CENTRE DE RECHERCHE EN INFORMATIQUE

FIRST EXPERIENCE ON SHARED-MEMORY MULTI-PROCESSOR MIMD MACHINE — BBN TC2000

Lei ZHOU
<zhou@cri.ensmp.fr>

September 1993

Document EMP–CRI E/176/CRI

First Experience on shared-memory multi-processor MIMD machine BBN TC2000

Lei Zhou
Centre de Recherche en Informatique
Ecole Nationale Supérieure des Mines de Paris

E-mail: zhou@cri.ensmp.fr

Septembre 1992

Abstract

Precise prediction of an application on a parallel computer is pretty important in performance evaluation. In this paper, we have experimented several sample tests on shared-memory multi-processor MIMD machine — BBN TC2000. We have found that the experience obtained from sequential machines can be easily expanded to parallel machine with minor modification of the algorithm. At the same time, several empirical algorithms have been extracted from the test results and proven to be correct by the experimentation.

1 Introduction

While performance evaluation on sequential machines had been the topic of evaluation for years, as parallel machines appear, which quickly replace the sequential machines and become the center of discussio.

We have gathered some experience of prediction on sequential machines, what we intend to do is to get some first-hand experience on parallel machines, i.e., practise some tests and observe the activities of the parallel machines. Because we don't have much time on observing the activities of parallel machines, we only executed a small group of sample programs to get some ideas about execution situation of parallel machines. Our goal is to look into how we could extend our experience obtained from sequential machines to the parallel ones.

When talked about parallel machines, some computer expert made the following observation about parallel machines with lots of processors.

> Suppose you consider a problem to take the form of a box. The volume of the box represents the computation of the problem. The surface area of the box represents the communication required to solve the problem. As you parallelize the problem over 2 processors, it's the same as cutting the box in half - the cut exposes more surface area without increasing the volume. The more you parallelize, the more surface area, or communication, becomes necessary.

Surely, it is the most intuitive description of the overheads caused by parallelizing the sequential programs.

Performance is an important aspect of many programs, those intended for sequential execution as well as those for parallel execution. However, when one sits down to write a parallel

```
/* Multiprocessor "Hello World" program */

#include <us.h>

long  *  nodecount;

PrintHello (dummy, index)
    int dummy, index;
{   printf ("Index=%d: Hello World from node #%d (= h/w node #%x)\n",
         index, PhysProcToUsProc(Proc_Node), Proc_Node);
    xmatomadd31( nodecount, -1 );
    while ( *nodecount != 0 ) LockWait (0);
}

main ()
{   InitializeUs ();
    nodecount = (long *) UsAlloc (sizeof (short));
    * nodecount = TotalProcsAvailable ();
    printf ("\nThere are %d nodes in this cluster\n\n",
           * nodecount);
    Share (& nodecount);
    GenOnI (PrintHello, * nodecount);
}
```

Figure 1: c us example1 source code

program, although with much more difficulties than sequential one, it is with the expectation that the parallel program will execute faster in parallel than not. Furthermore, there is an expectation that increasing the number of processors will result in a commensurate decrease in execution time. Actually, it is generally not the case.

We have a very simple multiprocessor program, see Figure 1, it is a multiprocessor version in C of the "Hello World" program in [4] and is only a little more complicated.

This program causes each processor to print out the greeting message:

```
Index=n: Hello World from node #m (= h/w node #%x)
```

The "Hello World" program uses **UsAlloc** to reserve space in globally shared memory for *nodecount*, a variable used for bookkeeping by the processors. *nodecount* is initialized with the number of processors available to program, a number obtained with via **TotalProcsAvailable**. After using **Share** to propagate the location of *nodecount* to other processors, the program then uses **GenOnI** to generate tasks that print the "Hello" message for each processor. The only tricky part is ensuring that each processor performs exactly one task. In general, without some form of coordination, some processors could get more than one task and others might get none. For this program, the coordination is simple. After printing its message, each processor atomically decrements a counter maintained in globally shared-memory (*nodecount*) with **xmatomadd31** and then waits until the counter indicates that all messages have been printed. This guarantees that no processor finishes its task until all messages have been printed, therefore, all tasks are generated before any processor finishes. The program is shown in Figure 1.

An interesting fact is that if we eliminate the coordination part of the program shown in Figure 1, only three processors participate the work, one gets 6, one gets 4 and one 2. The

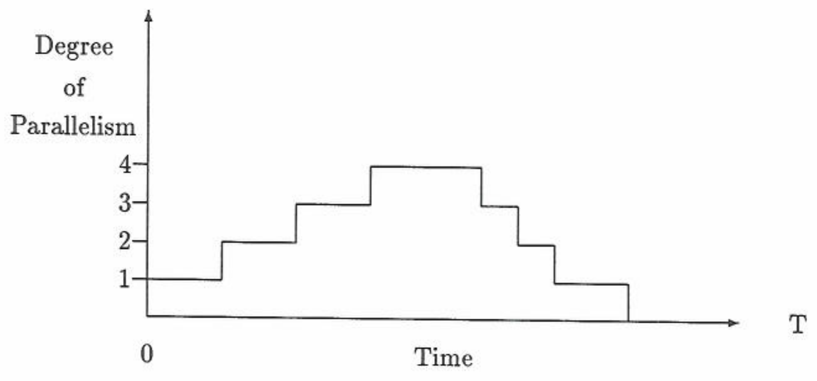

Figure 2: Parallism Profile of an Application

important fact is that it is 12 times faster than the one with coordination. This illustrates that if the user does not have much work to do on a parallel machine, the more processor the worse performance.

2 Background and Terminology

Generally speaking, researchers consider two main degradations of parallelism, *load imbalance* and *communication cost.* The former is due to uneven distribution of workload among processors, and is application-dependent. The latter is due to the communication processing and latency, and it depends on both the application and underlying parallel computer. To give an accurate performance measurement, both of the degradations need to be considered. Load imbalance is measured by *degree of parallilism.*

Definition 1 *Degree of Parallilism: An integer which indicates the maximum number of processors that can be busy computing at a particular instant in time.*

Apparently, degree of parallilism is function of time. Processors can not start to work at the same time. We can imagine that the processors get to work one by one until no more processors available on the machine, and when there are no more tasks, the processors gradually stop working. By drawing the degree of parallelism over the execution time of an algorithm, a graph can be obtained, see Figure 2. We refer to this graph as the parallelism profile.

Definition 2 *Parallelism Profile: The profile of execution time of an algorithm on a parallel machine.*

Communication cost is an important factor contributing to the complexity of a parallel algorithm. Unlike degree of parallelism, communication cost is machine-dependent. It depends on the communication netwotk, the routing scheme, and the adapted switching technique.

In our testing suite, we are considering a relatively simpler case. We now introduce some definitions here.

Definition 3 *Time of Task Tt: The execution time needed to finish one task.*

Of course, we assume that the tasks are identical.

Definition 4 *Time of Interval Ti: The waiting time between the two consecutive tasks on the same processor.*

We have known that all processors are same, so we can equally assume that waiting time is same on different processor.

Definition 5 *Alloted Number of Processor Np: The number of processors available for a certain job, which is alloted before the job gets started.*

Np can be alloted by **cluster** (see [2]) and can be obtained by **TotalProcsAvailable** during the execution.

Definition 6 *Effective Number of Processor Ne: The number of processors that fully participate the job.*

Ne in general case is equal to Np when the job is big enough. But when the job is not big enough, all processors available can not be all exploited, only a small part of them can be used.

Definition 7 *Formal Number of Processor Nf: The processor that only does one task during the work.*

The processor participates the job but it does only one and quits. Because of start-up overhead. In this case, more processors can only deteriorate the performance because the heave cost for start-up overhead.

Definition 8 *Desired Number of Processor Nd: Given the amount of tasks, the number of processors that can be effectively exploited to execute the job.*

That means under Nd, the more processors, the better performance; and beyond Nd, the more processors, the worse performance.

3 The Machine Description

Before we proceed our test program, we briefly introduce the machine — BBN TC2000 and its environments.

3.1 Basic Characteristics of TC2000

The TC2000 computer is a powerful new multiprocessor[1]. It builds upon BBN's experience in design of parallel processor and extends to a state-of-the-art machine. It has the following features:

1. It employs a number of **microprocessors**, each executing individually on the users' tasks in a controlled and coordinated way.
2. It employs **shared memory** to store information. All main memory of the machine is accessible to every processor.
3. The TC2000 processors access the shared memory through an interconnection network called the **Butterfly switch**. The switch provides a fast, efficient and effective access path.
4. The TC2000 design is **modular** and **scalable**. Processors, memory and IO capacitty can be added board by board, as needed by the application.
5. The TC2000 has a **balanced** architecture. The integer computation, floating point computation, memory and input/output capabilities are approximately equal in power.

3.2 Architecture of TC2000

The TC2000 architecture consists of **function boards** interconnected by a high performance **Butterfly Switch**. In addition, the **Test and Control System** (TCS) monitors the entire machine.

Each **function board** contains some or all of the following : a processor, cache and memory management unit, memory, a VMEbus interface, a switch interface, TCS circuitry and power supplies. The **processor** is Motorola 88000 chip group comprised of an 88100 CPU chip and at least two 88200 cache/memory management unit (CMMU) chips. One or two 88200 chip(s) handle instruction reference, and one other 88200 chip handles data reference. The **memory** is dynamic RAM, with parity, addressable as byte, halfword (2 bytes) or word (4 bytes), aligned on boundaries of the size being addressed.

3.3 nX Operating System

It is the operating system base on UNIX that runs on the TC2000 for multiprocess application.

There is a big difference between nX operating system and the one we use on SUN work-station. nX, just as you can imagine, is supposed to be an n-processors operating system, you need to specify how many processors (called nodes in parallel machines) are alloted to a certain application.

3.4 Uniform System Library

The TC2000 hardware and nX operatng system are a foundation on which a variety of software structures may be built. The Uniform System Library contains subroutines that can be used with either C or Fortran programs. The approach to memory management used by the Uniform System Library is based on two principles:

- Use a single address space shared by all processes to simplify programming; and
- Scatter application data across all memories of the machine to reduce memory contention.

The benefit of this strategy is that you can treat all processors as identical workers, each able to do any application task, since each has access to all application data. This greatly simplifies programming the machine, a benefit that far outweighs the modest cost.

The TC2000 multiprocessor can work very efficiently with individual tasks a few milleseconds in length; if necessary, it can also work on tasks in the hundred of microseconds. For shorter tasks, various overheads begin to interfere with good performance.

3.5 Xtra Programming Environment

The X Tools for Runtime Analysis (Xtra) programming environment is a software development environment (based on the X Window System) for debugging, analyzing, and tuning the performance of parallel programs. See [2] for more information. The Xtra environment includes (1) the TotalView debugger; (2) the ELOG library for generating user event logs; and (3) the Gist performance analyzer for analyzing event logs. The Xtra package also includes special macros and the *gisttext* tool for generating text event logs. This package is designed to help users develop parallel programs on the TC2000 system and port serial applications to the parallel environment of the TC2000 system.

3.5.1 TotalView Debugger

The principal advantage of the TotalView debugger over conventional UNIX debuggers (e.g. *dbx*) is that it was designed to debug the multiple processes of parallel programs. Furthermore, since the TotalView debugger runs under the X Window System, its graphic interface is easy to learn and ideal for conveying the concepts of parallel programs. I have not tried my hands on this debugger because of my major prediction work, but some people in newsgroup comp.parallel said this debugger is *very,very* impressive.

3.5.2 The ELOG library and Gist performance analyzer

I have used the other two components of the Xtra environment — the ELOG library and Gist performance analyzer. They were designed to tune the performance of multiprocess programs. The performance analyzer also has a graphic interface that is based on the X Window System.

We need to elaborate this utility because all the data we got for the BBN TC2000 are through this interface.

4 Gist Performance Analyzer

This utility is provided by the manufacturer. We will introduce how to create the events, obtain the data and analyze the them.

4.1 Logging Events with the Event Logging Library

An *event log* is a file containing a record of events for **each process** in your program. Each event recorded for a process contains:

- A time stamp, which is the real-time clock value when the event was recorded.
- An event code, which is an integer identifying the event.
- A user-defined 32-bit data item, which is used by the Gist program in analyzing the log.

In order to create an event log, you complete the following steps:

1. Decide what events should be logged from the program.
2. Understanding the library of event logging routines.
3. Instrument the program with the appropriate event logging routines.
4. Compile and link the program with the appropriate compiler switches.
5. Generate the event log by running the program.

4.1.1 Deciding on the Events to be logged

The event logging routines can be used for two purposes: 1) analyzing the performance of a program and 2) debugging a program. We only need to pay attention to several important points here:

- Log events for the major components of you program, such as procedures, functions and subroutines.
- Log events for any parts of your program that use multiprocess programming.
- If you application requires user input, gather it before you begin logging.

4.1.2 Understanding the Routines

The event logging library, **/usr/lib/libelog.a** consists of six routines. These routines initialize an event log, allocate space for individual processes to record events, specify what to do if a buffer overflows, define events to be logged, log events, and write the recorded events to the event log file. You must link your program to the **libelog.a** library in order to call the routines for the library.

It takes about 6 to 7 microseconds on the TC2000 to log an event. The amount of runtime overhead incurred by the event log instrumentation depends on how you use the event logging routines. The instrumentation in this work uses event-logging macros, it requires that ELOG be defined. If not, program will incur **no runtime overhead** by the presence of the event log instrumentation.

4.1.3 Instrumenting a Program

A sample C program, **simple.c** is located in the **/usr/example/gist** directory. This program generates tasks simply take up time in the processes. You can specify the number of tasks. the size of each task (e.g. the number of iterations of the task), and the number of the event log files.

4.1.4 Compiling and Linking the Program

For C programs, you compile in different ways depending on whether or not you define the symbol ELOG for expansion of the event logging macros. To compile the program with macros, do as follows:

```
cc -DELOG -o simple.c -lus -leleog simple.c
```

where

1. -DELOG defines the symbol ELOG (required when you use the C macros)
2. **-lus** links the prograam to the Uniform Systems library (required when you use the Uniform System)
3. **-lelog** links the program to the event logging library (required all the time).

4.1.5 Initializing the Event Log with giststart

giststart is a program that lets you initialize the event log file from the command line. Use this method when it is difficult or inconvenient to instrucment your program with a call to ELOG_INIT.

4.2 Output of Event Logs with the Gist

If Gist does not provide a function that you need to analyze your event log, you can write the event log to a text file. And then you can analyze the text file with standard operating system utilities, such as **awk, sed**, etc. or write your own program to perform the analysis desired. We use this method in our simple test.

Generally, it has two kinds of output formats, one is brief format, a relatively simpler format, and the other is gisttext format. The two formats will be simply introduced below.

```
0 0 1 200
0 750 3 160
c 1563 3 116
. .... . ...
. .... . ...
```

Figure 3: Brief Format of Gist Output

4.2.1 Brief Format

A text file for an event log in *brief* format contains four columns of information for each event:

1. The process label (in hexadecimal)
2. The time the event occurred, in microseconds
3. The event codes
4. The datum logged with the event

Figure 3 is the standard output of brief format, for this format is less useful than the following, we give up this format.

4.2.2 gisttext Format

A text file in *gisttext* format contains the same seven columns as the output produced by the **gisttext** program. They are:

1. The line type (always E, for Event)
2. The processor and process for the process trace
3. The time the event occurred, in microseconds
4. The elapsed time since the previous event, in microseconds
5. The unique integer(event code) identifying the event
6. A text string identifying the type of event
7. The formatted data item for the event

Figure 4 is the standard output of gisttext format. We will analyze this format in the following section.

4.3 Analyzing Event Logs with the Gist

This section describes how to analyze event logs that were generated by the event logging library introduced in the previous section.

Gist can display more information about each event in a process trace. (All of the information about an event cannot fit in the event box.) Specifically, Gist can display 1) the time the event occurred (in microseconds), 2) the event code, 3) the event name and 4) the data that is logged with the event. Now let us see the Figure 4, for the Node 0, at the 0 microseconds, the work starts and lasts 0, after that there is an interval (waiting time) 186 μsec and worker task #

```
E 0x0                     0      0  1  Start Generator
E 0x0                   186    186  2  Start Worker Task #0
E 0x0                   598    412  3  End Worker Task #0
E 0x0                   625     27  2  Start Worker Task #1
E 0xe                   998    373  2  Start Worker Task #2
E 0x0                  1031     33  3  End Worker Task #1
E 0x0                  1049     18  2  Start Worker Task #3
E 0xe                  1411    362  3  End Worker Task #2
E 0xe                  1432     21  2  Start Worker Task #4
E 0x0                  1455     23  3  End Worker Task #3
E 0x0                  1473     18  2  Start Worker Task #5
E 0xe                  1838    365  3  End Worker Task #4
E 0xe                  1854     16  2  Start Worker Task #6
E 0x0                  1879     25  3  End Worker Task #5
E 0x0                  1898     19  2  Start Worker Task #7
E 0xe                  2260    362  3  End Worker Task #6
E 0xe                  2277     17  2  Start Worker Task #8
```

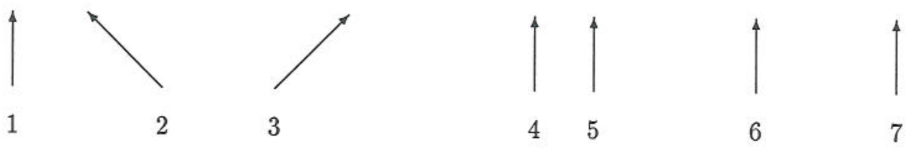

Figure 4: Gisttext Format of Gist Output

0 starts there, after 412 μsec, that is at 598 μsec worker task # 0 finishes. Node 0 begins the work again at 1049 for the worker task # 3 and ends at 1455. And the same thing for the rest of nodes.

So actually, we have every details about node's execution trace, i.e., for each node, how many jobs it has done, its overall start-up time and finish time, even start-up time and finish time for every task it has executed, we know exactly when it starts and finishes and interval between the two consecutive tasks.

It is convenient to write a awk script to analyze the output of gisttext. Figure 5 shows that script which will be used in the next section to analyze the gisttext.

4.4 Summary of the Analyzer

Up to now, we have every information necessary to analyze the program. A figure is worth thousands of words, so we draw a picture to illustrate the complete procedure.

Figure 6 is the complete procedure of the analysis. We have got two sorts of outputs after a bbn-ized program is run. 1) standard result of the program on issue, but it is out of our consideration, we simply ignore this output. And 2) side effect of the program. While the standard result is produced, logging file with events is formed too. As we have mentioned earlier, all information about each node is available in that file.

5 Experimentations

We have instrumented a sample C program, **simple.c** located in the **/usr/example/gist** directory. This program generates tasks simply take up time in the processes. You can specify

```
BEGIN { maxnu = 0 }
# $2 is the processor number
{ c=0 # convert proc number from hex to decimal
        for (i=3;i<=length($2);i++)
        {
        a=substr($2,i,1)
        if (a=="a") a=10
        if (a=="b") a=11
        if (a=="c") a=12
        if (a=="d") a=13
        if (a=="e") a=14
        if (a=="f") a=15
        a=a*1
        c=16*c+a
        }
# $3 is the time stamp
# event #5=start of task, event #7=end of task, c = proc number
# each time a processor finishes a task, compute it's duration
        if ($5==2) { debut[c]=$3 }
        if ($5==3)
{
line += 1 # finished number of the task
duree[c]=$3-debut[c]  # dural time of the task
   proc = c
   if ( length(begin[proc]) == 0 ) begin[proc] = line
   if ( end[proc] < line ) end[proc] = line
   total[proc] = total[proc] + duree[c]
   count[proc] += 1
   if ( maxnu < proc ) maxnu = proc
}
}
END { printf("\t Proc \t Ntask \t First \t Last \t Average Time Per Task\n")
      for ( i=0;i<= maxnu;i++)
printf("\t %d \t %d \t %d \t %d \t %f\n",\
i,count[i],begin[i],end[i],total[i]/count[i])
      for ( i=0;i<= maxnu;i++) {
sum_count += count[i]
sum_squa += count[i]*count[i]
      }
      av_count = sum_count/(maxnu+1);
      sigma = sqrt(sum_squa/(maxnu+1) - av_count*av_count)
      printf("For Task Distribution: %d tasks per processor\n",av_count)
      printf("The absolute standard deviation is %f\n\
              The relative standard deviation is %f %\n",\
              sigma, (sigma*100/av_count) );
}
```

Figure 5: Awk script to analyze the gisttext output

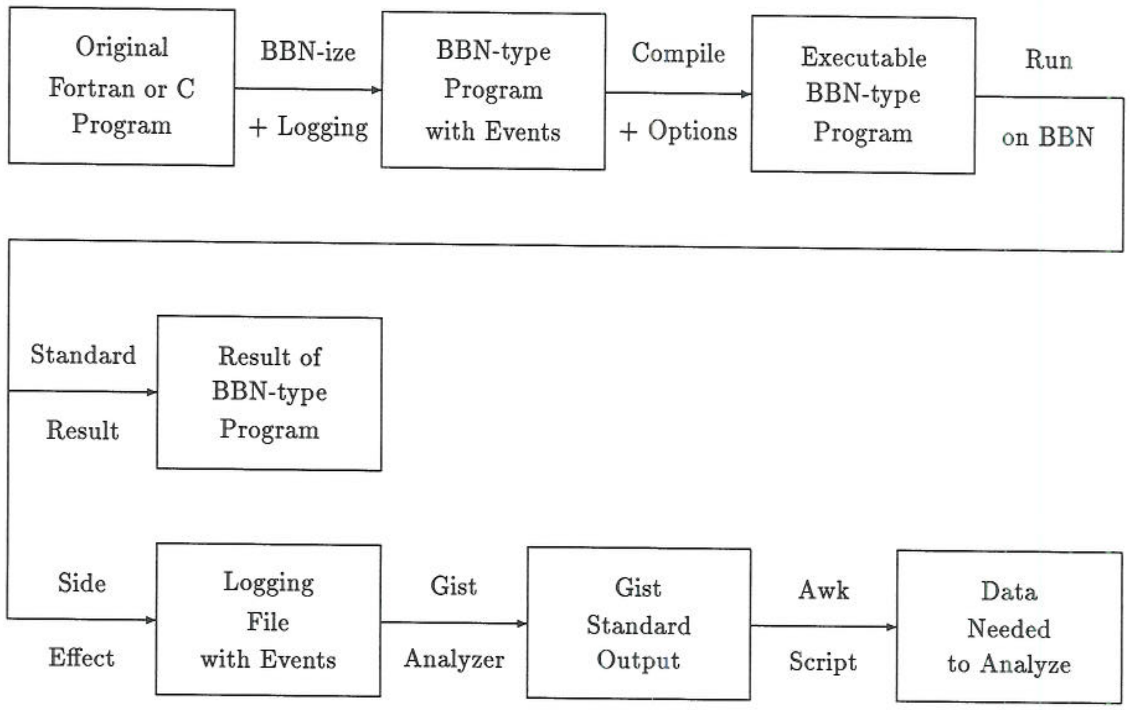

Figure 6: The Complete Procedure of Analysis

Proc	Ntask	StartTime	FinishTime	AverageTimePerTask	AverageInterval
0	1006	4370	4104378	4059.95	15.62
1	995	49942	4105011	4059.36	16.10
2	1000	25758	4101243	4059.93	15.57
3	996	44032	4103198	4059.88	15.61
4	999	31689	4103061	4059.84	15.62
5	1005	11605	4102828	4060.00	10.88
6	1002	19427	4102786	4059.37	15.86
7	997	37893	4101113	4059.60	15.86

Figure 7: Data of Test Program on 8 nodes

Proc	Ntask	StartTime	FinishTime	AverageTimePerTask	AverageInterval
0	1014	4672	4137212	4059.70	15.80
1	998	66113	4136782	4063.27	15.58
2	997	72894	4136147	4059.62	15.87
3	1008	29133	4137078	4059.50	15.85
4	1002	51676	4135487	4059.82	15.85
5	1013	12436	4136878	4060.34	11.18
6	994	86167	4136968	4059.50	15.77
7	989	104488	4135148	4059.86	15.65
8	988	110525	4137049	4059.48	15.97
9	1004	44618	4136409	4059.53	15.97
10	992	92379	4135544	4059.72	16.07
11	1000	58835	4134372	4060.05	15.50
12	1010	21324	4137709	4059.75	15.89
13	990	98309	4133768	4060.38	15.86
14	1006	37256	4137738	4060.02	16.02
15	995	79539	4135101	4060.20	15.75

Figure 8: Data of Test Program on 16 nodes

the number of tasks. the size of each task (e.g. the number of iterations of the task), and the number of the event log files.

5.1 Explanation of Test Program

We implemented the **simple.c** program to obtain the data for each processor.

We don't need to present all data obtained during the experimentation. We only present the three represetative ones, when number of processors is 8, 16 and 24 respectively.

See Figure 7, Figure 8 and Figure 9, we have to take notice that the first processor is numbered 0 instead of 1.

5.2 Startup Time

From the data we presented above, we can put all start-up times in a picture, as you can see in Figure 10. From the picture, we see that start-up times almost form a line for the start-up.

Definition 9 *Start-up line: The line formed by start-up points of all processors.*

Proc	Ntask	StartTime	FinishTime	AverageTimePerTask	AverageInterval
0	1021	4305	4166433	4059.61	16.93
1	990	135410	4170263	4059.78	15.84
2	982	167444	4169620	4059.96	15.59
3	1010	51743	4168428	4060.09	15.85
4	985	155121	4169089	4059.24	15.87
5	1020	13455	4166355	4060.80	10.68
6	995	114219	4169168	4059.70	15.64
7	1008	59747	4167856	4059.90	15.62
8	1002	83067	4166893	4059.93	15.76
9	986	148870	4167895	4060.04	16.06
10	981	170496	4168890	4059.83	16.02
11	1000	90978	4166817	4060.05	15.81
12	988	141954	4168847	4060.13	15.69
13	999	98534	4170290	4060.11	15.74
14	983	161295	4167995	4059.74	16.27
15	991	128607	4167651	4059.65	16.09
16	1012	44208	4168585	4059.95	15.54
17	1017	21664	4166695	4059.92	15.83
18	1016	29221	4169859	4059.64	15.81
19	1004	75595	4167422	4059.80	15.74
20	1006	67424	4167617	4059.83	15.92
21	993	121806	4168712	4059.55	15.90
22	1014	36647	4169501	4059.80	16.01
23	997	106553	4170137	4059.96	15.87

Figure 9: Data of Test Program on 24 nodes

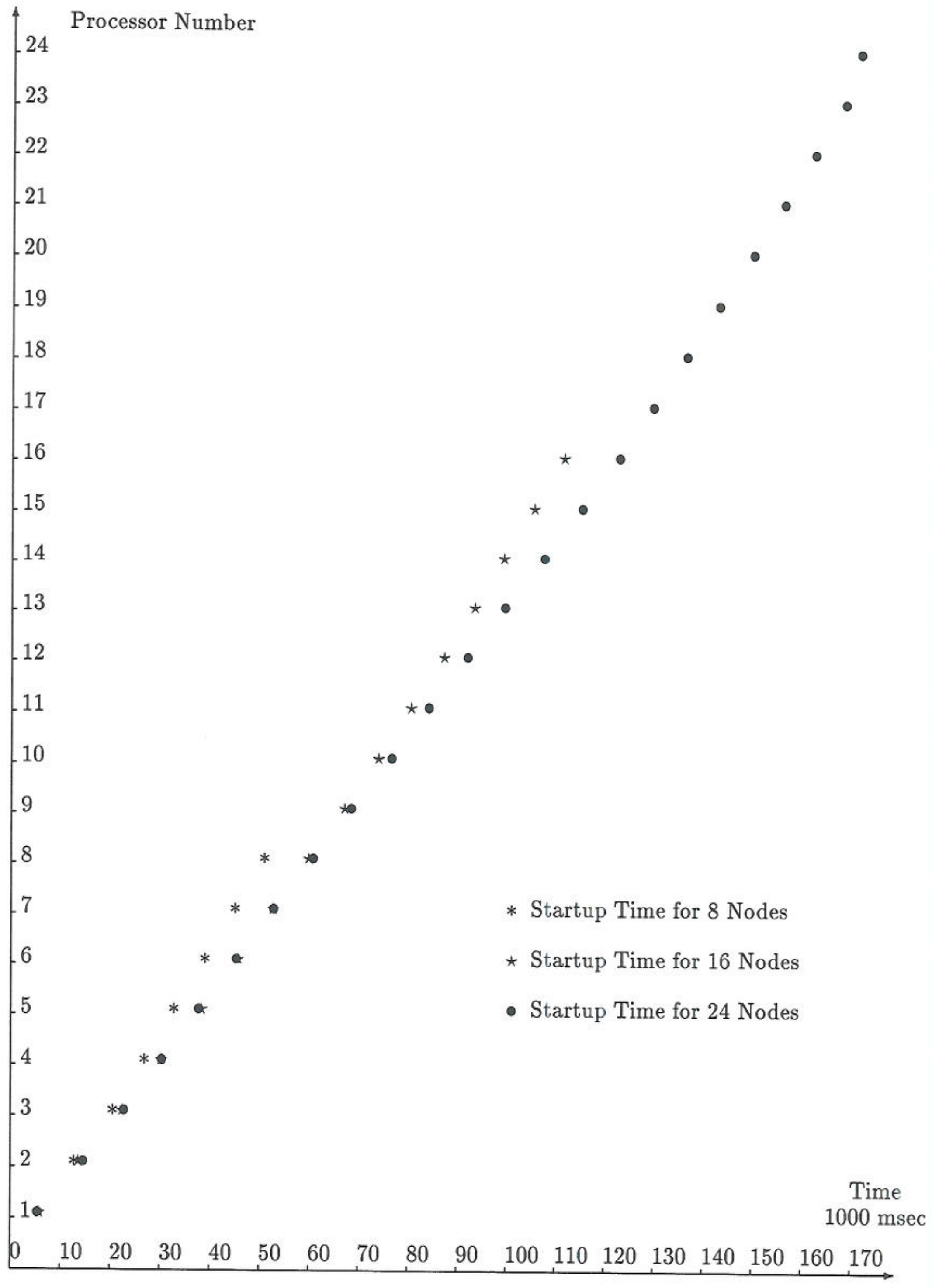

Figure 10: Startup Time for 8, 16 & 24 Nodes

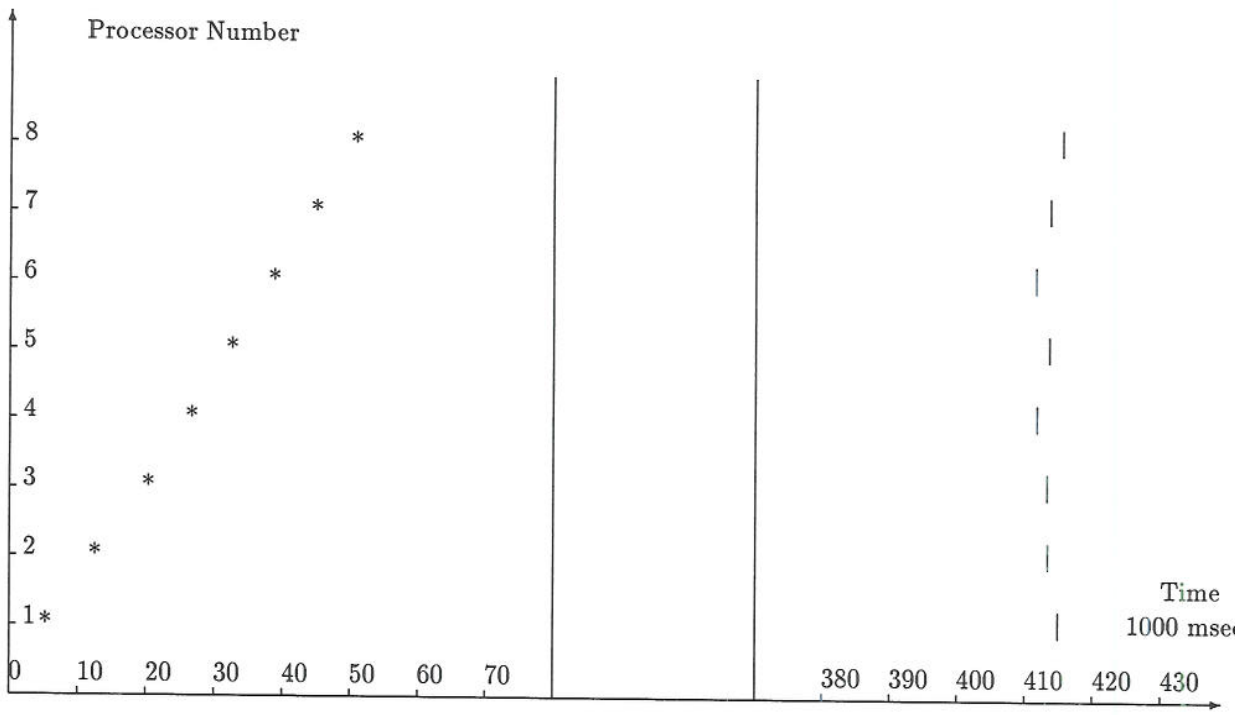

Figure 11: Startup Time and Finishing Time for 8 Nodes

Because we can suppose that start-up makes a straight line which starts from the origin, in order to facilitate the stating, we introduce the following definition:

Definition 10 *Start-up angle α: The angle formed by start-up line and processor-axis.*

So tgα has the physical meaning, the time to start another processor, so it introduces another definition:

Definition 11 *Start-up Slope Ta: The time needed to start up a processor.*

Here, from the data presented, we can conclude that start-up slope for the BBN TC2000 is about 7300 msec/node.

5.3 Finish Time

Now let us turn our attention to the finishing time of processors. We reuse the data for 8 nodes.

From Figure 11, we can observe that there won't be any idle processor (node) when there is a task in the waiting queue. i.e., the time between the node which finishes the job the first and the one which does the last is equal or smaller than average task time. So from the observation, we can simply assume that difference between the processor which finishes the work first and the one which does the last is exactly one task time Tt. At the same time, because we are only concerned with the surface surrounded by the start-up line and finish time, so we could sort the finish time in ascending order and we can also assume that finish time forms a line and difference between the processor which finishes the work first and the one which does the last is exactly one task time Tt. See Figure 12 is a theoritical model for the parallel execution.

We introduce another definition:

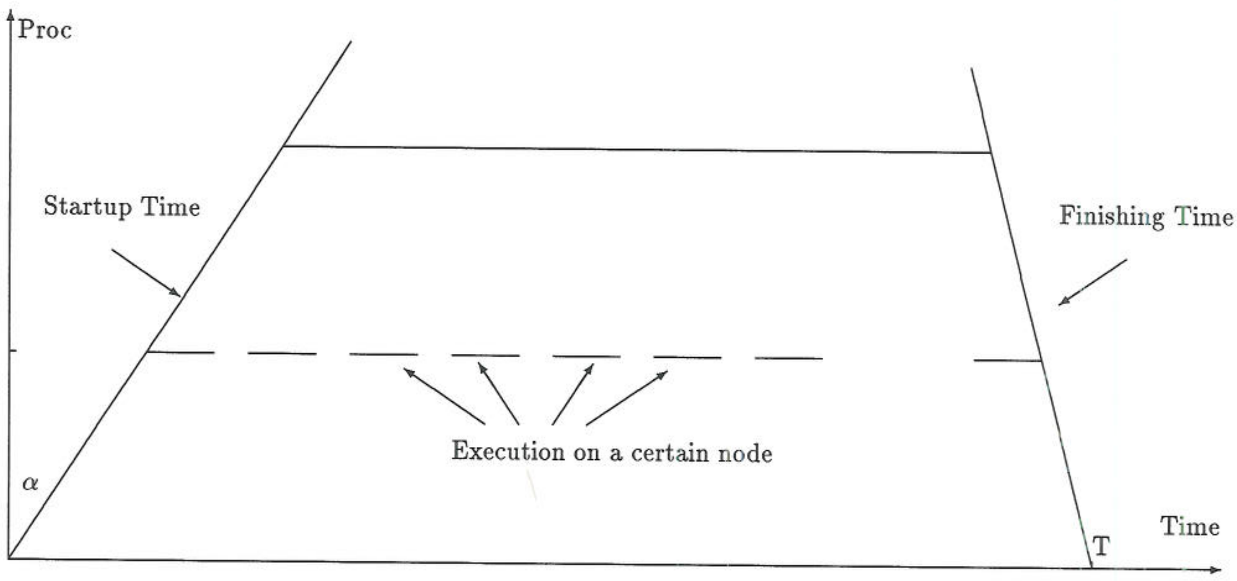

Figure 12: Startup Time and Finishing Time of Theoretical Model

Definition 12 *Finish line: The theoretical line which simulates the curve formed by finishing points according to ascending order. The difference the processor which finishes the work first and the one which does the last is exactly one task time Tt.*

From this later assumption, we can get the following algorithms for parallel machines.

6 Algorithm for Parallel Machines

Before we introduce our algorithms for the parallel machines, we summarize the assumptions we have made earlier.

1. Start-up is a straight line from the origin.
2. Start-up slope is a constant for a fixed BBN TC2000 machine.
3. Finish time form a line
4. the difference between the processor which finishes the work first and the one which does the last is exactly one task time Tt.

With all these assumptions, we describe the symbols we will use in our algorithms.

Np Number of processors available

Ne Effective Number of processors which participate

Nt Number of tasks

Tt Time for a task

Ti Time of interval or waiting time.

Ta Start-up slope tg(α)

Algorithm for Trapezoid Form

Now we are considering a general case, i.e., when the parallism profile takes the shape of trapezoid. We assume that all processors effectively participate the work, i.e. the number of tasks is greatly larger than the number of processors — $Nt \gg Np$; so all nodes participate i.e. Np = Ne and we assume that amount of time spent on a task and amount of waiting time be identical.

Under the above assumptions, for a node j, it has Nj tasks to execute, it has $(Nj - 1)$ intervals, one less than the number of tasks, the time is $Tj = Nj \cdot Tt + (Nj - 1) \cdot Ti = Nj \cdot (Tt + Ti) - Ti$.

Now consider that the trapezoid as a whole, we can imagine that $\sum_{j=1}^{Np} Nj = Nt$ and $\sum_{j=1}^{Np} 1 = Np$. So the sum of execution times for all the processors is:

$$\begin{aligned}
\sum_{j=1}^{Np} Tj &= \sum_{j=1}^{Np}(Nj \cdot Tt + (Nj - 1) \cdot Ti) \\
&= \sum_{j=1}^{Np}(Nj \cdot (Tt + Ti) - Ti) \\
&= (\sum_{j=1}^{Np} Nj) \cdot (Tt + Ti) - (\sum_{j=1}^{Np} 1) \cdot Ti \\
&= Nt \cdot (Tt + Ti) - Np \cdot Ti
\end{aligned}$$

It means that there are Nt tasks and because of there are Np processors, so there are $Nt - Np$ intervals, Np is largely less than the number of overall tasks, i.e., $Np \ll Nt$. Because the height is an unit, so from this point of view, we think the surface of the trapezoid is:

$$S1 = Nt \cdot (Tt + Ti) - Np \cdot Ti \tag{1}$$

The standard formula to get the surface is $Surface = Height \cdot (UpperLength + LowerLength)/2$. For the this trapezoid , we have $Height = Np$, which is the number of processors alloted (of course, $Nt \gg Np$); $LowerLength = T$, which is overall finishing time; and $UpperLength = T - Tt - Np \cdot Ta$

Here we have

$$\begin{aligned}
S2 &= Height \cdot (UpperLength + LowerLength)/2 \\
&= Np \cdot ((T - Tt - Np \cdot Ta) + T)/2
\end{aligned} \tag{2}$$

Both $S1$ in Equation 1 and $S2$ in Equation 2 denote the same amount of trapezoid surface from two different points of view, so they are equal: $S1 = S2$. Hence we have:

$$Nt \cdot (Tt + Ti) - Np \cdot Ti = Np \cdot ((T - Tt - Np \cdot Ta) + T)/2 \tag{3}$$

And this equation can be simplified to:

$$T = \underbrace{\frac{Np \cdot Ta}{2}}_{part1} + \underbrace{\frac{Nt}{Np} \cdot (Tt + Ti) - Ti +}_{part2} \underbrace{\frac{Tt}{2}}_{part3} \tag{4}$$

This equation has intuitive meaning: part 1 in Equation (4) is half of start-up line projected on the time-axis, part 2 is the average length of upper length and lower length and part 3 is half of finish line projected on time-axis. As shown in Figure 13.

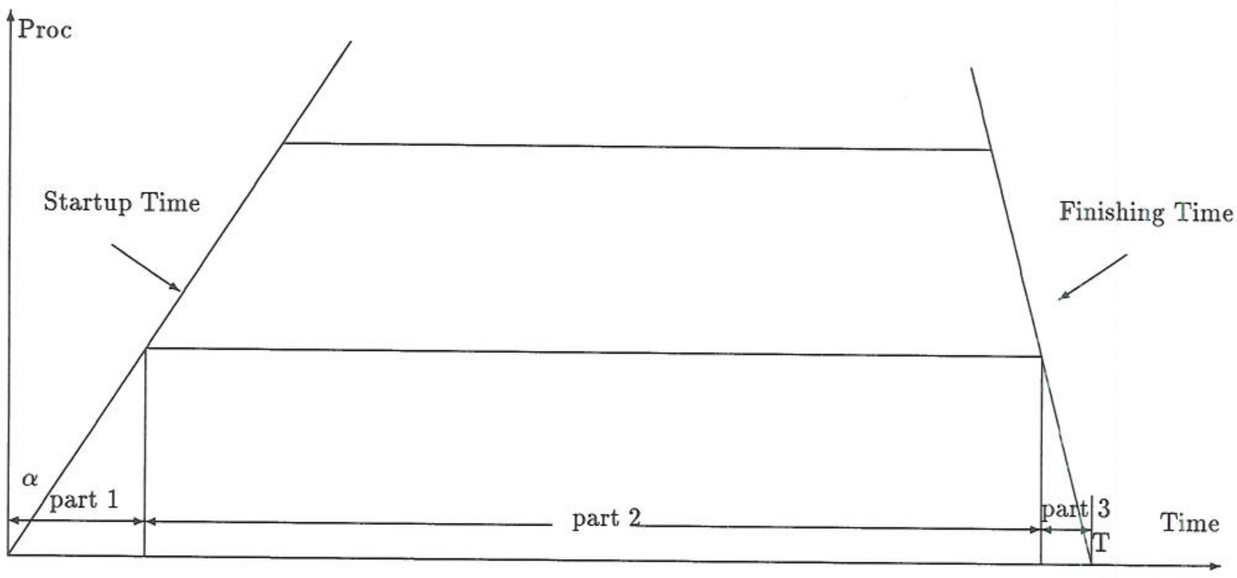

Figure 13: Startup Time and Finishing Time

Algorithm for Triangle Form

Now we consider the critical case, i.e., when the parallism profile takes the shape of triangle. First 12 processors have been allocated, and then we gradually increase the number of tasks while keeping the task size same.

Here we need to introduce another definition:

Definition 13 *Critical Finish Time Tc: The finish time when parallism profile takes the shape of triangle and all processors alloted are effective if given the fixed number of processors.*

Which means that if we increase a little more work, parallism profile will be the trapezoid, and if we decrease a little, not all processors are effective.

Now we use the different ways to compute the surface of the triangle. From the standard formula, the surface of critical triangle S3 is:

$$S3 = \frac{Tc \cdot Ne}{2} \tag{5}$$

and from the point of view of execution, the surface of critical triangle S4 is:

$$S4 = Nt \cdot (Tt + Ti) - Ne \cdot Ti \tag{6}$$

Certainly, $S3 = S4$ because both present the same amount of surface. We have:

$$\frac{Tc \cdot Ne}{2} = Nt \cdot (Tt + Ti) - Ne \cdot Ti \tag{7}$$

$$\tag{8}$$

while

$$Ta = \frac{Tc - Tt}{N} \tag{9}$$

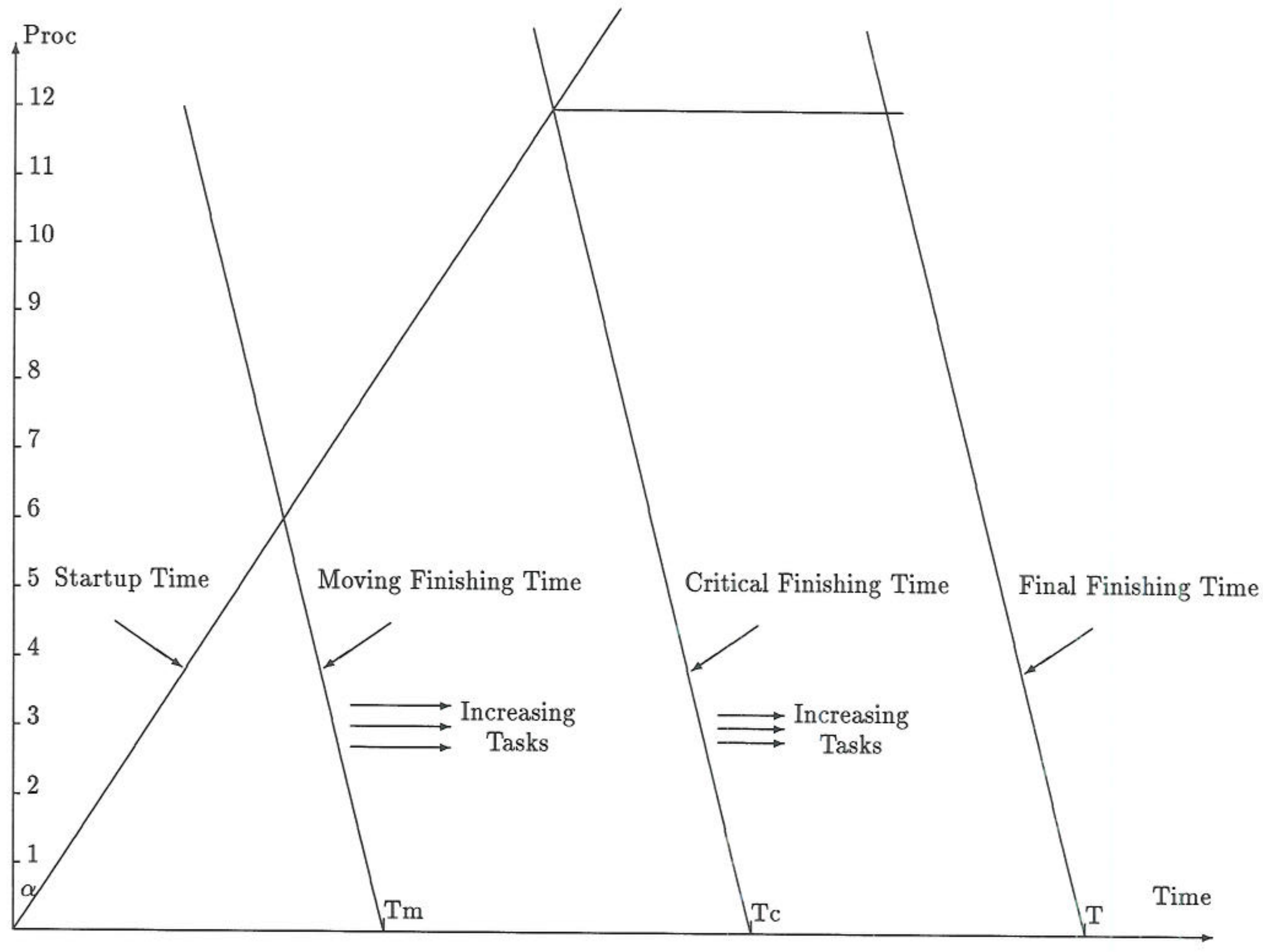

Figure 14: Triangle Form

Np	Experiment	Formula	Difference
8	4105011	4106084	0.03%
16	4137738	4135614	0.05%
24	4170290	4164614	0.14%

Table 1: Trapezoid Formulas Verification

After deduction, we have:

$$Tc^2 + (2 \cdot Ti - Tt) \cdot Tc - 2 \cdot Nt \cdot (Tt + Ti) \cdot Ta - 2 \cdot Ti \cdot Tt = 0 \tag{10}$$

Because, the total finishing time can not be negative, so the negative time does not make any sense, we neglect negative solution. Finally, we have unique solution:

$$Tc = \frac{-(2 \cdot Ti - Tt) + \sqrt{(2 \cdot Ti - Tt)^2 + 8 \cdot Nt \cdot (Tt + Ti) \cdot Ta + 8 \cdot Tt \cdot Ti}}{2} \tag{11}$$

Besides the final finishing time, we can also compute the desired number of processors.

$$Nd = \frac{(Tc - (Ti + Tt))}{Ta} \tag{12}$$

7 Verification of Formulas

In order to check our empirical formulas are adequate for the real world, we use the data obtained in Section 5 to compare.

Verification for Trapezoid Form

We only choose three cases when Np is 8, 16 and 24.

See Table 1, we can find that the difference is too satisfactory to believe. In effect, it is because task size is too small, so for a larger task size, bigger difference is expected.

Verification for Triangle Form

Now let us consider triangle form. See Figure 15, we gradually increment the total tasks number, `40*p` means the total task number is 40 times alloted number of processors, (in the example, it is 12). And then we have measuredFT which stands for measured finishing time, EffectP for effective number of processor, FormalP for formal number of processor, FTFormula1 for finishing time when using formula 1, which is Tc is Equation 11, FTFormula2 for finishing time of formula2, which is T in Equation 4, DesiredP for desired number of processor, which is Nd in Equation 12.

Maybe the Figure 15 is a little bit misleading, let us just extract some data from that figure and compare these data in Table 2.

8 Discussion

During the experimentation, we have observed something that we can not explain.

TotalTask	MeasuredFT	EffectP	FormalP	FTFormula1	FTFormula2	DesiredP	InterTime	TaskTime
40*p	14538	2	4	13596.48	44861.88	1.86	16.62	9.80
50*p	18051	2	7	16430.20	45348.77	2.25	20.99	9.88
60*p	18781	3	3	17861.55	45629.07	2.44	20.61	9.79
70*p	26292	3	9	24038.95	47112.18	3.29	37.49	9.76
80*p	24156	2	10	20097.85	46113.62	2.75	19.25	9.61
90*p	24190	3	9	22836.00	46786.21	3.12	23.24	9.89
100*p	28127	3	9	24610.48	47267.56	3.37	24.66	9.96
110*p	27247	3	9	24928.46	47356.88	3.41	22.34	9.95
120*p	27716	4	8	26430.93	47797.92	3.62	23.31	9.96
180*p	34605	5	7	34003.53	50412.94	4.65	26.41	10.30
190*p	35368	5	7	34663.08	50671.40	4.74	26.12	10.02
200*p	38801	4	8	33858.11	50355.09	4.63	22.90	9.85
210*p	39325	5	7	36478.14	51408.86	4.99	25.98	10.23
220*p	42143	4	8	35797.46	51126.75	4.90	23.51	9.77
230*p	39076	4	8	36471.70	51404.83	4.99	22.59	10.45
240*p	40526	5	7	38637.38	52334.78	5.29	24.95	10.59
250*p	43103	5	7	38797.55	52405.13	5.31	23.96	10.44
260*p	42669	6	6	41258.31	53531.47	5.65	26.86	10.55
270*p	46249	5	7	40884.16	53355.00	5.60	24.79	10.58
280*p	50867	7	5	49381.67	57743.07	6.76	39.24	10.54
290*p	48599	6	6	43308.60	54521.31	5.93	26.06	10.89
300*p	49289	6	6	46043.14	55918.41	6.30	29.64	10.73
310*p	46818	6	6	44695.75	55218.43	6.12	26.24	10.58
320*p	48375	7	5	49060.71	57558.68	6.71	32.09	10.88
330*p	53975	8	4	51719.31	59090.78	7.08	35.49	10.83
340*p	51953	7	5	50070.46	58129.85	6.85	31.26	10.87
350*p	53042	5	7	48394.65	57185.44	6.62	27.60	10.63
360*p	56590	7	5	53865.54	60384.62	7.37	34.83	11.22
370*p	94815	4	8	43990.61	54857.99	6.02	19.93	9.95
380*p	61136	8	4	58249.23	63194.87	7.97	40.31	10.72
390*p	55396	8	4	56179.14	61838.79	7.69	35.19	11.05
400*p	59231	7	5	53208.79	59979.76	7.28	29.55	10.88
410*p	59816	5	7	51022.77	58676.29	6.98	25.62	10.65
420*p	65658	9	3	62359.30	66027.05	8.54	41.84	11.07
430*p	67213	6	6	57051.22	62401.07	7.81	32.85	10.39
440*p	63268	8	4	57691.89	62820.62	7.90	32.37	10.85
450*p	73360	11	1	72766.81	74069.86	9.96	55.66	11.59
460*p	73287	10	2	73045.55	74301.15	10.00	54.63	11.66
470*p	69647	9	3	65186.14	68085.26	8.92	40.64	11.02
480*p	75210	11	1	74671.37	75673.21	10.22	55.09	11.30
490*p	74598	10	2	68117.77	70318.88	9.32	43.02	11.08
500*p	78052	11	1	74955.26	75913.97	10.26	52.94	11.27
510*p	76573	11	1	76685.03	77413.55	10.50	54.63	11.27
520*p	77898	11	1	74682.08	75678.12	10.22	50.15	11.15
530*p	86900	12	0	86509.37	86584.54	11.84	69.18	11.53
550*p	102887	12	0	99080.03	99935.51	13.56	91.88	10.18

Figure 15: Triangle Formulas Verification

Ne	Experiment	Formula	Difference
3	24190	22836	5.93%
4	27716	26430	4.87%
5	34605	34003	1.74%
6	42669	41258	3.42%
7	50867	49382	3.01%
8	53975	51719	4.36%
9	65658	62359	5.29%
10	73287	73045	0.33%

Table 2: Trapezoid Formulas Verification

8.1 Processor

There is always at least one processor which needs less waiting time considerably and at the same time does more job than the rest. As you can see from Figure 7 and Figure 8, the 5th (logical number) processor in both experiments. We have asked the machine owner in Toulouse, they replied that all processors are identical. We once tried to locate physical number of the fastest processor in one suite, and tried the same experimentation again without the that processor. It turned out that another processor emerged as the fastest one.

8.2 Intertask Time of Triangle Form

We can not explain why waiting time of triangle is going up steadily while incrementing the number of tasks.

We are guessing that more communication cost is needed when more tasks are involved.

9 Summary

From the experiments we have done, we can see that:

1. Time for each task is identical on each processor.
2. Waiting time between the consecutive two tasks is identical on each processor. i.e., the interval between the adjacent two tasks is same.
3. Starting time for nodes forms a line.
4. There won't be any idle processor (node) when there is a task in the waiting queue. i.e., the time between the mode which finishes the job the first and the one which does the last is equal or smaller than average task time.

As we have claimed several times, this experimentation is the first step towards the parallel machine to see whether it is feasible to use our experience on parallel machines. Therefore all these observations consolidate the idea that performance on parallel machines can be predicted and assure that the experience we got from the sequential machines can be expanded to parallel ones. This method looks very promising.

10 Acknowledgements

I would to express my special thanks to CERFACS in Toulouse for letting us practice the experimentations on BBN TC2000 machine and providing helps when we were stuck. Also I am grateful to Ms. Danny Durand for initiating this work and guiding me all along the experimentation. I also find that the discussion with François Irigoin and Fabien Coelho is very enlightening.

References

[1] *"Inside the TC2000 Computer"*, Revision 1.0 BBN Advance Computers Inc. 1990

[2] *"Using the Xtra Programming Environment"*, Revision 2.0 BBN Advance Computers Inc. 1990

[3] *"Uniform System Programming in C"*, Revision 2.0 BBN Advance Computers Inc. March 1990

[4] Brian W. Kernighan and Dennis M. Ritchie *"The C Programming Language"*, Second edition 1988 Prentice Hall

[5] R.H. Thomas and W. Crowther *"The Uniform System: A approach to runtime support for large scale shared memory parallel processors"*, ICPP'88