Hiding Communication Latency with non-SPMD, Graph-Based Execution

Jacob Sorensen and Scott B. Baden

Department of Computer Science and Engineering
University of California, San Diego
9500 Gilman Drive
La Jolla, CA 92093-0404 USA
http://www.cse.ucsd.edu/groups/hpcl/scg

Abstract. Reformulating an algorithm to mask communication delays is crucial in maintaining scalability, but traditional solutions embed the overlap strategy into the application. We present an alternative approach based on dataflow, that factors the overlap strategy out of the application. Using this approach we are able to reduce communication delays, meeting and in many cases exceeding performance obtained with traditional hand coded applications.

Key words: parallel programming, latency tolerance, non-SPMD, coarse grain dataflow

1 Introduction

Spurred on by the multi-core processor, scalable systems have the potential to enable simulations of remarkable fidelity and complexity, leading to new scientific discovery. However, improvements in processor performance amplify the cost of off-chip data motion, and applications must cope by with this trend by tolerating latency. Implementing and tuning an application to overlap communication with computation is daunting for the domain scientist, and a challenge even for the expert programmer. Traditionally, the overlap strategy is embedded into the application, and relies on split phase coding. Software development is tedious and prone to error, and the application suffers from non-robust performance.

Overlap strategies expose opportunities to mask communication costs by relaxing the total ordering imposed by traditional bulk synchronous implementation, e.g. with MPI [1]. A compiler may be able to determine a suitable partial ordering in some cases, but often the partial orderings are difficult to analyze, even by hand.

A natural way to realize partial orderings is by means of a task precedence graph, or *task graph* for short. Once the program has been expressed in terms of a task graph, a scheduler executes the partially ordered tasks according to the flow of data, e.g. dataflow [2–5], realizing overlap automatically. We have implemented this approach with a run time library, with run time services to support the data flow semantics via background threads. These services reorder communication

and computation tasks dynamically according to the flow of information, automatically choreographing communication to move it out of the critical path of computation. We have implemented applications using this approach, enabling us to factor policy and scheduling decisions out of the application code.

This paper makes two contributions. First, we show that a task precedence graph formulation is able to support latency tolerance, meeting and in many cases exceeding performance of traditional split phase encodings. Moreover, this formulation separates implementation policy from correctness. Second, we show that the approach supports application specific scheduling without affecting correctness of user code. Our task precedence graph supports performance metadata [6] that may be used to improve performance, in particular, to fine tune the scheduler.

2 Task Precedence Graph Representation

Scientific applications spend most of their time executing loop nests. We may describe a loop nest using an *Iteration Space Graph* (ISG), which represents the underlying dependence structure. Any scheme to parallelize an application must preserve the dependencies in the constituent ISGs. We may construct a task precedence graph using this ISG in which each task corresponds to a region of the iteration space. We call the resultant graph a *TaskGraph*. In a classic MPI implementation tasks are usually mapped 1:1 to processors. However, in order to mask data transfer delays, Little's law [7] prescribes that we render many tasks for each processor. As a result, we must solve a scheduling problem. Herein lies the difficulty: classic overlap strategies rely on split phase algorithms that embed overlap strategy into the application source code, resulting in high software development costs, and difficulty in porting code to new hardware.

Alternatively, we may execute the TaskGraph under the dataflow semantics [2–4]. Parallelism arises among independent tasks and interdependent tasks are enabled according to the flow of data among them. There is no need to embed scheduling policies into the application since these are handled by background services.

We have constructed a library, called *Thyme*, which enables us to implement applications in this way. Thyme avoids the need to write complicated split phase algorithms and decouples the overlap strategy from application correctness. Although Thyme's API may be used to develop applications directly, we envision that it will ultimately become part of a run time support library for a compiler. Space limitations prevent us from discussing this API, which, together with a detailed presentation of the implementation, will be described elsewhere.

Thyme is currently implemented as a C++ class library on top of MPI and pthreads, and implements two primary datatypes – *Task* and *TaskGraph*. A Thyme program constructs and executes a set of TaskGraphs under the control of the *Run Time Services* which process task completions and arrivals, move data among dependent tasks, and invoke a Scheduler. The services are decentralized, and run on all processing modules, communicating as necessary to manage Task-

Graph execution. A TaskGraph is a distributed data structure and it executes according to the owner computes rule. Each task is assigned an owning processing module, but any processor within the module may execute the task, since processors share memory within a node.

The Scheduler maintains a priority queue of tasks from the TaskGraph that are ready but not yet executing. Each task has an associated *priority*. This information exists as performance meta-data [6] decorating the task graph. Meta-data may be be specified by the programmer in order to improve performance, but does not affect correctness since task graph execution preserves all dependence constraints. Thus, the programmer is free to explore application-specific scheduling without reformulating the application. (The Thyme user may substitute their own scheduler in place of the default in order to further tune application performance.)

3 Experiments

We ran on two large-scale systems. *DataStar,* located at the San Diego Supercomputer Center, is an IBM system running AIX 5.2, with 8-way nodes containing 1.5 GHz Power4+ processors with 16 Gigabytes of shared memory, connected by a Federation switch. *Thunder,* located at Lawrence Livermore National Laboratory, is a Linux cluster running CHAOS version 3.3, with a Quadrics QSNET-II interconnect based on Elan4 and Elite4 components. Each "Madison Tiger4" node comprises four Itanium2 CPUs running at 1.4 GHz, with 8 Gigabytes of shared memory. We compiled on DataStar using `mpCC_r`, which invoked version 8.0 of the `xlC_r` compiler. IBM's `ESSL` version 4.2.0.3 provided the high performance matrix multiply routine `dgemm()` and FFTW 3.0 provided the FFT. We compiled on Thunder with `mpiicpc`, which invoked `icpc 9.1`. Intel's M̂KL 8.1.1 provided `dgemm` and FFT (the latter through the FFTW 3.0 interface).

We tested Thyme's ability to achieve overlap with three applications coming from Colella's seven application motifs [8]. Jacobi3D, a 3-D iterative Poisson solver (Dirichlet Boundary Conditions); MMULT, Matrix Multiplication, and the NAS-FT parallel benchmark, ver. 3.0 [9], which is dominated by a 3D Fast Fourier Transform. For each application we compared the Thyme implementation against two variants written with MPI. The *baseline* variant (BASE) uses blocking communication and does not attempt to overlap communication with computation. The *explicit overlap* variant (OLAP) uses asynchronous nonblocking communication to implement a split-phase algorithm to overlap communication with computation. The *Thyme* variant does not make MPI calls, since the run time services handle data motion automatically. These services commandeer one core per node to carry out their activities. Thus, although Thyme uses the same number of processors as non-Thyme variants in our experiments, fewer processors actually perform the computation.

We also report an *IDEAL* running time, which is the time required to perform computational work only. We obtained this time by disabling communication in BASE. Although the computed results are incorrect, the amount of computa-

tional work performed is not affected. This allows us to indirectly measure the cost of communication and thus establish an upper bound on the potential for improving performance by masking data motion costs.

Jacobi3D. *Jacobi3D* iteratively updates a 3D mesh using a 7-point stencil. We split the mesh uniformly and employ ghost cells to store border data from (up to) six neighboring sub-domains. The BASE variant used an $8 \times 4 \times 8$ processor geometry which is optimal for 256 processors. Ghost cells are exchanged prior to each iteration using `SendRecv`. The OLAP variant pre-fetches ghost cells [10]. It further subdivides each processor's subdomain into an inner core and an outer annulus. The annulus is a thin shell, one cell thick, encircling the inner core and inscribed inside the ghost region. Unlike the outer annulus, the computation on the inner core does not depend on the ghost cells; it is relaxed simultaneously with ghost cell exchange. Once all the ghost cells have all arrived, the outer annulus is then relaxed. The *Thyme* variant subdivides the mesh into many more tasks than processors. It uses a hierarchical decomposition to split the data first over nodes, and then over processors within a node. We used different *node* geometries on Thunder's 4-way nodes than on DataStar's 8-way nodes: $4 \times 4 \times 4$ and $4 \times 2 \times 4$, respectively. The *processor* geometries were the same on both platforms: $4 \times 4 \times 4$. Each task relaxes one block and depends on up to seven others from the previous iteration: the block in the current position plus up to six nearest neighbors. In the BASE and OLAP variants each processor stores its mesh as one contiguous memory area. The Thyme variant stores its data as separate contiguous blocks, improving cache locality.

We ran Jacobi3D for 25 iterations on 256 processors using problem sizes varying from 320^3 to 1600^3, beyond which communication is not a significant bottleneck. Fig. 1 (top) shows that the THYME variant enjoys a clear performance advantage over BASE, overlapping 43-78% of the communication on Thunder and 10-80% on DataStar. The OLAP variant not only failed to improve the running time but actually increased it. As noted by Baden and Shalit, the thin outer annulus has large strides and this slows down the updates to cells in the outer annulus considerably [11].

Thyme's graph-based execution model is well suited to this application. Rather than using thin annular faces, we break up the mesh into numerous small cubes, each stored in a contiguous area of memory. Relaxation exhibits good cache locality over these cubes, avoiding the computation time penalty imposed by OLAP's thin outer annulus.

We observed that execution from one iteration was frequently intermingled with that of the next, such that iteration boundaries no longer serve as precise synchronization point. We found no discernible pattern in task execution order from run to run. This implies that flexibility in scheduling may be helping performance and that an optimal schedule may be difficult to predict.

Matrix Multiplication. *MMULT* computes the matrix product $C = A \times B$, formulating the algorithm as a sequence of blocked outer products and sub-

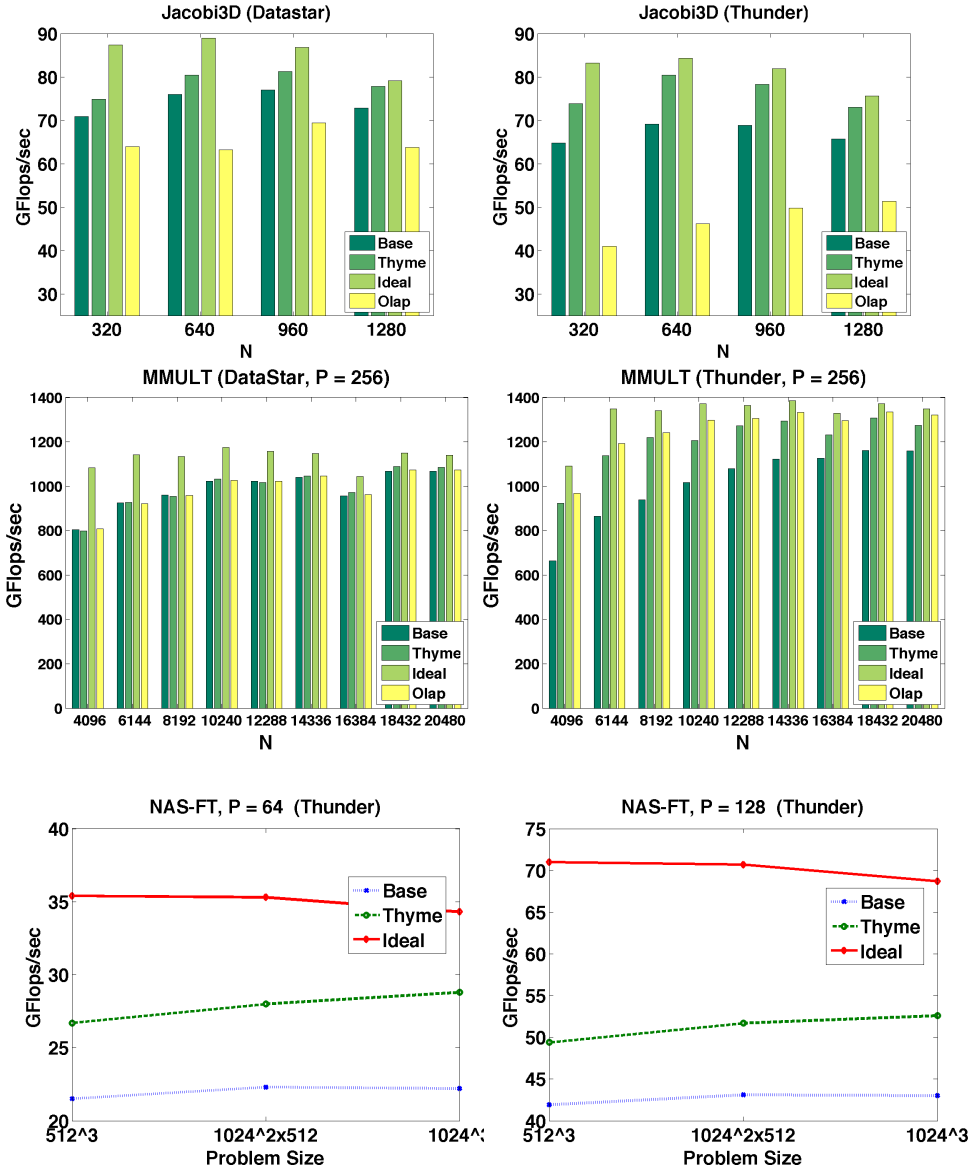

Fig. 1. Data-driven execution improves performance by overlapping communication with computation. Results are shown for Jacobi3D (top), MMULT (middle) running on DataStar (left) and Thunder (right), and for NAS-FT running on 64 and 128 processors of Thunder (bottom, left and right, respectively). No OLAP variant was implemented for FT.

dividing the matrices over a square 2-d processing geometry. The strategy is similar to SUMMA [12]. Each submatrix is further subdivided into panels of width (or height) n_b. The algorithm proceeds in $N/(n_b\sqrt{P})$ steps, where the processing geometry has P virtual processors. The panels circulate by row (for A) and by column (for B) and each processor computes a partial matrix product which is summed into its local portion of C (matrix multiply-add, *mpy-add*). Communication involves four nearest neighbors, proceeds in just one direction, and is periodic. BASE performs each mpy-add before transferring a panel of A and B; OLAP initiates panel transfer and then performs the mpy-add in parallel. THYME mirrors the behavior of the OLAP variant; the graph dependencies cause the panels to be passed around their respective rows and columns in pipelined fashion. Each task performs a mpy-add and depends on the left neighbor to receive a panel of A and the upper neighbor to receive a panel of B, passing, after execution, A to the right and B downwards. We used the vendor-provided `dgemm` routine to carry out mpy-adds and used two panels ($n_b = 2$) per block, which was optimal.

We ran MMULT on 256 processors with problem sizes ranging from $4,096^2$ to $20,480^2$, the largest size where communication had a significant affect on performance. Fig. 1 (middle) shows the result. THYME achieves near-optimal performance on Thunder, overlapping 61-93% of the communication. The results are not as clear-cut on DataStar, where the Thyme and baseline variants realize more or less the same performance. This is true because mpy-adds are 17-39% slower than on DataStar than on Thunder (except for the smallest problem size, where DataStar's mpy-adds were faster). Thus, DataStar runs incur a smaller fraction of communication time, and with less communication, there is less benefit to hiding it.

Thyme could not improve on the hand-coded OLAP variant. This is true because the communication pattern is highly constrained to nearest neighbors, and thus the flexibility of Thyme's data driven model doesn't offer an improvement. However, this flexibility doesn't *penalize* performance either. Unlike the hand-coded OLAP variant, the Thyme variant is free from split phase coding and embedded policy decisions, enhancing performance portability.

We used performance meta-data to guide task scheduling in the THYME variant of MMULT. The scheduler's priority queue exhibits LIFO behavior when the task priorities are equal. Thus, newly-ready tasks get scheduled before older readied tasks. While generally beneficial for cache locality, this behavior disrupts the pipeline of block transfers directed by the TaskGraph, tending to serialize computation. Processors would execute tasks over newly arrived blocks, starving neighboring processing nodes and causing long wait times. We were able to improve performance, without having to reformulate the application, by simply assigning older tasks a higher priority than newly ready ones. This entailed annotating the graph with appropriate meta-data. The effect is to alter the scheduler's behavior toward a FIFO without affecting correctness. This behavior was friendlier to the pipeline structure of the graph and resulted in a 20% performance improvement.

NAS-FT. The NAS-FT benchmark includes a costly transpose operation for setting up the FFT, and thus can benefit greatly from masking communication delays. The publicly available code [9] serves as the BASE variant. NAS-FT employs a 1-d virtual processor geometry in the Z dimension such that each processor receives a *slab* of the mesh. Each slab is further subdivided into 2D *sheets* of size $NX \times NY \times 1$, where NX and NY are the leading dimensions. The algorithm has 3 steps. After performing a 2D FFT on each of the NZ sheets, BASE invokes a total exchange `AllToAll` to transpose the data so each processor has slabs of the complete Z dimension. The processors complete the transform by computing 1-d FFTs along the Z dimension. The Thyme variant treats each sheet as a task. Thus, communication is much finer grained than with BASE and is interleaved with communication. The graph dependencies force the final 1-d FFTs to wait for all the 2D transforms to complete, so there are NX 1D transform tasks over sheets of size $1 \times NY \times NZ$.

We measured the execution time for three different problem sizes over various numbers of processors and ran for 20 iterations. The problem sizes were 512^3 (NAS-FT Class C), plus two larger sizes to test the effects of increasing the sheet size ($1024 \times 1024 \times 512$) and the number of sheets (1024^3). Current results are from Thunder only. We did not attempt to reformulate the 2224 line BASE variant to overlap communication with computation (OLAP).

Fig. 1 (bottom) shows that Thyme is able to hide communication significantly–37-65%–depending on the problem size and number of processors. The application incurs a 35-43% communication overhead.

There is room for improvement in NAS FT, in particular, to employ a 2-d "stick" decomposition in lieu of the 1-d decomposition used currently. Thyme is currently implemented using MPI, however, and the difficulty in realizing overlap with a stick decomposition under MPI has been documented [13]. We are investigating an alternative implementation to support sticks and thus improve scalability.

4 Discussion

Graph-based execution models began appearing in the 1970s with classic dataflow [2–4]. A large grain variant followed [5]. SMARTS [14] integrated task and data parallelism and provided an API for coarse-grain macro-dataflow. It has been demonstrated on shared memory only. OSCAR[15] had similar goals to SMARTS, but operated on static (compile time) graphs. CILK[16] and Mentat[17] treated functional parallelism. SciRun[18] and UIntah[19] support graphical composition of data flow graphs of components and dynamic load balancing of task graphs. Tarragon (Cicotti and Baden) [20] supports fine grain communication and was originally targeted to cell microphysiology. Husbands and Yelick [21] demonstrated thread scheduling techniques for tolerating latency in dense LU factorization.

In the context of this prior work (and similar to Tarragon) Thyme's contribution is a systematic approach to supporting latency tolerance on distributed

memory via the dataflow model, and the ability to modify scheduling behavior by annotating the graph with meta-data. Compared with traditional split phased encoding, Thyme provides an abstract description of the underlying virtual process structure that may be manipulated to optimize execution

To understand the role that the graph can play in tolerating latency, consider Charm++ [22] and Adaptive MPI [23]. Charm++ supports overlap via processor virtualization and asynchronous remote method invocation on shared C++ objects. AMPI (Adaptive MPI) employs virtualized MPI processes and is built on top of Charm++. None of these models employ an explicit dataflow graph to realize overlap. Charm++ employs a more general model than Thyme, embedding dependence information in the form of remote method invocations involving global objects. The task structure is implicit, however, and cannot be manipulated as a free-standing object as with Thyme. The Charm++ developers allude to difficulties with an implicit call graph, in particular, in coupling multiple graphs. Charisma [24] was developed to meet this need.

When a Charm++ method invocation blocks, or an AMPI receive blocks, the thread yields to another, which may block for the same reason. Indeed, our efforts to run an MPI variant of Jacobi3D under AMPI on DataStar failed to improve performance–and actually slowed it down in some cases. Thyme's meta-data offer an improvement over virtualization by taking the guesswork out of scheduling. They inform the scheduler about tasks that have all their input data ready, avoiding the guesswork of the "block and yield" model. If the number of cores per processor continues to grow over time, then the significance of informed scheduling will continue to grow as well.

5 Conclusions and Future Work

We have demonstrated that a data-driven formulation enables an application to tolerate latency without embedding scheduling and other policy decisions into the code. The approach provides an opportunity for increased performance because it allows the user to experiment with alternative implementation policies, including application-specific task scheduling in MMULT and tuned data decompositions in Jacobi3D. Thyme admits the use of newly arrived data to enable computation, masking the latency of data still in transit.

Our current implementation is restricted to Cartesian geometries, but covers a range of uniform and irregular problems including: uniform and multi-level finite-difference methods, such as structured adaptive mesh refinement and multigrid, and mesh-based particle methods (but not "tree codes" [25]). So called unstructured finite element methods need a different type of iteration space representation, but the general principles apply.

Systems with many core CPUs will benefit from Thyme's programming model, in particular, thread-aware schedulers that treat symbiosis and cache locality. A steady growth in on-chip parallelism will put pressure on communication subsystems, while at the same time providing an opportunity to optimize execution by expending inexpensive processing cycles.

Although the Thyme API is compact–only a couple of thousand lines of C++– programmers can obtain the benefits of the model without having to learn an entirely new API. We envision that Thyme will become part of a run time support library for a compiler or application library. To this end, we are currently investigating source-to-source translation techniques using Quinlan's ROSE [26] infrastructure. ROSE enables the user to access and transform the abstract syntax tree (AST); full knowledge of function arguments and dependence information is available for subsequent analysis and transformation. Translation support can realize semantic level optimizations on the Thyme library classes and automatically generate calls to the Thyme API. The application programmer can thereby obtain the benefits of Thyme's graph-driven execution model while remaining aloof of many of the details.

Acknowledgments

The authors wish to thank Daniel Quinlan for supervising Jacob Sorensen during visits to the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory. Dan's contributions to this work were invaluable. This work was supported by the United States Department of Energy under Award Number DE-FG02-05ER25706 and Prime Contract No. W-7405-ENG-48. Computer time on DataStar was provided by the San Diego Supercomputer Center and the University of California, San Diego.

References

1. Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A high-performance, portable implementation of the MPI message passing interface standard. Technical report, Argonne National Laboratory, Argonne, IL (1997) `http://www.mcs.anl.gov/mpi/mpich/`.
2. Dennis, J.: Data flow supercomputers. IEEE Computer **13**(11) (1980) 48–56
3. J.R. Gurd and C. C. Kirkham and I. Watson: The Manchester prototype dataflow computer. Communications of the ACM **28**(1) (January 1985) 34–52
4. : Executing a program on the mit tagged-token dataflow architecture. IEEE Transactions on Computers **39**(3) (March 1990) 300–318
5. Babb, R.G., I.: Parallel processing with large-grain data flow technique. Computer **17**(7) (July 1984) 55–61
6. Kelly, P., Beckmann, O., Field, A., Baden, S.: Themis: Component dependence metadata in adaptive parallel applications. Parallel Processing Letters **11**(4) (December 2001) 455–470
7. Little, J.D.C.: A proof of the queuing formula L = λ W. Operations Research **9** (1961) 383–387
8. Colella, P.: Defining software requirements for scientific computing (2004)
9. NASA Advanced Supercomputing Division: NAS parallel benchmarks http://www.nas.nasa.gov/Resources/Software/npb.html.
10. Sawdey, A.C., O'Keefe, M.T., Jones, W.B.: A general programming model for developing scalable ocean circulation applications. In: Proceedings of the ECMWF Workshop on the Use of Parallel Processors in Meteorology. (January 1997)

11. Baden, S.B., Shalit, D.: Performance tradeoffs in multi-tier formulation of a finite difference method. In: Proc. 2001 International Conference on Computational Science, San Francisco, CA (May 2001)
12. van de Geign, R., Watts, J.: SUMMA: Scalable universal matrix multiplication algorithm. Concurrency: Practice and Experience **9**(4) (April 1997) 255–74
13. Iancu, C.C., Strohmaier, E.: Optimizing communication overlap for high-speed networks. In: Proc. 12th ACM SIGPLAN Symp on Principles and Practice of Parallel Prog (PPoPP '07), New York, NY, USA, ACM Press (2007) 35–45
14. Vajracharya, S., Karmesin, S., Beckman, P., Crotinger, J., Malony, A., Shende, S., Oldehoeft, R., Smith, S.: Smarts: Exploting temporal locality and parallelism through vertical execution. In: International Conference on Supercomputing. (1999)
15. Kasahara, H., Yoshida, A.: A data-localization compilation scheme using partial-static task assignment for fortran coarse-grain parallel processing. Parallel Computing **24** (1998) 579–596
16. Leiserson, C., Randall, K., Zhou, Y.: Cilk: An efficient multithreaded runtime system. In: Proc. Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), Santa Barbara, CA (July 1995)
17. Grimshaw, A.S., Weissman, J.B., Strayer, W.T.: Portable run-time support for dynamic object-oriented parallel processing. ACM Transactions on Computer Systems **14**(2) (1996) 139–170
18. Johnson, C., Parker, S., Weinstein, D., Heffernan, S.: Component-based, problem solving environments for large-scale scientific computing. Concurrency and Computation: Practice and Experience **14**(13-15) (2002) 1337–1349
19. McCorquodale, J., de St. Germain, D., Parker, S., Johnson, C.: The uintah parallelism infrastructure: A performance evaluation. In: High Performance Computing 2001, Seattle (March 2001)
20. Cicotti, P., Baden, S.B.: Asynchronous programming with tarragon. In: Proc. 15th IEEE International Symposium on High Performance Distributed Computing, Paris, France (Jun. 2006)
21. Husbands, P., Yelick, K.: Multithreading and one-sided communication in parallel lu factorization. In: SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing, Reno, Nevada, ACM (Nov. 2007)
22. Phillips, J.C., Zheng, G., Kumar, S., Kalé, L.V.V.: NAMD: Biomolecular simulation on thousands of processors. In: Proceedings of SC 2002. (2002)
23. Huang, C., Lawlor, O., Kalé, L.: Adaptive mpi. In: Proc. 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03). (2003)
24. Huang, C., Kale, L.: Charisma: orchestrating migratable parallel objects. In: HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing, New York, NY, USA, ACM Press (2007) 75–84
25. Warren, M.S., Salmon, J.K.: A parallel hashed oct-tree n-body algorithm. In: Supercomputing. (1993) 12–21
26. Quinlan, D., Miller, B., Philip, B., Schordan, M.: A c++ infrastructure for automatic introduction and translation of openmp directives. In: Proc. 16th Inte. Parallel and Distrib. Proc. Symp. (April 2002) 105–114