OREGON STATE UNIVERSITY COMPUTER SCIENCE DEPARTMENT

A Benchmark Suite for Parallel Processors: Part I

Tieh-Jun Sun
Dr. Ted G. Lewis
Department of Computer Science
Oregon State University
Corvallis, OR 97331-3902

89-70-3

A Benchmark Suite for Parallel Processors :

Part I

by

Sun, Tieh-Jun

A Research Project Submitted in Partial
Fulfillment of the Degree of Master of Science

Major Professor
Dr. Ted. G. Lewis

Department of Computer Science
Oregon State University
Corvallis, OR. 97331-3902

April 3, 1989

TABLE OF CONTENT

Abstract

Advanced computer architectures are centered around the parallel computer systems. This project is focused on the experiment on two parallel computer architectures : Sequent Balance 21000 shared memory multiprocessor and Cogent XTM distributed system.

A set of benchmark programs are implemented using "C" language and Dynix parallel programming library on the Sequent system and "Linda" parallel programming primitives on the Cogent XTM.

In this paper, the following seven programs are discussed and the benchmark results are presented.

1. Parallel Matrix Multiplication
2. Disk File I/O
3. Saturating
4. Parallel Enumeration Sort
5. Memory Transfer
6. System Math Functions
7. Linpack routines

1. Introduction

A benchmark is a real program which is run on real data on the actual machine. Commonly, a benchmark program is a starting point in the performance evaluation process. As such, they must be widely accepted and be treated as "standard" [Martin and Riganati, 1988].

Since there is no official standard for benchmarks [Nicholls,1988], a thorough evaluation study using benchmark programs should involve the careful selection of a series of benchmarks typical of the job characteristics at the installation [Deitel, 1984].

The National Bureau of Standard (NBS) held a workshop in June 1985 to discuss the techniques for measuring and evaluating parallel computers. At the conclusion of the workshop, a committee was formed to support, promote, and critique the NBS approach to a benchmarking collection [Martin and Riganati, 1988].

Many benchmark programs have been published in the NBS system :

1. LINPACK - The benchmark program from Argonne National Lab. It is a set of routines that analyze and solve systems of linear equations and linear least squares problems (both single precision and double precision).

2. STONES - WHETSTONES and DHRYSTONES.

3. LIVERMORE - LIVERMORE loop program from Livermore National Lab.

4. ETA - Scientific codes for vector machines, contributed by M. Humphrey of ETA.

5. FERMI - Codes used in an equipment procurement at Fermi National Accelerator Lab.

6. JRR - Numerical problem set for parallel and vector machines by John Rice. This is not a executable version but is for illustrating programming techniques.

7. MENDEZ - Raul Mendez's benchmarks used on Japanese machines.

8. NASAAMES - The NASA kernal benchmark from NASA/Ames research center.

NBS invites organizations to use the programs currently in the system and to contribute other benchmark programs. Some organizations also have evolved their own benchmark suite. For example, BYTE has collected a set of programs that gauges system performance for microcomputers. The classical programs like Dhrystone, Whetstone, Sieve, and Linpack are included in BYTE benchmark suite [BYTE, 1988].

By reviewing the NBS benchmarks and some recently published benchmarks, we realized that they are originally designed for single-processor computers. JRR is the only one that illustrates the technique for parallel or vector computers but it is not an executable code. The BYTE new benchmark set is designed and implemented for personal computers.

In 1987, Levitan proposed the following tasks to be used as the beginning of a performance suite for evaluating the communication structure of parallel computer architectures [Levitan, 1987] :

1. Broadcasting - A processor sends a message to all other processors in the system.

2. Reporting - Gather information about the state of the network to a central location.

3. Extreme finding - Find the largest or smallest value from a set of items distributed one to a processor in the network.

4. Sorting - Sort a sequence of elements using parallel processors.

5. Packing - It is a task of moving data from higher numbered processors to lower numbered processors.

6. Saturating - This is the task of each processor sends a message to every other processors. It is a many to many message passing.

7. MST - Compute the minimum spanning tree of a graph.

In this study, we implemented a benchmark suite consisting of 6 of the 7 programs suggested by Levitan. Packing was removed from the suite because some parallel computers do not allow a programmer to control which processor each process executes on.

Four additional programs are implemented in the suite in addition to Levitan's programs :

1. Parallel Matrix Multiplication - Multiply two matrices using multiple processors.

2. Disk File I/O - Have multiple processes issue I/O to a common disk file to test for I/O bottleneck.

3. Memory Transfer - Move data from one memory location to another memory location.

4. Mathematical Function - Test the performance of system built-in mathematical functions.

Finally, to have a complete benchmark suite, we included the classical benchmark programs: Dhrystone, Whetstone, Sieve and Linpack routines.

We tested and evaluated the 15 programs by running them on the Sequent Balance system and the Cogent XTM workstation. The Sequent benchmark programs are written in C programming language with Dynix system parallel programming library. The benchmark programs for the XTM workstation are written via C++ programming language and use LINDA parallel programming primitives.

Performance Metrics

The performance improvement of parallel algorithm PA over sequential algorithm SA can be measured by the speedup of PA over SA. Formally, if SA takes Ts time units to execute and PA takes Tp(i) time units to execute using i parallel processors the speedup of PA over SA is defined as:

$$Sp(i) = Ts/Tp(i)$$

All benchmark results are computed by taking the average run time of 10 program executions measured in seconds.

2. Hardware and Software environment

2.1 SEQUENT BALANCE 21000 system [Sequent, 1985]:

The BALANCE system is a multiprocessor, a computer that incorporates multiple identical processors (CPUs) and a single shared memory. The BALANCE CPUs are general-purpose, 32-bit microprocessors. Following are the characteristics of BALANCE architecture :

a. *True multiprocessor.* The BALANCE is a true multiprocessor computer system not an array processor.

b. *Tightly coupled.* All processors share a single pool of memory, to enhance resource sharing and communication among different processes.

c. *Common bus.* All processors, memory modules, and I/O controllers plug into a single high-speed bus, making it simple to add processors, memory and I/O bandwidth.

d. *Symmetric.* All processors are identical, and all processors can execute both user code and kernal (operating system) code.

e. *Transparent.* Programs written for a single-processor system can run on a BALANCE system without modifications for multiprocessing support. Processors can be added or removed without modifying the operating system or user applications.

f. *Dynamic load balancing.* Processors automatically schedule themselves to ensure that all processors are kept busy as long as there are executable processes available. When a processor stops executing one process (e.g., because that process is finished or is waiting for I/O operation), it begins executing the next available process in the system-wide run queue.

g. *Shared memory.* An application can consist of multiple processes, all accessing shared data structures in memory.

h. *Hardware support for mutual exclusion.* To support exclusive access to shared data structure, the system includes one or more set of 16K user-accessible hardware locks.

The BALANCE system can include from 4 to 40 processors and can be configured with 4 to 28 MB of memory. In addition, each CPU has 8 KB of local RAM and 8 KB of cache RAM, both of which greatly reduce the number of times the processor must access system memory.

BALANCE computers run DYNIX operating system, a version of UNIX 4.2 bsd that also supports most utilities, libraries, and system calls provided by UNIX system V. The BALANCE language software includes multitasking extensions to C, PASCAL, and FORTRAN. The DYNIX parallel programming library includes routines to create, synchronize, and terminate parallel processes from C, PASCAL, and FORTRAN programs.

The DYNIX parallel library includes three sets of routines: a microtasking library, a set of routines for general use with data partitioning programs, and a set of routines for memory allocation. The three sets of routines are declared in two header files : ***microtask.h*** and ***parallel.h***.

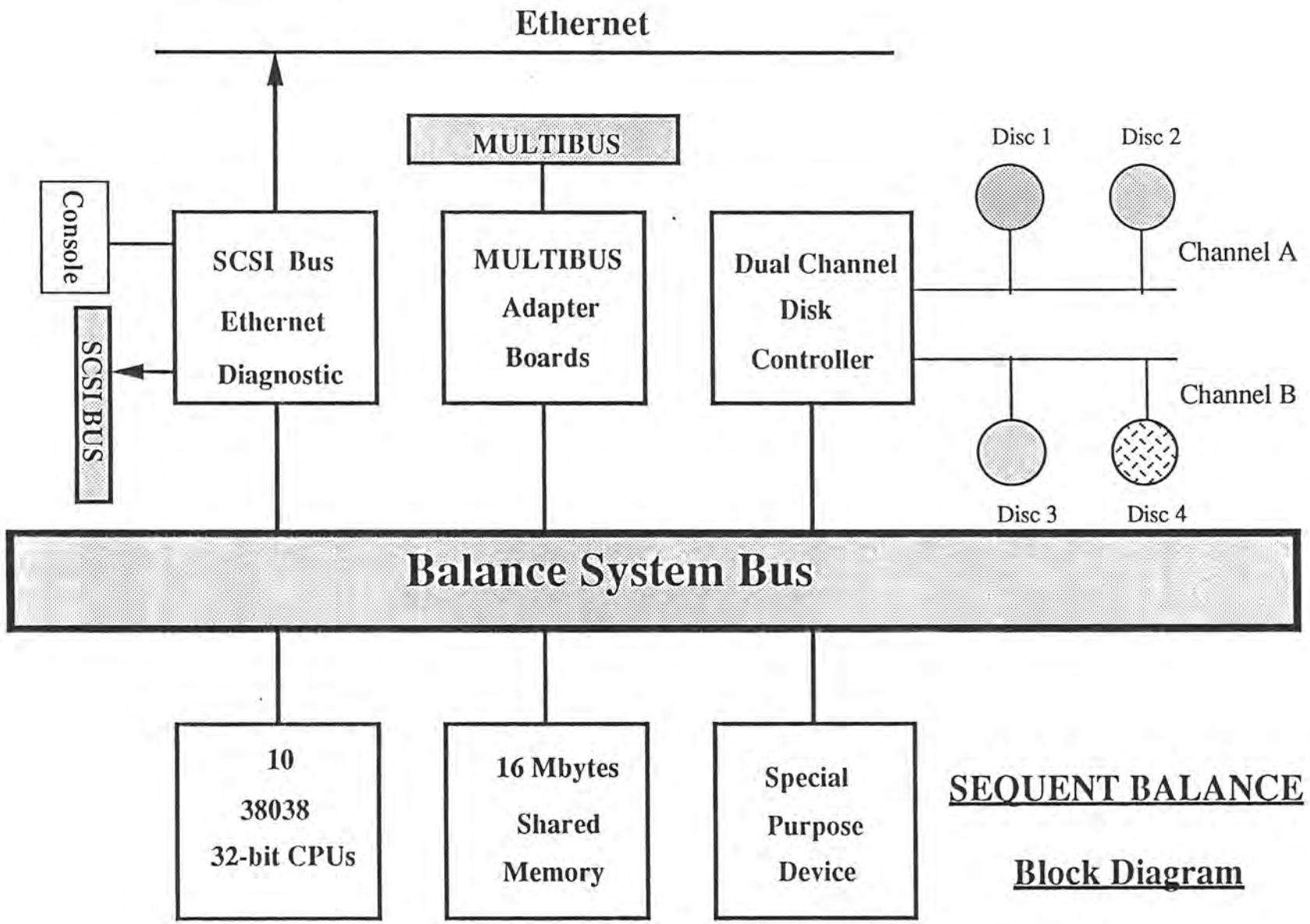
Two multitasking programming methods can be applied to most of the applications :

1. **Data Partitioning** - Involves creating multiple, identical processes and assigning a portion of the data to each process. Data partitioning is appropriate for applications that require loops to perform calculations on arrays or matrices. Data partitioning is done by executing the loop iterations in parallel.

The **m_set_proc(nprocs)** initializes a specified number of processes, and **m_fork**() executes the child processes in parallel. When the processes have been completed, the parent process calls **m_kill_proc**() to terminate the forked processes. The processes can be scheduled statically or dynamically. If the programmer knows that the computing time is approximately the same for each iteration of a loop, then static scheduling can be used. The static scheduling simply divides the loop iterations evenly among the processes. Dynamic scheduling can be applied when the computing time is expected to vary for each iteration of the loop. With dynamic scheduling, the loop iterations are treated as a task queue, and each process removes one or more iterations from the queue, executes those iterations, and returns for more work. The task queue can be implemented by using **m_next**() routine.

2. Function Partitioning - On the other hand, this method involves creating multiple unique processes and having them simultaneously perform different operations on a shared data set. This method is appropriate for applications that include many unique subroutines that need to be executed in parallel.

In this case, A lock will be always required to protect a critical region to prevent multiple processes modifying the same data set at the same time. We can use **s_init_lock**, **s_lock**, and **s_unlock** to initialize, lock, and release a lock. The process synchronization can be done by using the library routines **s_init_barrier** and **s_wait_barrier** which initialize a barrier and cause processes to spin until all related processes arrive at the synchronization point.



SEQUENT BALANCE

Block Diagram

2.2 COGENT XTM workstation [Cogent, 1988]:

The XTM Modular Supercomputer is a flexible, incrementally expandable architecture. The initial version allows configuring systems with a wide range of processing, user interface, and input/output capabilities. Systems are configured using the following modules and cards :

Workstation

The workstation module is housed in a cabinet about the size of an Apple Macintosh II. The workstation is attached to an external display, keyboard and mouse. The cabinet contains dual INMOS T800 transputers connected with transputer links. Each T800 transputer includes a 32-bit integer CPU, a 64-bit floating point unit that operates in parallel with the CPU, 4 KB of static memory, 4 MB of main memory, and four DMA ports into the communication network.

Each workstation also contains input/output devices including serial ports, 90 MB or 190 MB Winchester disk, and 800 KB floppy disk drive.

Resource Server

The Resource Server module contains a 16-slot backplane. The slots are connected with a high performance 32-bit communication bus and crossbar switch. The Resource Server may be populated with any combination of Compute and Communication cards.

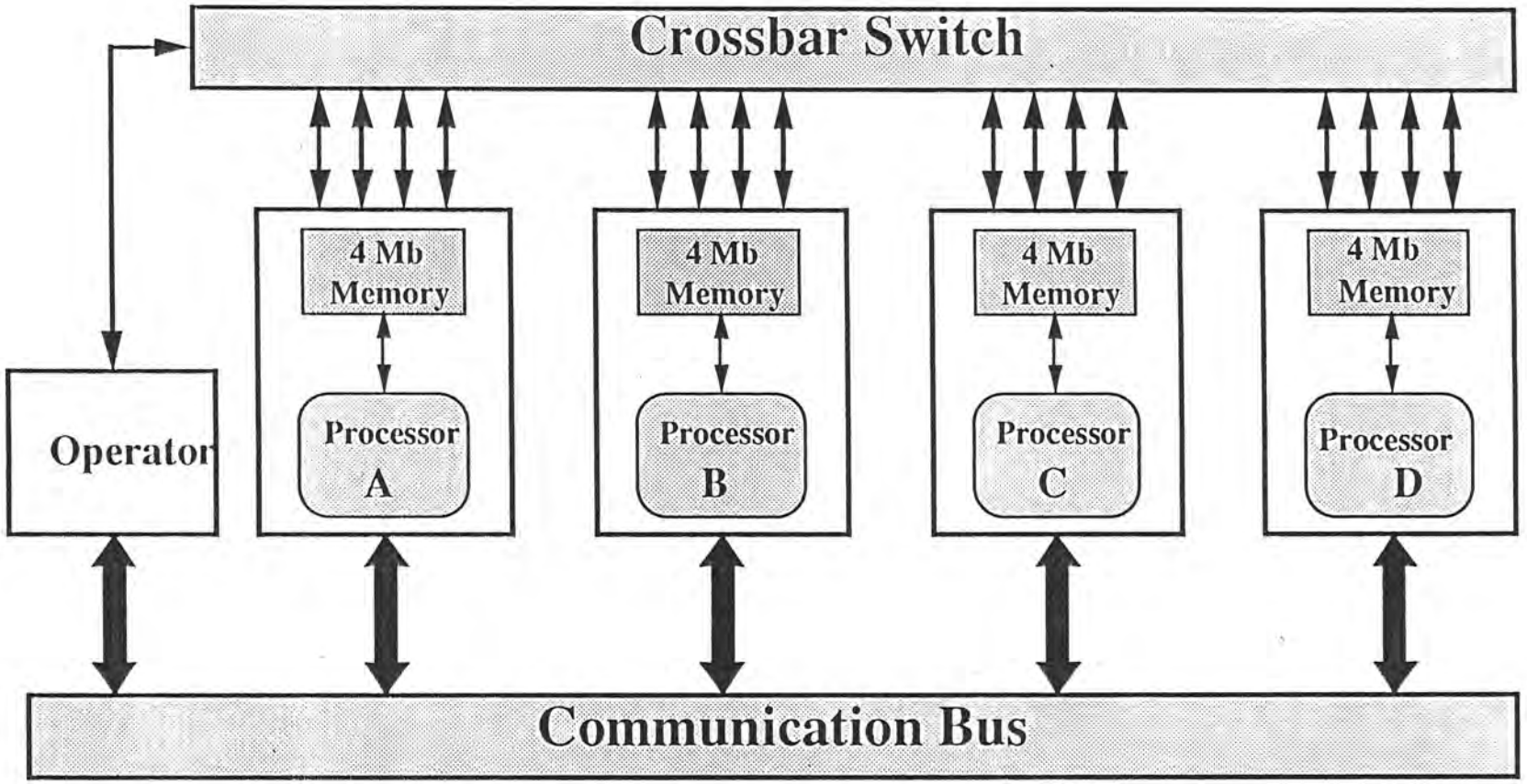
Compute Card

A Compute Card occupies a single Resource Server slot and holds two T800 processors. Each processor has 4 MB of memory and interface to the LindaBus parallel communication bus. Transputer links are connected to a crossbar switch in the Resource Server, supporting communication between individual processors.

Communication Card

A Communication Card occupies a single Resource Server slot. It multiplexes the parallel bus and transputer links onto a full duplex 100 MB/s fiber optic datalink, and it features an on-board T800 gateway processor. This bandwidth and processing capability combine to support flexible configurations of Workstations and Resource Servers.

The XTM workstation series offers modular expansion from a low cost workstation containing 2 processors to 334 processors supercomputing system offering over 500 MFLOPS peak performance, with the same set of software running on all configurations.



Cogent XTM communication system

XTM workstation runs XTMOS parallel operating system which is very compatible with UNIX. C++ , which is a superset of C, is the primary programming language for programming the XTM workstation. It is anticipated that system will offer FORTRAN, ADA, LISP, PROLOG and other traditional languages.

The Cogent software approach emphasizes a new operating system concept developed at Yale University by David Gelernter, called Linda [Gel, 1988], this parallel programming environment has been implemented for both distributed systems and multiprocessor.

The basis of Linda is logically-shared tuple memory through which processes communicate. Processes never exchange messages directly. Instead, a process with data to communicate adds it as a tuple to the shared tuple space, and a process expecting data seeks it in that tuple space. The tuple spaces hold two kinds of tuples. Process tuples are under active evaluation; data tuples are passive. To run a Linda program, we enter a process tuple into tuple space; it creates other process tuples. The process tuples execute in parallel and exchange data by generating, reading and consuming data tuples.

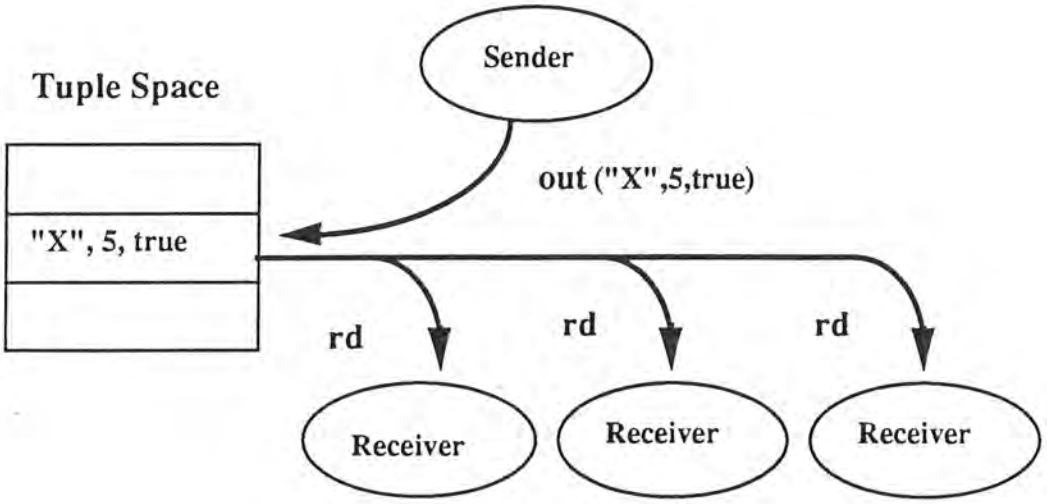
Four principal operations can be performed on tuple space :

1. *OUT* is an operation that creates a tuple and places it in tuple space.

2. *IN* is the reverse of *OUT*. It specifies a tuple that it desires, in the form of a template, and the computer matches the template against all tuples existing in the tuple space. If a matching tuple is found, it is removed from the tuple space and is returned to the requesting process. When no tuple is found to match, the requesting process is suspended until another process, through an *OUT* operation, creates a matching tuple.

3. *READ* operation is identical to *IN* except that the matching tuple is not removed from the tuple space.

4. *EVAL* is a specialized form of *OUT* that creates an active tuple instead of a data tuple.



An example of message broadcast using Linda

2.3 CASE Tools for Parallel Programming

2.3.1 ELGDF design language

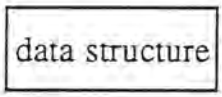
In June 1988, Hesham El-Rewini introduced a design language, ELGDF, that allows representation of a wide variety of parallel programs [Hesham, 1988]. The syntax is graphical and hierarchical to allow construction and viewing of programs. ELGDF language facilitates describing parallel programs in a natural way for both shared memory model as well as message passing model. The syntax poses high level structures such as loops, branches, fans, and replicators.

The ELGDF serves as the foundation of a parallel programming environment under development at Oregon State University. The complete syntax of ELGDF helps the program designers to deal with parallelism is the manner most natural to the problem at hand. It also helps as a way to capture parallel program designs for the purpose of analysis such as task scheduling. Thus the goal of ELGDF is two fold : 1) A program design notation and CASE tool, and 2) A software description notation for use by automated schedulers.

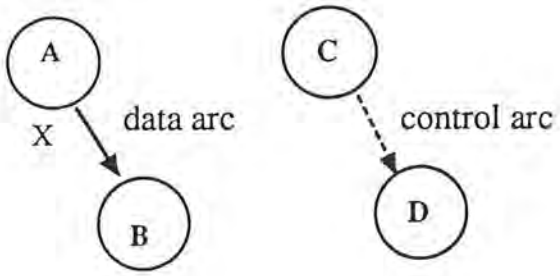
The basic constructs of ELGDF are briefly described as follows :

(1) Nodes - A node is represented by a "bubble". A simple node consists of sequentially executed code and is carried out by one processor. A compound node is a decomposable high level abstraction of a subnetwork of the program.

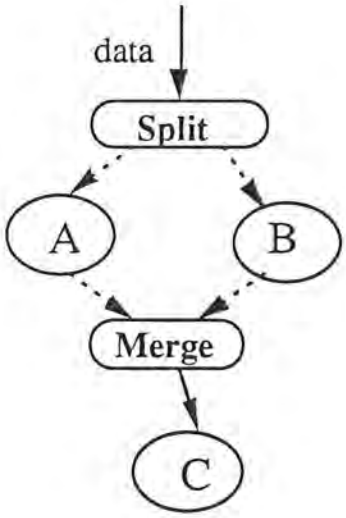
(2) Storage - A storage cell represents the data structure to be read or written by a simple node.



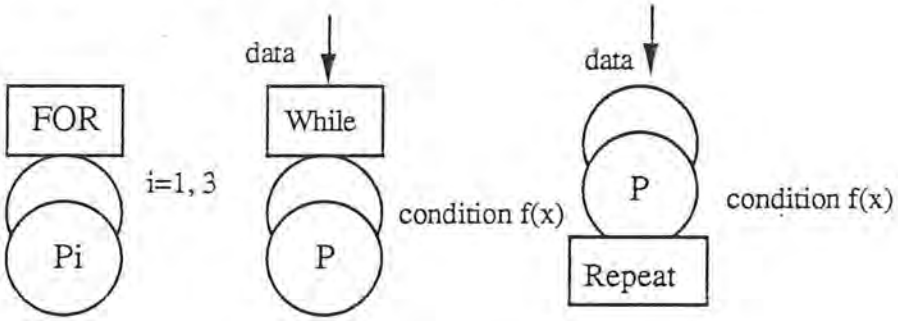
(3) Arc - A control arc expresses sequencing and transfer of control among nodes. A data arc can carry data from one node to another or can connect a node to a storage construct.



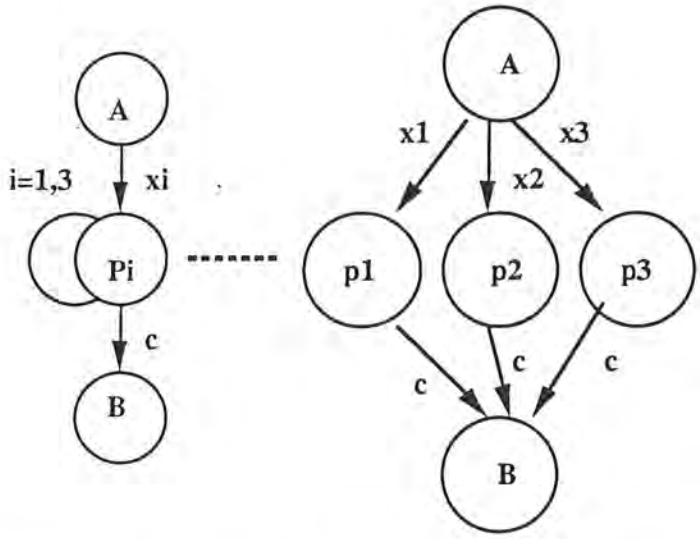
(4) Split and Merge - Split and Merge are special purpose simple nodes for representing conditional branching.



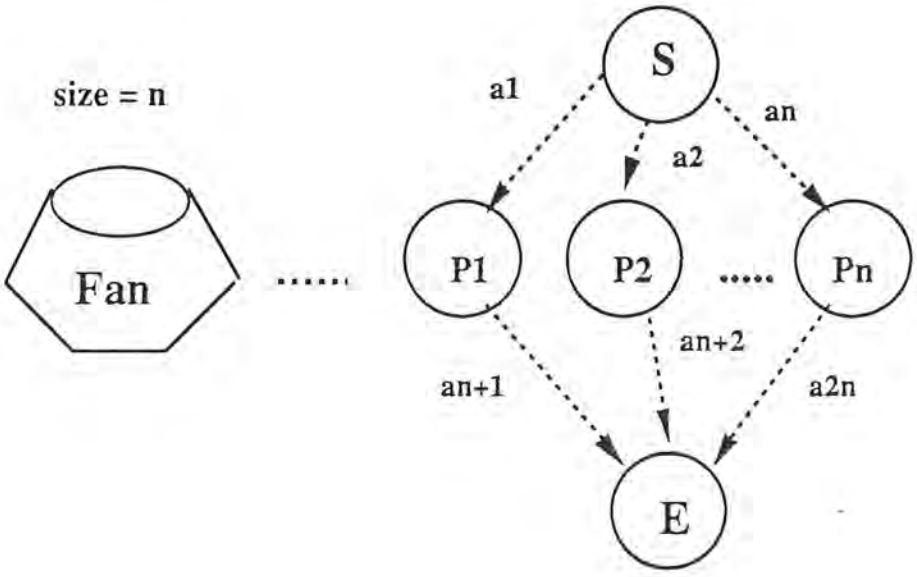
(5) Loops - a loop can represent FOR, WHILE, or REPEAT.



(6) Replicators - A replicator allows program designers to represent concurrent loop iterations compactly.



(7) Fan - Fan is another form of convenient structures in ELGDF, in which designer can use the skeleton prepared as fan and then define its constituents later. A fan of size N is composed of a start node (S), N parallel nodes Pi , i=[1..], 2N control arcs and a end node (E).



2.3.2 DSH task scheduler for parallel systems

DSH (Duplication Scheduling Heuristic) was proposed by Boontee and Ted Lewis in 1987 and their idea is implemented by Julie Chiu at Oregon State University recently [Boontee, 1987].

Distributing parallel tasks to as many processors as possible tends to increase the communication delay, which contributes to overall execution time. In short, there is a trade-off between maximizing parallelism and minimizing communication delay. We call this the max-min problem.

DSH duplicates tasks to reduce the cost of communication. The duplication of a task is called the task duplication concept (TDC). The TDC solves the max-min problem by duplicating the task nodes that influence the communication delay.

On a shared memory multiprocessor computer system, one process can not communicate to others directly. In fact, the shared memory is served as a communication center where processes send messages to and retrieve messages from. The Sequent Balance 21000 system is a typical shared memory system. So we simply ignored the communication delay when applying DSH to the tasks executed on the shared memory system.

Julie's program takes the execution time of each task (node) as input and produces a Gantt chart which tells us how the multiple processes are scheduled and also indicates the total time units for the whole task. We can then compare the time units from the Gantt chart with the actual program execution time. See the matrix multiplication program and enumeration sort program as examples.

3. Benchmark suite

3.1 Parallel Matrix Multiplication

Matrix multiplication is one of the most common operations in many application programs. Furthermore, it is common to have several of these operations in the middle of a large interactive process. In that case, any speedup in the multiplication process can save considerable execution time. The implementation of parallel version matrix multiplication as a benchmark program is to reflect the computation abilities of multiprocessor machines.

Implementation

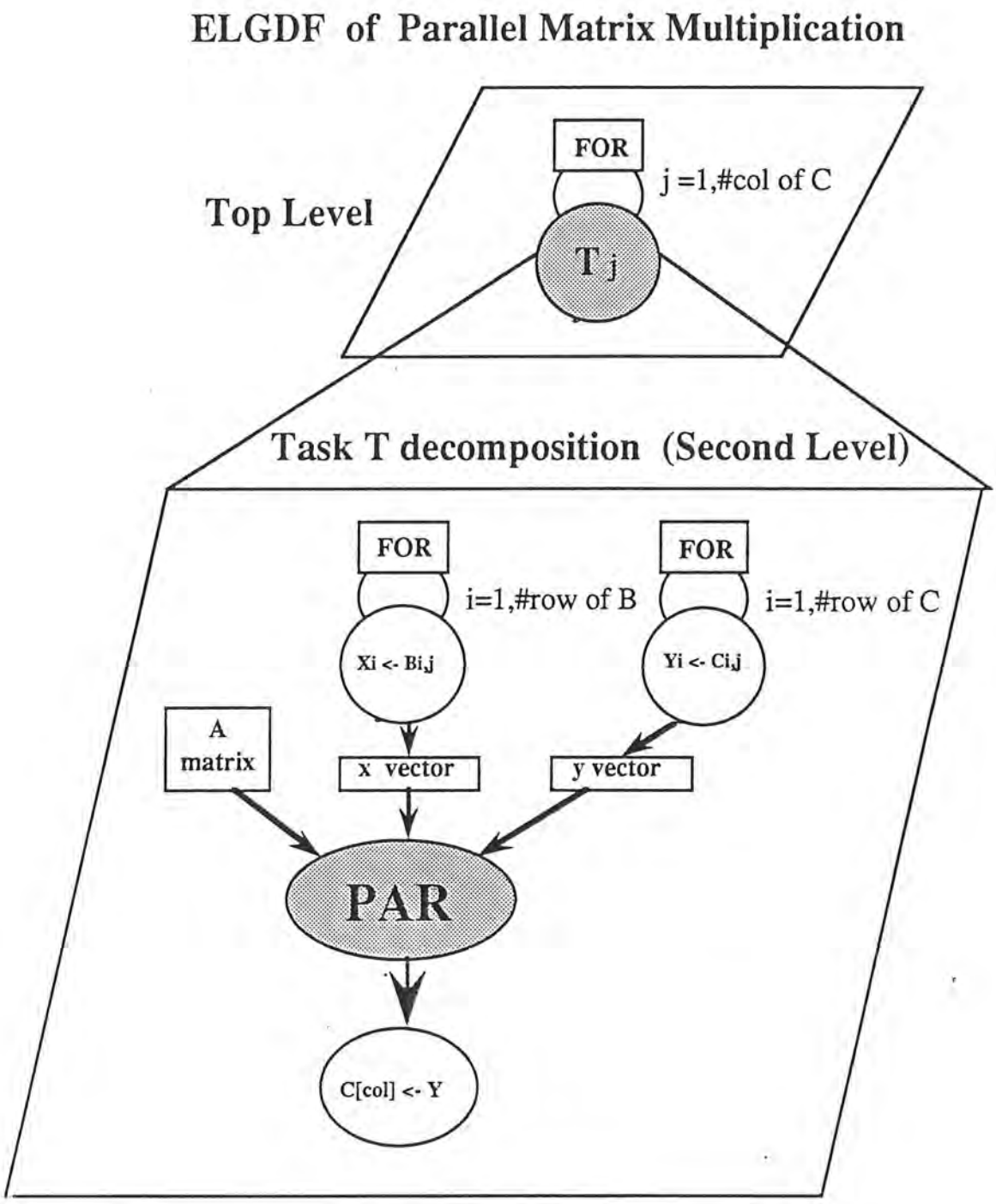
1. Sequent Balance 21000 shared memory system:

The matrix multiplication benchmark program takes matrix A times matrix B and stores the result in matrix C. The three matrices are declared as shared structures. The parallelism comes when multiple processes execute loop iterations simultaneously. Dynamic process scheduling is applied, every process consumes subtasks (part of the loop iterations) from a task queue (all the loop iterations to be executed). Dynamic scheduling keeps all the processes busy until they finish all the tasks in the task queue.

2. Cogent XTM workstation:

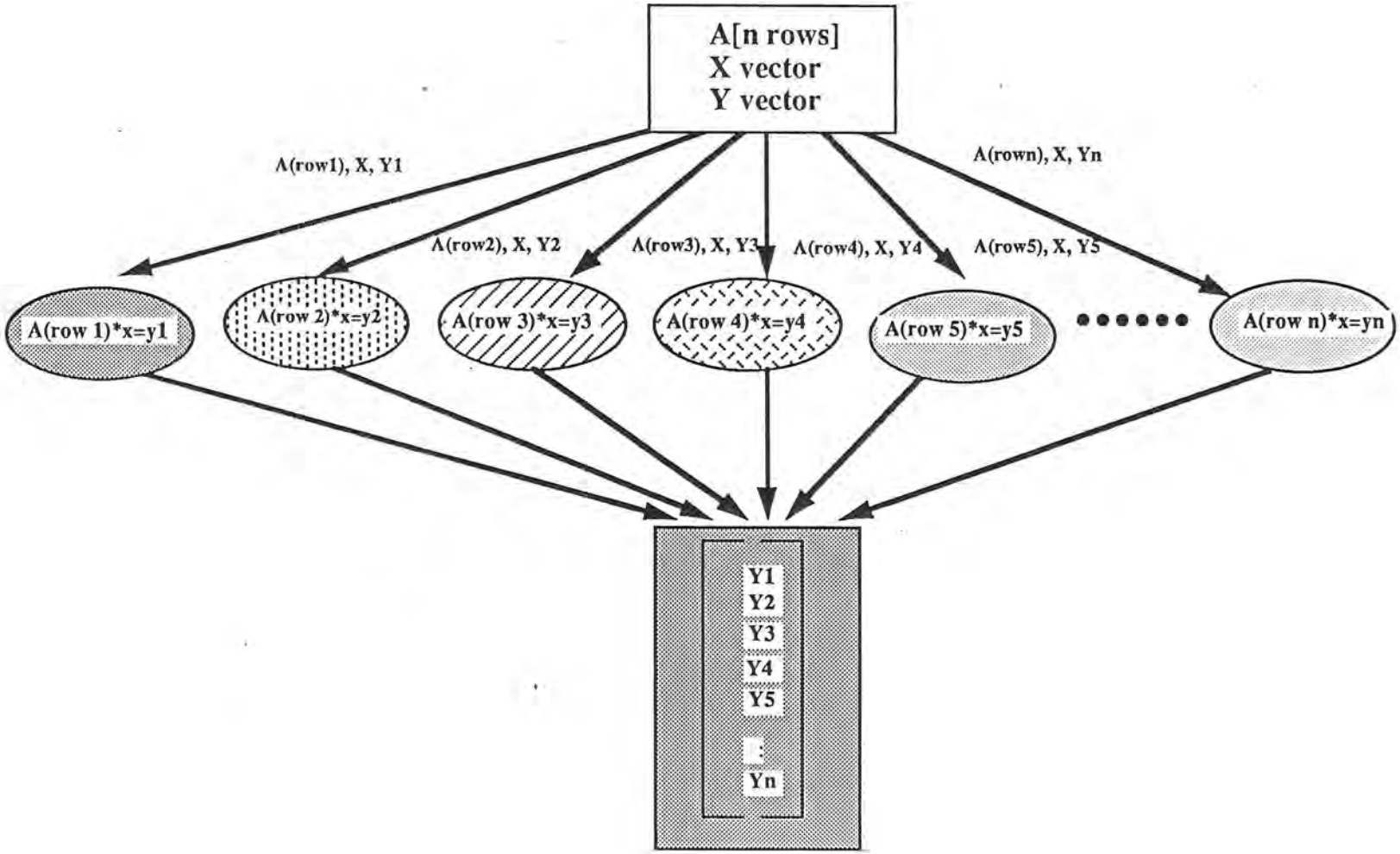
With Linda primitives, we can apply the same algorithm we used for Sequent Balance shared memory system. First we create a global tuple space. The parent process then *OUT* s all the rows of matrix A and a column of matrix B and C to this tuple space. We can certainly treat this tuple space as a task queue and have worker processes to consume the tasks simultaneously. Each worker process *IN* s a row of matrix A to do vector multiplication with a column of matrix B and get a column of target matrix C. Since Linda does not support primitives to implement dynamic process scheduling and we know that the computing time is approximately the same for each iteration of the loop, we apply static scheduling technique by simply divide the loop iterations evenly among processes.

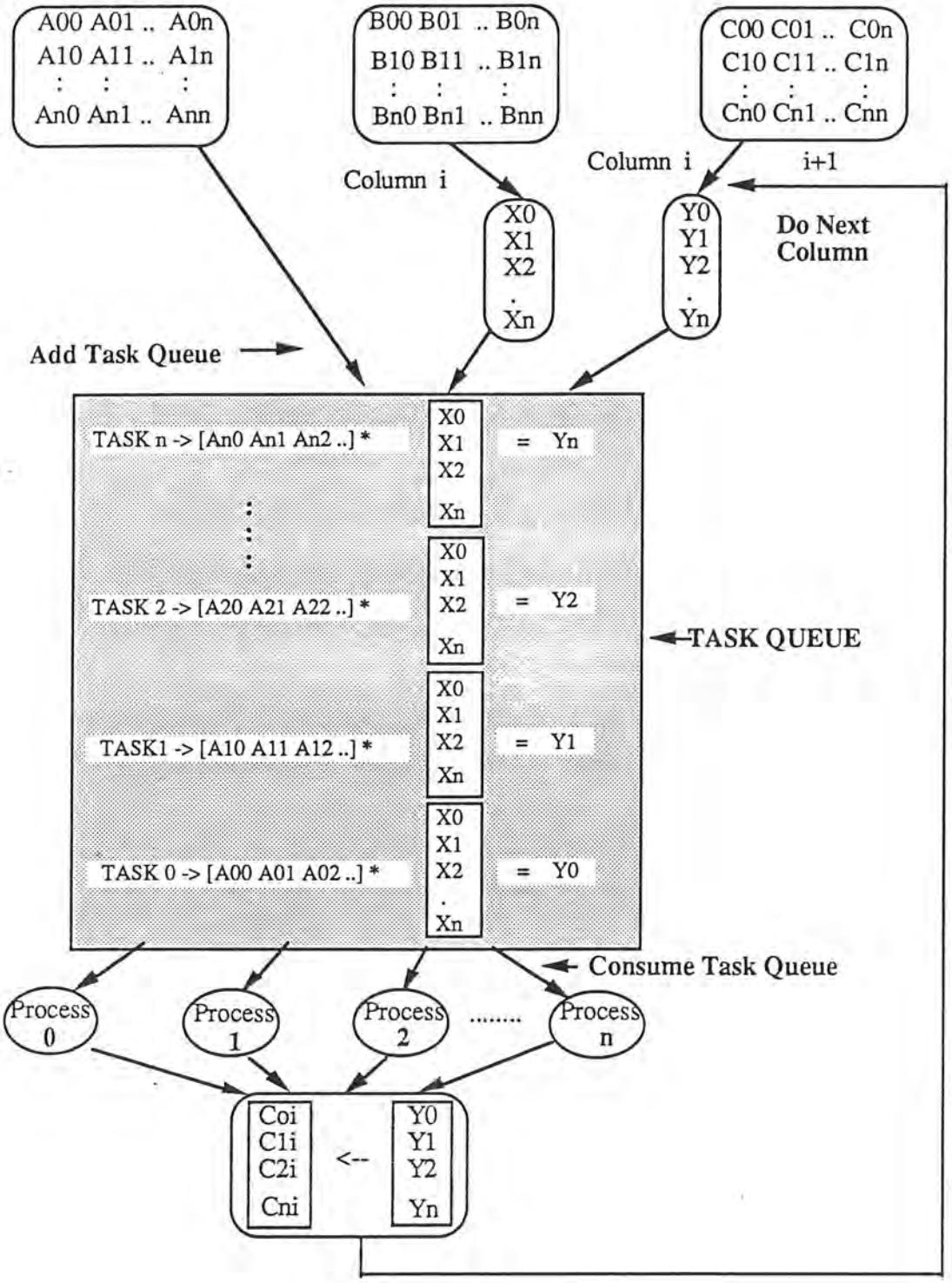
See ELGDF 1 and Task graph 1.



ELGDF 1 Parallel Matrix Multiplication

PAR task decomposition





Task Graph Parallel Matrix Multiplication

Algorithm [Hwang and Briggs, 1984]

```
for j = 1 to number of columns of target matrix
    for i = 1 to number of rows of target matrix
        C_ij = 0      /* initialize a column of target matrix and a  */
        Y_ij = 0      /* temp vector which represents a column       */
    end i loop        /* of target matrix                            */

    for k = 1 to number of rows of B
        X_k = B_kj    /* initialize a temp vector which will be      */
    end k loop        /* used to perform vector multiplication       */


    PARBEGIN
        while (task queue is not empty)

        /* X which is a column of matrix B times a row of A  */
        /* and store result in vector Y which will be a      */
        /* column of target matrix C                         */

            begin
                for l = 1 to number of rows of B
                    /* pid is the process id for each  */
                    /* child process                   */
                    Y_pid = Y_pid + A_pid ,l * X_l
                end l loop
                /* every child process sends result to  */
                /* target matrix                        */
                C_pid,j = Y_pid
            end
    PAREND
end j loop
```

Result

1. Sequent Balance 21000 system :

Sequent processes access different matrix index of the shared matrix in the shared memory simultaneously, there is no memory fetch conflict. In this case, it is not required to apply mutual exclusion. The result includes only the computation time without the process communication overhead. The speedup then can be obtained very close to linear model when the matrix size is big enough (96 X 96 or bigger). See Tables 3.1.1 - 3.1.11, Figures 3.1.1 - 3.1.2 and Gantt chart 1.

#pe / child	1	2	3	4	5	6	7	8	9
1	0.73	0.38	0.25	0.20	0.18	0.15	0.16	0.08	0.10
2		0.36	0.25	0.19	0.16	0.15	0.12	0.09	0.16
3			0.24	0.20	0.15	0.14	0.14	0.06	0.21
4				0.18	0.15	0.13	0.12	0.06	0.23
5					0.18	0.13	0.12	0.07	0.23
6						0.14	0.11	0.08	0.19
7							0.14	0.06	0.16
8								0.09	0.22
9									0.16
Total	0.76	0.41	0.28	0.23	0.22	0.18	0.19	0.20	0.22

Table 3.1.1 Matrix Multiplication single precision 24x24

#pe / child	1	2	3	4	5	6	7	8	9
1	5.44	2.78	1.87	1.42	1.17	0.95	0.82	0.75	0.66
2		2.77	1.85	1.40	1.18	0.96	0.75	0.73	0.68
3			1.86	1.41	1.21	0.96	0.92	0.75	0.66
4				1.42	1.17	0.96	0.90	0.74	0.69
5					1.17	0.97	0.84	0.74	0.67
6						0.98	0.90	0.72	0.69
7							0.94	0.71	0.70
8								0.75	0.68
9									0.70
Total	5.53	2.85	1.92	1.49	1.25	1.07	0.97	0.83	0.86

Table 3.1.2 Matrix Multiplication single precision 48x48

child \ #pe	1	2	3	4	5	6	7	8	9
1	42.31	21.58	14.31	10.88	8.59	7.27	6.28	5.49	4.87
2		21.57	14.39	10.87	8.59	7.23	6.30	5.47	4.88
3			14.33	10.85	8.63	7.29	6.28	5.48	4.97
4				10.85	8.60	7.27	6.27	5.51	4.96
5					8.61	7.24	6.29	5.51	5.03
6						7.29	6.35	5.49	5.01
7							6.30	5.48	4.95
8								5.48	4.98
9									5.00
Total	42.38	21.71	14.50	11.01	9.08	7.42	6.50	5.65	5.21

Table 3.1.3 Matrix Multiplication single precision 96x96

child \ #pe	1	2	3	4	5	6	7	8	9
1	381.67	193.96	128.09	96.58	76.75	64.28	54.79	48.37	42.90
2		193.80	127.92	96.45	76.85	64.47	55.33	48.33	43.09
3			128.03	96.46	76.87	64.07	54.80	48.31	42.85
4				96.48	76.89	64.17	55.33	48.38	43.16
5					76.86	64.32	54.99	48.32	42.92
6						64.28	54.99	48.33	43.02
7							55.05	48.31	43.22
8								48.36	42.89
9									43.06
Total	381.93	194.39	129.08	96.85	77.23	65.87	56.14	48.71	44.58

Table 3.1.4 Matrix Multiplication single precision 200x200

#PE	24 X 24		48 X 48		96 X 96		200 X 200	
	time	speedup	time	speedup	time	speedup	time	speedup
1	0.76	1.00	5.53	1.00	42.38	1.00	381.93	1.00
2	0.41	1.85	2.85	1.94	21.71	1.96	194.39	1.97
3	0.28	2.71	1.92	2.88	14.50	2.92	129.08	2.96
4	0.23	3.30	1.49	3.71	11.01	3.84	96.85	3.94
5	0.22	3.45	1.25	4.42	9.08	4.66	77.23	4.95
6	0.18	4.20	1.07	5.17	7.42	5.71	65.87	5.80
7	0.19	4.00	0.97	5.70	6.50	6.52	56.14	6.80
8	0.20	3.80	0.83	6.66	5.65	7.50	48.71	7.84
9	0.22	3.45	0.86	6.43	5.21	8.13	44.58	8.56

Table 3.1.5 Parallel Matrix Multiplication
(32-bit integer operation)

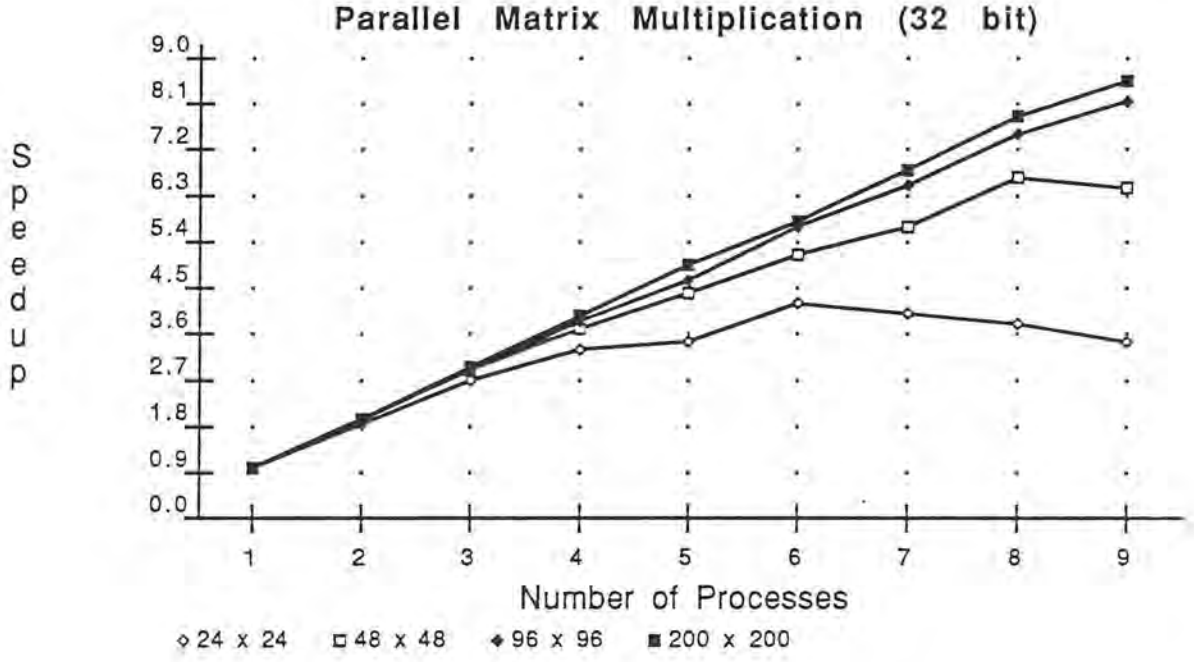
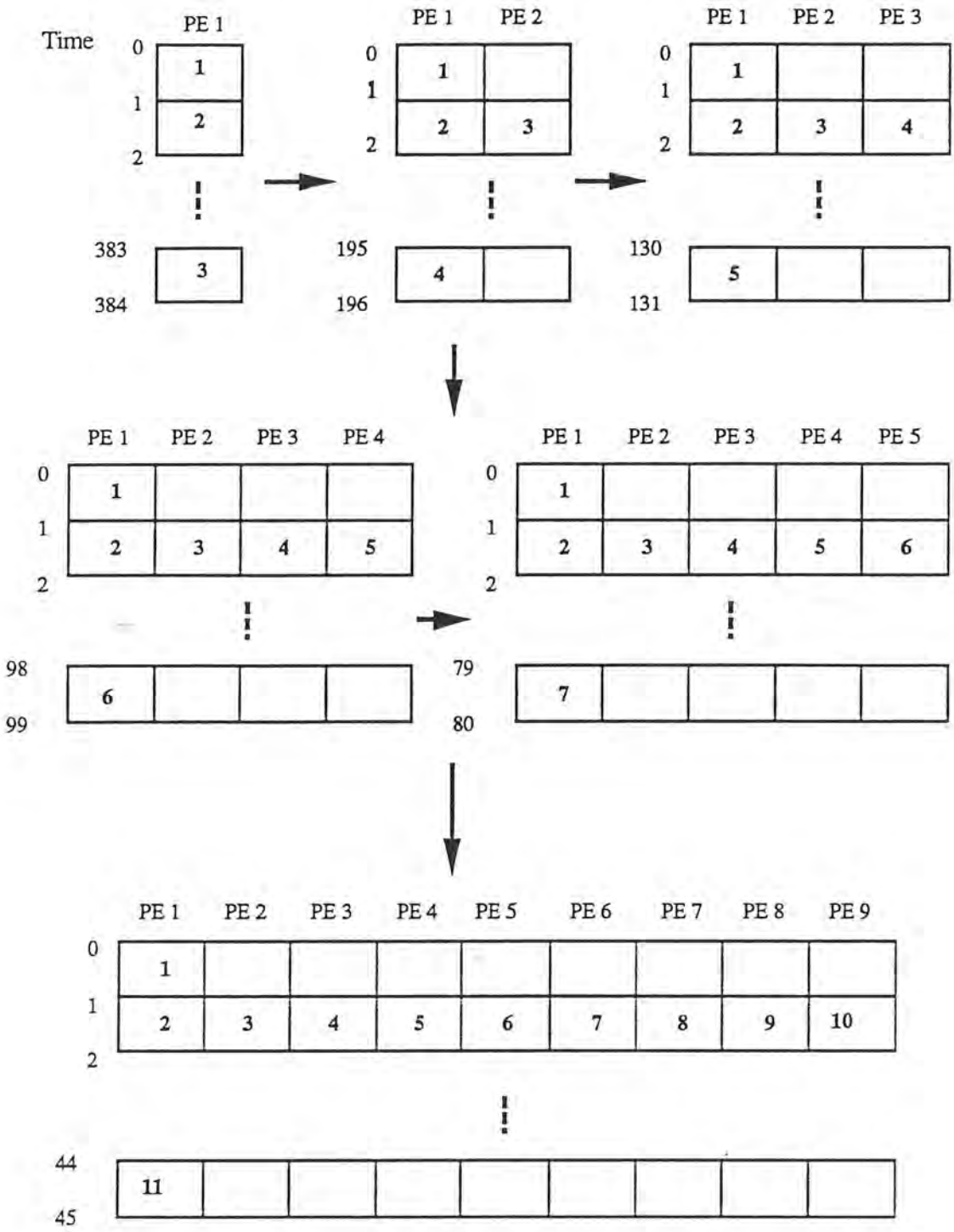


Figure 3.1.1 Speedup of Parallel Matrix Multiplication
(32 bit integer operation)



Gantt Chart 1 Process scheduling of Parallel Matrix Multiplication
Starting node is processed at time unit 1.
Parallel nodes are executed at time unit 2.
Ending node is done at time unit 43 (9 processes).

#PE	24 X 24		48 X 48		96 X 96		200 X 200	
	time	speedup	time	speedup	time	speedup	time	speedup
1	78	1.00	556	1.00	425	1.00	384	1.00
2	43	1.81	288	1.93	219	1.94	196	1.96
3	33	2.36	195	2.85	149	2.85	131	2.93
4	26	3.00	153	3.63	114	3.72	99	3.88
5	24	3.25	129	4.31	94	4.52	80	4.80
6	19	4.10	111	5.01	78	4.88	67	5.71
7	22	3.54	102	5.54	69	6.16	59	6.51
8	23	3.39	87	6.40	59	7.20	51	7.53
9	25	3.12	90	6.17	56	7.59	45	8.53

Table 3.1.6 **Gantt Chart time unit** for Parallel Matrix Multiplication (32-bit integer operation)

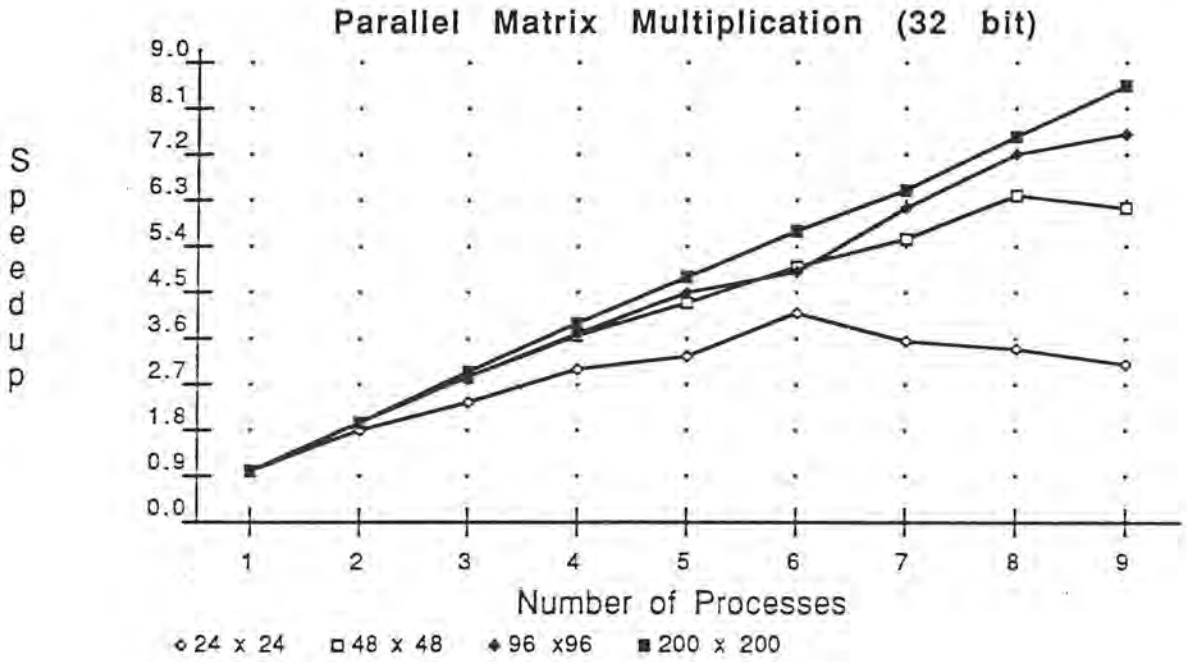


Figure 3.1.2 **Gantt Chart time unit** Speedup of Parallel Matrix Multiplication (32 bit integer operation)

#PE	24 X 24		48 X 48		96 X 96		200 X 200	
	time	speedup	time	speedup	time	speedup	time	speedup
1	0.97	1.00	7.22	1.00	57.30	1.00	515.06	1.00
2	0.52	1.87	3.75	1.93	28.75	1.99	258.93	1.99
3	0.48	2.02	2.58	2.80	19.26	2.98	171.27	2.98
4	0.30	3.23	1.95	3.70	14.56	3.94	128.70	3.99
5	0.25	3.88	1.63	4.43	12.03	4.76	102.98	4.98
6	0.23	4.22	1.36	5.31	9.86	5.81	86.55	5.95
7	0.22	4.41	1.21	5.97	8.58	6.68	74.85	6.88
8	0.19	5.11	1.20	6.01	7.58	7.56	65.72	7.83
9	0.21	4.62	1.37	5.27	7.07	8.10	59.72	8.62

Table 3.1.7 Parallel Matrix Multiplication
(64-bit double precision operation)

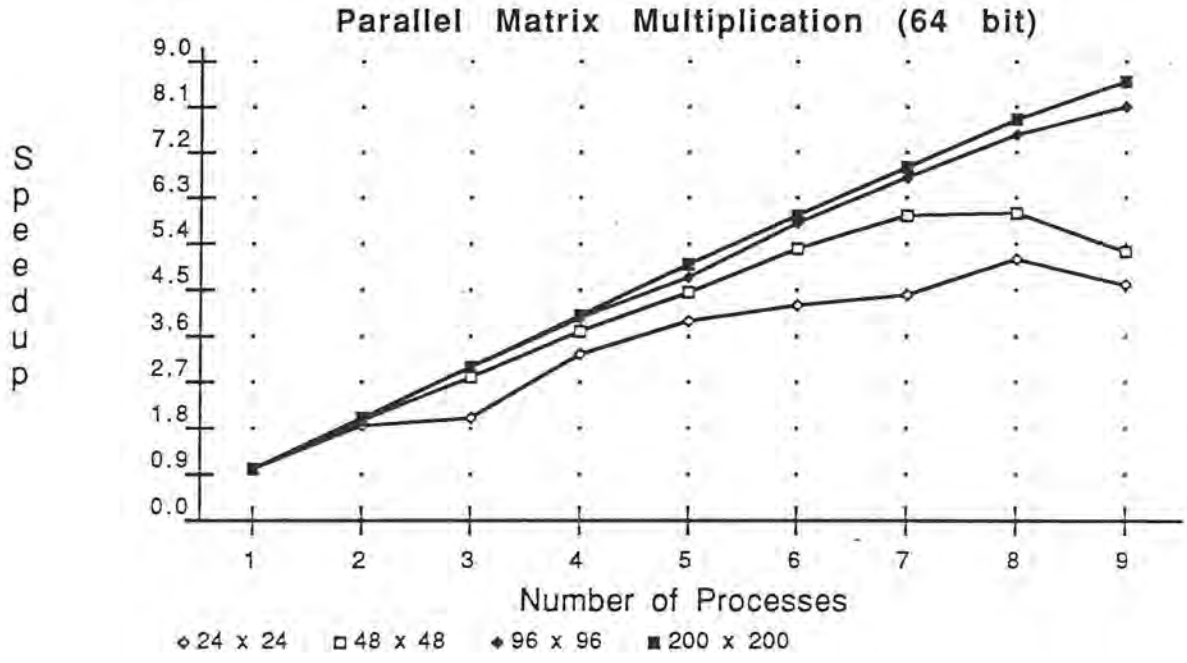


Figure 3.1.2 Speedup of Parallel Matrix Multiplication
(64 bit double precision operation)

2. Cogent XTM workstation :

Because of the overhead of Linda operation, the single processor XTM presents a slower execution speed than Sequent when the size of data is small. However, the XTM performs a 15 to 20 percent faster than Sequent by increasing the data size to 200 or more.

	24 X 24	48 X 48	96 X 96	200 X 200
	time	time	time	time
SP (32-bit)	2.43	11.01	55.01	354.56
DP (64-bit)	2.48	11.60	59.62	415.47

Table 3.1.8 XTM Matrix Multiplication
(Single Processor)

3.2 Disk File I/O

As I/O is responsible for more and more of the time we spend at the computer, it becomes more important for a user to know the I/O performance of his/her computers. An I/O benchmark program helps people obtain a simple and practical result.

The number of components involved with any storage access makes accurate performance testing very difficult. In general, the storage device itself, the device controller, and the computer's operating system are involved with any transaction.

I/O in parallel only comes when each storage device has its own control channel. In this case, many processes can issue I/O from/to different disks through different control channels. If I/O in parallel is working on a common disk file by more than one process, it competes with itself for both the control channel, disk arm and disk heads which can only be in one physical position at a time.

Instead of using sequential I/O testing, a parallel I/O benchmark program is introduced. This parallel I/O benchmark program is based on the idea of testing the I/O bottleneck and congestion (traffic problem occurs when multiple processes issue I/O to a common disk file through a single channel). The program forces many processes to perform I/O to a common disk file. Each process accesses different offsets of the common file at the same time. However, this is still logical parallelism.

Although all processes access different offsets of the common disk file simultaneously, the accesses are actually done sequentially, because there is only one disk controller, one disk arm, and a single disk head handling this I/O operation. This logical parallel I/O benchmark program thus gives us a combination performance of system bus, disk controller, I/O control channel, and storage device itself.

Implementation

This program use "C" low level I/O routines such as CREATE, OPEN, READ, WRITE, LSEEK and CLOSE to perform disk file read/write. A 2 MB disk file is created and read by all processes. Each process then writes what it has read to another 2 MB disk file.

Result

1. Sequent Balance 21000 system :

#PE	4 KB buffer	8 KB buffer	16 KB buffer	32 KB buffer
1	15.42	12.34	11.47	10.40
2	14.45	11.65	11.22	10.92
3	14.98	12.12	11.94	10.36
4	14.36	12.64	11.95	11.22
5	15.04	12.51	11.32	11.25
6	15.33	13.10	11.87	10.80
7	16.31	13.23	12.33	10.51
8	17.40	14.08	12.11	11.34
9	17.31	13.71	12.22	11.46

Table 3.2.1 Run time of Disk File I/O

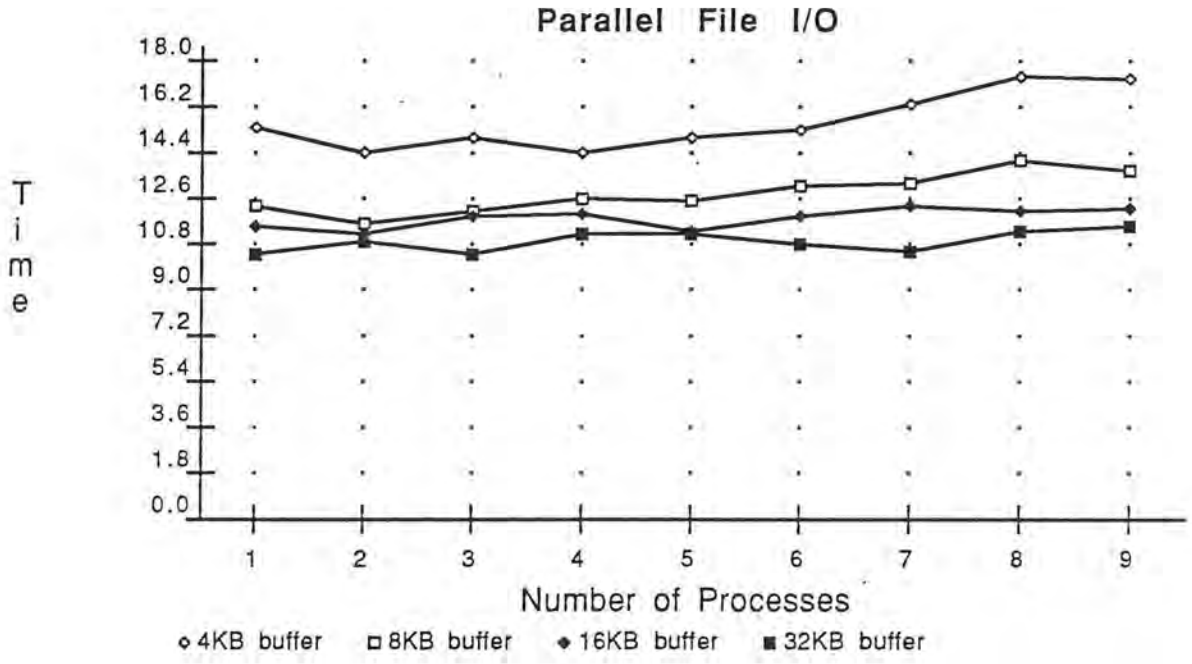


Figure 3.2.1 Result of Parallel Disk File I/O
(Sequent Balance 21000 system)

From the graph shown above, I/O in parallel without hardware support does not show any speedup. This program should be executed under sigle user mode to avoid other users share system resource. The run time shown above is not collected in the single user mode. Instead, we use UNIX "team" command to gain the highest priority to the command. The run time of single processor on XTM with 32KB buffer size is 11.30 seconds.

3.3 Many to many process message passing - Saturating

Saturating is the task of each processor sending a message to every other processor [Levitan, 1987]. The message passing among multiple processors is an important factor to affect the performance of multiprocessor machine. Choosing this task as a benchmark program is to measure the interprocess communication overhead.

On the Sequent system, interprocess communication is through shared memory. XTM system is a distribute system but its process communication takes advantage of using LINDA tuple space. It is a simulation of shared memory. Like broadcasting and reporting, saturating is a basic and essential task which is widely used in parallel algorithms.

Implementation

1. Sequent Balance 21000 shared memory system:

A two dimensional array is created in shared memory to facilitate "send" and "receive" operations. The "send" operation is each process sends a message which is a structure containing process ID and an array to an entire column. Since a row is used for one process to receive message from all other processes, the "receive" operation then will be done by each process retrieving data from an entire row and storing contents to process's local space.

The parallelism comes when multiple processes perform "send" and "receive" operations simultaneously. A synchronization point (barrier) is set to guarantee that processes complete "send" operations before they can start "receive" operations.

2. Cogent XTM workstation:

On the XTM system, processes never exchange message directly. Instead, a process with data to communicate adds it to the shared tuple space, and a process expecting data seeks it in that tuple space. We create equal number of send workers and receive workers. All send workers send multiple copies of its data along with their ID to the tuple space. Linda *IN* operation provides a good syncronization feature which allows receive workers wait until a desired tuple is *OUT*ed by any one of the send workers. Therefore, we do not need to set a syncronization point (barrier) to have all receive works start after all send process are terminated.

Result

1. Sequent Balance 21000 system :

#PE	1000 times array[100]		1000 times array[50]		500 times array[200]		500 times array[100]	
	time	speedup	time	speedup	time	speedup	time	speedup
1	18.88	1.00	9.39	1.00	17.04	1.00	9.23	1.00
2	29.04	0.65	15.54	0.61	29.25	0.59	14.45	0.64
3	40.70	0.46	21.32	0.44	40.19	0.42	20.46	0.45
4	51.88	0.36	27.66	0.34	51.70	0.33	25.95	0.35
5	62.16	0.30	34.16	0.27	63.40	0.27	31.57	0.29
6	74.30	0.25	38.60	0.24	74.39	0.23	37.14	0.25
7	85.33	0.22	44.91	0.20	86.40	0.19	43.10	0.21
8	97.95	0.19	50.41	0.18	97.78	0.17	49.13	0.19
9	110.15	0.17	56.77	0.16	111.40	0.15	54.41	0.17

Table 3.3.1 Many to many process message passing

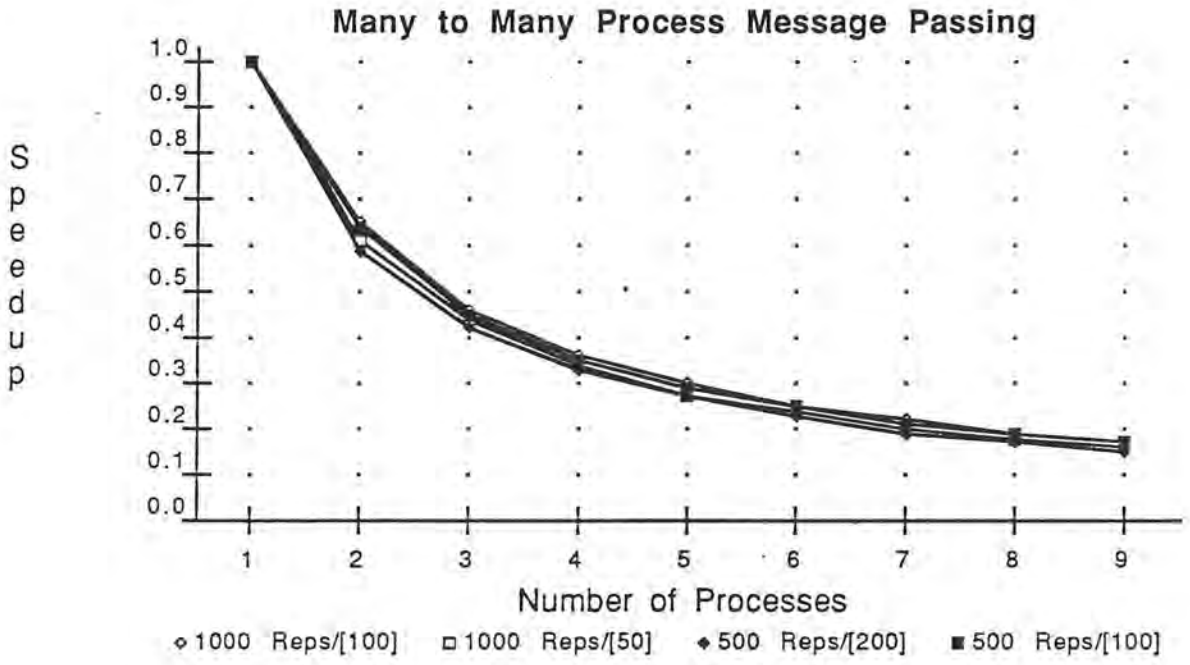


Figure 3.3.1 Speedup of Many to Many Process Message Passing (Sequent balance 21000)

2. Cogent XTM workstation:

#PE	1000 times array[100]	1000 times array[50]	500 times array[200]	500 times array[100]
1	8.30	7.13	5.17	4.15

Table 3.3.2 XTM process message passing

Unlike other parallel programs, this message passing benchmark program will not show us any speedup. Instead, adding more processes will saturate the system bus and then increase the program execution time. Since there is only a single processor available on XTM, the result of many to many message passing is meaningless at this moment. See Tables 3.3.1 - 3.3.2 and Figures 3.3.1- 3.3.2.

3.4 Parallel Enumeration Sort

Sorting has been widely used in many applications and selected as a benchmark program for years. On a multiprocessor machine, sorting can be done in a much faster way while many processes sharing the subtasks. How fast is the program is not the first concern when we design a sorting benchmark program. Instead, the benchmark program should be able to reflect the speed of comparison operations, speed of executing loop iterations and the speed of accessing shared memory. The Parallel Enumeration Sort [Akl, 1985] program uses the most straight forward technique to sort a sequence of elements. Although the algorithm is not efficient at all, for the purpose of the performance testing, enumeration sort does much more comparisons, loop iterations and shared memory access than other faster sorting algorithms.

Implementation

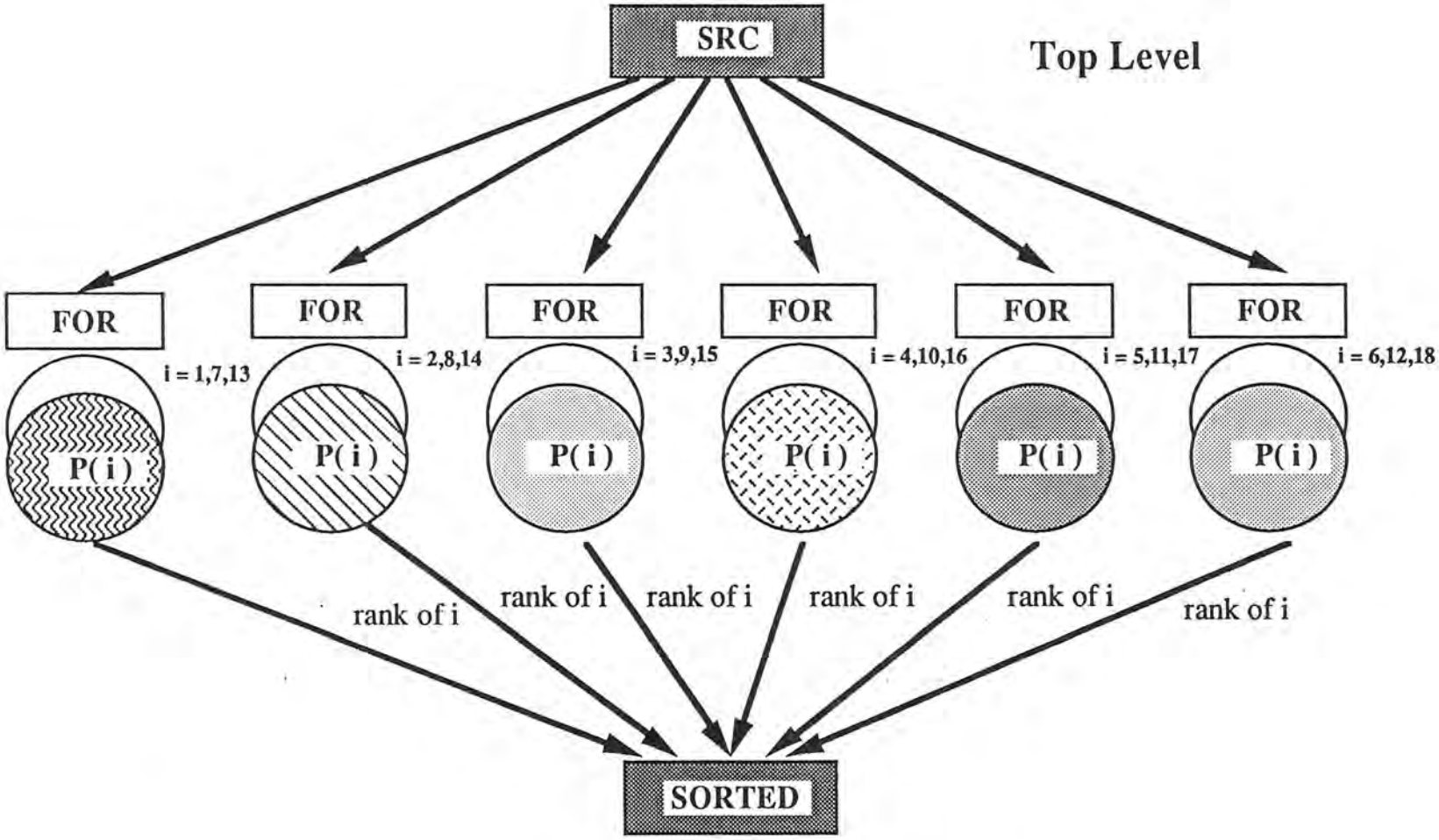
Parallel enumeration sort uses a simple algorithm which compares an element with all other elements and gets a rank for this element. Whenever the rank of a element is decided, this element is sent to its position in the sorted target array according to its rank. Both original array and sorted array are declared as shared structure in the shared memory for Sequent system. Linda implementation of the enumeration sort has a global tuple space where the whole unsorted array and each individual elements reside. Worker process *IN*s an individual element and *READ*s a copy of the whole unsorted array. After comparing each element with all the elements in the array, a rank of an individual element is decided and sent to the target array in the global tuple space.

See ELGDF 2 and Task graph 2.

Algorithm

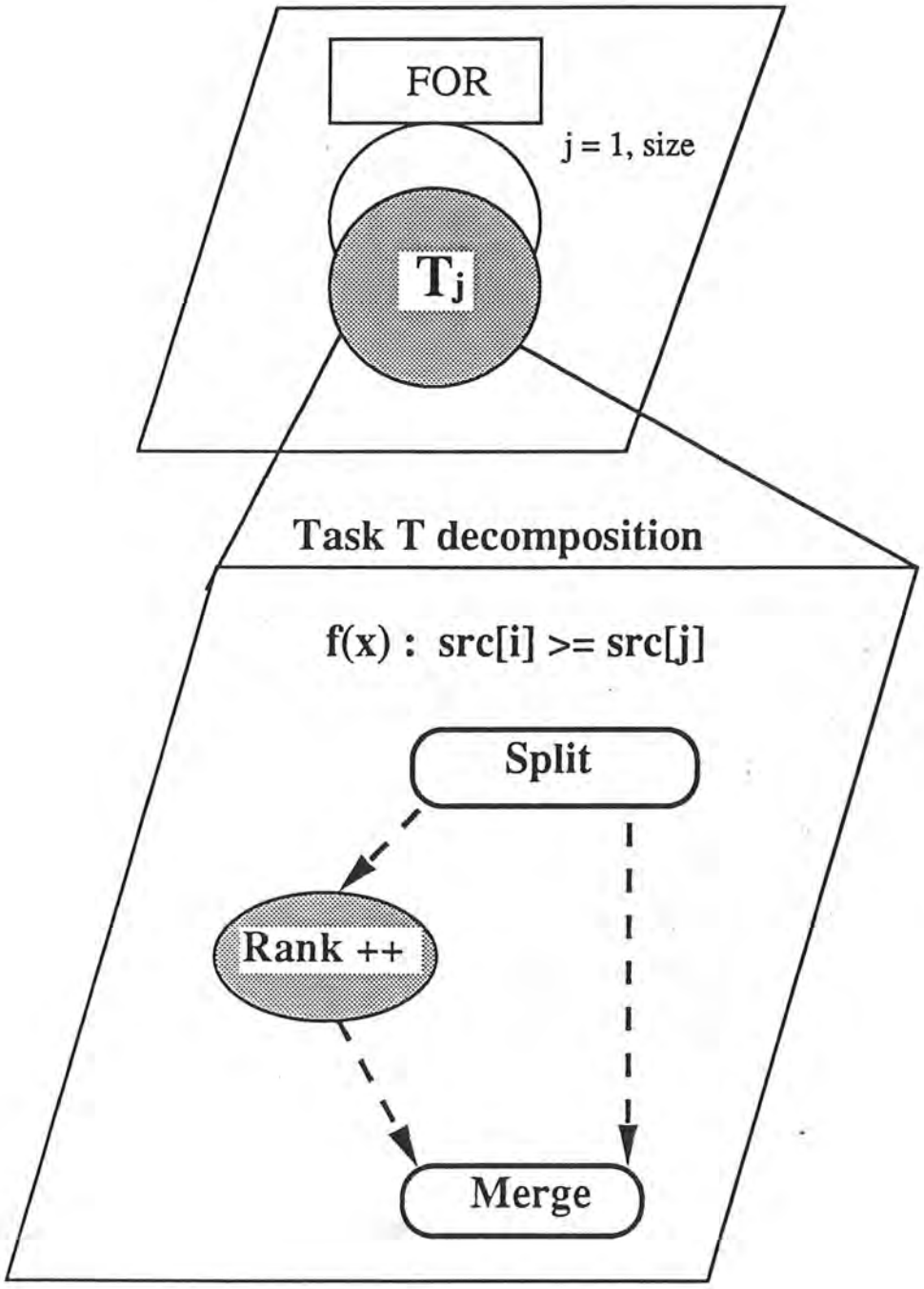
```
(1) for  i = 1 to number of processes
      create process i
(2) process i
          rank <- -1
          for j = 1 to number of elements do
                if (source[i] > source[j])
                      rank <- rank + 1
                else if (source[i] = source[j]) and (i>j)
                      rank <- rank + 1
          end j loop
          /* now send an element to its position in the sorted array */
          sorted[rank+1] <- source[i]
```

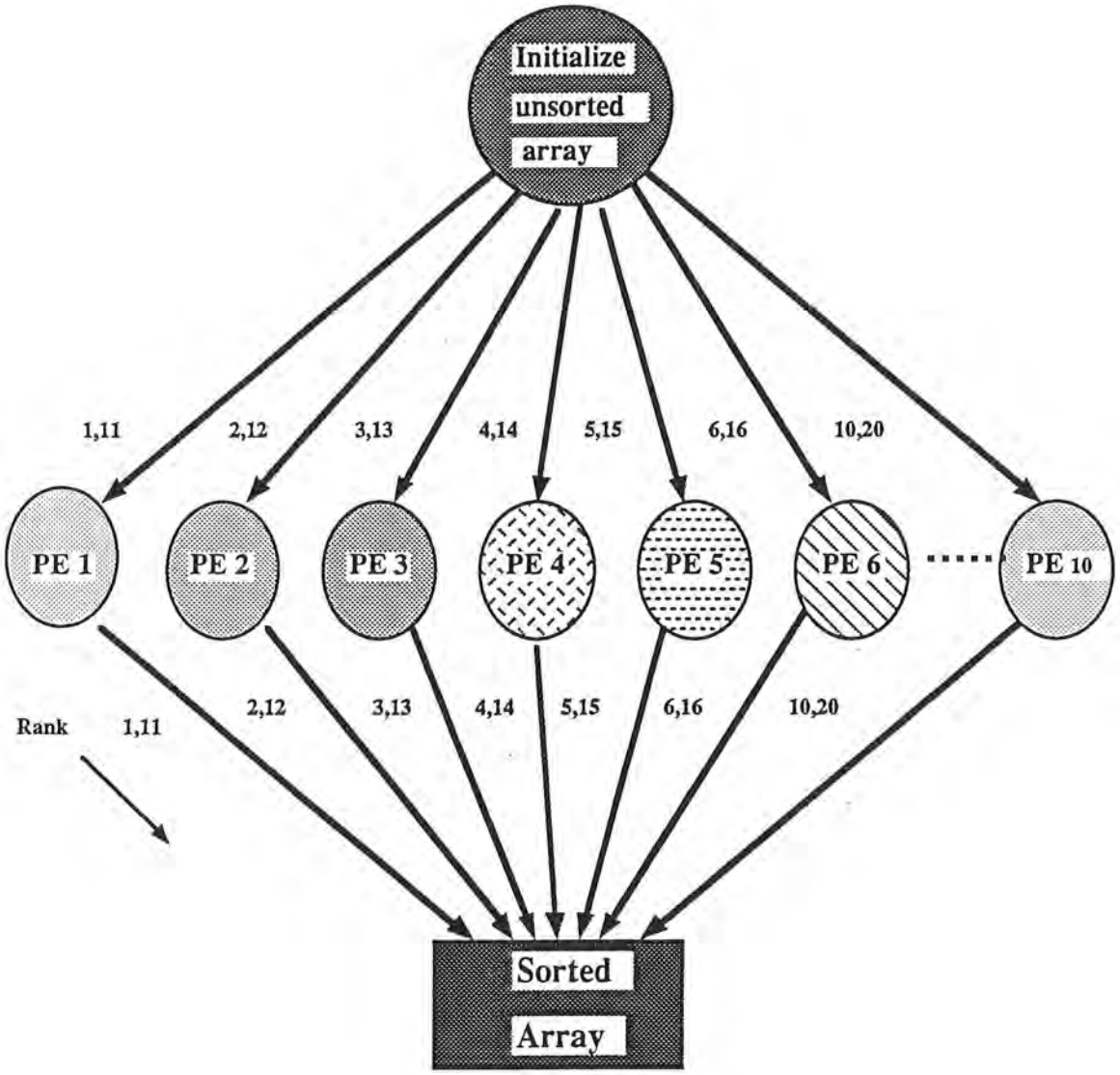
ELGDF of Parallel Enumeration Sort



ELGDF 2 Parallel Enumeration Sort

Child process decomposition





Task Graph 2 **Parallel Enumeration Sort**
(sort 20 elements by 10 processes)

Result

1. Sequent Balance 21000 system:

child \ #pe	1	2	3	4	5	6	7	8	9
1	0.41	0.21	0.15	0.10	0.08	0.08	0.06	0.05	0.07
2		0.21	0.14	0.10	0.08	0.07	0.07	0.05	0.08
3			0.13	0.11	0.08	0.07	0.06	0.06	0.07
4				0.11	0.09	0.07	0.06	0.05	0.07
5					0.08	0.07	0.06	0.05	0.08
6						0.07	0.06	0.05	0.08
7							0.06	0.05	0.09
8								0.06	0.08
9									0.08
Total	0.43	0.23	0.17	0.12	0.11	0.10	0.09	0.08	0.11

Table 3.4.1 Parallel Enumeration Sort (128 elements)

child \ #pe	1	2	3	4	5	6	7	8	9
1	1.64	0.82	0.55	0.41	0.34	0.28	0.24	0.21	0.19
2		0.82	0.54	0.40	0.33	0.27	0.23	0.21	0.18
3			0.55	0.41	0.32	0.27	0.24	0.21	0.18
4				0.41	0.33	0.28	0.24	0.20	0.19
5					0.32	0.27	0.23	0.21	0.18
6						0.27	0.23	0.20	0.19
7							0.24	0.20	0.18
8								0.20	0.18
9									0.18
Total	1.65	0.84	0.57	0.45	0.35	0.30	0.26	0.23	0.21

Table 3.4.2 Parallel Enumeration Sort (256 elements)

child \ #pe	1	2	3	4	5	6	7	8	9
1	26.01	13.04	8.80	6.51	5.19	4.40	3.75	3.25	2.90
2		13.08	8.65	6.55	5.29	4.34	3.77	3.28	2.87
3			8.67	6.48	5.22	4.34	3.74	3.22	2.88
4				6.49	5.25	4.42	3.72	3.28	2.94
5					5.19	4.32	3.74	3.27	2.92
6						4.34	3.70	3.27	2.91
7							3.71	3.26	2.97
8								3.22	2.85
9									2.88
Total	26.30	13.19	8.83	6.67	5.31	4.45	3.80	3.32	2.99

Table 3.4.3 Parallel Enumeration Sort (1024 elements)

#PE	SIZE=128		SIZE=256		SIZE=1024	
	time	speedup	time	speedup	time	speedup
1	0.43	1.00	1.65	1.00	26.30	1.00
2	0.23	1.87	0.84	1.96	13.19	1.99
3	0.17	2.53	0.57	2.89	8.83	2.98
4	0.12	3.58	0.45	3.67	6.67	3.94
5	0.11	3.90	0.35	4.71	5.31	4.95
6	0.10	4.30	0.30	5.50	4.45	5.91
7	0.09	4.77	0.26	6.30	3.80	6.92
8	0.08	5.38	0.23	7.17	3.32	7.92
9	0.11	3.91	0.21	7.85	2.99	8.79

Table 3.4.4 execution time and speedup of Parallel Enumeration Sort

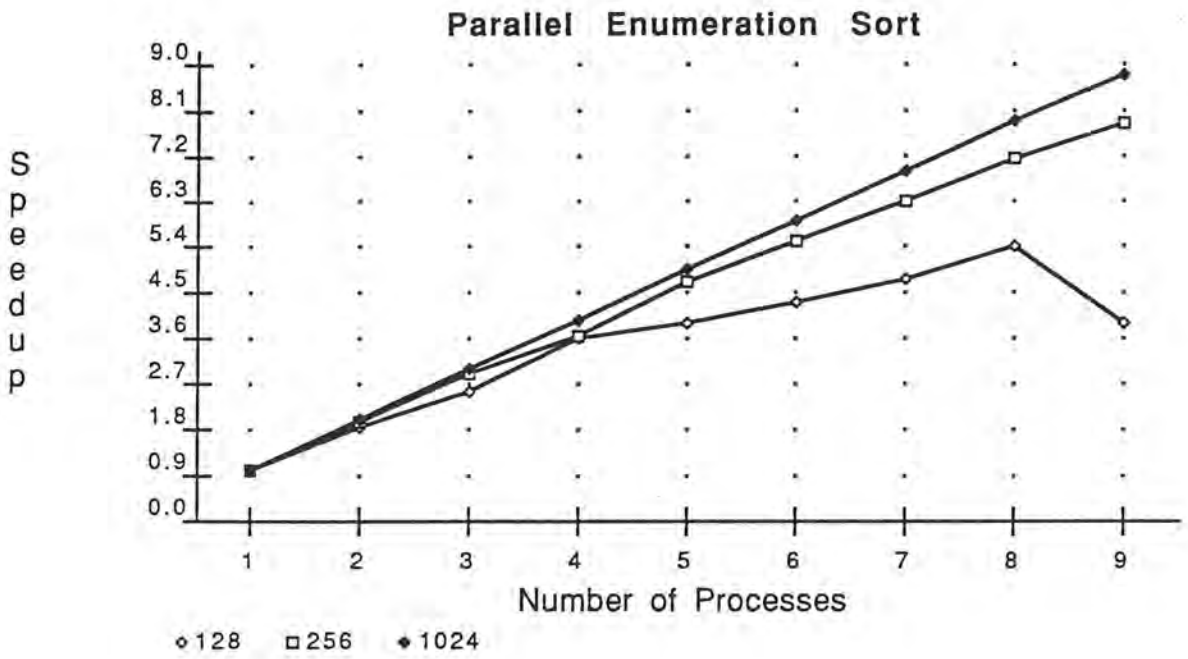
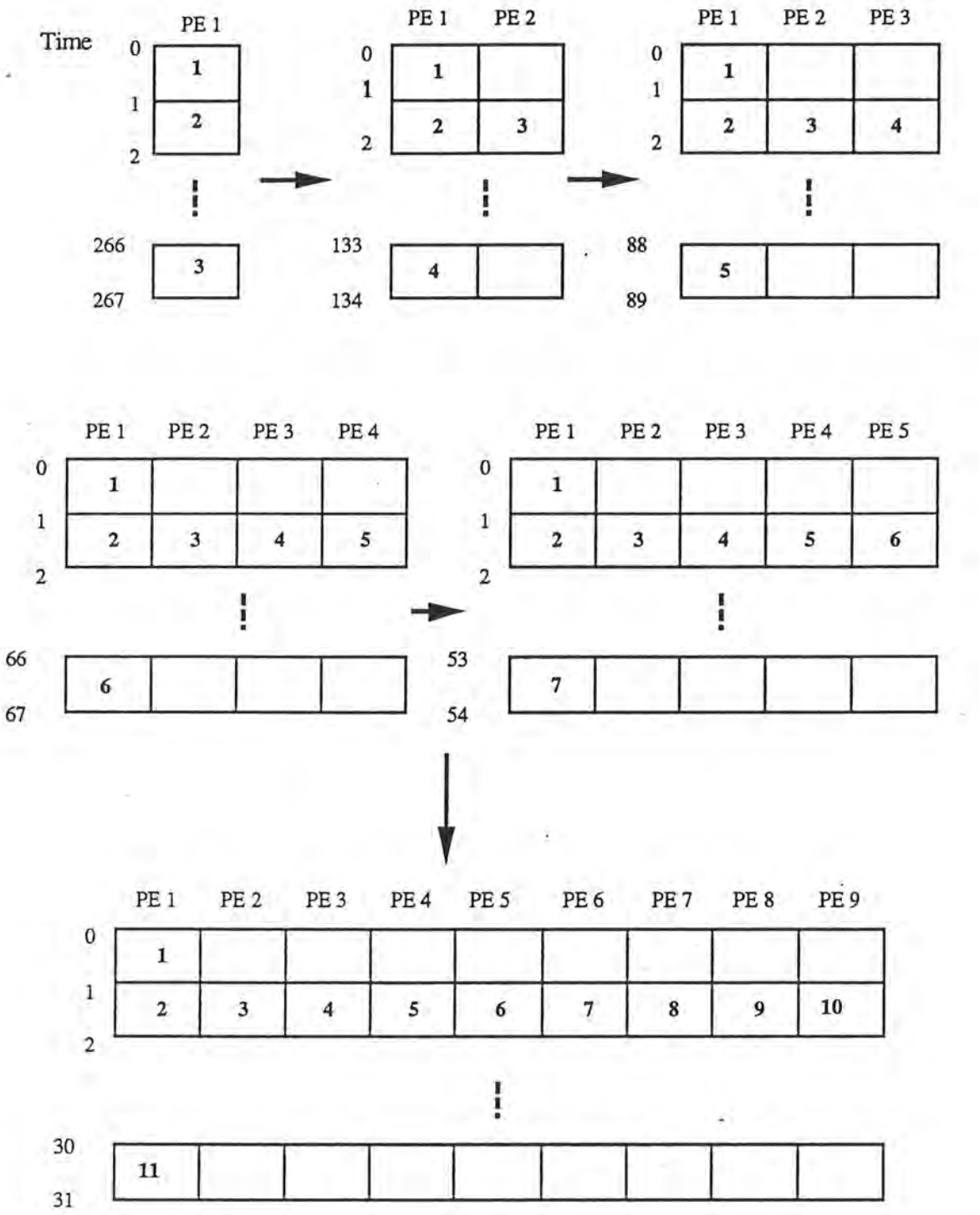


Figure 3.4.1 Speedup of Parallel Enumeration Sort (Sequent Balance 21000)

#PE	SIZE=128		SIZE=256		SIZE=1024	
	time	speedup	time	speedup	time	speedup
1	46	1.00	168	1.00	266	1.00
2	26	1.77	87	1.93	134	1.98
3	19	2.42	59	2.84	89	2.98
4	14	3.28	48	3.50	67	3.97
5	13	3.54	38	4.42	54	4.92
6	12	3.83	33	5.09	48	5.54
7	11	4.77	27	6.20	38	7.00
8	10	4.60	25	6.72	34	7.82
9	12	3.83	24	7.00	31	8.58

Table 3.4.5 **Gantt chart** time unit and speedup of Parallel Enumeration Sort



Gantt Chart 2 Process scheduling of Parallel Enumeration Sort
Starting node is processed at time unit 1.
Parallel nodes are executed at time unit 2.
Ending node is done at time unit 31 (9 processes).

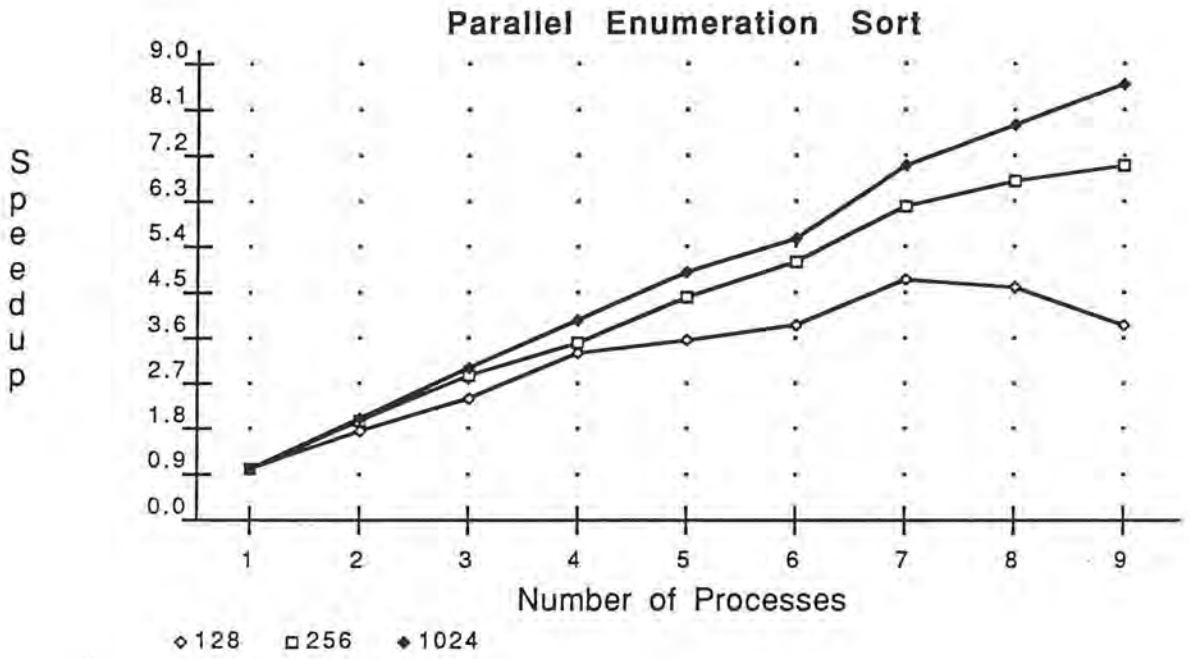


Figure 3.4.2 **Gantt Chart time unit** Speedup of Parallel Enumeration Sort

2. Cogent XTM workstation:

#PE	SIZE=128	SIZE=256	SIZE=1024
	time	time	time
1	0.32	1.02	13.76

Table 3.4.5 XTM Parallel Enumeration Sort

From the run time and speedup of Sequent shown above, we can see the speedup increases almost linearly with the increase in the number of processes used. The Gantt Chart result also shows the same speedup curve because there is no communication delay between processes on the shared memory system. Since the Linda operations are not heavily applied in this program, it thus results a 40 percent speedup over Sequent when the data size is increased to 1024. See Tables 3.4.1 - 3.4.5 and Figures 3.4.1- 3.4.2.

3.5 Memory data transfer

Recently, many multiprocessor machines have shared memory. The speed of moving data from one shared memory space to another shared memory space is concerned as an important factor which will affect the machine performance. For the distributed system such as XTM workstation, data transfer among processors' local memory is also a critical performance factor. This benchmark program is focused on testing the speed of data transfer among processors' local memory and data movement in the shared memory.

Implementation

1. Sequent Balance 21000 system:

The benchmark program for Sequent shared memory machine tests the speed of moving one byte, 2 bytes, 4 bytes and up to 4096 bytes from a shared memory location to another location. A source array and a destination array are declared to store the pointers to shared memory. Each process allocates shared memory space for source and destination locations and stores the pointers into source and destination arrays. Data transfer is the next task that moves the content of source location to the destination location. Program then releases the shared memory space after data have been transferred.

2. Cogent XTM workstation:

For the XTM workstation, each T800 transputer has its 4 MB local memory. Kernal Linda allows users to allocate transputer memory space and release it simply by calling system calls **k_allocate** and **k_free**. When program starts running, each process allocates a local memory space on a T800 transputer and *OUT*s the memory address to a global tuple space. Each process then *IN*s a memory address from tuple space. At this moment, each process then transfers the content of its local memory space to the address it just read in. Since we can not assign a process to a specific processor on the XTM system, the destination memory location is selected by withdrawing a memory address from tuple space randomly.

Result

1. Sequent Balance 21000 system:

#PE	1000 repetitions		10000 repetitions	
	time (sec.)	speedup	time (sec.)	speedup
1	9.80	1.00	91.34	1.00
2	10.77	0.90	99.38	0.91
3	11.61	0.84	107.10	0.85
4	12.94	0.75	116.12	0.78
5	14.13	0.69	123.57	0.73
6	15.20	0.64	134.83	0.67
7	16.32	0.60	146.41	0.62
8	17.71	0.55	156.33	0.58
9	19.27	0.50	174.57	0.52

Table 3.5.1 Memory transfer of Sequent Shared Memory System

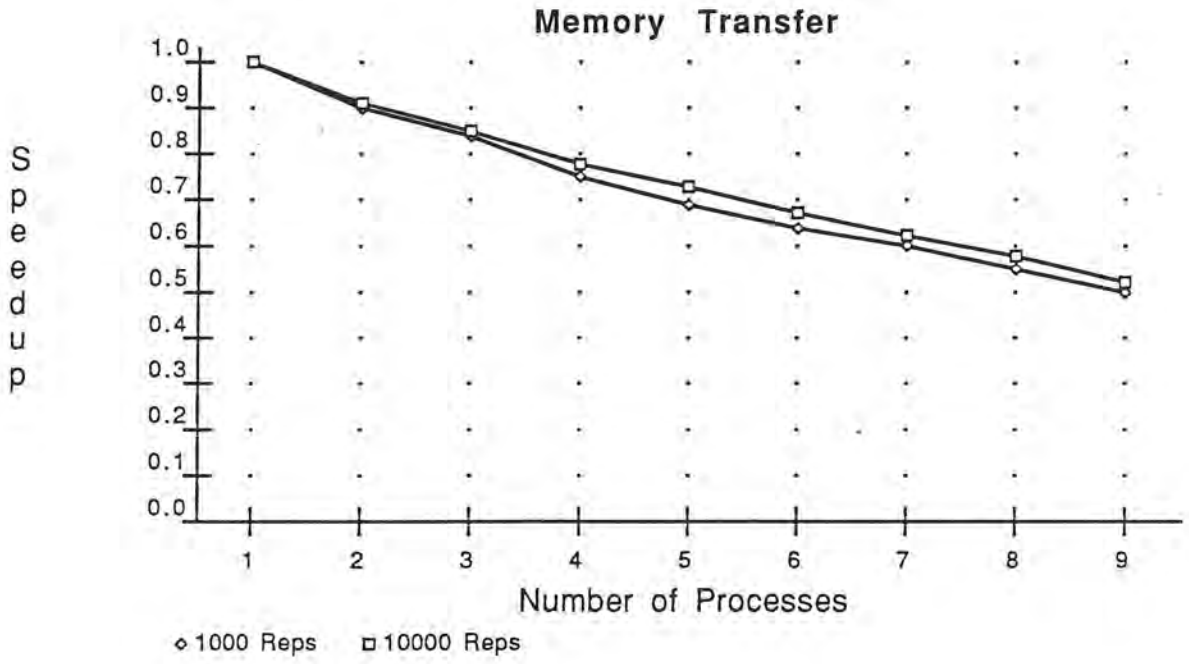


Figure 3.5.1 Memory transfer of Sequent Shared Memory System

2. Cogent XTM workstation:

#PE	1000 repetitions	10000 repetitions
	time (sec.)	time (sec.)
1	13	141

Table 3.5.2 Memory transfer of XTM distributed system

On the Sequent Balance System, programmers can not allocate a processor's local memory. So the test of memory transfer is actually moving data around the shared memory, instead of transferring data from one processor's memory space to another processor's memory space. Theoretically, the time used for each process to transfer data from one memory location to another memory location should be equal. Because of multiple processes access shared memory through system common bus at the same time, the overhead causes a negative speedup. See Table 3.5.1 and Figure 3.5.1.

3.6 System built-in Mathematical Functions

In "C" there is no built-in mathematical functions. Functions such as *SQRT(), EXP(), LOG(), SIN(), COS(),* and etc. usually occur in a special library. For example, our Sequent DYNIX system will make the mathematical library accessible if we give the ***-lm*** option in the command line when we compile the program.

Many scientific and engineering applications frequently use the system built-in mathematical functions. Therefore the purpose of implementing this benchmark program is to know how efficient the frequently used system built-in math functions are developed and how fast the processors can handle those math functions in your system.

Implementation

This program includes three categories of mathematical functions:

1. trigonometric functions:
   (1) *SIN(), COS(), TAN()*
   (2) *ASIN(), ACOS(), ATAN()*

2. logarithm and square root functions:
   (1) *EXP(), LOG(), LOG10(), POW()*
   (2) *SQRT()*

3. absolute value and approximate value functions:
   (1) *FABS()*
   (2) *FLOOR(), CEIL()*

All three sets of functions are executed 10000 iterations in sequential.

Algorithm

Trigonometric:

```
for i =1 to number of iterations do
        x = asin(sin(x) * cos(x));
        x = acos(sin(x) * cos(x));
        x = atan(tan(x));
        x /= const;
end i loop   /* each iteration almost transforms X back */
             /* to its initial value to prevent overflow   */
```

Logarithm and Square Root:

```
for i = 1 to number of iterations do
          x = sqrt(exp(log(log10(x))/const))
          x = pow(x,x)
          x *= const
        end i loop  /* each iteration transforms X back        */
                    /* to its initial value to prevent overflow */
                    /* or function return invalid value         */
```

Absolute and Approximate value:

```
for i = 1 to number of iterations do
          x = ceil(floor(x)/fabs(x))
          x *= fabs(const)
end i loop   /* each iteration transforms X back        */
             /* to its initial value to prevent overflow */
             /* or function return invalid value         */
```

Result

machine	OS	benchmark result
Sequent Balance 21000	DYNIX V3.0.14	42.53 sec.
Cogent XTM workstation	XTMOS 0.8x5	23.19 sec.

Table 3.6.1 Run time of Mathematic Functions

3.7 Linpack Routines [Linpack, 1986]

Linpack is a collection of FORTRAN subroutines developed by Jack Dongarra, Jim Bunch, Cleve Moler and Pete Stewart in 1979. (The name refers to the linear equations in the package.) It is a set of routines that analyze and solve systems of linear equations and linear least squares problems (both single precision and double precision).

The benchmark has now been run on engineering workstations and even some personal computers. Results are given in terms of Linpack KFLOPS (thousands of floating point operations per second). In May 1988, Bonnie Toy translated original FORTRAN routines to C and it has been published in the NBS benchmark collection. We have his C Linpack program run on Sequent Balance system and Cogent XTM workstation. The results are reported below:

Result

	Sequent/DYNIX	XTM/XTMOS	Repetitions
SPUNROLL	37 KFLOPS	256 KFLOP	10
DPUNROLL	35 KFLOPS	241 KFLOPS	10

Table 3.7.1 Linpack KFLOPS of 10 repetitions

* SPUNROLL : single precision with unrolled BLAS
* DPUNROLL : double precision with unrolled BLAS

Both Sequent Balance 21000 and XTM are not vector processors, so we run Linpack routines with UNROLL BLAS.

Note : BLAS - Basic Linear Algebra Subroutines such as dot product.

ROLL - all loops have simple accumulative expressions in them. This is used when testing vector processors.

Ex.
```
for (i =0; i<N; i++)
    sum = sum + x[i] * y[i]
```

UNROLL - inner loop gets expanded to do the equivalent of several iterations per loop.

Ex. for (i=0;i<N;i+=2)
sum = sum + x[i] *y[i] + x[i+1] * y[i+1]

4. Summary

Since there is no a standard language for all parallel computer systems, it is hard to compare the performance of two types of parallel computers via different parallel programming environments.

XTM presents a much better performance on those non-Linda benchmark programs such as Linpack and Math functions. This concludes that the overhead of Linda operations is a major factor to affect the hardware performance of XTM.

The basis of Linda is a logically shared memory through which processes communicate. Although it is easy to implement and the high speed of transputers can also overcome some of the overheads of Linda operations , it is still not very good to apply the concept of Linda to a distributed parallel computer system.

From the study on two types of parallel computer systems, shared memory system where all processors share a common memory space, and distributed system where processors communicate by sending messages, the shared memory systems are generally considered easier to use. This is mainly because accessing remote data via a simple memory reference is more familiar to programmers used to conventional sequential programming, than the message send required by distributed systems. The distributed systems are generally more flexible to expand, since there are no shared hardware resources to become overloaded.

There will be a significant increase in the use of languages that allow users to express explicit parallelism. Multiprocessors may perform highly parallel programs much faster than uniprocessors. Applications will no doubt be discovered in which the true multiprocessors becomes improtant.

5. References

[Akl,1985] Selim G. Akl. Parallel Sorting Algorithms, Academic Press Inc. 1985, pp 177-179

[Nicholls, 1988] Bill Nicholls. "That "B" word !", BYTE, June 1988, pp 207

[Boontee, 1987] Boontee Kruatrachue and Ted Lewis. "Duplication Scheduling Heuristic (DSH), A new precedence task scheduler for parallel systems" Tech. Report, Dept. of Computer Science, Oregon State University, 1987.

[BYTE, 1988] Richard Grehan, Tom Thompson, Curtis Franklin Jr., and George A. Stewart. "Introducing the New BYTE Benchmarks", BYTE, June 1988, pp 239

[Cogent,1988] Cogent Research. "XTM System Description", Cogent Research, Oct. 1988

[Deitel, 1984] Harvey M. Deitel. An introduction to operating systems, Addison-Wesley 1984, pp 361-362

[Gel, 1988] David Gelernter. "Linda the portable Parallel", Yale Research Report, Feb. 1987, rev. Jan. 1988

[Hesham, 1988] Hesham El-Rewini and Ted Lewis. "Software Development in Parallax : The ELGDF language", Tech. Report, Dept. of Computer Science, Oregon State University, 1988.

[Hwang and Briggs, 1984] Kai Hwang and Faye' A. Briggs. Computer architecture and parallel processing, McGraw-Hill Inc. 1984, pp 356-357

[Levitan, 1987] Steven P. Levitan. "Measuring communication structure in parallel architectures and algorithms", The characteristics of parallel algorithms, MIT press 1987

[Linpack, 1986] Omri Serlin. "MIPS, Dhrystones, and other tales",Datamation, June 1986, pp 118

[Martin and Riganati, 1988] Joanne L. Martin and John P, Riganati. "Computer community urged to pursue supercomputer benchmarking",Computer, Sep. 1988, pp 68-69

[Sequent, 1985] Sequent Computer Systems, Inc. Guide to Parallel Programming on Sequent computer systems, Sequent Computer Systems, Inc. 1985, pp 1-1 1-2

6. Appendix

Sequent Balance Benchmark programs :

1. Parallel Matrix Multiplication
2. Disk File I/O
3. Many to Many Process Message Passing - Saturating
4. Parallel Enumeration Sort
5. Memory Data Transfer
6. System built-in Mathematic Functions
7. Time File (routines to calculate the execution time)

XTM Linda Benchmark programs :

1. Parallel Matrix Multiplication
2. Disk File I/O
3. Many to Many Process Message Passing - Saturating
4. Parallel Enumeration Sort
5. Memory Data Transfer
6. System built-in Mathematic Functions

Sequent Benchmark programs

```
/*************************************************************/
/*                                                           */
/*     PARALLEL MATRIX MULTIPLICATION BENCHMARK PROGRAM      */
/*                                                           */
/*-----------------------------------------------------------*/
/*   This program is designed for the purpose of testing     */
/* the performance of matrix multiplication on the Sequent   */
/* multiprocessor computers.  The results are presented in   */
/* seconds.                                                  */
/*   This version uses dynamic process scheduling technique  */
/* to keep all processors busy.  Each process consums part   */
/* of the subtasks, vector multiplication, in the matrix.    */
/* The processes access different offsets of the matrix to   */
/* avoid the memory access conflict.  So the result only     */
/* reflect the processor computation speed without process   */
/* communication overhead.                                   */
/*-----------------------------------------------------------*/
/*          SEQUENT BALANCE 21000/DYNIX V3.0.14              */
/*                                                           */
/*  for 4-byte integer matrix multiplication                 */
/*     compile:  cc -DSP matrix.c -lpps -o single            */
/*     run : single  #process                                */
/*                                                           */
/*  for 8-byte double precision matrix multiplication        */
/*     compile:  cc -DDP matrix.c -lpps -o double            */
/*     run : double  #process                                */
/*                                                           */
/*-----------------------------------------------------------*/
/*  Written by: Sun Tieh-Jun                                 */
/*  Internet address : SUN@MIST.CS.ORST.EDU                  */
/*  Last modified : Feb. 20, 1989                            */
/*************************************************************/

#include <stdio.h>
#include <sys/time.h>
#include <parallel/microtask.h>
#include <parallel/parallel.h>

#include "/u1/sun/include/time.include"

#define N1 200              /* number of rows of M or A */
#define N2 200              /* number of columns of A or rows of B */
#define N3 200              /* number of columns of M or B */

#define LDA N1          /* the leading dimensions of A */
#define LDB N2          /* the leading dimensions of B */
#define LDM N1          /* the leading dimensions of M */

#define maxproc 9
#define BITE 1

#ifdef SP
#define PREC int
#define ZERO 0
#define PRECISION "4-byte integer"
#endif
```

```
#ifdef DP
#define PREC double
#define ZERO 0.0e0
#define PRECISION "8-byte Double"
  ndif

shared PREC M[N1][N3];    /* matrix of N1 rows and N3 columns */
shared PREC A[N1][N2];    /* matrix of N1 rows and N2 columns */
shared PREC B[N2][N3];    /* matrix of N2 rows and N3 columns */
shared long childtime[maxproc];  /* store run time of each child process */

long petime[maxproc];

main()
{
  void init_matrix(), mat_mul(), m_set_procs(), m_fork(), m_kill_procs();
  int i, j;
  int nprocs;   /* number of processes to be run in parallel */

  PREC x[LDB];    /* vector used to form y = y + A * x          */
  PREC y[LDM];    /* vector used to accumulate the product of A * x   */

  system("clear");
  printf("\n\n\nPlease enter number of processes to be run in parallel .... ");
  scanf("%d", &nprocs);

  init_matrix();

  printf("\nRandomly generate matrix done .....\n");
  printf("\nParallel Matrix Multiplication start ....\n");
  m_set_procs(nprocs);    /* set number of child processes to be created */

  for (j=0; j<N3; j++)
  {
     for(i=0; i<N1; i++)
       {
         M[i][j]=ZERO;
         y[i]=M[i][j];
       }
     for(i=0; i<N2; i++)
         x[i]=B[i][j];

     start_time(RUSAGE_SELF);
     m_fork(mat_mul, x, y, j);      /* creat child processes */
     stop_time(RUSAGE_SELF);
     accumulate_time();

     for (i=0; i<maxproc;i++)
         petime[i] = petime[i] + childtime[i];

  }

  m_kill_procs();

  system("clear");
  printf("     ****** Run Time of each child process ******\n\n");
  for (i=0; i<nprocs; i++)
```

```
        printf("          node %d ==> %d secs %d usecs. \n",i,
              (petime[i]-(petime[i]%micro))/micro,petime[i]%micro);

    printf("\n     Sequent Balance 21000/ Dynix V3.0.14/ 38038 CPUs\n\n");
    printf("     Parallel Matrix Multiplication benchmark result :\n");
    printf("     ..................................................... \n");
    printf("     time for matrix of size %d X %d ",N1,N3);
    printf("\n      %s",PRECISION);
    printf("\n      %d processes run in parallel. ", nprocs);
    printf("\n\n      ************** TOTAL RUN TIME **************\n");
    printf("                %d secs ",total_sec);
    printf("%d  microsecs\n\n",total_usec);
}


/* initialize matrix function */
void
init_matrix()
{
  PREC randgen();
  int i, j;

  for (i=0; i<N1; i++)
    {
      for (j=0; j<N2; j++)
        A[i][j]= randgen();
    }
  for (i=0; i<N2; i++)
    {
      for (j=0; j<N3; j++)
        B[i][j]= randgen();
    }
  for (i=0; i<maxproc; i++)
    petime[i]=0;
}


/* random number generator */
PREC randgen()
{
 long rand();
 PREC range = 32767;

 return((PREC)rand()/range);
}



/* form one component of y, such that y = y + A * x   */
void mat_mul(x, y, pass)
PREC x[LDB], y[LDM];
int pass;
{
   int i,j;        /* local loop index  */
   int id;       /* process id which is returned from m_next()  */
   int base,top;
```

```
  id = m_get_myid();

  start_time(RUSAGE_SELF);
  while((base=BITE*(m_next()-1)) < N1)
     {
       /* execute all iterations in bite of work */
       top=base+BITE;
       if(top>=N1)
          top=N1;
       for (i=base; i<top; i++)
         {
           /* do vector multiplication */
           for(j=0;j<N2;j++)
               y[i] = y[i] + A[i][j] * x[j];
           /* send a component of y back to the result matrix M */
           M[i][pass] = y[i];
         }
     }
   stop_time(RUSAGE_SELF);
   calc_time();
   childtime[id] = total_micro;
}
```

```
/*************************************************************/
/*                                                           */
/*            DISK FILE I/O BENCHMARK PROGRAM                */
/*                                                           */
/*-----------------------------------------------------------*/
/*   This benchmark program uses low level "C" routines      */
/* to perform disk file I/O.  This version may not be        */
/* portable to other machines.  For those systems without    */
/* the support of low level system I/O routines, you can     */
/* modify this program using "C" high level I/O routines.    */
/*   This program needs 4Mbytes disk space.  This program    */
/* be executed in single user mode to avoid others sharing   */
/* system resource. The result will tell the combination     */
/* performance of I/O controller, I/O channel, disk itself   */
/* and system bus.  Running this program with multiple       */
/* processes is to measure the congestion or bottleneck.     */
/*-----------------------------------------------------------*/
/*                                                           */
/*          SEQUENT BALANCE 21000/DYNIX V3.0.14              */
/*                                                           */
/*      Compile:  cc diskio.c -o diskio -lpps                */
/*                                                           */
/*   NOTE:  make sure you have 4 megabytes disk space        */
/*          available before you run the program.            */
/*          must include "timefile" to get the run time      */
/*-----------------------------------------------------------*/
/* Written by : Sun Tieh-Jun                                 */
/* Internet address: sun@mist.cs.orst.edu                    */
/* Last modified : Mar. 5, 1989                              */
/*************************************************************/

#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <parallel/parallel.h>
#include <parallel/microtask.h>
#include "timefile"

#define maxbuffersize 2L*1024L       /* buffer size */
#define filesize 2L*1024L*1024L  /* the file size 2megabyte */
#define transfersize 2L*1024L        /* number of bytes transfer at a time */


char *readfile = "/tmp/readfile";
char *writefile = "/tmp/writefile";

main()
{
  void m_fork(), m_kill_procs(), setup(), readwrite_par();
  int nprocs;
  int block;
  system("clear");
  printf("Disk file I/O benchmark program start ....\n\n");
  printf("please enter number of processes to be run in parallel : ");
  scanf("%d",&nprocs);
  printf("\n\nNow creating two 2-Mbytes files for read and write.....\n\n");
  setup();
```

```
  printf("File initial ok.  Start disk file I/O.....\n\n");
  block = filesize/transfersize;
  m_set_procs(nprocs);
  set_timer();
  start_timer();
  m_fork(readwrite_par,block);
  m_kill_procs();
  stop_timer();
  unlink(readfile);
  unlink(writefile);
  printf("\n\n        Sequent Balance 21000 Disk file I/O benchmark  \n");
  printf("        ******************************************\n");
  printf("                two 2-Mbytes disk files I/O\n");
  printf("\n               %d processes run in parallel. \n\n",nprocs);
  calc_time();


}

void
setup()
{
  int i;
  int fdr,fdw;
  unsigned int nbytes;
  char buffer[maxbuffersize];

  if ((fdr = creat(readfile,0640))<0)
    {
       printf("creat read file fail");
       exit(1);
     }

  close(fdr);

  if ((fdw = creat(writefile,0640))<0)
    {
       printf("creat write file fail");
       exit(1);
     }
  close(fdw);

  if((fdr = open(readfile,2))<0)
    {
      printf("can not open read file");
      exit(1);
    }


  lseek(fdr,0L,0);
  for (i=0; i<filesize/transfersize; i++)
    {
      if ((nbytes = write(fdr, buffer, transfersize))<0)
        {
          printf("initial file error");
          exit(1);
        }
```

```
    }
  close(fdr);
}

void
readwrite_par(blocks)
int blocks;
{
 int i,j;
 int id, nprocs;
 int fdrr, fdww;
 char buf[maxbuffersize];
 unsigned int nbytes;

  if ((fdrr=open(readfile,2))<0)
    {
      printf("can not open read file");
      exit(1);
    }
  if ((fdww=open(writefile,2))<0)
    {
      printf("can not open write file");
      exit(1);
    }

  nprocs = m_get_numprocs();
  id = m_get_myid();
  for (i=id; i<blocks; i+=nprocs)
    {

      lseek(fdrr,(long)(i*transfersize),0);
      if ((nbytes = read(fdrr,buf,transfersize))<0)
        {
          printf("read error");
          exit(1);
        }
      lseek(fdrr,0L,0);
      lseek(fdww,(long)(i*transfersize),0);
      if ((nbytes = write(fdww,buf,transfersize))<0)
        {
          printf("write error");
          exit(1);
        }
      lseek(fdww,0L,0);

    }
}
```

```
/*************************************************************/
/*                                                           */
/*  Many-To-Many processes message passing benchmark program */
/*                                                           */
/*-----------------------------------------------------------*/
/*     This program performs the task which every process    */
/*  sends message to all other processes.  This is a many    */
/*  to many communication.  Processes send and receive data  */
/*  by writing and reading the shared memory.  For those     */
/*  machine without shared memory, please use its own message */
/*  passing technique.  On sequent BALANCE 21000 system,     */
/*  message passing is taking advantage of accessing shared  */
/*  memory.                                                  */
/*     In the program, each process sends its ID along with  */
/*  a message to the shared memory.  After all processes     */
/*  completing "send", each process reads in all messages    */
/*  passed by all other processes from the shared memory.    */
/*-----------------------------------------------------------*/
/*            SEQUENT BALANCE 21000/DYNIX v3.0.14            */
/*                                                           */
/*       cc saturate.c -o saturate -lpps                     */
/*       run : saturate                                      */
/*       Note: must include "timefile" to get run time       */
/*-----------------------------------------------------------*/
/*  Written by Sun Tieh-Jun                                  */
/*  Internet address : sun@mist.cs.orst.edu                  */
/*  last modified : Jan 20, 1989                             */
/*************************************************************/

#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <parallel/parallel.h>
#include <parallel/microtask.h>
#include "timefile"

#define ITER 1000       /* number of iterations to be tested */
#define MAXCPU  9       /* the maximum number of CPUs can be used */
#define MESGSIZE 100    /* the size of message to be passed */

typedef struct {
        int pid;
        double message[MESGSIZE];
      } message_type;

double mesg[MESGSIZE];

shared message_type common[MAXCPU][MAXCPU];
shared sbarrier_t barrier;

sbarrier_t *bp;

main()
{
void  init(), saturate(), m_fork(), m_set_procs(), m_kill_procs();
int i;
int nprocs;     /*  number of processes */
```

```
system("clear");
printf("enter number of processes (1 - 9) to perform message passing : ");
scanf("%d",&nprocs);
 printf("\n\n%d-To-%d message passing start ......\n",nprocs,nprocs);
printf("\nSize = array of %d double\n",MESGSIZE);
printf("\nIteration = %d\n\n\n\n",ITER);
bp = &barrier;
s_init_barrier(bp,nprocs);
m_set_procs(nprocs);

init();

for (i=0;i<ITER;i++)
  {
   set_timer();
   start_timer();
   m_fork(saturate);
   stop_timer();
   accumulate_time();
 }
m_kill_procs();

printf("      Sequent Balance 21000 %d-To-%d Message Passing \n",nprocs,nprocs);
printf("     ***********************************************\n\n");
printf("              %d secs  %d microseconds\n\n",total_sec,total_usec);

}

void init()
{
int i;
for (i=0; i<MESGSIZE; i++)
      mesg[i] = (double) rand();
}


void
saturate()
{
int m_get_numprocs(), m_get_myid();
int i,n;
int id;
int nprocs;
message_type local[MAXCPU];

id = m_get_myid();
nprocs = m_get_numprocs();

for(i=0; i < nprocs; i++)  /* each process send its id and data out */
     {
       common[i][id].pid = id;
       for (n=0; n < MESGSIZE; n++)
         common[i][id].message[n] = mesg[n];
     }

s_wait_barrier(bp);       /* wait here until all processes complete sending */
```

```
for (i=0; i<nprocs; i++)  /* each process receive data from all others */
      {
         local[i].pid = common[id][i].pid;
         for (n=0; n < MESGSIZE; n++)
            local[i].message[n] = common[id][i].message[n];
       }
}
```

```
/*********************************************************/
/*                                                       */
/*    PARALLEL ENUMERATION SORT BENCHMARK PROGRAM        */
/*                                                       */
/*-------------------------------------------------------*/
/*       This benchmark program uses the parallel        */
/*  enumeration sort algorithm to test the speed of      */
/*  integer comparison and shared memory access.         */
/*                                                       */
/*  The program sorts a sequence of data by comparing    */
/*  each element with all the elements to obtain its     */
/*  rank in the sequence.  The unsorted array is in      */
/*  the shared memory for each individual element to     */
/*  compare with.  Data partitioning is used that        */
/*  each process performs more than one tasks (an        */
/*  element compares with whole unsortrd array).         */
/*-------------------------------------------------------*/
/*                                                       */
/*         SEQUENT BALANCE 21000/DYNIX v3.0.14           */
/*                                                       */
/*   compile : cc sort.c -lpps -o sort                   */
/*   run: sort                                           */
/*                                                       */
/*-------------------------------------------------------*/
/*   Written by : Sun Tieh-Jun                           */
/*   Internet address :  sun@mist.cs.orst.edu            */
/*   Last modified : February 15, 1989                   */
/*********************************************************/


#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <parallel/parallel.h>
#include <parallel/microtask.h>
#include "timefile"

#define SIZE 1024

shared int source[SIZE];      /* Original unsorted sequence */
shared int sorted[SIZE];      /* Sorted sequence */



main()
{
void initial(),sort(), m_fork(), m_set_procs(), m_kill_procs();
int i;
int nprocs;     /*  number of processes */

printf("Parallel sorting benchmark program start....\n\n");
printf("Please enter number of processes to be run in parallel: ");
scanf("%d", &nprocs);

printf("\n\nRandom generate %d elements .......\n\n",SIZE);
initial();
```

```
printf("Parallel sorting start.....\n\n");

set_timer();
start_timer();
m_set_procs(nprocs);
m_fork(sort);
m_kill_procs();

stop_timer();

printf("\n\n\n\nPARALLEL SORTING BENCHMARK RESULT\n");
printf("*********************************\n\n");
printf("%d elements sorted by %d processes\n\n",SIZE,nprocs);

calc_time();

}



void initial()
{
 int i;

 for (i=0; i<SIZE; i++)
   source[i] = rand() % 10000;
}

void sort()
{
 int m_get_myid(), m_get_numprocs();
 int i, j, k ;
 int id;
 int nprocs;

 nprocs = m_get_numprocs();   /* get number of processes */
 for (id= m_get_myid(); id<SIZE; id += nprocs)
   {
     k = -1;
     for (j=0; j<SIZE; j++)
         {
           if (source[id] > source[j])
               k++;
           else
             if ((source[id]==source[j]) && (id>j))
                k++;
         }
    /* Now, locate one element to its place in the sorted array */
    sorted[k+1] = source[id];
   }
}
```

```
/*************************************************************/
/*                                                           */
/*      SHARED MEMORY BYTES TRANSFER BENCHMARK PROGRAM       */
/*                                                           */
/*-----------------------------------------------------------*/
/*  This program tests the speed of moving a byte, 2 bytes,  */
/*  4 bytes, 8 bytes up to 4096 bytes from a shared memory   */
/*  location to another shared memory location.              */
/*-----------------------------------------------------------*/
/*                                                           */
/*            SEQUENT BALANCE 21000/DYNIX v3.0.14            */
/*                                                           */
/*    Compile : cc mtransfer.c -o transfer -lpps             */
/*        run : transfer                                     */
/*                                                           */
/*    note: include "timefile" to get the run time           */
/*-----------------------------------------------------------*/
/*  Written by : Sun Tieh-Jun                                */
/*  Internet address : sun@mist.cs.orst.edu                  */
/*  Last modified : Jan 20, 1989                             */
/*************************************************************/


#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <parallel/parallel.h>
#include <parallel/microtask.h>
#include "timefile"

#define MAXCPU 9
#define iteration 10000

/*  below are two arrays which store the pointers to a shared memory */
/*  location.  shsrcmem is for original location, shdesmem is for    */
/*  destination location.                                            */

shared char *shsrcmem[MAXCPU];
shared char *shdesmem[MAXCPU];


main()
{
  void m_fork(), m_kill_procs(), process();
  unsigned number_bytes;
  int i, nprocs;

  system ("clear");
  printf("Shared memory location transfer benchmark program start....\n\n");
  printf("Please enter number of processes to perform memory transfer :");
  scanf("%d",&nprocs);
  printf("Iteration = iteration");
  m_set_procs(nprocs);
  set_timer();
  start_timer();

  /* testing 1 byte, 2 bytes, 4 bytes and up to 4096 bytes movement */
```

```
  for (i=0;i<iteration;i++)
    {
      for (number_bytes=1; number_bytes<=4096; number_bytes*=2)
          m_fork(process,number_bytes);

    }

  stop_timer();
  m_kill_procs();

  printf("\n\n\n     Sequent Balance 21000 Shared memory transfer result\n");
  printf("      ************************************************\n\n");
  printf("  1byte, 2 bytes, 4 bytes and up to 4096 bytes are transfered\n");
  calc_time();

}

void
process(size)
unsigned size;
{
 void allocate(), move(), release();

 allocate(size);
 move();
 release();
}


void
allocate(size)
unsigned size;

  char *shmalloc();
  int m_get_myid();
  int id;

  id  = m_get_myid();

  /* allocate size bytes block for source and destination */
 if((shsrcmem[id] = shmalloc(size)) == 0)
   {
    printf("No free space for %d bytes source or out of bounds",size);
    exit(1);
  }

 if((shdesmem[id] = shmalloc(size)) == 0)
   {
    printf("No free space for %d bytes destination or out of bounds",size);
    exit(1);
  }


}
```

```
void
move()
{
  int m_get_myid();
  int id;
  int i;

  id = m_get_myid();
  /* move size bytes data from source to destination */

    *shdesmem[id] = *shsrcmem[id];

}


void
release()
{
 void shfree();
 int m_get_myid();
 int id;

 id = m_get_myid();
 /* release the shared memory */
 shfree(shsrcmem[id]);
 shfree(shdesmem[id]);

}
```

```
/*************************************************************/
/*                                                           */
/*         MATHEMATIC FUNCTION BENCHMARK PROGRAM             */
/*                                                           */
 *-----------------------------------------------------------*/
 *   This program performs the task to test the performance  */
/*  of executing mathematic functions of DYNIX system.       */
/*                                                           */
/*  (1) trigonometric functions :                            */
/*       sin(), cos(), tan(), asin(), acos(), atan()         */
/*                                                           */
/*  (2) logarithm and squart root :                          */
/*       exp(), log(), log10(), pow(), sqrt()                */
/*                                                           */
/*  (3) absolute value and approximate value                 */
/*       fabs(), floor(), ceil()                             */
/*                                                           */
/*-----------------------------------------------------------*/
/*                                                           */
/*          SEQUENT BALANCE 21000/DYNIX v3.0.14              */
/*                                                           */
/*    Compile : cc math.c -o math -lm                        */
/*       run : math                                          */
/*                                                           */
/*-----------------------------------------------------------*/
/*  Written by : Sun Tieh-Jun                                */
/*  Internet address : sun@mist.cs.orst.edu                  */
/*  Last modified :  Feb. 20,1989                            */
/*************************************************************/

#include <stdio.h>
#include <sys/time.h>
 include <sys/resource.h>
#include <math.h>
#include "timefile"

#define ITER 10000              /* number of iterations to be executed */
#define const1 0.50             /* these constants are used as initial */
#define const2 180.00           /* values for math functions.  This    */
#define const3 100.00           /* will make all functions return the  */
#define const4 25               /* valid values and avoid overflow.    */
#define const5 -1.4999

main()
{
 void trig(), log_sqrt(), absolute();

system("clear");
set_timer();
start_timer();

printf("System built-in Math function testing start ...\n\n");
printf("executing Trigonometric functions ... \n\n");
trig();
printf("executing Logarithm and Square Root functions ... \n\n");
log_sqrt();
printf("executing Absolute and Approximate value functions ...\n\n");
```

```
absolute();

stop_timer();
printf("\n\n\n          Sequent Balance 21000/Dynix v3.0.14 \n\n");
printf("        Built-in Mathematic Function benchmark result \n\n");
alc_time();

}


void
trig()
{
 double x = const1;
 int i;

 for(i=0; i<ITER; i++)              /*  each iteration almost transform   */
   {                                /*  x back to its initial value. The  */
     x = asin(sin(x)*cos(x));       /*  variation is very slow.           */
     x = acos(sin(x)*cos(x));
     x = atan(tan(x));
     x /= const2;
   }

}


void
log_sqrt()
{
 int i;
 double x = const3;

 for(i=0; i<ITER; i++)
   {
     x = sqrt(exp(log(log10(x))/const1));  /* each iteration transform x */
     x = pow(x,x);                         /* back to its initial value  */
     x *= const4;                          /* by multiplying a constant. */
   }
}


void
absolute()
{
 double x = const5;
 int i;

 for(i=0; i<ITER; i++)
   {
     x = ceil(floor(x)/fabs(x));    /*  each iteration transform x back */
     x *= fabs(const5);             /*  to its initial value.           */
   }

}
```

```
/************************************************************/
/*  Linpack Benchmark program used by Sun Tieh-Jun          */
/************************************************************/


/*

Translated to C by Bonnie Toy 5/88


To obtain rolled source BLAS, add -DROLL to the command lines.
To obtain unrolled source BLAS, add -DUNROLL to the command lines.

You must specify one of -DSP or -DDP to compile correctly.

You must specify one of -DROLL or -DUNROLL to compile correctly.

*/

#ifdef SP
#define REAL float
#define ZERO 0.0
#define ONE 1.0
#define PREC "Single "
#endif

#ifdef DP
#define REAL double
#define ZERO 0.0e0
#define ONE 1.0e0
#define PREC "Double "
#endif

#define NTIMES 10

#ifdef ROLL
#define ROLLING "Rolled "
#endif
#ifdef UNROLL
#define ROLLING "Unrolled "
#endif

#include <stdio.h>
#include <math.h>

static REAL time[8][6];

main ()
{
        static REAL aa[200][200],a[200][201],b[200],x[200];
        REAL cray,ops,total,norma,normx;
        REAL resid,residn,eps,tl,tm,tm2;
        REAL epslon(),second(),kf;
        static int ipvt[200],n,i,ntimes,info,lda,ldaa,kflops;

        lda = 201;
```

```
        ldaa = 200;
        cray = .056;
        n = 100;

        fprintf(stdout,ROLLING);fprintf(stdout,PREC);fprintf(stdout,"Precision Linpack\n\n'
        fprintf(stderr,ROLLING);fprintf(stderr,PREC);fprintf(stderr,"Precision Linpack\n\n'

        ops = (2.0e0*(n*n*n))/3.0 + 2.0*(n*n);

        matgen(a,lda,n,b,&norma);
        t1 = second();
        dgefa(a,lda,n,ipvt,&info);
        time[0][0] = second() - t1;
        t1 = second();
        dgesl(a,lda,n,ipvt,b,0);
        time[1][0] = second() - t1;
        total = time[0][0] + time[1][0];

/*      compute a residual to verify results.  */

        for (i = 0; i < n; i++) {
                x[i] = b[i];
        }
        matgen(a,lda,n,b,&norma);
        for (i = 0; i < n; i++) {
                b[i] = -b[i];
        }
        dmxpy(n,b,n,lda,x,a);
        resid = 0.0;
        normx = 0.0;
        for (i = 0; i < n; i++) {
                resid = (resid > fabs((double)b[i]))
                        ? resid : fabs((double)b[i]);
                normx = (normx > fabs((double)x[i]))
                        ? normx : fabs((double)x[i]);
        }
        eps = epslon((REAL)ONE);
        residn = resid/( n*norma*normx*eps );

        printf("     norm. resid      resid           machep");
        printf("         x[0]-1        x[n-1]-1\n");
        printf("  %8.1f      %16.8e%16.8e%16.8e%16.8e\n",
               (double)residn, (double)resid, (double)eps,
               (double)x[0]-1, (double)x[n-1]-1);

        fprintf(stderr,"    times are reported for matrices of order %5d\n",n);
        fprintf(stderr,"      dgefa      dgesl      total      kflops     unit");
        fprintf(stderr,"      ratio\n");

        time[2][0] = total;
        time[3][0] = ops/(1.0e3*total);
        time[4][0] = 2.0e3/time[3][0];
        time[5][0] = total/cray;

        fprintf(stderr," times for array with leading dimension of%5d\n",lda);
        print_time(0);
```

```
matgen(a,lda,n,b,&norma);
t1 = second();
dgefa(a,lda,n,ipvt,&info);
time[0][1] = second() - t1;
t1 = second();
dgesl(a,lda,n,ipvt,b,0);
time[1][1] = second() - t1;
total = time[0][1] + time[1][1];
time[2][1] = total;
time[3][1] = ops/(1.0e3*total);
time[4][1] = 2.0e3/time[3][1];
time[5][1] = total/cray;

matgen(a,lda,n,b,&norma);
t1 = second();
dgefa(a,lda,n,ipvt,&info);
time[0][2] = second() - t1;
t1 = second();
dgesl(a,lda,n,ipvt,b,0);
time[1][2] = second() - t1;
total = time[0][2] + time[1][2];
time[2][2] = total;
time[3][2] = ops/(1.0e3*total);
time[4][2] = 2.0e3/time[3][2];
time[5][2] = total/cray;

ntimes = NTIMES;
tm2 = 0.0;
t1 = second();

for (i = 0; i < ntimes; i++) {
        tm = second();
        matgen(a,lda,n,b,&norma);
        tm2 = tm2 + second() - tm;
        dgefa(a,lda,n,ipvt,&info);
}

time[0][3] = (second() - t1 - tm2)/ntimes;
t1 = second();

for (i = 0; i < ntimes; i++) {
        dgesl(a,lda,n,ipvt,b,0);
}

time[1][3] = (second() - t1)/ntimes;
total = time[0][3] + time[1][3];
time[2][3] = total;
time[3][3] = ops/(1.0e3*total);
time[4][3] = 2.0e3/time[3][3];
time[5][3] = total/cray;

print_time(1);
print_time(2);
print_time(3);

matgen(aa,ldaa,n,b,&norma);
t1 = second();
```

```
dgefa(aa,ldaa,n,ipvt,&info);
time[0][4] = second() - t1;
t1 = second();
dgesl(aa,ldaa,n,ipvt,b,0);
time[1][4] = second() - t1;
total = time[0][4] + time[1][4];
time[2][4] = total;
time[3][4] = ops/(1.0e3*total);
time[4][4] = 2.0e3/time[3][4];
time[5][4] = total/cray;

matgen(aa,ldaa,n,b,&norma);
t1 = second();
dgefa(aa,ldaa,n,ipvt,&info);
time[0][5] = second() - t1;
t1 = second();
dgesl(aa,ldaa,n,ipvt,b,0);
time[1][5] = second() - t1;
total = time[0][5] + time[1][5];
time[2][5] = total;
time[3][5] = ops/(1.0e3*total);
time[4][5] = 2.0e3/time[3][5];
time[5][5] = total/cray;

matgen(aa,ldaa,n,b,&norma);
t1 = second();
dgefa(aa,ldaa,n,ipvt,&info);
time[0][6] = second() - t1;
t1 = second();
dgesl(aa,ldaa,n,ipvt,b,0);
time[1][6] = second() - t1;
total = time[0][6] + time[1][6];
time[2][6] = total;
time[3][6] = ops/(1.0e3*total);
time[4][6] = 2.0e3/time[3][6];
time[5][6] = total/cray;

ntimes = NTIMES;
tm2 = 0;
t1 = second();
for (i = 0; i < ntimes; i++) {
        tm = second();
        matgen(aa,ldaa,n,b,&norma);
        tm2 = tm2 + second() - tm;
        dgefa(aa,ldaa,n,ipvt,&info);
}
time[0][7] = (second() - t1 - tm2)/ntimes;
t1 = second();
for (i = 0; i < ntimes; i++) {
        dgesl(aa,ldaa,n,ipvt,b,0);
}
time[1][7] = (second() - t1)/ntimes;
total = time[0][7] + time[1][7];
time[2][7] = total;
time[3][7] = ops/(1.0e3*total);
time[4][7] = 2.0e3/time[3][7];
time[5][7] = total/cray;
```

```
        /* the following code sequence implements the semantics of
           the Fortran intrinsics "nint(min(time[3][3],time[3][7]))"    */

        kf = (time[3][3] < time[3][7]) ? time[3][3] : time[3][7];
        kf = (kf > ZERO) ? (kf + .5) : (kf - .5);
        if (fabs((double)kf) < ONE)
                kflops = 0;
        else {
                kflops = floor(fabs((double)kf));
                if (kf < ZERO) kflops = -kflops;
        }

        fprintf(stderr," times for array with leading dimension of%4d\n",ldaa);
        print_time(4);
        print_time(5);
        print_time(6);
        print_time(7);
        fprintf(stderr,ROLLING);fprintf(stderr,PREC);
        fprintf(stderr," Precision %5d Kflops ; %d Reps \n",kflops,NTIMES);
}

/*----------------------*/
print_time (row)
int row;
{
fprintf(stderr,"%11.2f%11.2f%11.2f%11.0f%11.2f%11.2f\n",    (double)time[0][row],
       (double)time[1][row], (double)time[2][row], (double)time[3][row],
       (double)time[4][row], (double)time[5][row]);
}

/*----------------------*/
  tgen(a,lda,n,b,norma)
REAL a[],b[],*norma;
int lda, n;

/* We would like to declare a[][lda], but c does not allow it.  In this
function, references to a[i][j] are written a[lda*i+j].  */

{
        int init, i, j;

        init = 1325;
        *norma = 0.0;
        for (j = 0; j < n; j++) {
                for (i = 0; i < n; i++) {
                        init = 3125*init % 65536;
                        a[lda*j+i] = (init - 32768.0)/16384.0;
                        *norma = (a[lda*j+i] > *norma) ? a[lda*j+i] : *norma;
                }
        }
        for (i = 0; i < n; i++) {
          b[i] = 0.0;
        }
        for (j = 0; j < n; j++) {
                for (i = 0; i < n; i++) {
                        b[i] = b[i] + a[lda*j+i];
```

```
            }
        }
}

  ----------------------*/
dgefa(a,lda,n,ipvt,info)
REAL a[];
int lda,n,ipvt[],*info;

/* We would like to declare a[][lda], but c does not allow it.  In this
function, references to a[i][j] are written a[lda*i+j].  */
/*
     dgefa factors a double precision matrix by gaussian elimination.

     dgefa is usually called by dgeco, but it can be called
     directly with a saving in time if  rcond  is not needed.
     (time for dgeco) = (1 + 9/n)*(time for dgefa) .

     on entry

        a       REAL precision[n][lda]
                the matrix to be factored.

        lda     integer
                the leading dimension of the array  a .

        n       integer
                the order of the matrix  a .

     on return

        a       an upper triangular matrix and the multipliers
                which were used to obtain it.
                the factorization can be written  a = l*u  where
                l  is a product of permutation and unit lower
                triangular matrices and  u  is upper triangular.

        ipvt    integer[n]
                an integer vector of pivot indices.

        info    integer
                = 0  normal value.
                = k  if  u[k][k] .eq. 0.0 .  this is not an error
                     condition for this subroutine, but it does
                     indicate that dgesl or dgedi will divide by zero
                     if called.  use  rcond  in dgeco for a reliable
                     indication of singularity.

     linpack. this version dated 08/14/78 .
     cleve moler, university of new mexico, argonne national lab.

     functions

     blas daxpy,dscal,idamax
*/

{
```

```
/*     internal variables          */

REAL t;
int idamax(),j,k,kp1,l,nm1;


/*     gaussian elimination with partial pivoting       */

        *info = 0;
        nm1 = n - 1;
        if (nm1 >=  0) {
                for (k = 0; k < nm1; k++) {
                        kp1 = k + 1;

                        /* find l = pivot index */

                        l = idamax(n-k,&a[lda*k+k],1) + k;
                        ipvt[k] = l;

                        /* zero pivot implies this column already
                           triangularized */

                        if (a[lda*k+l] != ZERO) {

                                /* interchange if necessary */

                                if (l != k) {
                                        t = a[lda*k+l];
                                        a[lda*k+l] = a[lda*k+k];
                                        a[lda*k+k] = t;
                                }

                                /* compute multipliers */

                                t = -ONE/a[lda*k+k];
                                dscal(n-(k+1),t,&a[lda*k+k+1],1);

                                /* row elimination with column indexing */

                                for (j = kp1; j < n; j++) {
                                        t = a[lda*j+l];
                                        if (l != k) {
                                                a[lda*j+l] = a[lda*j+k];
                                                a[lda*j+k] = t;
                                        }
                                        daxpy(n-(k+1),t,&a[lda*k+k+1],1,
                                              &a[lda*j+k+1],1);
                                }
                        }
                        else {
                                *info = k;
                        }
                }
        }
        ipvt[n-1] = n-1;
        if (a[lda*(n-1)+(n-1)] == ZERO) *info = n-1;
}
```

```
/*----------------------*/

dgesl(a,lda,n,ipvt,b,job)
  t lda,n,ipvt[],job;
REAL a[],b[];

/* We would like to declare a[][lda], but c does not allow it.  In this
function, references to a[i][j] are written a[lda*i+j].  */

/*
     dgesl solves the double precision system
     a * x = b  or  trans(a) * x = b
     using the factors computed by dgeco or dgefa.

     on entry

        a       double precision[n][lda]
                the output from dgeco or dgefa.

        lda     integer
                the leading dimension of the array  a .

        n       integer
                the order of the matrix  a .

        ipvt    integer[n]
                the pivot vector from dgeco or dgefa.

        b       double precision[n]
                the right hand side vector.

        job     integer
                = 0         to solve  a*x = b ,
                = nonzero   to solve  trans(a)*x = b  where
                            trans(a)  is the transpose.

    on return

        b       the solution vector  x .

     error condition

        a division by zero will occur if the input factor contains a
        zero on the diagonal.  technically this indicates singularity
        but it is often caused by improper arguments or improper
        setting of lda .  it will not occur if the subroutines are
        called correctly and if dgeco has set rcond .gt. 0.0
        or dgefa has set info .eq. 0 .

     to compute  inverse(a) * c  where  c  is a matrix
     with  p  columns
           dgeco(a,lda,n,ipvt,rcond,z)
           if (!rcond is too small){
                for (j=0,j<p,j++)
                        dgesl(a,lda,n,ipvt,c[j][0],0);
           }
```

```
    linpack. this version dated 08/14/78 .
    cleve moler, university of new mexico, argonne national lab.

    functions

    blas daxpy,ddot
*/
{
/*     internal variables       */

    REAL ddot(),t;
    int k,kb,l,nm1;

    nm1 = n - 1;
    if (job == 0) {

            /* job = 0 , solve  a * x = b
               first solve  l*y = b          */

            if (nm1 >= 1) {
                    for (k = 0; k < nm1; k++) {
                            l = ipvt[k];
                            t = b[l];
                            if (l != k){
                                    b[l] = b[k];
                                    b[k] = t;
                            }
                            daxpy(n-(k+1),t,&a[lda*k+k+1],1,&b[k+1],1);
                    }
            }

            /* now solve  u*x = y */

            for (kb = 0; kb < n; kb++) {
                k = n - (kb + 1);
                b[k] = b[k]/a[lda*k+k];
                t = -b[k];
                daxpy(k,t,&a[lda*k+0],1,&b[0],1);
            }
    }
    else {

            /* job = nonzero, solve  trans(a) * x = b
               first solve  trans(u)*y = b                    */

            for (k = 0; k < n; k++) {
                    t = ddot(k,&a[lda*k+0],1,&b[0],1);
                    b[k] = (b[k] - t)/a[lda*k+k];
            }

            /* now solve trans(l)*x = y     */

            if (nm1 >= 1) {
                    for (kb = 1; kb < nm1; kb++) {
                            k = n - (kb+1);
                            b[k] = b[k] + ddot(n-(k+1),&a[lda*k+k+1],1,&
```

```
        /* cleanup odd group of four vectors */

        j = n2 % 8;
        if (j >= 4) {
                j = j - 1;
                for (i = 0; i < n1; i++)
                        y[i] = ((( (y[i])
                                + x[j-3]*m[ldm*(j-3)+i])
                                + x[j-2]*m[ldm*(j-2)+i])
                                + x[j-1]*m[ldm*(j-1)+i]) + x[j]*m[ldm*j+i];
        }

        /* cleanup odd group of eight vectors */

        j = n2 % 16;
        if (j >= 8) {
                j = j - 1;
                for (i = 0; i < n1; i++)
                        y[i] = (((((((  (y[i])
                                + x[j-7]*m[ldm*(j-7)+i]) + x[j-6]*m[ldm*(j-6)+i])
                                + x[j-5]*m[ldm*(j-5)+i]) + x[j-4]*m[ldm*(j-4)+i])
                                + x[j-3]*m[ldm*(j-3)+i]) + x[j-2]*m[ldm*(j-2)+i])
                                + x[j-1]*m[ldm*(j-1)+i]) + x[j]  *m[ldm*j+i];
        }

        /* main loop - groups of sixteen vectors */

        jmin = (n2%16)+16;
        for (j = jmin-1; j < n2; j = j + 16) {
                for (i = 0; i < n1; i++)
                        y[i] = ((((((((((((((  (y[i])
                                 + x[j-15]*m[ldm*(j-15)+i])
                                 + x[j-14]*m[ldm*(j-14)+i])
                                 + x[j-13]*m[ldm*(j-13)+i])
                                 + x[j-12]*m[ldm*(j-12)+i])
                                 + x[j-11]*m[ldm*(j-11)+i])
                                 + x[j-10]*m[ldm*(j-10)+i])
                                 + x[j- 9]*m[ldm*(j- 9)+i])
                                 + x[j- 8]*m[ldm*(j- 8)+i])
                                 + x[j- 7]*m[ldm*(j- 7)+i])
                                 + x[j- 6]*m[ldm*(j- 6)+i])
                                 + x[j- 5]*m[ldm*(j- 5)+i])
                                 + x[j- 4]*m[ldm*(j- 4)+i])
                                 + x[j- 3]*m[ldm*(j- 3)+i])
                                 + x[j- 2]*m[ldm*(j- 2)+i])
                                 + x[j- 1]*m[ldm*(j- 1)+i])
                                 + x[j]   *m[ldm*j+i];
        }
}

/*----------------------*/
REAL second()
{
#include <sys/time.h>
#include <sys/resource.h>

struct rusage ru;
```

```
REAL t ;

void getrusage();

  trusage(RUSAGE_SELF,&ru) ;

t = (REAL) (ru.ru_utime.tv_sec+ru.ru_stime.tv_sec) +
    ((REAL) (ru.ru_utime.tv_usec+ru.ru_stime.tv_usec))/1.0e6 ;
return t ;
}
```

```
/*****************************************************************/
/*  TIME file:                                                   */
/*                                                               */
/*  last modified: March 26, 1989                                */
/*                                                               */
/*  Written by:  Jisheng Shen & Tieh-Jun Sun                     */
/*                                                               */
/*****************************************************************/

#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>

#define  micro  1000000

long u_start_sec, u_start_usec;
long u_stop_sec, u_stop_usec;
long u_bench_sec, u_bench_usec;
long s_start_sec, s_start_usec;
long s_stop_sec, s_stop_usec;
long s_bench_sec, s_bench_usec;
long subtotal_usec; subtotal_sec, submicro;
long total_time = 0, total_sec, total_usec, total_micro;
long net_usec, net_sec;
long u_null_sec, u_null_usec;

typedef struct rusage ELEMENT;
typedef ELEMENT       *PTR;
PTR                   rusage;


void
op_time(who,loop)
int who,loop;
{
   int   i;
   rusage = (PTR) malloc(sizeof(ELEMENT));
   getrusage(who, rusage);
   u_start_sec = rusage->ru_utime.tv_sec;
   u_start_usec = rusage->ru_utime.tv_usec;

   for (i=0; i<loop; ++i);
   getrusage(who, rusage);
   u_null_sec = rusage->ru_utime.tv_sec - u_start_sec;
   u_null_usec = rusage->ru_utime.tv_usec - u_start_usec;
}

void
start_time(who)
int who;
{
   rusage = (PTR) malloc(sizeof(ELEMENT));
   getrusage(who, rusage);
   u_start_sec = rusage->ru_utime.tv_sec;
   u_start_usec = rusage->ru_utime.tv_usec;
   s_start_sec = rusage->ru_stime.tv_sec;
   s_start_usec = rusage->ru_stime.tv_usec;
```

```
}

void
stop_time(who)
{
    getrusage(who, rusage);
    u_stop_sec = rusage->ru_utime.tv_sec;
    u_stop_usec = rusage->ru_utime.tv_usec;
    s_stop_sec = rusage->ru_stime.tv_sec;
    s_stop_usec = rusage->ru_stime.tv_usec;
}

void
accumulate_time()
{
  free(rusage);
  if (u_stop_usec < u_start_usec) {
      u_stop_usec = u_stop_usec + micro;
      u_stop_sec--;
    }

  if (s_stop_usec < s_start_usec) {
      s_stop_usec = s_stop_usec + micro;
      s_stop_sec--;
    }

  u_bench_sec = u_stop_sec - u_start_sec;
  u_bench_usec = u_stop_usec - u_start_usec;
  s_bench_sec = s_stop_sec - s_start_sec;
  s_bench_usec = s_stop_usec - s_start_usec;
  subtotal_sec = u_bench_sec + s_bench_sec;
  subtotal_usec = u_bench_usec + s_bench_usec;
  submicro = subtotal_sec * micro + subtotal_usec;
  total_time = total_time + submicro;
  total_usec = total_time % micro;
  total_sec = (total_time - total_usec) / micro;

}

void
net_accumulate()
{
   net_sec = total_sec - u_null_sec;
   net_usec = total_usec - u_null_usec;
   if (net_usec < 0) {
       net_usec = net_usec + micro;
       net_sec--;
    }
}


void
calc_time()
{
  free(rusage);
  if (u_stop_usec < u_start_usec) {
```

```
        u_stop_usec = u_stop_usec + micro;
        u_stop_sec--;
      }

    if (s_stop_usec < s_start_usec) {
        s_stop_usec = s_stop_usec + micro;
        s_stop_sec--;
      }

    u_bench_sec = u_stop_sec - u_start_sec;
    u_bench_usec = u_stop_usec - u_start_usec;
    s_bench_sec = s_stop_sec - s_start_sec;
    s_bench_usec = s_stop_usec - s_start_usec;

    total_sec = u_bench_sec + s_bench_sec;
    total_usec = u_bench_usec + s_bench_usec;
    total_micro = total_sec * micro + total_usec;

    if (total_usec > micro) {
        total_usec = total_usec - micro;
        total_sec++;
      }

    if (total_usec < 0) {
        total_usec = total_usec + micro;
        total_sec--;
      }
}
```

XTM Linda benchmark programs

```
/*****************************************************************/
/*                                                               */
/*       PARALLEL MATRIX MULTIPLICATION BENCHMARK PROGRAM        */
/*                                                               */
/*---------------------------------------------------------------*/
                                                                 */
/*   For 4-byte integer matrix multiplication                    */
/*     compile : cc -DSP -o matrix matrix.c                      */
/*                                                               */
/*   For 8-byte double precision matrix multiplication           */
/*     compile : cc -DDP -o matrix matrix.c                      */
/*                                                               */
/*   run : matrix #process                                       */
/*---------------------------------------------------------------*/
/*   written by : Sun Tieh-Jun                                   */
/*   Internet address : sun@mist.cs.orst.edu                     */
/*   last modified : March 3, 1989                               */
/*****************************************************************/

#include <KLinda.h>
#include <stdio.h>

extern  int thread(char*,int, ...);
extern  Val environment();
extern  int rand();
extern  int timer();

#define WORKSPACE 6400
#define dictsize 32
#define LTS "lts"
#define ROWOFA "rowofA"
#define Xkey "x"
#define Ykey "y"
#define COUNT "count"
#define done -1

#define N1 200          /* number of rows of M and A */
#define N2 200          /* number of columns of A and rows of B */
#define N3 200          /* number of columns of M and B */

#define LDA N1       /* the leading dimension of A */
#define LDB N2       /* the leading dimension of B */
#define LDM N1       /* the leading dimension of M */

#ifdef SP
#define PREC int
#define ZERO 0
#define PRECISION "4-byte integer"
#endif


#ifdef DP
#define PREC double
#define ZERO 0.0e0
#define PRECISION "8-byte double"
#endif
```

```
PREC M[N1][N3];          /* matrix of N1 rows and N3 columns */
PREC A[N1][N2];          /* matrix of N1 rows and N2 columns */
PREC B[N2][N3];          /* matrix of N2 rows and N3 columns */

int nproc;           /* number of processes to be run in parallel */
 nt totaltime = 0;

main(int argc, char* argv[])
{
   void init_matrix(), mat_mul(), distribute(PREC [],PREC [],int);
   int i, j;


   PREC x[LDB]; /* vector used to form y = y + A * x  */
   PREC y[LDM]; /* vector used to accumulate the product of A * x */

   if (argc != 2)
     {
          fprintf(stderr,"usage: matmul nproc \n");
          exit(1);
      }

   if (sscanf(argv[1], "%d", &nproc) != 1)
     {
          fprintf(stderr, "failed to scan %d", argv[1]);
          exit(1);
      }

    init_matrix();      /* random generate matrix  */


   /* create a global dictionary */

   Val lts, env = environment();
   lts = createdict(dictsize);
   env.out(LTS,lts);

   for (j=0; j<N3; j++)
   {
      distribute(x,y,j);     /* pass a column of B to do parallel vector  */
    }                        /* multiplication with matrix A              */


   printf("XTM Parallel Matrix Multiplication by %d process\n",nproc);
   printf(" %d X %d  Matrix of %s \n",N1,N3,PRECISION);
   printf(" **** time = %f seconds ****", (float)totaltime/15625);


}



/* initialize matrix */
void init_matrix()
```

```
{
  PREC randgen();
  int i,j;

  for (i=0; i<N1; i++)
    {
          for (j=0; j<N2; j++)
             A[i][j] = randgen();
    }

  for (i=0; i<N2; i++)
    {
          for (j=0; j<N3; j++)
             B[i][j] = randgen();
    }
}


/*  random number generator */
PREC randgen()
 {
   PREC range = 256;

   return((PREC) rand()/range);
 }


/* pass a column of B and matrix A to perform parallel Matrix-Vector multiplication */
 void distribute(PREC x[LDB],PREC y[LDM],int pass)
 {
   int i;
   int startime, stoptime, benchtime;
   Val dummy;
   Val Arow,xblock,yblock;
   Val Receiveblock;
   PREC Mbuff[LDM];


    Val lts, env = environment();
    env.rd(LTS,lts);

    for (i=0; i<N1; i++)
        {
          M[i][pass] = ZERO;
          y[i] = M[i][pass];
        }
    for (i=0; i<N2; i++)
          x[i] = B[i][pass];


   /* send a row of A to each child process to do vector multiply */

   /* send first part of the combined key */
   for (i=0; i<N1; i++)
      lts.out(ROWOFA,i);

   /* send a row of A */
```

```
   for (i=0; i<N1; i++)
      {
       Arow = createblock(LDA*sizeof(PREC),(char*)A[i]);
       lts.out(i,Arow);
       }

   /* send x and y to child process */
    xblock = createblock(LDB*sizeof(PREC),(char*)x);
    yblock = createblock(LDM*sizeof(PREC),(char*)y);

    lts.out(Xkey,xblock);
    lts.out(Ykey,yblock);


    startime = timer();

   /* create multiple worker processes */
   for (i=0; i<nproc; i++)
      thread((char*)mat_mul,WORKSPACE);


   /* receive result which is a column of M */
   for (i=0; i<N1; i++)
     {
           lts.in(i+N1,Receiveblock);
           Receiveblock.gets(0,LDM*sizeof(PREC),(char*)Mbuff);
           M[i][pass] = Mbuff[i];
     }

    for (i=0;i<nproc;i++)
       lts.in(COUNT,dummy);

    stoptime = timer();
    benchtime = stoptime - startime;
    totaltime = totaltime + benchtime;

   /* remove x, y from the global dictionary after use */
   lts.undef(Xkey);
   lts.undef(Ykey);

 }


/* worker process which does vector multiplication */
void mat_mul()
{
 int i,j;
 Val row,tempblock;
 Val root,lts;
 PREC Abuff[LDA],Xbuff[LDB],Ybuff[LDM];
 Val xblock,yblock;
 Val SendBack;

/* get the path to access global dict */

 Val env = environment();
 env.rd(LTS,lts);
```

```
  for (i=0; i<N1/nproc; i++)
    {
      lts.in(ROWOFA,row);
      lts.in((int)row,tempblock);
      tempblock.gets(0,LDA*sizeof(PREC),(char*)Abuff);

      /* receive x, y from global dictionary */
      lts.rd(Xkey,xblock);
      xblock.gets(0,LDB*sizeof(PREC),(char*)Xbuff);
      lts.rd(Ykey,yblock);
      yblock.gets(0,LDM*sizeof(PREC),(char*)Ybuff);

      /* doing vector multiplication */
      for (j=0; j<N2; j++)
           Ybuff[(int)row] = Ybuff[(int)row] + Abuff[j] * Xbuff[j];


       /* send result back to parent */
       SendBack = createblock(LDM*sizeof(PREC),(char*)Ybuff);
       lts.out((int)row+N1,SendBack);

    }
   lts.out(COUNT,done);

}
```

```
/*********************************************************/
/*                                                       */
/*            DISK FILE I/O BENCHMARK PROGRAM            */
/*                                                       */
/*-------------------------------------------------------*/
/*   This benchmark program use low level I/O routines   */
/* with kernal Linda parallel programming primitives     */
/* to perform disk file I/O.  This version may not be    */
/* portable to other machines.  For those systems without */
/* the support of low level C I/O routines, please use   */
/* high level I/O routines (f** rooutines).              */
/*   This program need 4megabytes disk space.  Please run */
/* in single user mode to avoid others sharing system    */
/* resource.  The result is the combination performance  */
/* of I/O controller, I/O channel, disk itself and system */
/* bus.                                                  */
/*-------------------------------------------------------*/
/*   to compile this version for Cogent XTMOS :          */
/*        cc diskio.c -o diskio                          */
/*                                                       */
/*   NOTE:  make sure you have 4 megabytes disk space    */
/*          available before you run the program.        */
/*-------------------------------------------------------*/
/* written by : Sun Tieh-Jun                             */
/* Internet address : sun@mist.cs.orst.edu               */
/*                                                       */
/* last modified : March 3, 1989                         */
/*********************************************************/

#include <KLinda.h>
#include <stdio.h>
#include <syscall.h>
#include <fcntl.h>

extern Val environment();
extern  int timer();

#define dictsize 32
#define WORKSPACE 32000
#define LTS "lts"
#define ORDER "order"
#define COUNT "count"
#define done -1

#define maxbuffersize 32*1024   /* buffer size */
#define filesize 1024*1024*2    /* the file size 2megabyte */
#define transfersize 32*1024    /* number of bytes transfer at a time */


char *readfile = "/usr/sun/readfile";
char *writefile = "/usr/sun/writefile";
```

```
int nproc;

main(int argc, char* argv[])
{
  void setup(), readwrite(int);
  int i;
  int block;
  Val dummy;
  int  startime, stoptime, benchtime;


  if (argc != 2)
    {
        fprintf(stderr,"usage: I/O #proc\n");
        exit(1);
      }

  if (sscanf(argv[1], "%d", &nproc) != 1)
    {
        fprintf(stderr, "failed to scan %d", argv[1]);
        exit(1);
      }

  setup();

  block = filesize/transfersize;

  Val lts, env = environment();
  lts = createdict(dictsize);
  env.out(LTS,lts);

  for(i=0;i<nproc; i++)
    lts.out(ORDER,i);

  startime = timer();

  for (i=0; i<nproc; i++)
      thread((char*)readwrite,WORKSPACE,block);

  for (i=0;i<nproc;i++)
      lts.in(COUNT, dummy);

  stoptime = timer();
  benchtime = stoptime - startime;

  unlink(readfile);
  unlink(writefile);

  printf(" Two 2-Mbytes disk file read/write \n");
  printf("\n%d processes run in parallel. ",nproc);
  printf("\n\n\n\n Disk file I/O benchmark result \n");
  printf("*****************************************\n\n");
```

```
  printf("\n            time = %f seconds ", (float)benchtime/15625);

}


void
setup()
{
  int i;
  int fdr,fdw;
  unsigned int nbytes;
  char buffer[maxbuffersize];

  if ((fdr = creat(readfile,0640))<0)
    {
       printf("creat read file fail");
       exit(1);
     }

  close(fdr);

  if ((fdw = creat(writefile,0640))<0)
    {
       printf("creat write file fail");
       exit(1);
     }
  close(fdw);

  if((fdr = open(readfile,O_WRONLY))<0)
    {
      printf("can not open read file");
      exit(1);
    }


  lseek(fdr,0,0);
  for (i=0; i<filesize/transfersize; i++)
    {
      if ((nbytes = write(fdr, buffer, transfersize))<0)
       {
          printf("initial file error");
          exit(1);
        }
    }
  close(fdr);
}



void
readwrite(int blocks)
```

```
{
 int i;
 int fdr, fdw;
 Val id;
 char buf[maxbuffersize];
 unsigned int nbytes;


/* get path to access the global dictionary */

Val lts, env = environment();
env.rd(LTS,lts);


  if ((fdr=open(readfile,O_RDONLY))<0)
    {
      printf("can not open read file");
      exit(1);
    }
  if ((fdw=open(writefile,O_WRONLY))<0)
    {
      printf("can not open write file");
      exit(1);
    }

  lts.in(ORDER,id);

  for (i=(int)id; i<blocks; i+=nproc)
    {

      lseek(fdr,(i*transfersize),0);
      if ((nbytes = read(fdr,buf,transfersize))<0)
        {
          printf("read error");
          exit(1);
        }
      lseek(fdr,0,0);
      lseek(fdw,(i*transfersize),0);
      if ((nbytes = write(fdw,buf,transfersize))<0)
        {
          printf("write error");
          exit(1);
        }
      lseek(fdw,0,0);

    }

 lts.out(COUNT,done);
}
```

```
/*****************************************************************/
/*                                                               */
/*     Many-To-Many process message passing benchmark program    */
/*                                                               */
/*---------------------------------------------------------------*/
/*  This program performs message passing on an distributed      */
/* parallel computer system XTM.  Kernal Linda handles message   */
/* passing between two processors by using the global tuple      */
/* space.  Each process passes message to a global tuple space   */
/* and retrieve message also from that tuple space. The tuple    */
/* space is act as an shared memory space.                       */
/*---------------------------------------------------------------*/
/* compile : cc saturate.c -o sat                                */
/*     run : sat #process                                        */
/*---------------------------------------------------------------*/
/*  written by : Sun Tieh-Jun                                    */
/*  Internet address : sun@mist.cs.orst.edu                      */
/*  last modified : March 3, 1989                                */
/*****************************************************************/

#include <stdio.h>
#include <KLinda.h>

extern int thread(char*, int, ...);
extern Val environment();
extern int timer();

int rand();

 lefine PREC double
rdefine PRECISION "8-byte double"
#define arraysize 50 /* message size */
#define WORKSPACE 32000
#define dictsize 32
#define ITER 100
#define LTS "lts"
#define PID "processid"
#define BLOCK "block"
#define COUNT "count"
#define done -1


int nproc;  /* number of processes to be run in parallel */

main(int argc, char* argv[])
{
  void saturate();
  int i,loop;
  Val dummy;
  int startime, stoptime, benchtime,totaltime = 0;

  if (argc != 2)
```

```
    {
        fprintf(stderr,"usage: sat  nproc\n");
        exit(1);
      }

  if (sscanf(argv[1], "%d", &nproc) != 1)
    {
        fprintf(stderr, "failed to scan %d", argv[1]);
        exit(1);
      }

  /* create a global dictionry */

  Val lts, env = environment();
  lts = createdict(dictsize);
  env.out(LTS,lts);


for (loop=0;loop<ITER;loop++)
  {
    startime = timer();
    for (i=0; i<nproc; i++)
      thread((char*)saturate,WORKSPACE);

    for (i=0; i<nproc; i++)
      lts.in(COUNT,dummy);

    stoptime = timer();
    benchtime = stoptime - startime;
    totaltime +=  benchtime;
    printf("iter %d  done\n",loop);
  }


  printf("XTM %d to %d message passing result\n",nproc,nproc);
  printf("passing array of %d %s\n",arraysize,PRECISION);
  printf("Iteration = %d \n",ITER);
  printf("**************************************\n");
  printf("         time = %f seconds \n", (float)totaltime/15625);


}


/* random number generator */
PREC randgen()
{
  PREC range = 256;

  return((PREC) rand()/range);
}
```

```
/* processes saturating */
void saturate()

   PREC receive[arraysize];  /* local array for receiving */
   PREC send[arraysize];
   PREC randgen();
   Val Sendblock,Receiveblock,id;
   int i,j,procid;
   int k_getpid();


   procid = k_getpid();

   /* get path to access the global dictionary */
   Val lts, env = environment();
   env.rd(LTS,lts);

   for (i=0; i<arraysize; i++)
     send[i] = randgen();


    /* send array to global dictionary */
    for (i=0; i<nproc; i++)
     {
      lts.out(PID,procid);
      Sendblock = createblock(arraysize*sizeof(PREC),(char*)send);
      lts.out(procid, Sendblock);
     }


 /* receive array from global dictionary into local receive array */
    for (i=0; i<nproc; i++)
     {
       lts.in(PID,id);
       lts.in(id, Receiveblock);
       Receiveblock.gets(0,arraysize*sizeof(PREC),(char*)receive);
     }
   lts.out(COUNT,done);

}
```

```
/*************************************************************/
/*                                                           */
/*        PARALLEL ENUMERATION SORT BENCHMARK PROGRAM        */
/*                                                           */
/*-----------------------------------------------------------*/
   This program sorts a sequence of elements by multiple     */
/*  processes.  Each process get an element and compare it    */
/*  with all other elements to obtain a rank for it.  process */
/*  then send that element to its position according to its   */
/*  rank.  Multiple processes sort the sequence in parallel   */
/*  leads to a significant speedup.                          */
/*-----------------------------------------------------------*/
/*  written by : Sun Tieh-Jun                                */
/*  Internet address : sun@mist.cs.orst.edu                  */
/*                                                           */
/*  last modified : March 3,1989                             */
/*************************************************************/

#include <KLinda.h>
#include <stdio.h>

extern  int thread(char*,int, ...);
extern  Val environment();
extern  int rand();
extern  int timer();

#define WORKSPACE 32000
#define dictsize 1024
#define size 1024
#define LTS  "lts"
#define SKEY "skey"
#define COUNT "count"
#define done -1
#define SOURCE "source"

int nproc;


main(int argc, char* argv[])
{
void initial(), sort();
int startime, stoptime, benchtime;
int i;
Val result, dummy;
int target[size];

  if (argc != 2)
    {
          fprintf(stderr,"usage: sort #proc \n");
          exit(1);
     }


  if (sscanf(argv[1], "%d", &nproc) != 1)
    {
          fprintf(stderr, "failed to scan %d", argv[1]);
          exit(1);
```

```
    }

/* create a global dictionary */

  l lts, env = environment();
lts = createdict(dictsize);
env.out(LTS,lts);

initial();

startime = timer();

/* create multiple worker processes */
for (i=0; i<nproc; i++)
  thread((char*)sort,WORKSPACE);


/* receive the rank of each value and store those values */
/* to target according to its rank                       */

  for (i=0; i<size; i++)
    {
       lts.in(i+size, result);
       target[i] = (int) result;
    }

 /* syncronization point, all child processes are back */
  for (i=0;i<nproc;i++)
     lts.in(COUNT,dummy);

stoptime = timer();
  nchtime = stoptime - startime;


printf("XTM Parallel Enumeration sort");
printf("sort %d elements by %d processes :\n\n",size,nproc);
printf("**** RUN TIME ****\n");
printf(" time = %f seconds \n", (float)benchtime/15625);

}


void initial()
{
int i;
int src[size];
Val sourceblock;

/* get the path to access global dict */
 Val lts, env = environment();
 env.rd(LTS,lts);


/* generate random numbers and send them to dictionary */
/* printf("Unsort elements :\n");*/
for (i=0; i<size; i++)
```

```
    {
      src[i] = rand();

      lts.out(SKEY,i);
      lts.out(i,src[i]);
    }
  sourceblock = createblock(size*sizeof(int),(char*)src);
  lts.out(SOURCE,sourceblock);
  }


  void sort()
  {
  int i,j,k;
  Val index;
  Val temp;
  int local[size];
  Val localblock;

   /* get the path to access global dict */
   Val lts, env = environment();
   env.rd(LTS,lts);

  /* read in the original array to local */
  lts.rd(SOURCE, localblock);
  localblock.gets(0,size*sizeof(int),(char*)local);

  /* receive an element, compare it with all  */
  /* other elements and get a rank for it .   */
  for (i=0; i<size/nproc; i++)
    {
      lts.in(SKEY,index);
      lts.in(index,temp);
       k = -1;
      for (j=0; j<size; j++)
        {
             if ((int)temp > local[j])
               k++;
             else if ((int)temp == local[j]) && (index > j)
               k++;
        }
      /* send back the rank of this element */
      lts.out(k+size,temp);
    }
  lts.out(COUNT,done);
  }
```

```
/*************************************************************/
/*                                                           */
/*        MEMORY TRANSFER BENCHMARK PROGRAM                  */
/*                                                           */
/*-----------------------------------------------------------*/
/*  This program allocates local memory spaces of multiple   */
/*  T800 transputers and performs the data transfer among    */
/*  those transputer memory locations.                       */
/*-----------------------------------------------------------*/
/*  Compile : cc -o mtransfer mtransfer.c                    */
/*  run : mtransfer #proc                                    */
/*-----------------------------------------------------------*/
/* written by : Sun Tieh-Jun                                 */
/* Internet address : sun@mist.cs.orst.edu                   */
/*                                                           */
/* last modified : March 3, 1989                             */
/*************************************************************/

#include <KLinda.h>
#include <stdio.h>
#include <types.h>

#define WORKSPACE 32000
#define dictsize 1024
#define LTS "lts"
#define ITER 100
#define COUNT "count"
#define done -1

extern  int thread(char*,int, ...);
extern  Val environment();
extern  int timer();

typedef unsigned char BYTE;
typedef BYTE ptr;
typedef ptr *mptr;

int nproc;

main(int argc, char* argv[])
{
 int i,loop;
 int packet;
 Val dummy;
 int startime, stoptime, benchtime, totaltime=0;

 void sendproc(int), receiveproc(int);

 if (argc != 2)
     {
          fprintf(stderr,"usage: mtransfer #proc \n");
          exit(1);
```

```
    }

  if (sscanf(argv[1], "%d", &nproc) != 1)
    {
        fprintf(stderr, "failed to scan %d", argv[1]);
        exit(1);
    }

 /* create a global dictionary */

  Val lts, env = environment();
  lts = createdict(dictsize);
  env.out(LTS,lts);


 for(loop=0;loop<ITER;loop++)
   {
     for (packet=1; packet<=4096; packet*=2)
      {

       startime = timer();

       /* create multiple sender processes */
        for (i=0; i<nproc; i++)
          thread((char*)sendproc,WORKSPACE,packet);

       /* create multiple receiver processes */
        for (i=0; i<nproc; i++)
          thread((char*)receiveproc,WORKSPACE,packet);


        for (i=0;i<nproc*2;i++)
           lts.in(COUNT,dummy);

        stoptime = timer();
        benchtime = stoptime - startime;
        totaltime += benchtime;

      }
   }

  printf("XTM Memory transfer among %d processors\n",nproc*2);
  printf("** time = %f seconds **\n",(float)totaltime/15625);

}


void sendproc(int packet)
{
 Val Sendblock;
 BYTE *original;
```

```
 BYTE *malloc(int);
 int free(mptr);

 /* get the path to access global dict */

 Val lts, env = environment();
 env.rd(LTS,lts);

 /* allocate original MEMORY space */

  original = (mptr) malloc (sizeof(BYTE)*packet);

  Sendblock = createblock(sizeof(BYTE)*packet,(char*)original);
  lts.out(sizeof(BYTE)*packet,Sendblock);
  free(original);
  lts.out(COUNT,done);
}



void receiveproc(int packet)
{
 Val Receiveblock;
 BYTE *destination;
 BYTE *malloc(int);
 int free(mptr);

 /* get the path to access global dict */

 Val lts, env = environment();
 env.rd(LTS,lts);

 /* allocate destination space */
 destination = (mptr) malloc(sizeof(BYTE)*packet);
 lts.in(sizeof(BYTE)*packet,Receiveblock);
 Receiveblock.gets(0,sizeof(BYTE)*packet,(char*)destination);

 free(destination);
 lts.out(COUNT,done);
}
```

```
/************************************************************/
/*        MATH LIBRARY FUNCTION BENCHMARK PROGRAM           */
/*----------------------------------------------------------*/
/*  This benchmark program executes system support math     */
/*  functions.  There are three categories of functions are */
/*  tested :                                                */
/*           (1) trigonometric functions:                   */
/*                SIN(), COS(), TAN(), ASIN(), ACOS(), ATAN() */
/*                                                          */
/*           (2) logarithm and Square root functions :      */
/*                EXP(), LOG(), LOG10(), POW(), SQRT()      */
/*                                                          */
/*           (3) absolute and approximate value functions : */
/*                FABS(), FLOOR(), CEIL()                   */
/*                                                          */
/*  All functions are executed 10000 iterations.            */
/*----------------------------------------------------------*/
/*  written by : Sun Tieh-Jun                               */
/*  Internet address: sun@mist.cs.orst                      */
/*                                                          */
/*  last modified : March 3, 1989                           */
/************************************************************/

#include  <stdio.h>
#include  <math.h>

#define ITER 10000          /* number of iteration to be executed */


#define const1 0.50         /* these constants are used as initial  */
#define const2 180.00       /* value for math functions.  This will */
#define const3 100.00       /* keep allfunctions return the valid   */
#define const4 25           /* value and avoid overflow             */
#define const5 -1.4999

extern int timer();

main()
{
void trig(), log_sqrt(), absolute();
int  startime, stoptime, benchtime;
startime = timer();
trig();
log_sqrt();
absolute();
stoptime = timer();
benchtime = stoptime - startime;

printf("XTMOS system built-in Mathematical function benchmark. \n");
printf("\ntime = %f seconds ", (float)benchtime/15625);
}


void trig()
{
  double x = const1;
  int i;
```

```
  for (i=0; i<ITER; i++)                /* each iteration almost transform  */
    {                                   /* x back to its initial value.     */
          x = asin(sin(x)*cos(x));      /* The variation is very slow       */
          x = acos(sin(x)*cos(x));
          x = atan(tan(x));
          x /= const2;
        }

}


void log_sqrt()
{
double x = const3;
int i;

for (i=0; i<ITER; i++)                  /* each iteration exactly transform  */
  {                                     /* x back to its initial value.      */
    x = sqrt(exp(log(log10(x))/const1));
        x = pow(x,x);
        x *= const4;
  }
}



void absolute()
{
 double x = const5;
 int i;

  for (i=0; i<ITER; i++)                /* each iteration exactly transform  */
    {                                   /* x back to its initial value.      */
          x = ceil(floor(x)/fabs(x));
          x *=fabs(const5);
        }

}
```