HAL
archives-ouvertes.fr

Task-based programming for Seismic Imaging: Preliminary Results

Lionel Boillot, George Bosilca, Emmanuel Agullo, Henri Calandra

▶ To cite this version:

Lionel Boillot, George Bosilca, Emmanuel Agullo, Henri Calandra. Task-based programming for Seismic Imaging: Preliminary Results. IEEE 16th International Conference on High Performance Computing and Communications (HPCC), Aug 2014, Paris, France. pp.1259-1266, 10.1109/HPCC.2014.205. hal-01057580v2

HAL Id: hal-01057580
https://hal.inria.fr/hal-01057580v2

Submitted on 16 Mar 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Copyright

Task-based programming for Seismic Imaging: Preliminary Results

Lionel Boillot*, George Bosilca†, Emmanuel Agullo‡ and Henri Calandra§
*INRIA Bordeaux Sud-Ouest, Magique-3D project team
Avenue de l'Université, BP 1155, 64013 Pau cedex, France
†ICL, University of Tennessee, EECS department
1122 Volunteer Blvd, Knoxville TN 37996, USA
‡INRIA Bordeaux Sud-Ouest, HiePACS project team
200 avenue de la Vieille Tour, 33405 Talence cedex, France
§TOTAL EP, Depth Imaging and High Performance Computing
1201 Louisiana street, suite 1800, Houston TX 77057, USA

Abstract—**The level of hardware complexity of current supercomputers is forcing the High Performance Computing (HPC) community to reconsider parallel programming paradigms and standards. The high-level of hardware abstraction provided by task-based paradigms make them excellent candidates for writing portable codes that can consistently deliver high performance across a wide range of platforms. While this paradigm has proved efficient for achieving such goals for dense and sparse linear solvers, it is yet to be demonstrated that industrial parallel codes—relying on the classical Message Passing Interface (MPI) standard and that accumulate dozens of years of expertise (and countless lines of code)—may be revisited to turn them into efficient task-based programs. In this paper, we study the applicability of task-based programming in the case of a Reverse Time Migration (RTM) application for Seismic Imaging. The initial MPI-based application is turned into a task-based code executed on top of the PaRSEC runtime system. Preliminary results show that the approach is competitive with (and even potentially superior to) the original MPI code on a homogeneous multicore node, and can more efficiently exploit complex hardware such as a cache coherent Non Uniform Memory Access (ccNUMA) node or an Intel Xeon Phi accelerator.**

I. Introduction

As of today, most HPC applications are coded with programming paradigms designed specifically to be relatively close to the underneath hardware architecture. Undoubtedly relevant for achieving high performance, this prominent approach also allowed codes to remain compact while a single level of parallelism was at stake. As a result, HPC applications targeting shared memory machines have mainly been written with Posix threads (pthread), or, more recently, with OpenMP [1], whereas codes targeting distributed memory computers have relied on MPI. However, now that the level of hardware complexity is steadily increasing, staying at this low programming level requires reliance on multiple, sometimes conflicting, programming paradigms. In the past decade, many research groups and companies made the effort to port their MPI codes to MPI+thread in order to better fit the multicore paradigm and associated supercomputers. With the emergence of GPGPUs, or, more recently, co-processors in the TOP500 list [2], the same applications have further been extended to handle accelerators. Maintaining these codes while achieving high performance across platforms can then become an extremely complex, error and performance prone, time-consuming task.

Newer programming paradigms, especially task-based approaches, allow for a higher level of hardware abstraction. At the core of many numerical simulations, dense linear algebra kernels have been redesigned in terms of task-based algorithms for multicore [3], [4], accelerator-based [5]–[8] and distributed memory [9] machines. This research resulted in new production solvers such as PLASMA and MAGMA [10], FLAME [11], and, more recently, DPLASMA [12]. This paradigm has also been assessed in the context of more irregular algorithms such as sparse direct methods [13], [14], sparse iterative Krylov methods [15] or fast algorithms [16]–[19]. However, it is yet to be demonstrated that large industrial parallel codes accumulating dozens of years of expertise may be turned into efficient task-based programs. In this paper, we consider a highly optimized MPI-based Reverse Time Migration (RTM) industrial code for Seismic Imaging developed at Total, that we convert into a task-based program.

There exist different methods—and Application Programming Interfaces (APIs)—for programming task-based systems. The most prominent ones are sequential and parametrized task-based programming, respectively. Sequential task-based programs automatically infer dependencies from the sequence of tasks and data hazards [20], whereas parametrized task-based programming [21] requires the dependencies to be provided explicitly without the sequential order of the tasks. An exhaustive presentation of the runtime systems supporting task-based programming is out of the scope of this paper. Here, we only mention some of the most prominent projects onto which the above mentioned solvers have been designed; Quark [22], SMPSs [23], StarPU [24], and SuperMatrix [25] mainly (but not only) support sequential task-based programming, while P*a*RSEC [26] and CnC [27] mainly support parametrized task-based programming. The numerical scheme studied in this manuscript is a wave propagation discretized with a Discontinuous Galerkin (DG) method and a Leap-Frog time scheme [28] (see Section II). As a result, contrary to Finite Element Methods (FEM), there is no main matrix to invert, but only local subproblems to explicitly solve. It is thus natural to rely on a parametrized expression by expressing both tasks (how to compute a subdomain) and dependencies

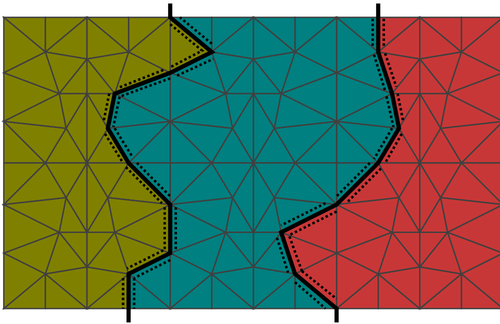


Fig. 1: Mesh partitioning of a surface into triangle elements. In the MPI reference code, over three processes, the whole domain is split into three subdomains (green, blue, and red from left to right). The dot lines represent the boundary data which have to be exchanged between the subdomains.

(how to exchange with subdomains at its interface) using a symbolic expression as further explained in Section IV. In addition to evaluating the complexity of porting the target application toward a task-based runtime, our goal was to study the behavior of the resulting task-based version in different scenarios. We are particularly interested in two types of execution environments, a ccNUMA node (Section V-C) and an Intel Xeon Phi co-processor (Section V-D). We rely on the P*a*RSEC runtime system, a distributed runtime specifically designed for complex hierarchical architectures (see Section III). Note that the comparison with other runtime systems, although certainly interesting, is not the objective of the paper. Instead, we compare the obtained preliminary results against the performance of the highly optimized MPI-based original code.

Starting from the original MPI-based wave propagation code (Section II) and the P*a*RSEC runtime system (Section III), our contribution consists of designing a task-based version of this wave propagation algorithm (Section IV) and assessing the benefits of using such an approach on top of P*a*RSEC on modern processors (Section V). The rest of the paper is organized as follows. Section II depicts the wave propagation scheme and discretization we use for performing the RTM. Section III explains how to program parametrized task-based applications with the P*a*RSEC runtime system with a special emphasis on how to efficiently exploit the architectures discussed in this study. Section IV then shows how to turn the wave propagation scheme into a parametrized task-based code. A preliminary performance study and comparison of the resulting task-based code with the original MPI code is presented in Section V before concluding in Section VI.

II. Wave propagation in a Nutshell

Modern seismic imaging is frequently performed with RTM techniques. Its core computational kernel is the solution of a wave propagation problem. In this study, we consider a velocity-stress formulation of an elastic wave problem [29] which we compute on an open bounded domain Ω of $\mathbb{R}^3$ during a period of time T. If we note $\mathbf{x} = (x, y, z) \in \Omega$ and $t \in [0, T]$ as the space and time variables, the velocity-stress formulation

1: *//– In sequential on the whole domain*
2: Initialization($mesh, matrix, v_h, \sigma_h$)
3: DomainDecomposition($mesh$)
4: *//– In parallel on each subdomain*
5: **for** $n = 1$ **to** $n_timesteps_T$ **do**
6: ExchangeBoundaryStress($\sigma_h^{n+1/2}$)
7: $v_h^{n+1} \leftarrow$ ComputeVelocity($v_h^n, \sigma_h^{n+1/2}, \Delta_t$)
8: ExchangeBoundaryVelocity(v_h^{n+1})
9: $\sigma_h^{n+3/2} \leftarrow$ ComputeStress($\sigma_h^{n+1/2}, v_h^{n+1}, \Delta_t$)
10: **end for**

Fig. 2: Parallel computation of the elastic wave propagation problem discretized in space and time with DG and Leap-Frog methods, respectively.

of the elastic wave equation can be written as:

$$\begin{cases} \rho(\mathbf{x})\partial_t v(\mathbf{x}, t) & = \nabla . \underline{\underline{\sigma}}(\mathbf{x}, t) \\ \partial_t \underline{\underline{\sigma}}(\mathbf{x}, t) & = \underline{\underline{C}}(\mathbf{x}) : \underline{\underline{\epsilon}}(v(\mathbf{x}, t)) \end{cases} \qquad (1)$$

with $\rho > 0$ the density, $v \in \mathbf{H}^1(\Omega \times [0, T])$ the unknown velocity field, $\underline{\underline{\sigma}} \in \underline{\underline{H}}_{div}(\Omega \times [0, T])$ the stress tensor, $\underline{\underline{C}}$ the stiffness tensor (elasticity coefficients), $\underline{\underline{\epsilon}}(v) = \frac{1}{2}(\boldsymbol{\nabla} v + (\boldsymbol{\nabla} v)^T)$ the strain tensor, ∇, and $\boldsymbol{\nabla}$ being the divergence and gradient operators, respectively.

We discretize this continuous problem in space and time as follows. A mesh generator partitions the domain Ω into a polygonal mesh Ω_h composed of tetrahedra K. As we rely on DG (and contrary to FEM), the functions v and $\underline{\underline{\sigma}}$ are approximated with discontinuous functions v_h and $\underline{\underline{\sigma}}_h$ that only satisfy $\{v_h, \underline{\underline{\sigma}}_h\} \in L^2(\Omega_h \times [0, T])$ on the global space, maintaining $\{v_h|_K, \underline{\underline{\sigma}}_h|_K\} \in \mathbb{P}^l$ only locally on each polyhedron K. $\mathbb{P}^q$ is the set of polynomials of a degree less than or equal to q. This order q can vary on every tetrahedron. The time discretization is performed with an implicit time scheme using a Leap-Frog method. The time domain $[0, T]$ is divided into time steps Δt. If v_h^n is the approximation of the velocity $v_h(t)$ at the discrete time $t = n\Delta t$, and $\underline{\underline{\sigma}}_h^{n+1/2}$ is the approximation of the stress tensor $\underline{\underline{\sigma}}_h(t)$ at the discrete time $t = (n + \frac{1}{2})\Delta t$, the discrete scheme of system (1) becomes:

$$\begin{cases} M_v \dfrac{v_h^{n+1} - v_h^n}{\Delta t} + R_{\underline{\underline{\sigma}}} \underline{\underline{\sigma}}_h^{n+1/2} & = 0 \qquad (2a) \\ M_{\underline{\underline{\sigma}}} \dfrac{\underline{\underline{\sigma}}_h^{n+3/2} - \underline{\underline{\sigma}}_h^{n+1/2}}{\Delta t} + R_v v_h^{n+1} & = 0 \qquad (2b) \end{cases}$$

Because DG leads to block-diagonal M_v and $M_{\underline{\underline{\sigma}}}$ matrices, their inversion may be performed locally. As a result, the method can be viewed as quasi-explicit, parallelism being extracted with a pattern similar to stencil computation. At each time step, the velocity and the stress tensor can be computed within a tetrahedron knowing the corresponding values at its interfaces (surface common with its neighbors) with equations (2a) and (2b), respectively. In a parallel computation, once the mesh has been computed (Initialization step at line 2 in Figure 2), the domain Ω_h of the whole mesh is split into multiple subdomains (DomainDecomposition step at line 3). Figure 1 shows the mesh partitioning of a surface into triangle elements and the decomposition into three

```
1: ComputeStress(it, d)
2:     it = 1 .. nb_timesteps
3:     d  = 1 .. nb_subdomains
4: READ V ← V Unpack_V(it, d, 1 .. nb_neigh(d))
5:            → (it != N) ? V Compute_V(it+1, d)
6: RW S ← S Compute_V(it, d)
7:          → (it != N) ? S Pack_S(it+1, d, 1 .. nb_neigh(d))
```

Fig. 3: JDF expression of a (simplified) ComputeStress task

subdomains. At each discrete time step n, processes exchange the stress tensor values at the boundaries of their subdomain with their neighbors (ExchangeBoundaryStress at line 6) and update the local value of the velocity (ComputeVelocity at line 7), based on Equation (2a). Processes then exchange these values (ExchangeBoundaryVelocity at line 8) and the time step is completed with a local update of the stress tensor (ComputeStress at line 8) based on Equation (2b).

This algorithm was implemented on top of MPI and optimized for integration in Total's seismic imaging code. The extremely parallel nature of the scheme allows for a high parallel efficiency. However, depending on the physical problem, the order of discretization of the elements can vary, modifying the number of degrees of freedom inside each polyhedron, and, hence, the computation time. This defines a weight for each domain, but due to heterogeneity of memory costs, computational capabilities (e.g., vectorization) and so on, estimating the computation time for a domain, on every kind of hardware architecture, is not a straightforward process. Because a process is associated with a single domain, load imbalance may occur. We assess the limit of this approach both on a homogeneous multicore architecture in Section V-B and on more complex architectures in sections V-C and V-D.

III. P*a*RSEC RUNTIME SYSTEM

Task-based runtime systems have properties that make them more versatile than legacy execution models. For example, as it manages the execution, a runtime can perform dynamic, opportunistic scheduling decisions. It can also orchestrate an adaptive response to ongoing conditions of the managed resources by detecting stress conditions (e.g., idling accelerators, load imbalance, network congestions, etc.), and adapting the way it maps and schedules computations onto resources. At the same time, a runtime can minimize inter-node data transfers across the network or intra-node data transfers between different memory banks. However, in order to efficiently maneuver these concepts, the runtime should have access to a large degree of parallelism, directly exposed by the algorithms. In this context, we investigated a framework which alleviates some of the challenges imposed by the changes at the hardware level described above, namely the P*a*RSEC runtime, a generic framework for architecture-aware scheduling and management of micro-tasks on distributed many-core heterogeneous architectures. We emphasize the fact that such an approach provides a portable way to adapt algorithms to future hardware trends.

A dynamic runtime is only one side of the necessary abstraction. Without access to the internals of the algorithms to

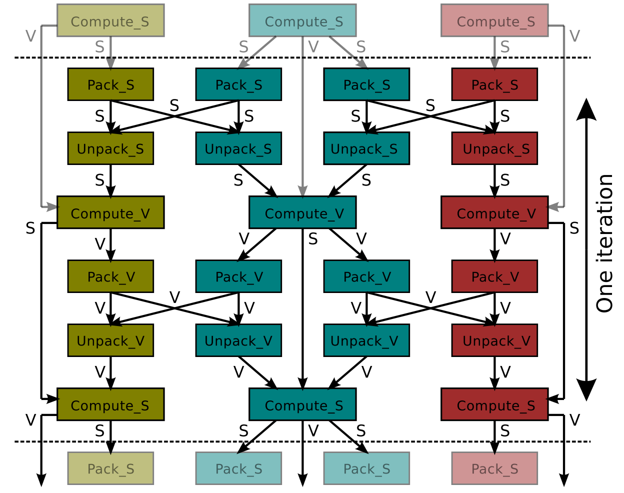


Fig. 4: One iteration of the task-based elastic wave propagation, using the domain decomposition in Figure 1.

expose the maximal parallelism, a runtime is bound to a limited view of the possible execution space. Thus, an efficient runtime must be supported by an algorithmic description capable of exposing the maximum concurrency available at the application level, allowing the runtime to keep all the computing units as busy as possible. This calls for an expression of the parallelism that is practical to end-users, expressive, and avoids cumbersome restrictions that prevent flexible scheduling of operations on heterogeneous hardware. For that, P*a*RSEC proposes a high-level dataflow expression and also permits the use of virtual processes in order to make the most of hierarchical architectures. We now present these two features.

Dataflow expression: A Parameterized Task Graph (PTG) [21] is a compact and convenient expression for representing a Directed Acyclic Graph (DAG) of tasks. The Job Data Flow (JDF) language is an extension of the PTG expression specifically designed for implementing task-based algorithms on top of the P*a*RSEC runtime system. It consists of a symbolic representation of the execution space, and of the data dependencies between tasks, allowing for a dynamic discovery, and generation, of the applications tasks. The JDF expresses the relationships between tasks in terms of annotated data flowing from one task to another. Without delving into the details, the language has a simple interface describing the prerequisite input flows for each task, and the resulting flows to be propagated upon the task completion. Another possible view is that the language allows us to mathematically describe (using a symbolic representation) the predecessors and successors of every possible task in the algorithm. An example of a JDF representation for one of the tasks of our target application (the ComputeStress task) is provided in Figure 3, and will be further described in the Section IV. Simply note that the runtime system parses this expression to ensure that the tasks are generated, and only marks them as ready to be executed once all their predecessors are completed.

Virtual processes: Another interesting capability of P*a*RSEC, especially critical on ccNUMA machines, is the deep integration with the underlying hardware configuration. Indeed, P*a*RSEC adapts the number and placement of the execution streams based on the architecture, allowing for a

simple, yet hierarchical, view of the target system. As a result, computational entities sharing certain levels of memory are aggregated together in a single, larger computational entity called a *virtual process*. Inside a virtual process, all the scheduling decisions and execution management infrastructure are shared, alleviating the cost of management, and decreasing the number of thread synchronization primitives required for a consistent and deterministic execution. The runtime integrates the memory hierarchy into the scheduling process, improving the locality of memory accesses and their reuses. We show how to exploit virtual processes in the context of our wave propagation scheme below. Note that work-stealing between virtual processes may still be performed in order to dynamically correct load imbalance, while maintaining locality.

IV. TASK-BASED FORMULATION OF THE ELASTIC WAVE PROPAGATION

Starting from the MPI-like formulation of the elastic wave propagation described in the algorithm provided in Figure 2, we now show how to turn it into a parametrized task-based algorithm and express it with a JDF expression. In order to more finely exploit hierarchical architectures, we also explain how to tune the granularity and match it with virtual processes.

Task-based algorithm: We assume that the Initialization and DomainDecomposition steps have been performed following lines 2 and 3 in Figure 2, respectively. The idea followed here is that the tasks will be applied on the subdomains, and then the frontiers will be exchanged between neighboring subdomains. The ExchangeBoundaryStress step (line 6 in Figure 2) is performed for each pair of neighboring subdomains. For instance, for the mesh and decomposition in the three subdomains depicted in Figure 1, the central (blue) subdomain packs the current value of the stress tensor on its interface, and exchanges it with both its left (green) and right (red) neighbors. Symmetrically, it unpacks the values obtained from the neighbors. Once it has unpacked the values at the interfaces with all neighbors (two in this case), the value of the velocity (ComputeVelocity) can be computed within the domain. This computation will then be followed by a second exchange, where the newly computed border values are exchanged following a similar scheme as depicted above. Using these newly acquired border conditions, the stress tensor can finally be evaluated within each domain (ComputeStress tasks). One iteration of the corresponding DAG is provided in Figure 4, with only triggered arrows. Overall, this execution scheme (computation + communication) is not inherently different compared with more traditional approaches, but the use of dataflow programming and, especially, the P*a*RSEC runtime system, allows for a completely dynamic execution, with no explicit synchronizations. Moreover, we emphasize that the dataflow of the application remains strictly the same for a fixed problem, and is completely independent of the number and type of available computing resources. Thus, from the application developer point of view, the code remains portable on any execution environment.

JDF expression: As discussed in Section III, the JDF expression consists of the expression of the respective tasks and dependencies. In order to describe this DAG in the JDF format, each task has to describe the dependencies with respect to other tasks. Figure 3 shows an example of a JDF description

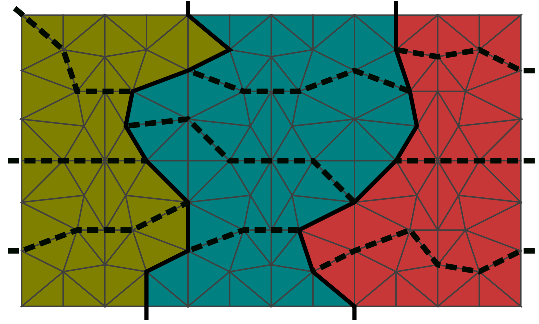


Fig. 5: Multi-leveled mesh partitioning of a surface into triangle elements as described in Section IV

for a (simplified version of) ComputeStress task, according to the DAG in Figure 4. The task is parametrized with indices it and d representing the iteration and domain numbers, respectively. The task works on two data: the velocity field V on the border, which is accessed in read mode (input data), and the stress tensor S within the subdomain, which is accessed in read/write mode (input/output data). The representation of the fact that the velocity field V on the border is obtained from the Unpack_V task(s) is represented by line 4: *READ V* $\leftarrow$ *V Unpack_V(it, d, 1 .. nb_neigh(d))*. The next task which needs the V data is Compute_V at the next timestep (if any); this is represented by line 5: $\rightarrow$ *(it != N) ? V Compute_V(it+1, d)*. The second data of the ComputeStress task is S. It comes from the Compute_V task of the current timestep, and will be overwritten. This is explained by line 6: *RW S* $\leftarrow$ *S Compute_V(it, d)*. This S data is then needed by the Pack_S task(s) of the next timestep (if any), as depicted in line 7: $\rightarrow$ *(it != N) ? S Pack_S(it+1, d, 1 .. nb_neigh(d))*. Altogether, these dependences (lines 4 to 7) allow for representing the data flow for the ComputeStress task. The other tasks are similar, establishing the dataflow of the application, which corresponds to the arrows in the DAG of Figure 4.

Granularity: In order to cope with hierarchical architectures, we actually depart from the MPI style of mesh partition, to a more refined two-level partitioning, with smaller regions. Figure 5 depicts such a possible hierarchical partitioning. Each color represents the original partitioning, as used by the MPI version of the code. We then divide these coarse subdomains (delimited by the plain black lines, for a total of 3 in this illustration) into finer subdomains (delimited by the dotted black lines, for a total of 12 in this illustration). Each coarse subdomain may then be mapped to a virtual process (see Section III). Ideally, we would possess more regions than the number of computing units, allowing more parallelism and finer grain dependencies between neighborhood regions. We will illustrate the impact of this hierarchical scheme on the behavior and performance of our wave propagation application in the next section.

V. EXPERIMENTAL STUDY

We now assess the scalability of the task-based parallelization proposed in Section IV. For reference, we also present the scalability of the MPI code presented in Section II. Note that their sequential performance is (almost) equal; subsequently a better speed-up also corresponds to a shorter execution time.

TABLE I. CCNUMA MEMORY DISTANCES TO BANK ZERO

bank	0	1	2	3	4	5
distance (to 0)	10	13	40	40	40	40

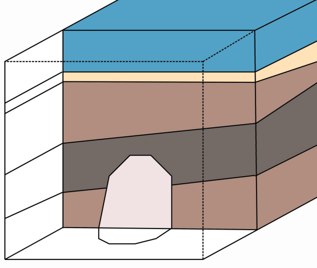


Fig. 6: Test case: $10 \times 10 \times 10\ km^3$ 3D cube.

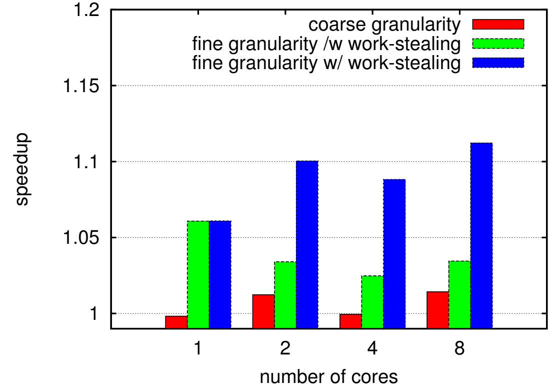


Fig. 7: 8-core Xeon speedup, granularity/work-stealing effects

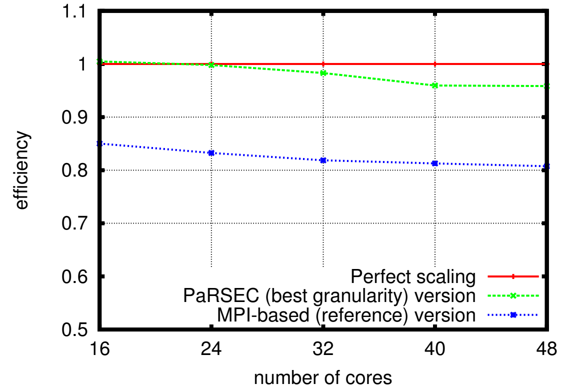


Fig. 8: ccNUMA Xeon efficiency (strong scalability)

A. Experimental setup

Hardware setup: We consider three hardware platforms:

- An Intel Xeon E7-8837 processor composed of 8 CPU cores running at 2.67 GHz. It has 24 MB of L3 cache, each CPU core has its own L1 and L2 caches of size 64 KB and 256 KB, respectively. We will refer to this machine as the *8-core Xeon* chip.
- A ccNUMA node composed of six Intel Xeon E7-8837 CPUs (with the same specifications as the above Xeon platform), supported by six memory banks with an increasing distance cost, as depicted Table I. We will refer to this machine as the *ccNUMA Xeon* node.
- An Intel Xeon Phi 7120P co-processor composed of 61 cores running at 1.238 GHz with four hardware threads each (244 hardware threads total). There is no L3 cache. L1 and L2 caches are exclusive to the CPU core and are of size 256 KB and 512 KB, respectively. We will refer to this machine as the *Xeon Phi* co-processor. Note that the code was ported natively and that results on this platform only involve the accelerator, but not the host processor.

Numerical setup: As represented in Figure 6, we consider a numerical case consisting of a 3D cube with an edge of 10 km and composed of multiple layers of physical materials. The goal of the elastic wave propagation is to compute the velocity and stress tensor evolutions (in time) within this volume. Note that these values do not have to be constant within a physical layer but the type of physical layer will drive the choice for the order of discretization of the elements (which may vary from 1 to 3 in our test case). We consider a test case composed of 800.000 tetrahedra.

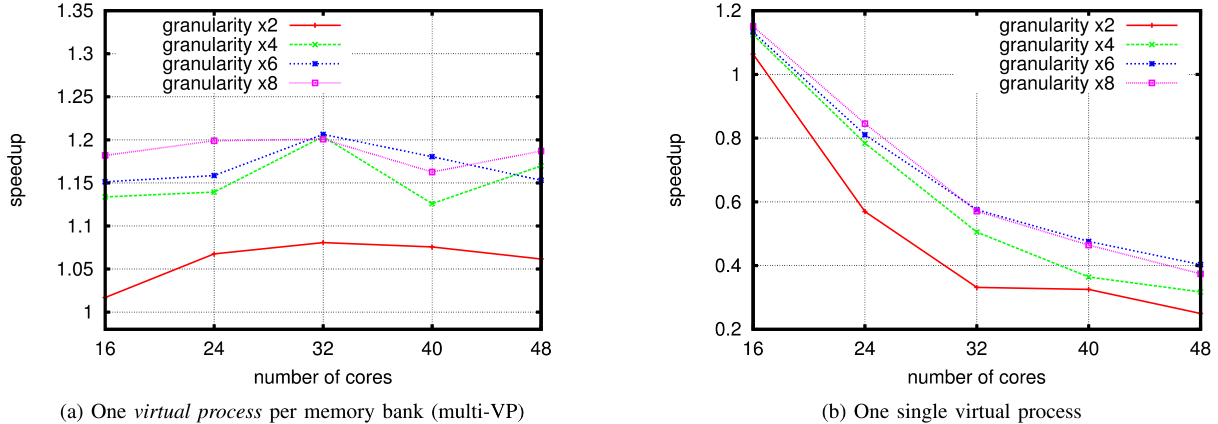
B. Performance on the 8-core Xeon chip

The first experiment aims to demonstrate that the benefits of the task-based approach can also be observed on a simple homogeneous test-case. Figure 7 represents the speed-up ($time_{MPI}/time_{PaRSEC}$) obtained with our task-based algorithm running on top of P*a*RSEC over the original, yet highly optimized, reference MPI-based code. Overall, we observe that even using the same domain decomposition as the MPI version, the P*a*RSEC version delivers slightly faster results. Decreasing the granularity of the domains, and thus increasing the number of domains, led to a small increase in the speedup. The main reason for this is the bigger opportunity to overlap computations, thanks to a finer granularity for the domains, resulting in better memory locality. Coupling finer grain domains with a work-stealing policy allows for improved speedup and gains.

C. Performance on the ccNUMA 48-core Xeon node

The second experiment measures the scalability of the implementation on a ccNUMA (cache coherent Non-Uniform Memory Access) node, with six memory banks and a total of 48 cores. In this configuration, P*a*RSEC integrates the NUMA hierarchy and creates separated groups of cores by enforcing the data locality. This capability, called *virtual process*, was mentioned earlier in Section III. By default, the work-stealing strategies are scoped to the local cores, i.e., to one memory bank, to minimize the bank transfers.

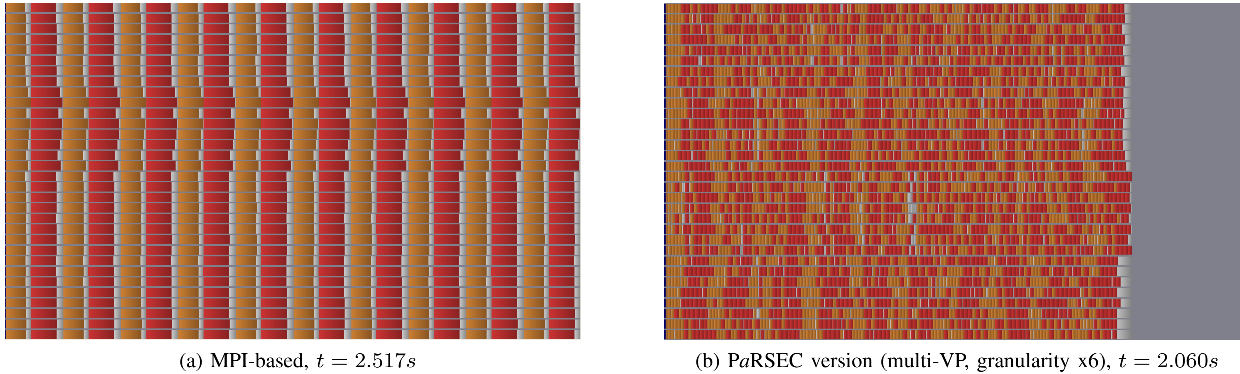
Figure 8 represents the strong scaling efficiency per computing unit obtained by the P*a*RSEC runtime and by the reference MPI-based code, from 16 to 48 cores, which means from 2 to 6 memory banks. The memory bank distances are listed in Table I. Each virtual process is associated with a memory bank to limit the work-stealing only on the local data. In addition, the P*a*RSEC results are obtained with a finer granularity than in the MPI-based code. We use the efficiency formula $time_{1core}/(\#cores * time_{\#cores})$ (the time



(a) One *virtual process* per memory bank (multi-VP)

(b) One single virtual process

Fig. 9: Impact of the memory hierarchy on the strong scaling



(a) MPI-based, $t = 2.517s$

(b) P*a*RSEC version (multi-VP, granularity x6), $t = 2.060s$

Fig. 10: Execution traces, on 32 cores for 10 timesteps, represented with the same scale. Each line represents a core activity, colors (red and orange) refer to computational tasks, gray sections mix idle time and communication (non-blocking). Dark gray on the right highlights the gain of P*a*RSEC over the reference MPI version.

on one core being slightly different between the MPI-version and the P*a*RSEC version when granularity is finer). The MPI version's efficiency is decreasing while increasing the number of computational resources, despite the smart mesh partitioning. Additional tests have highlighted the lack of load-balance between the MPI processes as one of the major causes of this loss of efficiency. On the other side, P*a*RSEC almost follows the perfect scaling thanks to the increased number of subdomains, exposing more intrinsic parallelism.

The previous experiments demand a more clear relationship of the impact of the domain decomposition on the overall performance. Figure 9 depicts the speedup of the P*a*RSEC version (higher is better) when the granularity of the domain decomposition is altered. The multiplier indicates the number of subdomains each original domain generates. Thus, a granularity of "8x" indicates that each original domain is subdivided into 8 subdomains. On the left, in Figure 9a, P*a*RSEC uses the virtual process's capability by allowing the work-stealing only on the local memory bank. The performance is constant according to the strong scalability. Moving away from the case "1 domain per computing unit" allows for a significant increase in performance, as the load balance of the algorithm is improved. On the right, in Figure 9b, the work-stealing is allowed everywhere on the memory banks. Due to distance costs, see Table I, this configuration is negative and implies no acceleration, and is even longer.

Figure 10 illustrates the dynamic scheduling of P*a*RSEC. The execution traces represent the core activity during the given time. This highlights the idle time and so the potential gain of an application. On the left, Figure 10a depicts the MPI-based version. The non-blocking communications correspond to waiting time and fast copies. This is represented in gray. It is recognizable that the application is load imbalanced, as some processes present longer computational tasks. On the right, Figure 10b shows a P*a*RSEC configuration with fine granularity and work-stealing through *virtual processes* bound on memory banks. The dark gray part, on the right, represents the gain over the MPI-based version.

D. *Performance on the Xeon Phi co-processor*

The following experiment focuses on new hardware—the Xeon Phi co-processor. On our particular model, one core is dedicated to card administration, see [32]. Thus, we limited our experiments to 60 cores, which corresponds to 240 hardware threads in hyper-threading mode. Unlike contemporary accelerators and GPUs, the Xeon Phi is composed of x86 cores and is capable of executing managed x86 code. Therefore,

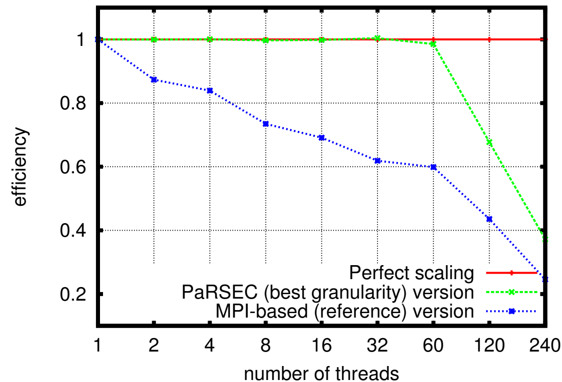


Fig. 11: Xeon Phi efficiency

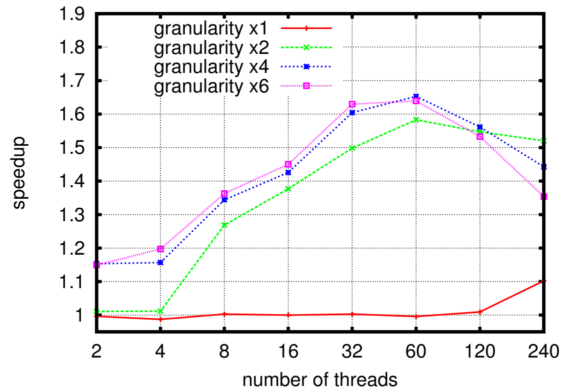


Fig. 12: Xeon Phi speedup with varying granularities

TABLE II. XEON PHI HYPERTHREADING-RELATIVE EFFICIENCY

Number of threads	**MPI**	**P*a*RSEC**
120	0.71	0.91
240	0.80	0.84

our code is executing on this platform without any additional development cost. This fact is true for both versions of the code—the original MPI version and the P*a*RSEC version.

Figure 11 represents the efficiency (higher is better) obtained while using the P*a*RSEC runtime system compared with the reference MPI-based code, with from 1 to 240 hardware threads. From 1 to 60 cores, with one hardware thread per core, neither the runtime nor the tasks themselves take advantage of the hyper-threading capability of the hardware. The P*a*RSEC results are obtained with a finer granularity than in the MPI-based code, with work-stealing. The memory is distributed over different locations on the card and automatically transferred to the processing units.

The MPI-based reference code suffers from declining load-balancing, but scales reasonably well up to 60 cores (with one hardware thread per core). As soon as the hyper-threading is activated, the efficiency drastically declines. On the other hand, the P*a*RSEC version follows the perfect scaling up to the 60 core limit, and then becomes similar to the MPI version, dropping pretty quickly with the use of hyper-threading. Table II presents the hyperthreading-relative efficiency, measured according to the time on one core, with the use of two or four hardware threads, respectively.

The impact of domain decomposition follows a similar trend as the ccNUMA platform, as indicated in Figure 12. A finer domain decomposition is indeed beneficial for improving the load balancing and decreasing the size of the computation domain (improving the cache usage). However, after a certain point, the overhead for data management and copies becomes a limiting factor for the granularity of the decomposition.

Based on these results, it becomes clear that the hyper-threads, at least on the Xeon Phi, should be used differently. Instead of promoting them to full compute resources, they should be used either by the tasks themselves to improve the memory bandwidth (by issuing more pending load/stores) for the algorithm, or by the runtime system to hide the scheduling overheads or the data movements. We expect that in both usage scenarios mentioned here, the usage of the extra units will be more beneficial than in the current study.

VI. CONCLUSION

While new programming paradigms have emerged for writing HPC applications at a higher level of hardware abstraction than MPI, pthread, or OpenMP, these paradigms have mostly been assessed in contexts where the codes could be, for the most part, rewritten from scratch. In this manuscript, we have studied the case of a production-quality MPI industrial code where the parallel scheme got translated into a parametrized task-based expression. We have shown that, by using a task-based programming paradigm, it is possible to achieve high-performance on three different architectures: an 8-core homogeneous Intel Xeon E7-8837 processor, a 48-core ccNUMA node composed of Intel Xeon E7-8837 CPUs, and an Intel Xeon Phi 7120P accelerator. Indeed, while the parallelization of the application is written at a high-level of abstraction, the resulting task graph is processed by a modern runtime system capable of efficiently exploiting the low-level details of the underneath hardware architecture.

The key to achieving high performance was to design a task-based algorithm with a tunable granularity. Compared to the initial MPI code, it becomes clear early on that there was a need to rely on an increased number of domains, enabling a better pipelining of the task, and providing the runtime with a more balanced application. We have indeed shown that the optimum trade-off corresponds to a case where the number of domains is larger than the number of available computational units. Furthermore, the work-stealing capability of the P*a*RSEC runtime system allowed us to achieve a better load balancing than the original MPI code, on which this capability is not present and cannot be implemented due to the code complexity.

Although very promising, our current results are still preliminary, and much work remains to be done to improve the performance portability. In the near term, we plan to address two shortcomings of the current study: dive deeper into the hyper-threading internals of the Xeon Phi, in order to continue our understanding of the benefits and limitations of the Xeon Phi hyper-threading support; and concurrently exploit the Xeon processor and the Xeon Phi co-processor. In this context, the runtime system will automatically transfer data between

host and device memories based on the most urgent data dependences. Additionally, we plan to continue our current push toward distributed architectures, and tackle distributed memory machines, and especially clusters of hybrid multicore nodes. Moreover, we hope that this study will encourage other groups to take the opportunity of turning their applications (or at least their computational-intensive sections) into task-based codes, toward a shift to more efficient and portable codes.

ACKNOWLEDGMENT

The authors acknowledge the support by the INRIA-TOTAL strategic action DIP (http://dip.inria.fr). This research used software resources supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-SC0010682.

REFERENCES

[1] OpenMP Architecture Review Board. OpenMP application program interface version 3.1, 2011.

[2] http://www.top500.org. *Top 500 Supercomputer Sites*, 2014.

[3] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. Parallel tiled QR factorization for multicore architectures. *Concurrency and Computation: Practice and Experience*, 20(13):1573–1590, 2008.

[4] G. Quintana-Ortí, E. S. Quintana-Ortí, E. Chan, F. G. Van Zee, and R. A. van de Geijn. Scheduling of QR factorization algorithms on SMP and multi-core architectures. In *Proceedings of PDP'08*, 2008.

[5] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Samuel Thibault, and Stanimire Tomov. Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In Wen mei W. Hwu, editor, *GPU Computing Gems*, volume 2. Morgan Kaufmann, September 2010.

[6] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Mathieu Faverge, Julien Langou, Hatem Ltaief, and Stanimire Tomov. LU factorization for accelerator-based systems. In Howard Jay Siegel and Amr El-Kadi, editors, *The 9th IEEE/ACS International Conference on Computer Systems and Applications, AICCSA*, pages 217–224. IEEE, 2011.

[7] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Mathieu Faverge, Hatem Ltaief, Samuel Thibault, and Stanimire Tomov. QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators. In *IPDPS*, pages 932–943. IEEE, 2011.

[8] Gregorio Quintana-Ortí, Francisco D. Igual, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Solving dense linear systems on platforms with multiple hardware accelerators. *ACM SIGPLAN Notices*, 44(4):121–130, April 2009.

[9] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Hérault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, Piotr Luszczek, Asim YarKhan, and Jack Dongarra. Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA. In *IPDPS Workshops*, pages 1432–1441. IEEE, 2011.

[10] Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180:012037, July 2009.

[11] Field G. Van Zee, Ernie Chan, Robert A. van de Geijn, Enrique S. Quintana-Orti, and Gregorio Quintana-Orti. The `libflame` Library for Dense Matrix Computations. *Computing in Science and Engineering*, 11(6):56–63, November/December 2009.

[12] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Herault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, Piotr Luszczek, Asim YarKhan, and Jack Dongarra. Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA. In *12th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC'11)*, 2011.

[13] Xavier Lacoste, Mathieu Faverge, Pierre Ramet, Samuel Thibault, and George Bosilca. Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes. In *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing Workshops and Phd Forum (IPDPSW'14), HCW 2014*, Phoenix, United-States, 2014.

[14] Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, and Florent Lopez. Multifrontal QR factorization for multicore architectures over runtime systems. In Felix Wolf, Bernd Mohr, and Dieter Mey, editors, *Euro-Par 2013 Parallel Processing*, volume 8097 of *Lecture Notes in Computer Science*, pages 521–532. Springer Berlin Heidelberg, 2013.

[15] Emmanuel Agullo, Luc Giraud, Abdou Guermouche, Stojce Nakov, and Jean Roman. Task-based Conjugate-Gradient for multi-GPUs platforms. Rapport de recherche RR-8192, INRIA, 2012.

[16] Hatem Ltaief and Rio Yokota. Data-driven execution of fast multipole methods. *CoRR*, arXiv:1203.0889, 2012. http://arxiv.org/abs/1203.0889.

[17] Emmanuel Agullo, Bérenger Bramas, Olivier Coulaud, Eric Darve, Matthias Messner, and Toru Takahashi. Task-Based FMM for Multicore Architectures. *SIAM Journal Scientific Computing*, 36(1):66–93, 2014.

[18] B. Lize, G. Sylvand, E. Agullo, and S. Thibault. A task-based H-matrix solver for acoustic and electromagnetic problems on multicore architectures. In *SciCADE, the International Conference on Scientific Computation and Differential Equations*, Valladolid, Spain, 2013.

[19] R. Kriemann. *H*-LU Factorization on Many-Core Systems. Preprint 5, Max-Planck-Institut für Mathematik in den Naturwissenschaften Leipzig, 2014.

[20] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* Morgan Kaufmann, 2001.

[21] Michel Cosnard, Emmanuel Jeannot, Tao Yang. Symbolic Partitionning and Scheduling of Parameterized Task Graphs. *IEEE International Conference on Parallel And Distributed Systems (ICPADS'98)*, 1998.

[22] A. YarKhan, J. Kurzak, and J. Dongarra. QUARK users' guide: QUeueing And Runtime for Kernels. Technical Report ICL-UT-11-02, Innovative Computing Laboratory, University of Tennessee, April 2011. http://icl.cs.utk.edu/projectsfiles/plasma/pubs/56-quark_users_guide.pdf.

[23] A. Duran, J. M. Perez, R. M. Ayguadé, E. amd Badia, and J. Labarta. Extending the OpenMP tasking model to allow dependent tasks. In *OpenMP in a New Era of Parallelism, 4th International Workshop, IWOMP 2008*, West Lafayette, IN, May 12-14 2008. Lecture Notes in Computer Science 5004:111-122.

[24] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, February 2011.

[25] E. Chan, E. S. Quintana-Orti, G. Gregorio Quintana-Orti, and R. van de Geijn. Supermatrix Out-of-Order Scheduling of Matrix Operations for SMP and Multi-Core Architectures. In *Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures SPAA'07*, pages 116–125, June 2007.

[26] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J.J. Dongarra. PaRSEC: Exploiting Heterogeneity to Enhance Scalability. *Computing in Science Engineering*, 15(6):36–45, Nov 2013.

[27] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Sağnak Taşirlar. Concurrent collections. *Sci. Program.*, 18(3-4):203–217, August 2010.

[28] Delcourte, S.; Fezoui, L. & Glinsky-Olivier, N. A high-order discontinuous Galerkin method for the seismic wave propagation. *ESAIM: Proceedings*, **27**:70-89, 2009

[29] J. Virieux. P-SV wave propagation in heterogeneous media: velocity-stress finite-difference method. *Geophysics*, 51:889–901, 1986.

[30] Bernacki, M. and Lanteri, S. and Piperno, S. Time-domain parallel simulation of heterogeneous wave propagation on unstructured grids using explicit, nondiffusive, discontinuous Galerkin methods. *J. Comput. Acoust.*, 14(1):57–81, 2006.

[31] C. Baldassari. Modélisation et simulation numérique pour la migration terrestre par équation d'ondes. *PhD thesis, Universit de Pau et des Pays de l'Adour*, 2009.

[32] J. Jeffers, J. Reinders. Intel Xeon Phi Coprocessor High-Performance Programming *Elsevier*, 2013.