Experimenting task-based runtimes on a legacy Computational Fluid Dynamics code with unstructured meshes

Emmanuel Jeannot[a], Yvan Fournier[b], Benjamin Lorendeau[a,b,*]

[a]Inria, LaBRI, Univ. Bordeaux, CNRS, Bordeaux-INP, France
[b]MFEE, EDF R&D, 6 Quai Watier, 78400 Chatou

Abstract

Advances in high performance computing hardware systems lead to higher levels of parallelism and optimizations in scientific applications and more specifically in computational fluid dynamics codes. To reduce the level of complexity that such architectures bring while attaining an acceptable amount of the parallelism offered by modern clusters, the task-based approach has gained a lot of popularity recently as it is expected to deliver portability and performance with a relatively simple programming model.

In this paper, we review and present the process of adapting part of *Code_Saturne*, our legacy code at EDF R&D into a task-based form using the PARSEC (Parallel Runtime Scheduling and Execution Control) framework.

We show first the adaptation of our prime algorithm to a simpler form to remove part of the complexity of our code and then present its task-based implementation. We compare performance of various forms of our code and discuss the perks of task-based runtimes in terms of scalability, ease of incremental deployment in a legacy CFD code, and maintainability.

Keywords: *Code_Saturne*, PaRSEC, runtime systems, tasks, unstructured meshes

1. Introduction

Large applications for parallel computers and more specifically unstructured Computational Fluid Dynamics codes are often based on distributed parallelism using runtime systems like the Message Passing Interface (MPI). This is the case of *Code_Saturne* [1], a CFD software using unstructured meshes. In the past decade this model had to be extended to a more refined parallelism with the arrival of NUMA architectures. Therefore most distributed scientific codes try to harness performance through MPI + X solutions such as MPI + OpenMP in order to improve shared memory performance as the number of cores per node increases. While OpenMP promise a method which offers interesting intranode performance using small code modifications, it often fails to deliver significant (or any) performance improvements on *Code_Saturne* and other scientific codes, though it does reduce the memory consumption per thread. When using a simple "loop-local" OpenMP model, as we increase the number of threads per MPI rank, performance drops rapidly, since many secondary loops are not threaded; and avoiding data races often requires specific renumbering strategies, which may not be easily adapted everywhere with a reasonable programming effort. These diminishing returns tend to limit the efforts which are worthwhile to spend in addition to the base MPI model.

Since the recent hybridization of clusters enforce us to adapt our code once again, the importance of finding a solution to provide *Code_Saturne* and CFD codes with satisfactory performance while being portable, maintainable and without needing optimizations that would lead to significantly more complex code is critical. As the post-petascale era has long been foreseen, runtime systems developers have been investigating other parallelism paradigms. Notably, the task-based approach has gained a lot of popularity recently as it is expected to deliver portability and performance with a relatively simple programming model.

One interesting aspect of the task-based approach is its ability to let developers visualize their problems as a set of smaller problems with inputs and outputs on which another small problem may depend. This data dependency, once clearly expressed, allows a task-based runtime system to build a graph of the algorithm set in a form that respects each problem dependencies, and then schedule this workflow on the available computing units. This way, it alleviates us from the burden of defining ourselves these graphs while providing theoretically better performance as work is scheduled at the earliest. Moreover, it emphasizes on choosing the proper granularity for each task, which is getting increasingly important as the number of computing units increases. However, this approach is best suitable for problems that can be expressed as a tiled algorithm such as the QR factorization [2] which has been broadly implemented using such programming model.

We set apart two task-based approaches as being mostly explored, which are sequential and parameterized task-based programming. The sequential model builds task dependency from the order of submission while the parameterized model uses algebraic data dependencies between task to schedule them readily. Several runtime systems have been proposed in both models: OMPSs[3], StarPU[4] and SuperMatrix[5] for the former model mostly and CnC[6] and PaRSEC[7] for the latter. As the set of problems we solve in *Code_Saturne* is broad, we chose to investigate in multiple solutions so as to determine which type of runtime is best to suit our needs. To this extent, we chose to compare a StarPU and PaRSEC implementation of our code on different areas.

As many teams are already dedicating their work to linear algebra solvers[8, 9, 10], we decided to focus on another part of the puzzle, namely our gradient reconstruction computation. As a significant portion of our main current numerical schemes, it has a high impact over the performance of our code and an intermediate computational intensity. As such, we propose in this article a review of different implementations of our gradient computation towards the implementation of a task-based approach through the use of task-based HPC runtime systems, and more specifically the PaRSEC and StarPU frameworks.

We show first the adaptation of our gradient reconstruction to a simpler form to remove part of the complexity of our code and then present its task-based implementation in both runtimes. We then compare performance of various forms of our code and discuss the perks of task-based runtimes in terms of scalability, ease of incremental deployment in a legacy CFD code, and maintainability.

2. *Code_Saturne* gradient reconstruction

Code_Saturne. Our code has been under development since 1997 by EDF R&D. The software is based on a co-located Finite Volume Method (FVM) that accepts three-dimensional meshes built with any type of cell (tetrahedral, hexahedral, prismatic, pyramidal, polyhedral) and with any type of grid structure (unstructured, block structured, hybrid). It is able to handle either incompressible or compressible flows with or without heat transfer and turbulence. *Code_Saturne* has been open-source (GPL) and available to any user since 2007. Parallel code coupling capabilities are handled in a specific subset of the code, the Parallel Location and Exchange (PLE) library (under Lesser General Public License (LGPL)).

Gradient reconstruction. The gradient reconstruction routine of *Code_Saturne* is detailed in [11] and is used in every time steps of our solvers so as to update physical variables on each cell such as pressure or velocity. It has low arithmetic intensity (about $\frac{1}{8}$ flop/bytes) and its threading race condition potential make it a good candidate for the experimentation of task-based runtimes as it is representative both in terms of common issues we would encounter

if we were to massively port *Code_Saturne* in a task-based form and in terms of overall performance representativeness (with the highest performance cost after our solvers). Moreover, its form cannot be made generic in a way that we could use third-party software to compute them. This allows us to effectively test task-based runtimes regarding their adequacy with an FVM-based with unstructured meshes CFD code.

$$\underbrace{\begin{bmatrix} C_{i,x\,x} & C_{i,x\,y} & C_{i,x\,z} \\ C_{i,y\,x} & C_{i,y\,y} & C_{i,y\,z} \\ C_{i,z\,x} & C_{i,z\,y} & C_{i,z\,z} \end{bmatrix}}_{\underline{\underline{C}}_i} \underbrace{\begin{bmatrix} G_{c,i,x} \\ G_{c,i,y} \\ G_{c,i,z} \end{bmatrix}}_{\underline{G}_{c,i}} = \underbrace{\begin{bmatrix} T_{i,x} \\ T_{i,y} \\ T_{i,z} \end{bmatrix}}_{\underline{T}_i} (1)$$

As seen in (1) it consists in the computation of the terms $\underline{T}_i$ and $\underline{\underline{C}}_i$ for each cell in order to compute our new gradients $\underline{G}_{c,i}$. The computation of $\underline{T}_i$ for a cell i requires data from all of its neighbors. However, the contribution to both cells of a given face has the same absolute value. Therefore the computation of $\underline{T}_i$ is performed face-wise instead of cell-wise. This implies a strong constraint for shared memory parallelism as several faces may contribute to the same $\underline{T}_i$ value (c.f. Fig. 1).

face contribution to $\underline{T}_i$ and $\underline{T}_j$ (rhsv)

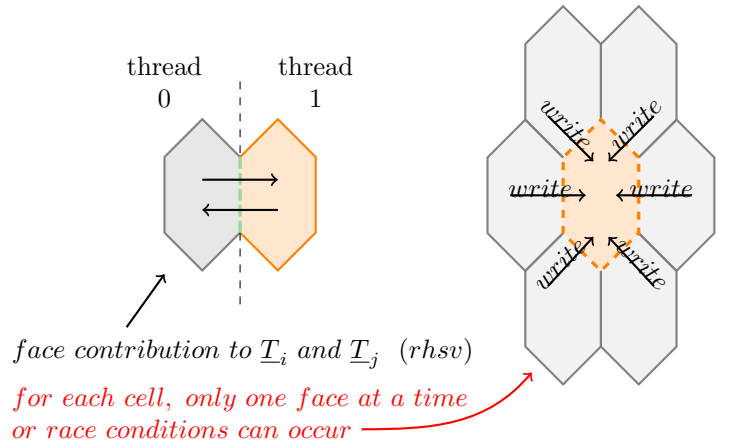*for each cell, only one face at a time or race conditions can occur*

Figure 1: Face wise contribution to cells for the computation of the rhsv term. Only one face of a given cell can be processed at a time, requiring developers to use coloring techniques for threaded parallelism which is expected to slightly increase frequency of cache misses.

To answer this problem when using thread based parallelism, *Code_Saturne* introduced in [12] the creation of groups of faces subsets where within a given group, any two faces belonging to two different subsets do not share cells. This way, each group of faces can be processed by separate threads using an OpenMP approach.

Because this approach leads to a lack of load-balancing with decreasing subsets size, we first decided to adapt our prime algorithm to a full cell-wise computation for all terms of our gradient reconstruction (c.f. Figure 5). A complete review of this adaptation from a face-wise to a cell-wise reconstruction is out of the scope of this article. However, this step can be considered as a first step

towards accelerators as it increases the number of operations to be performed as well as the arithmetic intensity of the algorithm. Performance is, however, similar on current platforms with no accelerators. Nonetheless by redesigning our algorithm, we pave the way to an easier transition from the Bulk Synchronous Parallel model to the Task-Based paradigm. Indeed as we ought to proceed the reconstruction of our gradients by blocks, each block representing a task, the data dependency between each block, as they share faces, makes the task definition a cumbersome process. The ideal solution for such problem would be an exclusion rule on each task definition to specify for a given task a set of task which cannot be executed at the same time. To our knowledge such capability does not exist as a mature feature in any task-based runtime systems.

Once the algorithm is adapted to avoid the race condition the building of the term $\underline{T}_i$ involves, the gradient reconstruction consists in merely the exchange of halo data and a set of identical tasks computing the reconstruction (c.f. Figure 2). Moreover, the exchange of halos is not needed for the computation of all cell indexes of (1) thus blocks of cells that do not need any halos could be computed earlier, allowing for the hiding or communication costs by overlapping them with available computation tasks. It is important to note the impacts brought by our cell numbering strategy (c.f. Figure 3) on our ability to determine which part of the mesh can be processed with or without neighboring data.
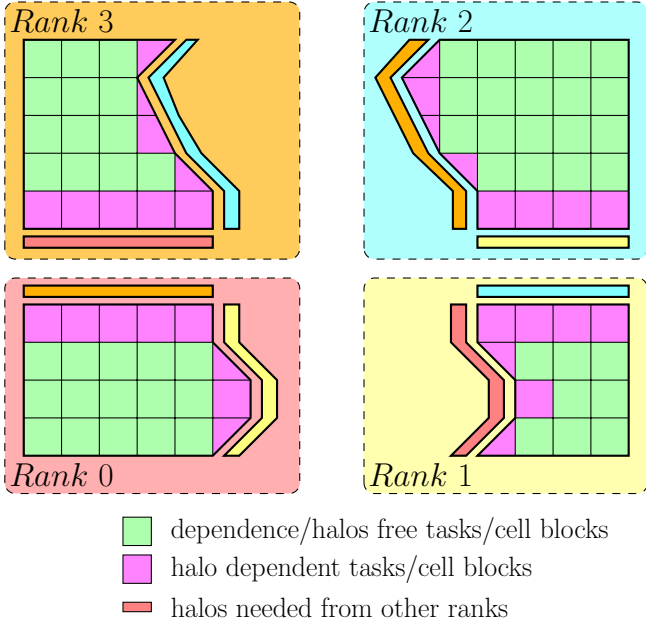

Figure 2: Gradient reconstructions tasks for each rank from a mesh perspective. Each task received a specific block of cells that may depend on halo reception. This dependency must be computed as cells and halo cells are numbered and indexed.
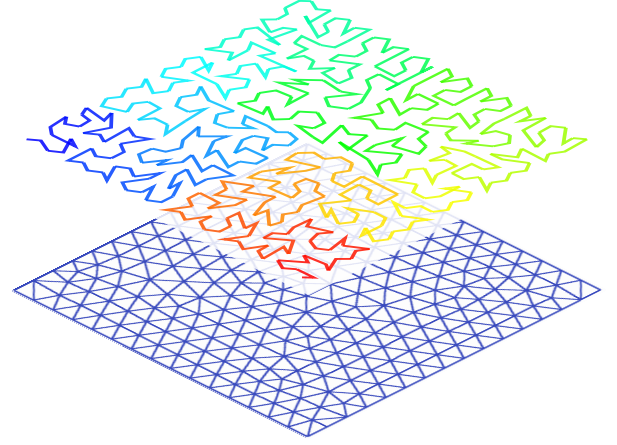

Figure 3: A regular mesh used in *Code_Saturne* with its Hilbert SFC cell numbering.

3. Task-based runtimes

Task based runtime systems are getting increasingly attractive as they offer a way to developers to be completely relieved of the complexity of exploiting current and future architectures. This is emphasized by the recent introduction of heterogeneous architectures in HPC clusters. In a task-based programming model computations are encapsulated in tasks that have no side effect and that communicate through messages. A task becomes ready when it has received, from all its predecessors, the requested data. The gain of parallelism comes from the fact that there can be many tasks per process (MPI ranks) and that all the ready ones are independent thus reducing synchronizations. Another advantage is that the same task (code) can be instantiated with different inputs reducing the development effort. By using a task model, new runtime systems can perform scheduling decisions based on several parameters such as performance models, knowledge of task characteristics and available hardware resources to find the best match at any given time and for each specific task.

Task expression. Both runtimes diverge in terms of task expression. The PaRSEC team chose to privilege an expression that removes users from most technicalities of other programming languages by providing their own, which perfectly fits algorithms which are expressible in a tile form (c.f. a 2D sweep example taken from [13], Figure 4). We expect any algorithm or code which do not fit in this reasoning such as *Code_Saturne* to experience trouble. In our case, describing tasks using PaRSEC abstraction proves to be challenging as its elegance must be balanced with the exploitation of our existing data indexes. Tasks inputs and outputs are defined using the same techniques, asking the developers to use a previously defined access function which translates tiled access to data to the correct in memory position. On the contrary, StarPU only differs slightly

3

```
T(a, b)
a = 0 .. ncx
b = 0 .. ncy

RW X <- (a != 0)   ? X T(a-1, b) : psi_x(b)
     -> (a != ncx) ? X T(a+1, b) : psi_x(b)
RW Y <- (b != 0)   ? Y T(a, b-1) : psi_y(a)
     -> (b != ncy) ? Y T(a, b+1) : psi_y(a)

BODY
{ computePhi ( X, Y, ... ); }
END
```

Figure 4: PaRSEC sample of a 2D Cartesian sweep. For a given tile/task, the inputs/outputs relation between tasks and geometry of the mesh is transparent.

```
Cell_Wise_Gradient_reconstruction(var)

Exchange halos (MPI_Sends and Recvs)
MPI_Barrier()


for i in cells
  compute C_i
  compute T_i
  compute G_{c,i} = C_i^{-1} · T_i // var's gradient for cell i
end for
```

Figure 5: Simplified algorithm of cell-wise gradient reconstruction

from classical C functions and tasks are merely a set of parameters to define on which a function will be called. Both StarPU and PaRSEC ask users to use partitioning of the workload through them for an optimal utilization. Indeed, expressing tasks and dependencies without fully using the data description through those runtimes disables part of the capabilities in terms of data transfers and data dependencies between tasks. Finally, the complexity hiding performed by PaRSEC when it comes to task expression has to be tempered by the beforehand data description that needs to be written for each data on which task will be working on.

Data dependencies. Both runtimes use data exchange from task to task as a mean of describing task dependencies. However the expression of those data dependencies varies between the two runtimes. In PaRSEC, dependencies are written using an algebraic form: a function of the parameter of a task describes the dependencies between different tasks. For instance in Fig. 4, we see that task T(a,b) receives X from T($a-1$,b) (unless $a = 0$) and send it to T($a+1$,b) (unless $a = $ ncx). Such expression has two strong advantages. First, it can be generated automatically from a sequential code as long as the control is static and the loop indices are affines (this is the case for the dense Cholesky factorization and the QR factorization) [14], then it is possible to build a symbolic schedule and execute it in a distributed way without having to process the meta-information during the execution [15, 16]. Hence, his approach relieves the runtime from the burden of handling a full task graph which can induce a high memory cost. At the opposite, StarPU builds its task graph during execution, as each task is submitted to the task graph separately, increasing its size step by step. Despite the potential memory overhead this solution brings, it also allows more flexibility and performance tweaking through scheduling, memory management and load balancing. This approach is also more natural to write as it is closer to the way sequential programs are written and does not require to express the symbolic formula that describes dependencies between tasks.

Partitioning. The first version of the PaRSEC and StarPU runtime systems were designed for shared memory systems. Indeed this did not require to parallelize the runtime system itself and it is much easier to handle data movement in that case as only its readiness has to be expressed. When both systems went to distributed memory environment the problem of partitioning the data and to efficiently schedule the tasks became extremely important. In PaRSEC the data distribution is explicit and given by the task description while in StarPU the data distribution is implicitly given by the code of the program. StarPU decides according to certain heuristics whose rank is best suited to own each data handles. This benefits cases where we need to maximize data re-use and minimize data movements.

In the case of unstructured grids such as it is with *Code_Saturne*, we do not expect to be able to use *Code_Saturne* existing solutions while benefiting from StarPU or PaRSEC partitioning methods. Indeed our code starts with a parallel partitioning of the whole mesh, thus making the knowledge of data distribution available only after it is already partitioned. Implementing a static distribution such as PaRSEC uses would be a humongous process as it would require us to redefine all our data layers through PaRSEC before the adaptation of any computing part of our code. Similarly making this transition with StarPU is an unnecessary problem yet simpler as we do not need a static knowledge of data ownership. In any case, given the code's size, only an incremental approach for improvement of the code's parallel performance may be considered viable. We state that – integrating a task-based runtime as a incremental process – using the shipped partitioning solution might not be advisable nor ever desirable. However, with such a conclusion, it is important to note how well each runtime performs in this scheme as it highly impacts how we see data movements between tasks. Indeed to answer this issue, data movements using PaRSEC are made throughout send and receive tasks while our data exchanges using StarPU simply mirror current pure MPI halo exchanges.

4

```
RECV(me, n)

me      = 0 .. WORLD−1
n       = 0 .. WORLD−1
n_idx   = %{ return find_rank_index(n);    %}
c       = %{ return halo_size_for_n(n_idx) %}
b       = %{ return cs_glob_n_blocks −1;   %}

: cs_halos(me, n, ..)

READ HALO <− (n == me) ? NULL
<− (n != me && !c) ? NULL
<− (n != me &&  c) ? HALO SEND(me, n)
−> (n != me &&  c) ? cs_halos(me, n, ..)
−> (n != me &&  c) ? HALO BLOCK_RECV_CTL(..)
```

Figure 6: PaRSEC sample of halos exchange (simplified)

4. Task-based gradient reconstruction

We identify several important points to focus on for the use of a task-based runtime system on *Code_Saturne*. As we expect the transitioning from our BSP model to the task parallelism model to be an incremental procedure, we investigate how well each runtime integrates in our legacy code. Their ability to handle unstructured meshes as well as their task dependency building mechanism are key features for *Code_Saturne*.

Integrating in a legacy code. *Code_Saturne* is a well-parallelized code using mostly MPI and as such, relies on several partitioning techniques to ensure a good load-balancing. This part can be handled through in-house space-filling curve algorithms (Morton or Hilbert) or through SCOTCH or Metis and their parallel versions. Using the task paradigm through PaRSEC and StarPU implies the handling of data movements from task to task in a well-determined form. StarPU uses data handle structures that are bound to each task and registered to StarPU. Those data handle structures, registered as input need and outputs for a given task are then used by StarPU to compute each task dependencies and the resulting Directed Acyclic task Graph. ParSEC has a similar method through its Parameterized Task Graph expression. Our gradient reconstruction does not fit in both solutions as our data structures are already spread across multiple MPI processes making data ownership expression a complex process. Moreover, our needs in data movements for the gradient reconstruction are strictly bound to the exchange of halos between MPI ranks prior to the computation and cannot be seen as a data dependency between one task to another.

Data exchanges and task dependencies. To perform this halo exchange we had to implement two different reasonings. In PaRSEC, our halos exchanges are seen as nearly empty tasks mimicking MPI sends and receives which are just describing where data are taken and where we store them.

```
for (rank = 0; rank < n_c_domains; rank++)
{
    start = start(rank);
    length = length(rank);

    if (need_to_send_to(rank))
    {
        struct cs_halo_handle *h = &halo_handle[n];

        h−>other_rank = halo−>c_domain_rank[rank];
        int shift = halo−>n_local_elts + start;

        starpu_vector_data_register(&h−>recv_handle,
                        STARPU_MAIN_RAM,
                        var + shift,
                        length,
                        sizeof(double));

        starpu_mpi_irecv_detached(h−>recv_handle,
                        h−>other_rank,
                        TAG,
                        cs_glob_mpi_comm,
                        recv_done,
                        arg);
    }
}
```

Figure 7: StarPU sample of halos exchange (simplified)

Using PaRSEC, data exchanges are performed from task to task, the preceding task transmitting data to its successors and so forth. Data transfers can be inter- or intra-node without any differences from a user point of view. They are described as an input/output view for each, so that based on all parameters defining the application domain of a task, we specify data that comes in, and data which goes out. As we can see on Figure 6 line 1, we have $my\_rank \times n$ RECV tasks, which is equal to $WORLD^2$ since both my_rank and n stretch from 0 to $WORLD − 1$. This is how we express that for each MPI rank, we may exchange to all other ranks. As dependencies and tasks in PaRSEC must be expressed before building our application, we are forced to define a range of tasks and dependencies that is larger than our needs. Indeed we create for each instance of PaRSEC a task set of size $WORLD$ to exchange our data while in reality each instance would need a subset with a significantly smaller size. This might prove to be an issue later as the number of PaRSEC instances increases.

We then have to distribute each $RECV(i, j)$ tasks on the correct MPI rank thanks to the distribution function $cs\_halos(i, j)$ which distributes each $RECV(i, *)$ task to the rank i. At this point, we created more $RECV$ tasks than what we need, since we do not know to whom we need to receive data. We thus need to determine the real need for each $RECV$ task by computing how many elements we need to receive for a given neighbor. This is where the term c is involved. It defines if and how many elements need to receive for the given task (this is used in the dataflow part

as well to specify the exchanging count but was removed due to lack of room). In the second part of Figure 6, we define the dataflow of our task. It defines in which case we received data from the $SEND$ task, and in which case we write and send data to other tasks.

However, performing halo exchanges with StarPU can be quite similar to our pure MPI based version (c.f. Figure 7) making the transition really easier. We can distinguish two StarPU calls, one to register a data handle, which will describe the characteristics of our data and another one wrapping an MPI_Irecv call using this so-called data handle with the possibility of using a callback, here $recv\_done$. This callback is then used to release tasks awaiting the reception of halo data. The release is performed by the submission of an empty task described before our gradient reconstruction so that each halo needing tasks will be declared as depending on this empty task completion. It is important to note that StarPU offers a facility called "starpu_task_insert" to handle the task dependency expression in a transparent way. However, this feature does not address unstructured cases with the same simplicity, as we ought to manage the definition of tasks with a varying amount of data handle structures depending on how many halos each task requires. In our case, the ability to specify by hand each task dependency and each data transfers with StarPU without depending on a higher level of abstraction proved to be a faster and easier solution to implement.

Handling unstructured meshes. The ability to process unstructured meshes implies some specificities slowing *Code Saturne* computations. Indeed, we work with sparse matrices which need to be ordered and indexed. With unstructured meshes, the neighbors of a given cell are not geometrically computable, meaning that defining data dependencies between one cell and another is not possible without maintaining extra information for each cell and will still fall far beyond the simplicity of expressing data dependencies with a code using structured meshes. To handle the building of a task graph which respects our data dependencies, we need to perform some code adaptation. Our mesh must be seen at a higher, coarser level corresponding to our task granularity, in which one block of cells corresponds to a single one, and data dependencies between each block is recomputed to fit this coarser view. We then determine for each block whether it needs data from other runtime instances as well as potential local, block to block dependencies. As our cells are numbered using space filling curves or graph-based techniques, we cannot determine if the neighborhood of a given cell contains halo cells without going through its connectivity, computing and storing for each block its dependencies from other MPI ranks. This is true for both StarPU and PaRSEC even if it is used differently.

In addition, even if our gradient reconstruction does not need to use specific data structures like any sort of Compressed Sparse Row storage, we observe the possibility in StarPU to register such data structures as runtime data handles, which should prove to be very handful in various parts of our code. To our knowledge, PaRSEC does not include such features.

5. Results

The experiments were carried out using Plafrim 2, an 88 node machine with a fat-tree network. Each node contains two Intel Xeon E5-2680 v3 processors (24 cores total, split in 4 NUMA nodes across 2 sockets with 6 cores each c.f Figure 8). Each node can use up to 128Gb of RAM and the Infiniband QDR TrueScale with a bandwidth of 40Gb/s.
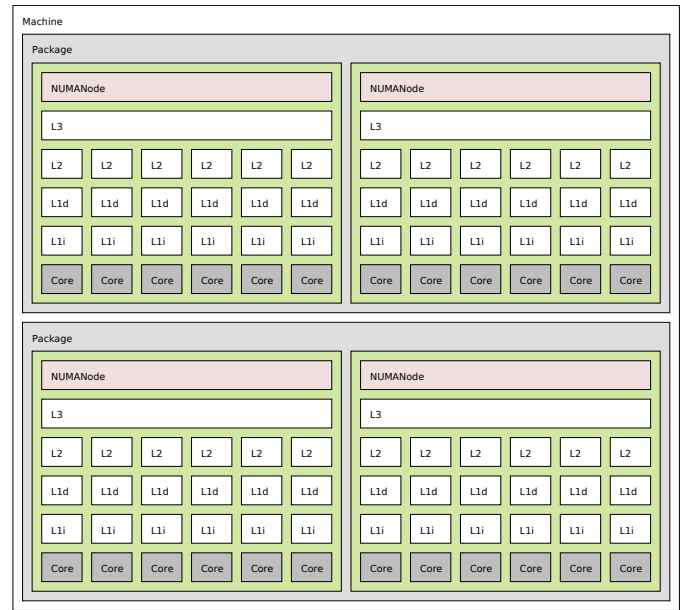

Figure 8: Node description of one Miriel node (lstopo output)

Intranode performances. In figure 9 we evaluate our implementations with PaRSEC and StarPU on a single node using a single MPI process, and thus, a single PaRSEC or StarPU instance. Our test case use a mesh of a total of 262 144 cells, with the total number of cells per thread varying from 131 072 to 10922. Optimal cells per thread-/process for a classical *Code Saturne* run is expected to stay between 60 000 to 30 000. Effective runtime of the gradient reconstruction is then compared with the optimal run case on one node, with 14 MPI processes and 2 OpenMP threads each. We also compare our results with the actual OpenMP version of the gradient reconstruction in the same configuration. We observe a slight performance gain for both runtime compared to the optimal MPI run case. On a single node, our StarPU implementation obtained the best results. It is interesting to note that both implementations already near the best MPI+OpenMP scenario with half the machine. This can be explained both by the

spreading of threads across another socket and potential additional data movements as well as the extreme scaling for this case, as starting from 12 threads up to 24, our ratio of cells per thread drops below 20 000.

cells per threads
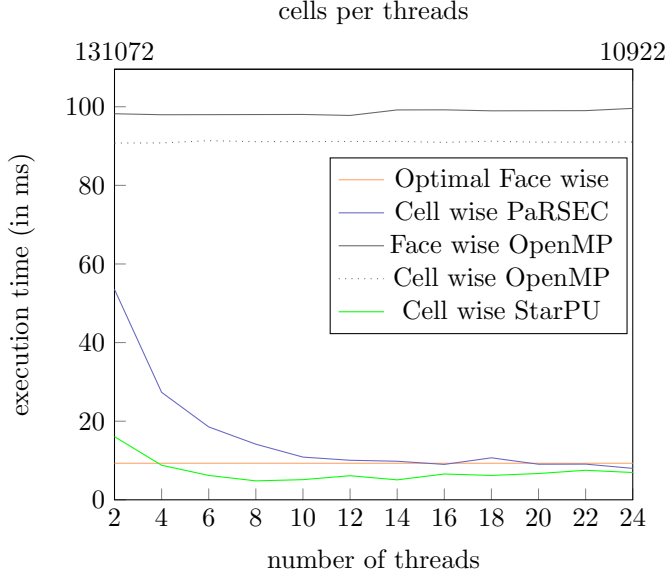
131072   10922

Figure 9: Intranode performance comparison

Internode performance. Our implementations were pushed on several nodes to further analyze their performance behaviors. Current implementations showed encouraging results but also displayed a discrepancy between the simplicity of the runtime as what seems to be the best fit for our implementations and the resulting performance. We identify the relation between each runtime and MPI as a potential issue. Indeed, in its current state our data exchanges are proving to strongly impact internode performance. As the intranode performance of our task-based implementations are faster than the best runcase of *Code_Saturne* and this issue to be a simple setback, we expect further results to be promising.

As we observed the best results with StarPU, we pushed various implementations in order to find the best overall solution between performance and ease of programming with *Code_Saturne*. We first show in Figure 10 that our implementation with StarPU on 2 ranks behaves less satisfyingly than MPI. Indeed we observe a tendency when using both PaRSEC or StarPU to degrade performance on multiple ranks. However we can observe that the overhead decreases from 50% to 10% as we increase the size of the mesh. Several aspects might be taken into account before any conclusions including the chosen task size: in our cases, we found that a task size of 4096 cells to be the best out of a test over 4096 to 16384 cells per task, which falls far under the recommended task granularity (more likely to be around 50000). Such granularity would, however, barely fit for our needs as the recommended granularity for each MPI rank in *Code_Saturne* is between 20000 to
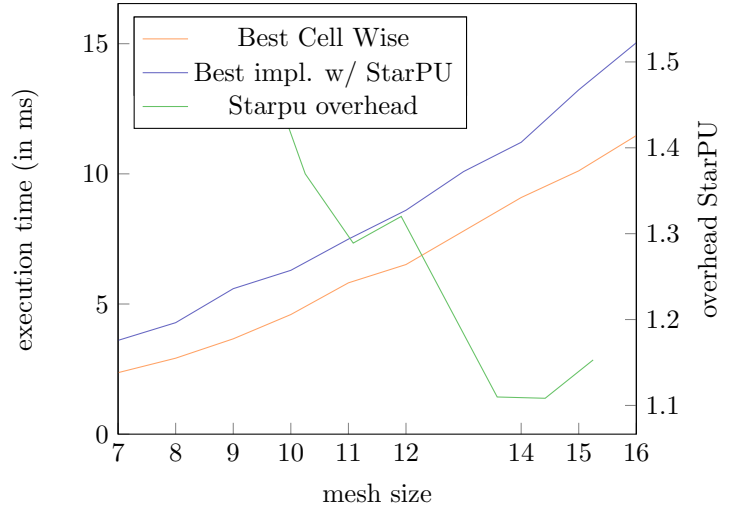

Figure 10: Performance comparison of MPI versus StarPU implementation of gradient reconstruction. We also show the overhead of the StarPU version as the ratio of the extra time needed over the total time of the execution.

80000, and would not allow to provide from the load balancing benefits of using many smaller tasks per core. This limitation can be seen as a consequence of the high arithmetic intensity focus of runtime systems such as StarPU compared to our low arithmetic intensity approach based on unstructured meshes. This observation can furthermore be supported by the Figure 11 which shows that in the same fashion as our MPI+OpenMP code, we tend to benefit from having more than one MPI/StarPU instance on one node. This can be explained by the strong effect of NUMA architectures on unstructured codes such as *Code_Saturne*. Figure 12 shows performance comparison of various gradient reconstruction implementations with StarPU such as using StarPU in a local fashion very similar to OpenMP and handling communications with MPI as well as an implementation with MPI communications as tasks (opposed to handle their completion as callbacks or the use of MPI_Test) and finally the expression of the gradient reconstruction of StarPU with global DAG as we had to do with PaRSEC. Finally Figures 13 and 14 shows a timeline of a gradient reconstruction performed on 2 ranks with StarPU. Overall scheduling of tasks is consistent but we sometimes observed a high variation of the time spent in the task submission area of the code which explains some of the performance instability.

6. Discussion

Using a task-oriented runtime rather than MPI tasks with OpenMP loops should allow for a simpler implementation of multilevel parallelism, with a smaller overhead than pure MPI. On a code already fully distributed with MPI, this approach should provide a relatively easy path forward, and using a task paradigm allows both better handling of parallelism opportunities, including easily accessible fine-grained control of computation and communication overlap, without the implicit synchronization barri-
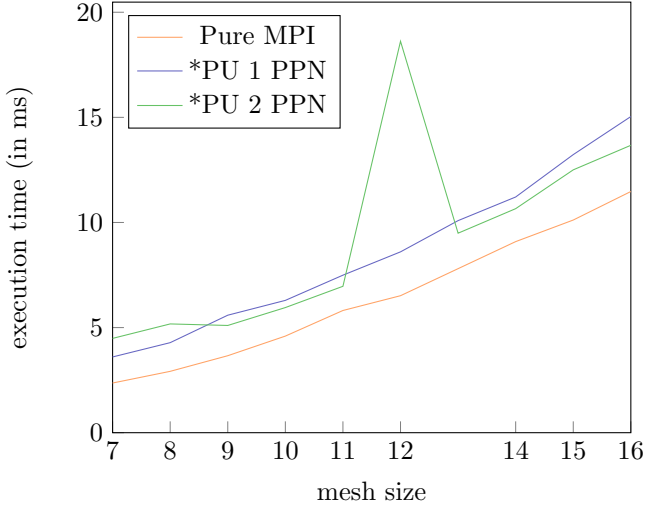
Figure 11: Performance comparison on 2 nodes with a selected task size of 4096 cells per task. Based on our observations of MPI+OpenMP *Code_Saturne* we try to reduce the performance gap we have with StarPU by placing two StarPU instances of one rank (2 process per node (PPN)). We observe a small gain which might be better with 4 PPN. The peak in execution time is explained later by a high sensibility to the operating system which sometimes arises.
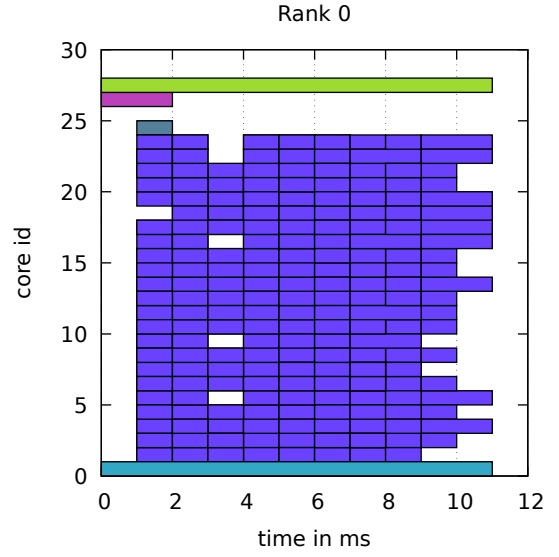
Figure 13: Timeline of one rank out of 2 of an execution of gradient reconstruction with StarPU on 2 ranks. In dark blue are represented the reconstruction task, in purple the time taken by the submission of all the tasks, in green the time from the beginning until the end of the StarPU barrier on all tasks which is similar to the time of the full reconstruction.
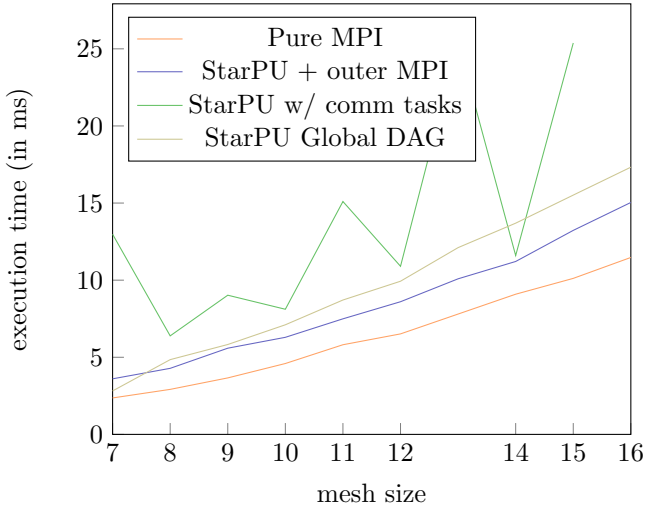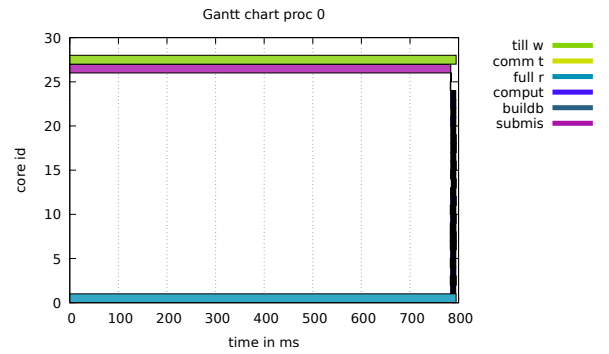

Figure 14: Another timeline with the exact same run case than figure13 but where the submission time took a very large amount of time. This is a strong indicator of the performance standard deviation we observed previously.

Figure 12: Performance of various gradient reconstruction implementations with StarPU compared against MPI+OpenMP.

ers inherent in loop-based OpenMP approaches, and without significant additional coding complexity relative to pure MPI. Also, this approach maps well to the optional offload of some operations to accelerators or other specialized hardware, and evolving from one to one mapping of processor cores to MPI rank based tasks to the distribution of finer-grained tasks across nodes allows for tuning of the task/data mapping relative to partition size so as to find a balance between data locality within a task and runtime latency.

Tested runtime systems are not fully mature and there are still some performance pitfalls, but the approach is promising. Both StarPU and PaRSEC offers interesting code writing features, with PaRSEC focusing on the simplicity of algorithm expression while StarPU offers a transitioning process from a legacy C code to a task-based version in a most readily way. For codes with low computational intensity and frequent communication requirements (latency-bound), improving the latency of task scheduling would improve performance. Initiating communication earlier to better overlap tasks would also be possible in some cases, but the required optimizations would add significant code complexity.

7. Conclusion

Adapting legacy codes to today's and future supercomputing systems is a very important yet difficult endeavor. Indeed, the increase of the number of cores as well as the emergence of accelerators make it more difficult for pure MPI programs to scale and achieve good performance. The MPI+OpenMP programming model is an attempt to solve this problem but, in the case of large code managing complex data structure (such as unstructured meshes in *Code_Saturne*), the performance gain is often hardly visible. In this paper we have explored an alternative approach using on task-based parallelism. This approach is highly appealing as it enables to remove most of the implicit barriers in loop-based OpenMP approaches. Moreover, task-based approaches combined with dynamic runtime systems enable online scheduling and hence are able to more naturally manage accelerators and more generally heterogeneous supercomputers.

In this paper, we have implemented the gradient computation of *Code_Saturne* (a leading CFD software) with two of the leading task-based runtime systems: PaRSEC and StarPU. Both systems have their respective advantages and drawbacks, but it was not the goal to compare them but rather to give a feedback and a review of porting of a strategic part of a legacy code onto such systems. From our experience we can draw several remarks. First, the code modification itself, once it has been debugged and optimized, is not extremely high. The changes are less important than expected and the obtained code is easy to read and maintain (especially for StarPU, whose model is closer to the C language than PaRSEC). Second, the obtained performances are promising. Early results show that substantial performance gains are achieved by both systems compared to the MPI version of the code, though not yet in all cases. Last, we see that these systems still lack some maturity for this class of application. Indeed, if writing a correct code is already a challenge due to the paradigm change and some discrepancies between the documentation and the version of the code, optimizing and debugging performance is still extremely difficult due to the lack of tools to give feedback to the user about performance bottlenecks.

We plan to carry on this work on several directions. First, we need to extend the experiments to larger settings with more nodes in order to see how both versions behave in terms of scalability and performance. We also want to clearly assess the advantages and drawbacks in terms of functional and non-functional features of both systems. As we started with a algorithmically simplified version of our gradient reconstruction, we also aim at performing the same procedure on our current face wise reconstruction. Lastly, if the task-based approach can deliver good performance at large scale, we would consider extending it to the whole code of *Code_Saturne*.

References

[1] F. Archambeau, N. Méchitoua, M. Sakiz, Code saturne: A finite volume code for the computation of turbulent incompressible flows-industrial applications, International Journal on Finite Volumes 1 (1) (2004) http–www.

[2] A. Buttari, J. Langou, J. Kurzak, J. Dongarra, Parallel tiled qr factorization for multicore architectures, Concurrency and Computation: Practice and Experience 20 (13) (2008) 1573–1590.

[3] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, J. Planas, Ompss: a proposal for programming heterogeneous multi-core architectures, Parallel Processing Letters 21 (02) (2011) 173–193.

[4] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier, Starpu: a unified platform for task scheduling on heterogeneous multicore architectures, Concurrency and Computation: Practice and Experience 23 (2) (2011) 187–198.

[5] E. Chan, E. S. Quintana-Orti, G. Quintana-Orti, R. Van De Geijn, Supermatrix out-of-order scheduling of matrix operations for smp and multi-core architectures, in: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures, ACM, 2007, pp. 116–125.

[6] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, et al., Concurrent collections, Scientific Programming 18 (3-4) (2010) 203–217.

[7] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, J. Dongarra, Dague: A generic distributed {DAG} engine for high performance computing, Parallel Computing 38 (12) (2012) 37 – 51, extensions for Next-Generation Parallel Programming Models. doi:http://dx.doi.org/10.1016/j.parco.2011.10.003.
URL www.sciencedirect.com/science/article/pii/S0167819111001347

[8] G. Quintana-Ortí, F. D. Igual, E. S. Quintana-Ortí, R. A. Van de Geijn, Solving dense linear systems on platforms with multiple hardware accelerators, in: ACM Sigplan Notices, Vol. 44, ACM, 2009, pp. 121–130.

[9] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, et al., Flexible development of dense linear algebra algorithms on massively parallel architectures with dplasma, in: Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on, IEEE, 2011, pp. 1432–1441.

[10] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, J. Langou, H. Ltaief, S. Tomov, Lu factorization for accelerator-based systems, in: Computer Systems and Applications (AICCSA), 2011 9th IEEE/ACS International Conference on, IEEE, 2011, pp. 217–224.

[11] Code saturne 5.0 theory guide.
URL http://code-saturne.org/cms/sites/default/files/docs/5.0/theory.pdf

[12] Y. Fournier, J. Bonelle, C. Moulinec, Z. Shang, A. Sunderland, J. Uribe, Optimizing code_saturne computations on petascale systems, Computers & Fluids 45 (1) (2011) 103–108.

[13] S. Moustafa, M. Faverge, L. Plagne, P. Ramet, 3d cartesian transport sweep for massively parallel architectures with parsec, in: Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International, IEEE, 2015, pp. 581–590.

[14] M. Cosnard, M. Loi, Automatic task graph generation techniques, in: System Sciences, 1995. Proceedings of the Twenty-Eighth Hawaii International Conference on, Vol. 2, IEEE, 1995, pp. 113–122.

[15] M. Cosnard, E. Jeannot, T. Yang, Compact dag representation and its symbolic scheduling, Journal of Parallel and Distributed Computing 64 (8) (2004) 921–935.

[16] E. Jeannot, Automatic multithreaded parallel program generation for message passing multiprocessors using parameterized task graphs, in: International ConferenceParallel Computing, 2001.

[17] E. Agullo, L. Giraud, A. Guermouche, S. Nakov, J. Roman, Task-based conjugate gradient: from multi-gpu towards heterogeneous architectures.

[18] P. Vezolle, J. Heyman, B. D'Amora, G. Braudaway, K. Magerlein, J. Magerlein, Y. Fournier, Accelerating computational fluid dynamics on the ibm blue gene/p supercomputer, in: Computer Architecture and High Performance Computing (SBAC-PAD), 2010 22nd International Symposium on, IEEE, 2010, pp. 159–166.