Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Profiling of Code_Saturne with HPCToolkit and TAU, and autotuning Kernels with Orio

B. Lindi[a*], T. Ponweiser[b], P. Jovanovic[c], T. Arslan[a]

[a]*Norwegian University of Science and Technology*
[b]*RISC Software GmbH*
A company of Johannes Kepler University Linz
[c]*Institute of Physics Belgrade*

Abstract

This study has profiled the application *Code Saturne*, which is part of the PRACE benchmark suite. The profiling has been carried out with the tools HPCtookit and Tuning and Analysis Utilities (TAU) with the target of finding compute kernels suitable for autotuning.
Autotuning is regarded as a necessary step in achieving sustainable performance at an Exascale level as Exascale systems most likely will have a heterogeneous runtime environment. A heterogeneous runtime environment imposes a parameter space for the applications run time behavior which cannot be explored by a traditional compiler. Neither can the run time behavior be explored manually by the developer/code owner as this will be too time consuming.
The tool Orio has been used for autotuning idenitified compute kernels. Orio has been used on traditional Intel processors, Intel Xeon Phi and NVIDIA GPUs.The compute kernels have a small contribution to the overall execution time for *Code Saturne*. By autotuning with Orio these kernels have been improved by 3-5%..

1. Introduction

The goals of this task have been to apply profiling tools and autotuning techniques to a real application important in PRACE. The application selected is *Code Saturne*, a Computaional Fluid Dynamics software package which is developed by Électrcité de France (EDF). *Code Saturne* is part of combined the PRACE and DEISA Unified European Application Benchmark Suite (UEABS).

Profiling and auto tuning is regarded as important and necessary steps for enabling a software for an exascale future. While profiling give insight to application behaviour, in an exascale environment will be highly dependent upon the input dataset. Autotuning offer methods to explore the parameter space governing the run time behavior for an exascale application in the combination with the dataset. As an application on an exascale system will be executed in a heterogeneous environment, the optimal execution path will not necessarily be known or found at a compile time. Search through different combinations of memory, cache and thread use, to mention a few parameters, needs to be carried out to find an optimal execution path, combining the best use of the different run time environments available.

1.1. On the road to exascale

State of the art multi-petascale systems comes in three hardware varieties[7]

- multicore systems like the Japanese K-computer
- manycore systems like JUQUEEN or the U.S Sequoia
- heterogeneous systems which combines traditional processor cores with accelerator processing units like the U.S systems Titan or Stampede

* Corresponding author. *E-mail address*: bjorn.lindi@ntnu.no

Of these technologies, only the line with accelerators seems to offer a path to Exascale. An Exascale system based on multicore technology does not seem feasible due to power requirements. The product line with embedded/manycore processors seems to end with the Blue Gene/Q design, which is the end of the Blue Gene line. This leaves us with the heterogeneous technology as the path forward towards Exascale performance.

Increasing application performance implies increased thread level-parallelism in the application. Huge performance gains can only be achieved by utilizing the abundance of threads which are offered by accelerator technologies like Graphic Processor Units (GPU) or many-core architectures like the Intel Xeon Phi. As pointed out in our previous report[8], adding an accelerator on to a compute node, increases the complexity of the run time system many fold. As heterogeneous systems contain two run time environments, the questions become what part of the application to run where, in what proportion.

Consequently code developers are faced with a landscape where the optimal executable cannot be produced by a traditional compiler alone. Different executions strategies needs to be considered, loops needs to be unrolled,possibly resulting in statements that can be executed in parallel by several threads, costly memory operations need to be minimized, and cache needs to be properly used both among traditional multi-core processors as well as co-processing. Hence, profiling becomes necessary for understanding code behaviour and identifying parts suitable for acceleration and autotuning becomes necessary for finding the executable with best performance. The search done with autotuning, may for a specific application even yield different executables for different workloads.

The profiling has been done with two different tools, *HPCToolkit*[3] and *Tuning and Analysis Utilities* (TAU)[5]. Auto-tuning has been carried out with *Orio*[6].

2. System setup and profiling

The work has been carried out on local workstations or on the Norwegian HPC-cluster “vilje”.”vilje” is a SGI ICE X system consisting of 1404 nodes connected with a Infiniband FDR fabric and a Lustre parallel file system. Each node has 32 GB of memory and two eight-core processors, giving a total of 16 cores per node. The processors are Intel Xeon 2670, running at 2.6 GHz with 20 MB L3-cache. The nodes are diskless.

SGI MPT 2.06 and MPT 2.09 have been used as MPI library for *Code_Saturne*. All our results are based on a standard build of *Code_Saturne* version 3.0.1, compiled with Intel compilers (version 14.0.1). We configured *Code_Saturne* for Intel MKL and large mesh support (`--with-blas --enable-long-gnum`). For profiling with *HPCToolkit*, a fully optimized debug build has been used (`-g -O3 -debug inline-debug-info`).

The software tools which have been employed are Tuning and Analysis Utilities (TAU) version 2.23, HPCToolkit version 5.3.2 and,Orio version 0.2.2.

2.1. The application Code Saturne

Code Saturne is an open source software package which can be used for CFD simulations. It solves the Navier-Stokes equations for different types of flow. For turbulence simulations can Renynolds-Averaged Navier-Stokes (RANS) metods or Large Eddy Simulations (LES) be used. The software have also specific modules for modelling combustion of coal or oil, semi-transparent radiative transfer, particle-tracking with Lagrangian methods, Joule effect, electrics arcs, weakly compressible flows, atmospheric flows, rotor/stator interaction for hydraulic machines

The software is developed by EDF.

2.2. Tuninig and Analysis Utilities (TAU)

TAU Performance System is a portfolio of tools for doing performance analysis of parallel programs written in Fortran, C, C++, UPC, Java, Python. TAU is capable of gathering performance information through instrumentation of functions, methods, basic blocks, and statements as well as event-based sampling. After a successful installation, TAU provides these scripts: *tau_f90.sh*, *tau_cc.sh*, and *tau_cxx.sh* to instrument and compile Fortran, C, and C++ programs respectively. TAU is developed by University of Oregon, Los Alamos National Laboratory and Forschungszentrum Jülich.

2.3. HPCToolkit

HPCToolkit is a suite of tools for performance measurement and analysis which is based on statistical sampling of timers and hardware performance counters. Program instrumentation is done by using the *hpcrun* launch script which inserts profiling code via LD_PRELOAD. To have performance information attributed to loops and source code lines (and not only on a function-level), debugging information needs to be included into the binary. Apart from that, no additional changes to the source code or to the build process are required. *HPCToolkit* is particularly well suited for performance analysis of compute kernel functions. It introduces low instrumentation overhead (approx. 3-5%) and yields accurate and detailed (source-line-level) performance information [3], [4].

2.4. Orio

Orio is a tool for auto-tuning of performance critical regions of code - typically at the level of C or Fortran loops. Orio focuses on thread-level ("intra-node") optimization and is therefore well suited for optimization of computation kernel functions.

Conceptionally, the input to *Orio* is an annotated computation specification written in some domain specific language. The output is an implementation of the given computation specification in a selected target language (currently C/C++, CUDA or OpenCL supported) which is optimized for a specific problem size and system architecture with respect to the given annotations.

As input language, currently just the so-called *loop-language* is implemented, which is a basically C with certain restrictions on how loop bounds and index increments may be formulated. However, future extensions of Orio may as well support higher-level input languages such as for example Matlab. Such languages allow concise representations of more complex computations (e.g. operations on matrices and vectors) and make possible the implementation of powerful higher-level code transformations, resulting ideally in both, reduced code development time and better runtime performance.

The typical workflow for performance optimization with Orio is as follows: Given an existing application and some run-time critical region of code, first a computation specification for Orio has to be created. In case that the original application is written in C, this step usually involves only marginal adaptations to the given original code.

Second, the computation specification has to be annotated in order to instruct Orio which code transformations to apply (e.g. loop unrolling) and which optimization parameters to use for these transformations (e.g. loop unroll factors). All optimization parameters together with their associated allowed value ranges give rise to an (exponentially large) *optimization parameter space*.

According to the given computation specification and optimization parameter space, Orio generates, compiles, executes and times many different program versions. As output it reports the optimization parameter configuration which led to the best observed performance as well as the associated generated code, which may manually be reintegrated back to the original application.

For this empirical tuning process, different search heuristics (e.g. Exhaustive search, random search, simulated annealing, etc.) and search limitations (e.g. maximum search time) can be selected. However, not all of them are currently implemented [6] and [11].

3. Performed work and results

3.1. Testing and benchmark cases

For identification of critical kernel functions which may serve as candidates for auto-tuning, two benchmark cases for *Code_Saturne* have been prepared. The first benchmark case, "T-junction", has been used for preliminary scalability and profiling results and for comparison and validation of the results obtained with the final (larger) "Tube bundle" case.

3.1.1. The T-junction case

We followed the official *Code_Saturne* tutorial case "Turbulent mixing in a T-junction",[2]. The modeled physical problem is turbulent mixing between hot and cold water inside a pipe which is composed of a T-junction and an elbow.

Two variants of this case have been created. For the first variant, we used the precise tutorial parameters and geometry, i.e. pipes with circular cross-sectional area. This yields an unstructured mesh with mixed cell types.

Unfortunately, refinement of this mesh beyond 4 million cells turns out to be problematic. For this reason, the mesh geometry has been simplified. By using cuboid building blocks, we obtain an easily scalable purely hexahedral mesh. Note that the area of the tube cross-section is roughly the same for both geometries and the physical parameters are the same for both test cases.

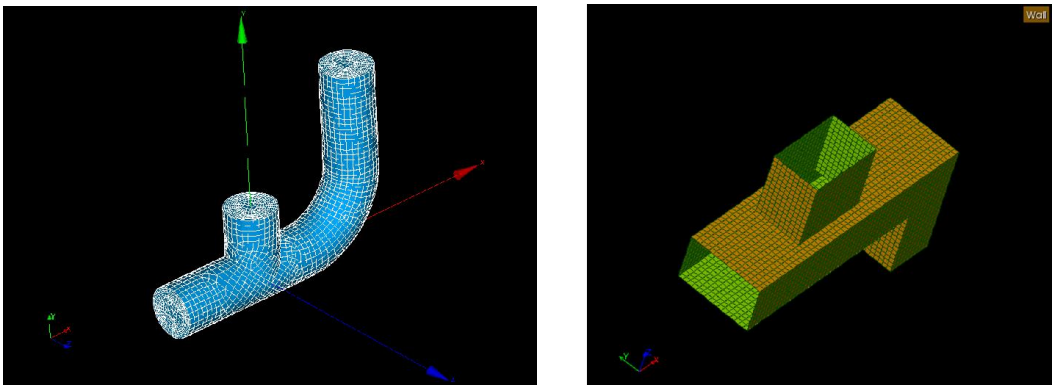


Figure 1: Tubular (left) and hexahedral (right) T-junction mesh geometry.

3.1.2. The Tube bundle case

The chosen benchmark case was based on the simulation of the flow over a staggered tube bundle as a part of a nuclear reactor with Large Eddy Simulation (LES). The benchmark problem has been selected for evaluation of the parallel performance of PRACE Tier-0 systems and the comparative results can be found in Moulinec et al. [9].

Figure 2 shows the computational domain, a subset of the tube bundle simulation. The tube diameter is D=22.7 mm and length is L= 64 mm. Reynolds number based on the bulk velocity and tube diameter (Re=UD/ν) is 18,000. The computational domain is created by extracting only one tube and the surrounding fluid part from flow field around the tube bundle. The faces of this domain are considered as periodic faces as seen in the figure.

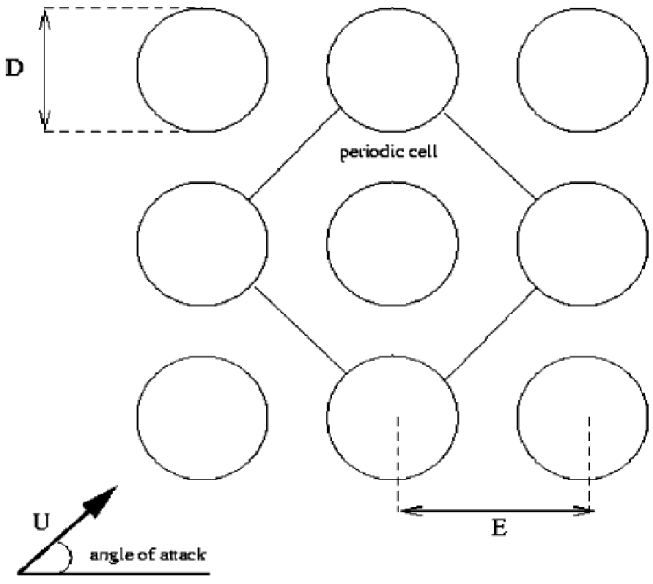


Figure 2: Flow over tube bundle

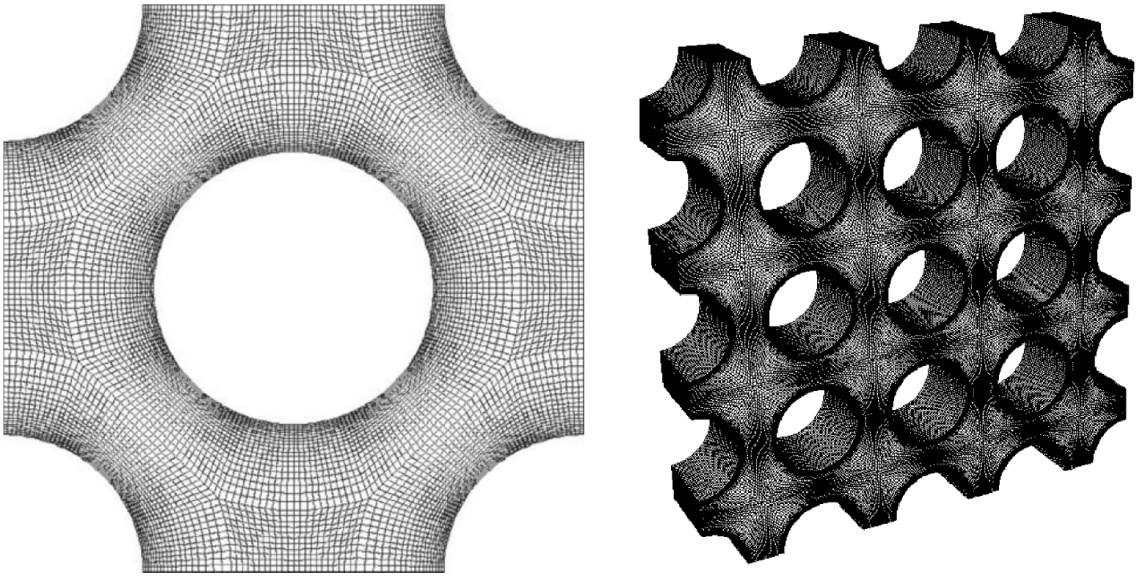


Figure 3: Mesh for one tube (left) and multiplied (3x3) mesh (right)

The resulting domain and the created mesh can be seen in Figure 3(left). In this example, the mesh contains 12,805,120 hexagonal cells. The mesh for a single tube is copied in a formation (3x3) to create a bundle which can be seen in Figure 3(right). With this methodology, it is possible to reach over 200 million cells with a small effort. *Code_Saturne* is able to copy or multiply one single mesh zone to create larger domains or merge different mesh zones which are created separately. This process can be done in parallel to avoid memory overflows. More details about the flow parameters and mesh can be found in Moulinec et al. [9]and Benhamadouche and Laurence [10]. In these benchmark tests, 4x4 (51 Million) and 16x16 (204 Million) configurations are used as seen inTable 1.

Type of mesh	Number of cells
Single tube(original)	12,805,120
2x2 full tubes	51,220,480
4x4 full tubes	204,881,920

Table 1: Mesh sizes for Tube bundle case

3.1.3. Preliminary scalability results

Code_Saturne is implemented as hybrid application, i.e. it offers parallelism at the level of message passing (MPI) and optionally threading (OpenMP.) In order to find a good ratio between number of MPI ranks and OpenMP threads for optimal computation efficiency, we first examined the T-Junction case with tubular pipe geometry, 218k cells mesh, running on 1 computation node. Varying the number of MPI ranks while leaving the total number of OpenMP threads (16) constant shows that the best efficiency is achieved with a purely MPI-parallelized version of Code_Saturne (seeTable 2). We therefore simply disabled OpenMP in the build process for all our further investigations.

MPI ranks	Threads per MPI rank	Wall clock (s)
16	1	105
8	2	120
4	4	143
2	8	172
1	16	325

Table 2: Parallel efficiency comparison when utilizing all 16 cores of one computation node: Best efficiency is achieved when using a non-hybrid (MPI-only) version of Code_Saturne.

3.2. Profiling and identification of performance critical kernel routines

3.2.1. TAU results

Simulations are performed for 100 time steps for each simulation. The time step size is 2×10^{-5} seconds. In Figure 4 and Figure 5, profiling results based on 2048 and 4096 CPUs are showed for the case for 204 million cells Tube bundle test case. For both calculations, *_mat_vec_p_l_native* and *_iterative_scalar_gradient* functions have the highest exclusive time disregarding MPI routines. The total wall clock time is around 1200 and 700 seconds for the cases with 2048 and 4096 cores respectively. Note that the profile figures show only the essential section of the profile listings, as a complete profile listing is to large, see Appendix B for a full profile listing.

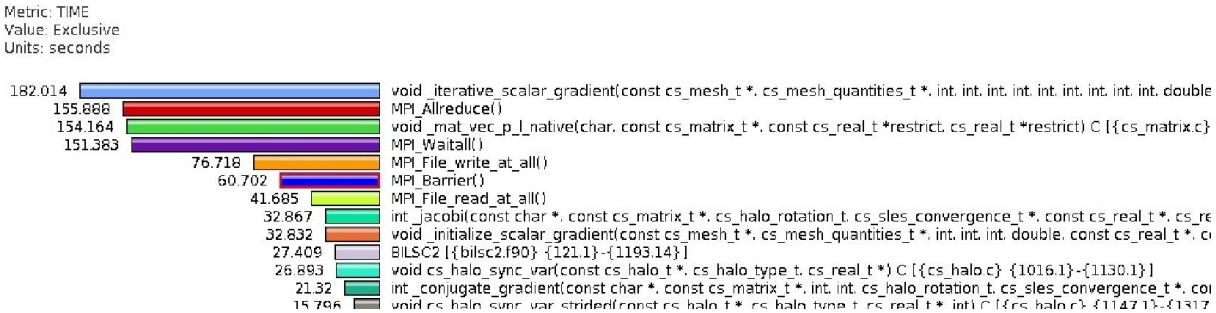


Figure 4: Profiles of functions (averaged on 2048 cores-128 nodes) from*Code_Saturne*, in decreasing order of exclusive time, 204M case.

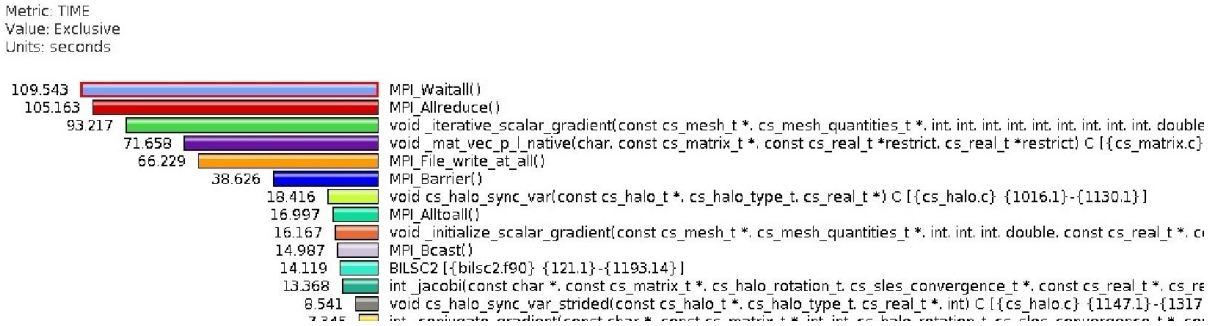


Figure 5: Profiles of functions (averaged on 4096 cores-256 nodes) from Code_Saturne, in decreasing order of exclusive time, 204M case.

In Figure 6, Figure 7 and Figure 8, profiling results based on 1024, 2048 and 4096 CPUs are showed for the case for 51 million Tube bundle test case. For both calculations, *_mat_vec_p_l_native* and *_iterative_scalar_gradient* functions again have the large exclusive time than the others. The total wall clock time is around 527, 368 and 302 seconds for the cases with 1024, 2048 cores and 4096 cores respectively. The exclusive time for MPI calls are highly increasing by CPU number and *_mat_vec_p_l_native* and *_iterative_scalar_gradient* functions has very small exclusive time for 51 million mesh especially when 4096 CPUs is used.

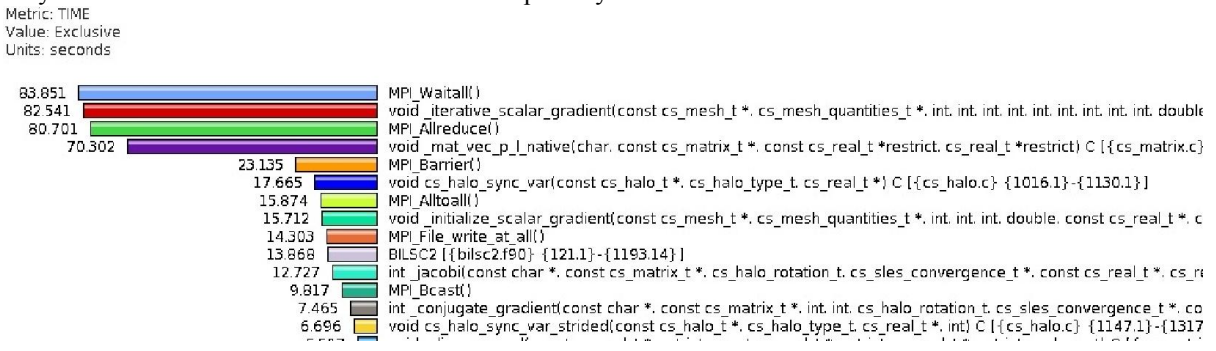


Figure 6: Profiles of functions (averaged on 1024 cores-64 nodes) from Code_Saturne, in decreasing order of exclusive time, 51M case.

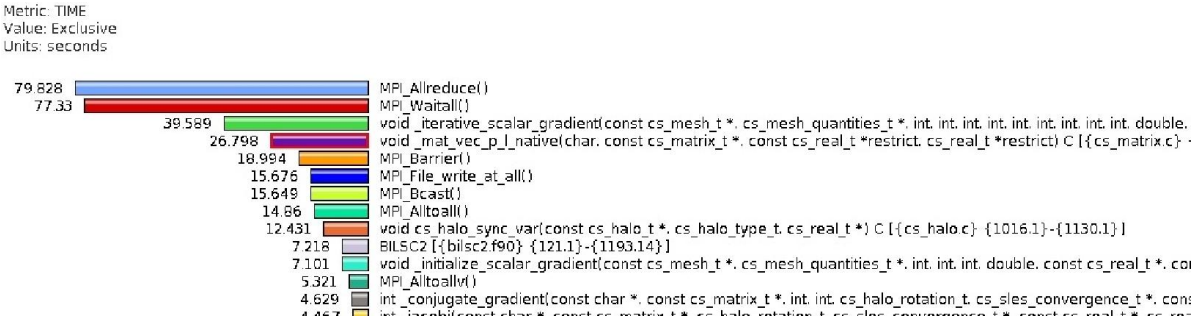


Figure 7: Profiles of functions (averaged on 2048 cores-128 nodes) from Code_Saturne, in decreasing order of exclusive time, 51M case.

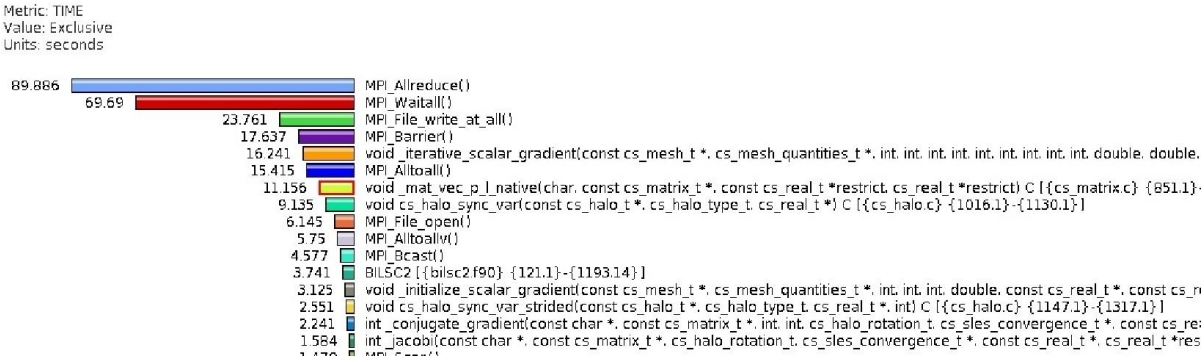


Figure 8: Profiles of functions (averaged on 4096 cores-256 nodes) from Code_Saturne, in decreasing order of exclusive time, 51M case.

3.2.2. HPCToolkit results

As a simple metric for identification of performance critical kernel routines, we select the number of processor cycles (PAPI_TOT_CYC) spent within a given routine *R*, excluding those cycles which are spent in routines called by *R*. We call this metric the exclusive time of a routine.

With *HPCToolkit* we examined both, the T-junction and the Tube bundle case. For the T-junction case, we use the purely hexahedral variant, meshed with 54 million cells. The mesh size for the the Tube bundle case is 200 million cells. The functions with highest exclusive time for these two test cases are listed in Table 3 and Table 4. In both cases, the routine *_mat_vec_p_l_native* has a relative high contribution to the overall runtime. The routine *_iterative_scalar_gradient* seems to be the most critical only for the Tube bundle test case. The explanation for this seems to lie in the fact that *_iterative_scalar_gradient* is significantly faster for perfectly orthogonal meshes than for more typical meshes with moderate non-orthogonality, like the Tube bundle case.

	256	512	1024	2048
pthread_spin_lock	3.5%	6.2%	13.9%	17.3%
poll_quicks	3.6%	6.0%	7.1%	8.8%
MPI_SGI_shared_progress	3.9%	6.4%	7.0%	8.4%
_mat_vec_p_l_native	**20.8%**	**16.2%**	**8.9%**	**4.5%**
_iterative_vector_gradient	4.6%	4.2%	3.6%	2.4%
_iterative_scalar_gradient	2.7%	2.4%	2.1%	1.4%
_conjugate_gradient	7.2%	4.1%	1.5%	0.8%
_polynomial_preconditionning	4.6%	2.9%	1.2%	0.5%
_diag_vec_p_l	4.5%	3.4%	1.3%	0.4%
cs_dot_xy_yz	3.1%	2.2%	0.9%	0.4%

Table 3: Hotspot routines by exclusive time for the 54M T-junction test case for different numbers of MPI processes.

	1024	2048	4096	8192
pthread_spin_lock	7.7%	11.1%	13.9%	17.9%
~unknown-proc~	4.0%	5.8%	7.3%	9.5%
MPI_SGI_shared_progress	3.7%	5.5%	6.7%	9.3%
_iterative_scalar_gradient	**17.3%**	**15.3%**	**13.3%**	**8.3%**
poll_quicks	3.1%	4.2%	5.5%	6.5%
_mat_vec_p_l_native	**16.1%**	**13.1%**	**10.1%**	**4.6%**
_initialize_scalar_gradient	3.4%	2.8%	2.3%	1.3%
_jacobi..0	3.3%	2.2%	1.5%	0.6%
_conjugate_gradient	2.9%	1.6%	0.8%	0.5%

Table 4: Hotspot routines for the 200M Tube bundle case for different numbers of MPI processes.

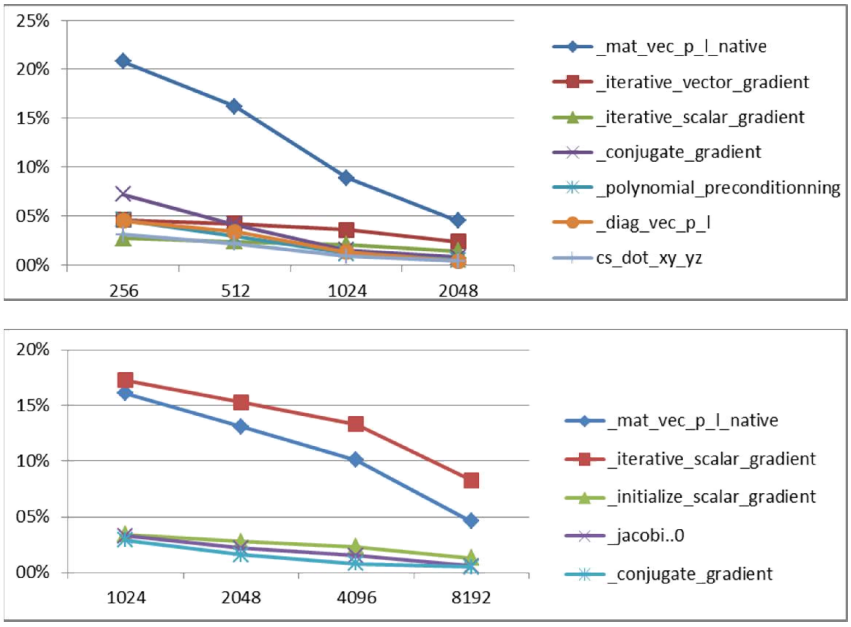


Figure 9: Visual comparison of hotspot routines for T-Junction (top) and Tube bundle (bottom) testcase.

3.2.3. Selected kernel routines for optimization

The above profiling results suggest putting our focus on the two kernel routines *_mat_vec_p_l_native* and *_iterative_scalar_gradient*. For the following, we introduce the abbreviations MVN (*_mat_vec_p_l_native*) and ISG (*_iterative_scalar_gradient*) for these two routines.

```
for (face_id = 0; face_id <= n_faces-1; face_id++) {
ii = face_cel_p[2*face_id] -1;
   jj = face_cel_p[2*face_id + 1] -1;
   y[ii] += xa[face_id] * x[jj];
   y[jj] += xa[face_id] * x[ii];
}
```

Figure 10: Runtime critical loop within the kernel routine *_mat_vec_p_l_native*.

MVN computes a matrix vector product $y = Ax$ for a sparse n-by-n matrix A, where n corresponds to the number of cells in the process local part of the mesh. Note that the matrix entries $a_{i,j}$ of A are nonzero only if $i = j$ or if the cells with indices i and j are adjacent. The most time-consuming part of MVN is the calculation of the off-diagonal portion of the matrix vector product, where there exist two different code paths for the symmetric and general case respectively. For our test cases, both code paths account for approximately half of the total runtime of MVN. For *Orio* optimization, we looked at the symmetric case only and selected the critical loop listed in Figure 10- the asymmetric case can be handled in a completely analogous way.

ISG serves for the computation of cell gradients. A special iterative method is employed for non-orthogonal meshes. ISG consists of more than 500 lines of C code, making it hard to apply *Orio* to the whole routine. We therefore only concentrate on the most runtime-critical nested loop within ISG, listed in Figure 11 and accounting for approximately 60% of the total runtime of ISG.

```
for (g_id = 0; g_id < n_i_groups; g_id++) {

# pragma omp parallel for private(face_id, ii, jj, pfac, fctb)
  for (t_id = 0; t_id < n_i_threads; t_id++) {

    for (face_id = i_group_index[(t_id*n_i_groups + g_id)*2];
       face_id < i_group_index[(t_id*n_i_groups + g_id)*2 + 1];
       face_id++) {

       ii = i_face_cells[face_id][0] - 1;
       jj = i_face_cells[face_id][1] - 1;

       pfac  =       weight[face_id]  * rhsv[ii][3]
              + (1.0-weight[face_id]) * rhsv[jj][3]
              + ( dofij[face_id][0] * (dpdxyz[ii][0]+dpdxyz[jj][0])
              +   dofij[face_id][1] * (dpdxyz[ii][1]+dpdxyz[jj][1])
              +   dofij[face_id][2] * (dpdxyz[ii][2]+dpdxyz[jj][2])) * 0.5;
       fctb[0] = pfac * i_face_normal[face_id][0];
       fctb[1] = pfac * i_face_normal[face_id][1];
       fctb[2] = pfac * i_face_normal[face_id][2];
       rhsv[ii][0] += fctb[0]; rhsv[ii][1] += fctb[1]; rhsv[ii][2] += fctb[2];
       rhsv[jj][0] -= fctb[0]; rhsv[jj][1] -= fctb[1]; rhsv[jj][2] -= fctb[2];

    } /* loop on faces */
  } /* loop on threads */
} /* loop on thread groups */
```

Figure 11: Runtime-critical loop within the kernel routine *_iterative_scalar_gradient*.

4. Optimization with Orio

4.1. Optimization for CPU

Following *Orio*'s standard workflow, we isolated the two performance critical loops of MVN and ISG (seeFigure 10 and Figure 11) in two separate input files for *Orio* and annotated the code with appropriate auto-tuning instructions.

In order to provide representative input data for the optimization process, we additionally inserted code into MVN and ISG for the file output of the relevant runtime data. We rebuilt this modified version of *Code_Saturne* and ran the 200 million cells Tube bundle test case with 1024 MPI processes, which we consider to be representative for typical use cases. The generated data files served as input for the *Orio* auto-tuning process.

In a first step, we applied only loop-unrolling to the selected loops of MVN and ISG. For MVN, the annotated code for Orio is listed in Figure 12. The *Orio*-generated code - *Orio* reported 5 as optimal unroll factor - can be seen in Figure 13. Note that both code snippets have been shortened. Indeed the tuned code is measurably faster than the original code. However the speedup is not particularly high: For both routines, MVN and ISG, the total speedup ranges approximately between 3 and 5%, comparing generated and original code with full compiler optimization (-O2 or -O3).

```
/*@ begin PerfTuning (
  def build { ... }
  def performance_counter { ... }
  def performance_params {
        param UF[] = range(1,8);
  }
  def search {
        arg algorithm = 'Exhaustive';
  }
  def input_params { }
  def input_vars
  {
        arg decl_file = 'decl.h';
        arg init_file = 'init.c';
  }
) @*/

  int face_id, ii, jj;
/*@ begin Loop( transform Unroll(ufactor=UF)

  for (face_id = 0; face_id <= n_faces-1; face_id++) {
ii = face_cel_p[2*face_id] -1;
     jj = face_cel_p[2*face_id + 1] -1;
     y[ii] += xa[face_id] * x[jj];
     y[jj] += xa[face_id] * x[ii];
}

) @*/
/*@ end @*/
/*@ end @*/
```

Figure 12: Orio input file for optimization (loop-unrolling) of *_mat_vec_p_l_native*.

```
for (face_id=0; face_id<=n_faces-5; face_id=face_id+5) {
ii=face_cel_p[2*face_id]-1;
   jj=face_cel_p[2*face_id+1]-1;
   y[ii]=y[ii]+xa[face_id]*x[jj];
   y[jj]=y[jj]+xa[face_id]*x[ii];
   ii=face_cel_p[2*(face_id+1)]-1;
   jj=face_cel_p[2*(face_id+1)+1]-1;
   y[ii]=y[ii]+xa[(face_id+1)]*x[jj];
   y[jj]=y[jj]+xa[(face_id+1)]*x[ii];
   ...
   ii=face_cel_p[2*(face_id+4)]-1;
   jj=face_cel_p[2*(face_id+4)+1]-1;
   y[ii]=y[ii]+xa[(face_id+4)]*x[jj];
   y[jj]=y[jj]+xa[(face_id+4)]*x[ii];
}
for (face_id=n_faces-((n_faces-(0))%5); face_id<=n_faces-1; face_id=face_id+1) {
   ii=face_cel_p[2*face_id]-1;
   jj=face_cel_p[2*face_id+1]-1;
   y[ii]=y[ii]+xa[face_id]*x[jj];
y[jj]=y[jj]+xa[face_id]*x[ii];
}
```

Figure 13: Orio-generated code for _mat_vec_p_l_native.

Unfortunately, despite the kind help and advice of Orio's main developer, Boyana Norris, we did not succeed in getting any further performance improvement by applying additional code transformations other than loop unrolling. Due to the irregular data access pattern in both selected loops, loop vectorization is not applicable. Moreover, loop parallelization has been excluded in the first place, because our initial investigations showed that *Code_Saturne* performs best when running with single-threaded MPI processes.

Indeed, a deeper analysis of MVN and ISG shows that both routines suffer from the same fundamental problem: Both selected loops iterate exactly once over all internal faces of the process-local part of the mesh. Note that the loop in ISG is a bit more advanced in that the iteration is partitioned into multiple passes, where in each pass

independent groups of faces are treated in parallel by multiple threads.[1] However, in our single-threaded testing setup, this degenerates essentially to same type of iteration as in MVN. The total number of iterations for the outer two loops in ISG is exactly one; the innermost loop iterates over all internal local mesh faces for our test case.

For both, MVN and ISG, the indices *ii* and *jj* refer to the two mesh cells which are adjacent to the mesh face of the current iteration. Analysis of the exported runtime data shows that these indices jump very arbitrarily from iteration to iteration. Consequently, the (read and write) access of associated cell data is very inefficient because cache misses occur with very high probability. It seems that things get even worse if this irregular data access pattern is carried out by multiple threads in parallel. Here, in addition to expensive load and store operations from/to main memory, also false sharing effects are observable (threads are frequently invalidating each others cache lines). We assume that this observation might be a possible explanation for the decreased efficiency when comparing the hybrid (MPI + OpenMP) variant of Code_Saturne with a purely MPI-parallelized version.

We come to the conclusion that for a further optimization of MVN and ISG, a smart reorganization of the involved data structures (such as for example the introduction of a thread-level mesh decomposition) as well as appropriate algorithmic reformulations are required. Clearly such fundamental changes to the code lie beyond the capabilities of Orio (and most likely beyond the capabilities of any other fully-automated tool available in the foreseeable future).

4.2. Optimization for GPU

In our benchmarks, we were unable to find any functions in *Code_Saturne* suitable for speedup on the GPU. The main problem was that the execution on GPU implies relatively slow data transfers between the host memory and the GPU memory, i.e. transfer of function arguments and results, which require the calculation to be very dense and long running on CPU in order to gain performance on the GPU. However, all the hotspots we identified in *Code_Saturne* were short running functions which are called many times. Also, *Orio*'s code transformation that supports CUDA is still under development and there are significant problems when trying to process anything more complex than the demo codes. Thus we have tried *Orio*'s GPU features on a synthetic test, which is the usual matrix-matrix multiplication. Even though the test does not have a direct relation to *Code_Saturne*, it can demonstrate *Orio*'s GPU features well while maintaining simplicity.

Orio provides kernel and host code generation and performance tuning for CUDA. In the latest revisions, at the time of this writing, there is also a skeleton for OpenCL support. Both features are implemented as submodules of the loop transformation module.

CUDA loop transformation takes the following parameters:

- *threadCount* - number of threads per block.
- *blockCount* - number of thread blocks
- *cacheBlocks* - boolean that controls whether to copy data that thread uses into shared memory.
- *pinHostMem* - whether to pin memory on the host, so it won't be paged out. This can speed up memory transfers because there is no need to copy data in host memory to a pinned buffer, but it should be used with care because the unpageable memory can fill up host memory pretty quickly.
- *streamCount* - number of streams for overlapped computation and communication.
- *unrollInner* - unroll inner loops.
- *preferL1Size* - on devices where L1 cache and shared (local) memory are in the same hardware, this configures the size of L1 cache.
- *dataOnDevice* - at the time of this writing this parameter is unused.

Value ranges for these parameters are specified in the *Orio*'s performance tuning annotation and can be explored using the standard autotuning feature. The performance tuning part is specified in the same way as for other transformation (e.g. previously mentioned Loop unrolling), with the addition of *build_command* argument in the build definition, which specifies how the CUDA compiler is to be invoked.

---

[1] In this context we call two faces independent, if they have no adjacent cell in common.

```
/*
 * A - m x N matrix of doubles.
 * B - N x p matrix of doubles.
 * C - m x p matrix that holds A*B
 */

/*@ begin Loop(transform CUDA(threadCount=256, blockCount=24,
                              preferL1Size=16, cacheBlocks=False)
for (i = 0; i <= m-1; i++) {
  for (j = 0; j <= p-1; j++) {
    s = 0.0;
    for (k = 0; k <= N-1; k++) {
      s += A[i*N+k]*B[k*p+j];
    }
    C[i*N+j] = s;
}
) @*/
/*@ end @*/
```

Figure 14: Matrix multiplication code annotated for Orio's CUDA transformation.

Figure 14 shows the annotated code which is used as an input to *Orio*'s CUDA transformation. Because of the way CUDA transformation generates the code, the variables declared in *PerfTuning* annotation as *input_vars* (*m*, *n*, *p*) become global host variables, and those in *input_params* (*M*, *N*, *P*) become defined as macros. This complicates the transformation since not all global variables used inside code loops are recognized as arguments for the device kernel, and that is the reason why the *N* used in the code was actually the macro for input parameter *N*. There are other inconveniences in working with the CUDA transformation, such as some unsupported parts of the C syntax, but these can be expected to improve since the feature is still being developed.

The generated CUDA kernel code is shown inFigure 15. Also a function for kernel launch is generated, but its contents are omitted for brevity. Depending on the performance tuning specification, Orio tries out many versions of the kernel code and launch dimensions, and generates the best performing ones along with the comments on their performance and selected parameters.

```
__global__ void orcu_kernel146(const int m, const int p, double* A, double* B,
double* C) {
  const int tid=blockIdx.x*blockDim.x+threadIdx.x;
  const int gsize=gridDim.x*blockDim.x;
  double s;
  int j, k;
  for (int i=tid; i<=m-1; i+=gsize) {
    for (j=0; j<=p-1; j++ ) {
      s=0.0;
      for (k=0; k<=N-1; k++ ) {
        s=s+A[i*N+k]*B[k*p+j];
      }
      C[i*N+j]=s;
    }
  }
}

void MatMatMult(double* A, double* B, double* C, int m, int n, int p) {
...
kernel launch code
...
}
```

Figure 15: Generated CUDA kernel and kernel launch function signature.

We conclude that this feature shows a lot of promise to eliminate much of the hard work around programming for CUDA, especially in searching for the optimal parameters for the underlying GPU. Unfortunately, as in the CPU optimization case, the organization of code and algorithms in *Code_Saturne* remain beyond capabilities of this automated transformation.

4.3. Acceleration of kernel functions for Intel Xeon Phi

Our latest attempt for increasing the performance of *Code_Saturne*'s kernel functions has been acceleration using Intel Xeon Phi coprocessors. For estimation of the optimization potential using this technology, we examine the performance of *Code_Saturne* running as single MPI process, which executes certain selected kernel routines on the coprocessor card. Unfortunately, *Orio*'s code generation support for Xeon Phi is still very preliminary. Therefore we manually inserted OpenMP 4.0 pragmas for offloading into *Code_Saturne*.

We selected *Code_Saturne*'s kernel routine *_jacobi* as a good candidate for offloading, as it shows a significant contribution to the total runtime when profiling the 200 million cells Tube bundle test case (seeFigure 16). Moreover, its call hierarchy is not too complex which enables a relatively straight-forward migration to Xeon Phi. It is important to note that there is one fundamental limitation connected to our choice: Execution of the function *_jacobi* involves an MPI reduction, which is not supported in an offload region[12]. For this reason, our modified code only runs in the case when MPI parallelization is disabled. Nevertheless, we consider our results as useful in that they yield an upper bound for the acceleration potential lying in the routine *_jacobi*.

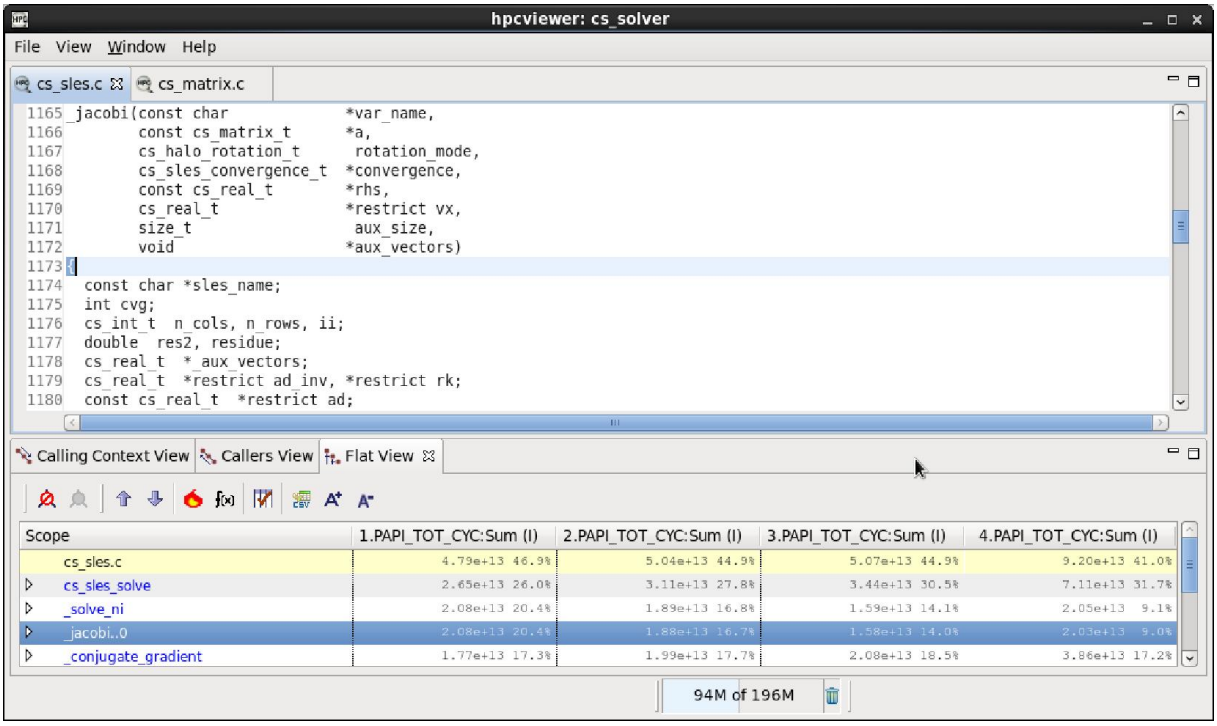


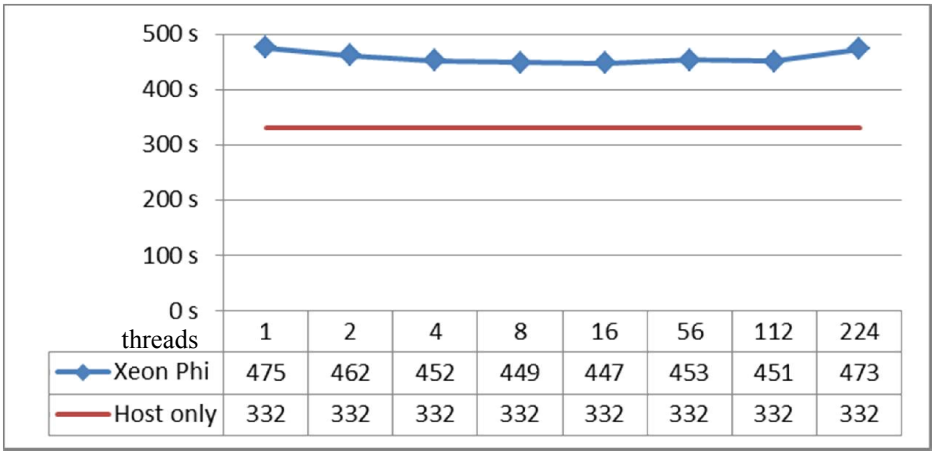
Figure 16: HPCToolkit profile for 200M Tube bundle test case for (1.) 1024, (2.) 2048, (3.) 4096 and (4.) 8192 processes.

For migration of *_jacobi* to Xeon Phi, we annotated all functions called by *_jacobi* with the OpenMP's "declare target" pragma accordingly (`#pragma omp declare target` and `#pragma omp end declare target`), which instructs the compiler to additionally emit code for execution on Xeon Phi. A limitation connected to offload functions is that only "plain old data" types (i.e. primitive datatypes as well as structs or fixed-sized arrays and combinations of those) are supported as function arguments. This makes it necessary to decompose more complex data structures (like the *Code_Saturne*'s mesh data structures) into their primitive components prior to passing them as parameters to the offload function. In our case, this decomposition is done directly in our modified version of the *_jacobi* function. The actual computation then is delegated to a newly introduced function *_jacobi_mic* (seeFigure 17) which is executed on the coprocessor.

In addition to these changes to the source code, also a slight modification to the standard build process is necessary, specifying Intel's *xiar* archiving tool and the parameter `-qoffload-build` in order to generate shared library variants for Xeon Phi execution[13].

```
#pragma omp target \
map(to:\
var_name[0:strlen(var_name)+1],\
   sles_name[0:strlen(sles_name)+1],\
   xa[0:(symmetric ? n_faces : 2 * n_faces)],\
   ad[0:n_rows],\
face_cel_p[0:2*n_faces],\
rhs[0:n_rows])\
map(tofrom:\
   convergence[0:1],\
   vx[0:n_rows])
cvg = _jacobi_mic(var_name, sles_name, xa, n_cols, n_rows, ad,
                  n_faces, n_cells_ext, symmetric, face_cel_p,
                  convergence, rhs, vx);
```

Figure 17: Offloading of _jacobi to Intel Xeon Phi using OpenMP 4.0.



threads	1	2	4	8	16	56	112	224
Xeon Phi	475	462	452	449	447	453	451	473
Host only	332	332	332	332	332	332	332	332

Figure 18: Total runtime (s) of T-junction testcase with single-threaded host-process.
The number of OpenMP threads for Xeon Phi is varied.

At the time of writing, we have approximate results for the overall runtime. Exact detailed profiling results have proven more challenging to achieve with our modified version of Code Saturne. Figure 18 shows the wall-clock time for the 1 million cells hexahedral T-junction case, running on 1 MPI rank (host process) with varying number of threads for the Xeon Phi card. As can be seen, the runtime not change significantly depending on the chosen number of Xeon Phi threads. Moreover, the best performance is still achieved by the reference run with the original (host-only) version of *Code_Saturne*. From these observations we deduce that the runtime of the modified *_jacobi* function is completely dominated by data transfer operations from and to the coprocessor. Clearly this data transfer overhead could be reduced by copying constant mesh data (the arrays `xa`, ad and `face_cel_p`) to the coprocessor only once.

According to our profiling results on the original (host-only) code version, 266 seconds (out of the total execution time of 332 seconds) are spent in routines other than *_jacobi*. The best observed runtime for the Xeon Phi code variant was 447 seconds (at 16 threads). Assuming that the execution time of the code outside *_jacobi* (which has not been changed) is the same for the original and modified code version, the times for *_jacobi* are 332-266 = 66s (host-only) vs. 447-266 = 181s (Xeon Phi). The constant mesh data structures (the arrays `xa`, `ad` and `face_cel_p`) currently make up about 70% of the total data transferred to/from the coprocessor. When copying this mesh data to the coprocessor only once at the initialization phase, we therefore would obtain 54 seconds (30% of 181s, disregarding the time spent in actual computation) as estimated lower bound for the best achievable runtime of *_jacobi* on Xeon Phi. Comparing this with the time of the host-only execution (66 seconds), we therefore do not expect any significant acceleration potential within *_jacobi* using Intel Xeon Phi.

5. Conclusion and Outlook

In this work we aimed for the optimization and acceleration of certain selected computation kernel routines of Code_Saturne. This goal is quite ambitious because as part of the PRACE application benchmark suite,

Code_Saturne already shows a very good performance and scalability behaviour compared to other widely known high-performance CFD codes.

Indeed we observed that the major performance limiting factor for Code_Saturne lies within MPI communications, which is not uncommon for typical well-performing HPC applications. Still, our investigations also revealed some optimization potential in kernel routines for sparse matrix vector products or gradient calculation. According to our insights, this optimization potential lies mostly in a smart reorganization of Code_Saturne's mesh data structures as well as in an appropriate algorithmic reformulations which allow more cache friendly data access patterns. Unfortunately, such a kind of optimization lies beyond the capabilities of auto-tuning tools such as Orio.

Nevertheless, we believe that code generation and auto-tuning tools such as Orio will get more and more essential for the development of HPC applications. Having future multi-petascale and (possibly) exascale systems in mind, the unbroken trend to higher degrees of heterogeneity and architectural complexity will impose a huge challenge to application programmers. The task of performance optimization, in particular keeping track of the ever-rising number of relevant optimization parameters, will be practically undoable without appropriate and widely automated tool support. Another trend, founded on architectural considerations for future HPC systems, is the increasing importance of alternative optimization objectives, such as memory accesses and energy efficiency [7].

With respect to heterogeneity, there is still much ground to cover for Orio. Its support for CUDA and OpenCL is still quite preliminary. Another yet unsolved challenge for Orio is its extension to a tool which covers not only thread-level optimization, but also optimization of MPI communications. Here, a promising approach might be the integration of higher-level input languages which allow more abstract formulations of distributed-memory algorithms. Regarding optimization with respect to memory accesses and energy efficiency, Orio seems to be on a track: The support of autotuning with respect to hardware performance counters has been announced. Additional studies for multi-objective optimization with Orio have already been carried out.[11].

References

[1] http://www.prace-ri.eu/ueabs#Code_Saturne
[2] http://code-saturne.org/cms/documentation/Tutorials
[3] http://hpctoolkit.org
[4] http://hpctoolkit.org/manual/HPCToolkit-users-manual.pdf
[5] http://www.cs.uoregon.edu/research/tau/home.php
[6] http://brnorris03.github.io/Orio/
[7] "No exascale for you" Horst Simon, Optical Interconnects Conference, May 6, 2013
[8] PRACE-3IP Deliverable 7.2.1 "A report on the survey of HPC tools and techniques"
[9] Moulinec, C., Sunderland, A. G., Kabelikova, P., Ronovsky, A., Vondrak, V., Turk, A., Aykanat, C., and Theodosiou, C., 2012, "Optimization of Code_Saturne for Petascale Simulations", PRACE white paper.
[10] Benhamadouche, S. and Laurence, D., 2003, "LES, coarse LES, and transient RANS comparisons on the flow across a tube bundle", International Journal of Heat and Fluid Flow, **24**(4), pp. 470-479.
[11] http://www.rce-cast.com/Podcast/rce-83-orio.html
[12] http://software.intel.com/en-us/articles/using-mpi-and-xeon-phi-offload-together
[13] http://software.intel.com/en-us/node/459136

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-312763.

Appendix A

Installation of Code Saturne with TAU implementation

Code_saturne can be compiled with different options/parameters as linear algebra libraries, partitioning methods, mesh importing formats, MPI/OpenMP by using several different external libraries. This document helps to develop and profile the code after a successful installation. In this page, practical information will be documented for installation on Linux. Users can download the source code and third part libraries from the links and follow the instructions. The detailed installation guide can be found on the official website of EFD: http://code-saturne.org/cms/sites/default/files/install-3.2.pdf

Downloading necessary files

The instructions at this page are based on the version 3.2.2 and the source files can be downloaded from here http://code-saturne.org/cms/download/3.2, for all other versions: http://code-saturne.org/releases/
Several optional libraries can be linked with the code for a more complete experience of Code_Saturne. These are:

Graphical Interface(GUI)

This option is not necessary for VILJE. It should be installed for local computers to use GUI.

- Libxml2 must be installed. (Can be downloaded here: http://xmlsoft.org/downloads.html)
- PyQt and Qt must be installed.(http://www.riverbankcomputing.co.uk/software/pyqt/intro)

Mesh format for pre- and post-processing

- CGNS, is library and generic file format for CFD data, It is useful for importing mesh files from ANSYS. The binary files for version (can be downloaded from http://cgns.sourceforge.net/)
- MED is library and file format for the tool SALOME which is an open-source software that provides a generic platform for Pre- and Post-Processing for numerical simulations. The binary files for version, can be downloaded from here http://www.salome-platform.org/
- HDF5 is a data model, library, and file format for storing and managing data. It supports an unlimited variety of data types, and is designed for flexible and efficient I/O and for high volume and complex data. It can be downloaded from here http://www.hdfgroup.org/HDF5/

The only compulsory pre-requisites needed are compilers for C/C++/Fortran (like GNU or Intel compilers) and a Python interpreter.

Parallel computing

- MPI, is necessary for parallel computations. OpenMPI, MPICH2, Intel MPI or MPT (from SGI) must be installed in the system. VILJE has MPT installed. For installation at a local computer, OpenMPI is recommended. The library should be compiled with same compiler which will be used for building *Code_Saturne* in order to avoid possible compilation errors.
- ParMETIS/METIS, is a set of programs for partitioning graphs, partitioning finite element/volume meshes. METIS is serial and ParMETIS is MPI based parallel version. (http://glaros.dtc.umn.edu/gkhome/views/metis)
- PT-Scotch/Scotch, is another partitioning library. PT-Scotch works in parallel. (http://www.labri.fr/perso/pelegrin/scotch/scotch_en.html)

The last versions of these libraries can be downloaded from their official website and built the binaries through the system compiler (it should be the same compiler that is used for building *Code_Saturne*). However the binary versions which is built for suitable Linux version can be downloaded from SALOME's website (it requires a free registration): http://www.salome-platform.org/downloads/current-version

After downloading and installing the SALOME, the binaries of libxml2, CGNS, MED, HDF5, METIS and Scotch will be created. Those libraries will be used to built the *Code_Saturne* which will be described later in more detail.

Profiling tool: TAU

TAU(Tuning and Analysis Utilities) and PDT (Program Database Toolkit) tools should be installed in the system. TAU Performance System is a portable profiling and tracing toolkit for performance analysis of parallel programs written in Fortran, C, C++, UPC, Java, Python.TAU is capable of gathering performance information through instrumentation of functions, methods, basic blocks, and statements as well as event-based sampling. After a successful installation, TAU provides these scripts: tau_f90.sh, tau_cc.sh, and tau_cxx.sh to instrument and compile Fortran, C, and C++ programs respectively. PDTis used for insertion of instrumentation for performance profiling and tracing for TAU.

Compiling

The libraries and source file should be copied in a folder that is accessible as a user and be extracted from the source tar file. The following script is prepared as an example to configure the *Code_Saturne* by using TAU scripts `tau_cc.sh, tau_cxx.sh` and `tau_f90.sh`

```
module load intelcomp/14.0.1 mpt/2.09
SOURCE=/home/ntnu/user/code_saturne/intel_mpt/ptscotch/mk_hd_cg_io_omp/code
_saturne-3.0.1
TARGET=/home/ntnu/user/code_saturne
${SOURCE}/configure --
prefix=${TARGET}/intel_mpt/ptscotch/mk_hd_cg_io_omp/target_v3.2.2 \
--with-mpi \
--with-blas=/sw/sdev/Modules/intelcomp/14.0.1/mkl \
--with-zlib \
--with-med-include=/home/ntnu/user/med-3.0.6/include \
--with-med-lib=/home/ntnu/user/med-3.0.6/lib \
--with-hdf5=/home/ntnu/user/hdf5-1.8.10 \
--with-cgns-include=/home/ntnu/user/cgnslib-3.1.3/include \
--with-cgns-lib=/home/ntnu/user/cgnslib-3.1.3/lib \
--disable-gui \
--enable-mpi-io \
--enable-long-gnum \
--enable-openmp \
--with-scotch=/home/ntnu/user/scotch/6.0.0 \
CC=tau_cc.sh FC=tau_f90.sh CXX=tau_cxx.sh | tee config.log
```

At this script the Intel compiler which is used to compile for TAU/PDT and SGI's MPT for MPI should be loaded as a module. The configuration options should be `CC=tau_cc.shCXX=tau_cxx.sh FC=tau_f90.sh`.It is recommended to build the version in a folder with a name representing the configuration options. Several builts with different options help to try several types of configurations. The GUI is disabled without using 'libxml2' option. `'--enable-mpi-io'` and `'--enable-openmp'` options activate MPI-IO and OpenMP.`--with-blas`option activates the BLAS (Basic Linear Algebra Subprograms) librarieswhich a set of low-level kernel subroutines that perform common linear algebra operations. Intel MKL library is installed at VILJE. Building *Code_Saturne* with this option may affect the performance of the code. Partition method is selected as Scotch by `--with-scotch` option.

After running the script, the logs can be tracked the configuration output at the log_config.txt and config.log files which is the default output. After configuring the options, for to build the code, run the two scripts below respectively and follow the log files (`make.log,makeInstall.log`)

```
module load mpt/2.09
module load intelcomp/14.0.1
make CC=tau_cc.sh FC=tau_f90.sh CXX=tau_cxx.sh | tee make.log
module load mpt/2.09
module load intelcomp/14.0.1
make installCC=tau_cc.sh FC=tau_f90.sh CXX=tau_cxx.sh | tee makeInstall.log
```

After a successful compilation, the environment setting should be set in .bashrc file as follows:

```
PATH=${PATH}:/home/ntnu/user/code_saturne/ptscotch/mk_hd_cg_io_omp/target_v
3.2.2/bin/:$PATH
```

where the path is pointing the binary file of the *Code_Saturne* based on the target at configuration options. Now '*c*`ode_saturne`' is ready as a command.

Appendix B

The profile figure lists all the profiled functions for the case with 51 million cells executed over 1024 cores.