Appears in IPPS/SPDP 98

Airshed Pollution Modeling: A Case Study in Application Development in an HPF Environment

Jaspal Subhlok, Peter Steenkiste, James Stichnoth* and Peter Lieu
School of Computer Science
Carnegie Mellon University
Pittsburgh PA 15213
{jass,prs,stichnot,pjl}@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 an HPF-based environment is highly suitable for developing complex applications, including multidisciplinary applications.

1 Introduction

This paper reports on the development of *Airshed*, a large air pollution modeling application [18], in the Fx parallel programming framework [28]. Our goal is to demonstrate that high level parallel programming languages like Fx and High Performance Fortran [13] 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.

*James Stichnoth is currently with Intel Corporation, Santa Clara, CA.

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.

This paper has three 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. The paper presents a simple performance model that captures application performance across this entire space. 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. Finally, we present a framework for coordinating external parallel modules with the Fx environment, and thereby opening up the programming model, an important consideration for interdisciplinary applications. The Fx Airshed air pollution modeling application was integrated with a parallel 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. We presents its performance in Section 3. In Sections 4, 5, and 6 we discuss the performance model, the use of task parallelism, and integration with foreign modules, respectively. We discuss related work in Section 7 and summarize in Section 8.

2 Parallel Airshed

We describe the Airshed application and outline its parallel 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 [18]. This model predicts the concentration of different chemicals in the atmosphere using their initial values and 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. The effect of air pollution control measures can be evaluated at a low cost making it possible to select the best strategy under a given set of constraints.

The URM model is based on the atmospheric diffusion equation,

$$\frac{\partial c_i}{\partial t} + \nabla.(\mathbf{u}c_i) = \nabla.(\mathbf{K}\nabla c_i) + f_i + S_i \qquad (1)$$

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

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

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 [19]. 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 [16].

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 to use 1-dimensional transport operators 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.

```
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

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 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 Parallel Fx implementation

The Airshed application was implemented using Fx data parallelism. Fx supports directives to guide the layout of array data onto a subset of nodes and allows for 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 [13] and further details are available in [25, 29].

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.

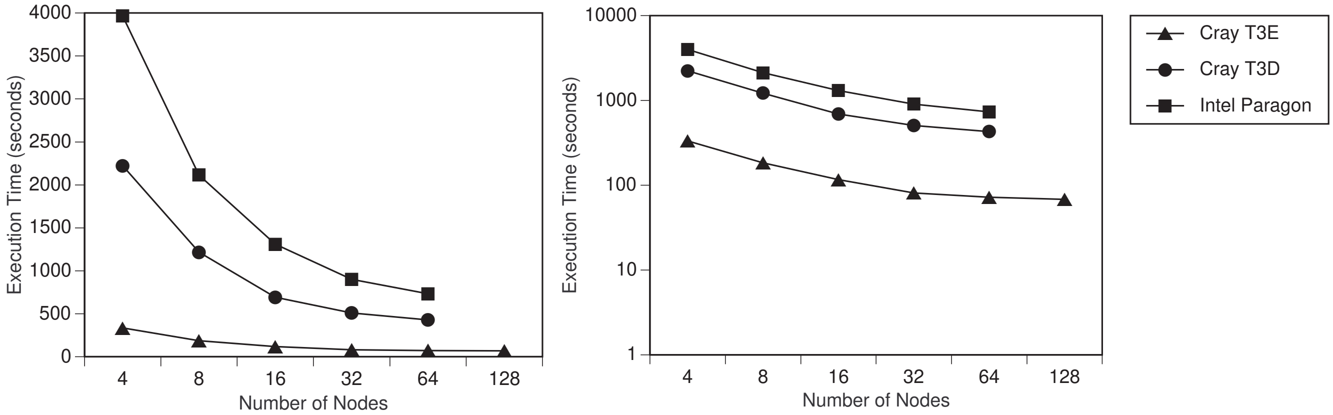
The *I/O processing* computations have limited parallelism and are handled sequentially in our implementation. The *chemistry* computation is independent for each grid point, and hence is parallelized along the "nodes" dimension; this means that it has a high degree of parallelism. The exception is the 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:



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

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

Transport $\rightarrow$ **Chemistry** $\rightarrow$ **Aerosol** $\rightarrow$ **Transport**
This results in the following data re-distribution steps in the main loop:

D_Repl $\rightarrow$ **D_Trans:** A(*,*,*) to A(*,BLOCK,*)

D_Trans $\rightarrow$ **D_Chem:** A(*,BLOCK,*) to A(*,*,BLOCK)

D_Chem $\rightarrow$ **D_Repl:** A(*,*,BLOCK) to A(*,*,*)

Note that a **D_Chem** $\rightarrow$ **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** $\rightarrow$ **D_Repl** $\rightarrow$ **D_Trans**.

3 Performance Results

In this section, we report our measurements of the performance of Airshed. In the next section, we show that a simple model can be used to explain and predict performance on different machines and for different numbers of nodes.

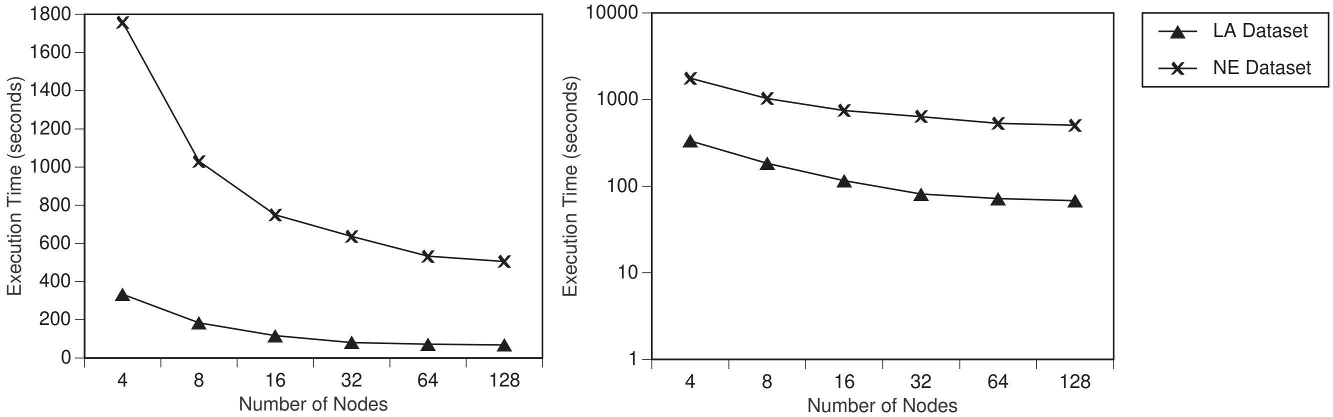
The execution times for the data parallel Airshed application executing on a Cray T3E, a Cray T3D and an Intel Paragon XP/S, using the Los Angeles basin data set, is presented in Figure 2. We observe significant speedups on all platforms, although the speedup is not linear. For example, on the Intel 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.

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.

To understand the performance better, we analyze 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 most 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 initial 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.

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 [17, 6]. 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} is however 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 [6], but because of their lower efficiency, they may not necessarily have better abso-



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

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

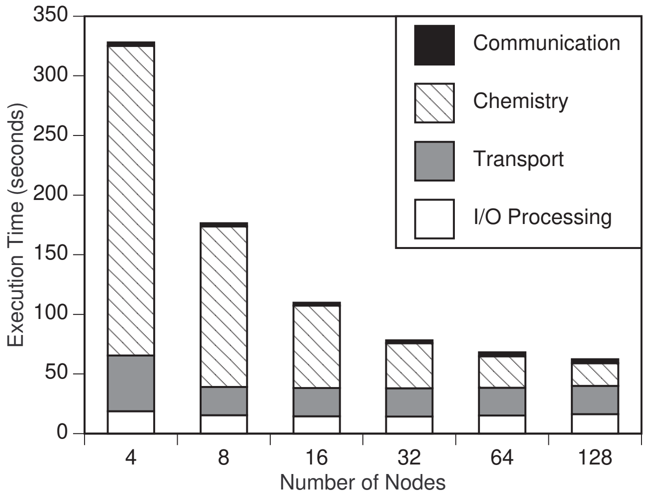


Figure 4: Scaling of the execution time of Airshed components on a Cray T3E for the LA data set

lute 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 multiscale version of Airshed is an important and challenging problem.

4 Predictable performance

We demonstrate that the performance of the Fx generated Airshed application on different machines and different numbers of nodes can be modeled in a simple way. As a result, the performance is broadly predictable, an important consideration in parallel programming. As is the case with most data parallel applications, Airshed can be divided into parallel computation phases and communication phases. We will discuss their performance models separately.

4.1 Computation performance model

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 generally 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 the minimum of 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.

The three main computation components of Airshed fit this simple model. We use the measurements represented by Figure 4 again for illustration. 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 constant since this computation is sequential. This general pattern was also observed for the Paragon and the T3D machines, as well as for the North East data set.

4.2 Communication performance model

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 costs is very important since they often limit 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:

$$C_t = L * m + G * b \quad (1)$$

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:

$$C_t = L * m + G * b + H * c \quad (2)$$

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 equation (2) 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 to send or receive the largest amount of data. (We are implicitly assuming here that maximum data transport corresponds to maximum communication time, but this assumption can be relaxed.) 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 only locally copied on the node (c). We now illustrate this process by analyzing the cost of redistributing the concentration array for each of the three communication steps in the main computational loop (Section 2.2) of Airshed. The time spent on each of these steps for the Los Angeles data set on the Cray T3E is plotted in Figure 5.

The step $D\_Repl \rightarrow D\_Trans$ converts a replicated array to an array that is distributed in a dimension with only 5 elements. This causes a local data copy but no actual transfer of data across nodes as the relevant data is locally available. For an array described by the dimensions $A(species, layers, nodes)$, Equation (2) shows that the cost of this operation is described by:

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

where P is the number of the processing nodes and W is the machine wordsize in bytes. Note that the *ceil* operation is required here and in future equations, since the node with the largest amount of data should be considered for cost computations when the data is divided unevenly. The meaning of the above equation is that the cost of this step is proportional to the maximum amount of 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, 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 observed in Figure 5, which is similar to the scaling of *transport* computation discussed earlier.

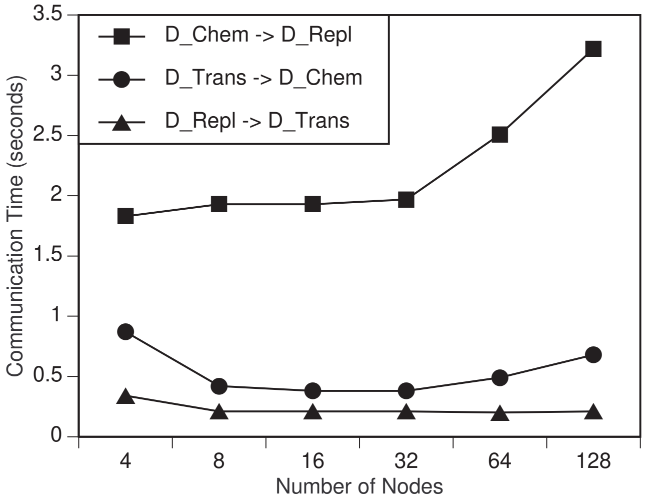


Figure 5: Scaling of communication steps in Airshed for the LA data set on the T3E

The redistribution step $D\_Trans \rightarrow D\_Chem$ copies an array distributed in the "layer" 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 is the number of nodes. Hence the cost of this operation, based on Equation (2) 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 it remains constant for large numbers of nodes. The change in the amount of data sent is responsible for the large drop in cost from 4 to 8 nodes in Figure 5. 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.

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.

4.3 Predictable performance

The models used in this discussion are, of course, very simple and not new. Our contribution is the insight that a parallelizing compiler responsible for generating code for the computational and communication phases of a problem can assist in the process of performance estimation. Given information on the features of the input data set, the compiler can easily 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 predict the execution time and scalability of an application with a fair degree of accuracy.

We estimated the performance parameters for the Cray T3E for the communication generated by Fx using measurements for a small number of nodes to be as follows:

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

We used these parameters in the equations for the communication steps described above and obtained the communication time estimates for Airshed executing with the LA dataset on a Cray T3E. For completeness, the communication times plotted represent 77 communication steps, and the *Wordsize* used on the T3E is 8 bytes. The results obtained are plotted in Figure 6 and compared to those obtained by measurements. We note that the estimated and measured values are close to each other, implying that the communication times for the entire spectrum of communication patterns and numbers of nodes are captured by three fundamental measurable machine parameters. Small differences between the two sets of values do exist, which is not surprising given the simple nature of the estimates.

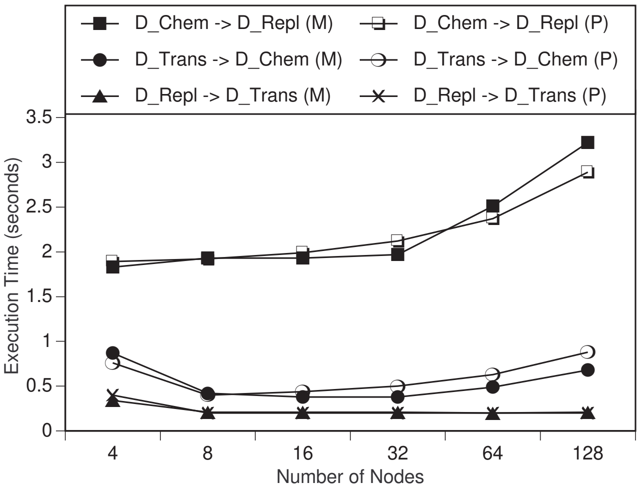


Figure 6: Predicted (P) and Measured (M) times for the the communication steps of Airshed with the LA data set on the T3E

We also obtained similar estimates for the computation phases. Figure 7 shows the estimates from the complete performance model and compares them with execution measurements. We note that the estimates and measured values match closely for the computation phases also. In fact, the values for the computation phases appear to be closer to the predictions than the communication phases. We conjecture that the reason is that estimation of computation times is less complex than that of communication times.

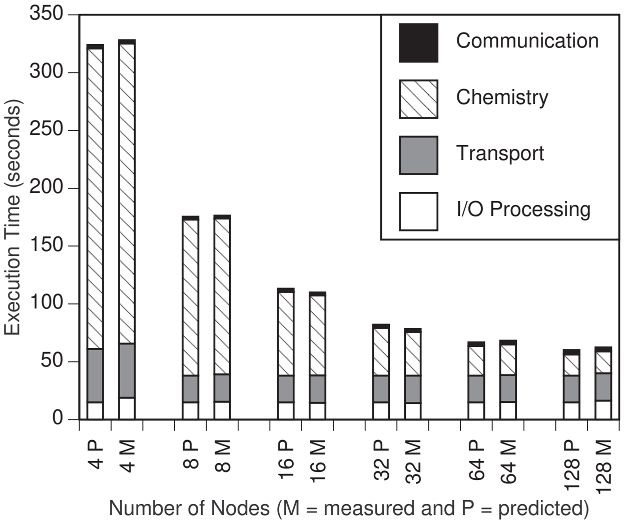


Figure 7: Predicted and Measured times for the computation phases of Airshed with the LA data set on the T3E

In summary, these results demonstrate that the execution behavior of the Fx generated Airshed application follows a predictable pattern. While we demonstrated this for only a single application, we believe that this process can be automated and extended to other applications. In particular, the results for the communication model (Figure 6) 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.), it is clear that a rough estimate of the execution time of an application can be obtained. This can be used to explain 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.

5 Task parallelism for Airshed

Task parallelism is supported in Fx by the use of mechanisms to distribute data structures onto subgroups of nodes, and a mechanism to specify execution on a subgroup of nodes. This allows independent sequential and data parallel routines to execute concurrently on disjoint groups of processors. The syntax and semantics of task parallelism in Fx are described in [28]. A similar task parallel construct is also an approved extension of High Performance Fortran [13].

Task parallelism makes it possible to reduce or eliminate the impact of application components with limited parallelism on the overall execution time. 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 [11, 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.

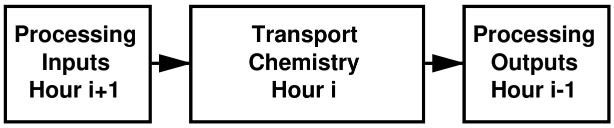


Figure 8: Pipelined task parallelism in Airshed

Given the dependencies between the input and output processing stages and the main computational loop, it is natural to use task parallelism to break up the computation in three pipelined stages, as shown in Figure 8. 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 9. The figure shows a significant improvement in scalability with the use of task parallelism. In particular, the execution time on 64 nodes was reduced by around 25%.

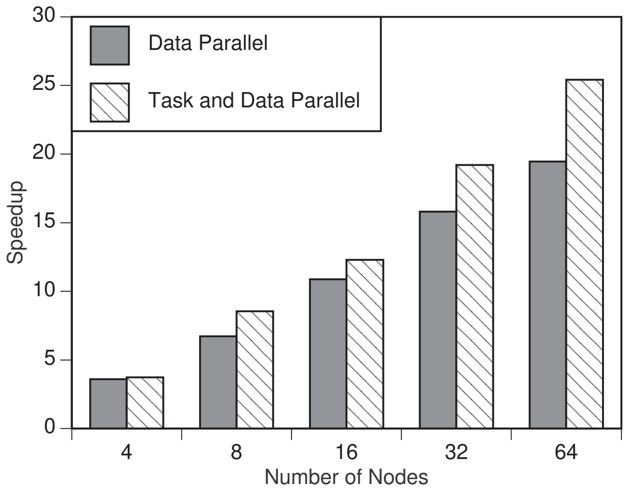


Figure 9: Speedup of Airshed on an Intel Paragon

6 Foreign module interface

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 the GEMS problem solving environment [22], as illustrated in Figure 10. However, different programs are often parallelized in different frameworks, and 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.

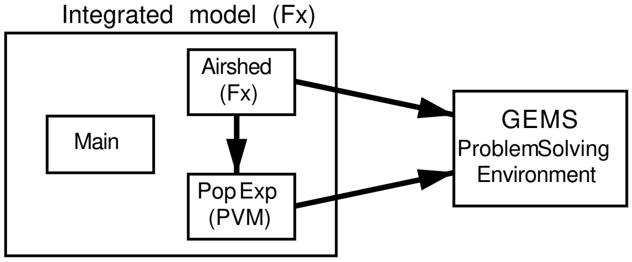


Figure 10: Combined Airshed and Population Exposure modules

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, 3, 10], 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. We will present our approach to the problem here, using Airshed+PopExp as an example. We include a comparison with other methods used in the HPF domain in the related work section.

We are investigating module integration based on the use of a common shared collective communication library for communication between a *native program* generated by a parallelizing compiler and an external *foreign module*. Our experimentation is with the Fx compiler system, hence the native programs in this discussion are actually Fx programs. In this model, a foreign module is an independent executable that may have been developed in a different language and parallelism model. In the native Fx program, a foreign modules is represented as a task. This task is associated with a node subgroup, that represents the nodes where the foreign module executes. Data is exchanged between the native Fx program and the foreign module through Fx variables that are mapped onto that subgroup. The foreign module uses a communication layer that it shares with the Fx runtime system to exchange data with the native Fx program. However, the internal communication model of the foreign module is independent of the communication model used by the Fx compiler. This approach is discussed in detail in [24]. We will summarize the key features here:

1. Modules written in different parallelism models can be integrated in a single framework. The main change in the foreign module is the insertion of calls to a shared library to input and output data from the native Fx program.

2. 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.

3. The interaction between the native Fx program and a foreign module is very general. Specifically, the interaction can be asynchronous, i.e., it is not of the form of a procedure call, and the foreign module can continue execution in parallel with the Fx program.

4. 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.

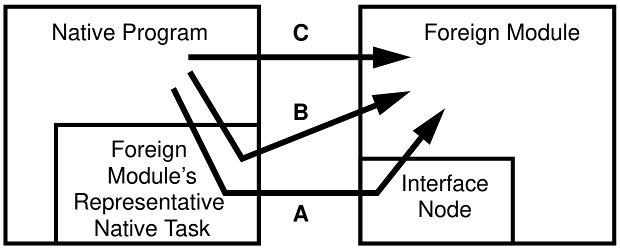


Figure 11: Optimization of communication with a foreign module

Support for synchronization and 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 all 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 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 [7].

Our prototype implementation of foreign modules is based on the simplest approach **A** above and it was used to integrate the PopExp program with Airshed. The structure of

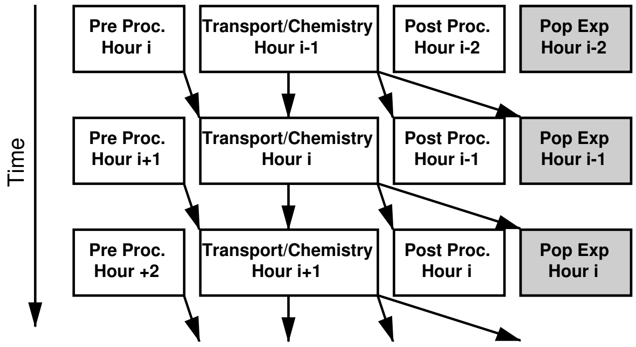


Figure 12: The structure of the Airshed and PopExp computation

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 discussed earlier. To quantify 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, extra overhead associated with the foreign module approach. However, it does not significantly impact overall performance, and as we described above, a more aggressive implementation could reduce this extra overhead if needed. Given the advantages in code reuse, this preliminary evaluation suggests that the foreign module approach is attractive.

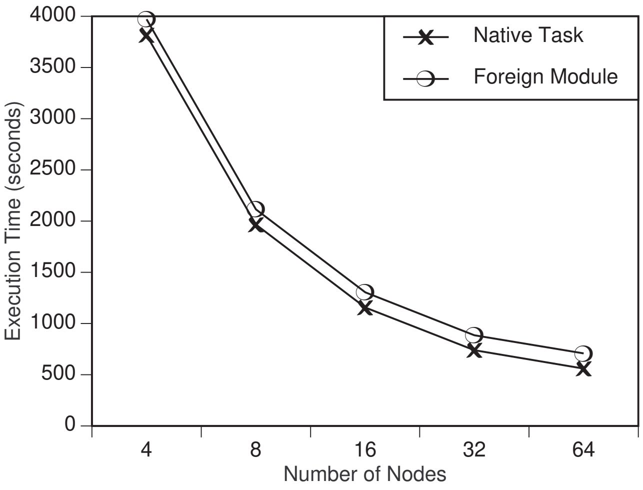


Figure 13: Performance comparison with PopExp as native and foreign module on an Intel Paragon

7 Related Work

Several research efforts have addressed compilation of data parallel languages, two of the pioneering ones being Fortran D [14] and Vienna Fortran [5]. Data parallelism in Fx, the framework used in this research, is discussed in [25, 29]. With the standardization of High Performance Fortran [13], 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 [4, 8, 12].

The Airshed model for pollution modeling was developed by McRae and Russel, and earlier parallel implementation, based on PVM, was developed by Russel et. al. [17]. A parallel implementation of the original uniform grid CIT model is described by Dabdub et. al. [6]. 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 [11] 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, 3, 10] but they are primarily targeted to communication between sequential processes. High Performance Fortran [15] 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. [9] 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 successfully solved in this environment.

Results are presented for the Airshed application executing on Intel Paragon and 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 provide evidence that the performance is broadly predictable and fits a simple model. 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 most suitable for developing such applications. We believe that the experience reported is representative of the development of multidisciplinary applications.

9 Acknowledgements

This work was supported by the Pittsburgh Supercomputing Center 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 used in this research. Thomas Gross, David O'Hallaron, Tom Stricker and Bwolen Yang contributed to the design and development of the Fx compiler system.

References

[1] Barbacci, M. Software Support for Heterogeneous Machines. Tech. Rep. SEI-86-TM-4, Software Engineering Institute, Carnegie Mellon University, May 1986.

[2] 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.

[3] 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.

[4] 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.

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

[6] Dabdub, D., and Seinfeld, J. Air quality modeling on massively parallel computers. *Atmospheric Environment 28*, 9 (1994), 1679–1687.

[7] 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.

[8] 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.

[9] 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).

[10] Foster, I., and Taylor, S. *Strand: New Concept in Parallel Programming.* Prentice Hall, Englewood Cliff, New Jersey, 1990.

[11] 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.

[12] 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).

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

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[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, p. Appears in this collection.

[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* (June 1997), ACM.

[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.