The Journal of Supercomputing (2020) 76:5636–5654
https://doi.org/10.1007/s11227-020-03172-7

Legacy code and parallel computing: updating and parallelizing a numerical model

Fernando G. Tinetti[1] · Maximiliano J. Perez[2] · Ariel Fraidenraich[3] · Adolfo E. Altenberg[4]

Published online: 23 January 2020

Abstract

In this paper, we present several important details in the process of legacy code parallelization, mostly related to the problem of maintaining numerical output of a legacy code while obtaining a balanced workload for parallel processing. Since we maintained the non-uniform mesh imposed by the original finite element code, we have to develop a specially designed data distribution among processors so that data restrictions are met in the finite element method. In particular, we introduce a data distribution method that is initially used in shared memory parallel processing and obtain better performance than the previous parallel program version. Besides, this method can be extended to other parallel platforms such as distributed memory parallel computers. We present results including several problems related to performance profiling on different (development and production) parallel platforms. The use of new and old parallel computing architectures leads to different behavior of the same code, which in all cases provides better performance in multiprocessor hardware.

Keywords Parallelization · Legacy code constraints · Mesh data distribution for FEM

✉ Fernando G. Tinetti
fernando@info.unlp.edu.ar; ftinetti@gmail.com

Maximiliano J. Perez
maxiperezunlam@gmail.com

Ariel Fraidenraich
afraide.1@gmail.com

Adolfo E. Altenberg
altenberg@gmail.com

[1] Fac. de Informática, UNLP, and CIC Prov. de Bs. As., La Plata, Buenos Aires, Argentina

[2] U. N. de La Matanza, San Justo, Buenos Aires, Argentina

[3] Universidad de Belgrano, Buenos Aires, Argentina

[4] UTN-FRBA, Buenos Aires, Argentina

1 Introduction

2D shallow water models have become standard in the numerical analysis of environmental flows. For flood wave analysis, a 2D shallow water representation provides an adequate resolution of the impact on flooding and the ability to include the effects of river branches, islands, tributaries, complex navigation channels and many geometric and topographic complexities. Besides, other environmental hazards, such as a flood or pollutant spill, can be predicted, which could be used to aid in the risk assessment and management control actions [2, 19, 26].

The flooding problem simulation was progressively developed until the end of the last century with the FEM (Finite Element Method) [19]. More specifically, the FEM is used to model geometry and edge conditions providing better efficiency [19] and abstraction than other simulation/modeling methods. This paper presents further parallelization and computing performance improvement for the numerical method than our previous work [25]. The shallow water numerical model is initially designed following a Taylor–Galerkin scheme [19, 26]. The spatial resolution is accomplished with weighted residuals (Galerkin method) and the temporal advance by means of a Taylor series approximation. The Taylor method is based on the knowledge of the Jacobian matrices that correspond to the projections of flows in the two orthogonal Cartesian coordinates [19, 26]. The differential system of shallow water equations is expressed conservatively by Eq. 1 as follows:

$$\frac{\partial U}{\partial t} + \frac{\partial F_i}{\partial X_i} - \frac{\partial Rd_i}{\partial X_i} = R_s, \quad i = 1, 2 \tag{1}$$

where

- $U(h, p, q)$ is the nodal vector of unknowns, where h is the water depth, p is the discharge flow in the x direction (longitudinal) and q is the discharge flow in the y direction (transversal).
- $F_i, i = 1, 2$ is the convective flow vector in x- and y-axes, respectively, as shown in Eqs. 2 and 3.
- $Rd_i, i = 1, 2$ is the diffusive vector.
- R_s is the source vector (containing the terms of topographic and friction variations) given by Eq. 4.

$$F_1 = \left(p, \frac{p^2}{h} + g\frac{h^2}{2}, \frac{pq}{h}\right)^T \tag{2}$$

$$F_2 = \left(q, \frac{pq}{h}, \frac{q^2}{h} + g\frac{h^2}{2}\right)^T \tag{3}$$

$$R_s = \left(0, -gh(S_0^x - S_f^x), -gh(S_0^y - S_f^y)^T\right) \tag{4}$$

The bed and friction slopes are given as

$$S_0^x = -\frac{\partial z}{\partial x} \tag{5}$$

$$S_0^y = -\frac{\partial z}{\partial y} \tag{6}$$

$$S_f^x = \frac{p}{h^3 c^2}\sqrt{p^2 + q^2} \tag{7}$$

$$S_f^y = \frac{q}{h^3 c^2}\sqrt{p^2 + q^2} \tag{8}$$

where $z = (x, y)$ is the bed elevation above an arbitrary domain and C is the Chézy coefficient of the bed resistance. The initial conditions are the natural quiet lake, and the boundary conditions are given by

- $P(0, y, t) = \mu(y, t)$, where μ is the velocity at the entry section.
- $h(L_x, y, t) = h_0$, where h_0 is the final depth at L_x (output section).
- $q(L_x, y, t) = 0$.
- $q(x, 0, t) = 0$, $q(x, L_y, t) = 0$, where these conditions represent the normal fluxes in the lateral boundary. The same boundary conditions are applied to a symmetric circular hole.

Simulations results provide the values of $U(h, p, q)$, the vector of nodal unknowns. The time evolution is modeled as a sequence of two half-steps of temporary advance, for which the solution of the first half-step is the initial condition of the second:

$$U^{n+\frac{1}{2}} = U^n + \frac{1}{2}\Delta t\left(R_s + \frac{\partial R_{di}}{\partial x_i} - \frac{\partial F_i}{\partial x}\right)_{(t_n)} \tag{9}$$

$$U^{n+1} = U^n - \Delta t\left(\frac{\partial F_i}{\partial x_i} - \frac{\partial R_{di}}{\partial x_i} - R_s\right)_{\left(t_{n+\frac{1}{2}}\right)} \tag{10}$$

The FEM briefly described above imposes simulations with a triangulation mesh similar to that shown in Fig. 1, representing finite element domain subdivision, with radial densification symmetry around a circular obstacle and a high densification in the lateral edges. Besides, the Taylor–Galerkin approximation—being an explicit method—imposes a small temporal increment in order to satisfy the Courant–Friedrichs–Lewy condition of stability [26]. And the small time step, in turn, implies a strong requirement on computing power for stable/useful results. The starting point of the work presented in this paper has been already explained in [25], and it was focused in the Fortran program that implements the FEM simulation with two objectives: (1) to enhance/update the Fortran source code in case it is needed and (2) to

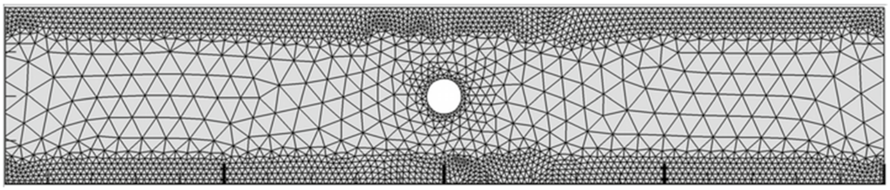

Fig. 1 Finite element mesh

speed up program execution, taking advantage of the multiprocessing facilities provided by current computing platforms. As it could be expected, both objectives were partially accomplished, because both imply an incremental work on legacy code (the original Fortran program). Thus, we continue the parallelization and the corresponding performance evaluation, with direct implication on the mesh partitioning among processors and the resulting workload balance. Even when the mesh partitioning in this paper is implemented and evaluated in shared memory parallel computers, the same underlying idea could be used for a distributed memory parallel computer implementation.

The rest of the paper is organized as follows. Section 3 briefly summarizes the work already made and published on this specific model [25]. Section 2 includes references to several related works on the shallow water numerical model in general. Section 4 discusses the general data parallel approach we have chosen as a global parallelization of the legacy code. Section 5 introduces the mesh/data distribution among processors for parallel processing. The mesh partition would introduce several changes in the legacy code and defines a specific pattern for computing. Section 6 introduces the specific performance evaluation experiments and explains several interesting results. Finally, Sect. 7 details the conclusions as well as some of the immediate work to enhance the current code and parallel processing.

2 Related work

There have been a number of papers reporting numerical methods to solve the 2D shallow water problem for flood wave analysis. The finite volume method for flood wave analysis [2] adequately represents the fluxes in physical approximations. A Taylor–Galerkin scheme is the underlying numerical approach used in [19, 26]. Since we are working on the legacy code, where this specific knowledge and previous developments are embedded in the code, we did not have to modify and/or develop any new research related to the numerical approach.

Domain decomposition and mesh generation methods to improve load balance have been used in [11], in numerical experiments applied to computational fluid dynamics (CFD) and electromagnetism. Furthermore, non-structured mesh (based on advancing front) and a large number of elements are used in [11], and a generalized method is presented in [9]. Our approach maintains the original mesh structure and densification defined by the expert scientific/users so that we do not change the mesh and/or mesh structure in any way. Maintaining the *original* mesh limits the approach to the partitioning and load balancing of the mesh *as is*, but we avoid having to deal with possible numerical errors and/or software bugs of new mesh configuration/approaches.

The influence of mesh partitioning on the solution of incompressible flow problems, e.g., flow around submarines, is studied applying software packages such as Code Saturn, Metis, ParMetis and Zoltran [20, 21]. Metis and ParMetis give the best load balance and speedup results. While we do not discard using software libraries such as ParMetis, we are currently working from the point of view of legacy software. We expect to acquire experience on the legacy software and all its characteristics as a previous step toward more global code changes, such as using external parallel/parallelization libraries.

As explained above, the focus of our approach is on maintaining the original source code and mesh structure, which includes specific densification characteristics, thus avoiding the portability and *translation* of source code for using already defined libraries and numerical approaches. Mainly, we constrained the work to the HPC/parallel computing field, maintaining the original numerical approach, and thus, we do not include any numerical artifacts/errors regarding the original numerical results.

As shown in [12], the mesh partitioning in the context of FEM for parallel processing is an NP-hard problem, and we take advantage of the mesh partitioning characteristics for aiding the partitioning. In particular, we have chosen an initial approach of a spatial/geometry-aware partitioning in terms of the definitions in [1, 13], along with *ad hoc* enhancements for unbalanced results and special cases, as explained in Sect. 5.1. We are considering further partition enhancements by taking into account different mesh densification in terms of local/greedy partitioning as described in [1, 6, 12].

Unlike the approaches in [4, 16], we do not consider a parallel initial mesh construction nor an *a priori* approach focused in distributed shared memory parallel computer. Instead, we consider the numerical approach and its corresponding mesh (including specific densification pattern/s, as in the problem in this paper) as departure points, and we propose a mesh partitioning independently of being used in a physically shared or distributed memory, including distributed shared memory architecture. Our approach avoids extra work required for verifying that the mesh generation does not introduce any numerical errors and implies the minimum source code change to existing programs (e.g., the large number of scientific legacy code currently used). This decision also excludes the generation of completely irregular mesh configurations, and we maintain the original mesh definition (including densification) so that our approach will not generate any numerical instability or unknown (side) effects of changing the underlying mesh.

3 Previous work on the legacy code

In this section, we will summarize the work already done on the code implementing the shallow water numerical model, mostly published in [25]. The initial phase was focused on enhancing the legacy code, by following the sequence:

- Porting the development and operation environments from Windows OS to the Linux OS. Most of the changes were related (as expected) to the compiler, basically to take advantage of the GNU compiler options, which include many new optimizations and are in continuous development.
- Identification of source code and subprograms important section/s, mostly using basic profiling, such as that provided by the GNU tools (gprof). Identifying important subprograms and the general runtime call graph helps to focus the work on the source code directly related to performance rather than trying to reengineer all the program at once.
- Some basic code updates from the Fortran 77 "coding style" to more modern coding style, e.g., code indentation and other code changes related to free-format lines, available since Fortran 90/95 standards. All of these changes are focused on improving code readability and maintainability (i.e., software engineering project improvements [24]).
- The most time-consuming subroutine was identified with profiling experiments. In that subroutine, several iteration structures (Fortran Do loops) were identified and parallelized using OpenMP [17]. Some of those Do loops were parallelized by using the knowledge of the expert users and programmers, for which we simplify the questions to "yes/no" answers in terms of numerical problems (related to data structures and data dependency problems). In all cases, semantics of source code has been preserved and the numerical output has been maintained identical to that of the original program.

The profile-based standard methodology previously described is, in some way, a "blind" technique, since the source code restructuring is made with little knowledge of its application. The numerical/scientific field experts were consulted about data flow and array access patterns in Fortran iterative structures. This methodology implied a strong programming effort by HPC (high-performance computing) expert programmers in order to introduce changes on the source code with no impact on the program output. Summarizing, while standard and well-known source code transformations such as core indentation can be carried out "blindly," several other transformations have to be supervised/aided by the numerical field experts. Performance was improved even when the optimized code was reduced to the single most time-consuming subroutine. This whole process was carried out in a "development computing environment" so that the "production computing environment" was not overloaded with development tasks and experiments. Table 1 shows the main characteristics of each computing platform. Performance experiments were carried out in both computing environments, and the results are summarized in Table 2, which provides more information in terms of possible performance improvements and research. Initially, it is worth taking into account three factors:

1. We obtained performance improvement with small source code changes.
2. The maximum number of processors (cores) is different in each environment: four in the production environment and eight in the development environment.
3. Only the most time-consuming subroutine has been parallelized.

Table 1 Production and development computing environments

	Production env.	Development env.
CPU	Intel I5-2310 2.9GHz	Intel Xeon 5405 @2.00GHz
Max. # of cores	4	8
CPU launch date	Q2'11	Q4'07

Table 2 Performance experiments, most time-consuming subroutine

	Sequential *t* (s)	Parallel *t* (s)	Speedup	Efficiency
Production env.	6000	3420	1.75	0.44
Development env.	13,500	1950	6.92	0.87

While the second factor explains at least some of the difference in the improvement obtained in the different platforms (more processors *naturally* imply more improvement), the difference in efficiency shows that the parallelization is more effective in the older platform (the one we have called "development environment"). We have taken this *version* of the program as a departure point for the work we explain in the following sections of this paper. Basically, we have chosen to deepen the parallelization process, based on the fact that

- At least in one of the computing environments, we obtained good scalability results (efficiency of about 0.87 with eight processors).
- One of the possible reasons of the low efficiency in the newer hardware (the one we have called "production environment") is that parallelizing a single subroutine may penalize other subroutines performance (e.g., by introducing memory footprints which imply higher cache miss ratio).

Finally, the global gain is relatively small, because there is only a single subroutine being parallelized. Table 3 summarizes performance gains in each environment using all available processors (4 in the production environment and 8 in the development one). The data shown in Table 3 are computed from data in Table V and Table VI in [25]. While the performance gain focuses on net/raw performance improvement, the efficiency highlights the fraction of the available resources effectively used. Also, recall that only one Fortran subroutine is parallelized: the most time-consuming one.

Besides, the *bare* performance result improvements, we found that in this case, GNU performance profiling tools were not good enough for obtaining acceptable profiling data. The results obtained by gprof (with data provided by gcc) were far from being reliable at least regarding the wall-clock time, so we used our own simple profiling by introducing instrumentation code in the program. The

Table 3 Performance experiments, global performance gain

	%Gain	Efficiency
Production env. (13,140 s vs. 11,820 s)	36	0.2
Development env. (31,560 s vs. 20,040 s)	10	0.28

instrumentation is maintained in the performance results we report below, so that we do not change the way in which we obtain performance data.

Summarizing, in our previous work we have acquired experience on the legacy code, parallelized a section of code (the most time-consuming subroutine) and used several performance analysis tools and techniques. However, the work constrained to a single subroutine and also constrained the global performance and efficiency gains, as shown in Table 3. We have found that further analysis on other time-consuming subroutines and specifically Fortran Do loops is too complex in this particular legacy code. Thus, we take a global approach to the parallelization process, a data parallel one, as explained below.

4 Data parallel approach

Taking advantage of the work described in the previous section, we decided to approach the parallelization of the remaining sequential code in the program. It is worth noting that, at this point, there are several pros and cons for a *larger-scale* (i.e., beyond a single subroutine) parallelization. As explained in the previous section, we have chosen to deepen the parallelization process. More specifically, our approach is focused on a larger-scale parallelization process in which more than a single subroutine would be handled. It is worth noting that, at this point, there are several points to remark:

- The rest of the code (other than the most time-consuming subroutine) remains as legacy code.
- Despite the previous item, we are able to take advantage of the experience obtained with the parallelization of a single routine:
    - Almost all the data structures are already known: data arrays and indirection arrays.
    - The coding style includes characteristics such as large number of source code lines by subroutine and large number of subroutine parameters/arguments.
- The source code changes should take into consideration inter-procedural side/collateral effects. Therefore, the optimization process becomes a global-scale work.

From the point of view of HPC, the FEM computing *naturally* leads to a data parallel processing pattern or data parallelism [18]. Data parallelism has been traditionally associated with SIMD (single-instruction multiple-data streams, [5]) hardware and the current GPU (Graphic Processing Units) implementations as explained in [8]. However, we have decided a more standard process/thread-oriented implementation for data parallelism considering:

- The thread-level computing allows a direct harnessing of multi-core facilities of current processors, such as those in the environments mentioned in the previous section. Furthermore, multi-core hardware computing facilities are made almost directly available to programmers via well-known and established programming specifications such as OpenMP, which we have already used.
- Once identified the threads operating on well-defined data or data structures, the SPMD (single-program multiple-data) processing model could be approached [23]. It is not a completely automatic process though and granularity plays a huge role in obtaining acceptable performance gains. Each parallelization scale should be taken initially as a hypothesis to be verified by experimenting on real applications. Distributed memory parallel platforms such as clusters of computers can be used for SPMD implementations based on the message passing programming model [14]. Also, distributed memory parallel architectures allow to easily overcome the problem of a limited (relatively small) shared memory parallel computer. Thus, distributed memory parallel architectures provide greater hardware scalability than shared memory ones.
- SIMD/GPU is not excluded, because each individual partition may be processed via a GPU, i.e., taking advantage of making the same operations in a single data region.
- Data parallelism is mostly focused on data dependencies for computing, which is easily achieved in a FEM approach by spatially partitioning the data in defined geographical regions. Figure 1 schematically shows how the whole simulation process is done by processing nodes belonging to a triangulation mesh. The rest of the section will discuss the alternatives for data distribution among threads.

Clearly, considering data parallelization implies a work of higher abstraction level than that of the previous section. It is not possible to implement data parallelization by looking at specific loops in specific subroutines as in [25]. Instead, we must define one or more conceptual data (mesh) partitions and, later, process them in parallel. Conceptually, data partitioning must be carried out independently of either a shared memory implementation (e.g., threading with OpenMP) or a distributed memory implementation (e.g., SPMD processing with MPI).

Even when data parallelism is independent of the parallel hardware and programming model at the conceptual level, it is not the case at the practical level. In practice, we have to take into account two important details:

1. We are working with a legacy application which, in turn, implies
    - A lot of knowledge about the problem and the numerical issues is already included in the program, and we should reuse all that knowledge without having to solve again the same problems.

- Developing (shared memory) threaded code from an existing serial code is less prone to error than developing another version of the program for SPMD computing. Programming SPMD implies to include explicit message passing operations, usually send and receive MPI functions, which allows processes in different computers to send and receive data, respectively.

2. Beyond and above reuse of knowledge, legacy source code issues and programming complexities, we necessarily have to deal with performance. The main objective for the whole parallel approach is, in general, to improve performance. In this context, we have the previous work as a good indication that at least some performance improvement is feasible.

We will approach a data parallel version of the whole program taking into account the explanations and considerations above. We do not approach the whole Fortran program as a software engineering source code upgrade project [10], e.g., for full Fortran 2003 [15] or Fortran 2008 [3] or Fortran 2018 [7] code. Instead, we focus on the triangularization mesh partitioning for data parallel processing, as explained in the next section.

5 Data distribution and balanced workload

The main ideas for the data parallel computing in this work are

- Taking advantage of the reuse of the legacy FEM code, so that numeric results are maintained stable, and comparable to the output of the original version.
- Having more partitions than processors in order to avoid data dependency conflicts neighboring partitions.
- Planning for computing non-neighboring data partitions in parallel (simultaneously), because they do not have any data dependency requirement.

Partitioning and planning are defined as preprocessing steps, i.e., the actual code of the program is not changed or adapted for any specific number of processors used at runtime. Partitions are defined and scheduled so that the program processes partitions according to the data provided as input. We also consider the characteristics of the triangularization mesh, as shown in Fig. 1, including radial densification symmetry around a circular obstacle and a high densification in the lateral edges. Those characteristics imply that density is symmetric in radial coordinates around obstacles and is greater in the lateral edges. Thus, a partitioning method cannot follow a traditional *striping* model by partitioning either the *x*-axis or the *y*-axis for equally sized areas as schematically shown in Figs. 2 and 3, respectively. Areas *containing* parts or the whole obstacle and/or lateral edges will have much more data to process than the other ones. And a partitioning that follows the *striping* model/s will impose an unbalanced parallel workload among processors. We analyze the non-uniform triangulation in order to define a partitioning scheme maintaining an acceptable balanced workload for parallel processing. The following description of the partitioning and planning preprocessing

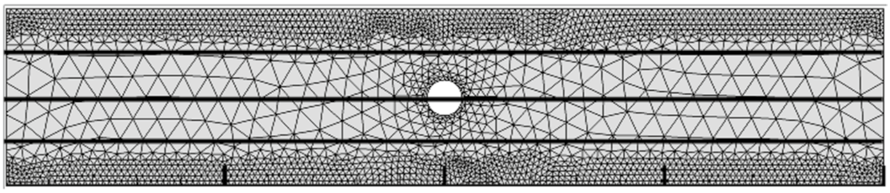

Fig. 2 Mesh horizontal partitioning

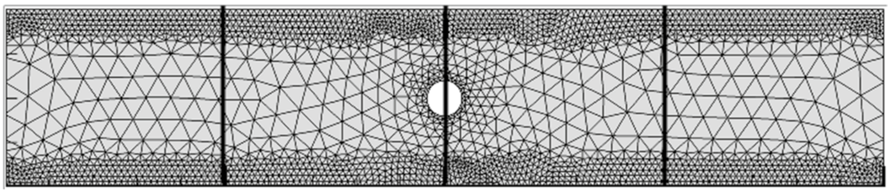

Fig. 3 Mesh vertical partitioning

steps is made as in an ideal scenario, i.e., assuming there are no issues for obtaining equally sized partitions. Basically, we define partitions by

- Radial densification symmetry around an obstacle would lead to a radial mesh partitioning relative to the obstacle.
- In general, the partitioning algorithm starts defining the number of partitions as twice the number of processors to be used.

Having a number of partitions as twice the number of processors allows planning the entire computing process as a sequence of two phases of simultaneous computing with all the available processors, i.e., with a single synchronization point. Thus, having p processors we will define $2 \times p$ almost equally sized partitions and an entire FEM computing step will be made with a single synchronization point, i.e., half of the non-neighboring partitions in the first phase and the other half of the non-neighboring partitions in a second computing phase. Both steps will use all the available processors without any intermediate synchronization, because partitions are non-neighbors, i.e., they do not have any computing data dependency. More specifically, we can identify each partition as $part_1, part_2, \dots, part_{2\times p}$, where $part_i$ has two neighboring partitions: $part_j$ and $part_k$ with

$$j = \begin{cases} i+1 & \text{if } i < 2 \times p \\ 1 & \text{if } i = 2 \times p \end{cases} \qquad k = \begin{cases} i-1 & \text{if } i > 1 \\ 2 \times p & \text{if } i = 1 \end{cases}$$

as shown in Fig. 4 for $p = 4$. According to this example, the first computing phase will process $part_i$ for $i = 1, 3, 5, 7$, the second computing phase will process $part_i$ for $i = 2, 4, 6, 8$, and every processor will be busy in each phase provided the partitions have (approximately) the same number of triangularization mesh nodes. Besides, if the partitions are almost equal, the intermediate synchronization time between the phases will be minimal.

5.1 Dealing with unbalanced partitioning and special cases

The previous description is made assuming that a mesh radial partitioning produces partitions containing almost the same number mesh nodes. However, a radial partition does not always *capture* the unbalance produced by high densification in the lateral edges. Then, we must define an adaptive algorithm for *balancing* the number of nodes in each partition. The algorithm uses a threshold, *delta*, for considering a balanced partitioning, so that Eq. 11

$$|\#nodes(part_i) - \#nodes(part_j)| \leq \delta \quad 1 \leq i,j \leq \#partitions \tag{11}$$

is used for defining a balanced partitioning, where

- $\#partitions$ is the total number of partitions.
- $\#nodes(part_k)$, $k = 1, \dots, \#partitions$, is the number of nodes (data to be processed) in partition $part_k$

The initial partitioning is made as explained, i.e., (a) the number of partitions is $2 \times p$, being p the total number of available processors, and (b) partition areas are defined subdividing the total geographical area in radial sectors, taking the obstacle as the central point, as shown in Fig. 5. There are several factors for generating different number of nodes in each partition, even beyond a δ threshold, such as

- The high number of triangles resulting from high densification of lateral edges, which is not *captured* by the radial criteria around the obstacle.
- Rectangular simulation areas such as that of Fig. 5 that naturally define several partitions with larger subareas than others.

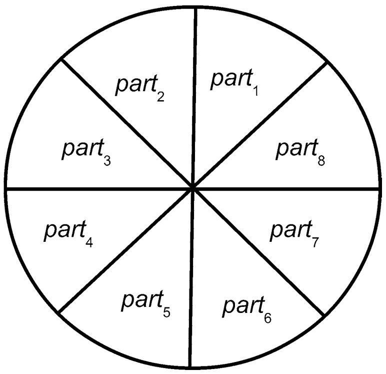

Fig. 4 Partitioning for $p = 4$

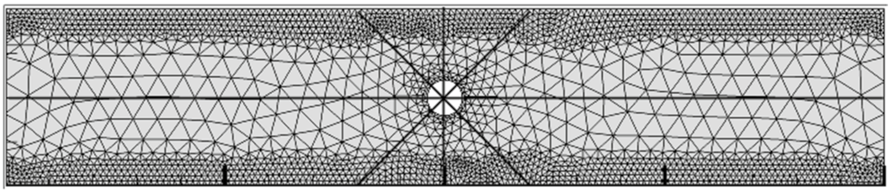

Fig. 5 Initial mesh partitioning

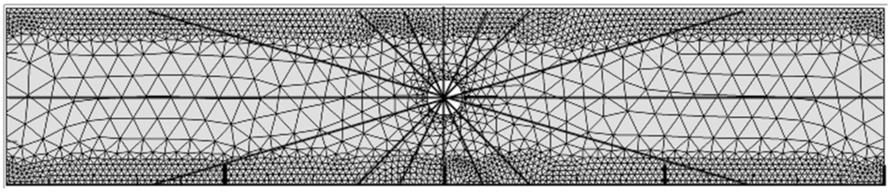

Fig. 6 Initial mesh partitioning

If the different number of triangles per partition is greater than a threshold δ for one or more partitions pair as defined in Eq. (11), we apply the following *(re)balance* algorithm:

1. Identify half of the partitions with less number of elements.
2. Compute avg_{half}, the average of elements of the partitions identified in the previous step.
3. For each of the unbalanced partitions $part_i$, compute the number of subdivisions in which $part_i$ will be subdivided as $subdiv = \#nodes(part_i)/avg_{half}$.
4. If the resulting partitioning is balanced according to Eq. (11), then nothing else needs to be done. Otherwise, go to step 1 using the actual partitioning.

Figure 6 schematically shows a possible result after using the balance algorithm given above on the initial partitioning shown in Fig. 5. The visual effect of Fig. 5 may not necessarily reflect the partitioning balance due to the non-regularity of the mesh and the δ threshold of *acceptable unbalance* between partitions. As expected, the δ threshold is defined in relative terms, as a % of acceptable difference between the number of nodes of any two partitions.

Even when theoretically each node belongs to a single mesh partition, several issues arise at the implementation level of abstraction, because a node actually means a triangle. A triangle is defined by three vertices, and depending on geometry, it is possible that a single triangle belongs to more than one partition as defined by the polar partitioning. Even when most of the triangles will have the vertices in a single partition, there are exceptions:

- Two vertices belong to a single partition, and the other vertex belongs to a different partition.

- All of the three vertices belong to different partitions. Triangle edges near the center of the radial partitioning are more likely to be in more than two partitions than those in the edges of the geographical area.

For the former case, in which two vertices belong to a single partition, we assign the triangle to the partition holding those two vertices. Triangles with vertices in three different partitions are defined as *special triangles*. Special triangles are included in a special partition and sequentially processed, in order to avoid erroneous numerical results because of data dependency issues in the parallel processing of partitions.

It is worth mentioning that the repartitioning algorithm has several practical limits and issues regarding a large number of mesh partitions:

- Too many partitions imply too little number of nodes per partition which, in turn, imply a small granularity and the corresponding penalty due to many synchronization points.
- As the number of partitions grows, the geographical area of each partition is smaller and the number of special triangles proportionally increases. We should maintain the number of special triangles relatively small given that they are processed sequentially, i.e., generating the corresponding performance penalty.

The previous limits/issues imply we have a δ threshold as large as that corresponding to a 25% of *acceptable* unbalance. It is worth noting that the unbalance is relative to half of the processing, since we start with a number of partitions as twice the number of available processors. Also, part of the unbalanced workload can be avoided by a correct ordering of partition processing, i.e., by the *appropriate* planning, as explained in the next subsection.

5.2 Planning: partitions processing scheduling

Once defined the mesh partitioning, it is necessary to schedule each partition processing, i.e., computing the FEM on actual data. Having a number of partitions of at least twice the number of processors available allows computing the whole data set with relatively little number of synchronizations. In the best scenario, we will have $2 \times p$ equally sized partitions for computing on p processors. And this best case for processing can be carried out with a single synchronization point as follows:

- Select and process a non-adjacent half of partitions, i.e., p partitions to be asynchronously processed in parallel. Figure 5 shows an example with 8 partitions, where partitions 1, 3, 5 and 7 do not have any data dependency between each other, and they contain non-neighboring geographical regions and, thus, can be processed asynchronously in parallel.
- Select and process the *second half* of the partitions, i.e., the remaining p partitions to be asynchronously processed in parallel. Following the example of Fig. 5, partitions 2, 4, 6 and 8 do not have any data dependency between each

other, and they contain non-neighboring geographical regions and, thus, can be processed asynchronously in parallel.

The single synchronization point is necessary between processing the first and second half of partitions, because partitions in different halves do contain processing data dependencies. Take into account that processing non-adjacent partitions allow to overcome the problem of potential synchronization at single triangle level. Otherwise, i.e., in case adjacent partitions are processed in parallel, data dependencies have to be controlled at each triangle so that numerical results are not affected depending on the relative order in which adjacent triangles are processed. Thus, ensuring non-adjacent partitions processing we avoid triangle-level synchronization and allow partition-level (i.e., sets of triangles) synchronization.

There are several scenarios in which there is more than a single synchronization point, and specific planning is required. A number of partitions multiple of p (number of processors available) are unlikely to happen whenever the (re)balance algorithm is used. Unbalanced partitions require repartitioning some of them, and the new total number of partitions tends to be below $6 \times p$. There are a number of reasons for that number, mostly related to the specific mesh and mesh densification areas, so it is possible that in other scenarios the final number of partitions will be different (while always greater than $2 \times p$). Besides, the so-called *special partition* (i.e., the set of triangles with vertices initially in more than two partitions) must be sequentially processed.

In case there is a special partition, it is sequentially processed, as explained above, which implies an "extra" synchronization point. However, since the number of partitions is relatively small for avoiding many synchronization points as well as small processing granularity, the number of triangles in the special partition is also relatively small, below 5% of the total number of triangles to be used in the whole computation. In case there are more than $2 \times p$ partitions due to the balance algorithm, there are more synchronization points. Basically, there may be as many computing phases as the total of partitions divided by p, the number of processors. There is some extra planning optimization as well, while maintaining that non-adjacent partitions are processed in each computing phase. We did not dig into much detail on extra optimization possibilities given that the number of total partitions we had to process is not too large in any of the experiments.

Summarizing, the different scenarios depending on partitioning, which, in turn, depends on the specific non-homogenous mesh triangularization, define the planning as

- Best case: two computing phases, with a single synchronization point between them.
- Number of partitions $prt > 2 \times p$ partitions: as many computing phases as $cp = \lceil prt/p \rceil$ with $cp - 1$ synchronization points.
- If there is a special partition, it always implies adding another (sequential) computing phase with its corresponding extra synchronization point.

6 Experimentation and analysis of results

We have carried out a relatively large number of performance experiments in order to evaluate our proposal. Since we have a large number of *raw* performance timing measurements, we present the performance results in terms of relative improvement enhancements for a better understanding the impact of our approach. We initially carried out sequential experiments to identify the time difference (i.e., overhead) produced by the new code arrangement. The processing time overhead was less than ten percent. There were no major source code changes, and the numerical results were consistent in all the experiments.

The first parallel computing experiments were carried out without any control of workload balance. We define the $2 \times p$ radially defined partitions as explained in the beginning of Sect. 5. The angle was chosen to keep $2 \times p$ partitions. Parallel computing is realized via OpenMP, mostly because only relatively small source code changes have been introduced in the legacy software. We follow the presentation of performance results as that introduced in Sect. 3, according to the so-called "development" and "production" environments, respectively, described in Table 1.

Table 4 shows performance results of the new parallel version, i.e., with radial partitioning. Experiments in the development environment are scaled from 1 to 8 processors running in parallel (with 2 and 4 processors as intermediate values). It is worth taking into account that the processors are already available in both platforms, i.e., the parallel computing version is able to take advantage of every computing facility in each parallel computer.

The performance gain is maximized with the rebalancing and planning methods also explained in the previous section. As a result, the performance gain is 54% in the production environment and 66% in the development environment. (Rebalancing and planning introduce a small additional performance gain.) Table 5 summarizes the global performance results of the initial parallel version in column labeled as "Initial %Gain" (those from Table 4) and the best parallel version in column labeled as "Best %Gain," in both hardware platforms. There are two main reasons for the large performance gain in this new parallel version as compared to the original version: (a) the enhanced workload balance, taking into account the radial density around the obstacle, and (b) the global parallel processing, where not only the most time-consuming subroutine is parallelized.

The global parallelization presented in this paper takes advantage of the previous work of parallelizing the most time-consuming subroutine [25], providing a general approach to partitioning and runtime planning. As a result, we have been able to enhance performance up to 54% and 66% in the production and development environments, respectively, without changing the original mesh definition, and with minor

Table 4 Performance gain experiments

	#Processors-%Gain		
Production env.	2p-40%	4p-51%	
Development env.	2p-35%	4p-47%	8p-61%

Table 5 Performance gain experiments

	Initial %Gain	Best %Gain
Production env.	36%	54%
Development env.	10%	66%

source code changes. The source code is available at https://bitbucket.org/maxiperezunlam/sw2d_paralelo/src/master/. If instructions about partitioning software, shallow water software or profiling files are required, please send an e-mail to the authors.

We plan to use this partitioning method for parallel processing even in distributed memory parallel computers. In this case, we will have to carefully consider

- Granularity issues, because explicit data transfers will be needed, thus adding overhead.
- Stronger source code changes, as compared to the OpenMP/shared memory approach in this paper.

We have also made experiments on a new production environment, with a better processor: Intel Core i7-7700HQ @2.8GHz, and performance results are summarized in Table 6. Results are very similar to those in Table 5 for the production environments. Performance improvements are maintained almost constant even when the new production environment is about five years newer than the previous one (Q2'11 vs. Q1'17) and a better processor model (i5-2310 vs. i7-7700HQ).

7 Conclusions and further work

We have presented the current state of a process, conceived to be of incremental nature, of performance enhancement via parallelization on a specific legacy code. We continued the work already started and published in a previous paper. We took advantage of previous experience on the legacy code, which has resulted in performance enhancements between 51 and 66%, obtained through a global parallel processing approach. We consider our approach as a *proof of concept* of what can be made on a legacy numerical code, expected to take advantage of current parallel computing hardware platforms. We have chosen to minimize the source code changes in order to avoid introducing bugs and numerical instabilities, while adding parallel computing. In particular, the mesh of points on which the sequential computation takes place is maintained unchanged, and uses a data parallel approach which allowed us to have better performance gain than in previous work.

Table 6 Performance gain experiments in new production environment

	#Processors-%Gain	
New production env.	2p-38%	4p-54%

We implemented a specific mesh partitioning taking into account non-uniform mesh densities, with a rebalancing algorithm for unbalanced partitions due to combined different mesh densifications. We pay particular attention to maintain the original mesh unchanged, i.e., we do not redefine the mesh nor mesh densification/s, and thus, the whole numerical approach is maintained as in the initial legacy code. Maintaining the numerical approach, in turn:

- Implies the minimum amount of source code changes, reducing the likelihood of introducing bugs in a source code hard to read and modify as every legacy code.
- Simplifies source code changes verification/validation, because the same numerical expected results should be found, and the mesh and numerical processing has not been changed at all.

Besides, we think the partitioning and rebalancing algorithms could be used in many any similar problems, i.e., one in which the mesh is not uniform, in the sense of having different densification areas. A priori, we do not think this approach could be adopted in other more complex scenarios, such as those in which the mesh is completely irregular. Also, since the partitioning is focused on distributing data, we expect to take advantage of the experience for a distributed memory parallel approach as a possible following enhancement. We also expect to work in identifying current performance penalties/bottlenecks for implementing further performance improvements.

We also plan to solve the control problem for the 2D shallow water equations [22] to get a regulated solution of flood wave propagation. And this new control routine should be optimized and parallelized with HPC techniques, including the resulting code in the original simulation work. Results will be verified using the open-source software Telemac2D-Mascaret.

References

1. Ansari SU, Hussain M, Mazhar S, Manzoor T, Siddiqui KJ, Abid M, Jamal H (2017) Mesh partitioning and efficient equation solving techniques by distributed finite element methods: a survey. Arch Comput Methods Eng 26(1):1–16
2. Bermúdez A, López X, Vázquez-Cendón ME (2017) Finite volume methods for multi-component Euler equations with source terms. Comput Fluids 156:113–134
3. Brainerd WS (2015) Guide to Fortran 2008 Programming, 2nd edn. Springer, Berlin
4. Feng D, Tsolakis C, Chernikov AN, Chrisochoides NP (2017) Scalable 3D hybrid parallel Delaunay image-to-mesh conversion algorithm for distributed shared memory architecture. Comput Aided Des 85:10–19
5. Flynn MJ (1972) Some computer organizations and their effectiveness. IEEE Trans Comput C 21(9):948–960. https://doi.org/10.1109/TC.1972.5009071
6. Hussain M, Abid M, Ahmad M, Hussain SF (2013) A parallel 2D stabilized finite element method for Darcy flow on distributed systems. World Appl Sci J 27(9):1119–1125
7. ISO, ISO/IEC 1539-1:2018 Information technology - Programming languages - Fortran - Part 1: Base language
8. Kirk DB, Wen-Mei WH (2012) Programming massively parallel processors: a hands-on approach, 2nd edn. Morgan Kaufmann, Burlington (ISBN 978-0-12-415992-1)

9. Kong F, Stogner RH, Gaston DR, Peterson JW, Permann CJ, Slaughter AE, Martineau RC (2018) A general-purpose hierarchical mesh partitioning method with node balancing strategies for large-scale numerical simulations. In: 9th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 18, Nov. 11–16, Kay Bailey Hutchison Convention Center, Dallas, TX, USA
10. Krommydas K, Sathre P, Sasanka R, Feng W (2018) A framework for auto-parallelization and code generation: an integrative case study with legacy FORTRAN codes. In: Proceedings of the 47th International Conference on Parallel Processing, ICPP 2018, Eugene, OR, USA. https://doi.org/10.1145/3225058.3225143
11. Larwood BG, Weatherill NP, Hassan O, Morgan K (2003) Domain Decomposition approach for parallel unstructured mesh generation. Int J Numer Methods Eng 58(2):177–188
12. LaSalle D, Karypis G (2013) Multi-threaded graph partitioning. In: 27th IEEE International Parallel and Distributed Processing Symposium (IPDPS), Boston. MA, United States. https://doi.org/10.1109/IPDPS.2013.50
13. Li X, Yu W, Liu C (2017) Geometry-aware partitioning of complex domains for parallel quad meshing. Comput Aided Des 85:20–33
14. Message Passing Interface Forum, MPI: A Message-Passing Interface Standard Version 3.1, 2015
15. Metcalf M, Reid J, Cohen M (2004) Fortran 95/2003 explained. Oxford University Press, Oxford
16. Nguyen C, Rhodes PJ (2018) TIPP: parallel Delaunay triangulation for large-scale datasets. In: SSDBM '18 Proceedings of the 30th International Conference on Scientific and Statistical Database Management, Bozen-Bolzano, Italy. https://doi.org/10.1145/3221269.3223034
17. OpenMP Architecture Review Board, OpenMP Application Programming Interface, Version 5.0, Nov. 2018. https://www.openmp.org/specifications/. Accessed Jan 2020
18. Pacheco P (2011) An introduction to parallel programming. Morgan Kaufmann, Burlington
19. Roig B (2007) One-step Taylor Galerkin methods for convection diffusion problems. Comput Appl Math 204:95–101
20. Shang Z (2013) Performance analysis of large scale parallel CFD computing based on Code Saturne. Comput Phys Commun 184:381–386
21. Shang Z (2014) Impact of mesh portioning methods in CFD for large scale parallel computing. Comput Fluids 103:1–5
22. Soumendra NK, Kiran P (2008) Finite volume model for shallow water equations with improved treatment of source terms. J Hydraul Eng 134:231–242
23. Sprenger S, Zeuch S, Leser U (2018) Exploiting automatic vectorization to employ SPMD on SIMD registers. In: 2018 IEEE 34th International Conference on Data Engineering Workshops (ICDEW), Paris, pp 90–95, https://doi.org/10.1109/ICDEW.2018.00022
24. Tinetti FG, Méndez M, De Giusti A (2013) Restructuring Fortran legacy applications for parallel computing in multiprocessors. J Supercomput 64(2):638–659. https://doi.org/10.1007/s11227-012-0863-x
25. Tinetti FG, Perez MJ, Fraidenraich A, Altenberg AE (2018) Experiences in parallelizing a numerical model. In: International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA'18, Las Vegas, NV, July 30–Aug. 2, ISBN: 1-60132-487-1, CSREA Press, pp 152–157
26. Zienkiewicz OC, Taylor RL (2005) The finite element method, vol 3, 6th edn. Butterworth-Heinemann, Oxford