Language and Run–time Support for Network Parallel Computing

Peter A. Dinda David R. O'Hallaron Jaspal Subhlok
Jon A. Webb Bwolen Yang
{pdinda, droh, jass, webb+, bwolen}@cs.cmu.edu

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. A central problem in 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, yet uses compiler knowledge to customize communication between tasks executing over the network. The run–time system is designed to support complete applications developed with different compilers and parallel program generators. It presents a standard communication interface for point–to–point transfer of distributed data sets between tasks. This allows the compiler to be portable, and enables communication generation without knowledge of exactly how the tasks 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 tasks 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 tasks that use non–standard data distributions. We outline several important applications of network parallel computing and describe the NetFx programming model and run–time system.

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

This research was sponsored in part by the Advanced Research Projects Agency/CSTO monitored by SPAWAR under contract N00039-93-C-0152, in part by the National Science Foundation under Grant ASC-9318163, and in part by a grant from the Intel Corporation.

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 efficient network parallel applications. This paper presents a programming model, a compiler, and a run–time system for the efficient generation and execution of network parallel applications.

Our programming model allows the programmer to express task parallelism among parallel tasks. Tasks are invocations of parallel procedures, called task procedures. The programmer specifies the interface of each task procedure, but may implement it in any language. Tasks are mapped onto the computing environment either by the compiler or the programmer. The compiler generates communication between tasks to satisfy data dependencies.

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 HiPPI 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–task communication interface. This interface provides mechanisms for transferring data between tasks 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 it 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 kinds of 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 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. This is the essence of network parallel computing.

This section illustrates the motivation for network parallel computing with three 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, and (3) an *earthquake ground motion modeling* application where the results are computed on a large parallel system and visualized on a graphics workstation.

Many common network computing environments, such as heterogeneous workstation clusters, or embedded systems constructed from commercially–available rack–mounted array processor boards also motivate network parallel computing because exploiting them may demand multiple languages and programming paradigms.

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 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 it is necessary to capture imagery at high rates, while in others, speed is less important than the portability of the imaging system.

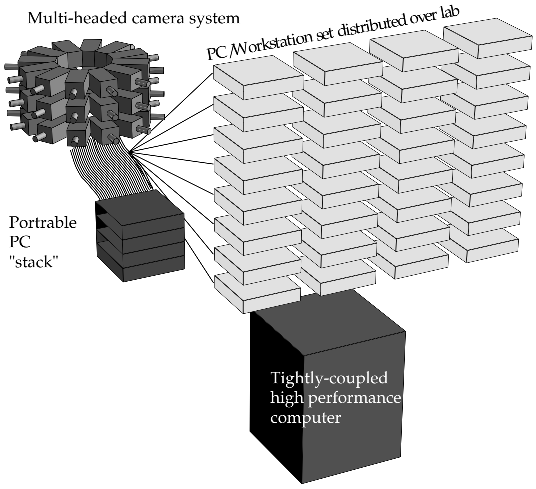

Fig. 1. Real-world modeling system

The system we envision is illustrated in Figure 1. The camera system is a detachable unit, and can be 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 HPF [9] or Fx [7], 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 will allow us to program the entire framework including communication across the network in a uniform portable manner.

2.2 Air quality modeling

Researchers at Carnegie Mellon developed a parallel implementation of the airshed environmental modeling application [14]. A goal of this work is to exploit heterogeneous 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 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

To address an earthquake modeling "grand challenge," 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.

3 Approach to network parallel computing

This section describes our approach to network parallel computing. Our goal is a compiler and run–time system that makes it possible to use multiple parallel languages and paradigms to effectively utilize the heterogeneous network parallel computing environment. The model of a network parallel application is a set of communicating tasks. Tasks are invocations of parallel task procedures, which may be written in any language. The data communicated between tasks is in the form of distributed data sets.

Figure 2 shows how a network parallel application is built. First, appropriate compilers and programming tools are used to generate task procedures and interoperability information. Each of these tools can also generate routines for a custom distribution management library, which will be discussed later. In addition to the task procedures, a tasking program, which describes how the tasks interact, is written. A special tool, the network task parallel compiler, combines the tasking program, task interoperability information, and information about the computing environment to produce a task management program, which is then linked to the task procedures, the custom distribution management library, a standard library, and the run–time system. The task management program orchestrates the different tasks of the application, and performs inter–task communication on their behalf by calling the run–time system.

Because both the distribution of data within a task and the mapping of tasks to nodes can be dynamic, much of the work involved in communicating distributed data sets between tasks is done by the run–time system, which presents the abstraction of a task–to–task communication of distributed data sets to the task management program. A major part of communicating distributed data sets is mapping from one distribution to another. While the standard library provides many methods for doing this, the run–time system can also exploit compile–time knowledge in the form of the custom distribution management library. This mechanism also allows the run–time system to be extended to support new kinds of data distributions. The run–time system is also responsible for global load balancing and miscellaneous utilities. The extensible run–time system for network parallel computing is the main contribution of this paper.

The run–time system is currently being used to support the NetFx compiler system, which can map data parallel Fx tasks onto different sequential and parallel machines over a network.

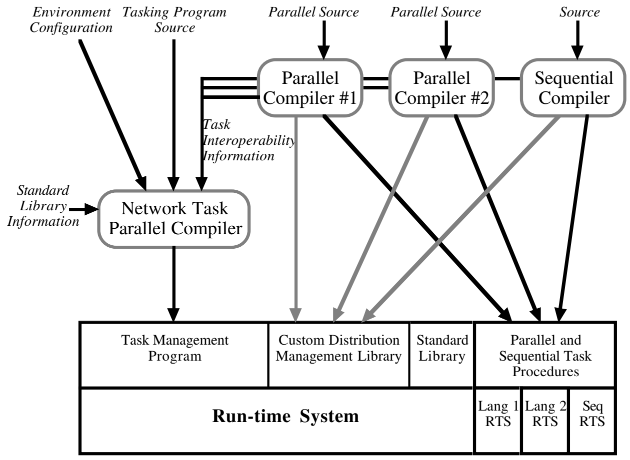

Fig. 2. Compilers and Run–time system for Network Parallel Computing

4 Programming model for NetFx

The current NetFx compiler uses the basic Fx programming model of integrated task and data parallelism [7, 18]. The main extension is that a data parallel task can be mapped to any sequential and parallel machines on a network. Support for data parallelism is similar to that provided in High Performance Fortran (HPF) and is described in [17, 20]. Task parallelism is used to map data parallel tasks onto different, possibly heterogeneous, groups of nodes. In the following, we briefly outline how task parallelism is expressed and compiled in NetFx.

4.1 Task parallelism

Task parallelism is expressed using compiler directives and no new language extensions are introduced. The current version of Fx relies on the user to identify and specify the side effects of the task procedures. Directives are also used to guide mapping of the tasks. This is important for performance and also to ensure that tasks execute on computers for which compiled versions are available. We describe the task parallelism directives and illustrate their use.

Parallel sections: Calls to task procedures are permitted only in special code regions called *parallel sections*, denoted by a `begin parallel/end parallel` pair. For example, the parallel section for a 2D FFT program 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
          enddo
c$        end parallel
```

The code inside a parallel section can only contain loops and task procedure calls. Every call to a task procedure inside a parallel section corresponds to a data parallel *task* that can execute in parallel with other tasks subject to dependence constraints. Outside the parallel section, the program executes as a single data parallel thread. The execution of a parallel section is consistent with the sequential execution of its contents.

Input/output directives: The user includes `input` and `output` directives to specify the side–effects of executing a task procedure, i.e., the data space that the procedure accesses and modifies.

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 task procedure `rowffts` can potentially use values of, and write to the parameter array `A`. Similarly, the input and output directives for the call to `colffts` have the form:

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

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

Mapping: Labeling task procedure 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 tasks 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 procedures for different architectures, the memory requirements of the task procedures, 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 relatively simple situations [19]. In general, the programmer has to supply the mapping information.

The mapping information specifies where each task executes. Mapping information is given for each call statement in a parallel section. Thus all executions of a particular task procedure call statement (i.e., each task associated with the statement) are mapped in the same way. Currently, the mapping information is static and all mapping is done at compile–time. However, the run–time system is designed to support dynamic task mapping as well. 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 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. For the purpose of this paper, we will assume that a mechanism for mapping any task to any group of available processors exists.

4.2 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 map each task procedure call to a set of nodes. At run–time, executing the call statement amounts to receiving input data sets, executing the task procedure (i.e., invoking a task), and sending output data sets. These steps are identical for compiling for a single massively parallel machine (as in Fx) or for a heterogeneous network (as in NetFx).

Currently, the NetFx programming model and compiler are the same as Fx: the placement of all tasks, the distribution of data inside each task, and the characteristics of the communication network are fixed and known at compile time. The programming model and compiler are being extended so a wider variety of of network parallel applications can be programmed efficiently. The NetFx run–time system is designed to address the realities of network parallel computing: the network is inherently a shared resource whose usage pattern changes dynamically, the mapping of the tasks can change at run–time, and data distributions may not be known at compile time. This dynamicity means the NetFx compiler does not have all the information to generate optimal inter–task communication. Instead, communication is managed jointly by the compiler and the run–time system. The compiler generates calls to the run–time system for communication between tasks, and may also generate address computation functions (section 5.3) that the run–time system uses to translate between data distributions. The details of the communication are managed by the run–time system, and are discussed in the next section.

5 Run–time support for task parallelism

The main function of the run–time system is the efficient transfer of distributed data sets between tasks with few constraints on processors, networks, languages, tasks, 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–task 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 inter–task communication step is accomplished using the run–time system.

5.1 Design challenges

The run–time system is designed for very general network computing, although it is presently being used mainly to manage data parallel NetFx tasks on a network. To meet this goal, several design challenges must be overcome. These include multiple languages and programming paradigms, dynamic program and data mappings, and processor and network heterogeneity. Each of these challenges is discussed separately.

Multiple languages and programming paradigms: Each task procedure in the program may be written in a different parallel language or paradigm. There are two major motivations for this. First, the heterogeneity of the network parallel computing environment means that no one language or paradigm is "best." For example, the environment may include both shared memory and distributed memory computers. The second motivation is multidisciplinary applications. Each component of a multidisciplinary application may be best expressed using a particular language or paradigm. For example, one component may require the sparse matrix support available in Archimedes [16], while another may require the fast dense matrix support available in Fx, and a third may be best expressed with nested data parallelism in NESL [4].

Regardless of what 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 Fx task, where it is implemented as a set of local Fortran subvectors distributed over the processors, to a DOME [1] task, where it is implemented as a distributed C++ object. The challenge is to translate between the two implementations. 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 tasks may change due to global load balancing or to support a run–time automatic mapping tool. The run–time system must support 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 the 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 (TCP/IP) routing [15], but this mechanism cannot make use of the parallel nature of inter–task communications because there is no notion of a parallel connection in TCP/IP. Furthermore, TCP/IP communication over a high speed network adaptor may be significantly less efficient than native communication. 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 these design challenges described in 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 program generators. The main abstraction is that of communicating distributed data sets between tasks mapped to groups of nodes.

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 better 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 independent of the others, and reduces overall complexity. For example, the communication subsystem, which actually performs parallel data transfers, can be developed independently by the network specialists since it is related to the other subsystems only through a well defined interface. Similarly, the development of distribution mapping methods is simplified because it is separated from actual data transfer.

5.3 Inter–task communication details

The goal of inter–task communication is to transport a distributed data set from a source task to a destination task. The elements of the data set are partitioned across the processes of a task 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 task is constant throughout the execution of the program, but its distribution may change. If the distribution can change, we say that its *binding* is *dynamic*, otherwise it is *static*.

To send a data set from a source task to a destination task, 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 abstract data types are compatible. For example, if the source abstract data type is an array, then the destination abstract data type must be an array of the same size and shape. The two end–point distribution types define a method for translating between the two member distributions. This method is called an *address computation function* and the process of executing it is *address computation*. Given two distributions of the appropriate type and a process pair, the address computation function returns an *offset relation* between offsets into the chunk of elements owned by each process, for corresponding global array elements. 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 task. 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 task 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 a program generator and the run–time system. It is a two way interface: the program 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 3. The first two of these are exported by the run–time system, while the third is exported by the program itself.

Inter–task communication interface: This interface consists of simple, high–level, non–blocking collective send and receive calls for transporting whole distributed data sets between tasks. It is required that if more than one data set is transferred between a pair of tasks, the order of the sends and receives on the tasks be the same. The arguments of the send and receive calls include: source or target task IDs, the address of the first data set element local to the calling process, the data type of each element, the sender and receiver distribution types and distributions, and an implementation specific hint. There is a single call to wait for all outstanding receives to complete, which allows a task to complete input communication before executing.

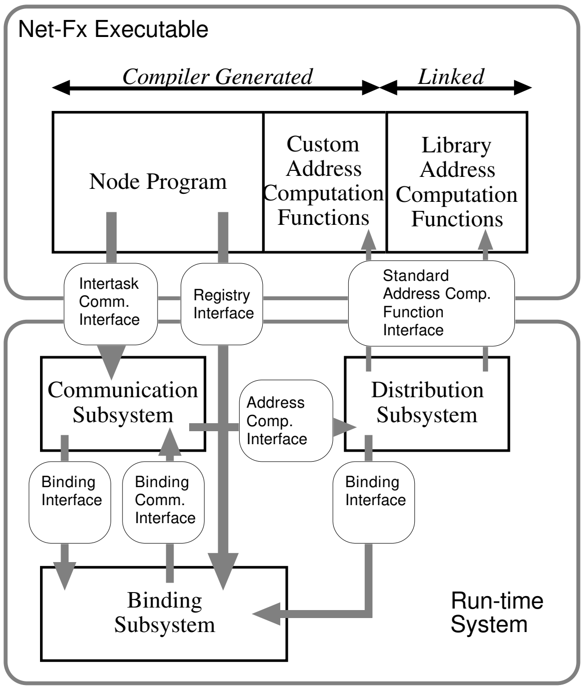

Fig. 3. The run–time system: subsystems, interfaces, and relation to the NetFx Executable

Registry interface: Before performing sends and receives, the program must describe the characteristics of relevant tasks, distribution types, and custom address computation functions to the run–time system using the registry interface. The program explicitly registers the initial task mapping and whether it is static or dynamic. Registration of a 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 two main 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 be able to generate a specialized address computation function for a particular communication step. In either case, the program must also register custom address computation functions for any communication that involves a new distribution type. For example, suppose task 1 communicates from a new distribution type DTX to predefined types $DT2$ on task 2 and $DT3$ on task 3. All three tasks must register distribution type DTX. Task 1 must register new address computation functions $\lambda(DTX, DT2)$ and $\lambda(DTX, DT3)$, task 2 must register $\lambda(DTX, DT2)$, and task 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 provide two functionalities. First, it must be able to generate an offset relation between a source and destination processor. Additionally, it should be able to directly assemble and disassemble messages between the two.

5.5 Internal structure

The run–time system presents the abstraction of inter–task communication of a distributed data set between uniquely labeled tasks. Such communication requires task and data distribution binding, to establish the current state of the computation, address computation, to determine what data to send and where to send it, and efficient low level communication over heterogeneous networks. These tasks are segregated into three subsystems — binding, distribution, and communication — with well defined interfaces between them. The intention is to decouple the implementations of the subsystems so that each can be specialized for particular network computing environments. For example, we would like to encourage the development of communication subsystems by members of the networking community. By encapsulating address computation and binding for them, we greatly simplify their task. The relationships of the subsystems are shown in Figure 3 and discussed below.

Communication subsystem: The communication subsystem is the heart of the run–time system and exports the previously discussed inter–task communication interface to the program. The subsystem drives the run–time system by making use of the binding subsystem to resolve task 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 task, distribution, and address computation function bindings and supplies them to the communication subsystem and the distribution subsystem through the *binding interface.* It uses the communication subsystem's binding communication interface if it is necessary to perform an inter–task communication step 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.

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 pairs of 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 either 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 directly assembled or disassembled instead.

5.6 Example

We illustrate how an inter–task send using a custom address computation function is performed. Figure 4 shows the steps graphically using figure 3. On the right are the actual calls performed by the node program and those performed internally for the inter–task send. The calls are numbered to correspond to the steps discussed in this section. There is a dependence from task $T1$ to task $T2$ and the compiler has generated a send from $T1$ to $T2$ (and a matching receive in $T2$, which is not discussed here.) The compiler also produced a custom address computation function $ADXFUNC$ for this communication. At run–time, $T1$'s first step is to use the registry interface to register the initial values of all relevant bindings. The task bindings for $T1$ and $T2$ are registered as static, as are the distribution types $DT1$ and $DT2$. Finally, $T1$ registers $ADXFUNC$ as the address computation function for $DT1$ and $DT2$.

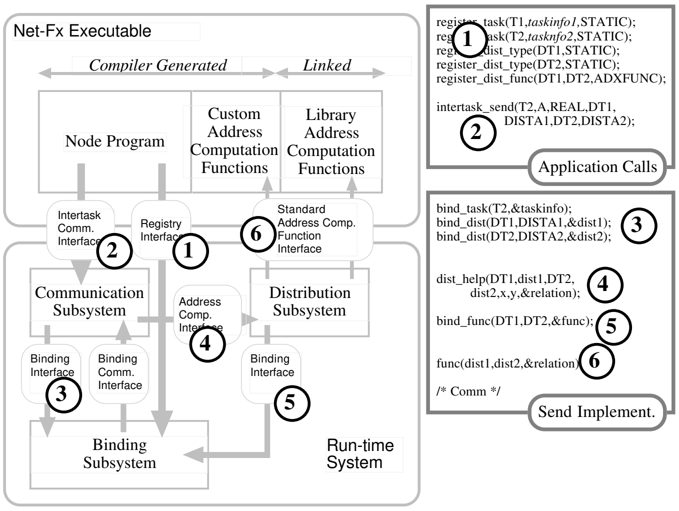

Fig. 4. Example of the steps followed for an inter–task send using a custom address computation function.

The second step (which may be repeated many times) is to invoke a send of the distributed data set A to $T2$. The inter–task send call includes the target task number, $T2$, 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 tasks 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 cache the address relations to avoid steps 4–6 in future inter–task sends.

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 pair of (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 NetFx is Gigabit Nectar [2] developed at Carnegie Mellon. A variety of communication interfaces have been developed for Nectar and we are incorporating them in the run–time system. Currently we are able to execute heterogeneous 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 in the paper is currently under development.

7 Related work

An overview of the issues involved in heterogeneous computing, especially compiler–based application mapping, is provided in [12]. In [13], the authors identify four goals for extending compilers to support and exploit heterogeneity and review six compilers using these criteria.

High level run–time support for communicating data parallel tasks is a fairly new research area. Most comparable to our work is the Schooner system [10]. Schooner combines a type specification language, a heterogeneous RPC mechanism, and a run–time system to permit procedures written in arbitrary languages to interact in a heterogeneous machine environment. A parallel procedure must be encapsulated in a sequential procedure, which acts as a proxy for communication, therefore Schooner can not perform parallel data transfers like our system. Also, the Schooner run–time system is not extensible and so cannot take advantage of compiler knowledge.

The goal of Converse [11] is to allow tasks written in different parallel languages to be linked together. Language run–time systems implemented on top of Converse can coexist. The core functionality is based on message–driven execution: run–time systems register handlers that are executed when properly tagged messages arrive. A well defined machine interface layer makes Converse itself highly portable. We feel that Converse and our system are orthogonal.

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 a 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 an 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 more general tasking paradigms than our scheme, but appear to significantly increase the communication volume and result in unnecessary synchronization between tasks due to monitor semantics.

The library described in [3] 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. An initialization step synchronizes sending and receiving tasks, and can be amortized over many executions. Only static task 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. The NetFx run–time system presents a simple abstraction for task-to-task 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 data parallel tasks 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. HOMER, P. T., AND SCHLICHTING, R. D. A software platform for constructing scientific applications from heterogeneous resources. *Journal of Parallel and Distributed Computing 21*, 3 (June 1994), 301–315.
11. KALE, L. V., BHANDARKAR, M., JAGATHESAN, N., AND KRISHNAN, S. Converse: An interoperable framework for parallel programming. Tech. rep., University of Illinois, 1995. http://charm.cs.uiuc.edu/research/interop.html.
12. KHOKHAR, A., PRASANNA, V., SHAABAN, M., AND WANG, C. Heterogeneous computing: Challenges and opportunities. *Computer 26*, 6 (June 1993), 18–27.
13. MCKINLEY, K. S., SINGHAI, S. K., WEAVER, G. E., AND WEEMS, C. C. Compiler architectures for heterogeneous systems. In *Eighth International Workshop on Languages and Compilers for for Parallel Computing* (1995). To appear.
14. 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.
15. NARTEN, T. Internet routing. In *Proceedings of SIGCOMM'89* (September 1989), Austin, TX.
16. 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.
17. 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.
18. 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.
19. 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.
20. 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).