Appears in PPOPP 97

A New Model for Integrated Nested Task and Data Parallel Programming

Jaspal Subhlok and Bwolen Yang
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
{jass,bwolen}@cs.cmu.edu

Abstract

High Performance Fortran (HPF) has emerged as a standard language for data parallel computing. However, a wide variety of scientific applications are best programmed by a combination of task and data parallelism. Therefore, a good model of task parallelism is important for continued success of HPF for parallel programming. This paper presents a task parallelism model that is simple, elegant, and relatively easy to implement in an HPF environment. Task parallelism is exploited by mechanisms for dividing processors into subgroups and mapping computations and data onto processor subgroups. This model of task parallelism has been implemented in the Fx compiler at Carnegie Mellon University. The paper addresses the main issues in compiling integrated task and data parallel programs and reports on the use of this model for programming various flat and nested task structures. Performance results are presented for a set of programs spanning signal processing, image processing, computer vision and environment modeling. A variant of this task model is a new approved extension of HPF and this paper offers insight into the power of expression and ease of implementation of this extension.

1 Introduction

Use of high level languages is gaining popularity as a means to portable and efficient parallel programming. Research and development in parallel languages has primarily focused on data parallelism [4, 11, 18] and High Performance Fortran [12] has emerged as a promising standard language for parallel programming. A disadvantage of using a parallel language like HPF, as compared to explicit parallel programming using, e.g., a message passing library like MPI [8], is that the user is constrained by the models of parallelism supported by the language. Application developers are often reluctant to use HPF because many applications do not completely fit the HPF data parallelism model. This paper presents a simple model of task parallelism which significantly extends the domain of applications that can be efficiently programmed in a language like HPF. A variation of this model is now an approved extension of HPF [10].

Most parallel applications contain data parallelism, but a large number of such applications need a combination of task and data parallelism to represent the natural computation structure or to enhance performance. Pipelines of data parallel tasks [5], a common computation structure in image processing, signal processing, and computer vision, require a combination of task and data parallelism to achieve good performance. The reason is that individual data sets for these applications are often small, and therefore, a pure data parallel implementation may not scale well. Multiblock codes containing irregularly structured regular meshes [1] are more naturally programmed as interacting tasks with each task representing a regular mesh, rather than as a single large irregular application. Tree structured algorithms like Barnes-Hut for N-body problems [2] need task parallelism to recursively divide the available processors to solve the recursively generated subproblems. Multidisciplinary applications like air quality modeling [13] are often modeled as sets of interacting modules, where each module represents a different scientific discipline. Finally, applications that interact with external devices like cameras and display devices need special tasks for such activities.

Integration of task and data parallelism is currently an active area of research and several compiler based and runtime system based approaches have been proposed [3, 6, 7, 9]. Compiler based solutions presented in the literature are either limited in terms of the forms of task parallelism structures they can support or have not demonstrated that they can be compiled efficiently. Runtime solutions have the advantage that they do not require new compiler technology, but they require that the programmer manage task parallelism at a lower level than data parallelism, and lack the advantages and flexibility of a single programming and compilation framework.

This paper presents a new model for integrated task and data parallelism. The central idea is to support the ability to divide the executing processors into *processor subgroups* and map data and computation onto these subgroups. A proces-

Effort sponsored by the Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0287. 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 necessarily representing the official policies or endorsements, either expressed or implied, of the Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

sor subgroup can be further divided into smaller subgroups allowing nested parallelism. The language extensions proposed are simple and minimal yet powerful enough for the expression of most common forms of mixed task and data parallelism including nested data parallelism. We show that this model can be compiled efficiently in an HPF framework with relative ease and present results from a diverse set of applications that use task and data parallelism.

The paper is organized as follows. Section 2 introduces our model of task parallelism and Section 3 shows how it is used to program different task structures. Section 4 outlines some of the issues in implementing this model efficiently. Section 5 describes the implementation and performance results from a set of applications. Section 6 discusses related work and Section 7 contains conclusions.

2 Model

We present our model of integrated task and data parallelism which has been implemented in the Fx parallelizing compiler. Fx provides directives to guide the layout of data arrays onto groups of processors to help exploit data parallelism [18]. Supported data distributions include block, cyclic and block-cyclic. Loop parallelism is expressed by a special parallel loop construct that combines loop and reduction parallelism [24]. Since Fx is conceptually similar to HPF in terms of data parallelism, and the details are not directly relevant to this research, we will not discuss them any further.

Task parallelism in our model is based on the concept of a *processor subgroup*, which is a collection of processors executing in data parallel manner. An instance of a program statement is executed by a processor subgroup, which may include all processors executing the program, or only a subset of them. The processor subgroup executing a statement is the *current* processor subgroup, and the corresponding processors are the *current* processors[1]. Task parallelism is obtained by dividing the current processors into processor subgroups and performing independent data parallel computations on disjoint processor subgroups.

In this section we describe the directives for task parallelism. These directives are used to divide the current processors into subgroups, map data structures onto processor subgroups, and specify independent task parallel computations. We also describe the task parallel execution model obtained by using the directives.

2.1 Task parallelism directives

Task parallelism is expressed using *declaration* directives that appear in the declarative section of a procedure, and *execution* directives that appear in the executable section of a procedure. The declaration directives are `TASK_PARTITION` directive, which defines a template for dividing processors into subgroups, and `SUBGROUP` directive, which maps variables onto processor subgroups. The execution directives include `TASK_REGION` directives, which partition current processors and define the task parallelism region, and `ON SUBGROUP` directives which specify the execution of a code block on a processor subgroup.

[1] A similar concept is described with *active* processors in HPF terminology [10]

Declaration directives

A `TASK_PARTITION` directive provides a template for partitioning the current group of processors into subgroups. As an example:

```
TASK_PARTITION myPart:: some(5), many(n - 5)
```

declares a partition named `myPart` (partition name is optional if there is only one task partition in a procedure) by dividing the current processors into two named subgroups: subgroup `some` is assigned `5` processors and subgroup `many` is assigned `(n-5)` processors. An important special case is obtained by replacing variable `n` with `NUMBER_OF_PROCESSORS()`, a function that returns the number of processors in the current group:

```
TASK_PARTITION myPart:: some(5),
                    many(NUMBER_OF_PROCESSORS() - 5)
```

This directive assigns `5` processors to subgroup `some` and the remaining processors to subgroup `many`.

The expressions in a task partition directive can use formal procedure parameters, and hence the partitioning can be different on different invocations of a procedure, depending on the values of the actual parameters. A subprogram unit can have multiple task partition directives to declare multiple templates for partitioning the current processor group.

A `SUBGROUP` directive is used to map (or attach) variables to a named processor subgroup formed with a task partition directive. These mapped variables will be referred to as *subgroup variables*. As an example:

```
SUBGROUP(some):: some_low, some_high
```

maps two variables `some_low` and `some_high` to the processor subgroup `some`. Each variable can be mapped to at most one processor subgroup. Variables that are not explicitly mapped to a processor subgroup will be mapped to all processors in the current processor group.

Alignment directives can be used only among variables mapped to the same subgroup and distribution directives are with respect to their corresponding processor subgroup. In the above example:

```
DISTRIBUTE some_low(BLOCK)
```

will distribute the elements of array `some_low` in block fashion *only* among the processors assigned to subgroup `some`.

Execution directives

`TASK_REGION` directives are used to activate and deactivate a partition of current processors into subgroups. A block of code delimited by `BEGIN TASK_REGION` and `END TASK_REGION` directives is a task region. A task region is a single–entry single–exit block of code that can include `ON SUBGROUP` regions. The `BEGIN TASK_REGION partitionName` directive activates the partitioning of the current processors into subgroups declared by the corresponding `TASK_PARTITION partitionName` statement and an `END TASK_REGION` directive deactivates this partition. The partition name is optional if there is only one `TASK_PARTITION` declaration, and hence only one processor partitioning template, in a procedure. Variables mapped to a subgroup in a partitioning template can be accessed only inside a corresponding task region.

An ON SUBGROUP region is a single–entry single–exit block of code delimited by ON SUBGROUP subgroupName and END ON, and is used to advise the compiler that code in this block should be executed on the processors of the named subgroup. The code inside an ON SUBGROUP region can only access variables mapped to that subgroup *except* for replicated variables that are read-only for the duration of the ON region. Note that there are no special restrictions on variable access inside a task region but outside of ON SUBGROUP blocks; i.e., all variables in the scope of the current procedure and all variables mapped to the subgroups of the current partition can be accessed. Further, code in two ON SUBGROUP blocks directed to execute on different subgroups cannot have *interfering I/O*, which intuitively means that there should be no concurrent accesses to the same files.

Lexical nesting is not permitted in task regions or ON SUBGROUP regions. However, a procedure called from an ON SUBGROUP region can partition its processors with another task region directive. Thus, dynamic nested partitioning of processors is allowed.

The following code illustrates simple usage of task regions. We assume that there is only one TASK_PARTITION statement and hence there is no explicit partition name. The start of a task region with statement BEGIN TASK_REGION activates the partitioning of current processors into subgroups some and many. The corresponding subgroup variables are also activated. The assignment "some_low = ..." will be executed on processors of subgroup some and assignment "many_high = f(many_low)" will be executed on processors of subgroup many. The assignment "many_low = some_low" is not in an ON SUBGROUP block and will be executed on processors assigned to both subgroups, which is all the processors executing the task region.

```
TASK_PARTITION:: some(5),
                 many(NUMBER_OF_PROCESSORS() - 5)
SUBGROUP(some):: some_low, some_high
SUBGROUP(many):: many_low, many_high
   .....
BEGIN TASK_REGION
   ON SUBGROUP some
      some_low = ...
   END ON
   many_low = some_low
   ON SUBGROUP many
      many_high = f(many_low)
   END ON
   ...
END TASK_REGION
```

2.2 Execution Model

The task parallelism directives are in the form of assertions about the code and hints to the compiler, and hence do not introduce any new semantics. The compiler uses these directives and program analysis to exploit task parallelism. The user asserts that the code in ON SUBGROUP regions accesses only local subgroup variables (with the exception of read only scalars) and that code in different ON SUBGROUP regions cannot have I/O related dependences. The compiler can therefore safely schedule code in different ON SUBGROUP regions in parallel. Further, whenever the compiler can determine that only a subset of current processors are required to execute a code block outside ON SUBGROUP blocks, it uses that information to allow non-participating processors to skip that code safely.

To explain the execution model, we classify the code inside a task region into two scopes: the subgroup scope and the parent scope. All the code blocks that are enclosed in ON SUBGROUP regions belong to the *subgroup* scope. Code blocks that do not belong to any ON SUBGROUP region are in the *parent* scope.

The execution of a subprogram with a task region proceeds as follows. The code is executed normally in data parallel mode on current processors until a task region is encountered. Inside the task region, code blocks in subgroup scope are executed only by the processor subgroup specified in the corresponding ON SUBGROUP directive. Processors not belonging to the named subgroup of an ON SUBGROUP region can skip past the region. The code in the parent scope is executed by all current processors, which includes the processors in all the subgroups of the task region, in normal data parallel mode. It is important to note that normal mechanisms to ensure legal data parallel execution still have to be used but their scope is constrained to the current subgroup scope or parent scope. For example, at the start of the execution of the task region in the program in Section 2.1, the many processors will immediately skip to the statement many_low = some_low since they are not required for the execution of the ON SUBGROUP some region. However, the statement many_low = some_low itself will not be executed until some processors also reach there, as is required for any legal execution that respects dependences. The actual synchronization mechanism is implementation dependent. It may be a barrier, a subset barrier, or based on message passing, and would typically be the same as that used for normal data parallel execution.

In the subgroup scope, the executing processor subgroup of an ON SUBGROUP block has read and write access to variables mapped onto that subgroup, and read-only access to replicated variables of its parent processor group. All unmapped scalar variables are automatically replicated by the compiler. All subgroup and parent variables are accessible in the parent scope of a task region.

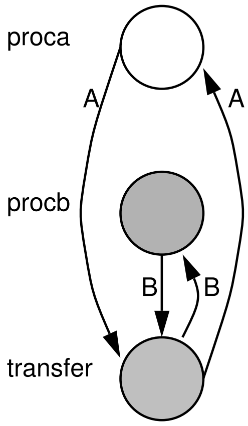
Computations only involving replicated scalar variables are automatically replicated on all executing processors, and are therefore performed asynchronously on all processors without synchronization or communication. If the compiler can determine that certain subgroups do not need to participate in the execution of a block of code in the parent scope, they can skip that code and continue to execute other parts of the program. An important special case is a simple array assignment statement where the compiler can trivially determine the participating and non-participating processor subgroups. This case is important for exploiting pipelined task parallelism and will be illustrated in later sections.

3 Usage

Task region and associated directives discussed in the previous section present a simple yet powerful model to write integrated task and data parallel programs. We illustrate how this model can be used to exploit task parallelism in four basic computation structures: parallel sections, data parallel pipeline, replicated data parallelism, and divide-and-conquer style nested parallelism. We omit declarations of arrays and their alignments and distributions for conciseness when they are not relevant to the discussion.

```
do i = 1,m
   CALL proca(A)
   CALL procb(B)
   CALL transfer(A,B)
enddo
```

(a)



(b)

```
TASK_PARTITION :: Agroup(nA), Bgroup(nB)
SUBGROUP(Agroup):: A
SUBGROUP(Bgroup):: B

BEGIN TASK_REGION
   do i = 1,m
      ON SUBGROUP Agroup
         CALL proca(A)
      END ON
      ON SUBGROUP Bgroup
         CALL procb(B)
      END ON
      CALL transfer(A,B)
   enddo
END TASK_REGION
```

(c)

Figure 1: Parallel section and join computation

3.1 Parallel Sections

A task region can be used to divide the processors into disjoint subgroups for performing independent computations, simulating what is often referred to as *parallel sections.* This form of task parallelism is relatively straightforward and useful in many application scenarios, an example being multiblock applications.

We present an example in Figure 1 where two procedures can execute in parallel on disjoint data sets but need to exchange data between invocations. This structure is common in multiblock applications where parts of the computation domain can be processed independently, but boundary elements have to be exchanged periodically. Figure 1(a) shows the high level code for this example and Figure 1(b) shows the data dependence between the procedures for one computation step. The procedures `proca` and `procb` execute one computation step in each invocation and their data sets are updated through procedure `transfer` between invocations.

An abbreviated task parallel program that expresses this computation is shown in Figure 1(c). The executing processors are divided into two subgroups `Agroup` and `Bgroup`, and variables `A` and `B` are mapped to these processor subgroups, respectively. When the execution of the task region starts, subroutines `proca` and `procb` start processing independently on processor subgroups `Agroup` and `Bgroup`, respectively. All processors combine to execute subroutine `transfer` after `proca` and `procb` complete execution. This sequence is repeated for `m` iterations.

3.2 Data Parallel Pipeline

Task regions can be used to implement pipelined data parallel computations where a stream of input data sets is processed by a series of functions. This is a common computation structure in signal processing, image processing and computer vision and it is often important to exploit task parallelism to achieve acceptable performance.

We use *FFT-Hist*, an image processing kernel, to illustrate how a data parallel pipeline is programmed with task regions. This program takes a stream of complex arrays as input. For each array, a two dimensional fast Fourier transform (FFT) is performed, followed by computation of histograms. The main loop of the program is shown in Figure 2(a) and the corresponding task graph is shown in Figure 2(b). For each iteration of the loop, the `cffts` function inputs the array `A` and performs 1D FFTs on the columns, the `rffts` function performs 1D FFTs on the rows, and the `hist` function computes histograms and outputs the results.

A high level task and data parallel implementation of this pipeline is shown in Figure 2(c). The `TASK_PARTITION` statement is used to divide the processors into three named subgroups, `G1`, `G2` and `G3` that are assigned `Pc`,`Pr` and `Ph` processors, respectively, corresponding to the 3 computation stages for column and row FFTs and histograms. A sequence of `SUBGROUP` statements is used to map the variables `A1`, `A2` and `A3` onto subgroups `G1`, `G2` and `G3`, respectively. Along with a `TASK_REGION` directive, `ON SUBGROUP` directives are used to map the computations, i.e., the three subroutines `cffts, rffts` and `hist`, onto the three processor subgroups. Data is exchanged between subgroups by assignments of the form `A2=A1` which are `not` in any `ON SUBGROUP` region, and therefore, can access all data.

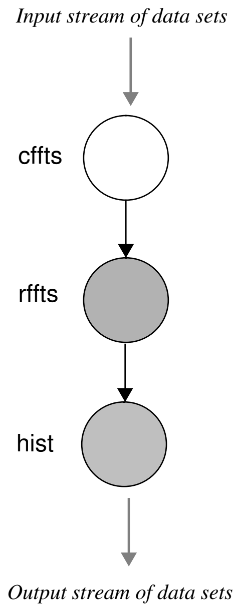
Once the pipeline is started up, the processors in each subgroup repeatedly perform computations and data exchanges, with each processor subgroup working on a different data set. To exploit maximal parallelism, it is important that assignment statements are executed only by necessary processors. e.g., processors in subgroups `G2` and `G3` should synchronize and participate in the execution of the assignment `A3=A2`, but processors in subgroup `P1` should skip past this assignment. Fortunately, this is a relatively simple optimization for a compiler.

3.3 Replicated data parallel computations

If a data parallel computation is repeated for a sequence of data sets, it is also possible to divide the processors into subgroups and assign the entire processing of each data set to one of the processor subgroups. We consider the example *FFT-Hist* computation in Figure 2 again. If each of the m data sets can be processed independently, then we can divide the processors into two subgroups and assign alternate data sets to each subgroup. The code for such an implementation is shown in Figure 3. The processors are divided into

```
do i = 1,m
   call cffts(A)
   call rffts(A)
   call hist(A)
enddo
```

(a)



(b)

```
TASK_PARTITION:: G1(Pc), G2(Pr), G3(Ph)
SUBGROUP(G1):: A1
SUBGROUP(G2):: A2
SUBGROUP(G3):: A3
BEGIN TASK_REGION
   do i = 1,m
      ON SUBGROUP G1
         cffts(A1)
      END ON
      A2 = A1
      ON SUBGROUP G2
         rffts(A2)
      END ON
      A3 = A2
      ON SUBGROUP G3
         hist(A3)
      END ON
   enddo
END TASK_REGION
```

(c)

Figure 2: FFT-Hist as a 3 stage data parallel pipeline

```
TASK_PARTITION:: G1(N/2), G2(N/2)
SUBGROUP(G1):: A1
SUBGROUP(G2):: A2
BEGIN TASK_REGION
   do i = 1,M/2
      ON SUBGROUP G1
         cffts(A1)
         rffts(A1)
         hist(A1)
      END ON
      ON SUBGROUP G2
         cffts(A2)
         rffts(A2)
         hist(A2)
      END ON
   enddo
END TASK_REGION
```

Figure 3: FFT-Hist as a replicated data parallel computation

equal size subgroups `G1` and `G2`, and each one performs the complete computation with calls to `cffts`, `rffts` and `hist` for half of the data sets.

Replication is a special case of *parallel sections* and is used to improve scalability of stream based computations, in a manner similar to data parallel pipelines. Replication can be used in combination with pipelining, and we will present the results of this combination for example applications.

3.4 Nested Parallelism

Task regions can be dynamically nested, i.e., a procedure call made from an `ON SUBGROUP` region in a task region can further subdivide the executing processors using another task region directive. This allows the exploitation of nested parallelism, one form of it being dynamic tree structured divide-and-conquer computations. As a simple concrete example, we show how *quicksort* can be implemented by recursively partitioning the input array of keys around a pivot and assigning proportionate number of processors to the two new arrays obtained as a result of partitioning.

Figure 4 shows quicksort implemented using task parallelism directives. Subroutine `qsort` picks a pivot key using the function `pick_pivot`, and determines the number of elements less than the pivot element using the function `count_less_than_pivot`. A suitable partitioning of processors to sort these two sets of elements is determined with the procedure `compute_subgroup_sizes`. Then the procedure `qsort_helper` is called with this information.

Subroutine `qsort_helper` declares a partition `qsortPart` of current processors with subgroups `greaterEqG` and `lessG`, and declares the appropriate size arrays `aGreaterEq` and `aLess` to be mapped onto these processor subgroups. In the corresponding task region, procedure `pick_less_than_pivot` is called to copy elements smaller than the pivot to the subgroup array `aLess`. Similarly, `pick_greater_equal_to_pivot` copies elements greater than or equal to the pivot to the array `aGreaterEq`. Then the subroutine `qsort` is invoked separately on these two arrays in the two `ON SUBGROUP` regions. Finally, the results are merged with subroutine `merge_result`.

When the program executes, the array to be sorted is recursively partitioned around pivots, and the processors are subdivided proportionately to perform sorting on the partitioned subarrays. In the end, the subgroups and the corresponding array sections are recursively merged to obtain the final result.

```
SUBROUTINE qsort(a, n)
    INTEGER n, a(n)
    INTEGER pivot, nLess, p1, p2

    if (n .eq. 1) return
    if (NUMBER_OF_PROCESSORS() .eq. 1) then
        call qsort_sequential(a, n)
        return
    endif
    pivot = pick_pivot(a, n)
    nLess = count_less_than_pivot(a, n)
    call compute_subgroup_sizes(n, nLess, p1, p2)
    call qsort_helper(a, n, nLess, n-nLess, p1, p2, pivot)
END

SUBROUTINE qsort_helper(a, n, nLess, nGreaterEq, p1, p2, pivot)
    INTEGER n, a(n), nLess, nGreaterEq, p1, p2, pivot
    TASK_PARTITION qsortPart:: lessG(p1), greaterEqG(p2)
    INTEGER aLess(nLess), aGreaterEq(nGreaterEq)
    SUBGROUP(lessG):: aLess
    SUBGROUP(greaterEqG):: aGreaterEq

    BEGIN TASK_REGION qsortPart
        CALL pick_less_than_pivot(aLess, nLess, a, n, pivot)
        CALL pick_greater_equal_to_pivot(aGreaterEq, nGreaterEq, a, n, pivot)
        ON SUBGROUP lessG
            call qsort(aLess, nLess)
        END ON SUBGROUP
        ON SUBGROUP greaterEqG
            call qsort(aGreaterEq, nGreaterEq)
        END ON SUBGROUP
        call merge_result(a, aLess, aGreaterEq)
    END TASK_REGION
END
```

Figure 4: Quick Sort using dynamically nested task parallelism

4 Implementation

The integrated task and data parallelism model presented in this paper has been implemented in the Fx compiler at Carnegie Mellon. The implementation of data parallelism in Fx is discussed in [18, 24]. An earlier version of task parallelism is discussed in [9] and related work in automatic mapping in [21, 22]. In this section, we outline the major issues in implementing our new task parallelism model in Fx.

Our model of task parallelism allows the programmer to assert that a computation can be efficiently and correctly performed by a subgroup of processors. The main challenge in implementing task parallelism is to develop a simple and efficient mechanism to perform computations independently on processor subgroups, thereby allowing the exploitation of data parallelism inside subgroups and task parallelism between subgroups. The specific design goals of the Fx implementation were as follows:

- Compilation of task parallelism should be orthogonal to compilation of data parallelism, i.e., the data parallel compiler should not be significantly affected by the addition of task parallelism.
- Computation and communication inside a subgroup should only use the processors assigned to the subgroup, i.e., the compiler should be able to exploit the assertion that a subgroup computation can be localized to the subgroup processors.
- Computation and communication outside of a subgroup (in parent scope) should only involve the necessary processors, i.e., the compiler and the runtime system should infer the set of processors required to execute a code block and allow the remaining processors to skip past that code block.

In the rest of this section we outline the techniques used to achieve these goals and discuss other relevant issues for implementing task parallelism.

Processor mappings

We use the concept of processor mappings to localize computation and communication to processor subgroups, and to insulate the data parallel compilation module from the implementation of task parallelism. A processor mapping maps virtual processors to physical processors. All data parallel compilation is done for a subgroup of virtual processors and virtual processors are mapped to physical processors during execution. The actual set of physical processors for a computation is identified only at runtime. For computations in the parent scope, the implementation conservatively identifies the participating processor subgroup, while in the subgroup scope, this information is explicitly provided by the corresponding `ON SUBGROUP` statement.

We outline the implementation of task parallelism using processor mappings. At startup, the processor mapping (which represents the group of all processors) is an identity mapping, i.e., every virtual processor ID corresponds to the physical processor ID. Each `TASK_PARTITION` template defines a new processor mapping for the executing processors. The `BEGIN TASK_REGION` directive activates the processor mappings of the corresponding task partition template. The `ON SUBGROUP` directive instantiates the processor mapping for the specified processor subgroup. In the instantiation process, each processor of the subgroup changes its virtual processor ID based on the new processor mapping. To support nested parallelism, each processor maintains a stack of virtual to physical processor mappings. This stack tracks all processor groups that a processor belongs to, with the top of the stack representing the current processor group. When a processor enters an `ON SUBGROUP` region, the processor mapping for the new subgroup that it belongs to is pushed onto this stack. When a processor leaves a task region, the corresponding processor mapping is popped off the stack.

This concept of processor mapping allows compilation, management of data distributions, computation of collective communication patterns, and barrier synchronizations to be all done in terms of a group of virtual processors. Hence implementation of data parallelism is done in terms of virtual processors and is not directly related to task parallelism. At runtime, computation and communication are automatically mapped to the relevant executing processors. A task partition specifies the number of processors to be assigned to each subgroup and the implementation is free to choose any such legal assignment. The Fx implementation attempts to choose a mapping that minimizes communication and synchronization overheads. The layer of indirection offered by processor mappings makes integration of this task parallelism model to an existing HPF compiler relatively easy.

Identification of minimal processor subsets

A computation in the parent scope of a task region can use data from any processor. However, it may only need data from a subset of processors. To exploit maximal task parallelism, it is important for an implementation to identify the set of processors required to execute a computation in the parent scope and allow the remaining processors to bypass the computation. Note that for computations inside the subgroup scope of a task region, the set of procesors that contain the relevant data is explicitly specified, and therefore no compiler analysis is required.

The example in Figure 2 illustrates the importance of recognizing the processors required to execute an array assignment. In Figure 2(c), the array assignment statement `A2=A1` only requires processors in subgroups `G1` and `G2` for execution. If the compiler correctly recognizes this, then the subgroup `G3` processors can continue past this statement and execute other code before the statement is executed. This is essential for full exploitation of task parallelism in this example.

In general, it is not possible to identify the processors that may have data required for a general computation like the execution of a procedure call. However, for simple statements like a full array assignment, the set of processors required to execute it can be determined easily by the compiler. In our experience, the ability to handle simple statements is sufficient to exploit maximal task parallelism in most cases.

Localization

Localization of computation, communication, and synchronization to a subgroup of processors is critical for exploiting task parallelism. The basic communication model in Fx is direct deposit of data by a sender to a receiver's memory space with the use of global barriers for synchronization [19]. To support task parallelism, the barriers are replaced by subset barriers on subgroups of virtual processors. At runtime, the physical processors are identified and only those processors participate in a barrier.

We should point out that the details of localization are strongly related to the underlying communication and synchronization framework. For example, if conventional message passing was used, the actual communication will implicitly involve only the necessary processors. However, such an implementation must ensure that empty messages are not exchanged with processors that are not participating in a communication step.

Replicated Computations

For effectively exploiting task parallelism in this model, the Fx compiler replicates all unmapped scalars and replicates all computations exclusively involving replicated variables. Replication of scalar computations is legal since it simply involves execution of the original sequential code on all processors, and preferable since it eliminates synchronization and communication between processors. A simple alternative is that one processor performs the computations and broadcasts the results to all processors. This approach is not recommended since it incurs extra synchronization and communication overhead and may prevent full exploitation of task parallelism due to unnecessary synchronization.

Replication is particularly important for loop induction and loop bound variables that are active inside task regions and we will illustrate this with the *FFT-Hist* example in Figure 2. The loop induction variable `i` in the program in Figure 2(c) must be replicated for exploiting task parallelism. If `i` is replicated, the loop control statements (check and increment for `i`) are executed on all processors subgroups independently, while the relevant processor subgroups participate in the execution of `ON SUBGROUP` and other code blocks inside the loop. However, if `i` was mapped to a single processor, then the value of `i` must be computed and broadcast at the beginning of every iteration by the processor that owns it. This leads to unnecessary synchronization that prevents pipelined task parallelism between loop iterations to be exploited.

Implication for I/O

In some parallel system implementations, I/O is performed through a single processor of the system. Task parallelism in the presence of I/O assumes that all processors can perform I/O independently and this paradigm has to be supported, although it is not necessary that each processor be able to physically perform all I/O operations independently. One simple solution is to have a single designated I/O processor that performs all I/O but is not considered an executing processor and hence does not have any execution related dependences.

SPMD or MIMD code generation

Another issue for the compiler is whether or not the same code image should execute on all processors. Since different processor groups may need different variables, a naive SPMD implementation is likely to be wasteful of memory since it must allocate all variables on all processors. The Fx compiler generates SPMD code and uses dynamic memory allocation to reduce the memory overhead at the cost of some added complexity. An alternative is to use different code images for different processor subsets. This will have better memory efficiency but add more implementation complexity.

5 Results

The model of task parallelism presented in this paper has been used to develop a variety of integrated task and data parallel programs using the Fx compiler. We now discuss the implementation and results from a set of programs. All performance results are presented for an Intel Paragon machine.

5.1 Sensor–based programs

We present results for the FFT-Hist program discussed in Section 3 and two other realistic task parallel programs: *narrowband tracking radar* and *multibaseline stereo*. The narrowband tracking radar benchmark was developed by researchers at MIT Lincoln Labs [17]. The processing of an input data set by the radar program consists of four steps: a *corner turn* to form a transposed matrix, independent row-FFTs, scaling, and thresholding. The multibaseline stereo uses an algorithm developed at Carnegie Mellon to measure depth in a scene accurately by using multiple cameras [15, 23]. Input consists of three (or more) images acquired from the cameras. The major computation steps are the following: finding a set of *difference images* by computing the sum of squared differences between corresponding pixels of match images, finding *error images* formed by computing the sum of a surrounding window of pixels for each pixel, and finding the *depth image* by computing the minimum element for each corresponding pixel for a set of images.

We will skip further details of these applications, which are described in [5] and corresponding results from an earlier Fx implementation of task parallelism are discussed in [20]. The structure of each of these three programs is a pipeline of data parallel tasks that processes streams of independent input data sets. The usage models for data parallel pipelines and replicated computations described in section 3.2 and section 3.3, respectively, can be used to build various task and data parallel implementations of these programs.

For processing of streams of input data sets, there are two performance criteria: 1) *Throughput*, which is the rate of processing data sets, and 2) *Latency*, which is the time taken to process an individual data set. Use of task parallelism in the form of pipelining or replication improves the throughput, but can increase the latency. Our programming model allows us to target the development of such applications to specific performance goals. Figure 5 shows that the best way to map the FFT-Hist example depends on the throughput requirement of the application. If there is no throughput requirement (i.e. minimizing latency is the only goal) then the simple data parallel mapping on the left is optimal, but with different throughput requirements, the best mapping changes. This model, along with the use of mapping algorithms presented in [21, 22], allows us to automatically determine the best mapping for a program for different performance goals. Table 1 shows the results obtained for this set of applications by setting a throughput goal and optimizing for minimum latency.

We observe from Table 1 that a factor of 3 improvement in throughput was achieved for the 256x256 FFT-Hist kernel with less than 20% increase in latency, but a 25% improvement in throughput for the 512x512 version of the program increased the latency by over 60%. These results illustrate that the benefits of task parallelism in this form vary widely and are higher for smaller data sets. A significant improvement in throughput was achieved for both the radar and the stereo program, but an increase in latency was observed only for the stereo program. The reason is that the improvement in throughput of the radar program was achieved by using processors that could not be used in the data parallel program because of the structure of parallelization. The main point is that integrated task and data parallelism offers a variety of ways in which such applications can be mapped, and facilitates the development of automatic tools to achieve different performance goals.

5.2 Airshed simulation

Airshed air quality simulation model developed by McRae and Russell [13] models the formation, reaction, and transport of atmospheric pollutants and related chemical species. The main data structure used in airshed simulation is a concentration matrix whose dimensions (with typical values) are number of atmospheric layers (5), number of grid points (500-5000) and number of chemical species (35). The simulation proceeds as follows. Every hour, a new set of initial conditions are input and a preprocessing phase is executed. This is followed by the main computation phase which consists of iteration over three simulation steps. The first step simulates particle transport, the second simulates chemical calculations, and the third also simulates particle transport. The number of iterations is determined at runtime depending on the hourly input and this phase is followed by the hourly output phase. High level structure of this application is captured by the following code:

```
DO i = 1,nhrs
   CALL inputhour(A)
   CALL pretrans(A)
   DO i = 1,nsteps
      CALL transport(A)
      CALL chemistry(A)
      CALL transport(A)
   ENDDO
   CALL outputhour(A)
ENDDO
```

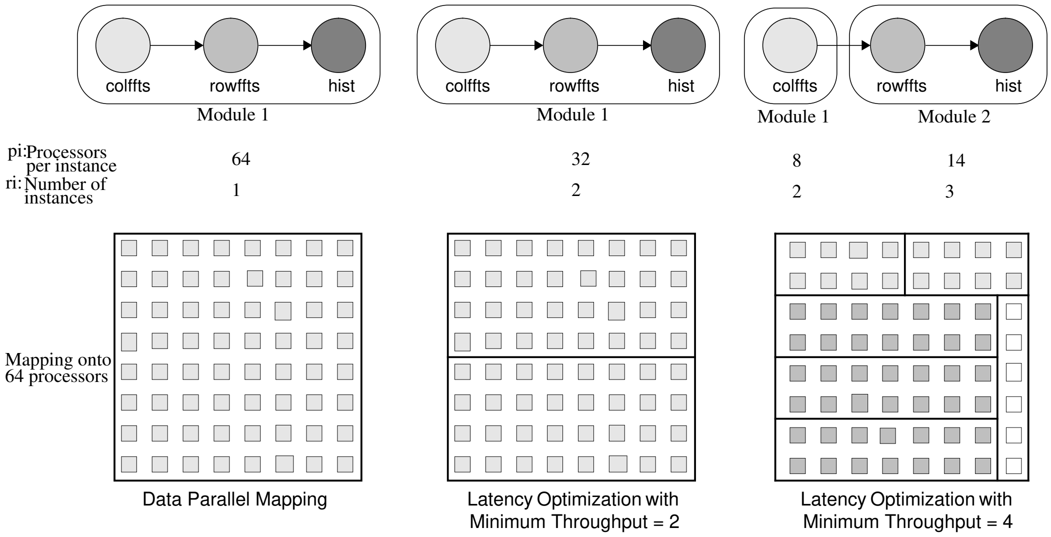


Figure 5: Mappings of a 512x512 FFT-Hist program on an Intel Paragon

Problem Description		Data Parallel		Best Task-Data Parallel		
Name of Program	Size of dataset	Throughput datasets/sec	Latency seconds	Throughput Constraint datasets/sec	Actual Throughput datasets/sec	Best Latency seconds
FFT-Hist	256x256	3.90	.256	8	13.3	.293
FFT-Hist	512x512	1.99	.502	2	2.48	.807
Radar	512x10x4	23.4	.043	50	70.2	.043
Stereo	256x240	3.64	.275	10	11.67	.514

Table 1: Performance results on 64 nodes of an Intel Paragon

A data parallel version of the airshed model has been developed with Fx and used over a period of time. One of the reasons this application did not scale well for larger number of processors was that the input and output processing phases are mainly sequential. These phases consume well under 2% of the total time in sequential execution, but become the main bottleneck once rest of the computation is speeded up by more than a factor of 10 by exploiting data parallelism. We used task parallelism to alleviate the effects of this performance problem. The input and output phases were separated into separate tasks and each one was placed on a separate processor subgroup. The input subgroup would preprocess the *next* input data set and transfer it when the main computation needed it. Similarly, the main computation would transfer raw output data to the output task and continue. The output task would process and physically output the data. The impact of using data and task parallelism in this fashion on the airshed application is plotted in Figure 6. The figure shows a significant improvement in scalability with the use of task parallelism. In particular, the execution time on 64 processors was reduced by around 25%.

5.3 Barnes-Hut

We present an abbreviated discussion of our ongoing implementation of Barnes-Hut algorithm, since it is an important application for integrated task and data parallelism. We expect to report on it in more detail in the future.

Figure 7 shows how a variation of Barnes-Hut algorithm can be implemented recursively with task parallelism. The function `build_bh_tree` builds a balanced binary tree of cells by evenly partitioning the particles along each axis, one at a time. This partitioning step is very similar to the partitioning in *quicksort* discussed in section 3, so further details are omitted.

The force computation is done by recursively dividing particles into subgroups, with each subgroup owning one half of the particles and a partial tree which covers the particles plus a copy of the top k level of the Barnes-Hut tree, where k is a fixed constant. The force vector for each particle is calculated at the lowest level of the recursion tree, where each subgroup has only one processor. If the force vector computation of a particle requires traversal of the missing (non-local) part of the Barnes-Hut tree, the particle is placed in a *worklist* of particles to be processed later by the parent subgroup. When the children complete, the parent subgroup will try to compute the force vector for the particles in the worklist based on its own more complete version of the tree, and pass a potentially reduced worklist to its parent.

The force computation for a particle can potentially propagate all the way to the root of the tree, but in general, the size of the worklist that is passed from a child to a parent will be small. In particular, for n particles uniformly distributed in a 3-dimensional cube, the total size of the work-

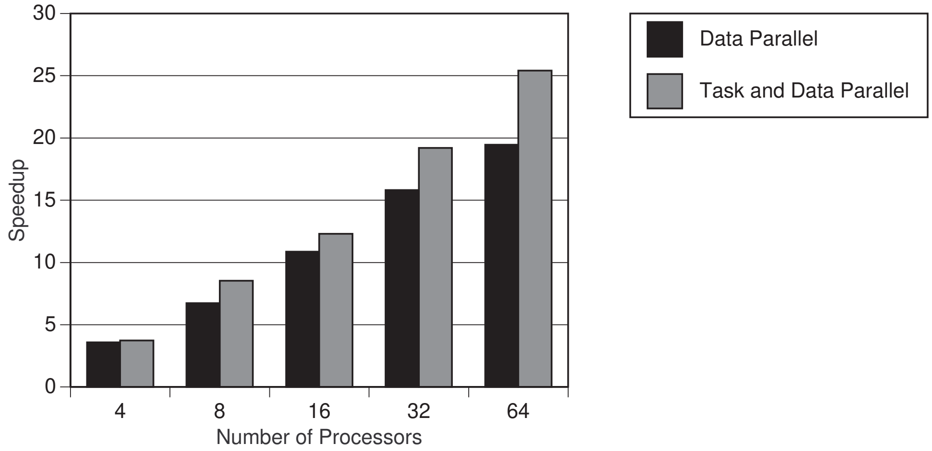


Figure 6: Speedup of Airshed application on an Intel Paragon

list is expected to be $O(n^{\frac{2}{3}})$. Furthermore, the size of the worklist can be reduced by controlling the number of replicated layers k. For p processors, k should be at least $\log(p)$ to avoid excessive communication. And since 2^k nodes will be replicated, k should be within a small constant multiple of $\log(p)$ to avoid excessive space overhead. The expected running time of this algorithm is $O(\frac{n}{p}\log(n))$ for a uniform distribution, and the memory usage is bounded by $O(n)$.

6 Related Work

In recent years, several ideas and proposals have addressed integration of task and data parallelism. We shall state just a few of them and discuss the relative merits of the model presented in this paper.

In Opus [3, 14], task parallelism is obtained by invocation of *task programs* (like subroutines) onto a specified set of processor resources. This is broadly similar to our approach but the task parallelism can only be exploited at subroutine call level. Opus also uses *shared data abstraction* to exchange data between executing tasks. While some form of sharing between tasks is indeed desirable, it makes the language and its implementation more complex and there is not enough evidence yet to justify the added complexity.

Earlier work on Fx [9] was based on exploiting task parallelism by specifying *task subroutines* and their input and output parameters, but leaving all the details of task management to the compiler. While this model is somewhat higher level than the model presented in this paper, it makes it more difficult to program complex forms of task parallelism (e.g., nested parallelism) as the user has less direct control over the resources.

Recently task parallelism has been added to High Performance Fortran as an approved extension. Both HPF task parallelism and our model are based on the basic idea of providing mechanisms for mapping data and computations onto processor subgroups[2]. One of the goals of this paper is to contribute to the development of compilers that support HPF task parallelism by illustrating the implementation and results from our task parallelism model.

[2]This is because of the strong interaction between the two design efforts.

The differences between HPF and Fx task parallelism partially emanate from the somewhat different design philosophies of a language standard and a research project. HPF has a general *ON* construct which can be used in different contexts including task regions. HPF task subgroups do not have to be declared explicitly, and execution on a subset of processors is specified by describing the processor subset in the relevant *ON* clause. In Fx, *ON* can be used only in the context of a task region, and the processor subgroup corresponding to an *ON* clause must be explicitly declared. HPF task parallelism requires minimal new syntax as no new declarative syntax is needed. It offers more flexibility as it allows the programmer to specify execution on a subset of processors that may have been computed during the execution of the current procedure, but it also offers less flexibility since only rectilinear subsets of the current processor arrangement can be processor subgroups.

A more detailed comparison of the two approaches is beyond the scope of this paper. As stated earlier, HPF requires a general implementation of *ON* constructs and it does not give the compiler writer some of the declarative information that we have used to help build a simple yet efficient implementation. However, we do believe that HPF task parallelism can also be implemented efficiently, at least for most common patterns of task parallelism. We expect commercial implementations of HPF task parallelism to become available in the future and shed more light on the HPF model and the significance of the differences between Fx and HPF task parallelism models.

An alternate approach to integrated task and data parallelism is use of a coordination language to communicate between tasks. An example of such an approach is presented in [7] where a subset of MPI is used to coordinate HPF tasks. This approach has the obvious and important advantage that the data parallel language does not have to be modified at all, but consequently, it does not offer a single clear programming model or semantic notion. For a task and data parallel program developed using this approach, there is no semantically equivalent sequential or data parallel program; while in our model, such a program is obtained simply by ignoring the task parallelism directives. Further, a coordination language does not allow compiler optimizations on the whole program, or allow dynamic load management by reassigning processors to different tasks within a program.

```
      SUBROUTINE bh(particles, nParticles)
          call build_bh_tree(particles, n, bhTree)
          call compute_force(particles, n, bhTree, nullWorkList)
          for each particle, update position based on force vector
      END
      SUBROUTINE build_bh_tree(particles, n, bhTree)
C         Build a binary tree by repeatedly partitioning the particles along one axis,
C         i.e., first partition along the x axis, then y, then z, and repeat.
C         The particles will be sorted based on the ordering of the leaves of the bhTree.
      END
      SUBROUTINE compute_force(particles, n, bhTree, workList)
          if (NUMBER_OF_PROCESSORS() .eq. 1) then
              for each particle p, call sequential_force_computation(p, bhTree, workList)
              return
          endif
          call compute_numbers_of_subgroup_processors(n, p1, p2)
          call compute_force_interaction_helper(particles, n, bhTree, p1, p2)
      END
      SUBROUTINE compute_force_helper(particles, n, bhTree, p1, p2, workList)
          TASK_PARTITION bhPart:: subTreeG1(p1), subTreeG2(p2)
          SUBGROUP(subTreeG1):: bhTree1, particles1, workList1
          SUBGROUP(subTreeG2):: bhTree2, particles2, workList2
          BEGIN TASK_REGION bhPart
              partition_bh_tree(bhTree, n, bhTree1, bhTree2)
              particles1 = particles(1:n/2)
              particles2 = particles(n/2+1:n)
              ON SUBGROUP bhTreeG1
                  call compute_force(particles1, n/2, bhTree1, workList1)
              END ON SUBGROUP
              ON SUBGROUP bhTreeG2
                  call compute_force(particles2, n-n/2, bhTree2, workList2)
              END ON SUBGROUP
              for each particle p in workList1 and workList2
                  call sequential_force_computation(p, bhTree, workList)
          END TASK_REGION
      END
      SUBROUTINE partition_bh_tree(bhTree, n, bhTree1, bhTree2)
C         Replicate top k levels of bhTree onto bhTree1 and bhTree2 (where k is a fixed constant
C         constant ≥ log(Total # of processors)).  Place the 1st half of the leaves
C         and all of their ancestors onto bhTree1 and similarly for the 2nd half.
C         Mark all branches leading to missing subtrees in bhTree1 and bhTree2 as remote.
      END
      SUBROUTINE sequential_force_computation(p, bhTree, workList)
C         Perform the sequential force computation with p on tree bhTree.  If p needs to
C         access a tree branch that is marked remote, then place p on workList and return.
      END
```

Figure 7: Barnes-Hut algorithm with dynamic nested task parallelism

7 Conclusions

We have presented a new model for writing integrated task and data parallel programs and show how it can be used to program different forms of task parallelism, including nested parallelism. This model is relatively simple and intuitive, and is designed as an extension to data parallel languages like High Performance Fortran. A related model is now an approved extension of HPF 2.0. Our model has been implemented in the Fx compiler. We argue that this model is relatively easy to implement and we have outlined some of the important implementation issues.

We demonstrate how this model of task parallelism is used to improve the performance of a variety of parallel programs. Data parallel pipelining and replication with task parallelism can be used to improve the performance and scalability of sensor based applications, and to tune such applications to specific performance goals. We present results from a FFT-Hist kernel, a narrowband tracking radar program, and a multibaseline stereo program, to illustrate the value of task parallelism for sensor based applications. Task parallelism was used to improve the performance of Airshed application by separating the input and output stages into independent tasks. Finally, we have demonstrated how this model of task parallelism is used to build tree structured applications like N-body problems.

Task parallelism can extend the power of a data parallel language like HPF and make it significantly more useful. We have demonstrated the advantages of integrated task and data parallelism and presented a simple and powerful way to achieve them.

8 Acknowledgements

We would like to thank the current and former members of the Fx compiler team at Carnegie Mellon, in particular Peter Dinda, Thomas Gross, Peter Lieu, David O'Hallaron, James Stichnoth, Tom Stricker and Gary Vondran, for their many contributions to the research presented in this paper. This work also benefited significantly from discussions in the control parallelism subgroup of the HPFF forum which was led by Dr. Rob Schreiber.

References

[1] Agrawal, G., Sussman, A., and Saltz, J. An integrated runtime and compile-time approach for parallelizing structured and block structured applications. *IEEE Transactions on Parallel and Distributed Systems 6*, 7 (July 1995), 747–754.

[2] Barnes, J., and P.Hut. A hierarchical O(N log N) force calculation algorithm. *Nature 4*, 324 (1986), 446–449.

[3] Chapman, B., Mehrotra, P., Van Rosendale, J., and Zima, H. A software architecture for multidisciplinary applications: Integrating task and data parallelism. Tech. Rep. 94-18, ICASE, NASA Langley Research Center, Hampton, VA, Mar. 1994.

[4] Chapman, B., Mehrotra, P., and Zima, H. Programming in Vienna Fortran. *Scientific Programming 1*, 1 (Aug. 1992), 31–50.

[5] Dinda, P., Gross, T., O'Hallaron, D., Segall, E., Stichnoth, J., Subhlok, J., Webb, J., and Yang, B. The CMU task parallel program suite. Tech. Rep. CMU-CS-94-131, School of Computer Science, Carnegie Mellon University, Mar. 1994.

[6] Foster, I., Avalani, B., Choudhary, A., and Xu, M. A compilation system that integrates High Performance Fortran and Fortran M. In *Proceeding of 1994 Scalable High Performance Computing Conference* (Knoxville, TN, October 1994), pp. 293–300.

[7] Foster, I., Kohr, D., Krishnaiyer, R., and Choudhary, A. Double standards: Bringing task parallelism to HPF via the Message Passing Interface. In *Supercomputing '96* (Pittsburgh, PA, November 1996).

[8] Gropp, W., Lusk, E., and Skjellum, A. *Using MPI: Portable parallel processing with the Message Passing Interface*. The MIT Press, Cambridge, MA, 1994.

[9] Gross, T., O'Hallaron, D., and Subhlok, J. Task parallelism in a High Performance Fortran framework. *IEEE Parallel & Distributed Technology*, 3 (1994), 16–26.

[10] High Performance Fortran Forum. *High Performance Fortran Language Specification, Draft Version 2.0*, Dec. 1996.

[11] Hiranandani, S., Kennedy, K., and Tseng, C. Compiling fortran D for MIMD distributed-memory machines. *Communications of the ACM 35*, 8 (August 1992), 66–80.

[12] Koelbel, C., Loveman, D., Steele, G., and Zosel, M. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.

[13] McRae, G., Russell, A., and Harley, R. *CIT Photochemical Airshed Model - Systems Manual*. Carnegie Mellon University, Pittsburgh, PA, and California Institute of Technology, Pasadena, CA, Feb. 1992.

[14] Mehrotra, P., and Haines, M. An overview of the Opus language and runtime system. Tech. Rep. 94-39, ICASE, NASA Langley Research Center, Hampton, VA, May 1994.

[15] Okutomi, M., and Kanade, T. A multiple-baseline stereo. *IEEE Transactions on Pattern Analysis and Machine Intelligence 15*, 4 (1993), 353–363.

[16] Ramaswamy, S., Sapatnekar, S., and Banerjee, P. A convex programming approach for exploiting data and functional parallelism. In *Proceedings of the 1994 International Conference on Parallel Processing* (St Charles, IL, August 1994), vol. 2, pp. 116–125.

[17] Shaw, G., Gabel, R., Martinez, D., Rocco, A., Pohlig, S., Gerber, A., Noonan, J., and Teitelbaum, K. Multiprocessors for radar signal processing. Tech. Rep. 961, MIT Lincoln Laboratory, Nov. 1992.

[18] Stichnoth, J., O'Hallaron, D., and Gross, T. Generating communication for array statements: Design, implementation, and evaluation. *Journal of Parallel and Distributed Computing 21*, 1 (1994), 150–159.

[19] Stricker, T., Stichnoth, J., O'Hallaron, D., Hinrichs, S., and Gross, T. Decoupling synchronization and data transfer in message passing systems of parallel computers. In *Proceedings of the 1995 International Conference on Supercomputing* (Barcelona, July 1995), ACM, pp. 1–10.

[20] Subhlok, J., O'Hallaron, D., Gross, T., Dinda, P., and Webb, J. Communication and memory requirements as the basis for mapping task and data parallel programs. In *Supercomputing '94* (Washington, DC, November 1994), pp. 330–339.

[21] Subhlok, J., and Vondran, G. Optimal mapping of sequences of data parallel tasks. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Santa Barbara, CA, July 1995), pp. 134–143.

[22] Subhlok, J., and Vondran, G. Optimal latency–throughput tradeoffs for data parallel pipelines. In *Eighth Annual ACM Symposium on Parallel Algorithms and Architectures* (Padua, Italy, June 1996), pp. 62–71.

[23] Webb, J. Latency and bandwidth consideration in parallel robotics image processing. In *Supercomputing '93* (Portland, OR, Nov. 1993), pp. 230–239.

[24] Yang, B., Webb, J., Stichnoth, J., O'Hallaron, D., and Gross, T. Do&merge: Integrating parallel loops and reductions. In *Sixth Annual Workshop on Languages and Compilers for Parallel Computing* (Portland, Oregon, Aug 1993).