Exact Dependence Analysis for Increased Communication Overlap ⋆

Simone Pellegrini[1], Torsten Hoefler[2,3], and Thomas Fahringer[1]

[1] University of Innsbruck – DPS at Innsbruck, Austria,
{spellegrini,tf}@dps.uibk.ac.at
[2] University of Illinois at Urbana-Champaign, IL, USA,
htor@illinois.edu
[3] Department of Computer Science, ETH Zurich, Switzerland
htor@inf.ethz.ch

Abstract. MPI programs are often challenged to scale up to several million cores. In doing so, the programmer tunes every aspect of the application code. However, for large applications, this is often not practical and expensive tracing tools and post-mortem analysis are employed to guide the tuning efforts finding hot-spots and performance bottlenecks. In this paper we revive the use of compiler analysis techniques to automatically unveil opportunities for communication/computation overlap using the result of exact data dependence analysis provided by the polyhedral model. We apply our technique to a 5-point stencil code showing performance improvements up to 28% using 512 cores.

Keywords: Message passing, Compiler Analysis, Data Dependence Analysis, Polyhedral Model

1 Introduction

The Message Passing Interface (MPI [12]) Standard defines a distributed memory library interface for use in performance-critical environments such as High Performance Computing (HPC). One of its main strengths is that the interface spans several abstraction layers, from very low level constructs (e.g., point-to-point messaging or simple one sided accesses) to high level performance-portable functionality (e.g., collective operations or derived datatypes). Highly optimized implementations exist for several supercomputer architectures and interconnects (e.g., Myrinet, InfiniBand). Performance and scalability are becoming critical aspects for tackling the challenges of exascale computing [14]. Thus, most of the latest research efforts have been spent in the runtime system.

However, the runtime system cannot overcome performance bugs in the application code. Performance analysis and profiling tools have been proposed over the years with the goal of helping developers to improve scalability of their codes [13, 15, 7]. Such tools are often very helpful to determine performance bottlenecks or root causes for performance issues, however, the programmer has to adapt the code eventually. In addition, tracing and post mortem analysis, may

⋆ This research has been partially funded by the Austrian Research Promotion Agency under contract nr. 824925 (OpenCore) and under contract 834307 (AutoCore).

```
1  for(unsigned iter=0; iter<NUM_ITERS; iter++) {
2  S0  MPI_Sendrecv(&A[ROWS-2][0], COLS, MPI_DOUBLE, top, 0,
3            &A[0][0], COLS, MPI_DOUBLE, bottom, 0, MPI_COMM_WORLD, &s);
4  S1  MPI_Sendrecv(&A[1][0], COLS, MPI_DOUBLE, bottom, 1,
5            &A[ROWS-1][0], COLS, MPI_DOUBLE, top, 1, MPI_COMM_WORLD, &s);
6    for(unsigned i = 1; i<ROWS-1; ++i)
7      for(unsigned j = 1; j<COLS-1; ++j)
8  S2        tmp[i][j] = A[i][j] + 1/4*(A[i+1][j]+A[i-1][j]+A[i][j-1]+A[i][j+1]);
9    double** ttemp=A; A=tmp; tmp=ttemp; // swap arrays
10 }
```

Listing 1.1. 5-points stencil code

be extremely time- and resource-consuming. Tuning the code for a particular architecture (e.g., determine software pipeline depths and optimal loop arrangement) is thus a very labor-intensive process and is often simply not applied in production environments.

Compiler technology has been used in the past to optimize MPI programs [6, 4, 2, 3, 11]. The main idea is to extend the compiler analysis module to understand the semantics of MPI routines and treat them not just like a library call but as a language construct. In doing so, existing compiler analysis can be utilized to uncover optimization potentials hidden within the input code.

In this work we show an approach based on compiler analysis, and specifically *exact* data dependency analysis to maximize the computation/communication overlap for a given input code. Indeed, increasing the time window on which computation and communication can be performed in parallel (or overlapped) is one of the well known rules of thumb used to optimize MPI codes. As opposed to the previous compiler-based approaches, we utilize finer-grain *exact* analyses using the polyhedral model [1]. Unlike the traditional dependence graph, which contains data dependency information between the program statements, the *dependence polyhedron* lists dependencies on the basis of *statement instances* [16]. An instance of a statement is a particular dynamic execution of that statement. For example, the body of a loop has as many instances as there are iterations. By using this more detailed analysis our approach increases the overlap window between generating the data or buffer availability and the final consumption of the data.

2 Motivation and State of the Art

MPI programs often exhibit recurring code patterns which are direct consequences of the programming paradigm. For example, many programs read the data right after receiving it from a peer process by iterating over the received array elements. Similarly, data is usually sent right after the sender process finishes the computation that writes to array elements being transmitted. A concrete and relevant example is represented by a standard parallelization of a 5-point stencil code depicted in Listing 1.1. Stencil codes are very important in computational sciences and we show a common way to parallelize such a code [9]. We have communication statements at the beginning of the loop, statements `S0` and `S1`, which exchange data being computed in the previous iteration. Right after the

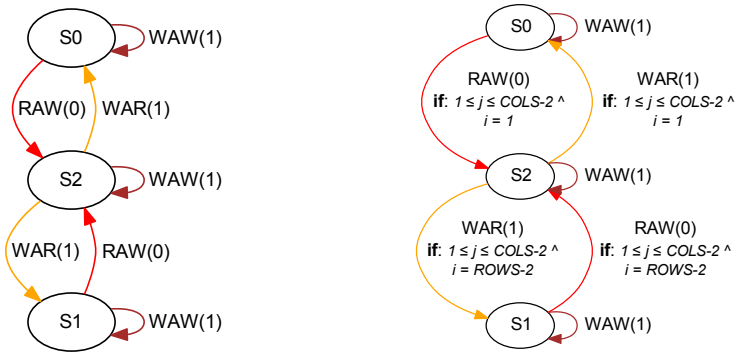
communication is performed, data is updated by a computational loop, statement `S2`. In both case the compiler sees a *true*, or *Read-After-Write* (RAW), data dependence on the elements of array `A` from statement `S0` to `S2` and between `S1` and `S2`.

Traditional compiler analyses usually derive dependence information on a per-statement basis. For the 5-point stencil code in Listing 1.1 the data dependence graph (DDG) built by classical data dependence analysis [10] is represented in Figure 1(a). We neglect, in this analysis, the swap statements in line 10 since it introduces data dependencies between successive iterations of the `iter` loop which are irrelevant since our focus is in maximizing the overlap within the loop body. The DDG shows three types of dependencies present in the code:

RAW : Read-After-Write dependencies (a.k.a. true-dependencies);
WAR : Write-After-Read dependencies (a.k.a. anti-dependencies);
WAW : Write-After-Write dependencies (a.k.a. output-dependencies);

Each dependence type is associated with a *distance vector* represented in brackets which, in the case of non loop-carried dependencies, is zero. We see that there are two, non loop-carried, RAW dependencies from statement `S0` to `S2` and between `S1` and `S2`, respectively. This is caused by the receive operation (implicit in the `MPI_Sendrecv` routine) writing elements of the array `A`. More precisely, the receive operation in `S0` writes `A`'s array elements in the range $A[0][0 : COLS)$. Same elements which are going to be read later in the first iteration of the stencil loop – and thus Read-After-Write – by statement `S2`. Although correct,these results are too conservative and coarse grained *inhibiting any kind of automatic optimization*. As a matter of fact, every dependence in the DDG exists for all the dynamic executions, or instances, of interested statements, however this is not the case. For example the dependence between `S0` and `S2` only applies to the first iteration of the stencil loop, all the remaining dynamic executions of statement `S2` are not dependent on `S0`. Similar considerations can be done for statement `S1`, for which the data dependence applies solely to the last iteration of the stencil loop.

The polyhedral model enables novel data analysis and transformation techniques by representing dependencies at the finest detail in an instance-based fashion. This technique is also referred to as *exact* data dependence analysis [16]. This allow a compiler to relax some of the constraints and apply more aggressive transformations at the array element level which would not be supported by a more coarse level of analysis at the object level. An example of the dependence polyhedron for the stencil code is shown in Figure 1(b). The graph contains the exact same key dependencies but it carries more information for each of them. An *expression predicate* states which subset of the statement instances are affected by the dependence. When the predicate is missing, then the dependence applies to every instance of that couple of statements. For example, the non loop-carried RAW dependence between statements `S0` and `S2` exists for all the instances of `S2` where iterator `i` is 1 and `j` is between 1 and `COLS-2` inclusive. This means that the remaining instance of the stencil loop are not dependent on the communication statements and therefore can be used to hide communication costs.



(a) Statement-based DDG (b) Instance-based DDG

Fig. 1. Data Dependency Graph (DDG) for 5-points stencil code in Listing 1.1

3 The Polyhedral Model and Integration of MPI Semantics

The polyhedral model represents, in an algebraic way, the execution of a program composed of arbitrary nested loops with affine loop indexes. It captures both the control-flow and data-flow of a program using three compact linear algebraic structures, i.e. the *iteration domain*, the *scheduling* (or *scattering*) *function* and the *access function*. The main idea is to define, for a statement S, a *space* in Z^N where each point correspond to an execution, or *instance*, of S. The value of the coordinates of a point within this space represents the value of the N loop iterators spawning statement S. In order to keep the representation compact, the space, called *polyhedron*, is defined by a set of bounding *affine inequalities*.

Iteration Domain The space on which a statement is defined is also referred to as its *Iteration Domain*, $\mathcal{D}_S$. For example consider the stencil code in Listing 1.1. Each statement is defined within an iteration domain which is bound by the surrounding control flow statements. For example the iteration domains for `S0`, `S1` and `S2` are defined as follows:

$$\mathcal{D}_{S_0} = \{ iter \mid 0 \leq iter < NUM\_ITERS \}$$
$$\mathcal{D}_{S_1} = \{ iter \mid 0 \leq iter < NUM\_ITERS \}$$
$$\mathcal{D}_{S_2} = \{ iter, i, j \mid 0 \leq iter < NUM\_ITERS \wedge 1 \leq i < ROWS - 1 \wedge 1 \leq j < COLS - 1 \}$$

Iteration domains are represented by an integer matrix A, multiplied by a so called *iteration vector* $\boldsymbol{x}$. The iteration vector determine the dimensionality of the space on which a statement is defined (composed by the loop iterators enclosing that statement). For example the iteration domain for statement `S2` in Listing 1.1 is defined by the vector $\boldsymbol{x}_{S_2} = \begin{pmatrix} iter \\ i \\ j \end{pmatrix}$.

Scheduling Function The second piece of information which is required to describe the semantics of a program are the so-called *scheduling* (or *scattering*)

functions. Intuitively, statements belonging to a loop body, and subject to the same control flow, will share identical iteration domains. The information of the order on which statement instances are executed is not represented. A *schedule*, $\theta(\boldsymbol{x})$, is a function which associates a logical *execution date*, or *time-stamp*, to each instance of a statement. This allows the ordering of the instances defined by the iteration domain and furthermore it defines an execution order for instances of different statements.

Access Function One last function is also required to capture the data locations on which a statement operates. The *access* (or *subscript*) *function* describes the index expression utilized to access an array, and therefore memory locations, within a statement. For compactness reasons, it is represented as a matrix. Access functions also store the information whether a particular memory location is being read (i.e., `USE`) or written (i.e., `DEF`). This kind of information is utilized by the polyhedral model to compute exact data dependency analysis for a given input code.

3.1 Instance-based Data Dependence Analysis

A statement R *depends* on a statement S if there exists an operation $S(\boldsymbol{x_1})$, an operation $R(\boldsymbol{x_2})$, and a memory location m such that:

- $S(\boldsymbol{x_1})$ and $R(\boldsymbol{x_2})$ refer to the same memory location m, and at least one of them writes to that location;
- $\boldsymbol{x_1}$ and $\boldsymbol{x_2}$ respectively belong to the iteration domain of S and R;
- $S(\boldsymbol{x_1})$ precedes $R(\boldsymbol{x_2})$.

Dependence information is computed on the basis of the three data structures presented earlier in this section. Intuitively, every point of the iteration domain is projected into a different space using the affine linear transformation represented by the access functions. The domain of this transformation is defined by the statement instances and the co-domain is the memory elements being accessed by that particular statement. Intersecting the co-domains obtained for every statement yields the set of memory elements for which a data dependence may occur. Finally combining this information with the statement execution dates, given by the scheduling matrix, makes it possible to determine the *source* and the *sink* for every dependence.

This complex capability to perform data dependence analysis is implemented in the majority of the libraries supporting the polyhedral model. In our work we utilized the Integer Set Library (ISL) [17] currently employed in several mainstream compilers like GCC and LLVM.

3.2 Limitations of the Polyhedral Model

As mentioned above, the polyhedral model requires affine constraints to describe control- and data-flow. Thus, not every program can be completely represented in the polyhedral model. To maximize the applicability to arbitrary

programs, the program is typically split into *Static Control Parts* (SCoPs) that are defined to be the maximal set of consecutive instructions such that: loop bounds, conditionals, and subscript expression are all affine functions of the surrounding loop iterators and global variables; loop iterators and global variables cannot be modified [1]. Girbal et al. demonstrated that SCoPs capture a large portion of the computation time in scientific applications [8].

3.3 Integration of MPI Semantics

Another limitation of a SCoP is the absence of any function or library call. However, if the body of the invoked function is available at compile time, *inlining* can be used to increase the size of the SCoP. This technique is applicable only if the function is not recursive and it has a single-entry and single-exit point. In order to overcome the problem with library routines, for which the source code is not available at compile time, our compiler pre-processes the input program and replaces MPI communication routines with semantically equivalent loop statements. Indeed our prototype deals with `MPI_Send` and `MPI_Recv` statements using plain datatypes for now. Under these circumstances, a `send(&buff[offset], size)` operation is semantically equivalent to a for loop *reading*, i.e., `USE`, elements `buff[i]` $\forall\, i \in [offset, offset + size)$. Similarly, a `receive(&buff[offset], size)` operation will be replaced by a loop *writing*, i.e., `DEF`, the same range of array elements. In this form, programs containing MPI routines can be handled by the polyhedral model and existing analysis and transformation tools can be utilized. While this transformation is sound for most MPI codes, it neglects the message tag and the communicator. To maintain the original relative ordering, additional data dependencies must be introduced between the generated loops for communication routines to enforce MPI's matching rules. Determining the value of the message tag at a specific program point requires, in the general case, dataflow analysis reaching beyond the SCoP boundaries (e.g. *aliasing* detection).

4 Implementation and Evaluation

In this section we propose a compiler transformation which based on the result of the instance based data dependence analysis obtained by the polyhedral model, maximize the communication/computation overlap by accordingly transforming the input program.

4.1 Implementation

The entire approach is implemented in the *Insieme Compiler and Runtime* infrastructure [5]. The *Insieme* project aims to provide an easy to use, powerful framework for source-to-source transformations and program analysis for heterogeneous multi-core parallel computers. It consists primarily of two components: the *Insieme Compiler* and the *Insieme Runtime System*. The *Insieme Compiler*,

Algorithm 1 Transformation flow for maximizing communication/computation overlap

```
1: Input: P = Syntax Tree of the input program; MOD = modified AST
2: Output: T = Syntax Tree of the transformed program
3: T = P
4: repeat
5:     MOD = false; G = extractDDG(T)
6:     for all dep ∈ G do
7:         if dist(dep) is 0 && src(dep) is MPI routine && sink(dep) is loop body && dep applies to a subset of the
           instances then
8:             T = applyLoopFission(T, sink(dep), findCut(dep)); MOD = true
9:         end if
10:    end for
11: until MOD is false
12: for all dep ∈ G do
13:    if dist(dep) is 0 && src(dep) is MPI routine && sink(dep) is loop body then
14:        {COMM_STMT, WAIT_STMT} = toAsynchronous(src(dep))
15:        T = removeStmt(T, src(dep))
16:        T = moveToEarliestSchedule(T, COMM_STMT)
17:        T = moveToLatestSchedule(T, {WAIT_STMT, sink(dep)})
18:    end if
19: end for
```

on which our work relies, fully integrates the polyhedral model analysis and transformations and provides a foundation for source-to-source program optimization. Its architecture is designed to support the processing of hybrid input codes that can include MPI, OpenMP and OpenCL written in C/C++.

Normal Form Before applying any transformation, the input code is preprocessed into a normal form. In this, an MPI program only contains `MPI_Send` and `MPI_Recv` statements so that successive steps of the analysis process are simplified. It is worth noting that the normalized program could have different buffering requirements and therefore may lead to deadlocks if executed. However, the program is kept in this normalized form only for the sake of performing static analysis. The shape of an MPI program in normal form is described by the following rules:

- Non-blocking point-to-point operations are rewritten to use the corresponding blocking version. This is obtained by replacing every asynchronous routine with the synchronous counterpart and by removing every `MPI_Wait` statement in the input code.
- `MPI_Sendrecv` operations are split into the corresponding `MPI_Send` and `MPI_Recv` operations.
- `MPI_Ssend`, `MPI_Rsend` or `MPI_Bsend` are rewritten to plain `MPI_Send`.

Handling of MPI Routine Semantics Once the program is in normal form, we replace MPI statements with their semantically equivalent loops as described in Section 3.3. From this representation of the input program (which does not contain MPI statements anymore), we proceed with the extraction of the SCoP

```
1  for(unsigned iter=0; iter<NUM_ITERS; iter++) {
2    MPI_Request __req0, __req1;
3    MPI_Irecv(&A[0][0],COLS,MPI_DOUBLE,bottom,0,com,&__req0);
4    MPI_Irecv(&A[ROWS-1][0],COLS,MPI_DOUBLE,top,1,com,&__req1);
5    MPI_Send(&A[1][0],COLS,MPI_DOUBLE,bottom,1,com);
6    MPI_Send(&A[ROWS-2][0],COLS,MPI_DOUBLE,top,0,com);
7    // stencil loop after fission
8    for(unsigned i = 2; i<ROWS-2; ++i)
9      for(unsigned j = 1; j<COLS-1; ++j)
10       tmp[i][j] = A[i][j] + 1/4*(A[i+1][j]+A[i-1][j]+A[i][j-1]+A[i][j+1]);
11   MPI_Wait(&__req0, MPI_STATUS_IGNORE);
12   // first iteration of stencil
13   for(unsigned j = 1; j<COLS-1; ++j)
14       tmp[1][j] = A[1][j] + 1/4*(A[2][j]+A[0][j]+A[1][j-1]+A[1][j+1]);
15   MPI_Wait(&__req1, MPI_STATUS_IGNORE);
16   // last iteration of stencil loop
17   for(unsigned j = 1; j<COLS-1; ++j)
18       tmp[ROWS-2][j] = A[ROWS-2][j] + 1/4*(A[ROWS-1][j]+
19               A[ROWS-3][j]+A[ROWS-2][j-1]+A[ROWS-2][j+1]);
20   double** ttemp=A; A=tmp; tmp=ttemp; // swap arrays
21 }
```

Listing 1.2. 5-points stencil code after code optimization

and the dependence polyhedron associated to it. In doing so we keep a link to the communication statement being replaced internally.

Code transformation Once the instance-based DDG is generated, we apply a sequence of transformations as described in Algorithm 1. The idea is to iterate through all the non loop-carried dependencies which have an MPI communication statement as the source and a loop body as sink. If the dependence applies to a subset of the instances of the sink then we split the loop, applying the loop fission transformation [10], at the range provided by the dependence analysis. In this way the iterations which are dependent on the MPI communication statement are isolated into a new loop statement. Notice that fission is possible as long as there are no dependencies in the loop body that conflict with the transformation being applied. The transformation framework in the Insieme Compiler implements a pre-condition analysis which determine whether a transformation can be safely applied.

The procedure repeats until a fix-point is reached where every dependence in the DDG applies to all the instances of the source and sink statement. The next step is to consider all dependencies between communication statements and computational loops based on the transformed code. For each of them, the source of the dependence – the communication statement – is removed from the code and the corresponding asynchronous version of the routine is scheduled in its earliest position (which is determined by constraints in the DDG). Listing 1.2 shows the transformed stencil code from Listing 1.1. The receive is scheduled at the beginning of the loop body as shown in lines 3 and 4. The loop depending on the communication statement, i.e., the sink, is scheduled lazily prepending to it an `MPI_Wait` operation placed to preserve the semantics of the program, lines 11 – 19 of Listing 1.2. The remaining non-dependent loop iterations will be, by the end of the transformation, confined between the issuing of the asynchronous

VSC2				LEO3			
# of MPI	Original (in msecs.)	Transformed (in msecs.)	Improvement (in %)	# of MPI	Original (in msecs.)	Transformed (in msecs.)	Improvement (in %)
16	219	218	0.3	12	264.9	264.5	0.09
32	89.7	89.0	0.8	24	118.7	118.9	-0.01
64	35.1	32.0	9.5	48	37.4	37.0	0.9
128	20.1	17.9	12.6	96	21.0	20.2	4.0
256	13.1	11.5	13.5	192	11.3	9.8	15.3
512	12.0	9.3	27.9	384	7.6	6.4	19.1

Table 1. Evaluation of the transformed code on the VSC2 and LEO3 cluster, fixed problem size of 4Kx4K and `NUM_ITERS=10`

communication operations and the consumption of the received data (lines 8–10). Therefore maximizing the overlap window.

The transformation can be easily extended to take into account loop-carried dependencies, in that case the distance of the data dependence, d, defines the number of loop cycles which can be executed between the source and the sink of the dependence. This can be handled by automatically allocating an array of d requests objects for each communication routine where the `MPI_Wait` statement of a request generated by a communication statement at iteration i happens at iteration $i+d$. This transformation, also known as *software pipelining* [10], requires additional control code, therefore overhead, to be inserted by the compiler to correctly fill and unload the pipeline. A compiler can employ static heuristics in order to determine when software pipelining is beneficial for a given input code.

4.2 Evaluation

We tested the transformed 5-point stencil code, depicted in Listing 1.2, on two production clusters and compared its execution time with the original code shown in Listing 1.1. The (*i*) Vienna Supercomputing Cluster 2 (VSC2) is a HPC system which consists of 1,314 nodes, with 2 AMD 8-cores Opteron 6132 HE processors each; the (*ii*) LEO3 cluster which consists of 162 compute nodes, with 2 Intel 6-cores Xeon X5650 CPUs. Both clusters use InfiniBand 4x QDR high speed interconnect.

The code has been executed keeping the problem size constant, 4K by 4K elements, and varying the number of MPI processes, results for both architectures are shown in Table 1. We see that, as expected, the transformed code has overall a better performance. Additionally, the improvement increases with the number of cores since the smaller problem slice assigned to each processor is, the more dominant the communication overhead becomes. Since our transformation aims at hiding communication costs, its benefit grows as the computation/communication ratio diminishes.

5 Conclusions and Outlook

In this paper we showed a compiler optimization which leverages instance-based data dependence analysis, based on the polyhedral model, to isolate loop iterations which are dependent on MPI communication statements. Consecutive proper rescheduling of statements allows the communication/computation overlap to be maximized.

Differently from classic data dependence analysis results, which state dependence relationships at statement level, our approach finds overlap opportunities within loop iterations and therefore at a more finer grain level.

We implemented the entire approach in the Insieme source-to-source compiler [5] and showed how the transformed code has an improved performance, up to 28% faster with 512 cores, because of the increased overlap.

Acknowledgments. This work was supported by the Austrian Ministry of Science BMWF as part of the UniInfrastrukturprogramm of the Research Platform Scientific Computing at the University of Innsbruck. Furthermore, the computational results presented have been achieved in part using the Vienna Scientific Cluster (VSC).

References

1. Benabderrahmane, M.W., et al.: The polyhedral model is more widely applicable than you think. In: Proc. of the Intl. Conf. on Compiler Constr. LNCS (Mar 2010)
2. Danalis, A., Pollock, L., Swany, M.: Automatic MPI application transformation with ASPhALT. In: Par. and Distr. Proc. Symp., IPDPS 2007. pp. 1 –8 (Mar 2007)
3. Danalis, A., Kim, K.Y., Pollock, L., Swany, M.: Transformations to parallel codes for communication-computation overlap. In: Proceedings of the 2005 ACM/IEEE conference on Supercomputing. pp. 58–. SC '05, Washington, DC, USA (2005)
4. Danalis, A., Pollock, L., Swany, M., Cavazos, J.: MPI-aware compiler optimizations for improving communication-computation overlap. In: Proceedings of the 23rd international conference on Supercomputing. pp. 316–325. ICS '09 (2009)
5. Distributed and Parallel Systems Group, University of Innsbruck: Insieme Comiler and Runtime Infrastructure, `http://insieme-compiler.org`
6. Fahringer, T., Mehofer, E.: Buffer-safe communication optimization based on data flow analysis and performance prediction. In: Proc. of the 1997 Intl. Conf. on Parallel Architectures and Compilation Techniques, PACT'97. pp. 189–200 (1997)
7. Geimer, M., Wolf, F., Wylie, B.J.N., Ábrahám, E., Becker, D., Mohr, B.: The Scalasca performance toolset architecture. CCPE Journal 22(6), 702–719 (Apr 2010)
8. Girbal, S., et al.: Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. Intl. Journal of Par. Progr. 34(3), 261–317 (Jun 2006)
9. Gropp, W., Lusk, E., Skjellum, A.: Using MPI (2nd ed.): portable parallel programming with the message-passing interface. MIT Press, Cambridge, MA, USA (1999)
10. Kennedy, K., Allen, J.R.: Optimizing compilers for modern architectures: a dependence-based approach. San Francisco, CA, USA (2002)
11. Knüpfer, A., et al.: The vampir performance analysis tool-set. In: Tools for High Performance Computing, pp. 139–155 (2008)
12. MPI Forum: MPI: A Message-Passing Interface Standard. Version 2.2 (September 4th 2009), available at: `http://www.mpi-forum.org` (Dec 2009)
13. Shende, S.S., Malony, A.D.: The Tau Parallel Performance System. Int. J. High Perform. Comput. Appl. 20(2), 287–311 (May 2006)
14. Thakur, R., Balaji, P., Buntinas, D., Goodell, D., Gropp, W., Hoefler, T., Kumar, S., Lusk, E., Traeff, J.L.: MPI at Exascale. In: Procceedings of SciDAC 2010 (Jun 2010)
15. Truong, H.L., Fahringer, T.: SCALEA: A performance analysis tool for distributed and parallel programs. In: Euro-Par 2002, LNCS, vol. 2400, pp. 41–55 (2002)
16. Vasilache, N., Cohen, A., Bastoul, C., Girbal, S.: Violated dependence analysis. In: In ACM ICS (2006)
17. Verdoolaege, S.: isl: An Integer Set Library for the Polyhedral Model. In: Mathematical Software ICMS 2010, LNCS, vol. 6327, pp. 299–302 (2010)