Compiler Synthesis of Task Graphs for Parallel Program Performance Prediction

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

Task graphs and their equivalents have proved to be a valuable abstraction for representing the execution of parallel programs in a number of different applications. Perhaps the most widespread use of task graphs has been for performance modeling of parallel programs, including quantitative analytical models [30, 22, 29, 28, 2], theoretical and abstract analytical models [15], and program simulation [14, 4]. A second important use of task graphs is in parallel programming systems. Parallel programming environments such as PYRROS [31], CODE [27], HENCE [27], and Jade [23] have used task graphs at three different levels: as a programming notation for expressing parallelism, as an internal representation in the compiler for computation partitioning and communication generation, and as a runtime representation for scheduling and execution of parallel programs. Although the task graphs used in these systems differ in representation and semantics (e.g., whether task graph edges capture purely precedence constraints or also dataflow requirements), there are close similarities. Perhaps most importantly, they all capture the parallel structure of a program separately from the sequential computations, by breaking down the program into computational "tasks", precedence relations between tasks, and (in some cases) explicit communication or synchronization operations between tasks.

If task graph representations could be constructed automatically, via compiler support, for common parallel programming standards such as Message-Passing Interface (MPI), High Performance Fortran (HPF), and OpenMP, the techniques and systems described above would become available to a much wider range of programs than they are currently. None of the systems cited above, or other systems we know, provide the capability to extract task graphs automatically for such common programming standards. The only system we know of that attempts to do so is the interpretive framework of Parashar et al. [29], which constructs task graphs for a limited class of HPF programs compiled by the Syracuse Fortran 90D compiler. Their task graph representation is restricted to a very simple loosely synchronous computational model with alternating phases of global communication and parallel computation (the model used in their compiler). Many programs use more general communication patterns that would require a much more sophisticated task graph representation and much more complex compiler techniques to synthesize such task graphs.

In the POEMS project [3], we are developing compiler-supported techniques to model the end-to-end performance characteristics of a large-scale parallel application on a large parallel system, using a combination of analytical, simulation and hybrid models, and models at multiple levels of resolution for individual components. Within the context of this work, we have developed a task-graph-based application representation that is used as a common vehicle to support a wide range of modeling techniques. This representation is capable of capturing arbitrary message-passing programs, including general communication patterns, and it is a symbolic representation designed to be computable at compile-time. Because of the information it captures, this representation is not specific to performance modeling, and can be easily generalized to serve as a compile-time internal representation or a run-time representation for program execution. The design of this application representation is described in detail in [1], and is summarized in Section 2.

This paper describes how parallelizing compiler technology can be used to automate the process of constructing this task graph representation for HPF programs compiled to MPI (and, in the near future, for existing MPI programs directly). In particular, this paper makes the following contributions:

- We describe compiler techniques to derive a static, *symbolic* task graph representing the MPI code generated for a given HPF program. A key aspect of this process is the use of symbolic integer sets and mappings to capture a number of dynamic task instances or edge instances as a single node or edge at compile time. These techniques allow the compiler to describe sophisticated computation partitionings and communication and synchronization patterns in symbolic terms.
- We describe how standard analysis techniques can be used to condense the task graph and simplify the control flow, whenever less than fully detailed information suffices (as in many performance modeling applications in practice).
- Finally, we describe an approach to instantiate a dynamic task graph representation from the static task graph, based on a novel use of code generation from symbolic integer sets.

In addition to the above techniques, which to our knowledge are new, the compiler also uses standard techniques to compute symbolic scaling functions for task computation times and message communication volumes.

The techniques described above have been implemented in the Rice dHPF compiler system. The dHPF compiler compiles HPF programs to MPI for message-passing systems, using aggressive techniques for computation partitioning and communication optimization [7, 5, 25]. This implementation was recently used in a joint project with the parallel simulation group at UCLA, in which we developed compiler-based techniques to improve the scalability of simulation of message passing programs [4]. In that work, we showed how compiler information captured in the task graph can be used to reduce the memory and time requirements for simulating message-passing programs in detail using In each application we examined, these reductions enable us to simulate system sizes or problem sizes *10 to 100 times larger* than could be simulated before using state-of-the-art parallel discrete event simulation, with little loss in accuracy. In the context of the present paper, these results illustrate the potential importance of automatically constructing task graphs for widely used programming standards. In particular, these optimized simulation techniques would be completely impractical without compiler support to automate the process of simulating a given program. In Section 4, we briefly describe representative results from the above work to illustrate how task graphs were used in that study.

The next section briefly describes the key features of our static and dynamic task graph representations. Section 3 is the major technical section, which presents the compiler techniques described above to construct the task graph representations. Section 4 illustrates the use of compiler-generated task graphs for simulation mentioned above. We conclude with a brief overview of related work (Section 5) and a discussion of future plans (Section 6).

2 Background: The Task Graph Representation

The POEMS project [3] aims to create a performance modeling environment for the end-to-end modeling of large parallel applications on complex parallel and distributed systems. The project supports a wide range of performance modeling techniques including detailed execution-driven simulation of message-passing programs and scalable I/O [12], program-driven simulation of single-node performance and memory hierarchies, abstract analytical models of parallel programs such as LogP and LoPC [17], and detailed analytical models of parallel program performance based on deterministic task graph analysis [2]. The wide range of modeling techniques in POEMS, and the goal of integrating multiple modeling paradigms make it challenging, if not impossible, for the end-user to generate the required workload information manually for a large-scale application. Thus, since the conception of the project, it has been deemed essential to use compiler support to simplify and partially automate the process of constructing the workload information.

We have designed a common task graph based program representation designed to meet all these gaols [1]. This representation provides a *uniform* platform for capturing the parallel structure of a program as well as its associated workloads for different modeling techniques. The representation uses two flavors of a task graph. The *static task graph* provides a compile-time representation that is designed to be synthesized using parallelizing compiler technology, and can be used directly for some types of modeling studies. The *dynamic task graph* provides a more detailed description of runtime program behavior for a specific program input, and can be used for more precise performance evaluation for a particular input. The design of these representations is described in detail in [1] and is briefly summarized here.

The specific information we aim to collect for a given program includes:

- The detailed computation partitioning and communication structure of the program, described statically i.e., in symbolic terms.
- Source code for individual tasks to support source-code-driven uses such as detailed program-driven simulation of memory hierarchy performance.
- Scaling functions that describe how computation and communication scale as a function of program inputs and processor configuration.
- Optionally, the detailed dynamic behavior of the parallel program, for specified a program input (in which we include the processor configuration).

The Static Task Graph: The static task graph (STG) captures the static parallel structure of a program. The qualification 'static' denotes that the graph is defined only by the program *per se*, thus, it is independent of runtime input values, intermediate program results, and processor configuration. Each node (or task) of the graph may represent one of the following main types: control flow nodes for loops and branches, procedure calls, communication, or pure computation. Edges between nodes may denote control flow within a processor or synchronization between different processors (due to communication tasks). For example, the STG for a simple parallel program is shown in figure 1, and is explained in more detail in the next Section.

A key aspect of the static task graph is that each node in the STG actually represents a set of instances of the task, one per processor that executes the task at runtime. Similarly, an edge in the STG actually represents a set of edge instances connecting pairs of dynamic node instances. We use symbolic integer sets to describe the set of instances for a given node, e.g., a task executed by P processors would be described by the set: $\{[p] : 0 \leq p \leq P-1\}$. Similarly, we use symbolic integer mappings to describe the edge instances, e.g., an edge from a `SEND` task on processor p to a `RECV` task on processor $q = p+1$ (i.e., each processor sends data to its right neighbor, if any) would be described by the mapping: $\{[p] \rightarrow [q] : q = p+1 \wedge 0 \leq p < P-1\}$. This kind of mapping enables precise symbolic representations of arbitrary regular communication patterns. Irregular patterns (i.e., data-dependent patterns that cannot be determined until runtime) have to be represented as an all-to-all communication, which is the best that can be done statically.

To capture high level communication patterns where possible (e.g., shift, pipeline, broadcast, etc. [24]) we group the communication operations in the program into related groups, each describing a single "logical communication event". A communication event descriptor, kept separate from the STG, captures all information about a single communication event. This includes the communication pattern, the set of communication tasks involved, and a symbolic expression for the communication size. The CPU components of each communication event are represented explicitly as communication tasks in the STG, allowing us to use task graph edges between these tasks to explicitly capture the synchronization implied by the underlying communication calls. The number of communication nodes and edges depends on the communication pattern and also on the type of message passing calls used. This technique does not work for MPI receive operations that use a wildcard message tag (because the matching send cannot be easily identified). It does work for receive operations that use a wildcard for the sending processor, but the symbolic mapping on the communication edges may be an all-to-all mapping (for the processors that execute the send and receive statements). Making the communication tasks explicit in the STG has proved valuable also because it allows us to describe arbitrary interleavings (i.e., overlaps) of communication and computation tasks on individual processors and across processors.

In addition to the symbolic sets and mappings above, each node and communication event in the STG includes a symbolic scaling function that describes how the task computation time or the message size scales as a function of program variables.

The STG of a program containing multiple procedures is represented as a number of unconnected graphs, each corresponding to a single procedure. Each call site is represented by a CALL task that identifies the called procedure by name.

The Dynamic Task Graph: The dynamic task graph (DTG) is a directed acyclic graph that captures the execution behavior of a program on a given input and given processor configuration. This representation is important for detailed performance modeling because it corresponds closely with the actual execution behavior being modeled by a particular program performance model (whether using detailed simulation or abstract analytical models).

The nodes of a dynamic task graph are computational tasks and individual communication tasks. In particular, the DTG does not contain control flow nodes (loops, branches, jumps, and jump targets). It can be thought of as being instantiated from the static task graph by unrolling all the loops, resolving all the branches, and instantiating all the instances of parallel tasks, edges, and communication events.

There are two approaches to making this representation tractable for large-scale programs, and these approaches can be combined: (1) we can condense tasks allocated to a process between synchronization points so that only (relatively) coarse-grain parallel tasks are explicitly represented, and (2) if necessary, we can compute the dynamic task graph "on the fly," rather than precomputing it and storing it offline. We describe techniques to automatically condense the task graph in Section 3.2. The approach to instantiate the task graph on-the-fly is outside the scope of this paper, but is a direct extension of the compile-time instantiation of the dynamic task graph described in Section 3.3.

3 Compiler Techniques for Synthesizing the Task Graph epresentations

As noted in the Introduction, there are three major aspects to synthesizing our task graph representation for a parallel program: (1) synthesizing the static, symbolic task graph; (2) condensing the task graph; and (3) optionally instantiating a dynamic task graph representing an execution on a particular program input. Each of these steps relies on information about the message-passing program gathered by the compiler, although for many programs the third step can be performed purely by inspecting the static task graph, as explained in Section 3.3. These steps are described in detail in the following three subsections.

3.1 Synthesizing the Static Task Graph

There are four key steps that need be performed in synthesizing the static task graph (STG):

- generating computation and control-flow nodes;
- generating communication tasks for each logical communication event;
- generating symbolic sets describing the processors that execute each task, and symbolic mappings describing the pattern of communication edges; and
- eliminating excess control flow edges.

Generating computation and control-flow nodes in the STG can be done in a single preorder traversal of the internal representation for each procedure; in our case, the representation is an Abstract Syntax Tree (AST). STG nodes are created as appropriate statements are encountered in the AST. Thus, program statements, such as `DO`, `IF`, `CALL`, `PROGRAM/FUNCTION/SUBROUTINE`, `STOP/RETURN`, trigger the creation of a single node in the graph; encountering one of the first two also leads to the creation of an ENDDO-NODE or an ENDIF-NODE, a THEN-NODE and an ELSE-NODE, respectively. Any contiguous sequence of other computation statements *that are executed by the same set of processors* are grouped into a single computational task (contiguous implies that they are not interrupted by any of the above statements or by communication).

Identifying statements that are computed by the same set of processors is a critical aspect of the above step. This information is derived from the computation partitioning phase of the compiler. This information is translated into a symbolic integer set [7] that is included with each task. By having a general representation of the set of processors associated with each task, our representation can describe sophisticated computation partitioning strategies. The explicit set representation also enables us to check equality by direct set operations, in order to group statements into tasks. These processors sets are also essential for the fourth step listed above, namely eliminating excess control-flow edges between tasks, so as to expose

```
CHPF$ DISTRIBUTE A(*,BLOCK)
      DO I=2,N
         DO J=1,M-1
            A(I,J) = A(I-1,J+1)
         ENDDO
      ENDDO
```

(a) HPF source code fragment

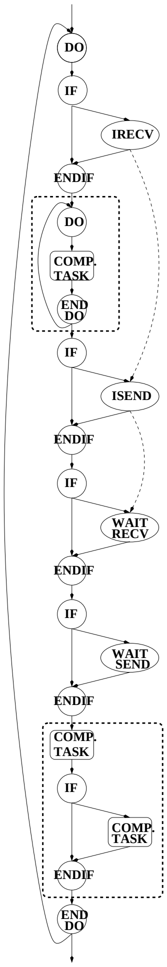
```
blk = block size per processor
DO I=2,N
    IF (myid < P-1)
        irecv B(i, myid*blk+blk+1) from myid+1

    ! Execute local iterations of j-loop
    DO J=myid*blk+1, min(myid*blk+blk-1, M-1)
        A(I,J) = A(I-1,J+1)
    ENDDO

    IF (myid > 0) isend B(i, myid*blk+1) to myid-1
    IF (myid < P-1) wait-recv
    IF (myid > 0) wait-send

    ! Execute non-local iterations of j-loop
    J=myid*blk+blk
    IF (J <= M-1)
        A(I,J) = A(I-1,J+1)
ENDDO
```

(b) Unoptimized MPI code generated by dHPF



(c) Static task graph

Figure 1: An example of generating the communication tasks.

program parallelism. In particular, a control flow edge is retained between two tasks *only if* the intersection of their processor sets is not empty. Otherwise, the sink node is connected to its most immediate ancestor in the STG for which the result of this intersection is a non-empty set.

When the first communication statement for a logical communication event is encountered, the communication event descriptor and all the communication tasks that are pertinent to this single event are built. The processor mappings for the synchronization between the tasks are also built at this time.

For an explicit message-passing program, the computation partitioning information can be derived by analyzing the control-flow expressions that depend on process id variables. The communication pattern information has to be extracted by recognizing the communication calls syntactically, analyzing their arguments, and identifying the matching send and receive calls. In principle, both the control-flow and the communication can be written in a manner that is too complex for the compiler to decipher, and some message passing programs will probably not be analyzable. But in most of the programs we have looked at, the control-flow idioms for partitioning the computation and the types of message passing operations that are used are fairly simple. We believe that the required analysis to construct the STG would be feasible with standard interprocedural symbolic analysis techniques [18, 19].

To illustrate the communication information built by the compiler, consider the simple HPF example, which, along with the MPI parallel code generated by the dHPF compiler, are shown on the left-hand side of Figure 1. The parallelization of the code requires the boundary values of array A along the j dimension to be communicated inside the I loop. (In practice, the compiler pipelines the communication in larger blocks by strip-mining the I loop [20, 6], but that is omitted to simplify the example.) The corresponding STG is shown on the right-hand side of the figure. Solid lines represent control flow edges and dashed lines represent interprocessor synchronization. In this example, the compiler uses the non-blocking MPI communication primitives. The two dashed lines show that the wait-recv operation cannot complete until the isend is executed, and the isend cannot complete until the irecv is issued by the receiver (the latter is true because our target MPI library uses sender-side buffering).[1] Also, the compiler interleaves the communication tasks and computation so as to overlap waiting time at the isend with the computation of local loop iterations, i.e., the iterations that do not read or write any off-processor data. The use of explicit communication tasks within the task graph allows this overlap to be captured precisely in the task graph. The dashed edge between the isend and the wait-recv tasks is associated with the processor mapping: $\{[p_0] \rightarrow [q_0] : q_0 = p_0 - 1 \wedge 0 \leq q_0 < p - 1\}$, denoting that each processor receives data from its "right" neighbor, except the rightmost boundary processor. The other dashed edge has the inverse mapping, i.e., $q_0 = p_0 + 1$.

Finally, the compiler constructs the symbolic scaling functions for each task and communication event, using direct symbolic analysis of loop bounds and message sizes. For a communication event, the scaling function is simply an expression for the message size. For each DO-NODE the scaling function describes the number of iterations executed by each processor, as a function of processor id variables and other symbolic program variables. In the simple example above, the scaling functions for the two DO nodes are N-1 and min(myid*blk+blk-1,M-1) - (myid*blk+1) + 1, respectively. For a computational task, the scaling function is a single parameter representing the workload corresponding to the task. At this stage of the task graph construction, no further handling (such as symbolic iteration counting for more complex loop bounds), takes place.

3.2 Condensing Nodes of 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 less detailed graph will be sufficient. For instance, a coarse-grain modeling approach could assume that all operations of a single process between two communication points constitute a single task.

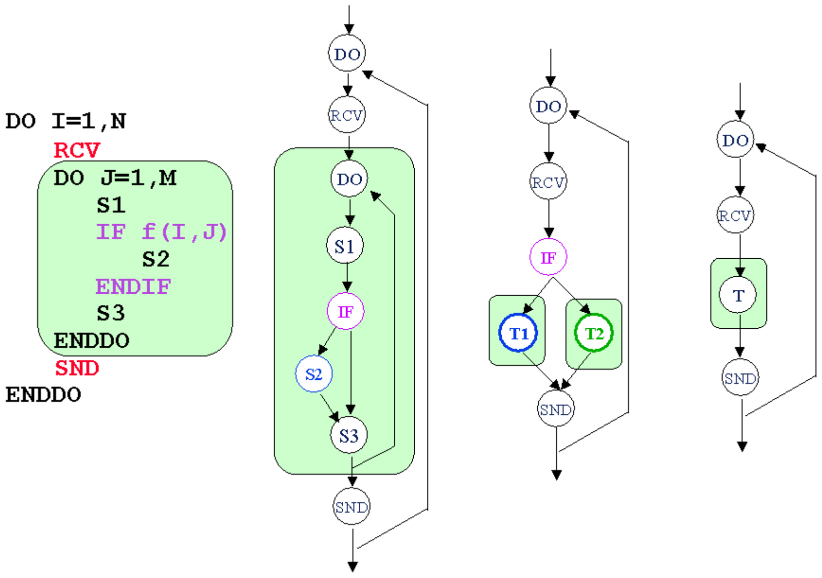
In order to add this functionality, the compiler traverses the STG and marks contiguous nodes, connected by a control-flow edge, that do not correspond to communication (or in the case of compound nodes, do not include communication). Such sequences of nodes are then collapsed and replaced in the STG by a single *condensed* task. Note that such a task will have a single point of entry and a single point of exit. For example, the two large dotted rectangles in Figure 1 (c) correspond to sequences of tasks that can be collapsed into a single condensed task.

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 individual collapsed tasks, each multiplied by the symbolic number of iterations of the surrounding loops, where appropriate (only if these loops are also collapsed).

In cases with conditional control flow, tasks can sometimes be condensed with no loss of accuracy, using sophisticated compiler analysis. 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 together.

To illustrate, consider the code shown in Figure 2. Let w_1, w_2, w_3 represent the workload (i.e., the scaling function) for statements S1, S2 and S3, respectively. The shaded code between the two communication statements is a candidate for collapsing into a single task. The logical condition f(I,J) of the IF statement may depend on either I or J or both. The initial version of the static task graph is shown in part (b) of the figure. The graphs in (c) and (d) show different cases depending on what variables determine the value of f:

[1] More precisely, the isend task should be broken into two tasks, one that performs local initialization and does not depend on the irecv, and a second one that can only be initiated once the irecv has been issued but does not block the local computation on the sending node. This would be simply require introducing additional communication tasks into the task graph.



```
DO I=1,N
   RCV
   DO J=1,M
      S1
      IF f(I,J)
         S2
      ENDIF
      S3
   ENDDO
   SND
ENDDO
```

(a) Source code *(b) Initial STG* *(c) Collapsed STG if* `f=G(I)` *(d) Collapsed STG if* `f=Y(J)`

Figure 2: Collapsing Branches of the Static Task Graph.

- If `f` is a function of `I` only, the `IF` statement can be moved outside the `J` loop and the `J` loop can be collapsed with no loss of accuracy. In this case, we are left with two separate tasks, representing the two possible versions of the `J` loop, as shown in part (c) of the figure. These two tasks have deterministic execution time expressions given by $M \times (w_1 + w_2 + w_3)$, $M \times (w_1 + w_3)$, respectively.

- If `f` is a function of `J` only, the shaded code can be condensed into a single task as shown in part (d) of the figure. The scaling function of the task T would simply be $M \times w$, where w is the workload inside the `J` loop body per iteration of the `I` loop. The key here is that w is constant across different iterations of the `I` loop.

- Finally, if `f` is a function of both `I` and `J`, condensing the shaded code can only be done if we introduce a branching probability parameter. If $p(\texttt{I})$ represents the probability that `S2` will be executed for a given value of `I`, then the entire shaded region can be condensed into a single task (as in part (d)) with a combined scaling function given by $\texttt{M}\times(w_1 + p(\texttt{I}) \times w_2 + w_3)$. This probabilistic expression for execution time can lead to inaccuracies, e.g., in estimates of synchronization delays between processes. If the task graph is not condensed, it remains as in part (b). The decision to condense this task graph therefore should depend on the goals of the modeling study.

These three cases can be differentiated using well-known but aggressive dataflow analysis (the analysis may have to be done interprocedurally in some cases). In fact, we note that the first two cases correspond directly to loop unswitching and identifying loop-invariant code respectively, except that only the static task graph is modified and the code itself is not transformed.

A key point to note is that in the first two cases, there is no resulting loss of accuracy in condensing the task graph. For example, in the ASCI benchmark Sweep3D [3] used in Section 4, the one significant branch is in fact of the first type, which can be pulled out of the task and enclosing loops (the analysis would have to be interprocedural because the enclosing loops are not in the same procedure as the branch).

3.3 Instantiating the Dynamic Task Graph

As noted in Section 2, the dynamic task graph (DTG) is essentially an instantiation of the STG representing a single execution for a particular input. The DTG is an acyclic graph containing no control-flow nodes.

The time for instantiating the DTG grows linearly with the number of task instances in the execution of the program, but much less computation per task is usually required for the instantiation than for the actual execution. This is an optional step that can be performed when required for detailed performance prediction.

The information required to instantiate the DTG varies significantly across programs. For a regular, non-adaptive code, the parallel execution behavior of the program can usually be determined directly from the program input (in which we include the processor configuration parameters). In such cases, the dynamic task graph can be instantiated directly from the STG once the program input is specified. In general, and particularly in adaptive codes, the parallel execution behavior (and therefore the DTG) may depend on *intermediate computational results* of the program. For example, this could happen in a parallel n-body problem if the communication pattern changed as the positions of the bodies evolved during the execution of the program. In the current work, we focus on the techniques needed to instantiate the DTG in the former case, i.e., that of regular non-adaptive codes. These techniques are also valuable in the latter case, but they must be applied at runtime when the intermediate values needed are known. The issues to be faced in that case are briefly discussed later in this section.

There are two main aspects to instantiating the DTG:

- enumerating the outcomes of all the control flow nodes, effectively by unrolling the DO nodes and resolving the dynamic instances of the branch nodes; and
- enumerating the dynamic instances of each node and edge in the STG.

These are discussed in turn below. Of these, the second step is significantly more challenging in terms of the compile-time techniques required, particularly for sophisticated message passing programs with general computation partitioning strategies and communication patterns.

Interpreting control-flow in the static task graph Enumerating the outcomes of all the control flow nodes in an execution can be accomplished by a symbolic interpretation of the control flow of the program for each process. First, we must enumerate loop index values and resolve the dynamic instances of branch conditions that have not been collapsed in the STG. This requires evaluating the values of these symbolic expressions. We can perform this evaluation directly *at compile-time* when these quantities are determined solely by input values, surrounding loop index variables, and processor id variables. Under these conditions, we know all the required variable values in the expressions, as follows. The input variable values are specified externally. The loop index values are explicitly enumerated for all DO nodes that are retained in the static task graph. The processor id variables are explicitly enumerated for each parallel task using the symbolic processor sets, as discussed below. Therefore, we can evaluate the relevant symbolic expressions for enumerating the control-flow outcomes.

We assumed above that key symbolic quantities were determined solely by input values, surrounding loop index variables, and processor id variables. These requirements only apply to those loop bounds and branch conditions that are retained in the collapsed static task graph (i.e., which affect the parallel task graph structure of the code), and *not* to loops and branches that have been collapsed because they only affect the internal computational results of a task. With the exception of a few common algorithmic constructs, we find these requirements to be satisfied by a fairly large class of regular scientific applications. For example, in a collection of codes including the three NAS application benchmarks (SP, BT and LU), an ASCI benchmark Sweep3D [3], and other standard data-parallel codes such as Erlebacher [7] and the SPEC benchmark Tomcatv, the sole exceptions were terminating conditions testing convergence in the outermost timestep loops. In such cases, we would rely on the user to specify a fixed number of time steps for which the program performance would be modeled.

More generally, and particularly for adaptive codes, we expect the parallel structure to depend on intermediate computational results. This would require generating the DTG on the fly, e.g., when performing program-driven simulation during which the actual computational results would be computed. In this case, the most efficient approach to synthesizing the DTG would be to use program slicing to isolate the computations that do affect the parallel control flow. (This is very similar to the use of slicing for optimizing parallel simulation as described in Section 4.) These extensions are outside the scope of this paper.

Enumerating the symbolic sets and mappings The second challenge is that we must enumerate all instances of each parallel task and each communication edge between tasks. These instances are described by symbolic sets and mappings respectively. In the context of complex computation partitioning strategies and arbitrary communication patterns, this presents a much more difficult problem at compile-time (i.e., without executing the actual program).

For example, consider the loop fragment from the NAS LU benchmark shown in Figure 3. The compiler automatically chooses a sophisticated computation partitioning, denoted by the ON HOME descriptors for each statement in the figure. For example, the ON HOME descriptor for the assignment to `flux` indicates that the instance of the statement in iteration `(k,j,i,m)` should be executed by the processors that own either of the array elements `rsd(m,i-1,j,k)` or `rsd(m,i+1,j,k)`. This means that each boundary iteration of the statement will be replicated among the two adjacent processors. This replication eliminates the need for highly expensive inner-loop communication for the privatizable array `flux` [5]. Communication is still required for each reference to array u, all of which are coalesced by the compiler into a single logical communication event. The communication pattern is equivalent to two SHIFT operations in opposite directions. Part (b) of the figure shows the set of processors that must execute the SEND communication task ($ProcsThatSend$), as well as the mapping between processors executing the SEND and those executing the corresponding `wait-recv` ($SendToRecvProcsMap$). (Note that both these quantities are described in terms of symbolic integer sets, parameterized by the variables `jst`, `jend`, `nx` and `nz`.) Each of these sets combines the information for both SHIFT operations. Correctly instantiating the communication tasks and edges for such communication patterns in a *pattern-driven* manner can be difficult and error-prone, and would be limited to some predetermined class of patterns that is unlikely to include such complex patterns.

Instead, we develop a novel and general solution to this problem that is based on an unusual use of code generation from integer sets. In ordinary compilation, dHPF and other advanced parallelizing compilers use code generation from integer sets to synthesize loop nests that are executed *at runtime*, e.g., for a parallel loop nest or for packing and unpacking data for communication [7, 8, 9, 13]. If we could invoke the same capability but execute the generated loop nests *at compile-time*, we could use the synthesized loop nests to enumerate the required tasks and edges.

Implementing this approach, however, proved to be a non-trivial task. Most importantly, each of the sets are parameterized by several variables, including input variables and loop index variables (e.g., the two sets above are parameterized by `jst`, `jend`, `nx` and `nz`). This means that the set must be enumerated separately for each combination of these variable values that occurs during the execution of the original program. We solve this problem as follows. We first generate a subroutine for each integer set that we want to enumerate, and make the parameters arguments of the subroutine. Then (still at compile-time), we compile, link, and invoke this code in a separate process.[2] The desired combinations of variable values for each node and edge are automatically available when interpreting the control-flow of the task graph as described earlier. Therefore, during this interpretation, we simply invoke the desired subroutine in the other process to enumerate the ids for a node or the id pairs for an edge.

To illustrate this approach, Figure 3(c) shows the subroutine generated to enumerate the elements of the set $ProcsThatSend$ described earlier. The loop nest in this subroutine is generated directly from the symbolic integer set in part (b) of the figure. This loop nest enumerates the instances of the SEND task, which in this case is one task per processor executing the SEND. This subroutine is parameterized by `jst`, `jend`, `nx` and `nz`. In this example, these are all unique values determined by the program input. In general, however, these could depend on loop index variables of some outer loop and the subroutine has to be invoked for each combination of values of its arguments.

Overall, the use of code generated from symbolic sets enables us to support a broad class of computation partitionings and communication patterns in a uniform manner. This approach fits nicely with the core capabilities of advanced parallelizing compilers.

[2] We could simplify this step if we modified the back-end of the integer set package to generate an interpreted language like Lisp rather than Fortran.

4 Status and esults

4.1 Status

We have successfully implemented the techniques described above in the dHPF compiler. We have extended the dHPF compiler to synthesize a static task graph for the MPI code generated by dHPF, including the symbolic processor sets and mappings for communication tasks and edges, and the scaling functions for loop nodes. In computing the condensed static task graph, we collapse all DO-NODEs or sequences of computational tasks that do not contain any communication or any IF-NODEs (We would rely on user intervention to collapse IF-NODEs). We also compute the combined scaling function for the collapsed tasks.

We have also partially implemented the support to instantiate dynamic task graphs at compile-time. In particular, we are able to enumerate the task instances and control-flow edges. We also synthesize the code from symbolic integer-sets required to enumerate the edge mappings at compile-time. We do not yet link in this code to enumerate the edges at compile-time.

Because of the aggressive computation partitioning and communication strategies used by dHPF, capturing the resulting MPI code requires the full generality of our task graph representation. This gives us confidence that we can synthesize task graphs for a wide range of explicit message-passing programs as well (including all the ones we have examined so far).

4.2 Effectiveness of task graph abstraction and condensing

In order to illustrate the size of the static task graph generated and the effectiveness of condensing the task graph, Table 1 lists these statistics for the STG produced by the dHPF compiler for three HPF benchmarks: Tomcatv (from SPEC92), jacobi (a simple 2D Jacobi iterator PDE solver), and expl (Livermore Loop # 18). The effect of condensing the task graph on reducing the number of loops (DO-NODE) and computational tasks (COMP-TASK) can be easily observed. After condensing, most of the remaining tasks are either IF-NODEs and dummy nodes (e.g., ENDIF-NODE, etc.) or communication tasks (which are never condensed), since we opted for a detailed representation of communication behavior, rather than compromise on the accuracy of the representation.

The reader can check the task graphs for the above codes at `http://www.cs.man.ac.uk/~rizos/taskgraph/`

	tomcatv	jacobi	expl
Number of lines of source HPF program	227	64	94
Number of lines of output parallel MPI program	1850	1156	3722
Total number of tasks (1st pass)	247	122	225
# COMM-NODE	54	36	114
# DO-NODE	18	13	17
# COMP-TASK	39	16	23
Total number of tasks (after condensing)	193	83	174
# DO-NODE	3	1	3
# COMP-TASK	20	5	6

Table 1: Size of STG for various example codes before and after condensing.

4.3 Task-graph-driven simulation of MPI programs

The most important application of our compiler-synthesized task graphs to date has been for improving the state of the art of parallel simulation of message-passing programs [4]. Those results are briefly summarized here because they provide the best illustration of the correctness and benefits of the compiler-synthesized task graphs. This work was performed in collaboration with the parallel simulation group at UCLA, using MPI-Sim, a direct-execution parallel simulator for MPI programs [11].

The basic strategy in using the STG for program simulation is to generate an abstracted MPI program from the STG where all computation corresponding to a computational task is eliminated, *except those*

Benchmark	% Error in prediction vs. measurement				
	#Procs = 4	8	16	32	64
Tomcatv	−5.44	15.75	11.79	8.50	9.27
Sweep3D	−7.01	−4.97	9.02	9.80	5.13
NAS SP, class A	−2.59		−1.24	7.11	6.10
NAS SP, class C			0.09	−14.01	−1.58

Table 2: Validation of the compiler-generated task graphs using a simulator for MPI programs.

computations whose results are required to determine the control-flow, communication behavior, and task scaling functions. We refer to the eliminated computations as *redundant computations* (from the point of view of performance prediction), because the results of these computations do not affect the performance of the program. We use the task scaling functions to generate simple symbolic estimates for the execution time of the redundant computations. The simulator can avoid simulating the redundant computations, and simply needs to advance the simulation clock by an amount equal to the estimated execution time of the computation. The simulator can even avoid allocating memory for program data that is referenced only in redundant computations.

The key to implementing this strategy lies in identifying non-redundant computations. To do so, we must first identify the values in the program that determine program performance. These are exactly those variable values that appear in the control-flow expressions, communication descriptors, and scaling functions (both for task times and for communication volume) of the STG. Thus, using the STG makes these values very easy to identify. We can then use a standard program slicing algorithm [21] to isolate the computations that affect these values. We then generate the simplified MPI code by including all the control-flow that appears in the static task graph, all the communication calls, and the non-redundant computations identified by program slicing. All the remaining (i.e., redundant) computations in each computational task are replaced with a single call to a special simulator delay function which simply advances the simulator clock by a specified amount. The argument to this function is a symbolic expression for the estimated execution time of the redundant computation. Note that the simulator continues to simulate the communication behavior in detail. Again, the reader can check the compiler-synthesized versions of the above codes at `http://www.cs.man.ac.uk/~rizos/taskgraph/`

We have applied this methodology both to HPF programs (compiled to MPI by the dHPF compiler), and also to existing MPI programs. In the latter case, we generated the abstracted MPI program by hand, but were careful to imitate the code that would be obtained if the compiler synthesized an STG for the original MPI program. The benchmarks include an HPF version of Tomcatv, one of the SPEC92 benchmarks, and MPI versions of Sweep3D (a key ASCI benchmark) and NAS SP (one of the NAS application benchmarks).

Table 2 shows the percentage error in the execution times predicted by MPI-Sim using the simplified MPI code, compared with direct program measurement. As can be seen, the error was less than 16% in all cases tested, and less than 10% in most of these cases. This is important because the simplified MPI program can be thought of as simply an executable representation of the static task graph itself. These results show that the task graph abstraction very accurately captures the properties of the program that determine performance. We believe that the errors observed could be further reduced by applying more sophisticated techniques for estimating the execution time of redundant computations, particularly with simple estimates of cache behavior.

The benefits of using the task-graph-based simulation strategy were extremely impressive. For these benchmarks, 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 and problem sizes 10 to 100 times larger than is possible with the original simulator (with little loss of accuracy as shown by the validations above). In particular, this has allowed us to simulate MPI programs for parallel architectures with hundreds of processors *faster than real-time*, and has made it feasible to simulate execution of programs on machines with 10,000+ processors. These results are described in more detail in [4].

5 elated Work

As mentioned in the Introduction, there is a large body of work on the use of task graphs for various aspects of parallel systems but very little work on synthesizing task graphs for general-purpose parallel programs. The vast majority of performance models that use task graphs as inputs generally do not specify how the task graph should be constructed but assume that this has been done [30, 22, 28, 2, 15]. A more detailed discussion of a wide range of such models is available in [2].

The various compiler-based systems that use task graphs, namely PYRROS [31], CODE [27], HENCE [27], and Jade [23] construct task graphs by assuming special information from the programmer. In particular, PYRROS, CODE and HENCE all assume that the programmer specifies the task graph explicitly (CODE and HENCE actually use a graphical programming language to do so). In Jade, the programmer specifies input and output variables used by each task and the compiler uses this information to deduce the task dependences for the program.

The PlusPYR project has developed a task graph representation that has some similarities with ours (in particular, symbolic integer sets and mappings for describing task instances and communication and synchronization rules), along with compiler techniques to synthesize these task graphs. The key difference from our work is that they start with a limited class of *sequential* programs (annotated to identify the tasks) and use dependence analysis to compute dependences between tasks, and then derive communication and synchronization rules from these dependences. Therefore, their approach is essentially a form of simple automatic parallelization. In contrast, our goal is to generate task graphs for existing parallel programs with no special program annotations and with explicit communication. 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 support realistic HPF and MPI programs.

As noted in the Introduction, Parashar et al. [29] construct task graphs for HPF programs compiled by the Syracuse Fortran 90D compiler, but they are limited to a very simple, loosely synchronous computational model that would not support many message-passing and HPF programs in practice. In addition, their interpretive framework for performance prediction uses functional interpretation for instantiating a dynamic task graph, which is similar to our approach for instantiating control-flow. Like the task graph model, their interpretation and performance estimation are significantly simplified (compared with ours) because of the loosely synchronous computational model. For example, they do not need to capture sophisticated communication patterns and computation partitionings, as we do using code generation from integer sets.

Dikaiakos et al. [14] developed a tool called FAST that constructs task graphs from user-annotated parallel programs, performs advanced task scheduling and then uses abstract simulation of message passing to predict performance. Their task graph construction is fairly straightforward and is driven by user annotations describing the tasks and inter-task communication.

The PACE project [28] proposes a language and programming environment for parallel program performance prediction. Users are required to identify parallel subtasks and computation and communication patterns.

Finally, Fahringer [16], Armstrong and Eigenmann [10], Mendes and Reed [26] and many others have developed symbolic compile-time techniques for estimating execution time, communication volume and other metrics. The communication and computation scaling functions available in our static task graph are very similar to the symbolic information used by these techniques, and could be directly extended to support their analytical models.

6 Conclusion and Future Plans

In this paper, we described a methodology for automating the process of synthesizing task graphs for parallel programs, using sophisticated parallelizing compiler techniques. The techniques in this paper can be used *without* user intervention to construct task graphs message-passing programs compiled from HPF source programs, and we believe they extend directly to existing message-passing (e.g., MPI) programs as well. Such techniques can make a large body of existing research based on task graphs and equivalent representations applicable for these widely used programming standards.

Our immediate goals for the future are:

- to demonstrate that the techniques described in this paper can be applied to message-passing programs (using MPI), by extracting the requisite computation partitioning and communication information; and
- to couple the compiler-generated task graphs with the wide range of modeling approaches being used within the POEMS project, including analytical, simulation and hybrid models.

Acknowledgement: 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. We are particularly grateful to the UCLA team and especially Ewa Deelman for her efforts in validating the simulations of the abstracted MPI codes generated by the compiler.

This work was carried out while the authors were with the Computer Science Department at Rice University.

eferences

[1] V. Adve and R. Sakellariou. Application Representations for Multi-Paradigm Performance Modeling of Large-Scale Parallel Scientific Codes. *International Journal of High Performance Computing Applications*, to appear.

[2] V. Adve and M. K. Vernon. A Deterministic Model for Parallel Program Performance Evaluation. Technical Report CS-TR98-333, Computer Science Dept., Rice University, December 1998. Also available at `http://www-sal.cs.uiuc.edu/~vadve/Papers/detmodel.ps.gz`.

[3] V. S. Adve, R. Bagrodia, J. C. Browne, E. Deelman, A. Dube, E. Houstis, J. R. Rice, R. Sakellariou, D. Sundaram-Stukel, P. J. Teller, and M. K. Vernon. POEMS: End-to-End Performance Design of Large Parallel Adaptive Computational Systems. *IEEE Trans. on Software Engineering*, November 2000.

[4] V. S. Adve, R. Bagrodia, E. Deelman, T. Phan, and R. Sakellariou. Compiler- Supported Simulation of Highly Scalable Parallel Applications. In *Proceedings of Supercomputing '99*, Portland, Oregon, November 1999.

[5] Vikram Adve, Guohua Jin, John Mellor-Crummey, and Qing Yi. High Performance Fortran Compilation Techniques for Parallelizing Scientific Codes. In *Proceedings of SC98: High Performance Computing and Networking*, Orlando, FL, Nov 1998.

[6] Vikram Adve, Charles Koelbel, and John Mellor-Crummey. Compiler Support for Analysis and Tuning of Data Parallel Programs. In *Proceedings of the 1994 Workshop on Parallel Processing Tools and Environments*, May 1994.

[7] Vikram Adve and John 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.

[8] S. Amarasinghe and M. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, Albuquerque, NM, June 1993.

[9] C. Ancourt and F. Irigoin. Scanning polyhedra with do loops. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, April 1991.

[10] Brian Armstrong and Rudolf Eigenmann. Performance Forecasting: Towards a Methodology for Characterizing Large Computational Applications. In *Proc. of the Int'l Conf. on Parallel Processing*, pages 518–525, August 1998.

[11] R. Bagrodia, E. Deelman, S. Docy, and T. Phan. Performance prediction of large parallel applications using parallel simulation. In *Proc. Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, GA, 1999.

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

[13] P. Banerjee, J. Chandy, M. Gupta, E. Hodges, J. Holm, A. Lain, D. Palermo, S. Ramaswamy, and E. Su. The Paradigm compiler for distributed-memory multicomputers. *IEEE Computer*, 28(10):37–47, October 1995.

[14] 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.

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

[16] T. Fahringer and H. Zima. A static parameter based performance prediction tool for parallel programs. In *Proceedings of the 1993 ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.

[17] Matthew Frank, Mary Vernon, and Anant Agarwal. LoPC: Modeling Contention in Parallel Algorithms. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, Las Vegas, USA, August 1997.

[18] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[19] Paul Havlak. *Interprocedural Symbolic Analysis*. PhD thesis, Dept. of Computer Science, Rice University, May 1994. Also available as CRPC-TR94451 from the Center for Research on Parallel Computation and CS-TR94-228 from the Rice Department of Computer Science.

[20] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Evaluation of compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[21] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. on Programming Languages and Systems*, 12:26–60, 1990.

[22] Alex Kapelnikov, Richard R. Muntz, and Milos D. Ercegovac. A Modeling Methodology for the Analysis of Concurrent Sys tems and Computations. *Journal of Parallel and Distributed Computing*, 6:568–597, 1989.

[23] Monica S. Lam and Martin Rinard. Coarse-Grain Parallel Programming in Jade. In *Proc. Third ACM SIGPLAN Symposium on Principles and Practices of Para llel Programming*, pages 94–105, Williamsburg, VA, April 1991.

[24] J. Li and M. Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–376, July 1991.

[25] John Mellor-Crummey and Vikram Adve. Simplifying control flow in compiler-generated parallel code. *International Journal of Parallel Programming*, 26(5), 1998.

[26] Celso Mendes and Daniel Reed. Integrated Compilation and Scalability Analysis for Parallel Systems. In *Proc. of the Int'l Conference on Parallel Architectures and Compilation Techniques*, August 1998.

[27] 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.

[28] E. Papaefstathiou, D. J. Kerbyson, G. R. Nudd, T. J. Atherton, and J. S. Harper. An Introduction to the Layered Characterization for High Performance Systems. Research Report 335, University of Warwick, December 1997.

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

[30] Alexander Thomasian and Paul F. Bay. Analytic Queueing Network Models for Parallel Processing of Task Systems. *IEEE Trans. on Computers*, C-35(12):1045–1054, December 1986.

[31] 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.

```
CHPF$  distribute rsd(*,block,block,*)
CHPF$  distribute u(*,block,block,*)
CHPF$  distribute flux(*,block,block,*)
       do k = 2, nz-1
         do j = jst, jend
CHPF$      INDEPENDENT, NEW(flux)
           do indep_dummy_loop = 1, 1
             do i = 1, nx
               do m = 1, 5
                            !    ON HOME rsd(m,i-1,j,k) ⋃ rsd(m,i+1,j,k)
                 flux(m,i,j,k) = F( u(m,i,j,k) )
               enddo
             enddo
             do i = ist, iend
               do m = 1, 5
                            !    ON HOME rsd(m,i,j,k)
                 rsd(m,i,j,k) = G( flux(m,i-1,j,k), flux(m,i+1,j,k) )
               enddo
             enddo
           enddo
         enddo
       enddo
```

(a) Source code and computation partitioning

$$
\begin{aligned}
\mathit{ProcsThatSend} \quad = \quad & \{[p_0, p_1] : \mathtt{jst} \leq 17p_1 + 17, \mathtt{jend}, 65 \wedge 0 \leq p_0 \leq 3 \wedge 0 \leq p_1 \leq 3 \\
& \quad \wedge 17p_0 < \mathtt{nx} \wedge 17p_1 < \mathtt{jend} \wedge 3 \leq \mathtt{nz}\} \\
& \bigcup \{[p_0, p_1] : \mathtt{jst} \leq 17p_1 + 17, \mathtt{jend}, 65 \wedge 0 \leq p_0 \leq 3 \wedge 0 \leq p_1 \leq 3 \\
& \quad \wedge 17p_0 < \mathtt{nx} \wedge 17p_1 < \mathtt{jend} \wedge 3 \leq \mathtt{nz} \wedge 2 \leq \mathtt{nx} \wedge 17p_1 < \mathtt{jend}\}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{SendToRecvProcsMap} \quad = \quad & \{[p_0, p_1] \rightarrow [q_0, q_1] : q_1 = p_1 \wedge 0, q_0 - 1 \leq p_0 \leq 3 \\
& \quad \wedge \mathtt{jst} \leq 17p_1 + 17, 65, \mathtt{jend} \wedge 0 \leq p_1 \leq 3 \\
& \quad \wedge 17p_0 < \mathtt{nx} \wedge 3 \leq \mathtt{nz} \wedge 17q_0 \leq \mathtt{nx} \wedge 17p_1 < \mathtt{jend}\} \\
& \bigcup \{[p_0, p_1] \rightarrow [q_0, q_1] : 0, p_0 - 1 \leq q_0 \leq p_0 \leq 3 \wedge \mathtt{jst} \leq 17p_1 + 17, 65, \mathtt{jend} \\
& \quad \wedge 0 \leq p_1 \leq 3 \wedge 17p_0 < \mathtt{nx} \\
& \quad \wedge 3 \leq \mathtt{nz} \wedge 2 + 17q_0 \leq \mathtt{nx} \wedge 17p_1 < \mathtt{jend}\}
\end{aligned}
$$

(b) Processor sets and task mappings

```
subroutine ProcsThatSend(nx, nz, jst, jend)
integer nx, nz, jst, jend

if (jend >= 1 && jst <= jend && nz >= 3 && jst <= 65) then
  do p0 = 0,  min(intDiv(nx-1,17), 3)
    do p1 = max(intDiv(jst-17+16,17), 0),  min(intDiv(jend-1,17), 3)
      !  Emit SEND task for processor (p0,p1)
    enddo
  enddo
endif
```

(c) Parameterized code to enumerate the set `ProcsThatSend`

Figure 3: Example from NAS LU illustrating processor sets and task mappings for communication tasks. Problem size: 65 × 65 × 65; Processor configuration: 4 × 4.