Language and Run–time Support for Network Parallel Computing

Peter A. Dinda David R. O'Hallaron Jaspal Subhlok* Jon A. Webb Bwolen Yang

School of Computer Science
Carnegie Mellon University
Pittsburgh PA 15213

Abstract

Network parallel computing is the use of diverse computing resources interconnected by general purpose networks to run parallel applications. This paper describes NetFx, an extension of the Fx compiler system which uses the Fx model of task parallelism to distribute and manage computations across the sequential and parallel machines of a network. The central problem in compilation for network parallel computing is that the compiler is presented with a heterogeneous and dynamic target. Our approach is based on a novel run–time system that presents a simple communication interface to the compiler, but uses compiler knowledge to customize communication between tasks on heterogeneous machines over complex networks. The run–time system interface is sufficiently general to support different client compilers. The compiler generates code for data parallel modules with point–to–point transfer of distributed data sets between modules. This allows the compiler to be portable and enables communication generation without knowledge of exactly how the modules will be mapped at run–time, and what low level communication primitives will be used. The compiler also generates a set of custom routines, called address computation functions, to translate between different data distributions. The run–time system performs the actual communication using a mix of generic and custom address computation functions depending on run–time parameters like the type and number of nodes assigned to the communicating modules and the data distributions of the variables being communicated. This mechanism enables the run–time system to exploit compile–time optimizations, and enables the compiler to manage foreign modules that use non–standard data distributions. We describe the NetFx programming model, the communication interface, how NetFx programs are compiled to the interface, and the issues in implementing the interface.

1 Introduction

Network parallel computing is the use of diverse computing resources interconnected by general purpose networks to run parallel applications. The domain of effective network parallel applications is rapidly expanding because of considerable improvements in the latency and bandwidth characteristics of networks such as HiPPI and ATM. It is becoming practical to run applications such as scientific simulations, real-time applications and real-world modeling on network parallel computing environments. However, standard parallel languages like High Performance Fortran (HPF) [9] offer little support for network parallel computing. Further, multiple languages and programming paradigms may be necessary to fully realize the potential of the network parallel computing environment — yet there is little support for interoperability in today's parallel languages and paradigms. Without such support, the construction of an efficient, scalable, and environment–independent network parallel application is a difficult and error–prone task in which the applications programmer must wear many hats, including that of the network specialist.

The goal of our research is to make it easy to build network parallel applications that achieve high performance in many different environments. This paper presents a programming model for network parallel applications and the design of a compiler and a run–time system that enables efficient generation and execution of network parallel applications.

*Corresponding author. (jass@cs.cmu.edu , (412) 268 7893)

Our programming model allows the programmer to express task parallelism among parallel tasks. Tasks are instantiated as procedure calls. The programmer specifies the interface of the task, but may implement the task in any language. Tasks are grouped together into modules which are then mapped to the computing environment, either by the compiler or the programmer. The compiler generates communication between tasks to satisfy programmer–specified data dependencies between modules.

The most challenging part of compilation for a network is generating efficient communication. In a network parallel computing, multiple communication mechanisms must be supported for efficient communication. For example, crossing a network requires different mechanisms than communicating between the nodes of an MPP. More important, the network environment is expected to be dynamic. Run–time load balancing is important even for applications with a static computation structure since the external load on different machines can vary. This means the compiler has limited information about the run–time environment during code generation.

For generating efficient and portable communication in a dynamic run–time environment, it is important to distinguish between the aspects of the communication that are constant for the duration of the execution and those that are varying and may be known only at run–time. The communication steps between tasks are a property of the program and are determined at compile time, while the actual low–level communication calls may depend on the run–time environment.

Our solution to generating efficient communication involves defining an inter–module communication interface. This interface provides mechanisms for transferring data between modules that are mapped on groups of nodes. The compiler generates code for this communication interface and the run–time system ensures that appropriate communication calls are generated. The compiled program is portable across the machines and network protocols supported by the run–time system.

The key new idea is that the interface decouples the compilation from generation of low level communication which is essential for a network environment, yet provides a mechanism for the compiler to customize or extend the behavior of the run–time communication system. This mechanism allows the run–time system to exploit compile–time communication optimizations, and allows the run–time system to be painlessly extended to support new data distributions. The run–time system itself is not tied to the compiler, but may be used by other clients.

We are extending the Fx compiler system [7] to support the programming model and the run–time system. Fx already extends a data parallel language based on HPF with task parallelism. The extended version of Fx is called NetFx. We begin by motivating the need for network parallel computing with a set of example applications and then describe the NetFx language extensions, compiler strategy and the extensible run–time system.

2 Motivation for network parallel computing

The different components of a large application typically have different requirements for resources like I/O devices, sensors, processing power, communication bandwidth, memory, and software tools. Further, each component may be best expressed in a different language or paradigm. Attempting to satisfy all of these resource requirements in a single computing platform utilizing a single language can be impractical or even impossible. An attractive alternative is to build each component of the application using the appropriate language or programming paradigm and run it on the appropriate platform using high speed networks to communicate with other components. We refer to this type of computation as *network parallel computation.*

This section illustrates the motivation for network parallel computing with four applications: (1) a *real–world modeling* application where inputs are acquired from a camera system, processed on a large parallel system, and displayed remotely, (2) an *air quality modeling* application where different steps of the computation have different processing and communication requirements, (3) an *earthquake ground motion modeling* application where the results are computed on a large parallel system and visualized on a graphics workstation, and (4) a *real–time embedded system* constructed from commercially–available rack–mounted array processor boards.

2.1 Real–world modeling

High performance computers and their programming tools have traditionally been applied to problems in scientific modeling, including simulation analysis. A group of researchers at Carnegie Mellon is applying these same systems and tools to problems in *real–world modeling*, which means three–dimensional shape modeling of objects and environments in the real world by direct observations using computer vision techniques. Success of this approach could lead to, for example, replacement of complex and specialized sensory systems based on laser range–finding technology by much simpler, more flexible and scalable, but computationally more expensive, systems like stereo vision. In real–world modeling applications the sensor system plays a key role. In some situations (e.g., imaging moving objects) it is necessary to capture imagery at video rate (30 Hz) or faster. Capturing data at this high rate is complicated by the need to synchronously access several cameras[1]. In other situations (e.g., imaging building interiors) speed of image capture is not so important but the imaging system must be portable.

These conflicting requirements can be met through the application of modern networking technology. The system we envision is illustrated in Figure 1. The camera system is a detachable unit, and can be

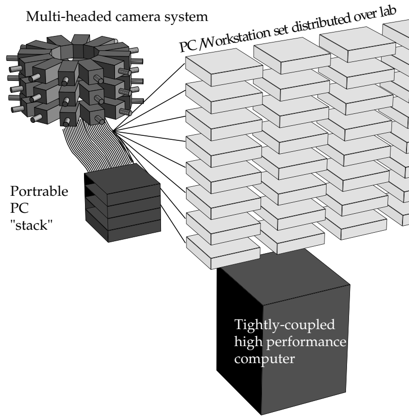

Figure 1: Real-world modeling system

attached either to a large number of workstations, giving a high frame capture rate, or to a smaller number of portable workstations or PCs, giving portability at the expense of speed. Once the images are captured, they are transformed into stereo vision data through the application of parallel stereo vision programmed in Fx, which can then be executed on the workstations themselves or on powerful tightly–coupled high performance computers. The resulting range imagery is then transformed into a three–dimensional model, which is another Fx program, and the resulting model can then be stored to disk or displayed remotely. NetFx implementation will allow us to program the entire framework including communication across the network in a uniform portable manner.

2.2 Air quality modeling

As part of a Grand Challenge grant on heterogeneous computing for large scale environmental modeling, researchers at Carnegie Mellon have adapted the airshed environmental model application [10] to run on parallel systems. The application consists of repeated application of three data parallel steps. The first step, reading input data from storage and preprocessing it, is I/O intensive. The second step, computing the interaction of different chemical species in the atmosphere, exhibits massive parallelism and little communication. The third step, computing the wind–blown transport of chemical species, has significant parallelism and communication. Each of these steps is suited to a different class of machine. For example, an Fx implementation of the chemistry step runs as fast on four Ethernet–connected DEC Alpha workstations as on 32 nodes of an Intel Paragon since the relatively slow communication is not

[1]In recent past, the most significant advances in stereo vision accuracy and reliability have come from the introduction of multiple cameras.

an important factor, but the transport step is expected to be more sensitive to the parameters of the communication system. Further, the input step is best run concurrently with the other stages. A NetFx implementation will allow us to use a single high level program for the entire application and the ability to use different mappings for execution.

2.3 Earthquake ground motion modeling

As part of a Grand Challenge grant on earthquake modeling, researchers at Carnegie Mellon are developing techniques for predicting the motion of the ground during strong earthquakes. The computation consists of two distinct steps: simulation and visualization. The simulation step uses a large, sparse, and irregular finite element model to generate a sequence of displacement vectors, one vector per time step. The visualization step manipulates the displacement vectors and produces a graphical display. The simulation step requires access to large amounts of memory, computational power, and low–latency high–bandwidth communications, while the visualization step requires access to a bit–mapped display and, for the more complex 3D models, a graphics accelerator. The natural partitioning of this application is to run the simulation step on a large tightly–coupled parallel system, and to run the visualization step on a graphics workstation.

2.4 Real–time embedded systems

Economic realities have created a pressing need for embedded systems that are built with commercial off–the–shelf components. The basic building block is typically a commercial array processor board that consists of a fixed number of processors connected by a fast onboard switch. For larger systems, multiple boards are mounted in a rack and connected by a bus or ring structure. For still larger systems, multiple racks are connected by a commercial network like FDDI or ATM.

These systems are used for real–time applications like sonar and radar, where the processing typically consists of reading from one or more sensors, and then manipulating the inputs in a sequence of stages. Each stage must be assigned enough processing resources to match the throughput and latency requirements of the overall system. An application that needs multiple racks to meet its real–time requirements will of necessity be a network parallel computation.

3 Our approach

Our programming model for network parallel computing environments exploits functional parallelism among parallel tasks. Because the machines in the environment are heterogeneous, tasks can be programmed using any language or programming paradigm in order to best exploit the hardware. The flow of data between tasks as well as the mapping of each task to the environment is described by a tasking program written in a specialized tasking language. The language is currently being defined. The tasking compiler combines the tasking program, and information about the tasks, the environment and standard support libraries to build an executable for the environment. The tasking compiler emits calls to a high performance run–time system customized for the environment in order to perform data transfers between tasks. The run–time system handles the low-level details of communication in a heterogeneous machine and network environment and can be extended by the tasking compiler, a language compiler or the programmer in order to specialize certain data transfers or to support data transfers that are not supported by the standard library. A task is free to perform local load balancing, moving data and computations between the machines allotted to it, but the run–time system is responsible for global load balancing by changing the machines allotted to a task. Figure 2 demonstrates how the system works.

3.1 NetFx approach

The NetFx programming model and compiler, described below, implements a limited subset of our approach. The programming model limits tasks to be Fx subroutines and restricts how they can be interconnected. However, heterogeneity is supported. The tasking language used is an extension of Fx parallel sections and and support for the language is embedded in the Fx compiler. NetFx uses the run–time

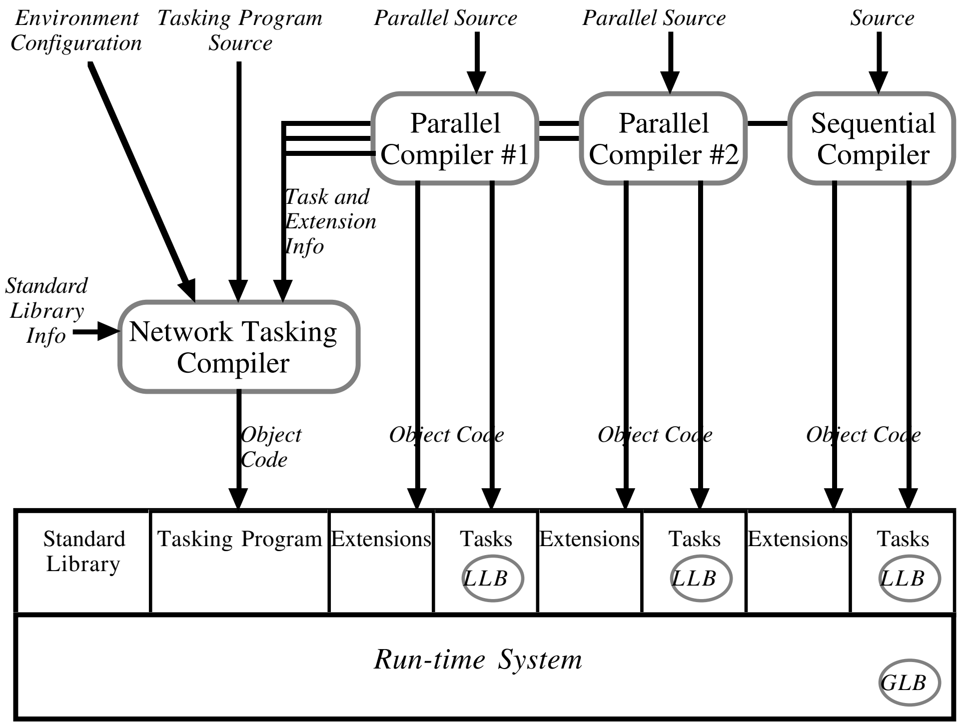

Figure 2: Overview of our approach

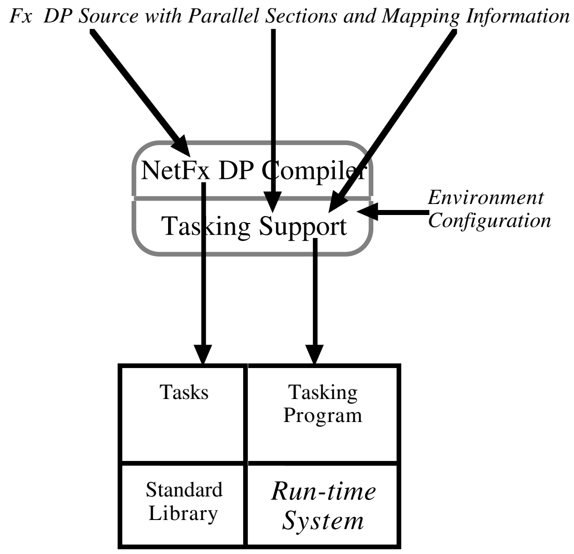

Figure 3: Overview of the NetFx approach

system described above, but no global (or local) load balancing is performed. Figure 3 demonstrates how the system works.

4 Programming model for NetFx

The initial implementation of the programming model is an extension of the Fx [7, 15] compiler system called NetFx. NetFx provides support for task and data parallelism but restricts language support to

Fx. Support for data parallelism is similar to that provided in High Performance Fortran (HPF) and is described in [13, 17]. Task parallelism is used to map data parallel subroutines onto different, possibly heterogeneous, groups of nodes. We outline how task parallelism is expressed in NetFx and illustrate how it is used to exploit coarse grain parallelism.

4.1 Task parallelism

Task parallelism is expressed solely using compiler directives and no new language extensions are introduced. To simplify the implementation, the current version of Fx relies on the user to identify the side effects of the task–subroutines and to specify them. Directives are also used to guide the compiler in mapping the program which is important for performance and also to ensure that tasks execute on computers for which the compiled versions are available. We describe the directives that are used to express task parallelism and illustrate their use. We use the FFT–Hist kernel as an example program. This program takes a sequence of arrays as input, and for each input array, it performs a 2D FFT by performing a set of 1D FFTs, first on columns and then on the rows, followed by histogram analysis.

Parallel sections

Calls to task–subroutines are permitted only in special code regions called *parallel sections*, denoted by a `begin parallel/end parallel` pair. For example, the parallel section for the FFT–Hist example has the following form:

```
c$      begin parallel
        do i = 1,m
           call colffts(A)
c$         input/output and mapping directives
           call rowffts(A)
c$         input/output and mapping directives
           call hist(A)
c$         input/output and mapping directives
        enddo
c$      end parallel
```

The code inside a parallel section can only contain loops and subroutine calls. These restrictions are necessary to make it possible to manage shared data and shared resources (including nodes) efficiently. Every task–subroutine call inside a parallel section corresponds to a (possibly data parallel) *task* that can execute in parallel with other tasks subject to dependence constraints.

A parallel section introduces a data parallel thread for each task–subroutine inside it. Outside the parallel section, the program executes as a single data parallel thread. The effect of executing a parallel section is the same as if its contents were executed sequentially.

Input/output directives

The user includes `input` and `output` directives to define the side–effects of a task–subroutine, i.e., the data space that the subroutine accesses and modifies. Every variable whose value at the call site may potentially be used by the called subroutine must be added to the input parameter list of the task–subroutine. Similarly, every variable whose value may be modified by the called subroutine must be included in the output parameter list. The variables in the input and output parameter lists can be distributed or replicated arrays, or scalars.

For example, the input and output directives for the call to `rowffts` have the form:

```
           call rowffts(A)
c$         input (A), output (A)
c$         mapping directives
```

This tells the compiler that the subroutine `rowffts` can potentially use values of, and write to the parameter array `A`. As another example, the input and output directives for the the call to `colffts` has the form:

```
            call colffts(A)
c$          output (A)
c$          mapping directives
```

This tells the compiler that subroutine `colffts` does not use the value of any parameter that is passed but can potentially write to array `A`

Mapping directives

Labeling task–subroutine calls and supplying input/output information is sufficient for the compiler to generate a correct parallel program. However, the performance of the program largely depends on how the task–subroutines are mapped onto the available computation resources on the network. The possible mappings are constrained by the availability of data parallel implementations of different task–subroutines for different architectures, the memory requirements of the task–subroutines, and the physical locations of devices like frame buffers. Ideally NetFx should be able to automatically map the program for correct execution and best possible performance, but that is a hard problem and NetFx is not able to do it except in some simple situations [16]. In general, the programmer has to supply the mapping information.

The mapping information specifies where each of the task–subroutines should execute. The details of the mapping directives are somewhat cumbersome and machine specific and we will not describe them here. For example, when mapping a task–subroutine to a network of workstations, it may be necessary to specify the names of the workstations to be used as a node, but when mapping to a part of a massively parallel machine like the Intel Paragon, it may be sufficient to specify the number of nodes to be used.

4.2 Programming coarse grain parallelism

NetFx allows the programmer to use fine grain data parallelism as well as coarse grain task parallelism. We illustrate how the task parallelism directives described above are used to exploit common forms of coarse grain parallelism.

Task pipelines

In image and signal processing programs , the main computation often consists of repeatedly executing an acyclic task graph. A simple example of such a program is the FFT–Hist example program, which we now present with directives:

```
c$        begin parallel
          do i = 1,m
            call colffts(A)
c$          output (A)
            call rowffts(A)
c$          input (A), output (A)
            call hist(A)
c$          input (A)
          enddo
c$        end parallel
```

In FFT–Hist, all data dependences in the program are forward and loop independent. Task–subroutine *colffts*, reads a data set from some input device, computes, and sends the output data set to *rowffts*, which receives the data set, computes, and sends the output data set to *hist*, which computes the final output data set. This allows an earlier stage, say *colffts* to process a new data set while *rowffts* is processing the previous data set, allowing coarse grain pipeline parallelism as shown in Figure 4. In related work, we have argued that such parallelism can be exploited efficiently and profitably for many applications including narrowband tracking radar and multibaseline stereo [14].

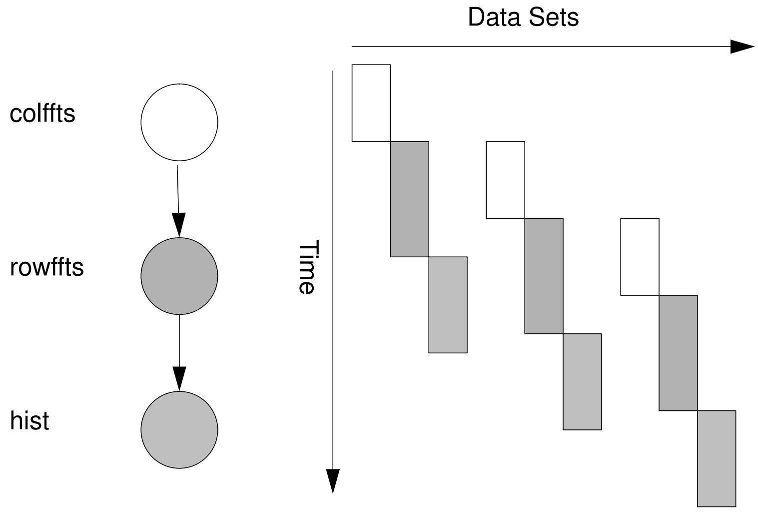

Figure 4: Dependences and coarse grain parallelism in FFT-Hist

Communicating tasks

In many scientific applications, different subroutines can execute in parallel, but need to transfer or exchange data between invocations. For example, two routines *fproca* and *fprocb* may work on different parts of a data structure and exchange boundary elements between successive invocations. A task parallel program that expresses this computation is as follows:

```
c$      begin parallel
        do i = 1,m
           call fproca(A, boundA, boundBx)
c$         output (boundA), input (boundBx)
           call fprocb(B, boundB, boundAx)
c$         output (boundB), input (boundAx)
           call ftransfer(boundA, boundB, boundAx, boundBx)
c$         input (boundA, boundB) output (boundAx, boundBx)
        enddo
c$      end parallel
```

We show the input/output directives only for the boundary elements for the purpose of illustration. The task dependence graph for this computation is shown in Figure 5. There is no direct dependence between task–subroutines *fproca* and *fprocb*, but they have a loop carried dependence on *ftransfer* due to variables *boundAx* and *boundBx* respectively. Task–subroutine *ftransfer* has a loop independent dependence on *fproca* and *fprocb*, due to variables *boundA* and *boundB*. Thus, the iterations of *fprocA* and *fprocB* execute in parallel and effectively exchange boundary elements using task-subroutine *ftransfer*. The computation schedule is illustrated in Figure 5. Note that the results obtained by task parallel execution are consistent with sequential execution.

4.3 Compilation

The compiler uses a data flow algorithm to analyze the dependences caused by input/output variables and precisely determines the data movement between tasks. The basic paradigm is that the results obtained must always be consistent with sequential execution. It then uses the mapping information to group task–subroutines into *modules*. All task–subroutines inside a module are mapped to the same set of nodes. A module repeatedly receives input, computes, and generates output.

The compiler generates calls to the run–time system for communication between modules, and may also generate address computation functions that the run–time system uses to translate between data

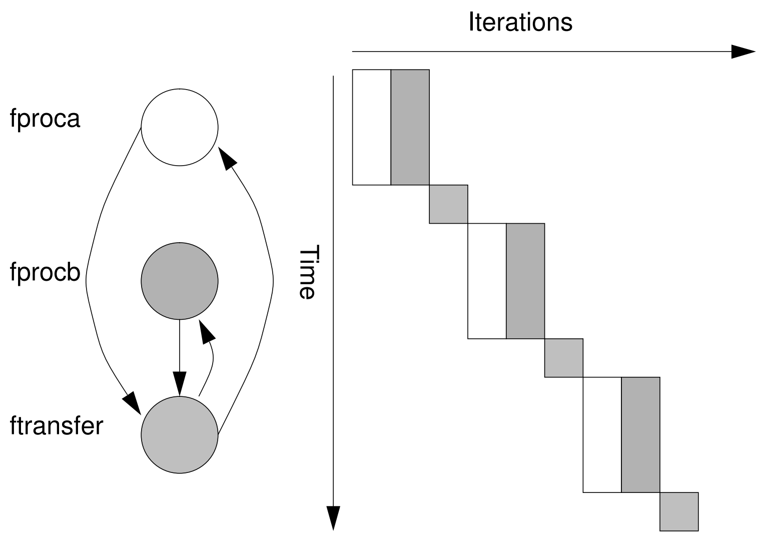

Figure 5: Dependences and coarse grain parallelism for a program with data exchange between parallel iterations

distributions. Inter–module communication management is done jointly by the compiler and the run–time system and the details are discussed in the next section. The involvement of the run–time system is necessary since the compiler may not know the mapping of the modules to the available nodes, the current distribution of data on a remote module, or the status of the (possibly general purpose) network at run-time. This information may change unpredictably for a variety of reasons, including load balancing, the use of non-NetFx task-subroutines, or other traffic on the network.

5 Run–time support for task parallelism

The main purpose of the run–time system is the efficient transfer of distributed data sets between modules with few constraints on processors, networks, languages, modules, or data. In this section, we first describe the design challenges of the run–time system and our approach to meeting them. Next, we explain the details of inter–module communication, and the programming interface exported to the compiler. This is followed by a discussion of the internal structure of the run–time system. Finally, we give an example of how an intermodule communication is accomplished using the run–time system.

5.1 Design Challenges

The main purpose of the run–time system is the efficient transfer of distributed data sets between modules. To meet this goal, several design challenges must be overcome. These include multiple languages and programming paradigms, dynamic program and data mappings, processor and network heterogeneity. Each of these challenges are discussed below.

Multiple languages and programming paradigms

Each task in the program may be written in a different parallel language or paradigm. There are two primary motivations for this. First, the heterogeneity of the network parallel computing environment means that no one language or paradigm is "best" for programming it. For example, the environment may include both shared memory and distributed memory computers. The second motivation is multi-disciplinary applications. Each component of a multidisciplinary application may be best expressed in a particular language or paradigm. For example, one component may require sparse matrix support such as Archimedes' [12], while another requires fast dense matrix support such as Fx's, and a third may be best expressed with nested data parallelism as in NESL [4].

Regardless of what the language or paradigm is used to implement the tasks, it should always be possible to transfer data between two tasks as long as the *abstract* data types are compatible. For example, it should be possible to send a vector from an HPF task to a DOME [1] task. The challenge is that the implementation vectors in HPF is very different from that in DOME. In general, if there are n implementations of an abstract data type, there are n^2 combinations of implementations that must be supported. Furthermore, it should be easy to extend the range of implementations.

Dynamic program and data mappings

In many languages and paradigms, the distribution of data can change dynamically. For example, DOME load balances vectors by shifting elements between processors. At another level, the assignment of processors to modules may change due to global load balancing or to a support a run–time automatic mapping tool. The run–time system should not restrict data transfer under these conditions.

Processor and network heterogeneity

Processor and network heterogeneity make efficient communication difficult. There are several reasons for this. First, processor heterogeneity may require that data representation be changed. Communication systems such as PVM [6] deal with this by converting to and from the XDR format, but this usually means additional copy steps. The second reason is that routing is complicated by an essentially arbitrary topology and many different link types and access mechanisms. A common solution is to rely on internet routing [11], but this mechanism cannot make use of the parallel nature of inter–module communications because there is no notion of a parallel connection in TCP/IP. Furthermore, the TCP/IP stack for a high speed network adaptor may be significantly less efficient than the native stack. Finally, MPPs typically do not support TCP/IP internally, resulting in a store–and–forward stop at an interface node, which increases latency.

5.2 Meeting the design challenges

Our approach to the design challenges described in section 5.1 has three components: a standard run–time system interface, run–time system extensibility, and run–time system modularity.

Standard run–time system interface

The run–time system exports a standard interface to the program generator. The main abstraction is that of communicating distributed data sets between modules.

Run–time system extensibility

To support multiple languages and paradigms, we provide a standard method for a compiler or programmer to extend the run–time system to support new combinations of data distributions. This is similar to how a generic sort routine allows an application to pass in `compare` and `swap` functions for a particular data type, but on a larger scale. This mechanism also provides a way for compile–time knowledge to be used by the run–time system for higher performance.

Run–time system modularity

The run–time system is composed of three subsystems with clearly defined interfaces between them. This allows each subsystem to be developed and optimized indepently of the others. The idea is that the communication subsystem, which actually performs (parallel) data transfers, is supplied with client subsystems that handle the details of mapping between data distributions and dynamic program and data mappings. This segregation of functionality opens the development of communication subsystems to the networking community. Conversely, the development of distribution mapping methods is simplified because communication is not part of the picture.

5.3 Details of Inter–module communication

The goal of inter–module communication is to transport a distributed data set from a source module to a destination module. The elements of the data set are partitioned across the processes of a module according to some rule. This rule is identified by the data set's distribution type and distribution. A *distribution type* identifies a class of similar rules, while a *distribution* identifies a specific member of the class. For example, the distribution type of HPF_ARRAY includes the distribution $(*, BLOCK)$. The run–time system assumes that the distribution type of the data set on each module is constant throughout the execution of the program, however its distribution may change. If the distribution can change, we say that its *binding* is *dynamic*, otherwise it is *static*. The initial value of both kinds of bindings are supplied at compile time.

To send the data set from a source module to a destination module, it is necessary to be able to translate between the distributions at the two end–points. It is only possible to translate between distributions whose global structures are compatible. For example, if the source global structure is an array, then the destination global structure must be an array of the same size and shape. The two end–point distribution types identify a method for translating between the two member distributions, a process called *address computation*. Given two distributions of the appropriate type and a process pair, the method, or *address computation function*, returns an *offset relation* between offsets into the chunk of elements owned by each process. This architecture–independent relation can be trivially converted to an architecture–dependent *address relation*, which maps memory addresses on one machine to memory addresses on the other.

The address computation function operates under the abstraction that a data set's elements are distributed over a consecutively numbered group of processes of the module. Once an address relation has been computed for each pair of processes, the abstract process numbers must be mapped to the actual nodes of the machines on the network. The properties of these module bindings are similar to those of the distribution bindings discussed above. Specifically, the bindings may be static or dynamic. Once the mappings are known, actual data transfer can begin.

5.4 Run–time System Interface

This section describes the programming interface between the program generator and the run–time system. It is a two way interface: the programmer generator emits calls to the run–time system, and the run–time system may itself call procedures produced by the program generator. The interface has three components as shown in Figure 6. The first two of these are exported by the run–time system, while the last is exported by the program itself.

Inter–module communication interface

This interface consists of simple, high–level, non–blocking collective send and receive calls for transporting whole distributed data sets between modules. It is required that if more than one data set is transferred between a pair of modules, the order of the sends and receives on the modules be the same. The arguments of the send and receive calls are

- Target (send) or source (receive) module
- Address of first element local to the the calling process
- Data type of each element
- Sender distribution type and distribution
- Receiver distribution type and distribution
- Hint (implementation specific)

There is a single call to wait for all outstanding receives to complete which allows a module to complete input communication before executing.

Registry interface

Before performing sends and receives, the program must describe the characteristics of relevant modules, distribution types, and custom address computation functions to the run–time system using the registry interface. The program explicitly registers the initial module mapping and whether it is static or dynamic. Registry of distribution type is required if the distribution type is not library-supported, or if the distribution is dynamic. Dynamic bindings are updated implicitly by examining send and receive calls. There are several reasons why a program may want to register its own distribution types. One reason is extensibility — the distribution type may not be supported by the library. Another is efficiency — the compiler may have been able to generate a specialized address computation function for a particular communication. In either case, the program must also register custom address computation functions for any communication that involves the new distribution type. For example, suppose module 1 communicates from a new distribution type DTX to predefined types $DT2$ on module 2 and $DT3$ on module 3. All three modules must register distribution type DTX. Module 1 must register new address computation functions $\lambda(DTX, DT2)$ and $\lambda(DTX, DT3)$, module 2 must register $\lambda(DTX, DT2)$, and module 3 must register $\lambda(DTX, DT3)$.

Standard address computation function interface

All address computation functions must conform to this interface. Each address computation function must be able to both return offset relations or assemble/disassemble message buffers. The run–time system decides which functionality is used.

5.5 Internal structure

The run–time system presents the abstraction of "module–to–module" communication of a distributed data set between uniquely labeled modules. This requires address computation, binding maintenance, and efficient communication over heterogeneous networks. These tasks are segregated into three subsystems (distribution, binding, and communication) with well defined interfaces between them. More than just good design practice, the ability to independently develop and enhance each subsystem is vital for extensibility and good performance. Clearly, the number of distribution types will grow over time as more languages and programming paradigms are developed and supported. Further, compile–time optimization of address computation is important for performance [13]. There are a variety of mechanisms that could be used to support dynamic bindings and it is unclear that any one is best for all environments. Finally, to achieve high performance, it will be necessary to specialize the communication system for different networks. By clearly defining the interfaces between subsystems, we make this specialization possible and easy. Indeed, all the subsystems are "plug–and–play." The internals of the run–time system are shown in Figure 6.

Communication subsystem

The communication subsystem is the heart of the run–time system and exports the previously discussed inter–module communication interface to the program. The subsystem drives the run–time system by making use of the binding subsystem to resolve module and distribution bindings and the distribution subsystem to compute address relations. In return, it supplies a simple *binding communication interface* to the binding subsystem. The communication subsystem performs all actual data transfer.

Binding subsystem

The binding subsystem resolves module, distribution, and address computation function bindings for the communication subsystem and the distribution subsystem through the *binding interface.* It uses the communication subsystem's binding communication interface if it needs to perform an inter–module communication step to to determine a binding (for example, to bind the distribution of an irregular data structure). The binding subsystem supplies the registry interface to the executable.

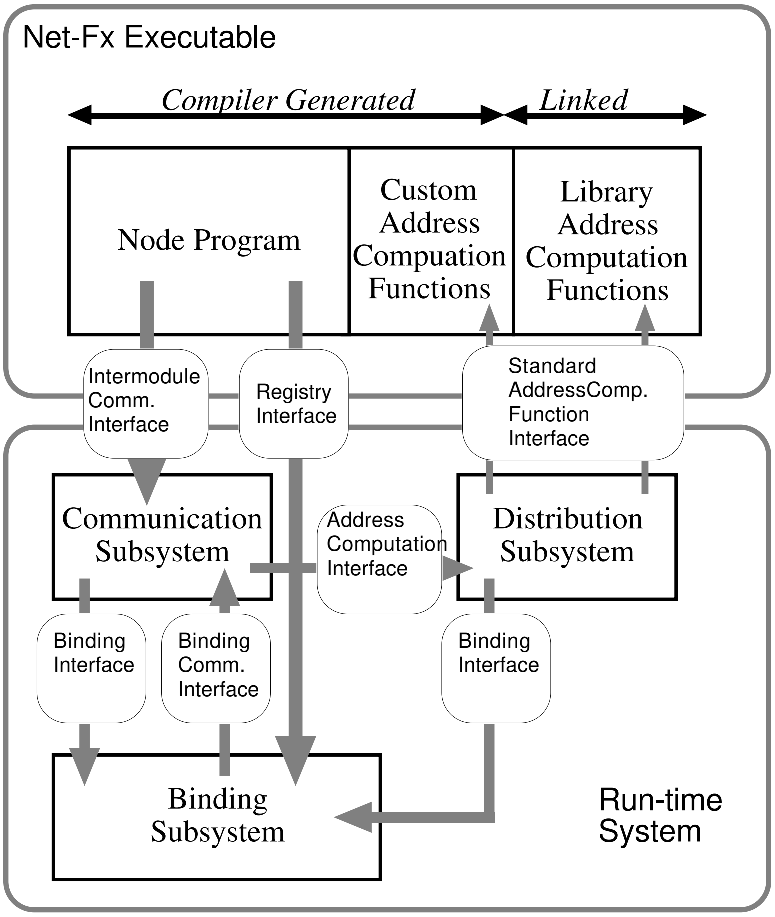

Figure 6: The run–time system: subsystems, interfaces, and relation to the NetFx Executable

Distribution subsystem

The distribution subsystem translates between sender and receiver distributions. It supplies the *address computation interface* to the communication system for this purpose. The binding subsystem is used to bind two distribution types and distributions to the appropriate address computation function. This function may be one of the *library address computation functions* linked to every NetFx Executable, or a compiler–generated *custom address computation function*. In every case, the address computation function must conform to the standard address computation function interface, discussed earlier. After finding the appropriate address computation function, it dispatches it, and conforms the resulting offset relation to the format desired by the communication subsystem and converts it to an address relation. (The communication system can request that a message be assembled or disassembled instead.)

5.6 Example

We illustrate how an inter–module send using a custom address computation function is performed. Figure 7 shows the steps graphically. On the right are the actual calls performed by the node program and those performed internally for the inter–module send. The calls are numbered to correspond to the interfaces used and to the steps discussed in this section. There is a dependence from module $M1$ to module $M2$ and the compiler has generated a send from $M1$ to $M2$ (and a matching receive in $M2$, which is not discussed here.) The compiler also produced a custom address computation function $DISTFUNC$ for this communication. At run time, $M1$'s first step is to use the registry interface to register the initial values of all relevant bindings. The module bindings for $M1$ and $M2$ are registered as static, as are

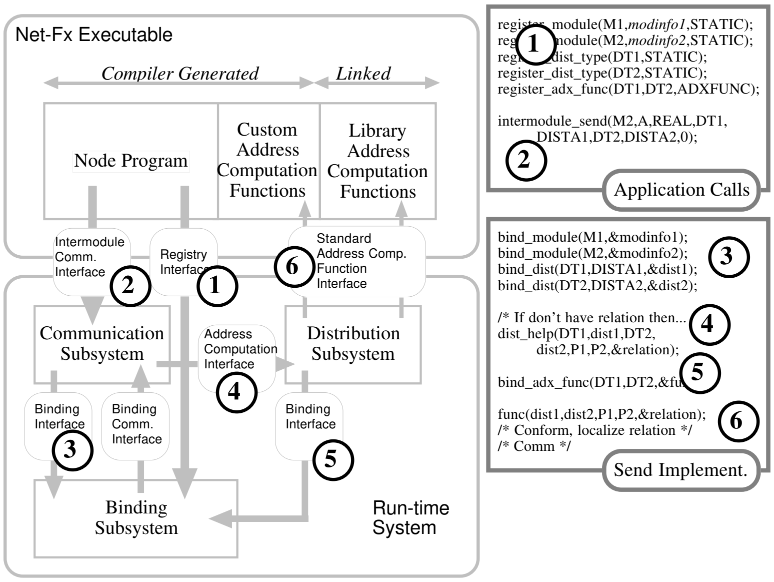

Figure 7: Example of the steps followed for an inter–module send using a custom address computation function.

the distribution types $DT1$ and $DT2$. Finally, $M1$ registers $ADXFUNC$ as the address computation function for $DT1$ and $DT2$.

The second step (which may be repeated many times) is to invoke a send of the distributed data set A to $M2$. The inter–module send call includes the target module number, $M2$, the first element of A that is local to the process, the data type of A's elements (REAL), the sender and receiver distribution types, $DT1$ and $DT2$, and the current sender and receiver distributions, $DISTA1$ and $DISTA2$. The third step is to bind the two modules to their current physical mappings, and the distribution types and distributions to their current distributions. Both of these steps are simple in this case because the bindings are static — they amount to local lookups. In the fourth step, the communication subsystem calls the distribution subsystem to build an address relation between each pair of sender and receiver processes $P1$ and $P2$. The distribution subsystem accomplishes this by binding the two distribution types, $DT1$ and $DT2$, to their corresponding address computation function, $ADXFUNC$ (step 5) and calling the function for each pair (step 6). The offset relations returned are conformed to the format the communication subsystem wanted and converted to address relations. Finally, the relations are used to perform the actual communication in a environment–dependent manner. The communication subsystem may decide to keep the address relations handy to avoid steps 4–6 the next time the inter–module send occurs.

6 Status and implementation

The Fx compiler currently supports task parallelism on the Intel iWarp, Intel Paragon, and on DEC Alpha workstations running PVM. Compiler analysis of parallel sections (although not mapping directive support or code emission) is the same for each target. The run–time system used on the Paragon and the Alphas was initially developed for the Alphas with an eye towards portability and serves as a prototype for the system discussed above. The prototype essentially has three components: a component that computes offset relations for any two (static) HPF distributions, a component that conforms these relations to a particular format and caches them, and a component that does communication using the relations. Porting this run–time system to the Paragon (and achieving good performance) required rewriting only the communication component. Further details of the library are available in [5].

The main target network for is Gigabit Nectar [2] developed at Carnegie Mellon. A variety of communication interfaces have been developed for Nectar and we are incorporating them in. Currently we are able to execute homogeneous Fx programs across Alpha workstations using Nectar as well as simple heterogeneous Fx programs on Alphas and Suns . The full heterogeneous run–time system described above is currently under development.

7 Related work

High level run–time support for communicating data parallel tasks is a fairly new research area. We compare our approach to two other run–time libraries that support communicating data parallel tasks. Opus [8] supports communication between tasks using *shared data abstractions* (SDAs). An SDA is essentially an independent object class whose private data is distributed over some set of threads and whose methods are protected with monitor semantics. For example, one could define a queue SDA whose elements are block–distributed vectors. To send a vector to another task, it is first sent to an SDA queue object (which requires a address computation) using the enqueue method. The receiving task would invoke the dequeue method which would cause the SDA queue object to send a data set (again requiring address computation computation). SDAs can support many more tasking paradigms than our scheme, but appears to significantly increase the communication volume and result in unnecessary synchronization between tasks due to monitor semantics.

Closer to our scheme is the library described in [3]. This library supports communication between data parallel tasks with collective sends and receives. Sends (and receives) are split between an initialization step which performs address computation and builds communication schedules, and an execution step that actually performs the send. Initializations synchronize sending and receiving tasks, but can be amortized over many executions. Only static module bindings and static HPF distributions are supported. All address computation is done with general purpose library routines.

8 Summary and conclusions

We address programming support for high performance networks which are emerging as an increasingly important form of parallel computing. We have presented NetFx, a compiler system that supports network parallel computing. The NetFx programming model is based on integrated support for task and data parallelism originally developed for the Fx compiler system. We also presented a run–time system for network parallel computing which presents a simple abstraction for module-to-module transfer of distributed data sets so that the compiler does not have to deal with the dynamic nature of the network computing environment. The run–time system allows the compiler and other tools that generate foreign data parallel modules to customize execution behavior. A clean interface between the compiler and the run–time system also allows independent development and retargetting of both. We believe that NetFx is the first tool to provide practical programming support for networks and will make network computing easier for non-specialists.

References

[1] Arabe, J., Beguelin, A., Lowekamp, B., E. Seligman, M. S., and Stephan, P. Dome: Parallel programming in a heterogeneous multi-user environment. Tech. Rep. CMU-CS-95-137, Carnegie Mellon University, April 1995.

[2] Arnould, E. A., Bitz, F. J., Cooper, E. C., Kung, H. T., Sansom, R. D., and Steenkiste, P. A. The design of nectar: A network backplane for heterogeneous multicomputers. In *Proceedings of the Third International Conference on Architecutural Support for Programming Languages and Operating Systems* (Apr. 1989), ACM.

[3] Avalini, B., Choudhary, A., Foster, I., Krishnaiyer, R., and Xu, M. A data transfer library for communicating data–parallel tasks. Case center tech report, Syracuse University, December 1995. To Appear.

[4] Blelloch, G. Nesl: A nested data-parallel language. CMU-CS 92-103, School of Computer Science, Carnegie Mellon University, 1992.

[5] Dinda, P. A., and O'Hallaron, D. R. The impact of address relation caching on the performance of deposit model communication. In *Third Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers* (1995). To appear.

[6] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Mancheck, R., and Sunderam, V. *PVM: Parallel Virtual Machine.* MIT Press, Cambridge, Massachusetts, 1994.

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

[8] 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).

[9] High Performance Fortran Forum. *High Performance Fortran Language Specification, Version 1.0*, May 1993.

[10] 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.

[11] Narten, T. Internet routing. In *Proceedings of SIGCOMM'89* (September 1989), Austin, TX.

[12] Shewchuk, J. R., and Ghattas, O. A compiler for parallel finite elemeent methods with domain–decomposed unstructured meshes. In *Proceedings of the Seventh International Conference on Domain Decomposition Methods in Scientific and Engineering Computing* (1994), College Park, PA.

[13] 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.

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

[15] Subhlok, J., Stichnoth, J., O'Hallaron, D., and Gross, T. Exploiting task and data parallelism on a multicomputer. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, CA, May 1993), pp. 13–22.

[16] 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). To appear.

[17] 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).