Coordinating Foreign Modules with a Parallelizing Compiler

Peter Steenkiste Jaspal Subhlok

June 97

CMU-CS-97-145

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Integrating task and data parallelism in a language framework has attracted considerable attention. Both the Fx language at Carnegie Mellon University and High Performance Fortran standard have adopted a simple model of task parallelism that allows dynamic assignment of processor subgroups to tasks in the application. However, the tasks must be written in the native language, i.e Fx or HPF. Large scientific parallel applications often use a mix of languages for a variety of reasons, including the reuse of existing code and the suitability of different languages for different modules. In this paper we demonstrate how a "native" parallelizing compiler can be used to create parallel applications that combine native modules with "foreign" modules written in a different parallel language. We argue that virtually all the advantages of translating a foreign module to the native language can be achieved by using the native compiler to coordinate the interactions with the foreign module. In particular, the ability to dynamically modify the assignment of processors among native modules and foreign modules is retained. The foreign module interacts with the native program through shared arguments and the changes in the code for the foreign module are minimal. We demonstrate how this idea allowed us to accelerate the development of the Airshed air quality model at Carnegie Mellon using the Fx language and parallelizing compiler. We also examine the tradeoffs between the nature of interaction permitted with the foreign module and the complexity of language and runtime support.

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.

Keywords: High Performance Fortran, coordination languages, task parallelism, interdisciplinary applications, application integration

1 Introduction

Many interdisciplinary applications require that several existing programs be combined into a larger application. An example of such an interdisciplinary system is a model that simulates the fate of pollutants in a multimedia environment coupled with an exposure assessment model that evaluates the impact of pollution on public health. Such a complex model would include as components a photochemical model, an emissions model, a water quality model, and an exposure model. While integrating independent programs is already a difficult task, this is further complicated by the fact that the individual programs will often be parallel programs, which, in general, have been parallelized using different programming tools. Moreover, these models are often expensive to execute, so an important goal of the integration is that the resulting model can be executed in parallel with reasonable efficiency.

There is a wide range of choices to accomplish this integration. A first approach is to use ad hoc solutions, such as running the programs independently and using file sharing for communication. The high communication cost and long-term maintenance costs make this approach unattractive. At the other end of the spectrum lies the option of merging the independent programs into a single program. This is in general very hard and time consuming (i.e. expensive) since the different programs will often use different languages, tools and runtime systems. Moreover, it requires fairly detailed knowledge of all programs, which is rarely present in any group. An intermediate solution is to use tools and languages to create "glue programs" that integrate the independent programs in a systematic way. Such programs are written in coordination languages such as Durra [2], Hence [3], Strand [12], and Linda [4]. These languages make it possible to build programs by connecting existing modules, based on separately specified interfaces for each module. Although these languages have had some success, they have not seen widespread use. We speculate that the main reasons are that a new and unfamiliar language has to be used and that the languages do not adequately address performance issues in general, and the use of parallelism in particular.

Our approach is similar to that of coordination languages, but we use an *existing* language that is designed for *efficient execution* and use of *parallelism*. Specifically, we propose to use High Performance Fortran [18, 16]. We use the Fx language and compiler [14, 27], a variant of HPF developed at Carnegie Mellon, to validate the concept. HPF is widely used and a lot of compiler technology [7, 17, 27] has been developed to support efficient execution of HPF programs. Following recent research in the integration of task and data parallelism [6, 9, 14, 22], a form of task parallelism has been added to HPF as an approved extension. We propose to use the HPF tasking mechanism to support the integration of independent parallel programs: external programs will be represented in the integrated application as tasks, which we will refer to as *foreign tasks or modules*.

A foreign module is an independent executable that may have been developed using a different language and parallelism model. The relationship between a foreign module and the main program is very flexible and is not necessarily of the form of a subroutine call. In particular, the foreign module can continue execution in parallel with the main program. In the native program, the foreign module has a representing processor subgroup and native and foreign modules exchange data through variables mapped onto that subgroup. This communication is implemented using a shared communication library that is based on the native runtime system. However, the internal communication model of the foreign module is independent of the communication model used by the native compiler.

This approach offers the advantages of a single integrated application without the effort of translating the application to one language or programming model. For example, a common task management library can optimize the allocation of nodes across the entire application including the foreign module. The interaction with the foreign module is general and asynchronous but it can be done efficiently since it uses a common communication fabric.

The remainder of this paper is organized as follows. We present the Fx task programming model and its integration with foreign modules in Section 2. In Section 3 we describe the implementation issues in supporting foreign modules. We present our motivating application and its performance in Section 4 and 5. We compare our approach to related research in Section 6 and summarize in Section 7.

2 Foreign modules in a parallel language

This section explains the concept of a foreign module and discusses how it increases the power and flexibility of a programming environment based on a high level parallel programming language. The "host" language framework is assumed to have support for data and task parallelism along the lines of Fx and High Performance Fortran (HPF), and will be referred to as the "native language". To make our discussion concrete, we will present examples using the Fx language and compiler. However, the concepts discussed are applicable to HPF and similar languages. We briefly describe the Fx programming model and discuss how it is enhanced to support foreign modules.

2.1 Native programming model

Data parallelism in Fx, as in HPF, is based on distributing data across processors. The data distributions supported include block, cyclic and block-cyclic. Loop parallelism is expressed by a special parallel loop construct that combines loop and reduction parallelism. Since the details of data parallelism are not directly relevant to this research, we will not discuss these any further and refer the interested reader to [27, 31].

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 [30]. We outline the main directives used to achieve these purposes.

- 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 is dynamic.

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

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

  maps variables `A1` and `B1` to 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
  ```

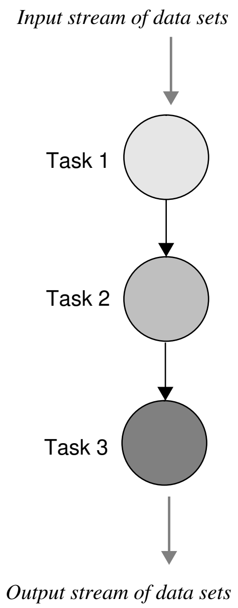
  directs that subroutine `task1` be executed on processors assigned to subgroup `part1` and subroutine `task2` be executed on processors assigned to subgroup `part2`. The code inside a subroutine called in an ON SUBGROUP region may further subdivide the processors. Hence task parallelism can be used to exploit nested parallelism.

We illustrate the use of task parallelism with an example. Figure 1(a) uses pseudocode to illustrate a simple pipeline containing 3 tasks, while Figure 1(b) shows the structure of the computation. Each task repeatedly receives its input data from a predecessor task, computes, and sends output to a successor task. Task 1 reads external input and Task 3 generates the final output. Thus, every iteration acts on a new data set and the corresponding read and write statements are inside routines Task1 and Task3, respectively.

A high level task and data parallel implementation of this pipeline in Fx is shown in Figure 1(c). The `TASK_PARTITION` statement is used to divide the processors into three named subgroups, and a `SUBGROUP` statement is used to map the variables `A1, A2` and `A3` onto named subgroups `P1, P2` and `P3`, respectively. `TASK_REGION` directives are used to define a task region, and `ON SUBGROUP` directives are used to map the computations, i.e. the three subroutines `task1, task2` and `task3` onto the three processor subgroups. Data is exchanged between subgroups by assignments of the form `A2=A1`, where the variables on one or both sides of the assignment have been mapped on a `SUBGROUP` region. The assignment statements are in a `TASK_REGION` region, but `not` in a `ON SUBGROUP` region, and they can therefore access all data. During execution, the three subgroups are typically working on different datasets in pipelined fashion.

```
do i = 1,m
   call task1(A)
   call task2(A)
   call task3(A)
enddo
```

(a)



(b)

```
TASK_PARTITION ::  P1(1),P2(x),P3(N-x)
SUBGROUP(part1) ::  A1
SUBGROUP(part2) ::  A2
SUBGROUP(part3) ::  A3
BEGIN TASK_REGION
   do i = 1,M
      ON SUBGROUP P1
         task1(A1)
      END ON
      A2 = A1
      ON SUBGROUP P2
         task2(A2)
      END ON
      A3 = A2
      ON SUBGROUP P3
         task3(A3)
      END ON
   enddo
END TASK_REGION
```

(c)

Figure 1: A 3 stage task and data parallel pipeline in Fx

The details of task parallelism in HPF are different from Fx, but the basic ideas are the same, hence the results of this paper carry over to HPF.

2.2 Motivation for foreign modules

The task parallelism model discussed above allows considerable flexibility in building parallel programs from modules, but only if all the modules are written in the same language. Large applications are often best developed by composing multiple modules written in different languages. In the example illustrated in Figure 1, we may wish to use a display module developed in a different parallel programming language and paradigm to view intermediate results, yielding the computation structure shown in Figure 2. The challenge is to integrate the display module TaskF with minimum effort while achieving high efficiency and maximum flexibility in its usage.

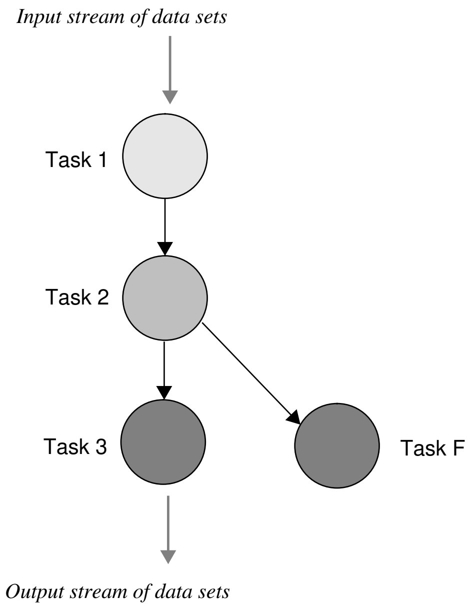


Figure 2: A pipeline with a foreign module

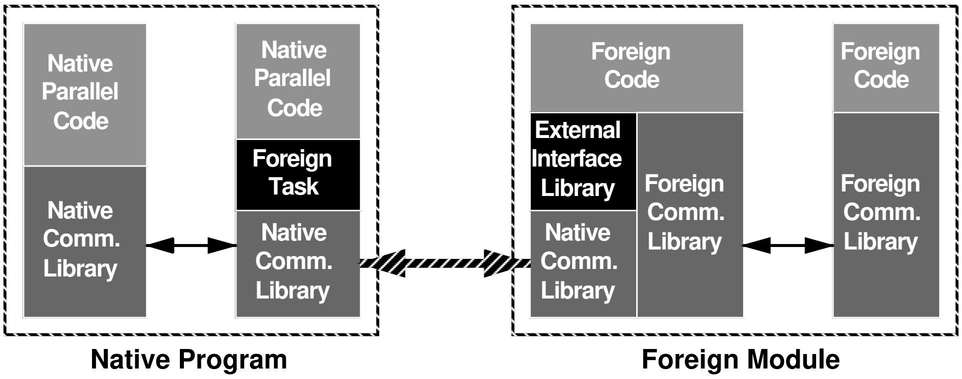


Figure 3: Architecture of foreign module interaction

Mechanisms for integrating a foreign module with another program range from rewriting it in the native language to allowing it to execute completely independently with data exchange through a common filesystem. Our solution is to allow the main program and the foreign module to execute independently, but to use the parallelizing compiler and the runtime system to provide efficient and flexible communication and coordination between the modules. This approach has the advantage that the foreign module code does not have to be rewritten in the native language or linked with the main program; if the foreign module uses a different parallel programming paradigm, these tasks can be complex and cumbersome. The foreign module can also execute asynchronously in parallel with the native program, increasing both flexibility and performance. Finally, if the foreign module is written in a way that it can use a varying number of compute nodes, processors can be allocated and reallocated dynamically between the native program and the foreign module. This approach requires "glue" code for the foreign module to interact efficiently with the main program and the runtime system, although some glue code will be needed with any method of integration.

We believe our approach is a cost-effective compromise between the easy but potentially inefficient approach of using separate programs, and the efficient but potentially expensive solution of full integration.

```
BEGIN TASK_REGION
      ...
   Af = A2
   ON SUBGROUP Pf
      call taskF(Af)
   END ON
   A3 = Af
END TASK_REGION
```

Figure 4: Call to a foreign task from a native module

2.3 Architecture

Figure 3 shows a high level picture of a native program integrated with a foreign module. The foreign module is an independent executable that executes concurrently with the main program and exchanges data with it at select points. The foreign module uses an *external interface library* that is based on the native runtime system to exchange data with the native modules of the application. However, the internal communication model of the foreign module is independent of the communication model used by the native compiler and runtime system. The native program consists of the parallel application code generated by the parallelizing compiler. The parallelizing compiler inserts calls in the application code to the native communication library, and to start up and initialize the parallel computation.

Let us assume that the foreign module started out as a stand-alone program. The foreign code is modified so that inputs are received from the native program and outputs returned to the native program, as needed. The foreign module's external interface library is responsible for implementing these calls in terms of the communication primitives supported by the native communication library. This code would typically replace part of the program's normal I/O code, e.g. calls to read and write to/from the file system. The foreign library also includes code for initialization and synchronization with the native program. There will be a single external interface library for each parallel language used by foreign modules, i.e., the external interface library is application independent.

In the native program, the foreign module is represented by a *foreign task*, which is in effect a stub for the foreign module. The interface to invoke a foreign task is identical to a call to a native task subroutine. For example, for the computation structure in Figure 2, a processor subgroup `Pf` is created and declared foreign. Inside the task region, the code from Figure 1 is modified by adding an assignment to a variable `Af` mapped to processor subgroup `Pf` (Figure 4). Assignments to and from the variable `Af` will be translated into communication with the foreign module, in the same way that they would for a native task.

3 Implementation

We describe how foreign module support can be implemented using an HPF-style system as the native platform. We discuss task invocation, communication optimizations, and processor allocation.

3.1 Process structure and foreign task invocation

A foreign module is either initiated by the native program or starts off as a separate executing program. Subsequently, it rendezvouses with the native program and exchanges data with it at select points. These points are marked by calls to the foreign task stub in the native program and calls to the external interface library in the foreign module. This model is more powerful than a "server" model where the nodes assigned to the foreign module run a server that periodically receives "execute" requests from the native program and returns results after it finishes execution of the parallel task. The server then sleeps until it receives the next execute request. The server model is simple to implement, although our "communicating parallel processes" design is not significantly more difficult to support. This generalization is of value in several situations. For example, foreign modules can continue to perform input/output activities such as driving a display device, or can perform independent computations like refining a solution, after the foreign task invocation has returned. Note that if the foreign module continues executing between invocations, which is supported under the general "parallel processes" model, it should not modify any data structures that are shared with native tasks, since this would violate the HPF programming model. Implementations of the general model should also consider that the foreign module may become less responsive to task execution requests, potentially slowing down the entire application.

The foreign module is represented in the application by a representative foreign task. Some changes are needed in the parallelizing compiler to support foreign modules since the compiler cannot analyze the code in the foreign module in the way that it can analyze native tasks. The native compiler needs another mechanism to collect essential information like the distribution of shared variables in the foreign module, and optional information such as the scaling properties of the foreign module as a function of the number of nodes. This information will generally be provided by the programmer or exported by the foreign module. A simple method is to associate a set of declarations with the foreign module that can be used by the native compiler. Note that for a specific class of foreign modules, the collection of this information could be very systematic, or even automatic. For example, for foreign modules that are based on a specific distributed object library such as Dome [1], this information could be automatically exported by the objects.

To invoke a foreign module at runtime, the native program calls the foreign task representing the foreign module. The foreign task sends a request to the processors assigned to the foreign module, provides the input data, and then waits for the results to be returned. Communication uses the native communication library and is discussed in more detail later.

3.2 Synchronization

An interesting challenge in adding foreign module support to an HPF-style system is synchronization. HPF implementations are based on SPMD programming model, i.e. each processor executes the same executable, so it has access to appropriate application control flow information. For example, information on the outcome of conditional instructions is available to the relevant processes as a result of the way the program code is generated by the parallelizing compiler. In an application that includes foreign modules, task synchronization (i.e. task invocation and termination) is handled by the foreign task, but synchronization required for communication, i.e. pairing up readers and writers, is harder to resolve.

The example in Figure 4 shows how input and output statements are used to move data into and out of tasks. In the simple case, input and output statements are executed unconditionally, and it is simple to determine what data has to be communicated with the task. Conditionally executed input or output statements are not a problem for native tasks, since all processes share the same control flow, but they are a problem for foreign modules. The problem is easy to solve for input statements: the runtime system can collect information on required input data, and then send the data when the foreign task (and module) is invoked. The synchronization problem is more difficult for output statements. One solution is to have the foreign module always return a superset of the results, but this might be expensive and inefficient. An alternative is to have output statements send "read requests" to the foreign module. The most efficient, but also most complex solution, is to include enough of the application flow control into the foreign module, for example as a code segment that is called by the foreign library, so that the foreign module can determine what output statements will be executed.

3.3 Communication

Support for synchronization and communication between a foreign module and the native program can be implemented in a variety of ways and presents a tradeoff between the complexity of the implementation on one hand, and programming flexibility and performance on the other. We shall outline the main options which are represented by different arrows in Figure 5. We shall only discuss the transfer of data from the native program to a foreign module, since the reverse scenarios are analogous.

We first describe the simple case represented by scenario **A** in Figure 5. In this case, the communication calls between the sender and receiver are always matched. The parallelizing compiler exports its model of communication generation for calls to foreign tasks and the user ensures that matching calls are made in

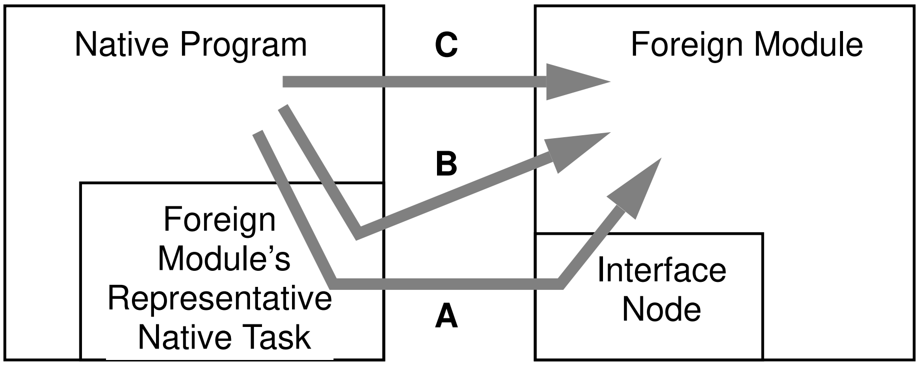


Figure 5: Optimizations for communication with a foreign module

the foreign module. The data is transferred from the native program to the representative foreign 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 the extra copies of data on the path from the native program to the foreign module. A special case that is particularly easy to implement arises when the number of processors assigned to the representative foreign task and the interface node of the foreign module are set to one since that obviates the need for parallel communication between the foreign module and the native task.

An important optimization to this mode of communication is represented by the scenario **B** in Figure 5. In this case the synchronization model stays the same, but 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. This optimization is clearly desirable since it eliminates one data transfer. However, implementation is more complex, especially if the data mappings inside the foreign module are not supported by the native compiler. For example, the foreign module may support sparse data structures which would imply that the native compiler and the native communication library have to be extended to support communication of distributed sparse data structures.

Finally, the most ambitious and most efficient scheme for communication is represented by the scenario **C** in Figure 5. In this case, the copying of data to the representative foreign task in the native program is completely eliminated and the data is directly transferred from the variables in the native program to the variables in the foreign module. This case is very hard to support in general since it requires that complete control flow information of the foreign module be made available to the native program and vice versa. The reason is that assignments to the variables of a foreign task can happen anywhere in the task region of a program, and to translate these without buffering requires that the native program know when it is safe to make the transfer, and the foreign module must be able to accept data in an unexpected order. However, further discussion is beyond the scope of this paper.

3.4 Processor allocation

In some applications, allocation of processors to tasks is an important optimization step. The parameters that determine a processor assignment include efficiency of tasks as a function of the number of processors, communication costs associated with a specific allocation, and system constraints (e.g. memory size). One of the main advantages of bringing foreign modules under the umbrella of a parallelizing compiler is that the compiler can optimize processor allocation across an entire application. To achieve this, the compiler must have access to the relevant task information. Since this information is typically obtained through profiling, this is not that different from the support needed for native tasks. Further, the foreign module must be written in such a way that it can execute on a variable number of processors, which is the case for most parallel codes today. Note that if the foreign code can be remapped dynamically, for example on a per call basis, then dynamic reallocation based on runtime information is possible. Hence the techniques developed for automatic mapping of tasks [22, 28, 29] can be applied to applications with foreign modules.

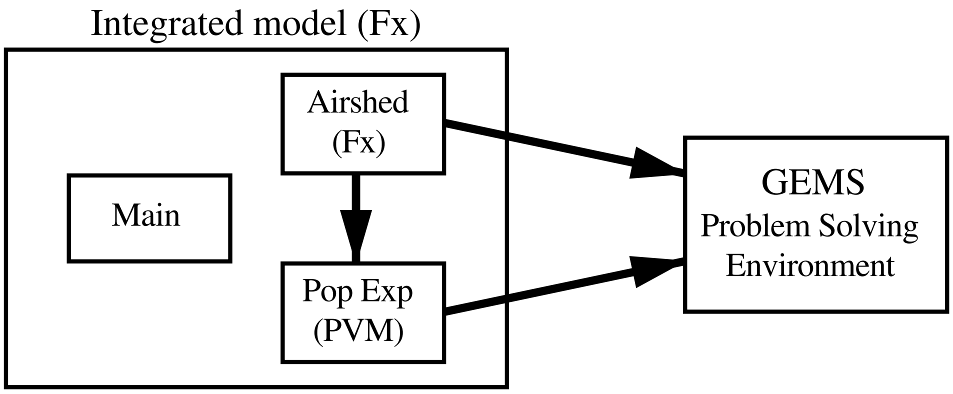


Figure 6: Integration of Airshed with a foreign module

4 Application example:Airshed

To test the feasibility of compiler-based support for foreign modules, we added a population exposure model, parallelized using PVM, to an air pollution model, parallelized using Fx.

The air pollution model we started with is the Urban Regional Model (URM) called Airshed, developed by McRae and Russell [20, 19]. The parallel Fx version of the model uses both task and data parallelism. Data parallelism is used to parallelize the modeling of the particle transport and the chemistry calculations. Task parallelism is used to execute pre-processing of inputs and post-processing of outputs in parallel with the main computation [30].

Air pollution models are often coupled with other models, such as surface or water quality models, to model more complex systems. In this experiment we coupled Airshed with a population exposure model, called PopExp, that uses the concentration data for chemicals, generated by Airshed, to calculate the impact on health. The population exposure model was written by an environmental scientist using explicit message passing (PVM) without the use of the Fx compiler. It takes as input a concentration array with chemical species and population data, and generates population exposure data. The parallel PopExp code follows a master-slave protocol: a master node distributes the data on concentrations of different chemicals to the slaves, which operate in parallel on different parts of the space.

Figure 6 shows how the integration was performed. We created a top level Fx program that invoked both Airshed and PopExp as tasks. The Airshed task is the original Airshed code written in Fx, while the PopExp task consists of a foreign task that represents the PVM parallel application in the Fx program. Airshed periodically sends hourly concentration data directly to the PopExp where it is received using a variant of the Fx communication library. The PopExp model then used PVM to distribute data internally and to support the parallel computation. Airshed is used by environmental scientists through the GEMS problem solving environment [25, 24] which allows them to model different scenarios and view and compare outputs. PopExp generates outputs in a such a way that it can be displayed in GEMS, so the integrated Airshed-PopExp application presents a single interface to users.

Figure 7 illustrates the parallelism in the integrated application. The boxes represent Fx tasks that can execute in parallel, each with internal data parallelism, and the arrows represent data dependencies that constrain execution order. Our implementation uses the simplest data transfer mechanism discussed in the previous section, i.e., the data is routed through a foreign task in the native program and an interface node in the foreign module, which may degrade performance.

Two versions of this integrated application have been developed. The first one executes on an Alpha cluster, where Fx uses PVM for native communication. The second one runs on an Intel Paragon, where Fx uses the NX libraries for native communication. The latter implementation supports dynamic reassignment of processors between the foreign module and the native program.

This example illustrates some of the advantages of compiler-supported foreign modules. First, it demonstrates that an application developed using a different computational model (explicit message passing using PVM) can be accommodated in this framework. Given the large number of existing explicitly-parallel mod-

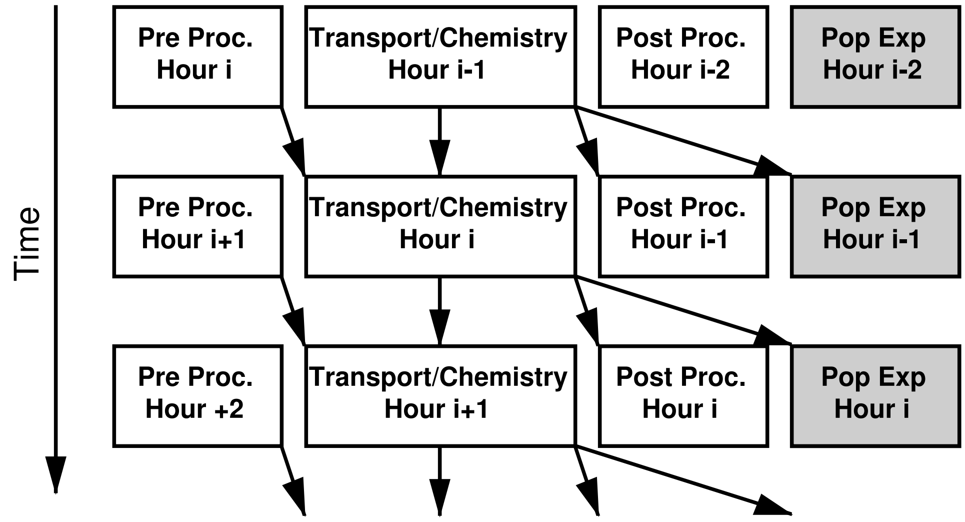


Figure 7: The structure of the combined Airshed-PopExp computation

ules, support for this class of applications is important. Second, we were able to bring up the integrated application quickly since it required minimal changes to Airshed and PopExp modules. Finally, we were able to change the balance of the number of nodes allocated to Airshed and PopExp using the existing Fx mechanisms.

5 Performance Evaluation

In this section, we quantify two performance measures for compiler-based foreign module support. First we look at the performance difference between using foreign modules and an all Fx implementation. Second, we illustrate how some of the performance advantages of integrated tools for parallel computing carry over to our approach to foreign modules. Specifically, we examine the value of the ability to dynamically adjust the processor allocation between a native Fx program and a foreign module. We use the Airshed model and the PopExp model executing on an Intel Paragon for all our measurements.

5.1 Performance characterization of foreign modules

To examine the value of translating a module to the native language as compared to integrating it as a foreign module, we developed an all Fx version of the Airshed/PopExp application in addition to the version in which Airshed is programmed in Fx and PopExp is a PVM foreign module. The purpose was to measure the extra overhead of calling PopExp as a foreign module and whether it justified the effort and expense of translating it to Fx. We verified that the stand alone versions of Fx and PVM PopExp had comparable performance.

The results obtained on an Intel Paragon are plotted in Figure 8. We observe that there is a fixed extra overhead associated with the foreign module approach. The reason is the extra copying in our current implementation of communication with a foreign module as discussed in Section 4, which should be reduced with the implementation of some of the communication optimizations discussed in Section 3. However, the overhead is not significant enough to be a large factor in the overall performance, and in most cases, it will not be sufficient to justify rewriting the foreign code.

5.2 Dynamic processor allocation

One of the interesting features of the population exposure module is that the amount of computation in it 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 multiple versions of PopExp with different amounts of computation. We then

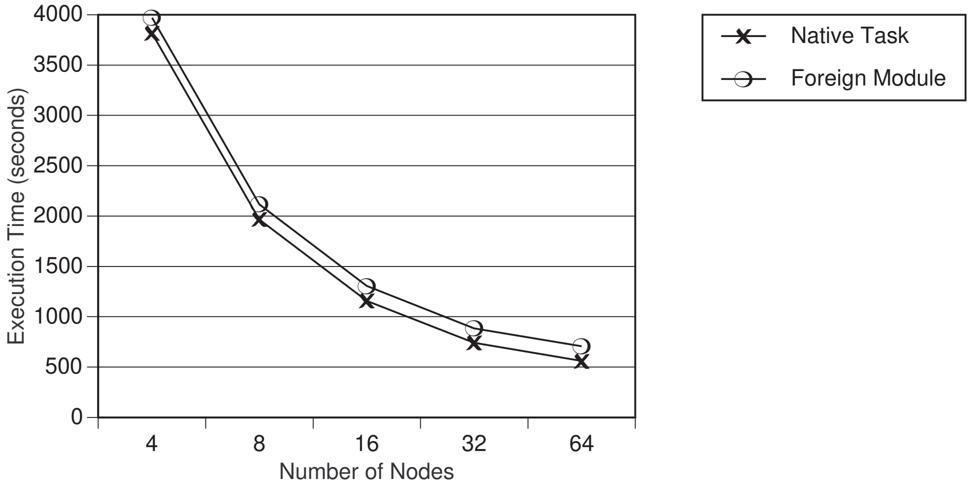


Figure 8: Performance comparison of PopExp as native and foreign module

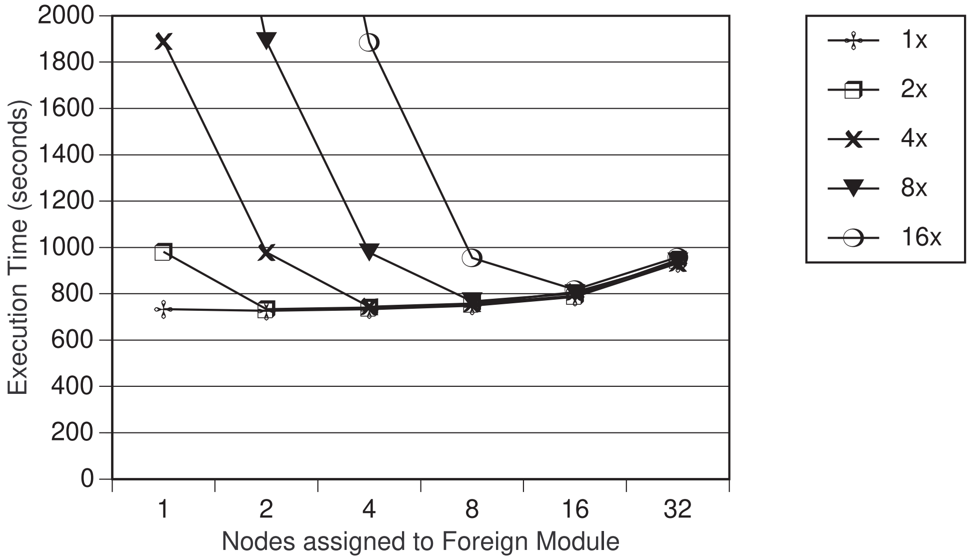


Figure 9: Performance with different processor assignments to foreign modules with different amounts of computation load

experimented with assigning different number of processors to the PopExp module and the Airshed module in the integrated application.

The results are shown in Figure 9. The graph shows the execution time as a function of the number of nodes assigned to the foreign module. Different curves are shown for different implementations 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 allocation is dependent on the version of the PopExp in use, and not surprisingly, the higher the amount of computation in the PopExp module, the larger the number of nodes allocated to it in the optimal mapping. This points out the value of the ability to load balance between a native program and a foreign module. This important feature can be supported by using a parallelizing compiler to coordinate foreign modules with the rest of the application.

6 Related Work

Support for foreign modules is a important problem in building large parallel applications and it is usually resolved in an ad-hoc manner. We review some of the main efforts that have proposed more systematic solutions.

Designers of High Performance Fortran [18] realized the importance of interaction with foreign code and provided a mechanism called an *EXTRINSIC* procedure call. Intuitively, this mechanism lays out the rules for transfer of control from HPF to a subroutine in another computation model, e.g., a C routine written in MPI, that is linked with the HPF executable. We are using an interface that is similar in nature to transfer control to another executable. However, our approach is designed to be more flexible and potentially more efficient. The native and foreign code need not be linked together, which can be a significant advantage if they use different programming models and runtime systems, and the foreign module can execute fairly independently from, and concurrently with, the native program. Moreover, the parallelizing compiler and runtime system can make certain optimizations, e.g. processor allocation and data movement, across the entire application.

Foster et. al. [11] use an MPI binding to HPF to enable multiple HPF executables to communicate using MPI collective communication operations. This approach is broadly similar to our approach and the main difference is 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 just a subroutine call in a task region.

Sharma et. al. [23] present a runtime approach to integrating heterogeneous codes. While the proposed system is somewhat similar to our system, we believe the use of a parallelizing compiler that has a global view of the entire application can result in better performance for many applications.

7 Conclusions

Components of large interdisciplinary applications often do not fit a single parallelism model and they are often developed using different paradigms. It is important to be able to build and orchestrate parallel programs using different types of components while maintaining high execution efficiency. We presented a simple approach to creating applications consisting of programs generated by a parallelizing compiler for a language like HPF, augmented with foreign modules developed with a different language and parallelism model. Our approach uses the parallelizing compiler as a coordination tool. The main advantage of this approach is that the involvement of the parallelizing compiler makes it possible to achieve the performance and flexibility of an application completely written in the native parallel language, even though some of the components are foreign modules written in different frameworks. For example, processor allocation and data movement can be optimized across the entire application.

We have used this approach to develop an integrated air quality modeling and related population exposure application. We present experimental results to validate that this is an effective approach to developing real parallel applications and presents a good tradeoff between development effort, performance, and usage flexibility.

References

[1] Arabe, J., Beguelin, A., Lowekamp, B., Seligman, E., Starkey, M., and Stephan., P. Dome: Parallel programming in a heterogeneous multi-user environment. Tech. Rep. CMU-CS-95-137, Computer Science Department, Carnegie Mellon University, April 1995. A condensed version of this paper has been presented at the International Parallel Processing Symposium, 1996.

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

[3] 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.

[4] Carriero, N., and Gelernter, D. Applications Experience with Linda. In *ACM Symposium on Parallel Programming: Experience with Applications, Languages, and Systems* (July 1988), ACM, pp. 173–187.

[5] 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.

[6] 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.

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

[8] 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).

[9] 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.

[10] Foster, I., Kesselman, C., and Tuecke, S. The nexus task–parallel runtime system. In *Proc. 1st International Workshop on Parallel Processing* (1994), pp. 457–462.

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

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

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

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

[15] Haines, M., Hes, B., Mehrotra, P., and Rosendale, J. V. Runtime support for data parallel tasks. In *Fifth Symposium on the Frontiers of Massively Parallel Computation* (1995).

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

[17] 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.

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

[19] Kumar, N., Russell, A., Segall, E., and Steenkiste, P. Parallel and distributed application of an urban and regional multiscale model. *Computers and Chemical Engineering* (1996), To appear.

[20] 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.

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

[22] 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.

[23] 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.

[24] Riedel, E., and Bruegge, B. Gems: Towards an object-oriented framework. In *Proceedings of the Eight Annual European Conference on Object-Oriented Programming (ECOOP '94)* (Bologna, July 1994).

[25] 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.

[26] Sharma, S., Ponnusamy, R., Moon, B., Hwang, Y., Das, R., and Saltz, J. Run-time and compile-time support for adaptive irregular problems. In *Supercomputing '94* (Washington, DC, Nov 1994), pp. 97–106.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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).