DE-FG03-93ER25183
CONF-9606296--1

A CRAY T3D PERFORMANCE STUDY

by

Asha Nallana and David R. Kincaid

CNA-283 May 1996

MASTER

To be presented at the "First Workshop on Numerical Analysis and Applications,"
Rousse, Bulgaria, June 24-27, 1996.

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, 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 product, 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 any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

A Cray T3D Performance Study

Asha Nallana[1] and David R. Kincaid[2]

[1] Interact, Inc., 9390 Research Blvd., Austin, TX 78758, USA
[2] Center for Numerical Analysis, The University of Texas at Austin, Austin, TX 78713–8510, USA kincaid@cs.utexas.edu

Abstract. We carry out a performance study using the Cray T3D parallel supercomputer to illustrate some important features of this machine. Timing experiments show the speed of various basic operations while more complicated operations give some measure of its parallel performance.

1 Introduction

Recently, high-performance computers have become an important tool for obtaining the solution of complex scientific problems. In spite of the enormous advances in performance of machines and method, they fall short of providing computational solutions to many important applications. To successfully solve these problems, one needs an increase in computational power of several orders of magnitude. Since the speed of the fastest processor already approaches the limits set by the laws of physics, such an increase will only be feasible through the integration of hundreds or thousands of powerful processors into a massively parallel computer. In principle, there is no limit to the aggregate speed of parallel computers, although the growing communication requirements limit the useful size for practical computer systems. Parallel computers are also superior to conventional systems if one considers their cost-effectiveness—a parallel machine employing off-the-shelf processors is usually much less expensive than a sophisticated serial computer.

The basic strategy for programming a massively parallel computer system is to assign the work to the appropriate data locations while keeping all the processors busy with the overall goal of solving the problem in the shortest possible time. Thus, the algorithm chosen must be highly local and parallel. The Cray T3D parallel supercomputer system is based upon the multiple instruction multiple data (MIMD) multiprocessor computational model. It also supports the single program multiple data (SPMD) and the single instruction multiple data (SIMD) computational models.

The objective of this research is to become acquainted with the Cray T3D computer and some of the modes of parallel programming available on it. To do this, we perform some numerical experiments and analyze their performance. In the report by Nallana [2], additional details concerning the Cray T3D are discussed.

2 Results

Following the procedures outline in [1], we present the results of some example programs run on the Cray T3D. While this computer supports several styles of parallel programming. we are mainly interested in *data sharing* and *work sharing*. Data sharing distributes data, such as an array, over the memories of the processing elements (PEs) using mostly implicit communication. The goal is to let as many PEs as possible perform operations on their own data rather than going off to another PE's memory to get the data since operations on local data are faster than those on remote data. Work sharing distributes the statements of the application program among the computer's PEs with the goal of executing them in parallel. For instance, iterations of a do-loop can be distributed among the processors. Work sharing provides a combination of implicit and explicit communication. We consider examples divided into two classes—timing and numerical.

2.1 Method of Timing an Operation

We begin with a discussion of a procedure for timing elementary operations. The basic idea is to measure the time `t` that the computer takes to do a large number `n` of the same operation so that the individual operation time is given by `t/n`. First, we set the initial time `t1` before doing any operation, then we perform the operation a large number (`r1`) of times. We measure the new time `t2` after this computation. These two function calls return the floating-point value of the real-time clock (in clock ticks). The difference between the two timings (`t2` − `t1`) is the elapsed time. This elapsed time includes the overhead for the loop control arithmetic which should not be included in the operation time since it is just an artifact of the technique we use to measure the time. To remove the loop overhead, we time the operation once again. This time we start with time `t3`, perform the computation again with a different number (`r2`) of repetitions, and measure the new time `t4`. Since the overhead for both do-loops is the same, the time for the loop control arithmetic is cancelled out by the expression ((`t2` − `t1`) − (`t4` − `t3`)) and we store it in `dtime`. Hence, the variable `dtime` represents the elapsed time for `nrep*dup` executions of the statement containing the operation. Here `nrep` is the number of repetitions in the timing loops. The variable `dup` is the difference between `r2` and `r1` which is the effective number of operations timed. We use (`r1, r2`) as either (16, 8) or (2, 1). We repeat this procedure `nsamp` times and collect `dtime` into `sumtime`. Expression `sumtime` contains the time for performing (`nsamp` ∗ `nrep` ∗ `dup`) repetitions of the operation. Thus, the average time for one execution of the operation is given by `sumtime` divided by `(nsamp*nrep*dup)`. Apart from calculating the average operation time, the code also calculates the average rate of executing the statement in millions of floating-point operations (`mflops`) which is (number of floating-point operations) divided by [(average operation time) ∗ 10^6]. In the following tables, all timing results are given in seconds.

2.2 Timing Examples

Arithmetic Operations. Using the procedure outline above, we begin by timing the basic arithmetic operations of addition, multiplication, and division. Values of the scalar quantities x and y used are 0.0 and 0.1, respectively. Here `(r1, r2) = (16, 2)`. We present the results in Table 1.

Table 1. Arithmetic operations: $x = x \, \mathrm{op} \, y$

	`nrep`	`nsamp`	`dup`
	16384	10	8
`op`	avg. op. time	`mflops`	x
+	3.9707E–08	25.18	1.0000E–01
*	3.9816E–08	25.12	0.0000E+00
/	4.1677E–07	2.40	0.0000E+00

Serial Dot Product. This example calculates the dot product of two vectors $\left(\mathrm{dp} = \mathbf{x}^T\mathbf{y} = \sum_{i=1}^{n} x_i * y_i\right)$ using the routine `SDOT` from the Basic Linear Algebra Subprograms (BLAS). In the code, each of the vector elements x_i and y_i are assigned the value 1.1. Here `(r1, r2) = (16, 2)`. The average operation time is for a single dot-product operation and we count $2n$ floating-point operations per dot-product operation. The results are given in Table 2.

Table 2. Serial Dot Product

n	`nrep`	`nsamp`
16384	50	10
avg. op. time	`mflops`	`dp`
1.2247E–03	26.8	1.9825E+04

Parallel Dot Product. This example calculates the dot product of two vectors but the difference between it and the routine discussed above is that in this one the computation is distributed over p nodes—each doing approximately $1/p$ of the computation. Hence, each node computes a portion of the dot product

$$x_i * y_i + x_{i+1} * y_{i+1} + \cdots + x_{i+k} * y_{i+k} \tag{1}$$

where $k \approx n/p$ and n is the length of the vector. We use the Cray MPP Fortran programming model knows as CRAFT and use some compiler directives such

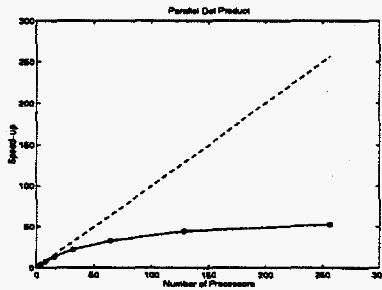

Fig. 1. Speed-up (Table 3)

Table 3. Parallel Dot Product

	n	`nrep`	`nsamp`
	16384	50	10
p	avg. op. time	`mflops`	dp
1	1.2276E–03	26.7	1.9825E+04
2	6.2260E–04	52.6	1.9825E+04
4	3.2108E–04	102.1	1.9825E+04
8	1.6921E–04	193.6	1.9825E+04
16	9.3824E–05	349.2	1.9825E+04
32	5.6204E–05	583.0	1.9825E+04
64	3.7174E–05	881.5	1.9825E+04
128	2.7604E–05	1187.1	1.9825E+04
256	2.3287E–05	1407.1	1.9825E+04

as `CDIR$ SHARED V(:BLOCK)`, `CDIR$ DOSHARED (K) ON V(K)`, `CDIR$ MASTER`, and `CDIR$ BARRIER`. The first one relates to the *data sharing* and it distributes the data among the memories of the various PEs ensuring that each processor works on its own data. The second one relates to *work sharing* and it causes the execution of different iterations of the loops to be distributed over different PEs with the goal of executing then in parallel. Communication among the PEs is mostly implicit. A subroutine call is used to force shared-to-private coercion which allows one to call the BLAS routine `SDOT` on the *local* data. Here `(r1, r2) = (2, 1)`. The results are given in Table 3.

Next in Table 4, we not only double the number of processors used but also double the problem size to give some indication of the *scalability*.

Rather than calling the routine `SDOT`, if we had written the dot-product calculation in Fortran, then various programming tricks would be necessary to obtain optimal performance on the Cray T3D; e.g., a four-way unrolled loop plus read-ahead would improve the number of cache hits. However, this version runs at approximately 15 `mflops` per node while the `SDOT` version goes at approximately 26 `mflops` per node.

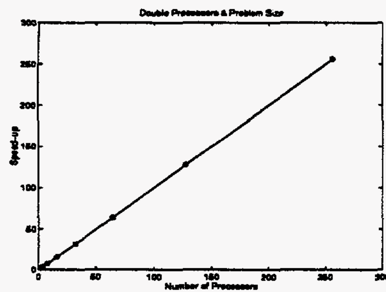

Fig. 2. Speed-up (Table 4)

Table 4. Parallel Dot Product: Scalability

		`nrep`	`nsamp`	
		50	10	
p	n	avg. op. time	`mflops`	dp
1	16384	1.2276E–03	26.7	1.9825E+04
2	32768	1.2281E–03	53.4	3.9649E+04
4	65536	1.2288E–03	106.7	7.9299E+04
8	131072	1.2284E–03	213.4	1.5860E+05
16	262144	1.2281E–03	426.9	3.1719E+05
32	524288	1.2283E–03	853.7	6.3439E+05
64	1048576	1.2281E–03	1707.7	1.2688E+06
128	2097152	1.2284E–03	3414.5	2.5376E+06
256	4194304	1.2283E–03	6829.4	5.0751E+06

Alternatively, one could use the Parallel BLAS (PBLAS) dot-product routine `PDDOT` from ScaLAPACK. The PBLAS are written as an internal component of this library so little effort was made to simplify their use. Also, the PBLAS is not a stand alone library and they require the use of an additional set of routines (BLACS) to handle the data distribution and communication. Consequently, their arguments are a bit difficult to understand if viewed only in the context of the PBLAS. Fortunately, an example of a dot-product program is available at the URL site: `http://www.netlib.org/blacs/BLACS/Examples.html`

Now we present an interesting numerical result. When timing a Fortran parallel code for computing the distributed dot product, we move the global sum to the end so that it is outside the timing loops. Hence, the code does just the multiplications and additions on p processors and, consequently, it is ideally parallelizable since the global sum communications are not timed. (Assuming an efficient global sum, the multiplications and additions should be the most time consuming part of the calculations.) The results are given in Table 5. We note that as the number of processors increases by powers of 2 the relative speed-up

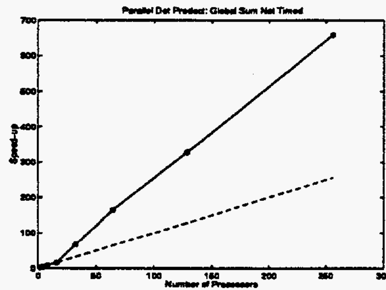

Fig. 3. Speed-up (Table 5)

Table 5. Parallel Dot Product: Golbal Sum Not Timed

	n	`nrep`	`nsamp`
	16000	50	10
p	avg. op. time	`mflops`	`dp`
1	3.2760E–03	9.8	1.9360E+04
2	1.6335E–03	19.6	1.9360E+04
4	8.1685E–04	39.2	1.9360E+04
8	4.0875E–04	78.3	1.9360E+04
16	2.1192E–04	151.0	1.9360E+04
32	4.9309E–05	649.0	1.9360E+04
64	1.9939E–05	1604.9	1.9360E+04
128	1.0034E–05	3189.3	1.9360E+04
256	4.9701E–06	6438.5	1.9360E+04

from $p-1$ to p processors, T_p/T_{p-1}, is approximately 2 for all cases except for 32 and 64 processors which are 4.3 and 2.43, respectively. Consequently, we obtain *superlinear* speed-up as shown in Fig. 3. The reason for this is that as the number of processors increases the amount of work per node decreases until at 32 processors the data just fits into high-speed cache. The data size is 2×16000 and on 32 PEs the data fits in the 8 Kbyte = 1 Kword cache: $32000/32 = 1000$. So for 32 PEs and above, the code is running from cache at approximately 20 `mflops` while for least than 32 PEs it is running from memory at approximately 10 `mflops`.

2.3 Numerical Examples

Next, we discuss the results of some parallel numerical examples; namely, polynomial evaluation and numerical integration.

Polynomial Evaluation. The first numerical code computes a short table of the values of the polynomial $x^3 + 2x^2 + 3x + 4$ for equally spaced argument

values on the interval [0,1]. Here all the processors share the computation for each argument. In this program, the argument values are dependent on the number of processors being used. Here we exploit both the data-sharing and the do-shared loops of the work-sharing method of programming. Special compiler directives **CDIR$ SHARED P_VALUE**, **CDIR$ DOSHARED**, and **CDIR$ MASTER** are used to assign the shared data and direct the parallel do-loops utilizing this data. From a sample output of this program with eight processors, we obtain the results in Table 6.

Table 6. Parallel Polynomial Evaluation

x Value	Polynomial Value	Processor
0.	4.	(PE 6)
0.14285714285714285	4.4723032069970845	(PE 4)
0.2857142857142857	5.0437317784256557	(PE 5)
0.42857142857142855	5.7317784256559765	(PE 7)
0.5714285714285714	6.5539358600583082	(PE 2)
0.71428571428571419	7.5276967930029146	(PE 3)
0.8571428571428571	8.6705539358600578	(PE 1)
1.	10.	(PE 0)

Numerical Integration. This example computes the value of the integral

$$\int_0^{\pi/2} \sin^3(x)dx \qquad (2)$$

using Simpson's rule with a fixed number of partition points. The exact value of the integral is $\int_0^{\pi/2} \sin^3 x\, dx = -\frac{1}{3}\cos x(\sin^2 x + 2)\big|_0^{\pi/2} = \frac{2}{3}$. In this example, the work is shared by all the nodes with each doing the same amount of work. In particular, each node is assigned to work on a different subinterval of the interval of integration obtaining a portion of the integral value. In the code, the outer do-loop uses the number of processors assigned to determine the limits for the inner do-loops which are executed in parallel. Contributions from each node are added as soon as they become available. The compiler directive **CDIR$ MASTER** is used to add all the contributions on a single processing element. In the results in Table 7, the eight processors finish in order 1, 6, 3, 4, 7, 5, 2, 0.

3 Summary

In doing performance tests, one has to think about the behavior of the cache. For example when putting code inside a do-loop for timing purposes, sometimes the vector lengths may exceed the cache size and result in a *cache miss* while other times they may be totally within the cache. If there is a cache misalignment

Table 7. Parallel Simpson's Rule

```
Number of processors = 8
Number of panels = 80
Lower limit = 0, Upper limit = 1.5707963267949001
Value of the Integral = 0.66666666635697946
```

there could be a *thrashing* of the cache. Since any of these situations may effect the timing results, one has to be very careful to determine what actually is going on before trying to analyze the results.

Our experience with CRAFT is that there are definite tricks that need to be used; e.g., shared-to-private coercion. The processors used in the T3D have limited memory bandwidth so the results can be disappointing using CRAFT. On the other hand, the new Cray T3E preserves the macroarchitecture and programming environment of the Cray T3D as well as having faster processing and communication speeds which are coupled with a larger memory. With the Cray T3E, the memory bandwidth is increased and there is a secondary cache with a three-way associativity. Moreover, the CRAFT model is simplified to improve its performance.

Comparing the results from the serial and the parallel dot-product routines, we come to the conclusion that parallel computation gives much improved performance. This is clearly evident from the speed-up graphs. This reiterated the effectiveness of parallelism and of the Cray T3D supercomputer, in particular.

Acknowledgments

This work was supported, in part, by the National Science Foundation grant CCR–9504954, Cray Research, Inc., grant LTR DTD, and the Texas Advanced Research Program grant TARP–266 with The University of Texas at Austin. The work was accomplished with the aid of the Cray T3D at the National Energy Research Supercomputer Center. We thank Alex Kluge and Robert Harkness of the Computation Center at The University of Texas at Austin. Also, Bob Numrich of Cray Research, Inc., was helpful in explaining some of our parallel timing results. In addition, we thank the University of Colorado High Performance Scientific Computing Group for the use of their excellent material [1].

References

1. Fosdick, Lloyd D., Jessup, Elizabeth R., Schauble, Carolyn S. C., Domick, Gitt: *An Introduction to High Performance Scientific Computing.* MIT Press, Boston, 1995. (URL: `http://www.cs.colorado.edu/95-96/courses/materials/hpsc.html`)
2. Nallana, Asha: *Cray-T3D Performance Study*, Report CNA-281, Center for Numerical Analysis, University of Texas at Austin, December 1995.

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