Data-Parallel Compiler Support for Multipartitioning

Daniel Chavarría-Miranda
Dept. of Computer Science
Rice University
Houston, TX
danich@cs.rice.edu

John Mellor-Crummey
Dept. of Computer Science
Rice University
Houston, TX
johnmc@cs.rice.edu

Trushar Sarang
Dept. of Computer Science
Rice University
Houston, TX
tsarang@cs.rice.edu

ABSTRACT

Strategies for partitioning an application's data determine both the range of suitable parallelizations and their potential efficiency. For multi-directional line-sweep computations, *multipartitioned* data distributions offer better parallel efficiency and scalability than block unipartitionings. This paper describes extensions to the Rice dHPF compiler for High Performance Fortran that enable it to support multipartitioned data distributions, and optimizations that enable dHPF to generate efficient multipartitioned code. We describe experiments applying these techniques to parallelize serial versions of the NAS SP and BT application benchmarks and show that the performance of the code generated by dHPF is within a few percent of that of hand-coded parallel implementations using multipartitioning.

1. INTRODUCTION

The strategies used to partition an application's data and computation for parallel execution play a fundamental role in determining both the range of the application's suitable parallelizations and their potential efficiency. High Performance Fortran (HPF) and OpenMP, two standard high-level models for parallel programming, provide a narrow set of choices for data partitioning and computation partitioning, respectively. While these partitionings can lead to good performance for loosely synchronous computations, they are problematic for more tightly-coupled codes.

Line-sweep computations, used to solve one-dimensional recurrences along each dimension of a multi-dimensional discretized domain, are an important class of tightly-coupled computations that are not supported well by the HPF and OpenMP partitioning choices. This style of computation is the basis for Alternating Direction Implicit (ADI) integration, a widely-used numerical technique for solving partial differential equations such as the Navier-Stokes equation [5, 11, 15]. Line-sweep computations are also the basis for a variety of other numerical methods and solution techniques [15]. Recurrences along each dimension of the data domain make this class of computations difficult to parallelize effectively.

To support effective parallelization of this class of computations, a sophisticated strategy for partitioning data and computation known as *multipartitioning* was developed [5, 11, 15]. Multipartitioning distributes arrays of two or more dimensions among a set of processors so that for computations performing a directional sweep along any one of the array's data dimensions, (1) all processors are active in each step of the computation, (2) load-balance is nearly perfect, and (3) only a modest amount of coarse-grain communication is needed. These properties are achieved by carefully assigning each processor a balanced number of tiles between each pair of adjacent hyperplanes that are defined by the cuts along any partitioned data dimension. Multipartitioning is described in more detail in Section 2.1.

A study by van der Wijngaart [17] of implementation strategies for hand-coding parallelizations of Alternating Direction Implicit Integration (ADI) found that 3D multipartitioning was superior to both static block partitionings using a wavefront parallization strategy, as well as a dynamic block partitioning strategy where repartitioning was performed by transposing between phases of the computation. Another study by Naik [16] examined differences between multipartitioning with static block partitioning strategies with respect to their tolerance of load imbalance. His findings show that three-dimensional partitionings are most tolerant of load imbalance, and that three-dimensional multipartitionings are the most tolerant of load imbalance as long as the number of processors is not overly large for the data size.

Our earlier research on data-parallel compilation technology to support effective, semi-automatic compiler-based parallelizations of ADI line-sweep computations focused on exploiting wavefront parallelism with static block partitionings by using a coarse-grain pipelining strategy [1]. Although the performance we achieved with this approach was superior to that achieved using the Portland Group's *pghpf* compiler [4] to compile variants of the tested applications parallelized using dynamic block partitionings, [1] both of the compiler-based parallelization strategies fell short of the performance achieved by hand-coded parallelizatiuons of the applications based on multipartitioning [1].

To try to close the gap between the performance achiev-

[1] Commercial data-parallel compilers such as pghpf lack support for coarse-grain pipelining. The Portland Group substantially rewrote variants of NAS SP and BT benchmark codes [3] to avoid programming idioms that their compiler could not parallelize effectively.

able with compiler-based parallelizations of line-sweep computations and the performance of hand-coded parallelizations, we began to investigate how data-parallel compilation techniques developed for block partitionings could be extended to support multipartitioning. In a previous paper [7], we described several of the basic compiler and runtime techniques necessary to generate code and support multipartitioned data distributions and evaluated a prototype implementation in the Rice dHPF compiler. Evaluation of our early prototype showed that although compiler-directed multipartitioning balanced work, there remained a substantial gap between its parallel performance and that of its hand-coded counterpart. Analysis of these results showed that the principal obstacles to better performance of such compiler-based multipartitionings were high communication frequency due to failure to aggregate messages headed in the same direction, high communication volume because of insufficient message coalescing and selection of inferior computation partitionings, and overly complex code for communication and computation.

This paper describes the design and implementation of a variety of optimizations that remove the performance bottlenecks we encountered with our first prototype implementation. Our new optimizations enable line-sweep codes parallelized by the dHPF compiler using compiler-directed multipartioning to achieve performance approaching that of hand-coded parallelizations based on multipartitioning. We have achieved performance which is on average (for processor counts from 1 to 36), within 2% of the hand-coded implementation for the class 'A' BT benchmark, and within 0.5% (on average) for the class 'B' BT benchmark. For SP, the end-to-end performance is within 15.63% of the hand-coded class 'A' benchmark, and within 16.78% for the class 'B' hand-coded benchmark.

Section 2 provides some background for our work with a brief introduction to multipartitioning and a brief overview of the Rice dHPF compiler and how it implements multipartitioning. Section 3 describes new optimizations we developed to make compiler-based multipartitioning efficient. Section 4 describes general partitioning-independent optimizations that we added to dHPF in our quest to achieve hand-coded performance with the complex code generated for compiler-based multiparitionings. Section 5 describes our experiences applying our compiler-based multipartitionings to the NAS SP and BT application benchmarks [3], two codes that use ADI integration to solve the Navier-Stokes equation in three dimensions, and compares our results with the performance of hand-coded multipartitionings. Section 7 presents our conclusions.

2. BACKGROUND

2.1 Multipartitioning

Multipartitionings for two and three dimensional data have been described in the literature by a number of researchers [5, 11, 15] "Diagonal" multi-partitionings, described in two dimensions by Johnsson, Saad and Shultz [11], and in three dimensions by Naik, Naik, and Nicoules [15], also generalize to higher dimensions. This style of multipartitionings involve involves partitioning data into $p^{\frac{d}{d-1}}$ tiles, where p is the number of processors and d is the number of partitioned array dimensions. Each processor handles $p^{\frac{1}{d-1}}$ tiles arranged along diagonals through each of the partitioned dimensions. Figure 1 shows a 3D multipartitioning for 16 processors; the number in each tile indicates the processor that owns the block. A restriction of diagonal multipartitionings is that for a d-dimensional diagonal multipartitioning to be applicable, the expression $p^{\frac{1}{d-1}}$ must be integral.

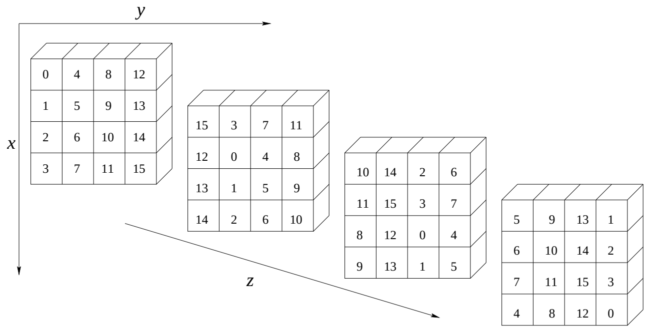

Figure 1: 3D Multipartitioning on 16 processors.

Recently, we developed an algorithm for computing generalized multipartitionings that avoids the integrality constraint of diagonal multipartitionings. Our result enables multipartitioning to be applied effectively in a much wider range of situations. Our algorithm determines how to best partition arrays of ρ dimensions, $\rho \geq 2$, on an arbitrary number of processors. The algorithm selects the appropriate number of partitioned dimensions d, where $2 \leq d \leq \rho$, determines how many partitions belong in each dimension of the array to minimize the total communication volume needed for a sweep along each partitioned dimension, ensures that each processor will have a balanced number of tiles in each hyperplane of the computation, and computes a mapping from data tiles to processors. When $\rho > 2$ our algorithm selects a two-dimensional multipartitioning only when a prime number of processors is used (unless the array dimensions are not balanced); otherwise, a more efficient higher-dimensional partitioning will be selected. A paper describing this result is in preparation.

2.2 The dHPF Compiler

This section provides a brief overview of the Rice dHPF compiler. More details about dHPF and its capabilities are available elsewhere [1, 2]. The dHPF compiler translates HPF programs into single-program-multiple-data node programs in Fortran 77 that use MPI message-passing primitives for communication.[2]

The dHPF compiler is based on an abstract equational framework that expresses data parallel program analyses and optimizations in terms of operations on sets of integer tuples. These tuples represent data elements, processors and iterations [2]. We use the Omega Library [13, 12] as the implementation technology for this framework. Our equational framework has enabled us to implement simple, concise, and general formulations of analyses and optimizations. Because of the generality of these formulations, it has been possible to implement a comprehensive collection of advanced optimizations to support semi-automatic paral-

[2]Earlier versions of dHPF also generated code for shared-memory systems prior to a rewrite of the communication generation subsystem.

lelization of scientific programs that are broadly applicable. Key features, analyses, and optimizations implemented in dHPF include:

- support for a generalization of the owner-computes rule that can specify that computation be performed on the home of one or more arbitrary references;
- independent computation partitionings for each program statement;
- automatic computation partition selection to minimize communication frequency;
- static loop partitioning and communication analysis for this general computation partitioning model;
- communication vectorization for arbitrary regular communication patterns;
- message coalescing for arbitrary affine references to an array;
- code generation for a symbolic number of processors;
- use of asynchronous communication primitives to provide latency tolerance;
- array padding to avoid cache conflict misses; and
- guard minimization strategies to improve code efficiency.

In addition to the optimizations described above, the dHPF compiler supports partial replication of computation to reduce communication frequency [1]. Partially-replicating assignments to a variable (e.g. replicating the computation of boundary planes of a partitioned array, or having multiple processors compute the assignment to a privatizable scalar in a loop iteration that they both must execute) can eliminate the need to communicate a variable between its definition and use. Since any replication of computation may be costly, dHPF only applies this technique automatically for privatizable scalars. For arrays, it partially replicates computation to avoid communication only under directive control [1]. A key technology that supports partially-replicated computation in the dHPF compiler is our general computation partitioning model which supports analysis and code generation of complex computation partitionings such as `ON HOME x(i,j,k), x(i,j+1,k), x(i,j-1,k), x(i+1,j,k), x(i-1,j,k), x(i,j,k+1), x(*,i,j,k-1)`, where `x` is an array multipartitioned in the `i`, `j`, and `k` dimensions.[3]

The integer set analysis framework used by dHPF supports block partitionings of arbitrary size onto a symbolic number of processors. This framework serves as the basis for dHPF's novel compiler support for multipartitioning. To support multipartitioned arrays, for the most part dHPF treats each data tile for a multipartitioned array simply as a block in a block partitioned array that is assigned to its own virtual processor. To support a diagonal multipartitioning in d dimensions, dHPF generates SPMD code so that each physical processor manages $p^{\frac{1}{d-1}}$ virtual processor tiles.

Each data tile for a multipartitioned array is extended as appropriate with overlap areas [10] to facilitate access to non-local data received from neighboring tiles. On each processor, all local tiles for a multipartitioned array are dynamically allocated as contiguous data. Storage is indexed in column-major order, where the leftmost dimensions are the original array dimensions and a new rightmost dimension corresponds to the *local tile index*. All communication and computation performed for a tile is defined in terms of the data mapped to that tile.

[3]This partitioning causes computation to be replicated along the boundaries in each partitioned dimension.

Code generation for multipartitioned loop nests is a two step process. First we generate code for executing a loop nest for a single tile. Then, we wrap this code in a loop that iterates over all the tiles assigned to a physical processor. Communication generation for tiles is handles similarly. Communication pinned inside a computational loop by data dependences is handled for a tile as if its virtual processor is a physical processor. Communication operations that are vectorized outside of all computational loops over data dimensions are each enclosed in their own tile enumeration loop.

3. MULTIPARTITIONING OPTIMIZATIONS

In this section we describe several key optimizations performed by the dHPF compiler that specifically target performance issues which arise in generating efficient compiler-parallelized code for multipartitioned computations. From a processor's perspective, a multipartitioned computation is organized as computation on each of a series of tiles. To avoid unnecessary serialization between physical processors, each processor's tile computations must be scheduled in the proper order. We describe our approach for tile scheduling in Section 3.1. To achieve good scalability, communication between processors must be as infrequent as possible. Sections 3.2 and 3.3 describe optimizations to reduce the frequency of communication for compiler-generated multipartitioned computations.

3.1 Tile Scheduling & Tile Loop Placement

To execute a loop nest operating on multipartitioned data, a processor performs a portion of the loop's computation for each of a sequence of virtual processor tiles. Each tile's computation may require communication. When tiles for a multipartitioned loop nest require communication for processor-crossing true data dependences, parallelism and overall performance is a function of the order in which tile computations are performed on each processor. To avoid unnecessary serialization, the tile enumeration order must be chosen carefully.

This problem of selecting the tile enumeration order is unique to multipartitioning. For block partitionings, tile enumeration order is trivial: each processor has only one block. For cyclic and block-cyclic partitionings, tile enumeration is straightforward—the enumeration is independent for each partitioned data dimension. For multipartitioned data arrays, tile enumeration order must be selected carefully because choosing to iterate through a processor's tiles in a particular direction along any one multipartitioned data dimension determines the iteration order for the other multipartitioned dimensions as well.

Figure 2 outlines the algorithm dHPF uses to select which multipartitioned data dimension will determine the tile enumeration order within a processor. If there is no communication for processor-crossing true dependences, then any tile enumeration order is equally good. When there is communication for processor-crossing true dependences, then the dimension along which the communication is flowing is selected as the tile dimension, and the tile iteration direction is selected to sweep along the tile dimension in the same direction as the communication. In one case, we note that any static tile schedule will cause deadlock; however, we can always avoid this case by exploiting the flexibility afforded by dHPF's general computation partitioning model. Namely,

```
ComputeTileEnumerationOrder(LoopNest)
  if ¬∃ processor-crossing dependences in LoopNest
    then
      tile iteration dimension = data dimension indexed
        by subscript with induction variable from
        shallowest loop
  else
    if comm. for processor-crossing dependences flows
      in both directions along a single dimension then
        error: any static tile schedule will cause deadlock
    else
      tile iteration dimension = data dimension along
        which comm. occurs which is indexed by
        induction variable of shallowest loop
  tile iteration direction = direction along which
    communication flows along tile dimension
```

Figure 2: Algorithm for selecting data dimension and direction for tile enumeration.

we can always choose computation partitionings so that all processor crossing dependences along only one dimension flow in the same direction.

As long as processor-crossing dependences flow along only one dimension of multipartitioned array, any tile enumeration order that does not sweep in the opposite direction of the communication will be correct. However, full parallelism will be realized for such computations only if the tile dimension is the same as the dimension along which communication occurs, and the tile enumeration order is the same as the direction of the communication. If a multipartitioned computation involves communication along more than one dimension, the computation will be partially serialized regardless of tile enumeration order.

3.2 Aggressive Communication Placement

In a loop nest that iterates over an array's data dimensions, processor-crossing true data dependences incident on a particular reference can be preserved by placing communication for that reference at the level determined by the data dependence. Loop-carried dependences can be preserved by placing communication within the carrying loop. Loop-independent dependences can be preserved by placing it at the loop level that is the least common ancestor of the dependence source and sink.

For a loop nest iterating iterating over a block partitioned array, communication that occurs only at a tile boundary can often be hoisted out of a bounds-reduced loop. However, hoisting such communication out of additional enclosing loops for block partitioned arrays could hurt performance: although it would reduce communication frequency, it would also reduce the potential for wavefront parallelism.

Loop nests iterating over the data dimensions of a multipartitioned array possess the unique property that if there is only communication along a single direction of a single data dimension, it can be hoisted out of all enclosing loops over data dimensions without reducing the available parallelism. In the dHPF compiler, we exploit this property to vectorize communication for processor crossing true dependences (both loop-carried and loop-independent) out of multipartitioned loops. This optimization enables programmers to nest loops over the dimensions of a multipartitioned array so that they will access memory using a stride-1 access pattern. In contrast, when a block distribution is used, either the programmer or the compiler must carefully arrange a loop nest with carried dependences to place the dependence-carrying loop at a level that balances communication frequency with wavefront parallelism; often this placement will be inconsistent with a stride-1 access pattern and hurt performance by reducing memory hierarchy utilization.

Although multipartitioned distributions enable full parallelism to be achieved for a loop nest that requires unidirectional communication along any one partitioned dimension, if processor-crossing true dependences flow in multiple directions or along multiple data dimensions, then the loop will be partially serialized.[4]

Communication placement in dHPF occurs in two steps. First, the compiler computes an initial placement in which data dependences are used to determine for each reference the loop level at which communication might be needed. This placement is safe for any data layout and any possible computation partitioning for the program's statements. Second, we apply a placement refinement algorithm that uses information about the data layout and computation partitionings selected to attempt to improve the initial placement.

Placement refinement performs a post-order traversal of the loops in a nest. For each loop, if all communication at the current loop level flows only along the tile iteration data dimension and direction (see Section 3.1), and the loop nest has only multipartitioned computation partitionings within, then all communication is hoisted out one level. We only hoist communication along the tile iteration dimension and direction because vectorizing other communication would increase serialization in a loop with communication in multiple dimensions or directions.

3.3 Aggregating Communication Across Tiles

A key property of multipartitioning distributions is that a single physical processor owns all of the tiles that are neighbors of a particular processor's tiles along any given direction. For example, consider the multipartitioning shown in Figure 1. The right neighboring tile (if any) of each of processor 1's tiles along the y dimension belongs to processor 5. Thus, if a processor's tiles need to shift data to their right neighbor along a particular dimension, the processor needs to send values to only one other processor.

This property must be exploited for multipartitioned computations to exhibit scalable performance. If not, a separate message would be sent per tile for a message that has been vectorized out of all loops over multipartitioned data with the result that a d-dimensional multipartitioning across p processors would require $O(p^{\frac{1}{d-1}})$ messages to perform one vectorized communication. Since the number of messages a processor must send using this strategy would grow in proportion to p, scalability would be poor.

To avoid this scaling problem, when the dHPF compiler generates code for each fully vectorized communication event, it packs data from all tiles on the owning processor and sends a single message to the corresponding recipient. This

[4]In such cases, loop distribution may be able to eliminate this serialization by converting the original loop nest into multiple loop nests with unidirectional, unidimensional communication that can realize full parallelism.

Receiver:	Sender:
∀ tiles, count msg bytes	∀ tiles, count data bytes
allocate a single buffer	allocate a single buffer
post a single receive	∀ tiles, pack data contiguously into buffer
	send data in a single msg
wait for data to arrive	wait for send to complete
∀ tiles, unpack data from buffer into storage for non-local data	

Figure 3: Communication event components and their interleaving to aggregate communication across tiles.

optimization is a major factor in reducing communication frequency, although it doesn't reduce its volume.

To support message aggregation across multiple tiles, the dHPF compiler splits communication tasks on both sending and receiving sides into a number of different components. Figure 3 shows the sequence of component operations executed by a sender and a receiver for a communication event. It also shows which components need to be executed for all the tiles a processor owns and which ones are executed as a single operation.

Sender's tasks:

Count: So that it can allocate a message buffer of the appropriate size, the sender determines the number of bytes that must be sent for each tile, and totals the sizes across all of its tiles. Typically, dHPF generates a closed formula for the send data per tile from the communication set.

Allocate: Allocate a buffer to hold the outgoing message.

Pack: Pack the data to be sent for each tile into the message buffer.

Initiate: Send the message buffer to the partner processor.

Wait: Wait for the message transmission to complete.

Receiver's tasks:

Count: The receiver counts the bytes of data to be received per tile and totals the sizes across all of its tiles.

Allocate: Allocate a buffer to hold the incoming message.

Initiate: Post a non-blocking receive operation.

Wait: Wait for message to arrive.

Unpack: Empty the communication buffer by scattering the received data into storage for non-local data (typically, shadow regions) for each tile.

For performance, dHPF uses asynchronous sends and receives. This requires that both the sender and receiver wait for the message transmission to complete separately from initiating the communication. The sender waits for transmission completion as its last operation. The receiver must wait for transmission completion before unpacking its message buffer.

Scheduling Communication

Low-overhead, high-efficiency communication requires avoiding serialization. Without careful scheduling of communication operations, the impact of serialization is significant. A principal cause of serialization we encountered was due to the interaction between communication scheduling within each SPMD process and the implementation of non-blocking message-passing communication primitives in the runtime library. According to the MPI standard, the completion of a send operation *may* be delayed, until a matching receive is posted. Our experience using MPI implementations by both IBM and SGI is that if a non-blocking `mpi_irecv` has not been posted before its matching `mpi_isend` is initiated, a substantial delay may occur.

To address this issue, dHPF generates code that posts MPI receive operations as soon as possible. In fact, dHPF posts receives for loop-carried communication before the dependence-carrying tile loop begins. (Of course, the receive completion check and data unpacking operations **must** to occur within the tile loop, along with the sender's operations.) Posting these receives early helps avoid unnecessary delays when the sender initiates communication. Separating communication into components facilitated reorganizing communication for this purpose.

4. GENERAL OPTIMIZATIONS

In our quest to tune dHPF's generated code to achieve hand-coded performance, we devised several new types of optimizations that improved the overall quality of our generated code.

Most of these optimizations are refinements of our analysis and code generation strategies that construct and manipulate sets of integer tuples using the Omega Library [12]. Omega represents sets of integer tuples with Presburger Formulae [9] as constraints. The set-based optimizations we describe here improve the ability of dHPF to analyze and optimize set representations that arise with complex programs. Omega operations used in equations in the following sections are defined in Appendix A.

4.1 Formulating Compact Sets

Our experience using the Omega library for data-parallel program analysis and code generation led us to recognize that compilation time and generated-code complexity are directly related to the complexity of the set representation. Omega contains a sophisticated procedure for generating code templates that enumerate the elements of a set [13]. However, if the sets are complicated, non-convex disjunctions then these code templates will be complex and inefficient. Within the dHPF compiler, we try to ensure that sets are represented in their simplest possible form, namely, as convex sets or as unions of simple, convex sets.

Our flexible computation partitioning strategy can give rise to disjunctive sets. For one of the benchmarks we studied, communication analysis failed as it tried to manipulate an overly complex set representation. The problem was caused by selecting partially-replicated non-owner-computes partitionings for loops with unrolled code. Computation partitioning analysis resulted in a set representation that was a disjunction of 25 conjuncts. Inspection of the set showed that its form was $\bigcup_{1\leq l_1,l_2\leq 5}\{[l_1,l_2,i,j,k] : i_l < i < i_u \ \&\ j_l < j < j_u \ \&\ k_l < k < k_u\}$, where i_l, i_u, j_l, j_u, k_l, and k_u are symbolic constants for the upper and lower bounds of i, j, and k, respectively. Intuitively, this can be represented compactly as a single conjunct where the constants in the first two positions of the tuples replaced by the ranges $1 \leq l_1, l_2 \leq 5$. We discovered that sets with compact ranges of constant terms could be collapsed by recomputing a set S as `Hull`$(S) - ($`Hull`$(S) - S)$. This process forces the set to be recomputed as a difference from a single conjunct, which has the effect of reducing the set to its most compact form.

4.2 Communication Factoring

A major set class within our analysis framework, are the sets that represent communication. For each logical communication event, we construct integer sets which describe both the data to be sent and received, and the partner processors involved in the communication event.

As a consequence of non-owner computes computation partitionings, a single reference can cause communication in multiple directions. In particular, with a block-style distribution in which computation has been partially-replicated for a strip around the block boundaries, the right-hand-side reference will cause communication both forward and backward along each of the partitioned data dimensions. While the closed-form representation of communication sets used in dHPF can represent the fact that nearest-neighbor communication is needed for all of a block's neighbors, the code generated for such communication is innefficient because for each neighbor, it must check which face of the non-convex communication set is needed. By factoring the communication event into simpler communication events—in this case, a separate one for each data dimension and each communication direction—the resulting communication is more efficient and the generated code is much shorter. After factoring the communication, each communication event involves manipulating only a simple convex set.

4.3 Communication Set Construction

Due to the use of advanced optimizations, such as partial computation replication, non-owner computes computation partitioning and communication event aggregation, the resulting data and processor sets can become unwieldy and costly to analyze.

By carefully constructing the sets to produce equivalent, but much more simple forms, compilation and code generation are sped up significantly and made to produce cleaner and simpler output. To achieve this, we group data references needed to construct the communication sets into equivalence classes. These equivalence classes are formed by inspecting the value number of each subscript expression in the communication-causing dimension and grouping conformant references together. References with different value numbers in partitioned dimensions should be grouped into different equivalence classes.

Splitting references into equivalence classes leads to a simple convex representation for each equivalence class. These resulting sets are then unioned together to produce the final result. This final resulting set is, of course, equivalent to one that would be obtained by processing the references in any order.

4.4 Communication Subsumption

Communication is initially scheduled independently for each data reference. This fact and the use of non-owner computes computation partitionings can produce separate logical communication events which correspond to sending and receiving the same data or subsets of it.

Identifying which logical communication events move subsets of data moved by other communication events enables us to eliminate redundant events, thus reducing the total communication volume. One difficulty in detecting redundancy is that communication events draw their representation from data references and computation partitionings with the result that the equivalence between two communication events is not easily determined.

For example, the data reference `a(i, j, k)` with a computation partitioning of `ON_HOME a(i+1, j, k)` causes communication for the last row of data in each tile. The data reference `a(i-1, j, k)` with a partitioning of `ON_HOME a(i, j, k)` requires exactly the same data to be communicated.

By analyzing the *shift offsets* of each communication event, a measure of how wide a panel of data will be communicated and by using the fact that multi-dimensional communication has already been split into its constituent dimensions, we can identify such cases and determine which communication events are covered by others and safely eliminate them.

4.5 General Communication Aggregation

Rather than sending separate messages for each distributed array to be communicated, the dHPF compiler is able to combine multiple messages addressed to the same recipient from a particular sender into a single message when certain criteria are satisfied. This optimization allows us to reduce communication frequency even more, at the minor cost of sending bulkier messages.

Aggregation Criteria

Two single logical shift communication events for different distributed arrays may be safely aggregated if the communication characteristics are *conformant*. First of all, both arrays should be mapped to the same HPF template. The communication should flow in one direction along the same dimension of the array. This guarantees that the partner processor sets are conformant too.

4.6 Code Generation

Generating SPMD code templates (that contain appropriately bounds-reduced loops and conditionals which partition the computation among the available processors) for vectors of disjunctive iteration spaces is a complex problem we encounter when generating code for iteration spaces that have been partially-replicated along data partitioning boundaries.

While Omega contains a sophisticated procedure for generating code templates for complex iteration spaces [13], we found that applying Omega's code generation algorithm to vectors of disjunctive iteration spaces often produced inefficient code templates in which a set of guards terms is repeatedly tested and placeholders for code fragments may repeat many times. In the dHPF compiler, we take the following steps to generate high-quality code for such complex iteration spaces.

Avoid testing conditions already known to be true. Exploit context information when generating a code template for an iteration space by eliminating any constraints from the iteration space that are universally true, or that the enclosing context makes true. We pass the intersection of universal known information and the computation partitioning for a scope's parent as known constraints when generating code for a statement's iteration space. Omega's code generation interface accepts one conjunct expressing such known constraints.

Avoid repeatedly testing the same conditions for multi-

ple iteration spaces. To generate code for a set of iteration spaces $I_j, 0 \leq j < n$, first we determine the satisfiability conditions S_j for each iteration space I_j as $S_j = \texttt{Project\_On\_Sym}(I_j)$ and compute any satisfiability constraints common to all iteration spaces as $C = \texttt{Hull}(\bigcup_j S_j)$. If these constraints are non-trivial, simplify each iteration space I_j by removing the common satisfiability constraints to create $I_j^{\prime}$, where $I_j^{\prime} = \texttt{Gist}(I_j, C)$. Next, separately generate code for C and the simplified iteration spaces $I_j^{\prime}$, for all j. Finally, enclose the code generated for all $I_j^{\prime}$s in a single copy of the conditional generated for C.

Avoid code replication for disjunctions. When all of the conjuncts in a disjunctive iteration space differ only in satisfiability constraints that are independent of the iteration space range, factor out common satisfiability constraints and enclose the iteration space in a conditional as described in the previous bullet, then generate a second conditional that tests the disjunction of the remaining satisfiability constraints $D = \bigcup_j \texttt{Gist}(S_j, C)$ and wrap it around the code generated for the simplified iteration space.

Avoid conditions in guards that repeat tests in loop bounds. Using the strategy described above for computing S_j, S_j can contain constraints that will also be enforced by the code generated for the simplified iteration space $I_j^{\prime}$. If $I_j^{\prime}$ contains a loop with one or more symbolic bounds, then the satisfiability constraints will contain a test comparing the upper and lower bounds to ensure the iteration set is non-empty. Since an empty loop will not be entered, it is not necessary for the satisfiability conditions to contain this test. We refined the strategy for computing S_j to avoid getting loop bound comparisons as part of the satisfiability conditions. The strategy for a refined S_j, which we denote R_j for simplicity, is $R_j = Gist(S_j, Project\_On\_Sym(I_j^{\prime}))$. This computation removes constraints that from S_j that $I_j^{\prime}$ will enforce and creates the unique satisfiability constraints we desire.

4.7 Generating Optmizable Code

The dHPF compiler translates an HPF code into SPMD Fortran 77 code with MPI message-passing communication. Our generated code will achieve high scalar performance only if the Fortran compiler on the target system can generate good object code for the SPMD code produced by dHPF. To date, we have focused on achieving high performance on a single target platform: an SGI Origin 2000 with MIPS R10000 processors using SGI's MIPSpro Fortran compiler. Obtaining scalar performance with our SPMD code that is competitive with that of the original sequential application proved difficult. Below we outline the key problems areas in which we had difficulty and the approaches required to boost SPMD scalar performance to the level of the sequential code. We believe that many of these backend code generation issues apply to other target platforms and are not necessarily exclusive to the SGI MIPS platform.

Prefetching. Software prefetching of data references is necessary for good performance on the MIPS R10000. Several adjustments to dHPF's generated code were necessary to ensure the SGI compiler added prefetches where necessary in the object code.

Use Cray pointer notation for dynamically-allocated arrays. The SGI compiler would not prefetch arrays indexed with linearized subscripts.

Avoid subscripts containing multiplication. The SGI compiler would not prefetch references of the form `a(i - mypid * bsize)`.

Guard against forward substitution of products into subscripts. We had to transform assignments `offset = mypid * bsize` into `offset = identity(mypid * bsize)`, where `identity` is a function call that returns its argument so offset was not forward substituted into subscripted references such as `a(i - offset)`.

Ensure back-end compiler infers no aliasing. We initialize Cray pointers for arrays as `loc(heap(offset))` where the *loc* intrinsic returns the address of its argument. The `heap` array is used as a Fortran-level representation of the dynamic memory space. Since different pointers were assigned using different offsets into `heap`, the back end compiler inferred aliasing. We were able to hide this by declaring `loc` as `external` so the compiler's knowledge of the semantics of the `loc` intrinsic was not applied.

Cache misses. Since sizes for multipartitioned array tiles are determined dynamically from the number of processors, the dHPF compiler needed to generate code to pad the tiles to guard against conflict misses. Our simple scheme merely padded each dimension to an odd length, which effectively diminishes potential conflict misses.

5. EXPERIMENTAL RESULTS

In this section, we describe results of our experiments using the dHPF compiler to generate optimized multipartitioned parallelizations of serial versions (NPB2.3-serial release) of the NAS BT and SP application benchmark codes [3] developed by NASA Ames.

BT and SP are two simulated computational fluid dynamics applications that solve systems of equations resulting from an approximately factored implicit finite-difference discretization of three-dimensional Navier-Stokes equations. The principal difference between the codes is that BT solves block-tridiagonal systems of 5x5 blocks, whereas SP solves scalar penta-diagonal systems resulting from full diagonalization of the approximately factored scheme [3]. Both codes consist of an initialization phase followed by an iterative computation. In each time step of the iterative computation, the codes calculate boundary conditions and then compute the right hand sides of the equations. Next, they solve banded systems in three computationally-intensive bidirectional sweeps along each of the three spatial dimensions. Finally, they update flow variables. Parallel implementations of these computations require communication during the setup phase and the forward and backward line sweeps along each dimension.

In our parallelizations of these codes with dHPF, the optimizations we described in Sections 3 and 4 removed most of the performance bottlenecks that we encountered previously [7] in code generated with our first prototype implementation of multipartitioning. We briefly outline where our optimizations were applicable with these codes, describe their impact on performance, compare the performance of our compiler-based parallelizations to that of hand-coded parallel implementations, and identify likely causes of remaining performance differences.

Our experiments with versions of the NAS SP and BT codes were performed on an SGI Origin 2000 node of ASCI Nirvana (128 250MHz R10000, 32KB (I)/32KB (D) L1, 4MB

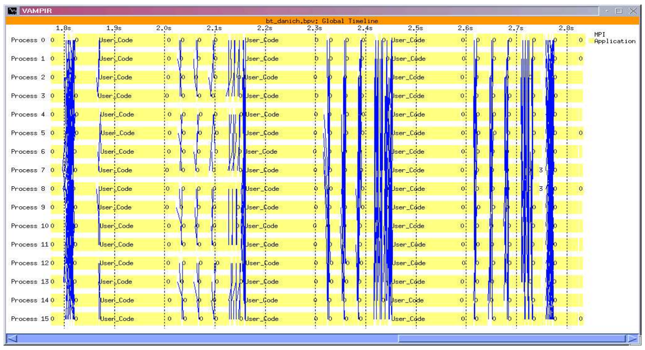

Figure 4: dHPF-generated NAS BT using 3D multipartitioning.

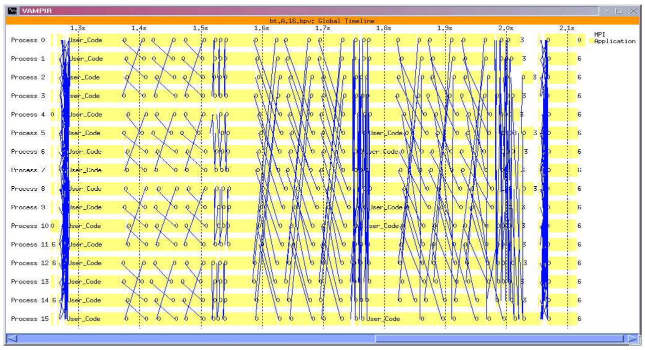

Figure 5: MPI hand-coded NAS BT using 3D multipartitioning.

L2 (unified)).

5.1 NAS BT

Effectively parallelizing the NPB2.3-serial version of BT with dHPF required coordinated application of a broad spectrum of analysis and code generation techniques. The support for multipartitioning and new optimizations in dHPF improved both the performance and scalability of the generated code compared to our earlier parallelizations [1] based on block partitionings. Both the parallel execution profile and the sequential performance of dHPF's compiler-multipartitioned code is nearly equivalent to the hand-coded version. Figure 4 shows a 16-processor parallel execution trace for one steady-state iteration of our compiler-generated multipartitioned code for class A problem size. Compared to the corresponding execution trace of the hand-coded multipartitioning shown in Figure 5, one can see that our dHPF-generated code achieves the same qualitative parallelization.

Using non-owner computes computation partitionings to partially-replicate computation along multipartitioned tile boundaries led to dramatic reductions of communication volume. In BT's `compute_rhs` subroutine, partially replicating the computation of the privatizable temporary arrays `rho_q`, `qs`, `us`, `vs`, `ws`, and `square` along the boundaries of a multipartitioned tile avoided communication of these six variables. No additional communication was needed to partially-replicate this computation because the boundary planes of the multipartitioned `u` array needed by the replicated computation were already being communicated in this routine. This optimization cut the communication volume of `compute_rhs` by nearly half.

# CPUs	hand-coded	dHPF	% diff.
1	1.06	1.09	-2.64
4	3.28	3.34	-1.77
9	7.73	7.26	6.14
16	14.21	13.49	5.10
25	21.08	20.66	1.98
36	29.78	28.77	3.40

Table 1: Comparison of hand-coded and dHPF speedups for NAS BT (class A).

# CPUs	hand-coded	dHPF	% diff.
1	0.98	0.92	5.85
4	3.37	2.91	13.48
9	4.91	5.63	-14.70
16	12.30	12.83	-4.34
25	19.09	19.91	-4.30
36	30.95	28.80	6.93

Table 2: Comparison of hand-coded and dHPF speedups for NAS BT (class B).

In BT's `lhsx`, `lhsy`, and `lhsz` subroutines, partially replicating computation along the partitioning boundaries of two arrays, `fjac` and `njac`, whose global dimensions are `(5, 5, IMAX, JMAX, KMAX)`, saved a factor of five in communication. Rather than communicating planes of computed values for these arrays across partitions in the `i`, `j`, and `k` dimensions, we communicated sections of `rhs(5, IMAX, JMAX, KMAX)`, which is a factor of five smaller, to replicate the computation along the partitioning boundaries.

Tables 1 and 2 compare speedups of NASA Ames' hand-coded parallelization using multipartitioning with those of our dHPF-generated code. All speedups are relative to the performance of the original sequential code. The columns labeled "% diff" show the difference between the speedup of our dHPF-generated version and the speedup of the hand-coded version, relative to speedup of the hand-coded version.

For the class A problem size, dHPF's generated code was at worst about 6% slower than the hand-coded version. For 36 processors (the largest number tested), our compiler-generated code was only roughly 3% slower. For the class B problem size, dHPF's generated code and the hand-coded version are also quite comparable. In half of the cases, dHPF-generated multipartitionings achieved better performance than the hand-coded version. For the largest number of processors tested, the dHPF-generated multipartitionings were 7% slower.

The excellent parallel performance and scalability of the compiler-generated code comes from partially-replicated computation partitionings, aggregation across tiles, aggregation between messages for different arrays, and aggressive communication placement. A measure of the effectiveness of these communication optimizations is that for a 16-processor class A execution, dHPF had only 1.5% higher communication volume, and 20% higher message frequency than the hand-coded implementation.

# CPUs	hand-coded	dHPF	% diff.
1	1.01	0.96	5.50
4	4.21	3.21	23.72
9	11.60	7.51	35.30
16	16.21	14.13	12.85
25	21.00	20.13	4.15
36	30.69	26.92	12.26

Table 3: Comparison of hand-coded and dHPF speedups for NAS SP (class A).

# CPUs	hand-coded	SP dHPF	% diff.
1	0.80	0.78	2.67
4	2.86	2.52	12.13
9	7.74	6.17	20.26
16	13.01	11.29	13.22
25	22.15	17.11	22.75
36	36.52	25.69	29.65

Table 4: Comparison of hand-coded and dHPF speedups for NAS SP (class B).

5.2 NAS SP

From a compiler's point of view, the NAS SP benchmark presents a significant challenge for achieving high performance. To generate efficient parallel code from a lightly modified serial version of this benchmark with a multipartitioning distribution, requires the use of several advanced compilation strategies [1, 7] including non-owner-computes computation partitionings, complex patterns of computation replication to reduce communication for privatizable arrays, aggressive communication coalescing and aggregation and a mix of complex computation partitionings. Despite the fact that the dynamic communication patterns of our compiler-generated multipartitioned parallelization, resemble those of the hand-coded parallelization, there is still a performance gap between our dHPF-generated code and the hand-coded MPI implementation. This gap comes from extra communication volume present in the compiler-generated code for SP's `lhs<xyz>` routines. Currently, the dHPF compiler uses a procedure-local communication placement analysis. This approach may schedule communication in each procedure even though the values might already be available as a result of prior communication. Interprocedural communication placement and interprocedural availability analysis would be needed to eliminate this additional communication automatically.

For SP, non-local values gathered by the `compute_rhs` routine cover the non-local values needed by the `lhsx`, `lhsy` and `lhsz` routines. In a 16-processor execution for a class A problem size, this unnecessary communication in `lhsx`, `lhsy` and `lhsz` causes the communication volume of our dHPF-generated code to be 14.5% higher than for the hand-coded parallelization. Although the additional volume is modest, this unnecessary communication also causes the communication frequency of the dHPF-generated code to be 74% higher than the hand-coded parallelization.

Like with BT, partially-replicating computation at the boundaries of multipartitioned tiles offered significant benefits for SP. In SP's `lhsx`, `lhsy`, and `lhsz` routines, replicating computation of boundary values of `cv`, a partitioned 1-dimensional vector aligned with a multipartitioned template, eliminated the need to communicate these boundary values at loop depth two between their definition and use. Although partially replicating computation of `cv` required communicating two additional planes of the three-dimensional multipartitioned array `us` in each of these routines, this communication was fully vectorizable, whereas the communication of `cv` we avoided was not.

In SP's `x_solve`, `y_solve`, and `z_solve` routines, we found that the dHPF compiler chooses to generate extra fully vectorized communication if this will reduce communication pinned inside a loop nest. With the multipartitioning distribution, communication inside a loop nest doesn't have such a high cost, and in the case of the sweep routines, a moderate increase in communication frequency, should be traded for a significantly reduced total communication volume. By compiling these routines using an owner-computes scheme, we effectively get a reduced communication volume with no penalty due to the communication vectorization property of multipartitioned loops.

Tables 3 and 4 show speedup measurements for SP taken in February 2001 for the class A and B problem sizes , respectively. All speedups shown in the figure are relative to the performance of the sequential code for the respective NPB2.3-serial distribution.

6. RELATED WORK

The closest similar distributions supported by parallelizing compilers in the literature are the generalized block distribution and variants thereof, including a general BLOCK distribution specified as an approved extension in the HPF2 standard [6, 14, 8]. These distributions divide an array into rectangular blocks of potentially unequal size, and assign a single block per processor. In contrast, multipartitioning assigns multiple blocks of identical size to each processor.

7. CONCLUSIONS

By using multipartitioning, hand-coded parallelizations of tightly-coupled line-sweep computations, such as the NAS SP and BT application benchmarks, achieve very good scalability as long as the problem size is large enough for each processor to have a reasonable amount of computation. Analysis and experiments with hand-coded line-sweep implementations found that multipartitioning has better scalability and performance than any block partitioning strategy [16, 17]. Therefore, it is not surprising that compiler-directed parallelizations of these applications using block partitionings have not achieved performance comparable to hand-coded implementations using multipartitioning [1]. With compiler-based support for multipartitioning, we have been able to narrow the gap between compiler-generated parallelizations for line-sweep computations and their hand-coded counterparts to just a few percent.

In our quest to match hand-coded performance with code generated by the dHPF compiler, it became clear that achieving high performance with compiler-generated parallelizations requires getting *all* of the details perfect. While compiler directed multipartitioned distributions lead to balanced computation, matching hand-coded performance requires closely matching the communication frequency and volume, along with the scalar performance characteristics as well. The optimizations described in this paper enabled us to match the scalar performance and significantly reduce the communication frequency and volume.

The success of our compiler optimizations is principally a result of our formalization of data-parallel analysis and code generation as manipulation of sets of integer tuples [2]. Using this general framework enables us to compose sophisticated optimization techniques with multipartitioned data distributions. For example, this approach enables us to compile a data-parallel program for a symbolic number of processors using a multipartitioned data distribution and still vectorize communication resulting from non-owner-computes computation partitionings that arise when partially replicating computation arrays at processor boundaries.

Our preliminary experiments with two representative line-sweep application codes show that our compiler technology has succeeded in simultaneously optimizing all aspects of program performance. We have demonstrated that it is possible to use data-parallel compiler technology to effectively parallelize third-party tightly-coupled sequential line-sweep applications (augmented with data layout directives) and yield performance that is competitive with sophisticated hand-coded parallel versions of these codes.

Acknowledgments

We thank Vikram Adve for his involvement in the early design and implementation discussions of this work.

This work has been supported by NASA Grant NAG 2-1181, sponsored by DARPA and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0159, and supported in part by the Los Alamos National Laboratory Computer Science Institute (LACSI) through LANL contract number 03891-99-23, as part of the prime contract (W-7405-ENG-36) between the Department of Energy and the Regents of the University of California. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of DARPA and Rome Laboratory or the U.S. Government.

8. REFERENCES

[1] V. Adve, G. Jin, J. Mellor-Crummey, and Q. Yi. High Performance Fortran Compilation Techniques for Parallelizing Scientific Codes. In *Proceedings of SC98: High Performance Computing and Networking*, Orlando, FL, Nov 1998.

[2] 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.

[3] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, Dec. 1995.

[4] Z. Bozkus, L. Meadows, S. Nakamoto, V. Schuster, and M. Young. Compiling High Performance Fortran. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 704–709, San Francisco, CA, Feb. 1995.

[5] J. Bruno and P. Cappello. Implementing the beam and warming method on the hypercube. In *Proceedings of 3rd Conference on Hypercube Concurrent Computers and Applications*, pages 1073–1087, Pasadena, CA, Jan. 1988.

[6] B. Chapman, P. Mehrotra, and H. Zima. Extending hpf for advanced data parallel applications. Technical Report TR 94-7, Institute for Software Technology and Parallel Systems, U niversity of Vienna, Austria, 1994.

[7] D. Chavarría-Miranda and J. Mellor-Crummey. Towards compiler support for scalable parallelism. In *Proceedings of the Fifth Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers*, Lecture Notes in Computer Science 1915, pages 272–284, Rochester, NY, May 2000. Springer-Verlag.

[8] N. Chrisochoides, I. Kodukula, and K. Pingali. Compiler and runtime support for irregular and adaptive applica tions. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, pages 317–324, Vienna, Austria, July 1997.

[9] D. C. Cooper. Theorem proving in arithmetic without multiplication. In *Machine Intelligence*, volume 7, pages 91–99, New York, 1972. American Elsevier.

[10] M. Gerndt. Updating distributed variables in local computations. *Concurrency: Practice and Experience*, 2(3):171–193, Sept. 1990.

[11] S. L. Johnsson, Y. Saad, and M. H. Schultz. Alternating direction methods on multiprocessors. *SIAM Journal of Scientific and Statistical Computing*, 8(5):686–700, 1987.

[12] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega Library Interface Guide. Technical report, Dept. of Computer Science, Univ. of Maryland, College Park, Apr. 1996.

[13] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *Frontiers '95: The 5th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, Feb. 1995.

[14] A. Müller and R. Rühl. Extending high performance fortran for the support of unstructu re computations. Technical Report TR-94-08, Ecole Polytechnique Fedérale de Zurich, November 1994.

[15] N. Naik, V. Naik, and M. Nicoules. Parallelization of a class of implicit finite-difference schemes in computational fluid dynamics. *International Journal of High Speed Computing*, 5(1):1–50, 1993.

[16] V. Naik. Performance effects of load imbalance in parallel CFD applications. In *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, 1992.

[17] R. F. Van der Wijngaart. Efficient implementation of a 3-dimensional ADI method on the iPSC/860. In *Proceedings of Supercomputing 1993*, pages 102–111. IEEE Computer Society Press, 1993.

APPENDIX

Definitions of Omega set operations

- Hull: If r_1 is a general integer set, hull(r_1) computes a single convex set that contains all elements in r_1, but often other elements as well.
- Gist: If $r_1 = \{x_1 \to y_1 \mid f_1(x_1, y_1)\}$ and $r_2 = \{x_2 \to y_2 \mid f_2(x_2, y_2)\}$ then the result of applying $\texttt{Gist}(r_1, r_2)$ is $\{x \leftarrow y \mid f(x, y)\}$. Where f is a Presburger formula such that $\forall x, y\ f(x, y) \wedge f_2(x, y) \Leftrightarrow f_1(x, y)$.
- Project_On_Sym: If $r = \{x \leftarrow y \mid f(x, y)\}$ then $\texttt{Project\_On\_Sym}(r) = \{\ [\ ]\ \exists x, y \text{ s.t. } f(x, y)\}$.