanism assigns the work to the cores on the base of the availability of data for processing, thus producing a directly acyclic graph with the possibility to explore this graph in many ways without block execution or putting a core in wait. Until now, the task technology is not enough widespread since in most cases programmers must rewrite at minimum the code, and in the worst case the whole algorithm.

[2]Multiple Program Multiple Data

Chapter 2

Parallelization and Restoration

In this chapter we report about our work that has allowed to obtain a parallel version of *SHYFEM* . We started from version 7.1.10 and all change done for parallelization are implemented in version 7.3.15. The first step of this work is done in parallel, we analyzed and done the first trial of parallelization, while ISMAR group transform the memory management from static with common blocks to dynamic with modules variable. This modernization permits a easier implementations of the parallelization. The two first sections of this chapter are referred to old *SHYFEM* version with common blocks, while the other section is referred to the dynamic allocation version. The transformation from static to dynamic left all variables global because the building of tree structure of memory is out of scope of this thesis. All modifications implemented have been subject to the regression test created by ISMAR group in order to maintain the correctness of scientific results.

2.1 Software Analysis

The initial step was the analysis of *SHYFEM* version 7.1.10.

The original code was based on procedure paradigm manly in FORTRAN 77 and FORTRAN 4 but years of programming experience have demonstrated these languages are not suitable to safely support shared memory programming paradigms. This is a notable weak point.

In the structure of the code we found other criticalities:

- only two OpenMP parallel regions were present but these are not effective;
- all variables are stored statically in common blocks;
- the software is composed by many different executables;
- the input dataset is split in many file and so it's error prone;
- IO and computational kernel are mixed together;

Function	Effective Time
conz3d	24.50%
tvd_fluxes	11.70%
dgelb	9.20%
vertical_flux_ie	5.20%
conzstab	4.20%
sp256v_intern	3.50%
tvd_get_upwind_c	2.80%
femintp	2.10%
assert_min_max_property	2.00%
is_r_nan	2.00%
nantest	1.80%
ilut	1.30%
mass_conserve	1.30%
loccoo	1.10%
locssp	1.00%
uvtopr	0.90%
....	

Table 2.1: Nador bechmark profiling, *SHYFEM* 7.1.10

We profiled the code with Intel VTune XE using Nador benchmark (not yet explained, see section 2.2.2) and were able to profile almost all features of the code. In table 2.1 are reported the results obtained on PICO with a serial run. Examining the code and considering the data in table 2.1 we can note that there are 3 parallelizable subroutines:

- conz3d
- conzstab
- sp256v

The most easy strategy to parallelize the code seems to split the elements and nodes associated of the grid among the PEs[1]. Inside the subroutines previously mentioned there is a loop over mesh elements, inside these loops other time consuming subroutines are called, for example dgelb in sp256v or tvd_fluxes in conz3d. The only way to improve performance of these nested subroutines is to optimize them, because there are just in a parallel loop.

[1]Processing Elements

2.2 First Parallelization

2.2.1 Subroutine parallelized

First of all we tried to parallelize with OpenMP thread approach the three most time consuming subroutines mentioned before. The easy and faster solution is to parallelize the computation over the elements of the mesh because in general each elements computation is independent to each other. The only problem that we could encounter is the concurrency access to memory.

sp256v

This subroutine A.1.1 solves an approximate solution to the Navier-Stoke equations and can be improved in a later stage. In this case we did a trivial parallelization on the loop over the elements of the grid (line 6 A.1.1) that contains a call to the function **sp256v_intern**. This function does not require a synchronization call over threads, because for each element the computation is independent. **sp256v_intern** contains only one call to another subroutine, **dgelb**, that solves a band system. **dgelb** is also one of the major time consuming subroutines, but we didn't parallelize it because is yet very fast ($10E-5-10E-7sec$) and all type of parallelization we can add increases the execution time.

conz3d

This subroutine computes new concentrations of tracer for each time step. First of all, the value for each elements is updated on its nodes, the updated values compose the linear system solved within the same subroutine. This algorithm is coded in two loops working on the same four arrays which create data dependencies among the loops. Therefore, the loop fusion technique to increment computational intensity of each iteration is not allowed and we enforce parallelism in two loops as shown in A.1.2. Outside of the parallel section there are only the variable assignments, while inside the parallel section we have three OpenMP constructs. The first OpenMP construct is an OMP_WORKSHARE that initializes to zero the shared matrix; the second is an OMP_LOOP over the elements; the last one is an OMP_DO over the nodes of the elements of the grid, which solves the linear system that has been set up in the previous loop. In order to solve the linear system over the nodes we need to accumulate 4 matrices $chigh, cn, cdiag, clow$ with an OMP_REDUCTION, because each element shares at least one node with other two elements. It is not easy to parallelize this subroutine, since, in addition to the synchronization, we had to manage reading and writing on many common blocks, which are used by the nested functions called from the loop on the elements: the use of common blocks generates many locks that doesn't allow the optimized access to the memory. To minimize data races we use a temporary array for common block when writing, and use direct read for the others. With regard to the call

num th	time to solution (sec)	spin time(% cputime)	Speed up
1	35.7	0	1
2	29.38	10	1.2
4	22.02	54	1.6
8	18.99	70	1.8
16	17.40	81	2.0

Table 2.2: Nador time to solution and spin time

of other subroutines, we tried to optimize the **tvd_fluxes** subroutine substituting four IF statement with arithmetic operations. With this optimization we achieved to pass from 12 % to 7 % of wall time spent in this function.

conzstab

Conzstab subroutine has the same structure as **conz3d,** but, unlike this, it has less workload and hasn't a function call. The parallelization of this subroutine is identical to the one explained previously.

2.2.2 Benchmark

In order to profile and understand the performance behavior of *SHYFEM* we considered three different benchmarks. We discuss the motivation and results of each benchmark in the next section.

Nador Benchmark

Nador is a benchmark based on Nador lagoon(Morocco); the dataset of this benchmark is pretty small, with 3289 elements, 1890 nodes, 8 levels and 3 trackers and 1481 time steps. In table 2.2 the strong scalability and the spin time are reported, while figure 2.1 is a screenshot of Intel VTune XE reporting the threads analysis and the timeline of simulation. In this benchmark the time spent in parallel region is 60%.

Black Sea Benchmark

The Black sea benchmark has 83938 elements, 43823 nodes, 27 levels, and performed 15 time steps. This benchmark is same describe as test case in section 1.1. In table 2.3 the strong scalability and spin time data until 8 threads are reported, in table 2.4 the time to solution and mean duration for each parallel section are reported. In this benchmark the time spent in parallel region is 39%.

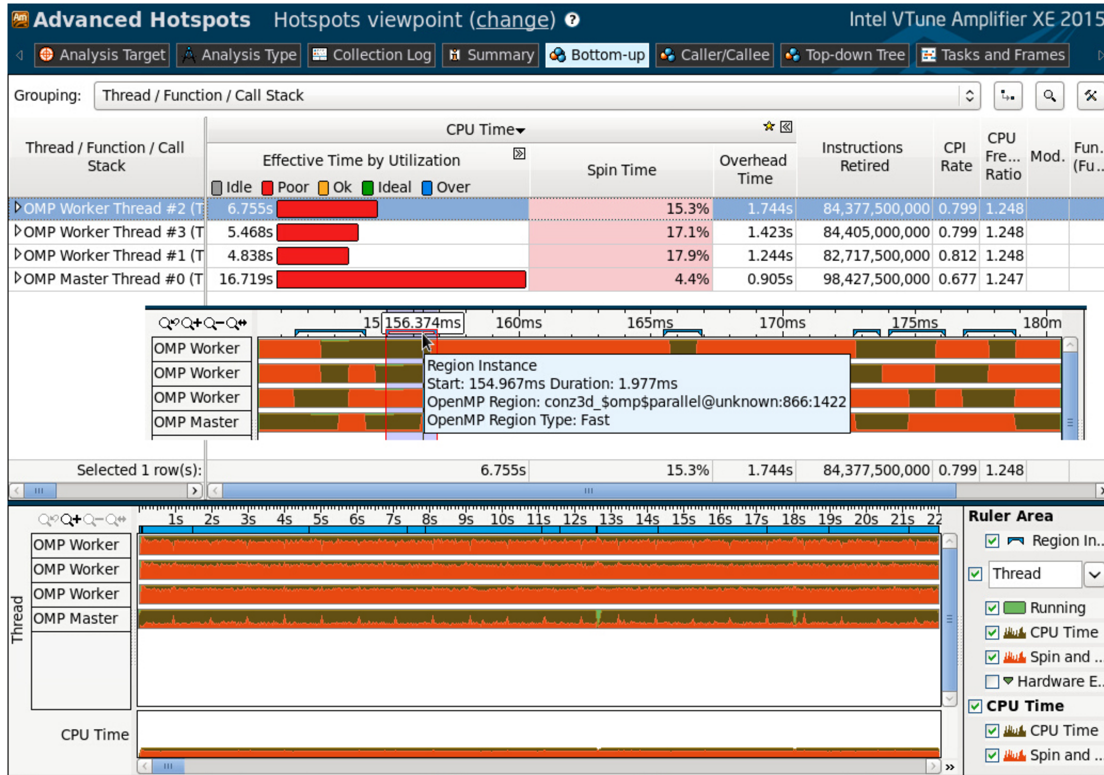


Figure 2.1: Nador benchmark, run on 4 cores and zoom on timeline

num th	time to solution (sec)	spin time(% cputime)	Speed up
1	46.54	0	1
2	37.06	24	1.3
4	33.68	35	1.4
8	33.10	53	1.4

Table 2.3: Black Sea time to solution and spin time

CONZ3D (A)

num th	time to solution (sec)	mean time (ms)
1	12.43	414
2	7.17	262
4	4.94	180
8	4.44	150

CONZSTAB (B)

num th	time to solution (sec)	mean time (ms)
1	3.55	112
2	2.50	100
4	2.43	90
8	2.32	80

SP256V (C)

num th	time to solution (sec)	mean time (ms)
1	5.44	360
2	2.78	180
4	1.49	100
8	0.79	52

Table 2.4: Black Sea parallel sections analysis

Big Black Sea benchmark

Big Black Sea benchmark has 268020 elements, 136749 nodes, 32 levels and does 69 time steps. We encountered a problem to execute this benchmark because with this dimension the software reaches a very large amount of memory. At the first run the operative system returned a signal 9; we tried to understand this behavior, but wherever we put a control print command in the main file, we always obtained the same reply. After deeper analysis we found the cause of the problem: the *bss* segment of this program is bigger than the memory that was requested on nodes. The *bss* is a portion of memory that is allocated before the call of the main program: if this memory is bigger than the memory available, the OS return the KILL signal. So we executed this benchmark on nodes of PICO with a large amount of memory. In table 2.5 the strong scalability and spin time data until 16 threads are reported. Figure 2.2 reports the thread analysis of 8 threads run: on the top we can see the CPU time for each thread and on the bottom the time line, where spin time and computational time are shown. In this benchmark the time spent in parallel region is 59%.

2.2.3 Analysis

First of all, after the parallelization, we tried to use a small benchmark, Nador, (2.2.2) in order to see the parallel performance. In table 2.2 the strong scalability and spin time for each configuration are reported. If we consider that in terms of computational time we have parallelized 60 % of simulation, we obtained a good

num th	time to solution (sec)	spin time(% cputime)	Speed up
1	528.8	0	1
2	426.6	10	1.2
4	398.2	26	1.3
8	355.0	45	1.5
16	362.0	61	1.5

Table 2.5: Big Black Sea time to solution and spin time

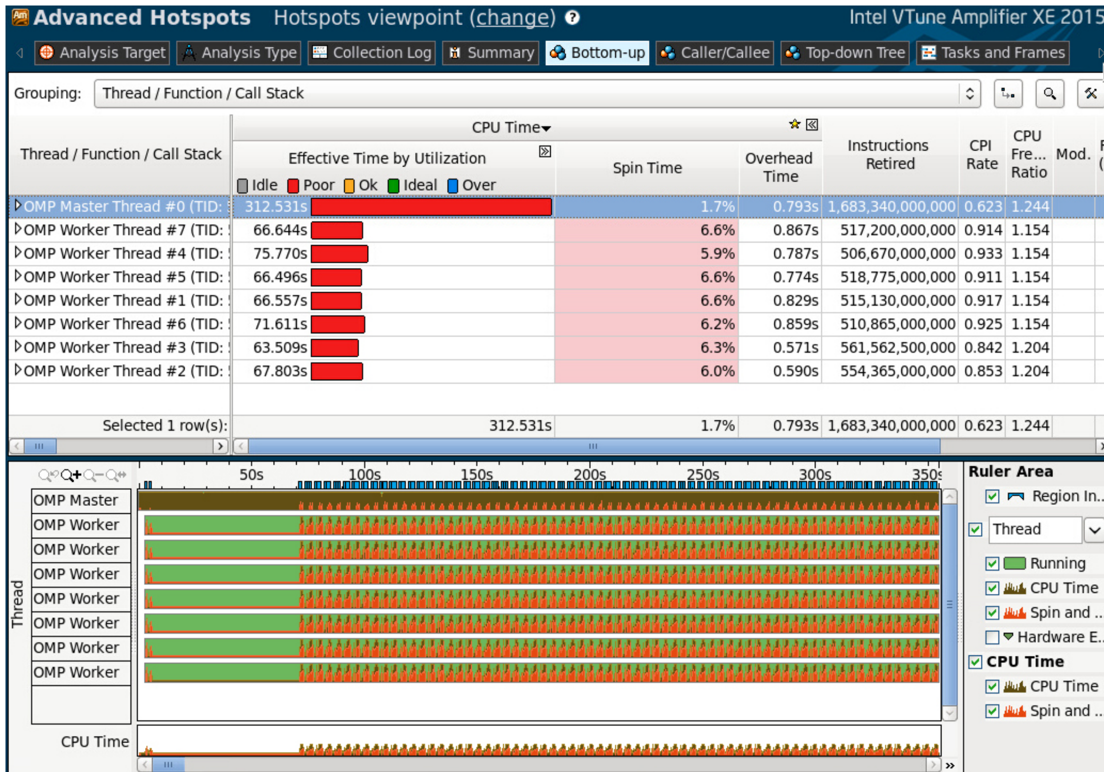


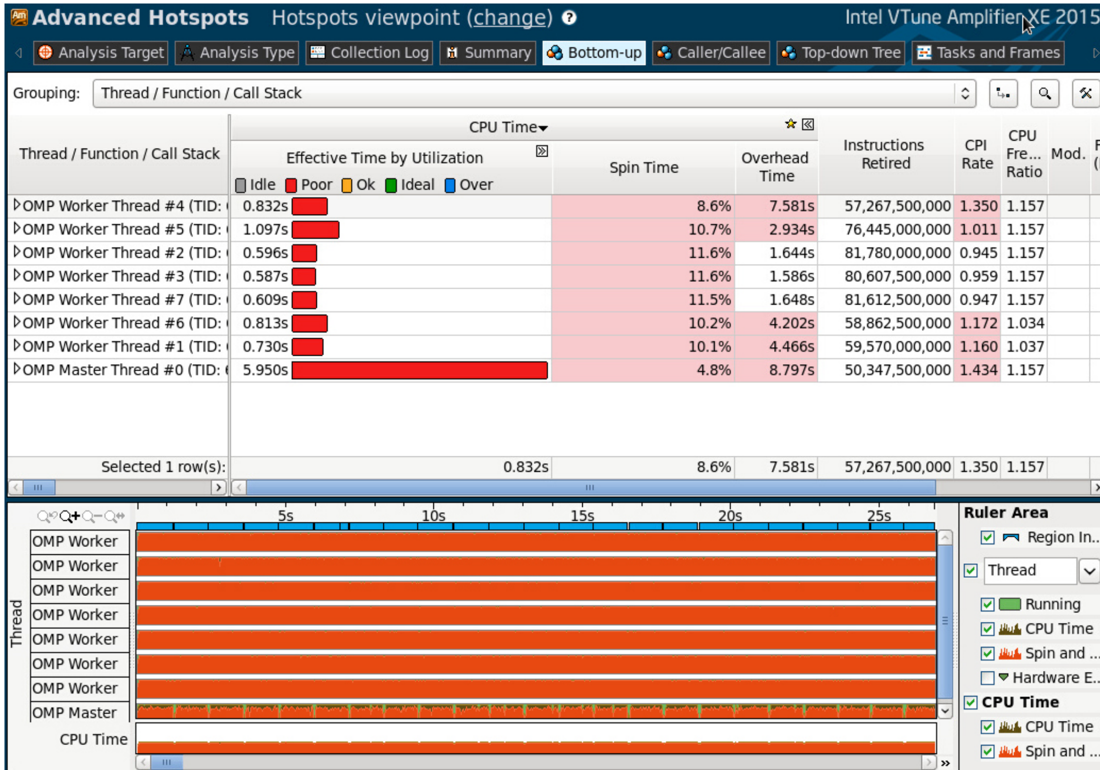
Figure 2.2: Big Black Sea benchmark Threads analysis

speed up: for Amdhal's law we can reach 2.4X as maximum speed up. After the execution time analysis with Intel VTune XE we tried to understand the causes of performance degradation. In figure 2.1 you can see the thread analysis (run on 4 threads): on the top it can be seen that there is a big time imbalance on the threads; on the bottom the time line shows that all threads except the Master have a large spin time. We also perceived that the parallel section is very short in terms of time (central zoom in figure 2.1), with 5 ms at most. *SHYFEM* on this benchmark has the same results of the below simple program, where the workload is undersized. In figure 2.3 A is shown the Intel VTune XE analysis of the simple program, and you can see that the distribution of CPU time and spin time is similar to *SHYFEM* 's. After this result, in order to demonstrate that the problem is the undersized workload, we tried to put a matrix multiplication (matrix size 4096x4096) in **conz3d** and conzstab subroutine: we can see in figure 2.3 B that spin time disappears and the distribution of CPU time among the core becomes uniform.

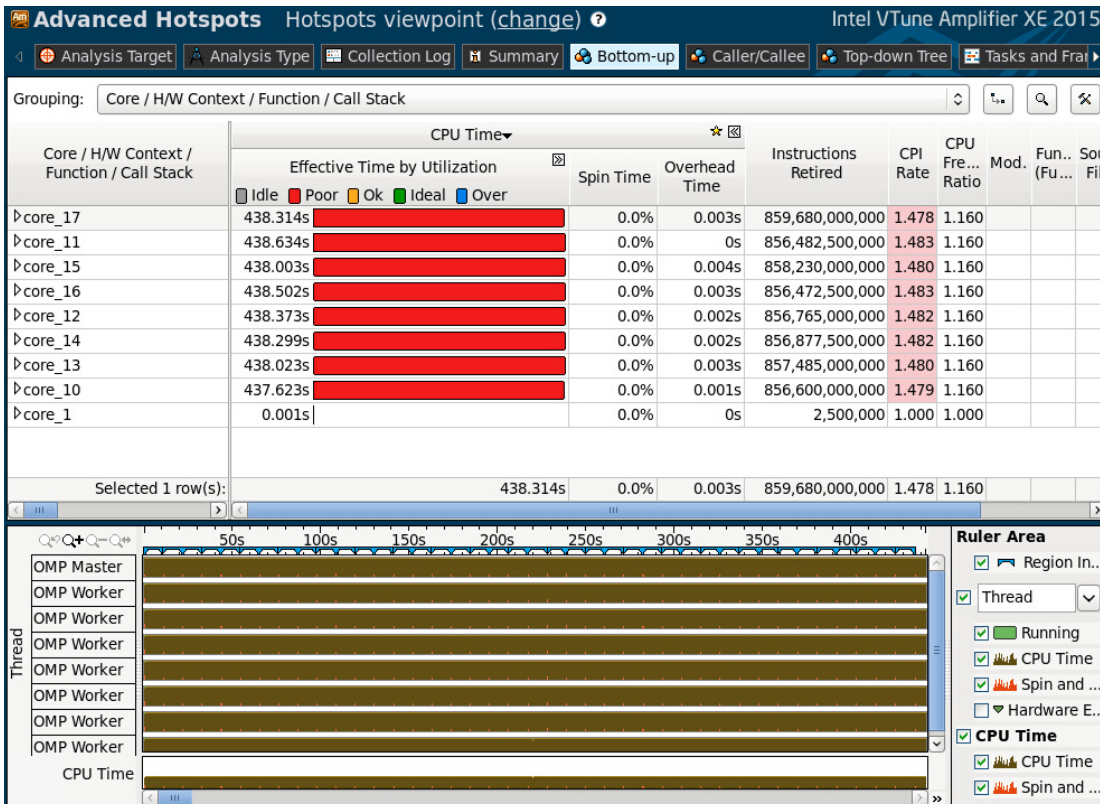
As mentioned before we tried a bigger benchmark than nador, Black sea benchmark (section 2.2.2). With this benchmark we analysed the global behaviour (table 2.3) and the single parallel section performance(table 2.4). In accordance with our hypothesis regarding small worksize problem, we obtained a smaller speed up than Nador but a consistent decrease of spin time; more interesting is the result of each parallel section. The **sp256v** (2.2.1) has linear scalability (table 2.4 C): this behaviour is due to the fact that there aren't synchronization calls among threads. In **conz3d** we haven't linear scalability and at 8 cores the subroutine doesn't scale any more. If we consider **conzstab** we noted that we don't have scalability, probably because, as reported in the last column in table 2.4 B, the time spent in this section is very small.

Finally we tested Black sea benchmark using a bigger mesh: "Big Black sea benchmark" (section 2.2.2). We can see that effectively in this larger simulation the spin time decreases even more (figure 2.2); there is a reduction of 20% respect Nador and 10% respect Black sea benchmark (e.g. in 4 threads run the spin time is reduced from 64% to 54%). Considering now the serial benchmark, we can see that the time spent in the parallel region is 59% and, for Amdhal's law, we must obtain a speed up of 2.4X. We tested scalability and we obtained the best results on 8 core, 355,1 sec for 69 iteration. The time to solution for serial run is 528.8 sec, 312.4 spent in parallel region and 216.4 spent in serial region. If we consider 8 core and we apply Amdahl's law we must obtain $216.4 + (312.4/8) = 255.5 sec$. In our run with 8 core we obtain instead 355 sec, the 29% higher than the theoretical result. Then we analysed the data referred only to the parallel section Intel VTune XE , which show that there is 12.5% of potential gain in parallel section due to spin time and overhead time and 17% due to unbalanced workload among threads (faster threads: 63.5 CPU sec, slower thread: 75.7 CPU sec). To reduce this unbalanced workload we tried a different type of OMP_SCHEDULE. The previous analysis is obtained with *guided* schedule. The best schedule results is obtained with *automatic*, and the unbalanced workload is reduced to 7%.

In the end, if we consider Nador benchmark (in other words, the worst case), we



(a)



(b)

Figure 2.3: Intel VTune XE analysis of (a) Simple program (b) Conz3d with Matrix multiplication

compared the initial version 7.1.10 with the modified version (OpenMP parallel threads + optimization) we obtain a speedup of 2.4X (this calculation has been done taking the run on 16 core with the new version and the serial run with started version).

2.3 Task Parallelization

For the reasons explained in previous section we decided to change the parallelization strategy. Our problem is that we have a lot of spin time due to a small computational load of SHYFEM. There are two reasons why *SHYFEM* has currently a low performance: first of all, it has a low workload; secondly, the paradigm used hitherto introduces implicit locks on the memory. It is unfortunately impossible to solve the first reason because is depended to dataset, therefore it is necessary to try to remove the locks. OpenMP task approach is our best choise meeting the constrain an because of stability and general availability of this paradigm with all compiler suite. The task approach is implemented in OpenMp library since 2006 but until now this technology hasn't had a big diffusion. This choice is aligned with the ongoing development of new high scalability libraries for multi-core processors.

2.3.1 OpenMP task

In this paragraph we describe the OpenMP task [1] technology that was introduced in OpenMP 3.0. Tasks uses an irregular concept of parallelism with respect to threads, since they encapsulate both the function and the data, i.e. when a thread encounters a task it can choose to execute it immediately or later, but the programmer can be sure that the work will be done. To better understand behavior of task it is worth to review the evolution of programming paradigm supported by OpenMP. The classic OpenMP approach is shared memory threads based: when the execution enters in a parallel section it uses a fork-join model to exploit the parallelism, but every thread accesses a shared memory. In OpenMP 2.5 the workshare directive(*parallel,for,sections*) is used to distribute work among threads and the work assigned is executed the thread itself. In OpenMP 3.0 the bunch of work is named *task* and there are two kind of this: implicit and explicit. An implicit task is generated by the workshare construct and executed on a team of threads initialized by a parallel construct, in the same way as OpenMP 2.0. The implicit tasks have been maintained in order to have compatibility with OpenMP legacy code since in the future all the old construct will be converted in implicit task. The new construct task in OpenMP 3.0 generates instead an *explicit task*. This kind of task is a portion of work that is defined by the programmer; after its generation, it will be added to a queue and when the scheduling point is encountered a scheduling mechanism manages the global task workload among the threads. The OpenMP directives *taskwait* and *taskgroup* specify a wait in the completion of child task of the current task, in other words this constructs

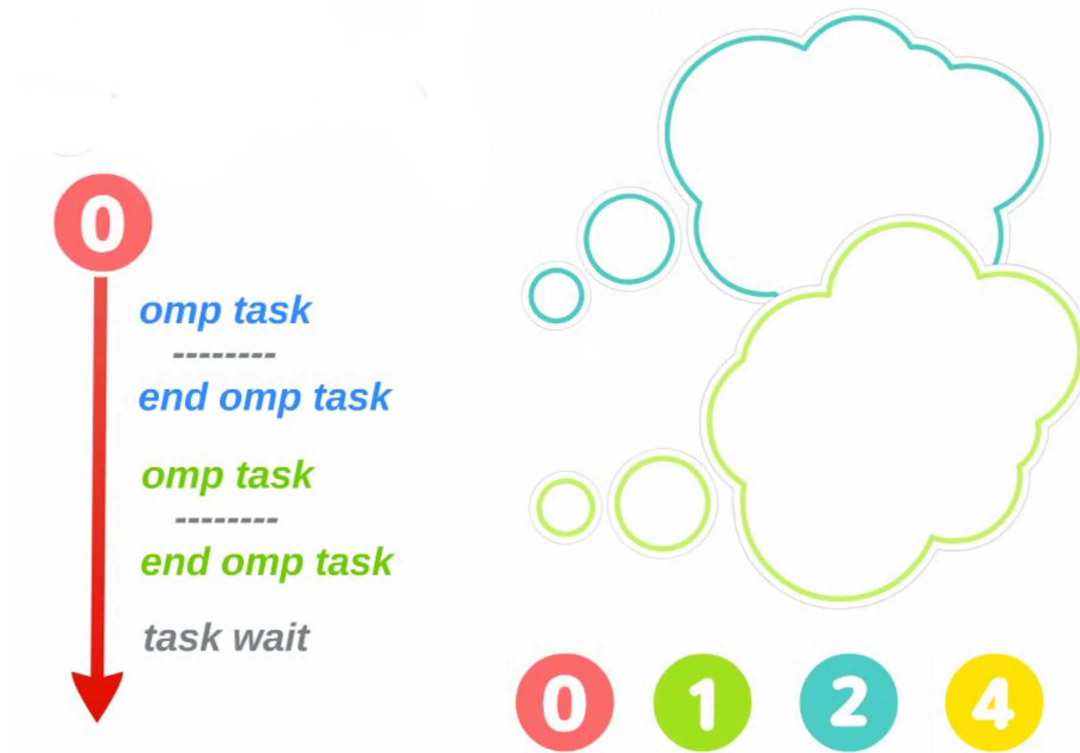


Figure 2.4: Ideal workflow of task programs

specify a scheduling point. It is important to pay attention to difference between scheduling point and barrier: the scheduling point forces threads to complete all task generated but doesn't imply that all threads arrive in the same point of the software, instead the barrier stops the work until all threads encounter it. In general we can think a task program as reported in figure 2.4: only one thread enters in a single (or master) section and spawns task, when this thread encounters a scheduling directive, it and all others threads start to execute tasks, when all tasks are executed only one thread continue the execution of the program. This is like a opportunistic paradigm where the available computational resources (threads) can stole the work to do from the queue masking the latency due to locked threads.

2.3.2 New parallel design

The new parallel design of *SHYFEM* has been conceived to obtain high scalability and to increase the dimension of the parallel part of the software. SHYFEM, in time loops, computes first of all the Hydrodinamic part and subsequently all the other parts, such as temperature, salinity and tracer. Thus, once computed the hydrodinamic part, all the other computations can be done in parallel. In figure 2.5 is shown the new scheme of parallelization.

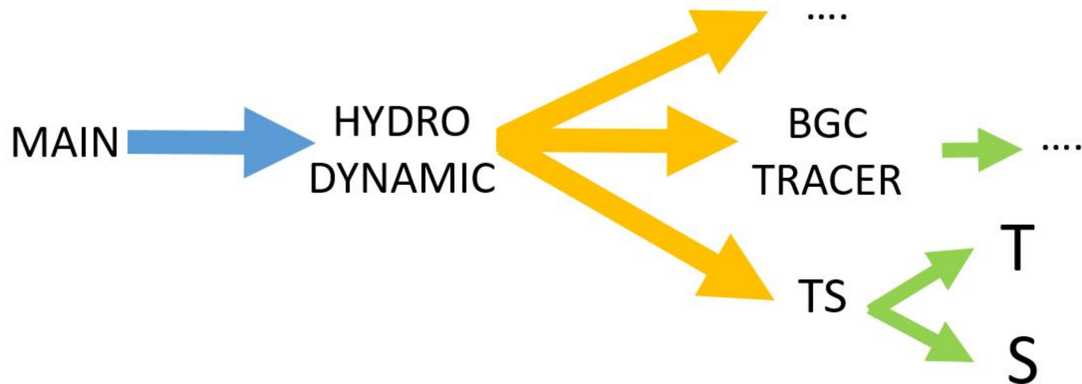


Figure 2.5: Shyfem new parallel design.

This design allows to increase the parallelism in two directions: vertical and horizontal. Horizontal parallelism depends intrinsically on the kind of calculus performed whose implementation activate different parts of *SHYFEM* . Vertical parallelism is based on the possibility to create OpenMP nested regions and nested tasks that work in parallel over the elements of the grid. We can divide *SHYFEM* in two parts, the first one named *hydrodinamic* part and the second *scalar* part: these two parts must be executed sequentially. In the hydrodinamic part there are two parallelizations possible:

- on the elements of the grid
- on the subroutine that work on the same elements (nested).

In the Scalar part the parallelization is exploitable in this way:

- on the different functions
- on the elements of grid
- on the subroutine that work on the same elements (nested).

Another advantage of this design is the extensibility, i.e. in region after hydrodinamic we can add more code in parallel beside T,S, biogeochemical tracer that only needed hydrodinamic results.

2.3.3 Hydrodinamic task

We implemented taskization on the most time consuming subroutine of hydrodinamic part: sp256v. The code reported in A.2.1 is very similar to the code explained in section 2.2.1 but presents 3 differences:

- In task code we found a single directive because all tasks are generated by only one thread.

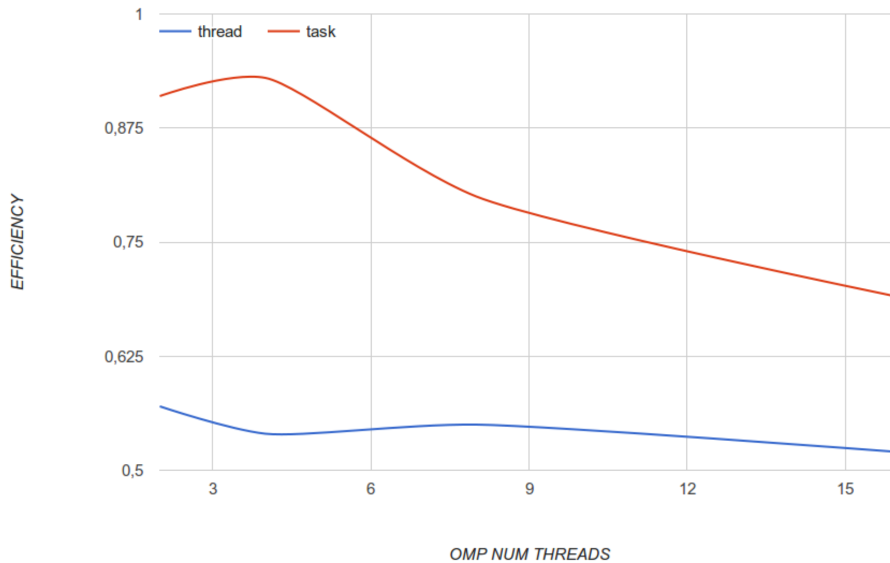


Figure 2.6: Sp256v comparison between task(red line) and threads (blu line) efficency, Big Black Sea benchmark with 1072 task, 250 chunk size

- The variable clause is defined in the task directive and not in the parallel directive, because in the first case the task include work and data.
- The elements loop is divided in two nested loops, since a task must contain a significant amount of work.

We expected from this subroutine the same scalability obtained from the thread version of sp256v, but we obtained a surprising result reported in figure 2.6. In this case the efficiency of tasks is greater than efficiency of threads. For task version we obtained that, when the number of threads increases, the performance decreases: this is probably because the time spent in task management increases; however we obtained a good result for efficiency, since the minimum obtained is 69 %. For threads version we obtained a value of efficiency around 50% for all threads number. We calculated the speed up on time to solution of whole code: with one thread we obtain 846,81 s while with 20 threads we obtained 788,72; thus, speed up is equal to 1.07 X. Using Intel VTune XE we calculated the time spent in the parallel region, the 9.6% of total: thus, for Amdhal's law we can reach a theoretical speedup of 1.1 X.

2.3.4 Scalar task

We implemented until now the task only in the two most time consuming parts of the scalar part: baroclinic(TS computation) subroutine and tracer subroutine(biogeochemical computation). We confined all this part in new subroutine *scalar* in shyfem.f file A.2.2

In this subroutine we activated nested parallelism because in barocl() (A.2.2) and

in tracer_compute() (A.2.2) we opened other two parallel sections and we used taskgroup directive, because we wanted to be sure that the task and its child tasks ended. We report barcol() subroutine in A.2.2

In this subroutine the software spawns two tasks, one for temperature and one for salinity. We set attributed DEFAULT to none because we encountered a lot of problem to let the compiler decide the value of OpenMP clause if not specify. Another note: we use IF clause into avoid the generation to empty task in case that input dataset required the computation of observables involved in the task. In tracer_compute subroutine, the software spawns a number of tasks equal to the number of tracers, as you can see in the code reported in A.2.2

The time to solution speedup results obtained for the scalar subroutine are reported in figure 2.7; in this figure it is shown the speedup obtained for a certain number of tracers (T+S+biogeochemical tracer). The speedup is calculated from the same run on 1 thread and on 20 threads. In the image we can observe an increase of speedup when the number of tasks increases, which is near to theoretical value when the number of tasks is ten times the number of threads. If we consider 800 tasks, we have a 7,8X speedup, with Intel VTune XE we calculate the time spent in parallel part and it is 88% of total time. For Amdhal's law the maximum speedup we can reach is 8.3X. We can observe that we have a good speedup when the number of task in ten times or more than number of threads, in other word we must overload the system. Note that this behavior as already been observed in other similar approaches like the one implemented in Charm++[3] used by NAMD code (even if the implementation of Char++ is perform using MPI only), where it is reported that an overload of a factor of 10 is on average required to mask latencies and not to impact too much the task management.

2.4 Another level of parallelism

In the previous section the exploitation of the horizontal parallelization has been explained, allowing the computation of T,S,biogeochemical tracer in parallel. Here we explain what has been done to add another nested level of parallelization over the elements of the grid. Whit the aim to increase the speedup even when the number of tasks is less than the number of threads.

2.4.1 Refactoring of conz3d

Initially we tried to insert the parallelization shown in section 2.2.1 for conz3d; later we tried to insert a task version of conz3d. Both versions produce a slowing down of the code: this behaviour is caused by various bugs found both in the code of the subroutines and in the structure of the algorithm. The structure of the algorithm causes implicit data races because of the elements sharing the nodes of the grids: in this way, when we compute in parallel the elements loop, the software reads and writes simultaneously in the same cell of memory. We choose to

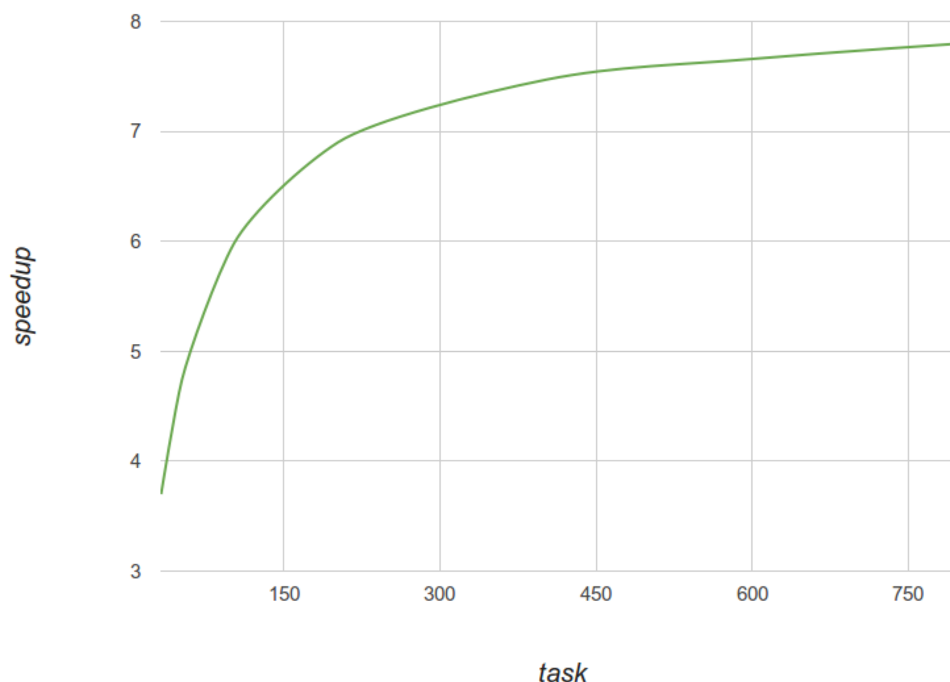


Figure 2.7: The speedup of whole code (Big Black Sea benchmark) with the benefit due to Scalar subroutine task parallelism

rewrite the subroutine in FORTRAN 90 and to clean it in order to simplify the reading to compiler; moreover, we split in two the other subroutine. Now conz3d calls inside conz3d_elements and conz3d_nodes; the first subroutine updates for each element the value on its nodes and the second solves the nodes system to calculate the new concentration, the code is shown in A.3 The concentration matrix cn,co,cdiag,clow,chigh is now allocated dynamically and it is passed to nested function. Cn is the only array returned to function caller. This refactoring per se has not given any improvement but allow us to implement new strategy of data distribution, as explained below, to solve the problem of data races.

2.4.2 New data distribution

Considering that we need to update the node data using the element computation, all elements share at least one node with another element creating the data races condition, that can not be solved by parallelizing in a trivial way the loop over them. We observe that, it is possible to partition the set of elements into subsets containing the elements that do not share nodes within of each of this subset the data races does not exist any longer and all computation can be performed in parallel. The dependency between different elements is than transformed in a dependencies among subset so that different subset can't not be run in parallel but need to be processed in predefined order depending on the algorithm that generate the subset. The idea of this concept is represented in 2.8, where we have a circular mesh and each subset is represented by one color: two triangles with the

same color does not share nodes. We created a new module in order to implement this domain decomposition, named mod_subset.f , which contains a second array, independent_subset , with a subset in each row. The number of subsets is one more than the number of sharing elements for node. To implement this we used a greedy algorithm that produces the following report:

```
----------- DOMAIN CLUSTERING INFORMATION -----------
NUM NODES =         1890  NUM ELEMENTS =         3289
MAX LINK =            8
SUBSET =            1  LENGTH =         582
SUBSET =            2  LENGTH =         564
SUBSET =            3  LENGTH =         505
SUBSET =            4  LENGTH =         455
SUBSET =            5  LENGTH =         422
SUBSET =            6  LENGTH =         380
SUBSET =            7  LENGTH =         269
SUBSET =            8  LENGTH =         102
SUBSET =            9  LENGTH =          10
NUM ITERATION GREEDY ALGO =           9
SUM SUBSET LENGTH =        3289
MAX LENGTH SUBSET =         582
TIME GREEDY =   0.186847925186157
ALL SUBSETS ARE INDIPENDENT
```

It is a characteristics of the greedy algorithm to have the firsts subset with more elements than the others. At the end of the table you can note that there is an independence test on subsets and the parameter "SUM SUBSET LENGTH" to confirm that all element is been assigned in one and just subset. The algorithm that distribute the data can have an impact on the computational time for a bigger mesh because it's complexity is n^2 cost (this is the worst case, in our case is never happen because the connectivity is nearest neighbor).

2.4.3 Nested Task

The subset subdivision permits to complete the vertical parallelization of the Scalar subroutine. Note that nested task does not require nested parallel region, indeed this is consider deprecated by the standard described in [1] and suboptimal for the performance since with introduce explicit syncronization point wich are not require a pure task approach [2]. The code of new conz3d has been renamed conz3d_omp and put in new file newcon_omp.f; the code is reported in appendix A.4. At line 15 in the code we have split the element loop in two, the first over the independent subset and the second, nested (line 16), over the elements in a single subset. Only the computation in a single subset is split in tasks, and the

[2]This implementation has been also discussed and validated with Federico Massaioli as one of the member of OpenMP task subcommittee

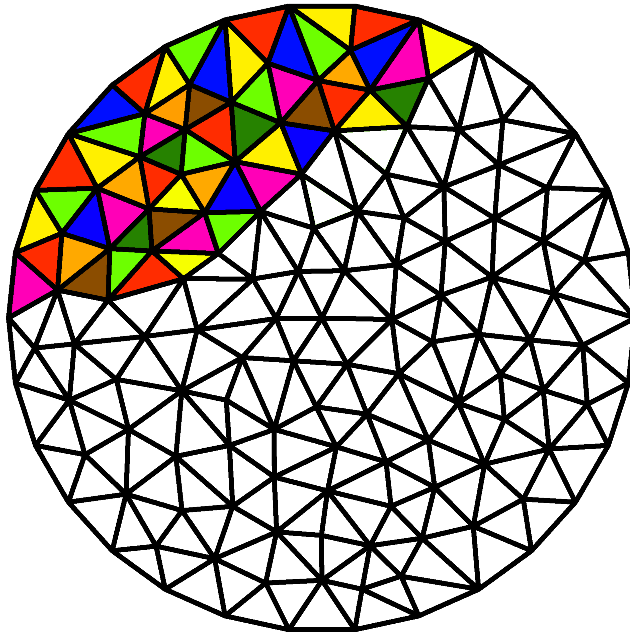


Figure 2.8: Example of domain decomposition of mesh in disjointed subset, each subset is represent by one color, there aren't elements of same color that share a node of grid.

TASKGROUP directive guarantees the finish of a subset computation before the begin of the successive. The chunk size of the second loop has been defined at line 12; in this way the number of tasks is at least ten times of the number of threads, as mentioned before. At line 18 we specify the task definition and we define the clause for all variables, in order to guide the compiler to do a better optimization. Each task executes a fine grain loop on the assigned elements. The introduction of chunk allow a better control of the computational load for each task and we can optimize the chunk size to meet the best scalability curve. At line 55 we have the loop over the nodes: this loop is parallelized as the element loop in the subset described above. We report the speedup result for conz3d_omp in figure 2.9, the blue and red lines are the results obtained with conz3d_omp on two different benchmark: nador (blu) and black sea (red). The orange line is the result obtained with conz3d threads version on black sea (sec 2.2.2). The results have been obtained using only one concentration in order to measure only the speed up of conz3d; given a single concentration only this part spawn tasks. If we take in consideration the red and orange line we deduce that task paradigm is lighter than threads approach: the two approaches seems equivalent for a low number of threads, but when the threads number increases synchronization, locks and waits become predominant in threads approach, therefore the speedup is limited, especially as in this case where the memory management is clumsy. Another point in favor to task approach is shown by the blue line, which tells us even the smallest benchmark with tasks has better results than the best threads benchmark.

In figure 2.10 we plotted the results concerning the speed up of time to solution

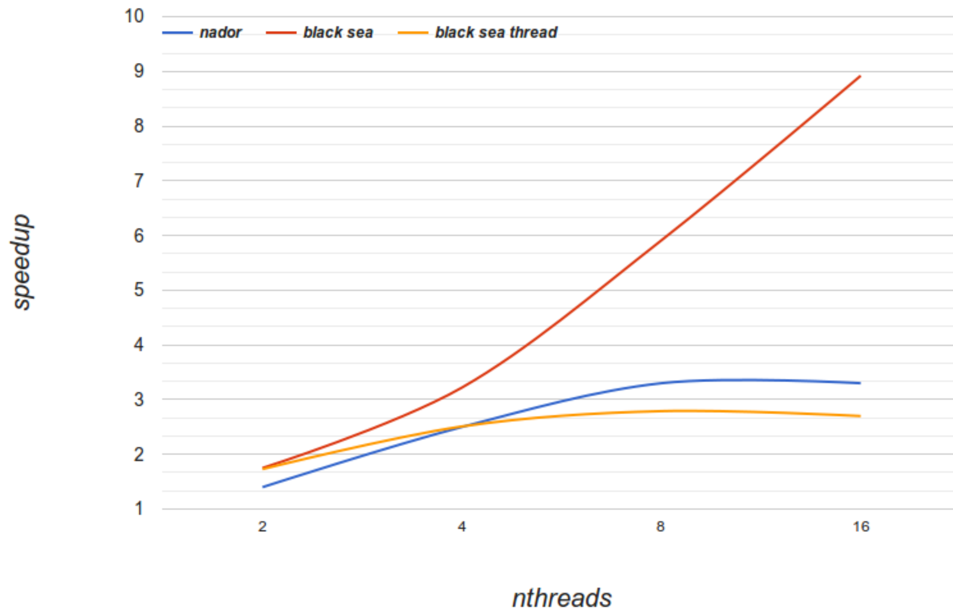


Figure 2.9: conz3d_omp speedup: in blu Nador benchmark, in red black sea benchmark and in orange the Black Sea results obtained with conz3d thread versione tab 2.4 A

for the whole simulation. We reported two cases, one blu line with 4 tracer and one with 8 tracer. In both cases all task parallel levels are enabled(the number of tracers is smaller than the number of threads). In both cases we obtained a increment of speedup due to the last level of parallelization with respect to theoretical maximum considering only horizontal level of parallelism. As expected the performance increased when the workload increases:

- performance increase significantly from 4 tracer to 8 tracer
- the increase of speedup when the third task level is active is better in case of 8 tracer (-26 % in time) than 4 tracer (-19 % in time).
- with 8 tracer we still observe a speedup to pass from 16 to 20 threads.

In figure 2.11 we show the spintime analysis obtained by Intel VTune XE on Black sea benchmark runned on 16 core. We can note that the spintime is concentrated in the serial region while in the bigger parallel region the spintime goes down to zero. The bigger parallel section corresponds to the *scalar* part and the smaller corresponds to the hydrodinamic part. In the hydrodinamic part the parallel section is still too small in time(∼50 ms) to exploit all cores, and then we have a small spin time, as it can be seen in the upper image of figure 2.12. The bigger parallel part instead lasts ∼ 700 ms and the software is able to exploit all cores (see the bottom image of figure 2.12). With Intel VTune XE we also measured the time spent in every parallel section: it is the 87.5% of the total time and therefore, so for Amdhal's law, we can obtain a maximum 8x of theoretical speedup.

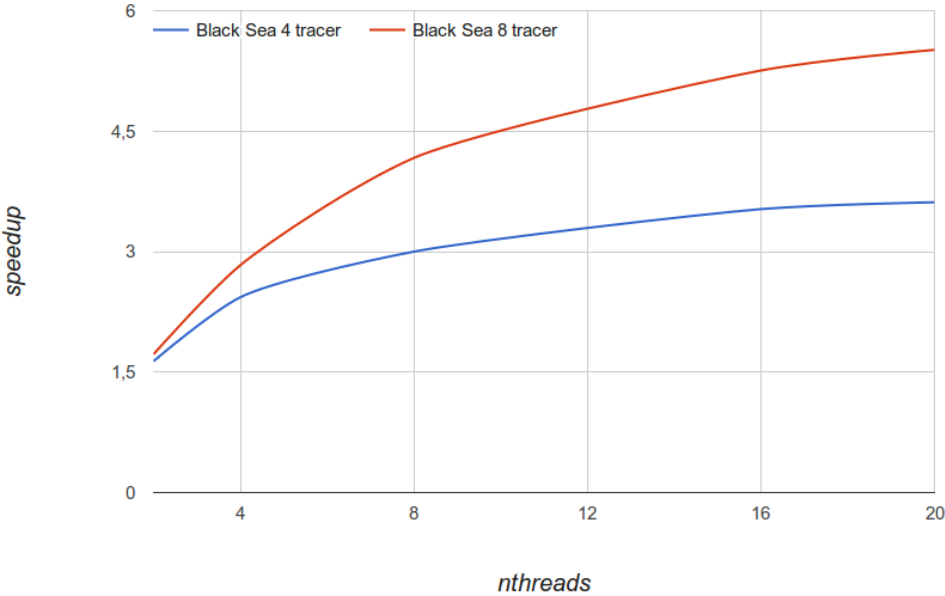


Figure 2.10: Time to solution speedup of whole program with three level of nested tasks implemented. In red the case of Black sea benchmark with 4 tracer, in blu the same benchmark with 8 tracer

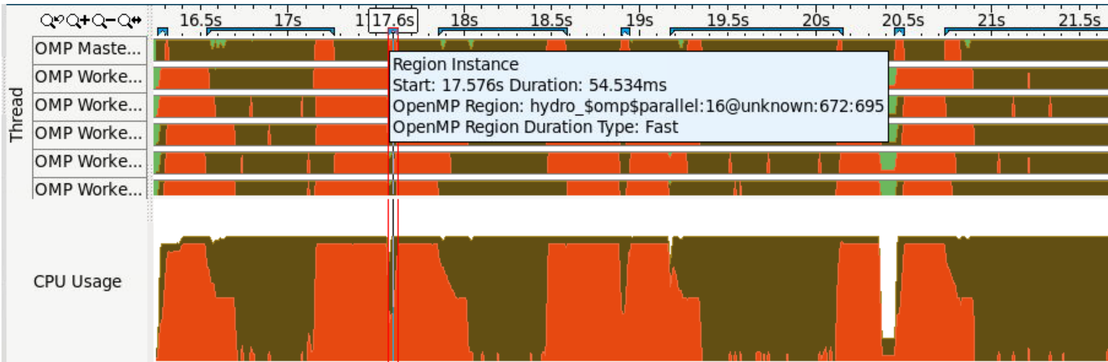


Figure 2.11: Nested task version: time line threads analysis on Black sea benchmark. In particular the spintime is displayed in orange.

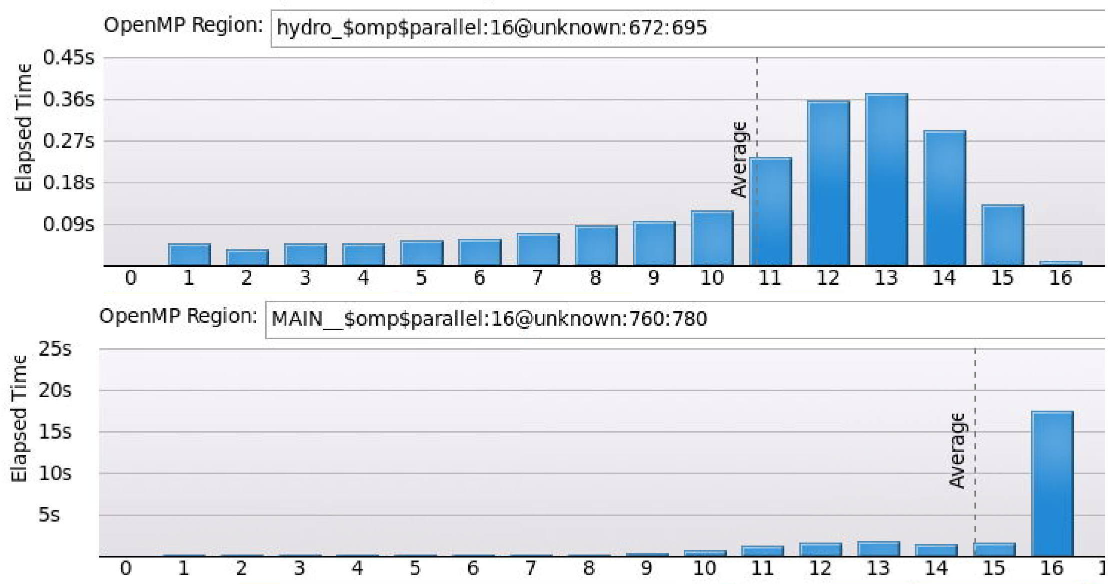


Figure 2.12: Nested task version: number of active threads across parallel region on Black sea benchmark, upper hydrodinamic section(mean 11 threads active), bottom scalar section(mean 13 threads active)

2.5 Debugging

As it is often the case in this work the most time consuming activity has been the debugging of the code, both reletad to the implementation of new features and of the original code containing hidden issues not exploited by the serial run. This happened because it 's easy to exploit hidden bug when introducing parallelization into a complex code with a long history of stratification of new features. We found quite common bugs as uninitialized variables, accesses on arrays out of bounds, or wrong assignment of formal subroutine's parameters. Another difficulty encountered in parallelization is the separation of the calculus part from the IO part. While we were working on the code, we had the sensation that it is was difficult for the compiler to untangle and optimize properly the code. In more than one case performance issues was solved just specify the intent of the variable , and many bugs have been found to be related to the correct dimension of arrays. As a proof that compiler is not able to fully optimize the code that it is we have obtained a large improvement in performace using the so called profile guided optimization. This optimization is a two steps process:

- compile with *prof-gen* flag so that at runtime the software generates a diagnostic file. You can do more than one run with different test cases could be done to have a more complete heuristic.
- compile with *prof-use* flag to generate an optimized executable using the diagnostic file generated.

We tried this optimization with Nador benchmark on PICO, which has been ran on 4 threads compiled with -O2 flag, and after the optimization we had a 1.3X of

speedup. In terms of time we gain 24% of the total. Other user of ISMAR reported that they obtained even a greater gain on another benchmark, in the order of 30% . From common experience this result is very high and shows the difficulties the compiler has to optimize the code in one time of compilation process.

Chapter 3

Conclusion

3.1 Results

The software we tried to optimize shows its age, the coding style and implementation reflect older architectures but provides reliable results. Due to the limited amount of time at our disposal we are satisfied by the results obtained. The main improvement is a relevant speedup achieved thanks to new parallelization paradigm by using OpenMP task. We began this work with *SHYFEM* v 7.1.10, which presented many bugs, data races and a very complicated structure, as we described in section 2.1. While ISMAR group was working in order to create a dynamic allocation version of *SHYFEM* , we tried to investigate the best way to parallelize the software. Initially we took in consideration the possibility to use OpenMP threads approach and we tried to implement it in the three most time-consuming subroutines: sp256v,conz3d,conzstab 2.2.1. We tested this OpenMP thread version on three different worksize benchmarks: Nador, Black Sea, Big Black Sea.
In case of Nador we obtain a smaller speedup in respect theoretical value, and we found that it is due to spintime, as its shown in figure 2.1. With the simple program reported in A.1.3 we reproduced the bad behaviour we saw in Nador: the problem revealed from the simple program is that the work is undersized, in other words the parallel sections are too fast and OpenMP is not able to exploit the power of parallelism creating an imbalance workload among the threads (see figure 2.3 A).
The Black Sea benchmark is bigger than Nador in terms of workload and we obtained a decrease of the spin time but a very smaller speedup of the time to solution. In this case we also analyzed the single parallel section and we understood that where it is requested a contention on part of memory when the number of threads increases the speed up goes down; maybe this behavior is due to a bad manage of memory done with common blocks.
With the biggest benchmark used, Big Black Sea, we had the best results in terms of spin time; this means that one of the principal problems is the low workload of *SHYFEM* , that is a memory bound program.

During this studies we spent a lot of time optimizing serial subroutines and in this part we obtain a 2.4 x of speed up using Nador benchmark with respect to version 7.1.10. When the dynamic allocation was completed we obtained a new version of *SHYFEM* , 7.1.81, with dynamic memory allocation; however, the allocation is global, which mean that all common blocks became module variables. In this way the management is lighter, but it does not solve the problem of the many dependencies between subroutines. Considering the previous results we understood that we need a technology with low impact on threads synchronization and access in memory in order to use the memory model implemented in *SHYFEM* 7.1.81. A good candidate is OpenMP task, but this requires a new design of parallelization because the tasks use a non structural idea of parallelization.
The new parallel design of *SHYFEM* divided the algorithm in two sequentially parts: Hydrodinamic and scalar. Into each part it is possible to use tasks to parallelize the work. In the Hydrodinamic part we used task as a thread in sp256v and we obtained surprising results, as it is shown in figure 2.6: the same parallelization is more efficient using tasks. In the scalar part we can exploit two kind of parallelizations; temperature, salinity and all tracers can become a task because it is not important the order of execution and in each of this task we can spawn other tasks in order to divide the work on elements.
In figure 2.7 we shown the speed up obtained by parallelizing the baroclinic(TS) and biogeochemical tracer part: we reached almost the theoretical peak, but this graph shows that the task technology seems to become efficient when the numbers of tasks is 10 times the number of threads. The old algorithm implemented in conz3d permits that a node shared between elements can be computed from two threads simultaneously, so the computation is serialized and we end up with many of data races.
In order to solve data races we introduced the domain decomposition (figure 2.8) explained in section 2.4.2: the algorithm produced a subset of elements disjointed and allowed the parallel computation into the same subset. The change of distribution allowed to insert in conz3d another level of parallelism: this step is fundamental in order to use *SHYFEM* when the number of tasks is smaller than the number of threads.
The results of new version of conz3d and parallelization are plotted in figure 2.9 and we can see that Nador, the worst case, with the new implementation is better than Black sea with the old implementation. The general behaviour of the OpenMP version of *SHYFEM* is reported in 2.10. The results show that one of the problems in exploiting the parallelism is always the low computational workload, but the results show also that the new version of *SHYFEM* is able to use all levels of parallelism and so all PEs available at least.
Another proof of this is the result reported in figure 3.1: this run has been performed on PICO's big memory node 2 on a old 4 socket with 10 cores/socket, the benchmark is Nador with 8 tracer(T,S and 6 biogeochemical tracer). It can be seen that the contribution of the grid nested parallelism is fundamental to exploit all the nodes's power and using task permits to have a nested tasking with light overhead.

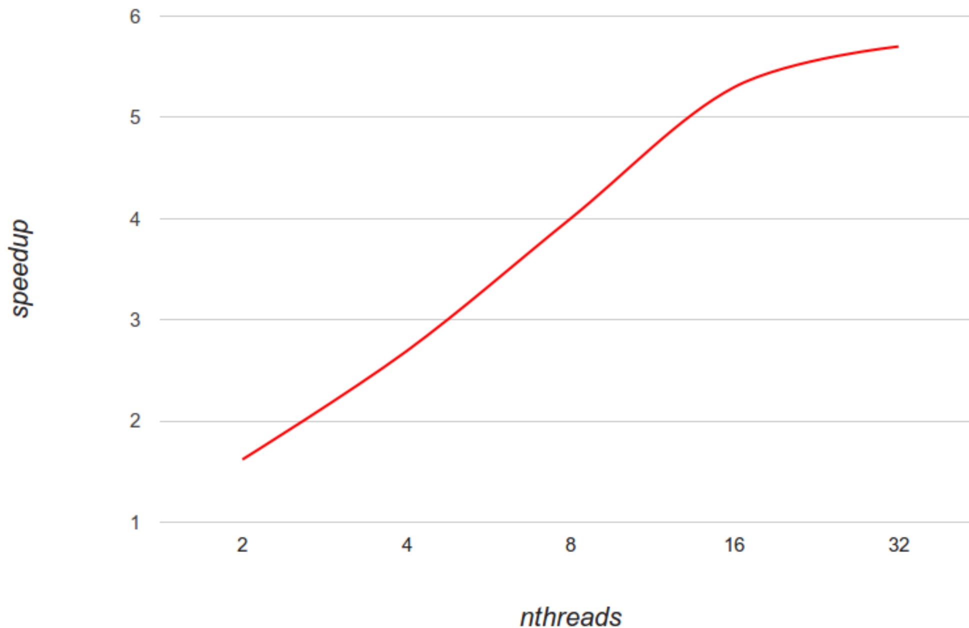


Figure 3.1: Speedup of *SHYFEM* , Black Sea benchmark with 8 tracers runned on PICO Big memory node

3.2 Future developments

The OpenMP tasks have resulted very powerful in order to parallelize software with old memory models, unfortunately even if we have now a dynamic memory allocation, all variables result global. The first thing to do before we continue the parallelization is to analyse the whole code, and then:

- create a schema of flowchart of software
- create a dependencies tree for all variable.
- identify old and unused part of code to be cleaned-up

After this action we will be able to rewrite *SHYFEM* in order to have a cleaner program that exploits data locality and uses efficiently the cache and the vectorization mechanism. As soon as these features will be implemented in the program, we will be able to perform a full fledged parallelization study and not a trial and error attempt as we have done so far in the modified subroutines. An interesting idea to consider is to make *SHYFEM* become a hybrid program, with MPI+OpenMP task. To achieve maximum performance, the recommended solution is to implement two nested levels of parallelism. The highest level is an MPI manager that distributes the elements of the mesh over HPC nodes while the deepest is the taskization of the whole program within each the single shared memory node. Unfortunately, to reach this goal we must spend a lot of time modifying the structure of the code and while we do this we can convert the whole code to Fortran 90 or later.

The actions previously mentioned have been thought in order to have a software at state of art; however we cannot forget that the development of *SHYFEM* is mainly done by scientists, and not HPC programmers: if we want to maintain a high quality of parallelization it is necessary to invest in HPC training for those scientists.
In September 2015 in Venice, we had a meeting in order to create a *SHYFEM* consortium and we have presented part of this work. Many research centers were present: OGS,CMCC,ISMAR,UniBo,CINECA and all agreed that *SHYFEM* is a very useful tool that will help coastal application in next years, so its parallelization and optimization is very important for development the of oceanographic and biogeochemical modelling sector.

Appendix A

Code

A.1 First parallelization

A.1.1 sp256v

```
1  !$OMP PARALLEL private(num_threads,myid,el_do,rest_do,init_do,end_do,
       ie)
2  !$&           firstprivate(bcolin,baroc,az,am,af,at,radv,
3  !$&                vismol,rrho0,dt)
4
5  !$OMP DO schedule(RUNTIME)
6  do ie=1,nel
7  call sp256v_intern(ie,bcolin,baroc,az,am,af,at,radv
8  +       ,vismol,rrho0,dt)
9
10 end do
11 !$OMP END DO
12 !$OMP BARRIER
13
14 !$OMP END PARALLEL
```

A.1.2 conz3d

```
1
2  ............
3
4  !$OMP PARALLEL PRIVATE(k,kn,ii,b,c,aj,aj4,aj12,ilevel,l,hn,ho,
5  !$OMP&                         rk3,cbm,ccm,itot,isum,fw,waux,
6  !$OMP&                         wdiff,rvptop,rvpbot,hmotop,hmobot,hmntop,
7  !$OMP&                         hmnbot,fd,w,flux_top,flux_bot,
8  !$OMP&                         flux_tot1,flux_tot,fl,iext,fnudge,cdummy,hmed,
9  !$OMP&                         cconz,qflux,mflux,loading,lstart,aux,
10 !$OMP&                         us,vs,t,myid,bdebug1,debug,bdebug,berror,
11 !$OMP&                         btvd,bgradup,btvdv,az,azt,ad,adt,aa,aat,
12 !$OMP&                  rstot,rso,rsn,rsot,rsnt,wws,dt,
13 !$OMP&                  alow,ahigh,adiag,u,v,i1,j1,k1)
14 !$OMP&              FIRSTPRIVATE(cn1_l,co1_l,co,aapar,wsink,nthreads)
```

```
15
16 ............
17
18 !$OMP DO PRIVATE(ie) schedule(RUNTIME)
19 !$OMP&     REDUCTION(+:chigh,cn,cdiag,clow)
20 do ie=1,nel
21 ............
22 call vertical_flux_ie(btvdv,ilevel,dt,wws,cl,wl,hold,vflux)
23 ............
24 call tvd_fluxes(ie,l,itot,isum,dt,cl,co1_l,gradxv,gradyv,u,v,f,fl)
25 ............
26 end do
27 !$OMP END DO
28
29
30 !$OMP DO schedule(RUNTIME)
31 do k=1,nkn
32 ............
33 end do
34 !$OMP END DO
35
36 ............
```

A.1.3 Simple program

```
1 PROGRAM main
2
3 use omp_lib
4
5 IMPLICIT NONE
6 integer :: I
7
8 DO i=1,10E6
9
10 call bad_scheduling()
11
12 END DO
13
14 STOP
15
16 END PROGRAM main
17
18 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
19
20 subroutine bad_scheduling()
21
22 use omp_lib
23
24 IMPLICIT NONE
25
26 DOUBLE PRECISION,DIMENSION(:),ALLOCATABLE :: array
27 INTEGER :: dim,i,j,k,myid,nthreads
28
29 dim = 20
30 ALLOCATE(array(dim))
```

```
31 !$OMP PARALLEL PRIVATE(myid)
32
33 myid =  omp_get_thread_num()
34
35 !$OMP DO PRIVATE(j)
36 DO j=1,dim
37 array(j) = COS(real(myid))
38 ENDDO
39 !$OMP END DO
40
41 !$OMP END PARALLEL
42
43 end subroutine bad_scheduling
```

A.2 OpenMP Task

A.2.1 Hydrodynamic task

```
1 !$OMP PARALLEL
2 !$OMP SINGLE
3
4 do ie=1,nel,250
5 !$OMP TASK FIRSTPRIVATE(ie,bcolin,baroc,az,am,af,at,radv
6 !$OMP&      ,vismol,rrho0,dt) PRIVATE(ies,iend)
7 !$OMP&      SHARED(nel)    DEFAULT(NONE)
8
9 iend = ie+249
10 if(iend .gt. nel) iend = nel
11 do ies=ie,iend
12
13 call sp256v_intern(ies,bcolin,baroc,az,am,af,at,radv
14 +       ,vismol,rrho0,dt)
15 end do
16 !$OMP END TASK
17 end do
18
19 !$OMP END SINGLE
20 !$OMP END PARALLEL
```

A.2.2 Scalar Task

scalar

```
1
2 subroutine scalar()
3
4 !$  use omp_lib
5
6 implicit none
7
8 real getpar
9
10 integer :: nscal,itemp,isalt,iconz,itvd
```

```
11
12
13 !$OMP PARALLEL
14 !$OMP SINGLE
15
16 nscal = 0
17
18 itemp = nint(getpar("itemp"))
19 isalt = nint(getpar("isalt"))
20 iconz = nint(getpar("iconz"))
21 itvd = nint(getpar("itvd"))
22
23 call tvd_init(itvd)
24
25 if(itemp .gt. 0) nscal = nscal +1
26 if(isalt .gt. 0) nscal = nscal +1
27 if(iconz .gt. 0) nscal = nscal + iconz
28
29 !$ call omp_set_nested(.TRUE.)
30
31
32 !$OMP TASKGROUP
33
34 !$OMP TASK
35 call barocl(1)
36 !$OMP END TASK
37
38 !$OMP TASK
39 call tracer_compute
40 !$OMP END TASK
41
42
43 !$OMP END TASKGROUP
44
45 !$OMP END SINGLE
46 !$OMP END PARALLEL
47 end subroutine
```

Barocl

```
1 subroutine barocl()
2 ...
3
4 !$OMP PARALLEL
5 !$OMP SINGLE
6
7
8 !$OMP TASK PRIVATE(what,dtime) FIRSTPRIVATE(thpar,wsink,robs,itemp,it
    )
9 !$OMP&        SHARED(idtemp,tempv,difhv,difv,difmol,tobsv) DEFAULT(NONE)
10 !$OMP&        IF(itemp > 0)
11
12 if( itemp .gt. 0 ) then
13 what = 'temp'
14 call scal_adv_nudge(what,0
15 +                                        ,tempv,idtemp
```

```
16 +                                   ,thpar,wsink
17 +                                   ,difhv,difv,difmol,tobsv,robs)
18 end if
19
20 !$OMP END TASK
21
22 !    call openmp_get_thread_num(tid)
23 !    !write(6,*) 'number of thread of salt: ',tid
24
25 !$OMP TASK PRIVATE(what,dtime) FIRSTPRIVATE(shpar,wsink,robs,isalt,it
     )
26 !$OMP&      SHARED(idsalt,saltv,difhv,difv,difmol,sobsv) DEFAULT(NONE)
27 !$OMP&      IF(isalt > 0)
28
29 if( isalt .gt. 0 ) then
30 what = 'salt'
31 call scal_adv_nudge(what,0
32 +                                   ,saltv,idsalt
33 +                                   ,shpar,wsink
34 +                                   ,difhv,difv,difmol,sobsv,robs)
35 end if
36
37 !$OMP END TASK
38
39 !$OMP END SINGLE
40 !$OMP END PARALLEL
41
42 ...
```

tracer_compute

```
1
2 subroutine tracer_compute()
3 ...
4
5 !$OMP PARALLEL
6 !$OMP SINGLE
7
8 do i=1,nvar
9
10 !$OMP TASK FIRSTPRIVATE(i,rkpar,wsink,difhv,difv,difmol,idconz,what,
11 !$OMP&      dt,nlvdi) SHARED(conzv,tauv,massv)  DEFAULT(NONE)
12
13 call scal_adv(what,i
14 +                                   ,conzv(1,1,i),idconz
15 +                                   ,rkpar,wsink
16 +                                   ,difhv,difv,difmol)
17
18 call decay_conz(dt,tauv(i),conzv(1,1,i))
19 call massconc(+1,conzv(1,1,i),nlvdi,massv(i))
20
21 !$OMP END TASK
22
23 end do
24
25 !$OMP END SINGLE
```

```
26 !$OMP END PARALLEL
27
28 ...
```

A.3 Modern conz3d

```
1 subroutine conz3d
2
3 ...
4 ALLOCATE(cn(nlvddi,nkn))
5 ALLOCATE(co(nlvddi,nkn))
6 ALLOCATE(cdiag(nlvddi,nkn))
7 ALLOCATE(clow(nlvddi,nkn))
8 ALLOCATE(chigh(nlvddi,nkn))
9
10 DO over elements
11 call conz3d_elements()
12 END DO
13
14 ...
15
16 DO over nodes
17 call conz3d_nodes()
18 END DO
19 ...
20 cn1 = cn
21 DEALLOCATE(cn)
22 DEALLOCATE(co)
23 DEALLOCATE(cdiag)
24 DEALLOCATE(clow)
25 DEALLOCATE(chigh)
26
27 end subroutine conz3d
```

A.4 conz3d_omp

```
1
2 subroutine conz3d_omp()
3
4 ...
5
6 nchunk = 1
7 nthreads = 1
8 !$  nthreads = omp_get_num_threads()
9
10 do i=1,subset_num    ! loop over indipendent subset
11
12 !$      nchunk = subset_el(i) / ( nthreads * 10 )
13
14 !$OMP TASKGROUP
15 do jel=1,subset_el(i),nchunk
16
17 !$OMP TASK FIRSTPRIVATE(jel,i) DEFAULT(NONE)
```

```
18 !$OMP& PRIVATE(j,ie)
19 !$OMP& SHARED(nlvddi,nlev,itvd,itvdv,istot,isact,aa,nchunk)
20 !$OMP& SHARED(difmol,robs,wsink,rload,ddt,rkpar,az,ad)
21 !$OMP& SHARED(azt,adt,aat,rso,rsn,rsot,rsnt,dt,nkn)
22 !$OMP& SHARED(cn,co,cdiag,clow,chigh,subset_el,cn1,co1)
23 !$OMP& SHARED(subset_num,indipendent_subset)
24 !$OMP& SHARED(difhv,cbound,gradxv,gradyv,cobs,load,difv,wsinkv)
25
26 do j=jel,jel+nchunk-1    ! loop over elements in subset
27 if(j .le. subset_el(i)) then
28 ie = indipendent_subset(j,i)
29
30 call conz3d_element(ie,cdiag,clow,chigh,cn,cn1
31 +       ,dt
32 +                        ,rkpar,difhv,difv
33 +       ,difmol,cbound
34 +       ,itvd,itvdv,gradxv,gradyv
35 +       ,cobs,robs
36 +       ,wsink,wsinkv
37 +       ,rload,load
38 +       ,az,ad,aa,azt,adt,aat
39 +       ,rso,rsn,rsot,rsnt
40 +       ,nlvddi,nlev)
41 end if
42 end do ! end loop over el in subset
43 !$OMP END TASK
44 end do
45
46 !$OMP END TASKGROUP
47
48 end do ! end loop over subset
49
50 !$       nchunk = nkn / ( nthreads * 10 )
51
52
53 !$OMP TASKGROUP
54 do knod=1,nkn,nchunk
55 !$OMP TASK FIRSTPRIVATE(knod) PRIVATE(k) DEFAULT(NONE)
56 !$OMP& SHARED(cn,cdiag,clow,chigh,cn1,cbound,load,nchunk,
57 !$OMP&                rload,ad,aa,dt,nlvddi,nkn)
58 do k=knod,knod+nchunk-1
59 if(k .le. nkn) then
60 call conz3d_nodes(k,cn,cdiag(:,k),clow(:,k),chigh(:,k),
61 +                                cn1,cbound,load,rload,
62 +                                ad,aa,dt,nlvddi)
63 endif
64 enddo
65 !$OMP END TASK
66 end do
67
68 !$OMP END TASKGROUP
69
70 cn1 = real(cn)
71
72 DEALLOCATE(cn)
73 DEALLOCATE(co)
```

```
74 DEALLOCATE(cdiag)
75 DEALLOCATE(clow)
76 DEALLOCATE(chigh)
77
78 end subroutine conz3d_omp
```

Bibliography

[1] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Yuan Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and Guansong Zhang. The design of openmp tasks. Parallel and Distributed Systems, IEEE Transactions on, 20(3):404–418, March 2009.

[2] Marco Bajo, Christian Ferrarin, Irina Dinu, Georg Umgiesser, and Adrian Stanica. The water circulation near the danube delta and the romanian coast modelled with finite elements. Continental Shelf Research, 78:62 – 74, 2014.

[3] Laxmikant V Kale, Anshu Arya Abhinav Bhatele Abhishek Gupta, Nikhil Jain, Pritish Jetley Jonathan Lifflander, Phil Miller, Yanhua Sun, and Ramprasad Venkataraman Lukasz Wesolowski Gengbin Zheng. Charm++ for productivity and performance. 2012.

[4] Vivek Sarkar, William Harrod, and Allan E Snavely. Software challenges in extreme scale systems. Journal of Physics: Conference Series, 180(1):012045, 2009.

[5] Georg Umgiesser, Donata Melaku Canu, Andrea Cucco, and Cosimo Solidoro. A finite element model for the Venice Lagoon. Development, set up, calibration and validation . Journal of Marine Systems, 51(1–4):123 – 145, 2004. Lagoon of Venice. Circulation, Water Exchange and Ecosystem Functioning.

[6] Georg Umgiesser, Christian Ferrarin, Andrea Cucco, Francesca De Pascalis, Debora Bellafiore, Michol Ghezzo, and Marco Bajo. Comparative hydrodynamics of 10 mediterranean lagoons by means of numerical modeling. Journal of Geophysical Research: Oceans, 119(4):2212–2226, 2014.