Parallelization of JStar Programs on a Distributed Computer

Simon Crosby

May 30, 2012

Abstract

In the past, the performance of sequential programs grew exponentially as the performance of CPUs increased with Moore's Law. Since 2005 however, performance improvements have come in the form of more parallel CPU cores. Writing parallel programs using existing programming languages can be difficult and error-prone. JStar is a new programming language that allows programs to be written in a naturally parallel way. The JStar project aims to produce compilers that can produce executables for a variety of architectures (such as many-core, GPUs and distributed computers). This thesis proposes a process for compiling these programs into distributed executables, and investigates various trade-offs and techniques for implementing JStar programs on a distributed computer. In this process, first a parallel design is created, then this design is expressed by a separate set of distribute statements that are combined with the original program to create a distributed program. The expressiveness and effectiveness of this approach is investigated for two case study JStar programs: (1) a prime number counting program (2) a version of Conway's Game of Life. Various designs were hand-translated into a distributed Java programs and benchmarks were run to assess the performance of different designs. For each of these case studies, parallel designs were found that achieved high levels of speedup.

Contents

Chapter 1

Introduction

This thesis examines how JStar programs can be compiled for distributed computers. JStar is a high level implicitly-parallel programming language designed to make writing parallel programs easier.

This thesis examines two different JStar case study programs and the implementation of these programs on a distributed computer. Benchmarks are carried out to measure the effect of various implementation decisions on performance.

1.1 State of Distributed Programming

Distributed computing has been for years the dominant architecture of high performance computing. It has long been uneconomic to build a single unit computer that can rival a distributed computer. Instead, many computing nodes are linked together through a local area network or interconnect to work in parallel.

Programs for distributed computers are traditionally written using low level programming languages, such as C or FORTRAN using MPI to communicate between compute nodes. These low level programming languages were used to get the best possible performance from expensive supercomputers.

This combination of message passing with imperative languages must be used with care. It has long been recognised that it is much harder for humans to intuitively understand programs that are composed of multiple interacting stateful tasks than it is to understand a single task[31]. Programs must be carefully written so that the program does not contain subtle concurrency bugs, such as race conditions and deadlock. However, distributed programs were generally written by HPC specialists who were aware of the pitfalls of distributed programming.

As Moore's Law has predicted, the number of transistors on a silicon chip has grown exponentially since the 1960s. Until very recently, this meant that the performance of sequential programs increased

exponentially as well. However around 2005, the trend of exponential growth of the performance of a single processor core came to an end.

As the performance of sequential programs on the latest microprocessors no longer doubles every 18 months or so, many people who are not HPC specialists are looking to parallel and distributed computing to increase the performance of their programs. The price of high performance computing has fallen drastically, as computing has become a utility available through the cloud.

Given the pitfalls of the "imperative with message passing" programming model, it has become a matter of urgency in the software industry to develop new distributed programming environments based on programming models that are more human friendly.

1.2 Goals for the JStar Project

JStar is a temporal logic programming language currently being developed at the University of Waikato. The goal of the JStar programming language is to make parallel programming easier, by allowing the programmer to write programs that are implicitly concurrent and the implementation will be able to, automatically or semi-automatically, convert the JStar program into implementations for various parallel computing platforms (such as many-core or distributed).

A previous University of Waikato project developed the language Starlog[14]. Starlog is the basis for the JStar IL (intermediate language). Starlog is a logic programming language that is augmented with timestamps. Timestamps allow the language to support I/O and mutable state while remaining completely declarative.

In Starlog, all data is stored in relations, with the compiler choosing an appropriate physical data structure for storing those relations[14].

In imperative languages, instructions conceptually are executed in sequential order. Although progress has been made toward automatic parallelization of sequential programs this is still largely an unsolved and very difficult problem. By contrast in Starlog all rules are conceptually evaluated concurrently, and the programmer must provide causal constraints as abstract timestamps to specify if there are any dependencies that must be satisfied before a rule is evaluated. The less restrictive the constraints the more parallelism there is in the Starlog program.

JStar has a Java like syntax (rather than a Prolog like syntax) to make it more attractive to developers. This syntax will be able to be compiled to an intermediate language on which a series of transformations will allow it to generate code for a variety of computational architectures, such as distributed computers, many-core and multiprocessor SMP computers, graphics processing units (GPUs) as well as single processor computers.

The process of making these transformations from a JStar architecture-independent program to a program that could run efficiently on a distributed computer requires knowledge how various decisions could affect the performance of the program. Examples of such decisions include how the computation should be split up onto the different cluster nodes to achieve the good performance and how much data to communicate between nodes at once to reduce communication overhead.

1.3 Goals for this Thesis

The objective of this thesis is to examine how JStar might be compiled into implementations for a distributed computer, and what techniques are useful to obtain efficient implementation of JStar programs. This thesis examines two different case studies, the Life and Primes JStar programs, to investigate different design choices and how they impact performance. For each of these case studies it is examined:

- how programmers might compile these case study programs to concrete distributed programs
- how different designs affect the performance of each program
- how different parameters (such as different communication protocols and message sizes) affect performance of the programs, and which parameter values result in good performance
- how well the different distributed programs are able to scale when using large numbers of compute nodes

The investigation was done by implementing the JStar programs primes and life in Java using the message passing library MPJ Express. Various different implementation options were benchmarked to determine how each option effects the performance of the distributed program. This data would be useful for optimising a future distributed JStar system.

Chapter 2

Related Work

This chapter explores similar efforts to develop declarative languages suitable for use for parallel and distributed programming. As there have been many different parallel and distributed programming projects, only a few will be surveyed in this chapter - those that have some similarity to or would be likely to influence the design of a distributed computer implementation of JStar. The JStar language is presented in Chapter 3.

2.1 Programming Languages

2.1.1 High-Productivity versus Efficiency-Oriented Distributed Programming

Languages and programming models used for distributed programming have traditionally emphasised performance over ease of use. Users who need to get the highest performance out of their limited access to distributed computers generally use low level languages and programming models, which allow a much greater degree of control over how programs are executed and greater latitude to tune programs to achieve high performance.

Users who need this level of performance and control often use either C/C++ or FORTRAN with a message passing library such as MPI. Although this low level of programming provides the programmer with total control, this comes at the cost of manually having to manage many aspects of the program.

To try to raise the level of abstraction of distributed programming and the productivity of programmers, new higher level distributed programming languages have been developed. These languages typically have several features to raise the level of abstraction:

- **Naturally Parallel Programming Model**

  Explicitly data-parallel loops and array operations are provided. These operations can be easily

parallelized.

- **Global View**

  A global view of the distributed system is presented to the programmer, where the activity of each individual compute note is controlled by the runtime. By contrast, MPI is a local view programming model where each activity of each individual compute node is separately controlled by the programmer.

- **Separate Specification of Data Distribution**

  The mechanism for specifying distribution of data across the distributed computer is separated from the program code. This separation of concerns makes the program more portable between different machines and easier to optimise the data distribution for a particular machine.

In late 1994, the High Performance FORTRAN language specification[17] was released. High Performance FORTRAN (HPF) significantly raised the level of abstraction for distributed data-parallel programming. However, when the language was released, the compiler technology was immature, which made the performance of HPF programs inferior to that of programs written using MPI. Most users abandoned HPF and returned to using MPI[25].

Another example of a high productivity distributed programming is ZPL[33]. ZPL is a data-parallel array programming language for distributed computing. ZPL provides separate data distribution constructs. ZPL's Grid construct allows the programmer to describe the distributed computer the program is running on. ZPL's Distribution construct allows the programmer to specify how data is distributed across the grid[2]. Active development of ZPL seems to have stopped around 2004, and the language has not seen widespread industrial use.

Further development of high productivity distributed programming languages has been bolstered by DARPA's HPCS (High Productivity Computing Systems) program. Supercomputer vendors have developed the high productivity languages Fortress[3] (Sun/Oracle), X10[13] (IBM) and Chapel[12] (Cray). However, now funding for the HPCS program has ended, it is not certain that these languages will be developed further.

2.1.2 Bloom and Dedalus

The JStar programming language semantics are based on Starlog, which is a bottom-up logic programming language based on Datalog[11]. Datalog programming languages are used in deductive databases. Starlog adds timestamps to bottom-up logic programming. By having timestamps, Starlog is able to emulate mutable state while remaining completely declarative.

Several other Datalog inspired programming languages have been created which make the bottom-up programming model the basis for a distributed programing language. One example is Bloom[23], a prototype of a distributed programming language. Bloom is a high level language for distributed systems.

Bloom is based on Dedalus, a temporally stratified Datalog variant. Dedalus differs from Starlog in that Dedalus is lower level. In Dedalus, location specifiers are used to specify on which node a tuple is stored. By comparison, in JStar partitioning of a relation across machines is not explicitly specified in the program and would be done later, either automatically or with some direction from the programmer.

Timestamps in Dedalus are also much more limited than in Starlog. In Starlog, an arbitrary expression can be used to give the stratification order. Starlog also allows multi-dimensional timestamps. In Starlog, looser stratification orders allow more parallelism. In Dedalus however, the timestamps only exist to control communication and synchronisation in the distributed system.

2.1.3 Parallel RDBMSs: Massive Distributed Data Parallelism

Parallel relational database management systems (RDBMS) have been highly successful in providing a form of implicit parallelism[15]. Programmers are able to execute SQL queries without having to explicitly say how their queries will be parallelised. Relational algebra on large data sets can generally be parallelised easily over a large number of shared-nothing compute nodes. How the query is executed in parallel is decided by the query optimiser. An example of a commercial shared-nothing parallel DBMS is Terradata.

JStar stores data in relations, as do relational databases. Relational databases were in fact a major inspiration for Starlog. Starlog did for programming what the relational database did for storing and querying data, which was to present a high level relational view of data while storing data in efficient data structures at the physical level[14].

The architecture of many parallel RDBMSs is a set of shared-nothing compute nodes[15]. This is also the architecture that will be used by a distributed version of JStar. As both parallel DBMS and JStar will both process large amounts of data stored in relations using shared-nothing compute nodes, some of the techniques used for parallel RDBMSs are likely to be useful for implementing a distributed JStar compiler.

2.1.4 Dataflow Languages

JStar aims to allow as much implicit parallelism as possible. The goal is that this implicit parallelism can be exploited automatically, possibly aided by the use of programmer provided hints or by autotuning[4]. This goal is similar to that of dataflow programming languages, such as SISAL[16] and Id[29]. JStar is

similar in some respects to dataflow languages, as the program rules are triggered by the insertion of tuples into the program database.

One of the problems of dataflow languages is that, although they could extract large amounts of fine-grained parallelism, the grain size of the parallelism is too small to justify creating a thread or process to exploit it. Exploiting this fine-grained parallelism would lead to an overall slowdown. This could also be a problem for parallelizing JStar programs where the stratification order is quite tight. In this situation all the pieces of parallelism may be too fine grained to be worth executing in parallel.

2.1.5 Parallel Logic Programming Languages

Work has been done on the parallelization of logic programming languages[22], some of which is applicable to creating a distributed implementation of JStar. Some general logic programming parallelization techniques have been found to be applicable to Starlog programs[14].

Although Clayton[14] only studied compilation of Starlog for uniprocessors, as part of his future work he identified several areas where parallelization could take place during compilation:

1. solving of goals in the body of a rule
2. evaluation of rules activated by the same trigger tuple
3. selection use of multiple minimal tuples in DELTA as trigger tuples

Clayton[14] noted that (1) is an example of And-parallelism, while (2) and (3) were examples of Or-parallelism.

2.2 Compiling High Level Algorithms to Concrete Implementations

In JStar, although the aim is to write programs with as much implicit parallelism as possible, the JStar programs are not explicit about how to parallelize the program across a distributed computer. This is in all likelihood going to be the most difficult part of compiling JStar programs for a distributed computer. The experiment programs presented in this thesis were transformed into distributed implementations by hand. Much more research is required to determine how concrete distributed implementations can automatically (or semi-automatically) be generated. It seems likely that the JStar implementation will need to have knowledge of parallel software architectures, and the compiler (or programmer) will need to select a suitable architecture for the given program or algorithm. For many programs, there will be several different possible software architectures which may be used. For example, the primes program

presented in Chapter 6, can be implemented either by using divide and conquer or a filter pipeline. The architecture chosen may have a significant impact upon performance. This can be seen in Chapter 6, where the map-reduce version of the program greatly outperformed the pipeline-filter version.

2.2.1 Parallel Evaluation Strategies

Trinder et al.[34] discuss a similar problem for the implicit parallelism support in Haskell. Their approach is to separate the algorithm from how it is parallelised, which they call evaluation strategies. They give the formula Algorithm + Strategy = Parallelism.

Trinder et al.[34] present several different parallel evaluation strategies:

- Pipelines: a series of functions which process a stream of data.
- Producer/Consumer: one process consumes a data structure generated by another process. The data structure acts as a buffer between the two processes.
- Divide and Conquer: A problem is divided up into smaller problems which are executed in parallel. The solutions to these smaller problems are combined to yield the final result.
- Data Oriented Parallelism: Items in the data structure are processed in parallel.

An distributed implementation of JStar will also separate the JStar program from the details of how the program is parallelised. In the future, a distributed JStar compiler could have a similar parallelization strategy mechanism. The compiler could apply parallelization strategies to JStar programs to determine how the program would be run in parallel across the compute nodes.

2.2.2 Parallel Patterns

Pattern languages are a way that expert knowledge of existing solutions can be codified and used by non-experts. Originally from the architecture community, this concept has been applied to software architecture[19]. "Our Pattern Language"[26] (OPL) provides a pattern language for the architecture of parallel programs. A distributed implementation of JStar would either need to incorporate a lot of this knowledge of existing solutions, and choose an appropriate pattern automatically or have it supplied by the programmer.

OPL defines a layered hierarchy of patterns:

- Structural Patterns and Computation Patterns

Structural Patterns define the overall organisation of the distributed program. Structural Patterns include map-reduce and pipe and filter. Computation Patterns describe different classes of computations. Computation Patterns are essentially the Motifs (formerly known as Dwarfs) described in the paper *"The landscape of parallel computing research: A view from Berkeley"*[4].

- Algorithm Strategy Patterns

  High level strategies for exploiting parallelism. For example, data parallelism and task parallelism.

- Implementation Strategy Patterns

  Patterns relating to how the program is organised, such as Single-Program Multiple Data (SPMD), fork/join and Bulk Synchronous Parallel (BSP), and data structures used in parallel programming, such as distributed arrays and distributed hash tables.

- Parallel Execution Patterns

  These patterns are low level mechanisms to support parallel execution, such as SIMD, MIMD and thread pools, as well as concurrent synchronisation mechanisms, such as message passing, mutual exclusion and transactional memory .

This pattern language has a large and growing set of patterns. These patterns have been in use for a long time, and OPL just documents their use. It would, in fact, probably be impossible to write a distributed program that did not use one of the patterns. Solutions described by these parallel patterns have been used in the experiment programs in Chapters 6 and 7. Two different versions of the primes program were written, one with the pipe-filter pattern and one with the map-reduce pattern. The life program is an example of the Structured Grid Pattern[26] and uses the Ghost Cell Pattern[27] (otherwise known as the Halo pattern), which reduces communication costs by sending the data from multiple generations in one message.

2.3 Linda and Coordination Languages

Linda is a coordination language for distributed and parallel programming. Linda allows processes to communicate through a globally shared associative memory called a tuplespace. Processes can communicate by adding and removing data from the tuplespace. Processes can block while waiting for a tuple, and when that tuple is entered into the tuplespace the process becomes unblocked and can read the data.

Linda separates the computation from coordination. The individual processes are written in a traditional programming language, communication and coordination between processes is done through the coordination language.

Some aspects of JStar are similar to Linda.

- JStar has a globally shared memory of tuples stored in relations, which is similar to Linda's tuplespace.
- In JStar, the generation of new tuples can trigger the evaluation of rules. In Linda, a process can become unblocked when a matching tuple is added to the tuplespace.
- JStar shares the same goal with with Linda, which is to separate the computation and coordination aspects of the program as far as is possible. One JStar program should ideally be able to run on uniprocessor, multicore and distributed computers. Further information required to implement the program on a specific parallel hardware architecture should be provided separately from the program.

Linda allows processes to remove tuples from the tuplespace, which introduces non-determinism and therefore the possibility of race conditions. JStar, by contrast, is deterministic.

Although both Linda and JStar provide a global database of tuples, there are considerable differences between the two. Linda provides a tuplespace, which processes written in existing programming languages can coordinate their activity. JStar provides a new programming language for writing parallel software. The emphasis of JStar is to analyse the structure of the program and then the compiler (possibly with some help from the programmer) can determine how the program will be parallelized and how each process coordinates with the others.

2.4 Autotuning for Performance

The aim of the JStar project is to create a very high level language which can be compiled to run on a variety of parallel hardware architectures, such as distributed computers, many-core or a multi-level combination of the two. When using lower level programming languages, the programmer has control over many low level aspects of the program that have an impact on performance. However, when using a more abstract language such as JStar, many of the decisions a programmer might have made when using a lower level programming language are now made by the JStar compiler or runtime. Many of these decisions about things like communication, allocation of work to nodes, granularity of task sizes, will have an impact on performance. To achieve good levels of performance good parameter values need to be chosen in these areas.

One approach to improve code generation is to rely on the compiler to analyse the program and use cost models and heuristics to choose good parameter values. However, compiler analysis is already quite complex for sequential code, and compiling code for distributed computers introduces many new

parameters which must be optimised[4]. Also, different distributed computers have different performance properties, such as interconnect latency and bandwidth and different numbers of cores per node. The program may have to be tuned for an individual distributed computer, which cannot be done by a static compiler model.

Autotuning has been identified as a process which can help optimise parameters to increase the performance of generated code for parallel hardware architectures[4]. Autotuning is an approach to optimisation where different variants of the program are benchmarked on the target architecture using typical data, and the performance of the different variations is compared. Using this information the compiler is able to choose between different options to achieve high performance [36]. Autotuning has been used with some success with scientific code, with the performance often being comparable and sometimes better than hand-tuned code[4].

Autotuning has been used in the compilation of Starlog programs for uni-processor machines to choose which data structure should be used to store data in relations[14]. Three different techniques were tried:

- Static cost analysis
- Autotuning based on regression analysis of benchmark runs
- Adaptive optimisation using data collected by profiling at runtime

Of these three techniques it was found that autotuning was the most accurate, consistent and robust technique. Clayton[14] notes "it seems there is no substitute to running a program to determine its bottlenecks". Adaptive optimisation at runtime was found to have a high overhead, which reduced performance.

It is hoped that this technique which, works well on a single processor machine, will be able to be used on a distributed computer as well. This will obviously introduce another dimension, as data structures can be divided across different compute nodes.

Chapter 3

The JStar Programming Language

The JStar programming language is a temporal logic programming language under development at the University of Waikato. As this thesis contains many different Starlog programs, this chapter provides an introduction to JStar and the intermediate language Starlog.

The aim of the JStar programming language is to allow parallel programs to be written more easily. Using JStar, programmers can write programs with implicit parallelism, which the compiler can exploit. JStar programs have a *temporal stratification order* that controls the order in which the rules are executed. The less restrictive this ordering, the more implicit parallelism will exist in the program.

3.1 The JStar Implementation Strategy

The JStar language is a high level programming language with a Java-like syntax. To compile programs in this language to executable code, first the program is translated into JStar's intermediate language, Starlog. After the program is translated into the Starlog intermediate language various transformations can be applied to the program. The aim of these transformations is to change the program from a architecture-independent program to a program specific to a particular hardware architecture (for example distributed, many-core, GPUs, or a combination of all three). This process is illustrated for the Pascal program in Chapter 4.

3.2 Starlog, the JStar Intermediate Language

In this thesis, we are interested in getting from the Starlog level to the distributed implementation level. That means that the code examples in this thesis are in Starlog rather than JStar. To help the reader understand these code examples, a quick tutorial on Starlog is provided.

In Starlog, data is represented as tuples contained inside a set of relations. This set of relations is called the *database*. The physical data structure which holds the data in these relations is chosen by the implementation (perhaps with some guidance from the programmer). This is the same way that data is represented in a relational database management system (RDBMS). Unlike a RDBMS however, this data is stored in memory rather than on disk. Storing data in relations provides a uniform representation of data. It also raises the level of abstraction for data storage by freeing the programmer from having to think about the data structures the data is stored in.

Starlog programs are *temporally stratified*. Each Starlog program has a *stratification order*. A stratification order is a partial ordering of all the tuples produced by the program. The position of a tuple within the stratification order is determined by the *timestamp* of the tuple. The timestamp of the tuple is determined by the data in the tuple. Starlog provides facilities for declaring what the timestamp of a tuple would be based on the data contained in the tuple.

In Starlog, all tuples that a rule accesses must have an equal or earlier timestamp than the tuples that a rule produces (if negation is used the timestamp must be strictly earlier). This is checked statically when a Starlog program is compiled.

Tuples produced in Starlog programs are immutable, they cannot be updated or deleted. Also, if the program passes the point in the stratification order when a particular tuple would be produced, and that tuple has not been produced, then that tuple can never be produced.

By ensuring that Starlog programs are temporally stratified, this allows the Starlog language to have two useful features:

- The stratification order controls the order in which the rules are evaluated. When it can be deduced that these rules can be executed independently then they can be executed concurrently. This makes the programming model implicitly parallel.

- Temporal stratification allows explicit control over the order of events in the program, which allows Starlog to mimic features from imperative languages such as I/O and updates while remaining declarative. Although tuples in Starlog are immutable, mutable variables can be simulated by adding an extra argument to the relation to act as a timestamp. This allows the values in the relation to be tracked over time.

3.3 Starlog Examples

To illustrate the Starlog language, in this section two different Starlog programs are presented: a program that calculates the sets of odd and even numbers (Section 3.3.1) and transitive closure of a graph (Section

```
% Even and Odd Numbers
type even(int).
type odd(int).

stratify even(N) by [N,even].
stratify odd(N) by [N,odd].
stratify even << odd.

even(Nb) <--
 range(0,20,Nb),
 not(even(Nb-1)).

odd(Nb) <--
 range(0,20,Nb),
 not(even(Nb)).
```

Figure 3.1: A Starlog program that calculates the sets of odd and even numbers between 0 (inclusive) and 20 (exclusive).

3.3.2).

3.3.1 Odd and Even Numbers

To illustrate the Starlog language, a simple example program that calculates the sets of even and odd numbers between 0 and 20 is given in Figure 3.1. Starlog programs contain three parts: the type declarations, the stratification declarations and the rules.

The type declarations declare the relations used in the program and the data types of the attributes of those relations. Type declarations begin with the `type` keyword. For the even number example in Figure 3.1, two relations were declared, `odd` and `even`. The declarations of these relations specifies that they both have one argument of type `int`.

Stratification declarations begin with the keyword `stratify`. Stratification declarations specify the stratification order of the Starlog program. To specify the stratification order for a relation, a list of terms is provided. Given a tuple, the list of terms is used to determine the tuple's place in the stratification order. All tuples produced in a Starlog program are ordered lexicographically using the lists given in the stratification declarations.

For example, in Figure 3.1 there is a stratification declaration for the relations `even` and `odd`. The stratification declarations for these relations are:

```
stratify even(N) by [N,even].
stratify odd(N) by [N,odd].
```

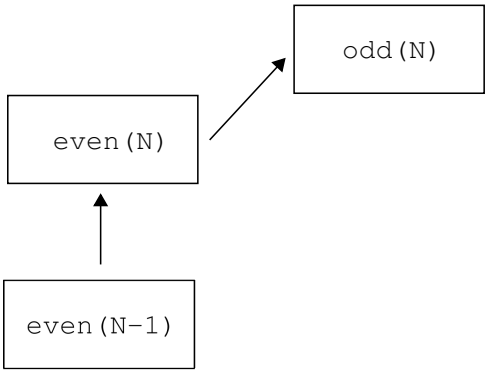


Figure 3.2: A graph of the causality links between tuples in the even and odd numbers program.

```
stratify even << odd.
```

This declaration says that the tuples in the relation `even` are stratified by `[N,even]`. This means that the tuples in the `even` relation are stratified first by N, which is bound to the first (and only) attribute of the tuple, and then by the constant `even`.

Similarly, the tuples in the `odd` relation are stratified by `[N,odd]`. Odd tuples are first stratified by N and then by the constant `odd`.

The declaration `stratify even << odd` specifies that the constant `even` is lower than the constant `odd`. Given the stratification orders given for `even` and `odd`, this means that the tuple `even(N)` is stratified earlier than `odd(N)`.

The next part of the program are the rules themselves. The algorithm used to calculate the `even` relation tuples is: if a number is not even, then the next number is even. For the `odd` relation: if a number is not even, then it is odd. The `range(N,M,I)` relation is a built-in relation which produces `I` values from N (inclusive) to M (exclusive). It is typically used in the same way that a for loop would be used in imperative languages.

In this program, to calculate `even(Nb)` depends on knowing if `even(Nb-1)` is true or false. This is why the stratification order is important in this example. The stratification order ensures that the tuples required by the body of a rule have been produced, or that they will never be produced, before the rule executes.

For example, when the tuple `even(0)` is produced at the beginning of the program, the `even` rule checks that `not(even(-1))` is true. It is known that it is true, because the tuple has not been produced. Since the program is past the point in the stratification order when it could be produced, it will never be produced.

For `odd`, the stratification order ensures that it is known if `even(Nb)` is true or false before calculating `odd(Nb)`.

```
% Transitive Closure
%--------------------------------------------------------------------
% Finds all paths that exist between any two nodes in a directed graph.
% path(X,Y,T) is the path between nodes X and Y that was found during
% iteration T of the program.

type path(int,int,int).
stratify path(_,_,T) by [T].

path(1,2,0) <-- true.
path(2,3,0) <-- true.
path(3,4,0) <-- true.
path(4,5,0) <-- true.
path(5,2,0) <-- true.

path(X,Y,T+1) <--
 path(X,Z,T),
 path(Z,Y,T2),
 T >= T2,
 not (path(X,Y,T3), T >= T3).
```

Figure 3.3: A Starlog program which calculates the transitive closure of a directed graph.

3.3.2 Transitive Closure

A Starlog program which calculates the transitive closure of a directed graph is shown in Figure 3.3.

The transitive closure program determines, for each pair (A,B) of vertices in the directed graph, if there is a path through the graph from A to B. Paths are stored in the relation `path(A,B,T)`. The `T` argument is an explicit timestamp field, which is the iteration of the rule on which the path between vertices A and B was found. This program is an example of how graph and tree programs can be implemented in Starlog using a recursive query. The transitive closure of the graph is calculated by doing a recursive self-join on the path relation.

Chapter 4

Implementing JStar on a Distributed Computer

4.1 Design of Distributed Programs

Foster[18] gives a methodological approach to designing a parallel algorithm. In this approach the algorithm design process is split into four stages:

- Partitioning: dividing the problem into many small subproblems
- Communication: determining how the subproblems should best communicate to ensure that the algorithm is parallel and to reduce bottlenecks
- Agglomeration: assembling the fine-grained partitioning of the subproblems into a larger grained parts that can be efficiently executed on a distributed computer
- Mapping: assigning the large grained pieces to nodes in the distributed computer so that the execution time is minimised

4.1.1 Partitioning

According to the above methodology, partitioning the problem means breaking it up into large numbers of parallel pieces. In the case of the Pascal's triangle, partitioning the algorithm into the calculation of each value would be a fine grained decomposition of the problem. There are two different ways to partition a program:

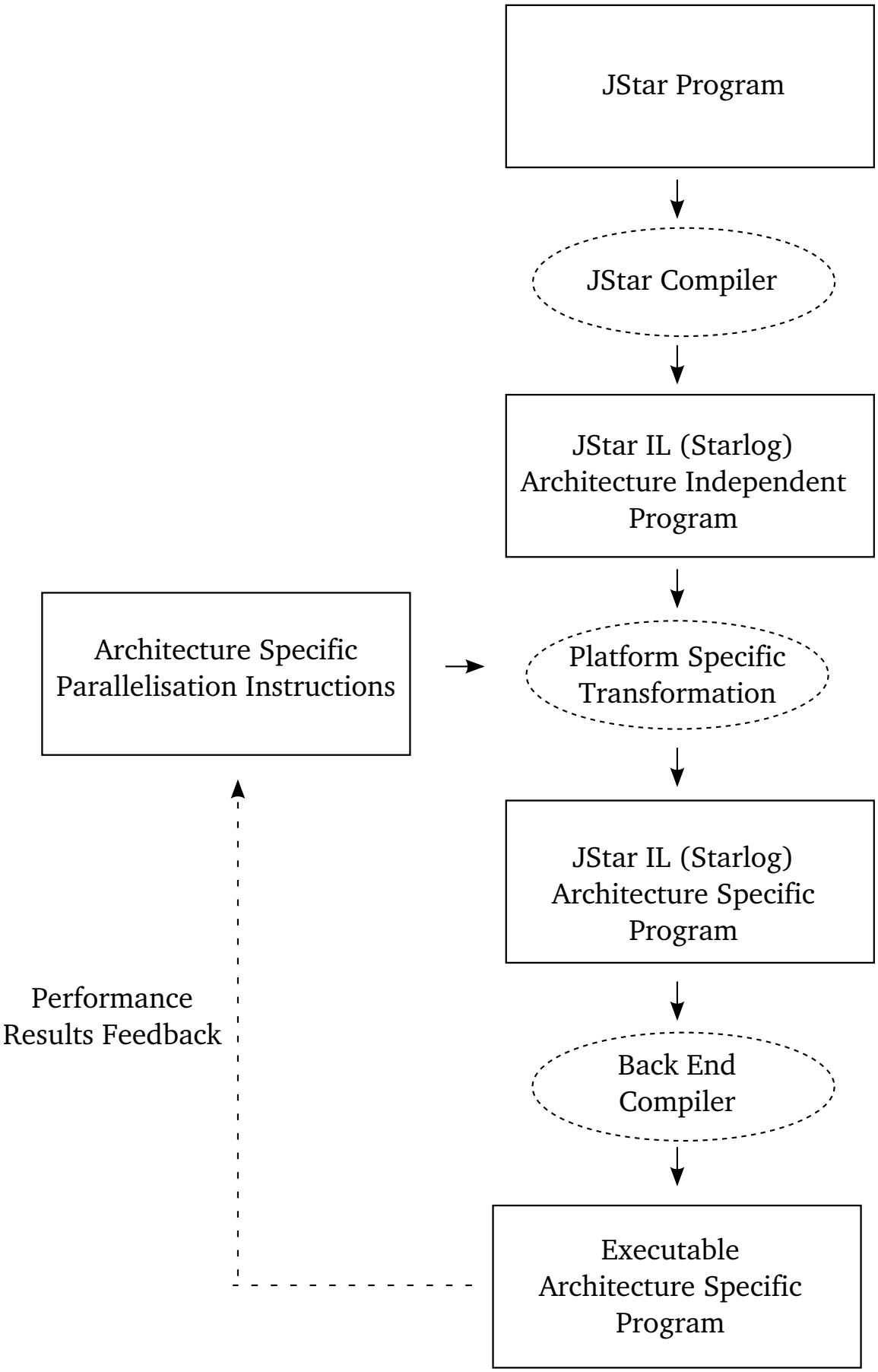


Figure 4.1: The architecture of the JStar system. First, JStar programs are compiled into an intermediate language (Starlog). Transformations are done on the Starlog program to change it from being parallel architecture independent to being specific to a particular parallel architecture. This architecture specific program is then compiled to a executable for a that particular parallel architecture.

- Domain Decomposition: the data space of the program is partitioned into pieces where each piece can be processed in parallel to achieve data parallelism. A domain decomposition of a JStar program involves duplicating the same code in each partition, but having each partition consume or produce different data tuples. A example of domain decomposition is ray tracing, where the computation is split into pixels that can be rendered separately.
- Functional Decomposition: the program is partitioned by the program code itself, where each partition runs a different part of the program code to achieve task parallelism. A functional decomposition of a JStar program would partition the program so that each partition ran different rules. An example of functional decomposition would be audio stream processing where a number of filters are applied to an audio signal, where each filter is run in parallel.

It is usual to apply both partitioning strategies to a program to achieve both data parallelism and task parallelism where it can be extracted.

4.1.2 Communication

The tasks identified in the partitioning stage are not normally independent: tasks need to communicate with other tasks. In the communication stage, how each task in the program communicates with other tasks is identified. Foster identifies four roughly orthogonal axes that categorise communication:

- **Local/Global**

  Local communication is when a node only communicates with a small number of "neighbour" tasks. Global communication requires large numbers of tasks to communicate.

- **Structured/Unstructured**

  Structured communication is when the tasks are organised in a regular pattern, such as a grid or tree structure. Unstructured communication is when there is no regular pattern of communication between tasks.

- **Static/Dynamic**

  Communication is static if the set of other tasks that a task communicates with does not change. Communication is dynamic if the other tasks a task will communicate with are variable and are not known until runtime.

- **Synchronous/Asynchronous**

  In synchronous communication, both the producer and consumer tasks know when to communicate with each other, and both tasks synchronise in a coordinated fashion. In asynchronous communication, the tasks do not know exactly when the other task will require data.

Some analysis of the Starlog program is required to determine what sorts of communication are used. Starlog by default uses global, unstructured, dynamic and asynchronous communication. A better performing implementation may be possible if it can be determined that communication is local, structured, static or synchronous.

4.1.3 Agglomeration

The aim of the partitioning process is to extract a large amount of parallelism. However, the overhead and communication costs of a large number of fined-grained tasks would result in poor performance. To achieve good performance fine-grained tasks must be merged into a smaller number of coarse grained tasks.

While the partitioning and communication stages of the process are largely conceptual, the agglomeration stage is the part of the process where the most is done to take the algorithm from a high level of abstraction to be an architecture-dependent program that is suitable for a distributed computer.

In this stage there can be several possible choices about how the fine grained tasks can be combined together into coarse grained tasks. Examples of this are seen in the case studies presented in Chapters 6 and 7. For the Life case study, the board could either be sliced up along one dimension, or divided into two dimensional blocks. Sometimes, different software architectures are possible. For the primes case study, two possible architectures were examined: a divide and conquer implementation where different subranges were checked by different nodes, and a pipeline filter where each node removed multiples of a different set of primes.

Sometimes, communication costs can be reduced by duplicating fine-grain tasks in multiple coarse-grain tasks. An example of this are halos[27] (otherwise known as ghost cell regions) discussed in Section 2.2.2. By duplicating computation by haloing, the number of messages sent is reduced.

4.1.4 Mapping

The mapping stage assigns the tasks to physical compute nodes. By allocating multiple tasks to one node many-core architectures can be exploited.

Foster identifies two goals for mapping tasks to nodes:

- Placing tasks that execute concurrently on different nodes.
- Reduce communication time by placing tasks that communicate with each other on the same node.

Foster notes that there can be a conflict between these two goals. Sometimes placing tasks which communicate with each other on the same node results in poor concurrency. In this case a trade-off has

to be made between increasing concurrency and reducing communication costs. An example of this is shown later in this chapter in Section 4.3.4.

4.1.5 Language Features for Specifying Agglomeration

As described in Section 2.1.1, many high productivity distributed programming languages allow the user to specify how the workload is divided up among the tasks. For example, ZPL allows the user to specify how an array is divided up among the tasks for processing.

The mechanisms in JStar allowing the programmer to specify where the computation occurs are more complex than those found in other languages. In JStar data is stored in relations, whereas the data in other distributed programming languages (such as ZPL) is stored in arrays. Array can easily be divided up by partitioning the array among the tasks so that each task can process one part of the array. In a distributed JStar program, the programmer must provide instructions to the compiler that determine how the relation is partitioned across the tasks based on the data in each tuple, and which tuples should be communicated to other tasks.

In a distributed implementation of JStar, the programmer needs to be able to control how the workload is divided among the high level tasks. In this thesis, a new language construct is proposed that allows the programmer to do this. This proposed language construct is the distribute command:

`distribute` *relation(attributes...)* `to` *set of tasks* `using` *expression*

The distribute command does not directly control where tuple values are calculated. Instead, it determines to which tasks the output tuple values are sent to. Output tuples in *relation* which match the *expression* are sent to the *set of tasks*.

For a rule to be triggered on a task all the necessary input tuples must already be on the task. By controlling which tasks output tuples are sent to using the distribute command, the programmer is able to control which rules are triggered on other tasks. The programmer can use this to divide a program so that it is computed in parallel across the tasks.

An example of using the distribute command for the Pascal's Triangle program is shown in Section 4.3.4.

One problem with this mechanism for distributing the program across different tasks is that the semantics of the program (and the output tuples produced) can be changed. This is usually not a desirable property, so there is room for further research to develop a mechanism which cannot change the meaning of the program, or to have proof obligations to prove that the distribute statement does not change the meaning of the program.

It is in principle possible to prove that the transformed program produces exactly the same tuples as

the original program by performing a global inductive proof over the stratification order of the program. However it would be preferable to have more modular proof obligations, or distributed design patterns, which are sufficient to ensure that the semantics are preserved.

Another way to gain confidence that the distribute statements work as intended is testing. As the distributed program is supposed to produce the same results as the original Starlog program, the original Starlog program can be used as a test oracle. This allows the distributed program to be tested automatically against the original Starlog program.

4.2 Creating a Distributed Executable from the Starlog Program

In this section, a process for parallelizing a JStar program is presented. This process involves first taking a program written in Starlog (which is architecture-independent) and transforming it to add architecture-specific details about how the program executes on a distributed computer. Examples of architecture specific details include how the computation is divided up between tasks and how the tasks communicate with each other. From this program with architecture specific details, a low-level imperative program with message passing is derived.

The process of creating a distributed executable involves:

1. Adding information to the original Starlog program to specify how the program is parallelized across tasks and how each task communicates with the others. This information is contained in distribute statements such as those in Figures 4.6 and 4.7. This step creates a distributed Starlog source code program as shown in Figure 4.8.

2. Using a Starlog compiler (extended with support for message passing communication) to compile the distributed Starlog source code to an executable.

4.2.1 Combining the Program Rules and the Distribute Statements

The next step of this process is to combine the rules of the program with the distribute statements. Combining the program rules with the distribute statement produces a distributed Starlog program. Combining the program rules and the distribute commands involves combining each of the rules with all applicable distribute statements. The distribute statements which are applicable to a given rule are the ones that apply to the relation in the rule's head.

When the program rules are combined with the distribute statements, an special extra task number attribute is added to the relation which determines which tasks the output tuples are sent to. When a

task generates a tuple from a rule, the tuple will be sent to the task given by the task number attribute.

A program rule, such as

relation`(`*attributes...*`)` `<--` *rule_clauses*`.`

with a distribute statement, such as

`distribute` *relation(attributes...)* `to` *set of tasks* `using` *distribute_clauses*`.`

is combined into:

relation`(`*set of tasks, attributes...*`)` `<--` *rule_clauses*`,`*distribute_clauses*`.`

Variable names which are bound to one of the attributes of the relation are renamed so they are the same in both *rule_clauses* and *distribute_clauses*. Other variable names in *rule_clauses* and *distribute_clauses* are uniquely renamed so that the variable names do not conflict.

Combining the program rules with the distribute statements in this way creates a single-program multiple data (SPMD) program which is run by each of the coarse-grain tasks. A SPMD program is one where each task runs the same program, but the program can identify which task it is running on perform the computation allocated to that task.

4.2.2 Compiling the SPMD program using a Starlog Compiler

Once a single-program multiple data program is created, this can be compiled using a Starlog compiler that includes support for communication using task number attributes like those in the previous section.

Some work has already been done on Starlog compilation and optimisation [14]. However, as of yet there is no Starlog compiler which can send output tuples to other tasks. The design of the existing Starlog compiler would need to be modified to send output tuples and receive them from other tasks.

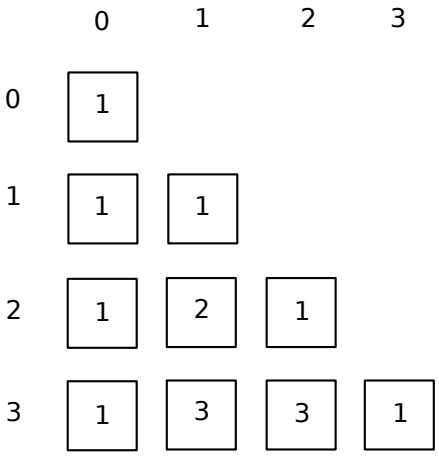
For the general case of compiling distributed Starlog programs there would also need to be some sort distributed coordination system, so each task knows when the other tasks had finished producing tuples at a certain level in the stratification order. The most general case might be quite expensive in terms of communication, because each task would need to coordinate with all the other tasks. However, for many specific examples it is possible to analyse a program to reduce the amount of communication to coordinate the tasks. For example, it may be possible to know from what other tasks a task will receive output tuples, or at what point in the stratification order. An example of this is shown in Section 4.4.1.

4.3 An Example: Pascal's Triangle

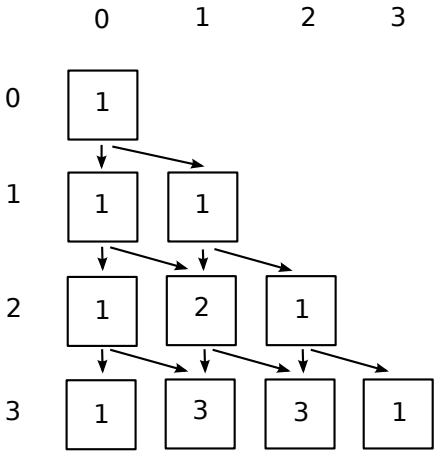
The Pascal's Triangle program generates rows of integer values, where the values at each end of the row are 1, and the other values are the sum of the two values above it in the preceding row. The Starlog program for Pascal's Triangle is shown in Figure 4.2.

```
% Pascal's Triangle
%-------------------------------------------------------------------------
% Generates X rows of pascals triangle where X is the number specified
% in rows(X). A pascal(R,C,N) tuple represents each element in the
% triangle where R is the row, C is the column number and N is the
% binomial coefficient.

stratify pascal(R,_,_) by [R,pascal].

type rows(int).
type pascal(int,int,int).

% Number of rows to generate.
rows(20) <-- true.

pascal(0,0,1) <-- true.

% Rule 1
% Generate a 1 as the first element in any row.
pascal(I,0,1) <--
 pascal(J,0,1),
 rows(Nn),
 I is J+1,
 Nn > I.

% Rule 2
% Generates a 1 as the last element in any row.
pascal(I,I,1) <--
 pascal(J,J,1),
 rows(Nn),
 I is J+1,
 Nn > I.

% Rule 3
% Generates row elements as the sum of the two
% elements in the previous row.
pascal(R+1,C,N1+N2) <--
 pascal(R,C,N1),
 rows(Nn),
 R+1 < Nn,
 pascal(R,C-1,N2).
```

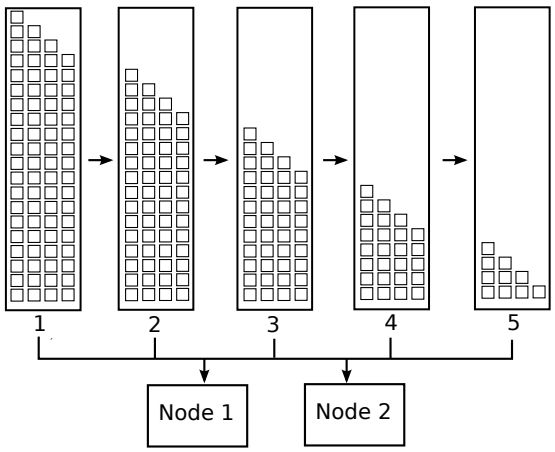
Figure 4.2: The Pascal's Triangle Starlog program



(a) Partitioning



(b) Communication



(c) Agglomeration and Mapping

Figure 4.3: Foster's methodology applied to the Pascal's Triangle program.

4.3.1 Partitioning

The Pascal's Triangle program can be partitioned into smaller threads using domain decomposition. The value at each row and column location can be calculated by an individual task. This exposes the maximum level of available parallelism.

4.3.2 Communication

After partitioning the program into many fine-grained tasks, then it is determined how each task communicates with the others.

The communication in the Pascal example is

- **Local**, as each task only communicates with neighbouring tasks.
- **Structured**, as Pascal's Triangle is a regular grid.
- **Static**, as communication channels between tasks are known in advance.
- **Synchronous**, as it is known at what point each task will send or receive data.

Rule 1 and 2 in the Pascal program do not require any input from other tasks, but the value they produce is required by the task that calculates the value in the next row.

Rule 3 in the Pascal program requires the two values above it in the triangle and thus needs to receive inputs from two tasks. This task in turn sends the result to two other tasks on the next row.

4.3.3 Agglomeration

To achieve good performance, the fine grained tasks that calculate Pascal's Triangle values need to be agglomerated together to create coarse-grain tasks. For this example, we will partition the triangle into vertical slices. Each slice can continue processing in parallel, sending and receiving messages to neighbouring tasks slices, as shown in Figure 4.3(c).

The performance could probably be increased using halos. Halos reduce the number of messages required, at the cost of duplicating some computation across tasks. By using halos, messages do not have to be sent and received for each row. Messages can be sent containing enough information to allow tasks to independently continue for multiple rows, as shown in Figure 4.4 . This has the potential to significantly reduce the amount of time spent communicating and increase the performance of the program.

A distributed JStar compiler will allow the programmer to specify how the fine-grain tasks are agglomerated into coarse-grain tasks. Some language constructs which allow the programmer to do this are discussed in Section 4.3.4.

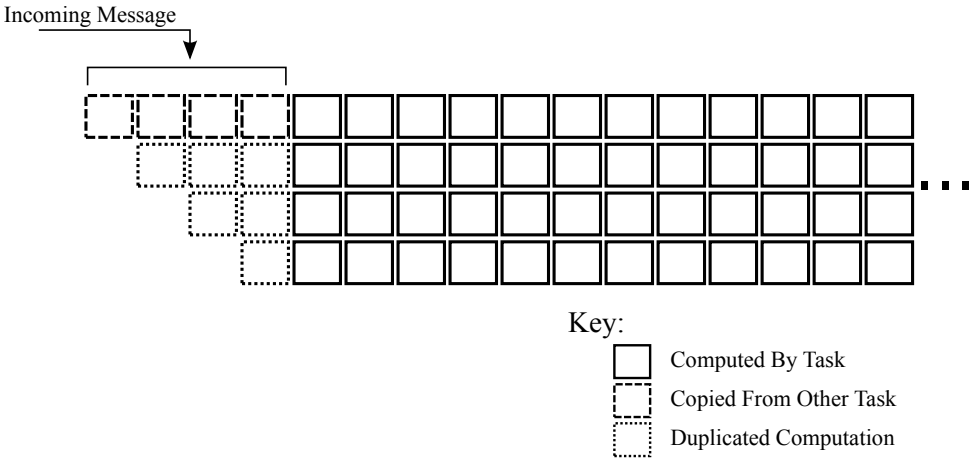
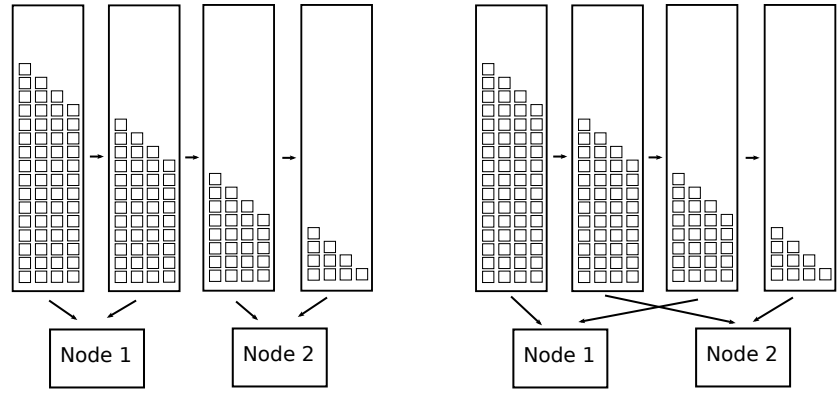


Figure 4.4: An example of using a halo for computing Pascal's triangle.



(a) Communicating tasks on the same node.

(b) Tasks allocated to increase concurrency.

Figure 4.5: Two different possible mappings.

4.3.4 Mapping

When deciding how to map the coarse-grain tasks Pascal's Triangle tasks to nodes, there is a conflict between the goal of placing tasks that communicate with each other and the goal of having the tasks executing concurrently.

Figure 4.5 shows two different possible mappings of coarse-grain tasks to nodes. The mapping shown in Figure 4.5(a) allocates tasks that communicate with one another to the same node. This reduces communications costs, but as all the tasks which compute the right half of the triangle are inactive until near the end of the program, concurrency is poor. The mapping shown in Figure 4.5(b) achieves a higher level of concurrency, but at the cost of more inter-node communications.

A distributed JStar implementation needs to provide some mechanism for specifying which coarse-grain task is assigned to which compute node. The distributed JStar runtime could accept a configuration file which allows the user to specify how tasks are assigned to nodes. The JStar runtime could accept

```
% Width of slice allocated to each task
slice_width(1000) <-- true.

% Assign each column to a task
distribute pascal(_,C,_) to T using
 slice_width(Sw), T is C // Sw.

% Send last column of slice to next task
distribute pascal(C,C,_) to T+1 using
 slice_width(Sw), C rem Sw = Sw - 1, curr_task(T).
```

Figure 4.6: Distribute commands for a distributed Pascal's Triangle program without haloing.

map commands such as:

`map` *task* `to` *node* `using` *expression*`.`

For example, to obtain the mapping shown in Figure X.X, the map command used would be:

```
map Task to Node using Node using num_tasks(NumTasks), Node = Task // NumTasks.
```

To obtain the mapping shown in Figure X.X, the map command used would be:

```
map Task to Node using num_nodes(NumNodes), Node = Task mod NumNodes.
```

4.3.5 Distribute Statements

In Section 4.1.5 the distribute command was introduced, which allows the programmer to specify how to partition the workload across the coarse-grain tasks. For the Pascal's Triangle program, the triangle has been split into vertical slices. Distribute statements which allocate a 1,000 column wide slice of the triangle to each task and communicate without using haloing are shown in Figure 4.6.

The first distribute statement in this example splits the triangle into slices. This distribute command specifies that all output tuples belong on the task given by the expression `C // Sw`, where Sw is the width of the slice allocated to each task (`//` is the integer division operator).

The second distribute statement sends the output tuples for the last column of each slice to the next task. This distribute command uses a special predicate `curr_task(T)`. `curr_task(T)` binds `T` to the âĂIJcurrentâĂİ task number, which is the task that the tuple was computed on. `T+1` sends the output tuple to the next task. This specifies the communication between tasks.

Together, these two distribute rules control the workload and communication for each task by directing where output tuples are sent. These two rules ensure that the output tuples of the allocated section of the triangle are on each task, triggering the calculation of the next row of Pascal's Triangle values.

```
% Width of slice allocated to each task
slice_width(1000) <-- true.

% Width of the halo region.
halo_size(10) <-- true.

% A) Divide the triangle into slices. This distribute allocates the
% first number in each column to a particular task. Together with
% distribute statements B and C, this ensures that the calculation of
% the rest of the column will also be triggered on the same task.

distribute pascal(C,C,_) to T using
 slice_width(Sw), T is C // Sw.

 % B) Copy across a part of the row every halo_size(N) rows. Together
 % with distribute statement C, this generates a halo effect.

distribute pascal(R,C,_) to T+1 using
 slice_width(Sw), halo_size(N), curr_task(T), R rem N = 0,
 range((T+1)*Sw - N, (T+1)*Sw, C), tasks(T+1).

% C) All tuples generated on a task are added to the tuple database
% on that task (except for Pascal rule 2, this is handled by A).
distribute pascal(R,C,_) to T using
 R > 0, R \= C, curr_task(T).
```

Figure 4.7: Distribute commands for a distributed Pascal's Triangle program with haloing.

4.3.6 Using Distribute Statements to Implement Haloing

A more complex example of using the distribute command is the Pascal's Triangle program with haloing. Haloing reduces the amount of communication required by duplicating some computation across nodes. In the case of Pascal's Triangle, each task (except for the task that calculates the leftmost slice) duplicates computation done by the task that calculates the slice to the left, as shown in Figure 4.4. This duplication can be achieved using distribute statements.

A set of distribute statements that partitions the triangle across the tasks using 1,000-column wide slices and uses a halo to reduce the number of messages sent to one every ten rows is shown in Figure 4.7.

The distribute statements in Figure 4.7 distribute each slice to a different task. The first of the distribute statements (marked A in Figure 4.7) puts the output tuples of rule 2 of the program (see Figure 4.2) on to the correct task.

The distribute statement B handles the communication. It sends a horizontal block of `N` pascal values to the next task every `N` rows (the constant `N` is from `halo_size(N)`). By sending this block of output tuples, this will cause the next task to duplicate some of computation for the next `N` rows, creating a

halo region.

The distribute statement C states that any pascal tuple a task calculates will be sent to the same task, except for when $Row = Column$ (which is handled by A). The effect of this rule is that it controls which other calculations are triggered. This rule allows the Pascal's Triangle values within the slice to be calculated as well as the duplicated calculation in the halo region. But the $Row \neq Column$ constraint prevents a task from calculating values in Pascal's Triangle that are in columns to the right of the slice allocated to that task.

One difference between the distribute statements for the haloing and non-haloing programs is that in the haloing version `curr_task(T)` is used to allocate slices to the tasks. For the version without haloing, it is sufficient to divide Pascal's Triangle into slices using the expression:

```
distribute pascal(_,C,_) to T using
 slice_width(Sw), T is C // Sw.
```

This command specifies that the task the output tuples are sent to is determined by the column number. However, this method of dividing the triangle between the tasks cannot be used with haloing. The calculation of the pascal tuples is being duplicated on the neighbouring task, and this distribute statement would force the output tuples of the duplicated computation to be sent back to the original task. Sending the output tuples back to the original task would serve no purpose, and the extra communication cost would defeat the point of haloing.

For the version using haloing, a different method is used to divide Pascal's Triangle among the tasks. A distribute statement instead specifies which task the first output tuple for each column is sent to. Another distribute statement places the pascal output tuples on the same node they are calculated on. By using the `curr_task(T)` predicate, it is known which task the tuple was calculated on. The distribute commands for the version with haloing use the `curr_task(T)` predicate to keep the output tuples on the same task where they were calculated. These haloed pascal tuples implicitly trigger the calculation of rest of the column on the same task.

4.4 Creating a Distributed Starlog Program

By applying Foster's methodology to the Pascal's Triangle example it has been shown informally how the program can be converted into a program suitable for a distributed computer.

```
% 1. pascal(0,0,1) combined with distribute statement A:
pascal(T,0,0,1) <-- true, 0 = 0, slice_width(Sw), T is 0 // Sw.

% 2. pascal(0,0,1) combined with distribute statement B:
pascal(T+1,0,0,1) <--  true,  slice_width(Sw), halo_size(N), curr_task(T),
 0 rem N = 0, tasks(T), range((T+1)*Sw - N, (T+1)*Sw, 0), tasks(T+1).

% 3. pascal(0,0,1) combined with distribute statement C (always false):
pascal(T,0,0,1) <-- true, tasks(T), 0 > 0, 0 \= 0, curr_task(T).

% 4. Rule 1 combined with distribute statement A:
pascal(T,I,0,1) <--  pascal(T,J,0,1), rows(Nn), I is J+1, Nn > I, I = 0,
 slice_width(Sw), T is 0 // Sw.

% 5. Rule 1 combined with distribute statement B:
pascal(T+1,I,0,1) <-- pascal(T,J,0,1), rows(Nn), I is J+1, Nn > I,
 slice_width(Sw), halo_size(N), curr_task(T), I rem N = 0,
 range((T+1)*Sw - N, (T+1)*Sw, 0), tasks(T+1).

% 6. Rule 1 combined with distribute statement C:
pascal(T,I,0,1) <-- pascal(T,J,0,1), rows(Nn), I is J+1, Nn > I, I > 0, I \= 0,
 curr_task(T).

% 7. Rule 2 combined with distribute statement A:
pascal(T,I,I,1) <-- I = I, pascal(T,J,J,1), rows(Nn), I is J+1, Nn > I,
 slice_width(Sw), T is I // Sw.

% 8. Rule 2 combined with distribute statement B:
pascal(T+1,I,I,1) <-- pascal(T,J,J,1), rows(Nn), I is J+1, Nn > I,
 slice_width(Sw), halo_size(N), curr_task(T), I rem N = 0,
 range((T+1)*Sw - N, (T+1)*Sw, I), tasks(T+1).

% 9. Rule 2 combined with distribute statement C (always false):
pascal(T,I,I,1) <-- pascal(T,J,J,1), rows(Nn), I is J+1, Nn > I, I > 0, I \= I,
 curr_task(T).

% 10. Rule 3 combined with distribute statement A:
pascal(T,R+1,C,N1+N2) <-- pascal(T,R,C,N1), rows(Nn), R+1 < Nn,
 pascal(T,R,C-1,N2), R+1 = C, slice_width(Sw), T is C // Sw.

% 11. Rule 3 combined with distribute statement B:
pascal(T+1,R+1,C,N1+N2) <-- pascal(T,R,C,N1), rows(Nn), R+1 < Nn,
 pascal(T,R,C-1,N2), slice_width(Sw), halo_size(N),  curr_task(T),
(R+1) rem N = 0, range((T+1)*Sw - N, (T+1)*Sw, C), tasks(T+1).

% 12. Rule 3 combined with distribute statement C:
pascal(T,R+1,C,N1+N2) <-- pascal(T,R,C,N1), rows(Nn), R+1 < Nn,
 pascal(T,R,C-1,N2), R+1 > 0, R+1 \= C, curr_task(T).
```

Figure 4.8: Combining the program rules with the distribute statements.

4.4.1 Combining Program Rules and Distribute Statements

To create a distributed single-program multiple data program, the program rules are combined with the distribute statements, as described in Section 4.2.1.

To specify how the Pascal's Triangle program is partitioned, an extra attribute is added to the `pascal` relation. This attribute is the task number of the task where the tuple is stored. If the tuple is stored on a different task, then the tuple is sent to that task. When this extra attribute is added, the `pascal(Row,Column,Value)` relation becomes `pascal(Task,Row,Column,Value)`.

The result of combining the pascal rules (from Figure 4.2) and the halo distribute rules (from Figure 4.7) is shown in Figure 4.8. The new rules created by combining the pascal rules with the distribute statements are labelled 1 to 12.

Currently, there is no distributed version of JStar. To test the distribute statements, a distributed environment was simulated inside the Starlog interpreter. This distributed environment was simulated by creating separate sets of pascal rules for each task. Each of these sets of rules simulates one task.

In each of these sets of rules, the output of the `curr_task(T)` predicate is hard-coded to the task number for that simulated task. The Task attribute for any of the input tuples from the `pascal` relation are also hard-coded to the task number of simulated task. This simulates only being able to access tuples that are stored on the same task.

Simulating a distributed environment allowed testing the distribute statements so that errors could be found and corrected. The distributed program produced the same results as the original program for several different test cases. Although testing is not as good as a proof, it is possible to have some confidence that the distributed program works as intended.

4.4.2 Compiling the Distributed Program

To compile the pascal SPMD program in Figure 4.8 a Starlog compiler which supports distributed programs (described in Section 4.2.2) is required.

There are various optimisations that can be done at this point. Some of these rules in can be eliminated or simplified. Rules 3 and 9 can be eliminated because they both have a clause which is always false.

Rule 1 can be simplified to `pascal(0,0,0,1) <-- true` . Rule 4 can be eliminated because the head of rule 4 simplifies to `pascal(0,0,0,1)`, which has already been proved true by rule 1.

Many of the rules in Figure 4.8 contain a lot of duplication, as they were created by combining each of the program rules with each of the distribute statements. The performance of this code could likely be improved by sharing duplicated code between rules.[14]. The existing Starlog compiler already does

an optimisation called *common prefix factorisation*[14]. Common prefix factorisation factors out shared sequences of instructions. This similar to the imperative programming language optimisation where repeated if statements with common elements in the conditions can be rewritten so that each element is only evaluated once.

Part of a distributed Starlog program is coordination between each of the tasks. This allows each of the tasks to know that they have received all the tuples they will ever receive at a particular point in the stratification order. For the pascal example, some static analysis of the program can help optimise the coordination between the tasks:

- Each task with a task number of `T` only sends to task `T+1`. This means that each task will only receive messages from `T-1` (except for the first task which does not receive any messages). Each task only has to coordinate with the previous task and the next task, which requires less communication than coordinating with all the tasks.

- As haloing is used, data is not sent every row. Every rule in Figure 4.8 only sends on rows where the remainder of the row number divided by the halo size is zero (i.e. the row number is divisible by the halo size). This is because of the clauses `halo_size(N), R rem N = 0` in each of the rules, which send the output tuples to the next node. Each task can assume it will not receive any tuples for rows where the row number is not divisible by the halo size.

As the program is stratified by row number, this will be translated into imperative machine code in which the outer loop iterates over the row number. The program would stop every number of rows to perform communication, along the lines of this pseudocode:

```
nextCommunicationRow = 0;
row=0
while row < rows
   // communicate this generation?
   if (row == nextCommunicationRow) {
       start sending data (nonblocking) to next
       receive data from previous
       finish sending data
       nextCommunicationRow += rowsPerMessage

   calculate next row
```

4.5 Conclusions

In this chapter, a process by which a JStar program can be transformed into a distributed imperative program is illustrated. The JStar program is first compiled to the JStar intermediate language, Starlog. From the Starlog program, a distributed version of the program is created. This is done using a parallel

program design methodology[18]. Using this design, a set of distribute statements is written that determines how the program is run across a distributed computer. By combining the program rules with the distribute statements a distributed single-program multiple data (SPMD) Starlog program is created. This SPMD program defines how the program will run on a distributed computer. This program is then compiled by a Starlog compiler which supports distributed programs to create an executable.

This chapter gives only a rough sketch of the process. To create a JStar compiler for distributed computers, some steps of this process (such as coordination between tasks and optimisation) need to be explored in more detail. In the future, more research into parallelization and compilation techniques will be required to develop a practical JStar compiler for distributed systems. Another topic which could be explored in the future is the possibility of automatically generating the distribute statements for JStar programs. This would further automate the process of creating distributed JStar programs.

Chapter 5

Communication

Efficient communication is critical to the performance of a distributed program. If processes running on different nodes do not communicate efficiently it is impossible to achieve good performance as time that could have been used doing computation will be wasted on communication overhead. This chapter investigates the impact of various factors on communication latency and throughput. The factors investigated are the communication operations used, the size of messages and the impact of flow control.

5.1 MPJ Express Message Passing Library

MPJ Express[5], a Java MPI-like message passing library, is the communication library used for the experiments in this thesis. MPJ Express provides a pure Java message passing library for distributed programs on Ethernet networks. Ethernet message passing in MPJ Express is implemented on top of the Java NIO (New I/O) classes.

Other available Java MPI-like message passing libraries are mpiJava[10] and MPJ/Ibis[9]. MPJ/Ibis is implemented on top of the Ibis[28] grid computing environment. MpiJava uses a C MPI library through JNI.

MPJ Express was chosen for these experiments as it is a popular and well supported library. The performance of MPJ Express is comparable to mpiJava and MPJ/Ibis[5].

5.1.1 MPJ Express Message Send Modes

MPJ Express, being modelled on MPI, provides the send and receive modes described in the MPI standard.

Send and receive operations can be either *blocking* or *nonblocking*. Blocking send and receive operations cause the program to block until the buffer is safe to write or read to. Nonblocking send operations

will return immediately. The programmer must either explicitly use a test operation to check if a buffer is safe to read or write, or use a wait operation which blocks until it is. Nonblocking send and receive operations are able to do other computation while waiting for the operation to complete.

There are four types of send operation: *standard*, *synchronous*, *ready* and *buffered*.

In the MPI standard, **standard mode** is loosely specified in order to allow more flexibility for implementers. Implementations of standard mode send are allowed to buffer the message on either the sender or receiver end, allowing the program to continue without waiting for the receiving task to perform a matching receive operation. Alternatively, implementations of standard mode send are also allowed to wait for the receiving task to begin a receive operation before sending the data.

In MPJ Express, the standard mode send operation buffers small messages, but waits for the receiving process to begin a receive operation for large message sizes. By default, large messages are messages 128 Kbytes or larger. This is because the MPJ Express standard mode send operation uses a different communication protocol for large messages than small messages. These communication protocols are discussed in Section 5.1.2.

This difference must be taken into account when using standard mode send. When sending small messages, which are buffered on the receiving node, it is assumed that the receiving node will have sufficient memory available to store incoming messages. In some situations more messages can be sent than there is available memory to buffer them, which causes the receiving process to crash. In these situations flow control is required to prevent the receiving process from being sent more messages than it is able to process.

Synchronous mode send, on the other hand, always waits for the receiving process to start a receive operation before sending data to the receiving process.

Buffered mode send allows the programmer to supply a buffer on the sender where messages can be buffered until they are sent.

Ready mode send is a send mode that has a precondition that the receiving task has already posted a receive operation.

5.1.2 MPJ Express Communication Protocols

There are two different protocols used in MPJ Express to implement these communication operations, the eager protocol and the rendezvous protocol. The eager protocol does not exchange control messages before transmitting the data, the rendezvous protocol does. For the rendezvous protocol, the sending process sends a ready-to-transmit control message and the receiving process sends a ready-to-receive control message. After the two processes exchange control messages and both are in the "ready to

communicate" state, the data is transmitted.

The eager protocol has lower latency than the rendezvous protocol as the initial exchange of control messages is not required. The eager protocol assumes that there is enough memory to store the incoming message.

The standard mode send operation uses the eager protocol for small messages, and the rendezvous protocol for larger messages. For smaller messages, using the eager protocol reduces the latency. For large messages, overhead of the rendezvous protocol is relatively insignificant. The message size at which the protocol change occurs is by default 128 Kbytes.

5.2 Performance of MPJ Express on Symphony

Several experiments were conducted to measure the performance of MPJ Express on the Symphony cluster at the University of Waikato.

The average latency was measured with a ping-pong experiment, where MPJ Express messages are sent back and forth between two nodes. The ping pong cycle, was repeated 10,000 times. The total time was divided by 20,000 which gives the average time to send one message. This experiment was repeated using different message sizes and both standard and synchronous send modes. The results of this experiment are shown in Figure 5.1. Here, it can be seen that standard mode has a lower latency than synchronous mode.

Also tested was the throughput of different send modes and message sizes. A task on one node sends 20,000 messages to a task on a another node. The throughput is calculated by measuring the amount of time taken to send the data from one node to another. The experiment was done with both standard and synchronous send modes using different message sizes. The results are shown in Figure 5.2.

For standard mode, the throughput when using message sizes less than 768 bytes is poor. For message sizes larger than 768, further increases in message size result in only small increases in throughput. For messages sizes larger than 128 KB, the standard mode switches from the eager to the rendezvous protocol, effectively making standard mode equivalent to synchronous mode.

Synchronous mode was found to have inferior throughput compared to standard mode for message sizes less than 128 KB. Unlike standard mode, where increases in throughput plateaued out for message sizes larger than 768 bytes, synchronous mode requires large message sizes to achieve good performance. As can be seen in Figure 5.2, when using 256 KB message sizes a throughput of 71.7 MB/s could be achieved, but only 13.9 MB/s when using 8 KB message sizes.

The choice of how much data to send in a single message can have a huge impact on performance. Sending each piece of data in a single message would be simpler, but would require sending large numbers

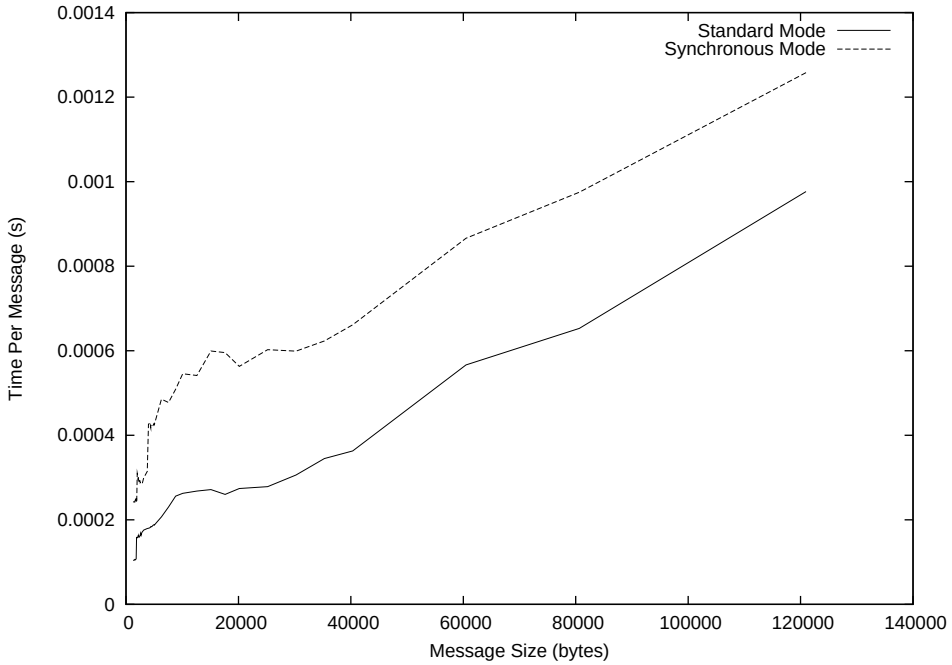


Figure 5.1: Average time taken per message in ping pong experiment.

of messages. However there is overhead to sending a message, so transmitting data in large numbers of small messages creates an unacceptable overhead. Batching data into larger messages is a necessity for achieving good performance.

5.3 Flow Control

Often in a distributed program, one task can produce data more quickly than the next can consume it. This is a problem, for example, in pipeline style programs such as the pipeline prime filter discussed in Chapter 6. If the data is sent using standard mode, then the data could overflow the buffer of the receiver task.

Experiments were conducted to measure the performance of two different flow control schemes.

5.3.1 Rendezvous Communication Based Flow Control

One solution to this problem is to replace standard mode send operations with synchronous send operations. Synchronous send mode uses the rendezvous protocol. This forces the sending task to wait until the receiving task posts a matching receive operation before transmitting the message. However, this would make the tasks unnecessarily sequential, as the sending task would have to wait until the receiving task posted a receive operation.

This limitation can be overcome by the receiving task posting a nonblocking receive operation before

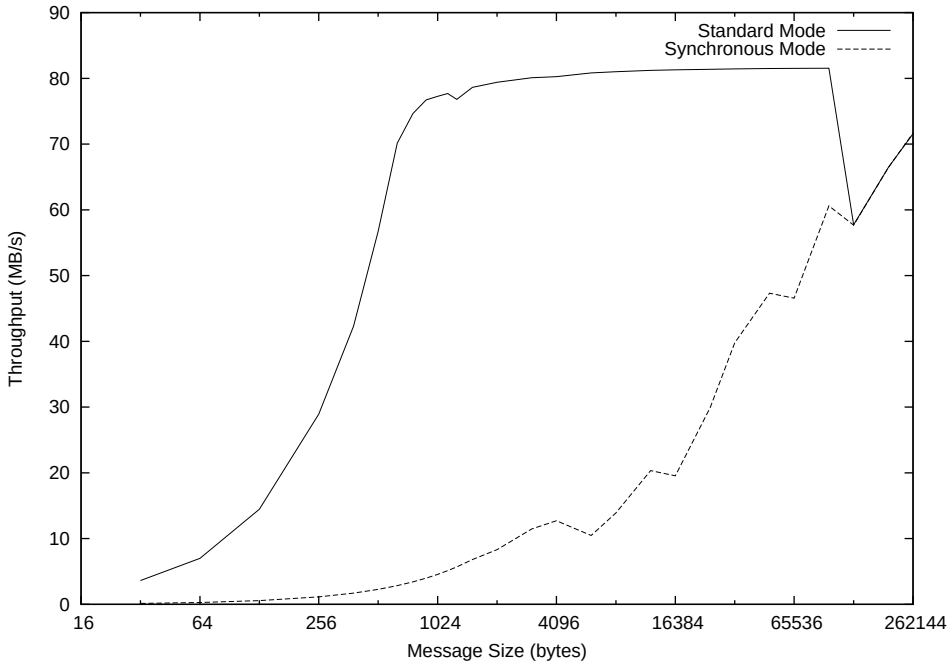


Figure 5.2: Throughput achieved with different message sizes.

doing other computation. This pre-posted receive operation allows the sending task to send the message while the receiver node is doing other computation.

A pool of these pre-posted receive operations can be used, which allows the sending task to get ahead of the receiving process. The pool of buffers is used on a round-robin basis. When all of the pool of pre-posted receive operations have completed the sending task must wait for the receiving task to re-post the receive operations before being able to transmit more messages.

Experiments were conducted to see if the performance could be increased by increasing the number of send and receive buffers used on the sender and the receiver. The throughput was between two nodes was measured using synchronous send mode. Each run was done three times and the median of the three runs was taken.

As can be seen in Figure 5.3, using multiple buffers does substantially increase the throughput for message sizes less than 256 KB.

5.3.2 Eager Communication Based Flow Control

One problem with the synchronous flow control system described in the previous section is that rendezvous protocol communication is slower than eager protocol communication. However eager protocol communication has no mechanism to prevent the receiver's buffer from overflowing. To prevent the sending process sending too much data to the receiver, there must be a separate back channel from the

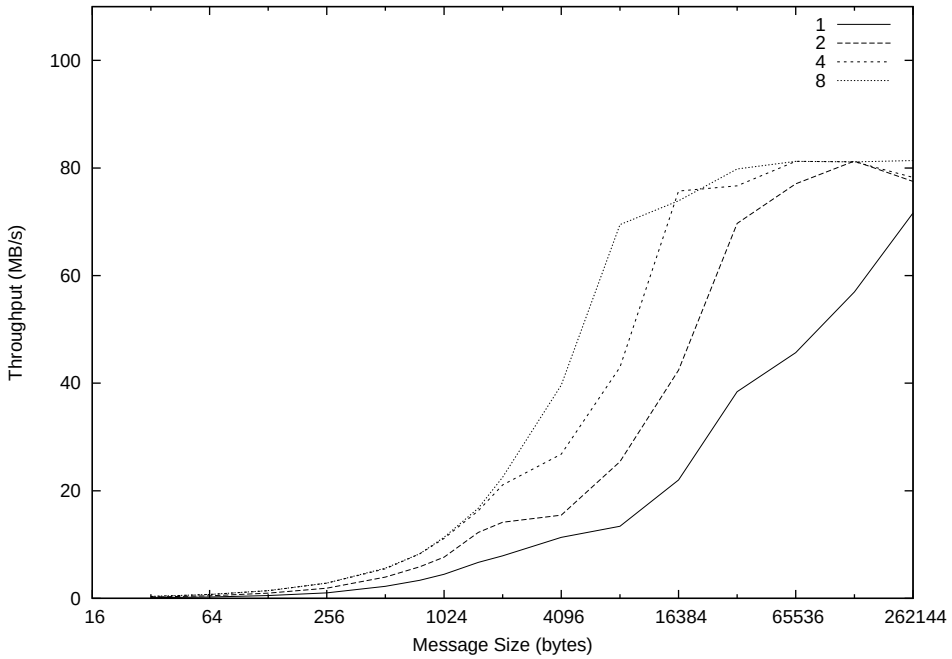


Figure 5.3: Throughput achieved using synchronous communication flow control with different message sizes.

receiver to the sender so that the receiver can notify the sender that data has been successfully received and more data can be sent.

Experiments were conducted to test the efficiency of eager protocol based flow control against that of rendezvous flow control. There are several parameters which may affect performance when using eager mode flow control:

- **Buffer Size**

  The size of the messages being sent.

- **Number of Buffers**

  The hypothesis is that the number of buffers used on the sender and the receiver will have no impact on the communication speed, because send buffers can be reused without waiting for the receiver process to finish receiving data.

- **Message to Acknowledgement Ratio**

  The message to acknowledgement ratio is the ratio of messages sent to acknowledgement messages received from the receiver process. The hypothesis is that this will have an significant effect on the performance when sending small messages, and using a high message to acknowledgement ratio will provide the best performance. However, for large message sizes the message to acknowledgement ratio will become less important.

- **Window Size**

  The window size is the amount of data that can be sent without receiving an acknowledgement. When this limit is reached the sending process must block until an acknowledgement for previously sent data is received.

To test the effect of these parameters on the performance a series of experiments was done. The first experiment measures various combinations of message size(64,128,256,512,768,1024,1536,2048,4096 8192,16384,32768,65536), number of buffers (1,2,4,8) and message to acknowledgement ratios (1,2,4,8,16,32,64). The window size was fixed at 512KB.

The second experiment measured the effect on the window size. As it had been found that the best number of buffers was 8, the number of buffers was fixed at 8 and runs were tried with different window sizes (1 KB, 2 KB, 8 KB, 32 KB, 128 KB and 512 KB).

To measure the throughput, 100,000 messages were sent and the total amount of data sent was divided by the time taken to send the messages.

5.3.3 Effect of Number of Buffers Used on Performance

Using more than one buffer did give a small increase in throughput for small (less than 1 KB) messages when using a message to acknowledgement ratio above 4. Using more than one buffer also increases throughput for larger (greater than 8 KB) message sizes. For message sizes between 1 KB and 8 KB the number of buffers used did not make much difference to the throughput achieved. The effect of the number of buffers used on throughput is shown in Figures 5.4, 5.5, 5.6 and5.7.

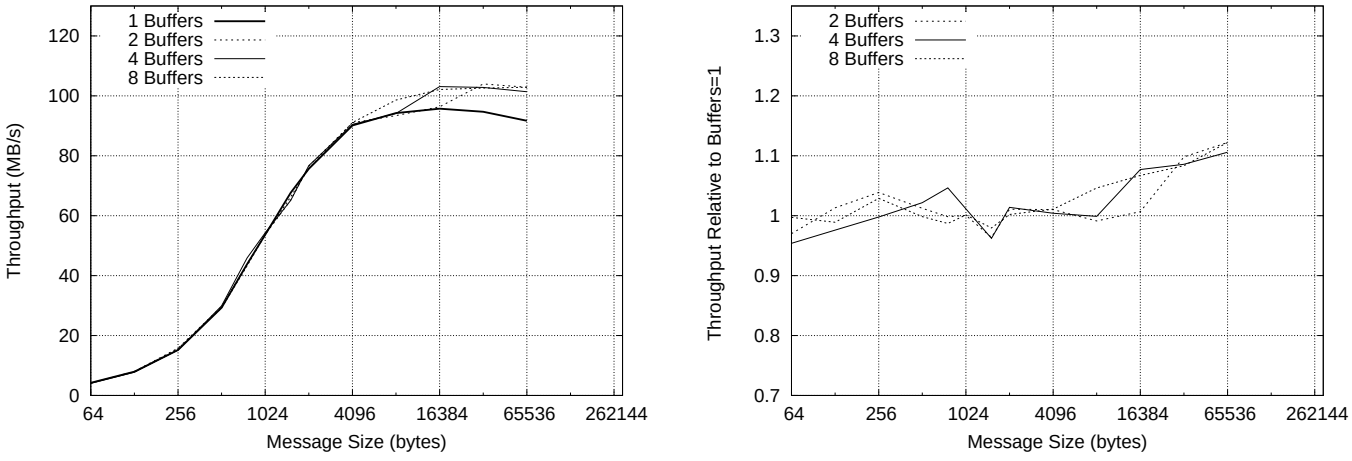


Figure 5.4: Effect on throughput of using different numbers of buffers, using a message to acknowledgement ratio of 1.

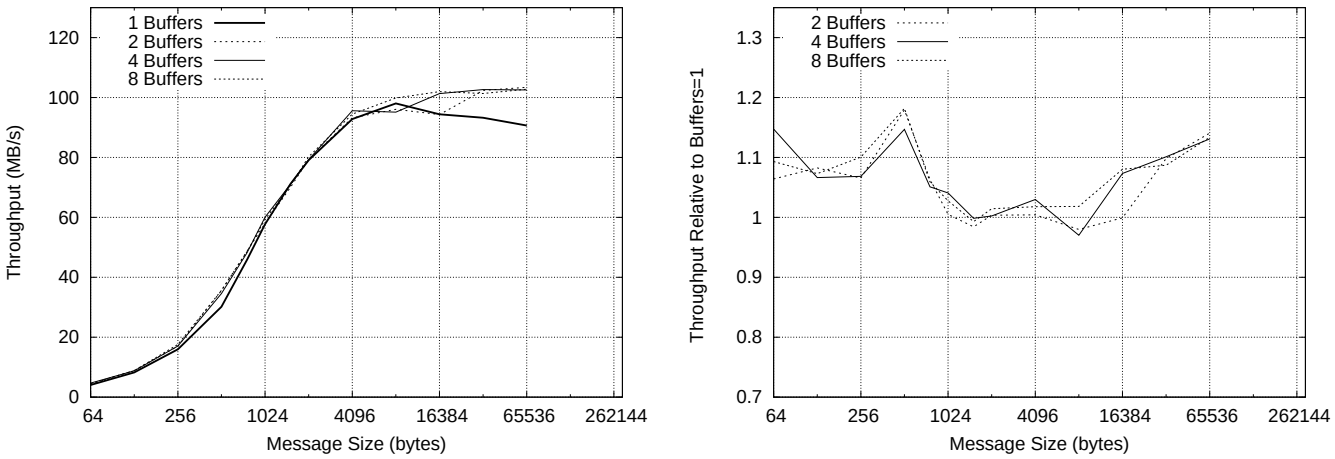


Figure 5.5: Effect on throughput of using different numbers of buffers, using a message to acknowledgement ratio of 4.

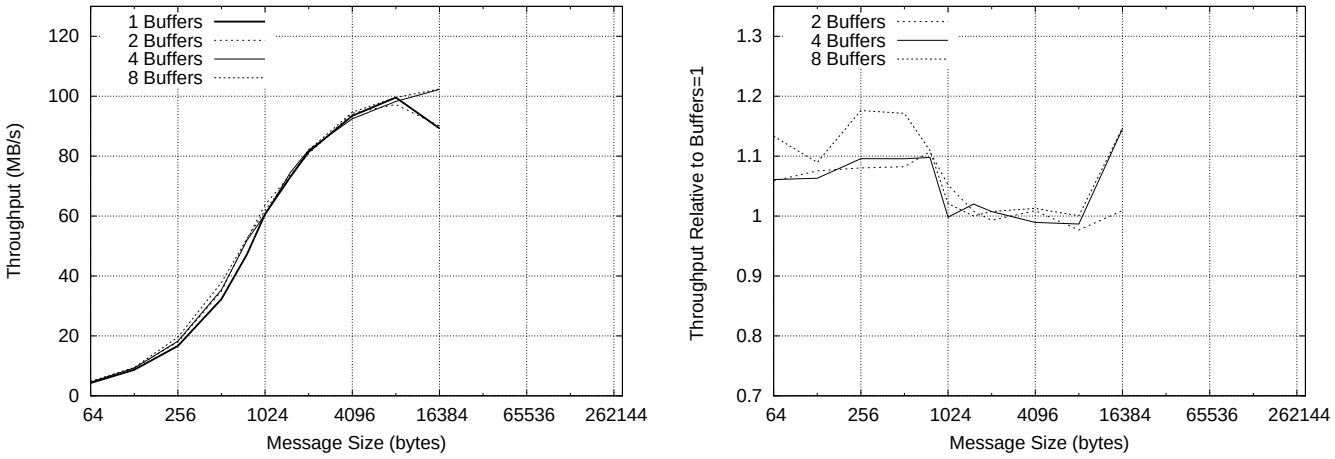


Figure 5.6: Effect on throughput of using different numbers of buffers, using a message to acknowledgement ratio of 16.

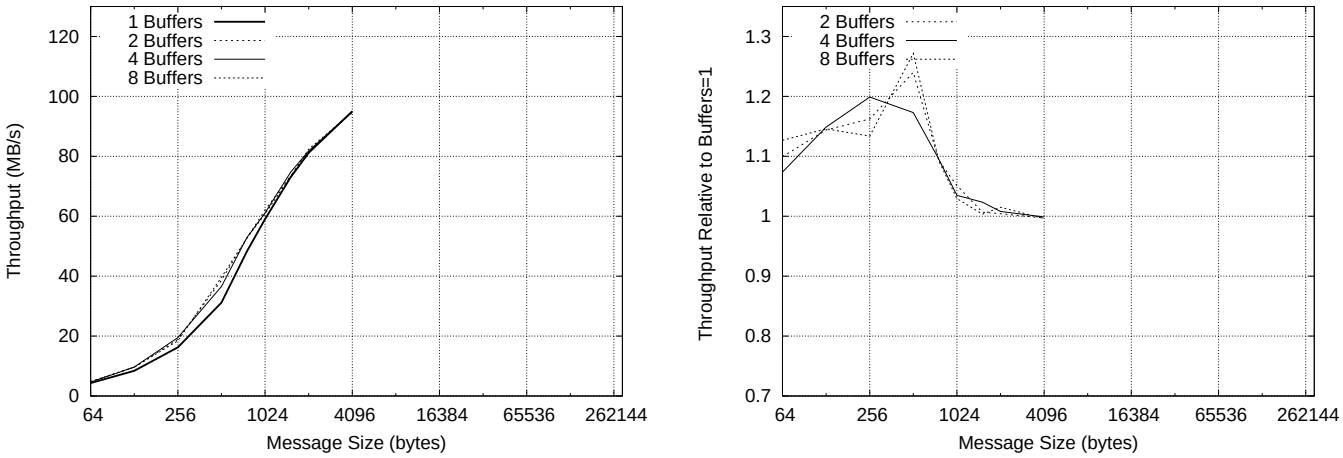


Figure 5.7: Effect on throughput of using different numbers of buffers, using a message to acknowledgement ratio of 64.

5.3.4 Effect of the Message to Acknowledgement Ratio on Performance

The message to acknowledgement ratio does have an small effect on throughput. In some cases there can be a 30% difference in performance for small message sizes. For larger message sizes changing the message to acknowledgement ratio seemed to have little effect. The effect of size of the message to acknowledgement ratio is shown in Figures 5.8, 5.9, 5.10 and 5.11.

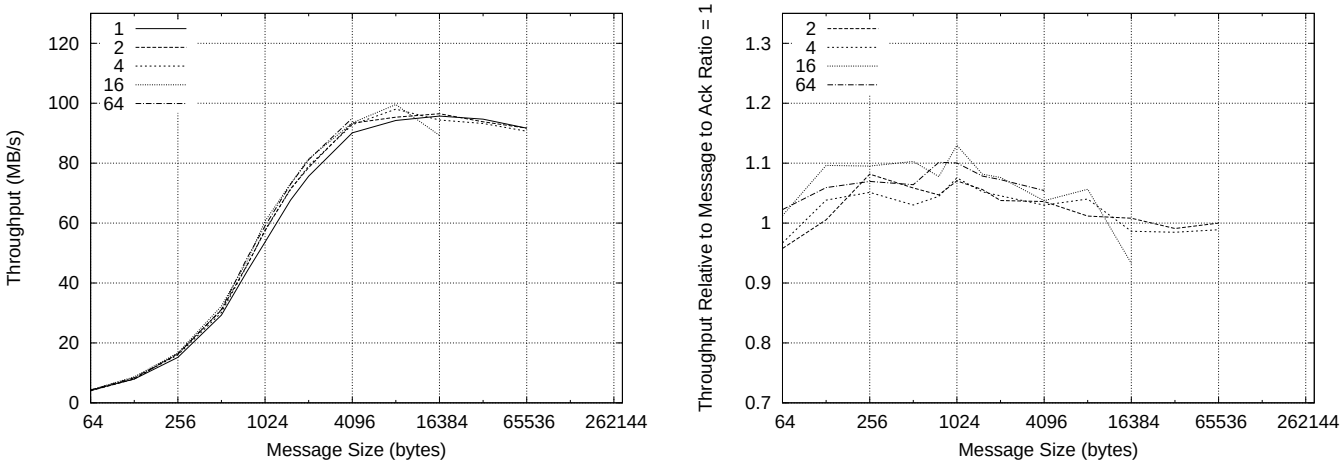


Figure 5.8: Effect on throughput of using different message to acknowledgement ratios, using 1 buffer.

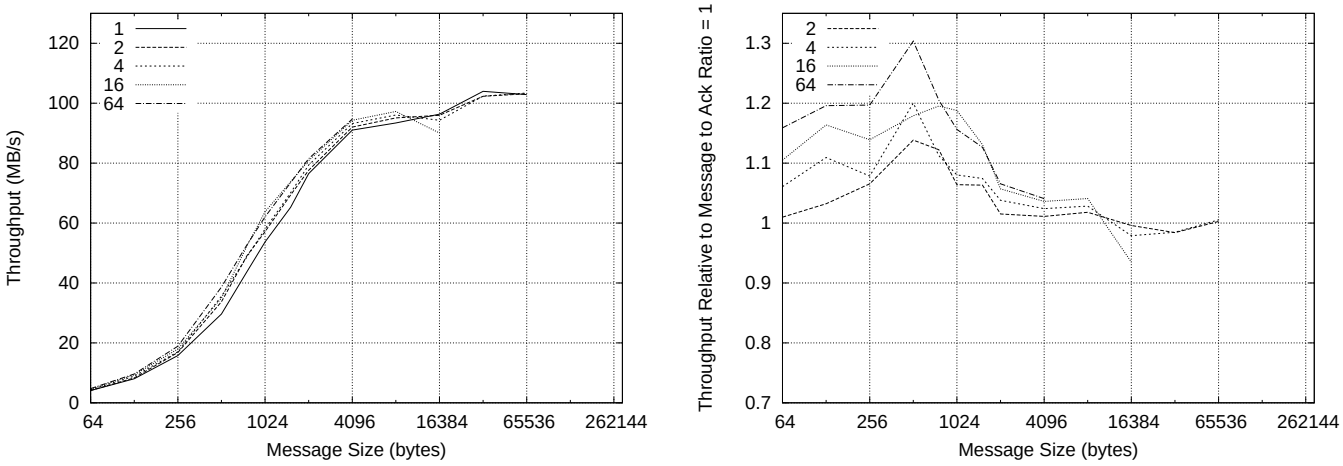


Figure 5.9: Effect on throughput of using different message to acknowledgement ratios, using 2 buffers.

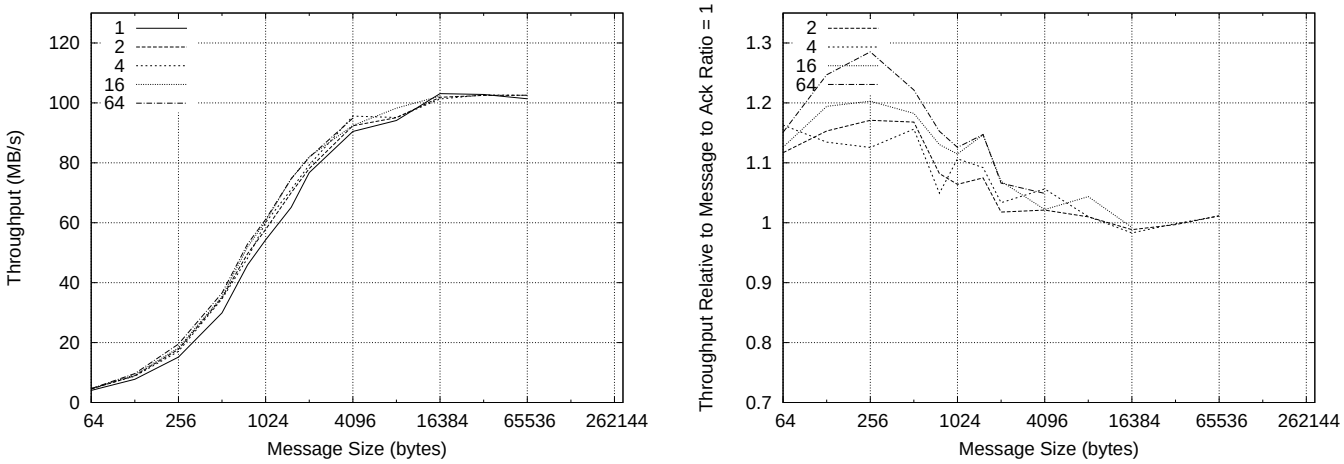


Figure 5.10: Effect on throughput of using different message to acknowledgement ratios, using 4 buffers.

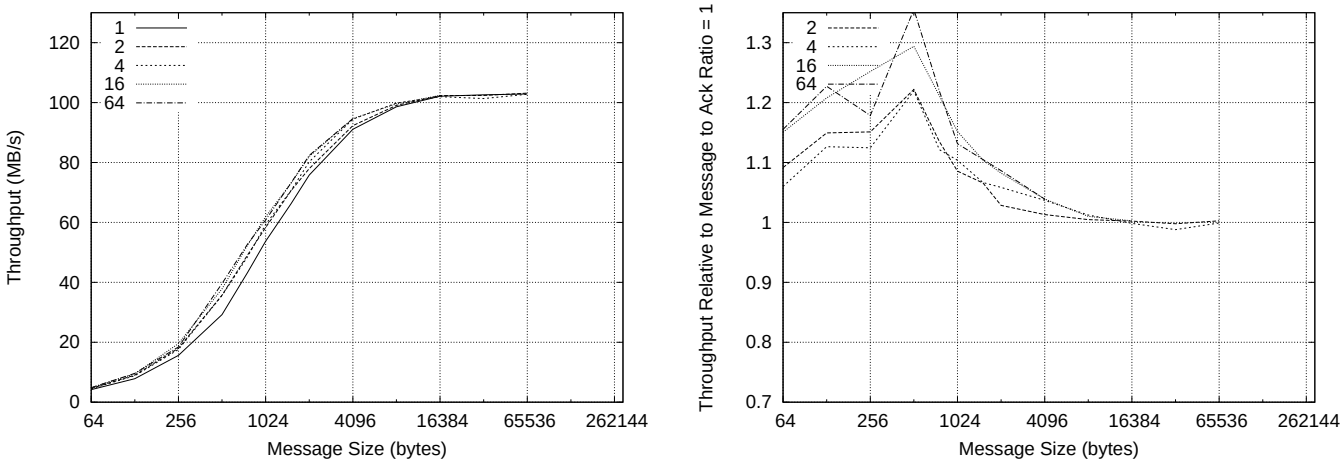


Figure 5.11: Effect on throughput of using different message to acknowledgement ratios, using 8 buffers.

5.3.5 Effect of Window Size on Performance

It was found that setting the window size too low had a negative impact on performance. A window size of 32 KB limited throughput to a little over 50 MB/s. A window size of 128 KB was found to be adequate, and further increases in window size did not increase the throughput achieved.

5.3.6 Comparison of Eager and Rendezvous Flow Control

The eager flow control scheme was found to be faster than the rendezvous flow control scheme, especially for small message sizes. For 1KB message sizes it was found to be 5.6 times faster. For larger message sizes the difference is smaller. For 128KB messages the eager flow control scheme is 1.3 times faster than the rendezvous flow control scheme.

5.4 Conclusions

In this chapter the performance of the MPJ Express communication library has been measured on the Symphony cluster at the University of Waikato. Experiments to determine impact on the latency and throughput of different messages sizes and send operations were run.

In this chapter it has been found that standard mode generally provides the best performance. For message sizes lower than 128 KB, standard mode using the eager communication protocol has a lower overhead and thus provides higher throughput and lower latency. For message sizes larger than 128 KB standard mode uses the rendezvous protocol, effectively making it equivalent to synchronous mode.

As sending a message has a certain overhead, performance can be improved if two small messages are packed together into a larger message. Standard mode has a much smaller overhead than synchronous mode, which makes it more efficient to use small message sizes. The maximum throughput achieved with standard mode was 81.5 MB/s, which was achieved with a message size of 49152 bytes. However, using a message size of 768 bytes achieved a throughput of 74.6 MB/s, which is 93% of the maximum achieved. Synchronous mode, on the other hand has a much larger overhead. This requires messages to be very large to achieve high throughput, as can been seen in Figure 5.2.

If a large amount of data is sent using standard mode, the receiving process might not have enough memory to buffer all incoming messages. Two flow control schemes were tried - one that used standard mode (which uses the eager protocol) and another which used synchronous mode (which uses the rendezvous protocol). The eager protocol scheme was found to have a higher throughput than the rendezvous scheme.

Chapter 6

Case Study: Primes

This chapter explores different ways of implementing the JStar primes program on a distributed computer. This program counts the number of primes less than a given integer using the Sieve of Eratosthenes algorithm.

6.1 Introduction

The Sieve of Eratosthenes is an ancient algorithm for finding prime numbers. It is attributed to Eratosthenes, a Greek scholar who lived from 276 to 194 BCE[30].

It finds prime numbers by first calculating the multiples of known prime numbers. Numbers which are not multiples of any prime are themselves prime numbers.

The sieve algorithm is as follows:

Given a range of numbers from 2 to Max:

1. Mark the first unmarked number as a prime
2. Mark all multiples of that number as non primes

Repeat steps 1 and 2 until all numbers are marked either prime or non prime.

An animation of this algorithm is shown in Figure 6.1.

6.1.1 Related Work

Wainwrite[35] investigates two different distributed Sieve of Eratosthenes algorithms on a hypercube computer. One algorithm is similar to the pipeline algorithm described in Section 6.6, the other is an algorithm similar to the divide search space algorithm described in Section 6.7. Although the hardware

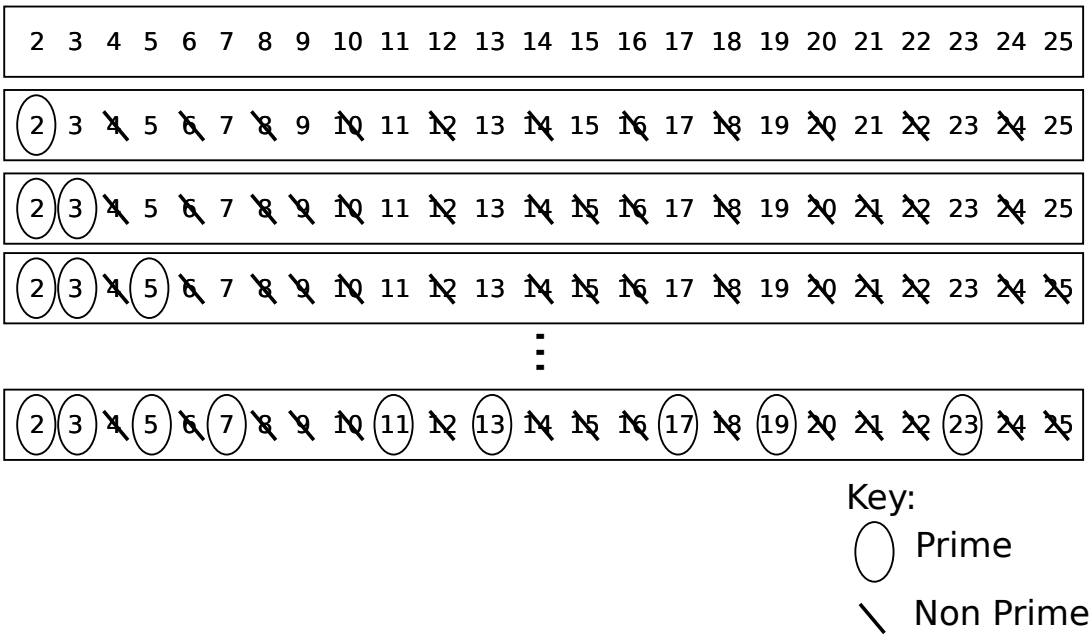


Figure 6.1: The process of the Sieve of Eratosthenes algorithm up to 25.

used for the experiments in this thesis is quite different, the conclusions about the pipeline implementation and the divide search space algorithm are quite similar.

6.2 Outline of the JStar Primes Program

A straightforward JStar implementation of the algorithm given in the introduction is presented in Figure 6.2. The program contains three relations: max(M), mult(M,P) and prime(P). The max relation serves the purpose of a constant.

The mult(M,P) relation is generated by the third rule in Figure 6.2. This rule generates multiples of primes previously found. This rule generates multiples with values of M, where $2 \cdot P \leq M < Max$. The prime rule checks each number from 1 to Max to see if it was a multiple of a previously found prime. If it is not, then it is itself a prime.

There is a mutual dependency between these relations. To know that a number is a prime, it first has to be known that it is not a multiple of any previously found prime. This in turn requires first calculating those primes.

This algorithm allows for some parallelism. The stratification order allows concurrently checking up to two times the highest prime discovered. Multiples of the primes that have been found can also be generated in parallel.

A more optimised version of the program is shown in Figure 6.3. This version of the program takes advantage of the fact that multiples less than P^2 will always be multiples of some other lower prime.

```
stratify num(N) by [N,num].
stratify mult(N,_,_)    by [N,mult].
stratify prime(N)     by [2*N,prime].
stratify num << prime.
stratify prime << mult.

type max(int).
type num(int).
type mult(int,int,int).
type prime(int).

max(5000) <-- true.
num(N) <-- max(Max), range(2,Max,N).
mult(M,P,I) <-- prime(P), max(Max), range(P, (Max//P)+1, I), M is I*P, M < Max.
prime(N) <-- num(N), N >= 2, not(mult(N,_,_)).
```

Figure 6.2: The primes JStar program.

```
stratify num(N) by [N*N,num].
stratify mult(N,_,_)    by [N,mult].
stratify prime(N)     by [N*N,prime].
stratify num << prime.
stratify prime << mult.

type max(int).
type num(int).
type mult(int,int,int).
type prime(int).

max(5000) <-- true.
num(N) <-- max(Max), range(2,Max,N).
mult(M,P,I) <--  prime(P), max(Max), range(P, (Max//P)+1, I), M is I*P, M < Max.
prime(N) <-- num(N), N >= 2, not(mult(N,_,_)).
```

Figure 6.3: The primes JStar program, improved to allow more parallelism. Changes from Figure 6.2 are shown in italics.

This makes it unnecessary to generate mult(M,P) tuples for $M < P^2$, as well as allowing for a weaker stratification order between the mult and prime relations.

In this program the stratification order allows the program to determine if numbers less than the square of the highest found prime are primes or not. This is a much weaker stratification order than the one that was used for the program in Figure 6.2. For the primes program this allows more potential parallelism and more flexibility in how the program can be implemented for a distributed computer.

As the program in Figure 6.3 is more efficient and allows more flexibility when implementing the program on a distributed computer, this version of the program was the one implemented for the experiments in this chapter.

6.3 Implementation of the JStar Primes Program as a Distributed Program

6.3.1 Partitioning

In Fosters' parallel algorithm design methodology the first step is to partition the program to achieve as much parallelism as possible. This is done by partitioning the program up into very fine-grained tasks.

To determine if each number is a prime, the JStar algorithm (Figure 6.3) checks that each number is not a multiple of any prime that has previously been found. As all the numbers have this same process applied to them, the program can be partitioned using domain decomposition.

The program could be partitioned so that each task checks one number to determine if it is prime. This would be quite a fine-grained decomposition, but it is possible to decompose this further. This task could be split into smaller tasks, where the checks to determine if the number is a multiple of a previously found prime (the mult rules in Figure 6.3) are each done in separate tasks. The output of each of these checks are collected by another task, which decides that the number is a prime only if it was not a multiple of any of the primes checked (the prime rule in Figure 6.3).

This further decomposition uncovers more parallelism and allows more flexibility in the agglomeration part of the methodology.

6.3.2 Communication

The next step in this process is to analyse which of these small tasks communicate with one another and which tasks must wait for which. In the case of the primes program, the task that determines if a number is a prime must wait for the tasks that check that it is not a multiple of a previously found prime. In

turn, these must wait for earlier prime tasks. These dependencies between tasks create a partial order in which these tasks must run.

6.3.3 Agglomeration

When partitioning the primes program a large number of fine-grain tasks are created. If all these tasks were executed the scheduling and communication overheads would be greater than the benefit from utilising parallelism. To reduce these overheads smaller tasks are agglomerated into larger tasks. By increasing the granularity of the tasks the scheduling and communication overheads are reduced.

The agglomeration stage is the part of the process where the decisions are made about how to implement the primes program as a program that is suitable for execution on a distributed computer. For the primes program, very different distributed programs can be written depending on how the small tasks identified in the partitioning process are combined into larger tasks. Two different methods of agglomeration were tried in this chapter.

- Pipeline

  One way of agglomerating these smaller tasks is to have each task to check for the multiples of a different set of primes. Each number is checked by all of the tasks. Each task ensures that the number is not a multiple of a prime allocated to that task. For example, the responsibility for filtering out multiples of 2 and 3 could be allocated to the first task, filtering out multiples of 5 and 7 could be allocated to the second task, and so forth.

  For the primes JStar program given in Figure 6.3, this means that the mult(M,P) tuples are distributed so that all mult tuples with the same value for P are generated by the same task.

  If each task checks for multiples of different set of prime numbers, then how should these tasks communicate with each other? One way is for the tasks to be organised as a filter pipeline, where each task checks for multiples of the primes allocated to that task. Numbers that are not filtered out (because they were not multiples of primes checked by that task) are sent on to the next task. Numbers that emerge from the final task are prime numbers, which can then be assigned to filter tasks.

  Another way that the tasks could be organised is that there could be a central organising task to which the filter tasks send the numbers that were not filtered out. The central organising task would find numbers not filtered out by any node, and those would be the primes. However, this scheme would require more communication than a pipeline and thus be less efficient.

- Dividing the search space

In this way of agglomerating the fine-grain tasks, each coarse-grain task checks a range of numbers for primes. Each task checks each number in the allocated range for multiples of all the prime numbers.

For example, if finding the number of primes less than 1,000 using 10 tasks, the first task would check all numbers from 2 to 99, the second task would check all numbers from 100 to 199, and so on.

The task counts the prime numbers found in the given range. All of the subtotals from the different tasks are collected and added together to come up with a final count.

6.3.4 Mapping

For the experiments in this chapter one filter task is mapped to each physical compute node.

6.4 Prime Filtering

In both of the above designs, the JStar prime counting program must check each number to see if it is a multiple of a previously found prime number. To do this it needs to calculate the multiples of previously found primes and store them in an appropriate data structure.

As there are several different data structures that can be used, an interface was created that all of the data structures implement:

```
void addPrime(long prime);
boolean lookForMult(long num);
void skipMultsSpeedup(long num);
```

For these data structures the numbers in the prime and mult relations are represented using 64 bit integers.

The addPrime method adds a prime number to the set of primes that this data structure generates multiples for.

The lookForMult method checks if a given number is a multiple of any of the prime numbers that have been added with addPrime. The num parameter of lookForMult is expected to be monotonically increasing. In other words, for each call to lookForMult the num parameter must be strictly higher than it was in the previous call.

The skipMultsSpeedup method skips generating multiples less than the value given. This is used in the Divide Search Space implementation in Section 6.7 to skip to the beginning of the sub range allocated to that filter task.

```
addPrime(num):
  add (prime,prime) to priority queue

lookForMult(num):
  while (true):
    if priority queue is empty:
      return false
    else
      remove pair from priority queue
      mult = multiplier * prime
      if (mult <= num):
        remove pair from priority queue
        add (multiplier + 1, prime) to priority queue
        if (mult == num):
          return true
      else:
        return false
```

Figure 6.4: Pseudocode for the priority queue filter.

Two different algorithms have been investigated in this report. One algorithm is based on a priority queue, which is discussed in Section 6.4.1. Another algorithm keeps an array in which each element represents a number to be checked. Non-primes are marked, and all the remaining unmarked numbers are primes. This algorithm is discussed in Section 6.4.2.

6.4.1 Priority Queue

In this data structure the next multiple of each prime is stored in a priority queue. This is implemented using the PriorityQueue class in the Java standard library. This priority queue implementation is implemented using a heap.

As the parameter for the lookForMult method always increases monotonically, multiples less than or equal to the parameter are removed from the priority queue and the next multiple of the prime is inserted. If a multiple was found that was equal to the parameter, the lookForMult method returns true to indicate that a multiple of a prime was found.

6.4.2 Mark and Check

In this data structure an array of boolean flags is used to keep track of which numbers are multiples and which numbers are not. For each prime that has been added to the data structure with the addPrime method, multiples of that prime are marked true in the array. All numbers that were not marked as

```
blockNum = 0
blockStartOffset = 0
blockEndOffset = blockSize
buffer = new boolean[blockSize]

addPrime(newPrime):
  add newPrime to list of primes
  set multiplier for prime to newPrime
  markMultiples(newPrime)

lookForMult(num):
  if num outside current block:
    advanceBlocks(num)
  index = num - blockStartOffset
  return block[index]

advanceBlock(num):
  while num outside current block:
    increment block
  recalcBlock()

recalcBlock():
  set all flags in buffer to false
  for each prime:
    markMultiples(prime)

markMultiples(prime):
  index = (prime * primeMultiplier) - blockStartOffset;

  while (index < 0):
    prime multiplier += 1;
    index = (primeValue * prime multiplier) - blockStartOffset;

  while (multInBlockIdx < blockSize):
    block[index] = true
    prime multiplier += 1
    index = (primeValue * prime multiplier) - blockStartOffset;
```

Figure 6.5: Pseudocode for the mark and check filter.

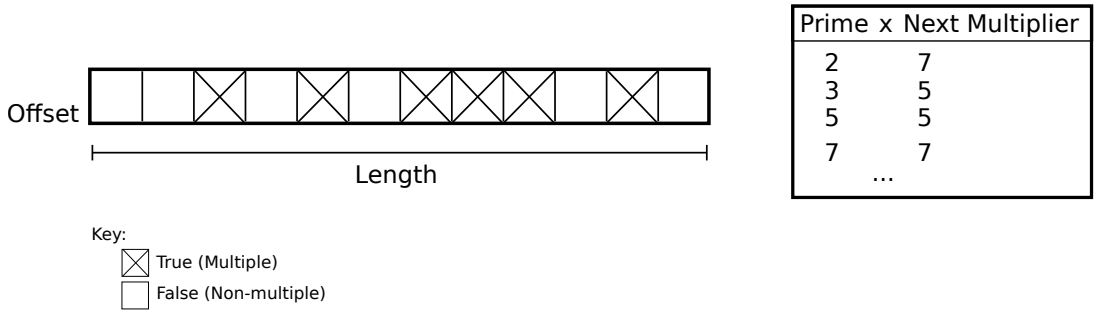


Figure 6.6: The Mark and Check data structure.

multiples are primes.

To avoid having to have a huge array, the search space is divided up into blocks. Each of these blocks are the same size, with the exception of the last block which may be shorter if the number range does not divide evenly by the block size. This allows a smaller array to be reused after all the non-multiples in a block have been counted.

Blocks are checked in ascending order. Once lookForMult is called with a parameter outside the current block, this means that the current block has finished being checked. The block number is increased until the parameter given in lookForMult is inside the block. All flags in the array are reset to false, the offset set to the beginning for the new block. Multiples marked in the new block.

When checking a very large range of numbers, storing an array containing a flag for each number is not practical. Allocating such a large array would require a very large amount of heap space. Also, in Java the maximum size for an array is 2^{31} elements.

Instead, a smaller array size can be used with an offset. After all flags in the array are checked the array can be reused. The offset is increased by the size of the array and all flags are reset to false in the array.

6.5 Single Node Runs

To calculate the speedup of the distributed versions of the prime counting program, the runtime of the prime counting program on a single node was measured. To measure runtime on a single node a variant of the prime counting program was created. This program shares the prime filtering code with the distributed versions, but only runs on one node and does not use the MPJ Express communication library. The runtime is measured by calling System.nanotime at the beginning and end of the program.

The runtime of the single node version of the program was measured using both the priority queue filter and the mark and check filter.

Detailed results of these experiments can be found in Appendix A.

6.5.1 Priority Queue Filter

The single node version of the program using the priority queue filter was run five times on five different compute nodes, counting primes up to 10 billion. The median runtime for counting the primes up to 10 billion is 7181.2 seconds.

To see if it is possible to gain any improvement over this runtime several optimisations were tested:

- Storing the value of multiple instead of recalculating it.

  The prime-multiplier pairs in the priority queue are ordered on the value of the prime multiplied by the multiplier. This value would need to be recalculated many times during operations on the heap. Instead of recalculating the value each time, it would likely be faster if it was calculated once and stored.

- Skip checking multiples of two.

  Instead of generating multiplies of two and then filtering them out, multiples of two can be skipped by checking only odd numbers from three onwards. As even numbers greater than 2 are skipped, it is not necessary to put multiples of two into the priority queue.

- Partitioning the priority queue into multiple priority queues.

  The complexity of inserting an item into a heap is O(log n). By dividing the priority queue into several smaller priority queues, the amount of time taken to insert prime-multiple pairs could be reduced. In these single node experiments, a series of priority queues are kept in an array. The maximum number of items stored in each priority queue is set to be $2^{arrayindex+3} - 1$, or {7, 15, 31, 63, 127, 255, ... }. These maximum number of items have been chosen because heaps containing these numbers are items are perfect binary heaps. Perfect binary heaps are heaps where all leaves are at the same depth.

  The priority queues are filled from the left of the array, so that multiples of smaller primes are filtered out by priority queues with fewer items stored in them. As multiples of small primes occur much more frequently than multiples of larger primes, they will have to be inserted into the priority queues much more frequently. Therefore it makes sense to store multiples of those primes in priority queues with a smaller number of items.

The effect of these optimisations on the runtime are given in Table 6.1.

Optimisations	Single Node Runtime (s)
None	7181.2
Skip Multiples of 2	5683.3
Storing $Prime \cdot Multiplier$	6103.1
Partitioning Priority Queue	3706.3
All of above optimisations	2864.8

Table 6.1: Single Node Runtimes for the priority queue algorithm with different optimisations.

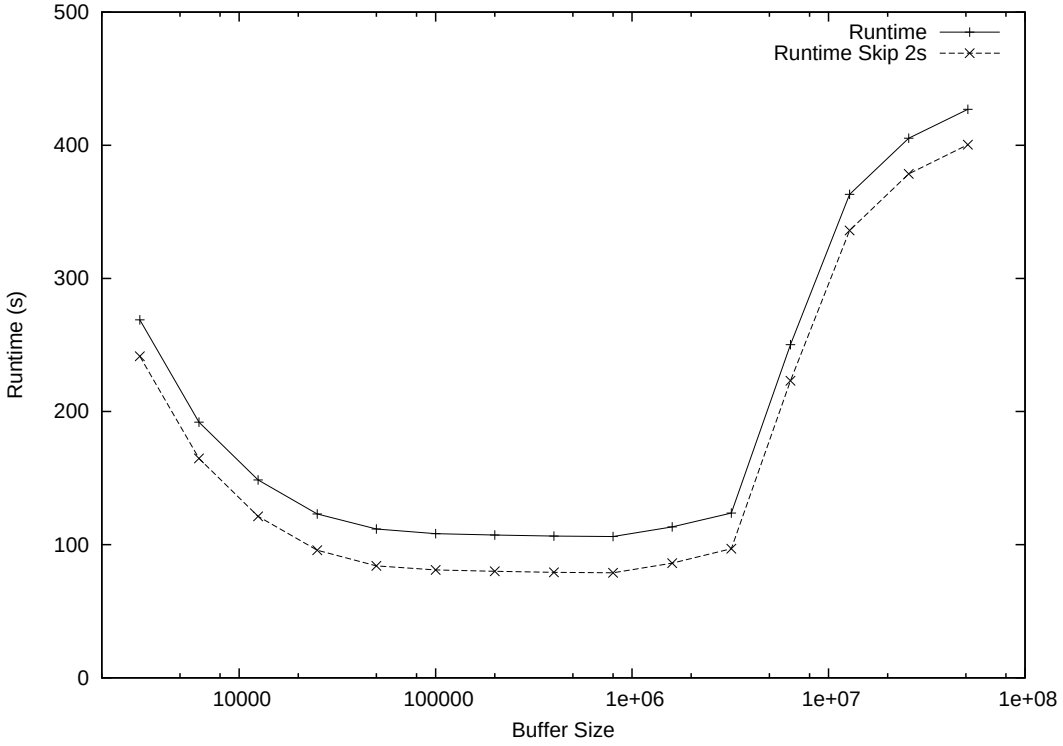


Figure 6.7: Median runtime of the single node implementation with the mark and check filter, using different buffer sizes.

6.5.2 Mark and Check Filter

The single node primes program was run using the mark and check filter with various buffer sizes. The program was run with various buffer sizes. Each experiment was run five times on five different compute nodes, counting up to 10 billion. The experiment was repeated with the optimisation of skipping the checking of multiples of two.

The median runtimes are shown in Figure 6.7.

6.5.3 Conclusions

For the priority queue algorithm, the optimisations discussed (storing the multiple rather than recalculating it, skipping multiples of two, partitioning the multiples into a number of different priority queues) significantly reduce the program runtime.

For the mark and check algorithm, using a buffer size that is too small can significantly increases the runtime required, as can be seen in Figure 6.7. For buffer sizes 200,000 up to 800,000 the program runtime slowly decreases as the buffer size becomes larger. For buffer sizes greater than 3,200,000 the program runtime sharply increases. As 800,000 is the optimal buffer size, all experiments in this chapter using the mark and check algorithm use this buffer size.

Clearly, from the above experiments, the mark and check algorithm has much higher performance. The mark and check algorithm with a buffer size of 800,000 skipping multiples of two is 36 times faster than the priority queue algorithm with all the discussed optimisations.

For calculating the speedup over the single node version of the program, the median runtime of the fastest single node program is used. This is the mark and check algorithm using a buffer size of 800,000 skipping multiples of two. The median runtime of this program is 78.9 seconds.

In Section 6.6, the distributed primes program using the priority queue algorithm is compared to the single node priority queue program. For calculating this speedup, median runtime of the single node program using the priority queue with all the discussed optimisations is used. The median runtime of this program is 2864.8 seconds.

6.6 Pipeline Filter

6.6.1 Introduction

This implementation of the primes counting program uses a pipeline, where a stream of numbers is sent from a generator task and passes through a series of filter nodes until they reach the end of the pipeline where they are counted. Each filter node is allocated a set of primes, and removes from the stream any

numbers that are multiples of those primes. As all numbers which are multiples of other prime numbers are removed, only prime numbers make it to the end of the pipeline.

This approach to implementing the prime counting program exploits the parallelism in the program by checking for the multiples of different sets of primes in different tasks, as described in Section 6.3.3.

In order to be able to allocate prime numbers to the filter nodes, those prime numbers must first be found. This is handled in this implementation by having an external organising task at the end of the pipeline which is not only responsible for counting the primes, but also allocating primes to the filter nodes. Primes for which $p^2 < max$ are counted then stored until they are allocated to a filter node, otherwise they are counted then discarded.

The limit of how far each node can filter is dictated by the stratification order. If p is the last prime allocated then the filter nodes can filter up to p^2.

Each filter node must not go past this point until the necessary primes are allocated to filter nodes. Otherwise, there would be a race condition as numbers could get through a filter node before a prime was allocated that would have filtered them out.

To ensure this doesn't happen, each filter node keeps track of up to what point the filter node can process incoming numbers without additional primes being allocated to filter nodes. When the incoming number is too high, then the filter node sends a message to the task responsible for allocating prime numbers requesting an update. Either the filter node can be allocated another prime number or be informed of the highest prime number already allocated to other filter nodes. Only after the filter node receives a reply can the filter node continue to process incoming numbers.

6.6.2 Distribute Statements for Implementing the Pipeline Filter

The distribute statements that produce the primes pipeline implementation are shown in Figure 6.8. To aid discussion, they are labelled D1, D2 and D3.

Distribute statement D1 generates all the numbers on the first task. The result of combining this distribute statement with the num(N) rule is:

```
num(T,N) <-- max(Max), range(2,Max,N), curr_task(T), T = 0 .
```

The effect of this is all num(N) with a range from 2 (inclusive) up to N (exclusive) are placed on the first node. The `curr_task(T)` constraint, with `T = 0`, restricts execution to node 0 only.

Distribute statement D2 sends num(N) tuples to the next task if N is not a multiple of the primes allocated to the current task. The result of combining the distribute statement with the num rule is:

```
num(T,N) <--
 max(Max), range(2,Max,N), curr_task(CurrT), num(CurrT,N),
 not(mult(CurrT,N,_,_)), T is CurrT + 1.
```

```
% D1
% Generate all numbers initially on the first task
distribute num(N) to T using
  curr_task(T),
  T = 0.

% D2
% Pass num(N) on to next task if it was not a multiple of
% primes on the current task
distribute num(N) to T using
  curr_task(CurrT),
  num(N),
  not(mult(N,_,_)),
  T is CurrT + 1.


% D3
% Place the prime rule on the last filter task - then distribute
% prime tuples according to prime_task_allocation(N,T)
distribute prime(N) to T using
  num_tasks(NumT),
  curr_task(CurrT),
  CurrT = NumT - 1,
  prime_task_allocation(N,T).


% The stratification order has been modified. The task number (T) is
% added to the stratification list. This is necessary to make the
% program strongly stratified.
stratify num(T,N) by [runtime,N*N,num,T].
```

Figure 6.8: Distribute statements for the pipeline filter implementation of the primes program.

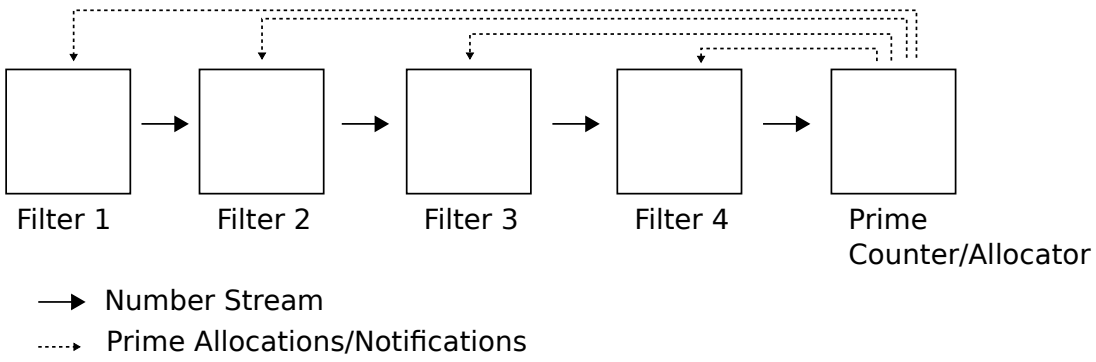


Figure 6.9: Filter nodes remove multiples of the primes allocated to them, ensuring only prime numbers reach the end of the pipeline. Primes that reach the end of the pipeline may be required by the filter nodes. The prime counter/allocator task assigns primes to filter nodes.

The clause `num(CurrT,N)` matches all `num(N)` tuples which have reached the current filter node without being filtered out. The `not(mult(CurrT,N,_,_))` clause filters out `num(N)` tuples which are multiples of the primes allocated to the current filter node.

The clause `T is CurrT + 1` calculates the task number of the next task by adding 1 to the task number of the current task.

Distribute statement D3 allocates primes to filter tasks. All numbers that are not filtered out by any of the tasks are primes. This distribute statements sends the `prime(N)` tuples to a filter task, which will cause that task to filter out multiples of that prime.

```
prime(T,N) <-- max(Max), num(CurrT,N), N >= 2, not(mult(CurrT,N,_,_)),
num_tasks(NumT), curr_task(CurrT), CurrT = NumT-1, prime_allocation(N,T).
```

The `CurrT = NumT-1` clause places this rule on the last task, which is the task with a task number equal to `NumT-1`. After the number is determined to be a prime (if `not(mult(CurrT,N,_,_))` is true), then the prime is allocated to one of the filter tasks. The allocation of primes to filter tasks is given by the relation `prime_task_allocation(N,T)`.

6.6.3 Communication Between Nodes

To reduce communication costs, the numbers sent to the next node down the filter pipeline are grouped together into larger messages. Before sending the numbers on to the next node, a filter node waits until the message buffer is full.

However this could cause the program to become deadlocked, as primes that need to be allocated to filter nodes are waiting in the filter pipeline. These primes will never reach the end of the pipeline, as the message buffers will never become full. The message buffers will never become full because the filter nodes can not send any more numbers until they are allocated new primes.

To ensure that the program can continue making progress, a flush request can be sent down the

```
canFilterUpto = 4

while true:
  num = get next num
  if num == flush flag:
    add flush flag to buffer
    send buffer
    continue

  if num == end of input flag:
    add end of input flag to buffer
    send buffer
    break

  if canFilterUpto <= num:
    add flush flag to buffer
    send buffer
    while canFilterUpto <= num:
      send new prime request to prime alloc task
      wait to receive reply
      if got new prime:
        allocate prime to this task
        canFilterUpto = prime ^ 2
      else
        canFilterUpto = lastPrimeAllocatedToOtherNodes ^ 2


  if num is not a multiple of a prime allocated to this node:
    add num to buffer
    send buffer if full
```

Figure 6.10: Pseudocode for the primes pipeline filter task.

pipeline which forces the filter nodes to immediately send the contents of the message buffer to the next node. This flush request is sent when a filter node requests an update from the prime allocating node. This ensures that all the primes still in the pipeline are flushed down to the prime allocating node.

6.6.4 Load Balancing

The best way to distribute the primes across the filter nodes in the pipeline to achieve the best performance is somewhat tricky. Having a lower number of primes on a node may reduce the amount of time spent checking if numbers are multiples of those primes. However, if less numbers are removed from the number stream then more time will be required to send those numbers down the pipeline.

Three different allocation schemes were tested in this chapter:

- Flat

Dividing the primes equally among the filter nodes.

- Triangle

  Placing linearly increasing numbers of primes along the pipeline

- Exponential

  Placing exponentially increasing numbers of primes along the pipeline

Lower prime numbers are allocated to earlier nodes in the filter pipeline, whereas larger prime numbers are allocated later on in the pipeline. This was done because the lower prime numbers will eliminate more numbers and therefore reduce the amount of time spent communicating numbers between nodes. Having low prime numbers on the first node eliminated many numbers so they did not have to be sent between nodes at all, which alleviated a major bottleneck in the pipeline.

As discussed in Section 6.5.1, using multiple different priority queues reduced the runtime of the single node primes program compared to using one priority queue. This same effect also occurs when using the priority queue with the pipeline, as each node has a different priority queue and filters out multiples of a different set of primes. How primes are allocated to nodes in the pipeline can affect the efficiency of the priority queue algorithm, and therefore the load on each node and the runtime of the program.

6.6.5 Experiments

A set of experiments were conducted to measure the level of speedup obtained using the pipeline implementation over the single node implementation. The runtime of the pipeline implementation was measured when run up to 10 billion when using 1, 2, 4, 8 and 16 nodes. For each different number of nodes, the experiment was run using the flat, triangle and exponential allocators. These experiments were done using the mark and check filter. Multiples of two were skipped, as described in Section 6.5.2.

Each combination of number of nodes and allocator was run 10 times, and the median was taken. The results for these experiments are given in Table A.4 in Appendix A. The median runtimes are shown in Figure 6.11.

The speedup is calculated using the single node runtime value given in Section 6.5. The speedups for these experiments are given in 6.12. To determine the amount of load on each node, the usertime is taken. The median usertimes of each node in the 16 node pipeline are shown in Figure 6.15.

The above experiment was repeated, except using the priority queue filter. The purpose of this experiment is to see if the pipeline implementation is able to achieve better speedups in more computationally

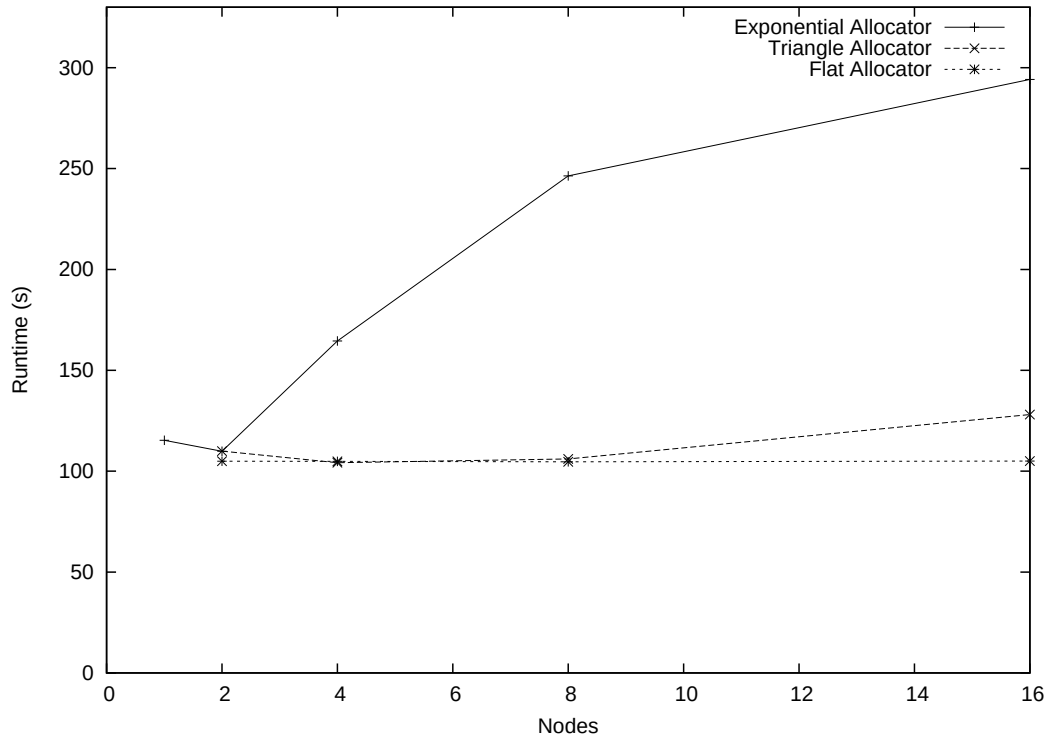


Figure 6.11: Runtime of the primes pipeline program with different allocators using the mark and check filter.

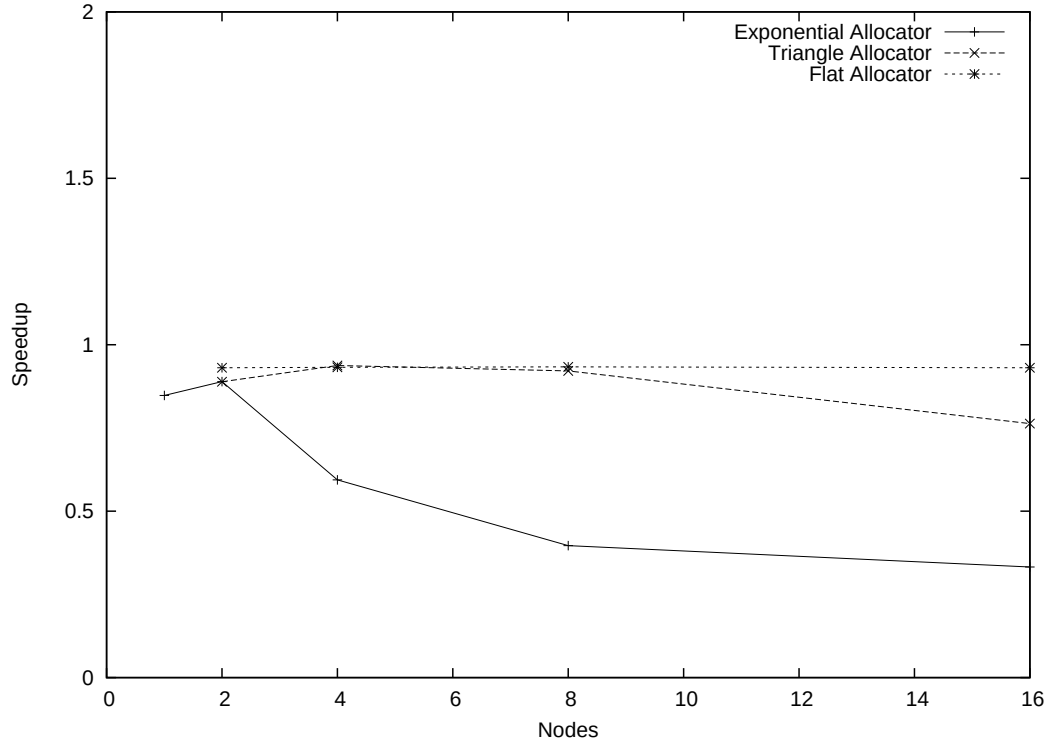


Figure 6.12: Speedup of the primes pipeline program with different allocators using the mark and check filter.

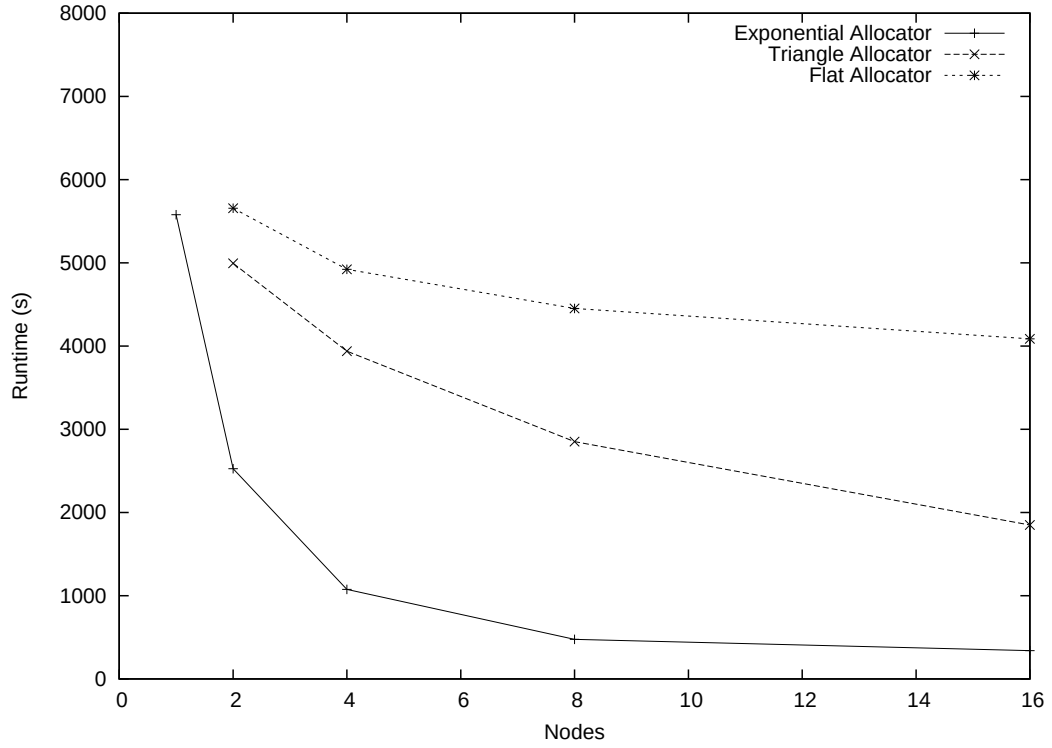


Figure 6.13: Runtimes of the primes pipeline program with different allocators using the priority queue filter.

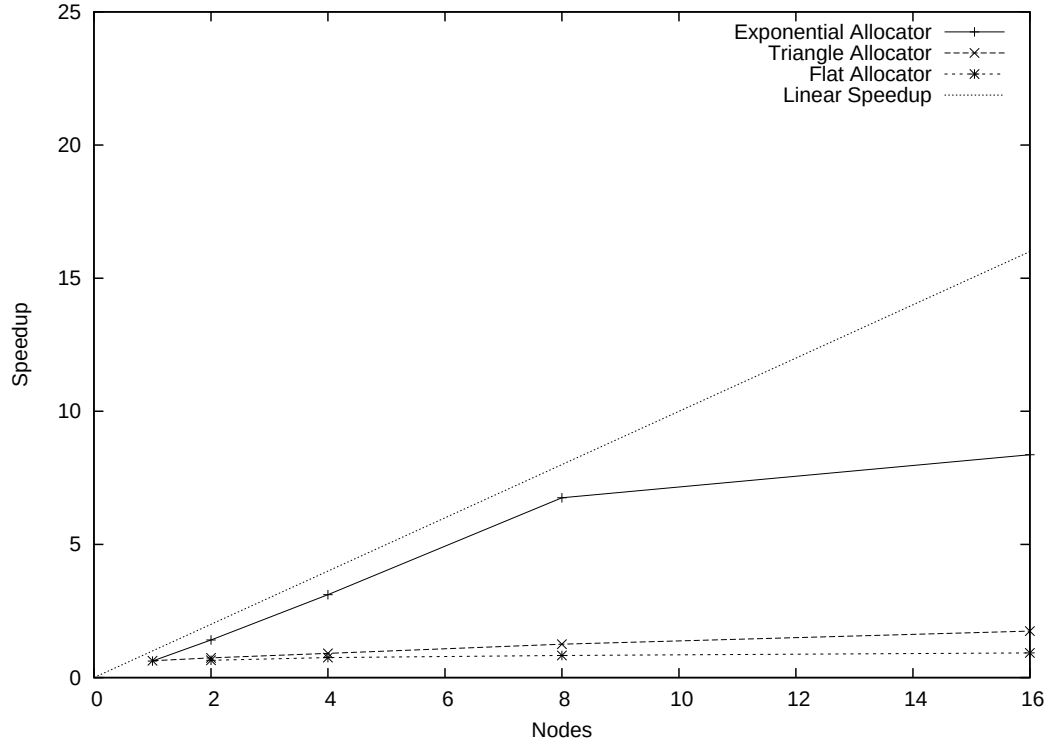


Figure 6.14: Speedup of the primes pipeline program with different allocators using the priority queue filter.

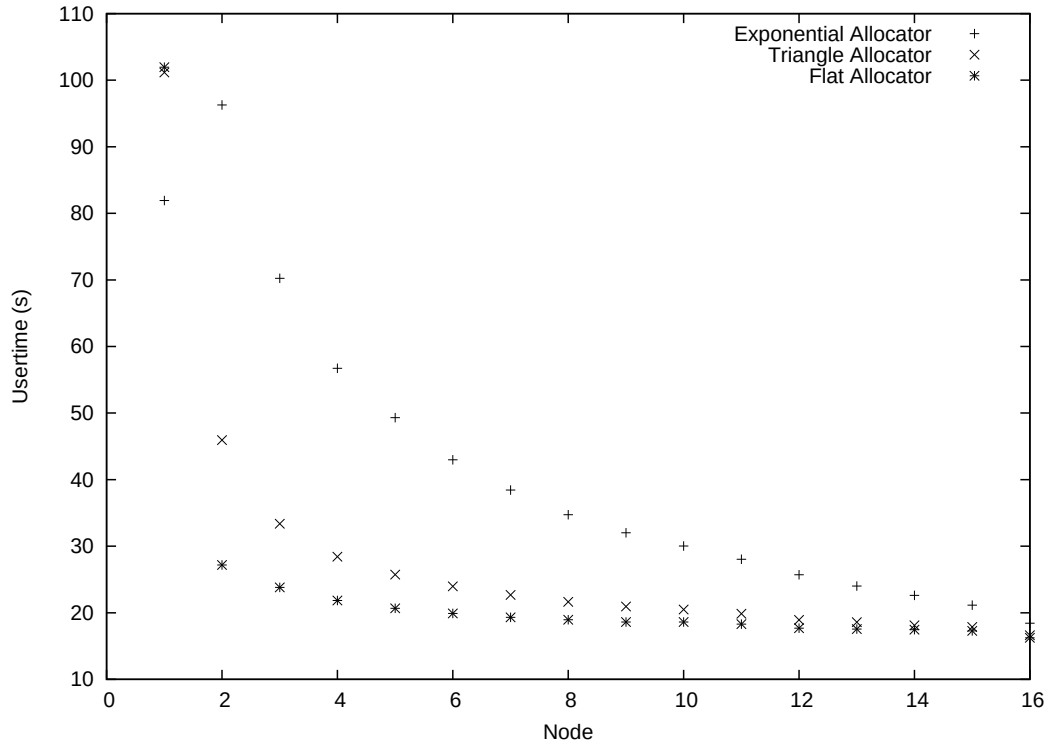


Figure 6.15: The median usertime of each node on the pipeline when run with 16 nodes using the mark and check algorithm.

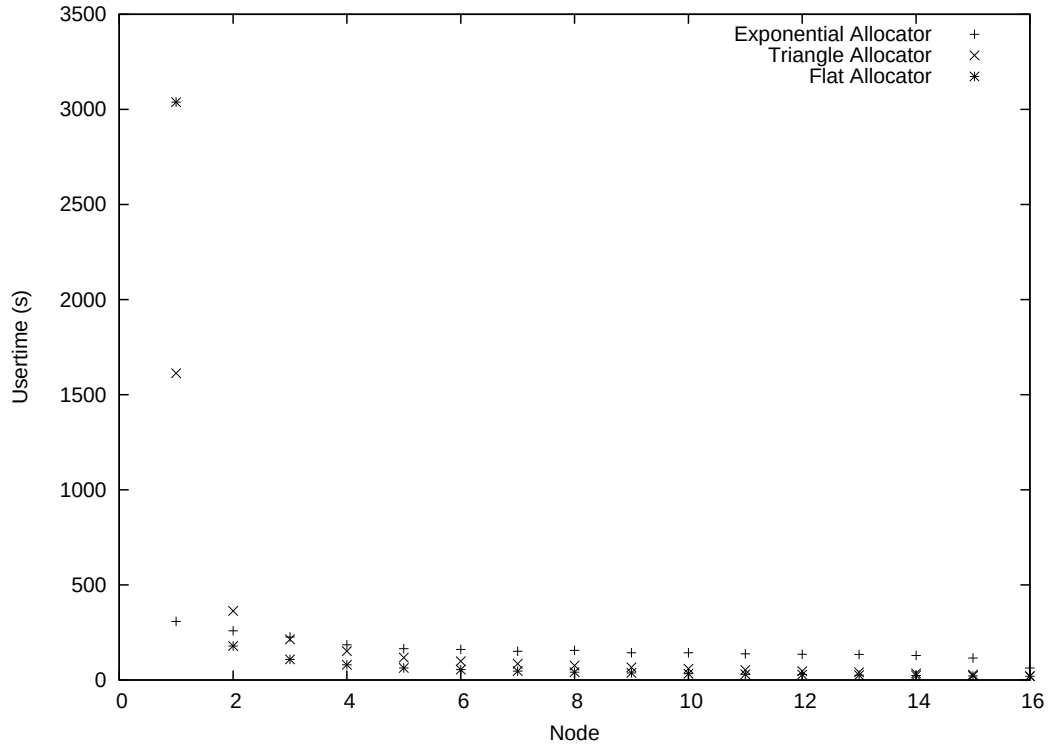


Figure 6.16: The median usertime of each node on the pipeline when run with 16 nodes using the priority queue algorithm.

intensive tasks. For these experiments, the skip multiples of two and stored multiple optimisations are used, as described in 6.5.1.

To calculate the speedup value, the single node median runtime using the priority queue filter is used as the single node runtime. The median runtimes are shown in Figure 6.13 (full results are shown in Table A.5 in Appendix A). The speedups are shown in Figure 6.14.

The median usertimes of each node in the 16 node pipeline are shown in Figure 6.16.

6.6.6 Conclusions

- No benefit over single node implementation when using the mark and check algorithm.

  The pipeline implementation using the mark and check filter algorithm is not able to achieve runtimes lower than the single node implementation using the mark and check algorithm. As both the pipeline and single node programs run faster using the mark and check algorithm, the fastest pipeline version is not competitive with the fastest single node version.

- The priority queue pipeline program is able to achieve speedup over the priority queue single node program.

  The priority queue pipeline version is able to achieve a good level of speedup over the single node priority queue program when using the exponential allocator using up to 8 nodes. When using 8 nodes it is possible to get a speedup of 6.7 over the single node version. When using 16 nodes it is only possible to get a speedup of 8.4.

- Low computation to communication ratio leads to poor performance of the pipeline when using the mark and check algorithm.

  The mark and check algorithm is efficient enough that the time spent sending and receiving the stream of numbers is similar or higher than the amount of time checking if a number is a multiple. As the amount of time saved by distributing the workload is less than the extra time required for communication, the final result is an increase in runtime rather than a reduction in runtime.

  This explains why the flat allocator had the lowest runtime and the exponential allocator had the highest. The flat allocator places the most primes on the first node, so it eliminates the most numbers without having to send them to the next node, avoiding the communication cost. The exponential allocator by contrast places the fewest primes on the first node and therefore has the cost of sending more numbers to the next node.

  In situations where there is higher ratio of computation to communication, a pipeline can achieve good levels of speedup.

The pipeline primes program using the priority queue filter algorithm achieve good levels of speedup over the single node primes program using priority queue filter algorithm, as shown in Figure 6.14. This algorithm is less efficient than the mark and check algorithm, which increases the computation to communication ratio.

Although the inefficiency of the algorithm increases the computation to communication ratio, there are still limits to how far the pipeline can be scaled up. Good speedup is achieved using 8 nodes, but the level of speedup using 16 nodes is less impressive. As more nodes are added the time spent communicating becomes the bottleneck which limits further gains in speedup. For a pipeline to be able to achieve high levels of speedup when using a large number of nodes, there must be enough work available to ensure all nodes can be kept busy.

- First node is the bottleneck of the pipeline.

  For the mark and check algorithm it is already been shown that placing a large number of primes on the first filter node is the best allocation, as the bottleneck is sending the numbers from the first node. However, this places a disproportionate amount of the workload on the first node and the leaves the rest of the nodes underutilised.

  For the priority queue algorithm the exponential allocator achieves the best runtime of the three different methods tested. Even though this allocator places very few primes on the first node, the sheer amount

6.7 Dividing the Search Space

6.7.1 Introduction

Another possible way to exploit the parallelism in the JStar primes program is to divide the search space. Each task would count the primes in one subrange, then the number of primes found in each subrange would be added together to give the total number of primes.

Instead of distributing different sets of primes to different tasks, all tasks check their subrange using the same set of primes. This set of primes is the set less than the square root of the maximum number, as explained in Section 6.2. Each task calculates this set of primes independently.

After calculating this initial set of primes, each task checks each number in it's subrange to check if it is a prime. The number of primes found is sent to the first filter task, which receives the counts of primes in each subrange and adds them together to give the total count. The pseudocode for this process is given in Figure 6.18.

6.7.2 Distribute Statements for Implementing the Divided Search Space Program

The distribute statements shown in Figure 6.17, implement the divided range prime search program. To aid discussion they have been labelled D1, D2, D3 and D4.

The first two rules (D1, D2) in Figure 6.17 calculate the initial set of primes (the primes required to calculate the rest of the primes). The other two rules (D3, D4) divide the search range into slices, and each task finds the primes within a given slice of the search range.

Distribute statement D1 combines with the mult rule, restricting it to generating `mult(M,P,I)` tuples where $M \leq \lceil\sqrt{Max}\rceil$. As M is generated by multiplying P by I, the distribute statement restricts I so that $P \cdot I \leq \lceil\sqrt{Max}\rceil$.

Likewise, distribute statement D3 restricts the mult rule, so that it only calculates `mult(M,P,I)` tuples where M is within the slice allocated to the current task.

Distribute statement D2 restricts `num(N)` so that `N` is in the initial set (i.e. $N \leq \lceil\sqrt{Max}\rceil$). This ensures that the primes in the initial set are generated on every node.

Distribute statement D4 ensures that the `num(N)` tuples are within the slice allocated to the task. This causes the program to generate the `prime(N)` tuple within that slice.

It might seem at first glance that the distribute statements are not necessary for both the `num` and `mult` rules, as the logic which divides the range into slices is duplicated in both distribute statements. But they are necessary because of the negation in the prime rule, `not(mult(N,_,_))`. If expression `not(mult(N,_,_))` is true, it may be that `N` is not a prime, but `mult(N,_,_)` may have been calculated on a different task. To have a correct program, distribute statements D2 and D4 are required so that incorrect `prime(N)` tuples are not generated.

6.7.3 Load Balancing

Load balancing in this implementation is far more straightforward than in the filter pipeline. The range of numbers to be checked for primes can be divided up evenly up among the nodes, assuming that all compute nodes have the same performance.

If the machines had varying performance then the search space could be divided into a larger number of subranges, and faster machines could be allocated a larger number of subranges.

6.7.4 Experiments

A set of experiments were run to measure the speedup achieved by this version of the program. The time taken to count the primes less than 10 billion was measured when using 2, 4, 8, 16 and 32 nodes.

```
% Statements D1 and D2 generate the set of "initial primes" (all
% primes less than sqrt(Max)) on each task.
% ------------------------------------------------------------

%    D1: Generate all multiples up to sqrt(Max) on each task
distribute mult(M,P,I) to T using
  max(Max), I =< ceil(sqrt(Max)) // P, curr_task(T).

%    D2: Search for primes up to sqrt(Max) on each task
distribute num(N) to T using
  max(Max), N =< ceil(sqrt(Max)), curr_task(T).




% Rules D3 and D4 allocate each task a "slice" of the total range to
% be searched. Each task finds primes within the slice allocated to it.
% ------------------------------------------------------------

% D3: Find multiples with M within the slice allocated to the task
distribute mult(M,P,I) to T using
  curr_task(T), num_tasks(NumT), max(Max),
  SliceSize is Max // NumT + 1,
  SliceStart is T*SliceSize,
  SliceEnd is (T+1) * SliceSize,
  SliceStartI is SliceStart // P,
  SliceEndI is (SliceEnd // P) + 1,
  I >= SliceStartI, I <= SliceEndI.

% D4 Find primes within the slice allocated to the task
distribute num(N) to T using
  curr_task(T), num_tasks(NumT), max(Max),
  SliceSize is Max // NumT + 1,
  SliceStart is T*SliceSize,
  SliceEnd is (T+1) * SliceSize,
  SliceStart =< N, N < SliceEnd.
```

Figure 6.17: Distribute statements for the divided sub-range implementation of the primes program.

```
// Find the initial set of primes
add 2 to initial primes
initial_primes_count = 1
i=3
while i * i < max:
  if i is not a multiple of initial primes:
    add i to initial primes
    initial_primes_count++
  i++

// Calculate the range of numbers checked by this task
i   = bottom of range checked by this task
top = top of range checked by this task

// Count primes in number range
prime_count = 0
while i < top:
  if i is not a multiple of initial primes:
    prime_count++
  i++

// Total prime counts from all tasks
if task number = 1:
  receive prime counts from all nodes
  total_prime_counts = total prime counts from all nodes
  primes = total_prime_counts + initial_primes_count
else:
  send prime_count to task 1
```

Figure 6.18: Pseudocode for the Divide Search Space implementation of the primes JStar program.

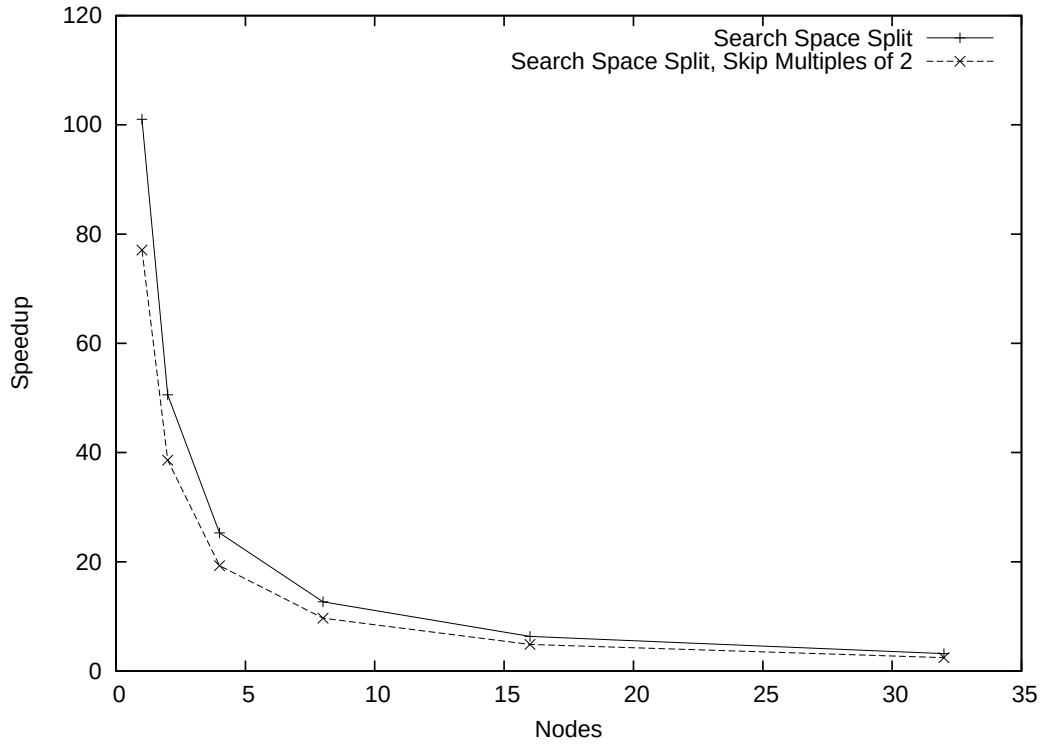


Figure 6.19: Runtime using the Search Range Distribution implementation.

The speedup is calculated using the single node runtime value given in Section 6.5.

6.7.5 Conclusions

The search range division implementation is able to achieve nearly linear speedup. The calculation of the initial set of primes (those less than the square root of the maximum) is duplicated across all nodes, but the amount of time required for this part of the program is very small. For example, when counting the primes up to 10 billion, the initial set of primes are the primes less than 100,000. This is only 0.001% of the total amount of work to be done. The only communication required is at the end of the program when the subtotals are collected and tallied.

6.8 Conclusions

From the results of the experiments presented in this chapter, it is clear that the partition range architecture is superior for the primes counting program.

- Communication costs limit the scalability of the pipeline primes program

  The partition range architecture has very little communication costs compared to the pipeline filter architecture. The high communication to computation ratio of the pipeline architecture means that

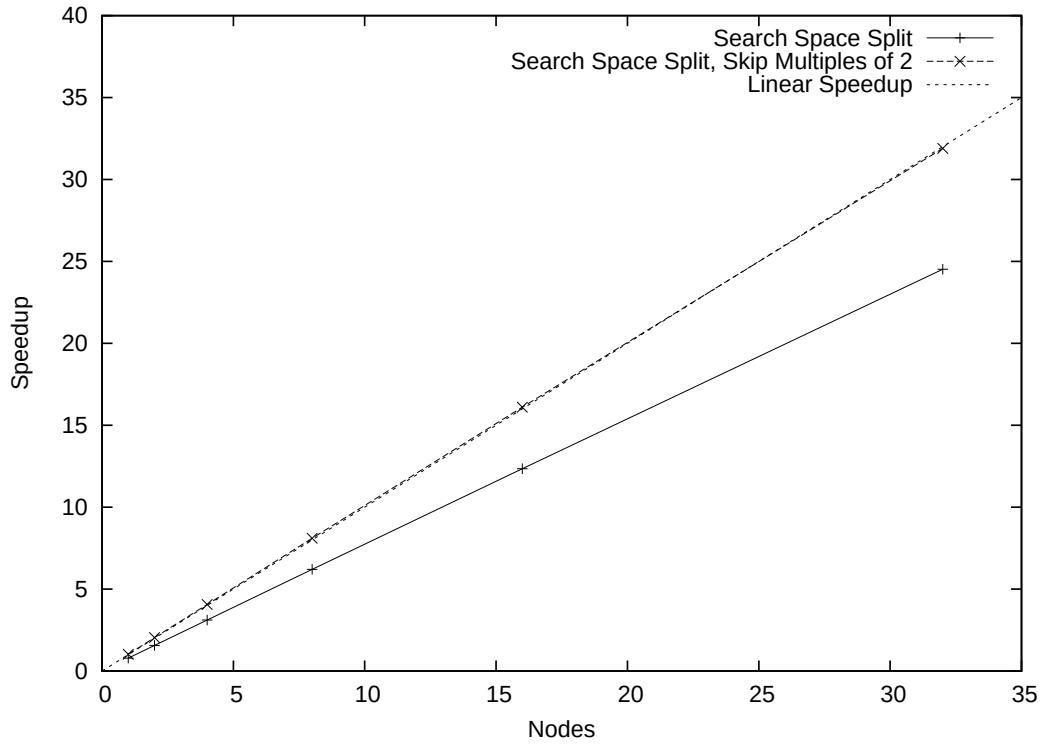


Figure 6.20: Speedup using the Search Range Distribution implementation.

the scalability of the pipeline is limited. Using the mark and check algorithm with the pipeline architecture is not able to achieve any speedup over the single node version.

Using the priority queue algorithm with the pipeline architecture is able to achieve close to linear speedup over the single node priority queue program when using 8 nodes or less. However, the program is not able to achieve good levels of speedup beyond 8 nodes.

The partition range architecture, on the other hand, scales very well. This program is basically embarrassing parallel. When using the mark and check algorithm with the partition range architecture linear speedup is obtained with 32 nodes, and excellent speedups should be possible even with very large numbers of nodes.

- The partition range architecture is easier to load balance than the pipeline architecture.

  The partition range architecture is much easier to load balance as the workload can easily be divided among the filter nodes. For the pipeline architecture, finding the optimal load balancing scheme is not as easy. Earlier nodes in the pipeline have to receive and send large amounts of data. For nodes at the end of the pipeline most of the original data has been eliminated. This can create a bottleneck at the beginning of the pipeline which leaves the rest of the nodes underutilised.

- JStar code must sometimes be written with the goal of implementation on a distributed computer

in mind.

In Section 6.2 two different JStar primes programs were considered. The first had been written for an implementation of Starlog for a standalone, single core computer. However, it would not be easy to transform this program into the Divide Search Space implementation. The original program had to be modified to make Divide Search Space implementation possible.

Chapter 7

Case Study: Conway's Game of Life

This chapter explores some of the choices that can be made when implementing the JStar Game of Life program on a distributed computer, and the effect of those choices on the implementation's performance.

7.1 Introduction

This section introduces the Game of Life and discusses different algorithms for computing it.

7.1.1 Background

The Game of Life is a cellular automaton represented as a two dimensional grid of cells. The game was invented by the British mathematician John Conway and was published in Scientific American in October 1970[20]. To "play" the game an initial layout is set out on the board. To advance the board by a generation the rules of the game are applied. The rules can be applied to the board repeatedly to calculate the subsequent generations. Through repeatedly applying the rules of the Game of Life one can observe how the pattern evolves.

7.1.2 Rules

Conway's Game of Life is played on a grid of squares where each square represents a cell. Each cell is surrounded by eight neighbouring cells. A cell can be either alive or empty. The following rules determine which cells are alive in the next generation:

- An alive cell with zero or one neighbouring alive cells dies from isolation, becoming empty in the next generation

- An alive cell with four or more neighbouring alive cells dies from overpopulation, becoming empty in the next generation
- An alive cell with two or three neighbouring alive cells will survive to the next generation
- An empty cell with three neighbouring alive cells will be alive in the next generation

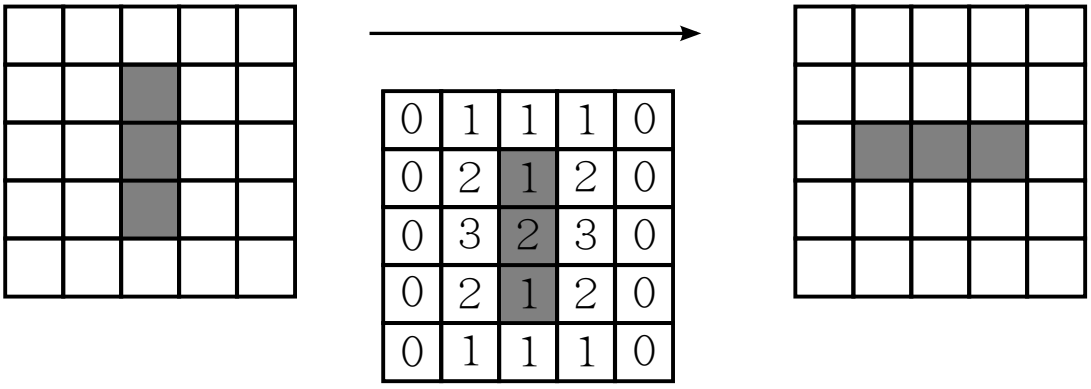


Figure 7.1: Calculating the next generation of an example life pattern. Grey squares are alive cells, white squares are empty cells. In this example two empty cells with three alive neighbours become alive in the next generation, and one live cell with two alive neighbours remains alive. Two living cells only have one alive neighbour, so do not survive to the next generation.

Although these rules are simple they are flexible enough to allow complex behaviour. The simple rules of the Game of Life have proved to be universal[7], having computational power equivalent to that of a Turing machine.

7.1.3 Life Algorithms

A simple algorithm for computing the Game of Life is to simply apply the above rules for every cell on every generation. There are quite sophisticated algorithms for computing the Game of Life which are far more optimised.

Various tricks can be applied to speed up the execution of the algorithm. One implementation[24] divides the board up into larger blocks. If there was no change in the block or on the edges of neighbouring blocks then all the cells in the block are in a stable state and the cells in the block will not change and do not need to be recomputed.

Hashlife[21] is a memoized version of the Game of Life where sub patterns are stored in a hashtable. In the Game of Life the same sub patterns often appear multiple times at different locations or at different generations. For example, there are repeating patterns called oscillators that cycle through the same cells being alive or dead after a fixed number of generations. As these behave in a predictable way the cycle only needs to be calculated once. After that it can be looked up in the hash table, greatly increasing the

execution speed. Also, if the same sub pattern appears in multiple locations on the board the entries in the hash table can be shared between them, which saves computation time. However, the algorithm is much more complex.

In this chapter the simple algorithm that applies the rules for each cell for every generation is used as a case study. The Starlog implementation of this algorithm is presented in Figure 7.2.

7.2 Implementation of the Starlog Life Program as a Distributed Program

7.2.1 Partitioning

Following Foster's parallel algorithm design methodology[18], the first step is to partition the computation into many small pieces in order to extract as much parallelism as possible. As the Game of Life consists of many cells which have the same process applied to them, domain decomposition can be applied to partition the computation. The Game of Life can be decomposed into cell value calculations, partitioning the computation so that each task computes whether a given cell is alive or dead at a given generation. In terms of the Starlog program given in the previous section, this is an application of the cell rule for differing values of X, Y and T. Each cell(X,Y,T) tuple is calculated by a different task.

7.2.2 Communication

The next step after partitioning the computation into smaller parts is to analyse how each task communicates with the others. Calculating the value of the next generation of a cell requires the value of the current cell generation as well as the value of the current cell generation of the eight neighbouring cells. In the Starlog Life program the stratification order ensures that a cell and its neighbours have been computed at time T before the next cell is computed at time T+1. In the Game of Life there is only local communication where each cell generation only needs to communicate with a small number of other program tasks. There is no global communication operation where many cell generations need to communicate at the same time. The algorithm is decentralised and the communication is distributed throughout the tasks. This lack of global communication allows the partitions of the program to run concurrently without any centralised bottleneck.

7.2.3 Agglomeration

Dividing the Game of Life algorithm into a large number of tasks where each task only calculates the value of one cell at one particular time uncovers a huge amount of parallelism. However when implemented

```
stratify cell(_,_,_,_,T) by [T,cell].
stratify neighbours(_,_,_,_,T) by [T,neighbours].
stratify cell << neighbours.
type cell(int,int,int,int,int).
type neighbours(int,int,int,int,int).
type board_size(int, int, int).

board_size(40, 20, 200) ::- true.

% count alive neighbours
neighbours(X,Y,N,T) <--
 cell(X,Y,V,T),
 board_size(Width, Height, Gens),
 X >= 0, Y >= 0, X < Width, Y < Height,
 cell((X-1) mod Width,(Y-1) mod Height,N1,T),
 cell((X-1) mod Width,Y,N2,T),
 cell((X-1)  mod Width,(Y+1) mod Height,N3,T),
 cell(X,(Y-1) mod Height,N4,T),
 cell(X,(Y+1) mod Height,N5,T),
 cell((X+1) mod Width,(Y-1) mod Height,N6,T),
 cell((X+1) mod Width,Y,N7,T),
 cell((X+1) mod Width,(Y+1) mod Height,N8,T),
 N is (N1+N2+N3+N4+N5+N6+N7+N8).

% Live cell with <2 neighbours dies
 cell(X,Y,1,PrevT), T is PrevT+1, board_size(Width, Height, Gens),
 T < Gens, neighbours(X,Y,N,PrevT), N < 2, V is 0.

% Live cell with 2,3 neighbours stays alive
 cell(X,Y,1,PrevT), T is PrevT+1, board_size(Width, Height, Gens),
 T < Gens, neighbours(X,Y,N,PrevT), N >= 2, N =< 3, V is 1.

% Live cell with >3 neighbours dies
 cell(X,Y,1,PrevT), T is PrevT+1, board_size(Width, Height, Gens),
 T < Gens, neighbours(X,Y,N,PrevT), N > 3, V is 0.

% Dead cell with 3 neighbours becomes alive
 cell(X,Y,0,PrevT), T is PrevT+1, board_size(Width, Height, Gens),
 T < Gens, neighbours(X,Y,N,PrevT), N = 3, V is 1.

% Dead cell with neighbours \= 3 is dead
 cell(X,Y,0,PrevT), T is PrevT+1, board_size(Width, Height, Gens),
 T < Gens, neighbours(X,Y,N,PrevT), N =\= 3, V is 0.
```

Figure 7.2: Starlog program implementing the Game of Life for a fixed board size. In this program the edges wrap around so that cells on the left edge of the board are neighbours with cells on the right edge. Similarly, cells on the top edge of the board are neighbours with those on the bottom.

on a distributed computer any performance benefit achieved through parallelism would be outweighed by the overhead of scheduling and communication between many small tasks. Agglomerating these smaller tasks into a few larger partitions reduces these overheads. When these overheads are reduced the parallelism can be exploited to achieve an increase in performance.

Achieving good performance requires combining these tasks into larger partitions in a way that the amount of time spent communicating data between tasked is minimised. Reducing the time spent communicating allows more time to be spent calculating the values of cells. The decisions made during the agglomeration stage for the Game of Life algorithm are focused toward increasing communication efficiency. Time spent transferring data between tasks can be reduced by reducing the amount of data sent between tasks and by sending the data using less messages to reduce the overhead of message passing.

In the Game of Life to calculate the next generation of a cell only the current value of the cell and the current values of its eight neighbours are required. The amount of data sent between tasks is reduced when the same cell as well neighbouring cells are computed by the same task. To do this the board can be divided up into blocks and each task will calculate the cells within that block. In this arrangement only the cells on the edge of the block have neighbouring cells computed by other tasks, reducing the amount of data that must be communicated between tasks.

When sending cell values between tasks it would be inefficient to send each value in an individual message. Instead, the values of all the cells on an edge of the board can be combined into a single message and sent after each generation, reducing the amount of communication overhead.

Such a computation pattern is an example of a structured grid parallel computation pattern[1]. In this parallel computation pattern, blocks exchange messages containing data on the edges where the blocks meet.

This can be taken further to allow enough data in one message to allow each task to continue computing cell values for multiple generations. This is done by using haloing. However, this requires duplicating the computation of some cell values in both tasks. Using haloing with the Game of Life program is discussed in more detail later in Section 7.6.

7.2.4 Mapping

The mapping stage is when tasks are assigned to compute nodes. For the experiments in this chapter each node is assigned one task.

As each of the tasks calculate the same number of cells, and assuming that each of the compute nodes have the same hardware (as they do in the experiments done in this chapter), then the program has been

statically load balanced across all the compute nodes.

7.3 Single Node Experiments

To calculate the speedup when using multiple nodes, the runtime when the program is run on a single node is required. To measure the runtime on a single node, a single node variant of the life program was created in which the edges of the board are copied from one side of the board to the other inside memory rather than being sent using the MPJ Express message passing library.

The single node runtime was measured by running the program with the board sizes 1280, 2560, 5120, 10240, 20480 on three different nodes, with three runs on each node to give a total of nine runs. To measure the runtime System.nanoTime was called at the start and end of the program.

The number of generations each experiment was run for was adjusted for each board size so that each experiment calculated the same number of cells. For example the 5120 by 5120 board is executed for 4096 generations and the 10240 by 10240 board is executed for 1024. As the 10240 by 10240 board is four times bigger than the 5120 by 5120 board it is only run for a quarter of the number of generations.

The results of these experiments are plotted in Figure 7.3 (the runtimes for all the experiment runs can be found in Table B.2 in Appendix B). The runtimes for each board configuration are generally quite consistent, however the maximum run for the 1280 by 1280 board size is quite a bit higher than the other runs for the same board size. There is a possibility that some anomaly, such a background daemon running, will cause runtime of the life program to be higher than usual.

It is also noticeable that the runtimes increase when the size of the board is very large. The runtime for 20480 by 20480 board is 4% higher than for the 1280 by 1280 board. This is likely to be a result of cache misses as less of the larger board sizes are able to fit into cache.

7.4 Distributed Life Program

7.4.1 Program Outline

A distributed version of the Game of Life was written to conduct experiments to assess the performance of various design decisions. The program was written in Java using the MPJ Express[5] message passing library.

Each task is responsible for calculating part of the board. The task asynchronously sends the cell values on the edge of its block to other tasks that require them. The task then waits to receive cell values from neighbouring tasks. After cell values arrive from neighbouring tasks the next generation cell values are calculated. These steps are repeated until the specified number of generations has been calculated.

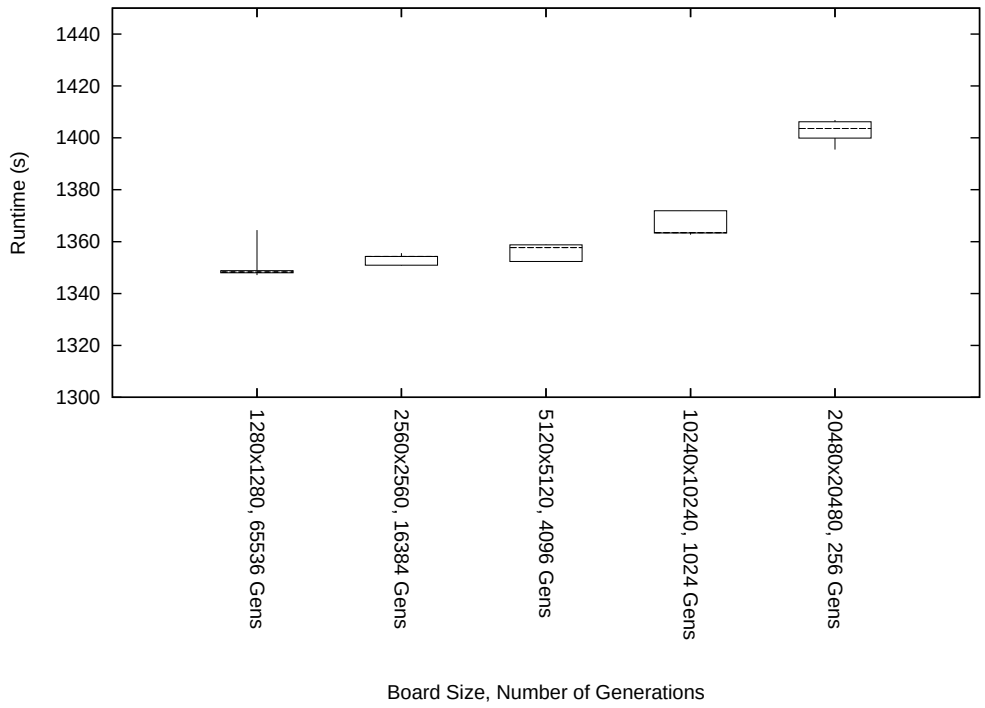


Figure 7.3: A box and whisker graph showing the runtimes of the single node version of life (for nine runs).

```
generation_counter = 0
while generation_counter < generations_to_run:
  send edges to neighbours
  receive edges from neighbours

  calculate next board generation
  generation_counter++
```

Figure 7.4: Pseudocode for the distributed life program.

These steps are the main loop which runs on each task. The pseudocode for this main loop is shown in Figure 7.4.

The pseudocode for calculating life cells on the board is shown in Figure 7.5. Each task's part of the board is stored in an array of arrays of byte. Two of these are used, one to store the current generation cell values and one for holding the next generation of the board.

For communication between tasks the cells to be transmitted are bit-packed into an array of byte, with each bit in the byte holding one cell value.

After each generation has been calculated the current generation reference is updated to refer to the array containing the newly calculated values as they become the new current generation. The next generation reference is updated to point to the old current generation array, so that the array will be

```
for (x=0; x < width; x++):
  for (y=0; y < height; y++):
    surroundingAlive = count live cells surrounding (x,y)
    if (x,y) is alive and (surroundingAlive = 2 or surroundingAlive = 3):
      next generation (x,y) = alive
    else if (x,y) is alive and surroundingAlive = 3:
      next generation (x,y) = alive
    else
      next generation (x,y) = empty
```

Figure 7.5: Pseudocode for calculating board life cells.

reused and overwritten by the new next generation values.

7.4.2 Measuring the Performance of the Distributed Program

When measuring the runtime of a computer program there are often factors that affect the runtime which are difficult for the experimenter to control. For example, most operating systems have background processes which may consume the computers resources during the experiment and cause the program to run more slowly than normal. This problem becomes more serious when measuring the runtime of the life program running on a distributed computer, as using multiple compute nodes increases the likelihood that one of the compute nodes will run more slowly than usual. When this occurs all the other compute nodes are held up as the neighbours of the slow node must wait to receive data, which in turn holds up their neighbours and so on. This makes the entire program run much more slowly than usual. In Figure 7.6 the effect of one compute node holding up the rest can be seen.

As the number of compute nodes used increases, this sort of slowdown becomes increasingly likely. When using 32 nodes, more than 20% of the experiment runs had at least one node whose calculation time exceeded the median node calculation time by more than 10% .

The problem with this is it makes it more difficult to measure the difference in runtime between different configurations as the difference in runtime is overshadowed by the slowdown caused by one node running more slowly than the others. Each board configuration is run twelve times times. However the slowdowns are large enough and frequent enough that they have a considerable impact on the mean value. The median value is usually less affected by runs with a slow node, but unfortunately slowdowns are common enough that for some experiments the majority of twelve runs had a slowdown, which means the median is affected.

To make the data from different experiments easier to compare, runs that contained an slow node that caused the runtime for that run to be abnormally high are excluded. To determine which runs

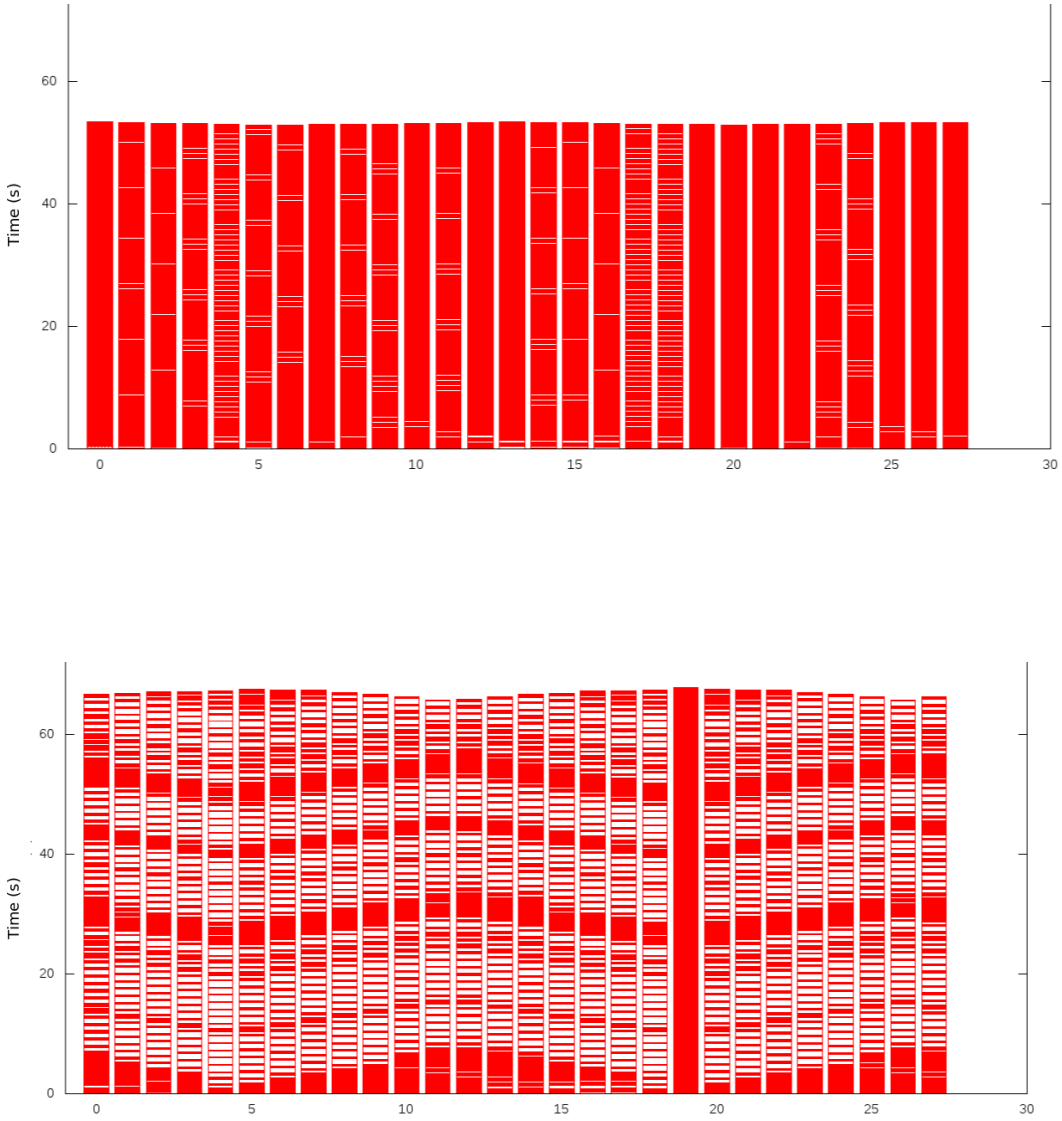


Figure 7.6: Timelines for two runs of a distributed life program. Each bar represents a compute node. The coloured area of the bar is time spent by the node computing life cells, the white area is time spent communicating with other nodes or waiting data to arrive from other nodes. In the top graph the nodes are all computing life cells at a similar rate and thus there no significant bottlenecks. In the bottom graph one compute node here is computing more slowly that the others. The other nodes are held up by this slow node and spend much of their time waiting for data to arrive. The delays propagate from node to node, as the nodes that are held up then hold up their neighbours. The layout that is used for this experiment is the 2-row brick layout which causes the double "V" pattern visible in the bottom graph.

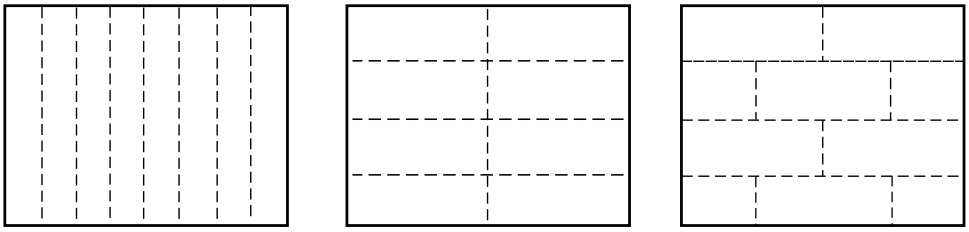


Figure 7.7: Slices, Grid and Brick layouts.

contained a slow node the time spent calculating life cells is measured using System.nanoTime. Any node which spends more than 6.1% more time calculating life cells compared to the median for that run is considered a slow node. All runs with such a node are excluded. The median is taken of the remaining runs.

The tolerance of 6.1% was chosen because this percentage discarded abnormally high runtimes but retained at least 3 runs for each experiment.

7.5 Division of Board Area Between Tasks

This section investigates different ways of dividing the life board up between tasks and the effect on the performance of the distributed life program.

7.5.1 Different Board Layouts

In Section 7.2.3 it was suggested that the amount of data that would be communicated between tasks would be reduced by computing neighbouring cells on the same task when possible, and that this could be done by dividing the life board into blocks. Each of these blocks would be allocated to a different tasks and be run on different compute nodes. The board could be divided up into blocks in different ways, or layouts.

There are two layouts that have been investigated in this report:

- Dividing the board into slices, where each partition has two neighbours.
- Dividing the board into a brick layout, where each partition has six neighbours.

Dividing the board up into a grid where each task has eight neighbours was considered but not implemented because it would require sending and receiving single-value messages containing the value of corner cells to the neighbours on the corners. We have previously seen that sending very small messages has a high amount of overhead and it was thought that the six neighbour layout would be more efficient.

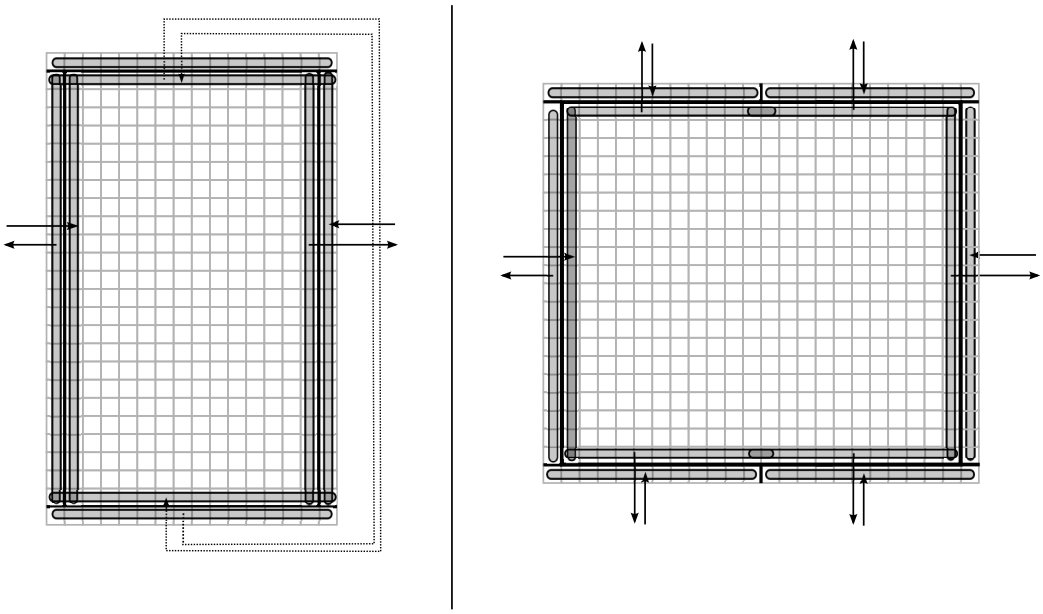


Figure 7.8: Communication between tasks using slices and brick layouts.

The slices layout has an advantage over the brick layout in that it only has two neighbours. This requires sending a message to each of the two neighbours containing the values of the cells on the edge of the block, and receiving two messages containing the cell values on edge of the neighbouring blocks. The bricks layout on the other hand requires sending six messages and receiving messages from other tasks.

However, as more tasks are added the slices get thinner. This means the "surface to area" ratio increases. The surface to area ratio is the ratio of the length of the sides of the block against the amount of cells that need to be computed inside that block. Having a lower surface to volume ratio will require less data to be transmitted between tasks. As we will see, surface area for the brick layout rises less quickly than the slices layout as more tasks are added.

We can derive an equation which compares the length of the edges for the slices and the brick layouts for square board sizes. This formula gives the proportion of the length of the edges when using the brick layout to the length of the edges when using the slices layout:

$$EdgeLengthProportion = \frac{BrickMessageEdgeLength}{SliceMessageEdgeLength}$$

$$SliceMessageEdgeLength = 2 \cdot Height$$

Tasks	Brick Layout 2 Rows		Brick Layout 4 Rows	
	Columns	Times Less Data Sent $\frac{1}{2} + \frac{1}{BrickColumns}$	Columns	Times Less Data Sent $\frac{1}{4} + \frac{1}{BrickColumns}$
4	2	1.00		
8	4	0.75		
16	8	0.63	4	0.50
32	16	0.56	8	0.38

Table 7.1: Differences in the edge to area ratio of 2 and 4 row brick layouts compared a slices layout of the same size. As more tasks are added the edge to area ratio of the slices layout grows faster as the board is divided into long thin strips.

$$BrickMessageEdgeLength = 2 \cdot \frac{Height}{BrickRows} + 2 \cdot \frac{Width}{BrickColumns}$$

By substituting the definition of SliceMessageEdgeLength and BrickMessageEdgeLength into this equation, and given that $Width = Height$, the equation can be simplified to:

$$EdgeLengthProportion = \frac{1}{BrickRows} + \frac{1}{BrickColumns}$$

As more tasks are added the edge to cells ratio the brick layout becomes better and better compared to the slice layout. However the brick layout requires sending messages to six different tasks compared to two for slices.

In the experiments in this thesis only square shaped boards were considered. The experiments in Section 7.5.3 measure the performance of both layouts for different board sizes and numbers of tasks.

The experiments were done on the Symphony cluster computer at the University of Waikato. When running these experiments the compute nodes are requested from the Torque cluster scheduler, and the compute nodes used in the experiments are the ones assigned by the scheduler. Not all experiments used the same set of nodes. Although the compute nodes have the same specifications it is possible there might be some variation between them. Also, different nodes are connected to different switches. The network latency is higher between nodes on different switches and the time spend communicating between nodes may be lengthened by network congestion.

7.5.2 Implementing the Board Layouts with Distribute Statements

The distribute statements that implement the slices layout is shown in Figure 7.9, and the distribute statements that implement the bricks layout in Figure 7.10.

Both the slices and bricks distribute statement have one statement which allocates each cell location

```
% Initial cell locations.
distribute cell(X,Y,V,T) to Task using
 board_size(Width,Height,Gens), num_tasks(Tasks),
 SliceWidth is Width // Tasks, Task is X // SliceWidth.

% Send left edge to task on left.
distribute cell(X,Y,V,T) to Task using
 curr_task(ThisTask), board_size(Width,Height,Gens), num_tasks(Tasks),
 SliceWidth is Width // Tasks,
 X = ThisTask * SliceWidth,
 Task is (ThisTask-1) mod Tasks.

% Send right edge to task on right.
distribute cell(X,Y,V,T) to Task using
 curr_task(ThisTask), board_size(Width,Height,Gens), num_tasks(Tasks),
 SliceWidth is Width // Tasks,
 X = (ThisTask+1) * SliceWidth - 1,
 Task is (ThisTask+1) mod Tasks.
```

Figure 7.9: Distribute statements which implement the slices layout.

on the grid to a particular location on the grid. The other statements send cells on the edge of a slice or brick to a neighbouring task.

The distribute statements shown in this chapter are for illustrative purposes, so they do not work for all board sizes. The slices distribute statement only works when the board width is divisible by the number of slices, so that each slice is the same width. The bricks layout requires the height of the board to be divisible by the number of rows of bricks, so that each row is the same height. The bricks layout also requires the width of the board to be divisible by the number of columns multiplied by two, so that width of the bricks is the same and can also be divided into two even pieces. Distribute statements which work for all board sizes are more complex, so in this thesis these simpler ones are shown that illustrate the general idea.

These distribute statements were tested for correctness by simulating a distributed computer in a JStar interpreter as described in Section 4.4.1.

7.5.3 Experiments to Determine the Effect of Board Layout on Runtime

To measure the performance improvement achieved when using multiple nodes a series of experiments were conducted. In these experiments the distributed version of the life program was run using different numbers of compute nodes, different board sizes and the board was divided up among these nodes using both the slices and bricks layouts discussed in Section 7.5.1. To calculate the speedup, or how many times faster the distributed version of the life program ran than a single computer implementation, the median

```
% Cell locations - which cell is calculated by which task.
distribute cell(X,Y,V,T) to Task using
 board_size(BsWidth,BsHeight,Gens), block_rows(BlkRows), block_cols(BlkCols),
 BlkWidth  is BsWidth  // BlkCols, BlkHeight is BsHeight // BlkRows,
 BlkRow is Y // BlkHeight, RowFirstX is (BlkRow rem 2) * BlkWidth // 2,
 BlkCol is ((X - RowFirstX) mod BsWidth) // BlkWidth,
 board_layout(BlkRow,BlkCol,Task).

% Send messages to upper left.
distribute cell(X,Y,V,T) to Task using
 curr_task(ThisTask), board_size(Width,Height,Gens), num_tasks(Tasks),
 board_layout(BlkRow,BlkCol,ThisTask), block_rows(BlkRows), block_cols(BlkCols),
 BlkWidth is Width // BlkCols, BlkHeight is Height // BlkRows,
 RowFirstX is (BlkRow rem 2) * (BlkWidth // 2),
 Y = BlkRow * BlkHeight,
 (X - RowFirstX) mod Width  >= BlkCol * BlkWidth,
 (X - RowFirstX) mod Width  <  BlkCol * BlkWidth + BlkWidth//2 + 1,
 TopLeftColSubtraction is ((BlkRow rem 2) + 1) rem 2,
 RecvBlkRow is (BlkRow-1) mod BlkRows,
 RecvBlkCol is (BlkCol-TopLeftColSubtraction) mod BlkCols,
 board_layout(RecvBlkRow,RecvBlkCol,Task).

% Send messages to upper right (not shown).
% Send messages to lower left (not shown).
% Send messages to lower right (not shown).
% Send messages to left (not shown).
% Send messages to right (not shown).
```

Figure 7.10: Distribute statements which implement the bricks layout. For brevity, only the upper left communication rule is shown. The `board_layout(Row,Column,TaskNum)` relation is used to look up a task number of a block given its row and column.

runtime of the single computer implementation was divided by the median runtime of the distributed computer version with the given parameters.

Each experiment was run twelve times. The runtime was measured by calling System.nanoTime at the beginning and at the end of the program on each node, and subtracting the time at the start of the program from the time at the end of the program. The highest of the node run times was taken to be the program run time.

To make the data easier to interpret, runs with abnormally slow nodes were discarded (as described in Section 7.4.2). The median of the remaining runtimes was taken to be the runtime for that experiment.

The speedup was calculated by dividing the median runtime of the single node version described in Section 7.3 by the median runtime (after removing abnormally slow runs). The speedup results are shown in Figures 7.11, 7.12 and 7.13.

The level of speedup obtained in these experiments is quite high, nearly as high as linear speedup.

The level of speedup obtained when using 32 nodes ranged from 21.5 for the 1280 by 1280 board using the 2-row bricks layout to 30.6 for the 20480 by 20480 board using the 2-row bricks layout.

Also it can been seen from the speedup graphs that larger board sizes are able to achieve a higher degree of speedup than smaller board sizes when using a large number of nodes. This is to be expected because smaller boards have a higher surface to volume than a larger board would when divided up among the same number of nodes. This means that a greater proportion of the runtime is spent communicating data between nodes.

The runtimes of these experiments can be found in Tables B.3, B.4, B.5, B.6 and B.7 in Appendix B.

7.5.4 Difference in Communication Time Between Layouts

To get a clearer picture of the difference in communication time between the different layouts, experiments similar those conducted in Section 7.5.3 were conducted, except that the part of the program that computes the life cells was deactivated. All the communication operations are still executed. The time spent inside the programs main loop was measured, rather than the runtime of the whole program so that any start up costs were not included. The time inside the main loop is spent sending and receiving messages from neighbouring nodes and copying the data received onto the life board. These experiments were repeated twelve times, and the median of the twelve runs was considered the runtime for that experiment. The runtimes were quite consistent, so there was no need to remove abnormally slow runs as was done in the experiments described in Section 7.5.3.

The results of these experiments are displayed in Table 7.2. The smaller board sizes that are run for a larger number of generations have a higher runtime. It is expected that smaller board sizes run for a

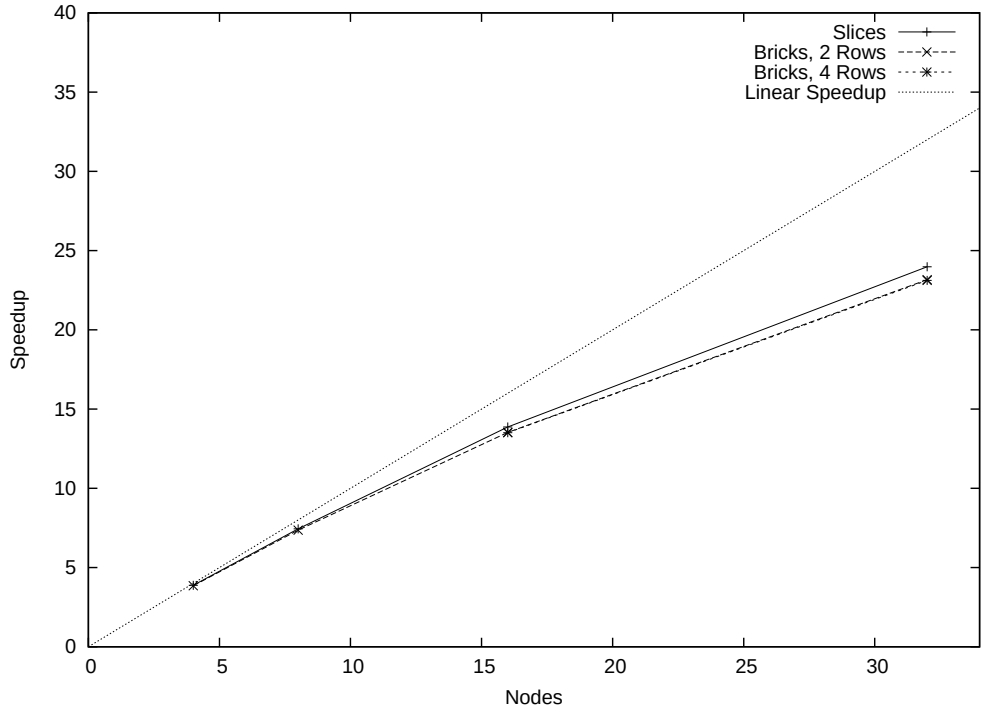


Figure 7.11: Speedup achieved using the slices, 2 row brick layout and the 4 row brick layout on a 1,280 by 1,280 board run for 1,024 generations.

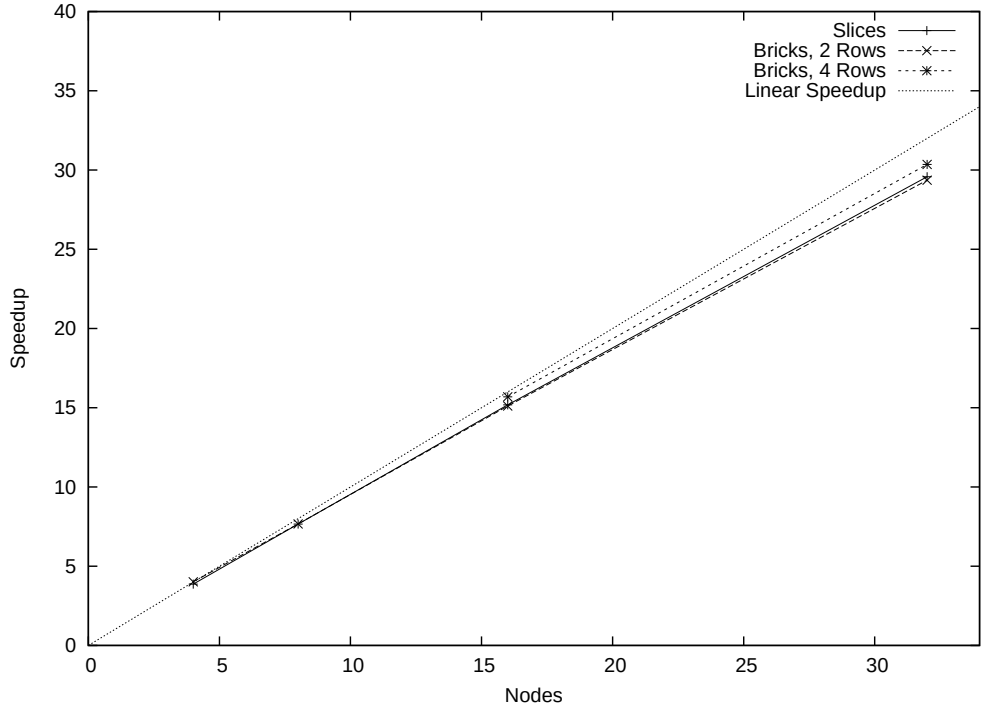


Figure 7.12: Speedup achieved using the slices, 2 row brick layout and the 4 row brick layout on a 5,120 by 5,120 board run for 4,096 generations.

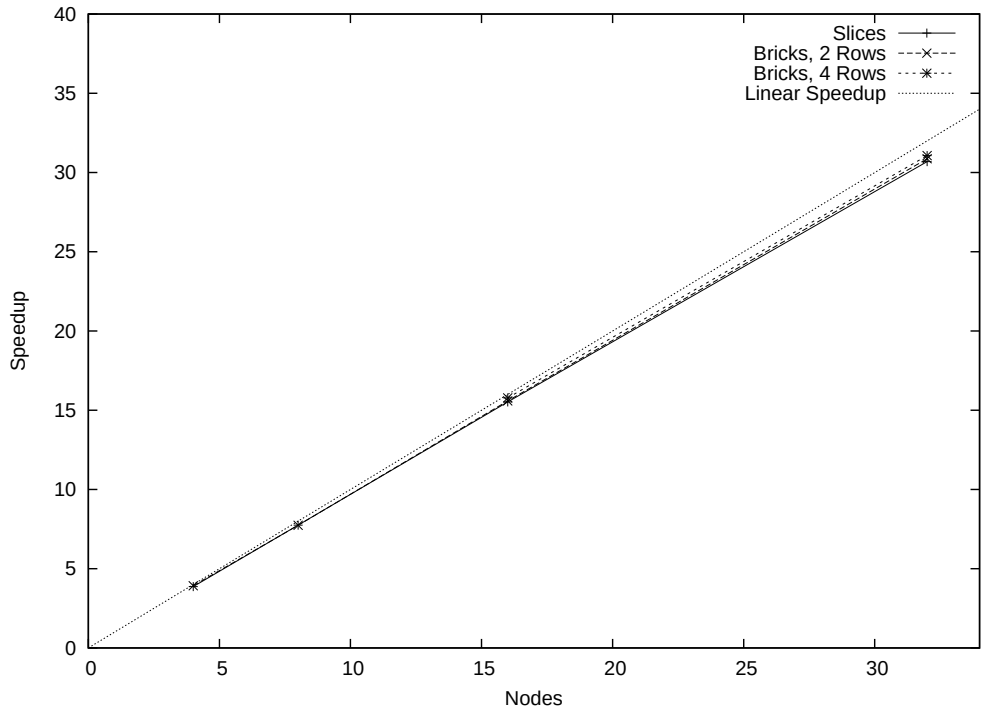


Figure 7.13: Speedup achieved using the slices, 2 row brick layout and the 4 row brick layout on a 20,480 by 20,480 board run for 256 generations.

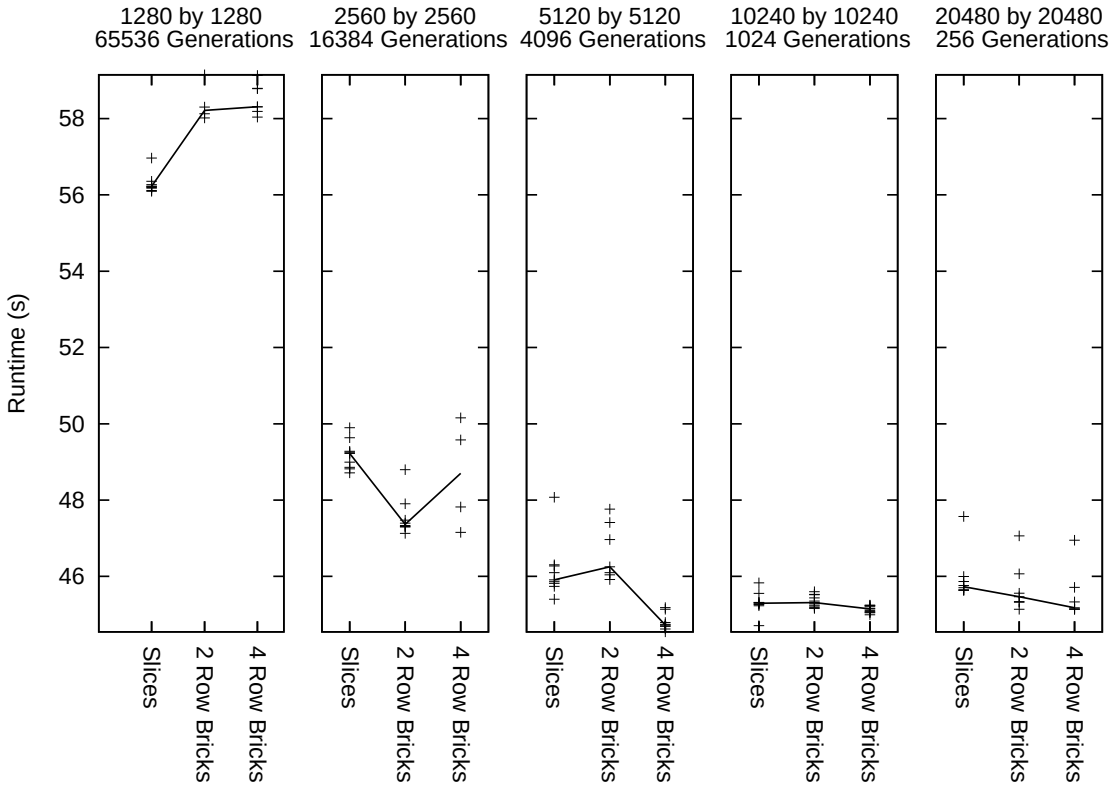


Figure 7.14: Median runtime of Life program run with 32 compute nodes using different board sizes and layouts.

Board	Nodes	Communication Only Times (s)			Communication Only Time Per Generation (ms)		
		S	B2	B4	S	B2	B4
1280 by 1280, 65536 Generations	4	8.78	11.83		0.13	0.18	
	8	9.29	11.35		0.14	0.17	
	16	9.77	11.35	11.43	0.15	0.17	0.17
	32	10.47	11.57	11.63	0.16	0.18	0.18
2560 by 2560, 16384 Generations	4	3.37	4.55		0.21	0.28	
	8	3.57	4.30		0.22	0.26	
	16	3.98	4.37	4.41	0.24	0.27	0.27
	32	4.37	4.51	4.53	0.27	0.28	0.28
5120 by 5120, 4096 Generations	4	1.67	2.18		0.41	0.53	
	8	1.82	2.13		0.44	0.52	
	16	2.01	2.23	2.27	0.49	0.55	0.55
	32	2.18	2.31	2.31	0.53	0.56	0.56
10240 by 10240, 1024 Generations	4	0.61	1.21		0.59	1.18	
	8	0.67	1.14		0.65	1.11	
	16	0.79	1.22	1.26	0.77	1.19	1.23
	32	0.90	1.27	1.26	0.88	1.24	1.23
20480 by 20480, 256 Generations	4	0.24	0.46		0.94	1.78	
	8	0.24	0.35		0.95	1.35	
	16	0.26	0.43	0.47	1.00	1.69	1.83
	32	0.35	0.49	0.48	1.39	1.90	1.86

Table 7.2: Runtimes when life cell computation is disabled.

larger number of generations would spend more time communicating as they send both more messages and more data. For all of the board sizes the runtime for slices is lower than that for bricks.

7.5.5 Conclusions

- **The distributed Game of Life program is able to achieve high speedup**

  When using four compute nodes, the speedup achieved for the 1280 by 1280 board size was 3.88 (using the slices layout). For the 5120 by 5120 board size the speedup achieved was 4.01 (using the 2-row brick layout). For the 20480 by 20480 board size the speedup achieved was 3.92 (using the 2-row brick layout). The figures here are given for the best performing layout in each case.

  When using 32 nodes the larger board sizes (5120 by 5120 and 20480 by 20480) achieved much higher speedup than the 1280 by 1280 board size. The 5120 by 5120 achieved a median speedup of 30.4 (using the 4-row brick layout). The 20480 by 20480 achieved a median speedup of 31.1 (using the 4-row brick layout). The 1280 by 1280 board achieved a median speedup of 24.0 using 32 nodes (using the slices layout).

Smaller board sizes have lower speedups than larger board sizes as they have a higher edge to area ratio and therefore spend a greater proportion of their time communicating rather than calculating life cells. Using a large number of nodes (such as 32) further increases the proportion of the time spent communicating. This is because the time spent calculating life cells is reduced by calculating the life cells in parallel across multiple nodes, while using multiple nodes does not reduce the communication time (as can be seen in Table 7.2). As the communication time is a much larger proportion of the runtime less speedup can be obtained.

- **For the 1280 by 1280 layout the slices layout is clearly faster than the brick layouts when using 32 nodes**

  This suggests that, for smaller board sizes, it is better to use the slices layout than the bricks layout (Figure 7.14). It seems that reducing the message passing overhead by sending fewer messages (as the slices layout does) reduces the communication time more than sending less data (as the brick layout does). For larger board sizes the 4-row brick layout does seem to be marginally better than the slices layout, but the difference is usually smaller than the variation between runs, so it is difficult to come to a definite conclusion.

- **The Communication Only Runtimes are faster for the slices layout than the brick layout**

  In Table 7.2 the communication only runtimes are always faster for slices than bricks. Communication only runtimes for slices layout increased when more nodes are added, but communication only runtimes for the bricks layout did not change significantly. It is not certain why this is.

7.6 Duplicating Cell Computations Across Tasks

7.6.1 Using Haloing to Reduce Number of Messages

To calculate the value of a cell, the previous generation values of all eight surrounding cells are required. As the cells on the outside edges of the block are sent to neighbouring blocks this means that messages need to be sent to and received from the neighbouring nodes on every generation. Sending messages takes time and communication time could be reduced if multiple small messages can be combined into larger messages. In order to reduce communication overhead, messages can be sent containing enough information so that each node can continue calculating cells for more than one generation without sending or receiving messages from other nodes.

This can be done by sending not just the values of the very outermost cells on the edges but also the

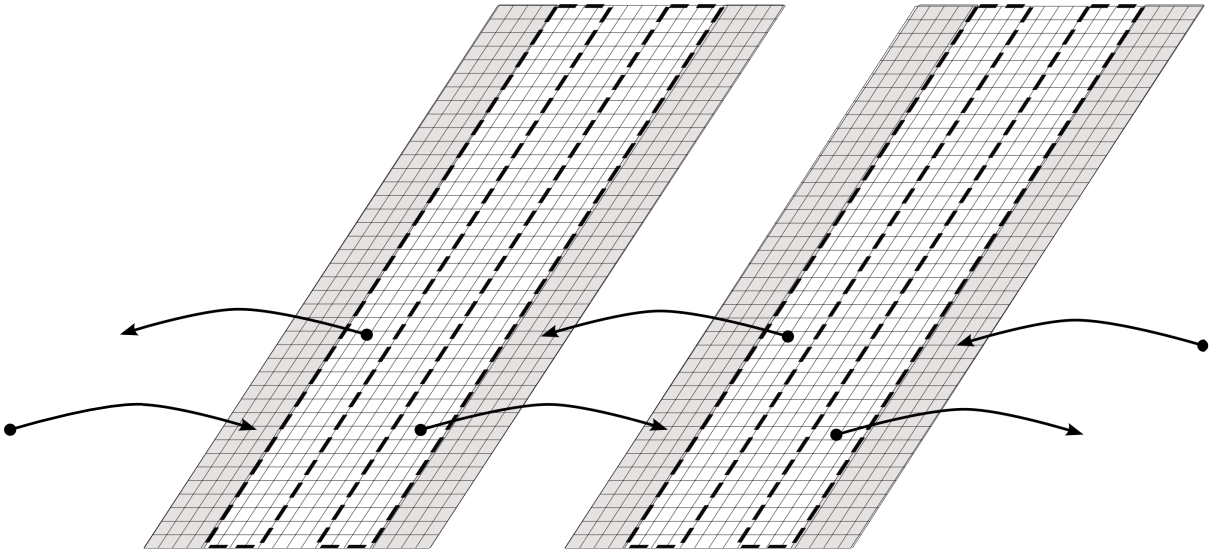


Figure 7.15: The slices layout being used with haloing. Cells from neighbouring nodes are copied into the shaded area.

ones that are on the next several adjacent layers, as can be seen in Figure 7.15. Although this reduces the communication overhead by requiring fewer messages, the cost of doing this is that some cell values have to be calculated independently on both nodes. In distributed computing, values are often recalculated locally if it can reduce time spent communicating.

Figure 7.16 shows a one dimensional rendition of this process. Multiple layers are copied to the board from neighbouring nodes. First the outside layers of the block are sent to neighbouring nodes and the outside layers of the block on neighbouring nodes are copied onto the board. The outside cell is missing neighbours and therefore the generation N+1 of this cell cannot be calculated. Effectively the width of the overlap is reduced by one cell each generation.

Then, generation N+2 is calculated. However now the cells one layer in also cannot be calculated because the cell value outside cells generation N+1 is missing. This process is continued until the overlapped cells cannot be calculated any further. Then cycle starts over and messages are sent to and received from the neighbouring nodes.

If the reduction in communication time gained by sending less messages is greater than the increase in computation time caused by calculating extra cells then the result will be an improvement in runtime.

The amount of cell calculations done per message when using a certain amount of overlap is:

$$\sum_{i=1}^{overlap-1} (\frac{width}{nodes} + 2 \cdot (i-1)) \cdot height$$

The amount of cell calculations done when using an overlap of 1 is $\frac{width}{nodes} \cdot height$. To compare this over the same number of generations to when overlap is being used, this is multiplied by the same amount of

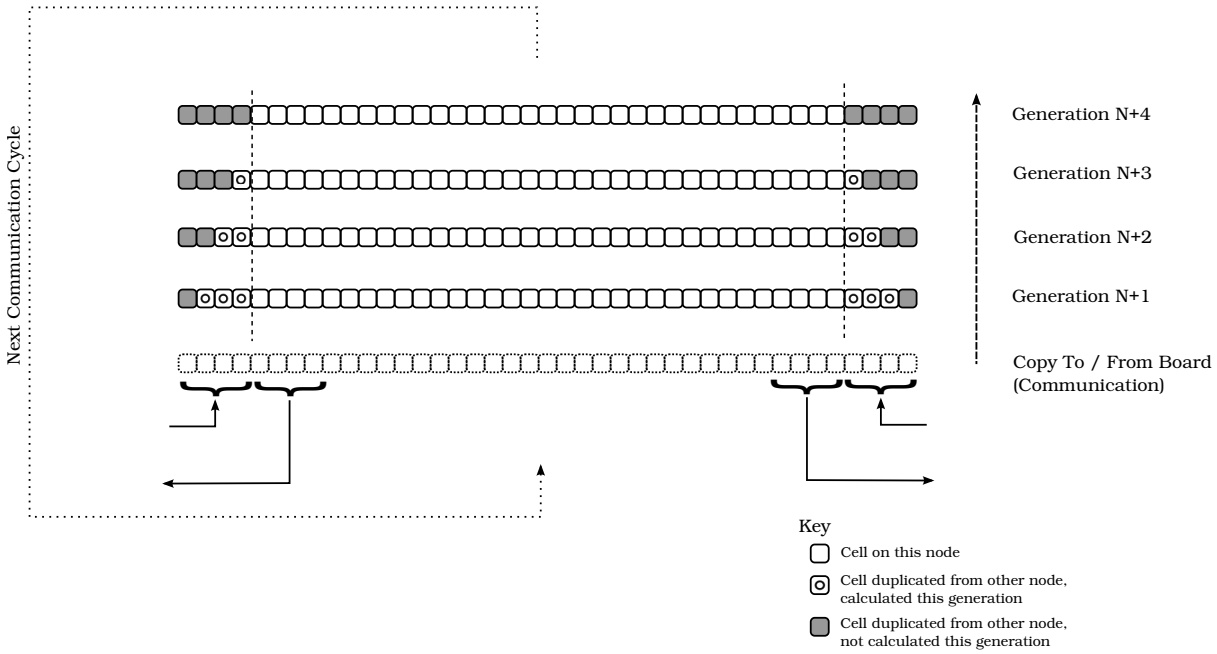


Figure 7.16: Communication example for a 4-layer halo. The outermost active layer becomes inactive after each generation.

overlap:

$$\frac{width}{nodes} \cdot height \cdot overlap$$

So, for a slices layout board the percentage extra cell computations required when using overlap is:

$$(\frac{\sum_{i=1}^{overlap-1} \frac{width}{nodes} + 2 \cdot (i-1)}{\frac{width}{nodes} \cdot overlap} - 1) \cdot 100$$

Height has been removed by simplification.

Likewise for the brick layout, the number of cell calculations per message done when using a certain amount of overlap is:

$$\sum_{i=1}^{overlap-1} (\frac{width}{cols} + 2 \cdot (i-1))(\frac{height}{rows} + 2 \cdot (i-1))$$

The number of cell calculations done with an overlap of 1 over the same number of generations is:

$$overlap \cdot \frac{width}{cols} \cdot \frac{height}{rows}$$

So, for a bricks layout board the percentage extra cell computations required when using overlap is:

Board	Overlap	Increase in Cell Computations vs. Overlap = 1 (%)									
		4 Nodes		8 Nodes		16 Nodes			32 Nodes		
		S	B2	S	B2	S	B2	B4	S	B2	B4
1280 by 1280	2	0.3	0.3	0.6	0.5	1.2	0.8	0.6	2.5	1.4	0.9
	4	0.9	0.9	1.9	1.4	3.8	2.4	1.9	7.5	4.2	2.8
	8	2.2	2.2	4.4	3.3	8.7	5.5	4.4	17.5	10.0	6.7
	16	4.7	4.8	9.4	7.2	18.8	12.0	9.7	37.5	21.7	14.7
5120 by 5120	2	0.1	0.1	0.2	0.1	0.3	0.2	0.2	0.6	0.4	0.2
	4	0.2	0.2	0.5	0.4	0.9	0.6	0.5	1.9	1.1	0.7
	8	0.5	0.5	1.1	0.8	2.2	1.4	1.1	4.4	2.5	1.6
	16	1.2	1.2	2.3	1.8	4.7	2.9	2.4	9.4	5.3	3.6
20480 by 20480	2	0.0	0.0	0.0	0.0	0.1	0.0	0.0	0.2	0.1	0.1
	4	0.1	0.1	0.1	0.1	0.2	0.1	0.1	0.5	0.3	0.2
	8	0.1	0.1	0.3	0.2	0.5	0.3	0.3	1.1	0.6	0.4
	16	0.3	0.3	0.6	0.4	1.2	0.7	0.6	2.3	1.3	0.9

Table 7.3: Percentage extra cell computations required for different amounts of overlap.

$$\left(\frac{\sum_{i=1}^{overlap-1}\left(\frac{width}{cols}+2\cdot(i-1)\right)\left(\frac{height}{rows}+2\cdot(i-1)\right)}{overlap\cdot\frac{width}{cols}\cdot\frac{height}{rows}}-1\right)\cdot 100$$

Table 7.3 shows the increase in the amount of cell computations (compared to using an overlap of 1) as a percentage. The table shows that the percentage of extra cells that need to be calculated when using the 4 row brick layout is much lower than when using the slices layout. This is because the 4 row brick layout has a lower surface to area ratio (discussed in Section 7.5.1). Haloing layouts that have a lower surface to area ratio not only communicate less data but also calculate fewer additional life cells. It is expected that the amount of time spent calculating will be proportional to the number of life cells to be calculated.

7.6.2 Implementing Haloing Using Distribute Statements

As shown in for the Pascal's Triangle example in Chapter 4, haloing can be implemented by using distribute statements. The distribute statements which implement haloing for the Game of Life program are shown in Figures 7.17 and 7.18.

Haloing is done in a similar way as for the Pascal's Triangle program, described in Section 4.3.6.

7.6.3 Experiments to Determine the Effect of Haloing on Program Runtime

A set of experiments was run to determine how using haloing affected runtime. Each combination of board size and layout tested in the previous experiments (in Section 7.5.3) was also tried with different

Board	Overlap	Increase in Time Spent Calculating Life Cells vs. Overlap = 1 (%)									
		4 Nodes		8 Nodes		16 Nodes			32 Nodes		
		S	B2	S	B2	S	B2	B4	S	B2	B4
1280 by 1280	2	0.2	-0.2	-0.5	-0.9	-0.8	-1.7	-2.1	-2.1	-3.8	-4.2
65536 Generations	4	0.5	0.1	0.5	-0.6	1.0	-1.6	-2.2	0.5	-3.4	-4.9
	8	1.4	0.8	2.4	0.7	4.6	1.0	-0.5	8.0	0.8	-2.4
	16	4.2	3.5	7.4	4.6	14.4	6.7	3.8	26.3	10.8	3.7
5120 by 5120	2	-1.6	0.0	-1.7	-2.6	-2.1	-0.1	-0.4	-1.9	0.4	0.4
4096 Generations	4	1.1	0.1	0.3	-2.3	-0.5	-1.8	-0.0	1.1	-1.2	-0.0
	8	0.7	1.1	0.7	-1.6	1.4	-0.9	0.7	3.2	0.3	0.6
	16	-0.4	1.8	0.9	-0.6	2.6	0.2	2.0	6.7	3.8	3.3
20480 by 20480	2	-0.6	0.5	0.1	-0.5	-0.2	1.1	0.4	-0.1	0.5	-1.0
256 Generations	4	-0.4	-1.0	0.2	0.5	-0.0	1.9	-0.0	0.2	2.3	0.3
	8	-0.4	-0.1	0.3	-1.3	0.3	2.2	1.6	0.9	3.1	-0.0
	16	-0.1	1.3	0.5	-0.2	1.0	1.8	1.5	2.2	4.0	0.8

Table 7.4: Percentage increase in average time spent by nodes calculating life cells.

```
% Set initial cell locations.
distribute cell(X,Y,V,T) to Task using
 T=0, board_size(Width,Height,Gens), num_tasks(Tasks),
 SliceWidth is Width // Tasks, Task is X // SliceWidth.

% Cells continue on the same task.
distribute cell(X,Y,V,T) to Task using
 curr_task(Task), T > 0.

% Send left edge of board section to the task on the left.
distribute cell(X,Y,V,T) to Task using
 curr_task(ThisTask), board_size(Width,Height,Gens), num_tasks(Tasks),

 T rem H = 0, % send messages every Nth generation

 SliceWidth is Width // Tasks, halo_size(H),
 X >= ThisTask * SliceWidth, X < ThisTask * SliceWidth + H,
 Task is (ThisTask-1) mod Tasks.

% Send right edge of board section to the task on the right.
distribute cell(X,Y,V,T) to Task using
 curr_task(ThisTask), board_size(Width,Height,Gens),
 num_tasks(Tasks), SliceWidth is Width // Tasks, halo_size(H),

 T rem H = 0, % send messages every Nth generation

 X >= (ThisTask+1) * SliceWidth - H, X < (ThisTask+1) * SliceWidth,
 Task is (ThisTask+1) mod Tasks.
```

Figure 7.17: Distribute statements which implement the slices layout with haloing.

```
% initial cell locations
distribute cell(X,Y,V,T) to Task using
 T=0, board_size(BsWidth,BsHeight,Gens), block_rows(BlkRows),
 block_cols(BlkCols), BlkWidth  is BsWidth // BlkCols,
 BlkHeight is BsHeight // BlkRows,

 BlkRow is Y // BlkHeight, RowFirstX is (BlkRow rem 2) * BlkWidth // 2,
 BlkCol is ((X - RowFirstX) mod BsWidth) // BlkWidth,
 board_layout(BlkRow,BlkCol,Task).

% cells continue on the same task
distribute cell(X,Y,V,T) to Task using
 curr_task(Task), T > 0.

% Send upper left section of board to upper left task.
distribute cell(X,Y,V,T) to Task using
 curr_task(ThisTask), num_tasks(Tasks),halo_size(H),

 T rem H = 0, % send messages every Nth generation

 board_size(Width,Height,Gens),
 board_layout(BlkRow,BlkCol,ThisTask),
 block_rows(BlkRows), block_cols(BlkCols),
 BlkWidth  is Width  // BlkCols, BlkHeight is Height // BlkRows,
 RowFirstX is (BlkRow rem 2) * (BlkWidth // 2),

 % select X and Y in the copy zone on this task
 Y >= BlkRow * BlkHeight, Y < BlkRow * BlkHeight + H,
 (X - RowFirstX) mod Width >= BlkCol * BlkWidth,
 (X - RowFirstX) mod Width <  BlkCol * BlkWidth + BlkWidth//2 + H,
 TopLeftColSubtraction is ((BlkRow rem 2) + 1) rem 2,
 RecvBlkRow is (BlkRow-1) mod BlkRows,
 RecvBlkCol is (BlkCol-TopLeftColSubtraction) mod BlkCols,
 board_layout(RecvBlkRow,RecvBlkCol,Task).


% Send upper right section of board to upper right task.
% (not shown)

% Send lower left section of board to lower left task.
% (not shown)

% Send lower right section of board to lower right task.
% (not shown)

% Send left section of board to left task.
% (not shown)

% Send right section of board to right task.
% (not shown)
```

Figure 7.18: Distribute statements which implement the bricks layout with haloing. The `board_layout(Row,Column,TaskNum)` relation is used to look up a task number of a block given its row and column. For brevity, only the upper left communication rule is shown.

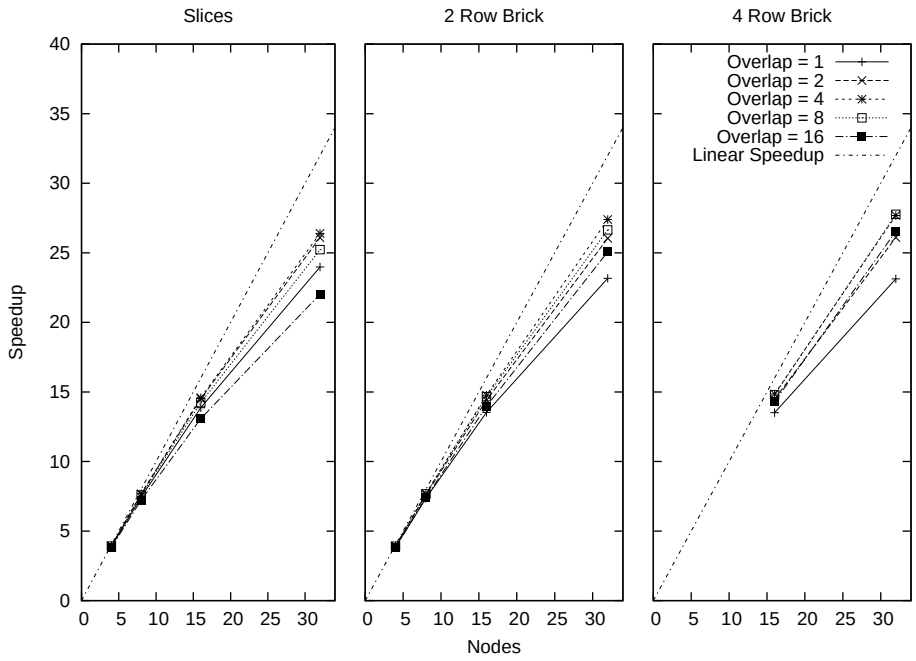


Figure 7.19: Speedup computing 1280 by 1280 for 65536 generations.

amounts of overlap. The number of layers overlap tried was 1 (no duplicated cell computation), 2, 4, 8 and 16. Each of these experiments was run twelve times. To make the data more consistent, runs with abnormally slow nodes were discarded (as described in Section 7.4.2). The median of the remaining runtimes was taken to be the runtime for that experiment.

These runtimes were compared to the single node runtimes (Section 7.3) to calculate the speedup value. The speedups of the experiments are shown in Figures 7.19, 7.20 and 7.21. More detailed data, including the runtimes of all experiment runs, can be found in Appendix B.

7.6.4 Difference in Communication Time Between Overlap

To get a clearer picture of the time spent communicating between nodes, experiments were run which measured the amount of time required to communicate the data between nodes. These experiments were similar to those in Section 7.6.3, except that the part of the program that calculates the life cells is deactivated. As the cell calculation is deactivated, the runtime consists of the time required to communicate data between nodes.

These experiments were conducted in the same way as those described in Section 7.5.4. The results for these experiments are shown in Tables 7.5, 7.6, 7.7 and in Figure 7.23.

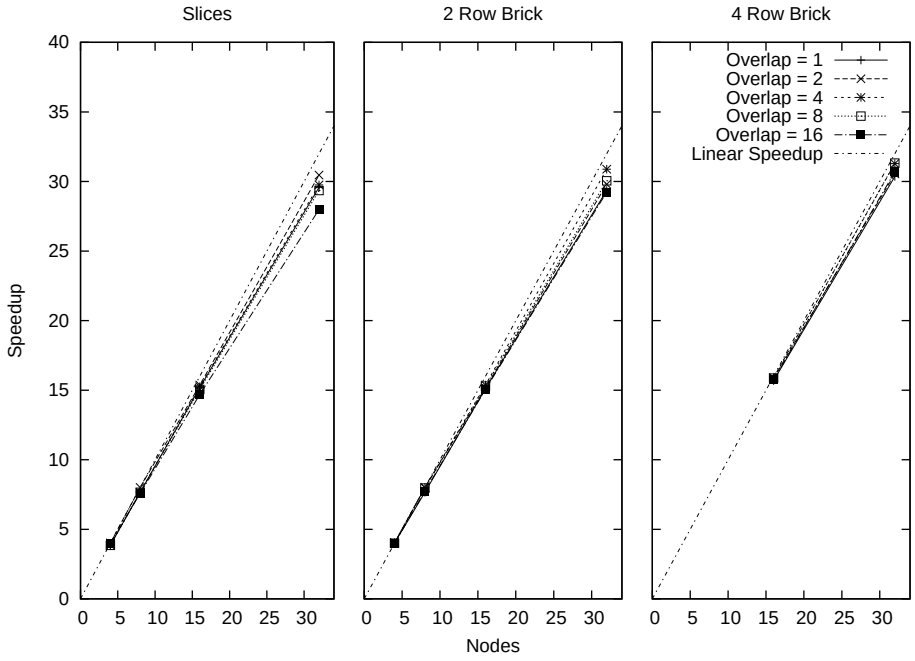


Figure 7.20: Speedup computing 5120 by 5120 for 4096 generations.

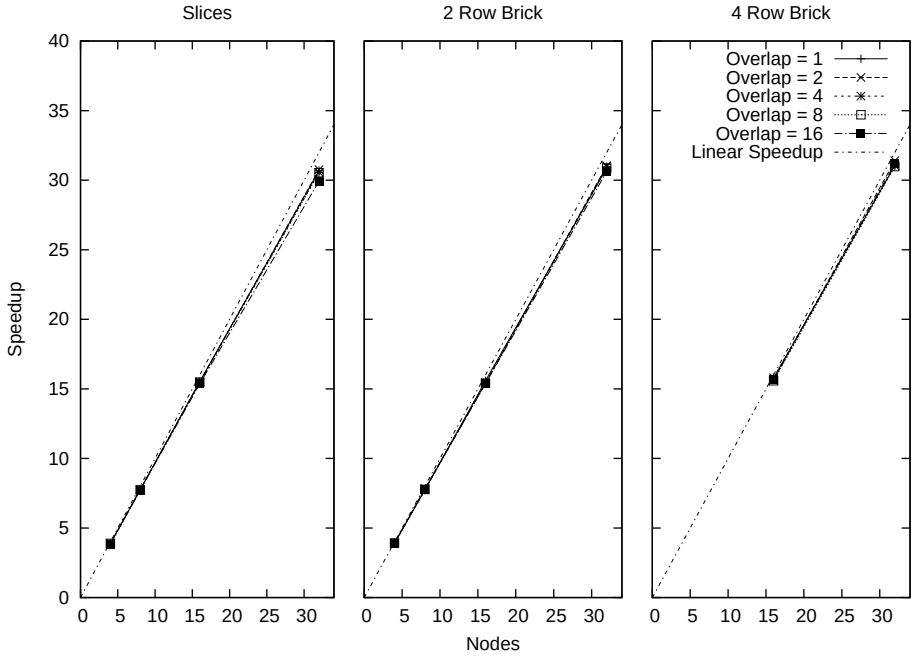


Figure 7.21: Speedup computing 20480 by 20480 for 256 generations.

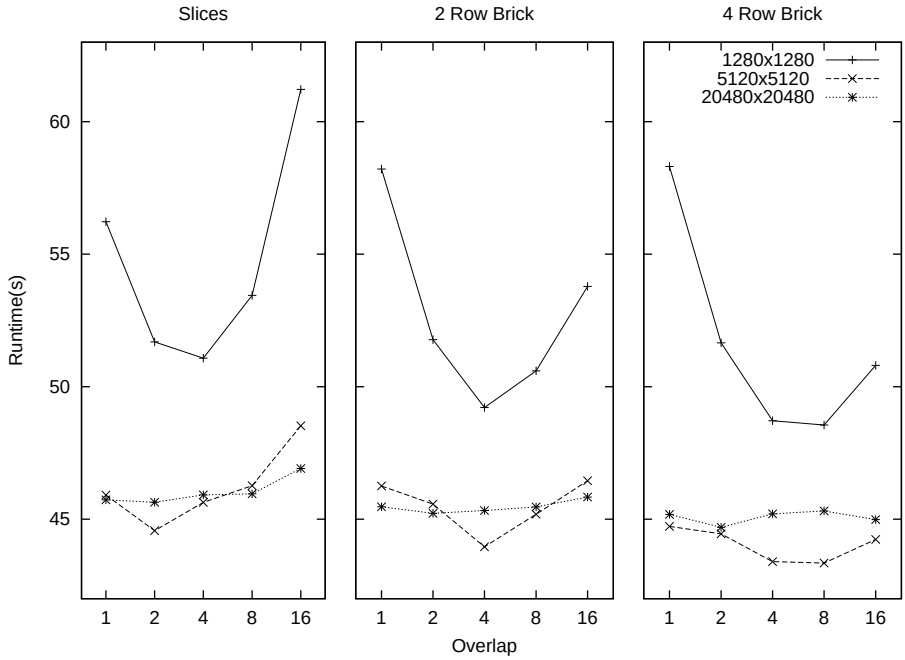


Figure 7.22: Median runtimes using 32 nodes after runs with slow nodes were discarded. Three different board sizes are shown, 1280 by 1280 for 65536 Generations, 5120 by 5120 for 4096 Generations and 20480 by 20480 for 256 Generations.

Board	Nodes	Overlap	Total Time (s)			Time Per Generation (ms)		
			S	B2	B4	S	B2	B4
1280	4	1	8.8	11.8		0.13	0.18	
by		2	5.7	7.1		0.09	0.11	
1280,		4	4.4	4.8		0.07	0.07	
65536		8	3.5	3.4		0.05	0.05	
Generations		16	3.2	2.6		0.05	0.04	
	8	1	9.3	11.4		0.14	0.17	
		2	6.0	6.9		0.09	0.10	
		4	4.6	4.6		0.07	0.07	
		8	3.7	3.4		0.06	0.05	
		16	3.3	2.7		0.05	0.04	
	16	1	9.8	11.4	11.4	0.15	0.17	0.17
		2	6.4	7.0	6.8	0.10	0.11	0.10
		4	5.0	4.7	4.6	0.08	0.07	0.07
		8	4.0	3.5	3.3	0.06	0.05	0.05
		16	3.5	2.8	2.7	0.05	0.04	0.04
	32	1	10.5	11.6	11.6	0.16	0.18	0.18
		2	7.0	7.1	6.9	0.11	0.11	0.11
		4	5.3	4.9	4.7	0.08	0.07	0.07
		8	4.4	3.7	3.3	0.07	0.06	0.05
		16	3.7	3.0	2.8	0.06	0.05	0.04

Table 7.5: Communication only runtimes for the 1280 by 1280 board run for 65536 generations.

Board	Nodes	Overlap	Total Time (s)			Time Per Generation (ms)		
			S	B2	B4	S	B2	B4
5120	4	1	1.7	2.2		0.413	0.539	
by		2	1.2	1.8		0.293	0.435	
5120		4	0.9	1.2		0.225	0.293	
4096		8	0.6	0.7		0.161	0.184	
Generations,		16	0.5	0.5		0.126	0.127	
	8	1	1.8	2.1		0.447	0.524	
		2	1.3	1.7		0.315	0.418	
		4	1.0	1.2		0.238	0.296	
		8	0.7	0.7		0.165	0.185	
		16	0.5	0.5		0.127	0.120	
	16	1	2.0	2.2	2.3	0.493	0.549	0.558
		2	1.4	1.8	1.8	0.350	0.447	0.450
		4	1.0	1.3	1.3	0.258	0.316	0.312
		8	0.7	0.8	0.7	0.178	0.204	0.186
		16	0.5	0.5	0.5	0.130	0.135	0.132
	32	1	2.2	2.3	2.3	0.535	0.567	0.567
		2	1.6	1.9	1.9	0.388	0.462	0.464
		4	1.2	1.4	1.3	0.289	0.334	0.323
		8	0.8	0.9	0.8	0.207	0.211	0.191
		16	0.6	0.6	0.5	0.149	0.151	0.136

Table 7.6: Communication only runtimes for the 5120 by 5120 board run for 4096 generations.

Board	Nodes	Overlap	Total Time (s)			Time Per Generation (ms)		
			S	B2	B4	S	B2	B4
20480	4	1	1.7	2.2		1.440	2.294	
by		2	1.2	1.8		1.171	1.861	
20480,		4	0.9	1.2		1.034	1.470	
256		8	0.6	0.7		0.974	1.208	
Generations		16	0.5	0.5		0.938	1.076	
	8	1	1.8	2.1		1.224	1.631	
		2	1.3	1.7		0.975	1.114	
		4	1.0	1.2		0.848	0.845	
		8	0.7	0.7		0.785	0.708	
		16	0.5	0.5		0.753	0.674	
	16	1	2.0	2.2	2.3	1.161	1.856	2.006
		2	1.4	1.8	1.8	0.894	0.963	0.992
		4	1.0	1.3	1.3	0.755	0.706	0.708
		8	0.7	0.8	0.7	0.672	0.619	0.566
		16	0.5	0.5	0.5	0.648	0.536	0.512
	32	1	2.2	2.3	2.3	1.493	2.016	1.977
		2	1.6	1.9	1.9	0.887	1.286	1.315
		4	1.2	1.4	1.3	0.719	0.673	0.634
		8	0.8	0.9	0.8	0.641	0.531	0.484
		16	0.6	0.6	0.5	0.625	0.495	0.408

Table 7.7: Communication only runtimes the 20480 by 20480 board run for 256 Generations.

Board	Layout	Overlap	Life Cell Calculation Clock Time Difference (s)	Communication Only Runtime Difference (s)	Life Cell Calculation Clock Time Difference + Communication Only Runtime Difference (s)	Actual Runtime Difference (s)
1280 by 1280, 65536 Generations	Slices	2	-1.0	-3.6	-4.5	-4.1
		4	0.2	-5.2	-4.9	-5.2
		8	3.6	-6.1	-2.5	-2.4
		16	11.9	-6.8	5.1	4.9
	2 Row Brick	2	-1.7	-4.4	-6.2	-6.4
		4	-1.6	-6.7	-8.3	-9.1
		8	0.4	-7.9	-7.5	-7.3
		16	5.0	-8.6	-3.6	-4.3
	4 Row Brick	2	-2.0	-4.7	-6.7	-6.7
		4	-2.3	-6.9	-9.1	-9.7
		8	-1.1	-8.3	-9.4	-9.6
		16	1.7	-8.9	-7.2	-7.6
5120 by 5120, 4096 Generations	Slices	2	-0.8	-0.6	-1.4	-1.8
		4	0.5	-1.0	-0.5	-0.5
		8	1.4	-1.3	0.0	0.3
		16	2.9	-1.6	1.3	2.4
	2 Row Brick	2	0.2	-0.4	-0.3	-0.8
		4	-0.5	-1.0	-1.5	-2.2
		8	0.1	-1.5	-1.3	-1.4
		16	1.6	-1.7	-0.1	-0.1
	4 Row Brick	2	0.2	-0.4	-0.2	0.0
		4	0.0	-1.0	-1.0	-1.1
		8	0.3	-1.5	-1.3	-1.2
		16	1.4	-1.8	-0.4	-0.3
20480 by 20480, 256 Generations	Slices	2	0.0	-0.2	-0.2	0.1
		4	0.1	-0.2	-0.1	0.1
		8	0.4	-0.2	0.2	0.2
		16	1.0	-0.2	0.8	1.0
	2 Row Brick	2	0.2	-0.2	0.0	-0.5
		4	1.0	-0.3	0.6	-0.1
		8	1.3	-0.4	1.0	0.0
		16	1.8	-0.4	1.4	0.4
	4 Row Brick	2	-0.4	-0.2	-0.6	-1.1
		4	0.1	-0.3	-0.2	-0.2
		8	0.0	-0.4	-0.4	-0.2
		16	0.3	-0.4	-0.1	-0.2

Table 7.8: Differences in seconds between using different levels of overlap and using no overlap. The sum of the differences in the time spent in the calculation phase of the program and the non compute runtime is similar to the difference in the program runtime.

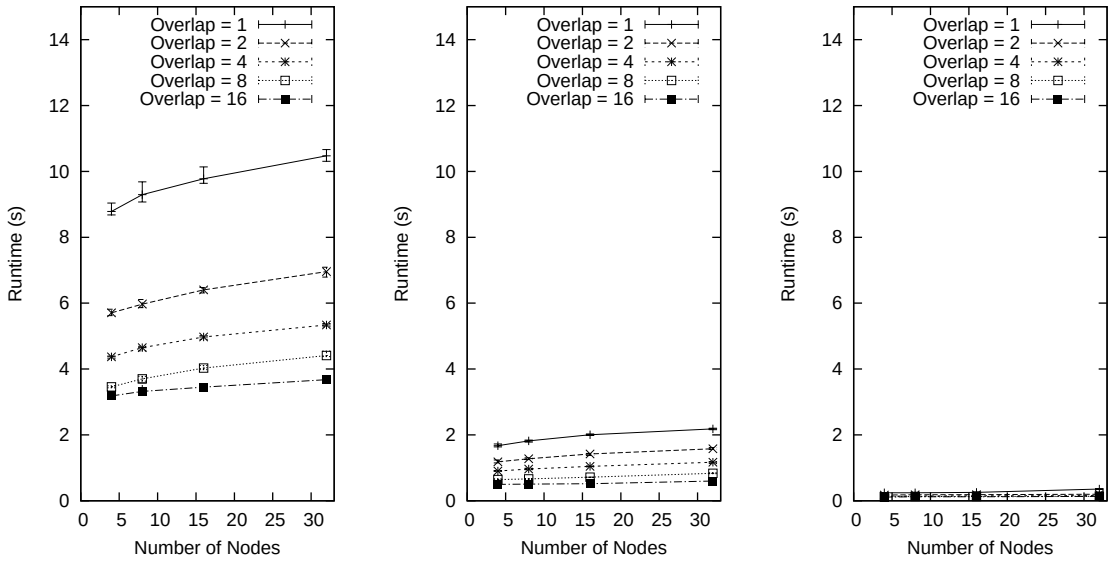


Figure 7.23: Communication only runtimes for 1280 by 1280 for 65536 Generations, 5120 by 5120 for 4096 Generations and 20480 by 20480 for 256 Generations. Using higher levels of overlap decreases non compute runtime.

7.6.5 Conclusions About Haloing

- **The performance of the distributed life program can be increased by using haloing**

  Figure 7.22 shows that it is possible to reduce the runtimes of the 1280 by 1280 and 5120 by 5120 boards by using haloing.

- **Small board sizes have a bigger increase in speedup than larger ones when haloing is used**

  The biggest reductions in runtimes are gained when haloing is used with smaller board sizes. Smaller boards send smaller messages, there is a greater benefit in grouping several generations worth of communication into one message. Also, communication is a larger proportion of overall runtime for smaller board sizes,

  Figure 7.22 shows that the 1280 by 1280 board had the largest reduction in runtime. The 5120 by 5120 board had a smaller but still noticable reduction in runtime. For the 20480 by 20480 board only small reductions in runtime were gained.

- **The effect of haloing on runtime follows a bathtub curve**

  This can be seen in Figure 7.22. There is a certain optimum level of overlap, which is between 2 and 8 depending on the board size and layout used. After this optimum level of overlap the cost of duplicating the calculation of cells on different nodes causes the runtime to rise.

  The effect of haloing on the time spent taken to calculate cells and to communicate data between nodes is shown in Table 7.8. Increasing the amount of overlap decreases the amount of time spent communicating data between nodes.

- **Extra time spent by each node computing life cells is not proportional to the number of extra cell calculations required because of haloing**

  Haloing duplicates the calculation of cells across nodes, so it was expected that this would increase the time spent calculating cells proportionally to the extra amount of cells each node had to calculate. Table 7.3 shows how many more cell calculations each node does at each level of overlap as a percentage, and Table 7.4 shows the average increase in the time taken by each node to calculate cells at each level of overlap.

  Surprisingly, the amount of time spent by nodes calculating life when using haloing is sometimes less than when using no overlap despite the fact that more cells are being calculated. This is shown in Table 7.4 as the negative percentages. This seems because more time is spent calculating life cells between sending and receiving messages, which reduces the scheduling overhead.

- **The lower surface to area ratio of the bricks layout means less cell calculation is required compared to the slices layout**

  Surface to area ratios of different board sizes and layouts are discussed in Section 7.5. When using haloing, a lower edge to area ratio reduces not only the amount of data that has to be transmitted, but also the number of cell computations that are duplicated on different nodes. This can be seen in Table 7.3, especially when using 32 nodes. For example, when using an overlap of 8 on the 1280 by 1280 board using 32 nodes with the slices layout an extra 17.5% cell calculations are done on each node compared to when not using haloing. When using the 4 row brick layout only an extra 6.7% cell calculations are done.

  This difference in the number of duplicated cell calculations can be seen in the difference in runtime between the two layouts. For the example of the 1280 by 1280 board using 32 nodes with an overlap of 8 the median runtime (after runs with slow nodes were removed) was 53.4 seconds when using the slices layout, but was 48.6 seconds when using the 4 row brick layout.

- The difference in runtime when using haloing is approximately equal to the sum of the difference in communication time (as measured in the non compute experiments) and the difference in time taken to calculate the life cells (measured using System.nanoTime inside the life program)

7.7 Conclusions

The Game of Life is an example of a structured grid[1] problem. These problems contain cells in a grid layout where the calculation of each cell requires information from neighbouring nodes. The conclusions given should be applicable to other structured grid problems.

- **High Level of Speedup Obtained**

  A high level of speedup could be obtained for all board sizes. Small boards achieved less speedup than larger boards. But even the smallest board size (1280 by 1280) achieved a speedup of 27.8 when using 32 nodes, which is quite a high level of speedup.

- **Slices Has the Lowest Communication Time Unless Using Very Large Board Sizes or Large Amounts of Overlap**

  Although the bricks layout had a lower edge-to-area ratio, the slices layout had the lowest communication time in most situations. When the data to be sent and received is small, the overhead of sending and receiving messages is high enough that it is better to send data in only two messages

(as the slices layout does) rather than six (as the bricks layout does), even though slices has a worse edge-to-area ratio.

The bricks layout had a lower communication time than the slices layout when the amount of data to be sent and received was large. The amount of data to be sent is large when large board sizes are used, or a large amount of overlap is used (which sends larger messages, but less often). However, as can be seen in Tables 7.5, 7.6 and 7.7, when the 4-row brick layout is better than the slices layout it is only better by the slightest of margins. In the experiments done in this chapter, a maximum of 32 nodes were used. The trends suggest that when using hundreds of nodes on very large board sizes, the bricks layout communication time would be considerably lower than for slices.

- **Haloing Task Edges Can Improve Performance**

  Haloing can improve performance, mostly by reducing the communication time. The communication time is improved because less messages need to be sent. Using haloing also increases the efficiency of the life cell calculation for small boards, as more cell calculations are done between sending and receiving messages, thus reducing the scheduling overhead.

  The cost of haloing node edges is that it duplicates cell calculation. If too many layers of edge are overlapped the cost of doing duplicate cell computations is higher, which outweighs the efficiencies gained.

  For other structured grid problems, how haloing node edges affects performance depends on how computationally intensive it is to calculate a cell. The more computationally intensive it is to calculate a cell, the less worthwhile it would be to use haloing.

- **Using Lower Edge-to-Area Layouts When Using Haloing Reduces Duplicated Computation**

  When overlapping node edges, each cell that adjoins a neighbouring cell must be duplicated on neighbouring cells. Having a lower edge-to-area ratio reduces the number of duplicated cell computations, which in turn reduces the runtime. When overlapping node edges, using a layout with a low edge-to-area ratio can noticeably reduce the runtime.

Chapter 8

Conclusions

The objective of this thesis is to examine how JStar might be compiled into implementations for a distributed computer, and what techniques are useful to obtain efficient implementation of JStar programs.

8.1 Creating and Compiling a Parallel Program

To examine how JStar programs may be compiled to a program suitable for a distributed computer, a possible process for doing this was presented in Chapter 4. This process starts by decomposing the original program into many fine-grain tasks, determining the communication required between each task, determining how these fine-grain tasks are agglomerated into larger coarse-grain tasks and allocating these coarse-grain tasks to compute nodes.

Using this distributed design, distribute statements are written which describe how the program is to be divided up into tasks and run on the distributed computer. The original Starlog program is then combined with the distribute statements to produce a single-program multiple data (SPMD) program.

A SPMD Starlog program could be compiled by a Starlog compiler that supports sending output tuples to other tasks. At the current time however, no Starlog compiler supports sending tuples to different tasks, so this part of the process was not demonstrated in this thesis. This issue is discussed in Section 4.2.2 as well as in the future work part of this Chapter (Section 8.4.1).

The process described in Chapter 4 illustrates how the compilation of a JStar program can work. This process has been demonstrated in this thesis by applying it to the Pascal's Triangle program in Chapter 4, the primes program in Chapter 6 and the Game of Life program in Chapter 7. This has shown that it is feasible to transform Starlog programs to distributed programs. It has also been shown that this transformation can be partially automated. However, writing a practical distributed compiler that can compile all JStar programs and produce distributed programs with performance comparable to

hand-written programs will require a lot of research before the techniques are well developed enough to make it practical.

8.2 Distribute Statements

A contribution of this thesis is a mechanism for allowing the programmer to specify the distributed design through annotations, and a way of combining those annotations with the original program to create a distributed program. These annotations are the distribute statements, which are presented in Chapter 4.

The distribute statements proved to be very flexible, and it was possible to create distribute statements to create all the distributed program designs presented in this thesis. In Chapter 6, it is shown how the primes program can be transformed into two completely different parallel implementations by supplying different distribute statements. In Chapter 7, both haloing and non-haloing versions of two different grid layouts are implemented by using different distribute statements with the original life Starlog program.

One problem with the distribute statements mechanism is that a distribute statement can alter the semantics of the program. It is possible that when a Starlog program is combined with the distribute statements, the tuples produced by the distributed program could be different from that produced by the original Starlog program. This is currently addressed by testing that the transformed program produces the same result as the original program. However, a more general solution would be desirable. Possible remedies for this problem could be investigated in future work. This is discussed in Section 8.4.2.

8.3 Performance Impacts of Distributed Program Design Decisions

During the process of compilation there are many different choices that must be made, which may have an effect on performance. The choices that can be made relate to how the workload of the program is divided among the coarse-grain tasks, how the tasks communicate and how they are allocated to compute nodes. Improving performance of a distributed program is often a matter of increasing the utilisation of the compute nodes or reducing the communication overhead. Decisions which affect performance are generally made during the agglomeration step of the design process.

To explore these various choices two case studies were examined, the Game of Life and a prime counting program. These programs were hand-translated into distributed Java programs. Performance experiments were done to access the performance implications of various design decisions.

8.3.1 How the Workload is Divided Up Among the Coarse-grain Tasks

One decision which was required in each of these case studies was how to divide the workload up among the coarse grain tasks. When doing this, it is important to take the communication costs into account. The amount of communication between tasks should be minimised by placing the fine-grain tasks which communicate with each other together in the same coarse-grain tasks when possible. The workload should be balanced among the compute nodes to reduce the amount of time tasks have to wait for data to be sent by other tasks.

For the prime counting program, two different programs were evaluated:

- A program which divides up the search range. In this program, each task calculates the initial set of primes that is sufficient to filter out the all the non-primes. Each task then counts the number of primes within its sub range. The subtotals of the primes counted are then sent to one task which adds the subtotals together to get the total number of primes.

- A filter pipeline program. In this program, each task is allocated a different set of primes. Each task filters out multiples of the primes allocated to the task. The tasks are organised as a pipeline so that the stream of possible primes flows through the pipeline, with non-primes being identified and removed by filter tasks. After the numbers have been through all the tasks, only primes remain.

The program which divided the search range up into subranges performed very well, being almost embarrassingly parallel. However, the pipeline version of the program performed much less well. The main reason for this was that the cost of the filtering operations was less than the cost of communication. Also, the primes pipeline proved to be difficult to load balance because most of the work was done by the tasks filtering out the low prime numbers, making it difficult to find enough work to keep the other compute nodes busy. Unless the load across the compute nodes is balanced then this will cause a bottleneck which forces compute nodes to wait rather than do productive work. However, pipelines could work quite well in other situations where the compute costs are much higher than the communication costs and the load can easily be balanced across the compute nodes.

For the Game of Life program, the board is divided up into pieces and each piece is handled by a different task. The size and shape of the pieces that the board is broken up into does have an effect on the communication time. Two different piece shapes were experimented with, vertical slices where each piece has two neighbours and a brick layout where each piece has six neighbours. Neither of these layouts was found to be universally superior to the other. Which one had the lowest communication cost depended on the interactions with other decisions, such as the size of each piece and the amount of haloing being used.

8.3.2 Haloing

One technique that significantly reduced communication time was haloing, or duplicating computation on multiple nodes. Haloing lowers the cost of communication by reducing the number of messages required. Haloing considerably reduced communication costs in the Game of Life case study. Haloing was found to yield the greatest improvement when tasks sent large numbers of small messages. For tasks that send a small number of large messages then the improvement is likely to be minimal or to reduce performance as the cost of duplicating computation exceeds any savings gained from reducing the number of messages.

8.3.3 Communication

It was found that small message sizes (less than 800 bytes or so) had relativity low throughput. If a large quantity of data is being sent using small messages, then communication overhead can be reduced by packing the small messages into larger messages.

Using the eager mode communication protocol rather than the synchronous mode protocol was found to significantly increase communication throughput, especially with small message sizes.

8.3.4 Allocation of Coarse Grain Tasks to Compute Nodes

In this thesis, each compute node has only one task allocated to it. The reason for doing this was to create an environment in which each compute node was a single processor. However, as compute nodes these days will certainly have many cores in one processor, allocating more than one task to each compute node will allow the program to take advantage of a compute node with multiple cores. Another approach that could be investigated in future work is to have multi-level parallelism, where inter-node communication could use message passing, but tasks on the same node could use shared-memory. This is discussed in Section 8.4.4.

In some situations, the load is not evenly balanced between tasks. This was the case for the Pascal triangle program presented in Chapter 4. In such cases allocating the tasks so that workload is balanced across compute nodes may increase performance.

8.4 Future Work

8.4.1 Creating a Distributed JStar Compiler

In the future more research will be required to develop a distributed JStar compiler. The processes described in Chapter 4 provides a starting point and would require many refinements.

The most difficult part of the compilation process is likely to be determining how the program can be run on a distributed computer, that is creating a distributed design from the JStar program. The distributed design can be specified manually by adding distribute statements, but ideally these distribute statements should be generated automatically. It must be determined how to split the workload up between tasks, and how the tasks communicate with each other. As completely automating this may be somewhat difficult, it is likely that any distributed JStar compiler would rely on some guidance from the programmer. Allowing the programmer to provide guidance to the compiler does give the programmer more flexibility and control.

In addition to programmer supplied information, a distributed JStar compiler should be able to use other sources of information to make decisions about how the program should be compiled. One source of information which a compiler could use is data from benchmarks. The effect on runtime performance of different alternatives could be measured and this data would help the compiler produce a better performing program.

By using the distributed design, the compiler would be able to convert the original JStar program into a explicitly distributed Starlog program, as shown in Chapter 4 . Once the original program has been converted into a single-program multiple data (SPMD) Starlog program, this could be compiled to executable code using existing Starlog compilation techniques[14]. Support for sending tuples between tasks would need to be added to existing Starlog compilers.

8.4.2 Verification of Distribute Statements

In Section 8.2 it was noted that distribute statements could change the resulting tuples produced by the program. Various ways of dealing with this problem are discussed in Section 4.1.5. These included:

- Inventing a new form of restricted distribute statements that is guaranteed to not change the tuples that the program produces, while still having enough flexibility to implement a wide variety of distributed programs.

- Performing a global inductive proof over the stratification order of the program to prove that it produces the same tuples as the original program. However, it may require a large amount of time and effort to produce the proof.

- Developing modular distributed design patterns that have already been verified, which could be applied to a range of problems. An example might be a pipeline filter, or dividing a grid among tasks. These design patterns could be implemented as high-level functions or macros that generate the necessary distribute statements.

- Testing the distributed program by using the original program as an test oracle to check that it produces the same results. A tool could be developed to automatically test the distributed program against the original Starlog program.

An area of future work is to investigate these options and others to see which is the most suitable way to ensure the correctness of distribute statements.

8.4.3 Researching Additional Case Studies

This thesis studied the Game of Life and the primes counting Starlog programs. The Game of Life program an example of a structured grid program and the primes program is a filtering and counting program. Other case studies will also need to be investigated to find additional techniques which will be required for obtaining efficient implementations of those types of programs.

8.4.4 Multi-Level Parallelism

A modern distributed computer offers different types of parallel hardware and different forms of parallelism which can be exploited to gain higher performance:

- **Inter-node Parallelism**: dividing the workload among different distributed memory compute nodes.
- **Intra-node Parallelism**: dividing the workload allocated to that node among the shared-memory CPU cores.
- **Vector Parallelism**: utilising vector processing hardware (such as GPUs) available on the compute node.

To write a compiler which was able to take advantage of multi-level parallelism, there will need to be more research to determine how JStar programs can be compiled for each of these parallel architectures, and the best way to partition the workload between CPU cores and GPUs located on different nodes.

Appendix A

Primes Case Study Results

		Runtime (No Optimisations)					Runtime (Skip Multiples of 2)				
Block Size	Run	Node 1	Node 2	Node 3	Node 4	Node 5	Node 1	Node 2	Node 3	Node 4	Node 5
3125	1	241.4	241.4	241.5	241.7	241.6	241.7	241.4	241.6	241.6	241.7
	2	241.8	241.4	241.5	241.5	241.4	241.6	241.5	241.7	241.4	241.3
	3	241.4	241.4	241.7	241.7	241.4	241.6	241.7	241.6	241.4	241.6
	4	241.2	241.7	241.7	241.6	241.6	241.6	241.4	241.4	241.4	241.6
	5	241.6	241.4	241.8	241.5	241.5	241.7	241.7	241.4	241.7	241.7
6250	1	165.3	165.0	164.5	164.9	164.6	164.9	165.0	164.6	165.3	164.6
	2	164.9	164.7	165.2	165.3	165.2	165.0	164.6	165.0	165.1	164.9
	3	164.6	164.8	164.7	164.7	165.0	164.6	164.8	164.7	164.6	164.7
	4	164.9	164.6	164.6	165.3	165.0	164.9	164.5	164.9	165.3	164.7
	5	164.9	164.6	164.6	164.8	164.6	165.3	164.7	165.3	164.9	164.8
12500	1	122.0	135.7	121.4	121.4	121.3	121.4	121.4	121.4	121.4	121.4
	2	121.4	121.4	121.4	121.4	121.5	122.1	121.4	121.3	121.4	121.4
	3	121.4	121.4	121.4	121.4	121.4	121.4	121.4	121.2	121.4	121.4
	4	122.0	121.4	121.4	121.2	121.5	121.4	121.4	121.4	121.4	121.3
	5	121.4	121.4	121.4	121.5	121.4	121.5	121.7	121.4	121.4	121.4
25000	1	95.7	95.7	95.7	95.6	95.6	95.6	95.7	95.5	96.0	95.7
	2	96.0	95.7	95.7	95.6	95.7	95.7	95.5	95.7	95.7	95.7
	3	95.7	95.6	95.7	95.7	95.6	95.7	96.0	95.7	95.7	95.9
	4	95.9	95.7	95.7	95.7	95.7	95.6	95.7	95.7	95.7	95.5
	5	95.6	95.7	95.6	95.6	95.7	95.7	95.7	95.7	95.5	95.7
50000	1	84.0	84.0	84.0	84.0	84.0	84.0	84.0	84.0	84.0	83.9
	2	84.0	84.0	84.0	84.5	83.9	84.0	84.0	83.9	84.0	84.0
	3	84.0	84.0	84.0	84.0	84.0	84.0	84.0	83.9	84.6	84.0
	4	84.0	83.9	84.5	83.9	84.0	84.0	83.9	84.0	84.0	84.0
	5	83.9	84.0	84.6	84.6	83.9	83.9	84.0	84.5	84.0	83.9
100000	1	80.8	80.8	80.8	80.8	80.8	80.8	80.7	80.7	80.7	80.6
	2	80.8	88.1	80.7	80.8	80.8	80.8	81.0	80.7	80.6	80.6
	3	80.8	80.8	80.8	80.8	80.8	80.8	80.8	80.7	81.0	80.7
	4	80.8	80.8	80.7	81.1	80.7	80.8	80.7	80.7	80.7	80.7
	5	80.8	80.8	80.8	81.1	80.8	80.8	81.0	80.7	80.7	80.6
200000	1	80.1	80.0	79.8	79.9	80.0	80.2	79.9	79.7	79.7	79.7
	2	79.8	79.8	80.0	80.0	79.5	80.2	79.7	79.8	79.6	79.7
	3	79.8	79.5	80.0	80.0	80.0	79.8	79.9	79.7	79.7	79.7
	4	79.8	79.8	79.8	80.0	80.0	79.8	79.8	79.7	79.7	79.8
	5	80.2	80.0	79.8	79.8	80.0	79.8	79.8	79.7	79.7	79.7
400000	1	79.2	79.2	80.6	79.2	79.1	79.2	79.2	79.1	80.6	79.2
	2	79.2	79.2	79.1	79.1	79.1	79.2	79.2	79.2	79.1	80.6
	3	80.6	79.1	79.2	79.1	79.1	79.2	79.2	79.2	79.2	79.1
	4	81.1	79.1	79.2	80.6	79.1	79.2	79.2	79.1	79.1	79.1
	5	79.2	79.2	79.2	79.2	79.2	79.1	79.2	80.6	79.2	79.2
800000	1	78.9	78.9	78.9	78.9	79.0	79.0	78.9	78.9	78.9	79.0
	2	78.9	78.9	78.9	78.9	78.9	78.9	78.9	78.9	78.9	79.0
	3	78.9	79.0	78.9	78.9	79.0	79.0	78.9	78.9	78.9	78.9
	4	78.9	78.9	78.9	78.9	78.9	78.9	78.9	78.9	78.9	78.9
	5	78.9	78.9	78.9	78.9	78.9	78.9	79.0	79.0	78.9	78.9

Table A.1: Runtimes in seconds for the single node version using the mark and check filter, with and without the skip multiples of 2 optimisation.

Optimizations	Runtimes				
No Optimisations	7124.44	7130.70	7131.68	7136.49	7142.12
	7151.70	7152.21	7153.23	7159.94	7167.10
	7170.84	7172.00	7172.61	7172.88	7175.88
	7176.07	7176.60	7177.46	7178.09	7178.20
	7178.32	7178.62	7178.92	7180.38	7181.13
	7181.21	7181.54	7181.92	7182.77	7182.94
	7183.83	7184.27	7184.48	7184.55	7184.74
	7186.48	7187.62	7187.88	7191.31	7191.61
	7193.77	7194.02	7195.02	7196.67	7197.19
	7200.74	7201.14	7204.77	7215.26	7417.44
Skip Multiples of 2	5606.19	5612.20	5675.60	5678.79	5679.17
	5679.26	5680.09	5680.40	5681.41	5681.59
	5682.46	5682.86	5683.26	5683.52	5684.15
	5685.83	5686.06	5687.46	5688.51	5693.18
	5693.70	5694.27	5695.99	5697.97	5710.70
Storing $Prime \cdot Multiplier$	6083.11	6089.35	6093.46	6095.02	6095.45
	6097.16	6098.01	6099.22	6099.67	6100.98
	6102.98	6102.99	6103.07	6103.46	6103.52
	6105.63	6106.18	6106.33	6107.67	6108.12
	6108.47	6108.84	6111.05	6124.26	6124.58
Partitioning Priority Queue	3599.70	3614.60	3628.04	3669.44	3675.05
	3677.29	3678.02	3678.34	3679.29	3680.08
	3680.56	3687.71	3706.28	3707.42	3708.63
	3713.98	3715.36	3716.31	3723.89	3725.03
	3725.65	3725.90	3745.14	3753.59	3754.96
All of Above Optimizations	2835.53	2844.75	2844.80	2854.68	2855.81
	2856.56	2860.70	2862.78	2862.92	2862.93
	2863.45	2864.21	2864.78	2866.44	2867.39
	2872.91	2873.17	2874.45	2874.83	2875.24
	2875.27	2875.77	2877.43	2877.53	2881.88

Table A.2: Runtimes in seconds for the single node version using the priority queue filter.

Run	Runtime				
	2 Nodes	4 Nodes	8 Nodes	16 Nodes	32 Nodes
1	49.82	25.17	12.75	6.63	3.24
2	49.79	25.17	12.74	6.62	3.33
3	49.93	25.16	12.73	6.62	3.32
4	49.93	25.15	12.74	6.62	3.33
5	49.96	25.15	12.74	6.62	3.33
6	49.94	25.15	12.70	6.62	3.33
7	49.93	25.16	12.74	6.43	3.33
8	49.93	25.16	12.74	6.62	3.32
9	49.97	25.15	12.74	6.62	3.25
10	49.93	25.24	13.05	6.43	3.25
11	49.93	25.16	12.75	6.62	3.24
12	49.93	25.08	12.74	6.59	3.33
13	49.95	25.17	12.74	6.52	3.33
14	49.93	25.16	12.74	6.62	3.33
15	49.94	25.16	12.74	6.47	3.33

Table A.3: Runtimes in seconds for the divide search range prime counting program.

Nodes	Run	Exponential	Triangle	Flat
2	1	104.93	109.15	109.85
	2	105.10	109.26	109.92
	3	104.97	111.44	109.39
	4	105.01	110.32	110.16
	5	104.98	110.40	109.04
	6	105.00	110.00	110.45
	7	104.96	110.18	109.91
	8	105.08	109.42	109.97
	9	104.90	109.22	107.98
	10	104.86	109.98	109.31
4	1	104.85	104.30	163.83
	2	104.87	104.19	164.85
	3	104.93	104.30	164.21
	4	104.80	104.31	163.69
	5	104.84	104.17	165.12
	6	104.83	104.18	164.88
	7	104.91	104.34	164.74
	8	104.79	104.27	165.29
	9	104.88	104.18	164.18
	10	104.82	104.21	164.41
8	1	104.79	106.11	246.87
	2	104.77	105.96	247.09
	3	104.64	106.21	248.28
	4	104.65	106.10	247.42
	5	104.70	106.03	247.14
	6	104.74	106.00	245.91
	7	104.66	106.11	245.67
	8	104.65	106.18	245.74
	9	104.74	106.08	245.89
	10	104.66	106.00	245.64
16	1	104.99	127.87	294.16
	2	105.04	127.98	294.06
	3	104.96	127.98	295.63
	4	104.99	128.35	294.22
	5	105.01	127.96	294.11
	6	104.98	128.06	294.25
	7	105.01	128.08	294.08
	8	105.08	128.27	294.30
	9	104.99	128.12	294.57
	10	105.00	128.29	294.01

Table A.4: Runtimes for the pipeline using the mark-and-check filter.

Nodes	Run	Exponential	Triangle	Flat
2	1	2527.03	4987.98	5657.50
	2	2527.19	4985.99	5674.31
	3	2525.62	5001.21	5648.37
	4	2525.16	5022.19	5654.91
4	1	1076.14	3932.35	4925.88
	2	1071.45	4029.52	4893.01
	3	1080.38	3932.12	4917.34
	4	1074.97	3945.89	4943.80
8	1	476.35	2851.72	4427.79
	2	474.14	2848.15	4453.43
	3	475.36	2852.92	4453.49
	4	477.91	2855.82	4450.48
16	1	340.57	1832.93	4103.25
	2	340.94	1868.07	4124.44
	3	338.26	1829.26	4069.64
	4	340.89	1888.03	4059.84

Table A.5: Runtimes for the pipeline using the priority queue filter.

Appendix B

Life Case Study Results

Board	Generations	Node	Run	Runtime
1280 by 1280	65536	1	1	1364.5
			2	1348.0
			3	1349.0
		2	1	1348.6
			2	1347.1
			3	1348.8
		3	1	1347.4
			2	1348.3
			3	1348.2
2560 by 2560	16384	1	1	1354.3
			2	1351.0
			3	1355.6
		2	1	1350.9
			2	1354.3
			3	1350.7
		3	1	1354.3
			2	1350.9
			3	1354.3
5120 by 5120	4096	1	1	1358.9
			2	1357.9
			3	1357.7
		2	1	1352.6
			2	1352.3
			3	1352.3
		3	1	1352.3
			2	1358.8
			3	1358.8
10240 by 10240	1024	1	1	1363.4
			2	1363.4
			3	1372.0
		2	1	1363.3
			2	1371.9
			3	1362.7
		3	1	1363.4
			2	1371.9
			3	1372.0
20480 by 20480	256	1	1	1403.3
			2	1406.3
			3	1406.9
		2	1	1405.2
			2	1397.8
			3	1406.2
		3	1	1403.6
			2	1395.5
			3	1399.9

Table B.1: Single Node Run Runtimes

Board	Generations	Median Runtime
1280 by 1280	65536	1348.3
2560 by 2560	16384	1354.3
5120 by 5120	4096	1357.7
10240 by 10240	1024	1363.4
20480 by 20480	256	1403.6

Table B.2: Single Node Run Runtimes

Run	4 Nodes		8 Nodes		16 Nodes			32 Nodes		
	Slices	Brick 2-Row	Slices	Brick 2-Row	Slices	Brick 2-Row	Brick 4-Row	Slices	Brick 2-Row	Brick 4-Row
1	**347.7**	**349.8**	**180.3**	**183.2**	**97.0**	**101.3**	**99.9**	*69.1*	**59.1**	*69.5*
2	**347.5**	**350.7**	**180.9**	**182.9**	**99.3**	**99.8**	**99.7**	**56.2**	*71.7*	**58.8**
3	**345.5**	**349.8**	*278.7*	**185.9**	**97.3**	**99.4**	**99.7**	**56.2**	*61.3*	*64.9*
4	**347.2**	**350.9**	**180.6**	**183.3**	**97.6**	**99.6**	**99.7**	*67.9*	*62.6*	*62.1*
5	*495.3*	**350.0**	**182.5**	**182.9**	*154.7*	**99.7**	**101.6**	**56.4**	**58.0**	**58.3**
6	**346.2**	**349.7**	*213.8*	**183.2**	**97.2**	**99.3**	**100.0**	**56.3**	*63.3*	**58.2**
7	**345.6**	*537.8*	**182.4**	**183.3**	**97.1**	**99.5**	*111.4*	**56.2**	*63.8*	*61.3*
8	*516.5*	**350.3**	**180.1**	**183.0**	**97.1**	**99.4**	**99.5**	**57.0**	*62.9*	**58.8**
9	**347.8**	**351.4**	*274.4*	**185.2**	**97.2**	*111.6*	**100.0**	**56.2**	*74.8*	**58.0**
10	**347.5**	**350.5**	**181.5**	**183.2**	**97.4**	**101.3**	**99.9**	*68.0*	*67.8*	**58.3**
11	**345.8**	**350.0**	**181.3**	**183.1**	*134.7*	**99.9**	**99.8**	**56.1**	**58.1**	*60.6*
12	**348.4**	**351.4**	**190.8**	**183.5**	**97.1**	**99.7**	**99.8**	**56.1**	**58.3**	**59.1**
Median	347.4	350.3	181.3	183.2	97.2	99.7	99.8	56.2	58.2	58.3
Speedup	3.9	3.8	7.4	7.4	13.9	13.5	13.5	24.0	23.2	23.1

Table B.3: 1280 by 1280, 65536 Generations, 1 overlap.

Run	4 Nodes		8 Nodes		16 Nodes			32 Nodes		
	Slices	Brick 2-Row	Slices	Brick 2-Row	Slices	Brick 2-Row	Brick 4-Row	Slices	Brick 2-Row	Brick 4-Row
1	**353.4**	**340.8**	**178.2**	**172.8**	**91.8**	**89.1**	**88.6**	**48.8**	*57.5*	**47.2**
2	**349.7**	**340.9**	**178.4**	**172.6**	**91.8**	**89.0**	**89.7**	*49.9*	**47.3**	*68.3*
3	**352.3**	**341.0**	*254.6*	**173.5**	**91.9**	**91.2**	**89.0**	**49.3**	**47.9**	**50.2**
4	**353.5**	**341.1**	**192.0**	**172.5**	**91.8**	**88.9**	**88.9**	**48.7**	**47.3**	**49.6**
5	**349.8**	**340.4**	**192.5**	**172.5**	**92.3**	**89.1**	**88.9**	**49.6**	**47.4**	*56.2*
6	**353.4**	*362.0*	*203.2*	**172.3**	**91.7**	*97.5*	*97.7*	**49.3**	**47.1**	*70.4*
7	**349.9**	**340.8**	**178.8**	*224.1*	*102.7*	*139.7*	**88.9**	**49.2**	*58.4*	*66.3*
8	**349.4**	**339.6**	**179.5**	**172.6**	**92.2**	*137.8*	**88.8**	*55.2*	**47.5**	*67.1*
9	**351.6**	**340.7**	**180.1**	**172.3**	**91.5**	**89.1**	*140.3*	**48.9**	**47.3**	*66.0*
10	**351.9**	**340.5**	**178.2**	*271.6*	**92.1**	**89.1**	**88.8**	*48.7*	*50.8*	*51.8*
11	**357.5**	**340.8**	**176.3**	**172.2**	**97.0**	**89.2**	**88.9**	**49.0**	**48.8**	*77.1*
12	*427.8*	**341.1**	*201.8*	*255.1*	**91.7**	**91.9**	**88.8**	**49.9**	*58.7*	**47.8**
Median	351.9	340.8	178.8	172.5	91.8	89.1	88.9	49.2	47.4	48.7
Speedup	3.8	4.0	7.6	7.9	14.8	15.2	15.2	27.5	28.6	27.8

Table B.4: 2560 by 2560, 16384 Generations, 1 overlap.

Run	4 Nodes		8 Nodes		16 Nodes			32 Nodes		
	Slices	Brick 2-Row	Slices	Brick 2-Row	Slices	Brick 2-Row	Brick 4-Row	Slices	Brick 2-Row	Brick 4-Row
1	**352.1**	**337.2**	**178.1**	**181.5**	**89.3**	**89.6**	**87.7**	**48.1**	*52.2*	**44.7**
2	**350.9**	**338.3**	**176.3**	**177.2**	**94.2**	**90.1**	**86.6**	**45.7**	*52.7*	**45.2**
3	**351.1**	**350.7**	**176.9**	**176.8**	**89.2**	**89.9**	**86.5**	**45.9**	**47.4**	*53.5*
4	**350.9**	**338.7**	**175.5**	**177.0**	*137.6*	*106.7*	**86.5**	*70.8*	**47.0**	**44.7**
5	**336.1**	**338.5**	**177.7**	**177.3**	**89.3**	**89.8**	**86.6**	*55.3*	**47.8**	**44.7**
6	**352.4**	**338.5**	**177.3**	**177.4**	**92.4**	**96.3**	**86.4**	**45.8**	**46.0**	**45.1**
7	**335.7**	**338.4**	**177.4**	**177.3**	**94.4**	**89.9**	**88.0**	**45.9**	**46.3**	**44.6**
8	**350.9**	**337.1**	*192.3*	**176.8**	**89.4**	**89.6**	**86.8**	*70.0*	**46.1**	*47.2*
9	**336.6**	**338.3**	**177.4**	**176.9**	**89.1**	**90.8**	**86.5**	**46.1**	*46.1*	**44.5**
10	**352.1**	**337.6**	**177.4**	**178.7**	**90.1**	**90.0**	**89.3**	**45.4**	*51.0*	**44.8**
11	**351.2**	**350.7**	**177.0**	**177.0**	**89.4**	**90.0**	*136.7*	**46.3**	**45.9**	*55.2*
12	**351.3**	**338.7**	**177.3**	**177.1**	**89.5**	**89.5**	**86.6**	**46.3**	*50.1*	**44.7**
Median	351.0	338.4	177.3	177.2	89.4	89.9	86.6	45.9	46.3	44.7
Speedup	3.9	4.0	7.7	7.7	15.2	15.1	15.7	29.6	29.4	30.4

Table B.5: 5120 by 5120, 4096 Generations, 1 overlap.

Run	4 Nodes		8 Nodes		16 Nodes			32 Nodes		
	Slices	Brick 2-Row	Slices	Brick 2-Row	Slices	Brick 2-Row	Brick 4-Row	Slices	Brick 2-Row	Brick 4-Row
1	**354.2**	**336.3**	**177.4**	**170.1**	**92.5**	**90.1**	**85.4**	**45.3**	**45.2**	**45.2**
2	**353.7**	**336.1**	**179.0**	**179.8**	**90.3**	*90.1*	**88.4**	**45.3**	**45.5**	**45.2**
3	**351.9**	**352.5**	*264.3*	**170.1**	**89.9**	**88.2**	**86.1**	**45.2**	**45.6**	**45.1**
4	**353.6**	**335.6**	**192.9**	**170.2**	**89.7**	**88.5**	**87.5**	**45.3**	**45.2**	**45.2**
5	**353.7**	**335.9**	**178.2**	**177.2**	**89.4**	**85.1**	**85.5**	**45.3**	**45.5**	*54.2*
6	**353.7**	**336.5**	**178.4**	**170.6**	**89.7**	**85.1**	**86.9**	*49.5*	**45.3**	**45.0**
7	**355.3**	**352.5**	*210.4*	**170.0**	*91.4*	**85.1**	**87.1**	**45.6**	**45.3**	**45.2**
8	*453.8*	**352.3**	**177.8**	*189.2*	**89.1**	**85.5**	**85.8**	*50.1*	**45.2**	**45.1**
9	**353.8**	**352.5**	**178.7**	**171.2**	**90.0**	**85.1**	*91.6*	**45.8**	**45.4**	**45.2**
10	**371.7**	**351.0**	**178.1**	**179.5**	**89.5**	**88.6**	**85.4**	*44.8*	**45.4**	**45.1**
11	**352.1**	**336.3**	**177.3**	**179.9**	*90.1*	**88.3**	**85.5**	**44.7**	*74.1*	**45.0**
12	**353.7**	**352.3**	*194.3*	*262.8*	*99.5*	**85.5**	**85.3**	**45.3**	**45.2**	**45.1**
Median	353.7	343.7	178.2	170.9	89.7	85.5	85.8	45.3	45.3	45.1
Speedup	3.9	4.0	7.7	8.0	15.2	15.9	15.9	30.1	30.1	30.2

Table B.6: 10240 by 10240, 1024 Generations, 1 overlap.

Run	4 Nodes		8 Nodes		16 Nodes			32 Nodes		
	Slices	Brick 2-Row	Slices	Brick 2-Row	Slices	Brick 2-Row	Brick 4-Row	Slices	Brick 2-Row	Brick 4-Row
1	**360.6**	**357.7**	**181.9**	**182.2**	*95.8*	*128.3*	**85.8**	**45.6**	**45.1**	*51.9*
2	**365.1**	**357.6**	**179.6**	**182.1**	**90.4**	**93.4**	**88.6**	*49.1*	*64.0*	*60.6*
3	**360.4**	**357.8**	**182.7**	**180.9**	**90.3**	**89.9**	*91.3*	**45.6**	**45.6**	**45.3**
4	**364.3**	**358.2**	**182.5**	*296.8*	**90.6**	**92.2**	**89.0**	**45.8**	**45.3**	**46.9**
5	**361.2**	**355.2**	**180.9**	**179.1**	**90.4**	**89.7**	**88.6**	*49.8*	*68.7*	*63.6*
6	**361.9**	**357.5**	**180.8**	**181.9**	**90.4**	**92.3**	**89.0**	**46.0**	**45.3**	**45.1**
7	**364.4**	**357.8**	**180.9**	**179.9**	**90.4**	*139.7*	**89.0**	**45.7**	**46.1**	**45.2**
8	**362.1**	**370.7**	**182.2**	**187.4**	**90.5**	**89.8**	**89.0**	**47.6**	*45.3*	*71.2*
9	**365.9**	**357.6**	**181.1**	*180.4*	**89.8**	**90.2**	**89.8**	**45.6**	**45.5**	**45.7**
10	**362.5**	**358.0**	*215.9*	**182.1**	**90.0**	**89.5**	**89.1**	*49.0*	**47.1**	*47.2*
11	**364.0**	**357.4**	**181.1**	**180.4**	**90.4**	**89.6**	**89.1**	*72.3*	*73.2*	**45.1**
12	**362.8**	**355.1**	**180.4**	**180.0**	**91.3**	**90.2**	**89.0**	**45.9**	*49.2*	**45.1**
Median	362.6	357.6	181.1	181.4	90.4	90.1	89.0	45.7	45.5	45.2
Speedup	3.9	3.9	7.8	7.7	15.5	15.6	15.8	30.7	30.9	31.1

Table B.7: 20480 by 20480, 256 Generations, 1 overlap.

Run	4 Nodes		8 Nodes		16 Nodes			32 Nodes		
	Slices	Brick 2-Row	Slices	Brick 2-Row	Slices	Brick 2-Row	Brick 4-Row	Slices	Brick 2-Row	Brick 4-Row
1	**343.1**	**345.3**	**176.0**	**176.7**	**94.0**	**93.3**	**92.9**	**51.5**	**51.8**	**52.7**
2	**343.2**	**365.6**	**176.1**	**177.4**	**92.8**	**93.4**	**97.9**	**51.6**	**51.7**	**51.2**
3	**342.6**	**344.3**	**176.0**	**176.7**	**92.8**	**93.6**	**93.1**	*61.3*	**51.7**	**51.9**
4	**343.3**	**344.3**	*208.4*	**177.0**	*128.1*	**93.7**	**93.0**	**52.2**	**51.7**	**51.4**
5	**342.4**	**347.0**	**176.7**	**181.8**	**92.9**	**95.8**	**93.3**	**53.3**	*72.6*	**51.4**
6	**343.7**	**345.0**	**176.1**	**177.3**	**92.6**	*136.7*	**93.1**	**51.8**	**51.9**	*58.6*
7	**342.6**	**344.6**	**178.4**	**177.5**	**94.1**	**93.5**	**93.1**	**51.6**	**53.6**	*54.6*
8	**343.2**	**344.2**	**175.9**	**176.9**	**92.9**	**93.1**	**93.1**	**53.5**	**51.7**	**52.3**
9	**343.5**	**344.1**	**175.8**	**177.3**	**96.9**	**94.3**	**93.1**	*73.3*	**51.8**	*61.3*
10	**343.5**	**344.3**	**176.8**	**176.7**	**92.9**	**93.5**	**93.2**	*58.0*	*63.3*	*66.2*
11	**343.4**	**344.2**	**175.5**	**176.6**	*100.9*	*114.7*	**93.1**	**51.6**	**52.1**	*57.1*
12	**343.0**	**344.4**	**176.2**	**178.6**	**94.3**	**93.5**	**92.8**	*63.6*	*74.0*	*69.6*
Median	343.2	344.4	176.1	177.2	92.9	93.5	93.1	51.7	51.8	51.7
Speedup	3.9	3.9	7.7	7.6	14.5	14.4	14.5	26.1	26.0	26.1

Table B.8: 1280 by 1280, 65536 Generations, 2 overlap.

Run	4 Nodes		8 Nodes		16 Nodes			32 Nodes		
	Slices	Brick 2-Row	Slices	Brick 2-Row	Slices	Brick 2-Row	Brick 4-Row	Slices	Brick 2-Row	Brick 4-Row
1	**360.9**	**336.2**	**177.4**	**171.2**	*138.2*	**87.7**	**87.2**	**47.6**	**45.8**	*68.1*
2	**338.0**	**336.4**	**175.8**	**171.3**	**92.5**	**87.6**	**87.6**	*70.2*	**45.8**	**45.9**
3	**352.6**	**336.2**	**177.8**	**171.8**	**92.0**	*101.7*	**87.6**	**48.2**	*69.7*	**45.7**
4	**339.0**	**338.4**	**177.0**	**171.2**	**92.0**	**87.4**	**88.1**	**47.9**	**46.2**	**45.7**
5	**352.6**	**338.3**	**178.6**	**171.6**	*130.1*	**87.6**	**87.4**	*65.6*	*63.5*	**45.6**
6	**350.5**	**338.3**	**177.5**	*253.8*	*139.6*	*133.8*	*99.0*	**47.9**	**46.4**	*63.5*
7	**352.5**	**338.3**	**177.5**	**177.1**	**91.5**	**87.5**	**91.6**	*113.0*	*70.3*	*73.4*
8	**350.0**	**336.0**	**177.1**	**171.4**	**90.6**	**87.4**	**87.4**	**47.6**	*72.3*	**45.6**
9	**353.8**	**336.2**	**177.2**	**171.3**	**94.0**	**87.5**	**87.3**	**47.8**	*62.0*	*63.8*
10	**353.0**	**336.1**	**177.2**	**171.3**	**91.2**	**87.5**	**87.2**	**47.6**	*74.8*	**46.0**
11	*517.8*	**336.2**	**177.4**	**171.1**	**91.7**	**87.5**	**87.3**	**47.8**	**45.8**	*70.2*
12	**350.2**	**336.4**	**176.9**	**179.3**	**91.6**	**87.5**	**87.3**	**47.6**	*52.8*	**45.7**
Median	352.5	336.3	177.3	171.3	91.7	87.5	87.4	47.8	45.8	45.7
Speedup	3.8	4.0	7.6	7.9	14.8	15.5	15.5	28.4	29.6	29.7

Table B.9: 2560 by 2560, 16384 Generations, 2 overlap.

Run	4 Nodes		8 Nodes		16 Nodes			32 Nodes		
	Slices	Brick 2-Row	Slices	Brick 2-Row	Slices	Brick 2-Row	Brick 4-Row	Slices	Brick 2-Row	Brick 4-Row
1	**338.1**	**338.9**	**179.6**	**169.9**	**88.0**	*136.2*	*101.7*	**45.1**	*55.7*	*70.7*
2	**333.7**	**337.6**	**169.3**	*213.7*	*103.5*	*96.7*	**85.9**	*68.8*	**45.6**	*47.2*
3	**344.8**	**337.3**	**169.2**	**169.6**	*112.0*	**89.6**	**85.8**	*59.0*	*54.1*	**44.2**
4	**338.1**	**339.1**	**169.4**	**170.1**	**88.5**	*138.0*	**85.8**	**43.8**	*70.8*	*69.0*
5	**338.3**	**338.1**	*233.8*	**170.1**	**88.4**	**89.3**	**86.3**	**44.7**	*50.3*	**45.8**
6	**337.5**	**337.3**	**175.8**	*208.5*	**88.4**	**89.4**	**85.5**	**43.5**	*52.2*	*47.8*
7	**338.0**	**337.3**	**178.5**	**169.5**	*136.2*	**89.2**	**85.7**	*50.6*	**45.5**	**45.6**
8	**338.1**	**336.8**	**182.6**	**169.9**	**87.9**	**89.6**	**85.7**	*55.2*	*56.7*	**44.4**
9	**337.5**	**337.4**	**176.3**	**169.8**	**85.9**	**89.1**	**85.8**	**44.5**	**45.4**	**44.4**
10	**334.1**	**337.6**	**169.0**	**169.9**	**89.0**	**90.5**	**85.7**	*69.1*	**46.1**	*66.1*
11	**351.5**	**337.2**	**169.4**	**169.6**	**93.1**	**89.7**	**85.8**	*77.7*	**46.6**	**45.1**
12	**337.6**	*400.1*	**169.5**	**170.0**	**88.3**	*104.8*	**85.8**	**44.7**	**45.5**	**44.2**
Median	338.0	337.4	169.5	169.9	88.4	89.5	85.8	44.6	45.6	44.4
Speedup	4.0	4.0	8.0	8.0	15.4	15.2	15.8	30.5	29.8	30.5

Table B.10: 5120 by 5120, 4096 Generations, 2 overlap.

Run	4 Nodes		8 Nodes		16 Nodes			32 Nodes		
	Slices	Brick 2-Row	Slices	Brick 2-Row	Slices	Brick 2-Row	Brick 4-Row	Slices	Brick 2-Row	Brick 4-Row
1	**357.9**	**336.3**	**177.9**	**167.9**	**92.1**	**88.6**	**85.3**	**45.1**	**44.6**	**44.4**
2	**353.8**	**355.3**	**180.9**	**176.8**	*186.7*	**88.4**	**91.2**	**47.5**	**44.6**	**43.1**
3	**356.9**	**339.8**	*180.8*	**176.2**	*90.7*	**88.2**	**85.2**	**45.1**	**43.1**	**43.0**
4	**356.4**	**335.5**	**178.4**	**167.8**	**88.9**	*108.8*	**85.3**	**47.4**	**44.3**	**43.2**
5	**360.3**	**337.2**	*178.5*	**169.3**	**91.1**	**88.4**	**85.3**	**45.2**	**45.7**	**44.2**
6	**355.6**	**335.2**	*178.4*	**167.9**	*113.8*	**88.2**	**90.0**	**45.3**	**44.4**	**43.0**
7	**357.9**	**338.0**	*224.3*	**169.9**	**89.6**	**88.2**	**85.0**	**45.5**	**46.7**	**43.1**
8	**355.5**	**337.0**	**177.7**	*253.3*	**90.5**	**88.0**	**85.0**	*45.1*	**44.8**	**43.1**
9	**355.5**	**334.8**	**177.6**	**176.2**	**89.8**	**88.0**	**85.0**	*45.4*	**44.9**	**43.0**
10	**356.8**	**339.6**	**178.2**	**168.0**	**90.7**	**88.3**	**88.3**	*71.1*	**44.4**	**42.8**
11	**360.5**	**354.7**	**178.4**	**177.0**	*139.6*	**89.9**	**85.2**	**44.8**	**44.7**	**42.9**
12	**358.7**	**339.9**	*261.0*	**168.0**	**89.5**	**85.2**	**85.3**	**46.1**	*48.0*	*65.2*
Median	356.9	337.6	178.2	169.3	90.1	88.2	85.3	45.3	44.6	43.1
Speedup	3.8	4.0	7.7	8.1	15.1	15.5	16.0	30.1	30.6	31.7

Table B.11: 10240 by 10240, 1024 Generations, 2 overlap.

Run	4 Nodes		8 Nodes		16 Nodes			32 Nodes		
	Slices	Brick 2-Row	Slices	Brick 2-Row	Slices	Brick 2-Row	Brick 4-Row	Slices	Brick 2-Row	Brick 4-Row
1	**359.4**	**360.7**	**180.5**	**178.8**	**90.7**	*101.8*	**90.0**	**45.6**	*45.2*	*49.2*
2	**360.4**	**360.4**	**180.9**	**179.0**	**91.1**	**92.9**	*90.2*	**47.8**	**45.0**	**44.7**
3	*567.4*	**364.1**	**181.9**	**178.9**	**90.5**	*90.6*	*90.2*	**46.1**	**45.1**	*53.8*
4	**360.0**	**360.7**	**181.3**	**179.0**	**89.7**	*90.5*	**89.6**	**45.5**	**45.4**	*72.9*
5	**359.6**	**361.2**	**182.0**	**178.9**	**90.4**	*91.3*	**89.7**	**45.4**	**45.2**	**44.7**
6	**359.9**	**360.2**	**180.6**	**179.2**	**89.7**	**90.9**	**85.8**	**45.9**	*56.4*	**44.6**
7	**359.7**	**364.2**	**184.2**	**178.6**	*100.6*	**90.1**	**85.9**	**45.6**	**45.5**	**44.8**
8	**357.7**	**357.7**	*288.2*	**179.0**	**89.9**	**94.4**	**89.6**	**45.6**	**45.2**	**44.7**
9	**367.0**	**363.9**	**180.9**	**178.8**	**94.1**	**90.9**	**90.8**	**45.6**	*45.8*	*62.3*
10	**378.7**	**360.4**	**180.5**	**179.5**	**90.6**	**91.1**	**85.7**	**46.3**	**45.3**	**42.8**
11	**364.2**	*360.7*	**182.3**	**178.8**	**90.5**	*91.1*	**89.5**	**47.7**	**45.4**	**44.5**
12	**359.4**	**359.6**	**182.1**	**178.9**	*96.4*	*90.0*	*114.2*	**45.5**	*48.3*	**44.7**
Median	359.9	360.7	181.3	178.9	90.5	91.0	89.6	45.6	45.2	44.7
Speedup	3.9	3.9	7.7	7.8	15.5	15.4	15.7	30.8	31.0	31.4

Table B.12: 20480 by 20480, 256 Generations, 2 overlap.

Run	4 Nodes		8 Nodes		16 Nodes			32 Nodes		
	Slices	Brick 2-Row	Slices	Brick 2-Row	Slices	Brick 2-Row	Brick 4-Row	Slices	Brick 2-Row	Brick 4-Row
1	**341.0**	**343.8**	**175.6**	*259.8*	**92.5**	**91.3**	**90.7**	**51.3**	*70.3*	*68.5*
2	**342.7**	**340.9**	**175.7**	**175.1**	**92.4**	**91.7**	**90.9**	**51.1**	**49.5**	**48.9**
3	**342.8**	**342.3**	**175.7**	**175.1**	**92.3**	**91.5**	**90.8**	**51.8**	*62.6*	*64.6*
4	**342.7**	**344.1**	**175.3**	**175.7**	**92.3**	**91.6**	**90.9**	**51.0**	*54.1*	*72.5*
5	**340.5**	**344.0**	**183.3**	**175.2**	**92.5**	**91.3**	**90.8**	**51.1**	**49.2**	**48.7**
6	**343.0**	**344.3**	*219.5*	**175.1**	**92.4**	**91.1**	**90.9**	**51.1**	*67.1*	*51.1*
7	**342.8**	**347.2**	**175.2**	**175.1**	**92.5**	*120.7*	**90.8**	**51.1**	**49.2**	**48.8**
8	**343.5**	**343.3**	*194.8*	**174.9**	**92.4**	**91.2**	**90.8**	**51.0**	**49.1**	**48.6**
9	**341.6**	**342.7**	**175.6**	**175.2**	*183.3*	**91.4**	*98.6*	*59.9*	**49.2**	**48.6**
10	**342.8**	**344.0**	**175.6**	**175.1**	**92.4**	**91.1**	**90.7**	**51.0**	**49.3**	**49.4**
11	**350.1**	**344.4**	**175.6**	**174.7**	*131.4*	**91.3**	**90.6**	**50.9**	**49.3**	**48.5**
12	*407.2*	**343.4**	**175.7**	**175.0**	**92.5**	**91.5**	**90.5**	*62.1*	**49.2**	*72.2*
Median	342.8	343.9	175.6	175.1	92.4	91.3	90.8	51.1	49.2	48.7
Speedup	3.9	3.9	7.7	7.7	14.6	14.8	14.9	26.4	27.4	27.7

Table B.13: 1280 by 1280, 65536 Generations, 4 overlap.

Run	4 Nodes		8 Nodes		16 Nodes			32 Nodes		
	Slices	Brick 2-Row	Slices	Brick 2-Row	Slices	Brick 2-Row	Brick 4-Row	Slices	Brick 2-Row	Brick 4-Row
1	**338.3**	**339.1**	**170.7**	**170.5**	**87.7**	**87.1**	**87.1**	*53.0*	**45.5**	**45.5**
2	**338.7**	**336.5**	*251.7*	**169.6**	**87.4**	**87.1**	**89.2**	**45.8**	**45.4**	**45.3**
3	**339.1**	**336.2**	**171.0**	**170.5**	**87.5**	*108.6*	**87.1**	*57.2*	*49.5*	*69.9*
4	**340.0**	**336.3**	**170.8**	**169.8**	**87.4**	**87.0**	**87.5**	**46.0**	*71.1*	*57.4*
5	**337.2**	**342.4**	**171.0**	**174.9**	**88.1**	**87.1**	**87.0**	*63.1*	**46.9**	**45.4**
6	**336.5**	**336.5**	*274.7*	**170.4**	**87.3**	*92.7*	**87.2**	**46.7**	**45.5**	**45.5**
7	*361.8*	*358.3*	**171.3**	**169.8**	**87.5**	*115.7*	**88.2**	*49.2*	**45.4**	**45.2**
8	**352.0**	**338.8**	**170.4**	*276.7*	**87.3**	**87.2**	*139.6*	**45.9**	**45.8**	**45.2**
9	**338.3**	**349.3**	**178.2**	**169.6**	**87.5**	**87.2**	**87.1**	**45.8**	**45.4**	**45.2**
10	**337.1**	**338.9**	**171.3**	**170.4**	**87.4**	**87.0**	**87.1**	**45.9**	**45.5**	**45.2**
11	**337.2**	**338.6**	**171.7**	**170.7**	**87.4**	**87.2**	**87.2**	**46.0**	**45.7**	*53.0*
12	**337.3**	**338.9**	**170.8**	**170.7**	**87.3**	**90.8**	*107.8*	**45.8**	*52.1*	*59.8*
Median	338.3	338.8	171.0	170.4	87.4	87.1	87.1	45.9	45.5	45.3
Speedup	4.0	4.0	7.9	7.9	15.5	15.5	15.5	29.5	29.8	29.9

Table B.14: 2560 by 2560, 16384 Generations, 4 overlap.

Run	4 Nodes		8 Nodes		16 Nodes			32 Nodes		
	Slices	Brick 2-Row	Slices	Brick 2-Row	Slices	Brick 2-Row	Brick 4-Row	Slices	Brick 2-Row	Brick 4-Row
1	**351.5**	**337.8**	**178.3**	**169.0**	**89.2**	**85.7**	**85.5**	*58.4*	*54.0*	**43.4**
2	**351.5**	**336.0**	**183.3**	**169.1**	**89.1**	**88.5**	**85.2**	**45.7**	**44.2**	**43.4**
3	**350.6**	**336.1**	**176.3**	**169.3**	*128.7*	*96.6*	**85.1**	*45.6*	*52.0*	**44.4**
4	**351.5**	**337.3**	*283.8*	**170.0**	**88.9**	**88.3**	**85.5**	*65.5*	**47.1**	**43.4**
5	**350.4**	**350.8**	**176.2**	**176.3**	**89.0**	**88.5**	**85.7**	*72.1*	**43.6**	**44.0**
6	**361.0**	*391.7*	**177.7**	**169.8**	**89.0**	**88.3**	**85.4**	**45.6**	**43.6**	**44.8**
7	**351.0**	**350.6**	**176.2**	**169.6**	**88.5**	*108.5*	**85.3**	**45.6**	**45.8**	**43.4**
8	**351.9**	**335.9**	**176.3**	**169.3**	**88.9**	*131.4*	*131.0*	**45.3**	**43.7**	**43.3**
9	**351.6**	**336.0**	**177.8**	**176.4**	**88.8**	**88.3**	**85.9**	*67.7*	**44.0**	**43.3**
10	**351.7**	**336.1**	*223.4*	**169.8**	**88.9**	**88.3**	**85.3**	**45.8**	**44.0**	**43.4**
11	**351.6**	**336.1**	**177.8**	**172.4**	**89.1**	**85.7**	**85.5**	*63.0*	*69.1*	**43.4**
12	**351.5**	**336.1**	**176.6**	**170.9**	**89.4**	**88.9**	**85.4**	**45.8**	**43.6**	**44.3**
Median	351.5	336.1	177.1	169.8	89.0	88.3	85.4	45.6	44.0	43.4
Speedup	3.9	4.0	7.7	8.0	15.3	15.4	15.9	29.8	30.9	31.3

Table B.15: 5120 by 5120, 4096 Generations, 4 overlap.

Run	4 Nodes		8 Nodes		16 Nodes			32 Nodes		
	Slices	Brick 2-Row	Slices	Brick 2-Row	Slices	Brick 2-Row	Brick 4-Row	Slices	Brick 2-Row	Brick 4-Row
1	**352.2**	**333.9**	**177.6**	**175.9**	**89.7**	**88.4**	**87.9**	**45.6**	**44.7**	**44.3**
2	**354.6**	**365.2**	**177.6**	**176.9**	**89.8**	**88.2**	**84.9**	*45.7*	*61.1*	**44.2**
3	*432.9*	**334.1**	*266.9*	**169.9**	**89.8**	*100.0*	**88.0**	*49.4*	**44.7**	*59.3*
4	**370.4**	**351.9**	**178.3**	*209.9*	**90.1**	**88.4**	**88.0**	*71.2*	**44.6**	**44.8**
5	**354.2**	**352.1**	**180.8**	**177.4**	**89.7**	**88.1**	**88.1**	*45.7*	**44.7**	**42.8**
6	**354.3**	**352.2**	**177.6**	**177.1**	**89.8**	**88.3**	*132.8*	**45.3**	*46.3*	*48.0*
7	**354.1**	**352.1**	**178.7**	**171.1**	**89.5**	*93.2*	**84.9**	*54.8*	**44.7**	**44.3**
8	**354.4**	**334.1**	**179.5**	**169.5**	**89.8**	**88.4**	**84.8**	*45.7*	**44.7**	**42.8**
9	*354.2*	**351.9**	**179.9**	**176.9**	**90.2**	**88.3**	**87.9**	**45.5**	*56.4*	*49.4*
10	**355.7**	**350.7**	**178.8**	**169.9**	*91.3*	**90.0**	**92.1**	*49.8*	**45.1**	**43.1**
11	**354.8**	**334.1**	**178.1**	**173.4**	*97.1*	**88.0**	**84.8**	*65.2*	**44.8**	**42.7**
12	**354.3**	**334.2**	**181.7**	**176.1**	**89.7**	**88.3**	**88.0**	**45.7**	**45.6**	**44.4**
Median	354.4	351.3	178.7	175.9	89.8	88.3	87.9	45.6	44.7	44.2
Speedup	3.8	3.9	7.6	7.8	15.2	15.4	15.5	29.9	30.5	30.8

Table B.16: 10240 by 10240, 1024 Generations, 4 overlap.

Run	4 Nodes		8 Nodes		16 Nodes			32 Nodes		
	Slices	Brick 2-Row	Slices	Brick 2-Row	Slices	Brick 2-Row	Brick 4-Row	Slices	Brick 2-Row	Brick 4-Row
1	**359.7**	**356.5**	**181.3**	**181.8**	**90.5**	*94.6*	**88.5**	*49.3*	**45.3**	**45.2**
2	**360.1**	**357.3**	**180.8**	**182.2**	**90.0**	**90.2**	*108.2*	*64.2*	**45.3**	*69.1*
3	**364.3**	**357.4**	**180.9**	**181.7**	**90.6**	**91.0**	**88.9**	**47.0**	**45.7**	**45.4**
4	**359.9**	**359.5**	**181.3**	**182.0**	**90.8**		**88.5**	**45.8**	**45.3**	**46.7**
5	**364.2**	**359.1**	*192.5*	**180.5**	**90.6**	**93.4**	**88.8**	**46.0**	**45.2**	**45.4**
6	**360.5**	**357.5**	**182.4**	**181.7**	**90.1**	**91.2**	**89.0**	**45.4**	**45.3**	**45.0**
7	**360.1**	**357.2**	**185.0**	**180.4**	*89.9*	*97.8*	**88.8**	**46.1**	**46.3**	*46.6*
8	**364.2**	*359.4*	**182.5**	**180.4**	**90.6**	**91.1**	*93.8*	**45.7**	**45.2**	**44.8**
9	**364.0**	*359.2*	**182.0**	**180.4**	**90.6**	**91.2**	**88.8**	**45.9**	**45.3**	**44.4**
10	**360.0**	**359.8**	**181.0**	**192.3**	**90.0**	*145.4*	**88.8**	**46.7**	**47.5**	**44.8**
11	**360.2**	*357.3*	**180.5**	**180.6**	**90.5**	*89.8*	**88.8**	**45.7**	**45.0**	*45.6*
12	**360.5**	**357.3**	**180.7**	**182.5**	**90.5**	**91.0**	*130.4*	**47.0**	**45.3**	**45.9**
Median	360.3	357.4	181.3	181.7	90.5	91.1	88.8	45.9	45.3	45.2
Speedup	3.9	3.9	7.7	7.7	15.5	15.4	15.8	30.6	31.0	31.1

Table B.17: 20480 by 20480, 256 Generations, 4 overlap.

Run	4 Nodes		8 Nodes		16 Nodes			32 Nodes		
	Slices	Brick 2-Row	Slices	Brick 2-Row	Slices	Brick 2-Row	Brick 4-Row	Slices	Brick 2-Row	Brick 4-Row
1	**343.9**	**343.2**	**177.6**	**175.6**	**94.5**	**92.0**	**91.5**	**53.4**	*68.4*	**50.6**
2	**344.8**	**343.0**	**177.4**	**176.0**	*104.6*	*152.2*	**91.1**	**53.4**	**50.0**	**49.0**
3	**346.5**	**345.4**	**178.2**	**176.0**	**98.0**	**92.0**	**91.1**	**53.5**	**52.0**	*53.8*
4	**344.9**	**344.9**	**177.7**	**178.8**	**94.4**	**91.9**	**91.1**	*57.3*	*56.7*	*76.2*
5	**345.4**	**345.4**	**178.1**	**175.6**	**95.1**	**91.8**	**91.1**	**53.4**	**49.9**	**48.6**
6	**343.9**	**345.5**	**177.2**	**175.6**	**94.8**	**92.0**	**91.0**	**53.7**	**51.2**	**48.6**
7	**343.5**	**343.2**	**178.0**	**175.7**	**94.3**	**96.4**	*134.3*	*67.2*	*66.0*	*70.6*
8	**344.6**	**343.1**	**177.6**	**176.3**	**94.5**	**92.0**	**91.0**	**53.4**	**54.1**	**48.6**
9	**343.8**	**345.2**	**182.6**	**175.7**	*107.0*	**92.0**	**91.1**	**55.3**	**49.9**	*54.8*
10	**344.2**	**344.9**	**183.8**	**175.8**	**97.2**	**91.9**	**91.4**	*80.8*	*75.7*	**48.5**
11	**343.9**	**345.0**	**177.4**	**176.1**	**94.8**	*103.4*	**91.2**	**53.4**	**52.1**	
12	**344.0**	**343.3**	**177.5**	**175.9**	*123.4*	**92.1**	**91.1**	**55.8**	**49.9**	**48.5**
Median	344.1	344.9	177.7	175.9	94.8	92.0	91.1	53.4	50.6	48.6
Speedup	3.9	3.9	7.6	7.7	14.2	14.7	14.8	25.2	26.6	27.8

Table B.18: 1280 by 1280, 65536 Generations, 8 overlap.

Run	4 Nodes		8 Nodes		16 Nodes			32 Nodes		
	Slices	Brick 2-Row	Slices	Brick 2-Row	Slices	Brick 2-Row	Brick 4-Row	Slices	Brick 2-Row	Brick 4-Row
1	**340.7**	**355.5**	**179.3**	**170.9**	**89.5**	**87.7**	**87.4**	**49.0**	*48.6*	**45.3**
2	**340.0**	**340.9**	**172.6**	**170.7**	**91.8**	**87.7**	**87.5**	**48.3**	**45.8**	**45.3**
3	**340.6**	*459.9*	**180.8**	**170.2**	**89.6**	**87.8**	**90.3**	*49.4*	**46.1**	*51.0*
4	**351.8**	**340.7**	**179.2**	**171.9**	**91.8**	*93.8*	*92.8*	**49.3**	**45.9**	*60.5*
5	**340.0**	**340.5**	**178.8**	**171.8**	**93.1**	**87.8**	**87.5**	*49.5*	*60.2*	**45.4**
6	**355.4**	**341.0**	**178.6**	*196.1*	*113.1*	**87.9**	**88.4**	*63.1*	**46.4**	**45.3**
7	**341.4**	**342.1**	**178.6**	**171.9**	**91.9**	*94.1*	**87.5**	**48.3**	**45.8**	**45.4**
8	**355.5**	**349.5**	**172.9**	**171.7**	**91.8**	**87.3**	**87.9**	**51.1**	*56.5*	**45.3**
9	**341.0**	**341.1**	**178.2**	**171.6**	**89.5**	*95.0*	*95.9*	*71.5*	**45.9**	**47.5**
10	**340.9**	**339.4**	**172.9**	**170.8**	**92.1**	**93.1**	**86.9**	*52.4*	**45.9**	**45.3**
11	**352.3**	**339.0**	**179.2**	**174.3**	**92.6**	**87.8**	**87.2**	*74.8*	**46.1**	**45.9**
12	**353.1**	**345.4**	**175.8**	**170.7**	**93.7**	**87.7**	*97.3*	**49.3**	**46.8**	**45.3**
Median	341.2	341.0	178.6	171.6	91.8	87.8	87.5	49.1	45.9	45.3
Speedup	4.0	4.0	7.6	7.9	14.7	15.4	15.5	27.6	29.5	29.9

Table B.19: 2560 by 2560, 16384 Generations, 8 overlap.

Run	4 Nodes		8 Nodes		16 Nodes			32 Nodes		
	Slices	Brick 2-Row	Slices	Brick 2-Row	Slices	Brick 2-Row	Brick 4-Row	Slices	Brick 2-Row	Brick 4-Row
1	**352.0**	**337.6**	**177.0**	**170.4**	*110.0*	*129.6*	**85.6**	**46.3**	**45.2**	**43.3**
2	*372.1*	**338.0**	*186.6*	**170.2**	**90.3**	**88.7**	**85.1**	**47.0**	**45.7**	*58.5*
3	**335.4**	**339.7**	**177.5**	**176.7**	**90.2**	**88.8**	**85.6**	**46.1**	**45.2**	**43.3**
4	**351.8**	**338.3**	**176.8**	**170.0**	**90.2**	**88.7**	**85.6**	**46.4**	**45.2**	**43.3**
5	**353.0**	**354.4**	**177.1**	**177.3**	*189.4*	**88.9**	**85.6**	*53.9*	**45.4**	**43.3**
6	**351.4**	**353.5**	**177.3**	**170.1**	**89.9**	**85.8**	**85.4**	**48.0**	**45.2**	**44.5**
7	**353.2**	**339.5**	**177.3**	**170.3**	**90.0**	**89.7**	**85.5**	**46.1**	**45.2**	**44.3**
8	**359.1**	**355.1**	**178.1**	*187.0*	**90.5**	*119.9*	**85.6**	*47.2*	**45.2**	**43.3**
9	**352.7**	**339.7**	*187.4*	**176.7**	*129.1*	**88.9**	**85.6**	**46.4**	**45.1**	**43.5**
10	**352.8**	**337.7**	**177.3**	**170.3**	**90.2**	**88.8**	**85.5**	**46.2**	**45.2**	**43.4**
11	**357.0**	**346.4**	**177.3**	**170.3**	**90.1**	**88.8**	**85.6**	**46.0**	**45.2**	**43.3**
12	**352.4**	**339.7**	**177.9**	**178.2**	**94.9**	**89.3**	**85.6**	**46.3**	**45.2**	**44.2**
Median	352.7	339.7	177.3	170.3	90.2	88.8	85.6	46.3	45.2	43.3
Speedup	3.8	4.0	7.7	8.0	15.1	15.3	15.9	29.3	30.0	31.3

Table B.20: 5120 by 5120, 4096 Generations, 8 overlap.

Run	4 Nodes		8 Nodes		16 Nodes			32 Nodes		
	Slices	Brick 2-Row	Slices	Brick 2-Row	Slices	Brick 2-Row	Brick 4-Row	Slices	Brick 2-Row	Brick 4-Row
1	**355.7**	**337.5**	**177.8**	**175.7**	**90.2**	**88.1**	**88.8**	**46.2**	**45.2**	**44.6**
2	**353.8**	**356.4**	*200.4*	**176.9**	**89.8**	*138.7*	**91.0**	**46.6**	**44.9**	**44.7**
3	**352.1**	**354.3**	**177.9**	**177.4**	**89.0**	**88.2**	**89.0**	*55.1*	**44.7**	*69.1*
4	**352.2**	**354.6**	*177.9*	**188.8**	*102.1*	**88.4**	**89.0**	**45.9**	**45.5**	**44.5**
5	**354.5**	**338.5**	**178.8**	**176.5**	**89.2**	**88.4**	**89.1**	**46.1**	**46.0**	*51.0*
6	**354.5**	**354.2**	**178.7**	**177.0**	**90.4**	**88.6**	**88.8**	**47.0**	**45.0**	**44.5**
7	**354.5**	**359.2**	**179.0**	**176.2**	**89.7**	**91.8**	*124.6*	*51.2*	*72.7*	**44.7**
8	*354.7*	**351.9**	**178.3**	**176.9**	**89.2**	**88.7**	*113.2*	*51.2*	**46.0**	**45.0**
9	**352.0**	**337.5**	**178.9**	**176.6**	**96.7**	**87.3**	**89.2**	**46.1**	*68.5*	**45.0**
10	*354.6*	**339.5**	*177.9*	**176.6**	*90.6*	**88.3**	**88.9**	**46.2**	**44.9**	**44.4**
11	**354.3**	**337.6**	**177.8**	**176.4**	**94.0**	*97.1*	**90.3**	**45.8**	*62.9*	**44.3**
12	**353.2**	**338.6**	**178.9**	**178.9**	**89.7**	**88.3**	**89.4**	**46.6**	*45.3*	**45.2**
Median	354.1	345.7	178.7	176.7	89.7	88.4	89.1	46.2	45.1	44.6
Speedup	3.9	3.9	7.6	7.7	15.2	15.4	15.3	29.5	30.3	30.5

Table B.21: 10240 by 10240, 1024 Generations, 8 overlap.

Run	4 Nodes		8 Nodes		16 Nodes			32 Nodes		
	Slices	Brick 2-Row	Slices	Brick 2-Row	Slices	Brick 2-Row	Brick 4-Row	Slices	Brick 2-Row	Brick 4-Row
1	**364.7**	**362.6**	**180.9**	**180.5**	**90.4**	**90.2**	*90.3*	*52.1*	*49.1*	*56.7*
2	**361.4**	**357.9**	**182.9**	**180.1**	**91.1**	**92.7**	**90.2**	**47.3**	**45.4**	*45.0*
3	**360.8**	**363.6**	**183.1**	*182.5*	**90.2**	**91.0**	*106.4*	**45.8**	*45.2*	**45.6**
4	**360.4**	*358.0*	**183.0**	*180.4*	**90.8**	**91.1**	*127.4*	**46.0**	**46.6**	**45.0**
5	*392.7*	*357.9*	**180.9**	**180.2**	**90.7**	**90.1**	*130.4*	**46.0**	*56.1*	*45.8*
6	**360.6**	**364.0**	**181.3**	*184.2*	**91.8**	**91.1**	**89.8**	*60.0*	*55.9*	*45.4*
7	**360.0**	**364.0**	**181.4**	*184.5*	**90.2**	*95.5*	**90.1**	*50.1*	**45.5**	*48.2*
8	**363.8**	**358.1**	*201.6*	**180.2**	**90.6**	**93.1**	**89.6**	*71.5*	*48.9*	*49.7*
9	**360.3**	**357.9**	**181.3**	**180.7**	**90.6**	**90.1**	**89.5**	*71.1*	**45.5**	*45.3*
10	**360.3**	**358.0**	**182.8**	**181.9**	**90.6**	**97.3**	**94.5**	*68.7*	**45.3**	*90.8*
11	**360.3**	**358.6**	**181.4**	**181.7**	**91.2**	**91.0**	*128.6*	**45.7**	**45.4**	*45.5*
12	**361.6**	**357.5**	**182.6**	**180.7**	**90.7**	*107.4*	**90.9**	**45.9**	**46.2**	*45.4*
Median	360.6	358.3	181.4	180.6	90.7	91.1	90.1	46.0	45.5	45.3
Speedup	3.9	3.9	7.7	7.8	15.5	15.4	15.6	30.5	30.9	31.0

Table B.22: 20480 by 20480, 256 Generations, 8 overlap.

Run	4 Nodes		8 Nodes		16 Nodes			32 Nodes		
	Slices	Brick 2-Row	Slices	Brick 2-Row	Slices	Brick 2-Row	Brick 4-Row	Slices	Brick 2-Row	Brick 4-Row
1	**353.8**	**351.3**	**185.9**	**184.1**	**106.7**	**96.3**	**94.2**	**61.2**	**53.7**	**50.9**
2	**353.7**	**353.4**	**185.9**	**182.0**	**102.8**	**100.6**	**93.7**	**61.2**	**53.7**	**50.6**
3	**352.5**	**353.7**	**186.5**	**182.2**	*162.8*	*109.9*	**94.2**	**61.2**	*59.5*	**51.8**
4	**353.6**	**355.2**	**186.8**	**181.7**	**103.1**	**97.3**	**94.5**	*66.1*	**56.6**	**50.8**
5	**353.8**	**353.6**	**186.4**	**181.8**	*141.9*	*105.1*	**94.3**	**61.3**	**54.1**	*63.1*
6	**352.9**	**353.4**	*201.8*	**182.4**	**102.7**	**96.4**	*150.1*	**61.1**	**53.8**	*56.4*
7	**353.7**	**352.5**	**186.4**	**181.6**	**103.0**	**96.5**	**94.3**	**61.4**	**53.8**	**51.1**
8	**353.2**	**352.5**	**192.5**	**181.8**	**102.9**	*106.8*	**94.3**	**61.2**	*84.9*	**50.8**
9	**352.9**	**352.5**	**186.2**	**194.3**	**103.3**	**96.3**	**94.4**	**61.3**	**53.9**	**50.7**
10	**353.9**	**354.0**	**186.5**	*297.9*	**103.3**	**98.8**	*116.6*	**61.2**	**54.3**	*64.9*
11	**353.0**	**352.5**	*199.3*	**181.8**	*168.6*	**96.5**	**94.3**	*71.6*	**53.6**	**50.7**
12	**353.6**	**352.2**	*261.9*	**183.2**	*110.3*	**96.3**	**94.4**	**61.2**	**53.7**	**50.8**
Median	353.6	353.0	186.4	182.0	103.1	96.5	94.3	61.2	53.8	50.8
Speedup	3.8	3.8	7.2	7.4	13.1	14.0	14.3	22.0	25.1	26.5

Table B.23: 1280 by 1280, 65536 Generations, 16 overlap.

Run	4 Nodes		8 Nodes		16 Nodes			32 Nodes		
	Slices	Brick 2-Row	Slices	Brick 2-Row	Slices	Brick 2-Row	Brick 4-Row	Slices	Brick 2-Row	Brick 4-Row
1	**355.9**	**344.2**	**192.9**	**176.1**	**92.9**	**91.0**	**89.0**	*74.6*	**48.7**	*66.2*
2	**356.4**	**344.3**	**176.7**	**175.2**	**93.0**	**90.0**	*105.1*	**53.0**	**47.8**	**46.3**
3	**344.2**	**342.7**	**176.2**	**175.1**	**96.1**	**90.1**	**88.9**	**52.2**	**47.9**	*50.6*
4	*486.9*	**342.6**	**184.3**	**175.3**	**92.9**	**90.2**	**89.0**	**51.8**	**47.8**	**46.3**
5	*484.8*	**344.3**	**182.4**	**178.2**	**92.9**	**91.6**	**89.6**	**52.5**	**48.1**	*53.6*
6	**356.4**	**344.2**	**176.2**	**175.5**	**93.0**	**90.9**	**89.1**	*64.7*	**47.7**	**46.1**
7	**360.6**	**342.6**	**176.6**	**175.2**	**93.0**	*101.4*	*108.7*	*55.6*	**47.7**	**46.2**
8	**343.6**	**351.5**	**182.3**	**175.1**	**96.6**	**90.0**	**89.3**	**53.2**	**48.1**	**46.2**
9	**343.8**	**344.3**	**182.8**	**175.0**	*110.0*	**90.6**	**88.8**	**52.5**	**47.8**	*58.9*
10	**344.2**	**344.3**	**182.5**	**175.2**	**95.3**	**90.9**	**88.9**	**52.6**	**47.7**	**47.3**
11	**344.3**	**344.3**	**176.2**	**174.7**	**95.5**	**90.1**	**88.8**	*56.9*	**47.8**	**46.2**
12	**356.8**	**342.8**	**176.6**	*217.0*	**92.7**	**90.2**	**89.8**	*62.3*	**47.8**	**46.4**
Median	350.1	344.2	179.5	175.2	93.0	90.2	89.0	52.5	47.8	46.3
Speedup	3.9	3.9	7.5	7.7	14.6	15.0	15.2	25.8	28.3	29.3

Table B.24: 2560 by 2560, 16384 Generations, 16 overlap.

Run	4 Nodes		8 Nodes		16 Nodes			32 Nodes		
	Slices	Brick 2-Row	Slices	Brick 2-Row	Slices	Brick 2-Row	Brick 4-Row	Slices	Brick 2-Row	Brick 4-Row
1	**341.5**	**342.3**	**173.1**	**171.7**	**92.3**	**90.2**	**86.3**	*48.4*	**46.5**	*51.7*
2	**353.3**	**341.9**	**179.3**	**171.9**	**91.4**	**90.4**	**86.3**	*48.4*	*48.3*	**45.1**
3	**337.4**	**341.5**	**179.3**	**184.7**	**92.5**	**86.7**	**88.7**	*48.4*	**46.4**	**44.0**
4	**340.7**	**357.1**	**179.4**	**178.9**	**95.1**	**87.9**	**87.2**	*62.5*	**47.6**	**45.5**
5	**341.0**	**357.1**	**179.5**	**179.3**	**91.8**	**89.4**	**86.3**	*62.7*	*51.6*	*47.3*
6	**337.1**	**340.9**	**179.6**	**171.7**	**92.5**	**90.3**	**86.5**	*55.3*	**46.4**	**44.0**
7	**337.3**	**340.9**	**179.2**	**171.0**	**91.6**	*106.2*	*137.5*	*53.3*	**46.5**	*69.3*
8	*556.9*	**341.0**	**179.0**	**179.3**	**92.5**	**86.8**	**86.2**	*48.4*	*51.6*	**44.0**
9	**353.9**	**341.5**	**179.4**	**178.7**	**92.4**	**90.2**	**86.2**	*60.4*	**46.5**	**44.1**
10	**354.9**	**341.4**	**179.3**	**171.7**	**93.3**	*103.5*	**86.3**	*48.7*	**46.4**	*47.1*
11	**337.6**	**340.9**	**178.9**	**178.8**	**91.7**	**90.4**	*93.3*	**48.5**	**46.4**	**45.0**
12	**340.5**	**341.5**	**178.8**	**170.6**	**92.5**	**90.3**	**86.2**	*55.9*	**46.4**	**44.4**
Median	340.7	341.5	179.3	175.3	92.4	90.2	86.3	48.5	46.4	44.2
Speedup	4.0	4.0	7.6	7.7	14.7	15.1	15.7	28.0	29.2	30.7

Table B.25: 5120 by 5120, 4096 Generations, 16 overlap.

Run	4 Nodes		8 Nodes		16 Nodes			32 Nodes		
	Slices	Brick 2-Row	Slices	Brick 2-Row	Slices	Brick 2-Row	Brick 4-Row	Slices	Brick 2-Row	Brick 4-Row
1	**356.8**	**340.7**	*321.9*	**179.0**	**91.6**	*126.0*	**90.5**	**47.2**	*47.7*	**46.1**
2	*355.9*	**353.2**	*179.4*	**169.7**	**91.2**	**89.6**	**89.4**	**47.2**	**45.2**	**47.4**
3	**358.6**	**337.0**	**180.0**	**177.9**	*91.2*	**85.4**	**85.7**	*47.0*	**45.3**	**45.0**
4	**355.6**	**355.0**	**179.3**	**170.3**	**91.1**	**89.7**	**85.6**	*56.9*	**45.3**	**44.9**
5	**355.7**	**354.7**	**179.1**	*180.3*	**90.3**	**89.7**	**85.5**	*47.0*	*47.3*	**45.2**
6	**355.5**	**337.1**	**179.4**	**178.2**	*91.7*	**85.0**	**85.7**	*47.2*	**48.4**	**46.6**
7	*356.5*	**353.1**	*179.0*	**169.9**	*92.6*	*90.2*	**85.9**	*47.2*	**45.4**	**44.9**
8	**355.8**	**339.8**	**179.2**	**177.8**	**90.7**	**89.5**	**89.6**	*47.0*	**45.4**	**45.0**
9	*358.9*	**353.1**	*180.1*	**178.5**	**91.1**	**90.9**	**85.7**	**47.2**	**45.3**	*51.5*
10	**372.0**	**354.8**	**180.0**	**179.1**	*91.5*	**89.3**	**89.6**	*47.3*	**45.3**	*65.5*
11	**355.8**	**353.1**	**180.0**	*208.2*	**91.5**	*103.1*	**89.7**	**47.3**	*53.0*	**45.0**
12	*409.4*	**353.3**	**180.7**	**170.5**	**92.0**	**89.5**	**85.8**	*47.0*	*60.0*	**45.0**
Median	355.8	353.1	179.7	177.8	91.2	89.5	85.8	47.2	45.3	45.0
Speedup	3.8	3.9	7.6	7.7	15.0	15.2	15.9	28.9	30.1	30.3

Table B.26: 10240 by 10240, 1024 Generations, 16 overlap.

Run	4 Nodes		8 Nodes		16 Nodes			32 Nodes		
	Slices	Brick 2-Row	Slices	Brick 2-Row	Slices	Brick 2-Row	Brick 4-Row	Slices	Brick 2-Row	Brick 4-Row
1	**365.2**	**362.2**	**181.2**	**180.6**	**91.3**	**91.0**	*141.4*	*61.6*	**46.4**	**45.1**
2	**365.1**	**365.3**	**182.0**	**180.2**	**92.1**	**91.5**	**89.5**	*50.0*	**45.6**	*51.1*
3	**360.4**	**361.8**	**181.4**	**179.3**	*115.9*	**91.5**	**89.6**	*69.9*	**46.0**	**44.9**
4	**364.6**	**365.0**	**181.7**	*202.0*	**91.1**	*90.7*	**89.5**	*50.5*	*69.1*	**45.9**
5	**360.7**	**361.9**	**181.5**	*191.8*	**91.4**	**90.4**	**89.5**	*49.9*	*73.5*	*56.6*
6	**365.1**	**362.4**	**182.0**	**179.9**	**94.2**	*90.5*	**89.7**	**46.4**	**46.5**	**44.9**
7	**364.9**	**361.0**	**183.5**	**180.8**	**90.8**	**91.6**	**89.6**	**47.3**	**47.1**	**46.1**
8	**365.5**	**364.3**	**187.4**	*185.4*	**91.5**	*131.3*	**89.7**	**46.5**	**45.6**	*71.5*
9	**364.8**	**362.4**	**180.0**	**180.0**	**91.4**	**90.2**	**89.7**	**46.5**	**45.7**	*47.2*
10	**361.4**	**365.0**	*215.4*	*180.0*	**91.3**	**90.1**	*124.0*	*66.0*	**46.5**	*46.8*
11	**361.3**	**361.9**	**183.3**	**181.1**	**90.9**	**91.3**	**89.6**	**47.4**	**45.7**	**45.0**
12	**360.5**	**361.7**	**183.4**	**181.5**	**90.6**	**91.4**	**89.5**	**47.9**	**45.6**	**44.9**
Median	364.7	362.3	182.0	180.4	91.3	91.3	89.6	46.9	45.8	45.0
Speedup	3.8	3.9	7.7	7.8	15.4	15.4	15.7	29.9	30.6	31.2

Table B.27: 20480 by 20480, 256 Generations, 16 overlap.

Bibliography

[1] Structured Grids - A View from Berkeley. Retrieved September 15, 2010, from http://view.eecs.berkeley.edu/wiki/Structured_Grids.

[2] Abstractions for Dynamic Data Distribution. *High-Level Programming Models and Supportive Environments, International Workshop on*, 0:42–51, 2004.

[3] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan willem Maessen, Sukyoung Ryu, Guy L. Steele, Joao Dias, Carl Eastlund, Christine Flood, Yossi Lev, and Cheryl Mccosh. Fortress in a Nutshell, 2007.

[4] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[5] Mark Baker, Bryan Carpenter, and Aamir Shafi. MPJ Express: Towards Thread Safe Java HPC. In *IEEE International Conference on Cluster Computing (Cluster 2006*, pages 25–28, 2006.

[6] Mark Baker, Bryan Carpenter, and Aamir Shafi. MPJ Express: Towards Thread Safe Java HPC. In *IEEE International Conference on Cluster Computing (Cluster 2006*, pages 25–28, 2006.

[7] E. Berlekamp, J. Conway, and R. Guy. *Winning Ways for your Mathematical Plays*, volume 4. Academic, 2nd edition, 2003.

[8] Shahid H. Bokhari. Multiprocessing the Sieve of Eratosthenes. *Computer*, 20:50–58, April 1987.

[9] Markus Bornemann, Rob V. Van Nieuwpoort, and Thilo Kielmann. MPJ/Ibis: A Flexible and Efficient Message Passing Platform for Java. In *In Euro PVM/MPI 2005, volume 3666 of LNCS*, pages 217–224. Springer-Verlag, 2005.

[10] Bryan Carpenter. mpiJava: A Java interface to MPI. In *In First UK Workshop on Java for High Performance Network Computing, Europar '98*, 1998.

[11] S. Ceri, G. Gottlob, and L. Tanca. What You Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Trans. on Knowl. and Data Eng.*, 1:146–166, March 1989.

[12] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.*, 21:291–312, August 2007.

[13] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an Object-Oriented Approach to Non-Uniform Cluster Computing. *SIGPLAN Not.*, 40:519–538, October 2005.

[14] Rodger J Clayton. *Compilation of Bottom-Up Evaluation for a Pure Logic Programming Language*. PhD thesis, University of Waikato, 2005.

[15] David DeWitt and Jim Gray. Parallel Database Systems: the Future of High Performance Database Systems. *Commun. ACM*, 35:85–98, June 1992.

[16] John T. Feo, David C. Cann, and Rodney R. Oldehoeft. A Report on the Sisal Language Project. *J. Parallel Distrib. Comput.*, 10:349–366, December 1990.

[17] High Performance Fortran Forum. High Performance Fortran Language Specification, 1994.

[18] Ian Foster. *Designing and Building Parallel Programs*. TODO.

[19] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, November 1994.

[20] Martin Gardner. Mathematical Games. *Scientific American*, October 1970.

[21] Gosper. Exploiting Regularities in Large Cellular Spaces. *Physica D: Nonlinear Phenomena*, 10(1-2):75–80, January 1984.

[22] Gopal Gupta, Khayri A. M. Ali, Mats Carlsson, and Manuel V. Hermenegildo. Parallel execution of prolog programs: A survey. *ACM Transactions on Programming Languages and Systems*, 23:2001, 1995.

[23] Joseph M. Hellerstein. The Declarative Imperative: Experiences and Conjectures in Distributed Logic. *SIGMOD Rec.*, 39:5–19, September 2010.

[24] Alan Hensel. About my Conway's Game of Life Applet, January 2001. Retrieved June 7, 2010, from http://www.ibiblio.org/lifepatterns/.

[25] Ken Kennedy, Charles Koelbel, and Hans Zima. The Rise and Fall of High Performance Fortran: an Historical Object Lesson. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 7–1–7–22, New York, NY, USA, 2007. ACM.

[26] Kurt Keutzer, Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. A Design Pattern Language for Engineering Parallel Software: Merging the PLPP and OPL Projects. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, ParaPLoP '10, pages 9:1–9:8, New York, NY, USA, 2010. ACM.

[27] Fredrik Berg Kjolstad and Marc Snir. Ghost Cell Pattern. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, ParaPLoP '10, pages 4:1–4:9, New York, NY, USA, 2010. ACM.

[28] Rob V. Van Nieuwpoort, Jason Maassen, Gosia WrzesiÅĎska, Rutger Hofman, Ceriel Jacobs, Thilo Kielmann, and Henri E. Bal. Ibis: A Flexible and Efficient Java-based Grid Programming Environment. In *Concurrency and Computation: Practice and Experience*, pages 7–8, 2005.

[29] Version Rishiyur, Rishiyur S. Nikhil, and Rishiyur S. Nikhil. ID Language Reference Manual, 1991.

[30] K.H Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill, 4th edition, 1999.

[31] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, Cambridge, MA, USA, 2004.

[32] Konstantinos Sagonas, Terrance Swift, and David S. Warren. XSB as an Efficient Deductive Database Engine. *SIGMOD Rec.*, 23:442–453, May 1994.

[33] Lawrence Snyder. The Design and Development of ZPL. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 8–1–8–37, New York, NY, USA, 2007. ACM.

[34] P.W. Trinder, K. Hammond, H.-W. Loidl, and S.L. Peyton Jones. Algorithm + Strategy = Parallelism. *JOURNAL OF FUNCTIONAL PROGRAMMING*, 8:23–60, 1998.

[35] R. L. Wainwright. Parallel Sieve Algorithms on a Hypercube Multiprocessor. In *Proceedings of the 17th conference on ACM Annual Computer Science Conference*, CSC '89, pages 232–238, New York, NY, USA, 1989. ACM.

[36] Hans Zima, Mary Hall, Chun Chen, and Cha Jaqueline. Model-Guided Autotuning of High-Productivity Languages for Petascale Computing. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, HPDC '09, pages 151–166, New York, NY, USA, 2009. ACM.