Data Movement and Control Substrate for parallel scientific computing*

Nikos Chrisochoides[1]**, Induprakas Kodukula[1] and Keshav Pingali[1]

Computer Science Department
Cornell University, Ithaca, NY 14853-3801

Abstract. In this paper, we describe the design and implementation of a data-movement and control substrate (DMCS) for network-based, homogeneous communication within a single multiprocessor. DMCS is an implementation of an API for communication and computation that has been proposed by the PORTS consortium. One of the goals of this consortium is to define an API that can support heterogeneous computing without undue performance penalties for homogeneous computing. Preliminary results in our implementation suggest that this is quite feasible. The DMCS implementation seeks to minimize the assumptions made about the homogeneous nature of its target architecture. Finally, we present some extensions to the API for PORTS that will improve the performance of sparse, adaptive and irregular type of numeric computations.
Keywords: parallel processing, runtime systems, communication, threads, networks

1 Introduction

The portability of programs across supercomputers has been addressed very successfully by MPI [8] which is intended to be an easy-to-use and attractive interface for the application programmer and tool developer. However, it is not intended to be a target for the runtime support systems software needed by compilers and problem solving environments, since this software requires an efficient (and perhaps inevitably, less friendly) substrate for point-to-point communication, collective communication and control operations. Issues not addressed by MPI, such as dynamic resource management, concurrency at the uniprocessor level and interoperability at the language level, need to be addressed by such substrates.

These issues are being addressed by a consortium called *POrtable Run-time Systems* (PORTS) [15] which consists of research universities, national laboratories, and computer vendors interested in advancing research for software

* This work supported by the Cornell Theory Center which receives major funding from the National Science Foundation, IBM corporation, New York State and members of the its Corporate Research Institute.
** Chrisochoides' current address is Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556.

substrates that provide support to compilers and tools for current and next generation supercomputers. Specific goals of the group are:

1. to promote the development of standard applications programming interfaces (APIs) in multithreading, communication, dynamic resource management, and performance measurement that will be used as a compiler target for various task-parallel and data-parallel languages,
2. to provide support for interoperability across parallel languages and problem solving environments, and
3. to encourage the development of a community code repository that will result from this consortium's activities.

To achieve these ends, the *PORTS* consortium has come up with several APIs. The first API, *ports_threads*, has already been agreed upon by the PORTS consortium. It comprises of a set of functions for lightweight thread management, modeled after a subset of the POSIX thread interface. An implementation of the *ports_threads* interface is available from Argonne National Laboratory [16]. In addition, a set of functions have been specified for timing and event logging, using the high resolution, synchronized clocks available on many shared and distributed memory supercomputers. The timer package is thread safe, but not thread aware. In other words, a correct implementation of this specification can be used in a preemptive thread environment, however the specification does not require threads. An extremely fast implementation of *ports_timing* is available from University of Oregon [24].

There is a proposed API for communication, and for the integration of communication with threads[17]. The *PORTS* consortium has been experimenting with four different approaches.

1. *Thread-to-thread* communication, supported by CHANT [2].
2. *Remote service request* communication, supported by NEXUS [11].
3. *Hybrid* communication, supported by TULIP [25].
4. The DMCS approach outlined in this paper.

In this context, our DMCS implementation accomplishes the following.

- We provide a simple mechanism for reducing the scheduling latency of urgent remote service requests, as well as the communication overhead associated with remote service requests for sparse, adaptive and irregular numeric computations.
- We isolate the interaction between threads and communication into a simple module which is easy to understand and modify, called *control*.
- We provide a global address space and a threaded model of execution that provides a common programming model for SMPs, clusters of SMPs and MPPs.
- We want a very lean and modular layer that allows us to "plug-and-play" with different module implementations from PORTS community.

The rest of the paper is organized as follows. In Section 2, we outline the architecture of the data-movement and communication substrate, including the interaction of the threads and communication modules. In Section 3, we discuss the implementation details of DMCS. Preliminary performance data for simple kernels from sparse and adaptive computations are given in Section 4. Finally, the related work and a summary with discussion are presented in Sections 5 and 6 respectively.

2 Architecture

DMCS consists of three modules: (i) a *threads* module (ii) a *communication* module and (iii) a *control* module. The threads and communication modules are independent, with some clearly defined interface requirements, while the control module is built on top of the point-to-point communication and thread primitives. The *threads* module supports the primitives defined by the PORTS consortium, *ports_threads*. We are using the implementation provided by the Argonne group, PORTS0 [16], augmented by an extra routine: *ports_thread_create_atonce*. The efficient implementation of this routine is necessary to minimize the scheduling latency of certain urgent, *remote service requests* [7]. Clearly, this extension can be implemented on top of the existing *ports_threads* primitives provided by PORTS0(for example, using the thread priority attributes), but for efficiency reasons, we choose to implement it, whenever possible, directly on the underlying thread package. A prototype is implemented on top of the QuickThreads [19], which has been ported to a wide variety of workstation and PC architectures.

The communication module provides the necessary support for the implementation of a global address space over both shared and distributed memory machines. Collective communication primitives are not considered in this paper. In the future, we plan to evaluate and use the "rope" primitives introduced in [10] and [20]. As in Split-C, NEXUS, TULIP and Cid, our communication abstraction is the *global pointer*. A global pointer essentially consists of a context number and a pointer to the local address space. The functionality of the communication module for point-to-point data-movement includes routines like *get/put* to initiate the transfer of data from/to a remote context to/from a local context. The interaction of the communication module and threads takes place in the *control* module which integrates the thread scheduler with the point-to-point communication mechanism. Figure 1 depicts the three modules of the DMCS and their interaction.

The *control* module provides support for remote procedure invocation, also known as *remote service requests* (RSRs). Remote procedures are either system/compiler procedures or user defined handlers. These handlers can be threaded or non-threaded. The threaded handlers can be either *urgent* (scheduled after a fixed quanta of time[3]) or can be *lazy* (scheduled only after all other computation and manager threads have been suspended or completed). The non-threaded

[3] The time interval of a timeslice or the time it takes for the next context-swithch of the current thread in the case of a non-preemptive scheduling environment.

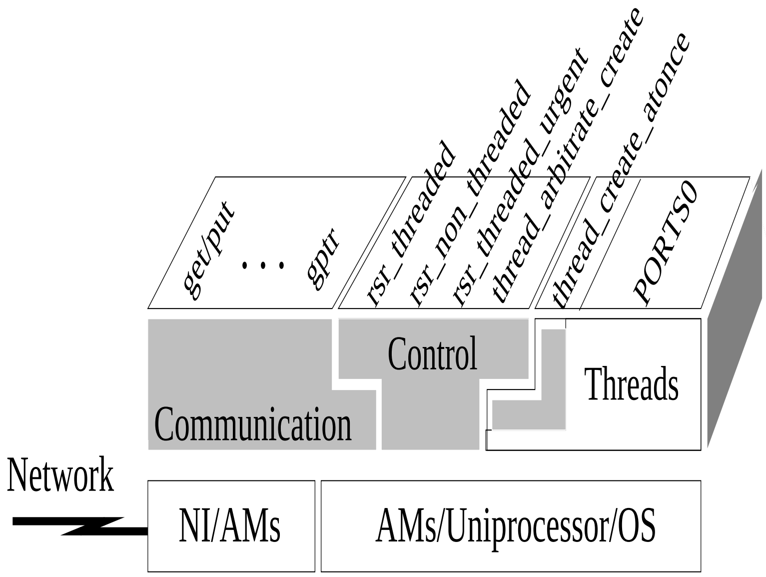

Fig. 1. Architecture

handlers are executed either as the message is being retrieved from the network interface[4], or after the message retrieval has been completed [1]. Finally, the control module provides some limited support for simple load balancing by allowing associating a window within which load on any processor can be balanced. This load-balancing support was chosen after experiments with the SplitThreads [23] system.

3 Implementation

In this section, we discuss the thread, communication and control modules of DMCS.

3.1 Thread subpackage

The underlying threads package consists of a user-level threads core called QuickThreads, which is a non-preemptive user-level threads package for thread creation and initialization. It provides no scheduling policies or mechanisms. It also lacks semaphores, monitors, non-blocking I/O etc. It provides machine dependent code for creating, running and stopping threads. It also provides an

[4] This works by overlapping computation and communication in the instruction level by interleaving the computation and flow of control that corresponds to the incoming message and the load/store operations needed to retrieve the message from the network interface.

easy interface for writing and porting thread packages. The higher level thread package has the responsibility of providing any additional functionality. Since the QuickThreads package is very flexible, clients can be designed with specific applications in mind and can be selectively tuned very easily.

DMCS implements a non-preemptive threads package. The thread routines in DMCS can be classified into *scheduling routines* and *management routines*, depending on their functionality. In DMCS, thread creation is separated from thread execution. DMCS maintains a queue of runnable threads, and thread creation routines simply insert a new thread into this runnable queue. Two different creation mechanism are provided (*dmcs_thread_create_atonce, dmcs_thread_create*) corresponding respectively to a low priority and a high priority for the newly created thread. It is important to note that these routines simply create and initialize a new thread, but do not actually run the thread. Thread management routines are responsible for actually running any threads in the run queue maintained by DMCS. The *dmcs_run* routine examines the run queue to check if it contains any threads, runs all of them to completion and then returns. Since DMCS provides non-preemptive threads, a *dmcs_yield* routine is provided to enable a thread to voluntarily de-schedule itself. For the sake of efficiency, DMCS also preallocates stack segments, which can be used by a thread creation routines instead of allocating memory on the fly. This leads to a more efficient thread creation routine. Finally, threads in DMCS can have several attributes that can be configured on a per-thread basis. Attributes currently implemented are the stack size of a thread, and whether the thread should use a pre-allocated stack, or do a fresh allocation for its stack.

3.2 Communication subpackage

The communication subpackage is implemented on top of a generic active message implementation on the SP-2[1, 21]. Active messages are a mechanism for asynchronous, low-overhead communication. The fundamental idea in Active Messages is that every message is sent along with a reference to a handler which is invoked on receipt of the message. The generic active message specification provides for small messages as well as bulk transfer routines. The implementation of this specification on the SP-2[21] is optimized so that small messages are delivered as efficiently as possible; for sufficiently large messages, the bandwidth attained is very close to the peak hardware bandwidth.

DMCS provides a homogeneous, data driven, asynchronous and efficient runtime environment. The communication subpackage of DMCS consists of three modules:

- *global pointers module*: DMCS provides the notion of a global pointer through which remote data can be accessed. When a program using DMCS runs on a N processors, each processor is assigned a unique integer id (known as a *dmcs_context*, and returned by the routine *dmcs_mycontext()*) in the range $0 \ldots N-1$. Any processor can access local data through a regular pointer. However, any remote data must be accessed through a *global pointer*. A global pointer is made up of a *dmcs_context* and a local pointer. Routines are provided to make a global pointer out of a local pointer and a *dmcs_context*, and also to extract these fields from a global pointer. This module is the only module that depends on the homogeneity of the underlying hardware (in the determination of unique context numbers).

– *Acknowledgement Variables*: Since DMCS is asynchronous by nature, a mechanism is provided to enable programs to find out about data transfer completion. This mechanism is an acknowledgement variable. An acknowledgement variable is represented as a 16-bit integer for efficiency reasons. An acknowledgement variable can have three states: *cleared*, *set*, and *unitialized*. To use an acknowledgement variable, a program must first request one using the routine *dmcs_newack()*, which returns an unused acknowledgement variable. This return value can be used as a handle to perform various operations on the acknowledgement variable. For example, *dmcs_testack()* checks if the acknowledgement variable has been set or not, and return immediately. On the other hand, *dmcs_waitack()* waits until the variable in question has been set. It is also possible to clear an acknowledgement variable using *dmcs_clearack()*. Finally, it is possible to "anticipate" the use of an acknowledgement variable in future data transfer using the function *dmcs_anticipateack()*. The last routine has an important role in one-sided data transfer operations, which will be described later. Finally, it is possible to use the same acknowledgement variable in conjunction with multiple data transfer operations. Use of an acknowledgement variable is always optional and a value of NULL for an argument of this type indicates that the particular acknowledgement variable in question is not being used.

– *Get and Put operations*: DMCS provides routines for one-sided communication. In other words, communication does not happen through a pair of matched sends and receives. Instead, get and put routines are provided for fetching remote data and storing remote data respectively. Both these operations are inherently asynchronous. Acknowledgement variables can be used to determine the state of one of these transfers. A Get operation transfers data from a source specified by a global pointer to a destination specified by a local pointer. A single acknowledgement variable passed as an argument to this routine is *set* when the data transfer operation is complete. A Put operation transfers data from a local buffer specified by a local pointer to a remote data buffer specified by a global pointer. A put operation has associated with it three acknowledgement variables: a *local_ack*, which is set when the local data buffer can be reused by the application program, a *remote_ack*, which is set on the processor that initiated the put operation to indicate that the put operation on the remote processor is complete, and finally a *remote_remote_ack* which is set on the remote processor to indicate that the put operation there has been completed. For the remote-processor intimation to work correctly, it must first anticipate this put operation by calling *dmcs_anticipateack()* on the acknowledgement variable specified as the *remote_remote_ack* in the put operation. As mentioned previously, all these acknowledgement variables are optional and a value of NULL can be passed as the corresponding argument to indicate dis-interest in that particular acknowledgement variable.

3.3 Control subpackage

The control subpackage is layered on top of the threads subpackage and the communication subpackage. The control subpackage consists of two sets of modules: remote-service requests and load-balancing routines.

- *Remote Service Requests*: DMCS provides several kinds of remote service requests. A remote service request consists of a remote context, a function to be executed at the remote context and the arguments to the function. In addition, a type argument is also passed, indicating the type of the remote service request. This type argument can take three possible values, and determines how the function at the remote end is executed. The function can be executed immediately on arrival, or it can be threaded. If the handler is threaded, then it can executed as either a low-priority thread or a high-priority thread. The type argument for the remote service request determines which of these modes of execution is performed.
  DMCS recognizes that passing a function as part of the remote service request may not be always desirable. This is the case in adaptive mesh refinement, where the components of a mesh are distributed on various processors. Different processors use different "interpolation functions" to transfer data between different grids at different stages of the mesh refinement process. When data related to a mesh is sent to another processor that needs it, it is not always known at the sender what kind of interpolation function the receiver needs to apply. Also, in a non-SPMD environment, a function address valid at the center may not be valid at the receiver. To provide a solution to this problem, DMCS provides the notion of an indexed remote service request. At any processor, a function can be "registered" with an integer tag. When a message with that particular tag is received, the function registered with that tag is invoked as the handler for the message. This mechanism enables different processes to register different handlers with the same tag and allows great flexibility.
  Some message passing system provide more efficient routines to transfer small amounts of data which are more efficient than using the generic bulk transfers. Keeping this in mind, DMCS optimizes the bulk transfer routines as well as the remote service request routines for small message sizes. For our implementation, this provides an improvement of almost 20% for small message sizes over the unoptimized case of using the same underlying generic bulk transfer routine. The generic form of the remote service request takes as arguments a remote context, a remote handler (either a function, or an index to one), a buffer of arguments and a length parameter counting the size of the arguments.

Finally, it is common in programs to transfer some data to a remote processor, followed by the invocation of a remote service request acting on the data just transferred. For such operations, it is more efficient to do the data transfer and the remote service invocation in one step, rather than transferring the data first, and then invoking the remote service request. This is because, the second approach usually involves some additional synchronization delays that can avoided by the first approach. For this purpose, DMCS provides *putw_rsr* routines, which combine the data transfer and the remote service action at the end of the data transfer. This is similar to the functionality specified in the generic active message specification.

– *Load balancing*: DMCS implements a simple parametrized load balancing primitive. The load on a processor is defined to be simply the number of threads on that processor. DMCS provides a primitive than enables a processor to start a new thread on the least loaded processor within a certain window size. The window size is a parameter that can be customized. Processors are divided into groups of size equal to the parameter. Load balancing is provided by DMCS within each of the groups. By tuning the window size parameter, load balancing can be confined to a small neighborhood or be over the entire machine. It may be noted that this load-balancing happens at thread creation, and a running thread is never migrated.

4 Performance Data

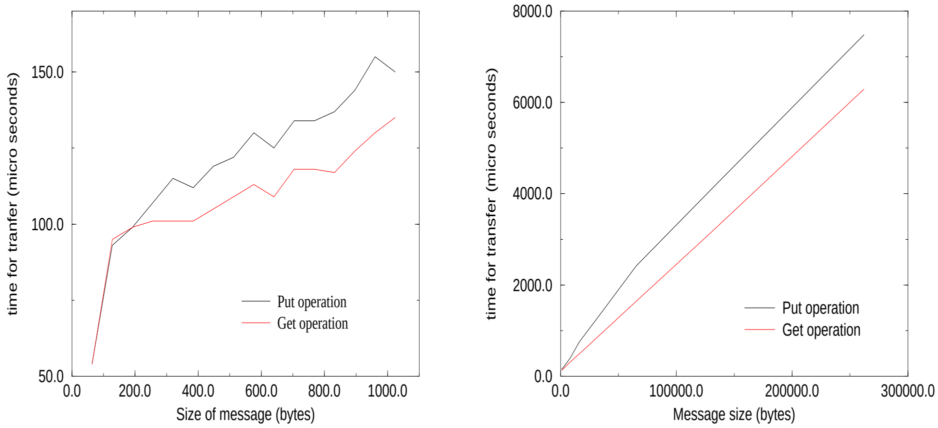

Fig. 2. Communication time as a function of message size: small message size(left), large message size(right)

In this section, we discuss the performance of our run time system on the IBM SP-2, comparing it with the performance of other systems.

The following are the basic parameters of our runtime system:

- Thread creation time = $12\mu s$
- Context switch time = $5.5\mu s$
- Peak Data transfer bandwidth = $33.6 MBytes/sec$
- One-way latency for a 0-byte message $29\mu s$.
- Time elapsed for a non-threaded null remote service request = $31\mu s$

The communication paramters of our runtime system parameters are very close (within 10%) of the underlying active message layer. For example, one-way latency for a 0-byte message in DMCS is $29\mu s$, which represents an overhead of 10% over the underlying active message latency of $26\mu s$. This compares with a handler to handler latency of $79\mu s$ in Nexus, which represents an overhead of 80% over the native MPL one-way latency of $44\mu s$. Similarly, the bandwidth achieved in DMCS for bulk transfers is $33.6 MBytes/sec$ for get operations, and $29 MBytes/sec$ for put operations. These numbers are also within 10% of the corresponding parameters of the active message layer. This verifies that the overhead introduced by the DMCS layer is quite small.

An advantage of the homogeneity of the target architecture is that it enables certain kind of optimizations in the runtime system implementation which are very hard otherwise. For example, in our implementation, we have recognized that certain kinds of remote service requests are very common. One such instance occurs in matrix vector multiplication, where the non-local portion of the vector needs to be fetched and then used in SAXPY operations. The usual manner to accomplish this is simply using a putw_rsr routine, with the remote service request handler taking care of the saxpy operations. However, by noting that the remote portion of the vector need not be stored, but can be simply used to compute the relevant portion of the matrix vector operations, after which only the results of the operations need to be stored, we can implement a more efficient matrix vector multiply operation in the runtime system itself. This routine avoids all buffering of the data and does the relevant computation in the hardware buffer itself. As can be seen from Figure 3, the performance saving can be quite substantial when the number of floating point operations is small for every data element, which is the case for sparse computations.

DMCS is being currently used to implement a task-parallel version of the Bowyer-Watson algorithm for mesh generation [14]. This algorithm provides an ideal mesh refinement strategy for a large class of unstructured mesh generation techniques on both sequential and parallel computers, by preventing the need for global mesh refinements. This application has been ported from an active message implementation to a PORTS implementation on top of DMCS. For the most part, this port was straightforward. The specialized matrix vector product routine of DMCS, described above, has been used to implement a sparse matrix-vector multiply routine for use in iterative solvers for large systems of linear equations.

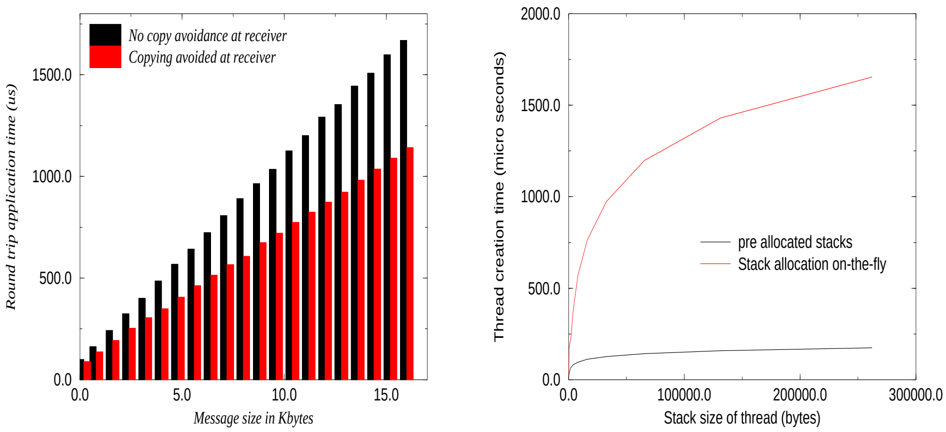

Fig. 3. Savings by copy avoidance at receiver(left), Comparison between preallocated and non preallocated threads(right)

5 Related Work

Three other software systems that integrate communication with threads are particularly interesting because they implement the same interface but have different design philosophies and objectives. CHANT implements thread-to-thread communication on top of portable message passing software layers such as p4 [13], PVM [18], and MPI [8]. The efficiency of this mechanism depends critically on the implementation of message polling or message delivery interrupt. There are three common approaches to polling for messages: (i) individual threads poll until all outstanding receives have been completed, (ii) the thread scheduler polls before every context switch on behalf of all threads, and (iii) a dedicated thread, called the *message thread*, polls for all registered receives. For portability, CHANT supports the first approach, since many thread packages do not allow their scheduler to be modified. Performance data in [2] indicate that there is little difference in performance between the first two polling approaches.

NEXUS decouples the specification of the destination of communication from the specification of the thread of control that responds to it. NEXUS supports the *remote service request* (RSR) driven communication paradigm which is based on the remote procedure call mechanism. The multithreaded system (or user) registers a message handler which is a new thread and is to be invoked upon receipt of an incoming message. The handler possesses a pointer to a user-level buffer into which the user wishes the message contents to be placed. The handler threads are scheduled in the same manner as computation threads. In a preemptive scheduling environment, each handler gets highest priority and will always get scheduled after a fixed quanta of time. In a non-preemptive environment, the handler thread gets assigned a low priority and gets scheduled only after other threads have suspended; thus, there is no bound on the waiting time for the handler in this case. The RSR driven communication paradigm is implemented in NEXUS which a portable multithreaded communication library for parallel language compilers and higher-level communication libraries [11].

TULIP's *hybrid* approach is essentially a combination of thread-to-thread and RSR driven communication paradigm [25]. The *hybrid* approach is essentially a combination of thread-to-thread and RSR driven communication paradigm and is supported by TULIP [25]. In the runtime substrate, TULIP provides basic communication via global pointers and remote service requests. Then, at the pC++ language level, there is the concept of threads and the notion of group thread actions [10, 20]. Communication is one module, the basic threads functions (i.e., creation, thread synchronization, etc) are in another module, and the two are combined into the *rope* module.

Finally, functionality similar to DMCS *threads* and *communication* and *control* modules are provided by a number of other runtime systems like Cid, Split-C, Cilk, and Multipol. Cid [3] and Split-C [22] are parallel extensions to C. Both systems support a global address space through the abstraction of the global pointer. They also implement asynchronous, one-sided communication, and multithreading (either in the language as in Cid, or through extensions as in SplitThreads [23]), and have mechanisms for overlapping computation with communication and synchronization latencies. Cilk [4] is similar but it targets a more restricted class of computations (*strict* computations). The scheduling policy is fixed, and for a certain class of applications is provably efficient with respect to time, space and communication. In contrast to the Cilk runtime system, Multipol [9] allows more flexibility to the programmer. For example, the programmer is free to use customized schedulers to accommodate application-specific scheduling policies for better performance, and can also specify how much of a thread state needs to be saved.

6 Conclusions

We have described the design and implementation of a data-movement and control substrate (DMCS) for network-based, homogeneous communication within a single multiprocessor, which implements an API for communication and computation defined by the PORTS committee. Unlike systems like Nexus, which are designed for a heterogenous environment, DMCS is targeted for a homogenous environment. Although the scope of DMCS is therefore restricted, it permits us to do optimizations which are difficult in heterogenous environments.

7 Acknowledgements

We thank Pete Beckman, Chi-chao Chang, Grzegorz Czajkowski, Thorsten von Eicken, Ian Foster, Dennis Gannon, Matthew Haines, L. V. Kale, Carl Kesselman, Piyush Mehrotra, and Steve Tuecke for valuable discussions.

References

1. Thorsten von Eicken, Davin E. Culler, Seth Cooper Goldstein, and Klaus Erik Schauser, Active Messages: a mechanism for integrated communication and computation *Proceedings of the 19th International Symposium on Computer Architecture, ACM Press*, May 1992.
2. Matthew Haines, David Cronk, and Piyush Mehrotra, On the design of Chant : A talking threads package, NASA CR-194903 ICASE Report No. 94-25, Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23681-0001, April 1994.
3. R.S. Nikhil, Cid: A Parallel, "Shared-Memory" C for Distributed Memory Machines. In Lecture Notes in Computer Science, vol 892.
4. Christopher F. Joerg. The Cilk system for Parallel Multithreaded Computing. Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, January, 1996.
5. L.V. Kale and M. Bhandarkar and N. Jagathesan and S. Krishnan and J. Yelon, CONVERSE: An Interoperability Framework for Parallel Programming, Parallel Programming Laboratory Report #95-2, Dept. of Computer Science, University of Illinois, March 1995
6. Nikos Chrisochoides and Nikos Pitsianis, FFT Sensitive Messages, to appear as Cornell Theory Center Technical Report, 1996.
7. Nikos Chrisochoides and Juan Miguel del Rosario, A Remote Service Protocol for Dynamic Load Balancing of Multithreaded Parallel Computations. Poster presentation in Frontiers'95.
8. MPI Forum, Message-Passing Interface Standard, April 15, 1994.
9. Runtime Support for Portable Distributed Data Structures C.-P. Wen, S. Chakrabarti, E. Deprit, Chih-Po Wen, A. Krishnamurthy, and K. Yelick. Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers, May 1995.
10. N. Sundaresan and L. Lee, An object-oriented thread model for parallel numerical applications. *Proceedings of the 2n Annual Object-Oriented Numerics Conference - OONSKI 94, Sunriver, Oregon, pp 291-308*, April 24-27 1994.
11. I. Foster, Carl Kesselman, Steve Tuecke, Portable Mechanisms for Multithreaded Distributed Computations Argonne National Laboratory, MCS-P494-0195.
12. Ian Foster, Carl Kesselman and Steven Tuecke, The NEXUS approach to integrating multithreading and communication, Argonne National Laboratory.
13. Ralph M. Butler, and Ewing L. Lusk, *User's Guide to p4 Parallel Programming System* Oct 1992, Mathematics and Computer Science division, Argonne National Laboratory.
14. Nikos Chrisochoides, Florian Sukup, Task parallel implementation of the Bowyer-Watson algorithm, CTC96TR235, Technical Report, Cornell Theory Center, 1996.
15. Portable Runtime System (PORTS) consortium, http://www.cs.uoregon.edu/research/paracomp/ports/
16. PORTS Level 0 Thread Modules from Argonne/CalTech, ftp://ftp.mcs.anl.gov/pub/ports/
17. A Proposal for PORTS Level 1 Communication Routines, http://www.cs.uoregon.edu/research/paracomp/ports
18. A. Belguelin, J. Dongarra, A. Geist, R. Manchek, S. Otto, and J. Walpore, PVM: Experiences, current status and future direction. Supercomputing'93 Proceedings, pp 765–6.

19. David Keppel, Tools and Techniques for Building Fast Portable Threads Package, UW-CSE-93-05-06, Technical Report, University of Washington at Seattle, 1993.
20. Data Parallel Programming in a Multithreaded Environment, **(Need authors...)***to appear i a Special Issue of Scientific Programming*, 1996.
21. Chichao Chang, Grzegorz Czajkowski, Chris Hawblitzell and Thorsten von Eicken, Low-latency communication on the IBM risc system/6000 SP. To appear in Supercomputing '96.
22. David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken and Katherine Yelick. Parallel Programming in Split-C. Supercomputing'93.
23. Veena Avula. SplitThreads - Split-C threads. Masters thesis, Cornell University. 1994.
24. Portable Clock and Timer Module from Oregon, http://www.cs.uoregon.edu/research/paracomp/ports
25. Pete Beckman and Dennis Gannon, Tulip: Parallel Run-time Support System for pC++, http://www.extreme.indiana.edu.

This article was processed using the LaTeX macro package with LLNCS style