2

Conf-910263--3

UCRL- JC-105425
PREPRINT

JAN 16 1991

BBN TC2000 ARCHITECTURE AND PROGRAMMING MODELS

Eugene D. Brooks III, Brent C. Gorda, Karen H. Warren, Tammy S. Welcome
Massively Parallel Computing Initiative
Lawrence Livermore National Laboratory
Livermore, CA 94550

This paper was prepared for submittal to
CompCon Spring '91, San Francisco, California
February 25-March 1, 1991

November, 1990

Lawrence Livermore National Laboratory

This is a preprint of a paper intended for publication in a journal or proceedings. Since changes may be made before publication, this preprint is made available with the understanding that it will not be cited or reproduced without the permission of the author.

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

UCRL-JC--105425

DE91 006235

BBN TC2000 Architecture and Programming Models*

Eugene D. Brooks III, Brent C. Gorda, Karen H. Warren, Tammy S. Welcome

Massively Parallel Computing Initiative
Lawrence Livermore National Laboratory
Livermore, California 94550

Abstract: The BBN TC2000 is a scalable general purpose parallel architecture capable of efficiently supporting both shared memory and message passing programming paradigms. We describe the TC2000 machine architecture and the programming models which we have implemented on it. The parallel programming models are implemented in a portable manner and will be useful on the scalable shared memory machines we expect to see in the future.

Keywords: Split-join, fork-join, shared memory, message passing, BBN TC2000, scalable multiprocessor

1 Introduction

Microprocessors have made incredible strides in performance in recent years and are beginning to overrun traditional supercomputer performance for scalar dominated application codes. It is expected that supercomputer class vector processing performance will appear in microprocessor form in the next few years. This development is enabling a new breed of supercomputers composed of hundreds, and in some cases thousands, of high performance microprocessors.

The BBN TC2000 is a scalable microprocessor based machine which provides a shared memory facility through a multi-staged interconnection network. It is very similar to the IBM RP3 architecture [1] but is currently commercially available. Because the machine supports both high bandwidth interleaved shared memory and large local memories, it is well suited to supporting both shared memory and message passing programming models.

We have implemented Fortran and C versions of the *split-join* [2] parallel programming paradigm on the BBN TC2000, and have provided an Argonne style message passing library within the split-join programming environment. The split-join parallel programming model is very similar to Harry Jordan's Force [3] and the IBM SPMD [4] programming model, the most significant difference being the support for team splitting and the arbitrary nesting of concurrency constructs. The split-join parallel programming model is implemented with the Parallel C Preprocessor (PCP) for the C programming language and with the Parallel FORTRAN Preprocessor (PFP) for FORTRAN. The Livermore Message Passing System (LMPS) is a message passing library which currently lives within the split-join programming environment, but could stand alone on a message passing machine if this were required.

The split-join parallel programming model is highly portable because a full featured version is easily implemented with a preprocessor and relatively little back end compiler support. An earlier version of PCP has been used on a variety of machines, including Sequent Symmetry, Sequent Balance, Alliant FX/8, SGI, Stellar, and Cray multiprocessors. PFP was written specifically for the large base of FORTRAN users who are participating in the Massively Parallel Computing Initiative at Lawrence Livermore National Laboratory. The PCP and PFP preprocessors have an option of emitting efficient serial code and this has been used to target both multiprocessors and uniprocessors with the same source code. We have found the split-join programming model to be a very good match to the BBN TC2000 architecture. The current areas of active work are extending the implementation of team splitting, which begins to get heavy use as the number of processors available climbs beyond a dozen or so. Users have found that one must exploit nested concurrency effectively if we are to successfully use large numbers of processors on general purpose applications.

The sections of this paper are as follows. The BBN

*Work performed under the auspices of the U. S. Department of Energy by the Lawrence Livermore National Laboratory under contract No. W-7405-ENG-48.

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

TC2000 hardware and capabilities are presented in Section 2. The split-join model, its memory model and the message passing model are described in Sections 3, 4, and 5. Specifics on how the implementation of these models take advantage of the architecture are included. The synchronization primitives offered in PCP and PFP are discussed in Section 6. Section 7 is on the debugging and performance monitoring abilities within the models. Finally the time and space scheduling mechanism used on our machine is described in section 8.

2 TC2000 architecture

The BBN TC2000 [5] is a scalable multiprocessor architecture which can support up to 512 computational nodes. The Motorola 88100 microprocessor is used in conjunction with three Motorola MC88200 chips to provide for a 32K byte code cache and a 16K byte data cache. The data caches are under programmer control with no hardware assistance for maintaining the coherence of shared data.

The processors are operated at clock speed 20 MHZ, providing a manufacturer's rating of 17 MIPS and a peak single precision floating point speed of 20 MFLOPS. Double precision floating point computation runs at a peak speed of 10 MFLOPS, because all data busses in the architecture are 32 bits wide. The size of the main memory on each computational node in our machine is 16 megabytes, although some earlier produced machines only have 4 megabytes.

The processors, with their 16 megabyte memories, are interconnected to each other in a PE-to-PE model by a variant of a multistage cube network [6] which BBN refers to as the "butterfly switch." In the TC2000, 8×8 switch nodes are used to construct the network. The 512 node configuration requires only three stages of switch nodes in the network, leading to a latency of about 2 microseconds for communication between arbitrary processors. This low latency is directly accessible to the programmer, through the shared memory paradigm that the machine provides. The network connections are 8 bits wide and are clocked at 38 MHZ.

The BBN TC2000 supports local memory, shared memory wherein successive cache lines reside on one card, and interleaved shared memory wherein successive cache lines are placed on successive cards and wrap around the machine. The contribution of each node to the interleaved shared memory pool is made at boot time, set via device registers in the interface to the switch which connects the processors. Any number of processors can be configured to contribute to the interleaved shared memory pool and it is useful and convenient to set the number of contributing processors to a prime number to avoid hot spot problems. The rest of the memory in each node can be used for either local memory or non-interleaved shared memory. This division is enforced by the memory management unit attached to the processor and is set at the time an application is run in a completely flexible way.

As noted above, the data caches in the TC2000 are under programmer control. The cacheability of sections of virtual address space can be adjusted at run time with system calls. System calls are also provided to flush regions of virtual address space from the data cache as required. This facility is used in the "data mover" routines of the message passing package to improve transfer rates, but the system call mechanism has too high an overhead to be used for fine grained concurrency control of specific variables. In general, local data is left cacheable and shared data is left not cacheable, with the possible exception of "write once - read many" variables (hot spots) that are efficiently handled as write through cacheable shared memory variables.

Unlike the message passing and SIMD machines which tend to restrict the user to a single programming model, the BBN TC2000 offers a rich architectural structure presenting many possibilities for the language developer. BBN offers their Uniform System [7] for C and Fortran, carried over from earlier machines which lacked hardware support for interleaved shared memory. They also offer a parallel extension of FORTRAN [8] which is a snapshot of the Parallel Computing Forum specification at some point in the past, this specification is still undergoing evolution at the current time. In addition to the BBN supplied programming models, we have made a substantial effort in developing programming models which fit the TC2000 architecture well and are implemented in a manner that preserves portability to scalable architectures possessing shared memory that we expect to see in the future.

3 The split-join model

In the traditional fork-join parallel programming model, a single processor starts the execution of the program and acquires more processors as concurrency is encountered in the code. The fork-join programming model has been quite useful on tightly coupled shared memory machines with relatively few processors, and some architectures such as the Alliant FX/8 and the Convex C2 provide special hardware to make the dispatch of slave processors happen as quickly as possible. Scalable machine architectures are not as tightly coupled and the cost of communication between processors, heavily used in the process of dispatching processors in the fork-join model, is relatively high. The BBN TC2000 is a realistic example of what one might expect in this regard, the latency of a cache hit on local memory is 3 clocks (pipelined at a rate of one per clock) where as the latency of a remote memory reference is roughly 40 clocks (not pipelined). If one must deal with a 40 clock latency for every memory reference required in the code used to dispatch processors, even an efficient spanning tree implementation can have substantial overhead.

In the split-join paradigm we deal with the high cost of processor dispatch, and the high cost of communication between processors, by minimizing their occurrence in the fundamental constructs of the programming model. All of the processors the job will ever acquire are dispatched at the start of the program and are immediately placed under the control of the programmer. This bunch of processors which loosely follow each other through the code is referred to as a *team* of processors. At this level, the programming model is quite similar to Harry Jordan's Force or the IBM SPMD model, except perhaps for the determinism which the user can establish with respect to which processors do what in the split-join paradigm.

Nested concurrency is exploited by the user through the construct called *team splitting*. In this construct, the user explicitly marks off separate blocks of work which can be executed independently of each other (it is assumed that each block of work is itself a job consisting of subtasks which can be executed in parallel), and possibly indicates the relative total amount of work in each block of code. When the team encounters the split blocks, it divides into a number of subteams matching the number of blocks of code and tackles each block of work with new teams having smaller numbers of processors. If the user has provided accurate loading information the processors finish their work at the same time and join back up into the parent team nearly simultaneously. The total number of processors is conserved in the team splitting process, an accurate analogy is that the processors "change sides" becoming members of a new team.

Splitting the size of the team into smaller subteams as nested concurrency is encountered is counter intuitive. The goal, however, in exploiting nested concurrency is to use a *fixed* number of processors more efficiently, not to use *more* processors. The split-join programming model is in some sense the *dual* of the fork-join model. One finds that one can usually accomplish the task at hand with either programming model. The advantage of the split-join programming model is its full featured, bottleneck free, implementation through a highly portable preprocessor.

4 The memory model

In the split-join programming paradigm, three types of memory are required to fully exploit the notion of team splitting. These are:

- memory which is private to a processor, *private* memory,
- memory which is shared among all processors, *shared* memory,
- and memory which is shared among the members of a given team or grouping of processors, but private to it, *teamprivate* memory.

Private memory is implemented on the processor which has access to it. Shared memory is implemented in the interleaved shared memory facility of the BBN TC2000. Teamprivate memory is allocated as an array in the interleaved shared memory, indexed by a *team descriptor* which is unique to a given team.

Team splitting is handled in a block structured way. Each time a processor "changes sides" and becomes a member of a new subteam, it computes a new team descriptor and its position in the new team without accessing any shared memory or synchronization resources. This leads to an efficient bottleneck free implementation of team splitting, the cost of which is completely independent of the number of processors in the team. As the processor computes a new team descriptor, it pushes the old one on a private stack for recovery when it reaches the end of its share of the work in the split block. Since a processor carries the team descriptors of all its antecedent teams on a stack, it has access to the team private memory of a parent team. This can be very useful in a situation where the tasks in the split blocks are to compute some results required by all the members of the parent team, but for which the use of the top level shared memory would pose an access hazard due to nested use of team splitting in a reentrant way.

5 Message passing

There is good deal of interest in message passing machines such as those offered by Intel or NCUBE.

To enable the use of message passing library code within the split-join programming model, and to provide for portability to message passing machines for those users who want to use a strict message passing paradigm, we have developed a message passing library which runs within the PCP/PFP run time environment.

The message calls are patterned after the Argonne message passing package [9], with the addition of broadcast support. The message passing library currently functions at the top most level in the split-join environment, but will be extended so that it can k used at any nesting level of team splitting. This will be useful for those applications which would like to use a message passing library routine deep in the bowels of some highly concurrent application.

6 Synchronization

Barrier synchronization and the notion of locks are provided in the PCP and PFP implementations of the split-join programming model. In barrier synchronization, all of the processors in a given team are forced to wait at the barrier until the last straggler has arrived. A bottleneck free software implementation [10] is used, requiring 30 to 40 microseconds to synchronize 32 processors. The execution time of the barrier scales as the log of the processor count. Each team has its own unique barrier.

The notion of a lock is also provided. A processor attempting to acquire a lock spin-waits until the the lock is unlocked and then indivisibly locks it. It then unlocks the lock, making it available to others when done with the critical region the lock was being used to protect. Locks may be located in the top level shared memory, or teamprivate memory, depending on the scope of the critical region which is being protected.

In the addition to the use of barriers and locks, the user may implement an event by simply spin-waiting on a location in shared, or teamprivate, memory to change. On a machine supporting coherent shared memory caches this is particularly effective and has no negative impact. If the machine lacks this support, as is the case for the BBN TC2000, one must be careful about the possibility of generating adverse impact on available memory bandwidth through the introduction of a hot spot.

7 Debugging and Performance Monitoring

Debugging a parallel program is typically the hardest problem faced by a programmer. Simple tools such as dbx are just not effective in displaying the state of execution. Reproducing complex timings under these conditions makes it very hard to debug.

The models presented here fit transparently on top of the system tools provided by BBN [11]. The **totalview** debugger gives the programmer a view of his PCP/PFP or LMPS source code. Running under the X Window System, the debugger shows each separate parallel stream with its own window and execution control. The programmer may debug a stream by itself or easily control the parallel program in a global manner.

The PCP, PFP, and LMPS programming models also allow a programmer to work with the BBN supplied performance monitoring tool **gist**. With **gist**, trigger points are inserted within the source, the program executed, and the results post analyzed with a graphical analysis tool. The user can view the resulting data from various time scales, from a global timing presentation down to a fine grained specific event timing.

8 Time sharing support

The BBN TC2000, as it arrives from the factory, supports only the notion of space sharing by dividing the processors into *clusters* as users request them. Once all of the free processors are allocated, a new user must wait until a cluster is relinquished before being able to run a parallel program. This is a common situation which occurs on all of the scalable parallel machines commercially available today. It leads to "sign up sheets" to allocate computer time, a throwback to the days we scheduled people onto supercomputers using similar mechanisms in the 50's. This is completely unworkable for a large institutional machine with hundreds of users.

Fortunately, the UNIX operating system runs in every node of the TC2000 and this provides an opportunity for the system support staff to create a *gang scheduler* which runs as a daemon outside of the operating system. When a parallel program starts up it communicates with the gang scheduler to inform it with regard to the number of processors required. The gang scheduler reserves a spatial slot in the machine for the job to run in, attempting to optimally fill any holes which exist within its schedule. The gang scheduler then proceeds to collectively start and stop jobs which contend for processor resources in a way which avoids live lock. The scheduler maintains a notion of fair share delivery of processor resources, and the length of time slices can be adjusted to optimize for interactive response or production throughput as the case may be.

9 Discussion

We have described the BBN TC2000 architecture and the programming models we have developed for use on it by research staff participants in the Massively Parallel Computing Initiative at Lawrence Livermore National Laboratory. These programming models are routinely used by the research staff in a wide range of efforts in computational physics, chemistry, engineering, and graphics applications. These applications include, but are not limited to:

- 3-D finite element neutron transport,
- cold matter Monte Carlo photonics,
- linear algebra,
- hot matter Monte Carlo photonics,
- the solution of Maxwell's curl equations,
- molecular dynamics for tool cutting,
- mixed-zone Eulerian hydrodynamics,
- geophysical fluid dynamics,
- parallelized image rendering,
- lattice-gas hydrodynamics,
- quantum mechanics,
- and plasma simulations for fusion energy devices.

The programming support is highly portable, having its roots in language support developed for earlier bus based shared memory multiprocessors such as the Sequent, Silicon Graphics, Stellar, and Alliant machines.

The key to high performance on the BBN TC2000, and any future scalable system supporting shared memory, is the efficient exploitation of data locality. The split-join parallel programming paradigm implemented via PCP and PFP support the user in pursuing data locality by providing explicit local memory in the programming model and a predictable execution environment wherein processors can be tiled onto a data set in a way which makes maximum use of data locality.

References

[1] G. F. Pfister, et al, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *Proc. of the 1985 International Conference on Parallel Processing*, pp. 764-771, August 20-23, 1985.

[2] E. D. Brooks III, *PCP: A Parallel Extension of C that is 99% Fat Free*, UCRL-99673, Lawrence Livermore National Laboratory, 1988.

[3] H. F. Jordan, *The Force: A Highly Portable Parallel Programming Language*, Proceeding of the International Conference on Parallel Processing, August, 1989.

[4] F. Darema, D. A. George, V. A. Norton and G. F. Pfister, *A single-program-multiple data computational model for EPEX/FORTRAN*, Parallel Computing, April, 1988.

[5] BBN Advanced Computers Inc., Inside the TC2000, Cambridge, MA, 1989.

[6] H. J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing*, 2nd edition, McGraw Hill, New York, 1990.

[7] BBN Advanced Computers Inc., Uniform System Programming in C, Cambridge, MA, 1990.

[8] BBN Advanced Computers Inc., TC2000 Fortran Reference, Cambridge, MA, 1989.

[9] Ewing Lusk, et al., *Portable Programs for Parallel Processors*, Holt, Rinehart and Winston, Inc., San Fransisco, 1987.

[10] D. Hensgen, R. Finkel, U. Manber, "Two Algorithms for Barrier Synchronization," *International Journal of Parallel Programming*, vol. 17(1), pp. 1-17, 1988.

[11] BBN Advanced Computers Inc., Debugging with the Xtra Tools, Cambridge, MA, 1990.

END

DATE FILMED

01 / 31 / 91