User co-scheduling for MPI + OpenMP applications using OpenMP semantics

Antoine CAPRA[1], Patrick CARRIBAULT[2], Marc PÉRACHE[2], and Julien JAEGER[2]

[1] ParaTools SAS, Bruyeres-le-Chatel, France,
capra@paratools.com,
[2] CEA, DAM, DIF, F-91297 Arpajon, France
{patrick.carribault,marc.perache,julien.jaeger}@cea.fr

Abstract. Recent evolutions in parallel architectures such as manycore processors are putting an end to the pure-MPI model. Simulations codes willing to productively use current and future supercomputers are bound to expose multiple levels of parallelisms inside and between nodes, combining different programming models (e.g., MPI+X). In this paper, we propose to discuss this evolution in the context of MPI+OpenMP which is a common hybridization approach. In particular, methods leveraging the OpenMP tasking constructs are presented in such hybrid context. Various approaches are discussed and compared considering codes trying to mix fine-grained computation and communications, taking advantage of recent evolutions in the OpenMP standard. Advantages and limitations of the approaches are detailed, including potential improvements to OpenMP in order ease both the integration and progress of MPI calls. These results are applied to a representative stencil code demonstrating improvements on the overall execution time thanks to an efficient mixing of MPI and OpenMP.

1 Introduction

Parallel scientific applications are designed to take advantage of the resources they are provided with. When considering current architectures, optimization spectrum is wide, ranging from vectorization at a core level to distributed operations involving millions of cores. Many-core processors leading to larger nodes and less memory per threads are now putting an increasing pressure on the pure MPI model[5], memory replication and communication overhead becoming impractical. Hybridization which consists in combining a shared-memory model with a distributed-memory one has now become a compulsory avenue when it comes to writing efficient parallel code. When considering MPI+X, OpenMP has become the de-facto standard. Moreover, legacy aspects when hybridizing existing programs have led to codes separating MPI and OpenMP aspects, most of the time an MPI code being adapted (often at loop level) to exhibit intra-node parallelism[6]. However, despite its clear practical aspect, this approach encounters limitations which will eventually prevent the program from scaling.

Indeed, by separating MPI and OpenMP phases, parallelism is generally bulk-synchronous, alternating between communication and computation phases. In such model, communications are done by a single thread, creating a loss of parallelism combined with extra fork-join overheads.

In this paper, we propose to mix MPI and OpenMP models, considering requirements originating from both runtimes. Our purpose is to question up to which point the OpenMP standard and the features it introduced in its last revisions provides facilities allowing the expression of programs invoking MPI functions inside parallel regions. How a program written with MPI+X in mind can express fine-grained parallelism and communication through OpenMP, in particular using tasks. After presenting hybrid tasking patterns, we show how this approach can be beneficial in particular by mitigating the bulk-synchronous nature of most MPI+OpenMP applications. In this process, we observed some limitations in existing OpenMP runtimes in terms of support, points which are used to propose some extensions to OpenMP oriented towards runtime stacking.

In order to express fine-grained parallelism, we consider the task-based model. This approach allows the expression of both MPI and computation phases. Iterations are seen as a directed graph mixing MPI and compute tasks. Tasks are vertexes in the graphs and edges dependencies between tasks. This leads to the expression of an MPI+OpenMP program as a Directed Acyclic Graph (DAG). We focus ourselves on the critical path arising in such graph when dealing with stencil-based computations, including spatial dependencies. In particular, our goal is to reduce the coupling arising from communications between distributed memory regions, doing as soon as possible parts of the computation which dependencies were satisfied. In this formalism, MPI tasks are the one leading to the highest parallel overhead, possibly delaying computation. We, therefore, show how the tasking patterns we introduce mitigates communication impact, giving a higher priority to MPI tasks and splitting computational border into multiple regions – eventually moving communications inside the parallel region.

In the rest of the paper, we first describe task-support in the context of OpenMP runtimes and discuss how it is beneficial to the expression of hybrid computation. Then, after exploring various alternatives, we present an approach leveraging OpenMP tasks with dependencies to mix MPI and computation. The approach is then validated using the stencil benchmark, demonstrating the impact of communication progress on the overlap. Eventually, after presenting related work, potentials improvements to the OpenMP standard for model mixing are presented in the light of the tasking model previously introduced. The paper then concludes with prospects linked to this work.

2 OpenMP Tasking

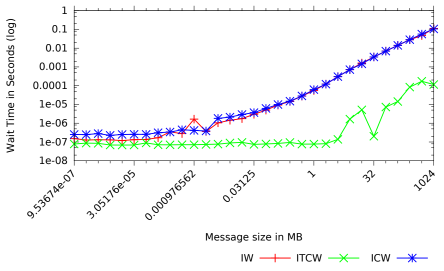
From a loop-oriented approach in its first implementations, the OpenMP standard has drastically evolved to propose and increasing variety of parallel constructs. Despite mostly used to parallelize loops in a transitional manner from an existing code, new constructions such as tasks are being considered by parallel programmers to improve the scalability of their applications.

One of the main drawbacks of the parallel loop approach is that it generally leaves sequential sections (in terms of shared-memory) inside parallel applications. Some loops may not be easily parallelized, exhibiting too complex dependencies, relying on external sequential libraries or simply not being parallelized yet. The progressive nature of the OpenMP parallelization being here both a strength and a weakness – a partially hybridized code underusing parallel resources. This point can be outlined by the well known Amdahl Law which states that the sequential part of a parallel code bound its strong scaling speedup, for example, if 20% of the code is sequential, maximum speedup is 5. It is then crucial to consider how OpenMP parallelization can be expressed in a manner mitigating the sequential part. Point that we discuss in this paper when considering future codes written with both OpenMP and MPI constraints in mind. OpenMP in its first version was not providing any task support, it was only at Version 3 that they were introduced. At this point, `task` and `taskwait` are defined and the `barrier` is a scheduling point for tied tasks. OpenMP 4 introduced `taskgroups`, dependencies with `depend` and removed implementation defined scheduling points for untied tasks. Eventually, the last version of the standard OpenMP 4.5 introduced `taskloop` and priorities. Consequently, since its definition, OpenMP has been improving its support for tasks which do provide a fine-grained way of defining parallel execution, functionalities that we will use in this paper to efficiently mix MPI and OpenMP.

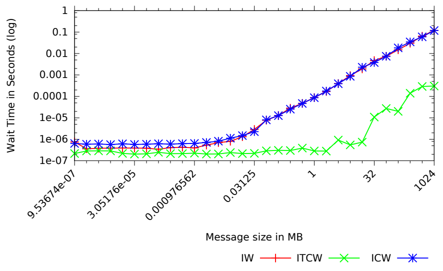
3 MPI+X alternatives

When mixing MPI and OpenMP one crucial aspect is how runtimes are going to mix together. One of the main interest of this hybridization is to be able to recover communications with computing in a more efficient manner. To do so, we first propose to measure communication duration in three different MPI scenarios. First, by emitting an asynchronous receive and directly waiting for it (IW). Second, Irecv then a parallel region where one OpenMP thread does Test calls while the others do the computation followed by wait (ITCW). And in the third case, by emitting the Irecv followed by a 500 μs computation and a wait (ICW). In this last case, we made sure that the overall computing had the same duration than in the second case – to allow direct comparison.

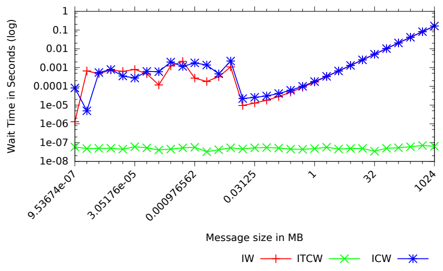
Looking at Figure 1.4, it is clear that if `MPI_Test` is not called, the MPI runtime does not take full advantage of the asynchronism, in this particular case, not directly waiting can be worse for small messages and comparable for larger ones. This is of course highly dependent on the underlying network. The reason why we repeated the measurement on top of Infiniband in an inter-node configuration. These results are presented in Figure 1(c), it can be seen that the same pattern than over the shared-memory segment arises, progress being compulsory for both MPICH and OpenMPI to take advantage of the overlap. Getting back to OpenMP integration this poses a problem. Indeed, it is not



(a) OpenMPI 2.1.0 (SHM)



(b) MPICH 3.3a2 (SHM)



(c) OpenMPI 1.8.8 (IB)

Fig. 1. Comparing our progress scenarios when running over both a shared memory segment and an Infiniband network (averaged 1000 times).

sufficient to emit communication and then wait for them, progress is needed to achieve good performance in a heterogeneous computation context. The rest of this Section is then dedicated to the expression of such progress inside OpenMP tasks.

```
  for( i = 0; i < MPI_COMM_NUM; i++ ){
   if( ! Atomic_load_int( tab_flags[i] ) ){
    MPI_Test( &(tab_reqs[i]), &mpiflag, MPI_STATUS_IGNORE );
     if(mpiflag && !Atomic_cas_int( &(tab_flag[i]), 0, 1 )){
       Atomic_incr_int(mpi_comm_complete )
       compute_ghost_associed_part(i);
     }
    }
  }
  finished = compute_core_part(); // yield function
}
```

Listing 1.1. MPI AWARE Select (loop splitting)

Our goal in this section is to progressively insert MPI calls inside the parallel region, this while accounting for the requirement of progressing the MPI runtime. To do so, we iteratively present our tasking patterns using an increasing number of functionalities. The extension of the OpenMP standard will allow us to submit an increasingly complete DAG of execution and thus to prioritize more effectively the tasks carrying out MPI actions.

If we had to insert MPI calls using OpenMP 2.0, the loop computing the core would be separated. Then border communications would be progresses using MPI_Test, and associated border computation triggered on completion. Then if communications have not competed yet, the core calculation can be used to recover communications. In order to extend this MPI query polling in the `MPI_THREAD_MULTIPLE` case, we have based our selection on the basis of an atomic value table. The calculation phase ends when all the MPI communications and the associated actions are realized: Computation of the border and MPI_Isend but also the core part. This process is illustrated in Listing 1.1. In this case, the execution path is constrained according to MPI dependencies. However, two computing functions are effectively parallelized internally at the price of a critical section choosing the next action based on communication completion. This reduced the potential overhead of MPI communications by constraining OpenMP behavior. Indeed, to be able to improve granularity, the core compute function would have to be chunked, in order to regularly progress and check communication dependencies. This code is, in fact, doing different kinds of tasks, encouraging us to rely on OpenMP tasks.

```
#pragma omp parallel
{
  #pragma omp for nowait
  for (i = 0; i < CORE_PART_NUM; i++)
  #pragma omp task
    compute_core_part(i);

  #pragma omp single nowait
  {
    while(mpi_comm_complete != MPI_COMM_NUM)
    {
      for( i = 0; i < MPI_COMM_NUM; i++ )
        if( ! Atomic_load_int( tab_flags[i] ) )
          #pragma omp task
            __progress_mpi_comm( i );
      #pragma omp taskyield
    }
  }
}
```

Listing 1.2. MPI AWARE Select (standard task)

As shown in Listing 1.2, an OpenMP task is dedicated to the progress of MPI communications. Moreover, thanks to the `taskyield`, MPI-related tasks are at most the number of MPI requests not completed. Dedicating a core (or task according to the OpenMP3.0 standard) to communications progression does not necessarily induce a penalty for the user code. For example, when considering architectures with a large number of cores such as the Intel KNL with 68 cores (272 hyper-threads). A hyper-thread, therefore, corresponds to 0.4% of a KNL – a totally acceptable overhead. As we can't modify the task scheduler, MPI progress will not be multi-threaded or prioritized. In most OpenMP implementations, an OpenMP thread performs its own tasks before stealing from other threads. In our scenario, stealing of communication tasks will only occur when a thread will have completed its own tasks – actually yielding the desired behavior. In this configuration without priority, only the stealing mechanism can give us a form of priority. When running this code, GOMP did not allow the `taskyield` construct. As far as Intel OpenMP is concerned, it was not providing expected performance gains. When waiting for communication we expected to schedule computing-related tasks. These runtime limitations required us to explore another task approach presented below in order to correctly progress communications.

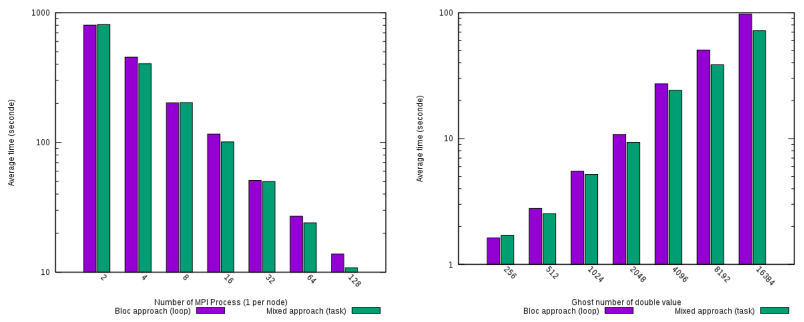
```
#pragma omp taskgroup
{
#pragma omp for nowait
for (i = 0; i < CORE_PART_NUM; i++)
#pragma omp task priority(1)
compute_core_part(i);

#pragma omp single nowait
{
  for( i = 0; i < MPI_COMM_NUM; i++ ){
#pragma omp task depend(inout: mpi_req ...) priority(100)
    {
      while( __mpi_request_not_match() )
      #pragma omp taskyield
    }
    if( i > MPI_COMM_SEND_NUM ){ // RECV REQUEST
#pragma omp task depend(inout:req_mpi ...) priority(100)
      {
        __compute_border_associate( i )
      }
#pragma omp task depend(inout: req_mpi ...) priority(100)
      {
        __send_ghost_associate( i );
      }
    }
  } // taskgroup
```

Listing 1.3. MPI AWARE Select (standard task)

Our initial idea was to rely on priorities and dependencies to pre-post MPI actions. To do so, valid and computable dependencies are required at compilation time. This leads to a problem when considering communications, a given MPI process may have a varying neighboring (mesh corners) while these dependencies have to be known at compilation time (no dynamic dependencies). In our example, `MPI_Requests` are static variables. Aware of `taskyield` limitations, we proposed in Listing 1.3, a version based on `single`, allowing us to force a thread to poll MPI Request, the `taskgroup` ensures that all threads participate in the execution of the sets of tasks, including the one testing for MPI communications. Eventually, the send task has two dependencies, ensuring that the previous send is complete before issuing the next.

4 Evaluation

In this evaluation Section, we demonstrate the interest of tasks when compared to loops. By avoiding successive fork-join, tasks are able to improve the overall scheduling. Our reference benchmark relies on a 1D splitting model over a $10e6$ double array evenly split between MPI processes. The ghost-cell array consists of 4096 doubles for each side with a periodic condition on the borders. In the loop-based version, Isend/Irecv are posted, the core part is computed, communications waited and then borders processed. In the task-based version, tasks are pushed immediately when the progress thread completes a test. We, therefore, end with a single parallel region with a computation split in tasks.



(a) Over MPI Processes (b) Over Ghost size (64 Processes)

Fig. 2. Comparison of our bulk-synchronous (loop parallelism) and our proposed task-based approach over both process count and message size (fixed at 64 MPI processes).

We ran our test on an Intel Sandy-bridge machine up to 2048 cores. Each dual-socket node gathers 16 cores on which we ran 16 OpenMP threads. In order to generate the results presented in Figure2, we realized 1000 timesteps and the execution time has been averaged 10 times. We observe that in this first case the task approach is better than the loop one when the number of nodes is higher than 16, this despite one core is used to progress communications. This is due to the increasing noise in MPI messages, creating irregularities in the communication scheme. Moreover, as the number of cores increases, the overall computation decreases (strong scaling), outlining this communication jitter. As far as the MPI overlap, we observed that is was almost null with the runtime that we used on the target machine (OpenMPI 1.8.8), justifying our efforts to integrate progress inside our parallel OpenMP constructs. Doing so, as shown in

Figure 2(b), provides performance gains increasing with overall communications size – demonstrating the importance of progressing MPI messages.

5 Progress and OpenMP

As we have seen in the previous section, not progressing MPI communication decreases performance. Consequently, in order to take advantage of asynchronous messages withing an MPI+OpenMP program, communications must be explicitly progressed through MPI runtime calls (Test, Probe). Not doing so shifts most of the message completion to the actual Wait operation (in the configurations we measured), removing any benefits from recovering communications. Acknowledging this issue, in this paper, we proposed task-based constructions overcoming this limitation. However, additional constructs in OpenMP may help solving this progress issue and more generally runtime stacking.

```
void idle( void *prequest ) {
  if(__mpi_request_match(prequest)){
    omp_trigger("ghost_done");
    return 0;
  }
  return 1;
}

#pragma omp parallel progress(idle, &request)
{
    #pragma omp noprogess
    {
        MPI_Wait(request, MPI_STATUS_IGNORE);
        omp_trigger("ghost_done");
    }
    #pragma omp task depends(inout:"ghost_done")
    {/* BORDER */}
    #pragma omp task
    {/* CENTER */}
}
```

Listing 1.4. Proposed implementation for a progress enabled OpenMP

As presented in Listing 1.4, OpenMP would gain in being enhanced with the notion of progress. Indeed, one could define what processing has to be done to satisfy task dependencies, letting the runtime invoke the `progress` function to trigger dependencies. In order to achieve this, two things are needed. First, the progress parameter of the parallel region, it defines which function should be called when the runtime is idle or switching between tasks. This is a function as it contains code which may not be executed if not compiled with OpenMP support if this function returns "0" it is not called anymore if it is "1" it continues to be called as there is work remaining. In this case, the otherwise ignored `noprogress` code section is executed, replacing the non-blocking progress calls with blocking ones. Eventually, we need *named dependencies* between tasks, this as we want another runtime to satisfy a dependency which cannot be known at compilation as an address, for example, *" ghost_done"*. To do so, we define `omp_trigger` which satisfies a named dependency. Using this simple construct, we are then able to express in a compact manner, a communication dependency with a direct fallback to a blocking version if OpenMP is not present. We are in the process of implementing this semantic to validate this abstraction which seems valid according to the experiments of this paper.

6 Related work

Architectures evolutions are putting an increasing pressure on the pure MPI model[5]. In this context, codes have to evolve in order to expose multiple levels of parallelism, inside and between nodes. This naturally led to the programming model mixing issue also called hybridization.

The first kind of hybridization appeared with accelerators such as GPUs addressed through Cuda or OpenCL[17]. This device, hosted by a regular node with its own cores have their own memory which is either remotely accessible or mapped in the host memory space. These devices are generally energy efficient and expose a high level of stream oriented parallelism. One issue with their use is the need for transfers to and from the device, requiring important programming efforts to manage data. Moreover, CPU resources might be underused if only used to move data, possibly outlining a need for another parallelism level. Pragma-based models such as OpenMP target[3,4], OpenACC[19], Xkaapi[11] and StarPU[2] proposed abstractions combining GPUs and CPUs in an efficient manner, abstracting data-movements.

MPI which is also an important component for large scale simulation did not reach such level of integration in shared-memory computations. Probably as the MPI process is the container for the shared computation, this combined with the fact that most programs evolved from MPI to handle new parallelism models[6]. For these practical reasons, there were fewer efforts to embed MPI in another model (e.g. X+MPI), but on the contrary to express parallelize inside MPI processes (MPI+X). The advent of many-core architectures first as accelerators such as the Intel MIC[9] first used through the OpenMP target directive and then as actual processors (with the Intel KNL) enforced the use of larger shared-memory contexts, requiring a form of hybridization. MPI + OPenMP is nowadays accounting for a large number of applications, nonetheless, neither MPI or OpenMP have collaborative behaviors, both of them are distinct runtimes with their respective ABI/API – point discussed in this paper. However, there are several programming models aimed at providing an unified view of heterogeneous or distributed architectures Coarray Fortran[16], Charm++[13], HPF[15], or languages such as Chapel[7], Fortress[1] or X10[8]. This, by leveraging various communication models, Message Passing, Partitioned and Global address space (respectively PGAS and GAS). Eventually, a complementary approach is based on Domain Specific Languages[10] aimed at abstracting parallelism expression[12] in order to "free" codes from programming model constraints, for example by targeting multiple models[18]. There is a wide range of such specialized languages with clear advantages, however, they transpose the dependency from a model to a dedicated language with its own constraints[14].

Model mixing and unified models, in general, is then a very active research area with a wide variety of approaches. In this paper we focused ourselves on two common building blocks, MPI and OpenMP, trying to see how MPI could be embedded inside OpenMP constructs in an efficient manner – taking a reverse approach when most efforts tend to embed OpenMP inside MPI.

7 Conclusion

In this paper, we first introduced the need for hybridization in parallel applications. Indeed, when scaling multiple nodes gathering hundreds of cores, the MPI+X paradigm becomes a compulsory approach to limit both memory and communication overhead. Then, after arguing that MPI+OpenMP codes generally relied on alternating phases between communication and compute, we explored alternative approaches relying on tasks. Indeed, from a node level point of view, the MPI phase becomes sequential if done outside of a parallel region. This induces a fork-join overhead and also puts an upper bound on the overall application speedup, the MPI part not being parallelized on all the computation units. Starting from this observation, the paper posed the question of how MPI could be included inside the parallel region for stencil-based applications with a relatively regular behavior. To do so, the paper relied on a representative mesh-based benchmark to implement three different approaches taking advantages of OpenMP tasks to mix MPI and OpenMP within a single parallel region.

The side effect of this approach is that the overall time-step becomes a Directed Acyclic Graph (DAG) which has to be scheduled by the OpenMP runtime. Such DAG made of tasks combines processing from both OpenMP (computation) and MPI (communication). However, such combination is not natural in OpenMP, particularly when considering `MPI_Request` handles which are generated dynamically during the execution. Indeed, OpenMP does not allow tasks dependencies to be expressed on the fly, instead, they have to be resolvable at compilation time. Consequently, in this paper, we proposed three different approaches to mixing OpenMP tasks and MPI despite this static dependency resolution. Approache that we both described and benchmarked in the context of our mesh-based benchmark. These measurements showed that the task-based approach was beneficial to the overall execution, in particular by allowing halo-cells to be processed immediately after communication completion, and also by limiting fork-joins which were present at each for loop in the initial implementation. However, as described in Section 3, we have seen that MPI progress became an issue, MPI runtimes working on communications only inside `MPI_Test` and `MPI_Wait` calls. This means that an efficient overlapping can only be achieved if MPI calls are regularly invoked during the computation as in our task-based examples.

8 Future Work

The paper showed that there are constructions favoring MPI+OpenMP, for example by expressing the computation as a DAG. However, the progress needs arising from this runtime stacking leads to a overlap problems which had to be circumvented using periodic tasks. This shows that an efficient use of OpenMP+MPI will need some interactions between runtimes in order to coordinate progress. For example, if the `taskyield` call could be defined as an arbitrary function, it would be possible for the OpenMP runtime to notify the MPI runtime that it may progress communications. This progress issue solved, dynamic (on request addresses) or label-based dependencies would be an alternative to the jump-table we relied on in this paper. This shows that there are side effects when combining two runtimes, necessarily requiring a form of interaction – need that was outlined in this paper.

References

1. Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.W., Ryu, S., Steele, G.L., Tobin-Hochstadt, S.: The Fortress Language Specification. Tech. rep., Sun Microsystems, Inc. (March 2008), version 1.0
2. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures, pp. 863–874. Springer Berlin Heidelberg, Berlin, Heidelberg (2009), `http://dx.doi.org/10.1007/978-3-642-03869-3_80`
3. Ayguade, E., Badia, R.M., Cabrera, D., Duran, A., Gonzalez, M., Igual, F., Jimenez, D., Labarta, J., Martorell, X., Mayo, R., Perez, J.M., Quintana-Ortí, E.S.: A Proposal to Extend the OpenMP Tasking Model for Heterogeneous Architectures, pp. 154–167. Springer Berlin Heidelberg, Berlin, Heidelberg (2009), `http://dx.doi.org/10.1007/978-3-642-02303-3_13`
4. Bertolli, C., Antao, S.F., Eichenberger, A.E., O'Brien, K., Sura, Z., Jacob, A.C., Chen, T., Sallenave, O.: Coordinating gpu threads for openmp 4.0 in llvm. In: Proceedings of the 2014 LLVM Compiler Infrastructure in HPC. pp. 12–21. LLVM-HPC '14, IEEE Press, Piscataway, NJ, USA (2014), `http://dx.doi.org/10.1109/LLVM-HPC.2014.10`
5. Besnard, J.B., Malony, A., Shende, S., Pérache, M., Carribault, P., Jaeger, J.: An mpi halo-cell implementation for zero-copy abstraction. In: Proceedings of the 22Nd European MPI Users' Group Meeting. pp. 3:1–3:9. EuroMPI '15, ACM, New York, NY, USA (2015), `http://doi.acm.org/10.1145/2802658.2802669`
6. Brunst, H., Mohr, B.: Performance Analysis of Large-Scale OpenMP and Hybrid MPI/OpenMP Applications with Vampir NG, pp. 5–14. Springer Berlin Heidelberg, Berlin, Heidelberg (2008), `http://dx.doi.org/10.1007/978-3-540-68555-5_1`
7. Chamberlain, B., Callahan, D., Zima, H.: Parallel programmability and the chapel language. The International Journal of High Performance Computing Applications 21(3), 291–312 (2007), `http://dx.doi.org/10.1177/1094342007078442`
8. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: An object-oriented approach to non-uniform cluster computing. SIGPLAN Not. 40(10), 519–538 (Oct 2005), `http://doi.acm.org/10.1145/1103845.1094852`

9. Duran, A., Klemm, M.: The intel many integrated core architecture. In: 2012 International Conference on High Performance Computing Simulation (HPCS). pp. 365–366 (July 2012)
10. Fowler, M.: Domain-specific languages. Pearson Education (2010)
11. Gautier, T., Lima, J.V.F., Maillard, N., Raffin, B.: Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing. pp. 1299–1308 (May 2013)
12. Hamidouche, K., Falcou, J., Etiemble, D.: Hybrid bulk synchronous parallelism library for clustered smp architectures. In: Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications. pp. 55–62. HLPP '10, ACM, New York, NY, USA (2010), `http://doi.acm.org/10.1145/1863482.1863494`
13. Kale, L.V., Krishnan, S.: Charm++: A portable concurrent object oriented system based on c++. SIGPLAN Not. 28(10), 91–108 (Oct 1993), `http://doi.acm.org/10.1145/167962.165874`
14. Karlin, I., Bhatele, A., Keasler, J., Chamberlain, B.L., Cohen, J., Devito, Z., Haque, R., Laney, D., Luke, E., Wang, F., Richards, D., Schulz, M., Still, C.H.: Exploring traditional and emerging parallel programming models using a proxy application. In: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing. pp. 919–932 (May 2013)
15. Loveman, D.B.: High performance fortran. IEEE Parallel Distributed Technology: Systems Applications 1(1), 25–42 (Feb 1993)
16. Numrich, R.W., Reid, J.: Co-array fortran for parallel programming. SIGPLAN Fortran Forum 17(2), 1–31 (Aug 1998), `http://doi.acm.org/10.1145/289918.289920`
17. Stone, J.E., Gohara, D., Shi, G.: Opencl: A parallel programming standard for heterogeneous computing systems. Computing in Science Engineering 12(3), 66–73 (May 2010)
18. Sujeeth, A.K., Rompf, T., Brown, K.J., Lee, H., Chafi, H., Popic, V., Wu, M., Prokopec, A., Jovanovic, V., Odersky, M., Olukotun, K.: Composition and Reuse with Compiled Domain-Specific Languages, pp. 52–78. Springer Berlin Heidelberg, Berlin, Heidelberg (2013), `http://dx.doi.org/10.1007/978-3-642-39038-8_3`
19. Wienke, S., Springer, P., Terboven, C., an Mey, D.: OpenACC — First Experiences with Real-World Applications, pp. 859–870. Springer Berlin Heidelberg, Berlin, Heidelberg (2012), `http://dx.doi.org/10.1007/978-3-642-32820-6_85`