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

Vikram Adve
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, U.S.A.
vadve@cs.uiuc.edu

Rizos Sakellariou
Department of Computer Science
University of Manchester
Manchester M13 9PL, U.K.
rizos@cs.man.ac.uk

1 Introduction

Recent years have seen significant strides in individual performance modeling techniques for parallel applications and systems. Each class of individual techniques has important strengths and weaknesses. Abstract, analytical models provide important insights into application performance and are usually extremely fast, but lack the ability to capture detailed performance behavior and most such models must be constructed manually which limits their accessibility to users. Program-driven simulation techniques can capture detailed performance behavior at all levels, and can be used automatically (i.e., with little user intervention) to model a given program, but can be extremely expensive for large-scale parallel programs and systems, not only in terms of simulation time but especially in their memory requirements. Finally, program measurement is an important tool for tuning existing programs and for parameterizing and validating performance models, and it can be very effective for some metrics (such as counting cache misses using on-chip counters), but it is inflexible, limited to a few metrics, and limited to available program and system configurations.

To overcome these limitations of individual modeling approaches, 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. Such an environment would support a variety of modeling techniques for each system component, and enable different models to be used for different system components within a single modeling study. Equally important, such an environment would include compiler support to automatically construct workload information that can drive the different modeling techniques, minimizing the need for user intervention.

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 [1]. In addition to the two basic goals stated above, another goal of POEMS is to enable compositional development of end-to-end performance models, using a specification language to describe the system components and a choice of model for each component, and using automatic data mediation techniques (specialized for specific model interactions) to interface the different component models. The project brings together a wide range of performance modeling techniques including detailed execution-driven simulation of message-passing programs and scalable I/O [7], program-driven simulation of single-node performance and memory hierarchies, abstract analytical models of parallel programs such as LogGP [21], and detailed analytical models of parallel program performance based on deterministic task graph analysis [5]. An overview of the POEMS framework and methodology is available in [1]. Several components of the framework are described elsewhere, including the specification language for compositional development of models [8], an overview of the parallel simulation capability for detailed performance prediction of large scale scientific applications [6], and the LogGP model for Sweep3D (an ASCI neutron transport code that has been an initial driving application for the POEMS project) [21].

An important challenge in developing such a comprehensive performance modeling environment is designing an application representation that can support the two basic goals mentioned above. In particular, 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. Instead, we require an abstract representation of the program structure that can enable different modeling techniques to be used for different program components (e.g., computational tasks, memory access behavior, and communication behavior). Nevertheless, the representation should 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. The application representation must also be efficient and flexible enough to capture complex, large-scale applications with high degrees of parallelism and sophisticated parallelization strategies.

To meet the second basic goal mentioned above, it must be possible to compute the application representation automatically for a given parallel program using parallelizing compiler technology. This requires a *compile-time* representation that is independent of program input values. In particular, this requires symbolic information about the program structure (e.g., numbers of parallel tasks, loop iterations, and message sizes). The representation must also capture part of the static control-flow in the program, in addition to the sequential computations and the parallel structure. It also requires that detailed information about a specific execution of the program on a particular input should be derivable from the static representation.

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 an 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 for large-scale parallel systems. 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 information required for different modeling techniques varies widely [1]. 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, simpler analytical modeling approaches (e.g., LogP and LogGP) 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 efficient and scalable enough to support modeling teraflop-scale 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 for such algorithms 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 [5, 12], task scheduling [22], and graphical programming languages [9, 14]. 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. Sections 3.2 and 3.3 briefly describe the components of the application representation. Section 3.4 then discusses how we expect to meet the challenges raised by the above goals. Finally, Section 3.5 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 (i.e., we assume there is one thread per process).

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

Static Task Graph or STG: (For a given program) A hierarchical graph in which each vertex is either a task node, a control-flow node (loop or branch), or a call node representing a subgraph for a called procedure. A task node captures either computation or communication. Each task node in the graph represents a set of parallel tasks (since the degree of parallelism may be unknown until execution time). Each edge represents a precedence between a pair of nodes, which may be enforced either by control-flow or synchronization.

Dynamic Task Graph or DTG: (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.

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).

A condensed static task graph can be defined analogously, where a sequence of task nodes can be collapsed if they do not include any communication and if every task node is instantiated into an identical set of dynamic task instances at run-time.

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 The Static Task Graph: A Compile-Time Representation

The *static task graph*, defined above, provides a concise description of program parallelism even for very large programs, and is designed to be synthesized using parallelizing compiler technology. 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 actually a collection of graphs, one per procedure. Each call site in the program is represented by an explicit call node in the graph, which provides information about the procedure being called and the actual parameters to the call (as symbolic expressions). Conceptually, such a node can be substituted by a sub-graph representing a particular instance of the called procedure.

The communication operations in the program are grouped into "logical communication events". For example, for a single logical SHIFT operation in a message passing program, all the send, receive, and wait operations implementing the SHIFT would be grouped into a single such event. Explicit communication task nodes are included in the STG to represent the computational overheads incurred by each thread when performing the communication operations. For each communication event, the set of tasks and the synchronization edges between them depend on the communication pattern and on the communication primitives used (e.g., blocking vs. nonblocking sends and receives). The use of explicit communication tasks has proved extremely useful for capturing complex communication patterns that can overlap communication with computation in arbitrary ways [4]. A communication event descriptor, kept separate from the STG, holds all the other information about each communication event including the logical pattern of communication (e.g., Shift, Pipelined Shift, Broadcast, Sum reduction, etc.), a symbolic expression describing the communication size, and the task indices for the communication task nodes in the STG.

Each task node (computation or communication) may actually represent a set of parallel tasks since the number of parallel tasks may not be known at compile-time. Each parallel task node therefore contains a symbolic integer set describing the set of parallel tasks 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 tasks $\{[i] : 1 \leq i \leq P-1\}$; the RECEIVE task node represents the set of tasks $\{[i] : 0 \leq i \leq P-2\}$.

Because each task node 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 that correspond to processors pairs that communicate during the SHIFT.

Finally, the scalar execution behavior of individual tasks is described both by the source code of the task itself, and more abstractly by symbolic scaling functions. The scaling function for a task node describes 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. The scaling function for each computational task includes a symbolic parameter representing the actual task execution time per loop iteration. After synthesizing the STG, the compiler can generate an instrumented version of the parallel program to measure this per-iteration time directly for all computational task nodes in the program. Additional numerical parameters can be included in the STG, both architecture-independent parameters such as the number of floating point operations between consecutive memory operations, or architecture-dependent parameters such as the number of cache misses or the actual task execution time (per loop iteration).

3.3 The Dynamic Task Graph: A Run-Time Representation

The *dynamic task graph* provides a detailed description of the parallel structure of a program for a particular program input, which can be used for abstract or detailed performance modeling [5]. The DTG is acyclic and in particular, no control-flow nodes (loops or branches) are included in the graph. Except for a few pathological examples, the DTG is *independent* of the task scheduling. In particular, different task scheduling strategies can lead to different execution behavior and therefore different performance for the same DTG. This ability of the DTG to capture the intrinsic parallel structure of the program separate from the task scheduling can be powerful for comparing alternative scheduling strategies, particularly for shared memory programs that often use sophisticated dynamic and semi-static task scheduling strategies [5].

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 DTG thus obtained 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, once the computation time per loop iteration is predicted analytically or measured. The challenges in computing the dynamic task graph and in ensuring efficient handling of large problem sizes are discussed below.

3.4 Challenges Addressed by the Application Representation

As noted in Section 2, there are a few major challenges that must be addressed if the application representation is to be used 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 compiler techniques for computing the representation are described in more detail in [4].

To address size and scalability, our main solution is to use the static task graph as the basis of the application representation. The STG is a concise (symbolic) representation whose size is only proportional to the size of the source code, and independent of the degree of parallelism or the size of the program's input data set. The STG can be efficiently computed and stored even for very large programs. Furthermore, we believe the STG can suffice for enabling many modeling approaches including supporting efficient compiler-driven simulation [2], abstract analytical models such as LogP and LogGP [1], as well as sophisticated hybrid models, e.g., combining processor simulation with analytical or simulation models of communication performance [1].

Some detailed modeling techniques (e.g., models that capture sophisticated task scheduling strategies [5], however, require the dynamic task graph. One approach to ensure that the size of the DTG is manageable for very large problems is to use the condensed DTG 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 3.5. The condensed graph seems to be a natural representation for message-passing codes because the communication (which also implies synchronization) is usually written explicitly between processes. Furthermore, message-passing programs typically use static scheduling so constructing the condensed graph is not difficult.

For many shared memory programs, however, there are some significant tradeoffs in using the condensed DTG because the condensed graph depends on the actual allocation of fine-grain tasks to threads. Most importantly, the condensed graph would be difficult construct for dynamic scheduling techniques where computing the allocation would require a detailed prediction of the execution sequence of all tasks in the program. For shared-memory programs, this be a significant drawback because such programs sometimes use sophisticated dynamic task scheduling strategies and there can be important performance tradeoffs to be evaluated in choosing a strategy that achieves high performance [5].

An alternative approach for computing the DTG for very large codes is to instantiate the graph "on the fly" during the model solution, instead of precomputing it. (This is similar to the runtime instantiation of tasks in graphical parallel languages such as CODE [14].) 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 second main challenge, namely supporting adaptive codes, arises because the parallel behavior of such codes depends on intermediate computational results of the program. This could mean that a significant part of the computation has to be executed to construct the DTG and to estimate communication and load balancing parameters for intermediate stages of execution. Although the STG is independent of runtime results, any modeling technique that uses the STG would have to account for the runtime behavior in examining and using the STG and would be faced with the same difficulty (e.g., see [2]). There are two possible approaches to this problem. First, for any execution-driven modeling study, we can determine the runtime parallel behavior on the fly from intermediate results of the execution. Alternatively, for models that require the DTG, we can precompute and store the DTG during an actual execution of the program, and measure the values of the above parameters. Both these approaches would require significant additional compiler support to instrument the program for collecting the relevant information at execution time.

Finally, an important goal in designing the application representation, as mentioned earlier, is to be able to compute the representation automatically using parallelizing compiler technology. The key to achieving this goal is our use of a static task graph that captures the parallel structure of the program using extensive symbolic information and static control-flow information. In particular, the use of symbolic integer sets and mappings is crucial for representing all the dynamic instances of a task node or a task graph edge. The ability to synthesize code directly from this representation is valuable for instantiating task nodes and edges for a given program input.

We have extended the Rice dHPF compiler infrastructure [3] to construct the static task graph for programs in High Performance Fortran (HPF), and to instantiate the dynamic task graph [4]. This implementation was used in a successful collaboration with the parallel simulation group at UCLA. This work has led to a very substantial improvement to the state of the art of parallel simulation of message passing programs [2]. The compiler implementation and its use for supporting efficient simulation are described briefly in Section 4.

3.5 An Example of Task Graphs: Sweep3d

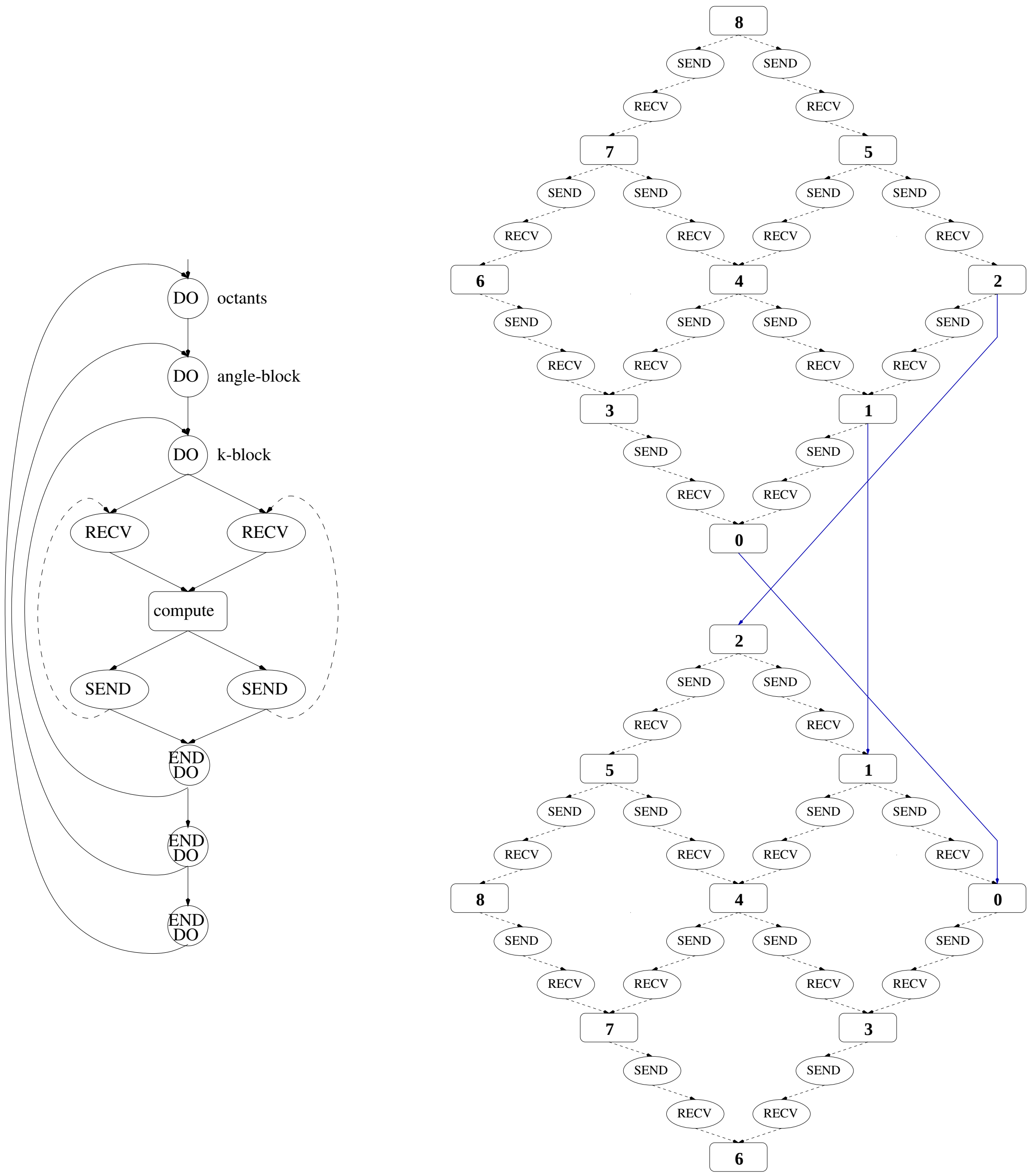
In order to illustrate the above, consider again Sweep3D (which was mentioned in the Introduction) [13] The main body of the code, that is the subroutine `sweep()`, consists of a wavefront computation on a 3-dimensional grid of cells. The subroutine computes the flux of neutron particles through each 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 3.4), the size of the dynamic task graph can be kept reasonable. Thus, for the the subroutine `sweep()` with a 50^3 problem size

[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.

on a 2×3 processor grid, the dynamic task graph would contain over 24×10^6 tasks, whereas the condensed 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.
- 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.
- Interfaces to generate workload information for different modeling techniques in the performance modeling environment.

The application representation has been implemented in an extension of the Rice dHPF compiler. The compiler constructs task graphs for MPI programs generated by the dHPF compiler from an input HPF program, and successfully captures the sophisticated computation partitionings and optimized communication patterns generated by the compiler. The compiler techniques used in this implementation are described in more detail elsewhere [4]. Briefly, the compiler first synthesizes the static task graph and associated symbolic information, after the code has been transformed for parallelization. The compiler then optionally instantiates a dynamic task graph from the static task graph, if this is required for a particular modeling study. This instantiation 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.) Second, if the compiler instantiates the dynamic task graph, it further condenses 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.

We have successfully used the compiler generated static task graphs to improve the performance of MPI-Sim, a direct-execution parallel simulator for MPI programs developed at UCLA [2]. In order to integrate the two systems, additional support has been added to the dHPF compiler. Based on the static task graph, the compiler identifies those computations within the computational tasks whose *results* can affect the performance of the program (e.g., computations that compute loop bounds or affect the outcomes of significant branches). The compiler then generates an abstracted MPI program from the static task graph in which these "essential" computations are retained but the other computations are replaced with symbolic estimates of their execution time. These symbolic estimates are derived directly from the scaling functions for each computational task along with the per-iteration execution time for the task. The compiler also generates an instrumented version of the parallel MPI program which measures the per-iteration execution time for the tasks. The non-essential computations do not have to be simulated in detail, and instead the simulator's clock can simply be advanced by the estimated execution time. Furthermore, the data used only in non-essential computations do not have to be allocated, leading to potentially large savings in the memory usage of the simulator.

Experimental results showed dramatic improvements in the simulator's performance, without any significant loss in the accuracy of the predictions. For the benchmark programs we studied, the optimized simulator requires factors of 5 to 2000 less memory and up to a factor of 10 less time to execute than the original simulator. These dramatic savings allow us to simulate systems or problem sizes 10 to 100 times larger than is possible with the original simulator, with little loss in accuracy. For further details, the reader is referred to [2].

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 [1, 4, 6, 8, 21].

Many previous simulation-based environments have been used for studying parallel program performance and for modeling parallel systems (e.g., WWT [18], Maisie [7], SimOS [20], and RSIM [15]). 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 [11] and Parashar's interpretive framework [17], 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 [19]. 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.

The PACE performance toolset [16] 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. This system too is restricted to a loosely synchronous communication model.

Finally, the PlusPyr project [10] has proposed a parameterized task graph as a compact, problem size independent representation of some frequently used directed acyclic task graphs. Their representation has some important similarities with ours (most notably the use of symbolic integer sets for describing task instances, and the use of symbolic execution time estimates). However, their representation is mainly intended for program parallelization and scheduling (PlusPyr is used as a front-end for the Pyrros task scheduling tool [22]). Therefore, their task graph representation is designed to first extract fine-grain parallelism from sequential programs using dependence analysis, and then to derive communication and synchronization rules from these dependences. In contrast, our representation is designed to capture the structure of arbitrary message-passing parallel programs independent of how the parallelization was performed. It is geared towards the support of detailed performance modeling. A second major difference is that they assume a simple parallel execution model in which a task receives all inputs from other tasks in parallel and sends all outputs to other tasks in parallel. In contrast, we capture much more general communication behavior in order to describe realistic message-passing programs.

6 Conclusion

This paper presented an overview of the design principles of an application representation that can support an integrated performance modeling environment for large-scale parallel systems. This representation is based on a task graph abstraction, and is designed to:

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

The dHPF compiler has been extended to construct the task graph representation automatically for MPI programs generated by the dHPF compiler from an HPF source program. The compiler-generated static task graph has been used successfully to obtain very substantial improvements to the state-of-the-art of parallel simulation of message passing programs.

In our ongoing work, we aim to explore other uses of the compiler-synthesized task graph representation. We are integrating the representation into the POEMS environment where it will form an application-level model component within an overall model. The task graphs have already been interfaced with the parallel simulator MPI-Sim for the work referred to above. We also aim to integrate the task graph representation with an execution driven processor simulator to model individual task performance on future systems. The processor simulation model and the message-passing simulation model could then be combined; in fact, the task graph representation provides a common representation that makes it straightforward to combine modeling techniques in this manner. If successful, we believe that this work would lead to the first comprehensive and fully automatic 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. The work described in the paper was carried out while the authors were with the Department of Computer Science at Rice University.

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, and by the NSF Next Generation Software program under grant number EIA-9975024.

References

[1] V. Adve, R. Bagrodia, J. C. Browne, E. Deelman, A. Dube, E. Houstis, J. Rice, R. Sakellariou, D. Sundaram-Stukel, P. Teller, and M. K. Vernon. POEMS: End-to-End Performance Design of Large Parallel Adaptive Computational Systems. To appear in *IEEE Trans. on Software Engineering (special issue on Software and Performance)*.

[2] V. Adve, R. Bagrodia, E. Deelman, T. Phan, and R. Sakellariou. Compiler-Supported Simulation of Highly Scalable Parallel Applications. In *Proceedings of SC99: High Performance Networking and Computing*, Portland, OR, Nov. 1999.

[3] 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.

[4] 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. Revised May 2000. Also available via `http://www-sal.cs.uiuc.edu/~vadve/Papers/tgsynthesis.ps`.

[5] 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. Also available via `http://www-sal.cs.uiuc.edu/~vadve/Papers/detmodel.ps.gz`

[6] R. Bagrodia, E. Deelman, S. Docy, and T. Phan. Performance Prediction of Large Parallel Applications Using Parallel Simulations. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, May 1999.

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

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

[9] 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.

[10] M. Cosnard and M. Loi. Automatic Task Graph Generation Techniques. *Parallel Processing Letters*, 5(4):527–538, Dec. 1995.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] D. Sundaram-Stukel and M. K. Vernon. Predictive Analysis of a Wavefront Application Using LogGP. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, May 1999.

[22] 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.