Airshed Pollution Modeling in an HPF Style Environment

Jaspal Subhlok
Department of Computer Science
University of Houston
Houston, TX 77098
jaspal@cs.uh.edu

Peter Steenkiste
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
prs@cs.cmu.edu

Abstract

In this paper, we describe our experience with developing Airshed, a large pollution modeling application, in the Fx programming environment. We demonstrate that high level parallel programming languages like Fx and High Performance Fortran offer a simple and attractive model for developing portable and efficient parallel applications. Performance results are presented for the Airshed application executing on Intel Paragon and Cray T3D and T3E parallel computers. The results demonstrate that the application is "performance portable", i.e., it achieves good and consistent performance across different architectures, and that the performance can be explained and predicted using a simple model for the communication and computation phases in the program. We also show how task parallelism was used to alleviate I/O related bottlenecks, an important consideration in many applications. Finally, we demonstrate how external parallel modules developed using different parallelization methods can be integrated in a relatively simple and flexible way with modules developed in the Fx compiler framework. Overall, our experience demonstrates that a high level parallel programming environment based on a language like HPF is suitable for developing complex multidisciplinary applications.

Keywords: Parallel programming, parallelizing compilers, High Performance Fortran, task parallelism, pollution modeling, Airshed, module integration, multidisciplinary applications

1 Introduction

This paper reports on the development of *Airshed*, a large air pollution modeling application [17], in the Fx parallel programming framework [28]. Our goal is to demonstrate that high level parallel programming languages like Fx and High Performance Fortran [12, 14] offer an efficient and portable, yet relatively simple, model for developing parallel applications. We demonstrate that a variety of problems commonly associated with high level parallel programming, specifically poor and unpredictable performance and a restricted programming model, can in fact be dealt with effectively in this framework.

This paper has four main components, each representing an important feature that we would like to see in parallel programming environments. First, we show that the Airshed application exhibits good performance with low overheads across a variety of parallel computers, including the Cray T3D, the Cray T3E, and the Intel Paragon. The application ported easily across platforms and exhibits a predictable performance pattern across different platforms, different input data sets, and different numbers of nodes. Second, we show how task parallelism can be used to reduce the impact of some performance bottlenecks, and in particular, how it was used to alleviate I/O processing bottlenecks in Airshed. This points to the importance of providing both data and task parallelism in a single system. Third, the paper presents a simple performance model that captures application performance across different architectures, numbers of nodes, and input sizes. We show how the model can be used to explain the performance of the Airshed application. Finally, we present a framework for coordinating external parallel modules with the Fx environment, thereby opening up the programming model. This is an important consideration for interdisciplinary applications. The Fx Airshed air pollution modeling application was integrated with a parallel

A preliminary version of this work was presented at the 12th International Parallel Processing Symposium, under the title: "Airshed Pollution Modeling: A Case Study in Application Development in an HPF Environment".

population exposure model written in PVM using this framework without significant rewriting and without a significant impact on performance.

The remainder of this paper is organized as follows. We first describe the Airshed application and its parallelization in Fx in Section 2 and its performance in Section 3. Use of task parallelism in Airshed is described in Section 4. Section 5 describes a simple performance model for Fx generated programs and analyzes the performance of the Airshed application in the context of this model. Section 6 discusses the integration of Fx programs with foreign modules. We discuss related work in Section 7 and summarize in Section 8.

2 Parallel Airshed

We describe the Airshed application and Fx programming framework and then discuss the parallel Airshed implementation in Fx.

2.1 Airshed

The Airshed air pollution modeling application is an "Urban Regional Model" (URM) that models the formation, reaction, and transport of atmospheric pollutants and related chemical species. It is a multiscale grid version of the CIT Airshed model [17]. This model predicts the concentration of different chemicals in the atmosphere using their initial values, hourly input of sun and wind conditions, and release of additional chemicals. An important use of Airshed is to help in the development of environmental policies by simulating the impact of air pollution control measures.

The URM model is based on the atmospheric diffusion equation,

$$\frac{\partial c_i}{\partial t} + \nabla.(\mathbf{u}c_i) = \nabla.(\mathbf{K}\nabla c_i) + f_i + S_i$$

Here, c_i is the concentration of the ith pollutant among p species, i.e., $i = 1, ..., p$, $\mathbf{u}$ describes the velocity field, $\mathbf{K}$ is the diffusivity tensor, $f_i(c_1, ..., c_p)$ is the chemical reaction term and S_i is the net source term. The operator splitting method is used to solve Equation (1). The solution is advanced in time as

$$c^{n+1} = L_{xy}\,(\wedge t/2)\,L_{cz}\,(\wedge t)\,L_{xy}\,(\wedge t/2)\,c^n$$

L_{xy} is the two dimensional horizontal transport operator. L_{cz} is the chemistry and vertical transport operator; they are combined because they involve similar computations on similar timescales. The Streamline Upwind Petrov-Galerkin (SUPG) finite element method is used for the solution of horizontal transport [18]. For the chemistry and vertical transport equations, the hybrid scheme of Young and Boris [30] for stiff systems of ordinary differential equations is used. A detailed description of the model can be found in [15].

Airshed uses a multiscale grid instead of a uniform grid, since, to provide a given accuracy, a well-chosen multiscale grid is computationally significantly more efficient than a uniform grid, as it requires evaluation of the L_{cz} operator at fewer points. This is especially important for a URM since it covers areas with very different characteristics (e.g. city versus open space). It is, however, not clear how 1-dimensional transport operators could be used with multiscale grids, due to their non-uniform sampling and internal dependences, so Airshed uses a 2-dimensional horizontal operator instead of a 1-dimensional method. It turns out that in conditions where significant cross-flow components exist, as in the larger heterogeneous geographic regions that are targeted by Airshed, a 2-dimensional method can also use a larger time step than a 1-dimensional method to achieve the same accuracy, which also helps efficiency.

Figure 1 illustrates the Airshed computation. 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 iterates over a number of time steps determined at runtime based on the hourly inputs. In each iteration, the model simulates horizontal particle transport for half a time step, then simulates chemical reactions, and then again simulates particle transport for half a time step.

The main data structure used in the Airshed simulation is a 3-dimensional array representing the concentration of the species in the volume being modeled. The three dimensions are horizontal grid nodes (the points distributed over a 2D space are represented in a 1D array), vertical layers, and the chemical species. The typical values for these dimensions are 500-10000 points, 5-20 layers and 30-100 species. The particular data sets used in the experiments

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

Figure 1: High level code for the Airshed simulation

reported in this paper represent the Los Angeles basin (700 points, 5 layers, 35 species) and North Eastern United States (3328 points, 5 layers, 35 species).

2.2 Fx programming framework

We now outline the Fx parallel programming framework. Fx supports layout of array data in block, cyclic and block-cyclic distributions. Loop parallelism is expressed by a parallel loop construct that combines loop and reduction parallelism. Data parallelism in Fx is similar to that in High Performance Fortran [12] and will not be discussed further. Details of Fx data parallelism are available in [19, 25, 29].

Task parallelism is supported in Fx by the use of mechanisms to distribute data structures onto subgroups of processors and a mechanism to specify the execution of code on a subgroup of processors. We will state the directives used to achieve these purposes individually:

A `TASK_PARTITION` directive is used to partition the current group of processors into named subgroups, e.g.

```
TASK_PARTITION::  part1(1), part2(x), part3(N-x)
```

defines a partition of the current processors into subgroups `part1, part2` and `part3`, that are assigned `1,` `x` and `N-x` processors, respectively. Note that `x` and `N` can be procedure parameters and hence the actual processor assignment may be dynamic.

A `SUBGROUP` directive maps variables to a named processor subgroup, e.g.

```
SUBGROUP(part1)::  A1, B1
```

maps variables `A1` and `B1` onto the processors assigned to subgroup `part1`. The details of mapping, (e.g. `BLOCK, CYCLIC`) are determined by separate directives for alignment and distribution.

A `BEGIN TASK_REGION` and `END TASK_REGION` pair defines a task parallel part of a program, or a *task region*. A task region can contain blocks of code delimited by `ON SUBGROUP subgroupname` and `END ON` pairs, which is the code directed to execute on a named subgroup. Such code can only access variables mapped to that subgroup.

The following code:

```
BEGIN TASK_REGION
   ON SUBGROUP part1
      call task1(A1)
   END ON
   ON SUBGROUP part2
      call task2(A2)
   END ON
END TASK_REGION
```

directs that subroutine `task1` be executed on processors assigned to subgroup `part1` and subroutine `task2` be executed on processors assigned to subgroup `part2`.

This approach to task parallelism is sufficient to capture many application structures including parallel and pipelined data parallel tasks. The details of the model are presented in [28]. Task parallelism is also an extended feature of HPF-2 [12], and although the syntax of HPF task parallelism is significantly different from Fx, the basic ideas are similar and this discussion applies.

2.3 Parallel Fx implementation

The Airshed application was implemented primarily using Fx data parallelism. Use of task task parallelism to alleviate the I/O overhead is discussed later in this paper. We group the Airshed computation steps outlined in Figure 1 into three classes. We refer to the routines `inputhour`, `pretrans` and `outputhour` collectively as *I/O processing*. The two `transport` calls that simulate horizontal transport are referred to as *transport* computations, and the calls to `chemistry`, which combines aerosol computations, chemical reactions and vertical transport, as *chemistry* computations. We represent the concentration matrix by $A(species, layers,\ nodes)$, which is $A(35, 5, 700)$ for the Los Angeles data set, and $A(35, 5, 3328)$ for the North East data set. We can now characterize the main computation and communication phases in the data parallel implementation.

The *I/O processing* computations have limited parallelism and are handled sequentially in our implementation. The *chemistry* computation is independent for each grid point, and hence is parallelized along the "nodes" dimension; this means that it has a high degree of parallelism. The exception is the aerosol computation, which happens at the end of the chemistry phase. It cannot be parallelized and is therefore replicated. While the aerosol computation consumes a negligible portion of the total computation time, it has a significant impact, since it forces the redistribution of the concentration array. The transport computation is independent for each layer, and hence is parallelized along the "layers" dimension. In practice, this means that there is a high degree of parallelism in the chemistry computation, but only limited parallelism in the transport computation.

The natural data distribution for the concentration array A in different computation phases is as follows:

1. I/O processing and the aerosol computation: Replicated or A(*,*,*)
2. Transport phase: A(*,BLOCK,*)
3. Chemistry phase: A(*,*,BLOCK)

We will refer to these distributions as **D_Repl**, **D_Trans**, and **D_Chem**, respectively.

In the main loop we have the following sequence of computational steps requiring different data distributions:
Transport → Chemistry → Aerosol → Transport

This results in the following data redistribution steps in the main loop:

D_Repl → D_Trans: A(*,*,*) to A(*,BLOCK,*)

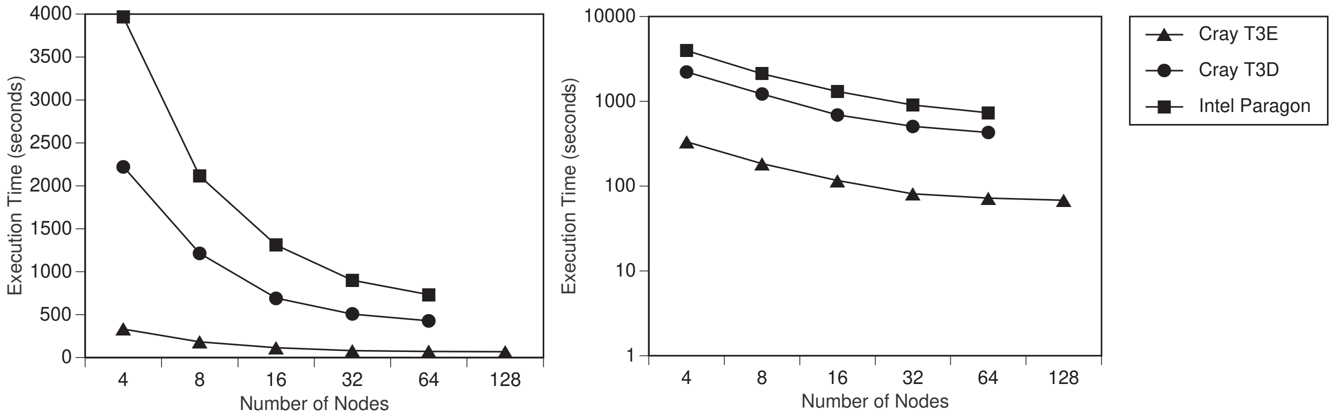
D_Trans → D_Chem: A(*,BLOCK,*) to A(*,*,BLOCK)

D_Chem → D_Repl: A(*,*,BLOCK) to A(*,*,*)

Note that a **D_Chem → D_Trans** redistribution is not required since the aerosol computation at the end of *chemistry* requires that the data be replicated, so the sequence of redistributions used to go from *chemistry* to *transport* is **D_Chem → D_Repl → D_Trans**.

3 Performance results

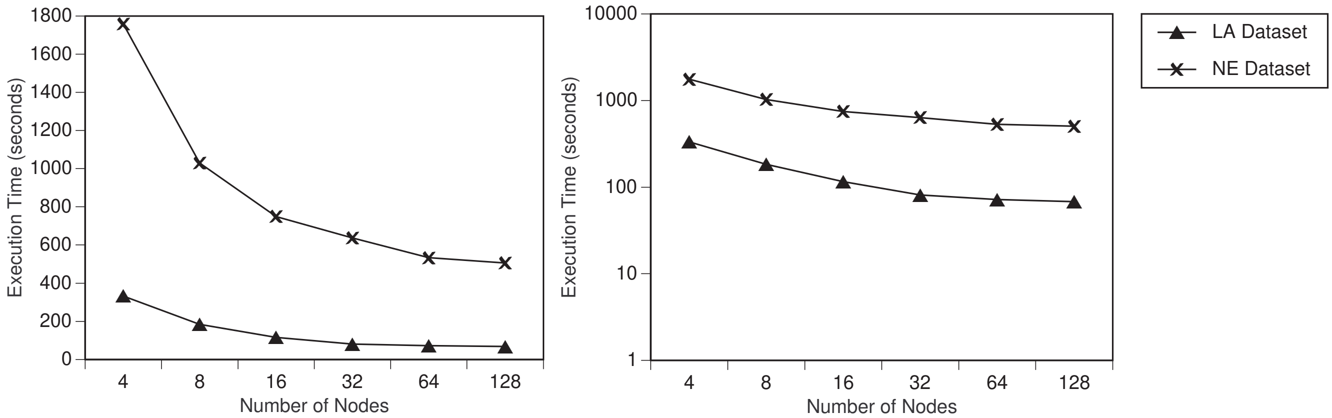
The execution times for the data parallel Airshed application executing on a Cray T3E, a Cray T3D and an Intel Paragon XP/S, using the Los Angeles basin data set, is presented in Figure 2. For reference, the execution time on a single processor 266 MHz DEC Alpha, a high-end workstation at the time the experiments were made, was around 2050 seconds. We observe significant speedups on all platforms, although the speedup is not linear. For example, on the Intel Paragon, increasing the number of nodes from 4 to 32, i.e. a factor of 8, results in a reduction of execution time from around 4000 seconds to around 900 seconds, a speedup of around 4.5. On the logarithmic scale, we observe that the curves representing the execution times are nearly parallel for the 3 machines, implying that the qualitative speedup behavior is the same for all machines. This illustrates that the application is *performance portable* in the sense that it not only executes on a variety of architectures, but it also achieves good performance on all of them, and follows similar performance patterns. The Cray T3D is just under a factor of 2 faster than the Intel Paragon, and the Cray T3E is approximately a factor of 10 faster than the Intel Paragon. These differences are fairly independent of the number of nodes used, and are not surprising considering the age and the processor and communication technologies used in these machines.



(a) time in linear scale *(b) time in log scale*

Figure 2: Execution times for the Airshed application with the LA data set

In Figure 3, we show the Airshed execution times for both the Los Angeles basin data set and the larger North East United States data set, executing on the Cray T3E. We observe that the qualitative execution behavior is similar for the two data sets. In particular, the logarithmic graph shows that they follow broadly similar speedup patterns.



(a) time in linear scale *(b) time in log scale*

Figure 3: Airshed execution times on the Cray T3E for the data sets representing Los Angeles basin and North East United States

To understand the performance better, we examine how the execution time of the different application components changes with the number of nodes. We select the Cray T3E and the Los Angeles basin data set as an example combination and show the time spent in different parts of the application in Figure 4. We observe that maximum amount of time is spent in *chemistry* computations, followed by *transport* computations and *I/O processing* computations. We also observe that these phases exhibit very different scaling behavior: the chemistry computation scales well to large numbers of processors, the execution time of the transport computation appears to scale only up to eight processors, and the time spent in I/O processing remains virtually constant. This result matches our earlier observation that there is a large degree of parallelism in the chemistry phase, the degree of parallelism in the transport phase is bounded by the number of *layers* (five in this data set), and I/O is essentially sequential. Most important, communication accounts for a very small fraction of the total time, implying that the compiler is generating efficient communication code, which is a major challenge in the implementation of languages like HPF.

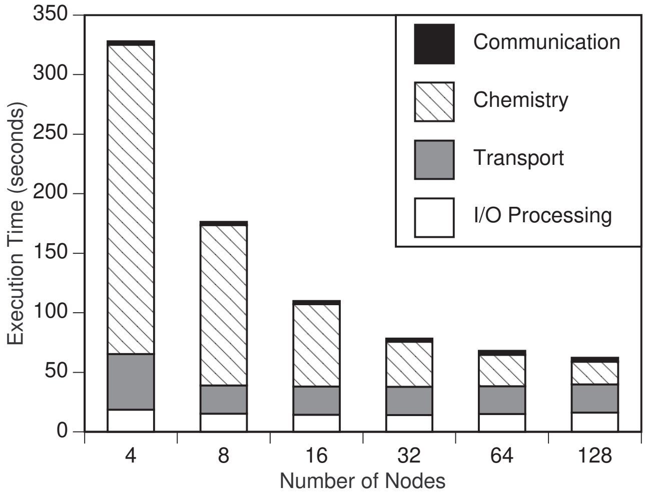


Figure 4: Scaling of the execution time of Airshed components on the Cray T3E for the LA data set

The limited scalability of the transport computations is related to the algorithmic choices made in the Airshed model. As we explained in Section 2, the use of a 2-dimensional operator over a multiscale grid significantly improves efficiency compared with 1-dimensional operators. Unfortunately, it has the drawback that it exhibits a much lower degree of parallelism than 1-dimensional uniform grid transport operators [16, 5]. With a 1-dimensional uniform grid transport, both the L_x and L_y operators can be parallelized over the layers and over one dimension of the grid, resulting in a relatively high degree of parallelism. The 2-dimensional L_{xy}, however, is difficult to parallelize, so the degree of parallelism is restricted to the number of layers. Overall, this means that models based on a uniform grid and 1-dimensional operators will offer better speedups [5], but because of their lower efficiency, they may not necessarily have better absolute performance. In fact, related research [23] appears to indicate that the improved parallelization does not make up for the reduced sequential performance. The relevance here is that the parallelization of the multiscale version of Airshed is an important and challenging problem.

4 Task parallelism for Airshed

Task parallelism is supported in Fx with mechanisms to distribute data structures onto subgroups of nodes and a mechanism to specify execution on a subgroup of nodes, as outlined in section 2.2. This allows independent sequential and data parallel routines to execute concurrently on disjoint groups of processors. Task parallelism makes it possible to reduce or eliminate the impact of application components with limited parallelism on the overall execution time by executing them concurrently with other computations. In related research, task parallelism has been used to minimize the impact of application modules that do not scale well because of high communication overheads [10, 28]. In the Airshed application, we have used task parallelism to alleviate the I/O bottlenecks. I/O is an important consideration in many applications and use of task parallelism for improving I/O behavior can be simpler and more effective than alternative techniques like explicit buffering in the program and the use of asynchronous I/O.

We observe from Figure 4 that one of the factors that limits the scalability of Airshed is the sequential *I/O processing* computations, which become a bottleneck for larger numbers of nodes. This trend was observed on all platforms. For example, on the Intel Paragon, *I/O processing* consumes well under 2% of the total time in sequential execution, but becomes a bottleneck consuming over 30% of the execution time on 64 nodes.

Given the dependencies between the input and output processing stages and the main computation loop, it is natural to use task parallelism to break up the computation in three pipelined stages, as shown in Figure 5. The input and output phases were separated into different tasks, and each placed on a separate node subgroup. When the main computation is performed on the *current* data set, the input subgroup reads and preprocesses the *next* input data set, while the output subgroup processes and writes the *previous* data set. The impact of using this form of task parallelism on the performance of Airshed on an Intel Paragon is plotted in Figure 6. The figure shows

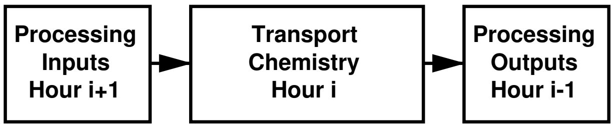


Figure 5: Pipelined task parallelism in Airshed

a significant improvement in scalability with the use of task parallelism. In particular, the execution time on 64 nodes was reduced by around 25%. This performance improvement due to task parallelism is analyzed in the next section.

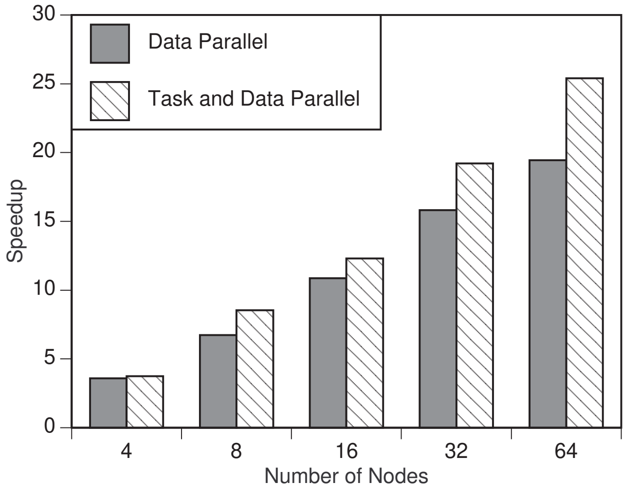


Figure 6: Speedup of Airshed on the Intel Paragon

5 Performance model

A common complaint about high level parallel programming systems such as Fx is that the observed application performance is often hard to explain because of the complexity of the compiler and the runtime system. We argue that when communication and other compiler overhead patterns are analyzed, predictable performance can be achieved at least for some regular applications. We present a simple performance model for programs generated by the Fx compiler and show that it predicts the performance of the Airshed application fairly accurately. We will first discuss the performance models for the computation and communication steps, since these are distinct phases in data parallel applications, and then apply the model to the Airshed application.

5.1 Computation performance

The execution time of a communication free data parallel code segment is determined by the total amount of computation to be performed, the rate of performing computations on a node, and the degree of useful parallelism. The total amount of computation is considered to be fixed for a given data set and the execution rate is determined by the type of the processor node. These two are jointly captured by the sequential execution time. The degree of useful parallelism is determined by the available parallelism and the number of nodes. Hence, the parallel compute time on a given architecture is simply the sequential execution time divided by the amount of useful parallelism.

Given a computation phase with sequential execution time T_{seq}, and effective parallelism of P_f, the parallel execution time T_x is simply as follows:

$$T_x = T_{seq}/P_f$$

When a computation is parallelized along an array dimension of size *dimsize* on P processors, the effective parallelism P_f will either be limited by the number of processor or the dimension of the array, so P_f can be estimated as:

$$P_f = min(P, dimsize)$$

The above equation is valid only if the elements of the distributed dimension are distributed equally across the processors. In general, we have to account for the fact that the assignment of elements to processors can be unequal, which is especially important for small values of $dimsize$, and the precise expression for P_f becomes:

$$P_f = dimsize/ceil(dimsize/P)$$

When task parallelism is also employed, pipeline parallelism can be exploited in addition to data parallelism to improve execution time. In this pipelined data parallel structure, each pipeline stage executes on a disjoint set of processors. The effective execution time of a set of k stages is determined by the *bottleneck* stage, and is simply the maximum parallel execution time among the stages as follows:

$$T_x = max(T_x^1, T_x^2, T_x^3, ...T_x^k)$$

In the above formulation, execution time also includes the time to send and receive data. Also, if the number of data sets that go through the pipeline is small, it is important to account for the time to start up and flush the pipeline. This can be achieved by increasing the number of times that the pipeline is clocked by the depth of the pipeline.

5.2 Communication performance

We now characterize the cost of the communication steps, which are the data redistribution steps in an Fx program. The communication costs do not cause significant performance degradation for Airshed, partly as a result of good communication code generated by Fx, and also because of the balanced computation and communication architectures of the machines used. However, in general, accurately modeling communication cost is very important since it often limits performance.

We first observe that, on today's high performance interconnection networks, communication performance is typically limited by the communication overhead on the end-points, and not by the aggregate bandwidth of the actual interconnect. As a result, the communication time is determined by the cost of assembling/disassembling messages, generating headers, and other overheads. The cost of sending a set of m messages and a total of b bytes can be modeled as:

$$tc = L * m + G * b$$

where L (latency component) is the effect of latency and message startup costs, and G (bandwidth component) is determined by the cost of operations that have to be performed on each byte, such as copying data and transferring messages between the memory and the actual interconnect.

A logical communication phase, which is a data redistribution in Fx, can also involve significant local copying of data, whose impact should not be ignored. Therefore, we extend the above equation to include c, the number of bytes that are locally copied but not communicated to another node, as follows:

$$tc = L * m + G * b + H * c$$

where H is determined by the cost of local copying. All the above parameters can be estimated for a specific system based on the communication system performance, the instruction execution rate, and the memory bandwidth.

Our model of the communication cost is based on the above equation and on the observation that the overall time of a communication phase is determined by the node that has the highest communication load. There are basically two steps involved in modeling the cost of a specific communication phase. First we have to identify the node that has the maximum communication time, which will typically be the node that has to send or receive the largest amount of data. Next, we have to determine how much data (b) and how many messages (m) that node has to send or receive, as well as the number of bytes that are locally copied on the node (c).

We now consider the aggregate communication cost associated with redistribution of multidimensional data arrays. For simplicity, we will consider only redistribution of full arrays and not array sections, and assume that the initial and final array mappings are either replicated or distributed along a single dimension. These assumptions are true for virtually all array redistribution steps in practice.

We first focus on the case where both the initial and final mapping are distributed, i.e., we consider a redistribution step for an array of size $arraysize$ on P processors, where the initial distribution has $dimsizeI$ elements in the distributed dimension and the desired final distribution has $dimsizeF$ elements in the distributed dimension. We use a communication model where all data to be exchanged between a specific pair of processors is aggregated, and therefore, at most one message is exchanged between a particular pair of processors. Hence the number of messages sent or received by a processor is bounded by P. However, the actual number of processors that own parts of an array is also bounded by $dimsize$, the number of elements in the distributed dimension. Therefore, the maximum number of messages sent (received) by a processor in a redistribution step is:

$$m_{max} = max(P, dimsize)$$

where $dimisize$ corresponds to $dimsizeF$ ($dimsizeI$) of the final (initial) distribution.

The other important parameter is the maximum number of bytes sent and received by a processor. The elements of an array may not be evenly divided among all the processors for two reasons. The $dimsize$ may be smaller than the number of processors leaving some processors unused, and it may not be evenly divisible by the number of processors. Taking these factors into account, the maximum number of bytes sent (received) by a processors is:

$$b_{max} = (arraysize/dimsize) * ceil(dimsize/P)$$

where $dimisize$ corresponds to $dimsizeF$ ($dimsizeI$) of the initial (final) distribution. Note that the above simplifies to $arraysize/P$ for the case where $dimsize$ is greater than P and is divisible by P.

Finally, we need to determine the number of bytes that are copied locally but not sent to other processors. This quantity is relatively small and can be ignored for normal redistributions. However, in the special case that the array is replicated in the initial distribution, the cost of the local copy comprises the entire cost of redistribution. The maximum number of bytes that are locally copied in this case is as follows:

$$c_{max} = (arraysize/dimsizeF) * ceil(dimsizeF/P)$$

Another special case is a replicated final distribution. In this case the sending processors must send their entire data to every receiving processor, so they have to send P messages, and the number of bytes sent is a factor of P higher than in the case where the final distribution is distributed. Every processor virtually receives all $arraysize$ elements.

The total communication cost is determined by the sum of the maximum cost of the three components discussed above, and can be stated as:

$$T_c = L * m_{max} + G * b_{max} + H * c_{max}$$

The communication costs associated with a general redistribution step, including the cases where one of the distributions is a replication, is summarized in Table 1.

Type of Redistribution	Communication Time: Send	Communication Time: Receive
Distributed TO Replicated	$L * P +$ $G * (P * (arraysize/dimsizeI) * ceil(dimsizeI/P))$	$L * min(P, dimsizeI) +$ $G * arraysize$
Distributed-I TO Distributed-F	$L * min(P, dimsizeF) +$ $G * (arraysize/dimsizeI) * ceil(dimsizeI/P)$	$L * min(P, dimsize1) +$ $G * (arraysize/dimsizeF) * ceil(dimsizeF/P)$
Replicated TO Distributed	$H * ((arraysize/dimsizeF) * ceil(dimsizeF/P))$	

Table 1: Table summarizing the cost of array redistribution. $arraysize$ is the total array size in bytes, $dimisze$ is the number of elements in the distributed dimension, and P is the number of processors. L is the latency parameter, G the bandwidth parameter, and H the local copy parameter for the parallel system.

5.3 Performance parameters

The models used in this discussion are, of course, very simple and not new. Our contribution is to show that a parallelizing compiler responsible for generating code for the computation and communication phases of an application can use this framework to generate a performance model for a parallel application. Given information on the features of the input data set, users or tools can then use this performance model to calculate the degree of useful parallelism of each computation phase, and the amount of data and the number of messages to be communicated by each node. Combined with the information about the characteristics of the target architecture, possibly obtained by executing the program sequentially and on small numbers of nodes, it is possible to estimate the execution time and scalability of an application.

To be concrete, we estimated the performance parameters for the Cray T3E for the communication generated by Fx using measurements for a small number of nodes and the results are as follows:

$$\begin{aligned} L &= 5.2 * 10^{-5} seconds/message \\ G &= 2.47 * 10^{-8} seconds/byte \\ H &= 2.04 * 10^{-8} seconds/byte \end{aligned}$$

These parameters give us a concrete performance model that we shall use to analyze the Airshed application.

5.4 Airshed performance analysis

We demonstrate that the performance of the Fx generated Airshed application broadly follows the execution model developed in the previous section.

5.4.1 Computation phases

The Airshed program has 3 main computation components, *chemistry*, *transport* and *I/O processing*. The predicted and measured execution times for these components for the LA data set on the T3E are plotted in Figure 7.

The *chemistry* computation scales almost linearly with the number of nodes, which is to be expected since it has a high degree of parallelism. The *transport* computation, however, has a limited degree of parallelism (five for the LA data set). It speeds up by a factor of two when doubling the number of nodes from four to eight, as the maximum workload on a node is halved. However, the execution time remains virtually unchanged as more nodes are added. The *I/O processing* time is essentially constant since this computation is sequential.

We note in Figure 7 that the execution model presented in the previous section predicts the computation time fairly accurately. This general pattern was also observed for the Paragon and the T3D machines, as well as for the North East data set.

5.4.2 Communication phases

We restate that the main communication steps in the Airshed application are as follows:

D_Repl → D_Trans: A(*,*,*) to A(*,BLOCK,*)

D_Trans → D_Chem: A(*,BLOCK,*) to A(*,*,BLOCK)

D_Chem → D_Repl: A(*,*,BLOCK) to A(*,*,*)

The measured and predicted times for these communication steps are charted in Figure 8.

The step $D\_Repl \rightarrow D\_Trans$ redistributes a replicated array to an array distributed in the "layers" dimension, which has only 5 elements for the LA input data set. This causes a local data copy but no actual transfer of data across nodes as the relevant data is locally available. For the array described by the dimensions $A(species, layers, nodes)$, it follows from Table 1 that the cost of this operation is:

$$C_t = H * ceil(layers/min(layers, P)) * species * nodes * W$$

where P is the number of the processing nodes and W is the machine wordsize in bytes. Note that the *ceil* operation is required since the node with the largest amount of data should be considered for cost computations when the data is divided unevenly. The above equation states that the cost of this step is proportional to the maximum amount of

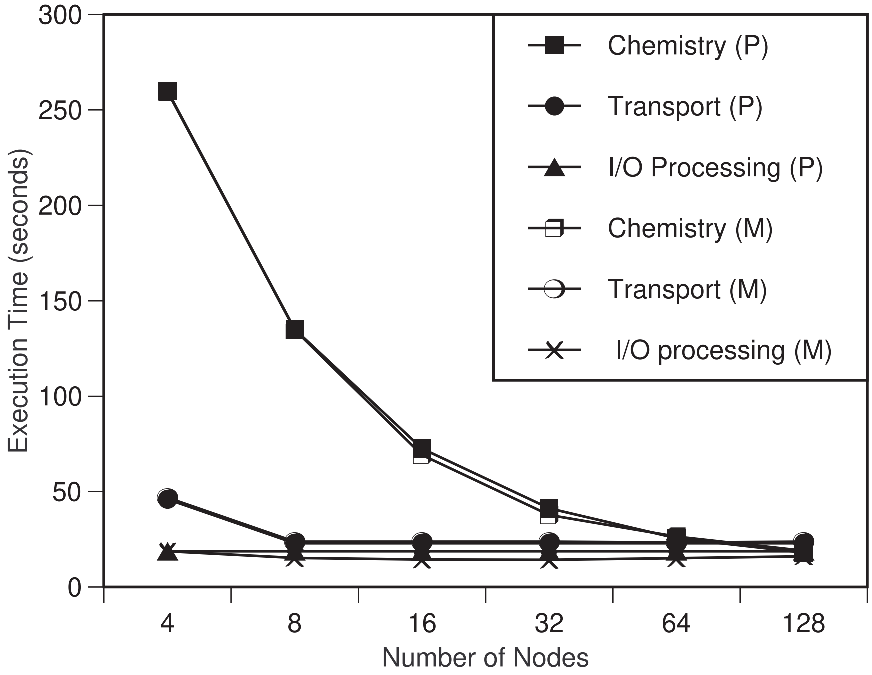


Figure 7: Predicted (P) and Measured (M) computation times for the LA data set on the T3E

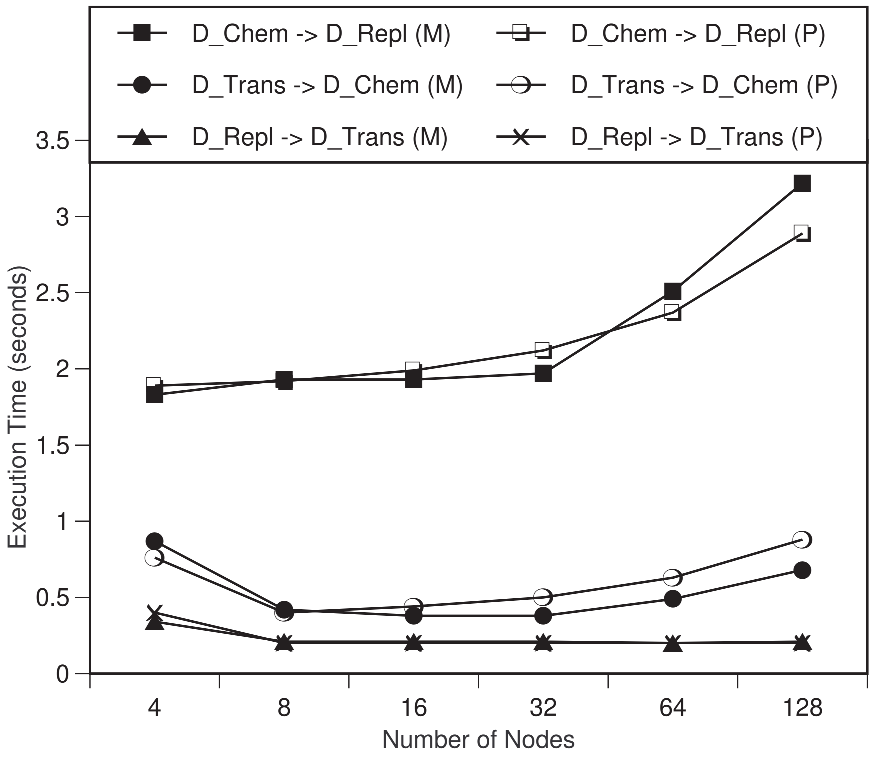


Figure 8: Predicted (P) and Measured (M) communication times for the LA data set on the T3E

local data on a node in the distribution D_Trans, which is determined by the maximum number of layers allocated to a node for the transport computation. For the LA input set, since the number of layers is 5, the amount of data is reduced from 2 layers to 1 layer when going from 4 to 8 nodes, but remains constant after that. This is the pattern predicted and observed in Figure 8, which is similar to the scaling of *transport* computation discussed earlier.

The redistribution step $D\_Trans \rightarrow D\_Chem$ copies an array distributed in the "layers" dimension with 5 elements to the "nodes" dimension with 700 elements. The cost of this step is dominated by the sending cost, as the maximum amount of data sent by a node exceeds the amount of data received by any node, since there are fewer senders than receivers. The number of messages that a sender has to send equals the total number of nodes. The cost of this operation, based on Table 1 is:

$$C_t = L * P + G * ceil(layers/min(layers, P)) * species * nodes * W$$

For the LA input set, as the number of nodes is increased from 4 to 8, the maximum amount of data sent by a node decreases from 2 layers to 1 layer, but then remains the same after that. The change in the amount of data sent is responsible for the large drop in cost from 4 to 8 nodes in Figure 8. For larger numbers of nodes, the amount of data sent stays the same, but a larger number of (smaller) messages are sent. This increased latency component is responsible for the gradual increase in cost beyond 8 nodes. We again observe that the predicted and measured communication times are fairly close to each other.

In the redistribution step $D\_Chem \rightarrow D\_Repl$, each node has to receive the entire data array, so the cost is higher than the previous two steps and is dominated by the receiving cost. The number of messages sent by a node, and the number of messages received by a node, are both bounded by the number of nodes. The cost equation is:

$$C_t = 2L * P + G * layers * species * nodes * W$$

As the number of nodes is increased, the volume of data to be received remains unchanged but the number of messages increases. We observe in the graph that the communication cost is relatively high to start with and gradually increases because of the increased latency component of the cost. This pattern is seen in measured as well as predicted times for this communication step.

5.4.3 Complete application

Finally we estimate the performance of the complete Airshed application and compare it with actual measurements in Figure 9. We first observe that the predicted time for data parallel Airshed is quite accurate. This is not surprising since the total execution time is simply the sum of all the computation and communication times, and we illustrated earlier that they can be predicted fairly accurately.

Figure 9 also compares the predicted and measured performance of Airshed with integrated task and data parallelism. The performance of this version of Airshed was estimated based on the execution model for pipelined data parallel computations described in the beginning of this section. The total execution time is dominated by the main computation phase that runs on nearly all processors since only two processors are assigned to *I/O processing*, one each to the input and output phases. We again notice that the estimated and measured times are fairly close to each other.

In summary, these results demonstrate that the execution behavior of the Fx generated Airshed application follows a predictable pattern. Small differences between the two sets of values do exist, which is not surprising given the simple nature of the estimates. While we have demonstrated performance predictability for only one application, we believe that this process can be automated and extended to other applications. In particular, the results for the communication model are expected to be fairly application independent since they apply to communication patterns generated by the Fx compiler. While there are a number of factors that can make the estimates inaccurate (e.g. cache usage patterns, communication schedule conflicts, etc.), the results for Airshed suggest that a rough estimate of the execution time of an application can be obtained. This can be used to explain the basic patterns and trends in the observed performance to users. Alternatively, the measurements obtained by executing an application on a small number of nodes can be used to extrapolate the performance to larger numbers of nodes. This is an interesting and important case since small parallel computers are fairly widely available as development platforms, while large ones are the domain of a select set of institutions like supercomputing centers. This model-based approach will be much less effective for less regular applications, or for systems that manage parallelism at runtime.

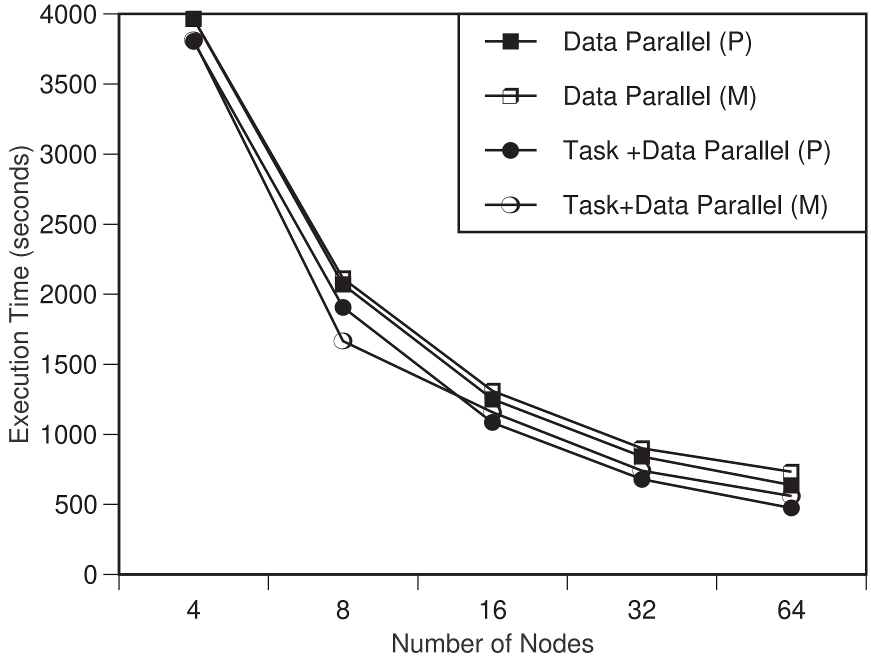


Figure 9: Predicted (P) and Measured (M) execution times of the complete data parallel Airshed application and the integrated task and data parallel version

6 Foreign module interface

We present our approach to developing integrated applications that include Fx and other *foreign* modules and show how it is employed for the Airshed application. The goal is to ease the development of applications that require concurrent execution of Fx and foreign modules, possibly on disjoint sets of processors.

6.1 Motivation

Many interdisciplinary applications require that several existing programs be combined into a larger application. Airshed is often coupled with a population exposure model (PopExp), a computation that uses the concentration data for chemicals generated by Airshed to calculate the impact on health. Population exposure calculations can be very expensive and are often also parallelized. Environmental scientists would like to use an efficient integrated version of these two programs, through a problem solving environment like GEMS [22], as illustrated in Figure 10. However, different programs are often parallelized in different frameworks. For example, in this case, Fx was used for Airshed and PVM for PopExp. We believe that integrating parallel programs with external parallel modules is a fundamental problem in parallel computing, and we will argue that the use of parallelizing compilers can actually help in the solution process.

Integration of existing programs is a challenge for all parallel programming environments. Solutions to accomplish this integration range from ad-hoc ones like running multiple programs independently and using file sharing for communication, to expensive ones like merging independent programs into a single program. Both approaches have obvious disadvantages: inefficient execution or a high cost in program development. Some of the more attractive approaches include the use of coordination languages [1, 2, 9], which are designed to build programs by combining existing modules. However, coordination languages have not been widely used, probably in part because familiarity with a new language is required. Using Airshed+PopExp as an example, we will present an approach that allows different parallel modules to execute in parallel with limited effort by the programmer. A comparison with other methods used in the HPF domain is included in the discussion of related work.

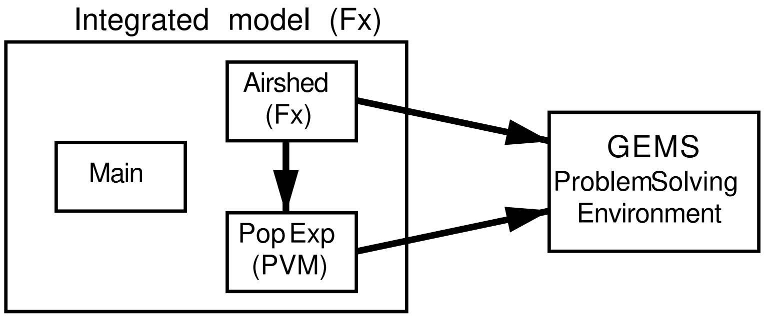


Figure 10: Combined Airshed and Population Exposure modules

6.2 Approach

The idea is to use task parallelism support in a parallelizing compiler and its runtime system to facilitate the integration of modules. The integration is based on a *native program* that will be compiled by the parallelizing compiler and that invokes the different modules. Modules can be either native modules, i.e., modules written in the language of the compiler, or external *foreign modules*, i.e., independent executables that may have been developed in a different language and parallelism model. In our approach, foreign modules are represented as tasks in the native program. These tasks, which we will call representative tasks since they represents the foreign modules in the native program, have characteristics that are similar to those of other native tasks. For example, a representative task is associated with a node subgroup that represents the nodes where the foreign module executes. Similarly, data is exchanged between the native program and the representative task (of a foreign module) through shared variables that are mapped onto that subgroup. Our experimentation is with the Fx compiler system, hence the native programs in this discussion are Fx programs.

The central idea of our approach to foreign modules is that the the communication library used by the Fx compiler is exposed to the foreign modules as the *external interface library*. Hence the Fx program interacts with the foreign module as it does with other Fx tasks, while the foreign module interacts through this external interface library which is implemented with the internal Fx communication library. Note that the foreign module may use a different library for its internal communication. This approach has the following key features:

1. Modules written in different parallelism models can be integrated in a single framework, without having to rewrite the foreign code. The main change in the foreign module is the insertion of calls to an external interface library to input and output data from the native Fx program.
2. The programming model of the native Fx module is not changed. A representative Fx task is assigned a set of nodes and the communication is achieved by reading and writing variables mapped to that task, which is identical to interaction with other Fx tasks.
3. The Fx compiler maintains a global view of the application. With the knowledge of computation and communication characteristics of a foreign module, the techniques used in Fx to manage processor allocation among tasks [26, 27] can be extended to foreign modules.

6.3 Design tradeoffs

We briefly look at two critical design options in integrating foreign modules. A more complete discussion can be found in [24].

Support for communication between a foreign module and the native program presents a tradeoff between the complexity of implementation on one hand, and programming flexibility and performance on the other. Figure 11 outlines some of the options. In the simple case represented by the scenario **A**, the data is transferred from the native program to the representative task, then to a designated *interface node* in the foreign module, which in turn distributes it to the other nodes of the foreign module. This is the easiest model to implement but may be inefficient because of extra data copies. Scenario **B** is an important optimization: data is transferred directly to all the nodes of the foreign module. This requires that the topology of the foreign module and the data distribution inside it be exposed to the native compiler for communication generation. Finally, the most complex and potentially

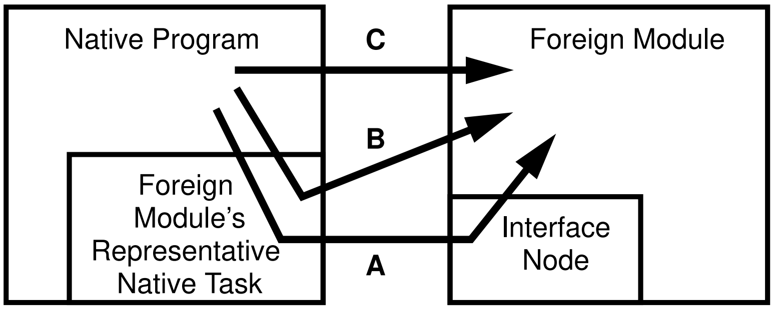


Figure 11: Optimization of communication with a foreign module

most efficient implementation is represented by the scenario **C**, where the data is directly transferred from the variables in the native program to the variables in the foreign module. Some of the challenges of such an efficient implementation are discussed in [6].

Implementations of foreign module support can also differ in the generality of the interactions between the native and foreign modules. In the simplest case, the foreign module executes when it is invoked and goes to sleep after it returns its results to the main module; it is idle and stateless in between invocations. However, in general, the interaction can be asynchronous, and the foreign module can continue execution in parallel with the Fx program. This may be useful, for example, if the foreign module involves I/O. Such an asynchronous model is much more powerful but it causes additional complexity.

6.4 Example

We added a population exposure program called PopExp as a foreign module to Airshed. PopExp started out as a stand-alone parallel program based on a master-slave paradigm using PVM for communication. In the original program, pollution data was read from a file. These read calls were replaced by calls to an external interface library exported by Fx that receives the data on the fly from the Airshed program. Note that the external interface library is application independent, i.e., a single external interface library can be used by foreign modules in different parallel languages. The interface library also includes the necessary calls for initialization and synchronization with the native program.

Our prototype implementation of foreign modules is based on the simplest approach **A** of Figure 11: the master collects all the data from Airshed and then distributes it to its slaves using PVM communication statements. The structure of the combined Airshed–PopExp processing is illustrated in Figure 12. In the integrated implementation, PopExp is effectively treated like a task in the same way as I/O stages were tasks in the task parallel implementation of Airshed discussed earlier.

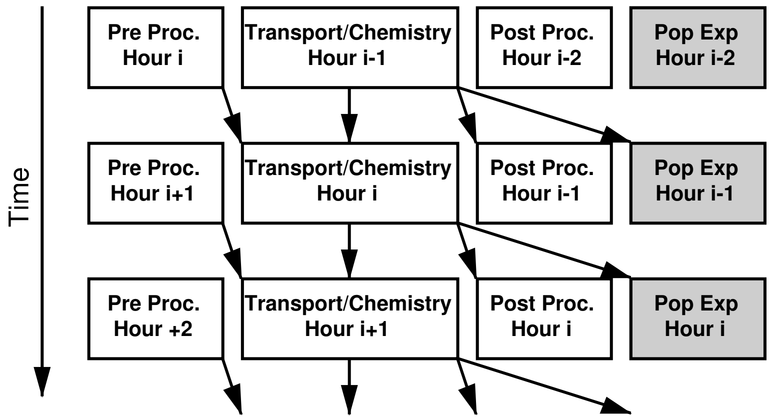


Figure 12: The structure of the Airshed–PopExp computation

6.5 Evaluation

To validate the performance of our approach to integration, we developed an all Fx version of the Airshed–PopExp application, in addition to a version in which Airshed is programmed in Fx and PopExp is a PVM foreign module. (We verified that the Fx and PVM versions of PopExp had the same performance behavior when executing independently.) The performance of both implementations is shown in Figure 13. We observe that there is a fixed, relatively small, overhead associated with the foreign module approach. This overhead is the result of the extra sequential data transfer required through the PopExp master; in the Fx-only program, there is no PopExp master and data is transferred directly to the PopExp data-parallel slaves. This overhead does not significantly impact overall performance, and as we discussed earlier, it can be reduced or eliminated by a more aggressive implementation. Given the advantages in code reuse, this preliminary evaluation suggests that the foreign module approach is attractive.

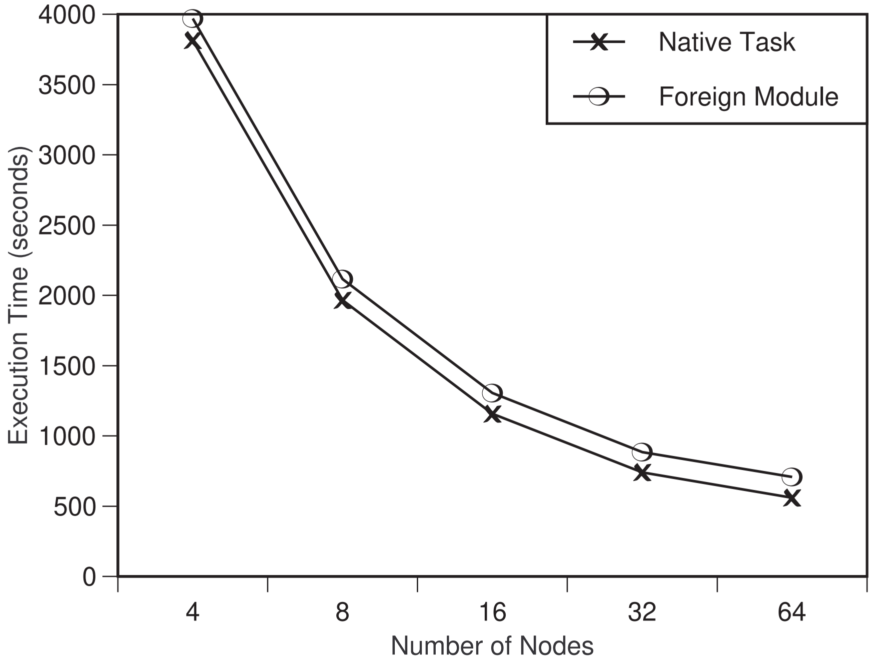


Figure 13: Performance comparison with PopExp as native and foreign module on the Intel Paragon

In general, the amount of computation in the population exposure module varies with the scenario in which it is used, as well as the input data set. To simulate different usages of this module, we developed versions of PopExp with different computational costs, and then experimented with assigning different numbers of processors to the PopExp and Airshed modules. Figure 14 shows the execution time as a function of the number of nodes assigned to the foreign module. Different curves are shown for different versions of the foreign module; the labels 2x, 4x, etc. represent the amount of computation, relative to the base PopExp computation. As expected, each curve goes through a minimum, corresponding to a balanced node assignment. If too few nodes are assigned to the PopExp calculation, it can become the bottleneck. If too many nodes are assigned to the PopExp calculation, the main Airshed computation becomes the bottleneck. In the latter scenario, all curves merge since the amount of computation in PopExp no longer affects the execution time. We observe that the optimal node allocation depends on the computational cost of PopExp. The point is that the appropriate load balancing between the native and foreign module can be supported in this framework.

7 Related work

Several research efforts have addressed compilation of data parallel languages, two of the pioneering ones being Fortran D [13] and Vienna Fortran [4]. Data parallelism in Fx, the framework used in this research, is discussed

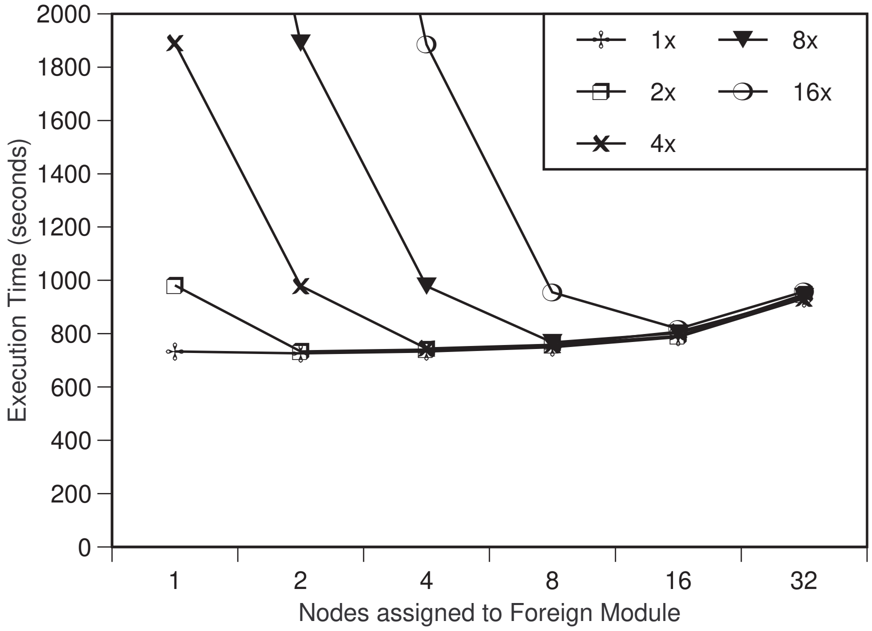


Figure 14: Performance with different processor assignments to the foreign module with different computation loads

in [25, 29]. With the standardization of High Performance Fortran [12], several commercial compilers for the language are also available. Support for task parallelism in Fx is discussed in [28], and the subject has also been addressed by several other research efforts, including [3, 7, 11].

The Airshed model for pollution modeling was developed by McRae et. al. [17], and a parallel implementation, based on PVM, was developed by Kumar et. al. [16]. A parallel implementation of the original uniform grid CIT model is described by Dabdub et. al. [5]. To our knowledge, this is the first implementation of Airshed in a high level parallel programming framework, and that exploits both data and task parallelism.

Task parallelism has been used to alleviate various kinds of performance bottlenecks [10] and some research efforts have addressed the automatic use of mixed parallelism [20, 26, 27]. However, we believe that the use of task parallelism to alleviate I/O bottlenecks is a novel idea that is being investigated in the Fx project.

Several coordination languages have been developed for application development [1, 2, 9] but they are primarily targeted to communication between sequential processes. High Performance Fortran [14] allows control to be transferred to external routines in other languages using EXTRINSIC procedure calls. Our approach blends such a mechanism with task parallelism and uses a common communication interface between a native Fx program and a foreign module. The result is that a foreign module can potentially execute concurrently with the main computation, offering significantly more flexibility. Foster et. al. [8] use an MPI binding to HPF to enable multiple HPF executables to communicate using MPI collective communication operations. This approach is somewhat similar to our approach, with the difference that our system is more closely integrated with the parallelizing compiler. In particular, the interface from the Fx/HPF compiler is not an explicit call to a collective communication library but a subroutine call in a task region, which is an existing HPF concept. A similar idea of using runtime communication libraries to integrate existing parallel code modules is explored in [21].

8 Conclusions

This paper reports on the development of the Airshed pollution modeling application in the Fx programming environment. The paper has demonstrated that large parallel applications can be successfully developed in a high level parallel programming environment based on High Performance Fortran. We specifically address problems associated with HPF programming relating to performance and limitations of the programming model, and show that they can be resolved in this environment.

Results are presented for the Airshed application executing on the Intel Paragon and the Cray T3D and T3E parallel computers. The data sets used represent Los Angeles basin and North Eastern United States. The results demonstrate that the performance obtained is good and that the performance tradeoffs are a result of the choice of the algorithm and not due to overheads associated with the use of a high level language. Further, we showed how Airshed performance can be explained using a simple model of computation and communication times, suggesting that for some applications, our Fx based framework offers predictable performance. We also show how task parallelism was used to alleviate I/O related bottlenecks, an important consideration in many applications. Finally, we demonstrate how external modules can be integrated in a relatively simple and flexible way in the Fx compiler framework.

In summary, we have developed an efficient implementation of the Airshed model and provided evidence that an HPF based environment is suitable for developing such applications. We believe that the experience reported is representative of development of multidisciplinary applications.

9 Acknowledgments

Peter Lieu performed most of the experiments in this research. Pittsburgh Supercomputing Center supported this work in the form of grants to use their supercomputing resources. We would like to acknowledge the contributions of Ted Russell and Ed Segall to the Airshed implementation. Thomas Gross, David O'Hallaron, James Stichnoth, Tom Stricker and Bwolen Yang were the key developers of the Fx compiler system. This research was sponsored in part by the Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0287.

References

[1] BEGUELIN, A., DONGARRA, J. J., GEIST, G. A., MANCHEK, R., AND SUNDERAM, V. S. Graphical development tools for network-based concurrent supercomputing. In *Proceedings of Supercomputing '91* (Albequerque, November 1991), IEEE, pp. 435–444.

[2] CARRIERO, N., AND GELERNTER, D. Applications experience with Linda. In *Proceedings of the ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages and Systems* (New Haven, CT, July 1988), pp. 173–187.

[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] DABDUB, D., AND SEINFELD, J. Air quality modeling on massively parallel computers. *Atmospheric Environment 28*, 9 (1994), 1679–1687.

[6] DINDA, P., O'HALLARON, D., SUBHLOK, J., WEBB, J., AND YANG, B. Language and runtime support for network parallel computing. In *Eighth Workshop on Languages and Compilers for Parallel Computing* (Columbus, Ohio, Aug 1995). Proceedings published as Springer Verlag Lecture Notes in Computer Science, No 1033, pages 534–550.

[7] 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.

[8] FOSTER, I., KOHR, D., KRISHNAIYER, R., AND CHOUDHARY, A. Double standards: Bringing task parallelism to HPF via the Message Passing Interface. In *Proceedings of Supercomputing '96* (Pittsburgh, PA, November 1996).

[9] FOSTER, I., AND TAYLOR, S. *Strand: New Concept in Parallel Programming*. Prentice Hall, Englewood Cliff, New Jersey, 1990.

[10] GROSS, T., O'HALLARON, D., AND SUBHLOK, J. Task parallelism in a High Performance Fortran framework. *IEEE Parallel & Distributed Technology 2*, 3 (Fall 1994), 16–26.

[11] HAINES, M., HES, B., MEHROTRA, P., AND VAN ROSENDALE, J. Runtime support for data parallel tasks. In *Fifth Symposium on the Frontiers of Massively Parallel Computation* (1995), pp. 432–439.

[12] HIGH PERFORMANCE FORTRAN FORUM. *High Performance Fortran Language Specification, Version 2.0*, Dec. 1996.

[13] 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.

[14] KOELBEL, C., LOVEMAN, D., STEELE, G., AND ZOSEL, M. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.

[15] KUMAR, N., ODMAN, M. T., AND RUSSELL, A. G. Multiscale air quality modeling: Application to southern california. *Journal of Geophysical Research 99* (1994), 5385–5397.

[16] KUMAR, N., RUSSELL, A., SEGALL, E., AND STEENKISTE, P. Parallel and distributed application of an urban and regional multiscale model. *Computers and Chemical Engineering 21*, 4 (December 1996), 399–408.

[17] 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.

[18] ODMAN, M. T., AND RUSSELL, A. G. A multiscale finite element pollutant transport scheme for urban and regional modeling. *Atmospheric Environment 25A* (1991), 2385–2394.

[19] O'HALLARON, D., WEBB, J., AND SUBHLOK, J. Performance issues in High Performance Fortran implementations of sensor-based applications. *Scientific Programming 6*, 1 (Spring 1997), 59–72.

[20] 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.

[21] RANGANATHAN, M., ACHARYA, A., EDJLALI, G., SUSSMAN, A., AND SALTZ, J. Runtime coupling of data-parallel programs. Tech. Rep. CS-TR-3565, Department of Computer Science, University of Maryland, December 1995.

[22] RIEDEL, E., BRUEGGE, B., RUSSELL, A., AND MCRAE, G. Developing GEMS: An environmental modeling system. *IEEE Computational Science and Engineering 2*, 3 (Fall 1995), 55–68.

[23] SEGALL, E., STEENKISTE, P., KUMAR, N., AND RUSSELL, A. Portability and scalability of a distributed multiscale air quality model. In *U.S. EPA Next Generation Environmental Models Computation Methods (NGEMCOM) Workshop* (Cape May, NJ, August 1995), EPA.

[24] STEENKISTE, P., AND SUBHLOK, J. Coordinating foreign modules with a parallelizing compiler. Tech. Rep. CMU-CS-97-145, School of Computer Science, Carnegie Mellon University, June 1997.

[25] 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.

[26] 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.

[27] 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.

[28] SUBHLOK, J., AND YANG, B. A new model for integrated nested task and data parallel programming. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Las Vegas, NV, June 1997), pp. 1–12.

[29] 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).

[30] YOUNG, T. R., AND BORIS, J. P. A numerical technique for solving ordinary differential equations associated with the chemical kinetics of reactive flow problems. *Journal of Physics and Chemistry 81* (1977), 2424–2427.