A Performance Characterization of UPC vs. MPI

Anup Tapadia
Fallon Chen
CSE 260 Fall 2006 Final Project

Introduction

Background and Motivation

One of the most widely used language libraries for parallel computation is MPI. MPI is based on the single program multiple data model (SPMD) and uses message passing to communicate between nodes. The advantage of using such a model is a relatively low level view of the code and system, so that a programmer can easily fine tune code for performance. The disadvantage of using this model is that the use of the message passing interface often obscures the actual algorithm being implemented, which makes debugging and maintenance of the code difficult and time consuming. As a result, MPI tends to be used for scientific applications, since for most scientific applications performance is more important than ease of reuse and maintenance.

There are several languages and libraries that go in the opposite direction of MPI. They obscure communication by using threads and a shared address space and focus on making programming more simple. As a result, these global address space (GAS) languages and libraries scale less easily than MPI, but are easier to use and maintain.

Partitioned global address space (PGAS) languages attempt to find a happy medium between the GAS and message passing models by providing both high performance and ease of programming. UPC is one such language. UPC is an explicit extension of ANSI C for a PGAS model. By providing a partitioned view of the global address space, the programmer can still fine tune code for performance. The global view of address space causes the algorithm being implemented to be more visible in code and therefore easier to debug and understand. Research by Kathy Yelick and Parry Husbands shows that UPC performs comparably with MPI, suggesting that UPC is a parallel programming language where both high performance and ease of development are both obtainable. In this project, we explored whether or not this was true.

Goals and Approach

We conducted our exploration of UPC by implementing several programs in both MPI and UPC, and comparing their latency, bandwidth and scalability as well as how easy it was to develop these programs in MPI and UPC. These programs are the Monte-Carlo calculation of pi, the ring program, binary sort, matrix multiply and connected components labeling. These programs were chosen because they either tasked the communication of UPC (ring, binary sort, pi) or the ease of using the partitioned global address space (matrix multiply, connected components labeling).

Testing was done on DataStar at the San Diego Super Computer Center, as well as on a small 8 node cluster called spindel.

Results and Interpretation

Monte Carlo Pi Calculation

Description

Monte Carlo methods can be thought of as statistical simulation methods that utilize a sequences of random numbers to perform the simulation. The name "Monte Carlo" was coined by Nicholas Constantine Metropolis (1915-1999) and inspired by Stanslaw Ulam (1909-1986), because of the similarity of statistical simulation to games of chance, and because Monte Carlo is a center for gambling and games of chance. In a typical process one compute the number of points in a set **A** that lies inside box **R**. The ratio of the number of points that fall inside **A** to the total number of points tried is equal to the ratio of the two areas (or volume in 3 dimensions). The accuracy of the ratio depends on the number of points used, with more points leading to a more accurate value. Monte Carlo methods have several applications and one of them is calculation of Pi.

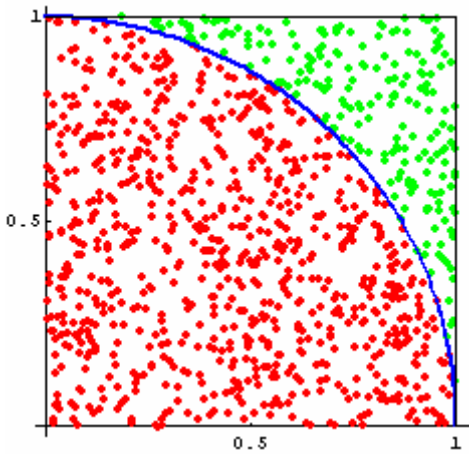


Fig 1. Monte Carlo Pi Calcuation

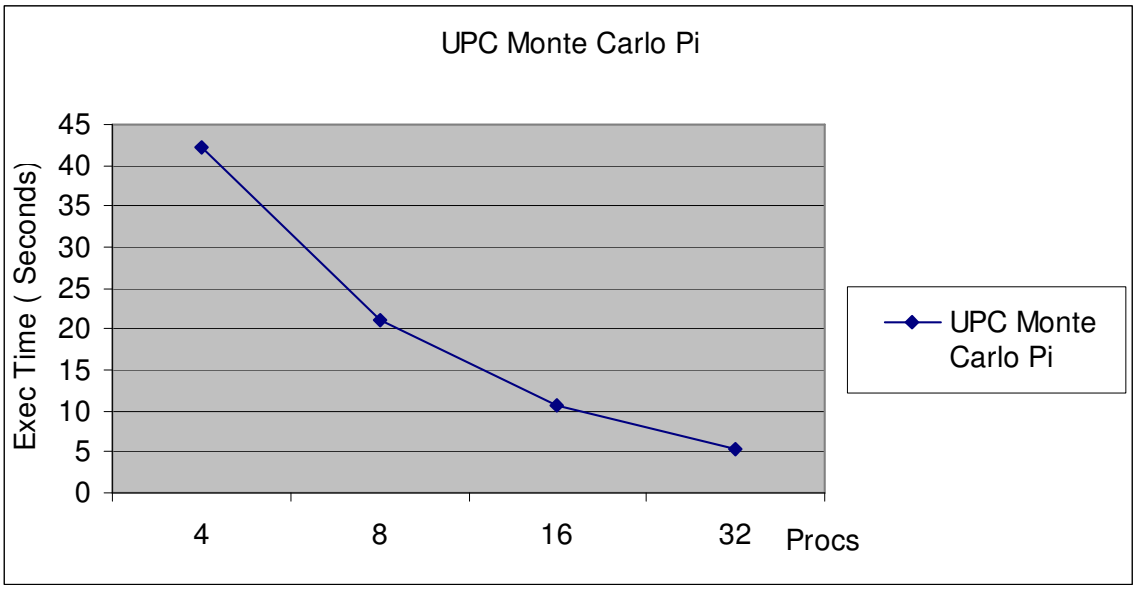
Serial Algorithm for Pi calculation using Monte Carlo methods –

1. Fix the number of throws t.
2. Randomly throw darts at x,y positions in a unit square
3. If x2 + y2 < 1, then point is inside circle.
4. Repeat step 2 and 3 t times.
5. Compute ratio: # points inside / # points total
6. p = 4*ratio

The parallel algorithm works as follows –

1. Fix the number of throws t.
2. Randomly throw darts at x,y positions in a unit square
3. If x2 + y2 < 1, then point is inside circle.
4. Repeat step 2 and 3 (t / total # of THREADS) times.
5. Compute ratio: # points inside / # points total
6. All threads add ratios to a shared variable total_ratios
7. p = 4*total_ratios

Results



Graph 1 – Scaling of UPC Monte Carlo Pi on 32 processors (DataStar)

Table 1 - Data for Graph- 1

Procs	Reading 1 (Seconds)	Reading 2 (Seconds)	Reading 3 (Seconds)	Reading 4 (Seconds)	Reading 5 (Seconds)	Average Exec Time(seconds)
4	42.196622	42.19647	42.19552	42.19777	42.19886	42.1970466
8	21.13776	21.1335	21.20228	21.13252	21.20862	21.1629354
16	10.571301	10.57928	10.62557	10.57393	10.57283	10.5845806
32	5.294534	5.312962	5.290094	5.289933	5.296752	5.296855

Table 2 - Processor Geometry configurations

Procs	Processor Geometry
4	1 node x 4 procs
8	1 node x 8 procs
16	2 nodes x 8 procs
32	4 nodes x 8 procs

Analysis

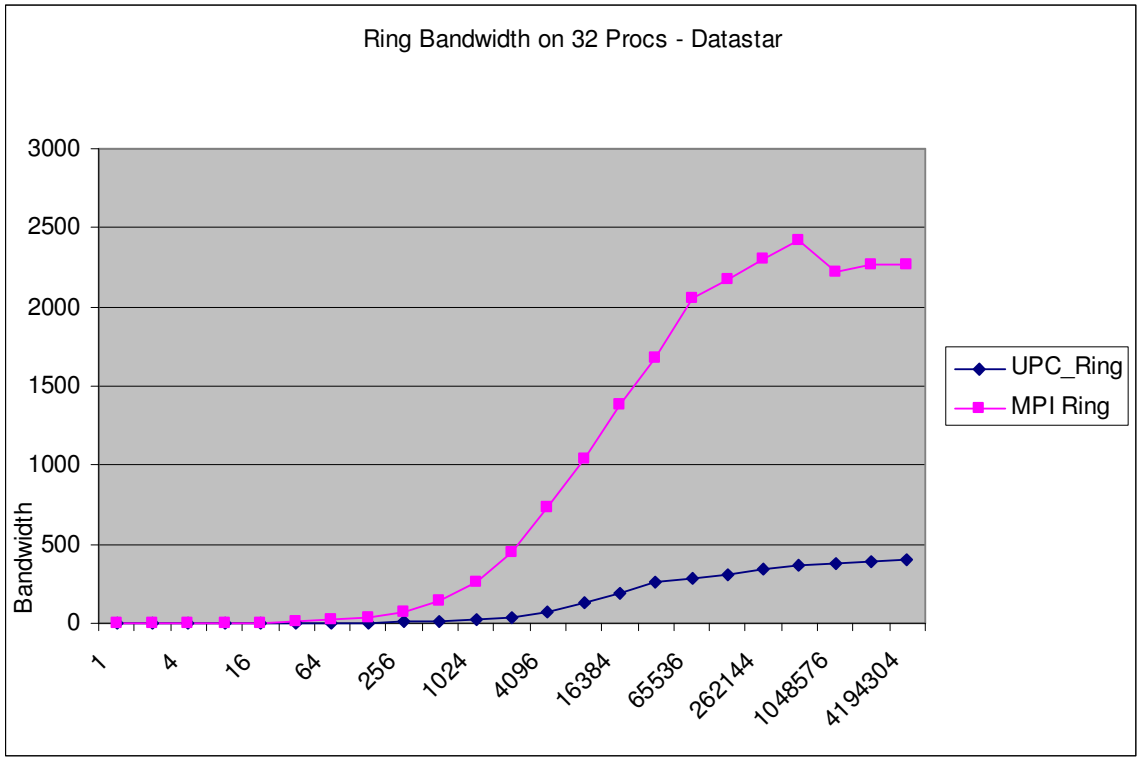
Although the Monte Carlo Method is often useful for solving problems in physics and mathematics which cannot be solved by analytical means, but it is a rather slow method of calculating pi. But it was useful as a study application for UPC. The UPC code was easy to write due to the shared memory architecture. Each process would acquire locks and operate in the shared memory region. This relieved the need for implicit synchronization between processes. Graph 1 shows that the UPC program scaled well when the number of processors were increased. It can also be observed that due to the architecture of DataStar, the program scaled really well when the number of processors increased from 4 to 8 as they were present on the same compute node. But after the computation goes beyond the node boundaries, the scaling is relatively less and goes on decreasing as the numbers of nodes are increased.

Ring Program

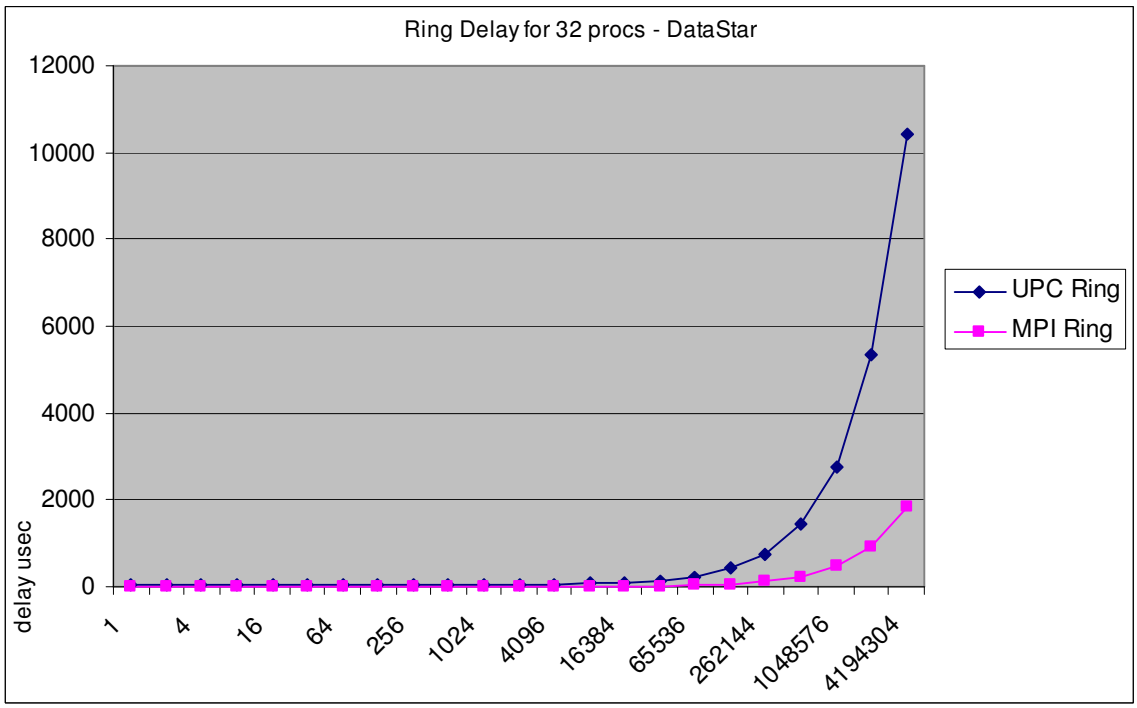
Description

Ring is a popular program used in parallel programming benchmarks for MPI. This program treats the processors as if connected in a ring. Process 0 sends a message to successor process 1, and so on until process P-1, which then sends the message back to process 0, which is P-1's successor in mod P arithmetic. The cycle then repeats. Messages of various sizes are passed around the ring, starting with 1 byte long messages and progressing in powers of two up to a user-specified limit. We implemented the UPC version of Ring and following are the comparative results with MPI.

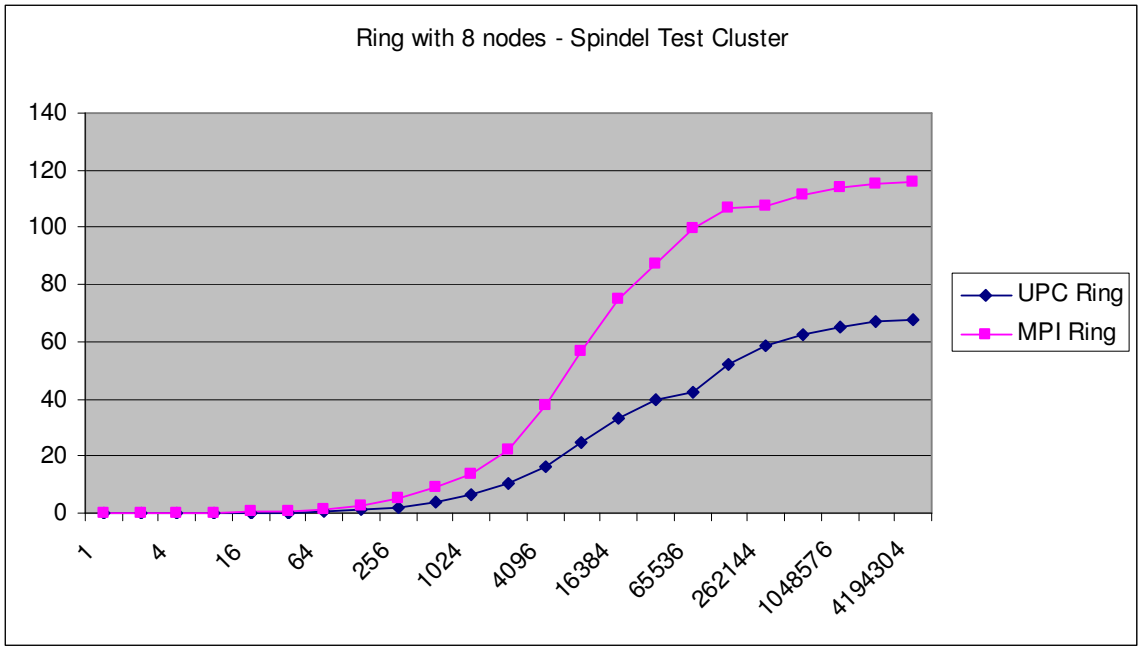
Results



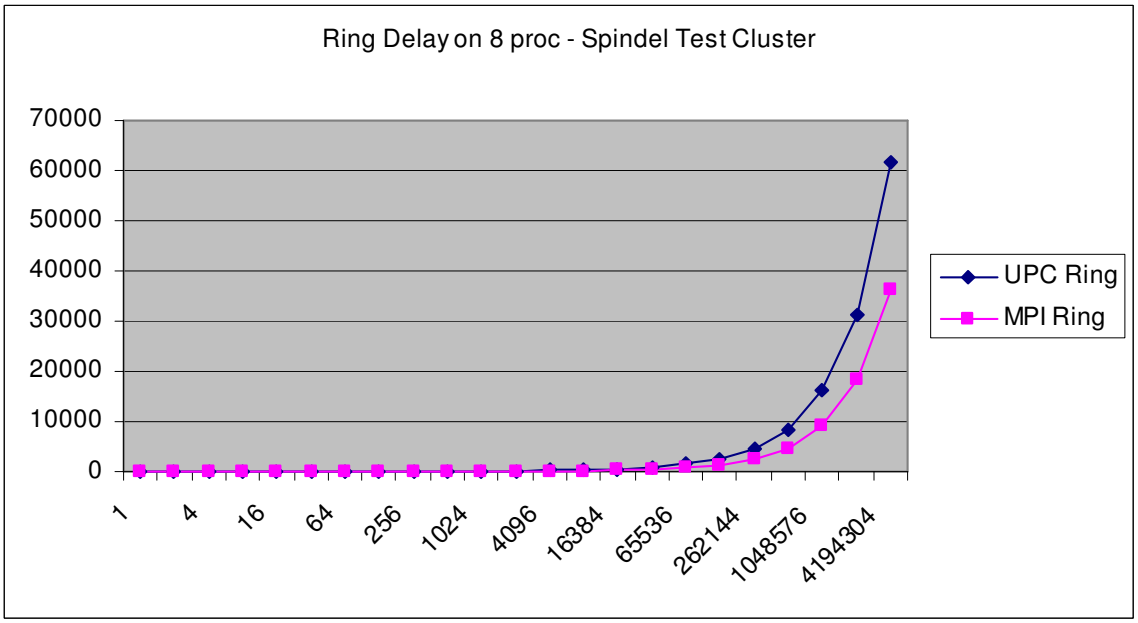
Graph 2 : Ring Bandwidth on increasing message size (DataStar 8x4)



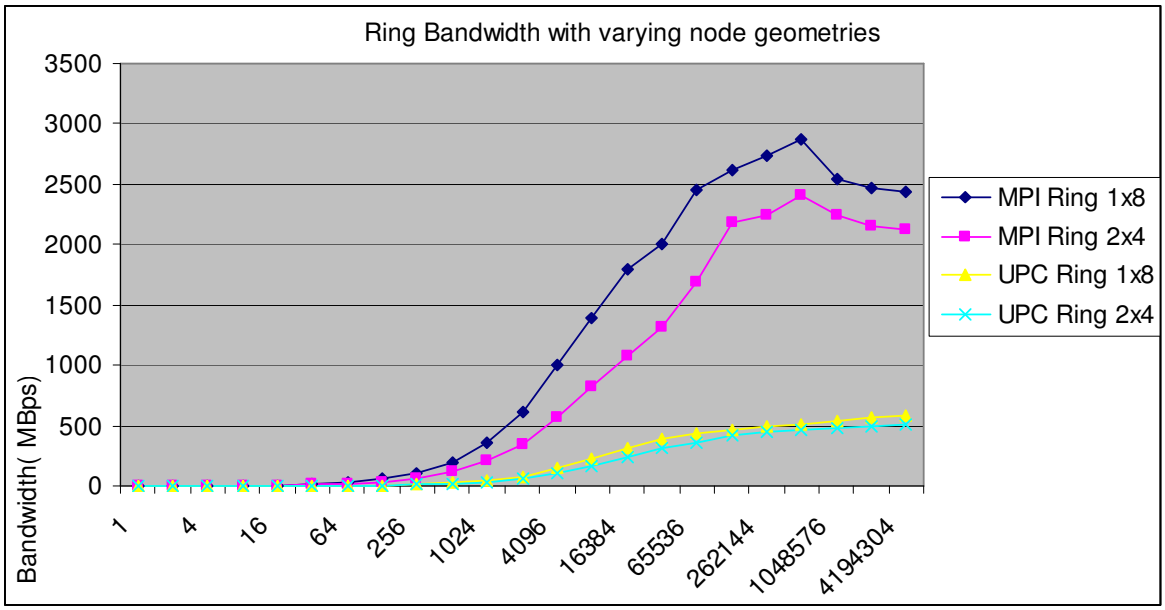
Graph 3 : Ring delay on increasing message size (DataStar 8x4)



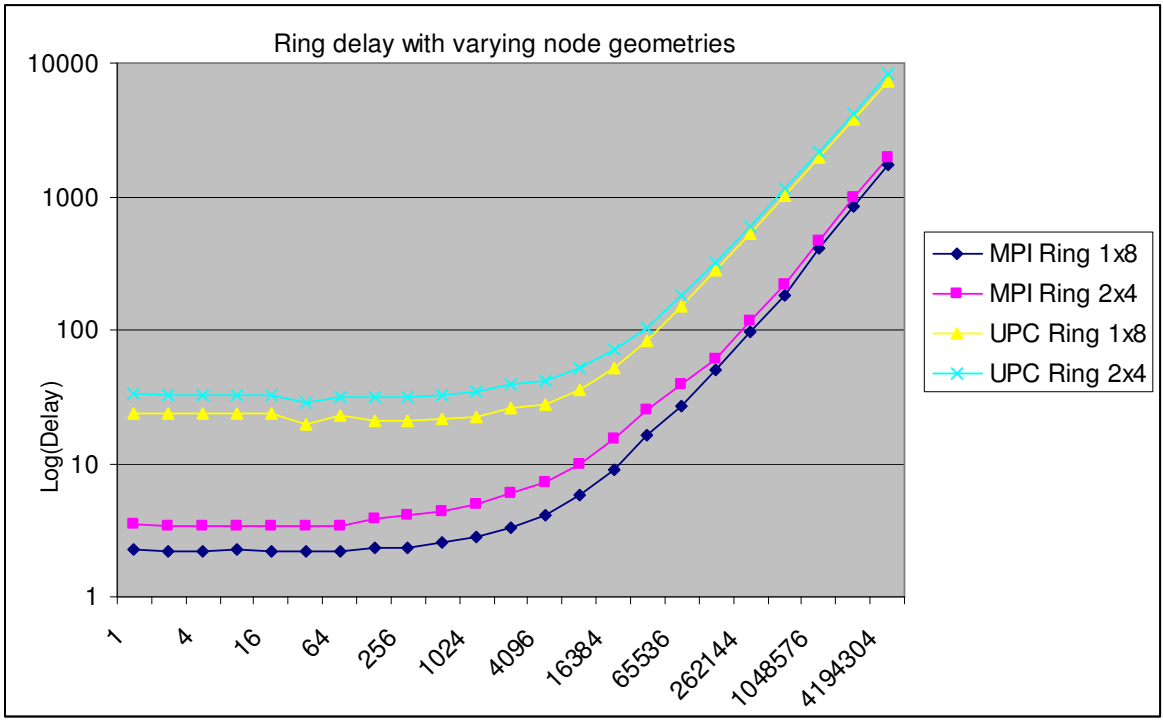
Graph 4 : Ring Bandwidth on increasing message size (Spindel Clusters)



Graph 5 : Ring delay on increasing message size (Spindel Clusters)



Graph 6: Ring Bandwidth with different node geometries on DataStar



Graph 7: Ring Delay with different node geometries on DataStar

Analysis

Graph 2 shows that peak bandwidth on DataStar with MPI was around 2500 MBps where as with UPC it was only about 500 MBps. This shows that UPC code was approximately 5 times less efficient. Also the bandwidth peaked out earlier in UPC than in MPI.

Graph 3 shows that the delay with UPC was greater than that with MPI. This may be due to the use of single process number 0 to move memories across shared memory spaces. There would have been an overhead for process 0 to instruct each process every time it was their turn to copy the memory locations resulting in higher delays.

Graph 4 shows that the peak bandwidth with Spindel was around 120 MBps with MPI and it was around 70 MBps for UPC. This shows that UPC code was approximately 12/7 times slower which is better than DataStar.

Graph 5 shows that the delay with UPC was greater than that with MPI. The explanation can be same as that of graph 3.

Graph 6 and 7 show the delay and bandwidth for the MPI ring and UPC ring. It can be observed that due to the architecture of DataStar node connectivity, the bandwidth decreases and the delay increases if we cross the local node boundaries.

The results favored MPI over UPC in all the cases. This may be due to the way code was written in UPC. Ring is probably not the right program to benchmark performance comparisons of bandwidth and delay between MPI and UPC because of the way these languages approach the problem are different. Due to shared memory architecture, UPC does not need to pass messages from one processor to another and I found it difficult to instruct each processor to independently copy some piece of memory when the previous one was done. Hence the UPC code contained only single process 0 which used to move information between different affinities of shared memories. There was also some problem with the GASNET driver on DataStar. That might be the reason to explain that UPC performed better on Spindel than on Datstar.

Binary Sort

Description

Binary sort is a method of sorting an array of elements by dividing them into small computational blocks in a tree manner. This method divides a given sequence into two halves (until only one element remains) and sorts both halves recursively. The two halves are then merged together to form a sorted sequence.

Following is a pseudo serial code for binary sort –

```
sorted-sequence BinarySort (sequence)
{  if (# elements in sequence > 1)
   {  seqA = first half of sequence
      seqB = second half of sequence
      BinarySort (seqA);
      BinarySort (seqB);
      sorted-sequence = merge (seqA, seqB);
   }
   else sorted-sequence = sequence }
```

Following diagrams show an example of a sample tree formed during binary sort.

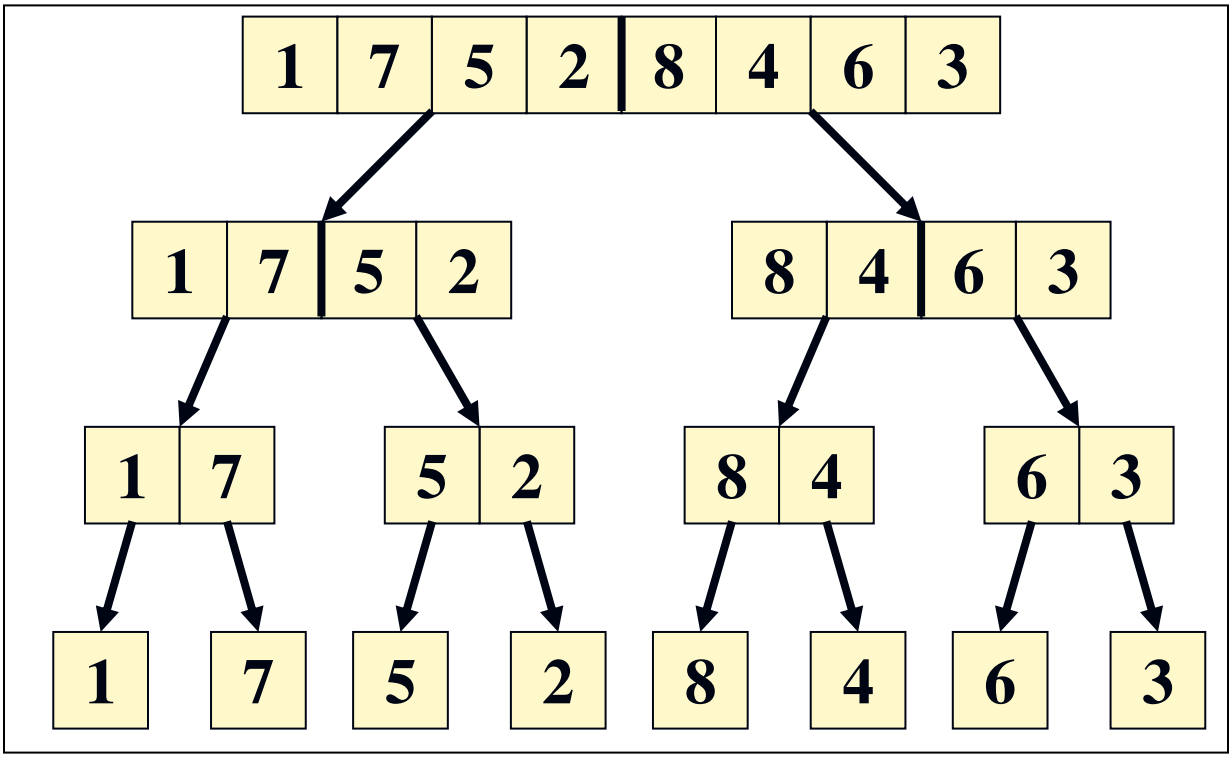


Figure 2 : Example – Sorting stage 1

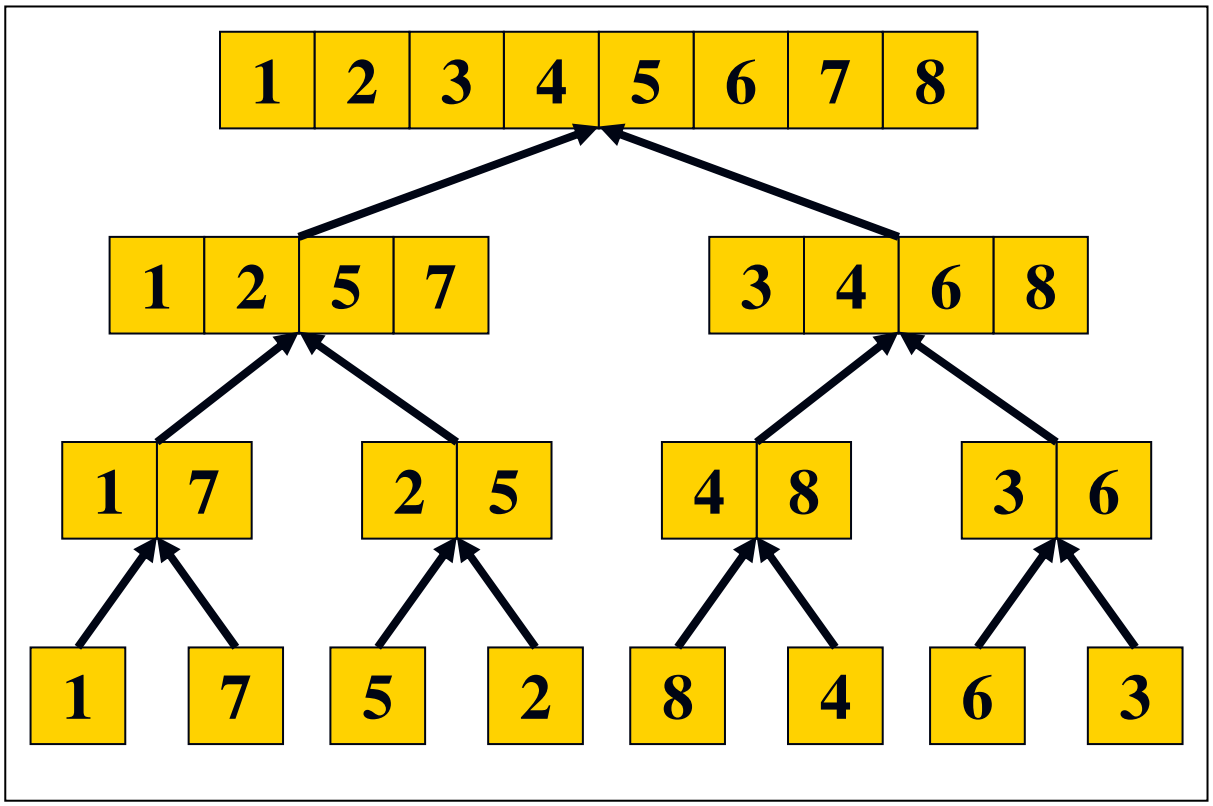
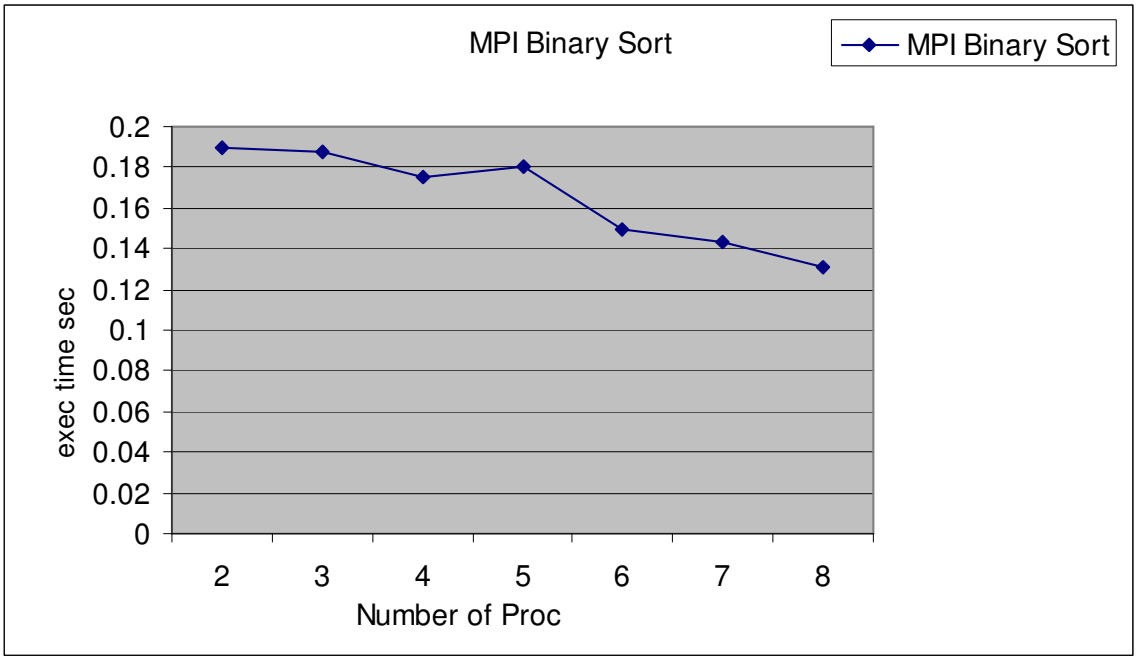


Figure 3 : Example – Sorting stage 2 . Completed Sort

For the parallel algorithm we divide work and gather the result. For the MPI implementation we divide work in two to two processors. Have each of these processors divide their work again, until either no data can be split again or no processors are available anymore. The UPC implementation was left incomplete because it was difficult to instruct the other processes directly to perform some task due to shared memory approach. Good processor geometry needs to be figured out to get this code executed in parallel using UPC and it would be different from the tree used in MPI implementation. I tried to work on one potential processor geometry but was unable to complete it due to time limitations.

Results



Graph 8 : MPI binary sort scaling performance

Analysis

Graph 8 shows that the code did not scale very effectively. The reason for this is that the parallel implementation does not achieve even workload on all processors the way the algorithm was designed. This algorithm can be improved to achieve even workload and optimize the run time and scaling in future. Shared memory with UPC can prove a better alternative for this.

Matrix Multiply

Description

Matrix multiply is a favorite example of those who like to demonstrate the ease of use of UPC. For certain cases, matrix multiply works out very neatly due to the way address partitioning works in UPC. The actual physical location of data is controlled by blocking. Each element of an array is distributed in a round robin fashion, dependent on the block size. For instance, a one dimensional array of length 8, with block size 1, distributed over 4 threads, would distributed in this way:

thread 0	***thread 1***	***thread 2***	***thread 3***
array[0],array[4]	array[1],array[5]	array[2],array[6]	array[3],array[7]

Blocking determines how many elements are distributed per thread per iteration. So the same example with a block size of 2 would look like this:

thread 0	***thread 1***	***thread 2***	***thread 3***
array[0],array[1]	array[2],array[3]	array[4],array[5]	array[6],array[7]

Since two dimensional arrays in UPC are stored in row major order as they are in UPC, a 8x4 array with block size 1 would look like this:

thread 0	***thread 1***	***thread 2***	***thread 3***
array[0,0]	array[0,1]	array[0,2]	array[0,3]
array[1,0]	array[1,1]	array[1,2]	array[1,3]
array[2,0]	array[2,1]	array[2,2]	array[2,3]
array[3,0]	array[3,1]	array[3,2]	array[3,3]
array[4,0]	array[4,1]	array[4,2]	array[4,3]
array[5,0]	array[5,1]	array[5,2]	array[5,3]
array[6,0]	array[6,1]	array[6,2]	array[6,3]
array[7,0]	array[7,1]	array[7,2]	array[7,3]

In other words, each thread now has the nth and nth+1 columns of the array, where n is the thread number.

If the block size for the same array is 8, however, we get :

thread 0	***thread 1***	***thread 2***	***thread 3***
array[0,0]	array[1,0]	array[2,0]	array[3,0]
array[0,1]	array[1,1]	array[2,1]	array[3,1]
array[0,2]	array[1,2]	array[2,2]	array[3,2]
array[0,3]	array[1,3]	array[2,3]	array[3,3]
array[0,4]	array[1,4]	array[2,4]	array[3,4]
array[0,5]	array[1,5]	array[2,5]	array[3,5]
array[0,6]	array[1,6]	array[2,6]	array[3,6]
array[0,7]	array[1,7]	array[2,7]	array[3,7]

Each thread now has the nth row of the array, where n is the thread number.

As a result, we can have a neat division of a two dimensional array by rows or by columns. If there are more rows than threads, or columns than threads, then each thread gets the n to (n + n/blocksize) columns or rows. For instance in the above example with a block size of 4, thread 0 got columns 0 and 1, thread 1 got columns 2 and 3, thread 2 got columns 4 and 5, and thread 4 got columns 6 and 7. This can be generalized for any number of threads and array dimensions. Note that this only works when the number of rows and columns in the array are a multiple of the number of threads—otherwise the arrays wrap around.

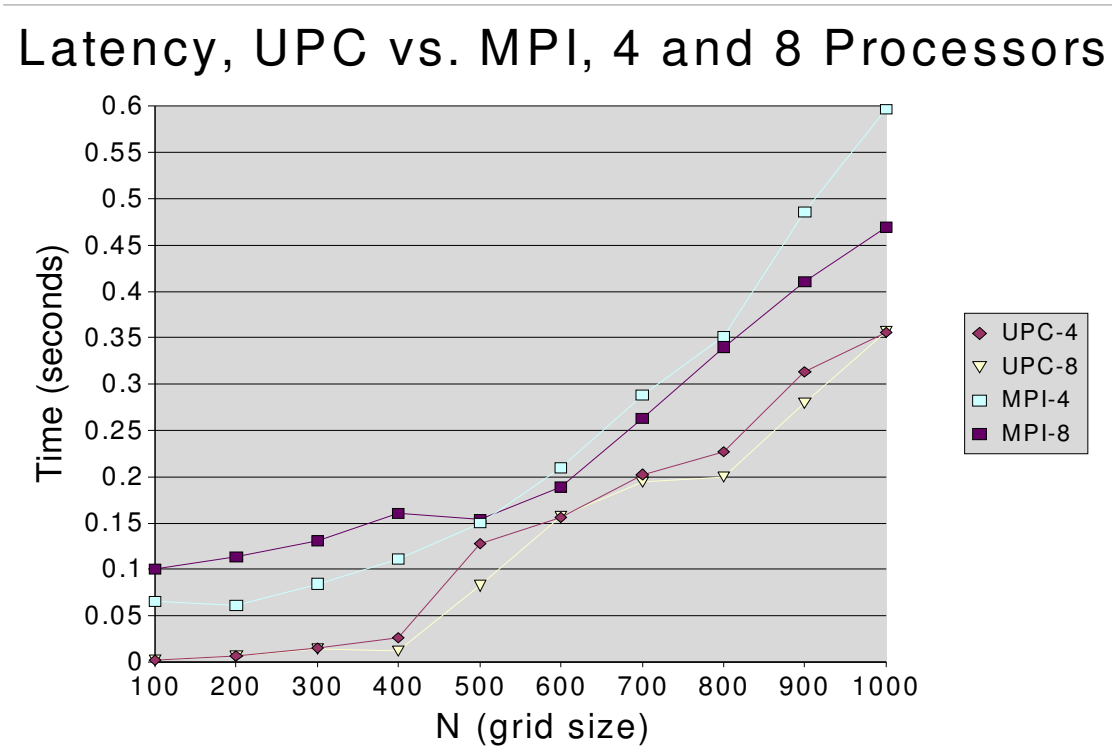
This makes storing the matrices for matrix multiplication simple. When we multiply a matrix A by a matrix B to produce a matrix C, we multiply each row of A by every column in B. In our code for matrix multiply, we made it so that each thread takes its local row(s) of A and gathers all the columns of B from all the other processors. The gathered columns of B are stored column wise in a single dimensional array. Multiplying the local row(s) of A by the columns of B produces the rows of C correlated with the thread id. Since B is stored column wise, accessing B for the multiply is very fast. The columns of B were collected by each thread using a UPC collective function called upc_broadcast, which collects the different parts of matrix B to reform a full version of matrix B in each thread.

Results

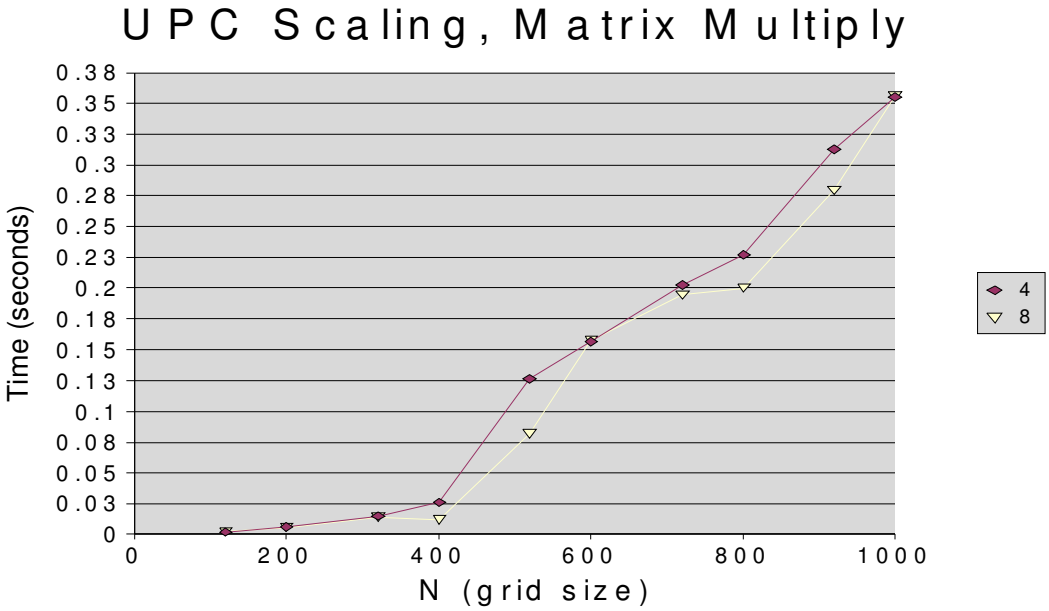
Results were collected on spindel, running on 4 and 8 nodes. There was a bug that we couldn't fix in time that prevented the matrix multiply from running on DataStar, so we only collected data for up to 8 nodes. The MPI matrix multiply was an implementation of SUMMA written by Stephen Fink.

Time in seconds for square matrix multiply of nxn arrays:

n	UPC-4 nodes	MPI-4 nodes	UPC-8 nodes	MPI-8 nodes
120	0.00198	0.06469	0.00224	0.09948
200	0.00600	0.06066	0.00571	0.11276
320	0.01498	0.08384	0.01408	0.12987
400	0.02581	0.11029	0.01161	0.16057
520	0.12646	0.14980	0.08186	0.15417
600	0.15658	0.20915	0.15736	0.18920
720	0.20234	0.28766	0.19413	0.26305
800	0.22717	0.35066	0.19974	0.33928
920	0.31249	0.48533	0.27940	0.41064
1000	0.35477	0.59657	0.35576	0.46970



\



Analysis

We found that in most cases the matrix multiply in UPC performed better than the matrix multiply in MPI. This is because the upc_all_broadcast function used to gather the columns of B is used only number of threads times, while the number of data exchanges used for each processor in the MPI algorithm is 2 *number of processor rows * number of processor columns data exchanges. Since there were more communication calls in the MPI code and the amount of data exchanged for both programs was relatively small, the overhead in MPI was larger than the overhead in UPC, which made the difference in the runtimes of the programs. A more effective test would run on larger values of n, so that the effect of overhead could be factored out.

The UPC was also much shorter than the MPI code (about 1/5 the size), and much more readable. This isn't a particularly good comparison, however, since the UPC code has several limitations on it that the MPI version does not. Specifically, the UPC code only works on nxn matrices where n is a multiple of the number of threads, and where n is small since it is limited by the square root of maximum block size (a constant in UPC, around 1MB).

Additionally, we found that the UPC code wasn't particularly scalable—although the performance on 8 processors was better than the performance on 4 processors, the difference isn't very pronounced. This probably resulted from the overhead of communicating among more processors. Better results would be obtained from testing larger values of n.

Connected Components Labeling

Description

To implement connected components labeling, we used a union find algorithm for UPC and a depth first search algorithm on MPI. The UPC code first divides the forest up among the threads then performs a the Suzuki algorithm for connected component labeling serially on each thread's piece. It then uses global arrays to share label data in the global relabeling of connected clusters. The global relabeling is performed using the Quad algorithm.

Current algorithm:

1. split up grid among threads
2. do serial labeling
3. exchange ghost cells with all neighbors
4. do a union-find with the labels of each neighbor
5. add new labels to global array of labels
6. wait until all threads are finished with steps 1-5
7. use global array of labels to relabel local grid

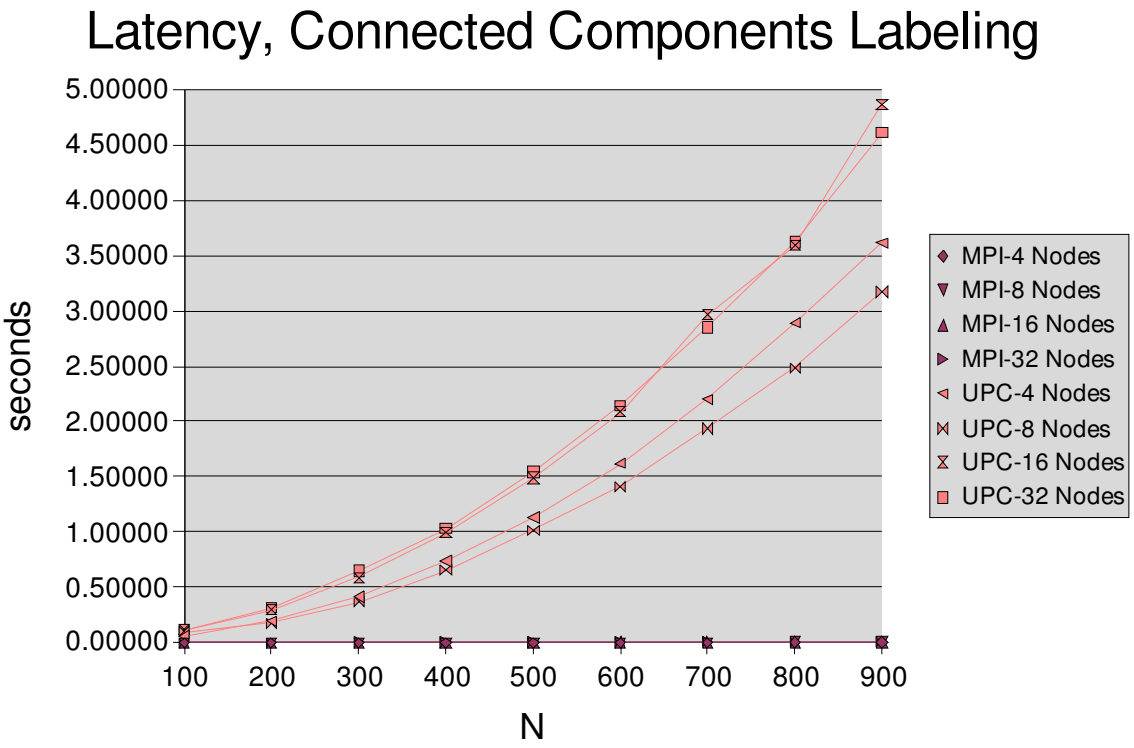
Results

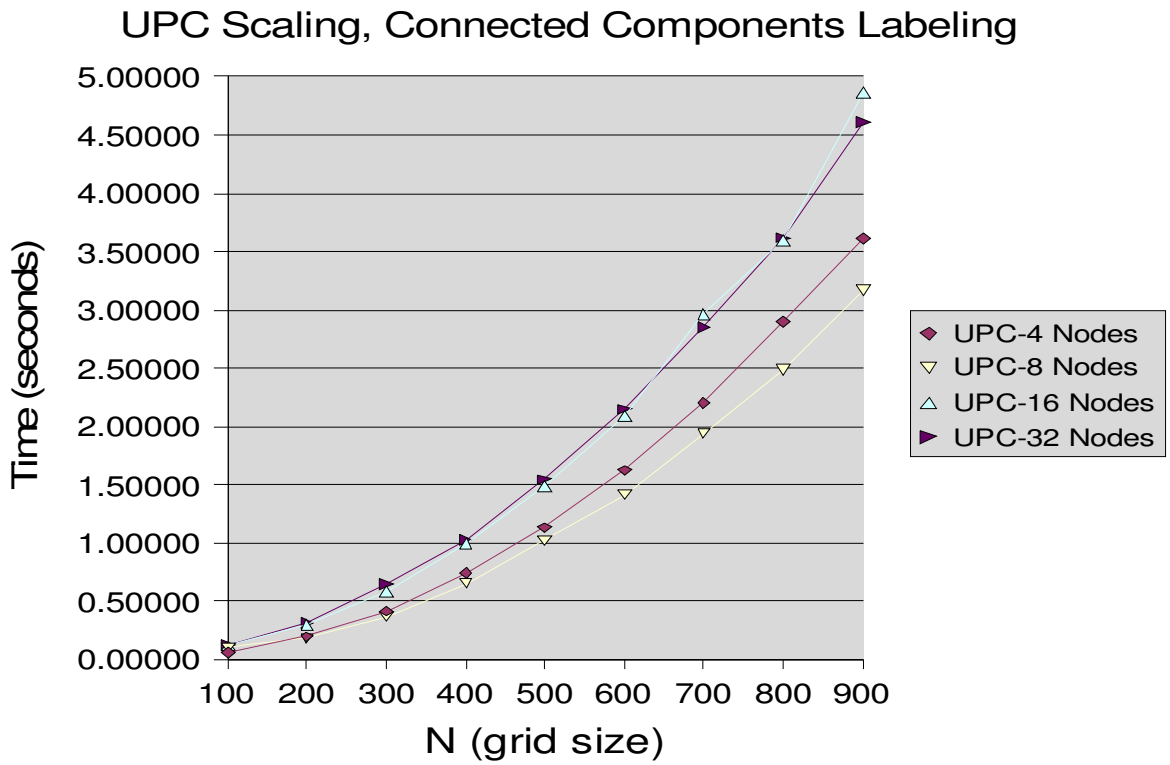
We ran connected components labeling on DataStar, on 4,8,16 and 32 processors, with NxN grids of sizes ranging from 100 to 900.

Number of Processors	***Geometry (node x tasks per node)***
4	1x4
8	1x8
16	2x8
32	4x8

N	MPI-4 Nodes	MPI-8 Nodes	MPI-16 Nodes	MPI-32 Nodes	MPI-64 Nodes
100	0.00023	0.00030	0.00029	0.00037	0.00229
200	0.00032	0.00038	0.00047	0.00204	0.00455
300	0.00055	0.00058	0.00140	0.00326	0.00644
400	0.00082	0.00084	0.00177	0.00366	0.00937
500	0.00119	0.00111	0.00241	0.00571	0.01123
600	0.00165	0.00178	0.00297	0.00754	0.01446
700	0.00215	0.00209	0.00354	0.00856	0.01464
800	0.00277	0.00288	0.00359	0.00893	0.01887
900	0.00337	0.00311	0.00471	0.01123	0.02013

N	UPC-4 Nodes	UPC-8 Nodes	UPC-16 Nodes	UPC-32 Nodes	MPI-64 Nodes
100	0.05925	0.10136	0.11637	0.11853	0.14663
200	0.19500	0.18148	0.29969	0.31761	0.35554
300	0.41521	0.36898	0.58723	0.64710	0.62040
400	0.73870	0.65482	0.99504	1.02796	1.06691
500	1.13180	1.02065	1.49242	1.54557	1.49843
600	1.62270	1.41440	2.08712	2.14727	2.08625
700	2.20027	1.93674	2.96146	2.85012	2.70834
800	2.89365	2.48331	3.59217	3.61860	3.46827
900	3.61412	3.17104	4.86866	4.60903	4.31663





(MPI run times are all in purple, UPC runtimes are all in pink)

Analysis

The UPC code does not scale well. This is probably due to the fact that the UPC code spends a lot of time accessing global arrays. More processors mean more overhead and contention for access to global space, so 16 and 32 processors take more time than 4 and 8 processors. Since the inputs are small, the amount of overhead is probably the biggest factor.
Currently, the global array is accessed through a for loop, though not sequentially:

```
for(i = 0; i < N;i++)
  globalarray[labels[i]] = x;
```

As a result the values being accessed are scattered across the threads and communication must be initiated for each new thread access. This quickly becomes slow around N=100 To fix this, I would try to use the UPC bulk access functions, upc_memget and upc_memput, which get and put arrays to and from shared memory. These functions will only copy from a single affinity (a single thread, essentially) at a time, so attention must be paid to where items are put in shared storage. For instance, if an array of size 8 is stored in memory across 4 threads, with a block size of 1, thread 0 will have items array[0] and array[4], so a upc_memget of 2 items from shared memory starting from array[0] will yield array[0] and array[4], rather than array[0] and array[1].

For the same reason, the UPC code is slower than the MPI code. This can probably be fixed if we stop using a global array of labels and instead use the shared arrays to faciliatate the exchange of ghost cells and ghost cell labels. In other words, make the exchange between easily read code and faster code.

It was easier to code this up than the MPI version because by eliminating the send and receive calls the connected component labeling algorithm was much more clear to read.

Progress & Milestones

Goals Modified or Not Met

We did manage to compare the performance and ease of coding of UPC and MPI, which was our main goal. However, we did not test scaling for very large numbers of processors or very large inputs.

Milestones Modified or Not met

We completed five out of the six programs we meant to do. The last program, the Game of Life was not completed due to time constraints. The time originally earmarked for developing the Game of Life program was spent on trying to develop a generalized matrix multiply using the SUMMA algorithm (which didn't work out and was scrapped in favor of the simple matrix multiply presented in this report). We also did not complete a binary sort implementation in UPC, because it is difficult to instruct each processor independently to develop the binary sort tree processor geometry.

Conclusions

We found that overall, UPC was an easier language to use for development. Being able to access different parts of a shared memory without having to explicitly find the processor that owned that piece of memory was very helpful in debugging. Although generally speaking, you need to know where the data resides in memory to fine tune the code. Another reason UPC was an easier language for development was that the readability of UPC code is also much better than the readability of MPI code. For example, copying one array of data in MPI from one node to another would require a call in the sending node to MPI_send, and a call in the other node to MPI_Recv. Both of these calls have to be synchronized, since the calls would block or exchange the wrong data if they were out of order. In contrast, copying one array of data in UPC from one node to another requires a single call to upc_memget to the shared memory location to the node that needs the data, or it can directly work on the global memory.

Coding in UPC required a mental model shift, since gathering data no longer requires synchronization of the nodes involved, and also is no longer thought of in terms of sending and receiving data, but simply taking data from the right location. Fine tuning the code requires finding better ways to lay out data in shared memory and making as much use of collective and bulk operations as possible, rather than minimizing the amount of communication and data sent at a time.

Performance was found to be not quite as good as that in MPI in most cases, but fine tuning and compiler optimizations might cause the code to yield better results. For instance, the original version of the matrix multiply would run about 10 seconds on a 100x100 grid. However, using blocking upc_memget and the upc_broadcastall collective caused the code to run in 0.001 seconds. In the case of embarrassingly parallel programs such as the pi calculation, performance of UPC can be better than that of MPI. But in the case of ring and binary sort, UPC runs more slowly than MPI code. In the case of connected components labeling, code that was written naively ran much more slowly than the MPI code. This happened because initiating communication and handling interference for direct access of many elements in a global array is very slow.

UPC is generally easier to debug and read, but more difficult than MPI to optimize. It nearly achieves its goal of being both easy to maintain and easy to achieve high performance in, but hasn't yet reached it.

Appendix

Data for Graph 2 (Data Star)

Bandwidth UPC Ring

Reading 1	Reading 2	Reading 3	Average
0.024	0.024	0.024	0.024
0.048	0.048	0.049	0.048333
0.097	0.097	0.097	0.097
0.194	0.194	0.194	0.194
0.384	0.387	0.388	0.386333
0.818	0.813	0.822	0.817667
1.613	1.605	1.619	1.612333
3.218	3.269	3.31	3.265667
6.295	6.203	6.277	6.258333
12.16	12.121	12.239	12.17333
23.437	23.477	23.461	23.45833
40.909	40.742	41.081	40.91067
76.499	76.468	75.516	76.161
124.593	123.917	124.139	124.2163
184.935	188.256	187.693	186.9613
259.252	259.534	258.615	259.1337
297.814	298.624	243.31	279.916
334.454	332.951	259.432	308.9457
357.982	354.285	328.158	346.8083
368.531	364.487	351.301	361.4397
371.646	399.977	372.211	381.278
379.737	424.059	381.521	395.1057
385.953	436.463	386.561	402.9923

Bandwidth MPI Ring

Reading 1	Reading 2	Reading 3	Average
0.29	0.3	0.33	0.306667
0.59	0.67	0.68	0.646667
1.07	1.35	1.34	1.253333
2.71	2.63	2.77	2.703333
5.37	5.35	5.4	5.373333
10.76	10.71	11.04	10.83667
19.04	21.15	21.25	20.48
35.59	39.24	38.9	37.91
75.13	75.54	74.8	75.15667
141.98	140.99	143.37	142.1133
257.39	257.72	257.62	257.5767
442.18	445.49	442.4	443.3567
718.83	727.52	738.6	728.3167
1031.42	1049.84	1053.39	1044.883
1384.03	1384.16	1380.4	1382.863
1671.23	1674.6	1671.5	1672.443

2081.11	2040.41	2045.37	2055.63
2186.39	2177.49	2146.97	2170.283
2332.72	2265.65	2306.5	2301.623
2410.53	2426.57	2433.09	2423.397
2255.77	2077.96	2315.39	2216.373
2275.81	2273.88	2270.03	2273.24
2272.5	2273.34	2272.38	2272.74

Data for Graph 3 (Datastar)

Delay MPI Ring

Reading 1	Reading 2	Reading 3	Average
0.29	0.3	0.33	0.306667
0.59	0.67	0.68	0.646667
1.07	1.35	1.34	1.253333
2.71	2.63	2.77	2.703333
5.37	5.35	5.4	5.373333
10.76	10.71	11.04	10.83667
19.04	21.15	21.25	20.48
35.59	39.24	38.9	37.91
75.13	75.54	74.8	75.15667
141.98	140.99	143.37	142.1133
257.39	257.72	257.62	257.5767
442.18	445.49	442.4	443.3567
718.83	727.52	738.6	728.3167
1031.42	1049.84	1053.39	1044.883
1384.03	1384.16	1380.4	1382.863
1671.23	1674.6	1671.5	1672.443
2081.11	2040.41	2045.37	2055.63
2186.39	2177.49	2146.97	2170.283
2332.72	2265.65	2306.5	2301.623
2410.53	2426.57	2433.09	2423.397
2255.77	2077.96	2315.39	2216.373
2275.81	2273.88	2270.03	2273.24
2272.5	2273.34	2272.38	2272.74

Delay UPC Ring

Reading 1	Reading 2	Reading 3	Average
42.2	41.8	41.5	41.83333
41.5	41.7	41.2	41.46667
41.4	41.3	41.1	41.26667
41.3	41.2	41.2	41.23333
41.7	41.3	41.3	41.43333
39.1	39.4	38.9	39.13333
39.7	39.9	39.5	39.7
39.8	39.2	38.7	39.23333
40.7	41.3	40.8	40.93333
42.1	42.2	41.8	42.03333

43.7	43.6	43.6	43.63333
50.1	50.3	49.9	50.1
53.5	53.6	54.2	53.76667
65	66.1	66	65.7
88	87	87.3	87.43333
126	126.3	126.7	126.3333
220	219.5	269.4	236.3
391	393.7	505.2	429.9667
732	739.9	798.8	756.9
1422	1438.4	1492.4	1450.933
2821	2621.6	2817.2	2753.267
5522	4945.4	5496.8	5321.4
10867	9609.8	10850.3	10442.37

Data for Graph 4 (Spindel)

Bandwidth UPC Ring

Reading 1	Reading 2	Reading 3	Average
0.009	0.009	0.009	0.009
0.019	0.019	0.019	0.019
0.037	0.037	0.037	0.037
0.074	0.074	0.075	0.074333
0.148	0.141	0.144	0.144333
0.301	0.301	0.301	0.301
0.585	0.586	0.576	0.582333
1.134	1.137	1.136	1.135667
2.134	2.134	2.137	2.135
3.857	3.846	3.76	3.821
6.322	6.348	6.239	6.303
10.401	10.46	10.446	10.43567
16.116	16.589	16.019	16.24133
24.251	24.886	25.684	24.94033
33.134	33.286	33.191	33.20367
40.096	39.621	40.295	40.004
42.186	41.891	42.575	42.21733
51.555	51.259	52.789	51.86767
58.596	58.229	58.535	58.45333
62.202	62.628	62.912	62.58067
65.307	64.8	65.257	65.12133
66.776	67.276	67.031	67.02767
67.813	67.761	67.79	67.788

Bandwidth MPI Ring

Reading 1	Reading 2	Reading 3	Average
0.02	0.03	0.03	0.026667
0.05	0.05	0.05	0.05
0.1	0.1	0.1	0.1
0.2	0.2	0.2	0.2

0.39	0.4	0.41	0.4
0.77	0.8	0.8	0.79
1.51	1.54	1.56	1.536667
2.82	2.96	2.96	2.913333
5.18	5.32	5.37	5.29
8.96	8.73	9.03	8.906667
13.82	13.89	13.6	13.77
22.18	22.41	22.4	22.33
37.3	37.53	37.68	37.50333
56.28	56.76	56.55	56.53
74.4	75.2	75.23	74.94333
87.18	87.28	87.74	87.4
99.43	99.81	99.98	99.74
106.84	107.44	107.04	107.1067
106.98	107.43	107.45	107.2867
110.85	111.47	111.53	111.2833
113.68	113.67	113.56	113.6367
114.87	115.24	115.05	115.0533
115.52	116.1	115.93	115.85

Data for Graph 5 (Spindel)

Delay MPI Ring

Reading 1	Reading 2	Reading 3	Average
45	39.2	39.4	41.2
40.7	39.1	39.4	39.73333
40.7	39.1	39.1	39.63333
40.8	39.3	39.3	39.8
41.3	39.6	39.5	40.13333
41.6	40	39.9	40.5
42.5	41.6	41.1	41.73333
45.3	43.2	43.3	43.93333
49.4	48.1	47.7	48.4
57.1	58.6	56.7	57.46667
74.1	73.7	75.3	74.36667
92.3	91.4	91.4	91.7
109.8	109.1	108.7	109.2
145.5	144.3	144.9	144.9
220.2	217.9	217.8	218.6333
375.9	375.5	373.5	374.9667
659.1	656.6	655.5	657.0667
1226.8	1220	1224.5	1223.767
2450.3	2440.1	2439.6	2443.333
4729.9	4703.4	4701.1	4711.467
9223.8	9224.8	9233.9	9227.5
18257.4	18198.9	18228	18228.1
36307.2	36125.6	36180.6	36204.47

Delay UPC Ring

Reading 1	Reading 2	Reading 3	Average
112.5	107.1	107.1	108.9
107.1	106.9	106.6	106.8667
107.4	107.2	106.7	107.1
107.6	107.7	106.9	107.4
108	113.2	110.8	110.6667
106.3	106.5	106.5	106.4333
109.5	109.2	111.2	109.9667
112.9	112.6	112.7	112.7333
119.9	120	119.8	119.9
132.7	133.1	136.2	134
162	161.3	164.1	162.4667
196.9	195.8	196.1	196.2667
254.2	246.9	255.7	252.2667
337.8	329.2	318.9	328.6333
494.5	492.2	493.6	493.4333
817.2	827	813.2	819.1333
1553.5	1564.5	1539.3	1552.433
2542.4	2557.1	2482.9	2527.467
4473.8	4501.9	4478.4	4484.7
8428.8	8371.4	8333.6	8377.933
16056.1	16181.8	16068.4	16102.1
31405.8	31172.4	31286.2	31288.13
61851.5	61898.8	61871.6	61873.97

Data for Graph 6

Bandwidth MPI Ring 1x8

Reading 1	Reading 2	Reading 3	Average
0.45	0.44	0.46	0.45
0.91	0.88	0.94	0.91
1.86	1.74	1.85	1.816667
3.59	3.52	3.58	3.563333
7.41	7.05	7.53	7.33
14.97	14.15	14.79	14.63667
29.22	27.49	29.78	28.83
55.63	53.04	56.31	54.99333
106.84	104.92	109.97	107.2433
194.87	196.97	206.89	199.5767
367.96	354.59	367.96	363.5033
630.69	609.38	613.79	617.9533
1024.52	973.29	991.98	996.5967
1355.21	1389.89	1443.69	1396.263
1835.15	1769.58	1801.95	1802.227
1847.61	2088.98	2084.7	2007.097
2328.78	2359.72	2657.75	2448.75
2681.69	2718.57	2466.48	2622.247

2668.04	2858.16	2670.31	2732.17
2858.83	2969.87	2767.37	2865.357
2543.85	2513.82	2563.86	2540.51
2502.34	2460.09	2451.57	2471.333
2450.39	2433.28	2429.26	2437.643

Bandwidth MPI Ring 2x4

Reading 1	Reading 2	Reading 3	Average
0.28	0.28	0.29	0.283333
0.58	0.58	0.58	0.58
1.16	1.19	1.19	1.18
2.33	2.34	2.36	2.343333
4.66	4.77	4.68	4.703333
9.37	9.51	9.45	9.443333
18.55	18.74	18.8	18.69667
32.64	33.52	33.59	33.25
62.33	63.28	63.21	62.94
117.56	118.1	119.45	118.37
208.91	208.75	211.46	209.7067
343.68	346	344.2	344.6267
566.43	571.42	573.02	570.29
820.92	830.22	836.54	829.2267
1067.94	1082.73	1090.92	1080.53
1301.46	1323.99	1331.14	1318.863
1669.46	1673.75	1711.65	1684.953
2181.45	2141.16	2220.02	2180.877
2288.37	2190.91	2230.62	2236.633
2474.82	2350.38	2384.77	2403.323
2331.72	2196.98	2207.38	2245.36
2172.93	2130.91	2155.78	2153.207
2137.25	2104.05	2133.53	2124.943

Bandwidth UPC Ring 1x8

Reading 1	Reading 2	Reading 3	Average
0.041	0.043	0.042	0.042
0.085	0.086	0.082	0.084333
0.173	0.171	0.171	0.171667
0.34	0.345	0.342	0.342333
0.681	0.69	0.675	0.682
1.657	1.62	1.645	1.640667
2.946	2.543	3.065	2.851333
6.062	5.985	6.067	6.038
12.119	12.142	12.076	12.11233
24.215	24.11	23.957	24.094
45.868	46.092	45.763	45.90767
84.992	85.293	70.269	80.18467
143.845	144.219	143.619	143.8943

228.595	229.387	225.303	227.7617
313.27	311.298	311.972	312.18
392.743	388.402	396.317	392.4873
427.711	431.271	433.722	430.9013
469.624	471.437	470.862	470.641
497.703	494.689	502.987	498.4597
509.863	508.272	506.629	508.2547
541.283	538.766	542.671	540.9067
557.309	563.835	563.639	561.5943
576.273	576.727	577.784	576.928

Bandwidth UPC Ring 2x4

Reading 1	Reading 2	Reading 3	Average
0.03	0.03	0.03	0.03
0.062	0.061	0.062	0.061667
0.123	0.122	0.123	0.122667
0.248	0.245	0.248	0.247
0.495	0.489	0.496	0.493333
1.128	1.117	1.126	1.123667
2.075	2.059	2.079	2.071
4.098	4.056	4.117	4.090333
8.112	8.031	8.109	8.084
15.781	15.601	15.796	15.726
29.926	29.91	29.76	29.86533
52.646	52.755	53.093	52.83133
98.435	98.125	98.636	98.39867
158.433	157.512	157.164	157.703
233.669	233.752	233.432	233.6177
316.027	315.464	316.065	315.852
357.663	354.951	354.443	355.6857
415.459	410.275	413.064	412.9327
444.5	437.675	447.766	443.3137
458.286	454.49	464.471	459.0823
481.541	478.959	479.657	480.0523
503.35	499.317	499.241	500.636
512.986	510.414	510.527	511.309

Data for Graph 7

Delay MPI Ring 1x8

Reading 1	Reading 2	Reading 3	Average
2.2	2.3	2.2	2.233333
2.2	2.3	2.1	2.2
2.1	2.3	2.2	2.2
2.2	2.3	2.2	2.233333
2.2	2.3	2.1	2.2
2.1	2.3	2.2	2.2

2.2	2.3	2.1	2.2
2.3	2.4	2.3	2.333333
2.4	2.4	2.3	2.366667
2.6	2.6	2.5	2.566667
2.8	2.9	2.8	2.833333
3.2	3.4	3.3	3.3
4	4.2	4.1	4.1
6	5.9	5.7	5.866667
8.9	9.3	9.1	9.1
17.7	15.7	15.7	16.36667
28.1	27.8	24.7	26.86667
48.9	48.2	53.1	50.06667
98.3	91.7	98.2	96.06667
183.4	176.5	189.5	183.1333
412.2	417.1	409	412.7667
838.1	852.5	855.4	848.6667
1711.7	1723.7	1726.6	1720.667

Delay MPI Ring 2x4

Reading 1	Reading 2	Reading 3	Average
3.6	3.6	3.4	3.533333
3.4	3.4	3.5	3.433333
3.5	3.4	3.4	3.433333
3.4	3.4	3.4	3.4
3.4	3.4	3.4	3.4
3.4	3.4	3.4	3.4
3.4	3.4	3.4	3.4
3.9	3.8	3.8	3.833333
4.1	4	4	4.033333
4.4	4.3	4.3	4.333333
4.9	4.9	4.8	4.866667
6	5.9	6	5.966667
7.2	7.2	7.1	7.166667
10	9.9	9.8	9.9
15.3	15.1	15	15.13333
25.2	24.7	24.6	24.83333
39.3	39.2	38.3	38.93333
60.1	61.2	59	60.1
114.6	119.7	117.5	117.2667
211.8	223.1	219.8	218.2333
449.7	477.3	475	467.3333
965.1	984.2	972.8	974.0333
1962.5	1993.4	1965.9	1973.933

Delay UPC Ring 1x8

Reading 1	Reading 2	Reading 3	Average
24.2	23.3	23.7	23.73333

23.6	23.2	24.4	23.73333
23.2	23.4	23.4	23.33333
23.5	23.2	23.4	23.36667
23.5	23.2	23.7	23.46667
19.3	19.8	19.5	19.53333
21.7	25.2	20.9	22.6
21.1	21.4	21.1	21.2
21.1	21.1	21.2	21.13333
21.1	21.2	21.4	21.23333
22.3	22.2	22.4	22.3
24.1	24	29.1	25.73333
28	28	28	28
35	35	36	35.33333
52	52	52	52
83	84	82	83
153	152	151	152
279	278	278	278.3333
526	529	521	525.3333
1028	1031	1034	1031
1937	1946	1932	1938.333
3763	3719	3720	3734
7278	7272	7259	7269.667

Delay UPC Ring 2x4

Reading 1	Reading 2	Reading 3	Average
33.3	32.9	33.1	33.1
32.5	32.9	32.5	32.63333
32.5	32.8	32.4	32.56667
32.3	32.6	32.3	32.4
32.3	32.7	32.3	32.43333
28.4	28.6	28.4	28.46667
30.8	31.1	30.8	30.9
31.2	31.6	31.1	31.3
31.6	31.9	31.6	31.7
32.4	32.8	32.4	32.53333
34.2	34.2	34.4	34.26667
38.9	38.8	38.6	38.76667
41.6	41.7	41.5	41.6
51	52	52	51.66667
70	70	70	70
103	103	103	103
183	184	184	183.6667
315	319	317	317
589	598	585	590.6667
1144	1153	1128	1141.667
2177	2189	2186	2184
4166	4200	4200	4188.667
8176	8217	8215	8202.667

Data for Graph 8

Procs	Average time for 5 runs
2	0.189281
3	0.187187
4	0.174884
5	0.180151
6	0.149497
7	0.143504
8	0.1309224