Application Representations for Multi-Paradigm Performance Modeling of Large-Scale Parallel Scientific Codes

Vikram Adve *and* Rizos Sakellariou
Department of Computer Science
Rice University
{adve,rizos}@cs.rice.edu

1 Introduction

Recent years have seen significant strides in individual performance modeling techniques for parallel applications and systems. Researchers are now beginning to focus on developing comprehensive, end-to-end modeling environments that bring together multiple techniques to enable practical performance modeling for large, real-world applications on very large scale systems. For example, the POEMS project aims to create such an environment for the end-to-end modeling of large parallel applications on complex parallel and distributed systems [8].

The key goal of the POEMS project is to enable compositional development of end-to-end performance models, based on component models from the three domains of application software, runtime systems, and hardware. Furthermore, in order to enable comprehensive and yet efficient modeling of large applications, the POEMS framework aims to integrate component models from multiple modeling paradigms (analytical modeling, simulation, and actual program execution), *and* to allow individual system components to be modeled at multiple levels of detail. The environment will support a wide range of performance modeling techniques including detailed execution-driven simulation of message-passing programs and scalable I/O [5], program-driven simulation of single-node performance and memory hierarchies, abstract analytical models of parallel programs such as LogGP [19], and detailed analytical models of parallel program performance based on deterministic task graph analysis [3].

An overview of the POEMS framework and methodology is available in [8]. Several components of the framework are described elsewhere in this issue including the specification language for compositional development of models [6], an overview of the parallel simulation capability for detailed performance prediction of large scale scientific applications [4], and the LogGP model for Sweep3D (an ASCI neutron transport code that has been an initial driving application for the POEMS project) [19].

An important part of a comprehensive performance modeling environment such as POEMS is an application representation that can support the diverse modeling capabilities envisaged for this environment. The application representation must provide a description of program behavior that can serve as a common source of workload information for any modeling technique or combination of modeling techniques to be used in a particular experiment. Directly using the application source code as this representation is inadequate because it can directly support only program-driven modeling techniques such as execution-driven simulation. For computing analytical model input parameters, it would be too expensive to parse and analyze the source code directly for each experiment. Instead, we require an abstract representation that can be efficiently converted into these model parameters, while also retaining sufficiently detailed information (including portions of source code) to enable detailed simulation studies. Ideally, the application representation would precisely capture all relevant information that affects the performance of the program so that the representation itself does not introduce any *a priori* approximations into the system. Finally, the application representation must be efficient and flexible enough to capture complex, large-scale applications with high degrees of parallelism and sophisticated parallelization strategies.

This paper describes the design of the application representation in the POEMS environment. We begin by describing in more detail the key goals that must be met by this design (Section 2). Section 3 then describes the main design features of the application representation in POEMS, and discusses how these design features meet the major challenges that must be faced for advanced, large-scale applications. We conclude with a brief overview of related work and a description of status and future plans.

2 Goals of the Application Representation

We can identify four key goals that must be met by the design of the application representation for a comprehensive performance modeling environment such as POEMS. These goals are as follows.

First, and most important, the application representation must be able to support a wide range of modeling techniques from abstract analytical models to detailed simulation. In particular, it should be possible to compute the workload for each of these modeling techniques from the application representation, as noted in the Introduction. The workload representations required for the different modeling techniques in POEMS vary widely [8]. The execution-driven simulation tools for modeling communication performance and memory hierarchy performance require access to the actual source code, both for individual sequential tasks and for communication operations. Deterministic task graph analysis requires a dynamic task graph representation consisting of sequential task nodes, task precedence edges, and communication events, together with numerical parameters describing task computation times and communication demands. Finally, the LogP and LogGP models have built-in information about the synchronization structure of the code and mainly require numerical parameters describing task computation times and communication demands.

Second, it must be possible to use parallelizing compiler technology to automate (partially or completely) the process of computing the application representation. This will be essential for large-scale real-world applications, where the size and complexity of the representation would make it impractical to compute it manually. It will also be essential if such a complex modeling environment is to be accessible to end-users. It is not reasonable to expect end-users to have a detailed understanding of the application representation, or to have any significant expertise in any of the modeling techniques being applied.

Third, the representation must be concise and efficient enough to support modeling terascale applications on very large parallel systems. This means that the representation must be able to capture program behavior for large problem sizes and system configurations, and for programs with high degrees of parallelism. Furthermore, the representation must be able to capture program behavior for adaptive algorithms, which are expected to be the algorithms of choice for large scale computations. The major challenge here is that the parallelism, communication and synchronization in these algorithms may not be predictable statically, but may depend on intermediate results during the evolution of the computation. This could mean, for example, that predicting the precise runtime behavior of the program might require actual execution of significant portions of the computation.

Finally, the representation should be flexible enough to support performance prediction studies that can predict the impact of changes to the application, particularly changes to the parallelization strategy, communication and scheduling. (Note that changes to system features will be captured by other components of POEMS from the operating system and hardware domains.)

3 The Application Representation in POEMS

The application representation in POEMS has been designed with a view towards meeting all of the goals described in the previous section. The representation is based on the *task graph*, a widely used representation of parallel programs for many purposes including performance modeling [3, 10], task scheduling [20], and graphical programming languages [12, 7]. The task graph provides an abstract yet precise description of parallelism, communication and synchronization, while permitting the sequential parts of the computation (the "tasks") to be represented at almost arbitrary levels of detail, ranging from a single execution time number to exact execution of object code. To meet the diverse goals of POEMS, we use two flavors of the task graph, the *static* and the *dynamic* task graph. These and other key terms are defined in the following section. Section 3.2 briefly describes the components of the application representation. Section 3.3 then discusses how we expect to meet the challenges raised by the above goals. Finally, Section 3.4 gives an example of the task graphs for Sweep3D.

3.1 Definitions of Key Terms

Task: A unit of work in a parallel program that is executed by a single thread, and such that any precedence relationship between a pair of tasks only arises at task boundaries.

Thread: A logical or actual entity that executes tasks of a program. In a multi-threaded system, threads will be actual operating-system entities scheduled onto processes. In a single-threaded application, threads may not actually exist but we use the term "thread" for uniformity (then threads and processes are logically mapped one-to-one).

Process: An operating system entity that "executes" the threads of a program. Also the entity that is scheduled onto processors.

Static Task Graph: (For a given program) A hierarchical graph in which each vertex represents either a task node, a control-flow node (loop or branch), or a subgraph. Each edge represents a precedence between a pair of nodes, which may be enforced either by control-flow or synchronization. Each task node in the graph may actually represent a set of parallel tasks (since the degree of parallelism may be unknown before execution time).

Informally, this is a representation of the *static* parallel structure of the program. In particular, this representation is defined only by the program and is independent of runtime input values or computational results. Note that such a graph must include control-flow (both loops and conditional branches) since it has to capture all possible executions of the given program. Furthermore, it must represent many quantities symbolically, such as the number of iterations of a loop.

The static task graph is hierarchical, i.e. any node may itself represent a subgraph, in order to simplify the representation of separate procedures.

Dynamic Task Graph: (For a program and a particular input) A directed acyclic graph in which each vertex represents a task and each edge represents a precedence between a pair of tasks. A task can begin execution only after all its predecessor tasks, if any, complete execution.

A dynamic task graph provides a representation of the *dynamic* parallel structure of the program, and therefore it depends on the specific program input.

Task Scheduling: The allocation of dynamic tasks to threads. The task scheduling *strategy* is implicitly or explicitly specified by the parallel program, even though the actual resulting schedules may be dynamic, i.e., data-dependent and/or timing-dependent. Examples of task scheduling strategies include static, block partitioning of loop iterations (as in Sweep3D) or dynamic scheduling techniques such as guided self-scheduling or explicit task queues.

Condensed Dynamic Task Graph: (For a program, a particular input, and a particular allocation of tasks to threads) A directed acyclic graph in which each vertex denotes a collection of tasks executed by a single thread, and each edge denotes a precedence between a pair of vertices (i.e. all the tasks in the vertex at the head of the edge must complete before any task in the vertex at the tail can begin execution).

This representation may be important because capturing all the fine-grain parallelism in the program (e.g., all individual loop iterations) as individual tasks might be too expensive for very large problems. The condensed graph essentially collapses all the tasks executed by a thread between synchronization points into a single condensed task. Note that this graph therefore depends on the specific allocation of tasks to threads. The tradeoffs in using the condensed dynamic task graph are described below.

3.2 Features of the Application Representation

As mentioned earlier, we represent the parallelism and communication in a parallel program using two flavors of the task graph that serve different purposes. The *static task graph* provides a compile-time representation that is designed to be synthesized using parallelizing compiler technology. The *dynamic task graph* provides a detailed description of runtime program behavior, which can be used for abstract or detailed performance modeling [3].

The static task graph provides a concise description of program parallelism even for very large programs. The types of graph vertices are listed in Section 3.1. Each task node in the graph can be either a computational task or a communication task. Communication tasks represent the processing overhead incurred by each thread when performing message-passing communication. Each vertex in the static task graph has substantial symbolic information associated with it to support performance modeling in subsequent stages.

Each task node may actually represent a set of parallel task instances since the number of parallel instances may not be known at compile-time. Each parallel task node therefore contains a symbolic integer set describing the set of parallel task instances possible at runtime. For example, consider a "left shift" communication operation on a 1-dimensional processor grid with P threads numbered $0, 1, \ldots, P-1$. In this pattern, each thread t sends data to thread $t-1, t > 0$. Therefore, the SEND task node in the static task graph represents the set of task instances $\{[i] : 1 \leq i \leq P-1\}$; the RECEIVE task node represents the set of instances $\{[i] : 0 \leq i \leq P-2\}$.

Each task node also contains a symbolic scaling function describing how task execution time varies with program input values and internal variables. Each loop node has symbolic expressions describing loop bounds and stride. Each conditional branch has a symbolic conditional branch expression. Each procedure of the program is represented as a separate subgraph, with special CALL nodes representing call sites. Finally, a communication event descriptor (separate from the graph) describes all the information related to a particular logical communication event, which is performed by several communication tasks in the graph. This descriptor includes a symbolic expression describing the communication size, as well as information about the logical pattern of communication (e.g., Shift, Pipelined Shift, Broadcast, etc.). The set of tasks required to implement a single logical communication event depends on the communication pattern and on the communication primitives used to perform the communication operations (e.g., blocking vs. nonblocking sends and receives).

Because each task in the static task graph may represent multiple parallel task instances, each edge of the static graph must also represent multiple edge instances. Furthermore, the mapping between task instances at the source and sink of the edge may not be a simple identity mapping. For example, in the 1-dimensional shift communication above, the mapping between SEND and RECEIVE task instances can be described as: $\{[i] \rightarrow [j] : 1 \leq i \leq P-1 \wedge j = i-1\}$ This symbolic integer mapping describes a set of edge instances connecting exactly those pairs of SEND and RECEIVE task instances corresponding to processors pairs that communicate during the SHIFT.

Finally, the scalar execution behavior of individual tasks is described either by the source code of the tasks itself, or more abstractly by a set of numerical parameters. The actual parameters required will depend on the modeling techniques to be used, and will vary from architecture-independent parameters such as the number of floating point operations between consecutive memory operations, to architecture-dependent parameters such as the number of cache misses or the actual task execution time (per loop iteration).

The dynamic task graph can be thought of as being instantiated from the static task graph for a particular program input by instantiating the parallel instances of the tasks, unrolling all the loops, and resolving all dynamic branch instances. The dynamic task graph provides a precise description of the actual runtime program behavior for a given program input. It describes the actual tasks executed at runtime, the precedences between them, and the precise communication operations executed. The symbolic scaling functions in the static task graph can be evaluated to provide cost estimates for the tasks, except that some constants (e.g., computation time per loop iteration) may have to be measured or predicted analytically. The challenges in computing the dynamic task graph and in ensuring efficient handling of large problem sizes are addressed below.

3.3 Challenges Addressed by the Application Representation

There are a few major challenges that must be addressed in using the application representation for large, real-world programs:

- The size and scalability of the representation for large problems;
- Representing adaptive codes; and
- Computing the representation automatically with a parallelizing compiler.

Below, we briefly discuss how the design of the representation addresses these challenges. The last of these (compiler techniques for computing the representation) is described in more detail in [2].

In terms of the size of the representation, the main challenge lies in the dynamic task graph. The size of the static task graph is only proportional to the size of the source code, and should not be an issue. To ensure that the size of the dynamic task graph is manageable for very large problems, we use a condensed form of the dynamic task graph defined earlier. Informally, this makes the size of each parallel phase proportional to the number of physical processors instead of being a function of the degree of fine-grain parallelism in the problem. For an example, see Section 3.4.

Nevertheless, there are some tradeoffs in using the condensed dynamic task graph. First, it must be instantiated separately for each number of processes or threads. Second, the condensed graph depends on the actual allocation of fine-grain tasks to threads and therefore may be practical to construct only for *static* task scheduling algorithms where this allocation can be computed statically. Message-passing programs, however, typically use static scheduling. Furthermore, the condensed graph seems to be a more natural representation for message-passing codes because the communication (which also implies synchronization) is usually written explicitly between processes. For shared-memory programs, the restriction to static scheduling can be a significant drawback, because there can be important performance tradeoffs between sophisticated dynamic task scheduling strategies for which the condensed graph would be very difficult to compute [3].

A second approach for supporting very large codes is to instantiate the dynamic task graph "on the fly," instead of precomputing it. (This is similar to the runtime instantiation of tasks in graphical parallel languages such as CODE [12]). Furthermore, this approach can be combined with the condensed graph described above. This approach may be too expensive, however, for simple analytical modeling since the cost of instantiating the graph may greatly outweigh the time savings in using simple analytical models.

The main challenge in supporting adaptive codes is that the dynamic task graph structure may depend on intermediate computational results of the program. This may imply that a significant part of the computation has to be executed to construct the graph and to estimate communication and load balancing parameters for intermediate stages of execution. There are two approaches to computing the dynamic task graph. First, we can precompute and store the graph during an actual execution of the program, and also measure the values of these parameters. Alternatively, for any execution-driven modeling study, we can synthesize the task graph on the fly using intermediate results of the execution. Both these approaches would require significant additional compiler support to instrument the program for collecting the relevant information at execution time.

Finally, a key goal mentioned earlier is to use parallelizing compiler technology to compute the static and dynamic task graph automatically for a given program. We have extended the Rice dHPF compiler infrastructure [1] to construct the static task graph for (non-adaptive) programs in High Performance Fortran (HPF), and to instantiate the dynamic task graph; further details are given in Section 4.

3.4 An Example of Task Graphs: Sweep3d

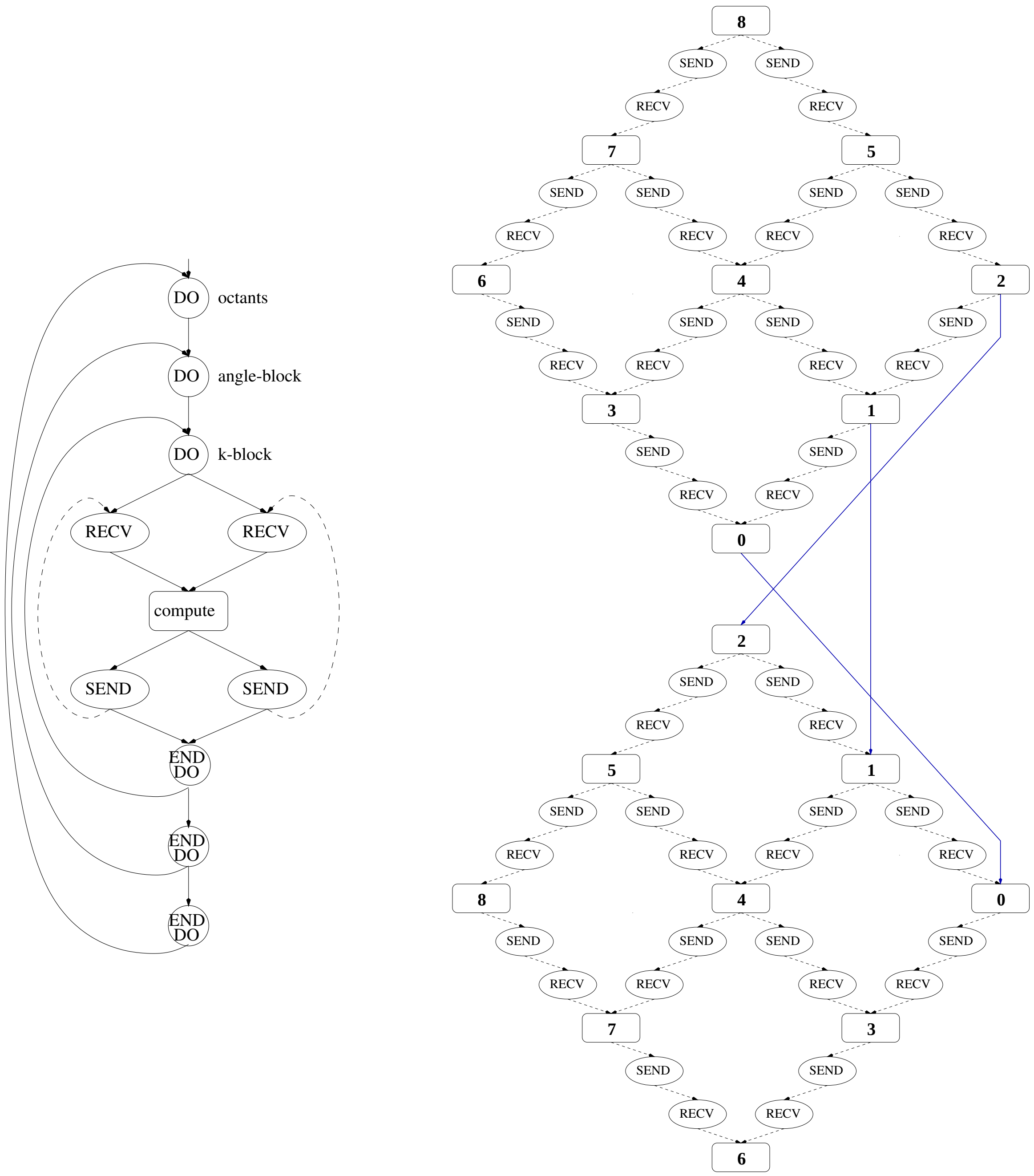
In order to illustrate the above, consider again Sweep3D (which was mentioned in the Introduction) [11] The main body of the code, that is the subroutine `sweep()`, consists of a wavefront computation on a 3-dimensional grid of cells. The subroutine computes the flux of neutron particles through each cell along several possible directions (discretized angles) of travel. The angles are grouped into 8 octants corresponding to the 8 diagonals of the cube. Along each angular direction, the flux of each interior cell depends on the fluxes of three neighboring cells. This corresponds to a 3-dimensional pipeline for each angle, with parallelism existing between angles within a single octant. The current version of the code partitions the i and j dimensions of the domain among the processors. To improve the balance between parallel utilization and communication in the pipelines, the code blocks the third (k) dimension and also uses blocks of angles within each octant.

The static task graph for the main body of the code is shown in Figure 1(a). Each node of the graph represents a different task node, where circles correspond to control flow operations, ellipses to communication operations, and rectangles to computation (each rectangle represents a condensed task node, viz., a task node in the static task graph that will be instantiated into several instances of condensed tasks in the condensed dynamic task graph). Solid lines denote those precedence edges of the task graph that are enforced implicitly by intraprocessor control flow while the dotted lines denote those that require interprocessor communication.[1] The program uses blocking communication operations (MPI_Send and MPI_Recv), each of which is represented by a single communication task.

Figure 1(b) shows part of the condensed dynamic task graph on a 3×3 processor grid (recall that loops are fully unrolled in the dynamic task graph). The graph corresponds to the last wavefront of octant 2 (top) and the first wavefront of octant 3 (bottom). The number inside each computation task corresponds to the processor executing that task.

Using the condensed form of the dynamic task graph described earlier (see Section 3.3), the size of the dynamic task graph can be kept reasonable. Thus, for the the subroutine `sweep()` with a 50^3 problem size on a 2×3 processor grid, the dynamic task graph would contain over 24×10^6 tasks, whereas the condensed

[1] Note that this distinction is possible only after the allocation of tasks to processes is known. This is known at compile-time in many message-passing programs, including Sweep3D, because they use a static partitioning of tasks to processes.



(a) Static task graph

(b) Part of dynamic task graph on a 3×3 processor grid.

Figure 1: Task graphs for Sweep3D. Solid lines depict control-flow; dashed lines depict communication.

dynamic task graph has only 3570 tasks.

4 Implementation and Status

The implementation of the application representation requires several software components. These include:

- A library implementing the task graph data structures, including the static and dynamic task graphs, and external representations for a symbol table, symbolic sets, and scaling functions.

- An extended version of the Rice dHPF compiler to construct the task graph representation for High Performance Fortran (HPF) and MPI codes, as described briefly below.
- A translator that converts the static or dynamic task graph into the graphical programming language, CODE [12]. CODE will be used for representing task graphs within the POEMS environment because it provides an existing infrastructure for editing, manipulating, and executing graph-based representations of parallel programs.
- Interfaces to generate workload information for each modeling technique in POEMS from the application representation in CODE.
- Performance measurement support to obtain numerical task measures required for a given model, such as serial execution time, cache miss rates, etc. This will require compiler support for instrumentation and runtime support for data collection. An alternative would be to use compiler support for *predicting* these quantities, which would be valuable for modeling future system design options.

At present, the data structures for static and dynamic task graphs have been constructed, and a prototype version of the dHPF compiler has been developed [2]. The dHPF compiler directly synthesizes the static task graph and associated symbolic information from the HPF program, after the code has been transformed for parallelization. The compiler also instantiates a dynamic task graph from the static task graph; this operation uses a key capability of the dHPF infrastructure, namely the ability to generate code to enumerate symbolic integer sets and mappings. Finally, the compiler incorporates techniques to condense the task graph as follows.

Condensing the task graph happens in two stages in dHPF. First, before instantiating the dynamic task graph, we significantly condense the *static* task graph to collapse any sequence of tasks (or loops) that do not include communication or "significant" branches. (Significant branches are any branches that can impact the execution time of the final condensed task.) This ensures that the initial dynamic task graph instantiated from this static graph is already significantly condensed. Then, after instantiating the dynamic task graph, we further condense this graph as follows. Note that, when instantiating the dynamic task graph, any significant branches are interpreted and resolved (i.e., eliminated from the graph). This may produce sequences of tasks allocated to the same process that are not interrupted by branches. These sequences are further condensed in this second step. The graph resulting from both the above steps is the final condensed task graph, as defined earlier.

Our two immediate goals are to translate this representation into the CODE language, and to interface it to the deterministic task graph analysis model [3] using preliminary support for performance measurement.

5 Related Work

We provide a very brief overview of related work, focusing on application representation issues for comprehensive parallel system modeling environments. Additional descriptions of work related to POEMS are available elsewhere [2, 4, 6, 8, 19].

Many previous simulation-based environments have been used for studying parallel program performance and for modeling parallel systems (e.g., WWT [16], Maisie [5], SimOS [18], and RSIM [13]). These environments have been based on program-driven simulation, where the application representation is simply the program itself. In these systems, there are no abstractions suitable for driving analytical models or more abstract simulation models. Such models will be crucial to make the study of large-scale applications and systems feasible.

Some compiler-driven tools for performance prediction, namely FAST [9] and Parashar's interpretive framework [15], have used more abstract graph-based representations of parallel programs similar to our static task graph. Parashar's environment also uses a functional interpretation technique for performance prediction which is similar to our compile-time instantiation of the dynamic task graph for POEMS, using the dHPF compiler. Parashar's framework, however, is limited to restricted parallel codes generated by their Fortran90D/HPF compiler, namely codes that use a loosely-synchronous communication model (i.e., alternating phases of computation and global communication) and perform computation partitioning using the owner-computes rule heuristic [17]. In addition, each of these environments focuses on a single performance prediction technique (simulation of message-passing in FAST; symbolic interpretation of analytical formulas in Parashar's framework) whereas our representation is designed to drive a wide range of modeling techniques in POEMS.

The PACE performance toolset [14] includes a language and runtime environment for parallel program performance prediction and analysis. The language requires users to describe manually the parallel subtasks and computation and communication patterns and can provide different levels of model abstraction. The system is restricted to a loosely synchronous communication model.

6 Conclusion

This paper presented an overview of the design principles of an application representation for an integrated performance modeling environment, such as the one envisioned by the POEMS research project. This representation is based on a task graph abstraction, which is designed in a way that:

- provides a common source of workload information for different modeling paradigms including analytical models, simulation models and measurement, and
- can be generated automatically or with minimal user intervention using compiler support.

In our ongoing work, we aim to integrate the different components of the POEMS framework including the compiler-synthesized application representation described in this paper, the specification language, and different performance prediction models. If successful, we believe that this work can lead to the first comprehensive performance prediction capability for very large-scale parallel applications and systems.

Acknowledgements

The authors would like to acknowledge the valuable input that several members of the POEMS project have provided to the development of the overall application representation.

This work was sponsored by DARPA/ITO under contract number N66001-97-C-8533, and supported in part by DARPA and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0159.

References

[1] V. Adve and J. Mellor-Crummey. Using Integer Sets for Data-Parallel Program Analysis and Optimization. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

[2] V. Adve and R. Sakellariou. Compiler Synthesis of Task Graphs for a Parallel System Performance Modeling Environment. Technical Report CS-TR98-332, Computer Science Dept., Rice University, Dec. 1998.

[3] V. Adve and M. K. Vernon. A Deterministic Model for Parallel Program Performance Evaluation. Technical Report CS-TR98-333, Computer Science Dept., Rice University, Dec. 1998.

[4] R. Bagrodia, E. Deelman, and T. Phan. Parallel simulation of large scale parallel applications. *International Journal of High-Performance and Scientific Applications* (this issue).

[5] R. Bagrodia and W. Liao. Maisie: A Language for Design of Efficient Discrete-Event Simulations. *IEEE Transactions on Software Engineering*, Apr. 1994.

[6] J. C. Browne and A. Dube. Compositional Development of Performance Models in POEMS. *International Journal of High-Performance and Scientific Applications* (this issue).

[7] J. C. Browne, S. I. Hyder, J. Dongarra, K. Moore, and P. Newton. Visual Programming and Debugging for Parallel Computing. *IEEE Parallel and Distributed Technology*, 3(1), Spring 1995.

[8] E. Deelman, A. Dube, A. Hoisie, Y. Luo, R. Oliver, D. Sunderam-Stukel, H. Wasserman, V. Adve, R. Bagrodia, J. C. Browne, E. Houstis, O. Lubeck, J. Rice, P. Teller, and M. K. Vernon. POEMS: End-to-End Performance Design of Large Parallel Adaptive Computational Systems. In *Proc. First Int'l Workshop on Software and Performance*, Oct. 1998.

[9] M. Dikaiakos, A. Rogers, and K. Steiglitz. FAST: A Functional Algorithm Simulation Testbed. In *International Workshop on Modelling, Analysis and Simulation of Computer and Telecommunication Systems – Mascots '94*, 1994.

[10] D. L. Eager, J. Zahorjan, and E. D. Lazowska. Speedup versus Efficiency in Parallel Systems. *IEEE Trans. on Computers*, C-38(3):408–423, March 1989.

[11] A. Hoisie, O. M. Lubeck, and H. J. Wasserman. Performance analysis of multidimensional wavefront algorithms with application to deterministic particle transport. Technical Report LA-UR-98-3316, Los Alamos National Laboratory, 1998.

[12] P. Newton and J. C. Browne. The CODE 2.0 Graphical Parallel Programming Language. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[13] V. S. Pai, P. Ranganathan, and S. V. Adve. The Impact of Instruction Level Parallelism on Multiprocessor Performance and Simulation Methodology. In *Proc. Third International Conference on High Performance Computer Architecture*, pages 72–83, 1997.

[14] E. Papaefstathiou, D. J. Kerbyson, G. R. Nudd, J. S. Harper, S. C. Perry, and D. V. Wilcox. A Performance Analysis Environment for Life. In *Second ACM SIGMETRICS Symposium on Parallel and Distributed Tools*, Oregon, Aug. 1998.

[15] M. Parashar, S. Hariri, T. Haupt, and G. Fox. Interpreting the Performance of HPF/Fortran 90D. In *Proceedings of Supercomputing '94*, Washington, D.C., Nov. 1994.

[16] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proc. 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 48–60, May 1993.

[17] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, Portland, OR, June 1989.

[18] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Complete Computer System Simulation: The SimOS Approach. *IEEE Parallel and Distributed Technology*, pages 34–43, Winter 1995.

[19] M. K. Vernon and D. Sunderam-Stukel. Predictive Analysis of a Complex Wavefront Application Using LogGP. *International Journal of High-Performance and Scientific Applications* (this issue).

[20] T. Yang and A. Gerasoulis. PYRROS: Static task scheduling and code generation for message passing multiprocessors. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.