The Simulation and Visualization of Parallel BMF Code

Paul Martinaitis

Department of Computer Science
University of Adelaide

SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE HONOURS DEGREE IN COMPUTER SCIENCE

November 1998

Rev 1.1

ABSTRACT

This project concerned the design and implementation of a tool which would enable the simulation and subsequent visualisation of parallel BMF code on a "virtual" architecture. This thesis will begin by giving a brief review of the wider context in which the project was conducted. The Bird-Meertens Formalism (or BMF) will be introduced from a programmers point of view and indications of the way in which BMF code may be parallelized through an automatic transformation process will be discussed. The design and construction of the simulator itself will then follow - along with the visualization methodology used. Results of the simulator will be presented as well as some extension made to the network module which facilitate the modeling of congestion.

ACKNOWLEDGEMENTS

I would firstly like to wholeheartedly thank my supervisor Brad Alexander for all his help and patience during the year and for introducing me to such an interesting area of study. I would like to also thank my parents who have always supported me through out my studies and were there to help when needed. My friends, especially Paul and Peter also deserve special mention for their support and good times.

TABLE OF CONTENTS

TABLE OF FIGURES

1. Introduction

Over the years, few areas of computer science research have shown as much promise in terms of increased performance and expansion in the domain of solvable problems than that of highly parallel computation. To date, a large number of innovative parallel architectures have been proposed with many of them having actually been constructed. Yet, in spite of this progress, few of these architectures are widely used in industry and the wider research community. A major reason for the under-utilization of these architectures lies not with the hardware as such, but rather with the high costs involved with the development and maintenance of the associated software. The difficulties with the software arise largely because in many cases, the programming models used on such machines are too architecturally specific; this increases the semantic gap between the high-level description of a problem and its implementation. Such a disparity not only increases the burden on the programmer but, more significantly, also makes the re-targeting of a program to different hardware platforms quite difficult. It is this latter point that is most troublesome because, for any given problem domain, the current best performing architecture at any given time can change quite rapidly and application software must be modified to keep up with such changes; it is this necessity to rewrite software for each new architecture that makes maintenance of parallel software so expensive. Clearly then, to facilitate the easier development of software that is more amenable to transfer between a variety of different architectures, less architecture specific models of computation are required.

1.1 Abstract Models of Parallel Computation

Skillicorn [17, 18] gives several key characteristics that a suitable parallel computational model should have if it is to be of greatest use. Firstly, the model needs to be sufficiently architecturally independent that it can be mapped to any of the commonly available parallel architectures. Secondly, the model should be Congruent; meaning that the complexity of a program at the model level should map in a 'linear' way to its implementation on the target machine. Finally, the model should be cognitively simple so that the programmer is insulated from the burden of dealing with low level aspects of the architecture such as synchronization, communications, topology etc.: such abstraction greatly increases productivity and code readability at low levels of parallelism and becomes a necessity with massive parallelism which would otherwise become impossible to reason about effectively.

1.2 The Bird-Meertens Formalism

Quite a number of models for parallel computation have been proposed, each necessarily being a compromise between the above-mentioned factors as well as a number of others such as expressiveness and ease of implementation[17]. The so-called data-parallel models[1] are considered to be among the simplest in terms understandability from the programmer's point of view as well as generally satisfying the criteria discussed above. The Bird-Meertens formalism[1, 2, 3, 8, 9, 15, 16, 17, 18, 19] or BMF is one such model that has a firm theoretical basis in categorical data types and their attendant operations.

This foundation gives it a number of additional useful properties including:

- Predictable computational patterns which assists both in the efficient implementation of BMF constructs and also in the reasoning about code behavior.

- Low minimum requirements on the underlying architecture resulting in lower implementation complexity and greater portability.

- Predominantly local communication patterns which serve to greatly increase the efficiency of implemented code.

- A completeness property which guarantees that any two equivalent computational forms (programs) may be transformed from one into another by the application of a finite number of algebraic steps.

[1] A data-parallel model is one which achieves parallelism through the application of the same instructions, simultaneously, over a set of data[5].

1.2.1 Completness

It is the completeness property which is perhaps the most significant feature of BMF as a programming model. What this property allows one to do is to take an initial program that is close to its specification and transform it into one that is computationally equivalent, but more efficient: because this process is a mathematical one, correctness of the process is assured. Furthermore, completeness guarantees that if an optimally efficient form of a program exists, then it can be reached in a finite number of steps: this combined with the fact that any step is reversible, means that the optimization of a program can be reduced to the traversal of a finite, albeit potentially very large, search space. Unfortunately it turns out that for any non-trivial program, the search space becomes intractably large thus precluding the use of blind search strategies.

1.3 Automated Optimization

As discussed above, expressing a program in BMF provides a sound basis for transforming a program from one form to another without affecting correctness. Such transformations are normally carried out with the aim of optimizing program performance. Performance enhancement is achieved by a combination of optimization of sequential code, cf. the data movement optimizer in the Adl project[3,4], and parallelization, or partitioning, of the code by exploiting the natural parallelism present in the BMF operators. At the present time, the large search space means that optimizations are performed by hand, particularly in the case of parallelization; current automatic optimization systems typically make use of a limited set of generally applicable transformations, however, it is hoped that in the future more sophisticated heuristic search techniques can be developed which will narrow the search space and make use of automatic optimization and parallelization more attractive.

Although the BMF model is reasonably architecturally independent, it must ultimately be implemented on a variety of real architectures, each having different physical characteristics. Such architectural differences will affect the degree to which particular program features and transformations impact upon program performance. For example, a system using a cross-bar network would generally experience substantially less congestion than, say, a binary tree. As a result when developing the heuristics, it is necessary to use knowledge of relationships between architectural parameters and transformations to steer the search in the ‘direction’ that will have the greatest probability of yielding the most effective optimizations on that particular architecture. At the present time, the required architecture specific information, available to guide us, is minimal; it is this gap in knowledge that the project attempts to go some way towards addressing.

1.4 Role of the Project

This project aims to the design and implement a tool which simulates the execution of parallel BMF code on a virtual architecture and allow subsequent visualisation, of an execution trace, using the Paragraph package. It is intended that such a tool will be used as an experimental platform to facilitate the exploration of the relationships between various architectural parameters and program transformations without the expense of actually of porting a full BMF system to each hardware platform. This investigation is carried out with the aim of increasing the knowledge base of architecture specific information which ultimately assists in the development more sophisticated heuristics leading to a greater use of automated optimization.

1.5 Project Overview

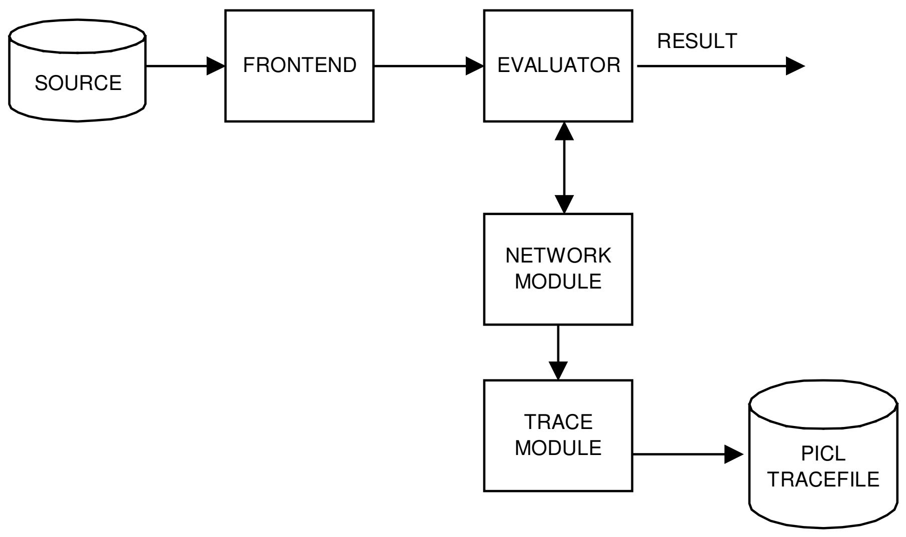


Figure 1-1: Logical Overview of the Simulator

Figure 1-1 shows the logical structure of and relationships between the major components of the simulator constructed in this project. A brief conceptual view of the way in which the simulator works is as follows: the Parallel BMF code is read in from an ASCII file, scanned and parsed by the Frontend and then sent to the Evaluator. The Evaluator simulates the parallel execution of the code producing the result (output) of the program as well as interacting with the Network Module to generate a PICL tracefile. Finally, Paragraph is used to visualize the execution of the program using the tracefile.

1.6 Overview of this Thesis

Chapter 2 introduces BMF 'source' code from an operational viewpoint, discussing the constructs available and how BMF programs may be written. Parallel BMF and the parallelization process will also be discussed. The details of the visualization methodology used by the simulator will be covered in chapter 3. In particular, the PICL tracefile format which may in some sense be regarded as the target code produced by the tool will be introduced and its relationship to the visualisation package Paragraph discussed. Chapter 4 will then look at the details of the simulator itself including how the parallelism and the interconnection network of the virtual architecture are simulated. The fundamental parallel operators of BMF will be examined in chapter 5 with visualisation results produced by the simulator being used to illustrate the nature of the operators.

Towards the end of the project, some extensions were made to the interconnection network model of the virtual architecture to take account of various different network topologies and congestion. Chapter 6 will outline how the more sophisticated modeling was carried out before highlighting the difference that the more realistic model makes to the visualisation results. The effects that different architectures can have on program performance will then be examined by way of some examples.
Chapter 7 will summarize the results and discuss the conclusions reached before outlining some future research directions that have arisen as a result of the work undertaken in this project.

2. The Bird-Meertens Formalism

2.1 Introduction

The Bird-Meertens Formalism (or BMF[8, 9]) is a theoretical framework, based on categorical data types, which enables the development of a well defined calculus to facilitate program transformation; such a theory has been built for a number of data types including: cons lists, concatenate lists, trees, arrays and bags[8, 14, 19, 20]. The BMF theory of concatenate lists, used exclusively in this project, is particularly well developed and its operations have potentially parallel implementations; this would not be possible with inherently sequential data types such as cons-lists. There are a number of different levels at which BMF can be discussed ranging from its theoretical underpinnings in Category theory through to its use as a practical programming language. In this chapter, the latter approach will be taken by providing an overview of the BMF source code used by the simulator whilst also introducing the notion of parallel BMF code.

2.2 Syntax

BMF, being a largely theoretical language, has no universally defined standard notation in which programs may be expressed; virtually any, practical, internally consistent notational system may be used. The syntactical form chosen to express the BMF source code for this project is essentially the same as that produced by the Adl compiler thus allowing automatically generated code to be fed straight into the simulator. Appendix A contains the definition of the syntax for the source code expressed in Extended Backus Naur Form (EBNF).

2.3 BMF Programs

A BMF program consists of a BMF expression juxtaposed with a single argument: in keeping with the functional nature of BMF, the argument encapsulates the entire input state of the program. The BMF expression in the program is either a single function or, more usually, a composition of functions. For example

```
B_Id  1
```

is a very simple program which applies the identity function to the argument literal 1. A more complex program, involving a number of operations would need to use the composition operator, B_Comp:

```
B_Comp ($Increment) (B_Id)  1
```

This program would first apply the identity function to the argument, **1**, and then apply the **Increment** function to the result of the identity function yielding **2**.

The simulator also has a library facility which allows a collection of pre-defined functions to be created and named for later use. Such functions are defined by using the following syntax:

```
$<function_name> = <BMF_Expression>
```

where <function_name> is the name of the function and <BMF_Expression> is the definition. To use the newly defined function in subsequent code, it is referenced by its name, prefixed with a dollar symbol. As an example, the following function could be defined thus:

```
$Trivial = B_Comp ($Increment) (B_id)
```

and then referenced by using $Trivial. When such a predefined function is used, the semantic meaning is equivalent to textual substitution of the function reference by its defining BMF expression. As such, the library mechanism is a facility to assist in the development of programs and in no way affects the referential transparency of the BMF code. That is to say, once a library function is assigned to a name, the same function will always be associated with that name; there is no dynamic reassignment or levels of scope.

2.4 Data Types

The following data types define the entities that may be used as input to, and appear as output from, BMF programs. The simple data types are: Integers, Real Numbers and Booleans.

2.4.1 Lists

A list or vector is an ordered collection of zero or more items of the *same* type. Syntactically, a list is a comma separated list of items enclosed by square brackets. i.e.

$$[a_1, a_2, \dots, a_n]$$

The empty list is denoted by [].

2.4.2 Tuples

Tuples are a collection of one or more items that may be of different types. Syntactically, a tuple is a comma separated list of items enclosed by braces. i.e.

$$\{a, b, c\}$$

where a, b, c can be different types.

2.5 Program Constructors

2.5.1 Composition

In BMF, a program consisting of a sequence of operations is expressed as a composition of functions. The B_Comp operator, introduced previously, provides this facility. From an operational viewpoint, a composition of two functions takes the output of the first function and pipes it as input into the second function. In more precise terms, if we have the functions F and G applied to an argument, x, then:

$$\text{B\_Comp} (F) \ (G) \ x \ \Leftrightarrow \ F (G (x))$$

Expressing composition in this way can make programs more difficult to read. An infix '.' is often used instead of B_Comp when describing programs.[2]

$$F.G \Leftrightarrow \text{B\_Comp} (F) (G)$$

2.5.2 All-Applied-To

As well as composition, another means of assembling functions in a program is by the use of the all-applied-to operators. There are two versions, B_Allvec and B_Alltup which use lists or tuples respectively.

B_AllVec

This operator takes a list of functions, applies them to individual copies of the input argument, and returns a list containing the results. If f_n defines a function and x is an argument then:

$$\text{B\_Allvec} [f_1, f_2, \ldots , f_n] \ x \ = \ [f_1(x), f_2(x), \ \ldots, f_n(x)]$$

[2] The two forms will be used interchangeably in this chapter, for the sake of clarity, but the infix form cannot be used as input to the simulator.

B_AllTup

B_Alltup is very similar in operation except that it uses tuples instead of lists. Once again, if f_n defines a function and x is an argument then:

$$\text{B\_Alltup } [f_1, f_2, \ldots, f_n]\ x \ = \ \{f_1(x), f_2(x), \ldots, f_n(x)\}$$

Notice that both B_Alltup and B_Allvec use square brackets to denote the input list or tuple respectively.

2.6 Constants

To generate an integer, real or boolean constant in a program, the B_Con function is used. For boolean constants, B_con takes the arguments B_True or B_False. For numerical constants, the value must be prefixed with the appropriate type identifier: either B_Int or B_Real.

2.7 Operators

The operations implemented in the source code have been broken up into two main groups: those appearing in the grammar under <b_exp>, called expressions, and those under <b_op>, sometimes informally referred to as 'operators'. The latter group, when used, are prefixed with the '**B_Op**'. The constructs appearing under <b_op> can be conceptually divided into two groups, Unary or Binary 'Operators', depending on whether they operate on a single argument or a single tuple of two arguments (i.e. a pair) respectively[3].

2.7.1 Binary 'Operators'

These operators all require a 2-tuple of arguments on which to operate. They can be divided into three main categories:

[3] The choice of which operators appear under B_Op is largely based on historical considerations and may be varied in the future.

Arithmetic

These operators provide the standard arithmetic operations and are denoted with the self-explanatory names of: B_Plus, B_Minus, B_Times and B_Divide. In this implementation, these operators are overloaded so that they may be used, transparently, with any combination of real and integer pairs. s an example of how a binary operator is used:

B_Op (B_Plus) {a, b}

would yield the result 'a+b'.

Boolean and Comparsion

All the standard boolean and comparison operators are provided:

Boolean Operations:

B_Neg, B_And, B_Or

Comparison Operators: (with their meaning shown to the right)

B_Eq	=
B_Gt	>
B_Lt	<
B_Le	<=
B_Ge	>=

These operators are also overloaded so that they may be used with any combination of integer and real arguments.

Aggregate Operators

B_Index

B_Index takes a tuple containing a list and an integer, as input, and returns the list item indexed by the integer.

B_Op(B_Index) { [A_0, A_1, …, A_I … A_N], I } = A_I

B_Project

P_Project re-orders a list according to a vector of index values thus allowing an arbitrary permutation of the list item to be performed. The first argument is the list to be projected. The second argument contains a list of indices: the index at each position in this list indicates which item of the input list is to go in that position.

B_Op(B_Project) { [A_0, A_1, …, A_N], [I0, I1, …, IN] } = [A_{I0}, A_{I1}, …,A_{IN}]

B_Distl

B_Distl takes in a pair consisting of a data item and a vector and distributes the data item over the vector in a pair-wise manner.

B_Op(B_Distl) {X, [A_0, A_1, …, A_N]} = [{X, A_0}, {X, A_1}, …, {X, A_N}]

where X can be any data item including another list or tuple.

2.7.2 Unary Operators

B_Neg

This operator performs negation on a boolean or numerical argument.

B_Length

Takes a list as an argument and returns its length.

B_Iota

B_Iota takes an integer argument, N, and returns the list [0, 1, 2, …, N-1]

2.7.3 Aggregate Operators over Lists

B_Map

Map applies a function to every element of the input list. So if *f* is a unary[4] function then map can be defined as:

$$B\_Map(f)\ [A_0, A_1, \ldots, A_N] = [f(A_0), f(A_1), \ldots, f(A_N)]$$

B_Reduce

Reduce takes in an associative operator and default value and then effectively inserts the operator between the list items. The operator is then applied thus folding the list back to a single value. If $\oplus$ is an associative operator and X the default value then:

$$B\_Reduce\ (\oplus)\ (X)\ [A_0, A_1, \ldots, A_N] = X \oplus A_0 \oplus A_1 \oplus \ldots \oplus A_N$$

and:

$$B\_Reduce\ (\oplus)\ (X)\ [\,] = X$$

The Reduce implemented here, and that implemented in the Adl project[5], makes no attempt to verify the associativity of the operator used. Additional forms of Reduce can be defined (as in the Adl project), such as directed left and right reductions[3, 8] which can use associative or non-associative operators - such variations are, however, inherently sequential.

Zip

Zip operates on a pair of lists merging them, in a pair-wise fashion, using a binary operator. If $\oplus$ denotes a binary operator then:

$$B\_Zip\ (\oplus)\ \{\ [A_0, A_1, \ldots, A_N], [B_0, B_1, \ldots, B_N]\ \} = \\ [\ (A_0 \oplus B_0), (A_1 \oplus B_1), \ldots, (A_N \oplus B_N)]$$

Scan

[4] By 'unary' in this context, we mean a function that operates on a single argument: eg. B_Neg, $Increment etc. - NOT a function that takes a single tuple of two arguments like, say: {a,b}

[5] In the Adl project, this type of reduce is called 'B_ReduceP'

Scan is also known as prefix or accumulate. From an operational viewpoint, scan can be viewed as the mapping of a reduction over the initial sub-lists of the input. That is, if $\oplus$ denotes an associative operator then:

$$\begin{aligned} &\text{B\_Scan}(\oplus)\ [A_0, A_1, \ldots, A_N] \\ &\quad\Leftrightarrow \text{B\_Map}\ (\ \text{B\_Reduce}\ (\oplus)\ )\ \ [\ [A_0], [A_0, A_1], \ldots, [A_0, A_1, \ldots, A_N]\ ] \\ &\quad= \ [\ A_0, (A_0 \oplus A_1), \ldots, (A_0 \oplus A_1 \oplus \ldots \oplus A_N)\ ] \end{aligned}$$

Split

The Split operation takes in an integer argument, P, and transforms the input list into a list of P sub-lists.[6] In performing the split, the data is partitioned as evenly as possible to make the sub-list lengths uniform.[7] As an example, if P = 3 and list length is divisible by P then all the sub-lists will be of equal length:

B_Split (B_Num 3) [1, 2, 3, 4, 5, 6] = [[1, 2], [3, 4], [5, 6]]

Alternatively, if the list length is not divisible by P then the first sub-list will be one item longer than the rest:

B_Split (B_Num 3) [1, 2, 3, 4, 5, 6, 7] = [[1, 2, 3], [4, 5], [6, 7]]

which still yields the most even distribution possible.

2.8 Parallel BMF Code

On of the primary advantages of BMF is that it enables the expression of a wide variety of parallel algorithms through the use of a small number of fundamental constructs: the parallel constructs are, in fact, data-parallel interpretations of serial aggregate operations, namely: Split, Map, Reduce, Scan, Zip, Distl and Project.

[6] It Should be noted that the split operation described here is quite distinct from the version introduced by Bird in {[8] p171] which splits a list into its head and tail components.

[7] Chapter 5 will describe the partitioning algorithm in detail.

Having relatively few, but highly expressive, parallel constructs not only reduces the cognitive burden on the programmer, but also increases portability because there are fewer parallel operations that need to be specifically implemented on the target architecture.

The parallel versions of the aggregate operators are distinguished from their serial counterparts by being prefixed with a 'P'. The list is the fundamental data-structure in parallel BMF, it being the only one that can be distributed across a machine.[8] In the following sections, the processor on which a particular list element resides will be labeled by a superscript; this is done to illustrate the operation of parallel BMF code but such information is not required by the programmer in designing the code.

From the programmer's point of view, the parallel versions of the aggregate operations have the same semantic meaning as their serial counter-parts except that the operations occur over distributed data; as a result, although the programmer has to use the appropriate operations, details of how the data is mapped and distributed onto the target machine are completely hidden. That is to say, a BMF program has the same semantic meaning whether it is expressed in terms of serial or parallel constructs. To illustrate this point, and to introduce the nature of parallel BMF program, consider the following example:

If we start with the list [1, 2, 3, 4, 5, 6] and split it three ways:

B_Split (B_Num 3) [1, 2, 3, 4, 5, 6] = [[1, 2], [3, 4], [5, 6]]

we can now increment each list item by mapping the map of the increment function over the list.

B_Map (B_Map ($Increment)) [[1, 2], [3, 4], [5, 6]]
= [[2, 3], [4, 5], [6, 7]]

On the other hand, if we were to distribute the input list over the machine using P_Split:

[8] We will see later that the P_Distl operator can distribute a scalar across the machine but only across an already distributed vector.

P_Split (B_Num 3) (B_Num 0) [1, 2, 3, 4, 5, 6]
= [$[1, 2]^{0}$, $[3, 4]^{1}$, $[5, 6]^{2}$]

where each sub-list is now on a different processor, as indicated by the superscripts, we would use a parallel map of the serial map of the increment function

P_Map (B_Map ($Increment)) [$[1, 2]^{0}$, $[3, 4]^{1}$, $[5, 6]^{2}$]
= [$[2, 3]^{0}$, $[4, 5]^{1}$, $[6, 7]^{2}$]

where the parallel map has the same semantics as the serial one except that it can operate on a distributed list.

As can be see from this example, the only difference between the two programs is that the outer map is either the serial or parallel version, as appropriate.

2.9 Parallel Operators

Having now seen an example of parallel BMF code, this section will describe, in qualitative terms, the parallel operators available in this implementation. A more detailed and precise account of their operation, and the trace results that they produce, will be given in Chapter 5.

2.9.1 P_Split

The split operation is the primary means by which data is spread across the machine. Like its serial counter-part, P_Split partitions an input vector into a list of sub-lists except that each sub-list is sent to a different processor. P_Split takes in two integer arguments, the first indicates the number of processors over which to spread the data and the second sets the stride (the distance between two adjacent processors). The stride argument is $\log_2$ of the actual stride; in this way, the stride is quantized to a power of two. The stride refers to the difference in processor id's between adjacent list nodes after the completion of the split operation[9]. For a three way split with a stride of 1 ($1 = 2^0$) :

P_Split (B_Num 3) (B_Num 0) [[A_0, A_1, …, A_N]]
= [[A_0, ..., A_A] ^{0}, [A_{A+1}, …, A_B] ^{1}, [A_{B+1}, …, A_N] ^{2}]

2.9.2 P_Map

P_Map is semantically equivalent to the serial version except that it can operate on a distributed list. Map is particularly efficient in the sense that it does not involve any data exchange between the processors.

2.9.3 P_Reduce

P_Reduce can be viewed (informally) as being the opposite of the P_Split operation because it allows a distributed list to be collected back together onto a single processor through the use of an associative operator[10]. From the programmer's point of view, it works in exactly the same way as B_Reduce except that no default value is required[11]. P_Reduce is commonly used with the list concatenate operator, B_Conc to gather a split list of sub-lists back to a single processor. For example taking the distributed list from the last example, we could gather it back to processor zero again by:

P_Reduce(B_Op (B_Conc) [[2, 3] ^{0}, [4, 5] ^{1}, [6, 7] ^{2}]
= [2, 3, 4, 5, 6, 7] ^{0}

The associativety of the operators used with P_Reduce allow the evaluations to occur in an arbitrary order without affecting the result: this fact permits P_Reduce to conduct many of the evaluations in parallel thus allowing the operation to be completed in $\log_2 N$ operations where N is the number of items in the list.

---

[9] The stride argument, and how it is used to facilitate nested splits will be described in section 5.3.

[10] Reduce with concatenate is often used after the parallel processing of a split list - such an operation is *not* the inverse of split, however, because the information regarding the lengths of the sub-lists is lost.

[11] This is assuming that there are at least two processors over which to reduce

2.9.4 P_Scan

P_Scan, like P_Reduce, operates on a distributed list except that the results is still a distributed list. As in the case of P_Reduce, the associativity of the operator used allows the development of an algorithm that can perform the scan in $\log_2 N$ steps.

2.9.5 P_Zip

P_Zip, like B_Zip, merges a pair of lists in a pair-wise fashion, using a binary operator. The two lists are distributed and must be conformable which is to say, they must be the same size and mapped across the same set of processors; this ensures that the two components of each 'evaluation pair' reside on the same processor so the 'zipping' operation can occur in parallel with no inter-processor data transfer. If $\oplus$ denotes a binary operator and A and B are a pair of distributed lists then:

$$\begin{aligned} &\text{P\_Zip}\ (\oplus)\ \{\ [A_0{}^0, A_1{}^1, \ldots, A_N{}^N], [B_0{}^0, B_1{}^1, \ldots, B_N{}^N]\ \} \\ &\qquad = [\ (A_0 \oplus B_0)^0, (A_1 \oplus B_1)^1, \ldots, (A_N \oplus B_N)^N\ ] \end{aligned}$$

Notice that each pair of list items in parenthesis reside on a separate processor so the evaluations can occur in parallel.

2.9.6 P_Distl

P_Distl is used to broadcast an item from one processor to many by sending the item to each node of a distributed list. Let X be any data item and A be a distributed list, then:

$$\begin{aligned} &\text{P\_Distl}\ \{\ X^0, [A_0{}^0, A_1{}^1, \ldots, A_N{}^N]\ \} \\ &\qquad = [\ \{X, A_0\}^0, \{X, A_1\}^1, \ldots, \{X, A_N\}^N\ ] \end{aligned}$$

2.9.7 P_Project

P_Project re-orders a distributed list according to a vector of index values. Whenever a list item changes position, an inter-processor data exchange will occur. For example:

$$\text{P\_Project } \{ [0, 1, \ldots, N], [A_0^{\,0}, A_1^{\,1}, \ldots, A_N^{\,N}] \} = [A_0^{\,0}, A_1^{\,1}, \ldots, A_N^{\,N}]$$

would produce no data exchange because the position of each list element remains unchanged whereas:

$$\text{P\_Project } \{ [A_0^{\,0}, A_1^{\,1}, \ldots, A_N^{\,N}], [N, N\text{-}1, \ldots, 1, 0] \}$$
$$= [A_N^{N}, A_{N-1}^{N-1}, \ldots, A_1^{1}, A_0^{0}]$$

affects a complete reversal of the list and will involve N data-exchanges.

2.10 Parallelisation of Serial BMF Code

In addition to providing a solid basis with which to express parallel algorithms, the formal framework of BMF provides a mechanism with which to parallelize serial code using a mathematical transformation process. The automation of such a process, as indicated in Chapter 1, appears to be intractably difficult except in the case of small, simple programs, or in programs displaying general, easily recognisable features. This section will give a brief introduction to the parallelisation process for some very simple BMF code to give some idea of the techniques involved.

Consider the problem of finding a series of factorials from Factorial(1) through to Factorial(N). Assuming that the Factorial function has already been defined, the following code would produce the required series:

```
B_Map ($Factorial)  [1, 2, 3, … N]
```

To transform this serial program into a parallel version over P processors, the following BMF identity will be made use of:

```
P_Reduce ( B_Op ( B_Conc ) .  P_Split (B_Num P ) (B_Num 0 )
          = B_Id      {Split Property}
```

Informally, the Split-Property states that if a list is distributed using split and then reduced back again using concatenate, there is no net change in the data; that is, the net effect is equivalent to the application of the identity function.

Given this fact, a program can be composed with a split followed by a reduce with concatenate without affecting the result. Therefore the program becomes:

```
 B_Map ($Factorial) . P_Reduce ( B_Op ( B_Conc ) .
        P_Split (B_Num P ) (B_Num 0 )
```

A further property can now be made use of:

```
B_Map (F) . P_Reduce (B_Op (B_Conc) )                {Map-Promotion}
        = P_Reduce (B_Op (B_Conc) ) . P_Map ( B_Map (F) )
```

Applying Map-Promotion to the Factorial program yields:

```
P_Reduce (B_Op (B_Conc) ) . P_Map (B_Map ($Factorial) ) .
        P_Split (B_Num P) (B_Num 0)
```

which is the final parallelized version. This example showed how parllelization can be performed through the application of a series of algebraic identities, specifically the Split-Property and Map-Promotion. Clearly a program of this size can easily be parallelized by inspection but for more realistically sized programs, the parallelization process becomes much more difficult.

3. Visualisation Methodology

3.1 Introduction

Having described the nature of the BMF source code in the previous chapter, the discussion will now turn towards what can, in some sense, be regarded as the target code of the simulator; namely the PICL format tracefile and the associated visualisation environment. This chapter will describe the relevant features of the PICL format, how it is used to record events in this project, and the inter-relationships between PICL and the visualisation package, Paragraph.

3.2 PICL

PICL [22] stands for 'Portable Instrumented Communication Library' and as its name suggests, not only defines a tracefile format, but also a generic message-passing library. The idea is that programs can be written using the generic PICL routines, rather than platform specific commands; this results in code that is more portable because it can be compiled and run on any architecture on which the PICL library has been implemented. PICL also provides tracing facilities so that any program that makes use of the generic routines can produce trace information automatically without having to modify its source code; this has the obvious advantage of eliminating the dangers involved with the manual insertion of instrumentation code throughout a potentially complex program. In this project, only the trace format of PICl is used as the trace events are generated directly by the simulator.

3.2.1 PICL Tracefile Format

The tracefile format provides a very general and extensible framework with there being a number of pre-defined trace events as well as facilities which enable customized 'user' events to be added. Some of the pre-defined events include categories such as: Inter-Process communications, Synchronization, File I/O and Resource Allocation. In this project, however, only a sub-set of the available events was required and will be described here. The PICL Documentation[22] contains the details for all the PICL events.

3.2.2 Basic Trace Record Structure

The tracefiles are written as an ASCII file with one trace record per line. Each trace record consists of a sequence of space separated ASCII fields all of which must appear on the same line. There are six basic fields which all records must contain:

1. **Record Type:**

   This field is either '-3' indicating the entry (beginning) of an event or '-4' which denotes the exit (end) of an event.

2. **Event Type:**

   This indicates the type of event that is being traced. The events that are used in this project are:

-21	Non-blocking send
-51	Blocking receive
-601	Processor has become idle.

   A more detailed description of these events will appear below

3. **Time Stamp**

   A floating point number which indicates the time at which the event occurred.

4. **Processor Id**

   The Id of the processor on which the event occurred

5. **Process Id**

   The process in which the event occurred. At present, neither PICL nor Paragraph support more than one process per processor. As a result, this field is always '-1' which refers to all processes on a given processor. (So in this case, -1 refers to the one and only process).

6. Number of Event Specific Data Fields

This field is an integer indicating how many event specific data fields follow. The number and nature of these fields is dependent on the *event type*; they are described in the next section.

When there are one or more event specific data fields (i.e. Field six is greater than zero), the following data descriptor is present to indicate the type of data contained in them:

Data Descriptor

The available types and the associated designator are:

0	Character
1	String
2	Integer
3	Long Integer
4	Single Precision Floating Point
5	Double Precision Floating Point

In this project, only integer data is required and so this field is always set to ‘2’
After the data descriptor, the event specific data fields follow.

3.2.3 Events and Event Specific DataFields:

The PICL events used in the project along with the interpretation of the specific data fields (if any) are shown below. The numbers is parenthesis are first two basic fields which identify the event.

Start Non-Blocking Send (-3 -21)

This event corresponds to a processor beginning to move data into the network; as such, any time between this event and event (-4 -21) is considered to be communications overhead.

Data Fields:

- *Message Length in Bytes*: The length of the message sent.

- *Type*: The type of the message expressed as an integer. This field allows different types of messages such as control versus data packets to be distinguished. In this implementation, the type is always ‘1’.
- *Destination Processor ID*: The processor to which the message is sent.

End Non-Blocking Send (-4 -21)

This event occurs when the transfer of data into the network, started by event (-3 -21), has been completed. This event has no extra data fields.

Start Blocking Receive (-3 -51)

Denotes a processor waiting for the arrival of a message. This is a blocking receive and so all the time between this event and the reception of the message (event -4 -51) is counted as communications overhead.

Data Fields:

- *Requested Type*: The type of message that is expected to be received.(Always ‘1’ in this implementation).

End Blocking Receive (-4 -51)

Denotes the arrival of the message which was initiated by event (-3 -51).

Data Fields:

- *Message Length*: Number of bytes received.
- *Type of message received*: Always ‘1’ in this implementation.
- *Source Processor*: Processor from which the data was sent.

Start Processor Idle (-3 -601)

This event occurs when the processor becomes idle.

End Processor Idle (-4 -601)

Occurs when the processor becomes Busy (not idle).

3.2.4 Example PICL Trace Records

Consider the following trace record:

```
-3  -21  4.500  4  -1  3  2  2  1  6
```

The first six numbers correspond to the basic fields described in section 3.2.2. They can be interpreted as:

-3 -51 : The event is the start of a send
4.500 : The Send began at time 4.500 seconds
4 : Processor 4 is sending
-1 : Process identifier which is always -1
3 : There are three event specific data fields following
2 : The type of the event specific data fields is Integer

The last three numbers are then the expected event specific data fields whose meanings are described in section 3.2.3

2 : The message being sent is two bytes long.
1 : The type of message being sent is one.
6 : The message is being sent to processor 6.

As another, simpler example:

```
-3  -601  5.000  12  -1  0
```

means that processor number twelve became idle at time 5.000 and conversely:

```
-4  -601  6.000  12  -1  0
```

indicates that processor twelve became busy (not idle) at time 6.000.

Notice that in this example, there are no event specific fields for event -601 as indicated by the zero in the last field.

3.3 The Trace Module

This section will describe the Trace Module which is the interface through which the rest of the simulator writes the tracefile. Figure 3-1 below, shows the logical structure of the simulator.

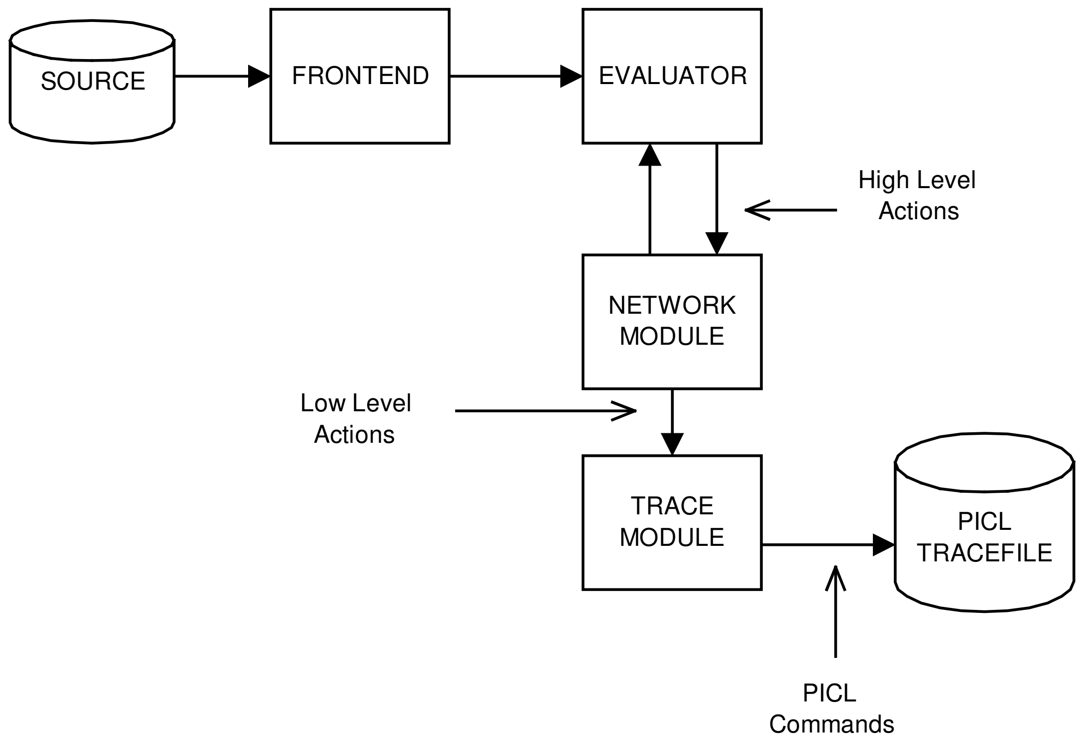


Figure 3-1: Logical Structure of Simulator

From a high-level point of view[12], the operation of the simulator can be briefly described as follows: the source code is read in by the front-end, parsed, and then feed to the Evaluator. The Evaluator simulates the parallel execution of the code, interacting with the network module whenever inter-processor communications or changes in processor state[13] occur through a set of 'High-Level' actions described in Chapter 5. The network module then, in turn, generates an appropriate sequence of

[12] The design and operation of the simulator will be discussed in detail in Chapter 5.

[13] That is, when a processor changes from being Busy to Idle or vice versa - see section 3.4.1 for more details.

low-level trace events which are converted to PICL and written to the tracefile by the Trace Module. The Trace Module provides an abstract interface to the PICL tracefile format by providing a procedural ‘wrapper’ for each trace event. Below is part of the Ada[7] specification for the Trace Module:

```
procedure Begin_Send (P_Id : Integer ; Time : Float ;
                      Size : Integer ; Ty : Integer;
                      Dest_Id : integer);

procedure End_Send (P_Id : Integer ; Time : Float);

procedure Begin_Recieve(P_Id : Integer ; Time : Float ;
                        Ty : integer);

procedure End_Recieve(P_Id : Integer ; Time : Float ;
                      Size : Integer; Ty : Integer ;
                      Source_Id : Integer);


procedure Start_Idle(P_Id : Integer ; Time : Float);

procedure Stop_Idle(P_Id : Integer ; Time : Float);
```

As can be seen, there is a procedure corresponding to each of the PICL events described in section 3.2. Each procedure simply takes in the required data as parameters and writes the appropriate ASCII sequence to the tracefile. For example the following call:

```
Begin_Receive(4, 6.000, 1)
```

would generate the following PICL sequence:

```
-3  -51  6.000  4  -1  1  2  1
```

and

```
Start_Busy( 2, 3.500 )
```

would generate:

```
-4  -601  3.500  2  -1  0
```

3.4 Paragraph

Paragraph[10, 11, 12] is a software package which provides facilities for the visualisation of trace information written in the PICL format. Although Paragraph was originally designed to visualize programs instrumented with the PICL library, there is no reason why it cannot be used with a PICL tracefile generated by some other means. This is the approach taken in this project where the PICL tracefile is generated by the simulator itself.

3.4.1 Events

The events occurring on a parallel machine which are relevant to Paragraph can be divided into two categories: Changes in the state of each processor and inter-processor communications.

Changes in Processor State

At any given time, each processing node of a parallel machine can be regarded as being in one of three mutually exclusive states: *busy*, *overhead* or *idle*. When Paragraph is used to visualize 'real' programs, a processor is said to be idle if waiting for the arrival of a message, in overhead if it is executing in the communications sub-system but not waiting for a message, and busy if doing anything else. So under this (default) scheme, a processor that is performing useful[14] computation and one which is occupied with non-communications overheads or sitting idle because it has nothing to do are all indistinguishable. In other words, a processor which is Busy has the *potential* to do useful computation, in that it is not occupied in the communications sub-system, but may not necessarily be doing so. Such a definition is used because the determination of whether a processor is doing useful computation or something else is considered to be too invasive[15].

[14] By useful, it is meant any computation which is required by the algorithm of the program.

[15] i.e. Such a measurement would cause an unacceptable perturbation and affect the accuracy of the overall tracing process.

In this project, as the tracefile is generated directly by the simulator, a more intuitive set of definitions can be used:

BUSY:
When the processor is actually performing useful computation.

OVERHEAD:
Time when the processor is processing communications overheads.

IDLE:
Anytime the processor is sitting idle either because it has
nothing to do, or is it waiting for a message to arrive from another processor.

The relationships between the three states and the PICL commands which cause the various transitions are summarized in Figure 3-2.

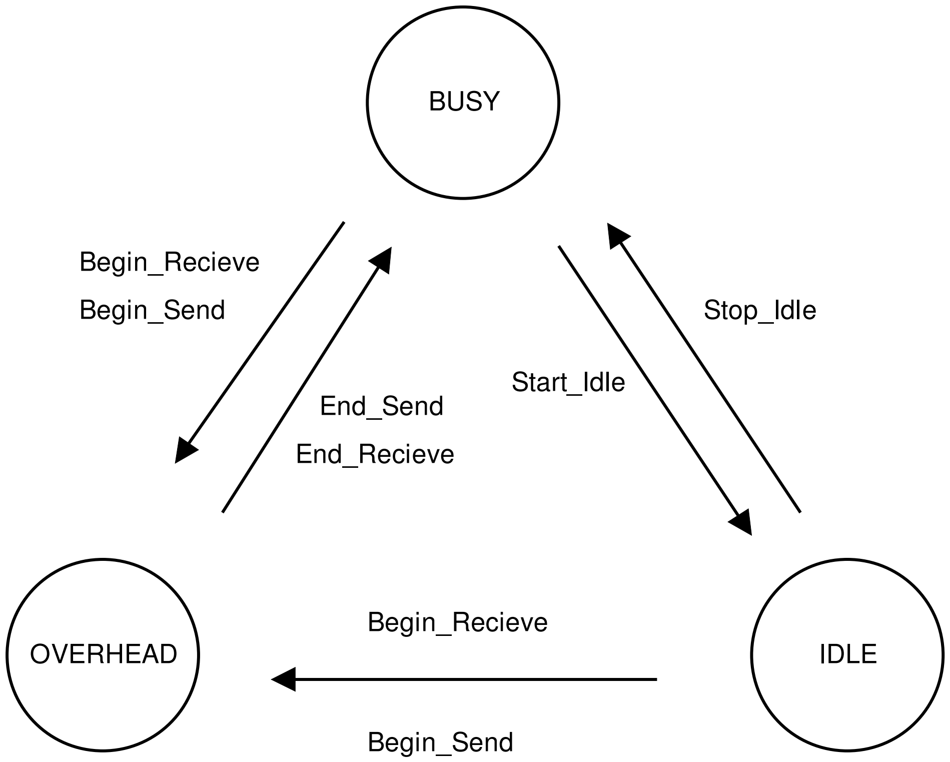


Figure 3-2: Processor State Transition Diagram

From the above diagram, it can be seen that to put the processor back into the Idle state, a Start_Idle command must be issued after every End_Receive, End_Send and at the completion of a computation.

Inter-Processor Communications

Inter-processor communications are characterized, in this project, by three phases:

Send

During this phase, the source processor transfers the data to be sent into the network buffer. This period is begun with a Begin_Send and ended with End_Send.

Transport

The transport time is the time taken for the message to travel through the network to the destination processor. During this phase, the destination processor remains idle and so no additional trace commands are issued.

Receive

Once the message has arrived at the destination processor, it is transferred from the network buffer. A Begin_Receive is issued at the start of the transfer and an End_Receive upon completion.

3.5 Paragraph Examples

To demonstrate how the events described in the previous section relate to the actual visualisation displays produced by Paragraph, consider the following, simple example:

> We have a four processor machine with the processor Id's running from zero to three. A single inter-processor send occurs, at time 1.0 from processor 0 to processor 2. Upon receiving the message, processor 2 then operates on the data for 5 time units. An appropriate sequence of trace events describing this situation could be:

```
start_Idle(0, 0.0);            {Set all Processors to idle
Start_Idle(1, 0.0);             state}
Start_Idle(2, 0.0);
```

```
4     Begin_Send(0, 1.0, 5, 1, 2);  {Initiate send from 0 to 2}
5     End_Send(0, 2.0);
6     Start_Idle16(0, 2.0);          {Enter idle state again}

      {The gap in time here represents the transport time
       through the network}

7     Begin_Recieve(2, 4.0, 1);     {Begin receiving data}
8     End_Recieve(2, 5.0, 5, 1, 0);
9     Stop_Idle(2, 5.0);            {Begin computation}
10    Start_Idle(2, 10.0);          {Finish computation}
```

If the resulting PICL tracefile for this example is fed into Paragraph, the following displays would be generated:

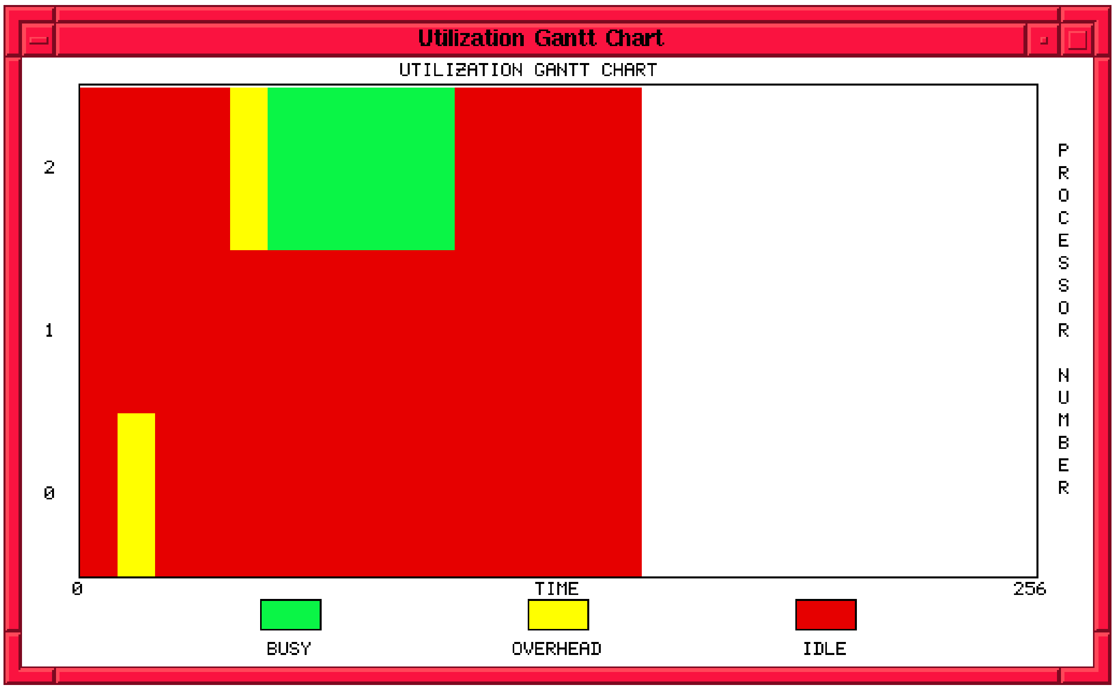


Figure 3-3: Example Gantt Chart

This display is called a 'Gantt Chart' and shows, at any given time, which of the three states each processor is in according to the colour. The correspondence between this display and the example can be seen as follows: all of the processor are in the idle state until processor 0 starts sending (line 4). Then there is a gap, in time, between

[16] Notice the Start_Idle here to prevent Paragraph from returning the processor to the Busy State. - see figure 3-2.

processor 0 finishing the send(line 5) and the message arriving at processor 2 (line 7). Processor 2 then finishes reading in the message(line 8) and enters the Busy state for five time units (lines 9 & 10).

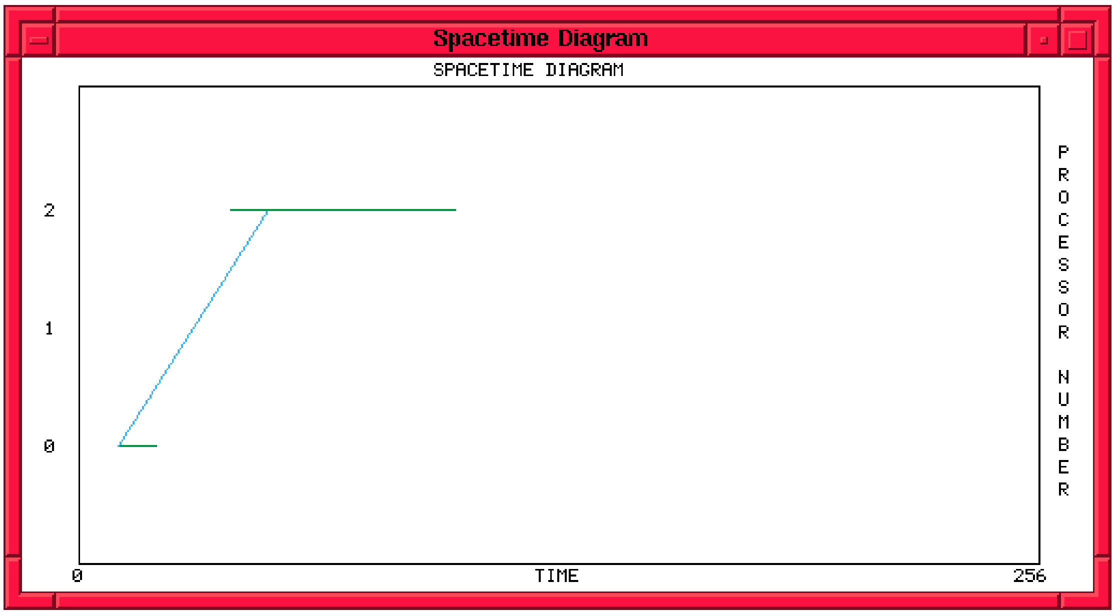


Figure 3-4: Example Space-Time Diagram

The Space-Time diagram is useful for visualizing the patterns of inter-processor communications. Whenever an inter-processor communication occurs, a line appears between the two processors starting at the beginning of the send (event -3 -21) to the completion of the receive (event -4 -51). Being a space-time diagram, for a given inter-processor distance, the steeper the line, the faster the communication was. The lines are also colour coded according to how large each message was. The horizontal lines correspond to times where the processor is either busy or in overhead. Looking at figure 3-4, the pattern shown clearly corresponds to that of the Gantt chart in figure 3-3; in particular, notice that the horizontal lines show the overheads involved in the send and receive with the line of processor 2 extending beyond the completion of the receive due to the extra Busy time.

Paragraph provides a number of additional displays which look at the same data, but from different point of view. Examples of the other displays will appear in chapters 5, 6 & 7 and their function will be explained there. The Paragraph manual[12] also provides a comprehensive overview of the available facilities.

4. The Parallel BMF Simulator

4.1 Overview

Figure 4-1 provides a logical overview of the simulator, separated into three areas. Having described the Source code and Visualisation Methodology previously, the present chapter will focus on the design and implementation of the core components of the simulator namely, those categorized in the diagram as Simulation & Evaluation. Together, these components encompass the entire functionality of the simulator save the final translation of the trace events into PICL which is performed by the Trace Module described in chapter 3.

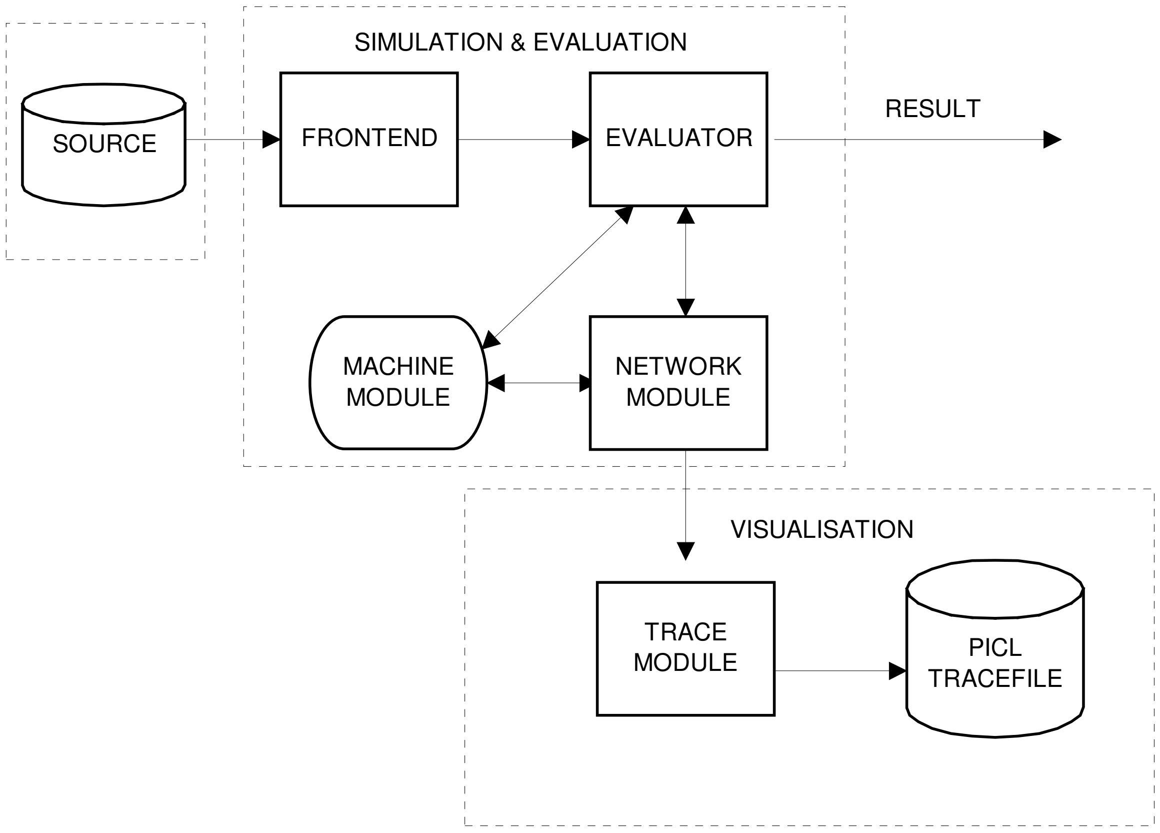


Figure 4-1: Logical Structure of the Simulator

4.2 Implementation of Datatypes

The data-types supported in this implementation can be divided into two main groups: simple types and aggregate structures. The simple types consist of integers, reals and booleans and the aggregates consist of the list and tuple structures.

4.2.1 Lists and Tuples

The representation of the list and tuple structures are very similar with both being implemented as a linked list of variant records [7]; the use of variant records allows lists and tuples of any (BMF) type to be represented. For example, consider the node definition of the list structure :

```
type List_Node(Selector: Lnode_Type := Integer_ty) is record
   Next : List := null;
   Proc_Id : Integer := 0;
   case selector is
      when list_Ty =>
       List_Val : List := null;
      when Tuple_Ty =>
       Tuple_Val : Tuple := null;
      when Integer_Ty =>
       Integer_Val : Integer := 0;
      when Real_Ty =>
       Real_Val : Float := 0.0;
      when Bool_Ty =>
       Bool_Val : Boolean := False;
   end case;
end record;
```

Figure 4-2: Ada List Node Definition

From this, it can be seen that through the choice of the appropriate selector, a list node can contain any of the BMF types including another list or tuple. A number of functions were written to work with and manipulate the tuple and list structures including, for example, functions to: find the length of a list, perform indexing, appending etc.

4.2.2 Values

To accommodate arbitrary BMF programs, the evaluator (described later) must be polymorphic in the sense that it can accept programs which may return any BMF type and also accept any BMF type as an argument. To achieve this, all BMF data both arguments and function results are wrapped within the following variant structure called a Value:

```
type Value(Selector: Lnode_Type := Integer_Ty) is record
   Proc_Id : Integer := 0;
    case selector is
       when list_Ty =>
        List_Val : List := null;
       when Tuple_Ty =>
       Tuple_Val : Tuple := null;
      when Integer_Ty =>
       Integer_Val : Integer := 0;
       when Real_Ty =>
       Real_Val : Float := 0.0;
       when Bool_Ty =>
       Bool_Val : Boolean := False;
    end case;
 end record;
```

Figure 4-3: Ada Value Specification

As in the case of list and tuple nodes, by the appropriate choice of the variant, Selector, a Value can contain data of any BMF type. It should be noted, that by using such a representation, any type checking that is performed has to be provided by each sub-evaluator[17] prior to the application of each operation. In this implementation, argument type checking is carried out for a few operations such as, for example, B_Length (checks for a list argument) and the boolean operators (checks for a tuple of booleans). It is anticipated that the simulator will primarily be used with automatically generated code which should already have appropriate typing thus negating the need, at this stage, for extensive argument checking to be performed.

[17] Section 4.5 covers the evaluator.

4.3 Front End

The 'front end' consists of the a scanner and a parser whose function is to convert the ASCII source code, representing a BMF program and its argument, into an abstract representation suitable for evaluation.

4.3.1 Scanner

The purpose of the scanner is to convert the ASCII source code into a stream of tokens which can then be processed by the parser. A token may be a 'reserved word' (such as B_Id, B_Map etc.), a special symbol such as a '(' or '[', or a literal value such as an integer, real or boolean.

4.3.2 Parser

The parser uses the stream of tokens from the scanner to construct an abstract syntax tree representing the program and in doing so, also checks the code for syntactic correctness. The design of the parser is based on the recursive descent method which is both easy to understand and implement, relatively efficient, and easily extendible. The recursive descent method fits in very naturally with the simple, non-ambiguous grammar of the source code[18]. On disadvantage of this parsing technique is that error recovery is more difficult than with table driven parsing but this is not a significant problem because it is anticipated that the simulator will work primarily with automatically generated code which should be correct.

Both the scanner and parser are sufficiently self-contained that they can be changed or modified to accommodate virtually any consistent form of source code notation without having to modify the rest of the system. For trivial changes such as, for example, changing the representation of the plus operation from 'B_Plus' to '+' , only the scanner needs to be updated. More extensive changes such as representing functional composition with an infix dot rather than the prefix 'B_Comp'[19] would involve modification of the parser as well.

[18] Looking at Appendix A, it is clear that the grammar is LL(1), which is to say that it can be deterministically parsed using only one look-ahead symbol.

[19] i.e. changing from the form B_Comp (A) (B) to A . B

4.4 Internal Representation of BMF Programs

As mentioned previously a BMF program, once parsed, is represented by an ‘abstract syntax tree’ or ‘parse tree’. The parse tree is implemented, literally, as a tree data structure with each node being polymorphic to accommodate the various BMF expressions (i.e. those constructs which appear under <b_exp> in the grammar). As with data type representations, the polymorphism of the parse tree is implemented by the use of a variant record as shown in figure 4-4. The variant component of the parse tree node takes the value of the BMF construct which the node represents.

```
type Ptree_Node(Selector : Pnode_Type := B_Id) is record

    case Selector is
       when B_Map | B_Zip | P_Zip | B_Comp | B_Reduce | B_Scan |
             B_Addr | P_Split | P_Reduce | P_Map |P_Scan =>

          C1 : Ptree := null;
          case Selector is
             when B_Comp | B_Reduce | B_Addr | P_Split  =>
                C2 : Ptree := null;
             when others => null;
          end case;

       when B_Alltup | B_Allvec =>
        C : PList;
       when B_Id | P_Distl | P_Project =>
        null;
       when B_Num =>
        Integer_Val : Integer := 0;
       when B_Con =>
        Constant_Value : Constant_Record;
       when B_Op =>
        Operator : Operator_Type;
    end case;
 end record;
```

Figure 4-4: Ada Specification of Parse Tree Node

Looking at the node specification, it can be seen that the BMF constructs can be divided into several categories according to what information is contained within their parse tree nodes:

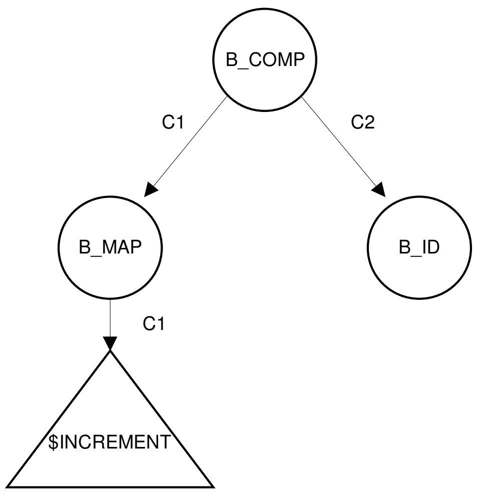
4.4.1 Higher Order Constructs

In the current context, 'higher-order' constructs are defined to be those which take one or more other BMF expressions as arguments with each such argument being represented by a separate child parse tree. It should be noted that the standard definition of a 'Higher Order' function is one which takes another function as its argument; this is slightly different to the definition used here because operators such as B_Addr have 'B_Num' nodes as children but such nodes do not represent functions. The number of child parse trees may be one in the case of constructs such as B_Map or two for constructs like B_Reduce. The 'all-applied-to' operators (B_Allvec and B_Alltup), may have any non-zero number of arguments[20] and so have a list of child parse trees which may be of any length.

For example, the parse tree for the program:

B_Comp (B_Map ($Increment)) (B_Id)

would appear, symbolically, as:



[20] This is a slight simplification as B_Allvec normally must have at least one argument and B_Alltup, at least two.

where the triangle is the parse tree representing the $Increment function and the arrow labels refer to the child designators shown in figure 4-4. Notice the B_Id function, which is not 'higher-order', has no children.

4.4.2 'Operators' under B_OP

As mentioned previously, for historical reasons, some of the BMF constructs appear under the <b_op> non-terminal in the grammar and are, therefore, prefixed in the syntax by 'B_Op'. Such constructs are represented by a specific tree node type, 'B-Op', with the node containing the type of the operator under the field 'Operator'.

4.4.3 Miscellaneous Constructs

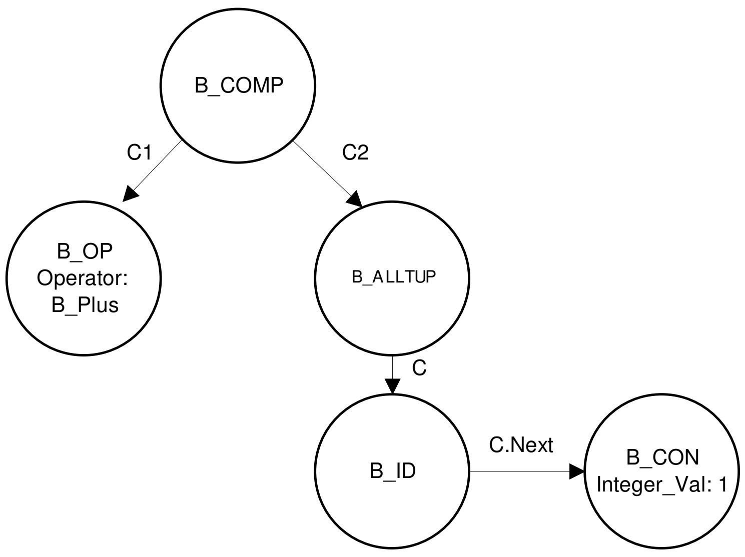
There are several miscellaneous BMF constructs which do not fit into the two previous categories: B_Id, P_Distl and P_Project have no associated extra information or children and so have a 'null' record entry. B_Num nodes store an integer value in a field called 'Integer_val' and B_Con nodes store a 'Constant_Value' which can be an integer, real, or boolean value.

The use of the above elements are illustrated in the following example:

The $Increment function, used a number of times previously, can be written:

$Increment = B_Comp (B_Op (B_Plus)) (B_Alltup [B_Id, B_Con (B_Int 1)]))

when parsed, $Increment can be represented by the following tree:



Notice how the B_Op and B_Con expressions contain the 'extra' information within the node ('B_Plus' in the case of B_Op and '1' is the case of B_Con). Also of note is that the B_Alltup node has a list of children with 'C' pointing to the first argument and 'C.next' referencing the second.

4.5 Evaluator

Once processed by the front-end, the parse tree representing the program is passed to the evaluator. The evaluator is, in many respects, the core of the simulator. It is responsible for the simulated execution of the parallel BMF code as well as, together with the network module, producing the necessary trace information reflecting any inter-processor communication and changes in processor state which may have occurred[21]. This section will concentrate on the evaluation methodology itself with the following two sections detailing the modeling of parallelism and the inter-connection network.

At the top level, the evaluator is invoked by a function call which has the following form:

```
Program_Result := Eval (Pt, Arg)
```

where: Pt is the parse tree representing the program
Arg is the argument of the program

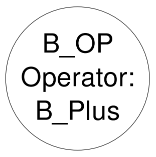
Both Arg and Program_Result are *Values*, as defined in section 4.2.2, to allow the evaluator to accept and return any BMF type. The Eval function effectively serves as an interface between the parser and sub-evaluators which perform the actual evaluations; there is one sub-evaluator for each construct listed under <b_exp> in the grammar. The Eval function invokes the appropriate sub-evaluator depending on the type of the parse tree node ; the type of the tree node is determined by the value of the variant (called Selector) - see figure 4-4. An overview of the way in which the evaluator works can be gained by the following examples:

[21] See section 3.4.1 for clarification of these events.

There are two main categories of functions which are evaluated: higher-order functions (which take other BMF expressions as arguments and hence have children in their parse tree) and those which do not:
The following, simple, program:

B_Op (B_Plus) {2, 3}

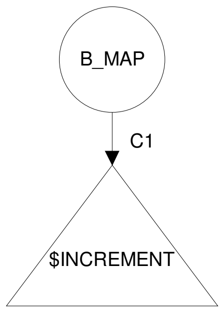
when parsed produces the following single node parse tree:



When Eval is invoked, with this node and the input data, it will call the appropriate sub-evaluator which, in this case, is Eval_B_Op as identified by the node selector: B_OP. Eval_B_Op then identifies the specific operation from the Operator field (in this case B_Plus) and executes the appropriate operation on the program argument.
In contrast, the following example is a higher-order function, B_Map:

B_Map ($INCREMENT) [1, 2, 3, 4]

which would produce the following parse tree:



Eval would invoke the sub-evaluator, Eval_B_Map, which would then perform the map operation in the following way:

```
For each item in input_list Do
    temp_node := Eval(Pt_Increment, item)
    append temp_node to result
End Do
```

where Pt_Increment is the parse tree of $Increment which is the child of the B_Map node referenced by C1. The necessity of the Eval function being polymorphic is highlighted, in this example, because it allowed the Eval function to be called recursively with a different function ($Increment).

4.6 Modelling of Parallel Execution

Simulated parallelism is facilitated in the evaluator by each data item being tagged with the Processor Id on which it resides and each virtual processor having its own clock stored in the *Machine Module*. When an operation is evaluated, the evaluator determines the location of the argument from its tag, and updates the corresponding virtual clock according to the length of the operation. To illustrate this, consider the following example:

P_Map (\$Increment) [1^0, 2^1, 3^2, 4^3]

The argument list is assumed to be distributed across four virtual processors as indicated by the superscripts. Evaluation proceeds in the same manner as with B_Map($Increment), described previously, except that, in this case, each list item is assumed to reside on a different processor and hence, a separate processor clock is incremented with each invocation of Eval($Increment, Item). This reduces the simulated execution time by a factor of four when compared to the case when a single processor is involved.

4.6.1 Tags

Lists are the only data structures capable of being spread across the machine and therefore each item within the list must have its own tag: the tag is referred to as 'Proc_Id' in the record definition for the list_node type as shown in Figure 4-2. Other types, including tuples, may not be spread across the machine and so only require a single tag which identifies the processor on which they reside; tags for such data are also called ‘Proc_Id’ which is a field in the Value record definition - see figure 4-2.

4.6.2 Machine Module

The state of the parallel machine is stored in a data structure called the ‘Machine Module’. Presently, the machine module is effectively an array of clocks, indexed by processor id. Such a design allows extra attributes of each processor to be stored: for example a ‘memory_used’ attribute could be added to facilitate the modeling of space consumption for each processor versus time. As can be seen from figure 4-1, both the evaluator and network modules access the machine module when they update the clocks.

With each invocation of the evaluator, an operation that consumes time proceeds in the following general manner:

- The current processor, P, is determined by inspecting the tag of the Item.
- The current time, T, is read from the appropriate virtual processor clock in the machine module.
- A Start_Busy[22](P, T) event is issued to signal the commencement of the computation.
- The specified operation is performed.
- The virtual processor clock is updated according to the length of the operation (T_{op}).
- An End_Busy(P, $T+T_{op}$) event is issued to signal the end of computation.

4.6.3 Inter-Process Communications

Parallelism, in BMF, involves two broad categories of operations: those such as P_Map which act upon pre-distributed data and those which involve the transfer of data via inter-processor communications such as P_Reduce and P_Split. Within the evaluator, the location of data is changed by simply altering the tag, however to

[22] Start_Busy is equivalent to Stop_Idle and Stop_Busy is equivalent to Start_Idle

simulate the effects of a real interconnection network, any inter-processor data movements are also associated with a call to the network module. These calls generate the appropriate trace information and also update the virtual clocks to take account of the communications delays.

4.7 Interconnection Network Modelling

There are two mechanisms by which the evaluator interacts with the network module: Normal and Delayed send.

4.7.1 Normal Send

For every inter-processor communication involved in executing the parallel operators, except P_Project and P_Scan, 'Normal Send' is invoked by the evaluator. Normal send has the following Ada specification:

```
function N_Send(Source, Dest : Integer ; Time : Float ;
                    Size, Ty : Integer) return Float;
```

The parameters are:	Source - Source Processor
	Dest - Destination Processor
	Time - The time at which the Send begins
	Size - The length of the message in Bytes
	Ty - The *type*[23] of message

Each call to N_Send generates all the trace events (described in chapter 3) associated with the communication by the following steps:

- Event Start_Send (Source, Time) is generated
- Send overhead time, T_{SOH} is calculated
- Send finish time $T_{SF} = Time + T_{SOH}$ is calculated
- Event End_Send (Source, T_{SF}) is generated
- The message arrival time at the destination processor:
  $T_{ARR} = T_{SF} + T_{Transport}$ is calculated
- Event Start_Receive (Dest, T_{ARR}) is generated

[23] i.e. the message type as defined in PICL - see section 3.2.3

- Receive overhead, T_{ROH} is calculated
- Receive Finish time $T_{RFT} = T_{ARR} + T_{ROH}$ is calculated
- Event End_Receive(Dest, T_{RFT}) is generated
- Clock of Destination processor is updated to T_{RFT}

The relationship between the various times is illustrated by the following line diagram:

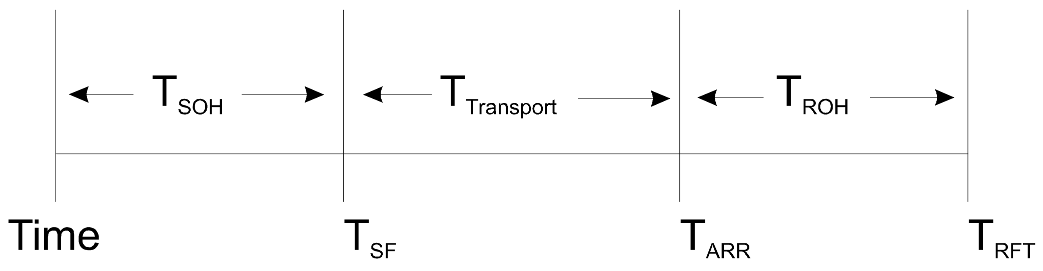


Figure 4-5: Communications Timeline

The cost model employed in calculating the overhead times, T_{SOH} and T_{ROH}, and the transport time $T_{Transport}$ will be discussed in section 4.7.3.

4.7.2 Delayed Send

The previous scheme of updating the clock of the destination processor and generating the associated trace information for the receive, in a single function call makes the assumption that the destination processor was *not* sending at the same time. This assumption holds for all the presently implemented parallel operators except P_Project and P_Scan.

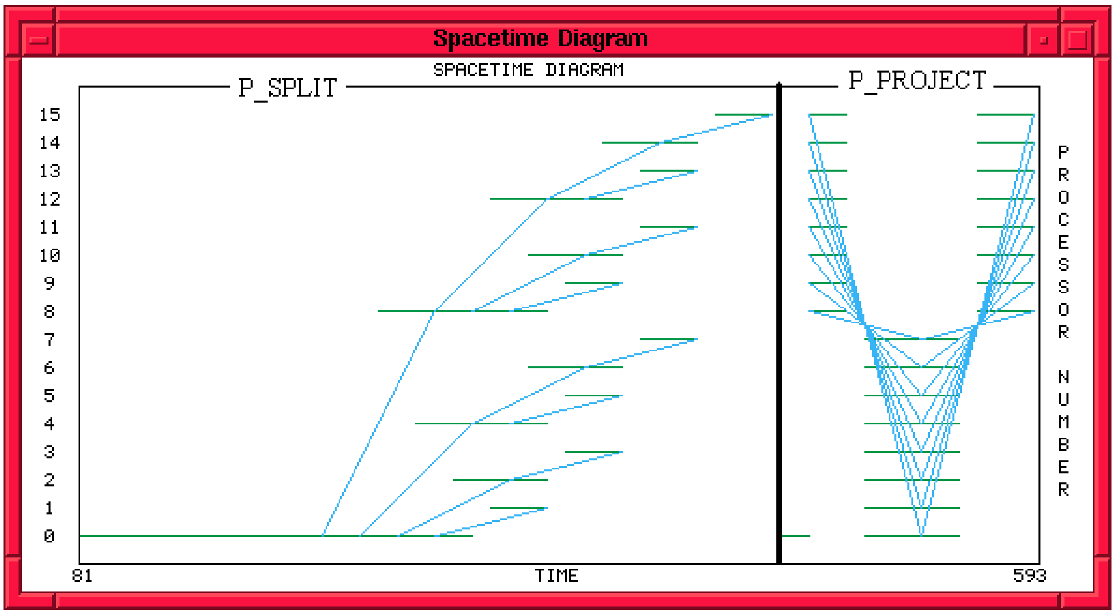


Figure 4-6: Space-Time Diagram of Project using Normal Send

Figure 4-6 shows a Space-Time diagram of a parallel Project operation affecting a full reversal of a distributed list using Normal Send. The left-hand side shows the P_SPLIT operation which distributes the input list across the machine[24]. During normal Project operations, all transmitting processors commence sending at the same time. However, it can be seen from figure 4-6, that processors 0 to 7 do not commence their send until they receive the data from processors 8 - 15. This behaviour, although semantically correct, does not accurately reflect the way in which Project actually operates. This abnormality can be explained as follows: When simulating Project, the evaluator calls N_Send starting with processor 15, proceeding in descending order through to processor 0. Each call to N_Send updates the clock of the sending processor as well as setting the clock of the receiving processor to the time at which the reception of the message would be complete. Consider the case of processors 15 and 0; when processor 15 sends to processor 0, the clock of processor 0 will be set to the time at which the reception of the message from 15 would be complete. When, in turn, processor 0 is called to send, it will commence from the current value of its clock rather at the beginning of the Project operation. A similar argument can be applied to processors 1 - 6.

[24] P_SPLIT and P_PROJECT will be described in more detail in chapter 5.

This problem can be alleviated by the use of the ‘Delayed Send’ procedure which allows the send operations for all processors to be completed before any receive operations are processed. Delayed send takes in the same parameters as N_Send:

```
function N_Send_D(Source, Dest : Integer ; Send_Start_Time : Float ;
                  Size, Ty : Integer;) return Float;
```

A call to N_Send_D performs the following steps:

- Event Start_Send (Source, Time) is generated
- Send overhead time, T_{SOH} is calculated
- Send finish time T_{SF} = Time + T_{SOH} is calculated
- Event End_Send (Source, T_{SF}) is generated

thus far, the steps are the same as for the N_Send. At this stage, the send finish time T_{SF}, and the other parameters (source, dest, size & Ty) are stored in a record which is added to a global linked list structure. This effectively completes the call to delayed Send.

Once all the send operation have been completed, the pending ‘receives’, stored in the linked list, are resolved though the invocation of the ‘Do_Recieves’ function which has the following specification:

```
      function Do_Recieves return float;
```

For each pending receive record in the list, Do_Recieves effectively resolves the operation by completing the last six steps of the Normal Send procedure.

Figure 4-7 shows a parallel Project operation using the Delayed Send operation: clearly, in this case, all the send operations commenced at the same time.

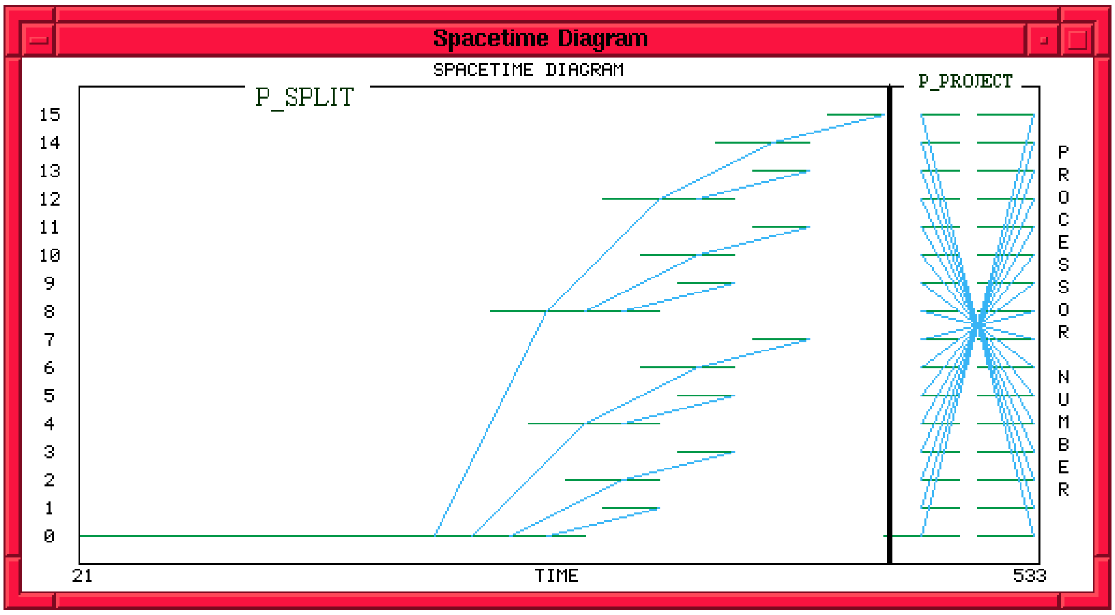


Figure 4-7: Space-Time Diagram of Project using delayed send.

4.7.3 Cost Modelling

There are three cost components which determine the timing of communication events: send and receive overheads and transport time. The overhead times are generally independent of network topology and in this implementation, are set, to be proportional to the message length. The transport time through the network is dependent on both the network topology, message length, and is subject to the effects of congestion; in this basic model, however, the transport time is fixed. The extended network model, dicussed in chapter 6, outlines enhancements made to enable both network topology and congestion to be taken into account when determining the transport time.

5. Results for Fundamental Constructs

5.1 Overview

The operation of the parallel BMF operators: P_Split, P_Distl, P_Reduce, P_Map, P_Zip, P_Scan & P_Project, was broadly described in chapter 2; in this chapter, a detailed description of the operation and implementation of the various operators will be given. Examples of simulated results using that various operators will be illustrated using Paragraph displays.

5.2 Comments on Visualisation Displays

In general, the total execution time, in virtual time units, for each program will differ. To sensibly use the available chart space, each program requires the choice of a different time-axis scaling factor. Thus when comparing charts between different programs, this should be taken into account.

5.3 P_Split

Currently, there are two parallel operators which distribute data from a single processor to many, namely: P_Split and P_Distl. Although there are a number of different distribution strategies which may be employed, this implementation uses a 'tree-like' pattern which enables the distribution operation to occur in $\log_2 P$ steps where P represents the number of processors over which the data is spread. This method is apparently optimal for networks which can be traversed in $O(\log_2 P)$ steps (such as Hypercube and tree networks): other topologies could well benefit from a different strategy.

The implementation of P_Split is based on the algorithm used in [1]. The operation proceeds in two stages: the first being the application of the serial split operator, B_Split to partition the input list into a 'list of lists' and the second, 'Split_Parallel' to actually simulate the distribution of the data across the machine as required for parallel operation.

5.3.1 B_Split

B_Split partitions the input vector according to the following algorithm from [1]:

The first sub-vector is created with length[25]: ceil (#list / P)

Subsequent sub-vectors are created, recursively with length: ceil (#tail / (p-1))

For example, Split(4) applied to a list of length 31 would be calculated as follows:

# sub-vector 1 = ceil (31 / 4) = 8

# sub-vector 2 = ceil (23 / 3) = 8

# sub-vector 3 = ceil (15 / 2) = 8

# sub-vector 4 = ceil (7 / 1) = 7

which is, combinatoriliy, the most even spread possible.

5.3.2 Split_Parallel

Split_Parallel takes a list of sub-vectors and simulates the distribution of the list across the machine by generating the appropriate sequence of calls to N_Send. The procedure has the following specification:

```
procedure Split_Parallel (L : List ; Seg_Length : Integer;
                           Stride : integer) is
```

where L is the pointer to the beginning of the list to be distributed, Seg_Length is the length of the segment to be processed [26], and Stride[27] is the final spacing of the list components expressed as a power of 2.

Figure 5-1 illustrates, in schematic form, the nature of the distribution pattern in the form of a space-time diagram:

[25] The list length function is sometimes denoted by '#'

[26] Seg_Length allows specification of how much of the list following L is to be processed.

[27] The role of stride facilitating the use of nested splits is discussed in Section 5.3.4

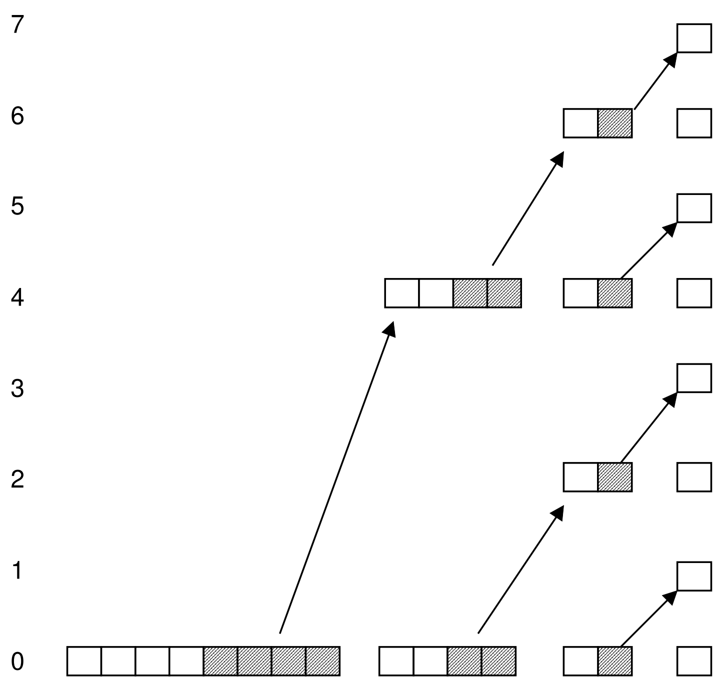


Figure 5-1: Data Distribution for P_Split Split Operation

The data initially resides on processor 0. For each stage, the shaded section indicates that portion of the list which is sent onto the next processor and the non-shaded section indicates that portion which remains. Broadly speaking, at every stage, each processor sends 'half' of its data to the next processor with this process continuing until each processor is left with only one list item. To generate such a distribution pattern, Split_Parallel calls itself recursively.

Each invocation of Split_Parallel determines the 'division point' (DP) of the input list (as indicated in Figure 5-2) as well as the lengths of the front and tail segments as shown. List nodes are arbitrarily numbered from zero upwards. DP is the number of the first node in the segment which is sent on to the next processor. DP_Ptr is a pointer to the DP node. Split_Parallel makes use of an auxillary function, LogP (x), which returns the largest power of 2 *strictly smaller* than x.

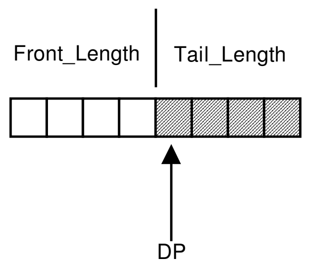


Figure 5-2: Variables used in Split_Parallel

The distribution algorithm proceeds as follows:

- The division point, DP is calculated from: DP = LogP (Seg_Length)
- If DP = 0 then return (i.e. input list has a length of one)
- Tail_Length is calculated from Seg_Length - DP (elements are numbered from zero)
- Calculate Dest_Id = Present_Id + Stride_Factor[28] * Div_Point)
- ‘Shift’ segment starting at DP to Dest_Id by updating tags
- Call N_Send (Present_Id, Dest_ID, ….) to generate trace information
- Recursively call Split_Parallel with tail_segment -
  Split_Parallel(DP_Ptr, Tail_Length, Stride)
- Front_Length = DP
- Recursively call Split_Parallel with front_segment -
  Split_Parallel (L, Front_Length, stride)

[28] Stride_Factor = 2^{Stride} i.e. stride is the final spacing of the distributed list expressed as a power of 2.

5.3.3 Simulation Results for P_Split

To generate the displays, the following BMF code was used:

```
P_Split (B_Num 8) (B_Num 0) [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

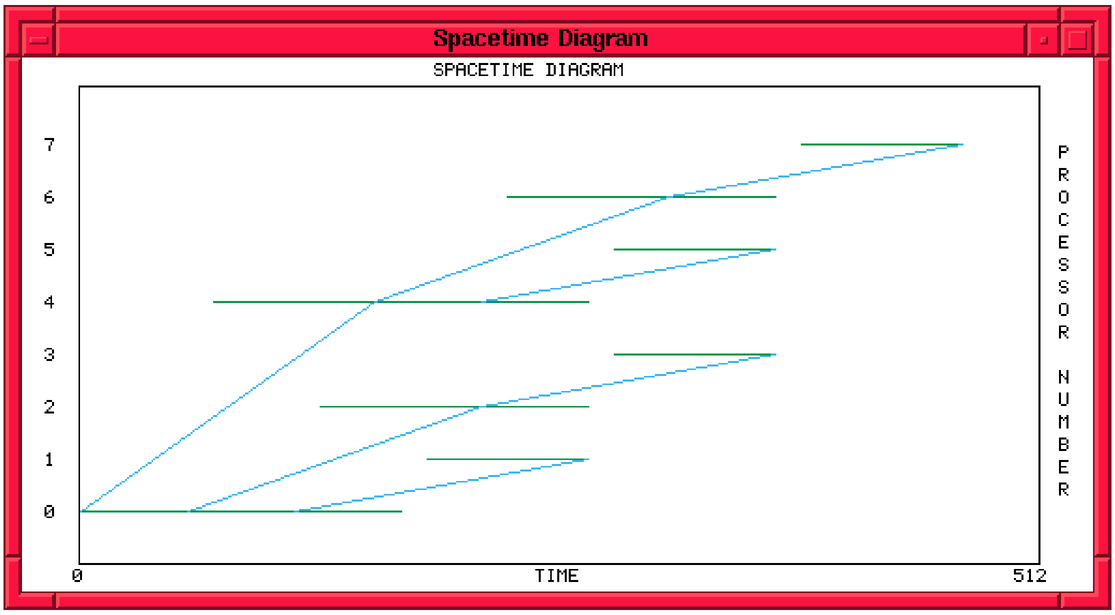


Figure 5-3: Space-Time Diagram of Split(8)

Figure 5-3 shows a Spacetime diagram[29] for a parallel split operation over 8 processors. The general form of the communication pattern corresponds to that shown in the schematic in Figure 5-1; however, the simulated display does not appear as uniform due to the effects of the finite overhead times involved with each message transmission. In particular, the message for processor 7 had to 'pass through' two intermediate nodes on its way and hence was the last to arrive. Also of note is that there is no busy time at the beginning of the operation because the action of the *B_Split* is not regarded as consuming 'real' computation time in this case.

[29] The Space-Time Diagram and Gantt chart were introduced in section 3.5

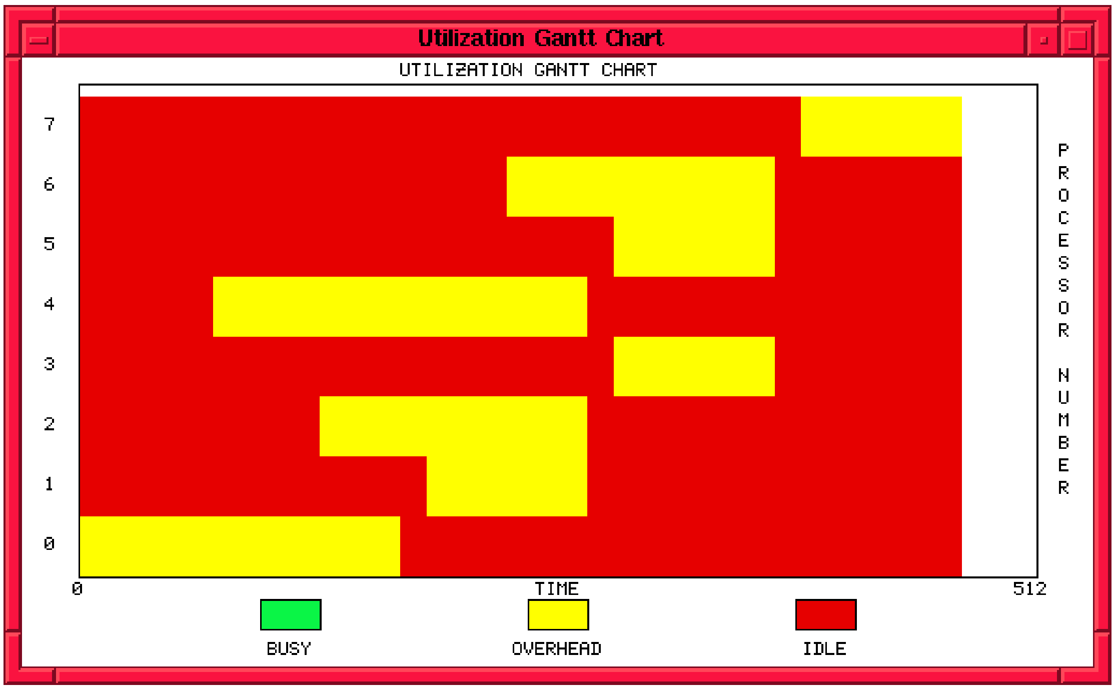


Figure 5-4: Gantt Chart for Split(8)

The Gantt chart, for a Split(8) is shown in Figure 5-4. Clearly, for each processor, the split operation consists mostly of idle time with a single burst of communications overhead. A parallel BMF operation is not considered complete until all processors involved in the particular operation have finished; in this way, a form of bulk synchronization is imposed upon BMF programs . In this case the next operation in the program cannot start until processor 7 has finished receiving.

The chart shown in Figure 5-5 illustrates the communications traffic density in number of messages versus time. Split produces a reasonably symmetrical peak in traffic density with the staggered falloff at the right-hand side of the peak due to the late arrivals.

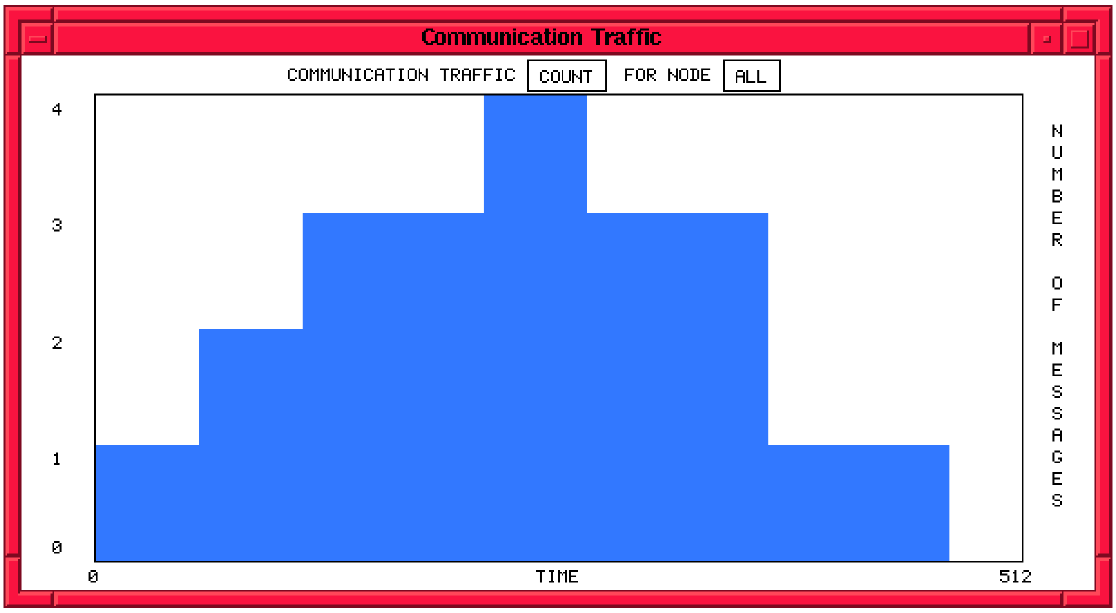


Figure 5-5: Traffic Volume for Split(8)

5.3.4 Nested Splits

The stride argument of P_Split is used to facilitate nested parallelism through the use of nested Splits. When nested splits occur, the higher level splits must have an appropriately larger stride argument than their 'children'. The quantitization of the stride interval to a 'power of 2' ensures that lists thus distributed are compatible with the reduce and scan algorithms explained later. A common use for nested splits is to distribute higher-dimensional lists across the machine to operate on all the elements in parallel. For example the following program spreads a 4x4 'matrix'[30] across the machine and increments every element by one before reducing the structure back to its original form. (The operation of P_Map and P_Reduce are discussed in sections 5.5 and 5.6 respectively).

```
        B_Comp (P_Reduce(B_Op(B_Conc))
           (B_Comp (P_Map(P_Map(P_Reduce(B_Op(B_Conc)))))
               (B_Comp( P_Map(P_Map(P_Map(B_Map($Increment)))))
                    (B_Comp (P_Map(B_Map(P_Split(B_Num 4)(B_Num 0))))
                                                (P_Split(B_Num 4)(B_Num 2)))))
```

```
[ [0, 1, 2, 3], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, 16]]
```

[30] Really a 'list of lists'

The program begins by splitting the input list over four processor, using a stride[31] of 4. This step produces the following intermediate result:

```
[ [[0, 1, 2, 3]]⁰, [[5, 6, 7, 8]]⁴,
  [[9, 10, 11, 12]⁸,[[13, 14, 15, 16]¹²] ]
```

To distribute each sub_list across the four intermediate processors, another, nested, split with a stride of one is mapped across the list yielding:

```
[ [[[0]⁰, [1]¹, [2]², [3]³]], [[[5]⁴, [6]⁵, [7]⁶, [8]⁷]],
  [[[9]⁸, [10]⁹, [11]¹⁰, [12]¹¹], [[[13]¹², [14]¹³,[15]¹⁴, [16]¹⁵] ] ]
```

Nested maps are then used to apply the $Increment function to each number before two reduces are conducted to restore the original form of the data.

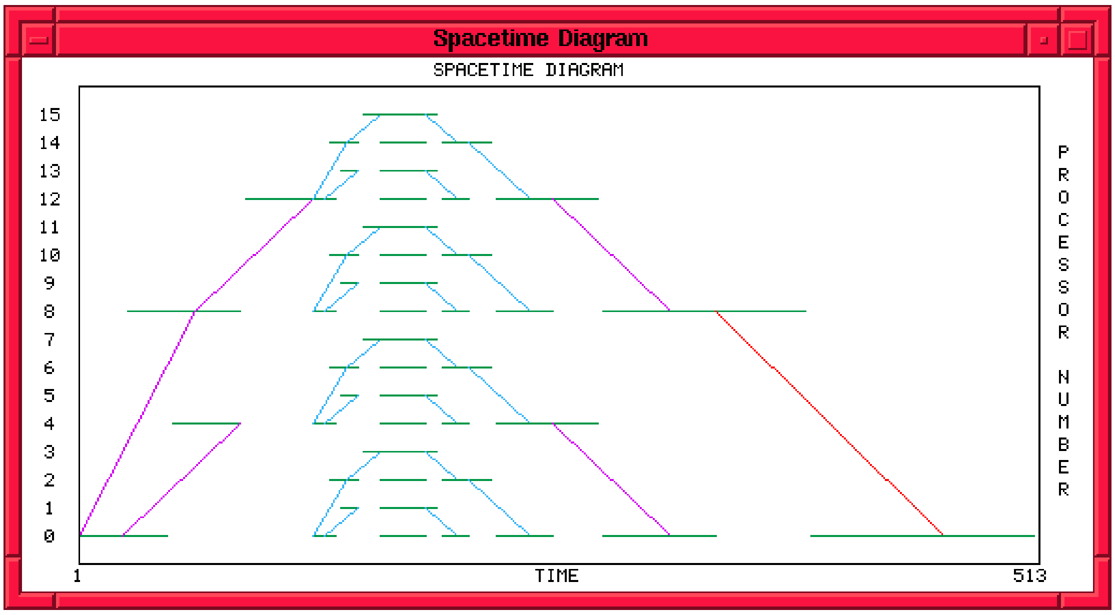


Figure 5-6: Space-Time Diagram for Nested Split

The above Space-Time diagram shows the operation of the two nested splits: the outer split sending to processors 4, 8 & 12 with the inner split then commencing to distribute the sub-lists across the four intermediate processors. Note that the colour of

[31] Recall that the stride argument is expressed as a power of 2 - i.e. An argument of 2 yields a stride of four.

the communication lines of the outer split are purple reflecting the larger amount of data being transmitted (4 bytes) as opposed to that of the inner split (1 byte).

5.4 P_Distl

The parallel Distl is a broadcast operation used to transmit a single value to every node of a pre-distributed list. The transmission pattern used to distribute the value is effectively the same as for P_Split except that in this case, the same value is being transmitted. The procedure which implements the communications patterns of Distl is essentially the same as Split_Parallel.

5.4.1 Results for P_Distl

To exemplify the operation of P_Distl, the following program was used:

```
B_Comp ( P_Distl)
            ( B_Comp ( B_Alltup[ B_Con (B_Int 2), B_Id] )
                        ( P_Split (B_Num 16) (B_Num 0) ) )
```

The program works as follows:

The input list is first distributed across the machine using P_Split. B_Alltup is then used to create a tuple consisting of the value to be distributed, 2, and the distributed list. P_Distl is then applied to the resultant tuple.

As an example, if the above program were used with a P_Split(4) instead of P_Split(16) and applied to the list [1, 2, 3, 4, 5, 6, 7, 8] then the state of the program before the application of P_Distl would be:

P_Distl {2, [$[1, 2]^0$, $[3, 4]^1$, $[5, 6]^2$, $[7, 8]^3$] }

the application of P_Distl would then yield:

[$\{2, [1, 2]\}^0$, $\{2, [3,4]\}^1$, $\{2, [5, 6]\}^2$, $\{2, [7, 8]\}^3$]

The results of the simulation of this program (with 16 processors) appear below:

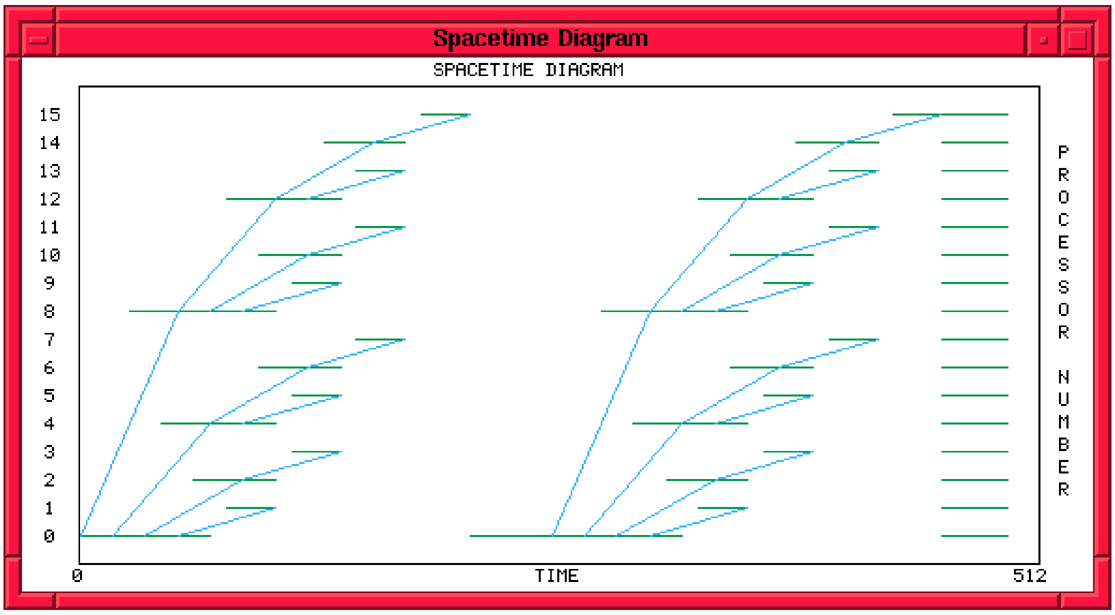


Figure 5-7: Space-Time Diagram of P_Distl

The trace for the initial P_Split operation appears on the left-hand side of Figure 5-7 with the trace for P_Distl appearing on the right. Clearly, the two communication patterns are almost identical in both shape and size.

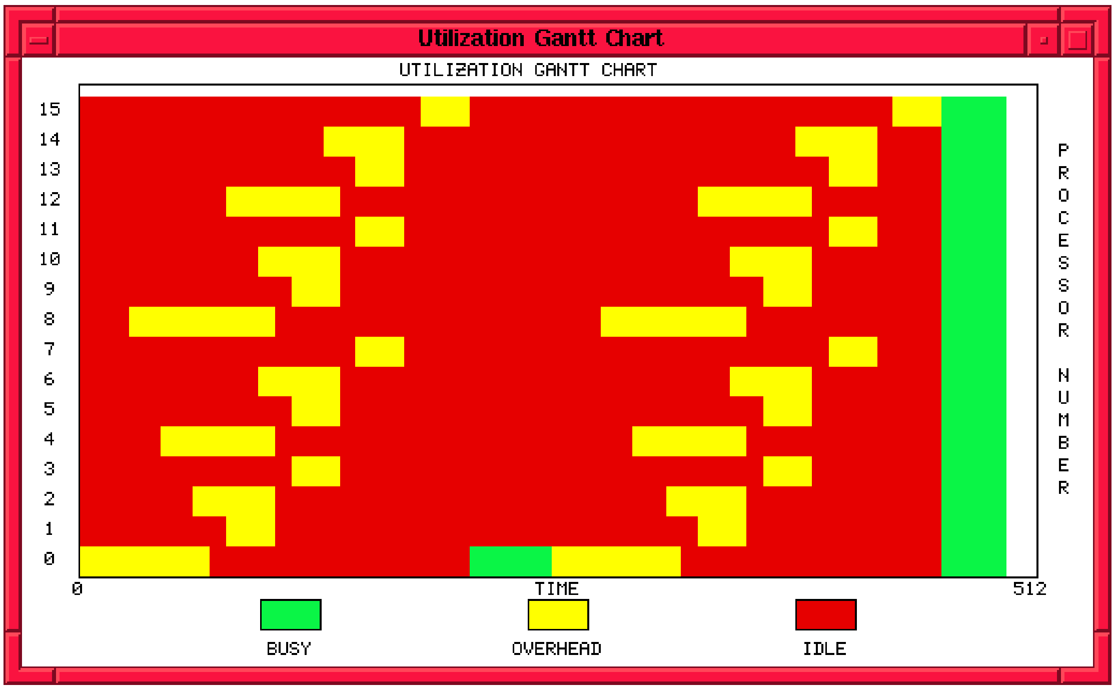


Figure 5-8: Gantt chart of P_Distl operation

Processor 0 is busy, at the center of the Gantt chart, just prior to commencing the send due to the execution of the B_Alltup and B_Id functions. All processors commence forming the result tuples once the distribution phase is complete, as indicated by the strip of busy time on the right hand side of the chart.

The size of the data to be distributed has an important bearing on the time taken to complete the operation. In above example, the size of the data being distributed was comparable to data being transmitted by the split operation. The program can be modified to distribute a sixteen item list in the following manner:

```
        B_Comp
          ( P_Distl)
          ( B_Comp ( B_Alltup[ B_Comp (B_Op (B_Iota)) (B_Con(B_Int 16)),B_Id]))
                       ( P_Split (B_Num 16) (B_Num 0) ) )
```

where B_Iota is used to create the list [0, 1, 2, …, 15] which will be distributed over the input list.

This program, when applied to a list of sixteen elements, produced the following results:

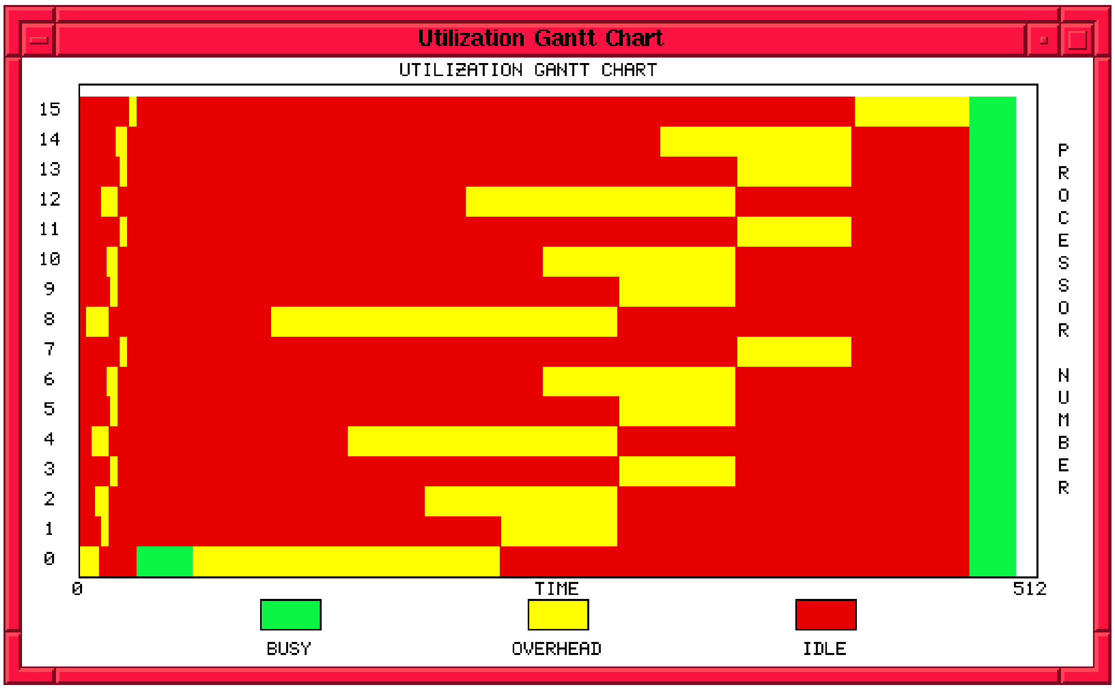


Figure 5-9: Gantt Chart of Large P_Distl Operation

In contrast to the original program, the P_Distl operation takes a much longer time than the P_Split. The split operation shown here took the same amount of time as the split in Figure 5-7, but the larger scaling factor used here makes it appear compressed. From this chart, it is clear that the sixteen-fold increase in data size resulted in a much larger execution time, for the P_Distl operation, due to increased communications overheads as well as greater computation time.

Shown in Figure 5-10, is the traffic volume versus time for the same program:

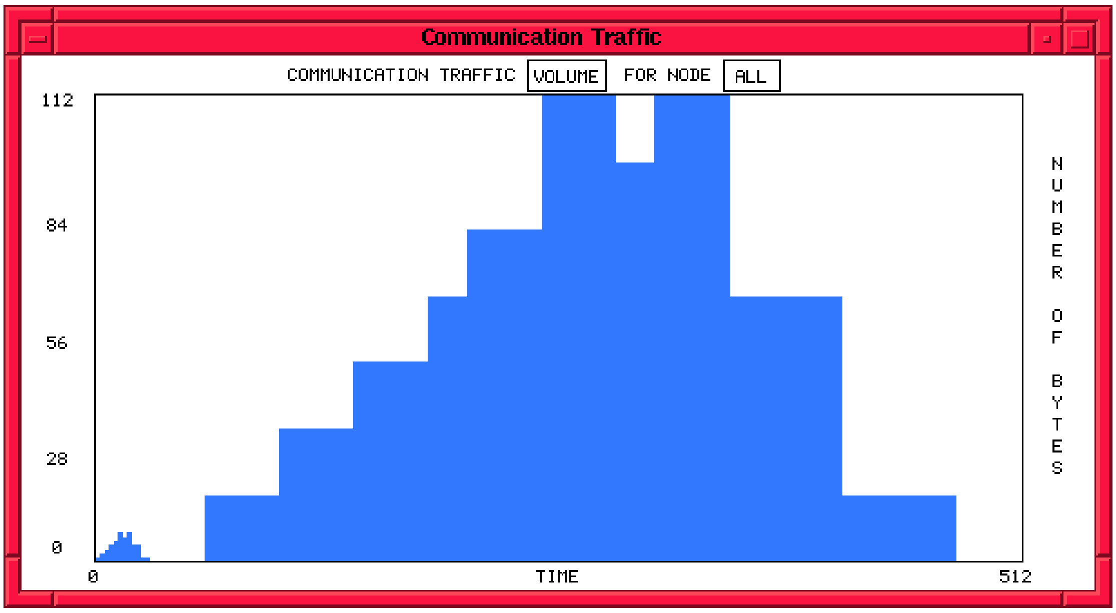


Figure 5-10: Traffic Volume for Large Distl

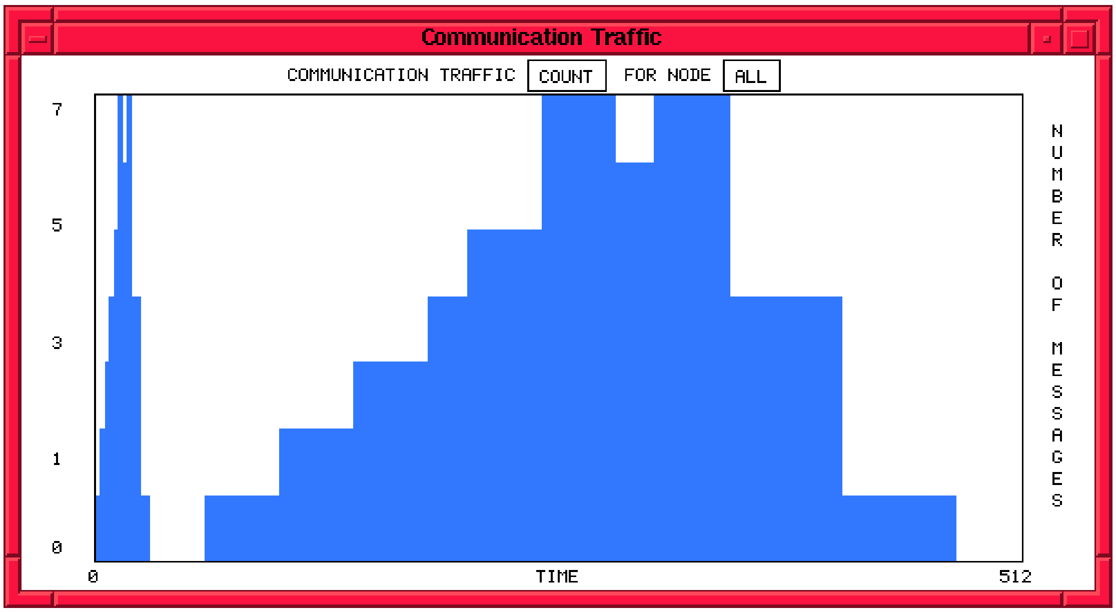


Figure 5-11: Traffic Count for Large Distl

As expected, the peak number of bytes traveling through the network, for the P_Distl operation (in Figure 5-10), greatly exceeds that of the split operation. The peak number of messages, however, remains the same for both operations but the value to be distributed was sixteen bytes long and the peak message count of seven therefore corresponds to a peak byte value of (16 x 7) = 112 bytes as shown in Figure 5-10.

5.5 P_Map

The parallel map operation facilitates parallel processing over a distributed list. The implementation of parallel map is essentially the same as that of the sequential version, B_Map, except that extra clock synchronization operations are required.

P_Map proceeds as follows:

```
Let $T_S$ be the start time of the map operation with the function F :
• The clocks of the processors on which the distributed list resides are
  synchronized to $T_S$.
• For each Item in Input List do
    • Eval( F, Item )          … apply function F to data Item
    • Time_Max = Max ( Time_Max, Eval_Time )
• Synchronize processor clocks to Time_Max.
```

Because every list Item is tagged with the Id of the processor on which it resides, each call to Eval will update the appropriate processor clock according to how long the particular operation took. The P_Map operation is considered to be complete when all processors have completed their task. Hence, the time taken by the last processor to finish, determines the time at which the P_Map operation completes. A record of the latest finishing time (Time_Max) is kept so that this value can be used to set the finishing time of the entire map operation.

5.6 P_Reduce

P_Reduce takes data, which is distributed over a number of parallel processors, and combines the data items with an associative operator, back to a single processor.

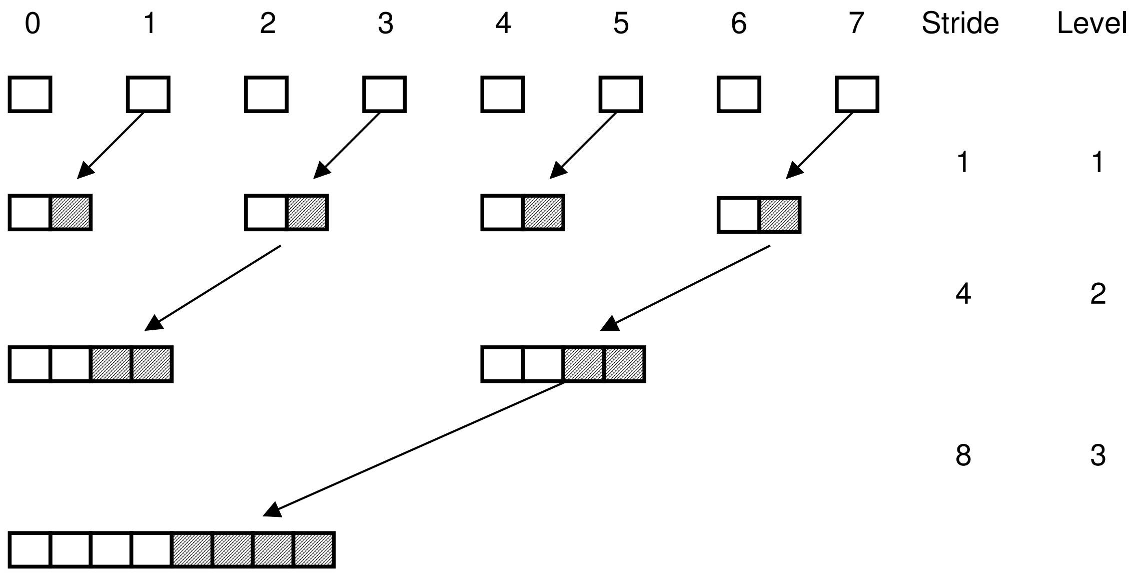


Figure 5-12: Schmatic representation of P_Reduce

The above diagram illustrates the operation of P_Reduce for a list distributed over eight processors. Although the diagram suggests a reduction with the concatenate operator, as shown by the increasing length of the list segments, the overall communication pattern is equally applicable to all operators. The operation proceeds in $\log_2 N$ steps, called levels, where N is the length of the list. Starting from level one:

- While the Length of the input list > 1 do
  - Set Stride equal to 2^{Level} . - as shown in the diagram.
  - For each Item in the list:
    - If Item-tag MOD Stride = 0 AND (not the last item) then
      - Initiate send *from* the next item's processor *to* the current one by calling N_Send
      - combine current processor's item with the received item using the operator with which the reduce was called.
  - Increment Level

The following program illustrates a common use of P_Reduce which is to 'gather' a split list back to a single processor, using the concatenate operator.

B_Comp (B_Comp (P_Reduce (B_Op (B_Conc)) (P_Map ($Increment)))
(P_Split (B_Num 16) (B_Num 16))

The following charts in Figures 5-12, 5-13, 5-14, 5-15, 5-16 & 5-17 display the behaviour of the above program:

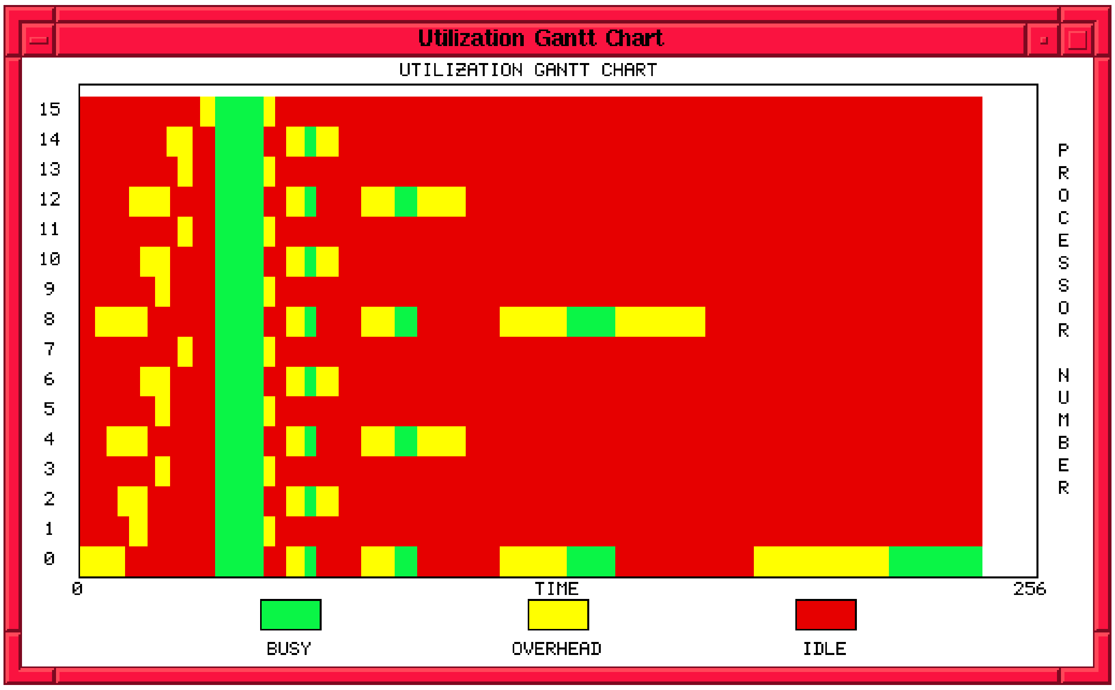


Figure 5-13: Gantt Chart of Split-Map-Reduce

Looking at the above chart, the split(16) can be seen distributing the input list across the machine. Once the split operation has finished, the parallel map applies the $Increment function to every list element in parallel as evidenced by the vertical strip of busy time. It is important to note that all the processors involved in the P_Map operation start at the same time, once processor 16 has received its message, this is due to the synchronization of the processor clocks at the commencement of P_Map.

Upon completion of the P_Map operation, the parallel reduction operation begins synchronously. The three components that make up each inter-processor communication can be seen by considering, for example, the point at which processor

1 sends to processor 0 (at level 1): there is a period of overhead when processor 1 sends the message, followed by idle time whilst the message is traveling through the network after which, there is a period of overhead when processor 0 receives the message. The communication overheads increase as the reduction progresses due to the increasing message sizes caused by the use of the concatenate operator. For the same reason, the busy time at each level of the reduction also increases because the time taken for a concatenate operation to complete is proportional to the argument size.

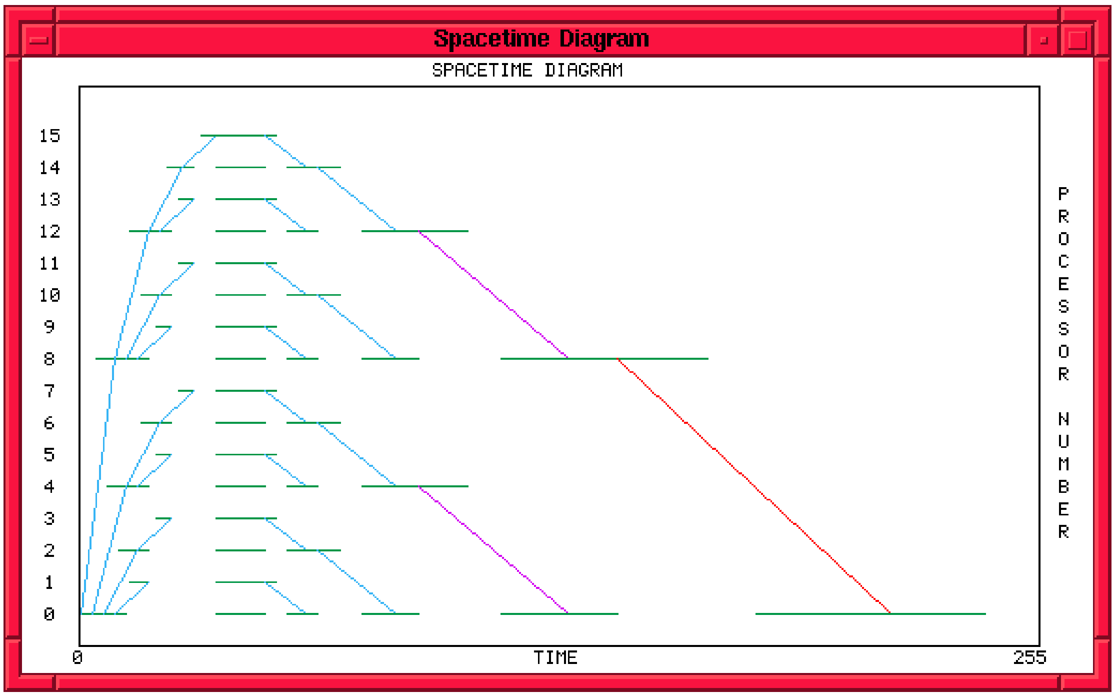


Figure 5-14: Space-Time Diagram of Split-Map-Reduce

The typical 'tree-like' distribution pattern of the split operation can be seen on the left of the chart after which there occurs a vertical region of horizontal lines representing the busy time of the parallel map, followed be the parallel reduction operation. In contrast to the previous examples, the lines representing communications change colour as the reduction operation proceeds; the colours are used to reflect the fact that the message length being transmitted increases due to the concatenation operation.

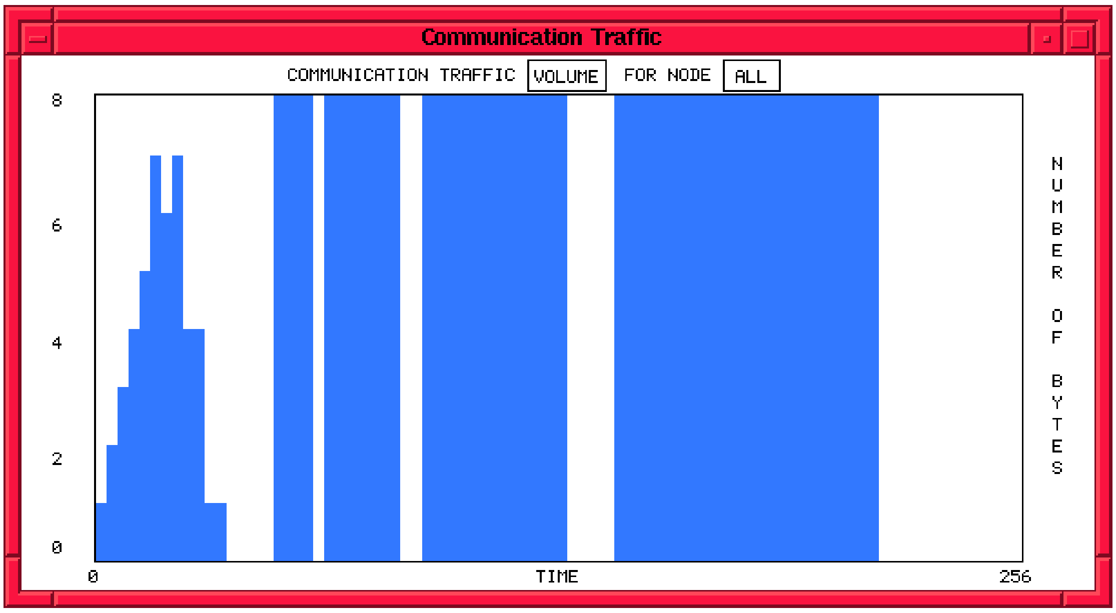


Figure 5-15: Traffic Volume for Split-Map-Reduce

Looking at the traffic volume versus time, the peak due to the split operation can be seen on the left. The reduction operation then proceeds causing a series of constant height bursts indicating that the total number of bytes being transmitted during each level of the reduction remains constant. This is commensurate with the fact that the concatenate operator keeps the *total* amount of data constant, only changing the message lengths by the joining of data items. The width of the bars, which represents communications time, increases as the as the message lengths increase.

The gaps between the communication bursts represent the busy time when the processors are performing the concatenation; note once again, that this busy time increases as the reduction level increases.

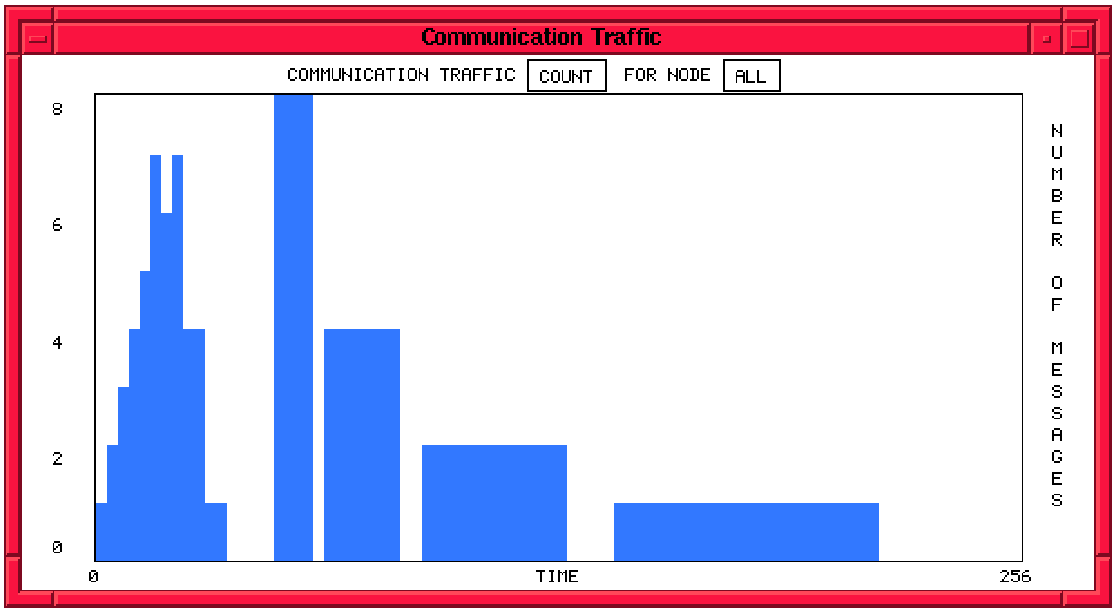


Figure 5-16: Traffic Count for Split-Map-Reduce

Although the total amount of data remains constant, when reducing with concatenate, the number of messages falls exponentially in keeping with the fact that parallel reduction is a $\log_2 N$ algorithm.

The basic interconnection model used in the simulator does not account for different network topologies or the effects of congestion. Paragraph allows one to *assume* a given network topology and superimpose the communication patterns upon the network so that a profile can be built up indicating how many messages are present in each segment at any given time. The following Figure shows the parallel reduction operation, at each level, assuming a switched binary tree topology: (the sequence proceeds from left to right - top to bottom).

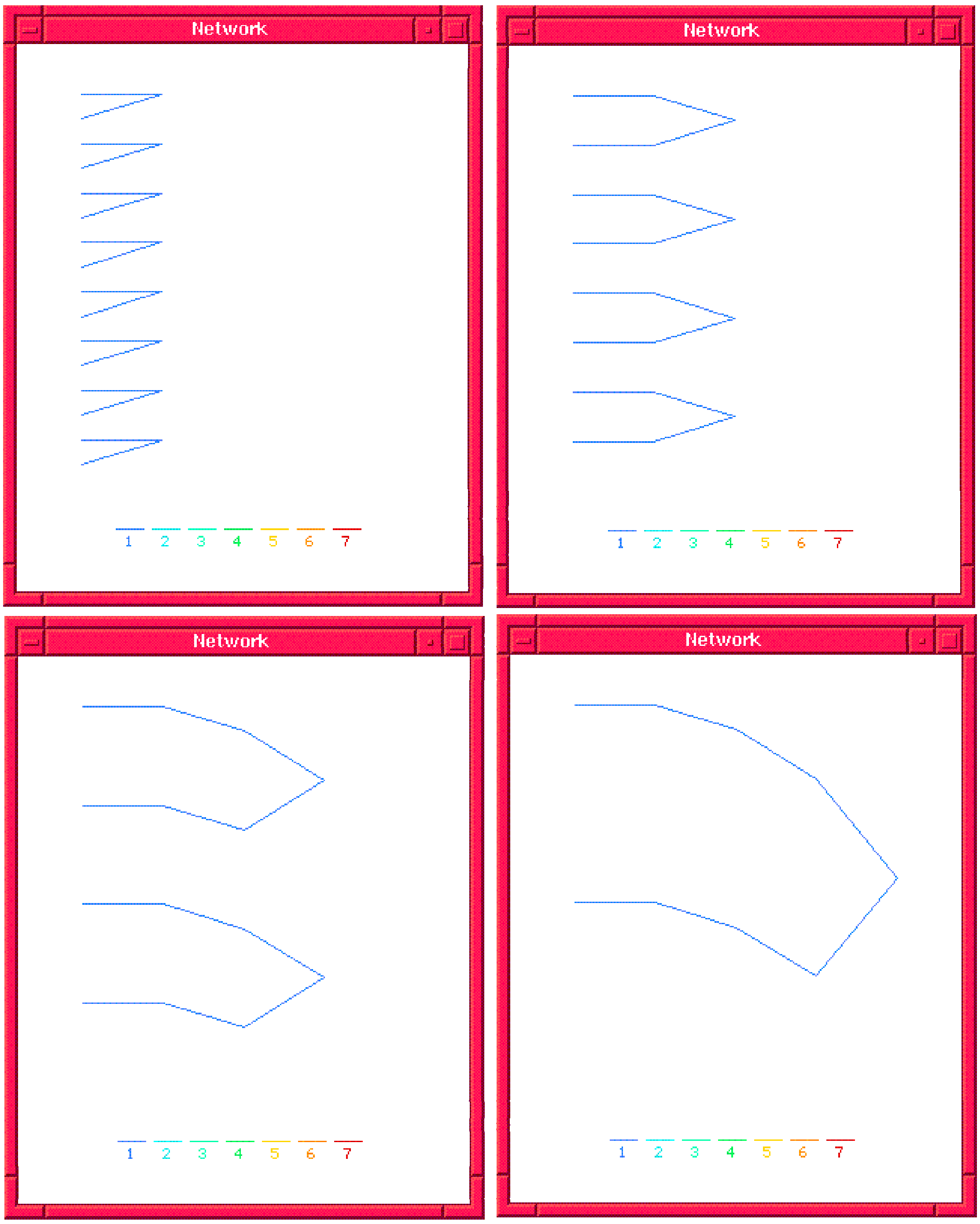


Figure 5-17: Binary Tree Network Animation for Reduce

The above figure illustrates that a reduction performed on such a network would not cause congestion because at no time, is there more than one message traveling through a given network segment.

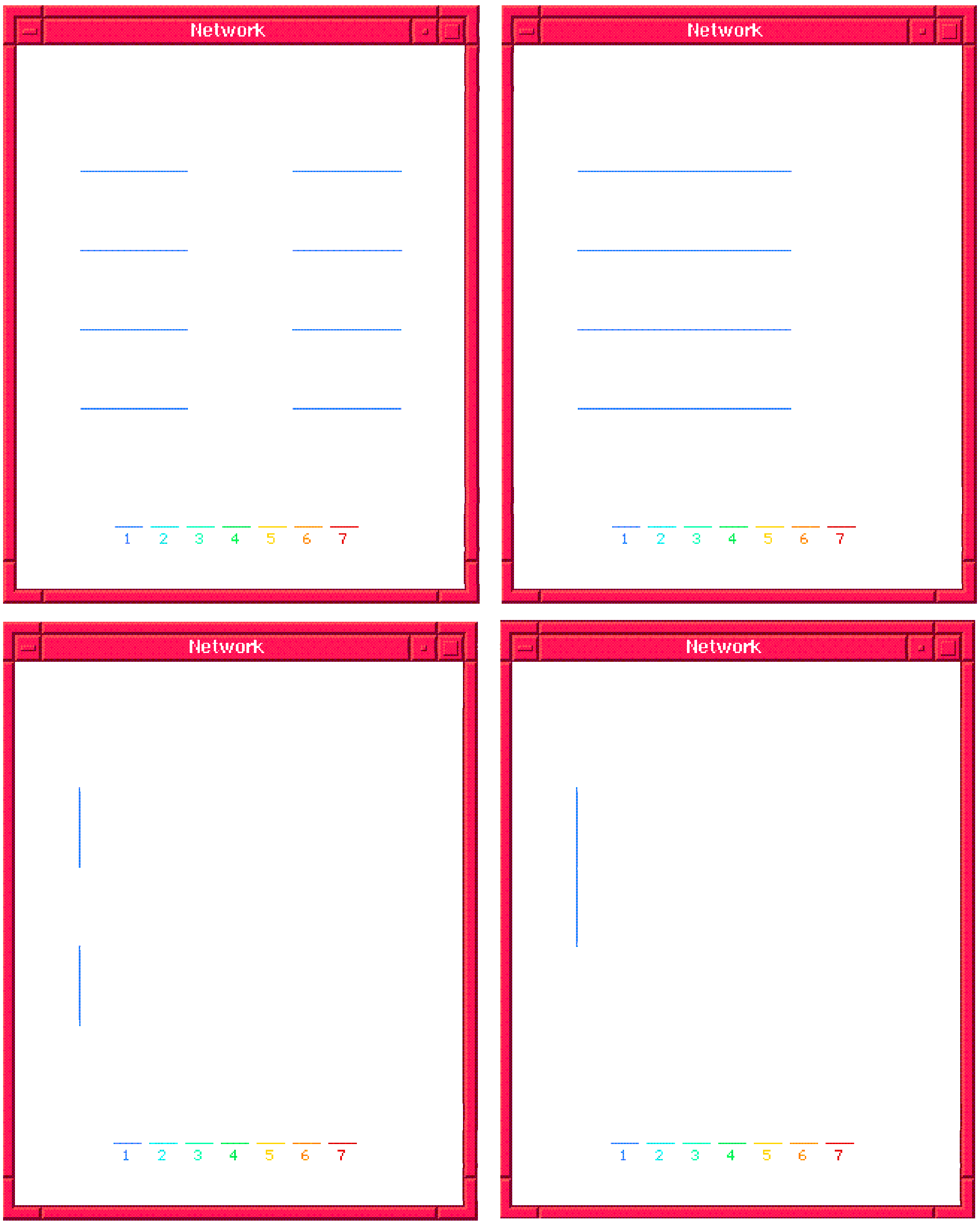


Figure 5-18: 4x4 Mesh Network Animation for Reduce

The above figure shows the four reduction levels, assuming a 4x4 Mesh network. Again, such an operation would cause no congestion and furthermore, the maximum path traveled in this network is three hops compared to eight in the case of the binary tree shown in Figure 5-17.

5.7 P_Zip

The parallel zip operation, P_Zip, is implemented in the same way as the sequential version except that, like P_Map, a clock synchronization occurs (for all processors involved) at the beginning end of the operation. P_Zip does not require any inter-processor communications to occur. Clock updates are handled by Eval, using the list item tags. (cf. P_Map in section 5.5) P_Zip raises an error if the input lists are not conformable[32].

The following program was written to illustrate the operation of parallel zip:

```
B_Comp
  (P_Zip (B_Op (B_Conc)))
  (B_Alltup [ P_Split (B_Num 16) (B_Num 0), P_Split (B_Num 16) (B_Num 0) ] )
```

The last line of the program, effectively makes two copies of the input list and splits them across the machine. For example if the list used only split(8) and was applied to the input: [1, 2, 3, 4, 5, 6, 7, 8] it would return:

$$\{ [[1]^0, [2]^1, [3]^2, [4]^3, [5]^4, [6]^5, [7]^6, [8]^7],$$
$$[[1]^0, [2]^1, [3]^2, [4]^3, [5]^4, [6]^5, [7]^6, [8]^7] \}$$ (note that these lists are conformable)

which when acted upon by P_Zip (B_OP (B_Conc)) would yield:

$$[[1, 1]^0, [2, 2]^1, [3, 3]^2, [4, 4]^3, [5, 5]^4, [6, 6]^5, [7, 7]^6, [8, 8]^7]$$

Notice that the resultant list is still distributed across the machine.
To generate the results shown is Figures 5-18 & 5-19, this program was applied to a list of length 16:

[32] That is, if the input lists are not the same size and / or reside on a different set of processors.

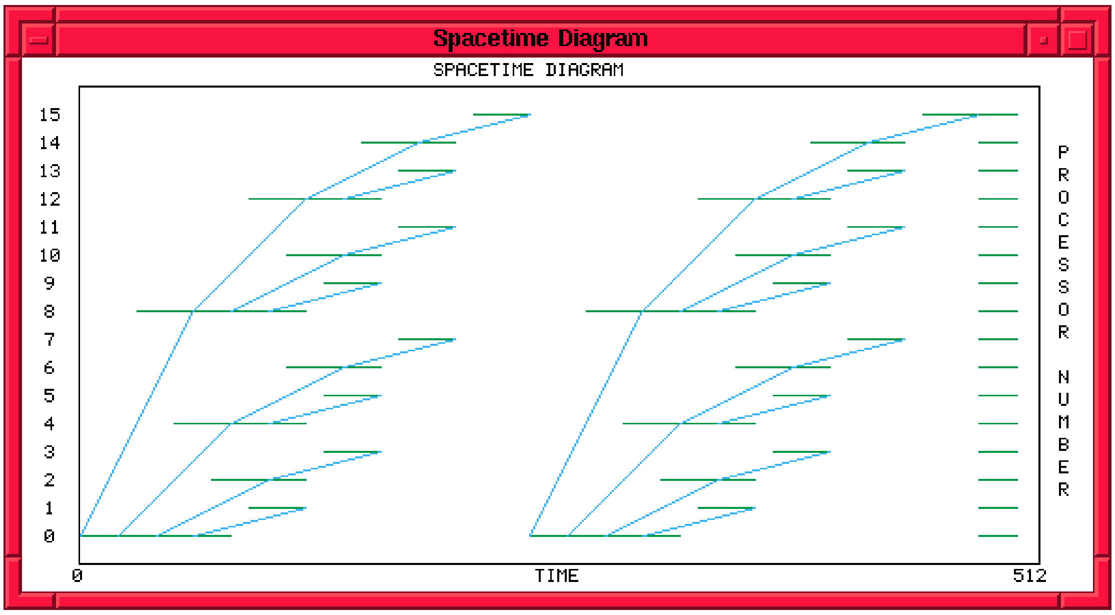


Figure 5-19: Space-Time diagram of Parallel Zip

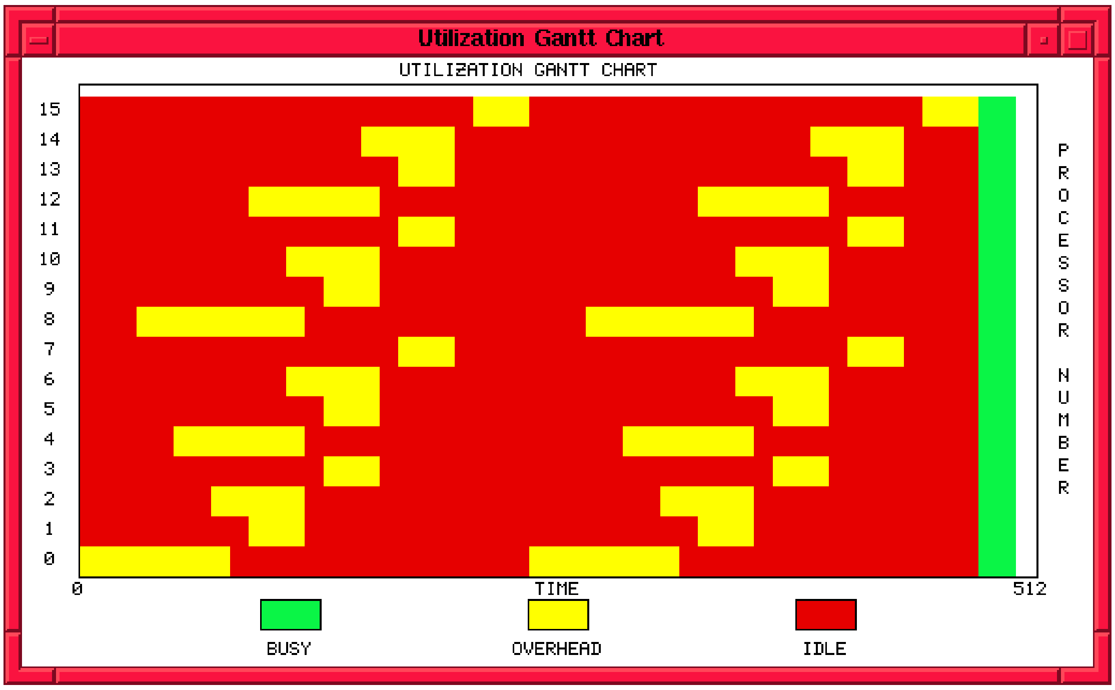


Figure 5-20: Gantt chart of Parallel Zip

The two spilt operation which distribute the input lists across the machine are clearly visible in both the space-time diagram and the Gantt chart. The Zip operation itself starts at the end of the second split operation as indicated by the vertical strip of busy time on the right hand side of the charts.

5.8 P_Scan

The schematic diagram below illustrates the operation of P_Scan for a list which has been distributed over eight processors. For the purposes of illustration, the exapmple shown is for a list of integers from 1 to 8 with the scan being performed with the addition operator. The general communication patterns, however are equally applicable to a scan performed with any suitable associative operator.

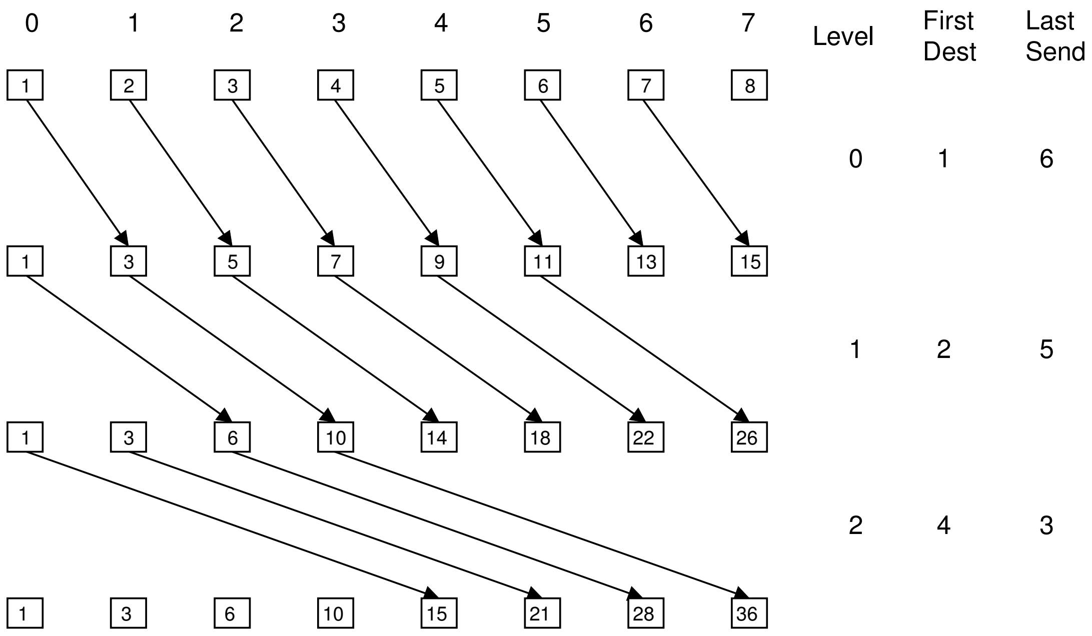


Figure 5-21: Schematic of Parallel Scan

The operation proceeds in $\log_2 N$ steps, called levels, where N is the length of the list. **In contrast to the case of P_Reduce, the starting level is zero rather one**. Starting from level zero:

- Do
    - First_Dest = 2^{Level} - as shown in the diagram
    - Current_Dest = First_Dest
    - Last_Sender = Length_Input_List - (First_Dest + 1)
    - Exit when Last_Sender <= 0
    - Source = 0
    - While Current_Dest <= Last_sender do

```
        • Initiate send from Source to Current_Dest by calling
          N_Send_D.
        • Item at Current_Dest = (Item at Source ⊕ Item at
                                              Current_Dest)
        • Current_Dest = Current_Dest + 1
    • Level = Level + 1
```

where ⊕ is the associative operator with which the scan was called.

To illustrate the operation of parallel scan, the following code was used:

```
$Flatten = P_Map (B_Reduce (B_Op (B_Conc) ) )


 B_Comp (P_Scan( B_Op ( B_Plus)))
         (B_Comp ($Flatten) (P_Split (B_Num 16) (B_Num 0)))
```

$Flatten converts a distributed 'list of lists' into a distributed list. For example:

$Flatten [[1]^0, [2]^1, [3]^2, [4]^3] = [1^0, 2^1, 3^2, 4^3]$

The flattening of the sub-lists makes reduce or scan with an arithmetic operator, such as plus, possible.

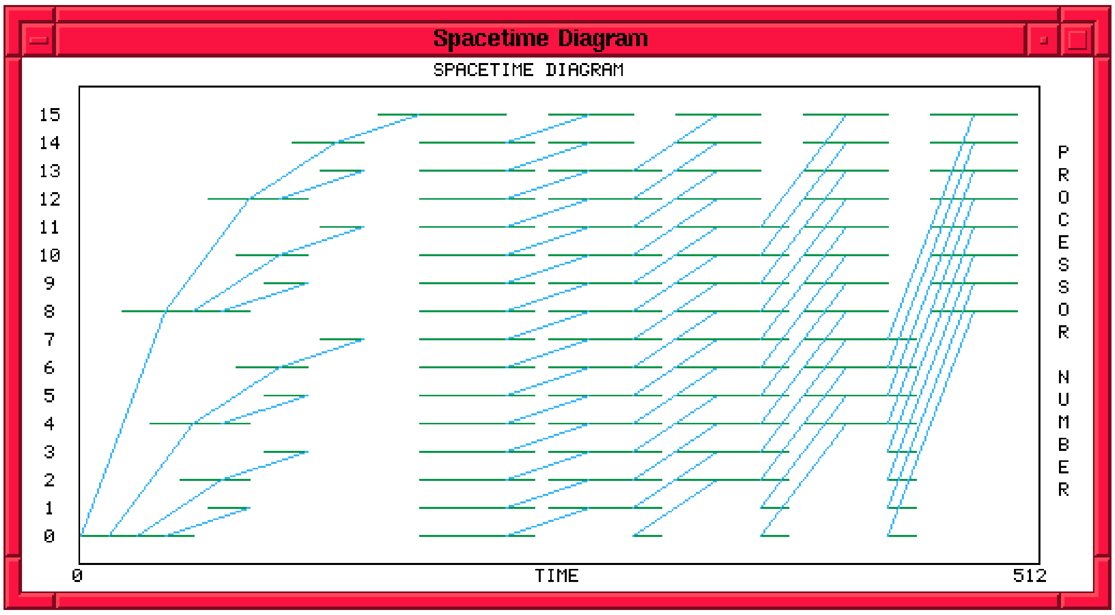


Figure 5-22: Space-Time Diagram for Parallel Scan with Plus

The communication patterns in the Space-Time diagram closely resemble those of the schematic (shown in Figure 5-21) except that this example is for 16 processors, rather than 8. Note that the left portion of the diagram shows the, now familiar, split operation which distributed the data over the machine.

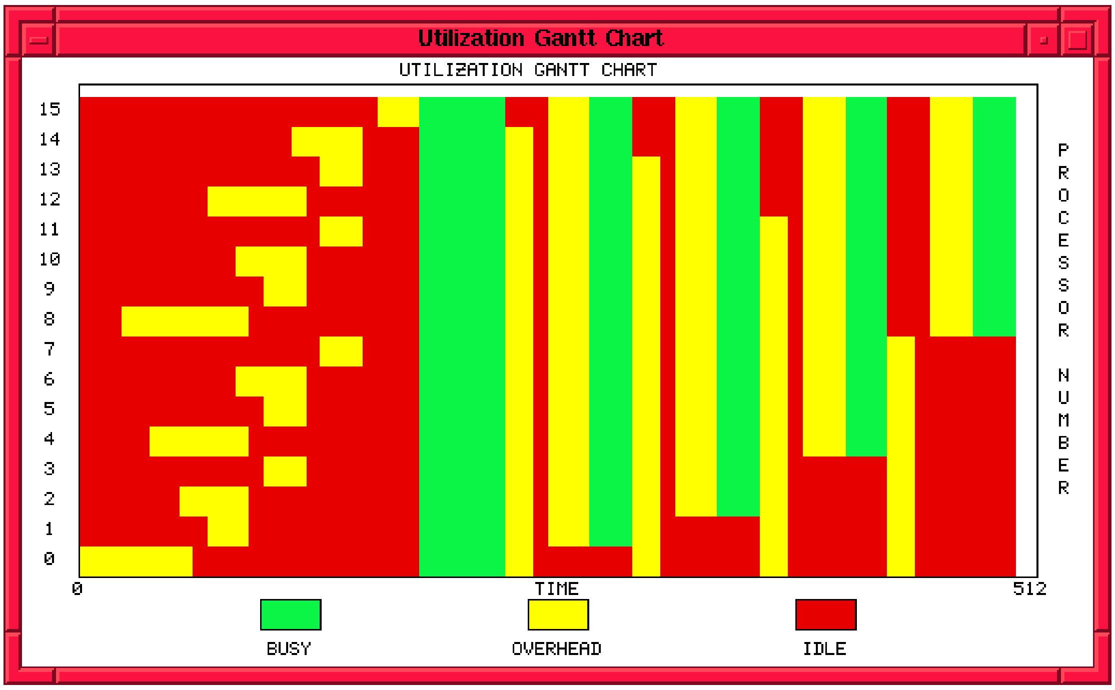


Figure 5-23: Gantt Chart of Scan with Plus

Looking now at the Gantt chart it can be seen that the first strip of busy time is wider than the rest; this is because it reflects the execution time of the flattening operation which occurs prior to the commencement of the scan. The busy times at each level of the scan remain constant because the operator used, namely B_Plus, executes in constant time regardless of the input values, provided they are of the same data type. In addition, as the scan proceeds, the data size remains the same which is reflected in the fact that the communication overheads remain constant.

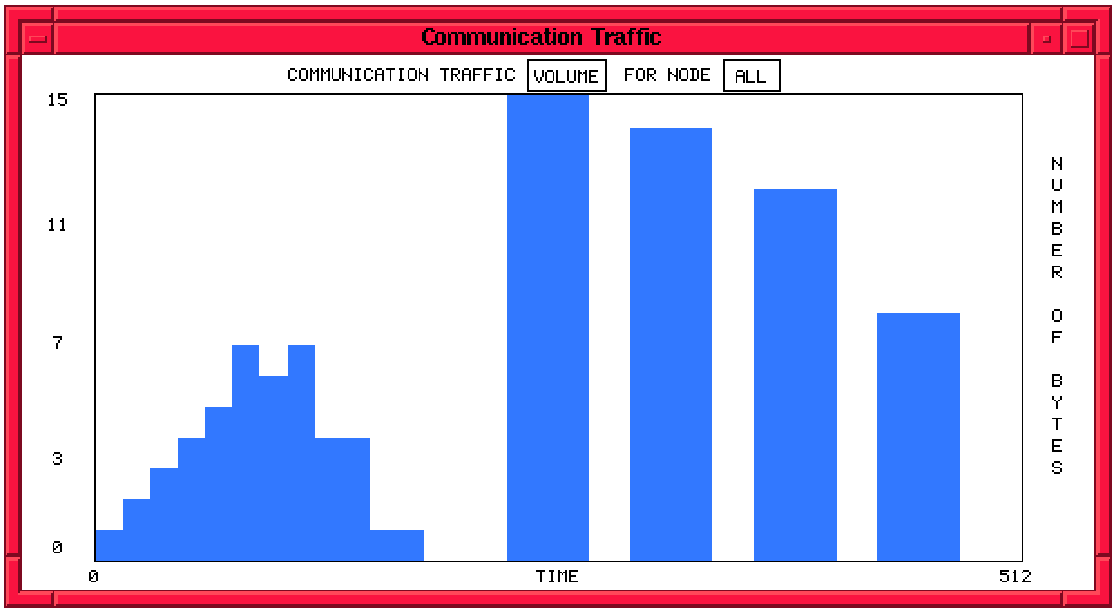


Figure 5-24: Traffic Volume for Scan with Plus

The Traffic volume chart shows that, like P_Reduce, the communications for P_Scan occur in a series of bursts which are separated by gaps during which computation occurs. In this example, because all messages are one byte long, the number of bytes during each burst, as indicated in Figure 5-24, equals the number of messages involved at that level. Reference to Figure 5-22 will show that there are 15 messages at the first level, 14 at the second, 12 at the third and 8 at the final level.

The nature of the operator used with scan can have an appreciable effect on the behaviour of the program as reflected in the following example which performs a parallel scan with the concatenation operator:

B_Comp (B_Scan (B_Op (B_Conc)))

(P_Split (B_Num 16) (B_Num 0))

A scan with concatenate produces a list of the initial sub-lists of the input so, for example, with the list $[1^0, 2^1, 3^2, 4^3]$, the following list would be produced:

$$[\ [1]^0, [1, 2]^1, [1, 2, 3]^2, [1, 2, 3, 4]^3\ ]$$

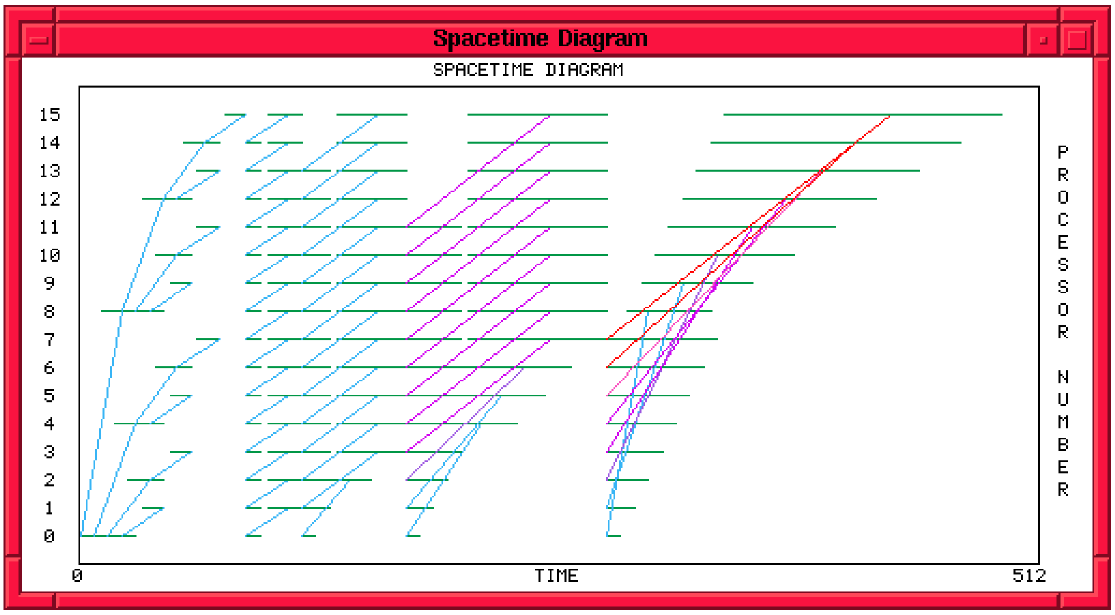


Figure 5-25: Space-Time Diagram for Scan with Concatenate

In contrast to the previous example, were the addition operator was used, concatenate results in an increased message size as the scan progresses. This is particularly noticeable with the increased skewing effect and change in the colour of the communications lines from blue to purple through to orange.

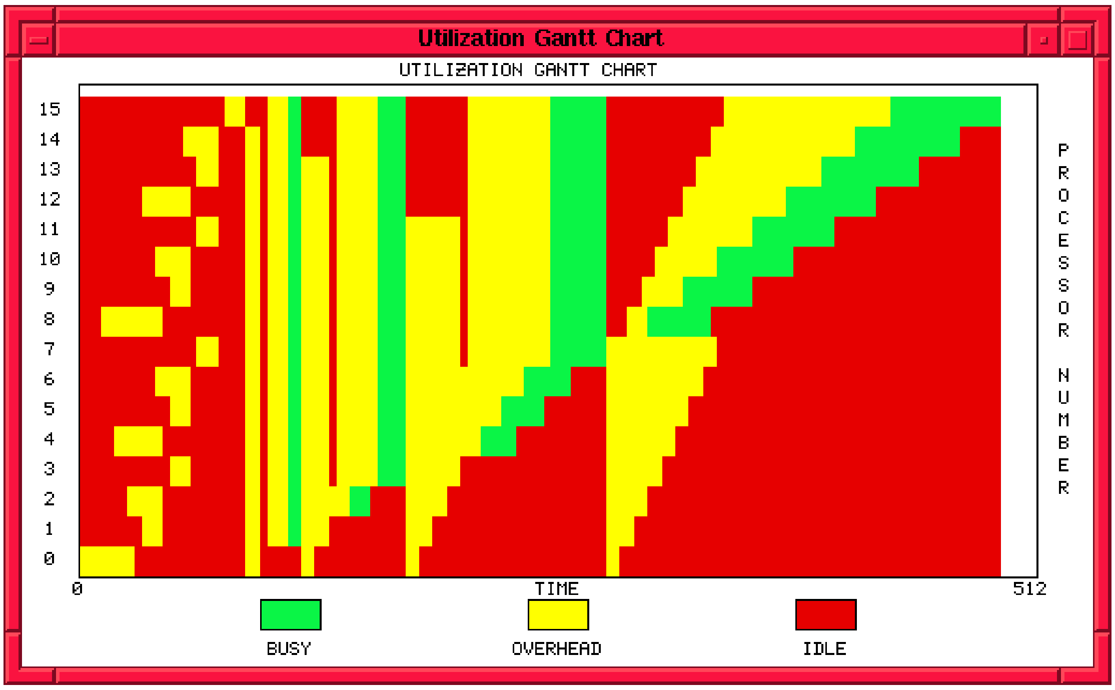


Figure 5-26: Gantt Chart of Scan with Concatenate

The Gantt chart of the same program highlights the skewing effect caused by the increasing communication overheads due to increasing message size as the operation proceeds. Furthermore, the busy time at each level increases because the execution time for concatenate is proportional to the size of its arguments.

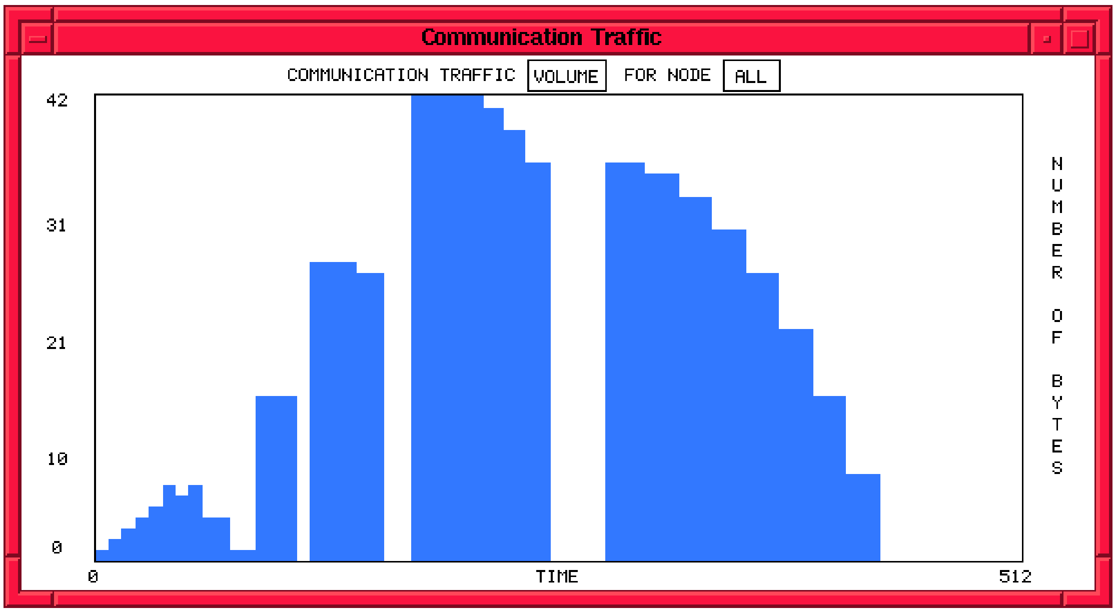


Figure 5-27: Traffic Volume for Parallel Scan with Concatenate

Analysis of the traffic volume shows that the amount of data being communicated during each burst rises from 15 for the first level to a peak value of 42 during the third and then down to an average of around 20 during the final burst. The ‘rounding’ of the tops of the last three bursts reflects the skewing of the communication patterns noted earlier.

5.9 P_Project

Parallel Project effects a permutation of a distributed list. The implementation of P_Project required the development of the delayed send procedure as described in section 4.7.2.

Projects takes in a pair of arguments: {Input_List, Index_List}
where: Input_List is the list to be permuted and Index_List the list of indicies.
The project operation proceeds as follows:

- Destination is set to the tag of the first item in the input list.
- Synchronize all processor clocks
- For each Index in Index_List Do
    - Current_item = L_Index (Input_List, Index)
    - Source = tag of Current_Item
    - Initiate Send from Source to Destination by calling N_Send_D
    - Change tag on Current_Item to Destination
    - Append Current_Item to the result list
    - Destination = Destination + 1

The following code was used to illustrate the operation of parallel project:

```
$Make_List = B_Comp (B_Op (B_Iota)) (B_Con (B_Int 16))
```

This function creates the list [0, 1, 2, …, 15].

```
B_Comp (P_Project)
        (B_Alltup [ B_Comp( P_Split (B_Num 16) (B_Num 0))
                                ($Make_List), B_Id ] )
```

The above program applies P_Project to a tuple consisting of a distributed list (created by make_list and distributed by split), and the index list which is supplied as input to the program. For example, if the following index list were provided as input:

[I_1, I_2, …, I_{16}]

then the state of the program before the application of the P_Project would be:

P_Project { [0^0, 1^1, 2^2 …, 15^{15}], [I_1, I_2, …, I_{16}] }

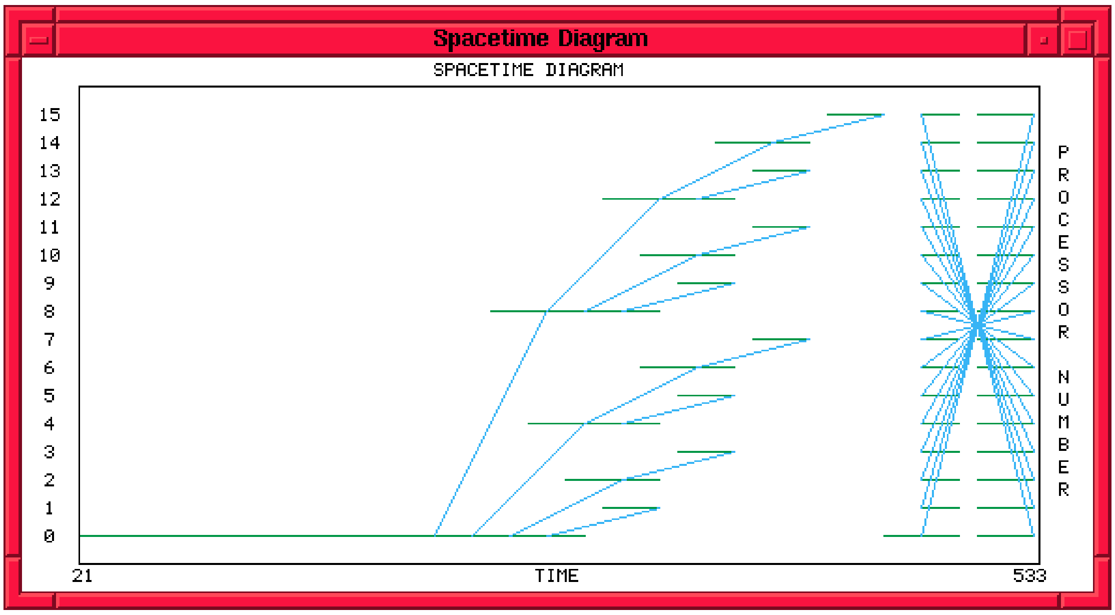


Figure 5-28: Space-Time Diagram for List Reversal

The above figure shows the distribution of the data across the machine followed by the parallel Project communications pattern for a full reversal of the list. Such a result was achieved by using an index list of the form:

[15, 14, 13, …, 2, 1, 0]

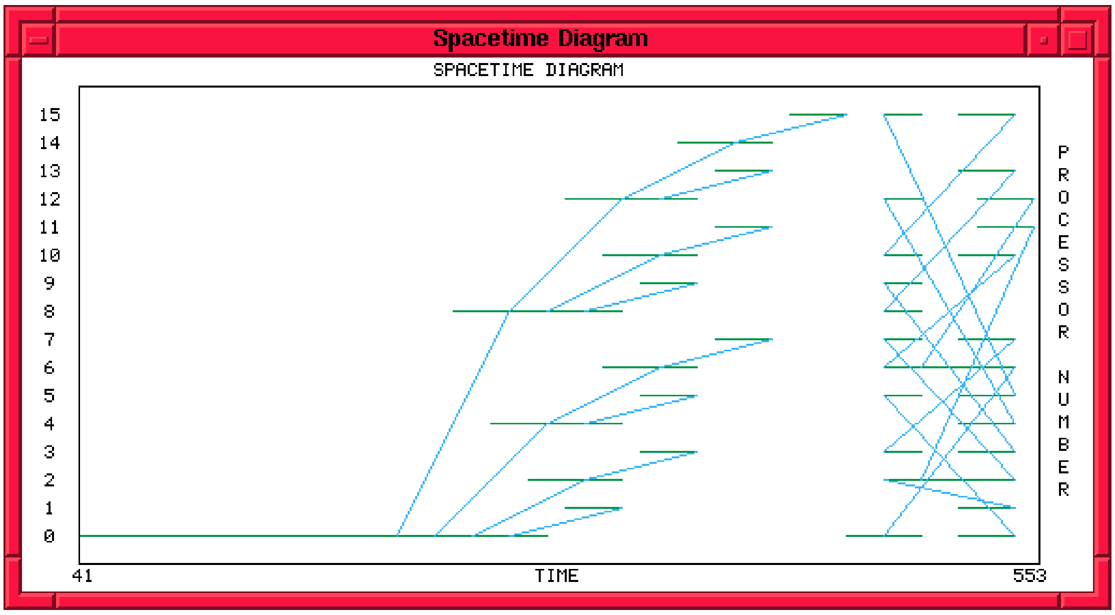


Figure 5-29: Space-Time Diagram for Random Permutation of List

Figure 5-29 illustrates the behaviour of P_Project when conducting a 'random' permutation of the list.

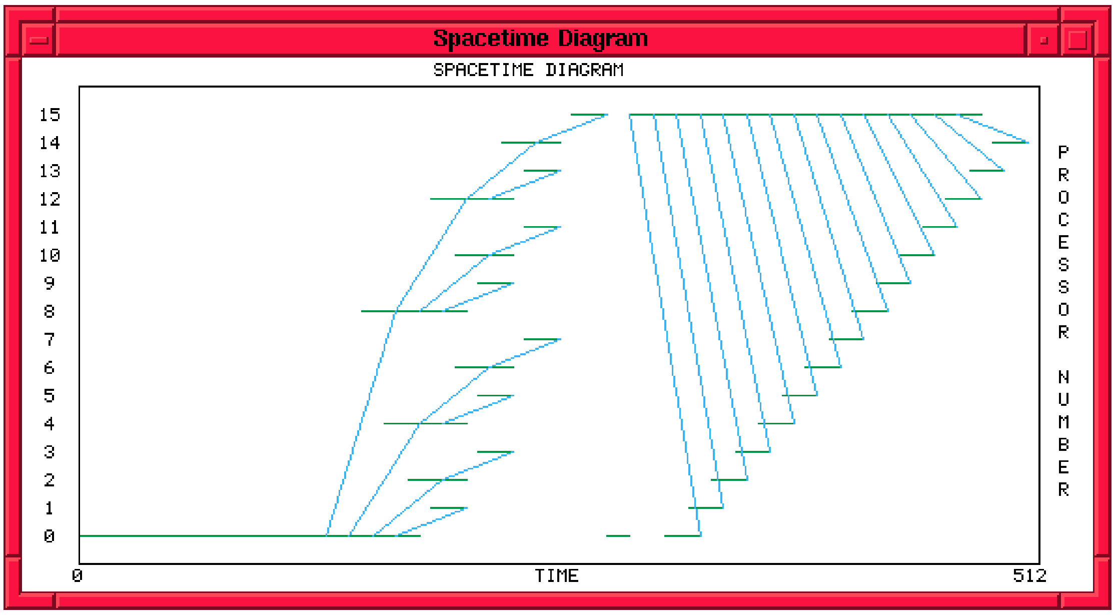


Figure 5-30: Space-Time Diagram of 'Broadcast' with Project

Finally, Figure 5-30 shows P_Project sending the same item (at list position 15) to every other processor. This method of broadcasting is very inefficient due to the overhead involved in sending the 15 separate messages (containing the same data) from the one processor.

6. Extensions to the Network Model

6.1 Overview

The basic network model, although yielding useful results, was deficient in two main areas: network topology and the effects of congestion were not taken into account. This chapter will outline the extensions made to the basic network model to address the above deficiencies.

6.2 Modelling of Network Topology

In the basic model, the transport time through the network was assumed to be constant regardless of the inter-processor distance between source and destination. At best, this model could have been modified to accommodate a 'fixed' mathematical function[33] which would return the transport time for a given source-destination pair but such a solution does not provide the generality needed to model a wide range of networks.

6.2.1 Representation of Network Topology

The extended model represents a network as a weighted directed graph with the weights on each link reflecting the relative bandwidth of each segment. The graph representing the network to be used by the simulator is created from an ASCII configuration file. Reference to the following example will illustrate how a network may defined using the file.

Figure 6-1 shows a schematic diagram for a four processor switched binary tree network with the processors being the rectangular boxes and the circles representing the internal nodes (switches). The processors are numbered first, starting from zero, after which the internal nodes may be numbered in any sequence. The number on each graph edge represents the relative bandwidth.

[33] i.e. We could, for example, use a function such as Transport_Time = (LOG_2 (ABS(Dest - Source)) to model a tree network.

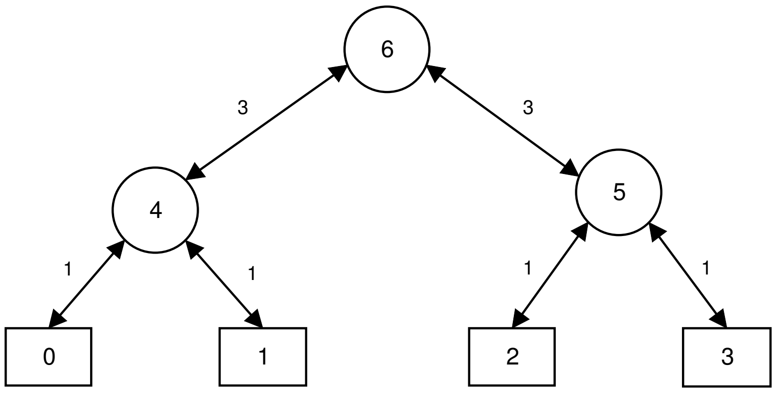


Figure 6-1: Binary Tree Network Schematic

To indicate the bi-directional nature of the links, the edges are represented by double-ended arrows. This graph would be expressed by the following configuration file:

```
7
0  4  1  -1
1  4  1  -1
2  5  1  -1
3  5  1  -1
4  0  1   1  1   6  3   -1
5  2  1   3  1   6  3   -1
6  4  3   5  3   -1
```

where the first number, 7, indicates the total number of nodes in the graph. It can be seen that each line consists of a node number and then a list of 'pairs' representing the edges; **each pair consists of the node to which the edge leads and the weight**. A '-1' is used to terminate the list for each node.

When initialized, the simulator uses the definition file to create the graph data structure, representing it in the form of an adjacency matrix.

6.2.2 Route Matrix

To facilitate topologically dependent cost modeling, a value for the cost of sending a message between any two given processors is required; in addition, congestion modeling will require a path-list to indicate the route by which a packet should travel when moving through the network from the source to the destination processor. This information is stored in a 'Route_Matrix' which contains a cost and path-list record for each source - destination combination. In the current implementation, 'shortest path routing' is used and therefore the Route_Matrix is generated, prior to the simulation commencing, by running Dijkstra's Algorithm [21] for each source-destination combination.

6.3 Modelling of Congestion

6.3.1 Link Contention

In a real network, there cannot be more than one packet traveling through a particular network link at any given time; congestion is caused when two or more packets contend for the same network link. The method employed to model such behaviour in this extended implementation is to keep a record of the time-intervals 'occupied' by packets, for each link, and ensure that no two such time-intervals overlap. A data structure called the Occupancy_Matrix stores, in the form of a linked list[34], a 'time-line' for each link in the network. Symbolically, each link record in the Occupancy_Matrix can be viewed as follows:

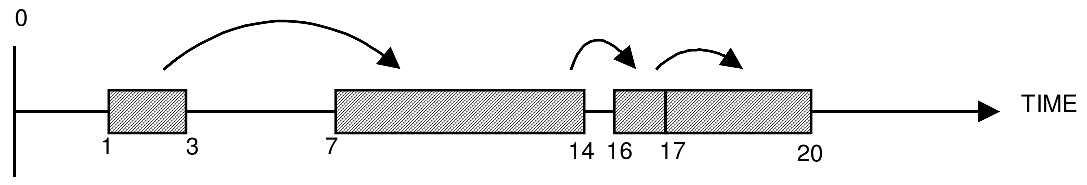


Figure 6-2: Occupancy Matrix Time-Line

The horizontal line represents time. Each hashed region indicates a period during which the link is occupied with the beginning and end times of each such interval indicated below each region. (In this example, four packets have already traveled through this particular link). When a packet 'arrives' at a link, the following steps occur:

- Let T_{INJ} be the time at which the packet arrives at the beginning of link.
- The travel time of the packet through the link, T_{TRAV}, is then calculated using the size of the packet and the bandwidth of the link.

[34] The linked list implementation avoids the necessity to descretize the time domain into a finite number of intervals as would be necessary with an array.

- The time-line is now scanned looking for the earliest interval, greater than or equal to T_{INJ}, which is at least T_{TRAV} wide.
- When the required interval is found, it is "marked out" on the time-line by adding another record to the linked list. The time at which the packet arrives at the next link is then set to be the end of the interval.

Referring back to the example shown in Figure 6-2, if a packet arrives at $T_{INJ} = 2$ and based on its size, $T_{TRAV} = 3$, then the earliest suitable interval after T_{INJ} is 3 to 6 as shown in Figure 6-3. In this case, the packet has been delayed by one time unit, due to congestion, arriving at the next link at time = 6 instead of 5.

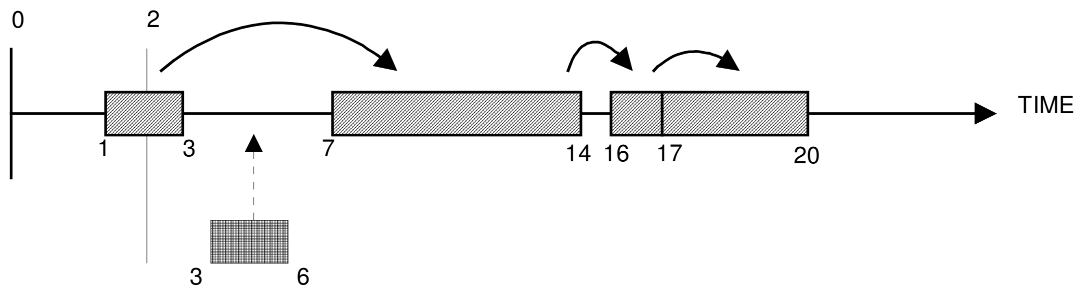


Figure 6-3: Insertion of New Time Interval into Matrix Time-Line

6.3.2 Network Package Extensions to Model Congestion

The above scheme provides the mechanism by which link contention is taken into account. To enable the above modeling to be incorporated into the simulator, the following procedure is used:

```
function Transmit_Packet(T_Inject : Float ;
                         Source, Dest, Size : Integer;
                         Rm : Route_Matrix_Ty ;
                         Om : Occupancy_Matrix_Ty) return Float;
```

Transmit_Packet, when called, uses the path_list to simulate the passage of the packet through each network link as it travels from source to destination, returning the time at which it arrives. Do_Receives was modified to call Transmit_Packet for each pending receive to ascertain the arrival time of the packet. All procedures that use

N_Send_D[35] (an hence also Do_Receives) namely, P_Scan and P_Project, automatically make use of the extended model. Any procedure which uses Delayed Send will automatically make use of the congestion model. Thus, those operators that already used Delayed Send, namely, P_Scan and P_Project automatically made use of the extended network model. However, congestion modeling for P_Reduce operations was incorporated by modifying it to use Delayed Send rather than Normal Send. To modify P_Split and P_Distl to use Delayed Send is difficult because of the way in which they are implemented and, because time did not permit, congestion modeling for these functions was not incorporated at this stage.

6.4 Examples using the Extended Network Model.

To illustrate the way in which the extended network model affects the simulation results, examples using P_Scan and P_Project will follow:

Figures 6-4 and 6-5 show the result of a scan with plus operation conducted using the basic and the extended network models respectively. For the extended model, a binary tree network was used.

[35] See section 4.7.2.

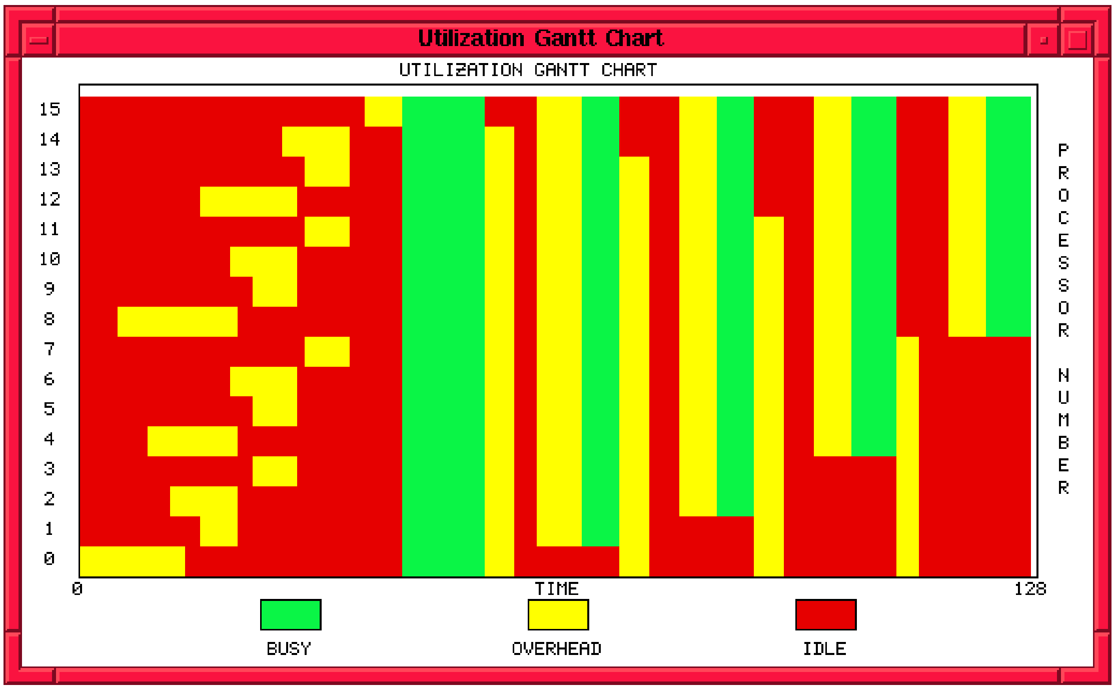


Figure 6-4: Gantt Chart of Scan using Basic Network Model

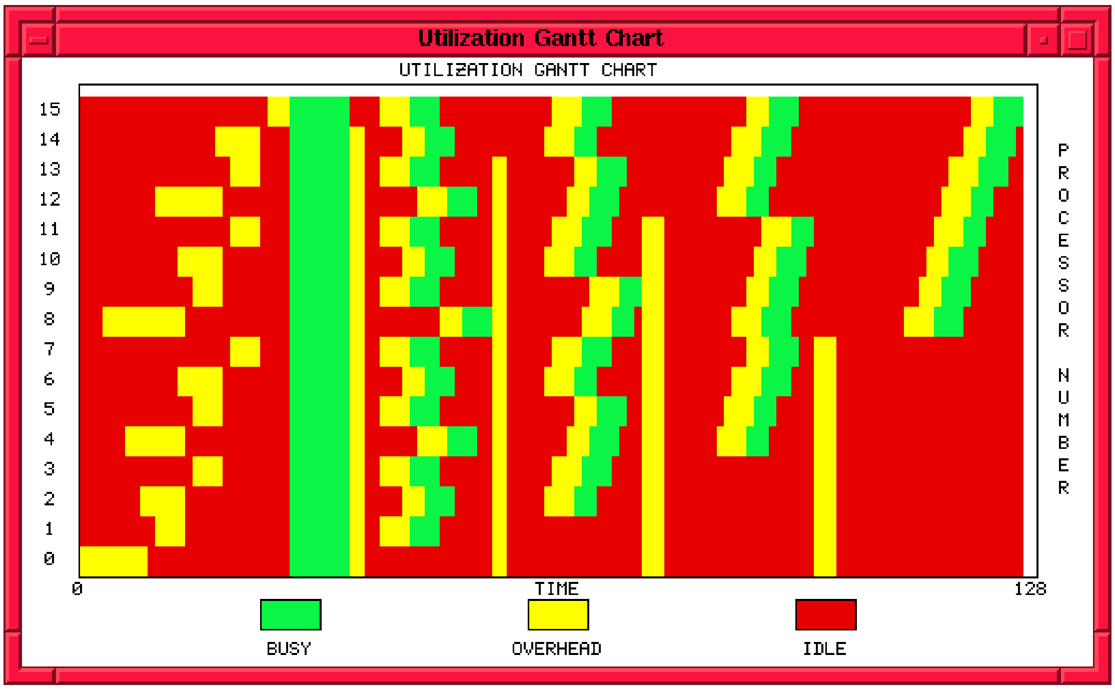


Figure 6-5: Gantt Chart of Scan on Binary Tree Network

The extended model has a significant effect on the nature of the results produced. There are two effects at work: the topological dependency of the path length between processors (imposed by the cost-model) and the congestion modeling. The effects of the cost model are best seen by considering the receives occurring at the first level of the scan: three messages, in particular, arrive later than all the others, namely the communications from processors: 3 to 4, 7 to 8, 11 to 12. This is due to the differences in communication path lengths as illustrated in the Binary Tree Network shown in Figure 6-6. The effects of congestion are distributed through the diagram but are most evident during the last stage of the scan. During this stage, processors 0 to 7 are transmitting to processors 8 to 15; the path lengths for these communications are the same therefore the staggering effect displayed is due purely to congestion. Figure 6-6 shows the state of the network during the final stage clearly showing that congestion should be expected, especially near the root of the tree.

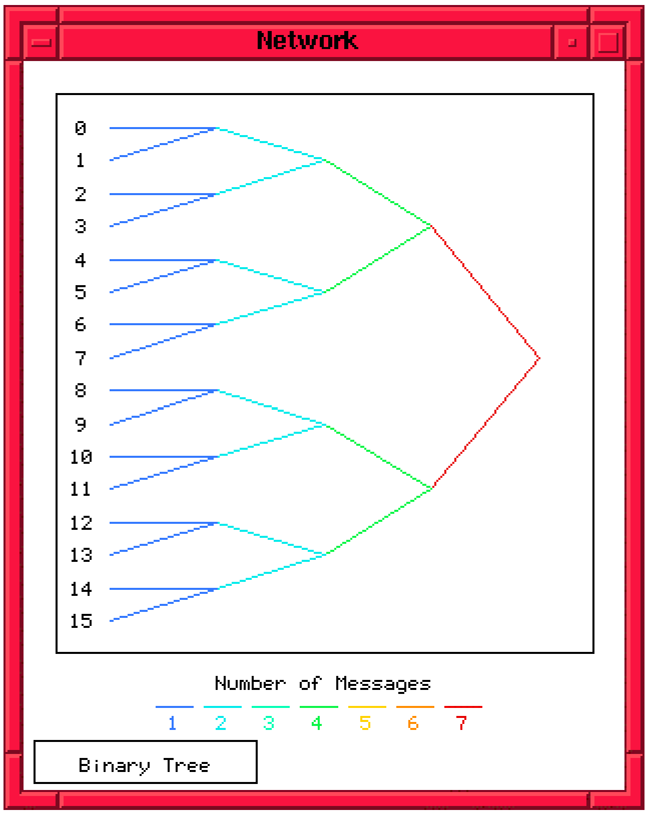


Figure 6-6: Network Diagram For Last Stage of Scan on a Binary Tree Network

To illustrate the effects that various choices of network topology can have on the execution of a program, parallel Project affecting the reversal of a list on a Binary Tree and a 4x4 Mesh network are compared below:

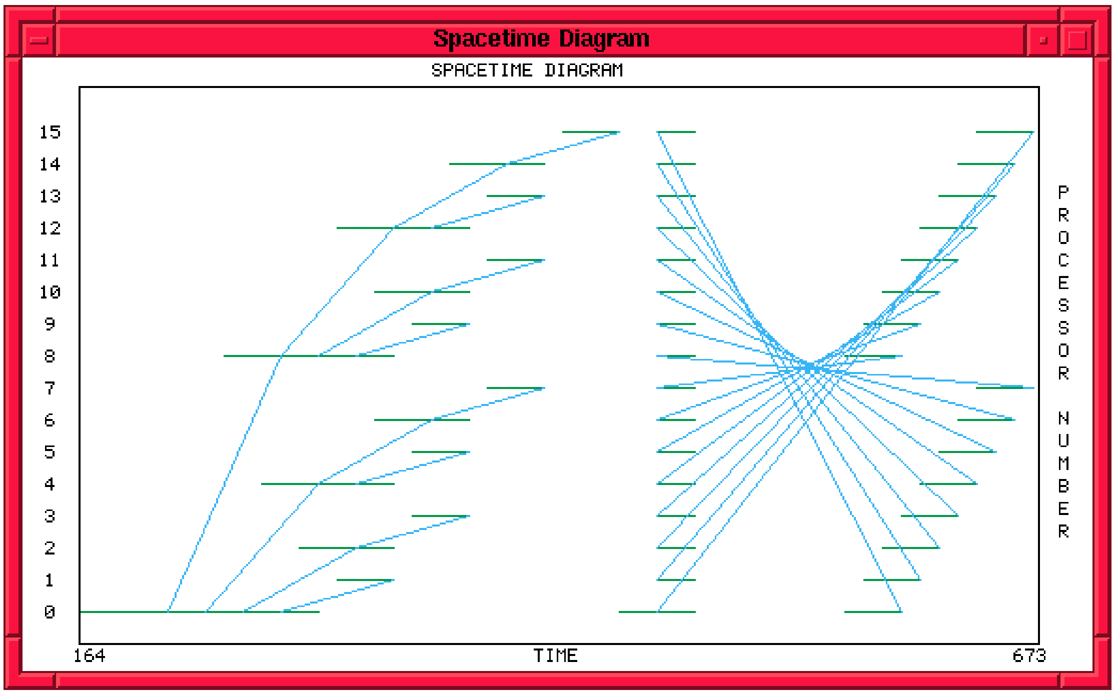


Figure 6-7: Space-Time Diagram for Project on Binary Tree Network

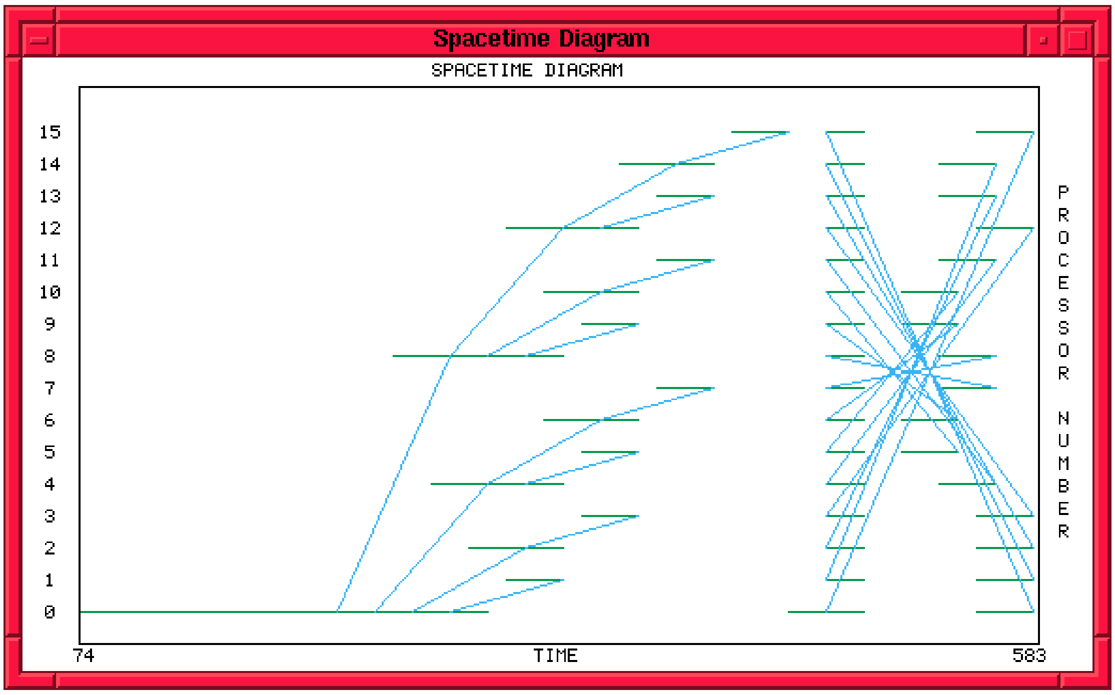
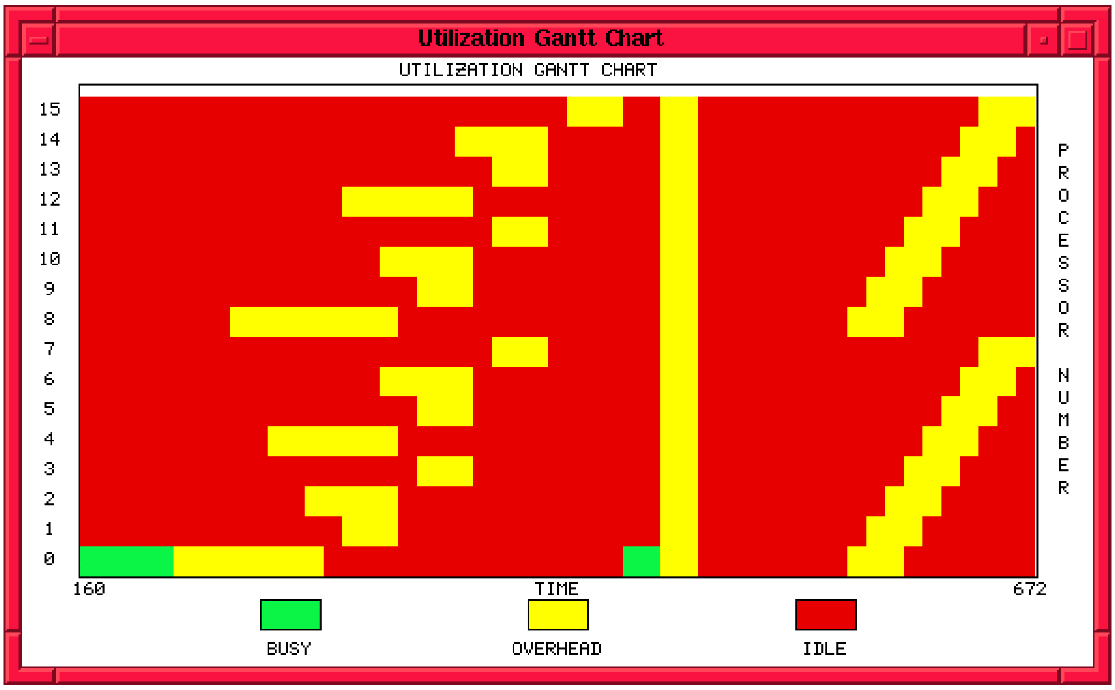


Figure 6-8: Space-Time Diagram of Project on 4x4 Mesh Network

Comparing the Figures 6-7 & 6-8, the stagering effect, due to congestion, is more pronounced in the case of the Binary Tree.



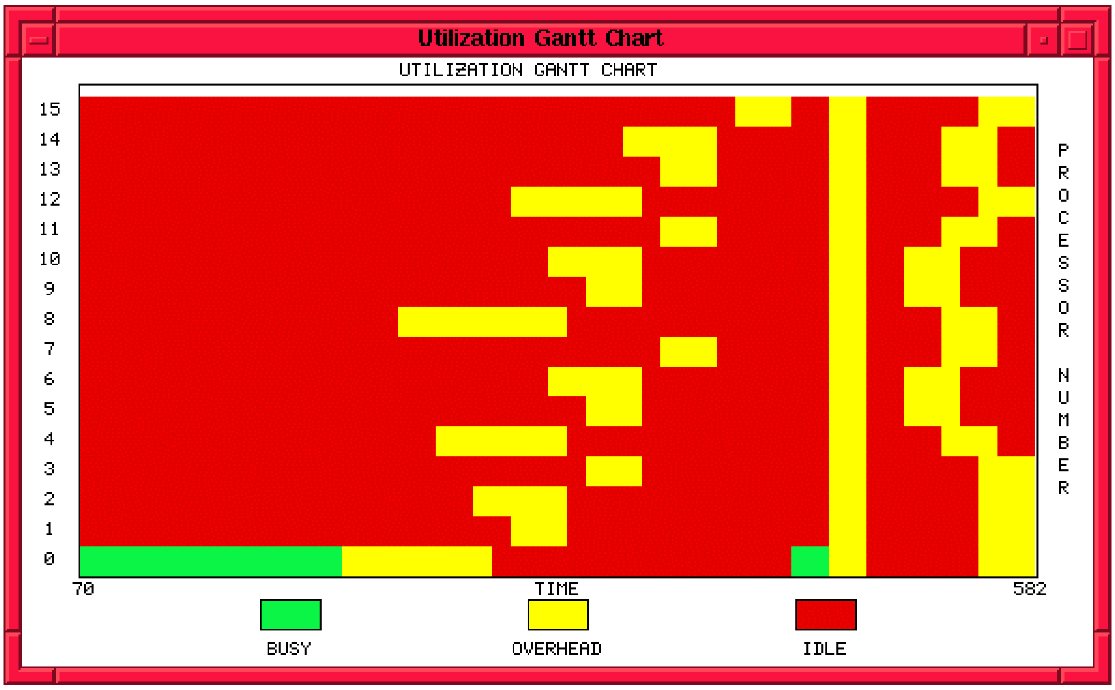


Figure 6-9: Gantt Chart for Project on 4x4 Mesh Network

Finally, comparison of the two Gantt charts further emphasizes the differences that the congestion model makes.

7. Conclusions

The simulator constructed in this project provides a solid basis for the simulation of parallel BMF code. The extensions to the network model to take into account differing network topologies, and the effects of congestion proved to be worthwhile. This network modeling is also capable of being enhanced in the future without extensive modifications being made to the rest of the system. Although only relatively small programs were simulated, indications are that the tool should be useful, in the future, to serve as a means of developing targeting strategies when building BMF implementations on real machines.

8. APPENDIX A

8.1 BNF Description of BMF Source Code

```
Program ::=  <b_exp> <v_data>
Function_Definition ::= $<Identifier> = <b_exp>

<b_exp> ::= $<Identifier> |
            B_Id |
            B_Comp ( <b_exp> ) ( <b_exp> )  |
            B_Con  ( <b_con> )  |
            B_Alltup [ <b_alltup> ] |
            B_Allvec [ <b_allvec> ] |
            B_Op  ( <b_op> )     |
            B_Addr  ( <b_num> ) ( <b_num> ) |
            B_Map  ( <b_exp> )     |
            B_Reduce ( <b_exp> )  ( <b_exp> ) |
            B_Scan  ( <b_exp> ) ( <b_exp> )  |
            B_Zip  ( <b_exp> ) |
            B_Split( <B_Num> ) |

            P_Map  ( <b_exp> ) |
            P_Reduce ( <b_exp> ) |
            P_Scan  ( <b_exp> ) |
            P_Zip  |
            P_Project |
            P_Split  ( <b_num> ) ( <b_num> ) |
            P_Distl

<b_alltup> ::= <b_exp> , {<b_exp>}*

<b_allvec> ::= e | <b_exp> {,<b_exp>}*

<b_num> ::= B_Num <Integer>

<b_op> ::=  B_Plus | B_Times | B_Minus | B_Divide | B_Uminus |
            B_Neg |
            B_And | B_Or | B_Eq | B_Gt | B_Lt |
            B_Length | B_Index | B_Iota |
             B_Distl | B_Project

<b_con> ::= B_int <num> | B_real <num> | B_True | B_False

<v_data> ::= <integer> | <real> | <boolean> | [ <v_vector> ] |
            { <v_tuple> }

<v_vector> ::= e | <v_data> {, <v_vector> }*

<v_tuple> ::= <v_data> {, <v_tuple>}*

<num> ::= <Integer> | <Real>
```

9. Bibliography

[1] Ahmer Ingrid, *Interpreter for Expressions in the Bird-Meertens Formalism*, Honours Thesis, Department of Computer Science, University of Adelaide, Australia, November 1996

[2] Alexander Brad, *Overview of BMF*, Technical Report, Department of Computer Science, University of Adelaide, Australia, September 1997

[3] Alexander Brad, *Mapping Adl to Bird-Meertens Formalism*, Technical Report, Department of Computer Science, University of Adelaide, August 1994, available from: http://www.cs.adelaide.edu.au/~adl/pubs.html

[4] Alexander Brad, Engelhard Dean and Wendelborn Andrew, *An Overview of the Adl Language Project* in Proceedings Conference on High Performance Functional Computing, Denver, Colorado, April 1995.

[5] Almasi George S. and Gottlieb Allan, *HIGHLY PARALLEL COMPUTING*, second edition, The Benjamin/Cummings Publishing Company, Inc, Redwood City, California, USA, 1994

[6] Backus John, *Can Programming Be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs*, ACM Turing Award Lecture, Communications of the ACM, vol 21, August 1978

[7] Barnes J.G.P., *Programming in Ada*, fourth edition, Addisson-Wesley Publishing Company, Wockingham, Inc, England, 1994

[8] Bird Richard S., *Lectures on Constructive Functional Programming*, in Lectures on Functional Programming, in Constructive Methods of Computing Science, NATO ASI Series Vol. F55, pp 151 - 216, 1989

[9] Gibbons Jeremy, *An Intoduction to the Bird-Meertens Formalism*, Technical Report, Departement of Computer Science, University of Auckland, New Zealand, November 1994

[10] Heath Michael T. and Finger Jennifer E., *Visualizing the Performance of Parallel Programs*, IEEE Software 8(5), September 1991, pp 29-39.

[11] Heath, Michael T., *Performance Visualization with ParaGraph*, In Proceedings of the Second Workshop on Environments and Tools for Parallel Scientific Computing, pages 221-230, SIAM, Philadelphia, 1994.

[12] Heath Michael T and Finger, Jennifer E., Pa*raGraph: A Tool for Visualizing Performance of Parallel Programs*, Software and User manual available at http://www.csar.uiuc.edu/software/paragraph/index.html

[13] Hillis Daniel W. and Steele Guy L. Jr, *DATA PARALLEL ALGORITHMS*, Communications of the ACM, vol 30, December 1986, pp 1170 - 1183

[14] Hoogendijk Paul F., Backhouse, Roland C., *Relational programming laws in the tree, list, bag, set hierachy,* Science of Computer Programming 22, 1994, pp 67 - 73

[15] Roe Paul, *Derivation of Efficient Parallel Programs* in Proceedings of the 17th Annual Computer Science Conference, Australian Computer Science Communications, Vol 16, No 1, Christchurch, NZ, Janurary 1994

[16] Skillicorn David B., *Architecture-Independent Parallel Computation*, IEEE Computer, December 1990, pp 38 - 50

[17] Skillicorn David B., *Parallelism and the Bird-Meertens Formalism*, Technical Report, Department of Computing and Information Science, Queen's University, Kingston, Canada, April 1992.
available from: http://citeseer.nj.nec.com/skillicorn92parallelism.html

[18] Skillicorn David B., *The Bird-Meertens Formalism as a Parallel Model*, in 'NATO ARW: Software for Parallel Computation', Kowalik, J.S. and Grandinetti, L. editors, vol 106, Springer-Verlag NATO ASI, 1993.

[19] Skillicorn David B., *Bulk Data Types for Architecture Independance*, Technical Report presented to: British Computer Society, Parallel Processing Specialist Group, May 1994

[20] Spivey Mike, *A Categorical Approach to the Theory of Lists*, Programming Research Group, Oxford University, Oxford, England, June 1987

[21] Tannenbaum Andrew S., *COMPUTER NETWORKS*, third edition, Prentice Hall International UK, London, England, 1996

[22] Worley Patrick H., *A New PICL Trace Format*, Technical Report ORNL/TM-12125, Oak Ridge National Laboratory, Mathematical Sciences Section, Oak Ridge, Tennessee, USA, September 1992

The following bibliography, although not specifically cited, were used as background reading.

Ajmone Marsen M., Balbo G. and Conte G., *PERFORMANCE MODELS of MULTIPROCESSOR SYSTEMS*, Massachussets Institute of Technology, MIT Press, London, England, 1988

Chuan-lin Wu and Tse-yun Feng, *Tutorial: Interconnecting Networks for PARALLEL AND DISTRIBUTED PROCESSING*, IEEE Computer Society Press, Washington, D.C., USA, 1984

Diller Antoni, *COMPILING FUNCTIONAL LANGUAGES*, John Wiley & Sons Ltd, Chichester, England, 1988

Kelly Paul, *Functional Programming for Loosely-Coupled Multiprocesors*, Pitman Publishing Company, London, England, 1989

Panda Dhabaleswar K., Basak Debashis, Dai Donglai, Kesavan Ram, Sivaram Rajeev, Banikazemi Mohammad and Moorthy Vijay, *Simulation of Modern Parallel Systems: A CSIM-Based Approach*, Technical Report, Department of Computer Science, The Ohio State University, Columbus, USA, July 1997

Peyton Jones Simon L., *The Implementation of Functional Programming Languages*, Prentice Hall International (UK) Ltd, Herdfordshire, UK, 1987

Reed Daniel A. and Fugimoto Richard M., *Multicomputer Networks: Message Based Parallel Processing*, Massachussets Institute of Technology, MIT Press, London, England, 1989

Sherson Isaac D. and Youssef Abdou S., *Interconnection Networks for High-Performance Parallel Computers*, IEEE Computer Society Press, Los Alaminotis, California, USA, 1994

Thompson Simon, *Miranda The Craft of Functional Programming*, Addison-Wesley, Harlow, England, 1995

Zomaya Albert Y., Editor, *Parallel & Distributed Computing Handbook*, McGraw-Hill, New York, USA, 1996