The Nexus Task-parallel Runtime System*

Ian Foster
Math & Computer Science
Argonne National Laboratory
Argonne, IL 60439
foster@mcs.anl.gov

Carl Kesselman
Beckman Institute
Caltech
Pasadena, CA 91125
carl@compbio.caltech.edu

Steven Tuecke
Math & Computer Science
Argonne National Laboratory
Argonne, IL 60439
tuecke@mcs.anl.gov

Abstract

A runtime system provides a parallel language compiler with an interface to the low-level facilities required to support interaction between concurrently executing program components. Nexus is a portable runtime system for task-parallel programming languages. Distinguishing features of Nexus include its support for multiple threads of control, dynamic processor acquisition, dynamic address space creation, a global memory model via interprocessor references, and asynchronous events. In addition, it supports heterogeneity at multiple levels, allowing a single computation to utilize different programming languages, executables, processors, and network protocols. Nexus is currently being used as a compiler target for two task-parallel languages: Fortran M and Compositional C++. In this paper, we present the Nexus design, outline techniques used to implement Nexus on parallel computers, show how it is used in compilers, and compare its performance with that of another runtime system.

1 Introduction

Compilers for parallel languages rely on the existence of a runtime system. The runtime system defines the compiler's view of a parallel computer: how computational resources are allocated and controlled and how parallel components of a program interact, communicate and synchronize with one another.

Most existing runtime systems support the single-program, multiple-data (SPMD) programming model used to implement data-parallel languages. In this model, each processor in a parallel computer executes a copy of the same program. Processors exchange data and synchronize with each other through calls to the runtime library, which typically is designed to optimize collective operations in which all processors communicate at the same time, in a structured fashion. A major research goal in this area is to identify common runtime systems that can be shared by a variety of SPMD systems.

Task-parallel computations extend the SPMD programming paradigm by allowing unrelated activities to take place concurrently. The need for task parallelism arises in time-dependent problems such as discrete-event simulation, in irregular problems such as sparse matrix problems, and in multidisciplinary simulations coupling multiple, possibly data-parallel, computations. Task-parallel programs may dynamically create multiple, potentially unrelated, threads of control. Communication and synchronization are between threads, rather than processors, and can occur asynchronously among any subset of threads and at any point in time. A compiler often has little global information about a task-parallel computation, so there are few opportunities for exploiting optimized collective operations.

The design of Nexus is shaped both by the requirements of task-parallel computations and by a desire to support the use of heterogeneous environments, in which heterogeneous collections of computers may be connected by heterogeneous networks. Other design goals include efficiency, portability across diverse systems, and support for interoperability of different compilers. It is not yet clear to what extent these various goals can be satisfied in a single runtime system: in particular, the need for efficiency may conflict with the need for portability and heterogeneity. Later in this paper, we present some preliminary performance results that address this question.

*This work was supported by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38, and by the National Science Foundation's Center for Research in Parallel Computation under Contract CCR-8809615.

As we describe in this paper, Nexus is already in use as a compiler target for two task-parallel languages: Fortran M [7] (FM) and Compositional C++ [3] (CC++). Our initial experiences have been gratifying in that the resulting compilers are considerably simpler than earlier prototypes that did not use Nexus services.

Space does not permit a detailed discussion of related work. However, we note that the Chant system [9] has similar design goals (but adopts different solutions).

2 Nexus Design and Implementation

Before describing the Nexus interface and implementation, we review the requirements and assumptions that motivated the Nexus design.

Nexus is intended as a *general-purpose runtime system* for task-parallel languages. While it currently contains no specialized support for data parallelism, data-parallel languages such as pC++ and HPF can in principle also use it as a runtime layer. Nexus is designed specifically as a *compiler target*, not as a library for use by application programmers. Consequently, the design favors efficiency over ease of use.

We believe that the future of parallel computing lies in *heterogeneous environments* in which diverse networks and communications protocols interconnect PCs, workstations, small shared-memory machines, and large-scale parallel computers. We also expect *heterogeneous applications* combining different programming languages, programming paradigms, and algorithms to become widespread.

Nexus abstractions need to be close to the hardware, in order to provide *efficiency* on machines that provide appropriate low-level support. Operations that occur frequently in task-parallel computations, such as thread creation, thread scheduling, and communication, need to be particularly fast. At the same time, Nexus abstractions must be easily layered on top of existing runtime mechanisms, so as to provide *portability* to machines that do not support Nexus abstractions directly. Communication mechanisms that were considered in designing Nexus include message passing, shared memory, distributed shared memory, and message-driven computation.

Finally, Nexus is intended to be a *lingua franca* for compilers, promoting reuse of code between compiler implementation as well as *interoperability* between code generated by different compilers.

Important issues purposefully not addressed in the initial design include reliability and fault tolerance, real-time issues, global resource allocation, replication, data and code migration, and scheduling policies. We expect to examine these issues in future research.

2.1 Core Abstractions

The Nexus interface is organized around five basic abstractions: nodes, contexts, threads, global pointers, and remote service requests. The associated services provide direct support for light-weight threading, address space management, communication, and synchronization [8]. A computation consists of a set of *threads*, each executing in an address space called a *context*. An individual thread executes a sequential program, which may read and write data shared with other threads executing in the same context. It can also generate asynchronous *remote service requests*, which invoke procedures in other contexts.

Nodes. The most basic abstraction in Nexus is that of a *node*. A node represents a physical processing resource. Consequently, the set of nodes allocated by a program determines the total processing power available to that computation. When a program using Nexus starts, an initial set of nodes is created; nodes can also be added or released dynamically. Programs do not execute directly on a node. Rather, as we will discuss below, computation takes place in a context, and it is the context that is mapped to a node.

Nexus provides a set of routines to create nodes on named computational resources, such as a symmetric shared-memory multiprocessor or a processor in a distributed-memory computer. A node specifies only a computational resource and does not imply any specific communication medium or protocol. This naming strategy is implementation dependent; however, a node can be manipulated in an implementation-independent manner once created.

Contexts. Computation takes place within an object called a *context*. Each context relates an executable code and one or more data segments to a node. Many contexts can be mapped onto a single node. Contexts cannot be migrated between nodes once created.

Contexts are created and destroyed dynamically. We anticipate context creation occurring frequently: perhaps every several thousand instructions. Consequently, context creation should be inexpensive: certainly less expensive then process creation under Unix. This is feasible because unlike Unix processes,

contexts do not guarantee protection. We note that the behavior of concurrent I/O operations within contexts is currently undefined.

Compiler-defined initialization code is executed automatically by Nexus when a context is created. Once initialization is complete, a context is inactive until a thread is created by an explicit remote service request to that context. The creation operation is synchronized to ensure that a context is not used before it is completely initialized. The separation of context creation and code execution is unique to Nexus and is a direct consequence of the requirements of task parallelism. All threads of control in a context are equivalent, and all computation is created asynchronously.

Threads. Computation takes place in one or more *threads* of control. A thread of control must be created within a context. Nexus distinguishes between two types of thread creation: within the same context as the currently executing thread and in a different context from the currently executing thread. We discuss thread creation between contexts below.

Nexus provides a routine for creating threads within the context of the currently executing thread. The number of threads that can be created within a context is limited only by the resources available. The thread routines in Nexus are modeled after a subset of the POSIX thread specification [10]. The operations supported include thread creation, termination, and yielding the current thread. Mutexes and condition variables are also provided for synchronization between threads within a context.

Basing Nexus on POSIX threads was a pragmatic choice: because most vendors support POSIX threads (or something similar), it allows Nexus to be implemented using vendor-supplied thread libraries. The drawback to this approach is that POSIX was designed as an application program interface, with features such as real-time scheduling support that may add overhead for parallel systems. A lower-level interface designed specifically as a compiler target would most likely result in better performance [1] and will be investigated in future research.

To summarize, the mapping of computation to physical processors is determined by both the mapping of threads to contexts and the mapping of contexts to nodes. The relationship between nodes, contexts, and threads is illustrated in Fig. 1.

Global Pointers. Nexus provides the compiler with a global namespace, by allowing a global name to be created for any address within a context. This

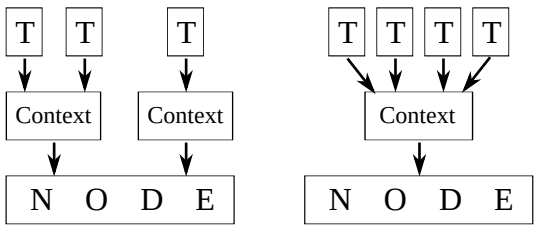

Figure 1: *Nodes, Contexts, and Threads*

name is called a *global pointer*. A global pointer can be moved between contexts, thus providing for a movable intercontext reference. Global pointers are used in conjunction with remote service requests to cause actions to take place on a different context. The use of global pointers was motivated by the following considerations.

- While the data-parallel programming model naturally associates communication with the section of code that generates or consumes data, task-parallel programs need to associate the communication with a specific data structure or a specific piece of code. A global namespace facilitates this.
- Dynamic behaviors are the rule in task-parallel computation. References to data structures need to be passed between contexts.
- Data structures other than arrays need to be supported. A general global pointer mechanism facilitates construction of complex, distributed data structures.
- Distributed-memory computers are beginning to provide direct hardware support for a global shared namespace. We wanted to reflect this trend in Nexus.

Global pointers can be used to implement data structures other than C pointers. For example, the FM compiler uses them to implement channels.

Remote Service Requests. A thread can request that an action be performed in a remote context by issuing a *remote service request*. A remote service request results in the execution of a special function, called a *handler*, in the context pointed to by a global pointer. The handler is invoked asynchronously in that context; no action, such as executing a receive,

needs to take place in the context in order for the handler to execute. A remote service request is not a remote procedure call, because there is no acknowledgement or return value from the call, and the thread that initiated the request does not block.

Remote service requests are similar in some respects to active messages [11]. They also differ in significant ways, however. Because active message handlers are designed to execute within an interrupt handler, there are restrictions on the ways in which they can modify the environment of a node. For example, they cannot call memory allocation routines. While these restrictions do not hinder the use of active messages for data transfer, they limit their utility as a mechanism for creating general threads of control. In contrast, remote service requests are more expensive but less restrictive. In particular, they can create threads of control, and two or more handlers can execute concurrently.

During a remote service request, data can be transferred between contexts by the use of a *buffer*. Data is inserted into a buffer and removed from a buffer through the use of packing and unpacking functions similar to those found in PVM and MPI [5, 6]. Invoking a remote service request is a three-step process:

1. The remote service request is initialized by providing a global pointer to an address in the destination context and the identifier for the handler in the remote context. A buffer is returned from the initialization operation.

2. Data to be passed to the remote handler is placed into the buffer. The buffer uses the global pointer provided at initialization to determine if any data conversion or encoding is required.

3. The remote service request is performed. In performing the request, Nexus uses the global pointer provided at initialization to determine what communication protocols can be used to communicate with the node on which the context resides.

The handler is invoked in the destination context with the local address component of the global pointer and the message buffer as arguments. In the most general form of remote service request, the handler runs in a new thread. However, a compiler can also specify that a handler is to execute in a preallocated thread if it knows that that handler will terminate without suspending. This avoids the need to allocate a new thread; in addition, if a parallel computer system allows handlers to read directly from the message interface, it avoids the copying to an intermediate buffer that would otherwise be necessary for thread-safe execution. As an example, a handler that implements the get and put operations found in Split-C [4] can take advantage of this optimization.

2.2 Implementation

In order to support heterogeneity, the Nexus implementation encapsulates thread and communication functions in thread and protocol modules, respectively, that implement a standard interface to low-level mechanisms (Fig. 2). Current thread modules include POSIX threads, DCE threads, C threads, and Solaris threads. Current protocol modules include local (intracontext) communication, TCP socket, and Intel NX message-passing. Protocol modules for MPI, PVM, SVR4 shared memory, Fiber Channel, IBM's EUI message-passing library, AAL-5 (ATM Adaptation Layer 5) for Asynchronous Transfer Mode (ATM), and the Cray T3D's get and put operations are planned or under development.

More than one communication mechanism can be used within a single program. For example, a context A might communicate with contexts B and C using two different communication mechanisms if B and C are located on different nodes. This functionality is supported as follows. When a protocol module is initialized, it creates a table containing the functions that implement the low-level interface and a small descriptor that specifies how this protocol is to be used. (Protocol descriptors are small objects: typically 4-5 words, depending on the protocol.) When a global pointer is created in a context, a list of descriptors for the protocols supported by this context is attached to the global pointer. The protocol descriptor list is part of the global pointer and is passed with the global pointer whenever it is transferred between contexts. A recipient of a global pointer can compare this protocol list with its local protocols to determine the best protocol to use when communicating on that global pointer.

Although some existing message-passing systems support limited network heterogeneity, none do so with the same generality. For example, PVM3 allows processors in a parallel computer to communicate with external computers by sending messages to the `pvmd` daemon process which acts as a message forwarder [5]. However, this approach is not optimal on machines such as the IBM SP1 and the Intel Paragon, whose nodes are able to support TCP directly, and it limits PVM programs to using just one protocol in addition to TCP. P4 has several special multiprotocol implementations, such as a version for the Paragon

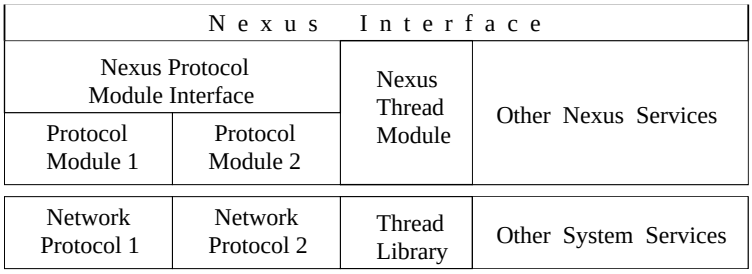

Figure 2: *Structure of Nexus Implementation*

that allows the nodes to use both NX and TCP [2]. But it does not allow arbitrary mixing of protocols.

3 Performance Studies

In this section, we present results of some preliminary Nexus performance studies. We note that the thrust of our development effort to date has been to provide a correct implementation of Nexus. No tuning or optimization work has been done at all. In addition, the operating system features used to implement Nexus are completely generic: we have not exploited even the simplest of operating system features, such as nonblocking I/O. Consequently, the results reported here should be viewed as suggestive of Nexus performance only, and are in no way conclusive.

The experiments that we describe are designed to show the cost of the Nexus communication abstraction as compared to traditional send and receive. Because Nexus-style communication is not supported on current machines, Nexus is implemented with send and receive. Thus, Nexus operations will have overhead compared to using send and receive. Our objective is to quantify this overhead. We note that support for Nexus can be build directly into the system software for a machine, in which case Nexus performance could meet or even exceed the performance of a traditional process-oriented send and receive based system. (We have started a development effort with the IBM T.J. Watson Research Center to explore this possibility.)

The experiments reported here compare the performance of a CC++ program compiled to use Nexus and a similar C++ program using PVM [5] for communication. The CC++ program uses a function call through a CC++ global pointer to transfer an array of double-precision floating-point numbers between two processor objects (Nexus contexts). We measure the cost both with remote thread creation and when a preallocated thread is used to execute the remote service request. The PVM program uses send and receive to transfer the array. Both systems are compiled with `-O3` using the Sun unbundled C and C++ compilers; neither performs data conversion. In both cases the data starts and finishes in a user-defined array. This array is circulated between the two endpoints repeatedly until the accumulated execution time is sufficient to measure accurately. Execution time is measured for a range of array sizes. The results of these experiments are summarized in Fig. 3.

We see that despite its lack of optimization, Nexus is competitive with PVM. Execution times are consistently lower by about 15 per cent when remote service requests are executed in a preallocated thread; this indicates that both latency and per-word transfer costs are lower. Not surprisingly, execution times are higher when a thread is created dynamically: by about 40 per cent for small messages and 10 to 20 per cent for larger messages.

4 Summary

Nexus is a runtime system for compilers of task-parallel programming languages. It provides an integrated treatment of multithreading, address space management, communication, and synchronization and supports heterogeneity in architectures and communication protocols.

Nexus is operational on networks of Unix workstations communicating over TCP/IP networks, the IBM SP1, and the Intel Paragon using NX; it is being ported to other platforms and communication protocols. Nexus has been used to implement two very different task-parallel programming languages: CC++ and Fortran M. In both cases, the experience with the basic abstractions has been positive: the overall

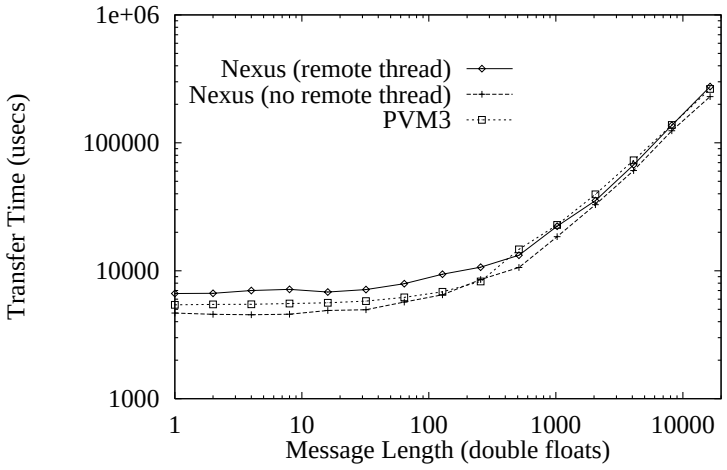

Figure 3: *Round-trip time as a function of message size between two Sun 10 workstations under Solaris 2.3 using an unloaded Ethernet.*

complexity of both compilers was reduced considerably compared to earlier prototypes that did not use Nexus facilities. In addition, we have been able to reuse code and have laid the foundations for interoperability between the two compilers. The preliminary performance studies reported in this paper suggest that Nexus facilities are competitive with other runtime systems.

References

[1] Peter Buhr and R. Stroobosscher. The μsystem: Providing light-weight concurrency on shared-memory multiprocessor systems running Unix. *Software Practice and Experience*, pages 929–964, September 1990.

[2] R. Butler and E. Lusk. Monitors, message, and clusters: The p4 parallel programming system. *Parallel Computing (to appear)*, 1994.

[3] K. Mani Chandy and Carl Kesselman. CC++: A declarative concurrent object oriented programming notation. In *Research Directions in Object Oriented Programming*. MIT Press, 1993.

[4] David Culler et al. Parallel programming in Split-C. In *Proc. Supercomputing '93*. ACM, 1993.

[5] J. Dongarra, G. Geist, R. Manchek, and V. Sunderam. Integrated PVM framework supports heterogeneous network computing. In *Computers in Physics*, April 1993.

[6] Message Passing Interface Forum. Document for a standard messge-passing interface, March 1994. (available from netlib).

[7] Ian Foster and K. Mani Chandy. Fortran M: A language for modular parallel programming. *J. Parallel and Distributed Computing*, 1994. to appear.

[8] Ian Foster, Carl Kesselman, Robert Olson, and Steve Tuecke. Nexus: An interoperability toolkit for parallel and distributed computer systems. Technical Report ANL/MCS-TM-189, Argonne National Laboratory, 1994.

[9] M. Haines, D. Cronk, and P. Mehrotra. On the design of Chant: A talking threads package. Technical Report 94-25, ICASE, 1994.

[10] IEEE. Threads extension for portable operating systems (draft 6), February 1992.

[11] Thorsten von Eicken, David Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *Proc. 19th Int'l Symposium on Computer Architecture*, May 1992.