Final Report (CS310)

Project Title:	Practical Study of Parallel Algorithms
Author:	Kevin Gordon (csuki@dcs.warwick.ac.uk, 9813328)
Project Supervisor:	Dr. Alexandre Tiskin (tiskin@dcs.warwick.ac.uk)
Project Website:	http://www.dcs.warwick.ac.uk/~csuki/
Course:	CS310 Computer Science Project
Year of Study:	2001-2002
Document Version:	0.18 FINAL DRAFT

Abstract

This project gives an introduction to the field of parallel computing and the parallel computing model: BSP (Bulk Synchronous Parallelism). The aim of this project was to carry out the implementation and analysis of two BSP problems. The parallel broadcast problem and parallel matrix multiplication were implemented, several experiments were run, and the results were analysed. Analysis of the results revealed that the experimental results didn't always match what the BSP model predicted, and this was highly dependent on both the type of architecture running the code and the optimisation of the code.

Keywords: BSP, Parallel Computing, Matrix Multiplication, BSP-lib, C, performance

Change Log

Version	Date	Description
0.1	11/04/02	Draft
0.10	25/04/02	Draft
0.18	1/05/02	Final Draft

Contents

Introduction

Parallel computing is a field in which computer software is run in parallel, using multiple processors. Using the BSP model (model of parallel programming) this paper discusses and analyses algorithms used in parallel computing.

4.1 Reasons for choosing the field of parallel computing

I had little previous knowledge of parallel computing before I began this project. My knowledge was very much a Hollywood tinted eyed view of what parallel computing was like. I found the topic, however, quite intriguing; because I was aware of the benefits that parallel computing could give in speeding up computations. I found the concurrency course that I had completed in the 2nd year of this course very interesting. It introduced the concepts of having separate processors working together on a problem, and how to get them to working together. However, this was purely on a single computer with virtual processors and the idea of taking programs and running them over several computers seemed very interesting. I had also heard of work under taken by a previous 3rd year student who took a sequential neural network and coded it in parallel. Experimentally this resulted in measurable speed increases.

During the course I've seen a lot of mathematical analysis of computing problems, but I've never really seen any of the code implemented and demonstrated. As such I was keen to do some experiments and compare the results to the mathematical analysis.

The example that caught my imagination was the 'Sieve of Eratosthenes' algorithm, for calculating prime numbers: You have several processors, the generator, and collector and sieve processors. The generator generates numbers increasing by one,

which it passes to the sieve. The first number each sieve receives is prime, each subsequent number it receives is tested as prime, if it can be divided by the number stored its thrown away, otherwise its passed on to the next sieve. Once complete, the number stored within each sieve is passed on to the collector. Each Sieve(i) dumps its first input to collect, and for i<N subsequently passes on to sieve(i+1) all non multiples of that number. Conceptually it is very interesting having discrete processors all working together in parallel on a particular problem.

4.2 Sequential computing versus parallel computing

An example of Sequential computing could be someone building a wall; it may take them 10 days to complete the task working on there own. However if you had 10 people working on the wall at the same time, and split the work up evenly, potentially you could complete the wall in a single day. This raises the interesting questions about how to organise the team of people, how to partition and share the work evenly, and how to manage them so they all work together.

In the context of this coursework I am using BSP-lib to manage the processes with the partitioning of the work and organisation of the processors is up to the coder. The structure of the BSP makes this process easier. Then of course you can have specialisation where different processes are performing particular jobs, and this carries on to the difficulty of designing parallel code. Also, that the implementation doesn't directly allow a speed up say divide the problem by 10 processors and you get a multiple of 10 speed up. Among others, these are issues/themes that were analysed.

4.3 Why parallel computing?

Sequential computing is limited by the hardware we have. There is some speed up to be gained from software. By improving the code and methodologies for coding you can increase the performance. However, none of these give huge leaps in performance. Thus, the performance of sequential computing is largely limited by the speed of the hardware we currently have. Of course simplicity in sequential computing has its advantages. Parallel computing offers another dimension to this that is not limited by the speed of a single processor, rather the design of the problem and the number of processors over which the problem is shared. This gives huge performance benefits in performance, without requiring cutting edge technology.

4.4 Aims and Objectives

"One of the promising trends in parallel computing is the BSP model. Many interesting algorithms have been designed for this model, but few of them implemented or studied from the practical point of view" – Dr. A. Tiskin.

The above quote gives the main objective for this project; that is to implement and study some interesting algorithms from a practical point of view. Firstly I wanted to extend my knowledge of parallel computing, secondly I wanted to learn how to run parallel software, and thirdly I wanted to learn how to write parallel code.

The main goal of this project was to learn something new about BSP algorithms, by investigating some of these BSP algorithms I intended to discover were the mathematical predictions would hold experimentally.

An aim of this project was to have as much information on the success and failures in order to allow another student to use it as a resource; to start a project on BSP or even continue the project from where I left off. During my research I looked at another project, which covered a similar area. I found the project inaccessible, hard to understand the tangible results achieved, and I did not understand the code that had been implemented. As such my aim is to make my project report as clear as possible so another student could pick it up and take it further.

4.5 What I did

Firstly I researched the area of BSP, initially reading about the mathematical model and then learning about how to use the BSP-lib software library. A lot of this material can be found in the background section. There is also some material on using the BSP-lib, which can be found in the appendices.

Next, I looked at the parallel broadcast problem. This is the problem of how to broadcast data between all the processors in the optimal time. There are two main algorithms that achieve this; the question is which one is optimal. I implemented the algorithms and carried out experiments to answer this question. I have analysed my results, and in this report you can find out the answer to this problem for the different architectures I looked at.

Finally I looked at parallel matrix multiplication, of which, there are numerous algorithms for parallel multiplication, I set out to implement some and find out which was the best. Only one parallel matrix multiplication algorithm was implemented, the results show that it performs worse than sequential matrix multiplication on the architecture tested on. The analysis section gives more details.

4.6 Structure of this report

The project report is split into three main sections; background information, the parallel broadcast problem and parallel matrix multiplication.

All the background information that I spent the first part of the project researching can be found in the background section. This gives an introduction to the BSP model, the mathematical model behind it, and the BSP software implementation that I used for this project. It also contains information about the architectures of the computers that I used for this project.

The second section contains the work I did on the parallel broadcast problem. It gives the background information about the problem. Included is an explanation of how the algorithms work, how they were implemented and how the code works. In the experimental method section I explain which experiments conducted, and how I went about setting up the experiments and capturing the results. The results and analysis can be found in the appropriate section.

The third main part of the project contains the work carried out on parallel matrix multiplication. It is structured similarly to the parallel broadcast section. Firstly I cover the background material. Then I explain what experiments were carried out. Finally the results and analysis for matrix multiplication are included.

4.7 Report Organisation

I suggest that the report should be read sequentially, the background information on BSP should be digested before reading the rest of the report. To understand the results and analysis of the two problems tackled I suggest the background information for each problem is read first.

Background

This section summarises the background material concerning the BSP model, BSP-lib, and the computer architectures that I used for this coursework.

5.1 What is the BSP Model?

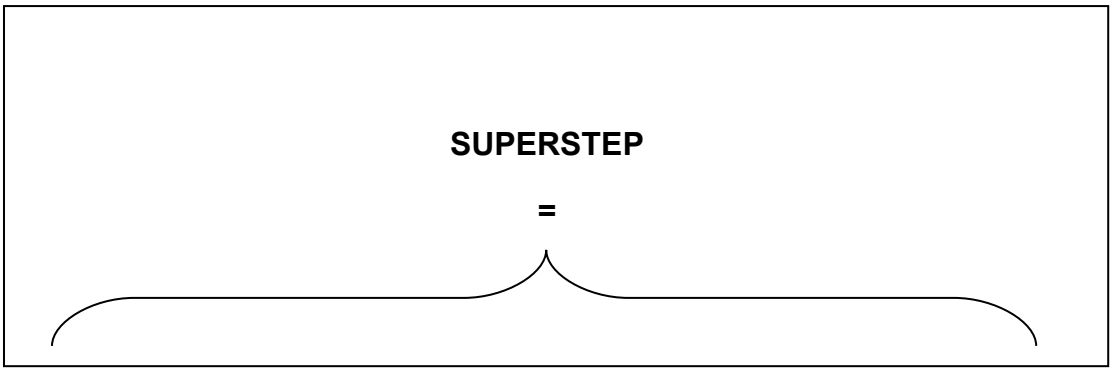
BSP stands for Bulk Synchronous Parallelism. It is a model or paradigm for parallel computing. There are a number of different models for parallel computing including blackboard, phone call/handshake, and e-mail type models. BSP is an example of an e-mail type model.

The blackboard model is where anyone can read or write on the blackboard, representing a global memory. The phone call model is a handshake type model; for example a person would call someone on the telephone. They then wait for the other person to receive the call and pick up the receiver, once done they can pass information. In this way a processor sends a message to another processor requesting a read or write, and then waits for that processor to respond.

The e-mail model can be thought of as a contrived system whereby workers are allowed to send and receive messages from 9:00 to 10:00. They then continue with there work, and send and receive e-mails the following day between 9:00 and 10:00. This is similar to how the BSP model works. So each process works independently, then at a set time the work stops and messages are exchanged between processors. Once the information is exchange work continues.

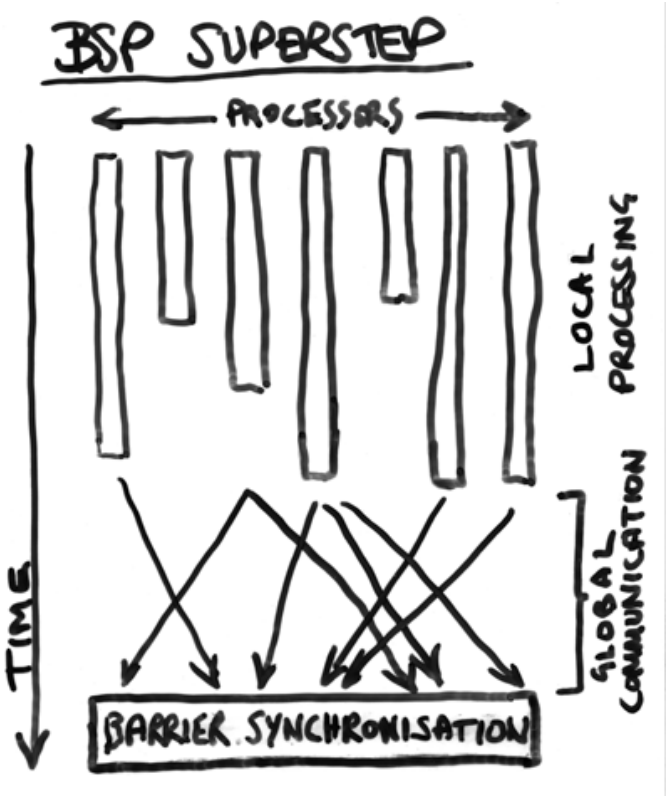
In BSP work is partitioned as the e-mail model suggests. These partitions are called super-steps, a super-step being the duration of time in which some amount of processing occurs locally, and then global communication and synchronization occurs.

5.2 The BSP Super-step



PROCESSING + COMMUNICATION + BARRIER SYNCHRONISATION

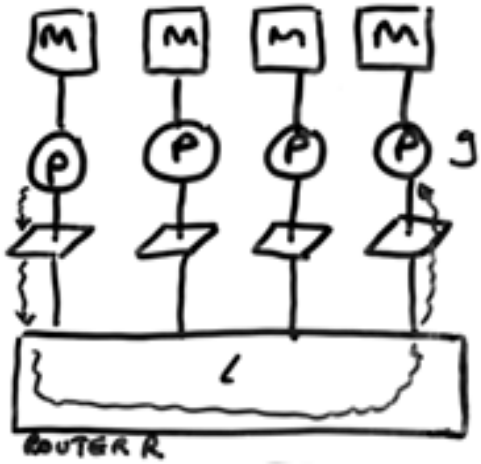
A super-step is made up of the local processing, global communication and barrier synchronization. During the processing stage each process processes data stored locally, and is unable to access data from any other processor. During communication processing stops and movement of data occurs; each processor sends and receives messages alternatively handling read and write requests.



As you can see from the above diagram of a BSP super-step, local processing is followed by global communication and finally barrier synchronization. It is important to note that read/write or send/receive calls are made throughout the local processing period, it is only during global communication in which data is transferred. Also barrier synchronization performs the important function of making all transferred data available to local processors once the global communication is complete.

5.3 The BSP Computer and its mathematical model

Having this fixed structure of super-steps has negligible affect on the performance of parallel programs. The advantages include having a very simple mathematical model in which to analyse programs.



The diagram above gives the conceptual idea behind the BSP computer or the BSP architecture. The computer architecture used by BSP can be though of as a series of p processors "Nodes", each with local memory, connected by a network. The model is independent of the topology of the network, as such we do not have to concern ourselves with the network topology when making calculations, this cost is incorporated in the constants for the equation. The cost model is based on two main parameters g and h.

p = number of processors

l = number of time steps for "Barrier Synchronisation" (latency)

g = permitivity of the communication network (gap).

$$g = \frac{\text{(Total number of local operations performed by all processors in 1 second)}}{\text{(Total number of words of data delivered by the communication network. In one second – in a situation of continuous traffic)}}$$

s = time of the super-step

So l and g are the parameters dependent on the architecture that you are using, the speed of the processors and the network are both factored in.

w = max(number of operations on any one processor during S)

h_s = max(number of messages sent in S by any processor)

h_r = max(number of messages received in S by any other processor)

$$s = w + \max\{ h_s, h_r \} . g + l$$

The total time required for a BSP computation is equal to the sum of each of the super-steps. An h relation is the communication pattern in which each processor sends or receives maximum of h messages.

S = sum of all the super-steps

$$S = W + H . g + l$$

W is the sum of all the local computations; H is the sum of all the h-relations. W, H and S can be thought of as functions of the network and the processors, f (n, p). The above equation for calculating the BSP computation time makes calculation as simple as for sequential programs. BSP is to parallel as Von Neumann's model was to sequential programming. The key advantageous of the BSP paradigm is that it gives us a stable framework in order to develop software, which is portable, and architecture independent.

5.4 BSP-lib, the BSP software library

The BSP-lib, BSP Programming library, is the implementation of BSP that I used for this project. Supplied by Oxford Parallel computing, it was the software library suggested by the project supervisor. It was convenient as it was already set up and

working on the architectures that I carried out testing on. Additionally my project supervisor provided support for the BSP-lib software library.

BSP-lib is easy to use, with some helpful material and a full on-line manual (see references). BSP programs can be written using the languages C or Fortran, for this project I chose to use C. Firstly because I had some prior knowledge concerning using C and secondly because there was a lot more tutorials and example information for BSP coded in C.

The code is written as Single Processor Multiple Data (SPMD). This means one block of code is written, which is run simultaneously on each processor you start.

BSP-lib provides two methods for communication, Bulk Synchronous Message Passing (BSMP) and Direct Remote Memory Access (DRMA). BSMP involves sending messages between processors and having the concept of a queue for each processor to receive messages into. DRMA allows each processor to carry out a get or put action on another processors memory, this is done through the process of registration, whereby each processor calls BSP registration primitive, which creates a global registration and allows processors to reference memory in other processors.

5.5 Computer Architectures used to run the parallel code

The main architecture used was a cluster of AMD K6-2 PCs and individual SUN workstations in the DCS department at the university of Warwick. BSP-lib allows you run code on a single machine simulating multiple processors, i.e. running concurrently on a single machine, which was useful for testing code initially before I learned how to run the code parallel. The system specification of the PCs was AMD K6-2, most

266MHz processors running at 281MHz, with 64MB of memory, running SunOS 5.8. The BSP parameters are as follows. These are the official parameters that come in the BSP-lib parameter database, limited to only 2 processors and 4 processors:

Cluster of Solaris Workstations

s (Mflops/s)	**P (no procs)**	**l (flops)**	**g (flops/word)**
17.421	2	83007.0	176.09
18.128	4	164505.0	237.21

* Source the BSP-lib parameter database

Running on a single Sun workstation gives the following values for the parameters:

Single Workstation

s (Mflops/s)	**P (no procs)**	**l (flops)**	**g (flops/word)**
39.650	2	689.6	15.8
39.600	3	1730.5	17.1
37.481	4	33064.9	1117.85

Cluster of Pentium Pro PCs (266Mhz Pentium Pros with 512K cache connected by 10Mbit Ethernet)

s (Mflops/s)	**P (no procs)**	**l (flops)**	**g (flops/word)**
61	1	85	1
61	2	52745	484.5
61	4	139981	1128.5
61	8	539159	1994.1
61	10	826054	2436.3
61	16	2884273	3614.6

Interestingly a dedicated parallel architecture is not required, as such the code can be run other the cluster of workstations that are in the DCS lab.

There were 70 lab workstations in total. BSP-lib was configured to allow 16 processors initially; this was reconfigured to 32 at a later date. All the communications across this network are at the same speed. The network comprises of 3 switches, one master and two slave switches, together they work as one large switch. Each system is plugged into one of the three switches which connect to a sun Ultra-5 which then plugs into the central network, connections are 100mb.

I used the Oscar supercomputer at Oxford University. I submitted my code and test scripts to my project supervisor who then submitted this to the Oscar computer to be run. The computer is a Silicon Graphics Cray Origin 2000 parallel computer. Tested with up to 64 processors. It has in fact got 96 processors in total. System specification includes processing power of 96 MIPS, 195 MHz, R10000 processors, each with 256 Mbytes of RAM. The BSP parameters for Oscar (an origin 2000 supercomputer);

Origin 2000

s (Mflops/s)	**p (no procs)**	**l (flops)**	**g (flops/word)**
100.7	1	286	1.36
	2	804	8.26
	3	1313	8.36
	4	1789	10.24
	5	2474	11.06

	6	2963	12.25
	7	3867	14.28

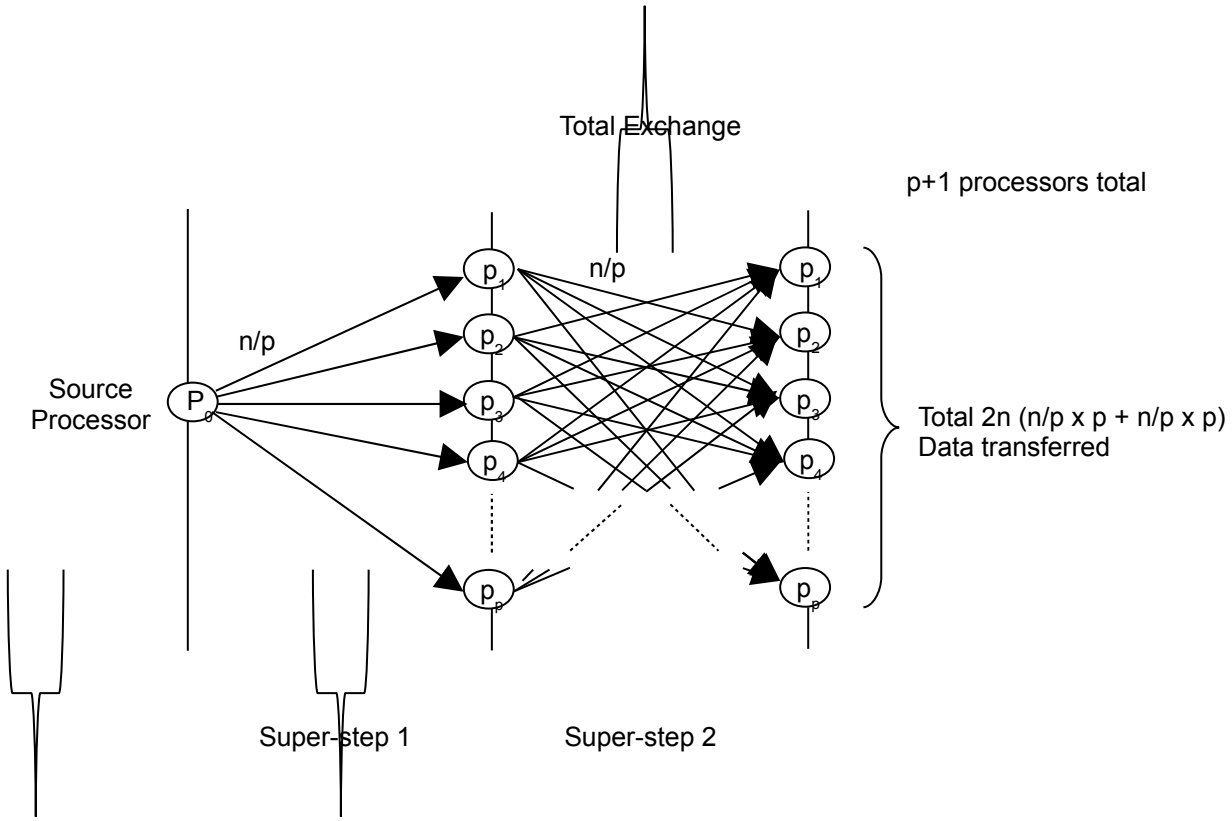
These values come from the parameter data published on the Oxford BSP-lib website (http://www.bsp-worldwide.org/implmnts/oxtool/params.html - origin)

Parallel Broadcast

Theory

During the first part of my project I looked at the parallel broadcast problem. The parallel broadcast problem concerns the transfer of data from one process to all other processes. Under certain circumstances a processor wishes to send data to all other processors taking part in the parallel computation. To make parallel applications as efficient as possible we would seek a way to reduce the amount of time it takes to send this amount of data. Potential we are sending n amount of data to p processors, which in total gives n x p data transferred. Of course this data transfer does not have to occur sequential, there are other algorithms then just sending out n x p data n of which is received by each processor; this method is called the direct method. Alternatively we can spread the cost over the several super-steps, reduce the amount of data sent by anyone processor, these two main algorithms are 2-Phase and Tree. 2-Phase is a two-step process in the first super step the source processor sends a fraction of the data to each processor. A total exchange then occurs whereby each processor then sends its share of the data to all other processors. In the tree method, being a binary tree, at each step data is sent to another processor, forming a tree, so on each super-step a tree of nodes is built up which are sending data to further processors. Each of these methods was implemented, with experiments carried out to seek out which was the optimal. As I will show the mathematical analysis suggests that 2-Phase is more efficient, the experiments carried out looked to prove or disprove this.

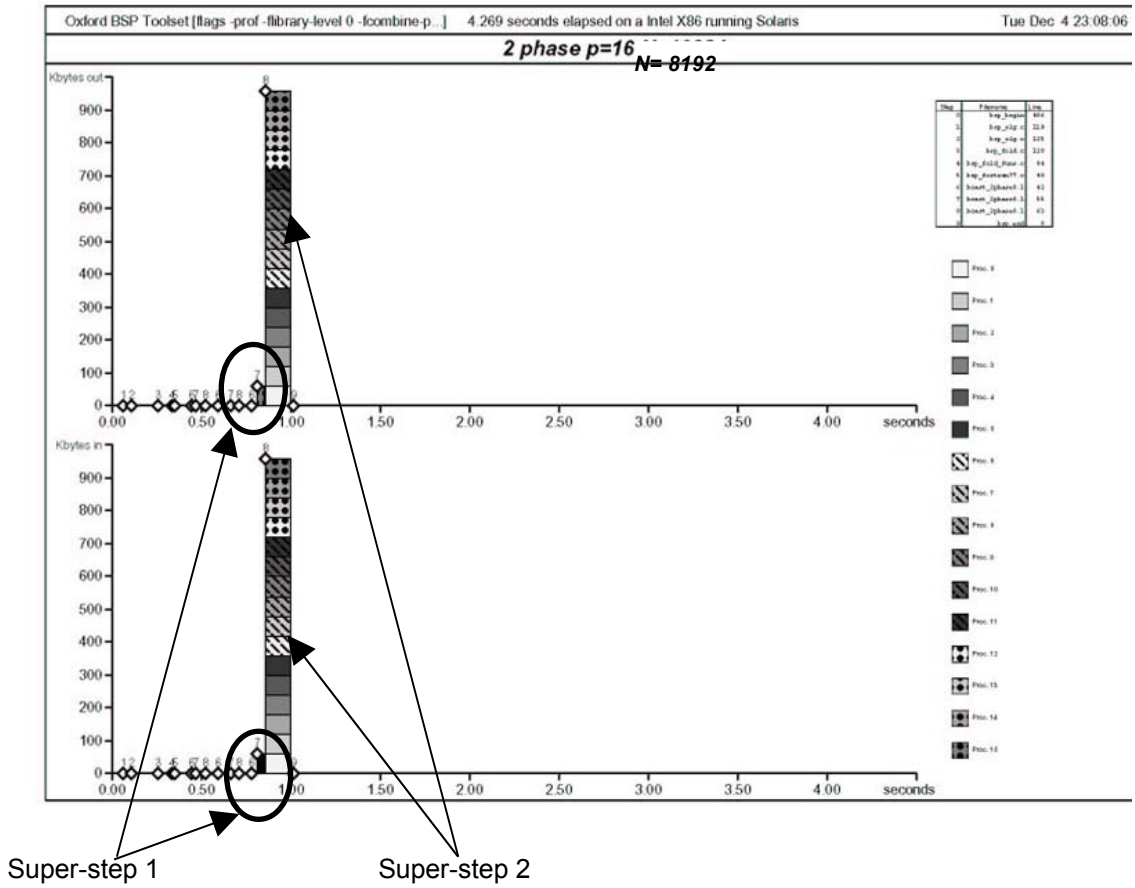
2-Phase Broadcast



The 2-Phase method takes two super-steps. During the first super-step the source processor divides up the data to be broadcast, and sends an equal amount to each processor, that is each processor then receives n/p amount of data. At the beginning of the second step the data to be broadcast is shared equally amongst all the processors, a total exchange then occurs. A total exchange is when each processor sends its data to all other processors. So each processor sends its share of the data (n/p) to p (to be more accurate p-1) other processors. At the end of the second super step each processor as receives p lots of n/p data, thus the broadcast is complete.

Communication cost	=	2Ng	(2n)
Synchronisation cost	=	2l	(2)

The communication cost of the 2-Phase algorithm is 2Ng, N being the total amount of data transferred, and g being the parameter g. I use the simplification 2n. During super-step 1 the source processor sends p lots of n/p amounts of data. The maximum amount of data received by any processor is n/p, but the maximum amount of data sent by any processor is p x n/p which equals n. Thus the h-relation is of order n. During the second super-step, a total exchange occurs, whereby each processor now sends out its n/p piece of data to all other processors, that is n/p x p which means the h-relation is of order n. Thus the communication cost is 2n (n from 1st super-step + n from 2nd super step). The synchronisation cost is 2l, as there are only two super-steps.



The BSP profile graph above (see appendices for how to read these graphs) shows the profile for the 2-Phase algorithm, working over 16 processors with 8,192 integers to be broadcast (64KB). During super-step 1, unfortunately its difficult to see on this diagram, but there is a small bar, a single processor sends out approximately 4KB of data to each processor (so the bar shows a total of 64KB data communicated). During the second super step each processor then sends out its 4KB of data to all other processors, so in total each processor sends out 64KB of data, and receives 64KB of data during the second phase. This second super step is called a total exchange. The broadcast is then complete.

The code is included in the appendix. The code is based on "bcast_ex4", part of the BSP Tutorial by Bill McColl and Jonathan Hill, oldww.comlab.ox.ac.uk/oucl/oxpara/courses/tutorials/

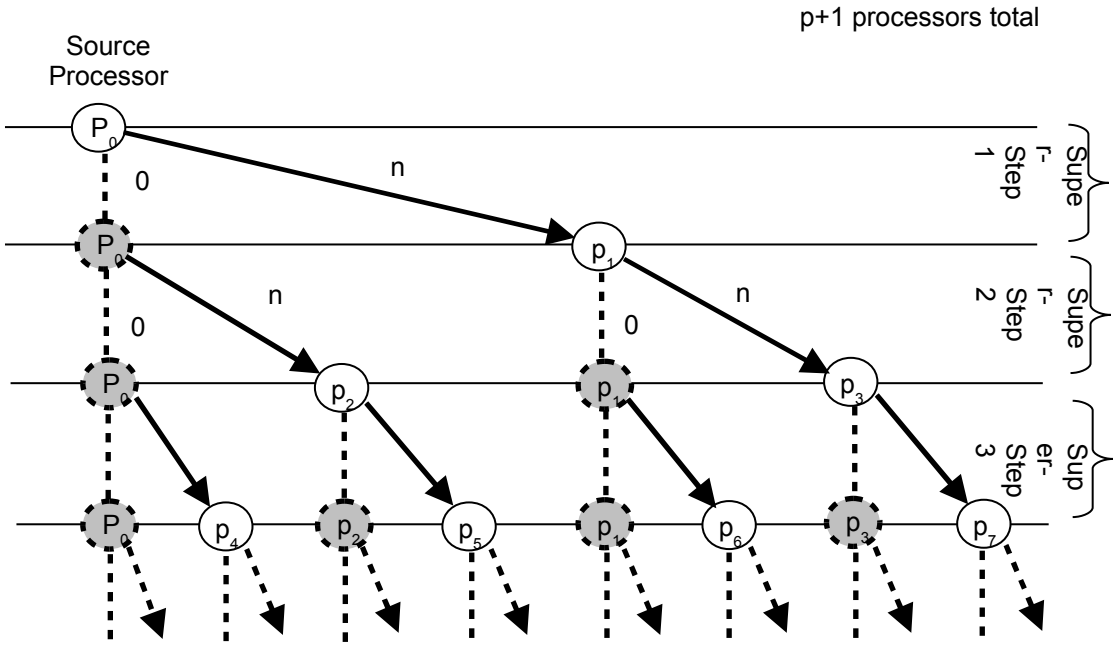
The parallel code for SPMD mode is split out into the start_spmd() function. The code to perform the broadcast can be found in the broadcast_ints() function, just before this is called bsp_dtime() is called to start the timing of the broadcast. This transfers a 1D array of integers from a specified source processor to all other processors. Firstly the destination array (on each processor) is registered. Then During the first main super-step (there are several other super-steps but they insignificant as they only involve registration/deregistration of data areas) the source processor does a bsp_put() on each of the processors with part of the array to be broadcast. A put is made to each processor in turn, the data transferred is at the offset given by (n_over_p * i) * sizeof(int). n_over_p is n/p the size of the data to be transferred, i is the processor number, and sizeof(int) gives the offset in bytes. As a minor technicality if p is not a factor of n then the last processor will get an array of slightly smaller length. At this point bsp_sync() is called, and this is the end of the first super-step.

During the second super-step a total exchange occurs amongst the processors. All processors execute a for loop, which involves calling bsp_put() on each processor and putting there part of data in the array of the destination processor. The array offset of the destination array is calculated depending on the processor number, which is the same offset as the data in the source array. The bsp_sync() function is called and that is the end of the second super-step. The destination array is then deregistered.

Processor zero then outputs the duration of the broadcast by calling bsp_dtime() to get the duration of time from the last time bsp_dtime() was called.

Tree Broadcast (Binary Tree)

The Type of tree broadcast I looked at was binary tree, whereby the communication pattern forms a binary tree. As you can see below;

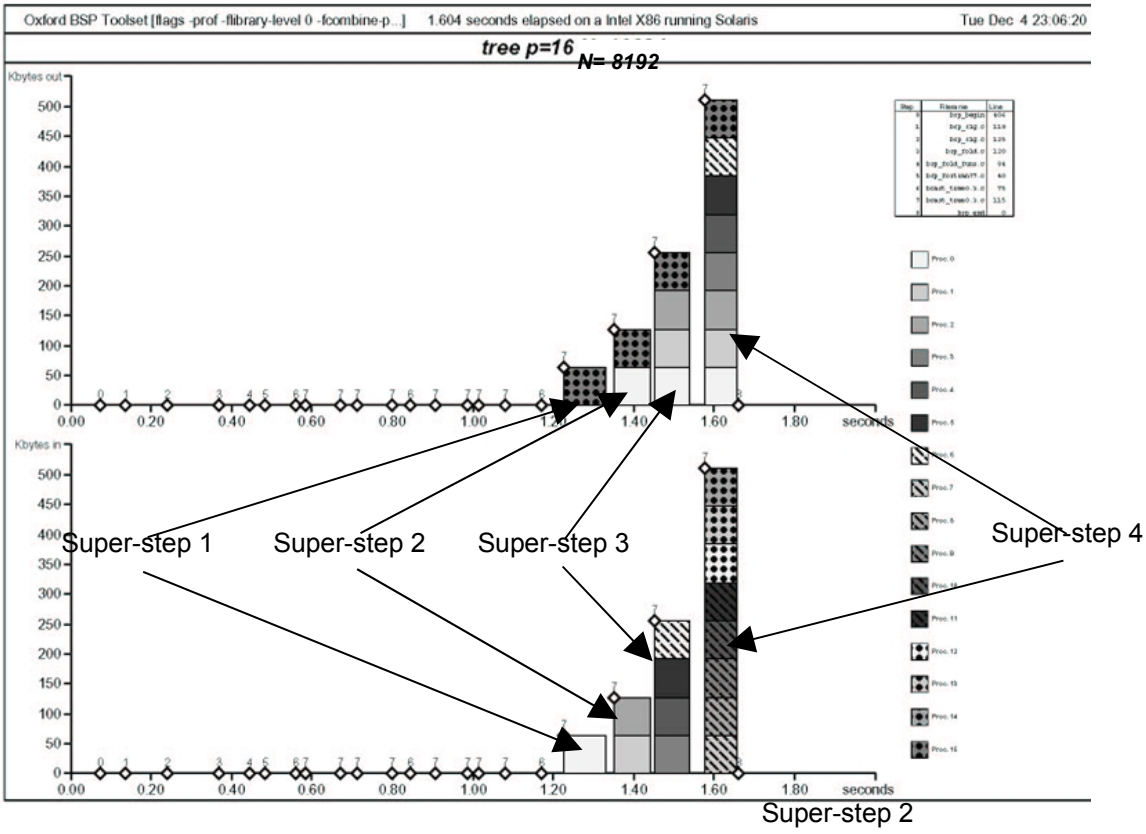


Conceptually the communication paths between the processors become a binary tree, as you can see from the above diagram. During the first super-step the source processor sends its data to another processor. These processors form the nodes of the tree, during the next super-step (super-step 2) both nodes then send data to a further two nodes, therefore there are now four nodes in the binary tree. At the next super-step (super-step 3), or next level of the tree each of the four nodes then sends to a further processor each, and there are now eight nodes in the binary tree. This continues depending on how many processors there are involved in the broadcast.

Note the nodes coloured grey with the dashes between, indicates they are involved in a particular super-step, but no data is being transferred to that processor.

Communication cost	=	(Log p) Ng	(nlog p)
Synchronisation cost	=	(log p)l	(log p)

The depth of the binary tree is proportional to logarithm of the number of processors. As such the number of super steps is calculated as log p. The communication costs are given by nlog p. This is because at each super-step a maximum of data size of n is transferred between any two processors. And because there is n data transferred per level of the binary tree, the total communication cost is the number of super steps multiplied by the amount of data per super-step (nlog p).



See the appendices for details on how to read BSP profile diagrams. The above BSP profile is for Tree broadcast with 16 processors involved in the broadcast of 8,192 integers (64Kbytes of data). During Super-step 1 processor 16 sends 64KB to Processor 0. P16 and P0 now have the 64KB of data, during super-step 2 P16 and P0 transfer 64KB of data to P1 and P2. At the start of super-step 3 there are now conceptually 4 nodes in this level of the binary tree, each of which sends 64KB of data to one other processor. In super-step 4 there are now conceptually 8 nodes in this level of the binary tree, each of which transmits the data to another processor. This works out quite nicely as we have a balanced binary tree.

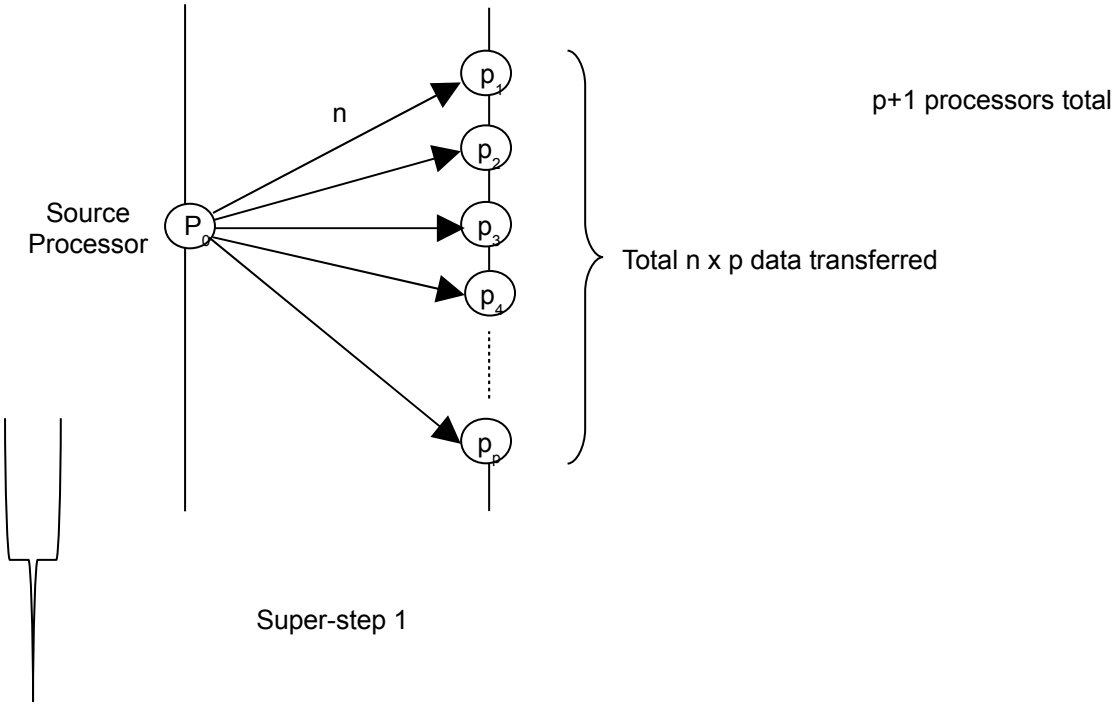
The code is based on the code for 2-phase broadcast. The difference being the broadcast_ints() function, and this code is based on and adapted from the code given in the paper "Broadcasting on the BSP model: Theory, Practice and Experience" by Alexandros V. Gerbessiotis.

The key thing to note is that at each super-step all processors that have received that data send the data onto another processor. Note the source processor can be any of the processors, so we use a temp_pid which gives the relative distance from the source pid. This is done using the expression ((pid-fromp)+nprocs)%nprocs), so to simplify the source processor can be thought of having pid 0, and then all subsequent processors having sequentially greater pids. The processors that have received the data are those with processor id less than what is called the ‘mask id’. This is just the number of processors that have received the data so far. Because it is a binary tree we can calculate the number of nodes at each level, which is 2^(depth of tree). And because each processor that has received data is a node at that depth of the tree the, therefore the process ids that have received data are less than the mask pid.

Each processor then sends the data to one other processor. (There is capability for n-ary type trees, but this is out of scope of my current discussion).

Direct Broadcast (1-Phase broadcast)

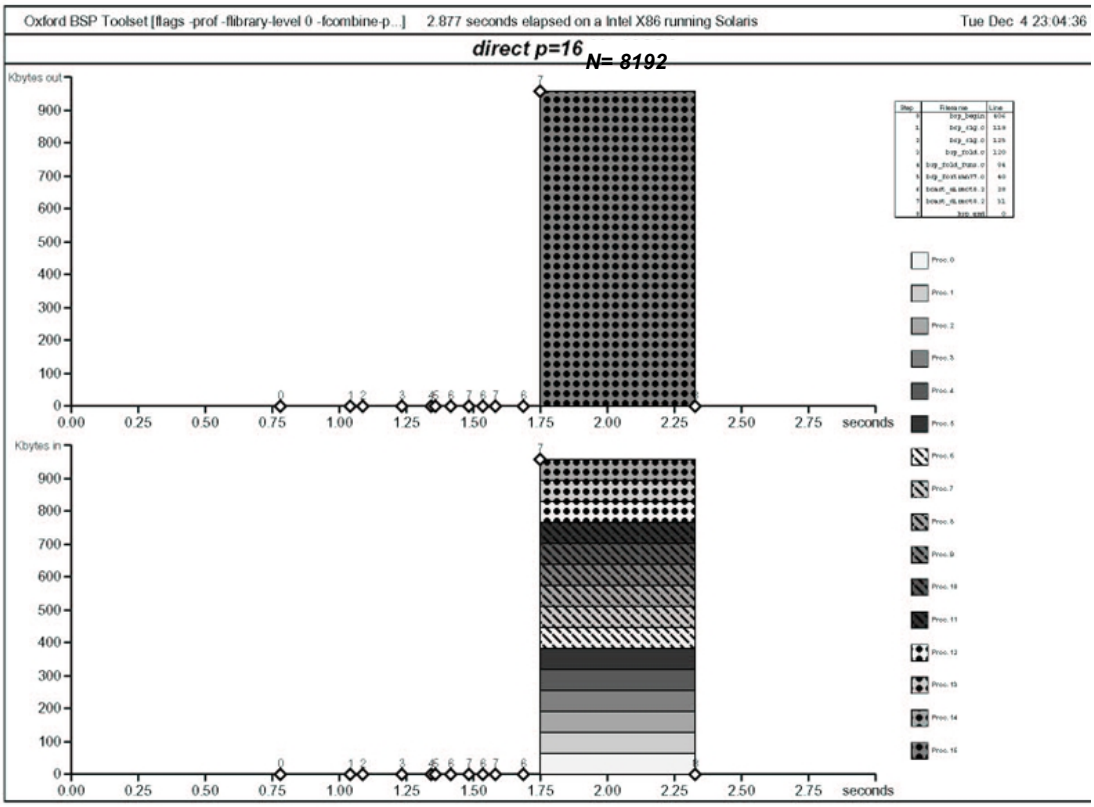
After looking at 2-Phase and tree, I then looked at the most basic broadcast method; namely direct broadcast, or 1-Phase broadcast. This is the naive method whereby the source processor sends all the data to all the processors. This occurs in 1 super-step and has n*p amount of data transferred. Thus theoretically it is an upper bound on the time that should be taken to compute the broadcast. It should be inefficient compared to the other methods, and thus can be used as a control.



As you can see the algorithm is a one step processor. The source processor sends the total n data to each processor. At the end of the 1st super step each processor has received n amount of data, thus in total n x p amount of data is transferred.

Communication cost	=	pNg	(np)
Synchronisation cost	=	l	(1)

The communication cost is given by the equation pNg, and the synchronisation cost given by l. As such communication is of order np, and synchronisation is order 1, as only one super-step occurs. So to send 10MB of data to 10 processors would cost a transfer of 100MB during one super-step.



See the appendices for an explanation of how to read the BSP profile graphs. What the above graph shows is the profile for the direct broadcast program. The experiment was run for 16 processors, with 8,192 integers being broadcast (64KB). Thus in total 15x(8192x8)/1024 = 960 Kilobytes was transferred in total, which you see from the top bar the source processor (processor 0), sends this data. Each of the other processors then receives 64KB of data, as you can see from the bar along the bottom, which is divided into 64KB seconds corresponding to the data received by each processor.

Notice that it appears data is only transferred to 15 processors, not the full 16 involved in the computation. This is because the source processor transfers data to itself, which just requires a local memory copy in the C code. As such it does not appear in the BSP profile as a data transfer.

The code, which can be found in the appendix, is based on the code for 2-phase broadcast. The difference being the function broadcast_ints(). Instead of the 2 super-step process, one super-step occurs, during this step the bsp_get() statement is called by each processor, to get the array from the source processor. Bsp_sync() is then called and the array is transferred to each of the processors.

Overall Equations

The overall costs;

Algorithm	**Communication**		**Synchronisation**	
	Equation	**Order**	**Equation**	**Order**
Direct (1-Phase)	pNg	(np)	l	(1)
Tree (Binary)	(Log p) Ng	(nlog p)	(log p)l	(log p)
2-Phase	2Ng	(2n)	2l	(2)

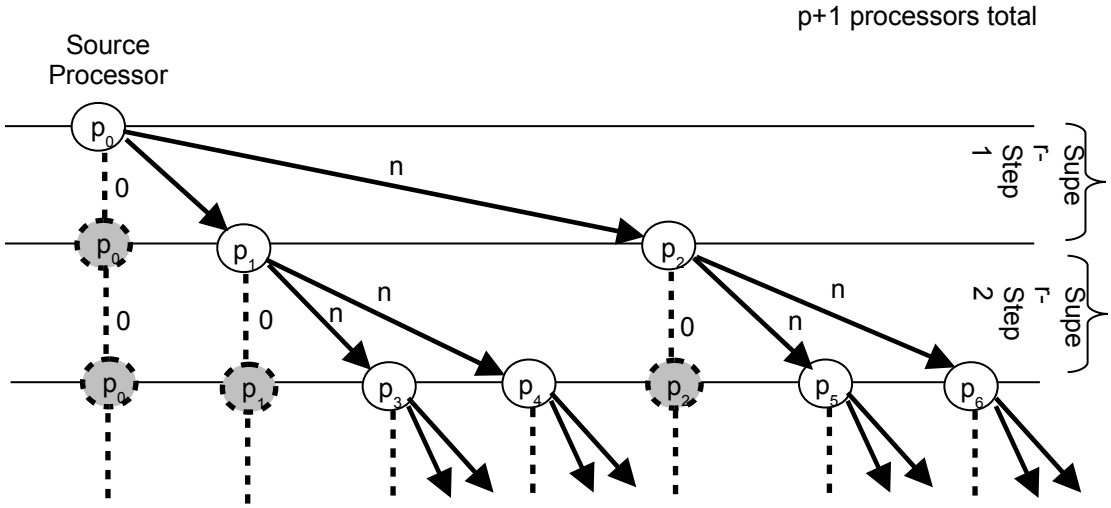
There are two claims made by the theoretical analysis:

1. When N is small ($N < 1 / (pg – 2g)$) use the one-stage broadcast (direct broadcast).
2. The 2-Phase broadcast is always better than the tree broadcast when $p > 2$ (it should be the same when p-2).

See method for the experiments carried out, and see the results for what experimental data I got.

Alternative Tree Broadcast

On setting up a repeat of the tests I believed that the tree method was working incorrectly. By studying the graphical profile I believed that the code was working incorrectly. This is because I miss-understood how the tree algorithm was working, I believed that each processor should be sending data to two other processors, and then those two processors send data to a further two processors each. This would mean that once a processor had sent the data it would become redundant. The mistake here is that I was forgetting that each processor stays as a node in the tree, and therefore for a binary tree each node only sends to one other processor. Sending to two processors would give you a ternary tree. However all processors that have sent data are then not used any further within the broadcast, so we have the algorithm;



This code has not yet been tested.

Internal Broadcast Primitive

I decided to also look at the internal broadcast primitive. This is built in to the BSP library. And I thought it would be a good idea to compare the code I am testing with the built in broadcast. I currently do not know by what method the BSP broadcast works, but would have expected it to be optimised for the procedure of broadcasting.

This code has not yet been tested.

Experimental Method

Composition of the timing data

The timing information given by the results needs to be explained further. The programs output a time from when the bsp_begin() has been called, and the timer commences when all the processors have started. The timer is then stopped before bsp_end() is called, thus the timing data outputted by the C code is purely the time taken for local processing, and communication and synchronisation costs. All start-up costs/shut down costs are not included in this timing information. This is why the Unix command time gives larger values for the timing, and the programs total running time is longer than the results shown. This is due to the start-up costs of the BSP programs. However these start-up costs do vary, and are not accurate, and it is not appropriate to include them in the total time for the programs. I have also chosen to leave them out of the overall results.

Preliminary tests and data-ranges

I Ran the experiments on Parallel broadcast 2-Phase and Tree on the DCS lab computers. These results can be seen in the results sections. The data sizes used were as follows;

Bytes in a Kilobyte	1024					
Kilobytes	32	64	128	256	512	1024
Bytes	32768	65536	131072	262144	524288	1048576
N (number of ints in array)	**8192**	**16384**	**32768**	**65536**	**131072**	**262144**

The choice was related to the experiments carried out in the study*****, to allow the results to be compared with that study. Firstly I wanted to get some results, to get some test data to see how the code was performing, and secondly I wanted a reasonable spread of data over a range of data sizes, the data sizes spreads over 3 orders of magnitude.

The number of processors used in the testing was constrained by the maximum number of processors that could be used on the DCS lab machines implementation of BSP-lib was 16 at this time. The second factor was that I was initially doing tests manually so didn't want to test over too large a range of processors. I chose 5, 10 and 16 processors to test on, which gave a reasonable range and would allow me to extrapolate. This gave me initial test values that gave me an idea of the different broadcasts relative performance, and also that they were working correctly. Some of my initial summary data can be seen:

n	**P**	**Tree (binary) /sec**	**2-Phase /sec**
1	2	0.00896	0.001090
		0.001006	0.001078
		0.001091	0.001146
		0.000733	0.001071
10,000,000	2	0.493	2.471
		0.506	2.449

So successfully got 2-phase and tree code running at this stage. The initial times showed the Tree algorithm to be faster than two-phase, and with greater data size it proved to be significantly faster. However a second test found the contrary that over a limited data range the 2-Phase broadcast was better:

n	**P**	**Tree (binary) /sec**	**2-Phase /sec**
3	8	0.091354	0.060914
		0.094823	
3,000	8	0.125569	0.099498
		0.114308	0.097929
		0.104811	0.123149
3	10	0.149614	
3	12	0.197964	0.099900
		0.167728	0.112238

		0.180177	0.082062

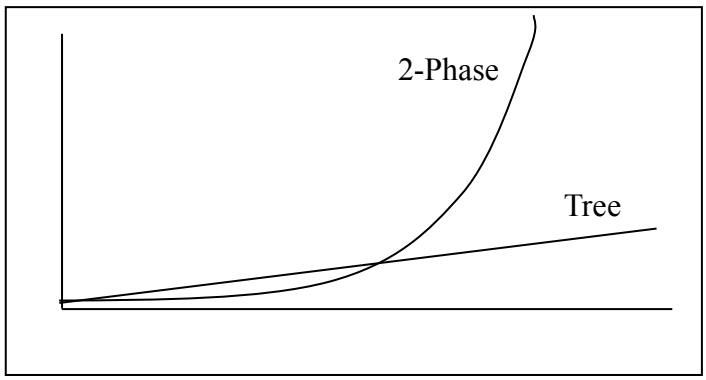
Shell scripts

Due to the large amount of data being generated, even over quite a small data range, and the duration of time required to carry out each one of the tests I developed some shell scripts that could run the tests automatically over a range of input values. Also it is difficult to draw any conclusions from these results, as there are few repetitions over a small range of data input, due to this limitation, I did not think they were statistically any good, hence why I set up the shell script to run many of repetitions.

The shell scripts can be seen in the appendix. These tests generated large output files of data, which required processing to get into a meaningful form, which can be seen in the results section. A full outline of how the data was processed can be found in the appendix. These scripts were later updated to work with the Oscar super-computer.

2-Phase versus Tree

The next tests I decided to run were based on the data ranges discussed above. Also I was aware that the code was outputting different amounts of data, as there was some debug information, I ran several tests to see if this was making a significant impact on the performance of the code. The tests proved it wasn't, but I stripped out the debug information and ran the code in non-verbose mode.



It was once I ran these tests I realised further investigation was required in order to verify the relationships within the data, and find appropriate explanations. On looking at the data the conclusion was that it diverged from what was predicted; in that the results showed parallel tree was better than 2-phase. It basically shows that 2-Phase is a lot worse than the tree method for large messages. This is in contrast to what the theory says, and also what the paper by Juuvlink and Rieping suggested. In this paper they suggest that BSP predicts that the 2-Phase algorithm is superior, there experiments show that for smaller data-sizes 2-phase is worse than tree, and for larger data sizes 2-Phase is better than tree. I've found 2-Phase better than tree but only for messages less than a certain size, the opposite of what this study found. This could be to do with the set-up of Unix workstations on the network. See results section for a discussion.

At this point I chose to investigate the direct broadcast method. As such I run tests over the same data range. You can see the results in the results section. I did not repeat the tests for the other two methods of broadcast. It was found that the direct broadcast method was better than the two supposedly optimum methods for parallel broadcast, however as the tests were carried out at different times they are not directly comparable.

Oscar Supercomputer testing

At this point the use of the Oscar supercomputer was made available to me. I run the tests for 2-phase and Tree. The results can be seen in the results section. The code did not need to be adapted, though the shell scripts had to be changed, see the appendix.

The tests for 2-Phase and Tree broadcast were run on the Oscar supercomputer this, was for a larger range of data values from 32KB to 65536KB (66 MB) of data.

Direct Broadcast Re-test

The direct broadcast test was re-run on the DCS computers. Firstly to act as a control for further tests, secondly to see what the performance was like relative to the previous tests, and finally to see the relationship between number of processors and the time to broadcast for a fixed data-size.

High Performance 2-Phase Broadcast

The final test carried out was high performance 2-Phase broadcast; the tests were run on the DCS computer. The aim of this test was to see whether use of buffered bsp primitives was causing the unexpected results in the experimental data. Secondly the test was run to get a more accurate breakdown of the timing of the program. The Bsp_timed() was used to time each of the super-steps in which the 2-phase broadcast was carried out, and separated out the initialisation code, and the extra super-steps for registration of variables. As such this test gives the timing information for the exact duration of the two super-steps in which broadcast is carried out.

Results And Analysis

Firstly I'm going to go through the two broadcast methods separately. I'll then make a comparison between the two types of broadcast. I'll ten go through the analysis of the test results for all the other tests performed. Please note that the raw data can be found in the appendix.

2-Phase versus Tree

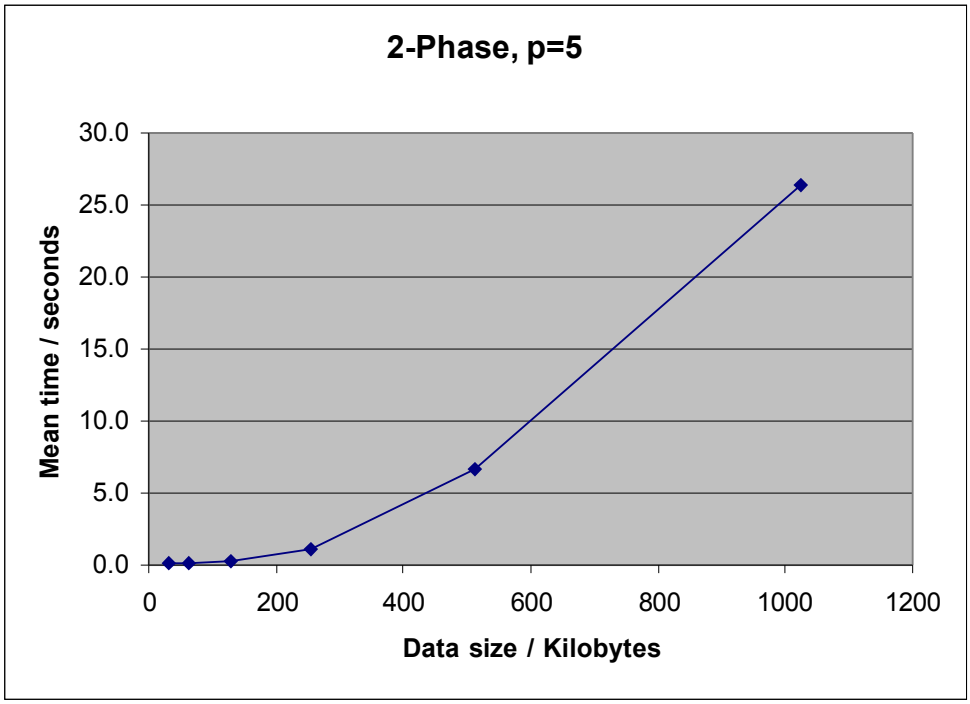
Firstly I'm going to look at the affect of changing the data sizes on the time to broadcast. I've included the data for the two broadcast methods as you can see below. This contains the experimental data, including the mean broadcast times, standard deviation, min and max, and results population – being the number of repetitions for that particular result.

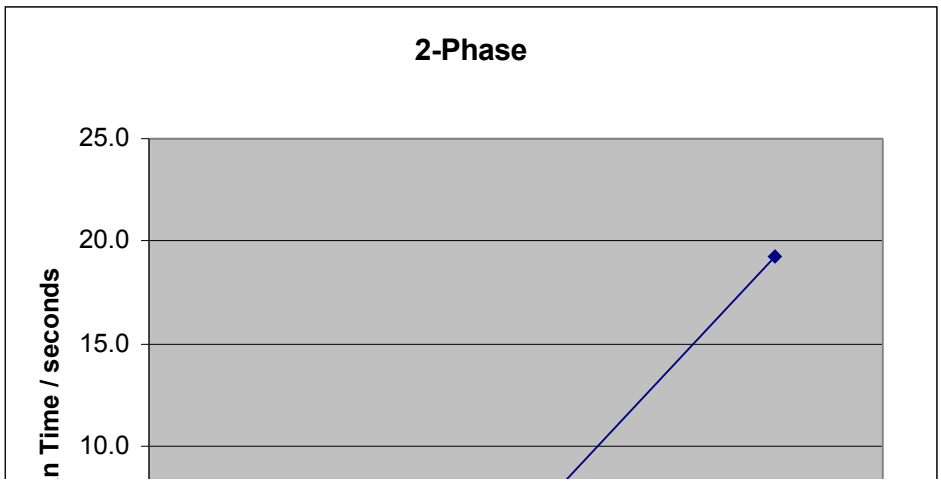
Analysis of 2-Phase broadcast

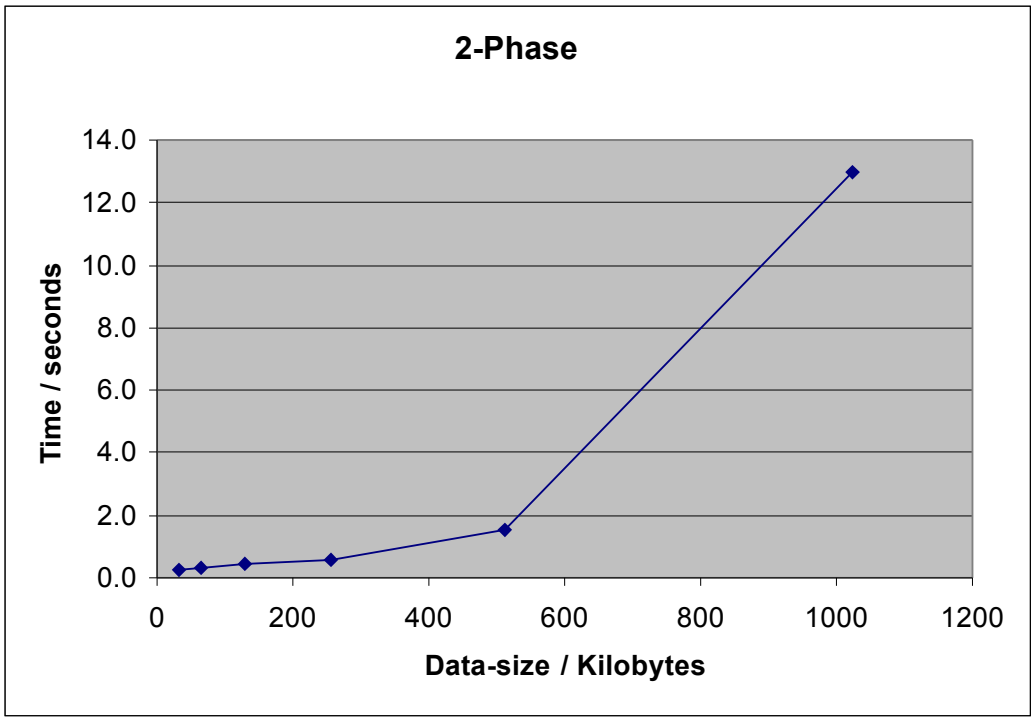
Broadcast Type:		2-Phase				
			Time /seconds			
Number of Processors	**Data size /Kilobytes**	**Results population**	**Mean**	**Standard Deviation**	**Min**	**Max**
5	32	6	0.085144	0.003240	0.082422	0.090693
5	64	6	0.127187	0.011166	0.112629	0.142551
5	128	7	0.215030	0.013613	0.192278	0.230880
5	256	6	1.073815	1.589247	0.376829	4.317352
5	512	4	6.679988	1.212657	5.776688	8.468453
5	1024	7	26.430414	3.000400	23.253664	31.087066
10	32	5	0.166666	0.019588	0.146870	0.196538
10	64	6	0.207706	0.021229	0.177019	0.228850
10	128	6	0.294307	0.018273	0.277004	0.322858
10	256	5	0.566967	0.072726	0.504733	0.685405
10	512	3	2.981274	1.236566	2.068619	4.388596
10	1024	3	19.298234	4.383064	15.381767	24.032654
16	32	5	0.275207	0.036164	0.217605	0.315697
16	64	4	0.333766	0.022228	0.309233	0.361478
16	128	5	0.458612	0.065447	0.398760	0.561652
16	256	3	0.600059	0.047819	0.567072	0.654900
16	512	4	1.543545	0.231198	1.286623	1.848100

16	1024	3	12.989134	1.944890	10.782883	14.455493

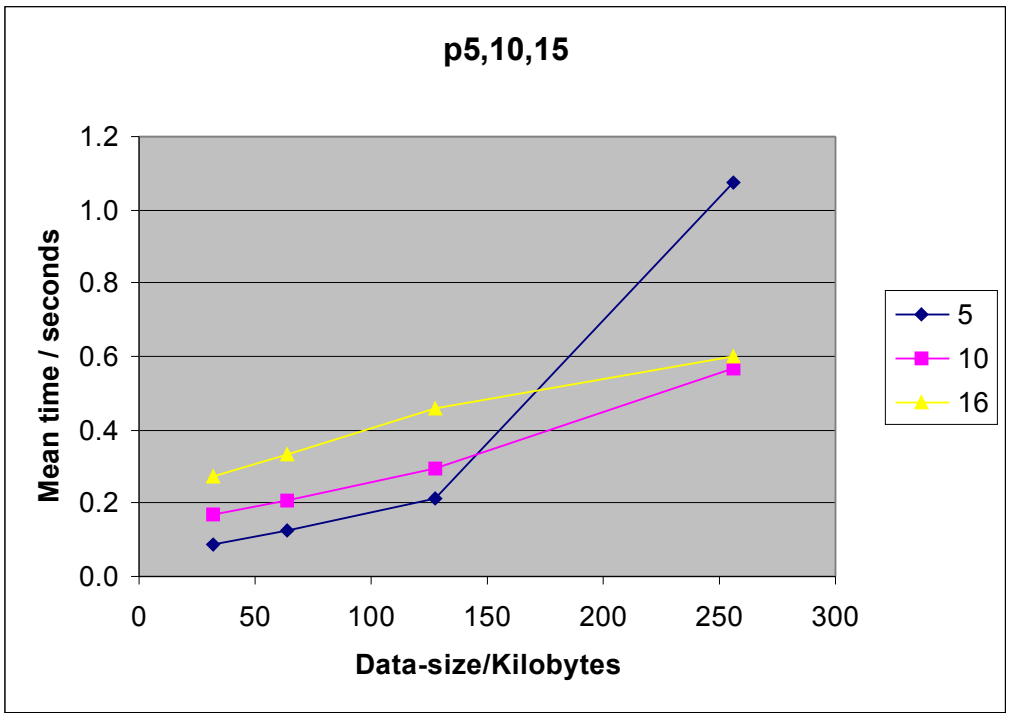
The results show a consistent increase in time with increasing data size broadcast.



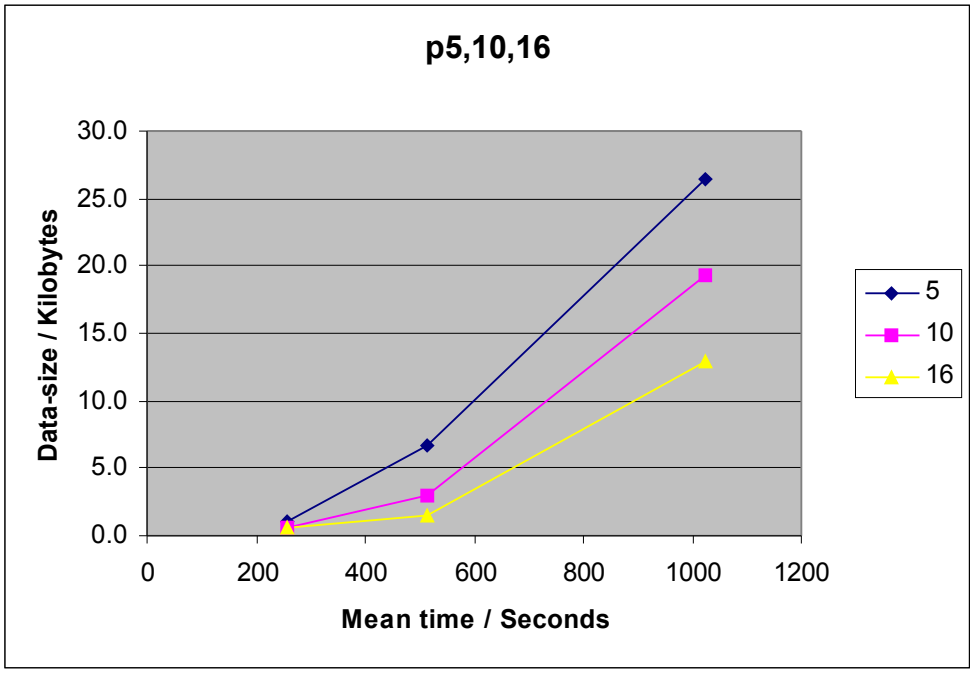




From looking at the above graphs it shows a curve, this indicates time is increasing in polynomial time or even exponentially relative to the data-size.



There is an anomaly in the results; increasing the number of processors is decreasing the time to broadcast. For 128KB and under a linear relationship can be seen, and data is broadcast quickest on 5 processors and slowest on 16 processors. After 128KB the times for 10 and 5 processors to broadcast quickly increases.



The above graph shows that the running time is decreasing with more processors.

Polynomial model

This is done by plotting the log of each of the variables on a graph; a straight-line graph indicates that the relationship is polynomial. The equation for the straight line is calculated using the approximation techniques on a Casio CFX-9850G graphic calculator.

y = time to broadcast / seconds
x = data size / kilobytes

Model 1 (Full Data)

p=5, $y = 2.605 \times 10^{-5} x^2 - 7.873 \times 10^{-4} x + 0.1556985$
p=10, $y = 2.5526 \times 10^{-5} x^2 - 7.978 \times 10^{-3} x + 0.6533288$
p=16, $y = 1.8787 \times 10^{-5} x^2 - 7.382 \times 10^{-3} x + 0.81076197$

Model 2 (Smaller Data Values)

p=5, $y = 2.341 \times 10^{-5} x^2 - 2.415 \times 10^{-3} x + 0.1556985$
p=5, $y = 3.5129 \times 10^{-6} x^2 - 7.6479 \times 10^{-4} x + 0.14069533$
p=5, $y = 1.4391 \times 10^{-3} x^2 + 0.24421613$

Broadcast Type:	2-Phase	
		Time /seconds

Number of Processors	Data size /Kilobytes	Mean	Model 1 prediction	Model 1 difference	Model 2 prediction	Model 2 difference
5	32	0.085144	-0.035058	0.120202	0.102390	**-0.017246**
5	64	0.127187	0.019774	0.107413	0.097026	**0.030161**
5	128	0.215030	0.289489	-0.074459	0.230128	**-0.015098**
5	256	1.073815	1.469125	-0.395310	1.071656	**0.002159**
5	512	6.679988	6.389214	0.290774	5.056010	1.623978
5	1024	26.430414	26.472670	**-0.042256**	22.229903	4.200511
10	32	0.166666	0.424171	-0.257506	0.145057	**0.021608**
10	64	0.207706	0.247291	-0.039585	0.155849	**0.051857**
10	128	0.294307	0.050363	0.243945	0.199015	**0.095292**
10	256	0.566967	0.283833	0.283134	0.371682	0.195285
10	512	2.981274	3.260081	-0.278807	1.062346	1.918928
10	1024	19.298234	19.249808	**0.048426**	3.825003	15.473231
16	32	0.275207	0.593776	-0.318569	0.290267	**-0.015060**
16	64	0.333766	0.415266	-0.081500	0.336319	**-0.002553**
16	128	0.458612	0.173672	0.284940	0.428421	**0.030191**
16	256	0.600059	0.152195	0.447864	0.612626	**-0.012567**
16	512	1.543545	1.956077	-0.412532	0.981035	0.562510
16	1024	12.989134	12.951191	**0.037943**	1.717855	11.271280

Both the models are compared in the above table, I've calculated the values as per each of the models and calculated the difference between the actual timing information and the models prediction. You can see that the predictions fit poorly, the model, which incorporates all the data, only really fits for the largest data-size 1024KB. As you can see the differences being 0.04, 0.05, and 0.04, so 26.47sec versus 26.43sec on 5 processors, 19.25 versus 19.30 on 10 processors and 12.95sec compared to 12.99sec on 16 processors. This is due to the influence of the largest result. Looking at the model calculated from just the smaller results (32KB – 256KB) the predictions fit a lot closer. For 5 processors the differences are 0.017, 0.030, 0.015, and 0.002, this gives accuracy to about 1 decimal place. For 10 processors the model differs further, 0.022, 0.051, 0.095, the predictions are still quite close to the model. As with 16 processors, the differences are less than 0.03 seconds, which makes the model accurate to 1 decimal place.

Overall the models do not fit closely, however the graphs do resemble a polynomial type distribution of results.

Conclusions

The experimental results appear to deviate from the BSP model. If we trust the experimental results this would be because the architecture used is not in fact a true BSP computer, and hence why the results are not modelled well by the BSP model. This could be due to the network topology, the transport protocol, etc. However we do not have enough data to reliably say that the relationship is exponential. It does appear that the parameters g and l are varying for different data-sizes so it would be better to look at it in terms that the relation should be linear, and work out the parameters g and l for this machine, and how they change for different data-sizes.

Another possibility is that the code is not optimal for this type of broadcast. The experiment is not designed perfectly. Firstly each repetition requires the program to be run from start to finish each time, there was no repetitions in the main loop to allow us to avoid the start up costs. Secondly the timing information is for the entire bsp_begin() bsp_end() block of code, and so includes any initialisation costs and computations. During this initialisation step two integers are broadcast, one that tells each processor how many integers are to be broadcast and one that tells each processor whether to run in verbose mode or not. Then an array is allocated on each processor and populated with data. This is a “for” loop, the number of times it goes through the loop is equal to the number of integers to broadcast. As such as the data-size increases this will be adding to the time to broadcast. Also the broadcast routine uses 3 supersteps instead of 2, as the array which receives the broadcast data has to be registered and there are some initial costs for calculating the number of integers each processors should receive. These all add to the timing to broadcast the number of

integers. A better experiment would have been to just time the 2 super-steps for which the program broadcasts data.

Also it may be due to the network, up to some data-size there data maybe cached/fit into a buffer, which improves the performance for smaller data-sizes. For larger data-sizes the broadcast would thus be un-buffered and performance would deteriate.

BSP Predicted values for 2-phase

So far I've commented on what the BSP model predicts that the behaviour of the algorithm should be. The actual values that are predicted are as follows.

Broadcast Type:		2-Phase		
Number of Processors	**Data size /Kilobytes**	**Mean Time**	**Data size – Previous data size ratio**	**Time - Previous time ratio**
5	32	0.085144		
5	64	0.127187	2	1.49
5	128	0.215030	2	1.69
5	256	1.073815	2	4.99
5	512	6.679988	2	6.22
5	1024	26.430414	2	3.96
10	32	0.166666		
10	64	0.207706	2	1.25
10	128	0.294307	2	1.42
10	256	0.566967	2	1.93
10	512	2.981274	2	5.26
10	1024	19.298234	2	6.47
16	32	0.275207		
16	64	0.333766	2	1.21
16	128	0.458612	2	1.37
16	256	0.600059	2	1.31
16	512	1.543545	2	2.57
16	1024	12.989134	2	8.42

The above table includes a column to show the ratio of the broadcast times, as the data size doubles. At each step we are doubling the data-size and for small values e.g. 32-64KB for 5 processors we see 1.5 and 1.7 times increase in the time. For 10 processors we see 1.25 times and 1.42 times increase in the time to broadcast. Then for 5 processors doubling the data-size from 128 to 256 causes the time to be broadcast to increase by 600% from 1.07 seconds to 6.68 seconds. Then doubling the data-size again from 512KB to 1024KB increases the broadcast time by about 400% (times 3.96). For 10 processors a similar pattern can be seen for doubling the data-size from 32Kb all the way through to 256Kb we get an increase in broadcast time up to a factor of almost 200% on each doubling of the data-size (as would be expected), x1.25, x1.42, x1.93, though all less than the expected doubling. Then doubling the data-size from 256Kb to 512Kb gives us an increase of 500% (5.26) and increasing from 512Kb to 1024Kb gives an increase of 600% (6.47).

For 16 processors, we can see a similar relationship less than 256Kb, doubling the data-size multiplies the broadcast time by 1.21, 1.37 and 1.31. Then for doubling data-sizes greater than 256Kb time more than doubles 2.57x, 8.42x.

So below 256Kb to broadcast the 2-Phase is performing better than BSP predicts as doubling the data-size increases the time to broadcast by a lesser factor. Broadcasting more than 256Kb, the 2-phase performs a lot worse than predicted by BSP. Performance appears to deteriorate rapidly with increasing of the data-size.

One interesting observation is the variety in the results as the data-size increases the variation of the results increases. On 5 processors the Standard deviation is less negligible for data of less than 256Kb (±0.0032sec, ±0.011sec, ±0.0136sec…) but for data-sizes greater than that the standard deviation increases ±1.58sec, ±1.21sec,

±3.00sec which is significantly larger, though 95% are within that range so we can still be pretty confident about the accuracy of the results.

Please note I was unable to get as many results for larger data-sizes as with more frequency, greater data-sizes would cause no timing information to be outputted.

For 10 processors the Standard deviation is small, under 0.08 seconds then for 512Kb and above the standard deviation is 1.23 and 4.38.

Interestingly the variation of the results for 16 processors is a lot smaller than for 5 and 10 processors. For 16 processors the Standard deviation is as small as 1.94sec, 0.23sec and 0.05sec for the data-sizes 1024Kb, 512Kb and 256Kb respectively.

The reason for the greater variation in the broadcast times I would suggest is due to the latency in the network. In terms of BSP it is the two parameters g and l, that determine incorporate the performance of the network. L is the “latency” and g is “gap”. The latency is the number of time steps for barrier synchronisation, and g is the permitivity of the communication network. Both parameters are assumed to be constant for a particular architecture. This may not be the case; in fact it is not the case on the cluster of Unix workstations. Potentially the performance of the network may vary with increasing data communicated on the network. This will be discussed later on in the report.

One strange observation from the results is the way in which the broadcast time varies for a particular data-size on different number of processors. These results show that the greater the number of processors the less time it takes to broadcast the same amount of data. This is an unexpected result. As the number of processors increases

the amount data transmitted in total by the network increases. Therefore performance of the network is expected to degrade.

e.g 2-phase broadcast p=5 N=1024 (datasize)

During total exchange each of the 5 processors sends 1024/5 Kb of data to each of the other 5 processors. So the maximum message size is 1024/5, but the total amount of data transmitted by the network is 5*1024/5*5 which is 1024*5. For 16 processors the maximum message length is 1024/5, but the total data transmitted is 1024*16. Therefore with an increase in the amount of data transmitted you would expect degradation in the network performance, not an improvement. I ran a series of tests on direct broadcast, the results can be seen later on in the report.

The BSP equation for 2-Phase broadcast is as follows;

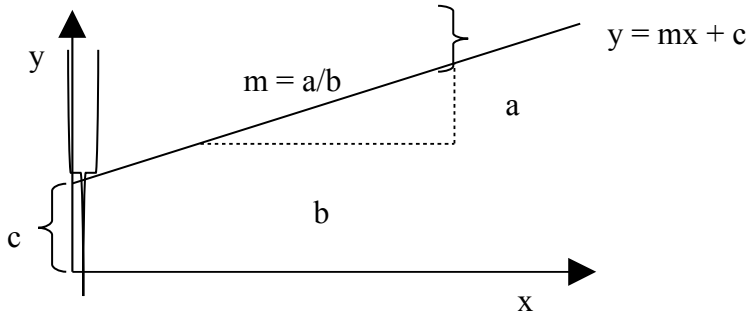
Comm. = 2Ng

Sync. = 2l

Therefore; time = 2Ng + 2l

time = a . data-size + b, where a and b are constants

In other words time is directly proportional to data-size, and it you would expect to find a linear relationship between time and data-size. In other words a straight-line graph such as below would have been expected.



One very strange observation is that the larger the number of processors, the less time the broadcast was taking on the DCS Lab workstations. This is a very strange result as with greater number of processors there are more processors to send data to, and therefore a greater amount of data transferred by the network on each super step, which would suggest that the broadcast times should increase. When the same code was run on the Oscar super computer it was found that the time to broadcast did in-fact increase with the more processors.

In fact to be more correct you would expect the 2-phase broadcast to be largely unaffected by the increase in the number of processors, as the cost is only based on the data size. The tree method on the other hand is based on the size of the data and the number of processors. Which means you would expect the times of the tree broadcast to diverge away from those for the 2-Phase broadcast.

		Time for type		
Number of processors	**Data-size /Kilobytes**	**TREE**	**2-PHASE**	**DIRECT**
5	32	0.129666	0.085144	**0.08934**
5	64	0.179916	0.127187	**0.124786**
5	128	0.496433	0.21503	**0.202098**
5	256	0.601885	1.073815	**0.349762**
5	512	2.461492	6.679988	**0.679784**
5	1024	7.531307	26.43041	**1.591121**
10	32	0.278171	0.166666	**0.138927**
10	64	0.385844	0.207706	**0.235066**
10	128	0.794472	0.294307	**0.376475**
10	256	1.696721	0.566967	**0.841741**

10	512	3.330218	2.981274	**1.777254**
10	1024	7.663237	19.29823	**4.025822**
16	32	0.376909	0.275207	**0.245722**
16	64	0.429697	0.333766	**0.381976**
16	128	0.711125	0.458612	
16	256	0.995928	0.600059	**1.145801**
16	512	2.171702	1.543545	**2.036489**
16	1024	5.15171	12.98913	**6.62627**

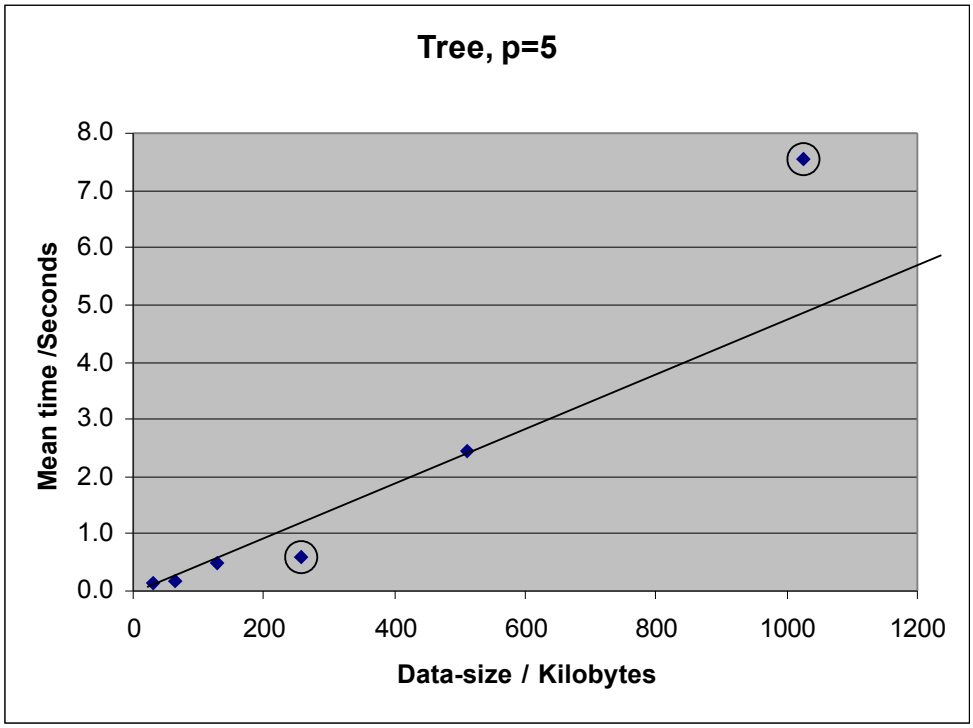
Analysis of the Tree Broadcast

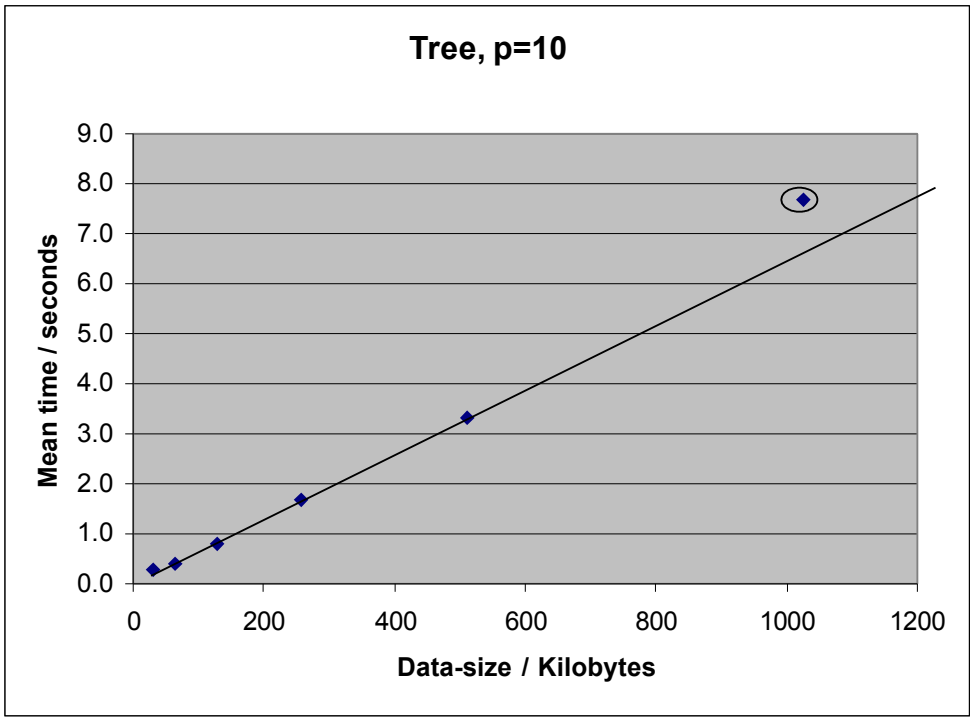
Broadcast Type:		TREE				
			Time /seconds			
Number of Processors	Data size /Kilobytes	Results population	Mean	Standard Deviation	Min	Max
5	32	6	0.129666	0.042368	0.097667	0.177714
5	64	6	0.179916	0.030544	0.159477	0.215028
5	128	7	0.496433	0.102329	0.378469	0.561297
5	256	6	0.601885	0.289138	0.384495	0.930027
5	512	4	2.461492	1.768308	1.214207	4.485178
5	1024	7	7.531307	2.939756	4.594662	10.474165
10	32	5	0.278171	0.029514	0.244122	0.296454
10	64	6	0.385844	0.016273	0.369036	0.401523
10	128	6	0.794472	0.050030	0.764564	0.852230
10	256	5	1.696721	0.145150	1.553768	1.843973
10	512	3	3.330218	0.483317	2.849067	3.815672
10	1024	3	7.663237	0.264015	7.506538	7.968055
16	32	5	0.376909	0.023506	0.352221	0.399021
16	64	4	0.429697	0.010706	0.422931	0.442040
16	128	5	0.711125	0.061774	0.646632	0.769762
16	256	3	0.995928	0.205393	0.851250	1.231017
16	512	4	2.171702	0.343445	1.857665	2.538459
16	1024	3	5.151710	0.265667	4.871218	5.399531

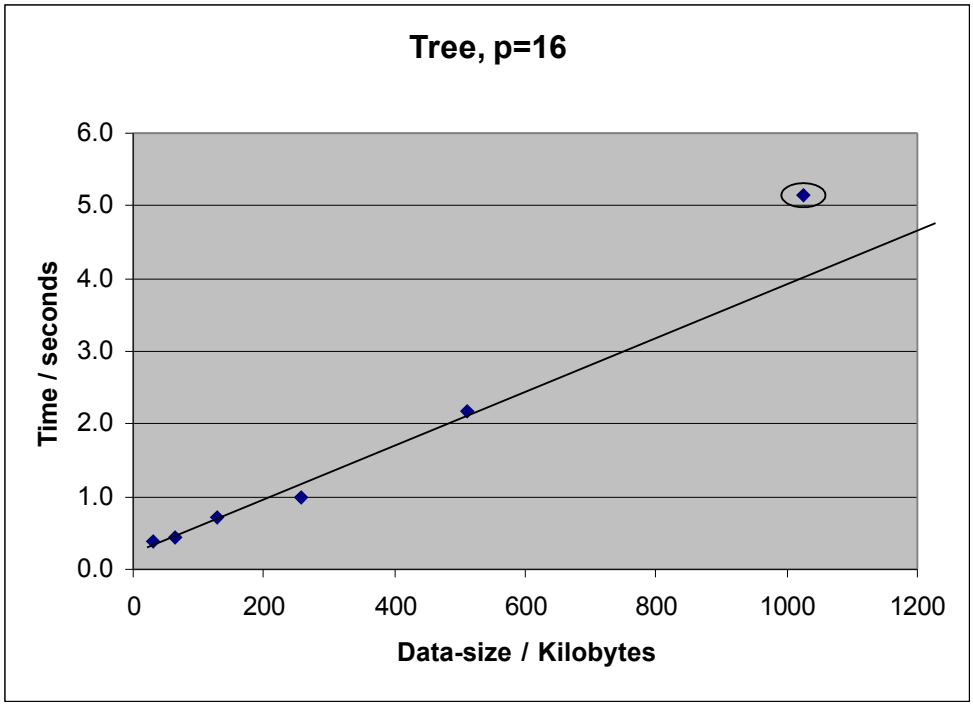
What we see is that with the increase in data-size, we get an increase in the time to broadcast. Unlike the 2-phase broadcast where there was the strange result of the time to broadcast decreasing with the number of processors, on this test the time increases with more processors, as would be expected. This is apart from 16 processors where the data-size is 1024KB, the time to broadcast is less than for 5 and 10 processors, 7.5 seconds on 5 processors, 7.7 seconds on 10 processors it then dipped down to 5.2 seconds for 16 processors.

p	5	10	16
Data-size / Kilobytes	1024	1024	1024
Time / seconds	7.5	7.7	5.2

The following graphs show the data-size versus the time to broadcast for different number of processors.



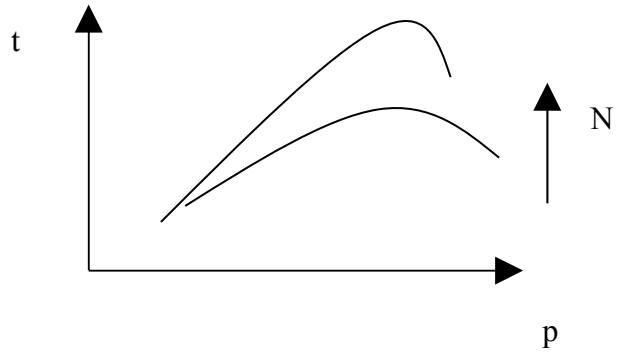




The graphs above show a linear relationship between data-size and broadcast time; broadcast time increases linearly with data-size, as predicted by the BSP model. Looking at the data I believe the times for data-size of 1024 to be spurious result, as together the other results can be approximated with a linear line of best fit. It is only the larger values that do not fit this relationship; this is highlighted on the above graphs. Including these data values in the results would require a logarithmic analysis of the data to identify the actual relationship.

I believe it is acceptable to ignore the largest data values as it is clearly seen something strange is happening for larger data sizes, as the time to broadcast is decreasing with number of processors. For this reasoning the timings for 512KB and on 16 processors should also not be accepted as it is less for that on 10 processors.

We seem to be getting the following pattern, which is a bit strange:



OK this is happening for datasizes less than 1024 KB. However, because the data of smaller sizes all fits the linear model, I am going to use that data.

With out the largest data values an equation of each of the graphs can be calculated, which will allow the parameters g and l to be approximated for the architecture the tests were run on.

Using the graphic calculator analysis we get the following equations:

N = data-size, t = time for broadcast

p = 5, $4.9807 \times 10^{-3} \times N - 0.0995853$

p = 10, $6.4912 \times 10^{-3} \times N + 7.9197 \times 10^{-3}$

p = 16, $3.7269 \times 10^{-3} \times N + 0.19764633$

As you can see from the following table comparing the the above models to the actual values gives a reasonably close fit, albeit for the larger data-sizes.

Broadcast Type:	**TREE**

Number of Processors	Data size /Kilobytes	Mean	Model prediction	Difference	Relative Error
5	32	0.129666	**0.0597971**	**0.069869**	**53.9%**
5	64	0.179916	**0.2191795**	**-0.039264**	**-21.8%**
5	128	0.496433	0.5379443	-0.041511	-8.4%
5	256	0.601885	**1.1754739**	**-0.573589**	**-95.3%**
5	512	2.461492	2.4505331	0.010959	0.4%
5	1024	7.531307	**5.0006515**	**2.530655**	**33.6%**
10	32	0.278171	**0.2156381**	**0.062533**	**22.5%**
10	64	0.385844	0.4233565	-0.037512	-9.7%
10	128	0.794472	0.8387933	-0.044321	-5.6%
10	256	1.696721	1.6696669	0.027054	1.6%
10	512	3.330218	3.3314141	-0.001196	0.0%
10	1024	7.663237	**6.6549085**	**1.008329**	**13.2%**
16	32	0.376909	**0.3169071**	**0.060002**	**15.9%**
16	64	0.429697	0.4361679	-0.006471	-1.5%
16	128	0.711125	0.6746895	0.036436	5.1%
16	256	0.995928	**1.1517327**	**-0.155805**	**-15.6%**
16	512	2.171702	2.1058191	0.065883	3.0%
16	1024	5.151710	**4.0139919**	**1.137718**	**22.1%**

The above data shows the mean times together with the models predictions, the difference in times, and the relative error. This is the difference divided by the mean times. Those data values in bold are those that I feel deviate from the model the most, those having over 15% relative error, meaning the model prediction is 115% or 85% of the actual value. The largest errors are for those data values which I have circled in the graphs above. For 5 processors that is the times for data-sizes 256KB and 1024KB, and as you can see from the table below the error is quite large.

Data-size	**Actual**	**Prediction**
256KB	0.601	1.175
512KB	7.531	5.000

The smallest value is also quite out with error of 0.07 seconds giving half the time expected.

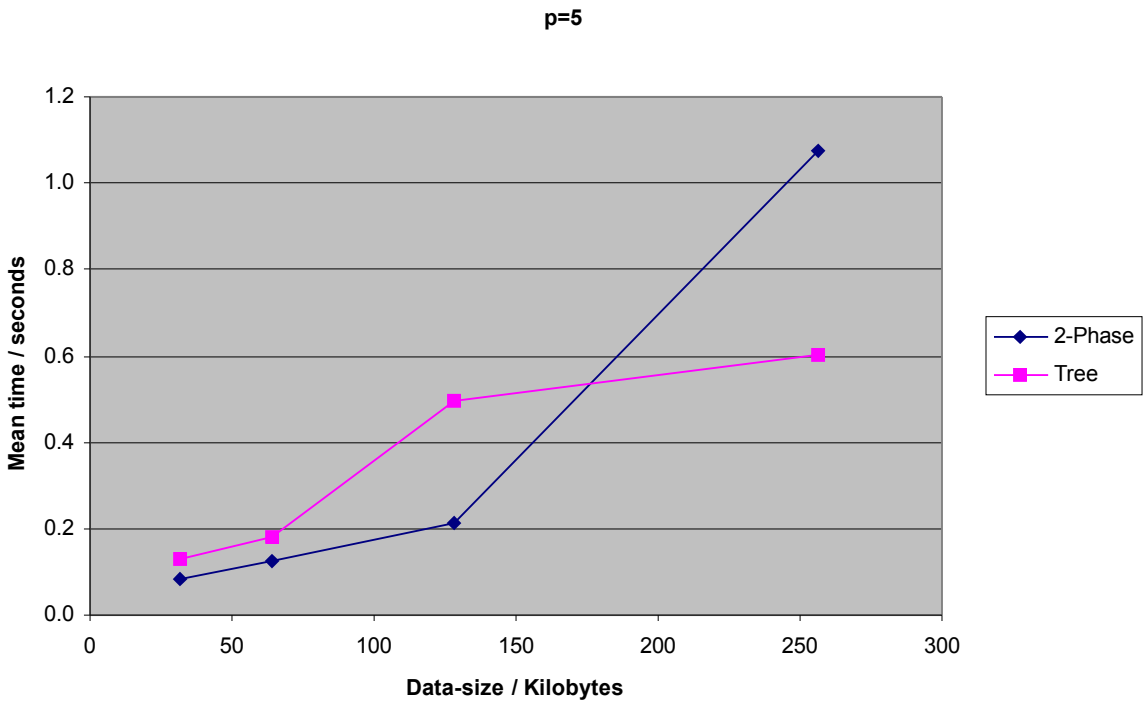
For 10 processors the largest data-size gives a time 1.008329 seconds out, but this is not a very large relative error (13.2%). For 16 processors the largest data-szie gives time 4.01 seconds instead of 5.15 seconds again only a 1 second error, but relatively

insignificant. The 256KB value has a larger error but this is also not highly significant. As such I think this model fits correctly.

2-Phase / Tree Model Compare

As you've seen the 2-phase algorithm does not appear to conform to the linearity in performance as predicted by the BSP model.

If you look at the data for Tree and 2-phase, it appears that 2-Phase is better for smaller data-sizes, whereas tree is better for larger data-sizes. For 5 processors we get;

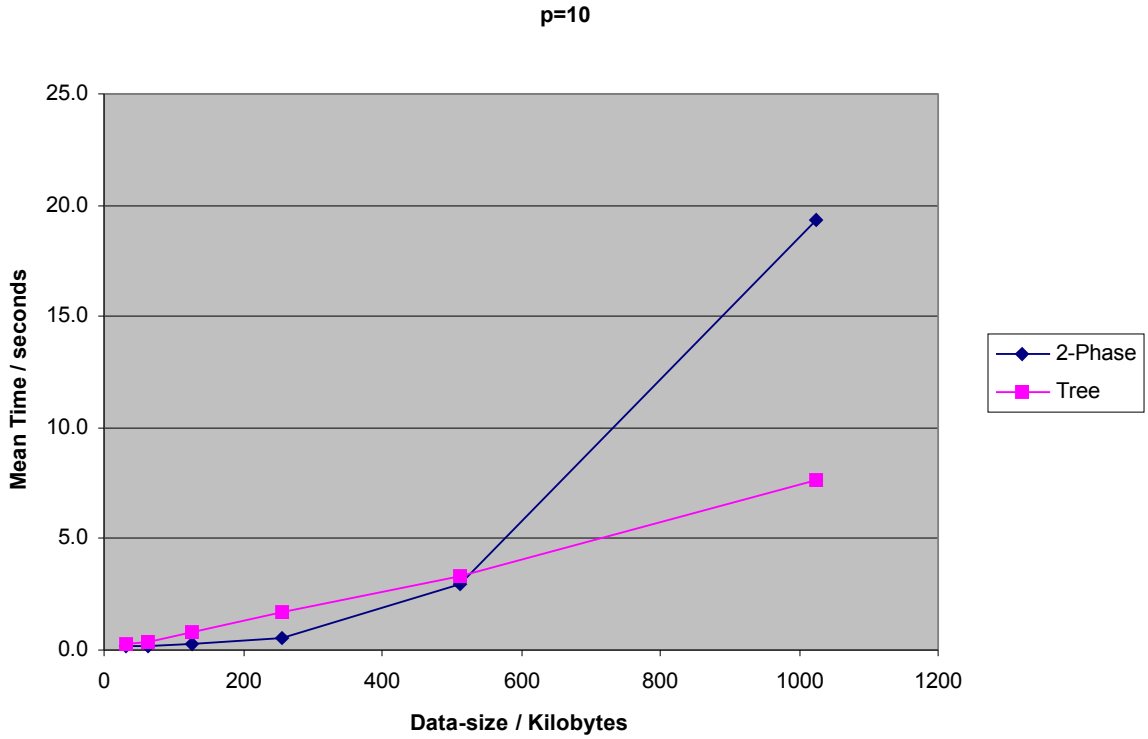


Data-Size/KB	2-Phase		Tree	Ratio
32	0.085	<	0.130	0.7
64	0.127	<	0.180	0.7
128	0.215	<	0.496	0.4
256	**1.077**	**>**	**0.602**	**1.8**
512	**6.680**	**>**	**2.461**	**2.7**
1024	**26.480**	**>**	**7.531**	**3.5**

There appears to be a cut off at 256KB where 2-phase has a shorter broadcast time than for tree broadcast, as you can see the ratio is up to half the time to broadcast e.g.

for 128KB 0.215 seconds for 2-phase and 0.496 seconds for tree. Then for over 256KB the time to broadcast using the 2-Phase method rapidly increases until for 1024KB its 3.5 times greater.

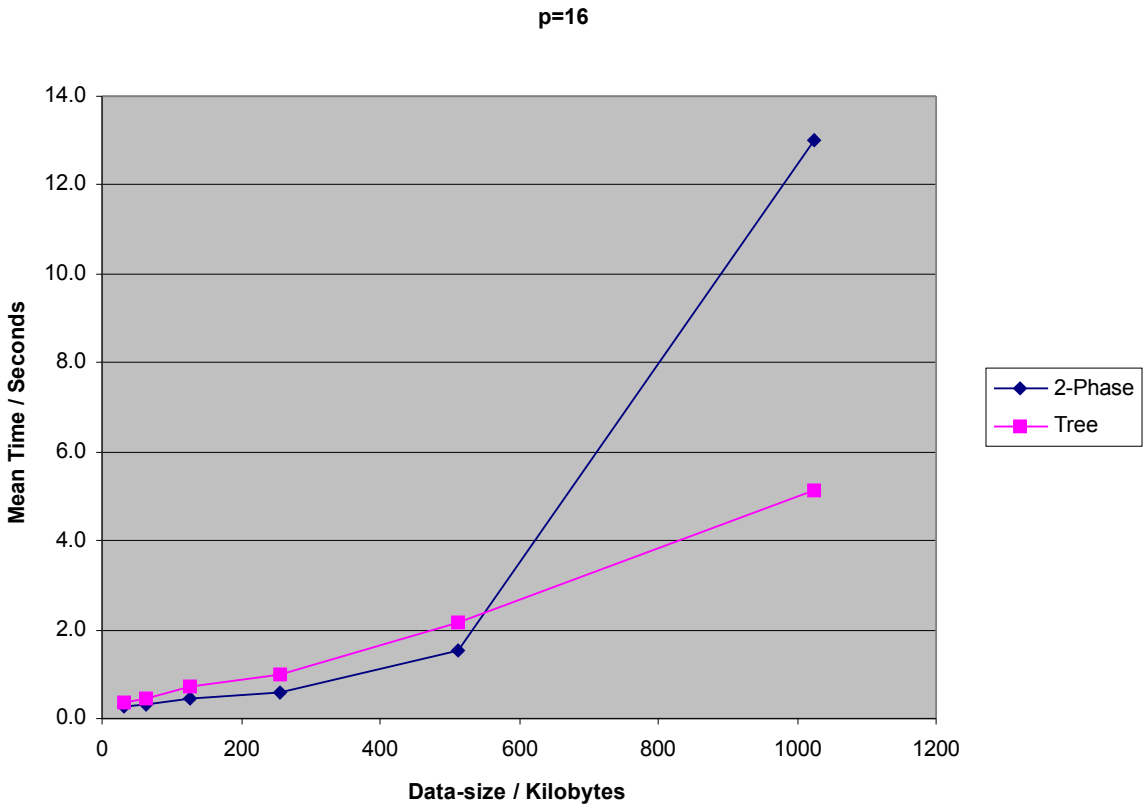
For 10 processors 2 phase performs better for all data-sizes apart from 1024KB, as you can see from the following table;



Data-Size/KB	2-Phase		Tree	Ratio
32	0.167	<	0.278	0.6
64	0.208	<	0.386	0.5
128	0.294	<	0.794	0.4
256	0.567	<	1.697	0.3
512	2.981	<	3.330	0.9
1024	**19.298**	**>**	**7.663**	**2.5**

As you can see the 2-phase ifs from 30% to 90% the time taken for tree method to broadcast.

For 16 processors we have:



Data-Size/KB	2-Phase		Tree	Ratio
32	0.275	<	0.377	0.7
64	0.334	<	0.430	0.8
128	0.459	<	0.711	0.6
256	0.600	<	0.996	0.6
512	1.543	<	2.172	0.7
1024	**12.989**	**>**	**5.152**	**2.5**

For data-sizes up to and including 512KB the 2-phase broadcast performs better, at about 0.6-0.8 of the times for the tree broadcast. Then for data-sizes of 1024KB 2-phase takes 2.5 times longer than for the tree method.

With increasing number of processors trees deteriorates relative to 2-phase. But this is balanced by the increasing amount of data transferred by 2-phase. The sharp increase in time to broadcast data of size 1024 KB is probably due to the large amount of data being transferred by the network.

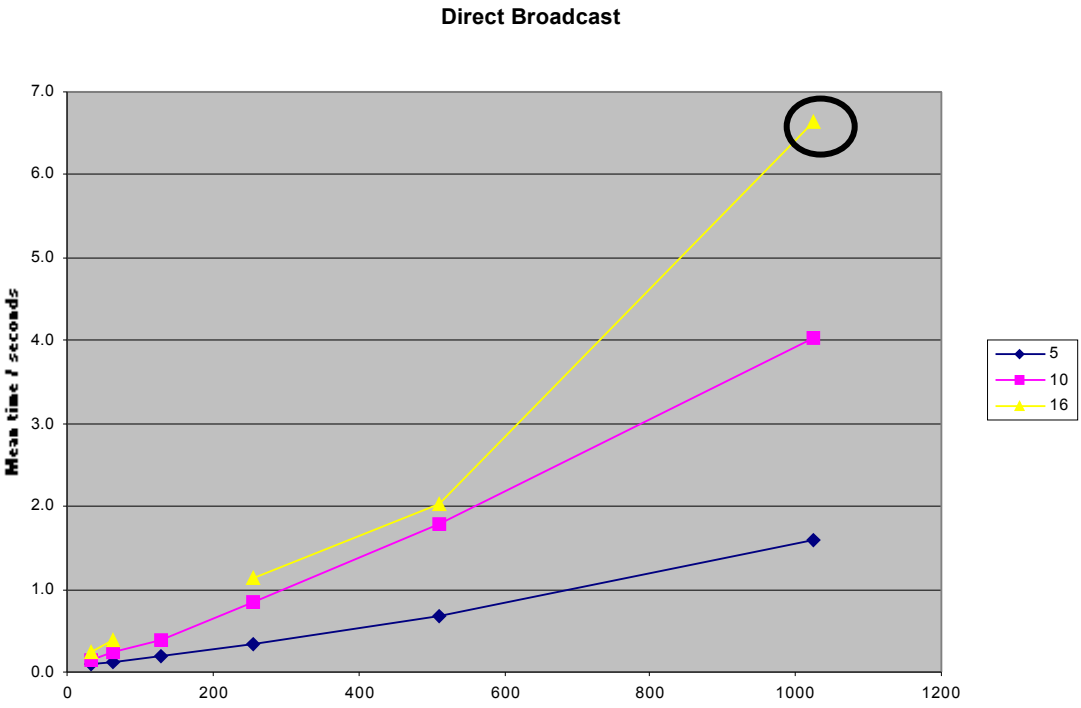
Analysis of Direct Broadcast

The tests for direct broadcast were run at a later date than for 2-phase and tree, and therefore are not necessarily directly comparable.

Broadcast Type:		DIRECT				
			Time /seconds			
Number of Processors	**Data size /Kilobytes**	**Results population**	**Mean**	**Standard Deviation**	**Min**	**Max**
5	32	1	0.089340	0.000000	0.089340	0.089340
5	64	3	0.124786	0.013194	0.115125	0.139819
5	128	3	0.202098	0.024396	0.183219	0.229644
5	256	3	0.349762	0.028278	0.318177	0.372727
5	512	3	0.679784	0.046773	0.652290	0.733790
5	1024	3	1.591121	0.091204	1.487382	1.658699
10	32	1	0.138927	0.000000	0.138927	0.138927
10	64	2	0.235066	0.003973	0.232257	0.237875
10	128	1	0.376475	0.000000	0.376475	0.376475
10	256	2	0.841741	0.010078	0.834614	0.848867
10	512	2	1.777254	0.211305	1.627838	1.926669
10	1024	3	4.025822	0.422803	3.552979	4.367490
16	32	1	0.245722	0.000000	0.245722	0.245722
16	64	2	0.381976	0.017434	0.369648	0.394304
16	256	2	1.145801	0.015168	1.135075	1.156526
16	512	2	2.036489	0.046048	2.003928	2.069050
16	1024	3	6.626270	3.539552	4.487570	10.711893

As you can see from the following graph direct broadcast behaves in the way expected. There is a linear relationship; as the data size increases the broadcast time

increases linearly. The graph also shows that the greater the number of processors the greater the broadcast time.



The BSP analysis gives the equation pNg+1, and as such we would expect a linear time graph, which is what the graph shows. The results fit the BSP model well. The equations for these graphs are as follows. I used the graphic calculator method to work out the following equations:

p=5, $1.5036 \times 10^{-3} \times N + 9.1774 \times 10^{-4}$

p=10, $3.9294 \times 10^{-3} \times N - 0.0877623$

p=16, $3.7308 \times 10^{-3} \times N + 0.14663509$

I'm ignoring the value for 16 processors, where the data-size was 1024KB, as I believe this to be a spurious result, as the other values fit a straight line graph.

The following table shows how close this model is to the experimental results;

Broadcast Type:	DIRECT				
		Time /seconds			
Number of Processors	**Data size /Kilobytes**	**Mean**	**Model 1 Predict**	**Difference**	**Relative error**
5	**32**	**0.089340**	**0.04903294**	**0.040**	**45.1%**
5	**64**	**0.124786**	**0.09714814**	**0.028**	**22.1%**
5	128	0.202098	0.19337854	0.009	4.3%
5	256	0.349762	0.38583934	-0.036	-10.3%
5	512	0.679784	0.77076094	-0.091	-13.4%
5	1024	1.591121	1.54060414	0.051	3.2%
10	**32**	**0.138927**	**0.0379785**	**0.101**	**72.7%**
10	**64**	**0.235066**	**0.1637193**	**0.071**	**30.4%**
10	128	0.376475	0.4152009	-0.039	-10.3%
10	256	0.841741	0.9181641	-0.076	-9.1%
10	512	1.777254	1.9240905	-0.147	-8.3%
10	1024	4.025822	3.9359433	0.090	2.2%
16	32	0.245722	0.26602069	-0.020	-8.3%
16	64	0.381976	0.38540629	-0.003	-0.9%
16	256	1.145801	1.10171989	0.044	3.8%
16	512	2.036489	2.05680469	-0.020	-1.0%
16	**1024**	**6.626270**	**3.96697429**	**2.659**	**40.1%**

The model is very close for 16 processors apart from 1024KB. For 5 processors and 10 processors, the model fits very closely apart from for 32KB and 64KB, where the models prediction is quite far off, up to 73%. However for such small values they are the same order of time, which means we can be reasonably confident with the reliability.

By comparing direct broadcast to 2-Phase and tree there are several trends, which I will now discuss.

Number of Processors	Data size /Kilobytes	2-Phase	Tree	Direct
5	32	**0.085144**	0.129666	0.089340
5	64	0.127187	0.179916	**0.124786**
5	128	0.215030	0.496433	**0.202098**
5	256	1.073815	0.601885	**0.349762**
5	512	6.679988	2.461492	**0.679784**
5	1024	26.430414	7.531307	**1.591121**
10	32	0.166666	0.278171	**0.138927**
10	64	**0.207706**	0.385844	0.235066
10	128	**0.294307**	0.794472	0.376475
10	256	**0.566967**	1.696721	0.841741
10	512	2.981274	3.330218	**1.777254**
10	1024	19.298234	7.663237	**4.025822**
16	32	0.275207	0.376909	**0.245722**
16	64	**0.333766**	0.429697	0.381976
16	128	**0.458612**	0.711125	
16	256	**0.600059**	0.995928	1.145801
16	512	**1.543545**	2.171702	2.036489
16	1024	12.989134	**5.151710**	6.626270

= minimum time

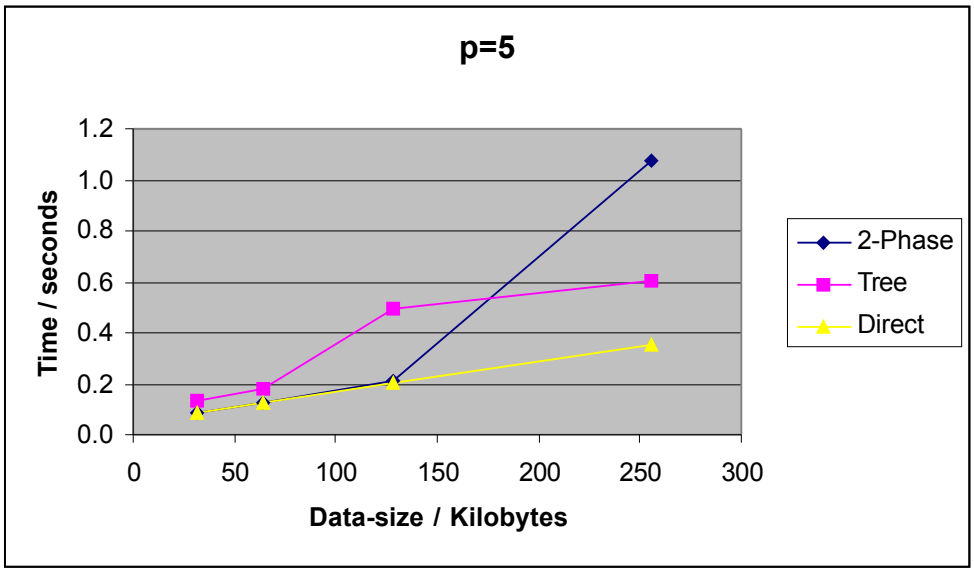
Firstly the less number of processors the better direct broadcast seems to perform. For p=5, direct broadcast has the smallest broadcast times, all accept for 32KB where it is 0.004 seconds worse than the 2-phase on average. However as the number of processors increases direct is best better for small data sizes of 23KB, having a smaller broadcast time for 10 and 16 processors. Then the 2-phase broadcast is better for size 64-256KB of data for 10 processors. Then for the larger data-sizes for 10 processors, 512KB and 1024KB direct is better. For 16 processors fro 1024KB tree is better but only slightly better than direct.

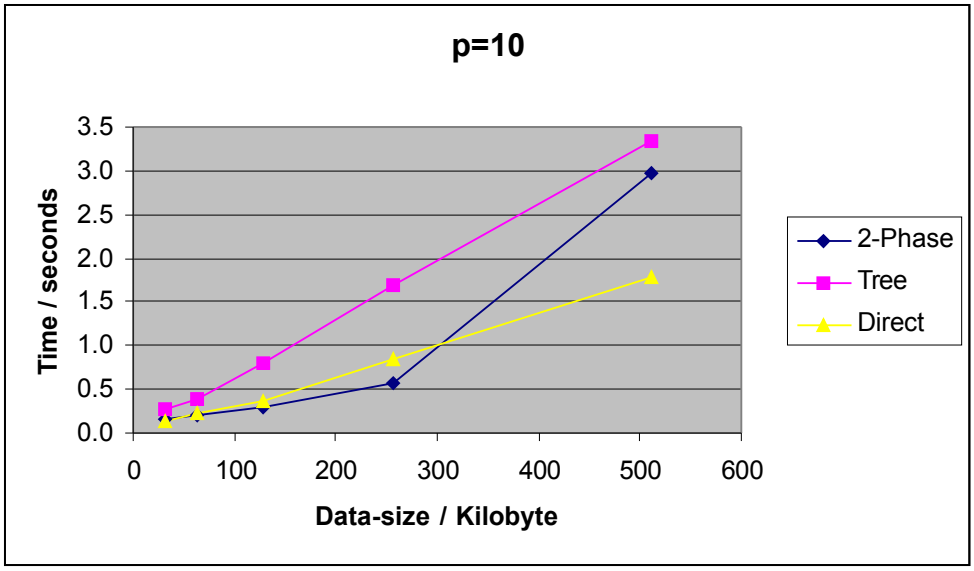
The results show that for smaller data-sizes direct broadcast is better than the other methods. For larger data-sizes direct broadcast also performed better. I believe this is due to spurious results for 2-phase and tree methods. Also the testing was carried out at a different time, and I believe this may be due to the performance of the network.

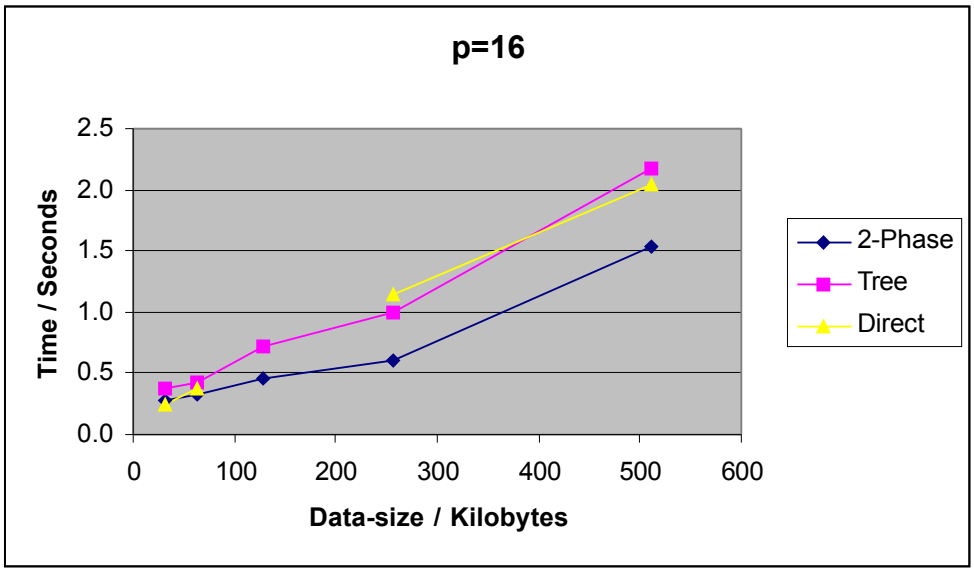
For direct broadcast the test was carried out at a different time, and poor performance of the network may have occurred when testing 2-phase and tree.

An odd result is that the direct broadcast actually performs better than the 2 supposedly optimal methods for broadcasting, potentially this maybe because two tests were carried out at different times, all 2-phase and tree were performed over a couple of days, the results for the direct method approximately a week later. I did not repeat whole test, only carried out testing on the direct broadcast and put this data together with the previous test, as you can see from the graphs above.

When N is small ($N < 1/(pg - 2g)$), it is better to use the direct broadcast method. Using g = 237.21 x 10-6 MFLOPS, for 5 processors direct broadcast should be used for N less than 11KB (1405 words = 1/g(5-2)), for 10 processors N less than 4KB (526 words = 1/g(10-2)) and for 16 processors N less than 2KB (264 words = 1/g(16-2)). The sizes are very small, and not in range of the test data used.




p=10
Time / seconds
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
100
200
300
400
500
600
Data-size / Kilobyte
2-Phase
Tree
Direct


p=16
Time / Seconds
2.5
2.0
1.5
1.0
0.5
0.0
0
100
200
300
400
500
600
Data-size / Kilobytes
2-Phase
Tree
Direct

2-phase Parameter Analysis

S=2Ng + 2l

S is normalised and gives the number of time steps, therefore it has to be divided by the performance measure of the architecture (in MFLOPS). BSP-lib gives the values of l and g to be;

s (Mflops/s)	P (no procs)	l (flops)	g (flops/word)
18.128	4	164505.0	237.21

The calculation is $t = \frac{2 \times N \times 256 \text{ words/KB} \times 237.21 \text{ FLOPS/word} + 164505 \text{ FLOPS}}{18.128 \times 10^6 \text{ FLOPS}}$

so $t = 6.6997 \times 10^{-3} \times N + 9.0746 \times 10^{-3}$

Note the 256 words/KB, to convert kilobytes to words.

The following table lists all the predicted values together with the experimental results, the difference and the percentage error (relative error).

2-Phase

Number of Processors	Data size /Kilobytes	Mean	BSP Model Prediction	Difference	Relative Error
5	32	0.085144	0.223463903	-0.138320	-162.5%
5	64	0.127187	0.437853171	-0.310666	-244.3%
5	128	0.215030	0.866631706	-0.651601	-303.0%
5	256	1.073815	1.724188775	-0.650374	-60.6%
5	512	6.679988	3.439302915	3.240685	48.5%
5	1024	26.430414	6.869531194	19.560883	74.0%

For the tree broadcast we get S = (log p)Ng + (log p)l
So $t = S/S_{dcs}$ S_{dcs} = speed of DCS machines
N = datasize
$t = \frac{\log p \times N \times 256 \text{ words/KB} \times 237.21 + \log p \times 164505}{18.128 \times 10^6}$
For p =5 then;
$t = 1.10049 \times 10^{-2} \times N + 2.722 \times 10^{-2}$

Tree

Number of Processors	Data size /Kilobytes	Mean	BSP Model Predict (s=18.128, l=164505, g=237.21)	Difference	Relative Error
5	32	0.129666	0.34880781	-0.219142	-169.0%
5	64	0.179916	0.67039171	-0.490476	-272.6%
5	128	0.496433	1.31355951	-0.817126	-164.6%
5	256	0.601885	2.59989512	-1.998010	-332.0%
5	512	2.461492	5.17256633	-2.711075	-110.1%
5	1024	7.531307	10.3179087	-2.786602	-37.0%

The predicted values using these parameters are no-where near. It is pointless using these parameters as realised that these are for SOLARIS machine, and we are using AMD machines. Therefore the values that should be the closest are those for a Pentium Pro NOW. The values can be found in the architecture section. Ok there is a large difference between the predicted by the BSP model and the experimental data. However the predictions made are of a similar order. For 2-phase the data-sizes <256KB the BSP model overestimates the time to broadcast. For datasizes > 256KB the times are underestimated. For tree we see that the times are overestimated for all data-sizes up to 1024KB. For 1024 KB the time is underestimated by the BSPmodel.

It is interesting to see what the model actually predicted;

Number of Processors	Data size /Kilobytes	2-Phase	Tree	Direct
5	32	0.223463903<	0.348807809<	0.535968055
5	64	0.437853171<	0.67039171<	1.071936055
5	128	0.866631706<	1.313559512<	2.143872055
5	256	1.724188775<	2.599895117<	4.287744055
5	512	3.439302915<	5.172566326<	8.575488055
5	1024	6.869531194<	10.31790874<	17.15097606

For this data range the BSP model predicts that the 2-Phase algorithm will be better then the tree algorithm.

Working out the intersection of the two equations gives the datasize for which Tree will be better than 2-Phase:

$t_{2\text{-phase}} = 6.6997 \times 10^{-3} \times N + 9.0746 \times 10^{-3}$
$t_{tree} = 1.10049 \times 10^{-2} \times N + 2.722 \times 10^{-2}$

Intersection therefore when $t_{2\text{-phase}} = t_{tree}$.

$6.6997 \times N + 9.0746 = 11.0049 \times N + 27.22$
$-18.145 = 4.3052 \times N$
$N = -4.2$ KB

Therefore predicts that for values of $N > 0$ 2Phase will perform better than the tree broadcast.

This is very interesting because the experimental results show this to be the case for all smaller data-sizes. Then for larger data-sizes it shows tree to be better. We can also look at what the BSP model predicts for direct Broadcast.

Direct Broadcast

$S = pNg + 1$
So $t_{direct} = S/S_{dcs} = (5 \times N \times 256 \text{ Words/KB} \times 237.21 + 1)/(18.128 \times 10^6)$
$t_{direct} = 1.6749 \times 10^{-2} \times N + 5.516 \times 10^{-8}$

Number of Processors	Data size /Kilobytes	Mean	BSP Model Predict (s=18.128, l=164505, g=237.21)	difference	Relative error
5	32	0.089340	0.535968055	-0.446628	-499.9%
5	64	0.124786	1.071936055	-0.947150	-759.0%
5	128	0.202098	2.143872055	-1.941774	-960.8%
5	256	0.349762	4.287744055	-3.937982	-1125.9%
5	512	0.679784	8.575488055	-7.895704	-1161.5%
5	1024	1.591121	17.15097606	-15.559855	-977.9%

For direct broadcast the BSP model has consistently overestimated the time to broadcast for all datasizes. Most of the sizes are an order of magnitude out.

$t_{direct} = 1.6749 \times 10^{-2} \times N + 5.516 \times 10^{-8}$
$t_{2\text{-}phase} = 6.6997 \times 10^{-3} \times N + 9.0746 \times 10^{-3}$
$t_{tree} = 1.10049 \times 10^{-2} \times N + 2.722 \times 10^{-2}$

Direct & 2-phase
$16.749 \times N + 5.516 \times 10^{-5} = 6.6997 \times N + 9.0746$
$10.0493 \times N = 9.075$
$N = 0.90$ KB

Direct & tree

$16.749 \times N + 5.516 \times 10^{-5} = 11.0049 \times N + 27.22$
$4.3052 \times N = -18.145$
$N = -4.2$ KB

This shows that for all $N > 0$ direct direct will perform worse than the tree broadcast. For $N < 0.90$ KB the direct broadcast will perform better than the 2-Phase broadcast.

This is limited by the choice of s, g and l. To this point I've used those given by the BSP-lib parameter database for p=4, and used it as an approximation for p=5. These values were achieved experimentally on a different architecture, and so will vary depending on the architecture used. Therefore I'm going to attemp to derive the values for the parameters from the experimental data.

Deriving the parameters

Experimental
p = 5, $t = 4.9807 \times 10^{-3} \times N - 0.0995853$
p = 10, $t = 6.4912 \times 10^{-3} \times N + 7.9197 \times 10^{-3}$
p = 16, $t = 3.7269 \times 10^{-3} \times N + 0.19764633$

For 5 processors the experimental and predicted were actually very close. The gradient being 4.98 seconds as opposed to 5.02 seconds but the constant is –9.96x10-2 instead of 2.72 x 10-2. That it is negative is an anomaly, and would indicate that this equation is not accurate.

So in form mx + c

$$c = \frac{(\log p)\, l}{S}$$

$$m = \frac{(\log p)\, g}{S}$$

x = N/256 (to convert words to kilobytes)

log p = log2 5 = 3

And assuming S = 18.128 x 10^6. Applying the equation gives us the following values for g and l:

Tree

p	m	c	g / FLOPS/word	l / FLOPS
5	0.0049807	-0.0995853	117.5652729	-601760.7728
10	0.0064912	0.0079197	114.914525	35892.0804
16	0.0037269	0.19764633	65.97777656	1194310.89

So the g value is very close, but the value for l, latency is inaccurate.

Analysis of repeated direct broadcast

After completing the test on Oscar I went about re-testing the code on the DCS machines. I started by tweaking the direct broadcast, the changes I made were that the repetitions of code were built in; as such there would be no repetition of start up costs. Once the program starts, it has a loop within the begin-end block that repeats the code. I did this as I felt it would give more consistent results. The test was run with just direct broadcast the results I got were:

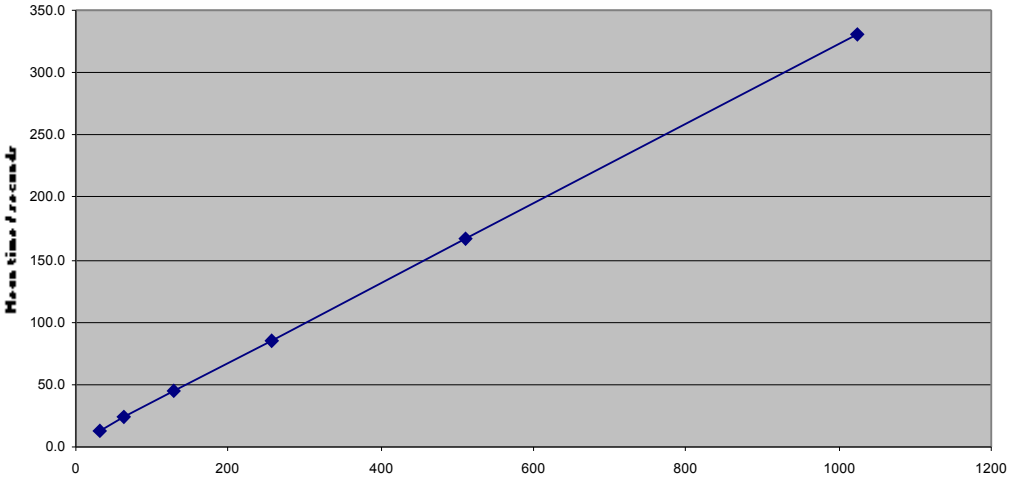
Broadcast Type:		direct_bcast				
			Time /seconds			
Number of Processors	**Data size /Kilobytes**	**Results population**	**Mean**	**Standard Deviation**	**Min**	**Max**
5	32	10	6.150998	0.141318	5.959550	6.394088
5	64	10	10.808641	0.189258	10.563412	11.220598
5	128	10	19.862605	0.181242	19.655555	20.278371
5	256	10	38.006262	0.127125	37.857871	38.226806

5	512	10	74.595997	0.113442	74.441867	74.761697
5	1024	10	147.673453	0.129154	147.492764	147.907464
10	32	10	13.402549	0.147544	13.211999	13.620541
10	64	10	23.907726	0.245907	23.472449	24.250634
10	128	10	44.291663	0.270328	44.014031	44.867640
10	256	20	85.499574	0.366955	84.939268	86.115177
10	512	10	167.540894	0.385760	167.160904	168.338211
10	1024	10	331.451901	0.469322	331.011989	332.487096
16	32	10	22.446436	0.592830	21.723329	23.344536
16	64	10	39.492787	0.419376	38.784685	39.966624
16	128	10	73.399689	0.644678	72.833290	74.935227
16	256	10	141.733362	0.388334	141.317034	142.470054
16	512	10	278.631036	0.747625	277.964335	279.903082
16	1024	10	552.071977	1.291957	550.886716	555.181606
32	32	10	44.852134	0.465414	44.434334	46.095088
32	64	10	80.956155	0.863045	80.149824	82.283957
32	128	10	150.348145	0.897455	149.895804	152.866773
32	256	10	292.888320	1.192067	291.182915	294.788522
32	512	10	573.908536	0.190179	573.703000	574.283573
32	1024	10	1139.894998	1.944502	1137.202142	1142.240558

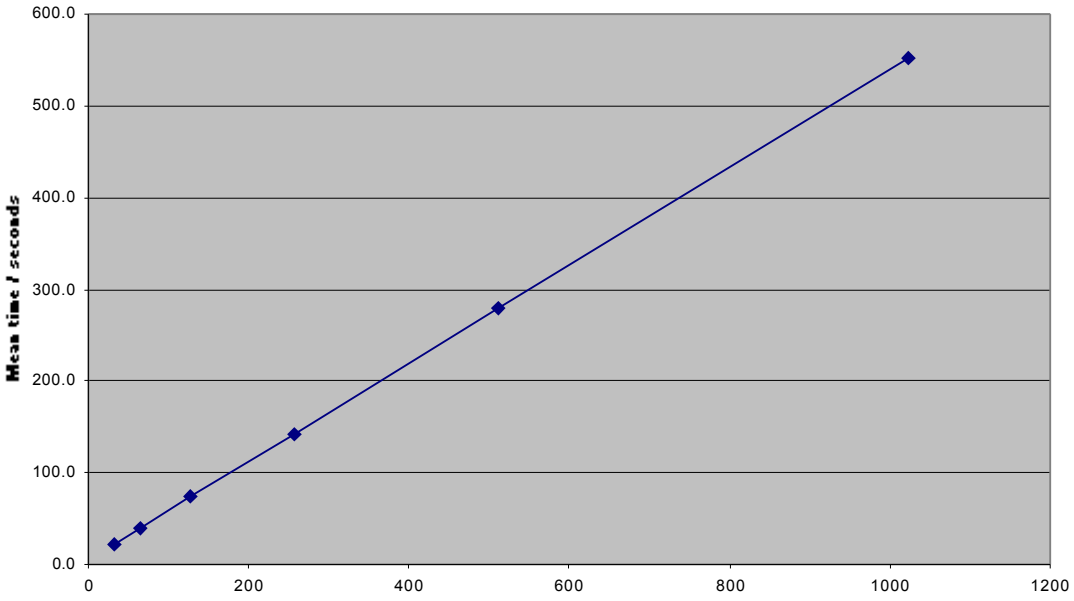
The data shows a similar pattern found for the previous testing of direct broadcast. The time of broadcast increases with increasing data-size, as you can see from the above table e.g. for 5 processors the time increases reasonably linearly from 6.2 seoncds for 32KB to 148 seconds for 1024KB broadcast. The standard deviation is a lot lower than in the previous results, indicating less variation in the results. This is due to the change in the code so the program doesn't have to be restarted each time, as such start up and finish costs have no impact on the broadcast times recorded. Also there is not such an impact on the network from one test to another.

The following graphs display the data-size versus the time for 5, 10, 16 and 32 processors.

p=10



p=16



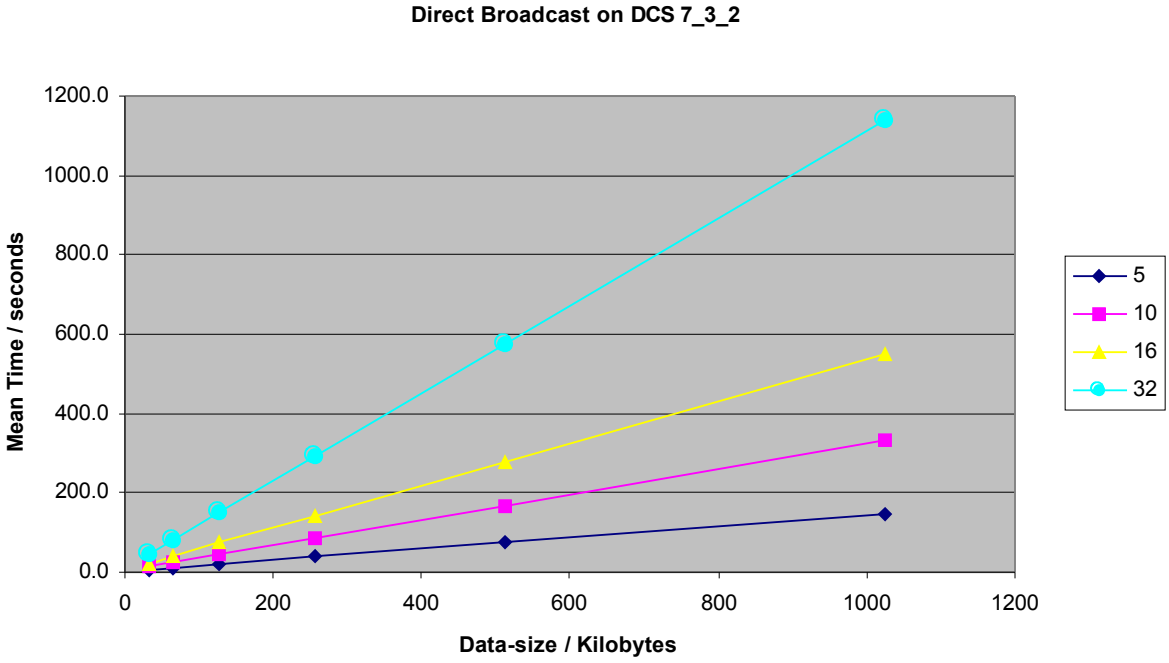
The results give an exact fit for a linear relationship on each graph. The BSP model predicts a linear relationship, as such the results fit closely to the BSP model. This is

the same as what was achieved on the first testing of the direct broadcast. The data fits even closer to a straight line. The equations for the above graphs are:

p=5, 0.14262734 x N + 1.59353934
p=10, 0.32051821 x N + 3.32159608
p=16, 0.53401106 x N + 5.20149825
p=32, 1.1034519 x N + 9.72048971

Number of Processors	Data size /Kilobytes	Mean	Model	Error	Relative Error
5	32	6.150998	6.15761422	-0.006616	-0.1%
5	64	10.808641	10.7216891	0.086952	0.8%
5	128	19.862605	19.84983886	0.012766	0.1%
5	256	38.006262	38.10613838	-0.099877	-0.3%
5	512	74.595997	74.61873742	-0.022741	0.0%
5	1024	147.673453	147.6439355	0.029518	0.0%
10	32	13.402549	13.5781788	-0.175629	-1.3%
10	64	23.907726	23.83476152	0.072965	0.3%
10	128	44.291663	44.34792696	-0.056264	-0.1%
10	256	85.499574	85.37425784	0.125316	0.1%
10	512	167.540894	167.4269196	0.113974	0.1%
10	1024	331.451901	331.5322431	-0.080343	0.0%
16	32	22.446436	22.28985217	0.156584	0.7%
16	64	39.492787	39.37820609	0.114581	0.3%
16	128	73.399689	73.55491393	-0.155225	-0.2%
16	256	141.733362	141.9083296	-0.174968	-0.1%
16	512	278.631036	278.615161	0.015875	0.0%
16	1024	552.071977	552.0288237	0.043153	0.0%
32	32	44.852134	45.03041579	-0.178282	-0.4%
32	64	80.956155	80.34034187	0.615814	0.8%
32	128	150.348145	150.960194	-0.612049	-0.4%
32	256	292.888320	292.1998984	0.688422	0.2%
32	512	573.908536	574.679307	-0.770771	-0.1%
32	1024	1139.894998	1139.638124	0.256874	0.0%

The following graph is the important one, as it tells us whether we've resolved the issue of decreasing broadcast time with increasing number of processors. This is infact untrue, as the original direct broadcast gave us the expected relationship that the more processors the greater the time to broadcast;



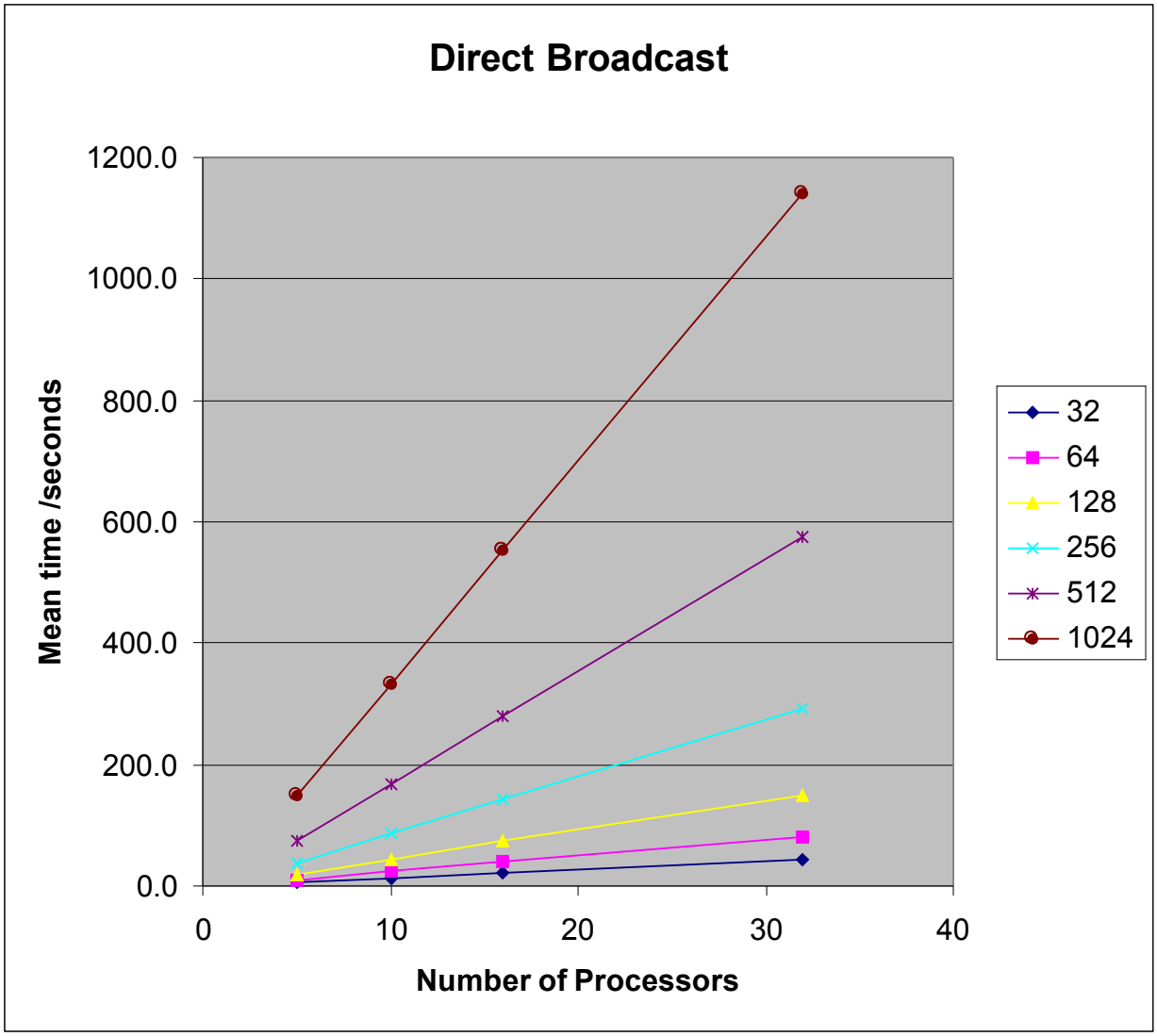
Potentially from this we can extract values fro the parameters g and l. It shows the more processors the greater time to broadcast e.g.

Ratio;		x2		x1.6		x2	
#processors	5		10		16		32
Time for 1024KB:	147.7		331.5		552.1		1139.9
Ratio;		x2.2		x1.7		x2.06	

The models fit almost exactly for all processors the error is under 0.1 seconds. For 32KB its under 0.8 seconds. Because the times are quite large this error is insignificant. An interesting point to make is that increasing the number of processors increases the time to broadcast by a similar ratio.

Ratio;		x2		x1.6		x2	
#processors	5		10		16		32
Time for 1024KB:	147.7		331.5		552.1		1139.9
Ratio;		x2.2		x1.7		x2.06	

So if you plot this data you get the following graph.



Interesting these all fulfil a linear relationship as well.

Its also interesting to note that the timings are a lot more than for the first set of tests on the dcs computers. See the previous results, you can see for 5 processors the time spans 0.09-1.59 seconds and for 16 processors the time spans 0.25 – 6.63 seconds. Whereas on these repeated tests the time spans from 6.15 – 147.7 seconds for 5 processors and 22.4 to 278.6 seconds for 16 processors. The reason for the significant increase in the timing to broadcast for the same code on the same architecture is due to the recompiling of the BSP-lib library. BSP-lib was recompiled to allow a maximum of 32 processors instead of 16, and this coincided with a drop in performance of BSP-lib.

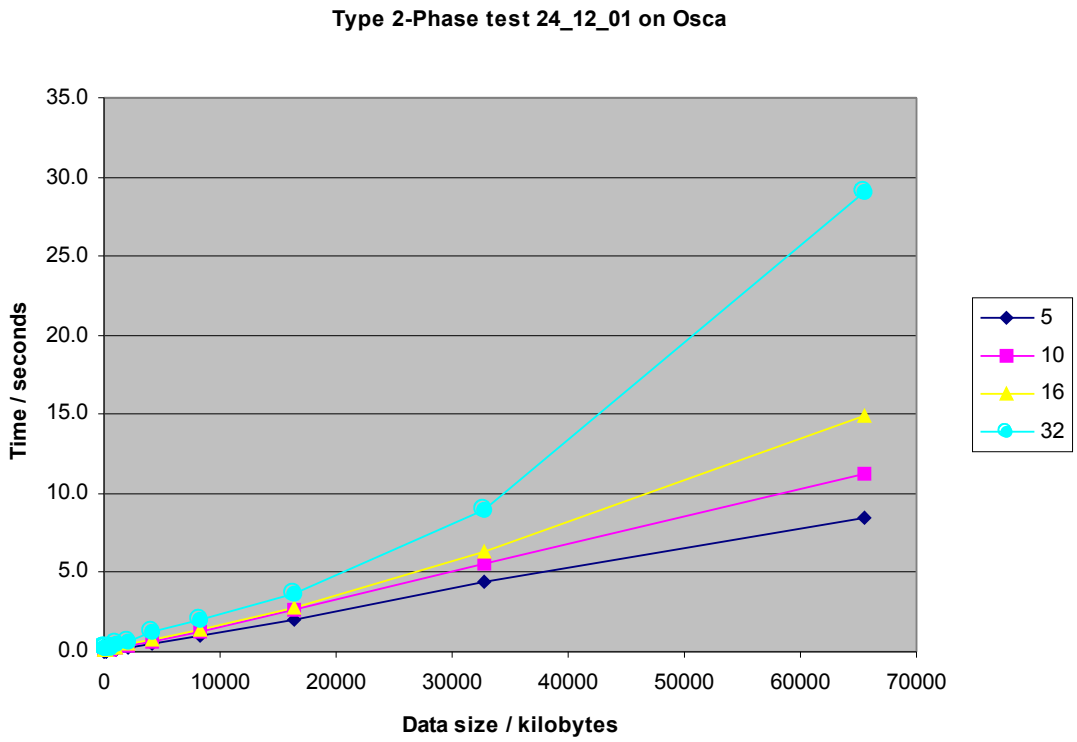
Oscar results

The results from running 2-phase and Tree broadcasts on the Oscar supercomputer can be seen in the following tables:

Test Number:		1				
Broadcast Type:		2-Phase				
			Time /seconds			
Number of Processors	**Data size /Kilobytes**	**Results population**	**Mean**	**Standard Deviation**	**Min**	**Max**
5	32	6	0.026221	0.004075	0.021965	0.033214
5	64	6	0.030999	0.007198	0.020674	0.042821
5	128	6	0.049244	0.017831	0.035106	0.082295
5	256	6	0.061215	0.016825	0.041323	0.090759
5	512	6	0.105579	0.010207	0.096170	0.121749
5	1024	6	0.182385	0.024792	0.140065	0.212970
5	2048	6	0.296570	0.043689	0.242812	0.350609
5	4096	6	0.507133	0.113249	0.406707	0.709326
5	8192	6	1.017317	0.157097	0.742261	1.225138
5	16384	6	2.059879	0.248794	1.782320	2.471447
5	32768	6	4.397077	0.438299	3.868737	4.993014
5	65536	6	8.491864	1.087605	6.565834	9.626013
10	32	6	0.076720	0.036961	0.050763	0.145436
10	64	6	0.067935	0.019546	0.033100	0.088339
10	128	6	0.072432	0.017293	0.052166	0.095692
10	256	6	0.099518	0.012649	0.090963	0.125074
10	512	6	0.138486	0.027146	0.109926	0.185631
10	1024	6	0.254884	0.055293	0.183313	0.352770
10	2048	6	0.377843	0.061270	0.297426	0.461644
10	4096	6	0.651093	0.111686	0.496841	0.802074
10	8192	6	1.308338	0.219450	0.950413	1.521338
10	16384	6	2.639103	0.375169	1.932726	2.934422
10	32768	6	5.540556	0.744395	4.776797	6.919716
10	65536	6	11.262612	1.033422	10.079018	13.035292
16	32	6	0.112904	0.023203	0.094112	0.151417
16	64	6	0.132935	0.017642	0.115798	0.164104
16	128	6	0.126031	0.020180	0.099364	0.157335
16	256	6	0.166656	0.048876	0.126326	0.263815
16	512	6	0.192110	0.017457	0.174080	0.211246
16	1024	6	0.268835	0.036781	0.203949	0.313202
16	2048	6	0.464059	0.050628	0.376426	0.519707
16	4096	6	0.788582	0.068419	0.663965	0.853838
16	8192	6	1.372406	0.148614	1.146046	1.503537
16	16384	6	2.828267	0.317841	2.447637	3.328016
16	32768	6	6.368352	1.437385	4.184222	7.897176
16	65536	6	14.857795	1.714152	11.642253	16.569317
32	32	3	0.210655	0.002827	0.207911	0.213559
32	64	3	0.266023	0.062258	0.194280	0.305869

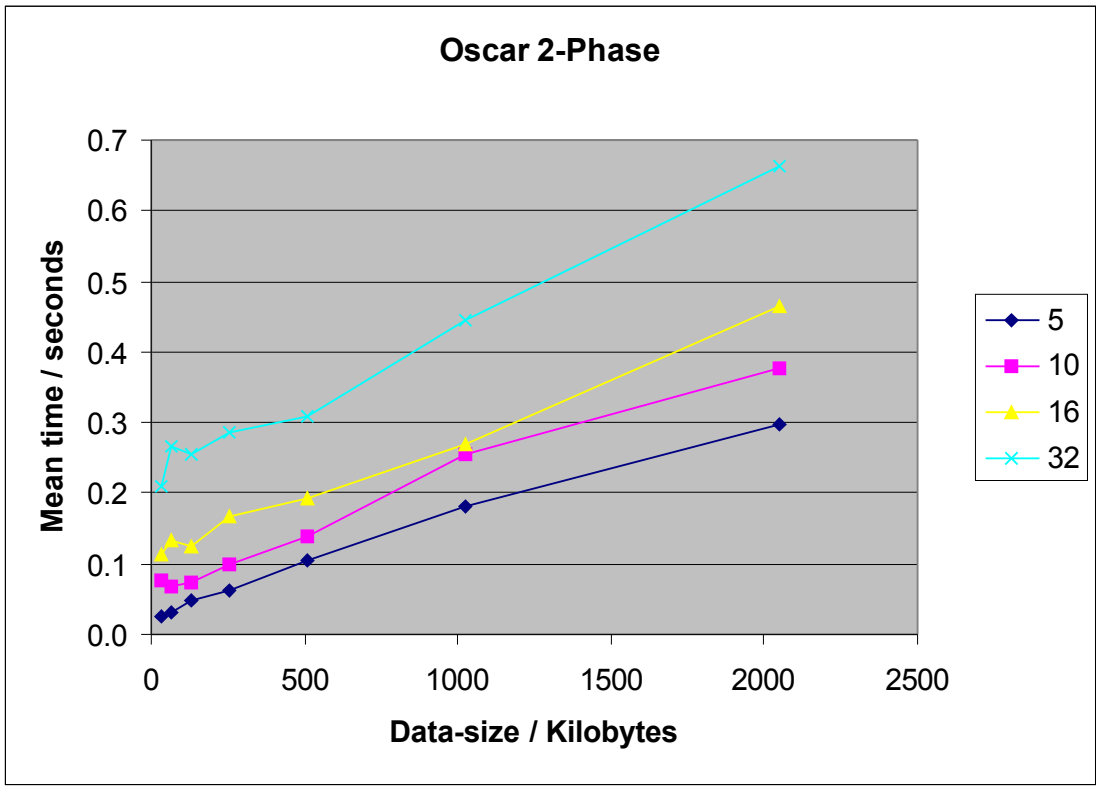
32	128	3	0.256465	0.029281	0.223849	0.280489
32	256	3	0.286169	0.026169	0.257185	0.308062
32	512	3	0.309501	0.024986	0.285786	0.335589
32	1024	3	0.445228	0.039551	0.400225	0.474463
32	2048	3	0.663807	0.094895	0.554236	0.719452
32	4096	3	1.244375	0.054327	1.182610	1.284755
32	8192	3	2.053418	0.078040	1.963319	2.099861
32	16384	3	3.670378	0.059031	3.623904	3.736799
32	32768	3	9.033085	0.157734	8.865194	9.178183
32	65536	3	29.016817	8.772399	20.564938	38.077970

The results for 2-Phase follow the pattern that with increasing number data-size the time to broadcast increases. Also the standard deviation is low, and increases as the data-size increases. Up to 2MB data broadcast the standard deviation is less than 0.05 seconds, which is a variation of up to +/-27% for 256KB. As the data-size increases the variation increases for data-size of 65MB it is 1.1 seconds, but this is a variation of +/-13%.



The above graph shows the data-size plotted versus the broadcast time, for each of the number of processors. Firstly it shows that with increases number of processors the

time to broadcast increases, as would be expected. It appears that for smaller data sizes (below 10MB) the broadcasts times are linear, for datasizes > 10MB the broadcast times on 5 and 10 processors increases linearly. For 16 processors there is a slight deviation from this relationship, with a slight curve, so the broadcast times for 32MB and 64MB are slightly larger than expected. For 32 processors the experimental results give a straight line up to 16MB, above 16MB there is an obvious curve, with the broadcast time for 32MB and 64MB much larger than would have been expected.



The above graph shows the smaller data-sizes of 2MB and less, the graph shows for each processor the data fits a linear relationship. There are a few anomalies for data-sizes less than 256KB. For example for 32 processors the time to broadcast jumps from 0.21+/-0.0028seconds for 32KB to 0.266+/-0.062seconds for 64KB and then falls to 0.256+/-0.029seconds for 128KB. A similar pattern can be seen for these data sizes on 16 processors. For 10 processors 32KB is broadcast in 0.077+/-0.04 seconds and then the broadcast time falls to 0.068+/-0.02 seconds for 64KB. Firstly for broadcast times this small there is little difference between the timings, and as such

for the smallest data-sizes the times are not as reliable as for the larger data sizes. Secondly there is a large variation in the data values that are slightly larger than expected, and it is negligible the amount they vary from what is expected.

As discussed previously for parallel broadcast on the cluster of workstations a linear relationship is expected between data-size and time to broadcast. Again on the Oscar supercomputer this is not what we find, we find a curve on the graph, which suggests the relationship is of polynomial time order. Because the Oscar supercomputer is a BSP computer, and can be accurately modelled by the BSP equations it indicates that the either the experimental results are wrong or the coding of the broadcast algorithm is wrong. There are many repetitions of the results so I believe they give an accurate reflection of the programs performance. However the biggest question lies over the code itself, the actual time measure is made up of just the broadcast routine for the set number of integers, but in this routine there are some local computations that mean we are not getting a time for the pure broadcast;

1) The destination into where the integers will be broadcast is registered in one superstep. This should perhaps be included in the calculation as it costs a small amount of communication cost for the super-step, I do not know the time cost for this, I will predict it is no more than broadcasting an integer, therefore is a constant (k_0 = 2words*g+l).
2) 2 statements, which include 2 assignments, 2 division/multiplications, 2 subtractions/additions and 1 function call to get the processors id name. I believe this is constant time per BSP computation (k_1).
3) During the 1st super-step of broadcast processor carries out a for loop. It loops p times, and a bsp_put is called, and 2 addition/subtractions and 4

multiplication/divisions and one if statement. This is thus a constant times p (k_2p) local computations.

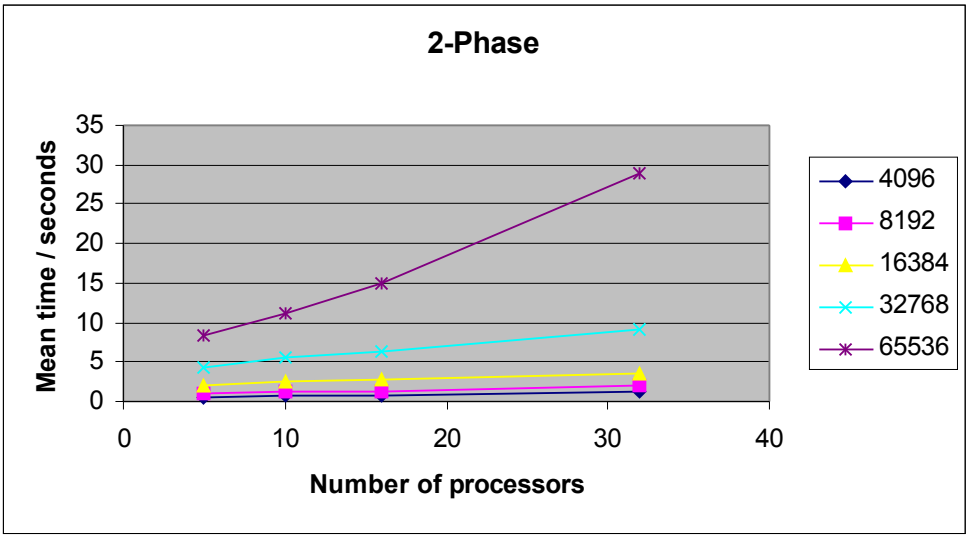
4) During super-step 2, each processor goes through a for loop p times, and carries out a put call. For each loop there are 2 addition/subtractions and 4 multiplication/divisions, 1 if statement, and 2 function calls to get the number of processors and processor 1d. As such there is a constant p times on each processor (k_3p).

Therefore to the equation $S = 2Ng + 2l$, I believe the first super-step contributes $2g + l$. And the local computations contribute $k_1+k_2p+k_3p$, So the overall equation for 2-phase is;

$S = k_1 + p(k_2+k_3) + 2(N+1)g + 3l$

What this means is that for any number of processors there is should be a slight increase in the timings compared to what the BSP model predicts, this is by $k_0 + k_1 + p(k_2+k_3)$. However this is a constant for any data-size. For increasing number of processors, it has an increasing contribution to the broadcast times.

To get the non-linear relationship, as we are getting in the above graph we would expect that is an extra term that is increasing relative to the data-size to be broadcast, but the code doesn't actually show this. This would suggest that the Oscar super computer on which tests where carried out does not necessarily fit the BSP model



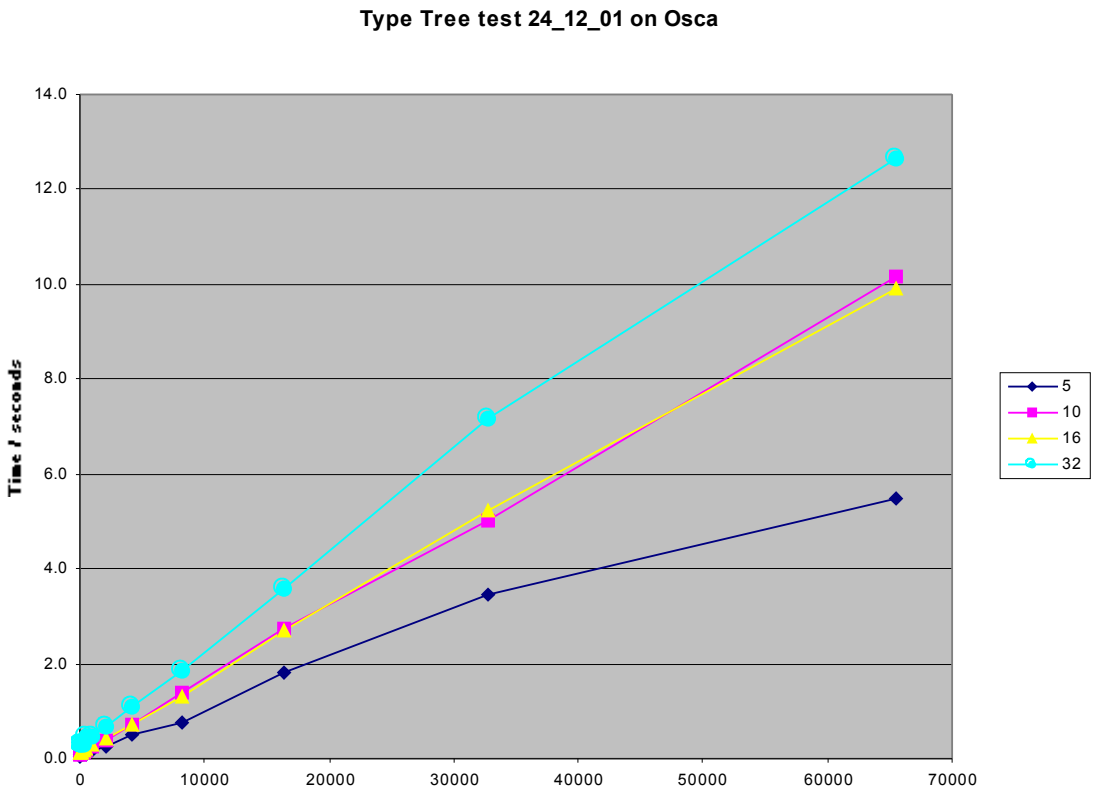
exactly; BSP assumes that the parameters of g and l are constant for a fixed number of processors, but we are seeing that with increasing data size there is an additional term increasing which is probably from the performance of the network degrading as we increase the data-size to broadcast.

The above graph confirms that the broadcast times are increase linearly with the number of processors. Basically the additional local computational terms are a function of the number of processors, as are the parameters g and l, the function being linear, and as such combined they give a linear time graph. The parameters g and l are also functions of the data-size as well.

Test Number:		1				
Broadcast Type:		TREE				
			Time /seconds			
Number of Processors	Data size /Kilobytes	Results population	Mean	Standard Deviation	Min	Max
5	32	6	0.024251	0.007455	0.018154	0.038387
5	64	6	0.067879	0.078309	0.024527	0.225348
5	128	6	0.064101	0.024128	0.028974	0.088472
5	256	6	0.076405	0.024007	0.052881	0.119302
5	512	6	0.080275	0.024466	0.057683	0.125726
5	1024	6	0.156454	0.022193	0.121438	0.174534
5	2048	6	0.260337	0.031414	0.210134	0.293670
5	4096	6	0.504599	0.082642	0.433535	0.659159
5	8192	6	0.775728	0.158901	0.586945	0.958606
5	16384	6	1.833675	0.229139	1.537354	2.162178
5	32768	6	3.460073	0.325175	3.028974	3.832838
5	65536	6	5.496548	0.775166	4.571277	6.554086
10	32	6	0.068361	0.017429	0.058582	0.103410
10	64	6	0.094543	0.046550	0.042307	0.161847
10	128	6	0.109345	0.045297	0.082332	0.200633
10	256	6	0.145979	0.044560	0.103125	0.220394
10	512	6	0.181249	0.046675	0.110926	0.233624
10	1024	6	0.273725	0.027106	0.242395	0.306704
10	2048	6	0.394229	0.039658	0.330514	0.441474
10	4096	6	0.715935	0.074673	0.610991	0.808508
10	8192	6	1.389207	0.101773	1.264367	1.535222
10	16384	6	2.747427	0.440341	2.330267	3.346606
10	32768	6	5.001431	0.588765	4.365764	5.928399
10	65536	6	10.182964	1.129071	8.623834	11.749302
16	32	6	0.140847	0.091205	0.063841	0.318677
16	64	6	0.140031	0.065184	0.093709	0.270584
16	128	6	0.132142	0.044258	0.076852	0.206752
16	256	6	0.161327	0.026364	0.129332	0.204647
16	512	6	0.191124	0.023140	0.164344	0.232180
16	1024	6	0.303888	0.040026	0.256274	0.368060
16	2048	6	0.421648	0.049737	0.347487	0.467174
16	4096	6	0.700671	0.115676	0.506898	0.856564
16	8192	6	1.327377	0.089230	1.231866	1.456102
16	16384	6	2.695816	0.512325	2.213976	3.666296
16	32768	6	5.224560	0.429176	4.753242	5.802970
16	65536	6	9.926595	0.665298	9.270066	11.156470
32	32	3	0.292017	0.075404	0.230181	0.376020
32	64	3	0.285766	0.096434	0.202280	0.391322
32	128	3	0.282742	0.063071	0.210298	0.325430
32	256	3	0.286766	0.046392	0.252250	0.339502
32	512	3	0.445770	0.075821	0.399082	0.533254
32	1024	3	0.471737	0.030065	0.437998	0.495689
32	2048	3	0.672753	0.062144	0.602377	0.720074
32	4096	3	1.105222	0.146289	1.004301	1.272992
32	8192	3	1.876232	0.058403	1.838969	1.943542
32	16384	3	3.604266	0.662782	3.093513	4.353229

32	32768	3	7.150435	0.808407	6.497822	8.054750
32	65536	3	12.667328	0.572733	12.097796	13.243210

The results for the tree broadcast follow the expected pattern. Broadcast times increase as the data-size increases and the variation of the times increases with the data-size, but the standard deviation is smaller for larger data-sizes than for the 2-Phase test, e.g. for 32 processors, with 64MB broadcast in 12.7+/-0.6 seconds which indicate the results are even more accurate than those for 2-phase broadcast.



The above graph shows the broadcast time versus the data-size broadcast for tree broadcast. The relationship appears to be linear for 10, 16 and 32 processors. For 5

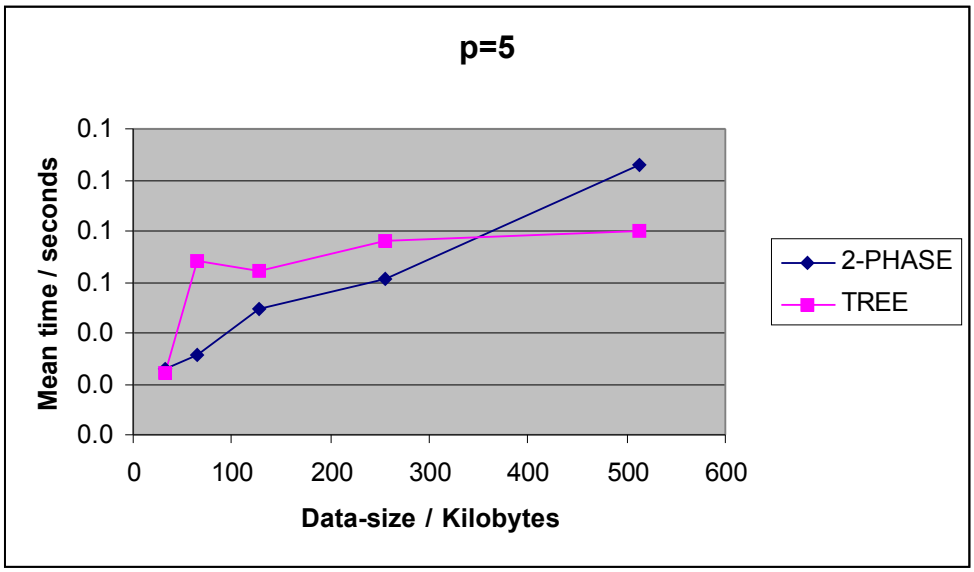
processors, the time to broadcast seems to tail off, however these results can still be approximated with a straight line.

There is a strange results that it appears 64MB is broadcast quicker 16 processors than 10 processors. The above results table shows that this occurs for all data-sizes greater than 2MB, for example 8MB is broadcast in 1.389seconds on 10 processors, and 1.33seconds on 16 processors. 32MB is broadcast in 5.00seconds on 10 processors and 5.22seconds on 16 processors. If you look at the full data range all the timings for 10 and 16 processors are very close, the standard deviation shows that the results vary over the same range. It would be interesting to do further tests to see if this is repeated. Potentially this is because for 10 and 16 processors there is the same number of super-steps ($\log_2 10 = 4$, $\log_2 16 = 4$), so the communication and synchronisation costs will very close, which would explain this anomaly.

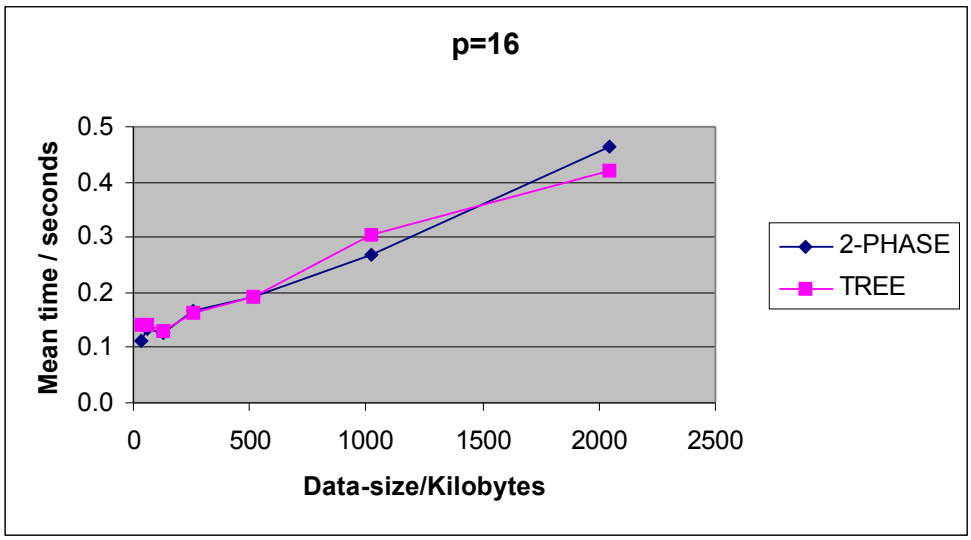
We can be reasonably confident with these results; the timing information is made up of just the broadcast routine. Again there is an extra super-step in which the data-type in which the data is broadcast into is registered. Then there are several local computations, which are adds/mults/ifs and there are a constant number each time the routine is run, so this is constant initialisation time. There are two code optimisations which may be the reason why the timings are linear relative to the data-size. A memory copy is carried out if the data is being sent to the same processor, and secondly the high performance put primitive is being used instead of the buffered put primitive which is used for the 2-phase broadcast. This could be what is causing the unexpected curved appearance in the graph for 2-phase broadcast.

The experimental results show the general trend that 2-Phase broadcast is better for smaller data-sizes, and the tree broadcast is better for larger data-sizes. For 5 processors for 256KB 2-phase takes 0.061s, tree takes 0.076s, then for 512KB 2-

phase takes 0.106s and tree takes 0.080s. For 10 processors the cut off point is 32MB, for 16 processors the cut off point is 2MB, up to which 2-phase is better. For 32 processors it is up to 4MB.



The above graph shows an example of this cut off point for 5 processors, being 256KB, for data-sizes of 256KB and less 2-Phase is better, for data-sizes greater than 256KB Tree is better. This is apart from for 32KB, where tree is slightly better than 2-Phase.



This is seen for all the number of processors on 16 processors, the timings are very close for small data-sizes. For 16 processors 2-phase and tree exchange being the most efficient for 1MB 2-phase is better for 2MB tree is better, then for 256KB Tree is better, for all the other between 256-2MB 2-phase is better < 256KB 2-phase is better.

Comparing the 2-Phase method with Tree running on the Oscar supercomputer, the tree method therefore appears to be better for larger data sizes, this varies depending on how many processors the code is running on. This is an unexpected result, however it can largely be explained by the different use of the bsp_put primitive. Tree uses the high performance non-buffered version, and as such is much more memory efficient. I believe the extra term in the 2-phase broadcast which is increasing the broadcast times is due to this buffering affect. It would be expected that if the test were run using the high performance 2-phase broadcast the graphs would show the linear relationship as suggested by the BSP model.

BSP predictions

Broadcast Type:			TREE						
# of Procs	ceil ($\log_2$ p)	Data size /Kilo-bytes	s / (MFLOPS /second)	g / (FLOPS/ word)	l / FLOPS	Mean time / seconds	BSP predicted time / seconds	Diff.	% error
5	3	32	100.7	11.06	2474	0.024251	0.002773	0.02	88.6%
5	3	64	100.7	11.06	2474	0.067879	0.005472	0.06	91.9%
5	3	128	100.7	11.06	2474	0.064101	0.010871	0.05	83.0%
5	3	256	100.7	11.06	2474	0.076405	0.021667	0.05	71.6%
5	3	512	100.7	11.06	2474	0.080275	0.043261	0.04	46.1%
5	3	1024	100.7	11.06	2474	0.156454	0.086448	0.07	44.7%
5	3	2048	100.7	11.06	2474	0.260337	0.172823	0.09	33.6%
5	3	4096	100.7	11.06	2474	0.504599	0.345573	0.16	31.5%
5	3	8192	100.7	11.06	2474	0.775728	0.691072	0.08	10.9%
5	3	16384	100.7	11.06	2474	1.833675	1.382070	0.45	24.6%
5	3	32768	100.7	11.06	2474	3.460073	2.764066	0.70	20.1%
5	3	65536	100.7	11.06	2474	5.496548	5.528058	-0.03	-0.6%

The above table shows the times calculated for the tree broadcast based on the parameters for the Origin 2000 machine from the BSP parameter database. It only goes up to 7 processors, so I've done the calculations for 5 processors. As you can see the predictions get closer the greater the data-size to broadcast. For 23KB there is an 88.6% error, 0.0027s is predicted and an average time of 0.024 seconds was underachieved. The model underestimates the time to broadcast. For 8MB the broadcast time was 0.776, the prediction is 0.691 seconds, which is just a 10.9 % error. The larger the data size the closer the prediction, for 64MB the broadcast time was 5.5seconds, with an error of 0.6%. This shows that the code is not optimal, and the local computations that occur do contribute to the running time of the program, for smaller data sizes they contribute a lot, for larger data-sizes the affect of these local computations is minimal. The difference between the experimental times and the predicted times increases with the data-size, this difference could be due to the initialisation of the broadcast routine. However it is may be that the parameters are

not an exact reflection of the actual parameters for the Oscar supercomputer, from the experimental data we can derive values for the parameters.

Parameter derivation

From the tree broadcast experimental results we get the following equations;

p = 5, $t = 4.9807 \times 10^{-3} \times N - 0.0995853$
p = 10, $t = 6.4912 \times 10^{-3} \times N + 7.9197 \times 10^{-3}$
p = 16, $t = 3.7269 \times 10^{-3} \times N + 0.19764633$
p = 32, $t = 3.7269 \times 10^{-3} \times N + 0.19764633$

So in form mx + c

$c = \frac{(\log p)\, l}{S}$, so $l = c\, S / (\log p)$

$m = \frac{(\log p)\, g}{S}$, so $g = m\, S / (\log p)$

x = N/256 (to convert words to kilobytes)

And assuming S = 100.7 x 10^6 FLOPS. Applying the equation gives us the following values for g and l:

Tree

Broadcast Type:			TREE			
# of Procs	**ceil ($\log_2$ p)**	**S**	**m / (seconds/kilobyte)**	**c / seconds**	**g = m S / (256 x log p)**	**l = c S / (log p)**
5	3	100.7E+06	8.6848E-05	1.18668E-01	11.387	3.98330E+06
10	4	100.7E+06	1.5344E-04	9.97165E-02	15.089	2.51036E+06
16	4	100.7E+06	1.5077E-04	9.99771E-01	14.827	2.51692E+07
32	5	100.7E+06	1.9261E-04	9.98808E-01	15.153	2.01160E+07

The above table shows the calculation of the g and l parameters based on the experimental results for tree broadcast. The value for g give by the parameter database is 11.06 FLOPS/word for 5 processors; experimentally g was 11.387 FLOPS/word,

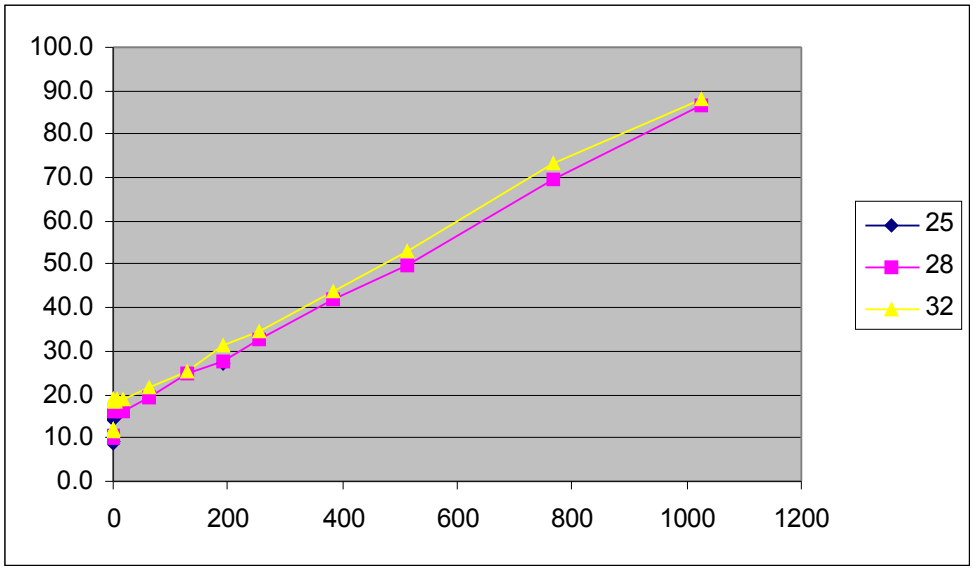
which is very close. There are no values for the parameters for the number of processors greater than 7. The values for the parameter l are not very close to the values quoted. For 5 processors l is 3.983 x 10^6 FLOPS experimentally, whereas the parameter database predicts a value of 3867 FLOPS. This may be due to nature of the experiment, the influence of an additional super-step and the cost of local computations.

High Performance 2-Phase Broadcast

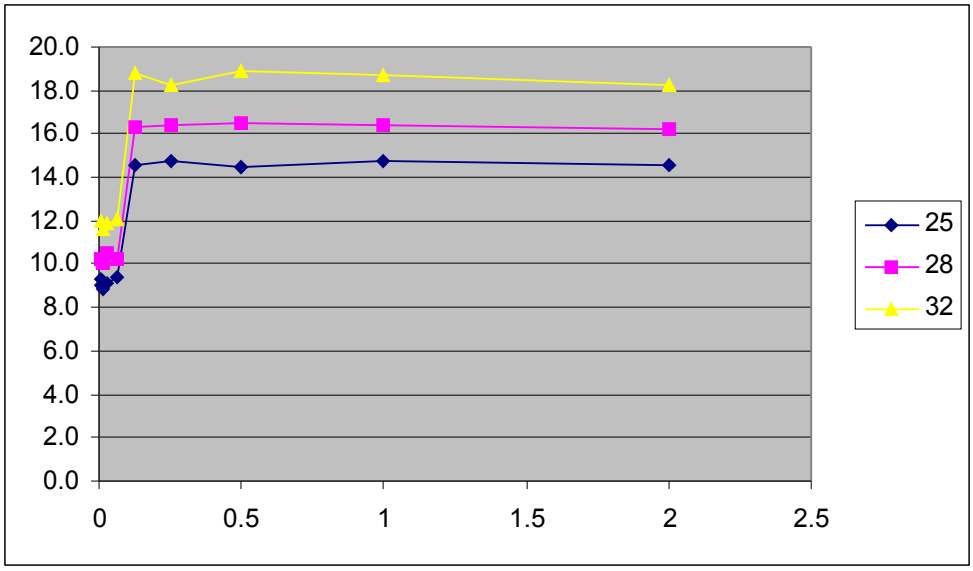
Test Number:			1			
Broadcast Type:			hp_2phase			
				Time /seconds		
Number of Processors	**Data size /Kilobytes**	N	**Results population**	**Mean total**	**Mean s1**	**Mean s2**
25	0.00390625	1	10	9.276172	3.011900	6.264271
25	0.0078125	2	10	9.065058	2.976490	6.088568
25	0.015625	4	10	8.875991	2.902829	5.973162
25	0.03125	8	10	9.099346	2.923553	6.175793
25	0.0625	16	10	9.436352	2.893470	6.542882
25	0.125	32	10	14.588146	6.375605	8.212542
25	0.25	64	10	14.787744	6.545176	8.242569
25	0.5	128	10	14.478328	6.366256	8.112072
25	1	256	10	14.717164	6.480327	8.236837
25	2	512	10	14.602264	6.406199	8.196065
25	192	49152	10	27.047601	9.766904	17.280697
28	256	65536	10	32.685620	11.819120	20.866500
28	384	98304	10	41.807176	14.032045	27.775131
28	512	131072	10	49.984990	16.232860	33.752130
28	768	196608	10	69.452770	21.402352	48.050418
28	0.00390625	1	20	10.253344	3.291344	6.962000
28	0.0078125	2	10	10.273329	3.262915	7.010414
28	0.015625	4	10	10.057448	3.270956	6.786492
28	0.03125	8	10	10.550571	3.309067	7.241504
28	0.0625	16	10	10.189140	3.201653	6.987487
28	0.125	32	10	16.273355	7.195607	9.077748
28	0.25	64	10	16.408416	7.347152	9.061264
28	0.5	128	10	16.543485	7.277025	9.266461
28	1	256	9	16.425524	7.286826	9.138698
28	2	512	10	16.240510	7.084491	9.156019
28	4	1024	10	16.725745	7.431902	9.293843
28	8	2048	10	16.893945	7.665404	9.228541
28	16	4096	10	16.242564	7.150392	9.092172
28	64	16384	10	19.167969	7.939524	11.228445
28	128	32768	10	24.726687	9.449196	15.277491

28	192	49152	10	27.449715	10.360614	17.089101
28	256	65536	10	32.685620	11.819120	20.866500
28	384	98304	10	41.807176	14.032045	27.775131
28	512	131072	10	49.984990	16.232860	33.752130
28	768	196608	10	69.452770	21.402352	48.050418
28	1024	262144	10	86.649378	26.066723	60.582656
32	0.0078125	2	9	11.968899	3.685533	8.283366
32	0.015625	4	10	11.592116	3.583650	8.008466
32	0.03125	8	10	11.845251	3.678703	8.166548
32	0.0625	16	10	12.084025	3.705850	8.378175
32	0.125	32	10	18.775649	8.527955	10.247690
32	0.25	64	10	18.260518	8.013729	10.246790
32	0.5	128	10	18.848687	8.508574	10.340113
32	1	256	10	18.697758	8.376075	10.321683
32	2	512	10	18.281658	7.912004	10.369654
32	4	1024	10	18.629047	8.274529	10.354518
32	8	2048	10	18.791349	8.347119	10.444229
32	16	4096	10	18.703896	8.339653	10.364243
32	64	16384	10	21.693639	9.317962	12.375677
32	128	32768	10	25.414498	10.557510	14.856988
32	192	49152	10	31.508677	12.281234	19.227442
32	256	65536	10	34.779502	13.303316	21.476186
32	384	98304	10	43.816580	16.603557	27.213023
32	512	131072	10	52.796656	19.064538	33.732118
32	768	196608	9	73.456648	24.864181	48.592467
32	1024	262144	10	87.931969	30.070793	57.861177

The results give the expected relationship of broadcast time increasing with data-size.



The above graph shows that the 2-Phase broadcast is now behaving as predicted by the BSP model. Also the greater the number of processors, the greater the broadcast time.



For small data-sizes, for example 2.5KB and less the broadcast times stay almost level at about 15s for 25 processors, 16.5s for 28 processors and 18.5s for 32 processors.

These results show that the cluster of AMD PCs does in fact act as a BSP computer, and can be modelled by the BSP mathematical analysis. The only difference is using the unbuffered bsp_put primitive. Thus the buffered bsp_put was causing the anomalies we found in the results for previous tests. Although informally it is suggested that the standard BSP primitive will cause some performance degragation, there is no discussion of the extent of its impact of broadcast times.

Parallel Matrix Multiplication

Theory

Investigation into the parallel broadcast problem went really well, I then went on to look at parallel matrix multiplication. Initially I looked at the All pairs shortest path problem, I soon found that matrix multiplication was a key component of these algorithms, and so decided to first investigate matrix multiplication, the work carried out on All pairs shortest path can be found in the appendix. During this part of the project I did not make as much progress as I had planned for. I spent time researching this. There are two main methods 2D and 3D matrix multiplication, the question being which one is optimal.

During my research I found many other different algorithms for matrix multiplication. I began by implementing the sequential matrix multiplication; this would act as a control. The next step was to implement the 1D matrix multiplication, which allowed me to develop the appropriate routines that could be used for higher order matrix multiplication.

7.1.1 Sequential Matrix Multiplication

Sequential matrix multiplication was implemented. I wanted to use this as a control to compare the performance of sequential matrix multiplication and parallel matrix multiplication. From this I generated some reliable test data that was used to verify the output from parallel matrix multiplication.

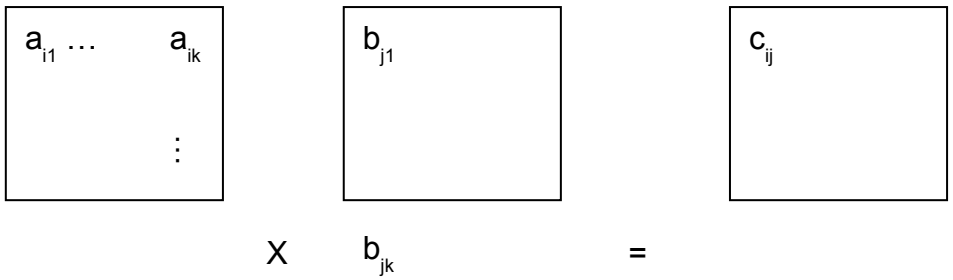
Matrix multiplication is based on the following pseudo code (see algorithms book):

```
Matrix-Multiply(A, B)
        If colums[A] = [B]
                Then error "incompatible dimensions"
                Else for i ← 1 to rows[A]
                        Do for j ← 1 to columns[B]
                                Do C[i,j] ← 0
                                        For k ← 1 to columns[A]
                                                Do C[I, j] ← C[i,j] + A[i, k].B[k,j]
                        Return C
```

i.e. for each I, j = 1, 2, … n we calculate:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

Where:



7.1.2 The Sequential matrix multiplication code

The design decision was made to store the matrices as 1D arrays. For sequential matrix multiplication, or naive method, the routines written for the local multiplications for parallel matrix multiplication were used. The code can be found in the appendix “matrix_seq.c”.

The routines are were:

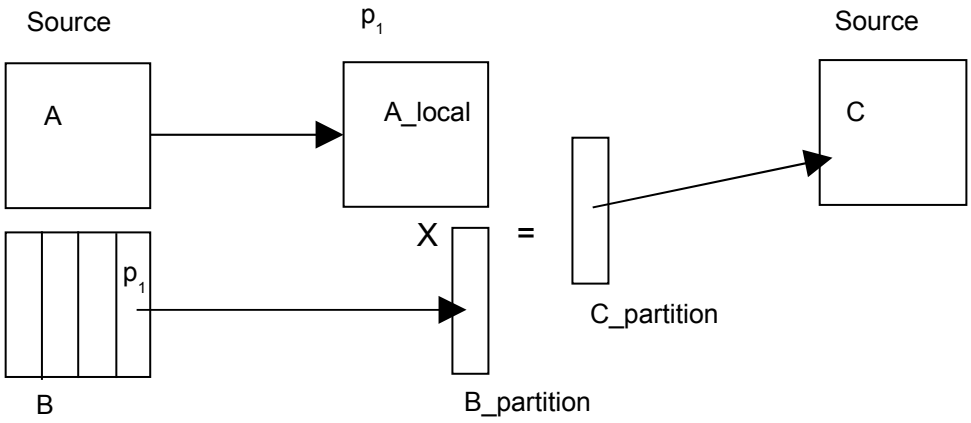
1) createMatrix(#rows, #cols) – Creates the 2D matrix as a 1D array
2) populateMatrix(#rows, #cols, matrix, row_seed, col_seed) – Populates a matrix with data.
3) multiplyMatrices(#rowsA, #colsA, #rowsB, #rowsA, A, B) – Multiply two matrices using the naive method.

The only complication is calculating the index numbers of the array, based on indexing values in the 2D matrix. To calculate the appropriate index the following equation is used:

Index = row_index * #cols + col_index

Note “#cols”, the number of columns, is equivalent to the size of the row.

7.1.3 1D Matrix Multiplication



1D matrix multiplication involves the 1D decomposition of the solution matrix, making each processor responsible for a column partition of the solution C. Each processor receives a column partition of the B matrix, and receives an entire copy of the A matrix. Locally the matrices are multiplied using the naive multiplication method. The solution partition that the processor has calculated locally (C_partition) is then returned to the source processor.

7.1.4 1D Matrix Multiplication code explained

The 1D matrix Multiplication algorithm is implemented as the code “Matrix1dpar.c”, which can be found in the appendix.

Super-step 1 and 2

During the first super-step matrices A and B are populated with data. Then locally on each processor, matrices are created. Each processor receives all of matrix A using the getMatrixPartition(). During the next super-step the B matrix is partitioned (using the setPartitionIndices(), which calculates the partition indices), and then sent to each of the processors, using the getMatrixPartition() routine.

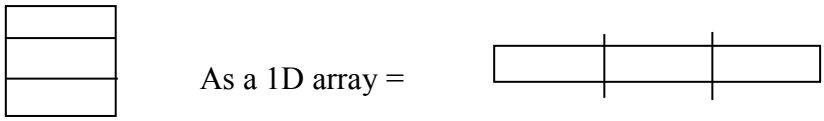
Super-step 3

During the next super-step a sequential matrix multiplication is carried out locally on each processor and a partition of the C matrix is created. Using the putMatrixPartition routine each processor puts the its partition of the C matrix on to the source processor.

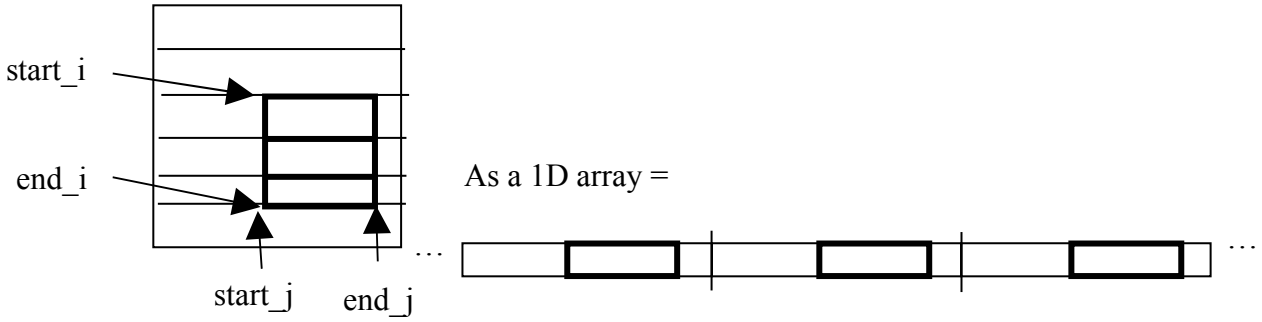
7.1.5 Communicating the matrices

The matrices are communicated using a series of gets and puts primitives by the two routines getMatrixPartition() and putMatrixPartition(). PutMatrixPartition() is a wrapper method for bsp_put. The matrices are communicated by splitting them up into a several 1D arrays, which are then communicated by a series of puts. This is because the matrix doesn't actually lie in contiguous memory. As such there is a put for each row of the matrix partition into the memory of the destination processor. There is a put primitive called for every row of the matrix to be communicated, the diagram below demonstrates how these routines work.

Partition of the matrix locally is represented as a 1D array:

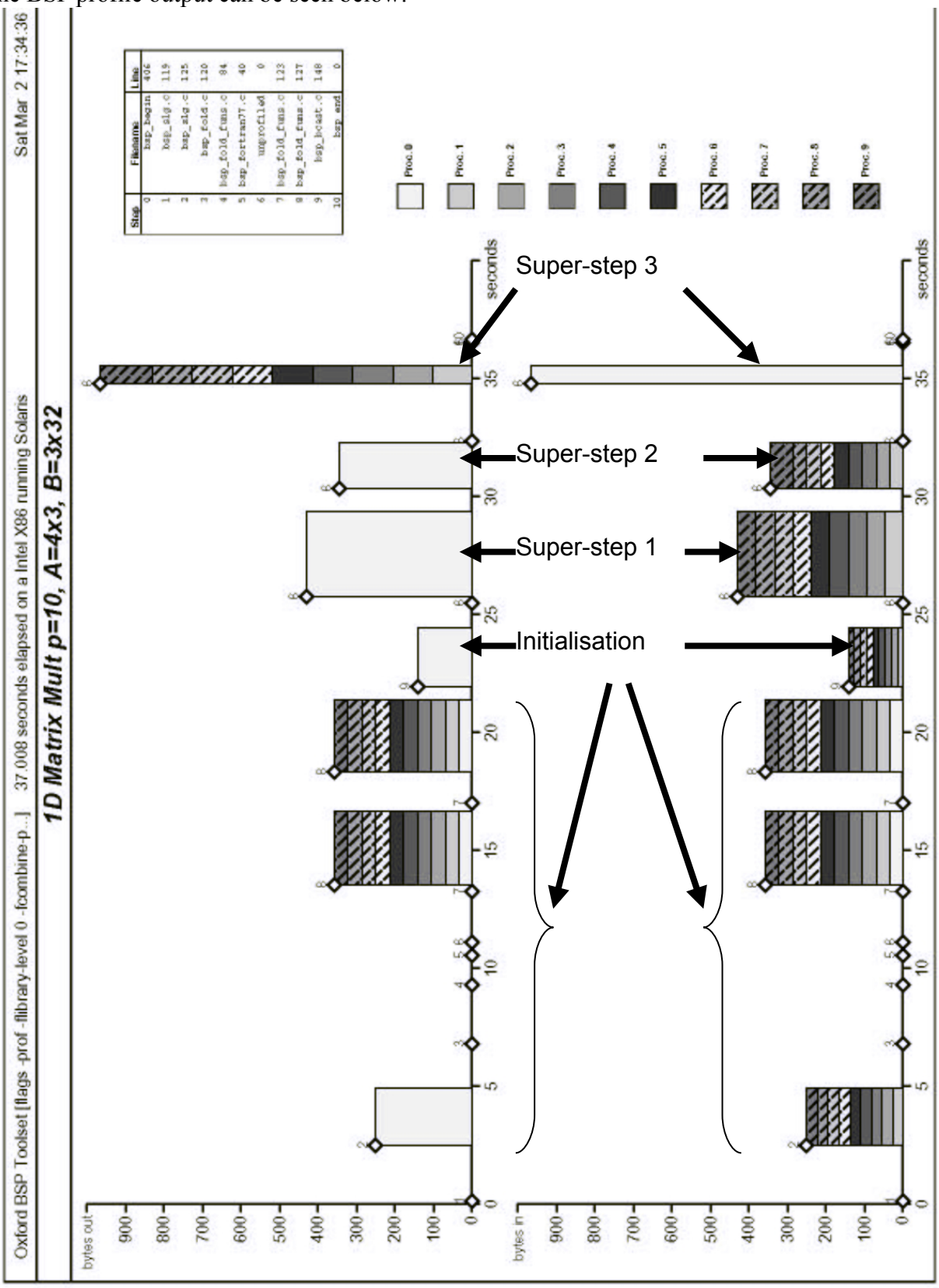


The full matrix is on the destination processor is like this:



7.1.6 1D Matrix Multiplication BSP Profile

The BSP profile output can be seen below:



This shows the 1D matrix multiplication on 10 processors multiplying A, a 4x3 matrix, by B a 3x32 matrix. All peaks before step 1 are initialisation cost. During step 2 matrix A is broadcast to each processor, so 148bytes (4x3x4bytes) are sent to each processor. In step 2 a partition of matrix B is sent to each processor. In this case B has 32 columns and therefore each processor receives a partition of 3 columns (32/10 = 3r2), so in total 32x3x4 = 384 bytes are sent by processor zero. Each processor then receives a 3x3 matrix apart from the last matrix, which receives a 3x5 matrix, which can be seen on the profile. In step 3 each processor sends its matrix partition back to the source processor. In total about 1 Kilobyte of data is sent in total.

7.1.7 Further work

I carried on and developed several other routines to make higher order matrix multiplication. Experiments were carried out on 1D matrix multiplication and sequential matrix multiplication see results for this.

This allowed me to develop appropriate routines that would be required for the higher degree matrix multiplication. This included how to store a matrix as a 1D array, how to store partitions, how to partition a matrix row-wise, column-wise and block-wise, and how to broadcast a matrix or partition to all other processors.

The code for 2D matrix multiplication has been included in the appendices, though it was not completed.

Experimental Method

Firstly I used the sequential matrix multiplication to create several matrix multiplication calculations in order to verify the data produced by parallel matrix multiplication. The verification data can be found in the appendix.

The matrix multiplication was then run sequentially, at this point DCS was not accessible, so the sequential matrix multiplication was run on my home PC to get some data to compare with the experimental data for 1D parallel matrix multiplication.

I then developed test scripts to test the 1D matrix multiplication. I chose the number of processors to use as 4, 16, 9, 25; they therefore all have integer square routes. This was forward planning to enable me to compare the results with 2D matrix multiplication.

The following table shows the data sizes I chose:

n=z=m	23	32	45	64	90	128	181	256
#ints (n^2)	529	1024	2025	4096	8100	16384	32761	65536
Data / bytes	4232	8192	16200	32768	64800	131072	262088	524288
Data / Kilobytes	4.133	8.000	15.820	32.000	63.281	128.000	255.945	512.000
Data / Megabytes	0.00	0.01	0.02	0.03	0.06	0.13	0.25	0.50

The table above shows the data-sizes used in the test. The dimensions was set to be equal, to allow all the data to be compared easily. The table shows the number of integers that will be broadcast in each super-step, and then how much data will be communicated. This ranges between 4Kilobytes and 512Kilobytes.

The results and analysis from this test can be found in the following section.

Results and Analysis

Preliminary results

p	M	z	n	Number of Integers per superstep	Kilobytes	Time / seconds
	4	5	7	4	31.5	0.25	4.259661
	4	18	16	12	240	1.88	7.610837
	1	500	700	400	315000	2460.94	43
	1	500	700	400	315000	2460.94	43
	1	500	500	500	250000	1953.13	38
	1	5000	500	500	1375000	10742.19	420
	4	5	5	5	25	0.20	3.9
	4	5	5	5	25	0.20	4.0
	4	4	4	4	16	0.13	3.7
	4	4	4	4	16	0.13	3.9
	4	4	4	4	16	0.13	3.467294
	4	40	40	40	1600	12.50	14.6
	4	40	40	40	1600	12.50	14.5
	4	400	400	400	160000	1250.00	193.6
	4	400	400	400	160000	1250.00	193.5
	9	40	40	40	1600	12.50	37.7
	9	40	40	40	1600	12.50	38.4
	16	40	40	40	1600	12.50	69.8
	16	40	40	40	1600	12.50	71.2

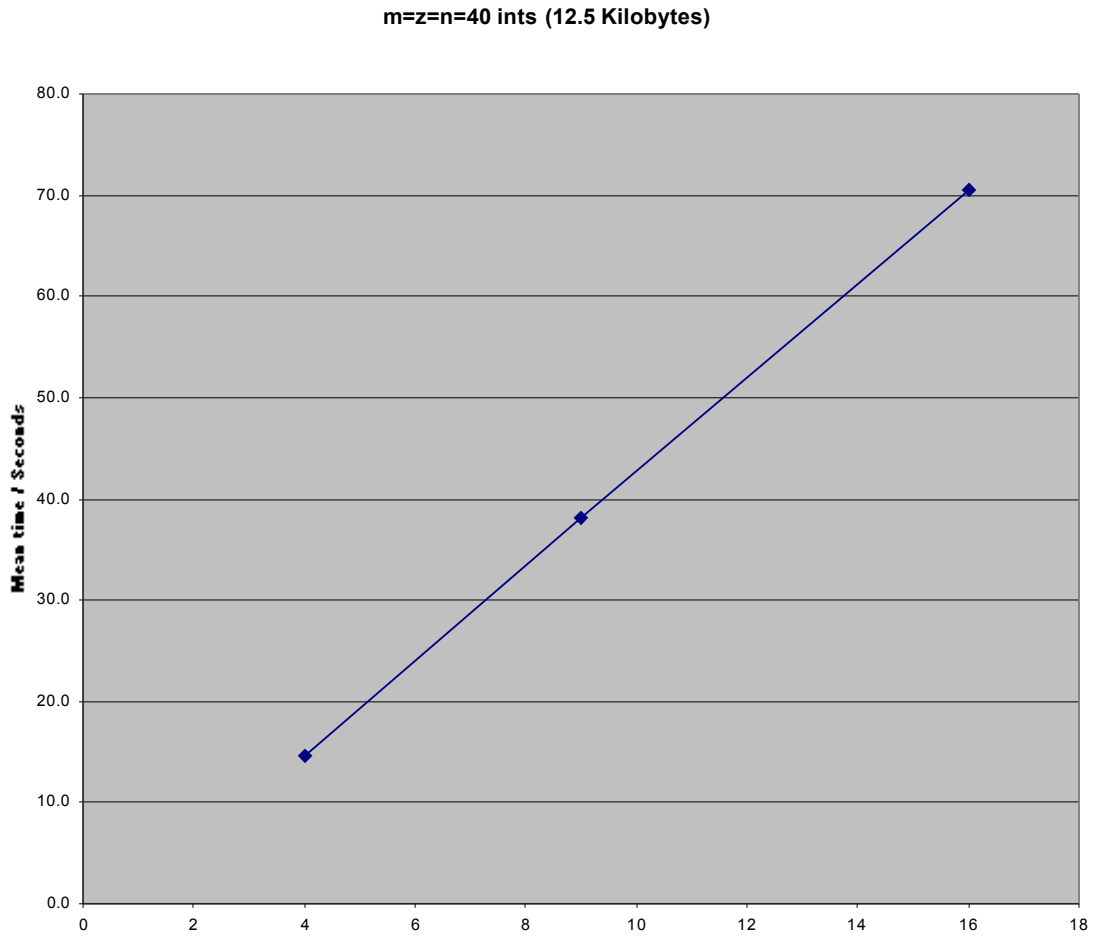
Its difficult to compare these results for sequential multiplication as there isn't a broad enough set of results. However on 1 processor to multiply two 500by500 matrices took 38 seconds, on 4 processors to multiply two 400by400 matrices took 193.6 seconds. This would indicate that the sequential code is better than 1D parallel. Firstly there is not enough data here to make any final conclusions, secondly the code is not optimised, and thirdly the timing does not exactly reflect what we are trying to measure.

The timing data is for the entire matrixMultiplication1D routine, as such this incurs the setting up of the data which populates the matrices with data, which takes n time steps, where n is the amount of data. Then the partition sizes are calculated which is a

couple of additions, multiplications and divisions, and is a constant number of time steps. This also includes the time it takes to register and broadcast matrix A and partition matrix B to each processor, which takes 2 super-steps. Then the matrices are multiplied locally this takes n*n*(n/p) time to process or n^3/p. Finally there is another super-step to communicate the matrices back to the source processor. So it is the super-step in which the matrices are multiplied locally that is what should have been measured. As such a lot of the time is made up from the constant time for initialisation steps, and time to broadcast the matrices between processor.

From the table there isn't enough data to be able to reliably compare all the data, however for a matrices dimensions of m=z=n=40 the test was run on 4 9 and 16 processors. This data can be summarised in the table below:

p	m=z=n	Data per superstep / Kilobytes	average time
4	40	12.5	14.6
9	40	12.5	38.05
16	40	12.5	70.5

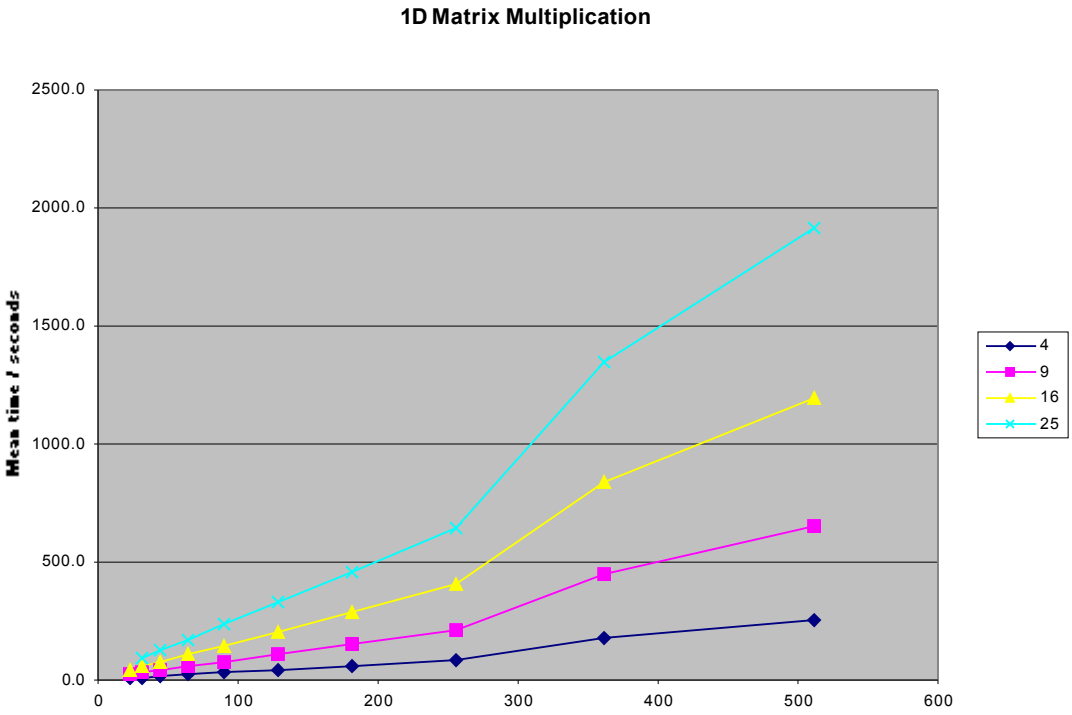


What this shows is a linear relationship, that with increasing number of processors the time to multiply increases. This is not what is expected, increasing the number of processors working on matrix multiplication should improve the running time. I believe this is due to the affect described above, that the timing includes the time to broadcast the data to all the processors and return the data. And as such the time is mainly made up of the time to broadcast the data, and this would explain why the graph follows a linear relationship as would be expected for broadcasting data.

Data from full test

Test Number:			1				
Multiplication type			1D_matrix_multiplication				
				Time /seconds			
#procs	**n=m=z**	**Data size /Kilobytes**	**Results pop.**	**Mean**	**Standard Deviation**	**Min**	**Max**
4	23	8.3	5	9.331910	0.030931	9.307257	9.370268
4	32	16.0	5	12.127513	0.043217	12.060373	12.160459
4	45	31.6	5	15.993408	0.048811	15.906439	16.020083
4	64	64.0	5	22.047528	0.202897	21.870942	22.312328
4	90	126.6	5	29.877765	0.035204	29.829636	29.921824
4	128	256.0	5	41.594784	0.066485	41.505253	41.671143
4	181	511.9	5	58.459964	0.116828	58.276315	58.596792
4	256	1024.0	5	83.057880	0.092407	82.970573	83.205222
4	362	2047.6	5	174.282712	0.151177	174.113057	174.461364
4	512	4096.0	5	256.495647	0.701098	255.843671	257.596372
9	23	8.3	5	24.472889	0.212341	24.260662	24.739076
9	32	16.0	5	31.695021	0.444683	31.166134	32.244919
9	45	31.6	5	42.244722	0.182185	42.060768	42.435667
9	64	64.0	5	57.394956	0.047364	57.337851	57.464060
9	90	126.6	5	78.996523	0.673901	78.250103	79.847156
9	128	256.0	5	109.837607	0.332348	109.405102	110.325499
9	181	511.9	5	152.243279	0.814116	151.320527	153.357262
9	256	1024.0	5	213.878477	0.526371	213.320791	214.719896
9	362	2047.6	5	452.830840	0.947336	451.471103	454.114122
9	512	4096.0	5	648.872460	1.134164	647.695282	650.405522
16	23	8.3	5	45.130162	0.575968	44.271315	45.892671
16	32	16.0	5	59.092403	0.683560	58.016339	59.920004
16	45	31.6	5	78.277955	0.374693	77.685083	78.722587
16	64	64.0	5	106.559686	0.238772	106.339473	106.943410
16	90	126.6	5	146.885847	0.666032	145.821218	147.605118
16	128	256.0	5	203.797711	0.916815	202.840241	205.060305
16	181	511.9	5	284.631322	1.268439	283.359162	286.219599
16	256	1024.0	5	402.763983	1.674920	400.332916	404.879716
16	362	2047.6	5	842.861272	1.082132	841.141754	844.090001
16	512	4096.0	5	1195.455364	5.349536	1189.384057	1203.580771
25	32	16.0	5	93.991954	0.619031	93.524798	95.065587
25	45	31.6	5	126.845844	0.606434	126.394196	127.873248
25	64	64.0	5	173.233621	0.581268	172.300525	173.856274
25	90	126.6	5	237.214204	1.244648	235.729130	238.661457
25	128	256.0	5	331.867220	1.559992	329.156507	333.179745
25	181	511.9	5	461.659978	0.613316	460.729823	462.417003
25	256	1024.0	5	646.557881	1.436972	644.703713	648.446592
25	362	2047.6	5	1343.423754	2.987899	1340.326901	1348.079546
25	512	4096.0	5	1915.321985	11.007033	1899.969815	1923.883069

The above table gives the test results for 1D matrix multiplication. They show 2 patterns firstly that the time to multiply increases with the size of the matrices. Secondly the results show that the time to multiply is in fact increasing with the number of processors used for any particular data-size.



The above graph shows how the time to multiply is increasing with the number of processors. For example to multiply two 512x512 matrices takes 1024.0seconds on 25 processors, and takes 1195sec for 16 processors, 648sec for 9 processors, and 256 seconds for 4 processors. Unfortunately the timings that were recorded were unreliable, as they don't just include the super-step where the local multiplications are carried out, it also includes the time for broadcasting. There are 3 super-steps in which data is broadcast, in the example mentioned above two 512x512 matrices being multiplied leads to approx 4096KB, 256KB and 4096KB being broadcast to each

processor in each of the 3 super-step respectively. For direct broadcast it takes 552 seconds for 16 processors to broadcast 1024KB of data, and 141.7 seconds to broadcast 256KB. This gives an estimate of 1245seconds to perform this program, which is slightly more than the running time we actually achieve. This just demonstrates how much of this time is made up of the 3 broadcasting super-steps.

Conclusion and suggestions for further work

Note this code was not optimised. The sending of the 2 matrices in steps 1 and 2 could occur in one super-step. Secondly there are two many registration calls. Only the matrix that is remote should be registered. This would improve the performance of the program. The bsp_put and bsp_get primitives are not high performance and so there will some inefficiency caused by the buffering of data.

The final problem is that the timing information is for the duration of the 1d matrix multiplication sub-routine, so it includes the timing for the matrices to be created, all the data to be sent to all the processors, processed locally, and then returned to the source processor. As such the running times are largely made up of the time to broadcast the data, rather than the time it takes to multiply the matrix.

However the results collected do show that the sequential matrix multiplication was better. This is because the wrong timing measurement was taken for matrix multiplication. However the times for the parallel multiplication do not give the total running times of these programs, and it is a fact that adding the communication costs does increase the total running time significantly. It would be interesting to see how using the more efficient 2D and 3D multiplication algorithms will improve this situation.

Conclusion

The project was a success on its minor objectives; I increased my knowledge of BSP and of parallel computing, as I believe is evident in this report. I learnt how to code and run parallel code using the BSP-lib.

I successfully tested code for the parallel broadcast problem, and 1D matrix multiplication. I found that 2-phase was worth than the tree broadcast for larger data-sizes, and that the 2-phase algorithm was not behaving as the BSP model predicts. Further analysis showed what an impact the buffered put had on the experimental results, and that using buffered puts the code will not follow the BSP model even on a dedicated parallel architecture such as Oscar. I also experimentally derived values for the BSP parameters, for the architectures tested on, these were found to be similar but more extensive than those found in the BSP parameter database.

I believe I developed a sound process for testing the algorithms implemented, and processing the data generated. These processes can be found in the appendices, there are still many areas I would have like to investigate further, these areas are discussed in the further work section.

Further Work

Matrix Multiplication

I found the area of matrix multiplication really interesting and it would be good to see some good results from this, and see how the practical compares to the theoretical. It

would be a shame to let the work done go to waste. Also there are may further ideas for parallel matrix to work on. Not only could Strassens algorithm be used to further optimise matrix multiplication, as the local matrix multiplication routines, or as a method for partitioning the matrix. But additionally there is a lot of work from CS321 course concerning other optimisations for matrix multiplication, and complex matrix multiplication, such as using Winigrads method.

DRMA vs. BSMP

All the code I implemented used DRMA (direct remote memory), one of two forms of communication that BSP-lib offers. The other form is BSMP (Bulk Synchronous Message passing). DRMA is optimised for large amounts of data being communicated, whereas BSMP is optimised for small packets of varying sizes. It would be interesting to investigate these primitives and how they compare experimentally, and in which situations which form of communication is optimal.

Sieve of Eratosthenes

I believe it would be interesting to implement Sieve of Eratosthenes on a parallel architecture. How to implement the conceptually distinct processes in SIMD code is interesting.

The lowered numbered sieves get most of the traffic, as most numbers can be discarded by the smaller sieves, thus larger sieve numbers get little traffic. This raises the issue of load balancing. There are two many sieves for each processor to map to just one sieve, so each processor would have several virtual processes. It would be

interesting to see how BSP-lib handles allocating the processors and virtual processors in order to balance the load.

Authors Assessment of the Project

I found the area both exciting and extremely interesting. I believe I succeeded in my aims for the project, as I developed a good working knowledge of BSP and the BSP-lib. Additional several interesting conclusions were made about results collected in this project, which showed me something new about the BSP model.

Unfortunately not as much progress was achieved, as I would have liked, however this is a situation that falls upon anyone conducting a project such as this. There is still a lot of work that I wanted to do, both on the parallel broadcast and matrix multiplication, and additionally other parallel algorithms. I believe that the project raises several interesting ideas for further study.

Acknowledgements

I'd like to thank my family for their help and support for me throughout this year, and my work on this project. Additionally I'd like to thank my project supervisor Dr. Alex Tiskin who was always happy to help, and made him-self available. Finally I'd like to thank two of my friends, David de Niese for his help with finding a project and Matthew Webb for his help with completing the final report.

Bibliography & References

Lafore R. Object Oriented Programming in Turbo C++. California: Waite Group Press; 1991. 741p

Lowe TL, Rounce JF. Calculations for A-Level Physics. Cheltenham: Stanley Thornes; 1992. 424p

Dongarra JJ, Gentzsch W. Editors. Computer Benchmarks. Amsterdam; Elsevier Science Publishers; 1993. 349p

Miller R, Stout QF. Parallel Algorithms for Regular Architectures. London: MIT; 1996. 286p

Leopold C. Parallel and Distributed Computing. USA: John Wiley & Sons; 2001. 229p

Bostock L, Chandler S. Pure Mathematics. Cheltenham: Stanely Thornes; 1994. 741p

Cormen TH, Leiserson CE, Rivest RL. Introduction to Algorithms. USA: MIT; 1999.

Laverick R. Implementation and analysis of Bulk-Synchronous Parallel Algorithms. 3rd Yr Rprt; 2000-2001.

A. Tiskin. The Design and Analysis of Bulk-Synchronous Parallel Algorithms. DPhil thesis. University of Oxford, 1999.

Juurlink B, Rieping I. Performance Relevant Issues for Parallel Computation Models.

Foster I. Case Study: Matrix Multiplication. <http://www-unix.mcs.anl.gov/dbpp/text/node45.html> Accessed 2001.

Gunnels J, Lin C, Morrow G, van de Geijn R. Analysis of a Class of Parallel Matrix Multiplication Algorithms. <http://www.cs.utexas.edu/users/plapack/papers/ipps98/ipps98.html> Accessed 2001.

Tiskin A. Parallel Algorithms. <http://www.dcs.warwick.ac.uk/~tiskin/teach/pa.html> Accessed 2001.

McLatchie B. University of Oxford Parallel Applications Centre. <http://oldwww.comlab.ox.ac.uk/oucl/oxpara/oxpara.htm> Accessed 2001.

BSP Worldwide. <http://www.bsp-worldwide.org/> Accessed 2001.

Ferencz A, Diament B. Parallel All-Pairs Shortest-Paths.

<http://www.cs.berkeley.edu/~ferencz/cs267/final/>

Foster I. Case Study: Shortets-Path Algorithms. <http://www-

unix.mcs.anl.gov/dbpp/text/node35.html>

Hill JMD, McColl WF. Questions and Answers about BSP. Oxford, 1996

Hill JMD, McColl B, Stefanescu DC, Goudreau MW, Lang K, Satish BR, Suel T, Tsantilas T, Bisseling

RH. BSPlib: The BSP Programming library. Elsevier Preprint, 1998.

The code for 2-phase parallel broadcast was based on “bcast_ex4”, part of the BSP Tutorial by Bill McColl and Johnathan Hill. <http:// oldwww.comlab.ox.ac.uk/oucl/oxpara/courses/tutorials/>

The code for the actual tree broadcast is based on and adapted from the code given in the paper "Broadcasting on the BSP model: Theory, Practice and Experience" by Alexandros V. Gerbessiotis.

Glossary

Term	Definition
BSP	**B**ulk **S**ynchronous **P**arallelism – Parallel programming model/ paradigm. Conceptually programs are made up of super-steps, which gives a save, simple coding model and mathematical model.
BSP-Lib	A BSP software library, which allows BSP code to be implemented. See introduction for more information.
Parallel computing	When a computational problem is shared and worked on by multiple computers/processors. I.e. the processors work in parallel.
Sequential computing	When a computational problem is worked on by a computer on its own.
FLOPS	Floating point operations per second. A common benchmark measurement for rating the speed of a micro proessor.
MFLOPS	Mega FLOPS = 1 Million FLOPS
GFLOPS	Giga FLOPS = 1 Billion FLOPS
DCS Lab	Department of Computer science computer laboratory, containing over 70 networked AMD PCs.
SIMD	Single Instruction Multiple Data, this is when you have the same piece of code working on several processors on different data in parallel.

Appendices

Unforeseen Problems during coursework

1) No access to Topaz for Several weeks – 2/10/02
    - During the Christmas holidays, therefore worked on research
    - Solution, waited until returned to university on 7/01/02

2) BSP-lib not compiling (8/1/02 – 11/01/02)
    - Didn't need to use significantly, it resolved itself by the 11th January

3) DCS Topaz non-accessible – 8/02/02

4) BSP-lib performance degradation (after February)
    - On rebuilding BSP-lib to allow for a maximum of 32 processors instead of 16, caused the performance of BSP-lib to fall.
    - Also affected how strict the compilation was, leading to changes required for several programs that were previously working. Strict on use of bsp_init() and an additional bsp_sync() was required at the end of any bsp_begin() – bsp_end() block.
    - Additional peaks appearing on profile graphs were due to initialisation, and were not the programs behaving incorrectly.

5) Home PC infected with Boot virus (11/03/02 – 12/03/02)
    - Computer was rebuilt over 2 days
    - Some data recovered from hard drive, and backups from DCS.

6) No access to lab machines during Easter holiday
    - Reason for access was to get some coursework files and to run a few tests to obtain values for the parameters s, l and g.

- Computers had been switched off, were turned back on, by request, on 12th April. So available for one week of the Easter holidays.

7) Virus and windows corrupt on home PC (27/04/02 - 28/4/02)
    - Unrecognised virus by Norton utilities
    - No data lost, windows rebuilt

Data conversion

1 word = 4 Bytes, on a 32-bit processor. (1word = 32 bits = 4 Bytes)

1 Byte = 8 bits

1 Kilobyte = 1024 Bytes

1 Kilobyte = 256 words

1 Integer = 8 Bytes

Processing the data

The test data from the output logs from the test programs were manually extracted into a MS Excel spreadsheet. On the first worksheet the data was put into the following table:

Broadcast type	**N**	**Size/bytes**	**Size/kilobytes**	**P**	**Time /seconds**
…					

On the next worksheet the data as processed for the statistical content, using array formulas, columns were added for the population (number of results), the average time, the standard deviation, and the minimum and maximum values.

The processed data was then put in a series of summary data tables, from this I generated the appropriate charts and tables that can be seen in the report.

Writing a simple BSP program

```
#include "bsp.h"
#include <stdio.h>
#include <stdlib.h>

void spmd_start();

int npes;

void spmd_start() {

   bsp_begin(npes);
   //PARALLEL CODE BEGIN

   printf("Hello world from processor %d\n", bsp_pid());
   bsp_sync();

   //PARALLEL CODE END
   bsp_end();
}

void main(int argc, char **argv) {
```

```
    bsp_init(spmd, argc, argv);
    npes = 5;
    spmd_start();

    exit(0);
}
```

BSP code is SPMD, that is single processor multiple data, so all parallel code will be run on each of the processors. A single processor runs the above code the bsp_init() function is used to initialise the BSP-library, and allows for dynamic process generation whereby each processor begins executing the code at the call to bsp_begin(). Note all processors will run all the code if bsp_init() is not used. Npes is the number of processors to be run. The spmd_start() function contains the code to be run parallel. At bsp_begin(npes) npes processors are started, each processing the code, which follows until the bsp_end() function is called. In this case the program prints a short message with each processors id number. A bsp_sync() is called, which calls the barrier synchronisation, which allows the processors to synchronise, and then exit when the bsp_end() method is called.

Compiling and running BSP programs

To compile use the bspcc program, this should be found in the bin directory of the B

SP-lib implementation you are using, set the path value appropriately.

- To compile code to run parallel, use the appropriate bspcc;

```
bspcc –o <executable filename> <filename>.c
```

- To compile code to run on a single machine use the shmemsysv switch;

```
bspcc –o -shmemsysv <executable filename> <filename>.c
```

Running the program parallel, first time you will need to do the following things.

1. Create a .rhosts file, with the computer host name, in the root directory. E.g.lab-05 csuki

2. Create a .bsptcphosts file, with each of the names of the machines in the following format;

```
host(lab-04);
host(lab-03);
host(lab-10);
```

   This file is used to give the names of all computers to use within the cluster you are using.

3. Set the environmental variables under each machine. The best way is to change the environment variables in your set up file. Alternatively log on to each computer to be used by the running code and type;

```
export PATH=$PATH:~tiksin/BSPX86/bin
export BSP_DEVICE=MPASS_UDPIP
```

   Note you should put in the appropriate path directory to the bin directory of the BSP-lib implementation you are using.

4. Run the BSP-lib daemon on each machine, this will handle the communication amongst the machines. You can log on to each machine and run bsplibd, alternatively you can use the all switch, which will run bsplibd on all machines listed in the .bsptcphost file.

You can the run the programs parallel by using bsprun.

```
bsprun –noload –npes 4 ./executable_name
```

The noload switch turns off the load manager. The npes switch sets the number of processors to be started.

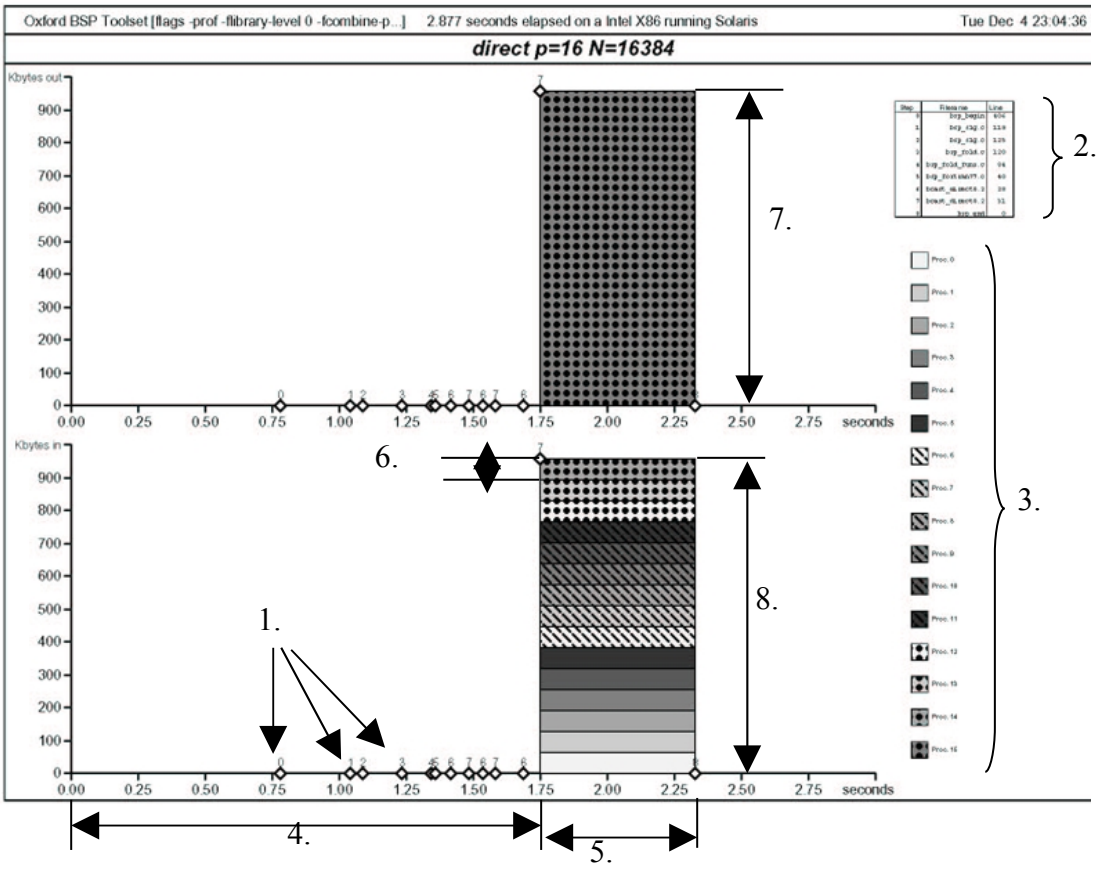
BSP Profile graphs

Generating the BSP Profile graphs

Compile the code with the "–prof" flag. Then use the "bspprof" program as follows:

```
Bspprof –title '<title>' PROF.ps <filename>.ps
```

Reading the BSP Profile graphs



Above is an example of the BSP Profile graph. The top graph shows the amount of data sent versus time by each processor. The bottom graph shows the amount of data received versus time per processor. As you can see from the graph during super-step 7, one processor sends out approximately 1000 Kbytes of data, which is shared equally between each of the other processors.

1. The beginning of each super-step is marked by a numbered diamond, which appears in the key see 2.
2. Table containing each of the super-step references, with the number of each super-step, the filename of the program the super-step is called from, and the line number from that file.
3. Key showing each processor represented by a particular shading pattern.
4. White space indicates time spent processing, and in this case there has been no data transferred between processors.
5. Duration of a super-step, between diamonds labelled 7 and diamond labelled 8. You can see data is transferred during this super-step.
6. The bar is divided into different shaded areas, each representing the data either sent as in this case or retrieved by a particular processor.
7. This bar shows the amount of data sent by each processor. In this case it shows one processor is sending out 1000 Kbytes of data.
8. This bar shows the amount of data received by each processor. In this case it shows each processor receives an n/p (amount of data divided by the number of processors) fraction of the data, approximately 100 Kbytes.

All Pairs Shortest Path

Initially I looked at the all pairs shortest path problem, implementing parallel. There are two main methods I looked at based on those discussed in Dr Alex Tiskin's thesis, the two methods being Floyd Warshall and repeated squares methods. The Floyd-Warshall method is based on Gaussian Elimination, the repeated squares method involves squaring the matrix representation of the graph, taking all paths of two, then squaring again. The repeated squares method offers a marginal asymptotic improvement; the problem is whether this occurs in practice.

I began by looking at the algorithms on a sequential computer, the repeated squares method was implemented, which I've included in the appendix for completeness. Both algorithms relate closely to matrix multiplication, I believed the best method to make these algorithms parallel was to look at the underlying parallel matrix multiplication. I then chose to investigate parallel matrix multiplication. Doing some research on this I found it looked very interesting, and so shifted my focus on to matrix multiplication. The plan was to come back to APSP if I had any time at the end of the project.

Source code Floppy disk

The source code and raw data can be found on the floppy disk attached to the project report. This disk contains 2 zip files, extracting the files will put them into folders to make it easy to browse the appropriate code or data.

Project presentation

The project presentation slides can be found at the back of the project report.