Available on-line at www.prace-ri.eu

Partnership for Advanced Computing in Europe

OpenMP Parallelization of the `Slilab` Code

Evghenii Gaburov[a], Minh Do-Quang[b], Lilit Axner[c]

[a]SURFsara, Science Park 140, 1098XG Amsterdam, the Netherlands
[b]Linné FLOW Centre, Mechanics Department, KTH, SE-100 44 Stockholm, Sweden
[c]KTH-PDC, SE-100 44 Stockholm, Sweden

Abstract

This white paper describes parallelization of the `Slilab` code with OpenMP for a shared-memory execution model when focusing on the multiphase phase flow simulations, such as fiber suspensions in turbulent channel flows. In such problems the motion of the "second phase - fibre" is frequently crossed over the distributed domain boundary of the "first phase - fluid", which in turn reduces the work-balance between the MPI ranks. The addition of OpenMP parallelization allows to minimize the number of MPI ranks in favor of a single-node parallelism, therefore mitigating MPI imbalance. With OpenMP parallelism in place, we also analyze performance of `Slilab` on Intel XeonPhi.

Project ID: `90631`

1. Introduction

The transport of particulate material by fluids is a problem with far-reaching consequences, and thus a long history of studies. When the particles are very small and neutrally buoyant, they tend to act as Lagrangian tracers and move with the local fluid velocity. Particles with a density greater than or less than the carrier fluid, however, tend to show different dynamics, such as preferential concentration and clustering in turbulent flow fields. Even particles that are neutrally buoyant can show dynamics different from the underlying flow when they are large compared to the smallest flow scales, since they filter the flow field in complex ways. Furthermore, fibers with a dipole charge, anionic or cationic, will also introduce a more complex motion than a normal fiber. In tissue manufacturing, a charge controller was used to control the fiber's attractive electrostatic forces, and to achieve optimum product properties and productivity. Recent experiments in Delft have shown that nylon fibers can significantly reduce the drag of water flow in a pipe. Therefore, understanding the phenomena occurring in the fiber suspension is essential to control industrial processes.

Numerical simulations provide access to the motion of the fibers along with the velocity of the carrier flow and allow a detailed study of the motion of fibers in complex flows. However, in the past they have been restricted to one way coupling, or using a drag model for the particle fluid interactions, or a statistical description of fiber ensembles. Experimental studies have not been able to access both fiber motion and fluid velocity in flows more complex than laminar shear flows. Several groups have studied orientation dynamics in flows with uniform velocity gradients, and investigated the effects of inertia, aspect ratio, and distance from solid boundaries.

Most existing models have neglected the direct interaction between fiber-fiber and fiber-wall. Some models take into account this phenomenon, but these models are still in a rudimentary form. In these models, when the fiber touches the wall or other fibers, it bounces off elastically, ignoring the change of linear and angular momentum before and after the interaction. However, Lee and Balachandar have shown that the angular slip velocity between particle and local fluid may give rise to a lift force perpendicular to the flow direction, which in turn changes the trajectory of the rotating particle. To achieve a complete understanding, a fully resolved direct numerical simulation (DNS) of fibers suspended in a turbulent flow is needed. But, fundamental phenomenological studies need the most accurate simulations on particle level. For example, there are 1million fibers in 1l of water when the concentration is 0.1%. Thus, in a simulation domain large enough for a fully developed turbulent channel flow, we need to have a number of fibers in the order of O(10000). Moreover, to achieve a fully resolved interaction model, the fiber surface also has to be fully resolved. Consequently, the total numbers of Lagrangian points grows dramatically, which explains why there are no fully resolved simulation of turbulent fiber flows reported so far.

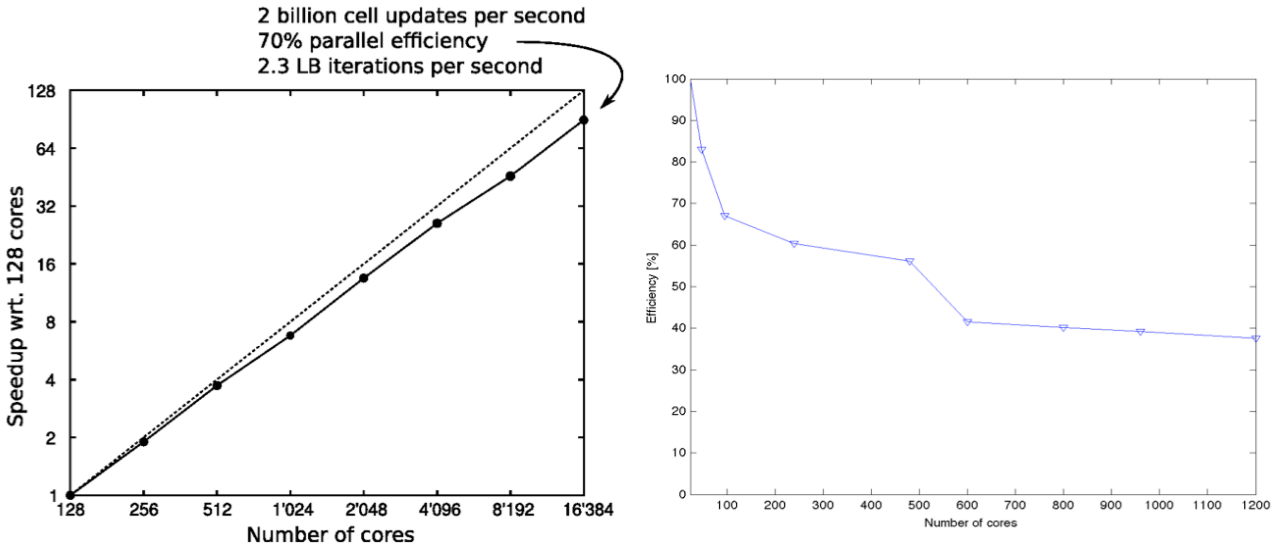

Fig. 1. Left panel: Speed-up of a single phase simulation performed on the CADMOS BG/P. A parallel efficiency of about 70% is obtained in the "strong" scaling case and on a span of two order of magnitude in the core count. Right panel: Parallel efficiency of `Slilab` code for two phase flows on the LINDGREN Cray XE6. The parallel efficiency is influenced by a relative size of the solid particles and the size of lattice partition.

2. Code description

The `Slilab` code is a Lattice-Boltzman method (LBM) code that has been developed at Mechanics department, Royal Institute of Technology, Sweden. This code is based on the `Palabos` library[1]–an open-source efficient parallel library based on the LBM. The `Palabos` library has been developed over many years, partly at the University of Geneva, and is proven to accurately simulate flow in complex geometries. A good parallel efficiency has been found on large-scale parallel computers, such as the Blue Gene/P (left panel of Fig. 1). Since the `Palabos` library is only parallelized with MPI, it has unsatisfactory parallel efficiency when used to simulate the two-phase flows.

To deal with the finite size particle - fluid interaction, `Slilab` used two state of the art techniques: 1) the node-base technique as the immersed boundary or the external boundary force method, 2) and the link-based technique as the lattice link bounce back method.

For the two phase flows, the parallel efficiency is influenced by the relative size of the solid particles and the lattice partition (right panel of the Fig. 1). Since most of the information is communicated only to the neighboring ranks, we have used `MPI_CART_CREATE` to map the ranks into a 3D grid topology. Therefore, the nearest-neighbor networks, such as Cray Gemini or BlueGene 3D torus interconnects, will help to improve the performance of the simulation. To ensure that the data used in a loop remains in the cache until it is reused, we divide the loop iteration into smaller blocks looping. Therefore, a computer with a bigger shared highest-level cache, usually referred to as the last level cache (LLC), is preferable for this code.

3. Parallelization with OpenMP

As described in the previous section, the `Slilab` code has a time-proven parallelization with MPI for distributed memory systems. However, modern CPUs are equipped with a dozen of cores on a single die, and it makes sense to take advantage of OpenMP parallelization to increase parallel efficiency of the application; for example, a single MPI ranks can be assigned to a single CPU, which will result in a tenfold reduction in the number of MPI ranks.

To parallelize `Slilab` code with OpenMP it is best to focus on a single MPI rank execution. Indeed, the MPI parallelization of `Slilab` has a satisfactory scaling to hundreds of nodes, each with a dozen of MPI ranks. Since addition of a new node adds computational resources (both flops and bandwidth) in the same proportion, it is natural to expect that if a single rank OpenMP version utilizes all resources provided by the CPU, then the resulting hybrid MPI+OpenMP code will maintain its linear scalability with MPI ranks.

The main hotspot function of the code, taking $p = 58\%$ of CPU time, was already parallelized with OpenMP in the original code. However, Amdahl's law dictates that with only this parallelization, the maximal speed-up with N processors will be $S(N) = 1/[(1-p) + p/N]$. Indeed, in the limit $S(N \to \infty) \approx 2.38$, which is much smaller than the number of available processing cores in a single node. Thus, we needed to identify additional serial parts of the code which require parallelization.

We profiled `Slilab` with the Intel VTune Amplifier to identify hotspot areas and to evaluate concurrency of the code. For this purpose we used GNU Compiler Collection 4.8.1, Intel MPI 4.1.3.048 and Intel VTune

[1] `http://www.palabos.org`

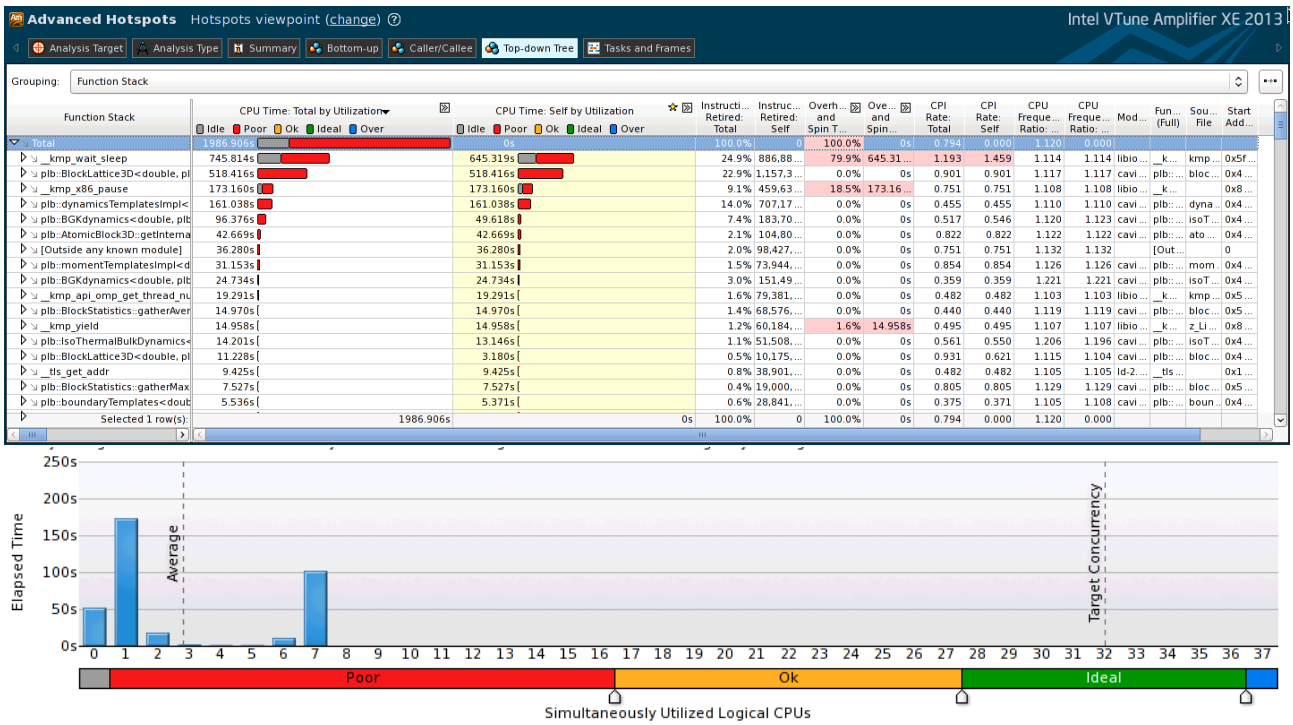

Fig. 2. Hotspot (top panel) and concurrency (bottom panel) analysis of the original `Slilab` code. The hotspot informations shows, by summing CPU time of `__kmp_wait_sleep` and `__kmp_x86_pause` functions, that CPU is idle nearly half of the time. The concurrency histogram represents breakdown of the application runtime, and each bar represents wall-clock time (vertical axis) spent in simultaneously utilizing a given number of cores (horizontal axis). The profiling data is assembled for application using 8 threads and a single socket, where each thread is pinned to a separate physical core.

Amplifier XE 2013 Update 15. We chose GNU Compiler 4.8.1 due to the superior performance of the generated code over Intel Compiler 14.0.2 on this application[2].

Unless mentioned otherwise, we used the following environment:

```
1 export OMP_SCHEDULE=static
2 export I_MPI_PIN=1
3 export I_MPI_PIN_DOMAIN=omp:compact
```

List 1. OpenMP and Intel MPI environmental variables used for SNB experiments.

The first variable declares that "`for`"-loops with "`#pragma omp for schedule(runtime)`" clause will use static scheduling. The second and third variables indicate that MPI ranks are placed in a compact manner while respecting OpenMP thread placement. An application with the following execution line "`OMP_NUM_THREADS=8 mpirun -np 4 ./a.out`" will schedule four MPI ranks next to each other, each with eight OpenMP threads. In particular, for a system with hyper-threading enabled (HT),the first MPI rank will use first four cores (0-3) with two threads per core due to HT, the 2nd MPI rank will use the second four cores (4-7), and the final MPI rank will use the last four cores (12-15), for a grand total 16 cores split into 4 MPI processes each with 8 OpenMP threads.

We show both hotspot (top panel) and concurrency histogram (bottom panels) in Fig. 2. From the hotspot information we observe that nearly 46% of the CPU time is idle. This is confirmed by the concurrency panel in which we see that nearly 2/3 of the time only one thread being used, which is obviously an inefficient use of CPU resources. The top panel in the Fig 2 provides us a list of functions which are the primary targets for the parallelization efforts.

Most of these functions consisted of loops that were easily parallelized with "`#pragma omp parallel for schedule(runtime) [collapse(2)]`" clauses. Some functions, however, made use of "`static`" variables to store data between function invocations. This generated race conditions, which were lifted by further promoting such "`static`" variables to become private for each threads. An "OpenMP"-way to achieve this was to use "`#pragma omp threadprivate`" clause for "`static`" variables. However, these variables were not of a plain-old-data type but rather C++ template classes. We found that while Intel C++ compiler [14.0.2] could digest such "thread private" declarations, the GNU C++ [4.8.1] compiler was refusing to compile the code by generating cryptic compilation errors. A work around for this was to use C++11 "`thread_local`" keyword with GNU C++ during variable declaration, which proved to work fine with OpenMP. The irony was that Intel C++ Compiler was unable to compile these declaration, and as a result we opted for a compile-time conditional rules, such that

[2] `gcc 4.8.1` performance was nearly 20% higher.

with Intel C++ compiler we used "`#pragma omp threadprivate`" clause after variable declaration, otherwise we took advantage of "`thread_local`" keyword during variable declaration. A code sample demonstrating this is shown below:

```
...
#ifdef __INTEL_COMPILER
#define THREAD_LOCAL
#define PARALLEL_MODE_OMP_INTEL_ONLY PARALLEL_MODE_OMP
#else
#define THREAD_LOCAL thread_local
#endif

template <typename Descriptor, plint index, plint value>
class SubIndex
{
private:
  SubIndex()
  {
    for (int iVel=0; iVel<Descriptor::q; ++iVel)
        if (Descriptor::c[iVel][index]==value)
            indices.push_back(iVel);
  }

  std::vector<plint> indices;

  template <typename Descriptor_, plint index_, plint value_>
  friend std::vector<plint> const& subIndex();
}

template <typename Descriptor, plint index, plint value>
std::vector<plint> const& subIndex()
{
  /* use "thread_local" keyword with non-Intel compilers */
  static THREAD_LOCAL SubIndex<Descriptor, index, value> subIndexSingleton;
#ifdef PARALLEL_MODE_OMP_INTEL_ONLY
  /* use "threadprivate" clause only with Intel C++ compiler */
#pragma omp threadprivate (subIndexSingleton)
#endif
  return subIndexSingleton.indices;
}
...
```

Finally, we also focussed on adding NUMA awareness to application. Without such, the code failed to scale beyond the single socket because data needed by CPU1, instead of being accessed directly, had to travel from CPU0 via QPI[3] which has much slower bandwidth compared to the direct main memory interface. The code had an allocator that used a single `new` C++ operator to allocate memory for the working data set, and such allocator also took care of data initialization. Unfortunately, since data initialization–first touch–was done by a single core, all memory where the data resided was mapped to the CPU0, and any access to this memory by other CPUs would have to travel via QPI from the CPU0. Considering that with two CPUs half of the data would not reside in the local memory of the other CPU, the resulting performance penalty could be quite substantial. To add NUMA awareness, we manually allocated memory via `malloc`, and then used parallel initialization as shown in the following code snippet:

```
#ifndef _NUMA_AWARE_
    rawData = new Cell<T,Descriptor> [nx*ny*nz];
#else /* use NUMA aware allocator */
    typedef Cell<T,Descriptor> cell_t;
    rawData = (cell_t*)malloc(sizeof(cell_t)*nx*ny*nz);
#ifdef PARALLEL_MODE_OMP
#pragma omp parallel for schedule(runtime) collapse(2)
#endif
    for (plint iX=0; iX<nx; ++iX)
      for (plint iY=0; iY<ny; ++iY)
        for (plint iZ=0; iZ<nz; ++iZ)
          rawData[iZ + nz*(iY + ny*iX)] = cell_t();
#endif
```

However, the degree of NUMA awareness of the application also depends on overall application parallelism. A mismatch between parallel strategies in the application and those of the NUMA-aware allocator would leave residual NUMA-effects that may hamper application performance. Nevertheless, this simple NUMA-aware initializer proved to be sufficient for dual- and quad-socket CPU configurations.

[3]QPI stand for Intel QuickPath Interconnect. Other CPU vendors may use different technologies for inter-CPU communication, such as AMD with HyperTransport.

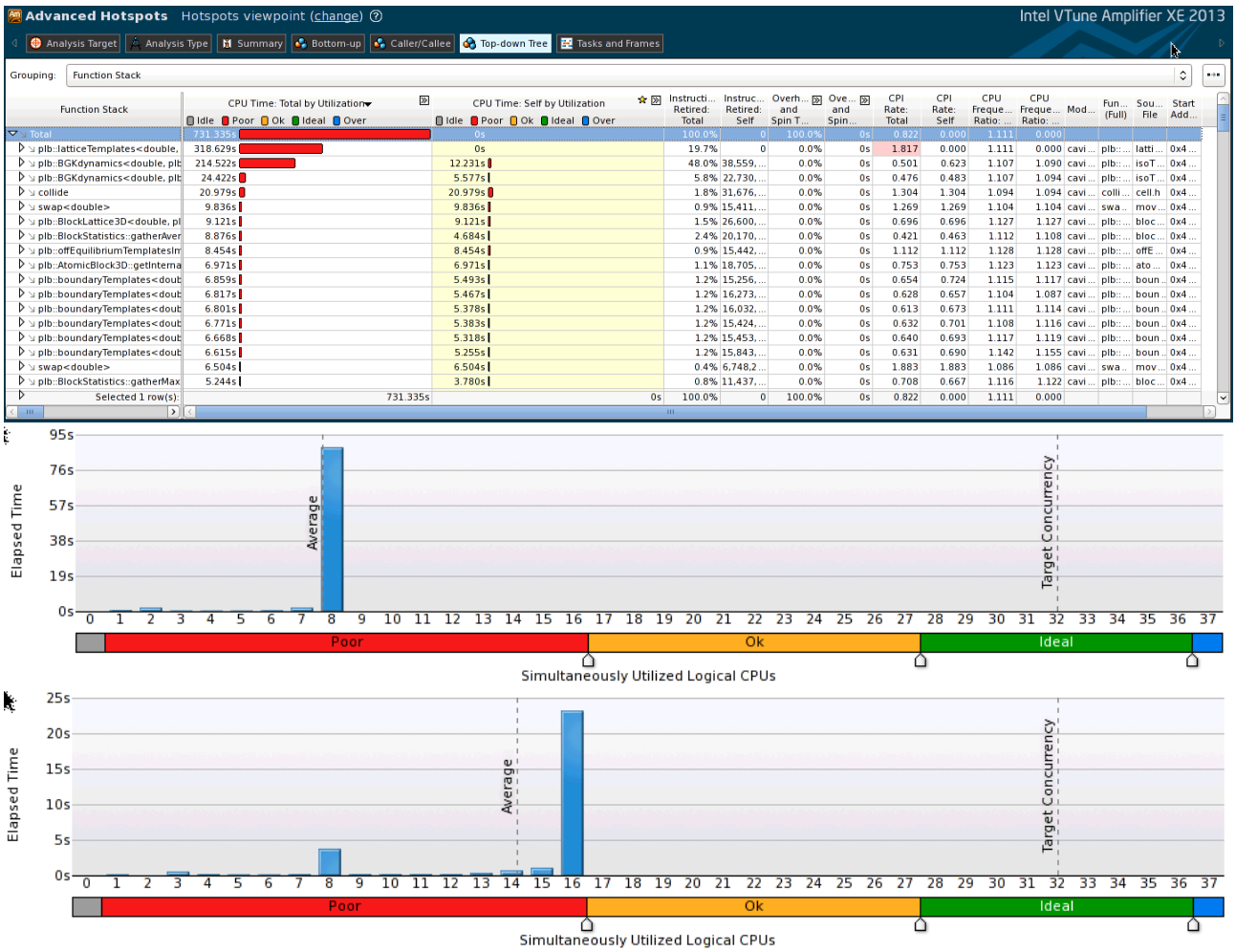

Fig. 3. Hotspot (the top panel) and concurrency (the bottom two panels) analysis of the OpenMP parallelized `Slilab` code. The description of each of the panel is similar to the one in Fig. 2, except that the very bottom panel shows concurrency histogram when using two sockets.

In Fig. 3 we show hotspot and concurrency data for the OpenMP parallel code, and in contrast to the old code, the CPU is active nearly 100% of the time. Indeed, the hotspot information doesn't show "`__kmp_wait_sleep`" and "`__kmp_x86_pause`" calls, during which processing cores are idle, and the bottom two panels show that the most of the time all cores available to application are active. These improvements in the application concurrency, result in a tangible reduction in the runtime: by comparing the total "Total" CPU time in the hotspot panel of Figs. 2 and 3, we see that the properly parallelized OpenMP version is 2.65x faster than the original code.

The bottom panel in the figure shows concurrency histogram when using two CPUs and 16 threads, with each thread being pinned to a separate physical core. There are two noticeable maxima, one at the 16 cores and a much smaller one at the 8 cores. The former implies that the code scales well from one to two sockets. However, the peak at the 8 cores is likely due to residual NUMA effect that is not addressed by our simple NUMA-aware allocator. While this causes a little performance impact on the dual- and quad-socket configurations with Intel Xeon CPUs, with larger number of NUMA domains, and especially if their relative distances becomes large enough, the performance damage from the residual NUMA effects may become worse.

4. Results

This section describes performance results of the OpenMP parallelization, in which we focus entirely on a single node scalability tests, either standalone or in a combination with MPI. Due to the bandwidth-bound nature of `Slilab` , it makes little sense to study its speed-up with the core count. The reasoning follows the fact that the memory bandwidth is a resource shared by all cores on a single socket[4]. In contrast to the floating-point operations per second (flops), which is a property of a core and therefore scales linearly with the core count, the addition of new cores from the same CPU above a certain threshold will not increase application bandwidth in the same proportion. As such there is no any fundamental reason to expect a linear speed-up with the core count on a single socket. However, an additional CPU socket provides the application with fresh memory bandwidth, thus it is natural to expect the application speed-up to increase linearly with the number of sockets. In addition

[4]At least for Intel CPU line at the time of writing.

to this scalability test, we also empirically measure the sustained application bandwidth and compared it with a practically achievable maximal value. This metric serves a dual purpose, a) as a measure of how efficiently the application utilizes available CPU resources, and b) as an estimate of how much (non-algorithmic) optimization potential remains in the application.

We used two different processor configurations for these experiments. The first configuration is a standard dual-socket workstation equipped with two Intel Xeon E5-2650L, with hyper-threading enabled and 32GB of four-channel 1600MHz DDR3 memory, to which we refer as SNB. The second configuration is a Intel XeonPhi 5110P co-processor, to which we refer as KNC. While originally the application was not intended to run efficiently on KNC, once the OpenMP parallelization was in place it was a simple task to compile the application for KNC in the native mode.

4.1. Performance on Intel Xeon

We used Stream benchmark[5] to measure maximal achievable bandwidth on SNB configuration. While the benchmark reports the bandwidth, we instead opted for `likwid` tool[6] for this purpose, mainly because we use this tool to measure sustained application bandwidth; this also serves as a kind of sanity test for the Stream benchmark and the `likwid` tool, which should report the same numbers within an acceptable confidence interval. By parsing the relevant output from "`likwid-perfctr -g MEM -t 300ms -c S0:0@S1:0`" command, we measure average bandwidth achieved during both the benchmark and the application runtime. Here, we focus on the Stream benchmark bandwidth with 16 threads placed on a CPU0, and with 32 threads equally distributed among both CPUs. In the former case, the achieved bandwidth was 35.7 GB/s, and in the latter case it was 69.2 GB/s. This shows a loss of about 3%, possibly due to QPI through which CPUs communicate with each other. Thus, for all practical intents and purposes we will be using 35.7GB/s and 69.2GB/s as a reference point for a maximal achievable bandwidth for our configuration[7].

To measure application performance we chose two representative problem sizes, a box with 256^3 and 128^3 cells. The smaller problem has 8x less work per core, which allowed to obtain better insight in the performance of hybrid parallelization strategies. Together with the environmental variables shown in List. 1 we used the following execution line:

```
1 OMP_NUM_THREADS=#threads mpirun -np #ranks x #S ./cavity3d.avx #size
```

where `#threads` is equal to the number of threads per MPI rank, `#ranks` is the number of MPI ranks per socket, `#S` is number of sockets in use, and `#size` is the problem size which is equal to either 256 or 128.

N_{ranks}	N_{threads}	1S [Mups]	BW [GB/s]	2S [Mups]	BW [GB/s]	Speed-up
1	16	43.2	29.6	77.4	55.1	1.79
2	8	41.6	29.3	77.8	57.7	1.87
4	4	40.8	30.1	77.8	58.6	1.91
8	2	40.8	30.7	76.3	61.0	1.87
16	1	39.7	31.8	73.9	62.2	1.86
1	8/8c*	37.0	24.9	66.4	46.1	1.80

Table 1. Performance of `Slilab` code on SNB using 256^3 problem size. The first and the second columns show the number of MPI ranks per socket and the number of threads per rank respectively. The third and fifth columns report sustained "science rate" of the application–the mega-updates per second [Mups]–for single and dual-socket runs respectively. The fourth and sixth columns show the sustained application bandwidth, which can be compared to a maximum of 35.7 and 69.2 GB/s for single and dual-socket runs. Finally, the last column shows speed-up from using two sockets. *The last line uses 8 threads with 8 cores per socket, which is equivalent to disabling hyper-threading.

The performance results for 256^3 problem are shown in Table 1. Here, the first and second columns correspond to `#threads` and `#ranks` respectively. The third and fifth columns report application science rate–the number of mega-updates per second [Mups]–for 1 socket and 2 socket configuration. In an application with, for example, 40 Mups, the average duration of an iteration is $256^3/(40 \cdot 10^6) \approx 0.42\,\text{sec}$. In other words, inverse of Mups scales linearly with the wall-clock time it takes for a single iteration update. The forth and sixth columns report the application performance in terms of the sustained application bandwidth. While such a number carries little relevance to the user of the application, when compared to the maximal achievable bandwidth this number gives a measure of how efficiently the application utilizes available bandwidth. In particular, we know that the maximal achievable bandwidth on such SNB configurations s 35.7 GB/s and 69.2 GB/s for single and dual-socket runs respectively. The two columns demonstrate that the application utilizes over 80% of this bandwidth, which is rather good. This means, the upside for further optimizations is rather limited to the remaining 20%, which on its own is unlikely to be achieved. Finally, the last column shows application speed-up from using two sockets, by doubling the number of MPI ranks. This table also indicates that the "sweet-performance-spot" is achieved with 4 threads per MPI ranks, and with 4 MPI ranks per socket for this

[5] `http://www.cs.virginia.edu/stream/`
[6] `https://code.google.com/p/likwid/`
[7] The peak bandwidth of our SNB configration is 51.2 GB/s per socket.

N_{ranks}	N_{threads}	1S [Mups]	BW [GB/s]	2S [Mups]	BW [GB/s]	Speed-up
1	16	35.1	27.8	63.0	40.7	1.79
2	8	36.1	27.0	67.5	45.9	1.87
4	4	37.2	30.0	76.2	45.2	2.05
8	2	34.2	25.6	63.8	51.1	1.86
16	1	35.2	28.1	66.8	53.0	1.90
1	8/8c*	32.8	20.4	55.8	36.2	1.70

Table 2. Performance of `Slilab` code on SNB using 128^3 problem size. The columns in this table carry the same meaning as the ones in Tab. 4.1.

particular run. It is also worth noticing in the last line in the table, that with enabled hyper-threading the science rate of the application increases by 16%.

The same exercise is repeated for 128^3 problem size, and the results are presented in the Table 2. In contrast to the previous case, we systematically observe both lower Mups and bandwidth. The primary reason is that with 8x less work, the "surface-to-volume" ratio of the problem increases, which results in a decrease of the relative fraction of the useful work. It was also interesting to observed that, similar to the previous case, the "sweet-performance-spot" is still obtained with 4 threads per rank. Furthermore, the speed-up when using two sockets is 2.05, which is above 1.93 we achieved with the Stream benchmark. Without detailed investigation we can only suspect this is due likely to the data being mostly able to fit in the last level lache, which is 20MB in size. Indeed, a quick look at memory utilization via "`top`" Linux command showed that the 128^3 problem requires 39MB of memory in total, thus confirming our suspicion. In addition, compared to the pure OpenMP and MPI runs, the hybrid "performance-sweet-spot" is over 14% faster, which demonstrates superiority of the hybrid parallelization strategies for small enough problems. Similar to the previous case, the small problem sizes also benefit from hyper-threading, albeit to a lesser degree.

4.2. Test-driving Intel XeonPhi

We demonstrated in the previous sections that the adopted OpenMP parallelization strategy was able to achieve good scalability and excellent concurrency on multiple cores. It is therefore tempting to test-drive parallelized `Slilab` code on KNC. It turned out that compiling `Slilab` with KNC as a target was a simple task: we only needed to add "`-mmic`" flag to the compilation and linking with Intel Compiler Suite, as well as to replace "`ar`" with "`xiar`" which allows to create static libraries with KNC object files.

Due to the code style choices, the Intel C++ compiler is unable to auto-vectorize the application (same applies to the compilation for CPU), which results in generation of a scalar code. However, due to the bandwidth-bound nature of the code the vectorization plays less important role compared to a compute-bound code. Therefore, this experiment can be viewed as an empirical test for the code scalability. Indeed, the Stream benchmark demonstrates that if bandwidth-bound code scales, it is able to sustain 100 GB/s on the Triad test with 224 threads/56 cores with vectorization disabled, which compares favorably to 136 GB/s on the same vectorized code. Thus, in theory, we should be able to get at least dual-socket SNB performance from XeonPhi. In practice, as we show below, a single XeonPhi is able to deliver performance similar to a single socket SNB.

We used the following environmental variables on KNC:

```
1 export OMP_SCHEDULE=static
2 export KMP_AFFINITY=compact,verbose
3 export KMP_PLACE_THREADS=56c4t
```

List 2. Intel OpenMP and MPI environmental variables

Here, the second variable tells Intel OpenMP implementation to place OpenMP threads in a compact manner. The last variable, tells the Intel OpenMP to use 56 core with 4 threads per core; we found inferior performance when using less than 4 threads per core.

After setting these environment variables, we ran the application XeonPhi with 256^3 problem using the following command "`./cavity3d.mic 256`". As with CPU, we profiled application with Intel VTune Amplifier, and the results are shown in the Fig. 4. From the figure we learn that the processor spends about 75% of the CPU time doing useful work, which is respectable for a code that has not been specifically tuned for XeonPhi. Indeed, the science rate of this particular run is nearly 32 Mups, which is comparable to a single socket Xeon. This reinforces the fact that the scalability is of the utmost importance in achieving a satisfactory performance on Xeon Phi. Nevertheless, this falls short of the expected two-socket performance, and there could be several reason for this. Without detailed inspection, we speculate that the lack of vectorization has a more pronounced impact on the application performance because the `Slilab` code has higher arithmetic intensity than the Stream benchmark. From the concurrency histogram we see that there is little left to gain from further improvement in parallelism, and the efforts must be concentrated in code refactoring that would make it suitable for auto-vectorization. Another potential issue is to recall that we found in CPU experiments that Intel C++ compiler generates 20% slower code compared to GNU C++ compiler. It is, therefore, not unreasonable to expect that the same performance penalty exists on Xeon Phi as well. Unfortunately, it is not possible to test this claim

N_{ranks}	N_{threads}	1 KNC [Mups]	2 KNC [Mups]	Speed-up
1	224	31.7	40.2	1.26
8	28	33.4	48.0	1.44
16	14	34.1	63.0	1.85
32	7	34.8	64.4	1.85
224	1	28.1	40.7	1.45

Table 3. Performance of `Slilab` code on XeonPhi 5110P using a grand total of 56 fully subscribed cores per device. The first column displays the number of MPI ranks per device, and the second column shows number of threads per MPI rank (compact thread scheduling is used). The third and fourth columns display code performance in mega-updates per second [Mups] when using one and two XeonPhi devices. Finally, the last column shows speed-up when using two XeonPhi cards.

since only Intel C++ compiler is capable to generate code for XeonPhi[8].

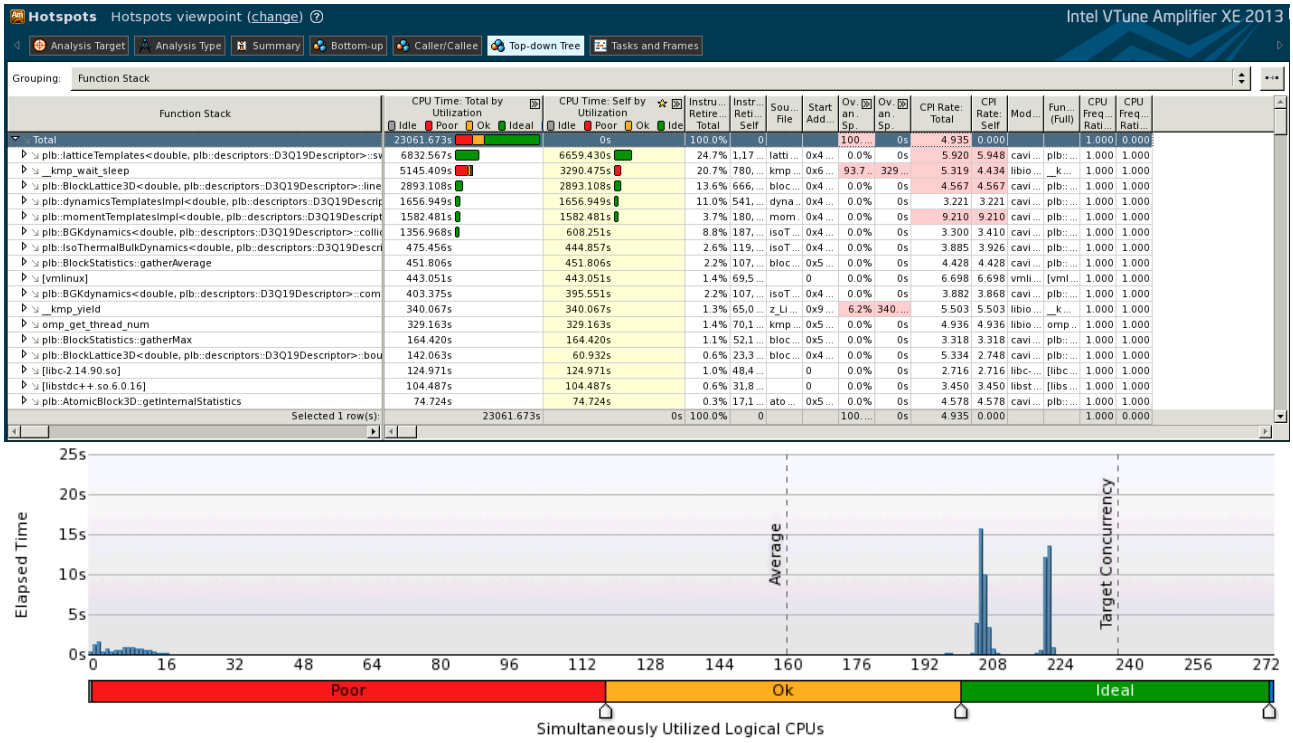

Fig. 4. Hotspot analysis (top panel) and concurrency histogram (bottom panel) of the OpenMP parallel `Slilab` code running on 56 XeonPhi cores with 4 threads per core, netting in a grand total of 224 threads. By counting the CPU Time spent in `__kmp_wait_sleep` and `__kmp_yield`, we see that the processor sits idle about 24% of the time. The concurrency histogram shows two strong peaks at at about 208 and 224 thread count, with a small tail of non-scalable code at the left. This shows that OpenMP code has a satisfactory scalability to 224 threads, but there is room for further improvement.

An interesting experiment can be made by taking the advantage of the hybrid parallelism. In particular, we split 224 threads is groups of N_{ranks} and N_{threads} threads per rank, such that the product of the two is equal to 224. The results of this experiment are shown in Table 3. The first two columns show the number of ranks per device, and the number of threads per rank respectively. The next two columns show `Slilab` performance in mega-updates per second [Mups] on one and two XeonPhi cards respectively. Finally, the last column shows the speed-up when using two devices. We didn't have `likwid` tool installed on XeonPhi, and therefore we were unable to measure the sustained application bandwidth. However, by using the correlation between the science rate and the sustained bandwidth from Table 1 , we estimated the sustained bandwidth for the best performance runs to be 30 and 55 GB/s when used with one and two XeonPhi cards respectively.

We observe that the "sweet-performance-spot" is achieved using 16 and 32 ranks per device with 14 and 7 threads per rank respectively. Furthermore, this rank-thread configuration also substantially improves speed-up when using two devices. There are several possible explanations for this behaviour, but without detailed investigation we can only venture a guess of what is the likely cause. We know from the Fig. 4, the pure OpenMP mode has an imbalance of about about 25%, which is due likely to serial parts in the application that perform poorly on XeonPhi. However, it is known that `Slilab` has good MPI scalability, and therefore by reducing number of OpenMP threads, while increasing the number of MPI ranks, we empirically located

[8]Intel has produced a `gcc` compiler capable of generating KNC code, but the compiler uses outdated and slow x87 instruction set for float point computations.

a "sweet-performance-spot" where OpenMP imbalance is minimal. The data from the Table 3 suggests that the imbalance is minimized with 14 to 7 threads per rank. The increase of the rank count allows a better load balancing, and when compounded with less imbalance due to the OpenMP in each rank, the speed-up on the two XeonPhi devices improves substantially. At the other end, by relying entirely on MPI for parallelism, we see the deterioration of the performance likely related to the fact that with 224 MPI ranks, the amount of work per rank becomes too small to hide MPI communication overhead.

Acknowledgements

This work was granted access to the HPC resources of SURFsara/Cartesius and Hybrid/CSC and EPCC/Archer and EPCC/Hector made available within the Distributed European Computing Initiative by the PRACE-2IP, receiving funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. RI-283493.