# *POEMS Report for Tasks 4.1 and 4.2*
# Compiler Synthesis of Task Graphs for Data-Parallel Applications
*Vikram Adve*

## 1 Introduction

The goal of task 4 is to develop a compiler-synthesized application representation that can be used as a common workload representation for the diverse modeling techniques in POEMS [2]. The POEMS researchers at Rice University[1] have designed such an application representation based on abstract static and dynamic task graphs, and developed and prototyped techniques to synthesize these task graphs in the Rice dHPF compiler [5,6]. The static task graph (STG) and its automatic synthesis via a parallelizing compiler now form a key component of the POEMS methodology [2]. These capabilities have subsequently had significant impact on the state of the art of simulation of highly scalable applications and systems [3], as described in more detail in the report for task 6 (and outlined briefly below).

Subtasks 4.1 and 4.2 collectively focus on task graphs for data-parallel codes written in High Performance Fortran (HPF) and automatically parallelized by dHPF. The two subtasks aimed to consider single-procedure and multi-procedure programs separately, but the research and software effort for these subtasks has largely been a common one. These subtasks have been completed, and the original goals of these subtasks have been met. This report briefly describes the effort for these two subtasks and the research results and software developed. In the near future, the compiler will be extended to generate task graphs for explicit message-passing codes (where possible), which forms subtask 4.3.

The next section describes the design of the task-graph-based application representation in POEMS. Section 3 describes the compiler techniques for synthesizing this application representation for data-parallel (HPF) programs automatically parallelized by dHPF. Section 4 briefly highlights the potential impact of this research, and the research and software artifacts resulting from this work. Finally, Section 5 describes our current research directions within POEMS.

## 2 Task Graph Representations

The design of the application representation in POEMS must meet four key goals [5]. First, it must provide a common source of workload information for the various modeling techniques envisaged in POEMS. Second, the representation must be computable *automatically* using parallelizing compiler technology, although manual intervention might be required for some programs. Third, the representation must be concise and efficient enough to support modeling terascale applications on very large parallel systems. 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.

The application representation we have designed is based on a combination of static and dynamic task graphs. The static task graph (STG) is a compact, *symbolic* graph representation of the static parallel structure of a program, in which each node represents a set of parallel tasks and each edge represents a set of edge instances. The nodes represent computational tasks, CPU components of communication, or control-flow branch nodes. The edges of the graph represent control flow, explicit communication (for message-passing events), or synchronization. The graph also includes extensive symbolic information in

[1] The lead investigator for POEMS at Rice University (Vikram Adve) has recently joined the faculty of the Computer Science Department at the University of Illinois at Urbana-Champaign, where this component of the POEMS research effort will continue as previously planned.

order to capture the structure of the parallel computation as a function of parameters such as the number of processors and relevant program input variables. For example, each parallel task node contains a symbolic integer set describing the set of task instances that are executed at runtime for given parameter values (e.g., $\{[i] : 0 \le i \le P-1\}$). Each edge between parallel task nodes contains a symbolic integer mapping describing the edge instances connecting task instances (e.g., the mapping $\{[i] \to [j] : 1 \le j \le P-1 \wedge j = i+1\}$ denotes *P*-1 edge instances connecting instance *i* of the node at the tail to instance *j* of the node at the head.). Finally, control-flow nodes contain symbolic information about loop bounds or branching conditions. Together, this information enables the STG to capture the parallel structure of the program *while remaining independent of program input values*. In addition, each computational task also has an associated scaling function describing how the computational work for the task scales as a function of relevant program variables. This information can be used to support simple extrapolation of performance metrics as a function of input size, for example.

The dynamic task graph (DTG) is a directed acyclic graph that captures the precise parallel structure of an execution of the application for a given input, and is instantiated for that particular input from the STG. The DTG is important for supporting detailed and precise modeling of parallel program behavior. For programs where the tasks are statically scheduled on to the processes (as in most message-passing programs), the DTG can be represented very concisely by combining all the tasks allocated to a particular process between synchronization points into a single "task." For other cases, the DTG may need to be computed on demand (from the STG) during model evaluation.

Several features of this design are key to meeting the goals stated above:

- The STG provides detailed information about both the sequential tasks and the parallel structure of the program (which is captured by the mappings of tasks to processes and the communication and synchronization edges between tasks). This provides sufficient information for instruction-driven, trace-driven, or execution-driven simulation approaches, while the abstractions of tasks and the graph structure support both simple and detailed analytical models.
- The static structure and the powerful symbolic representation used for task and edge mappings are both key to enabling automatic compiler synthesis of the STG for a wide range of parallel programs.
- The use of scaling functions is important for analytical modeling.
- The DTG supports more detailed analytical or simulation modeling of program performance than is possible with the static task graph alone, because the DTG describes the actual execution behavior of the program for a given set of input values.
- The demand-driven instantiation of the DTG from the STG can be used to support both large-scale problems and to enable evaluation of complex irregular applications.

## 3 Techniques for the Compiler Synthesis of Static and Dynamic Task Graphs

There are three key aspects to synthesizing task graphs for a given parallel program:

1. *Synthesizing a detailed static task graph for the program.* This includes generation of communication tasks for each communication event, and generation of symbolic sets describing the processors that execute each task, and symbolic mappings describing the pattern of communication for each communication event.
2. *Condensing the STG* to reduce the size of the graph and the potential size of dynamic task graphs.
3. *Optionally, instantiating a dynamic task graph* (for a given input and system) if needed for the modeling techniques used.

The techniques used for each of these are briefly described in turn below, after a brief description of the Rice dHPF compiler infrastructure.

### 3.1 Background: dHPF Compiler Infrastructure

The Rice dHPF compiler project aims to develop aggressive compiler analysis and optimization techniques for data-parallel languages like HPF. The compiler currently compiles HPF programs to message-passing programs using MPI (Message-Passing Interface), and preliminary versions have also been developed that compile to shared-memory and to software distributed shared memory systems.

The dHPF compiler is based on two powerful frameworks for parallel program optimization. A general computation partitioning framework permits a wide class of computation partitionings (CPs) to be represented within the compiler, and supports automatic CP selection, communication analysis, and code generation for any CP in this class. An abstract integer set framework [1] enables all of the key communication analysis and parallel program optimization problems within the compiler to be expressed in terms of abstract symbolic integer set equations, and much of the code generation is performed directly from the resulting symbolic sets. The set framework is implemented using the Omega library, a symbolic integer set package for compiler development [7].

Since the task graphs describe the structure of an explicitly parallel program, the task graphs currently synthesized by dHPF specifically describe the MPI programs generated by the dHPF compiler. The fact that such an MPI program was generated from an HPF program is not particularly relevant, and we will extend the compiler to synthesize task graphs for MPI programs directly as part of subtask 4.3. The key difference between generating task graphs for compiled or hand-written MPI programs is that, for compiled MPI programs, the dHPF compiler internally has extensive analysis information about the parallel structure of the program whereas, for hand-written programs, this information would have to be generated by analyzing the MPI code. We believe that the power of the computation partitioning and integer set frameworks makes it feasible to construct this information for a wide variety of MPI codes.

### 3.2 Compiler Synthesis of the Static Task Graph

The generation of the STG is directly based on the dHPF compiler's internal representation of the program. The tasks are initially identified as individual loop iterations of the program. The initial control-flow edges are directly extracted from the control-flow in the Abstract Syntax Tree (AST), and the communication tasks and related communication descriptors are directly obtained from the communication descriptors computed for the program.

For each static task node, a symbolic integer set describes the mapping of dynamic instances of that task to processors. This mapping is directly computed from the computation partitionings chosen by the compiler; this translation is a basic feature of the integer set framework in dHPF. (For hand-written MPI programs, the equivalent computation partitionings can be derived from analysis of the SPMD control-flow in the program, while the translation of these partitionings to integer sets is unchanged.)

For each parallel communication or synchronization edge, a symbolic integer set describes the pairs of task instances that are connected by instances of that edge. This mapping is derived directly based on the semantics of each communication pattern. For example, a non-blocking point-to-point message send and a matching non-blocking receive together require a particular set of communication tasks and edges.

The final step is to construct the scaling function for each computational task and each message size. These functions are derived by the compiler by direct analysis of the loop bounds, and primarily reflect the number of iterations executed by each processor as a function of processor id variables and other symbolic program variables. These scaling functions are meant to capture application information rather than act as actual performance models. During task graph construction, no further modeling (such as the effect of cache misses or processor pipelines) takes place.

### 3.3 Condensing the Static Task Graph

The process described above produces a first-cut version of the STG. For many typical modeling studies of parallel program performance, however, a much less detailed graph will be sufficient. For instance, a coarse-grain modeling approach could assume that all operations between two communication points constitute a single task. In some cases, this condensing can be done with no loss of accuracy. For example, no accuracy will be lost in cases where all dynamic instances of the resulting collapsed task have identical computational time; that is, the workload expressions are free of any input parameters whose value changes for different instances of that task. In other cases, condensing would result in some loss of accuracy, and the goals of the modeling study should be used to dictate the degree to which tasks are collapsed.

In order to preserve precision, the computation of the scaling function of the new condensed task is of particular importance. Ignoring conditional control flow for the moment, this scaling function is the symbolic sum of the scaling functions of the collapsed tasks, each multiplied by the symbolic number of iterations of the surrounding loops, where appropriate. Finally, in the case of conditional control flow, we can use compiler techniques based on dataflow and control-flow analysis to minimize the tradeoffs between accuracy and task graph size. These techniques are described in more detail in [6].

### 3.4 Instantiating a Dynamic Task Graph for Given Inputs

For detailed analytical or simulation modeling techniques that exploit a complete dynamic task graph (e.g., [1]), the DTG for a given set of input values can be instantiated from the STG at compile time for some programs and at runtime for any program. Performing this instantiation at compile-time permits extremely efficient modeling techniques [1], but requires sophisticated code generation capabilities. The dHPF compiler has been extended to perform this instantiation at compile-time, where possible.

The DTG can be instantiated from the STG for given program input values by unrolling all of the loops, resolving the dynamic instances of all the branches, and generating all the dynamic instances of parallel tasks, edges, and communication events. The dHPF compiler unrolls loops and resolves branches using the technique of abstract interpretation, for given program inputs.

The most challenging and creative part of this effort was in generating the dynamic instances of a node and or an edge, which requires enumerating the elements of the symbolic integer set describing these instances. Performing this enumeration directly from the set *at compile time* is extremely difficult, in general. A key feature of the integer set framework in dHPF, however, is the ability to generate code for a given integer set that enumerates the elements of that set. Therefore, we generate such code for all sets of interest (in Fortran77 form), and fork a separate process to compile, link, and execute this code *(still at compile time)*. The code for each set is put into a separate subroutine, parameterized by symbolic variables that occur in that set. (For example the set of tasks of a loop would be described in terms of variables that appear in the loop bounds, which are constant for each execution of that loop.) Each such subroutine is then invoked as many times as needed with different values of these external symbolic variables. Each invocation generates a collection of integer tuples corresponding to a collection of task or edge instances, one per tuple. This technique enables compile-time synthesis of the DTG for a much larger class of programs than would be otherwise possible.

In some cases, however, the DTG depends on more complex intermediate results of the program, e.g., for irregular or adaptive programs. In such cases, the DTG must instantiated during model execution, e.g., during execution-driven simulation or direct measurement. The DTG can either be instantiated *on the fly* for use within the same model or saved offline for use with other models.

## 4 Potential Impact of the Research

The most important impact of this research to date has been a significant advance in the state of the art of simulation of parallel programs, *making it possible for the first time to simulate highly scalable programs on very large-scale systems* [3]. The results of this work are described in more detail in the report for subtask 6.3.1 and in the attached paper. Briefly, the strategy developed in this work uses analytical models derived from the STG and compiler analysis for the computational parts of a program, while using detailed simulation of the communication behavior. We have used this strategy to integrate the MPI-Sim parallel simulator at UCLA with the Rice dHPF compiler infrastructure. The integrated system can simulate unmodified High Performance Fortran (HPF) programs compiled to the Message-Passing Interface standard (MPI) by the dHPF compiler, and we expect to simulate MPI programs as well (when task graph synthesis for MPI codes is completed). For a wide range of problem and system sizes, our results show that the optimized simulator has errors of less than 17% compared with direct program measurement in all the cases we studied, and typically much smaller errors. Furthermore, it 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 and problem sizes 10 to 100 times larger than is possible with the original simulator*. Furthermore, the compiler synthesis of the STG is key to this effort because it allows us to perform such optimized simulations *automatically* for unmodified user programs.

In a continuation of this work, we are currently examining whether compiler-based optimizations that greatly reduce memory usage can make optimistic parallel simulation much more efficient. Previous studies have shown optimistic parallel simulation to be ineffective for simulating parallel program performance. We believe that the dramatic memory reductions described above could greatly reduce the checkpointing and restart overheads inherent in optimistic simulation. We are also exploring a number of other compiler-driven strategies for further improving simulation efficiency for a variety of programs.

More generally, the compiler synthesis of task graphs can be used to automate a broad variety of performance modeling techniques, including analytical models, simulation, and hybrids of the two. The use of task graphs as the workload representation within POEMS will enable us to explore some of these strategies.

Finally, we believe that the automatic synthesis of task graphs also raises the possibility of using task graphs as a key compiler representation for parallel and distributed programs. The lack of such a representation has been a key hindrance in expanding compiler support for parallel and distributed systems. Exploring such a compiler representation is a goal of our future research.

### 4.1 Research and Software Artifacts Produced

The research results of this effort have been described in the following papers and technical reports:

- The design of the POEMS methodology, in which static and dynamic task graphs play a key role [2].
- The design of the application representation in POEMS [5].
- The techniques developed for synthesizing task graphs automatically using parallelizing compiler technology [6].
- The use of task graphs in compiler-supported simulation of parallel programs [3].

The major software artifact resulting from this effort has been an extended version of the dHPF compiler that synthesizes task graphs for data-parallel programs written in HPF. This will form a key component of the POEMS environment.

We have built on this software for the effort in subtask 6.3.1, extending the system to link the dHPF compiler with the MPI-Sim parallel simulator at UCLA, as described above.

## 5 Summary and Ongoing Work

The goals of subtasks 4.1 and 4.2 of the POEMS project plan have been achieved. The dHPF compiler synthesizes condensed static and dynamic task graphs for both single-procedure and multi-procedure HPF programs, along with analytical expressions describing the scaling behavior of task computation times. This task graph representation is key to enabling the wide variety of modeling techniques in POEMS. The compiler synthesis is essential to make the POEMS approach practical for a wide class of ordinary users by automating the process of program performance modeling. One major impact of this effort to date has been to enable simulation of much larger systems than was possible previously, even with parallel direct-execution simulators. It s now possible to simulate systems or problem sizes that are 10-100 times larger than previously possible, without any user intervention or changes to the target program.

Our research within POEMS continues along several directions:

- One effort consists of subtask 4.3, and is aimed at compiler synthesis of task graphs for existing (i.e., user-written) message-passing (MPI) programs. This will greatly expand the applicability of the techniques described above.
- Second, we are collaborating with the University of Texas to integrate static task graphs as component models described in the POEMS Specification Language (PSL). As part of this effort, we will integrate task graphs with a processor-level simulator (SimpleScalar) to enable simulation of individual task performance.
- The third major effort is to continue exploring compiler support for more efficient simulation, both for parallel and distributed programs, e.g., distributed multimedia programs such as a video-on-demand service.

## References


1. Vikram Adve, “Analyzing the Behavior and Performance of Parallel Programs”, Ph.D. Thesis, Univ. of Wisconsin-Madison, UW CS Tech. Rep. #1201, Dec. 1993.
2. Vikram Adve, Rajive Bagrodia, James Browne, Ewa Deelman, Aditya Dube, John Rice, Rizos Sakellariou, David Sundaram-Stukel, Patricia Teller and Mary Vernon. “POEMS: End-to-end Performance Design of Large Parallel Adaptive Computational Systems, ” to appear in *IEEE Transactions on Software Engineering*.
3. Vikram Adve, Rajive Bagrodia, Ewa Deelman, Thomas Phan, and Rizos Sakellariou. “Compiler-Supported Simulation of Highly Scalable Parallel Applications,” To appear in *Proceedings of SC99: High Performance Networking and Computing*,. Portland, OR, Nov. 1999.
4. Vikram Adve and J. Mellor-Crummey, “Using Integer Sets for Data-Parallel Program Analysis and Optimization”, *SIGPLAN98 Conference on Programming Language Design and Implementation (PLDI)*, Montreal, June 1998.
5. Vikram Adve and Rizos Sakellariou, “Application Representations for Multi-Paradigm Performance Modeling”, *International Journal of High Performance and Scientific Applications*, to appear.
6. Vikram Adve, and Rizos Sakellariou. “Compiler Synthesis of Task Graphs for a Parallel System Performance Modeling Environment,” Technical Report CS-TR-98-332, Computer Science Dept., Rice University, December 1998.
7. Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman and David Wonnacott,, “The Omega Library Interface Guide,” University of Maryland, April 1996.
8. S. Prakash and Rajive Bagrodia, “Using Parallel Simulation to Evaluate MPI Programs,” *Winter Simulation Conference* 1998.