Experimental Study on Time and Space Sharing on the PowerXplorer

Sulieman Bani-Ahmad
CWRU, Ohio, USA.
sulieman@case.edu

Ismail Ababneh
AABU, Mafraq, Jordan
Ismail@aabu.edu.jo

ABSTRACT

Scheduling algorithms in parallel computers fall into two basic categories: time and space sharing algorithms. Space-sharing based processor allocation algorithms can be contiguous or non-contiguous. Studies show that non-contiguous allocation is superior due to decrease in fragmentation. Other studies have reported that executing jobs on fewer processors (folding) can improve the performance of contiguous and non-contiguous allocation. However, the problem with folding is that it is not always applicable because of parallel programming languages and parallel operating systems limitations.

Most of previous studies used simulation. Our study is an experimental one for studying time and space sharing on a real parallel machine (the PowerXplorer), with eight processors arranged as a two-dimensional mesh. A set of five scientific applications with differing communication characteristics were implemented and executed using time and space sharing. The observed execution times were used to study and compare time-sharing and contiguous and non-contiguous space sharing *with* and *without* folding. Our study showed that time-sharing gave comparable results to space sharing allocation. Further, non-contiguous allocation gave better results than contiguous allocation when folding is not supported. However, when folding is supported contiguous allocation gave the best mean turnaround times.

Keywords: Scheduling algorithms, parallel programmin, parallel computing.

1 INTRODUCTION

A job in multiprogrammed parallel systems is characterized along two dimensions; the *length*, measured by the execution time and the *width* or *size* measured by the number of threads such that each thread is executed on a separate processor. Thus, resource sharing in parallel computers takes two levels, (i) *time sharing*, whereby a thread can be interrupted during execution by other threads of jobs running on the same processor, (ii) space sharing, where a thread has exclusive use of its processor until its execution is complete [11]. Space sharing can be contiguous or non-contiguous depending on whether the processors allocated to a single job are physically adjacent or not [13].

Comparison studies show that non-contiguous allocation is superior to contiguous allocation as the former produces less *external fragmentation* [13]. However, contiguous allocation is preferred as it provides maximum communication speed between threads of the same job [13].

Folding, that is executing jobs on fewer processors, can improve performance of contiguous and non-contiguous allocation [12]. The problem with folding is that it is not always applicable due to limitations of parallel programming languages and parallel operating systems [12].

Most previous studies used simulation to study and compare different scheduling processor allocation strategies. This study, however, is an experimental one for studying time and space sharing on a real parallel machine (the *PowerXplorer*) with eight processors arranged in a two dimensional mesh and uses *PARIX* operating system and parallel programming language. This study passed by the following stages:

(i) We selected a set of five scientific applications to implement and later use to comparatively assist different allocation strategies. Two of these applications, namely *Matrix multiplication* (MM) and *LU factorization*, are used in well-known parallel benchmarks like NAS [3] and GENESIS [2]. The other applications are (a) *2D Fast Fourier Transform* (2D FFT), which is used in image processing, (b) *Floyd shortest path algorithm* from graph theory, and (c) a *simulation of electromagnetic wave propagation in 2D space using Finite-difference in time domain* (FDTD) from Electromagnetics.

(ii) We implemented the selected applications in *PARIX* environment. The applications were run on the PowerXplorer using time sharing (TS) and both contiguous and non-contiguous space sharing (CSS, NCSS).

(iii) The execution times are used as inputs to a simulator to study and comparatively evaluate contiguous and non-contiguous space sharing (a) with and without folding, and (b) with bounded

folding, which limits *folding factor*. The folding factor for a particular job j that requests n_j is defined as the ratio between the number actually allocated to j and n_j.

The main contribution of this paper is that it comparatively assists time sharing and different processor strategies on a real parallel computer, namely the PowerXplorer. Our major findings are:

- TS gave close average execution times to CSS and sometimes performed better with large allocation sizes, i.e. when relatively large allocation sizes. Further, literature shows that TS (i) reduces the job's wait time (before allocation), (ii) dampens the effect of *idle state* of processors that occurs when a thread halts waiting for communication or I/O operations. Therefore, TS can be useful to *interactive parallel computer systems* that require low response times to user commands.
- Simulation shows that non-contiguous allocation acheives better results than contiguous allocation when folding is not supported. However, when folding is supported, contiguous allocation gives the best mean turnaround times.

The remainder of this paper is organized as follows: Section 2 presents the PowerXplorer and PARIX operating system and programming language. The studied processor allocation strategies are briefly described in section 3. Section 4 presents the implementation details of the five scientific applications used in the experimental part of this study. The simulation details are described in section 5. In section 6, we present our main experimental results and observations. Section 7 provides our conclusions.

2 The PowerXplorer and PARIX

The PowerXPlorer is a family of distributed memory systems from Parsytec (http://www.parsytec.com/). This system runs the PARIX operating system and is based on 8 processing units arranged in a 2D mesh, as illustrated in figure 1. Each processing unit has (i) one 80-MHz PowerPC 601 processor, (ii) 8 MB of local memory and (iii) a *transputer* for establishing and maintaining communication links

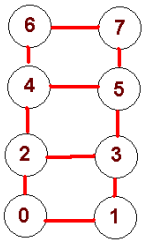


Figure 1: The 2D mesh of the PowerXplorer with processor IDs.

PARIX (PARallel UnIX extensions) [7] is the native operating system in the Parsytec PowerXplorer family. It provides UNIX functionality at the front-end with library extensions for the needs of the parallel system. The Parix software package comprises components for the program development environment (compilers, tools, etc.) and runtime environment (libraries). PARIX offers different types of synchronous and asynchronous communication [7].

3. Studied Scheduling Strategies

Besides *time sharing*, we study the performance of the following processors allocation strategies of space sharing.

(i) ***Space sharing without folding***: We test two strategies under this category, namely, *Contiguous Space Sharing* (CSS) and *Non-Contiguous Space Sharing* (NCSS). In general, a CSS strategy starts by allocating the exact requested number of processors. If it fails, the requesting job waits until enough idle processors become available. In NCSS however, if allocating a contiguous set fails, the algorithm looks for a non-contiguous set. If it fails again, the requesting job waits until the needed number of processors becomes available.

(ii) ***Space sharing with folding***: Folding is allocating a fewer number of processors than requested in case not enough idle processors are found. We test four strategies under this category:

(a) C*ontiguous space sharing with unbounded folding* (CSSUF), which puts no limits on the folding factor. We define the folding factor as the ratio between the requested number of processors and the actual nember allocated after folding. This strategy puts no limits on the folding factor, i.e. it may fold any job requesting n_i processors to one processor with folding factor $1/n_i$.

(b) *Non-contiguous space sharing with unbounded folding*, which differs from the previous one in that it relaxes the contiguity condition and tries to allocate a non-contiguous set of processors if a contiguous set is not available before folding.

(c) *Contiguous/non-contiguous space sharing with bounded folding* (CSSBF, NCSSBF), which are similar to the previous two except in that they allow maximum folding factors of k/n_i where n_i is the requested number of processors and k is the minimum number of processors that can be allocated to any job. We chose k=2 as we have relatively small mesh (8 processors). For example, we allowed folding factors of 1/4, 1/3 and ½ and 1 on the requests for 8, 6, 4 and 2 processors respectively.

4. Implementation Details of the Selected Applications

To facilitate parallelizing the five scientific applications, we chose the size of the input to be dividable by 1, 2, 4, 6 and 8. Further, at run time, one of the instantiated threads of the application, arbitrarily selected, takes divides the data input

between all other threads. The same thread is responsible for gathering the final results. We refer to this thread as the *main thread*.

Figure 3 shows the main loops of *Matrix Multiplication*. M_a and M_b are the input matrices to be multiplied and M_c is the result matrix. The matrices are of size 384*384 each.

The algorithm is parallelized by dividing the rows of M_a and the columns of M_b into *n* slices each, where *n* is the number of processors assigned to the application (1, 2, 4, 6 or 8). A total of *n* threads are instantiated at each processor. The main thread distributes the slices such that each thread initially receives a slice of M_a and another of M_b (*One-to-All communication*). The threads, in turn, start to compute the M_c slice assigned to them. During computation, *All-to-All communication* occurs between the threads as follows; thread of ID *j* (i) passes the M_b slice that it has to its neighbor with ID *j+1* and (ii) receives its *j-1*'s M_b slice and (iii) computes the corresponding portion of the M_c slice. The three steps mentioned above, (i), (ii) and (iii) are repeated *n-1* times. Finally, the *main thread* collects back M_c slices (*All-to-One communication*).

As it is quicker than asynchronous communication, we use *Synchronous Communication* in all applications, i.e. the two communicating threads should be at both sides of the communication channel simultaneously. To prevent deadlock due to communication, the threads with odd IDs start sending first and simultaneously the threads with even IDs receive. Next, the roles are reversed.

Figure 2 shows the main loops of Floyd's shortest path algorithm. $|V|$ is the number of vertices in the input graph *G(V,E)*, where *V and E* are the vertex and edge lists. We chose *|V|=384*. *M* initially is the edge-weight matrix of *G*. After Floyd's algorithm is applied on *G*, *M* becomes the shortest matrix.

At each iteration of loop *k* (figure 2), the distances between the vertices scanned by loops *i* and *j*, *M[i][j]*, are updated as follows: if the distance between vertices *i* and *j* passing by *k* is shorter than the current distance between *i* and *j*, then the entry *M[i][j]* is updated to the new shorter distance.

```
for(k=0;k<|V|;k++)
 for(i=0;i<|V|;i++)
  for(j=0;j<|V|;j++)
   {if(M[i][j] > (M[i][k]+M[k][j]))
      M[i][j] = M[i][k]+M[k][j];}
```

Figure 2: The main loops of Floyd's algorithm

```
for(i=0;i<Md;i++)
 for(j=0;j<Md;j++)
  for(k=0;k<Md;k++)
    Mc[i][j]+=Ma[i][k]*Mb[k][j];
```

Figure 3: The main loops of Matrix Multiplication procedure

Floyd's algorithm is parallelized by instantiating *n* threads. The rows of *M* are divided into *n* slices. The *main thread* passes each slice to its corresponding thread (*One-to-All communication*). During shortest-path computation, the thread that has the k^{th} row (figure 2) passes it to all other threads (*All-to-All communication*). The main thread gathers the updated slices of *M* (*All-to-One communication*).

Figure 4 shows the main loops of LU factorization application based on *Gaussian elimination*. Where *dim* is the size of the matrix (M_{lu}) to be factorized. After the elimination is performed, M_{lu}, will have both the *L* matrix (the values below the diagonal) and the *U* matrix (the values above the diagonal).

The application is parallelized by dividing M_{lu}'s columns into *n* slices, where *n* is the number of processors. The main thread distributes the slices and at the end collects it back (*One-to-All* and *All-to-One* communication). Depending on *k* (figure 4), the thread that determines the elimination factor sends the factor to all threads of interest to perform elimination (the threads that have any row below the k^{th} row).

```
for (k=0;k<dim-1;k++)
{
  for (i=k+1;i<dim;i++)
  {
     factor=Mlu[i][k]/Mlu[k][k];
     Mlu[i][k]=factor;
     for (j=k+1;j<dim;j++)
     {Mlu[i][j]=Mlu[i][j]-factor*Mlu[k][j];}
   }
}
```

Figure 4: The main loops of LU factorization code.

The 2D FFT is similar to the well-known 1D FFT accept that the former is performed on matrices instead of vectors. The 2D FFT is performed on a matrix *M* by implementing 1D FFT on the rows of *M* to produce *M'*, and then on the columns of *M'*. The dimension of the vectors should be of the power of two. Thus we choose *M* to be of size 1024*1024 in our experiments.

Parallelizing 2D FFT is conducted as follows; the rows of *M* are divided into *n* slices, where *n* is the number of processors allocated for the 2D FFT job. Each slice is of size *1024/n*1024*. The main thread distributes the slices between all other threads. Having done that, the two stages of 2D FFT are performed as follows:

(i) Each thread performs 1D FFT on the rows of the slice it has. Having done that, each thread further slices the columns of the result *1024/n*1024* matrix into *n* slices. The new slices are exchanged between the threads so that each thread receives a column slice of the complete transformed matrix of this stage, i.e. *M'*.

(ii) Again, 1D FFT is performed on the column-slices. At the end, the main thread collects back the column slices of the final transformed matrix.

Electromagnetic wave propagation simulation in space is done by iteratively computing two

components, namely, the *electric filed* and the *magnetic filed* components according to the four equations shown in figure 5. We chose to simulate electromagnetic wave propagation in 2D space as described in [5].

$$\frac{\partial D_z}{\partial t} = \frac{1}{\sqrt{\varepsilon_o \mu_o}} \left(\frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y} \right), \quad D_z(\omega) = \varepsilon^*(\omega).E_z(\omega)$$

$$\frac{\partial H_x}{\partial t} = -\frac{1}{\sqrt{\varepsilon_o \mu_o}} \frac{\partial E_z}{\partial y}$$

$$\frac{\partial H_y}{\partial t} = \frac{1}{\sqrt{\varepsilon_o \mu_o}} \frac{\partial E_z}{\partial x}$$

Figure 5: The four components of the electromagnetic field. μ_o is the free space permeability of the magnetic field. ε_o is the free space permittivity of the electrical field, H and E are the magnetic and electric fields respectively. And D is the electric flux density.

The simulation is conducted as follows: We initialize the four matrices D_z, E_z, H_x and H_y to zeros. Each entry in the matrix D_z represents the electric flux density at the corresponding point in space. Similarly, each entry of the matrices E_z, H_x and H_y represents the electric and magnetic fields components at the corresponding point in the space. A *wave pulse source* is placed in the space. After that, we iteratively compute the values of the different electromagnetic field components. Each iteration represents a moment of time elapsed. For more details, refer to [5].

The electromagnetic wave propagation is parallelized as follows: The 2D space to be studied is divided into *n* smaller subspaces of equal sizes. Each thread of the instantiated *n* threads is responsible of updating the four components, namely the *electric field* and the *electric flux density*, and the two magnetic field components. The threads that are responsible of contiguous small spaces exchange the values of the four components located at the boundaries.

5. Simulation Details

After we implemented the five applications described in section 4 above, we executed them on various numbers of processors considering time and space sharing. After that, we used the observed execution times to further study and experimentally evaluate different processor allocation algorithms with and without folding on the PowerXplorer.

To quantify and simulate the effect of non-contiguity on the execution time of applications, we experimentally compute the *non-contiguous to contiguous time ratio, nc2ctr* using the formula

$$nc2ctr = \max_{\substack{\forall n \in \{1,2,4,6,8\} \\ \forall A \in \{"MM","Floyd","LU","2DFFT","FDTD"\}}} (T_{nc}(A,n)/T_c(A,n))$$

where $T_{nc}(A,n) / T_c(A,n)$ are the execution times of application A on n non-contiguous/contiguous processors respectively. We experimentally found that $nc2ctr = 1.1$.

Jobs are served in *First In First Out* (FIFO) order. The contiguous allocation strategies used is the *First Fit* as described in [8]. Job sizes are chosen uniformly random from the set {1, 2, 4, 6, 8}. We also considered another distribution that favors small-size jobs. However, the observations of both were the same, thus we only report the case of *uniformly distributed* job sizes.

The outputs of the simulators are:

(i) The ***Average System Utilization*** (ASU), which is defined as $ASU = (\sum_0^T SU(t))/T$ where T is the simulation time. $SU(t)$ is the system utilization at time *t.* System utilization at any moment is computed as the number of the busy processors divided by the total number of processors in the system.

(ii) The ***Average Turnaround Time*** (ATT), which is defined as $ATT = (\sum_{j=1}^{N} TT(j))/N$ where $TT(j)$ is the turnaround time of job *j* and *N* is the total number of served jobs. $TT(j)$ is computed as $TT(j) = WT(j) + ET(j)$ where $WT(j)$ and $ET(j)$ are the *waiting* and *execution* times of job *j*, respectively.

The above parameters are measured with a *confidence interval* of 0.95 and a *maximum error level* of 0.05.

6. Experimental Results and Observations

Table 1 shows the execution times of the selected applications on 1, 2, 4, 6 and 8 processors.

	Matrix			Floyd			FDTD simulation		
n	**T**	**S**	**e**	**T**	**S**	**e**	**T**	**S**	**e**
1	49.4	1.00	1.00	58.3	1.00	1.00	106.	1.00	1.00
2	29.8	1.66	0.83	32.6	1.79	0.89	62.0	1.71	0.86
4	16.6	2.98	0.74	20.0	2.92	0.73	30.2	3.51	0.88
6	12.5	3.95	0.66	18.0	3.24	0.54	20.8	5.96	0.85
8	7.7	6.42	0.80	18.7	3.12	0.39	16.2	6.54	0.82

	LU factorization			2D FFT		
n	**T**	**S**	**e**	**T**	**S**	**e**
1	64.0	1.00	1.00	15.	1.00	1.00
2	47.6	1.34	0.67	8.2	1.83	0.92
4	28.4	2.25	0.56	7.9	1.90	0.48
6	19.4	3.30	0.55	-	-	- *
8	23.3	2.75	0.34	7.1	2.11	0.26

* Power of 2 processors is required for the FFT parallel algorithm.
T: execution time
S: Speed up
e: Efficiency

Table 1: Execution times, speed-up values and efficiency value of the selected applications. *Problem Sizes:* (**MM**) 384x385. (**Floyd**): 480 points (**FDTD**): 384x384. (**LU**): 768*768 (**2D FFT**): 1024x1024

Observation: (Table 1) Speed-up and efficiency values in MM and FDTD were high, however, they were low in Floyd, LU factorization and 2D FFT. The above observation results from the differences in the time spent by different applications performing communication compared to the time they spend on computation. The Speed-up is low in

the cases where *communication to computation ratio* is high [15], as in the case of Floyd's algorithm. The best speedup we obtained where in MM and FDTD, where the communication compared to computation times were low compared to the time spent in computation.

To support the above illustration, we compute the communication times in the case of MM and Floyd. We use the point-to-point communication model described in [14]. According to this model, the time needed to send a message of size *s* from one point to another point ($t(s)$ in seconds) is given by $t(s) = t_o + s / r_\infty$, where t_o is the *startup* or *latency* time. r_∞ is the *asymptotic bandwidth* defined as the maximum achievable bandwidth when message length approaches infinity [14].

The latency time of point-to-point communication in PARIX on the PowerXplorer is given by $t_o = 41 + 42 * p$ (in *micro seconds*) [14], where *p* is the number of processors in the path from the *source* to the *destination* (including the source and the destination points) [14]. The parameter r_∞ is also estimated by 1.05 MB/Sec [14]. Using this model and given the data input sizes of the implemented applications, one can estimate the communication times as in the next example.

Example: In the case of executing the MM application on 4 *contiguous* processors as illustrated in figure 6, and given that the input matrix is defined as an array of floats (each float variable occupies 4 bytes in memory), we can compute the All-to-All communication time as follows:

- a total of 4*(4-1) messages are sent.

- each message is of size *s*=384/4*384 (*float* /message)* 4 (bytes/*float*) / 2^{10} MBytes.

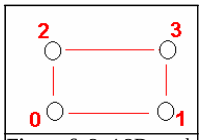


Figure 6: 2x4 2D mesh

Thus, the communication time of the 12 messages (without considering the communication latencies) is $12 * s / r_\infty = 12 * 0.09375 = 1.125$ seconds.

- The latency times vary depending on the *length of the path*. 4 of the 12 messages have a path length of 3. These four communication operations occur between MM threads running on processors 0 and 3 (two operations) and between the threads running on 1 and 2 (another two operations). The other 8 operations are each of length 2. Thus, the total latency time needed to send 12 messages is

$$t_o = 4(41 + 42*3) + 8(41 + 42*2) = 1167 \mu s .$$

- Finally, the total All-to-All communication time is 1.126 seconds. This forms 6.8% of the total execution time of MM on four contiguous processors.

Repeating same calculations for the 2D FFT on four processors, we notice that the communication time is around 3.001 seconds which forms around 40% of the total execution time which is 7.9 seconds. This explains the reason why efficiency may decrease as the number of processors increases.

6.1 Time Sharing and Serial Execution Runtimes

Table 2 shows the observed execution times on 8 processors of all the applications executed *serially* and using *time sharing*. For instance, to implement time sharing on the eight processors between MM and FDTD, we instantiated two threads on each processor, a thread for MM and another for FDTD.

Applications		TS		SE		
A1	**A2**	**A1**	**A2**	**A1**	**A2**	**Sum**
MM	**MM**	12.6	12.9	7.7	7.7	15.4
MM	**FLOYD**	9.3	25.3	7.7	18.7	26.4
MM	**LU**	10.3	28.4	7.7	23.3	31.0
MM	**FFT**	9.3	8.6	7.7	7.1	14.8
MM	**FDTD**	10.8	21.1	7.7	16.2	23.9
FLOYD	**FLOYD**	30.2	30.6	18.7	18.7	37.4
FLOYD	**LU**	31.0	32.7	18.7	23.3	42.0
FLOYD	**FFT**	21.7	7.6	18.7	7.1	25.8
FLOYD	**FDTD**	33.6	18.0	18.7	16.2	34.9
LU	**LU**	37.9	38.2	23.3	23.3	46.6
LU	**FFT**	25.2	7.3	23.3	7.1	30.4
LU	**FDTD**	37.0	9.5	23.3	16.2	39.5
FFT	**FFT**	9.2	9.9	7.1	7.1	14.2
FFT	**FDTD**	8.7	19.0	7.1	16.2	23.3
FDTD	**FDTD**	31.2	31.7	16.2	16.2	32.4

Table 2: The execution times in time sharing and serial execution strategies on eight processors.

Observation: (Table 2) Floyd takes 18.7 seconds when singly executed on eight processors. However, when executed using time sharing with MM and LU (that require 7.7 and 23.3 seconds respectively), Floyd's execution time raises to 25.3 and 31.0 seconds, respectively.

The illustration of the above observation is that applications with relatively short execution time relinquish its processor(s) early. Therefore, they have less effect on the execution time of the other applications running simultaneously on the same processor(s).

Observation: (Table 2) Finish time with time sharing is in average 15.3% less than serial execution.

For instance, the serial execution time of MM and FDTD was 23.9 seconds. With time sharing however the MM application finished after 10.8 seconds and the other after 21.1 seconds. 2.8 seconds where saved by time-sharing. Time-sharing threads uses the shared processor alternatively s.t. when a thread halts waiting for communication the other ready threads may resume execution, thus, less processor clock cycles are wasted.

We also observed that, as the number of processors allocated for the applications being executed in time-sharing manner decreases, the time difference between serial and the time-sharing execution decreases. The reason is that the time spent in communication increases as the number of processors used increases. Thus, the processor clock cycles while waiting for the communication process to occur/finish increase, which in turn provides an opportunity for more cycles to be utilized.

6.2 Contiguous and Non-contiguous Space Sharing

Table 3 shows the execution times of two applications sharing the space of eight processors (Contiguous allocation). Two configurations are presented: (i) the 4, 4 configuration (Case **A**), where each application is assigned 4 processors, and (ii) the 2, 6 configuration (Case **B**), where one application is assigned 2 processors and the other is assigned 6.

Applications		CSS Case A		CSS Case B	
A1	**A2**	**A1: 4**	**A2: 4**	**A1: 2**	**A2: 6**
MM	**MM**	16.6	16.6	29.8	12.5
MM	**FLOYD**	16.6	20.0	29.8	18.0
MM	**LU**	16.6	28.4	29.8	19.4
MM	**FFT**	16.6	7.9	29.8	-
MM	**FDTD**	16.6	30.2	29.8	20.8
FLOYD	**FLOYD**	20.0	20.0	32.6	18.0
FLOYD	**LU**	20.0	28.4	32.6	19.4
FLOYD	**FFT**	20.0	7.9	32.6	-
FLOYD	**FDTD**	20.0	30.2	32.6	20.8
LU	**LU**	28.4	28.4	47.6	19.4
LU	**FFT**	28.4	7.9	47.6	-
LU	**FDTD**	28.4	30.2	47.6	20.8
FFT	**FFT**	7.9	7.9	8.2	-
FFT	**FDTD**	7.9	30.2	8.2	20.8
FDTD	**FDTD**	30.2	30.2	62.0	20.8

Table 3: Execution times using contiguous space sharing on eight processors.

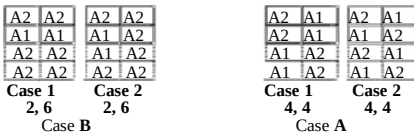


Figure 7: Four scenarios of allocating two jobs (A1, A2) on the PowerXplorer using (4, 4) and (6, 2)

Table 4 shows the same experiment conducted in table 3; this time the processors of each application are non-contiguous. Four configurations are tested (as illustrated in figure 7); two cases for each configuration, namely the (4,4) and (2, 6) configurations.

Observation: (Tables 3 and 4) Noncontiguous allocation of an application increases the execution time over contiguous allocation by a factor of 1.07 in average and 1.1 in maximum.

Non-contiguity creates message-passing contention when threads of two different applications compete over common communication channels. This results in more delay due to communication and thus increases the execution times of the applications.

Example: Consider the case shown in figure 8: the black and white circles represent processors allocated two different applications, A1 and A2. A1's threads are running on processors 0 and 3, and A2's threads are running on 1 and 2.

Assume the following scenario: A1's thread at processor 0 is sending messages to the other thread on processor 3 over the path (0 – 1 – 3). Simultaneously, A2's thread on processor 1 is sending a message to the other thread on 2 over the path (1 – 0 – 2).

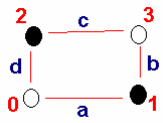


Figure 8: Message-passing contention

In this case, the two applications will compete over the link (0 -1) which is channel **a.** This results in

Applications		Case 1 (4,4)		Case 2 (4,4)		Case 1 (2,6)		Case 2 (2,6)	
A1	**A2**	**A1: 4p**	**A2: 4p**	**A1: 4p**	**A2: 4p**	**A1: 2p**	**A2: 6p**	**A1: 2p**	**A2: 6p**
MM	**MM**	17.1	17.1	17.0	17.0	29.8	12.6	29.8	12.6
MM	**FLOYD**	17.0	21.4	16.8	21.2	30.1	19.0	30.0	18.7
MM	**LU**	17.0	29.3	16.9	29.2	30.5	20.8	30.0	19.9
MM	**FFT**	17.7	8.1	16.7	8.1	-	-	-	-
MM	**FDTD**	16.6	30.3	16.6	30.2	29.8	20.8	29.8	20.8
FLOYD	**FLOYD**	21.6	21.6	21.7	21.7	33.0	19.0	33.1	18.6
FLOYD	**LU**	21.4	29.4	21.4	29.3	33.4	20.6	33.2	19.9
FLOYD	**FFT**	21.1	8.2	21.0	8.1	-	-	-	-
FLOYD	**FDTD**	21.2	30.5	21.2	30.4	32.9	20.9	33.0	20.9
LU	**LU**	30.3	30.3	30.3	30.3	49.0	20.7	48.7	19.8
LU	**FFT**	29.2	8.2	29.1	8.1	-	-	-	-
LU	**FDTD**	30.9	29.4	30.6	29.2	48.3	20.9	48.5	21.0
FFT	**FFT**	8.7	8.7	8.6	8.6	-	-	-	-
FFT	**FDTD**	8.0	30.3	7.9	30.2	8.2	20.8	8.2	20.9
FDTD	**FDTD**	30.7	30.7	30.7	30.7	62.2	20.8	62.2	20.8

Table 4: Execution times using non-contiguous space sharing on eight processors

.

Applications		2 processors		4		6		8		
A1	**A2**	TS	CSS	TS	CSS	TS	CSS	TS	CSS(4, 4,)	CSS(2,6)
MM	**MM**	58.10	49.40	30.90	29.80	22.35	23.20	12.75	16.60	21.15
MM	**FLOYD**	56.05	53.85	28.00	31.20	21.60	24.90	17.30	18.30	23.90
MM	**LU**	64.35	56.70	32.50	38.70	22.75	29.10	19.35	22.50	24.60
MM	**FFT**	22.30	32.20	14.30	19.00	-	18.85	8.95	12.25	-
MM	**FDTD**	73.35	77.70	35.90	45.90	25.20	30.00	15.95	23.40	25.30
FLOYD	**FLOYD**	63.60	58.30	36.25	32.60	29.65	26.30	30.40	20.00	25.30
FLOYD	**LU**	71.80	61.15	44.35	40.10	31.65	30.50	31.85	24.20	26.00
FLOYD	**FFT**	24.40	36.65	16.60	20.40	-	20.25	14.65	13.95	-
FLOYD	**FDTD**	84.10	82.15	43.50	47.30	30.45	31.40	25.80	25.10	26.70
LU	**LU**	94.90	64.00	54.90	47.60	35.15	38.00	38.05	28.40	33.50
LU	**FFT**	32.10	39.50	21.10	27.90	-	27.75	16.25	18.15	-
LU	**FDTD**	106.75	85.00	49.55	54.80	31.05	38.90	23.25	29.30	34.20
FFT	**FFT**	11.90	15.00	10.90	8.20	-	8.05	9.55	7.90	-
FFT	**FDTD**	39.70	60.50	21.95	35.10	-	19.20	13.85	19.05	14.50
FDTD	**FDTD**	123.80	106.00	59.20	62.00	40.55	46.10	31.45	30.20	41.40

Table 5: Average execution time with time-sharing and contiguous and non-contiguous space sharing

communication delay, and thus increases the overall execution time of both applications.

We denote to the maximum observed increase in execution time due to non-contiguous allocation over contiguous by ***nc2ctr***. We used this value in the simulation part (see section 5 for details).

6.3 Comparing Space and Time-Sharing Strategies

Table 5 demonstrates the average execution times on 2, 4, 6, and 8 processors using time-sharing (TS) and contiguous space sharing (CSS).

Observation: (Table 5) The ratio between execution time of time-sharing and space-sharing applications ranges between 0.95 and 1.3.

This shows that time-sharing performance is comparable to contiguous space sharing. For instance, the execution time of LU and Floyd on four processors using TS and CSS were 44.35 and

40.1 seconds, respectively. The difference was 4.25 seconds.

Observation: (Table 5) The difference between TS and CSS decreases as the number of processors allocated to the sharing applications increases.

TS may give shorter execution time than CSS in some cases (i) depending on the communication timing and behavior of the applications and (ii) due to low efficiency values at relatively large allocation sizes. For instance, in the case of MM and LU factorization executed on eight processors, the execution time was 19.35 seconds with TS and 22.5 and 24.6 with CSS on 2-6 and 4-4 distributions respectively.

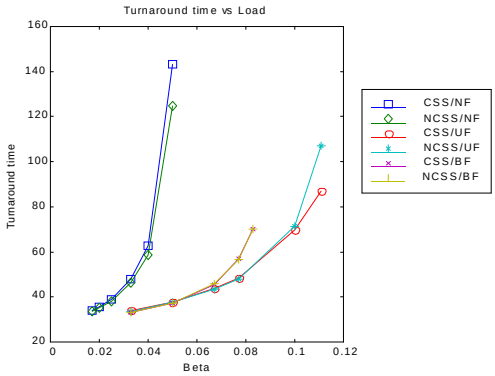


Figure 10: System load (Beta) vs. turnaround time of all allocation strategies

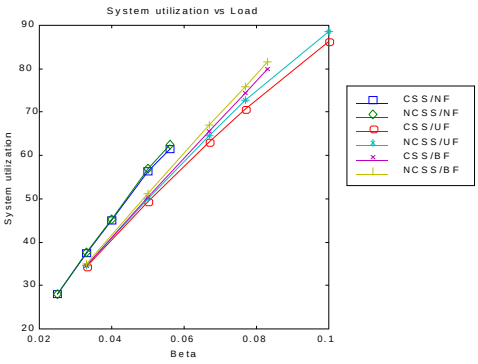
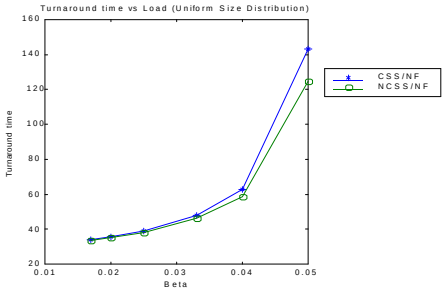
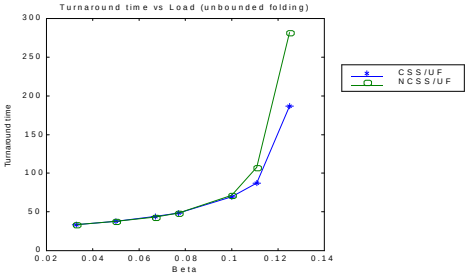


Figure 11: System load (Beta) vs. System utilization of all allocation strategies



(a)



(b)

Figure 9: Average turnaround time vs. system load for CSS and NCSS (a) without folding and (b) with unbounded folding

6.4 Simulation Results and Observations

Figure 9 shows the average turnaround time vs. system load (Beta, which is *the mean arrival rate* in *Poisson process*).

Observation: (Figure 7 (a) and (b)) Without folding, NCSS allocation gives shorter average turnaround time (ATT) than CSS. The CSS allocation, when unbounded folding is used, gives less ATT than NCSS allocation.

NCCSS outperforms CSS in terms of average turnaround time when folding is not supported. However, when folding is supported, CSS outperforms NCCSS and can sustain much higher system loads.

Figures 10 and 11 show the average turnaround time (figure 10) and system utilization (figure 11) vs. system load for CSS and NCSS (i) without folding (ii) with bounded folding and (iii) unbounded folding.

Observation: (Figures 10 and 11) Folding is useful as it provides low average turnaround time and high system utilization at high system loads.

Observation: (Figure 11) Unbounded folding gives higher system utilization than bounded folding, however, it gives noticeably higher average turnaround times at high system loads.

7. Conclusions and Future work

We experimentally demonstrated that time-sharing gives comparable results to space sharing allocation and may perform better at relatively large allocation size. Simulation, based on the real observed execution times of the selected applications, showed that non-contiguous allocation gave better results than contiguous allocation when folding is not supported. However, when folding is supported, contiguous allocation gave the best mean turnaround times.

As a future work, we may perform the same study on different computer architectures. Also, we may conduct the same experiment using different parallel programming languages to compare their performance with different processor allocation strategies.

8. References

[0] Fraij, Faris, Contiguous and Noncontiguous Space Sharing Strategies in Parallel Computers, Master thesis, al-Albayt University, Jordan, 1998

[1] Sulieman Bani-Ahmad, Experimental study on Time and Space Sharing on the PowerXplorer, Master thesis, AABU, Mafraq, Jordan.

[2] Addison C. A., Getov V. S., Hey A. J., Hockney R. W., and Wolton I. C., The GENESIS Distributed-Memory Benchmarks, Journal of Computer Benchmarks, Vol. 8, 1993, pp 257-271.

[3] Bailey, D. H., The NAS Parallel Benchmarks, Journal of Supercomputer Applications, vol. 5, no. 3, 1991, pp 66-73.

[4] Breshears, C. P. and Langston, M. A., Parallel Benchmarks and Comparison-based Computing. Proceedings of the International Conference on Parallel Computing, Gent, Belgium, September 1995.

[5] Sullivan, D. M. Electromagnetic Simulation using the FDTD Method. First edition. IEEE press, New York, 2000.

[6] Quinn, M. J. Parallel Computing Theory and Practice. Second edition, McGRAW-HILL Inc, New York, 1994.

[7] PARSYTEC EASTERN EUROPE GmbH, PARIX, version 1.3.1 -PPC Reference Manual, PARSYTEC EASTERN EUROPE GmbH, 1995.

[8] Zhu, Y. Efficient Processor Allocation Strategies for Mesh-connected Parallel Computers, Journal of parallel and distributed computing, Vol. 16, 1992, pp 328-337.

[9] D. Feitelson, L. Rudolph, and U. Schwiegelshohn. Parallel Job Scheduling -- a Status Report. In 10th Workshop on Job Scheduling Strategies for Parallel Processing, New-York, NY, June 2004.

[10]Foster, I. T., Designing and building parallel algorithms. First edition, Addison-Wesley Publishing Company, Massachusetts, 1995.

[11]Feitelson D., A Survey of Scheduling in Multiprogrammed Parallel Systems, IBM research report RC 19790 (87657), Revised version, 1995.

[12]Ismail, I. M. and Davis, J. A., Adaptive Run-to-completion Job Scheduling Policies for Parallel Computers, proceedings of International conference on electronics, circuits and systems, December 1995, pp 61-66 (a)

[13]Lo, V., Windish, K. Liu, W. and Nitzberg, B. Non-contiguous Processor Allocation Algorithms for Mesh-Connected Multiprocessors, IEEE transactions on parallel and distributed systems, 1997, pp 712-725.

[14]L P Santos, V Castro, A. Proenca. “Evaluation of the Communication Performance on a Parallel Processing System”. Proceedings of the 4th European PVM/MPI Users' Group Meeting, 1997.

[15]Mark Crovella, Ricardo Bianchini, Thomas J. LeBlanc: Using Communication-to-Computation Ratio in Parallel Program Design and Performance Prediction. SPDP 1992.

.