Tulip: A Portable Run-Time System for Object-Parallel Systems[1]

Peter Beckman Dennis Gannon
Computer Science Department, Indiana University, Bloomington, Indiana 47405

Abstract

This paper describes Tulip, a parallel run-time system used by the pC++ parallel programming language. Tulip has been implemented on a variety of scalable, MPP computers including the IBM SP2, Intel Paragon, HP/Convex SPP, Meiko CS2, SGI Power Challenge, and Cray T3D. Tulip differs from other data-parallel RTS implementations; it is designed to support the operations from object-parallel programming that require remote member function execution and load and store operations on remote objects. It is designed to provide the thinnest possible layer atop the vendor-supplied machine interface. That thin veneer can then be used by other run-time layers to build machine-independent class libraries, compiler back ends, and more sophisticated run-time support. Some preliminary performance measurements for the IBM SP2, SGI Power Challenge, and Cray T3D are given.

1 Introduction

Object Parallelism is a term which we use to describe a family of related approaches to programming parallel computers. This model extends data parallelism to object-oriented languages and systems. Object parallelism is defined by the concurrent application of parallel functions and operators to aggregate data structures (objects). In an object-parallel system, the programmer creates and manipulates *collection* or *container* objects that encapsulate a distributed data structure on a massively parallel computer. Examples of this style of programming include pC++ [1], a collection-based parallel programming language; Single Program Multiple Data (SPMD) C++ class libraries such as LPARX [2], P++ [3], the POOMA library [4] and CHAOS++ [5], that provide user-defined distributed data structures; and data-parallel extensions to the C++ Standard Templates Library such as the Amelia Vector Template Library (AVTL) [6].

While C++ is the most common language for this work, it is not the only way object parallelism may be expressed. There are many other parallel programming models that exploit object-oriented concepts. These include Mentat [7], CC++ [8], CORBA, Charm++ [9], and UC++ [10]. In these systems, the emphasis is on task-level parallelism with networked and heterogeneous systems.

In this paper, we describe the *Tulip* run-time system, which was designed to support object-parallelism on scalable, massively parallel processing (MPP) computers. Its design was governed by a set of objectives and requirements that were derived from our experience with the pC++ project. We were motivated by the following concerns:

- Tulip will serve as a *compiler target* as well as an API for parallel class library designers. Consequently, the design should be guided by optimization rather than end user's programming ease.
- Because the current standard for writing parallel class libraries is to use SPMD style, Tulip should be based on this model. However, it should also provide a path to a more general *multi-threaded* implementation style. In particular, future support for dynamic, nested data parallelism and mixed task-data parallelism will require multi-threaded implementations.
- Basic message passing should be a fundamental part of Tulip; the MPI standard [11] is now well supported on many systems. However, object-parallelism differs from data parallelism. Unlike uniform arrays, distributed complex-linked structures change size and shape over the lifetime of the program. Consequently, translating all data movements into predetermined send-receive pairs is not always easy. Furthermore, object-oriented style lends itself more naturally to a single-sided communication model where a member function is invoked through a pointer to an object. Active messages [12] and remote procedure call methods must be a component of Tulip.
- Tulip must provide a consistent interface across all the target platforms, but it should provide explicit support for hardware features that have been added to many parallel systems. A primary difference between an MPP architecture and a network of workstations is the special hardware for collective communication, barriers, and shared memory. Tulip should include primitives such as barrier and remote memory load and store, whose implementation can use special hardware when it is available.

The work described here addresses three of these four concerns. This paper does not consider the interface to multi-threaded execution. A working group, known as POrtable Run Time System (PORTS) [13], is currently considering the design of a multi-threaded RTS. We also emphasize that we have not considered issues related to nested data parallelism, which may be handled by a multi-threaded system or by compiler technology such as that pioneered in the design the NESL system [14]. Our focus is on the design problems associated with building a run-time system that supports remote data load and store, collective operations, and remote procedure calls (RPC) within an SPMD execution environment.

We are not the first to address these problems. *Active messages* provide a limited form of remote procedure call, i.e., messages invoke a remote handler with a few bytes of argument data. The message size is bounded by the architecture of the processor network interface adaptor. Messages can either interrupt the addressed processor, or a remote-queue and polling mechanism can be used. Our approach differs in that we allow arbitrary argument lengths and do not interface the low-level hardware. NEXUS [15] also uses remote handlers, but is designed for multi-threaded task parallelism on heterogeneous networks. NEXUS does not attempt to exploit special hardware for collective communication or

[1]This research is supported in part by: ARPA DABT63-94-C-0029 and Rome Labs Contract AF 30602-92-C-0135

remote shared memory. Internally, NEXUS uses the thread interface specified by the PORTS consortium. The future multi-threaded implementation of Tulip will share that layer.

In the paragraphs that follow, we will describe the abstract machine model, the interface to the software, and the implementation. We conclude with a description of a series of experiments.

2 Machine Architectures

Object-parallel systems built atop Tulip are designed to run on a wide range of architectures. That machine model has been generalized to a collection of symmetric multiprocessors (SMPs) sharing some communication network. An SMP is a basic building block; it contains one or more CPUs that can share memory and have cache coherency. The vendor-supplied interface to concurrency may be kernel-level threads, virtual processors, or allocation of physical CPU resources. Most workstation vendors provide high-end SMP servers with 2-8 processors. For Tulip, a *Node* is an SMP, possibly connected to other SMPs via a high speed network. A *Context* is an address space. A Unix process on a SMP would be a single context. Lightweight threads share a context. A machine such as the SP2 can support several contexts per node. To explore the design and implementation issues for Tulip, three very different machine architectures were selected. Below, is a quick tour of each machine.

The IBM SP2 is a traditional distributed-memory message passing supercomputer. However, unlike the TMC CM5, Intel Paragon, and Cray T3D, each compute node is a complete computer, with it's own disk and host name. At the present time, each POWER2-based SP node has only one CPU, and cannot be an SMP. This makes mapping the SP2 to the parallel virtual machine quite simple - each processor is one node.

IBM interconnects nodes with a multi-stage high performance switch (HPS) capable of 40 MB/sec bandwidth and application to application latency of 40 microseconds. There are several software interfaces for communication on the SP2; we used an experimental IBM implementation [16] of MPI.

The T3D is a distributed-memory 3D torus-connected compute array of DEC Alpha microprocessors. On the T3D at the Pittsburgh Supercomputer Center, each processing element (PE) is a 150 Mhz Alpha with 64 megabytes of local memory. Each pair of PE share connection to a high speed interconnection network, with a peak performance of 300 MB/sec along each link. A convenient interface to the interconnection hardware is Cray's C communication library. Tulip uses the library, which provides "shmem_get" and "shmem_put" functions for accessing remote "shared memory" without interrupting the node (network DMA).

The 75 Mhz. MIPS superscalar R8000 microprocessor chipset is the heart of SGI's Power Challenge (SGIPC). The SGIPC at the Indiana University Computer Science Department has 2 GB of total memory and 10 processors, each equipped with a 4 MB data cache. The SGIPC uses snooping cache coherency hardware and the high speed bus can sustain a total peak transfer rate of 1.2 GB/sec.

SGI compilers are designed to interface the hardware support for *piggyback reads*, which can combine read requests so that other processors can share the response, and *synchronization counters*, which count broadcast increment transactions and help improve barrier synchronization times. However, the software interface for parallelism is clumsy. The current OS (IRIX 6.1) has no standardized thread interface, so users are stuck with programming `SPROC`s, a multiprocessor version of `fork()`. Furthermore, there is no easy way to insist that an `SPROC` execute on a particular compute node. In true SMP style, the individual tasks are moved from processor to processor based on unseen heuristics within the operating system.

3 The Design of Tulip

Tulip provides functions for remote memory load/store, remote procedure invocation, collective operations, timers, process control, and tracing and profiling. In the subsections that follow we discuss the first three of these. We will not discuss timers, process control, or tracing and profiling in this paper.

The object-parallel execution model is based on the parallel invocation of class member functions for a distributed set of objects. Many operations use pointers to objects that may exist within the same context or in a remote context. To support this concept we use two types of pointers. A local pointer is a simple, untyped, memory address valid only within the context it was created. A global pointer is the tuple of context number and local pointer. A global pointer uniquely identifies any memory address in the computational hardware.

3.1 Remote Memory Operations

The Tulip functions tulip_Get() and tulip_Put() are very much like `memcpy()`. For tulip_Get(), the source address is a global pointer, and the destination is a local pointer. For tulip_Put() it is switched. Since the functions are non-blocking, an address to a *service handle* must be passed into both functions. Later, the handle may be used to check if the operation has completed; tulip_Probe() and tulip_Wait(), which are non-blocking and blocking respectively, take a handle as an arguement. Tulip_Barrier() does collective synchronization. Tulip supports other collective functions such as reduce and broadcast, but they are beyond the scope of this paper.

3.2 Remote Procedure Invocation

Even on a shared memory machine such as the SGIPC, writing a remote procedure call mechanism can be difficult. There are two ways to detect the arrival of a RPC request, poll for it, or rely on an interrupt mechanism. An interrupt or active message mechanism embodies the asynchronous nature of an RPC nicely. However, standard Unix signals must go through many operating system layers, and usually have high latency. Our experiments revealed that two processors of the SGIPC take about 33 ms of wall clock time to ping-pong a Unix signal between them. We decided to poll for requests rather than pay that latency. Since the SGIPC has snooping caches, polling for a RPC is very fast. Polling can either be done after a timer has expired (again, using a slow signal), or polls can be added to the source code. The pC++ compiler "sprinkles" calls to tulip_Poll() into the code. Periodic calls to tulip_Poll() were required on all the computers.

Rather than provide a way to "register" function names and pointers, we chose an interface only a compiler could

love. Global pointer member function invocations must be identified by the compiler, and set up for remote execution. There is only one handler for each node. That handler is written by the compiler, or by the library writer. When a node issues a *RemoteAction* request, a notice is sent to the destination. The receiver must execute tulip_Poll() to check for its arrival. After the receiver notices the request, it executes RemoteActionHandler(). RemoteAction() needs only three parameters, a context number where the handler should be run, an integer type field, and a buffer (usually used to encode procedure arguments). The type field is an easy way provide the RemoteActionHandler() with information about the buffer. Future versions of Tulip will provide function registration, so that library builders can each plug in a module.

3.3 RemoteAction()

On the T3D, after tulip_Poll() finds a RemoteAction(), it uses shared memory to copy the argument buffer, eliminating a buffer copy and extra synchronization. Shared memory machines can skip the explicit fetch of the argument buffer and simply use a pointer. Unfortunately, a space leak results unless the original buffer is freed after RemoteActionHandler() exits. An acklowledge (ACK) must be sent back to the caller, so the buffer can be freed and the service handle updated.

RemoteAction() over a message passing layer combines control and data into one message. This eliminates a round trip to set up matched send and receive pairs by forcing the messaging layer to buffer and hold the message until tulip_Poll() notices their arrival and posts a RECV() to pull them in from the system buffers. Since argument buffers are not large (or the buffer would be passed by global pointer reference), this generally not a problem. After the handler exits, an ACK message is sent to the caller. Tulip_Poll() detects the message, and the corresponding service handle is modified.

3.4 Barrier()

Tulip implements non-blocking barriers, so user-level threads may continue executing, and other data movement operations can be serviced. Unfortunately most vendor-supplied barrier synchronization functions are blocking. Call their barrier function, and control is not returned until all processors enter the barrier and synchronize. We cannot use those types of vendor supplied barriers. For the SGIPC and SP2, we were forced to write our own barrier, which could execute tulip_Poll() and detect service requests while waiting at the barrier. Cray saw the wisdom of a providing a non-blocking barrier function interface on the T3D. Calling `set_barrier()` simply sets a bit on the hardware and returns immediately. Processors can then execute `test_barrier()` and tulip_Poll() until the barrier is over.

On the SP2 and SGIPC, a simple fan-in/fan-out binary tree was constructed to provide barrier. Nodes entering the barrier notify their parent node they are waiting at the barrier. When all a node's children have so registered, the event is propagated. While the nodes wait for the fan-out message, they execute tulip_Poll(), and may service other requests. When the root node receives a message from its children, the barrier is done, and notification fans out in the reverse direction.

3.5 Tulip_Get() & Tulip_Put()

Tulip_Get() provides a "receiver driven" communication primitive. On shared memory machines, tulip_Get() should never be called, since the compiler should be smart enough to generate code for shared memory. Nevertheless, if used, it is basically a call to `memcpy()`. For network DMA machines such as the T3D, tulip_Get() calls the appropriate memory transfer function.

For message-passing machines like the SP2, tulip_Get() requires a rendezvous protocol to avoid buffer copies. Unneeded buffer copies – especially for large objects, **must** be avoided. If sender and receiver carefully choose a message type, and the receive is preposted, extra buffer copies are eliminated. To do this, a RECV() is posted, then a short control message is sent to the owner, informing it of the request for data. The sender then "fills" the preposted RECV(). This senario also permits computation and communication to overlap, since there are no blocking calls.

To agree on a message type to match up each send/receive pairs requires the message `type` field be split into two logical fields: a message ID (MID) and a message flavor. This technique is quite similar to the way *Chant* [19] uses MPI to send messages to threads, except in this case, the extra bits stolen from the message type field are not used to encode destination, but to distinguish between pending tulip_Get() messages. Each pending tulip_Get() must have a unique message ID, so Tulip can match senders and receivers, and detect which preposted receives have completed, and which corresponding service handles should be updated.

Tulip_Put() is very similar to tulip_Get(). On the T3D and other network DMA machines, tulip_Put() calls the vendor-supplied data transfer engine. Tulip_Put() is not atomic. Unpredictable results occur when simultaneous writes overlap.

Tulip_Put() on the SP2 is more complex that its tulip_Get() counterpart. To avoid extra buffer copies, an extra message must be ping-ponged between source and destination. The first message announces the intent of the sender. The receiver must post a RECV() in anticipation of the object data. A control message is returned to the sender indicating it is clear to send. After the receiver detects that the message has indeed arrived, it sends an ACK to the sender, to notify the node the transaction is complete.

4 Early Performance Results

4.1 Communication

In this section we present some of the early performance tests of Tulip. Figure 1 compares tulip_Get() on the SP2 and T3D with the native communication layer provided by the vendor. Remember that tulip_Get used "one sided" communication. On the SP2, the remote "owner" of the data simply executes tulip_Poll(). When a control message arrives, it decodes and services it. For the T3D, the owner of the data polls, but since network DMA communication occurs without owner participation, the owner remains idle.

For the native SP2 communication test, raw synchronous MPI_Ssend() and MPI_Recv() between two nodes was used. On the T3D, a call to shmem_get() was all that was necessary. For a 64K transfer, the T3D native layer achieved about 35 MB/sec bandwidth, while MPI on the SP2 got about 31.5 MB/sec.

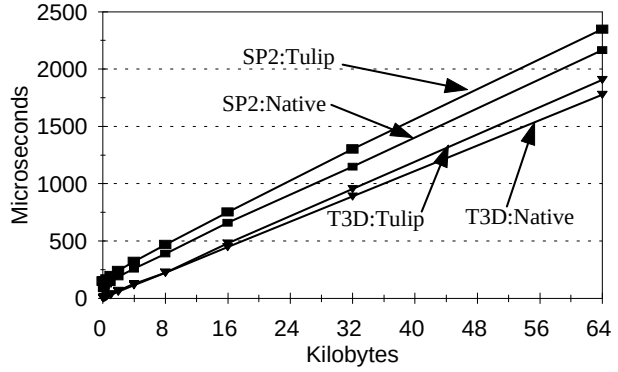

Figure 1: Tulip_Get() on the SP2 and T3D

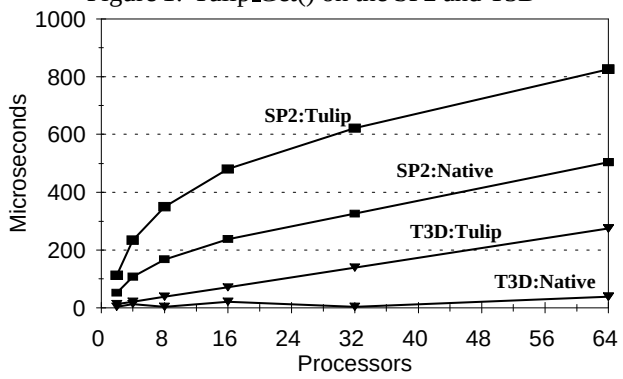

Figure 2: Barrier on the SP2 and T3D

Figure 1 indicates that a small, nearly constant overhead was associated with using tulip_Get() instead of the native communication layer. Because of the current design of tulip_Poll(), the added latency varies with the number of processors. Clearly, tulip_Poll() should get first priority as the RTS is optimized and tuned.

4.2 RemoteAction

In this test, a RemoteAction() request was sent to a node, which in turn, sent a RemoteAction() event back. This ping-pong was repeated, and the time averaged. For the SP2, with nodes waiting at the barrier and polling, the ping-pong RemoteAction() averaged about 165 microseconds. On the T3D, the time was 87.3. Dividing these numbers in two, gives the optimal one-way RemoteAction() latency at 82.5 and 43.7 microseconds for the SP2 and T3D respectively.

4.3 SP2 and T3D Barrier Synchronization

As described earlier, Tulip's barrier must be non-blocking, and execute tulip_Poll() between tests for completion. The following tests compare Tulip's barrier with the vendor-supplied barrier. The times represent the average wall clock time spent in the barrier

Several patterns for arrival of the nodes to the barrier were tested, including nearly-synchronized arrival (only one floating point operation between successive barrier calls), randomly staggered arrival (each node arrives at a different time), and late arrival (one node arrives at the barrier late). For the T3D and SP2, all three of these tests presented quite similar results. For brevity, Figure 2 only shows synchronized arrival, and compares the native barriers on the SP2 and T3D to Tulip's non-blocking versions.

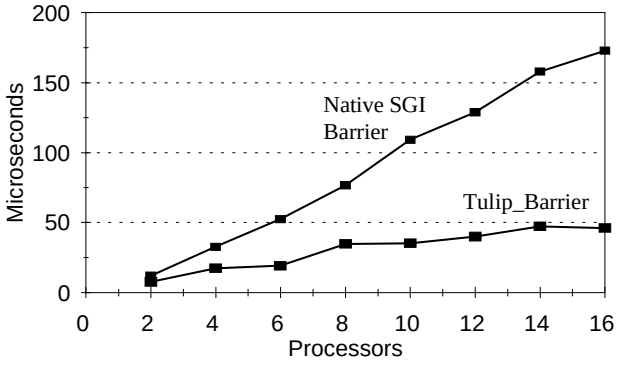

Figure 3: SGI Barrier

Notice that the native T3D barrier is blazingly fast. In all of our tests, from 2 to 64 nodes, the barrier was never slower than 4 microseconds. Cray Research uses a series of wires and fast logic to make their hardware-assisted barrier. This is a perfect example of why Tulip **must**, whenever possible, use the fast hardware provided by the vendor. Tulip's barrier on the T3D once again demonstrates that the delays added by tulip_Poll() should be minimized. For the SP2, Tulip's polling barrier was within a factor of about 2 to 3 from the MPI blocking barrier.

4.4 SGIPC Barrier Synchronization

Tulip's barrier for the SGIPC proved to be the biggest surprise. The first implementation, using a tree, had linear performance. Careful investigation revealed that our entire reduction tree was within a single 128 byte cache line – serializing our updates. A careful distribution of tree nodes improved performance. Our polling barrier outperformed the native SGI barrier.

Figure 3 compares Tulip's barrier to the SGI barrier for nearly-synchronized arrival of nodes. The near-linear performance of SGI's barrier suggests that their implementation does not use a tree, but rather all the nodes fight for control over some resource. The more nodes, the more contention.

5 Conclusions

In this paper we have described a simple portable, light-weight run-time system for scalable MPP architectures. The design emphasizes the types of operations that are important for implementing object-parallel extensions to the data-parallel model. Tulip incorporates remote memory fetch and store operations, remote procedure call, and collective communication and synchronization. The design of the primitives in this system are intended for use by a compiler or a parallel library designer. Consequently, they are to be as fast and efficient as possible. In particular, they are designed to take advantage of any architectural feature that the host parallel computer provides.

In our experiments we measured the performance of this design on three different machines. For the SP2, we built RPC and memory fetch/store operations on top of MPI. Adding those features has a cost; barrier synchronization is more complex, and about twice as slow. The cost of

fetch/store operations is nearly the same as MPI synchronized send/receive pairs. For the CRAY T3D, we used the existing hardware barrier to build an extended barrier that allows asynchronous communication (RPCs) to take place while processors wait. Our initial implementation is not very fast compared to the 4 microsecond hardware barrier, but it is still faster than the native MPI barrier on the SP2. Furthermore, we feel that the costs can be substantially reduced by a carefully tuned, low-level implementation.

One conclusion that can be drawn from this work is that an extension of the MPI standard to include active messages, RPC operations and remote memory load and store operations, would allow vendors to optimize these operations to match their special architectural features. Consequently, library designers and compiler writers would be given much greater opportunity to optimize performance for a given machine than they currently have.

References

[1] D. Gannon, F. Bodin, P. Beckman, S. Yang, and S. Narayana. *Distributed pC++: Basic ideas for an object parallel language*, Journal of Scientific Programming, Vol. 2, pp. 7-22 (1993).

[2] Scott B. Baden, Scott R. Kohn, Silvia M. Figueria, and Stephen J. Fink, *The LPARX User's Guide v1.0*, Technical report, Department of Computer Science and Engineering, University of California, San Diego, La Jolla, CA 92093-0114 USA, Apr. 1994.

[3] R. Parsons, D. Quinlan, *A++/P++ Array Classes for Architecture Independent Finite Difference Computations*, Proceedings, OONSKI, 1994. pp. 408-418.

[4] J. Reynders, D. Forslund, P. Hinker, M. Tholburn, D. Kilman, W. Humphrey, *Object Oriented Particle Simulation on Parallel Computers*, Proceedings, OONSKI, 1994. pp. 266-279.

[5] C. Chang, A. Sussman, J. Saltz, *Support for Distributed Dynamic Data Structures in C++*, Univ. of Maryland, Dept. of Computer Science Technical Report CS-TR-3266, 1995.

[6] T. Sheffler, *A Portable MPI-based parallel vector template library*, RIACS Technical Report 95.04, RIACS, NASA Ames Research Center. 1995.

[7] A. S. Grimshaw. *An introduction to parallel object-oriented programming with Mentat*, Technical Report 91 07, University of Virginia, 1991.

[8] K. M. Chandy and C. F. Kesselman. *CC++: A declarative concurrent object-oriented programming notation*, In Research Directions in Object-Oriented Programming, MIT Press, 1993.

[9] L. Kale, S. Krisnan, *Charm++: A Portable Concurrent Object Oriented System Based on C++*. Technical Report. Univ. of Illinois, Urbana-Champaign, 1994.

[10] UC++ - Europa Parallel C++ Project. WWW reference only: http://www.lpac.qmw.ac.uk/europa/.

[11] W. Gropp, E. Lusk and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1994.

[12] von Eicken, Thorsten; Culler, David E.; Goldstein, Seth Copen; Schauser, Klaus Erik. *Active Messages: a Mechanism for Integrated Communication and Computation.* UCB//CSD-92-675, March 1992. 20 pages.

[13] PORTS- POrtable Runtime System consortium. *The PORTS0 Interface*, Technical Report, Jan. 1995.

[14] Guy E. Blelloch, *NESL: A Nested Data-Parallel Language* CMU CS Technical Report CMU-CS-92-103.ps January 1992 36 pages

[15] Ian Foster, Carl Kesselman, Robert Olson, and Steven Tuecke *Nexus: An Interoperability Layer for Parallel and Distributed Computer Systems* Technical Memorandum ANL/MCS-TM-189, May 1994 TM189.dvi.Z TM189.ps.Z

[16] Hubertus Franke, *MPI-F An MPI Implementation for the IBM SP-1/SP-2*, Version 1.39, internal document, IBM T.J. Watson Research Center, Feb. 95,

[17] M. Haines, D. Cronk, P. Mehrotra, *On the design of Chant: A talking threads package*. Proceedings of Supercomputing 1994, Washington, D.C. Nov. 1994.