

Parallelizing Compiler for Matrix Expressions

by
José M. Vidal

SUBMITTED TO THE DEPARTMENT OF
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE
DEGREE OF
BACHELOR OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1990

© 1990 José M. Vidal

The author hereby grants to MIT permission to reproduce and to distribute copies
of this thesis document in whole or in part.

Signature of Author: _____

Department of Electrical Engineering and Computer Science
May 21, 1987

Certified by: _____

Anant Agarwal
Assistant Professor, Department of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by: _____

Leonard A. Gould
Chairman, Department Committee on Undergraduate Theses

Parallelizing Compiler for Matrix Expressions

by

José M. Vidal

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 1990 in partial fulfillment of the
requirements for the Degree of
Bachelor of Science in Computer Science and Engineering

ABSTRACT

We study the automatic compilation of matrix expressions on a fully connected, shared memory parallel machine such as the Encore Multimax. For this task we have developed a parallelizing compiler. This compiler takes as input a matrix expression written in Lisp syntax and outputs a Mul-T program that will execute the desired expression, given an arbitrary number of processors. Several different scheduling techniques are used for this task. Tests have been run to compare the performance of these techniques and the results of these tests are presented. The software was designed so as to allow the easy addition of new operations and scheduling techniques.

Thesis Supervisor: Anant Agarwal

Title: Professor, Department of Electrical Engineering and Computer Science

Contents

1	Introduction	1
2	Problem Specification and Solutions	3
2.1	Problem Specification	4
2.2	Possible Solutions	5
3	Software structure	10
3.1	Overall Design	11
3.2	Operations	13
3.2.1	Matrix addition and subtraction	13
3.2.2	Matrix multiply	14
3.2.3	LU decomposition	15
3.2.4	Matrix inverse	15
3.3	Data Abstraction	17
3.4	Parser	19
3.5	Code Generator	21

<i>CONTENTS</i>	4
4 Testing	28
4.1 Results	29
5 Conclusion	41
A Description of the files	43

Chapter 1

Introduction

In the recent years there has been an increased proliferation and preoccupation with parallel architectures. This trend is evident in the appearance of numerous parallel computers such as the Connection Machine, Encore Multimax, J-Machine and others. These machines hope to increase their computing power by having several processors acting in parallel. The idea is that each processor will execute a small part of the program so that the whole program can be finished in a short amount of time, even if the individual processors are not that fast. The task of assigning jobs to each processor is usually left up to the programmer. This task can get very unpleasant.

We consider the special case of scheduling a matrix expression. The compiler developed as part of this thesis takes as input a matrix expression and outputs an executable Mul-T program which explicitly schedules the different operations within the matrix expression. The operations supported by the compiler include matrix

addition, multiplication, subtraction, LU decomposition and inverse (although these last two do not perform row exchanges). We use several scheduling techniques. Each one of these has been implemented, and has been tested on several examples. The results of these tests show which scheduling techniques work better under specific circumstances.

These programs were designed as research tools to test the different schedulers but they can also be reliably used to execute matrix expressions if this is so desired. As they stand, these programs will be a great help to anyone who wants to solve matrix equations on the Encore Multimax or some other parallel machine that runs Mul-T.

Chapter 2

Problem Specification and Solutions

As was mentioned, this thesis proposes to implement a solution to the problem of scheduling a matrix expression given that we have an arbitrary number of processors. In this chapter we will present the specific scheduling problem of executing a matrix expressions on a multi-processor architecture. We will also explain the different scheduling techniques used for solving this problem.

2.1 Problem Specification

The problem can best be illustrated with an example. Suppose that one desires to calculate the value of formula 2.1 using p processors.

$$((A + A) + (B + B)) \times ((C + C) \times (D + D)) \quad (2.1)$$

Here A , B , C and D are matrices and the operations performed on them are matrix operations. All the matrices are different. The same name is used only for historical reasons. Finally, all operations are explicitly performed, no algebraic simplifications are used. When calculating the expression it is clear that a partial order must be followed. For example, the first multiplication, the one outside the parenthesis, can only be performed after all other operations have finished since it needs the values returned by these. Another factor that must be taken into account when scheduling this expression is the problem of how many processors are going to be allocated to each operation. We can then summarize the problem as that of scheduling all operations in a matrix expression by giving them a partial ordering and assigning to each one of them a specific number of processors. This has to be done while keeping in mind that the data dependencies have to be respected and that the time taken to execute the whole expression should be minimized.

The scheduling of the operations is accomplished with the use of *futures* which enable parallelism to be expressed. It is important to note, however, that by using

these constructs we incur an overhead penalty. We should also strive to minimize this overhead penalty.

2.2 Possible Solutions

When trying to schedule matrix expressions we are presented with various problems. The first aspect of the problem that should be solved is the parallelizing the operations themselves. Luckily, since these are matrix operations, they possess a great deal of parallelism. In matrix add, multiply and subtract all the individual adds and subtracts can be performed in parallel. This means that there are no data dependencies and all that we have to worry about is the assigning of equal number of simple operations to each processor. Other operations like matrix inverse are somewhat harder to parallelize.

After finding a way of performing the operations in parallel we are left with the problem of scheduling the different operations in a matrix expression. At first glance it would seem that a proper solution to the problem would be to allocate all processors to the first task, wait until this task is done and then allocate all processors to the second task and so on. This process would be repeated until all tasks or operations are done. The ordering of the tasks here would simply depend on the data dependencies. A program that implements this strategy would look for all the operations that it can execute (i.e. all the operations that have their data dependencies satisfied) and choose one of these to execute. The chosen task will then be given all processors. The

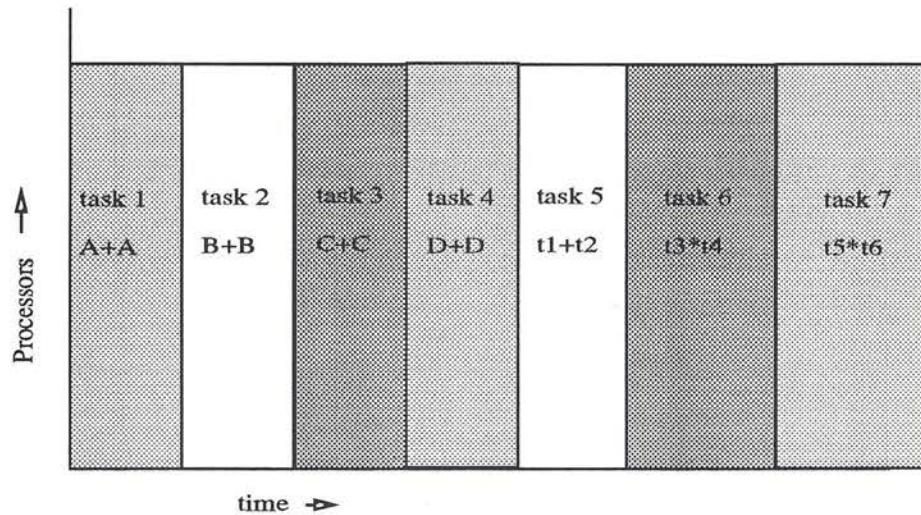


Figure 2.1: Scheduling all processors to each available task. Task numbers correspond to operations that have to be performed. T1 through T6 represent the results of tasks 1 through 6, respectively. Note that tasks 6 and 7 take longer to execute.

way this scheduler works can be seen in Figure 2.1, which illustrates how equation 2.1 would be parallelized using this scheme. Here the tasks are performed one after the other and each one of them uses all of the available processors. This solution was implemented, and it represents a basic solution to which the others were compared. The algorithm, which we called *Select1* basically serializes the computation of the expression. By doing this it fails to take advantage of the available parallelism. But, even more important, it creates a great number of *futures* and has to suffer due to all the overhead that these carry with them. In fact the number of *futures* that it creates is equal to the number of processors times the number of operations in the expression. Such a number could get really big. It is also undesirable to pay a cost that is directly proportional to the number of processors, since we want the elapsed

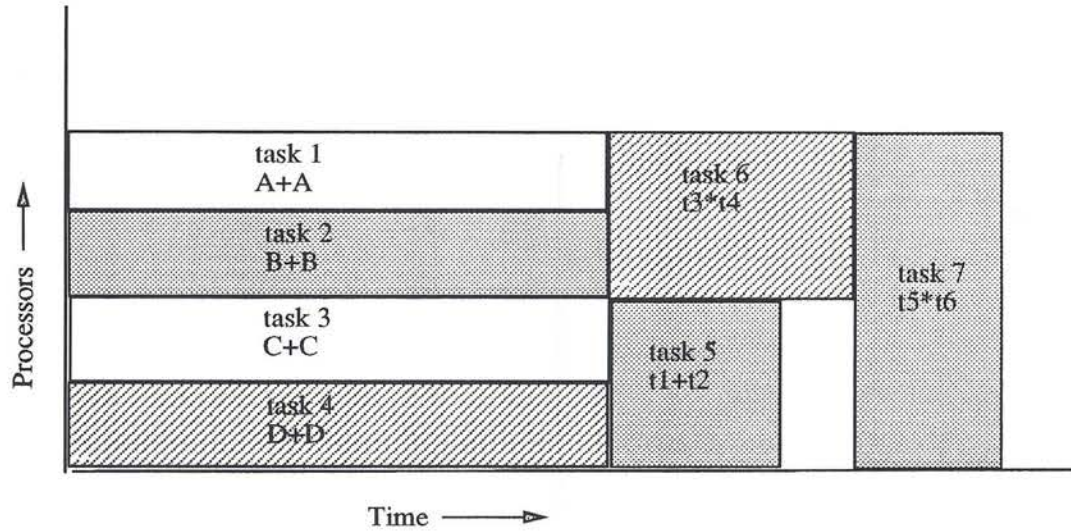


Figure 2.2: Scheduling all possible tasks by evenly dividing processors between them. Notice how task 5 finishes before task 6 since it is a simpler task.

time to be reduced by the addition of more processors.

What is desired is a scheduler that minimizes the amount of switching between processors. Each processor should handle the same task for as long as possible. This situation can be accomplished by distributing the processors. A scheduler that accomplishes this would try to execute all possible tasks at the same time.

This is what our second scheduler, *Selectall*, does. It finds all the tasks that can be executed and, instead of choosing only one of these to be executed, it executes all of them and divides the number of available processors evenly between them. In our example, we would start by performing all of the four innermost additions. Each one of these will be given $p/4$ processors where p is the total number of processors. Figure 2.2 shows how the processors are shared between the tasks. This might look

like a good scheduling technique but it can lead to a situation where there are idle processors. This will happen if some of the operations are more time consuming than others. In our example the left hand side of the multiplication will finish faster than the right hand side. This means that the processors allocated to compute the left hand side will be idle until the right hand side processors are finished. In Figure 2.2 we can, therefore, see that task 5 finishes before task 6 and the processors that were assigned to task 5 stay idle until task 7 can begin. A new scheduling technique similar to this one could be implemented, which takes this fact into account and gives more processors to the tasks that need them. This technique, however, was not implemented.

What was implemented, instead, was a slightly different technique which does, in fact, give more processors to those tasks that need them the most. We call this scheduling technique a *tree heuristic*, and it only works for expressions whose graphical representation in a tree. For equation 2.1 this heuristic allocates more processors to task 6 than to task 5. This way the harder task gets more processors and can execute faster. The scheduling technique first decides how hard the operations is and gives an appropriate number of processors to it. The number of processors assigned is determined by a heuristic that estimates how much time a certain operation will take. For a matrix expression we can define this heuristic to be a number proportional to the size of the matrices, times a constant of proportionality that depends on the operation being performed. The specifics of this heuristic will be discussed in the

next chapter.

The three scheduling heuristics implemented in this thesis are *Select1*, *Selectall* and *Tree heuristic*.

Chapter 3

Software structure

The ideas presented on the previous chapter were implemented on an Encore Multi-max machine using Mul-T. Mul-T is dialect of T. It has the same capabilities as T except that it also supports *futures*. This construct allows the user to express parallelism. By enclosing a task within a *future* we tell the machine to enqueue the task for later execution, possibly by a different processor, and continue execution without waiting for it to finish. The command *touch* forces a task to finish. *Futures* give the programmer the power to schedule and distribute tasks as he wishes. In this chapter we will explain the design and implementation techniques used for the different programs.

3.1 Overall Design

The program takes as input an expression written in lisp-like format and outputs a program written in Mul-T. There is, however, one intermediate representation between expressions and code, this representation takes the form of a canonical graph. These graphs, since they are derived directly from the lisp expressions, have the property of being directed acyclic graphs. These are used as canonical representations from which the compiler will derive various programs by applying the different specialized scheduling heuristics that were presented before. One of the reason for this representation is that it allows for the easy implementation of new scheduling algorithms. The graph also allows the compilers much more flexibility in accessing the different components and the structure of the expression. A block diagram of the system is shown in Figure 3.1.

The input format follows lisp's convention of preceding the operands by the operation, but only unary or binary operations are allowed. The different operations are represented by their respective symbols. Matrix addition is represented by $+$, subtraction by $-$, multiplication by $*$, LU decomposition by U and matrix inverse by $\%$. Figure 3.2 shows a sample input expression. Matrix inverse and LU decompositions do not perform row exchanges. This was not implemented due to time constraints.

The use of DAGs is not allowed at the moment. DAGs could be represented by using *let* expressions in the input expression. The use of *let* expressions is not supported right now but the program possesses a couple of functions designed for the interpretation

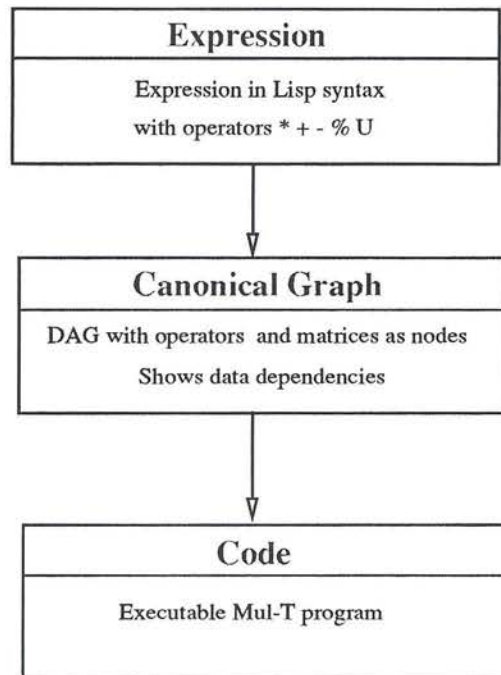


Figure 3.1: Block diagram of the system.

```
(* (+ (+ A A) (+ B B)) (* (+ C C) (+ D D)))
```

Figure 3.2: Sample input expression.

of *let* constructs. *Let* expressions poses the same syntax of a Lisp *let* expression. The implementation of DAGs will minimize redundant computation by allowing the user to specify reusable bindings.

The output format is in the form of executable Mul-T code. The compilers construct a program that includes even the matrices themselves, so that it can be run immediately. The output of the program is the result of calculating the given matrix expression. Figures 3.4 to 3.6 show some samples of the output programs. These will be explained in detail later.

3.2 Operations

Since all the operations are matrix operations they were parallelized by giving each processor a part of the matrix to calculate. This is especially true for matrix addition and subtraction. Matrix inversion and LU decomposition were somewhat more complicated to implement since they require some sequentiality.

3.2.1 Matrix addition and subtraction

These operations were implemented by dividing the number of processors between the number of rows in the operand matrices. If a matrix has m rows and is to be calculated with p processors then each processor is given $\lfloor p/m \rfloor$ rows to calculate. This is implemented by creating p futures, each of which will be handled by one processor. We assume that, on the average, there will be more rows than processors. It should be noted that the routine will never give more than one processor to each row. This will not affect the execution of the program as long as the previous assumption remains true. This assumption will prove to be right most of the time since the Encore machine which was being used, can use at most sixteen processors¹. The procedure specifications are as follows:

```
matadd = matrix-a matrix-b nproc
```

```
requires:matrix-a and matrix-b to be matrices.
```

¹If this constraint proves to be a problem then the `matadd` and `matsub` subroutines can be changed to distribute the actual elements of the matrix among the processors

effects:returns a matrix that is the sum of matrix-a and

matrix-b. It uses nproc processors.

matsub = matrix-a matrix-b nproc

requires:matrix-a and matrix-b to be matrices.

effects:returns a matrix that is the subtraction of matrix-a

from matrix-b. It uses nproc processors.

3.2.2 Matrix multiply

This procedure implements an optimal blocking technique which divides the matrix into a series of areas, each of which is assigned to one or more processors.² The procedure specifications are as follows:

matmul_block = mat-a mat-b mat-c st1 len1 st2 len2 st3 len3

p1 p2 p3

requires:mat-a, mat-b and mat-c are matrices that start

at st1, st2, st3 and have length of len1, len2, len3

respectively. p1, p2 and p3 represent the number of

processors available. They are set by compile-matmul.

modifies:mat-c

effects:changes mat-c to be the matrix multiplication of

mat-a and mat-b

²This subroutine was written by S. Prasanna.

3.2.3 LU decomposition

This procedure takes a matrix that has no zeros on its diagonal and that, otherwise requires no row exchanges, and returns the matrix in its upper triangular form. It implements row elimination by finding successive pivots. The procedure requires a certain amount of serialization. It first eliminates the numbers in the first column below the diagonal. It does this by performing simple row elimination with the first row. Since each row can be eliminated in parallel we can distribute all the rows between processors in the same way that was done in *matadd* and *matsub*. After assigning all rows to processors we make sure that they all have finished and then recursively repeat the problem for a submatrix that is equal to the first matrix without the first row and the first column. The specifications for this procedure are as follows:

```
dec = minrow maxrow mincol maxcol matrix p
```

requires: minrow and mincol are the minimum rows and column
numbers. maxrow and maxcol are the maximum.
matrix is the matrix and p the number of processors
effects: returns the upper triangular form of matrix 'matrix'

3.2.4 Matrix inverse

Matrix inverse is computed using gaussian elimination. Gaussian elimination was chosen above other methods because it provides a greater degree of parallelization and ease of implementation. Another method, like the determinant method, would

have proved too constraining in terms of the sequentiality that it imposes on the procedure. The matrix inverse routine first attaches an identity matrix to the input matrix, doubling the number of columns. This new matrix is then used as input to the LU decomposition subroutine. The output of this is then sent to another decomposition subroutine. This subroutine eliminates all the elements in the half above the diagonal of the original matrix. That is to say, it returns a matrix where the original input matrix (without the attached identity) has zeros everywhere except on the diagonal. This new matrix is sent to a final subroutine that divides each row by the number on it's row that is also part of the main diagonal. Finally, after this is done the input matrix has been turned into the identity matrix and the identity matrix has become the inverse of the input matrix. This second half of the matrix is returned. All these procedures are executed sequentially since the data dependencies require this to be so. It is very likely that a better scheme for computing matrix inverse can be devised. One such scheme might try to use back substitution instead of doing the second matrix decomposition. The specifications for this procedure are:

```
inverse = minrow maxrow mincol maxcol matrix p
```

```
requires:minrow and mincol are the minimum row and column
```

```
numbers. maxrow and maxcol are the maximum.
```

```
matrix is the matrix and p the number of processors
```

```
effects:returns the inverse matrix of matrix 'matrix'
```

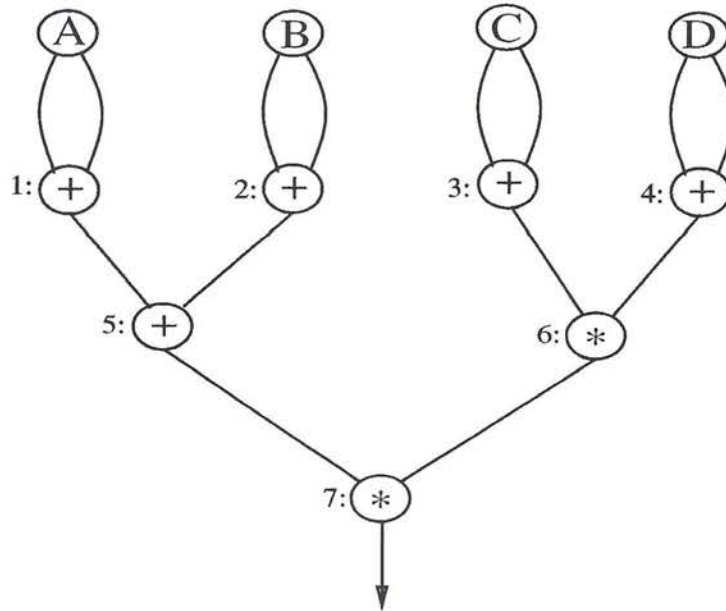



Figure 3.3: Expression 2.1 as seen in its graphical representation. Note that the numbers besides the nodes correspond to the task number.

3.3 Data Abstraction

The only data abstraction used in this program was a canonical graph, which was used to represent the input expression and the data dependencies in it. A graph is represented by the use of *nodes*. *Nodes* are the only data structure used. Each *node*, by itself, represents either a matrix, an operation or a whole expression. Equation 2.1 can be seen in its graphical representation in Figure 3.3. A *node* is a structure which holds a number of selector:value pairs, where the selectors and values are as follows:

type: Type of node, either prim, compound or io.

done: Flag to certify if a value for such a node has been calculated.

name: Holds the matrix or the name of a prim node.

optype: If the node is a prim node this contains the operation.

pred: Pointers to the predecessors of the node.

succ: Pointers to the successors of the node.

work: Number proportional to the amount of work performed by the node and its predecessors.

rows: Number of rows in the array that is the output of the node.

cols: Number of cols in the array that is the output of the node.

The *type* of a node is used to differentiate from the different instances of nodes. If the node contains a matrix, and therefore contains no predecessor pointers, its *type* is set to io node. A prim or primitive node does not contain a matrix but it does contain one or more pointers to its predecessors, along with the operation that it will perform on them. In place of the matrix it contains the temporary name that the code generator assigns to the value of the result of the operation. In Figure 3.3 the nodes with the letters on them, from A to D, are io nodes. The rest of the nodes are prim nodes. A compound node contains a graph and pointers to all the nodes within the graph plus specialized pointers to the last node in the graph, the output node, and the io nodes. Compound nodes allow for the storage of the graph within a single node. Some of the advantages of this representation include the fact that we can access all the nodes in the expression directly, without having to go through any indirection. The programs can also be expanded such that these compound nodes are used as io nodes for another expression.

The *work* selector contains the value of the work heuristic as applied to that node. This heuristic will be explained later. It is important to note that the nodes are

doubly linked. This gives a more powerful representation, one that is needed for the code generator and the schedulers to be able to generate the desired code.

3.4 Parser

The parser's job is to convert expressions into canonical graphs. Its operation is rather straight forward. It recursively follows, in a top-down manner, the lisp input expression while simultaneously constructing the desired nodes and the links between them. The main subroutine is called *make-nodes* and takes in as arguments a parent node and an expression. If the first element in this expression is an operation then the node is given the *type* of prim and its *pred* list points to it's predecessors. These are constructed recursively by one or more calls to *make-nodes*. If the expression is an atom, then the node created is an io node. In such a case the value of the expression (i.e. the matrix) is stored on the *name* selector. Once a node's predecessors have been computed then the node is given the appropriate number of rows and columns. These can be easily determined by looking at the operation and at the number of rows and columns in the preceding matrices.

The node is also given a work number that is calculated from its size along with the type of operation that it performs and the work done by its predecessors. The *work* number roughly represents the amount of time taken for the execution of the node and its subtree. All io nodes receive a *work* number that is equal to the zero. A primitive node gets a *work* number that is equal to the sum of the work that is

predecessors do plus a number that is proportional to the amount of work done by the node. Suppose that the input matrices to the node are M_1 and M_2 , with N_1 , N_2 and N_3 , N_4 rows and columns respectively. In case that the node's operation is addition then the work number equals $N_1 \times N_2$. If the operation is multiplication then the work number will be $N_1 \times N_2 \times N_4$. For matrix inverse we have to use a rough approximation. We set the work number to be equal to $2N^3/3$, where N is the number of rows on the matrix (assuming a square matrix). Finally for matrix decomposition we set it to be $N^3/3$. When *make-nodes* ends then the graph is completed.

Make-nodes was designed so as to make the implementation of general DAGs easier. Some extra subroutines were even constructed to exemplify how the parsing of these was to be accomplished. The addition of more operations is achieved by adding the desired operator to the global primitives constant and changing *make-nodes* to recognize this new primitive. This requires the programmer to come up with some heuristic for calculating the time that it would take to execute the desired operation given that we know the size of the matrices its operating on. The specifications for the parser procedure are as follows:

```
make-graph = exp
```

```
requires:exp be a lisp expression that contains only unary or
        binary operators. All operands are matrices. The legal
        operators are * + - % U.
```

```
effects:returns a graph that represents the input expression.
```

3.5 Code Generator

All the different scheduling algorithms explained in the previous chapter were implemented. The first two schedulers, the one that selects one node at a time and the one that selects all possible nodes, are based on the same algorithm. This algorithm takes the form of a simple loop:

- Find all nodes whose data dependencies are satisfied.
(We start with all the io nodes)
- If there are none then end the process.
- Call a select subroutine which will choose some of these nodes.
- Schedule the chosen nodes and mark them as done.
- Go to the first step.

The algorithm produces the code at the same time it is executing. It keeps recursively concatenating a series of let expressions which are functionally equivalent to the input expression. The first scheduler, which we call *select1*, uses the loop algorithm. Its select subroutine will only choose one of the possible nodes. It will then check to see if the node has any successors. If it does not then the scheduler is done and it then creates a program that executes the given operation. The operation's operands are the *name* fields of its predecessors. If the node does have successors then it will return a binding list which will bind a temporary name to the value that is the result of the execution of the operation. This name will also be assigned to the *name* field of the node. A sample program produced by this scheduler is shown in Figure 3.4.

The scheduler was run with eight processors on the expression seen in equation 2.1. The program sequentially creates all the seven *futures*, and gives each one of them eight processors.

The second algorithm, *selectall*, also uses the loop algorithm. It selects all possible nodes each time. This means that it creates one *future* for each node. But since we do not want to allocate more *futures* than processors then each *future* will get a number of processors that is equal to the total number of processors divided by the number of selected nodes. This strategy evenly distributes processors, other different techniques might be employed for the distribution of processors. This scheduler produced the program shown in Figure 3.5. This program, it can be seen, creates four initial *futures* to perform the for innermost additions (refer to equation 2.1), giving each one of these two processors. It then allocates two *futures* for the addition and the multiplication, giving each one of them four processors. Finally, it allocate a final *future* for the last multiplication, giving this one all eight processors. The specifications for these procedures are as follows:

```
compile-graph = graph nproc select
```

```
requires: graph is a graph created by make-graph, nproc
```

```
is the number of available processors. Select is either  
1 or 2.
```

```
effects: returns Mul-T code that will execute the desired graph.
```

```
If select is 1 it will use select1. If its 2 it will
```

```

(DEFINE (RUN1) (BLOCK
  (LET ((T.192
    (FUTURE
      (LET ((_TEMP_1 'MATRIX-A) (_TEMP_2 'MATRIX-A))
        (MATADD _TEMP_1 _TEMP_2 1 8))))))
  (LET ((T.193 (FUTURE
    (LET ((_TEMP_1 'MATRIX-B) (_TEMP_2 'MATRIX-B))
      (MATADD _TEMP_1 _TEMP_2 1 8))))))
    (TOUCH T.192) (TOUCH T.193)
    (LET ((T.194 (FUTURE
      (LET ((_TEMP_1 T.193) (_TEMP_2 T.192)
        (_TEMP_3 (MAKE-MATRIX 4 4 0)))
        (MATMUL_BLOCK _TEMP_1 _TEMP_2 _TEMP_3 0
          4 0 4 0 4 2 2 2))))))
      (LET ((T.195 (FUTURE
        (LET ((_TEMP_1 'MATRIX-C)
          (_TEMP_2 'MATRIX-C))
          (MATADD _TEMP_1 _TEMP_2 1 8))))))
        (LET ((T.196 (FUTURE
          (LET ((_TEMP_1 'MATRIX-D)
            (_TEMP_2 'MATRIX-D))
            (MATADD _TEMP_1 _TEMP_2 1 8))))))
          (TOUCH T.195) (TOUCH T.196)
          (LET ((T.197 (FUTURE
            (LET ((_TEMP_1 T.196)
              (_TEMP_2 T.195))
              (MATADD _TEMP_1 _TEMP_2 1 8))))))
            (TOUCH T.194) (TOUCH T.197)
            (LET ((T.198 (FUTURE (LET (
              (_TEMP_1 T.197) (_TEMP_2 T.194)
              (_TEMP_3 (MAKE-MATRIX 4 4 0)))
              (MATMUL_BLOCK _TEMP_1
                _TEMP_2 _TEMP_3
                0 4 0 4 0 4 2 2 2))))))
              (TOUCH T.198))))))))))

```

Figure 3.4: Sample output program using *Select1* with eight processors.


```
use selectall.
```

The third technique that was explained will is called *tree-heuristic*. It works only on expressions which are trees. This name comes from the fact that this scheduler looks at the whole tree and applies a heuristic to determine how many processors each node is going to get. For it to work we need each node to have its *work* selector set to an appropriate value.

This scheduler is designed by a recursive technique. It implements a top-down approach as opposed to the two previous techniques which implemented a bottom-up code generator. It starts by producing code that will execute the last operation, that is, the node with no successors. This operation is assigned all possible processors. The scheduler then calls itself recursively on all of the nodes predecessors³. We then split all the available processors among the subtrees giving them a number of processors that is proportional to the subtree load. We use a continuous approximation.⁴ Figure 3.6 shows a sample program produced by this scheduler. It shows how some operations get more processors than others. Specifically, we can see how the first two additions get only one processor each since this subtree can be calculated much faster than the other subtree with the multiplication. At the next level the multiplication gets six processors while the addition gets only two. This clearly reflects the impact that the size of the operations has on the scheduler.

```
compile-graph2 = graph nproc
```

³We are limiting this number to 2.

⁴This is explained by S. Prasanna. Ph.D. thesis.

```

(DEFINE (RUN2) (BLOCK
  (LET ((T.179 (FUTURE
    (LET ((_TEMP_1 'MATRIX-A) (_TEMP_2 'MATRIX-A))
      (MATADD _TEMP_1 _TEMP_2 1 2))))
    (T.180 (FUTURE
      (LET ((_TEMP_1 'MATRIX-B) (_TEMP_2 'MATRIX-B))
        (MATADD _TEMP_1 _TEMP_2 1 2))))
    (T.181 (FUTURE
      (LET ((_TEMP_1 'MATRIX-C) (_TEMP_2 'MATRIX-C))
        (MATADD _TEMP_1 _TEMP_2 1 2))))
    (T.182 (FUTURE
      (LET ((_TEMP_1 'MATRIX-D) (_TEMP_2 'MATRIX-D))
        (MATADD _TEMP_1 _TEMP_2 1 2))))
    (TOUCH T.180) (TOUCH T.179) (TOUCH T.182) (TOUCH T.181)
    (LET ((T.183
      (FUTURE
        (LET ((_TEMP_1 T.179) (_TEMP_2 T.180))
          (MATADD _TEMP_1 _TEMP_2 1 4))))
      (T.184 (FUTURE
        (LET ((_TEMP_1 T.181) (_TEMP_2 T.182)
          (_TEMP_3 (MAKE-MATRIX 4 4 0)))
          (MATMUL_BLOCK _TEMP_1 _TEMP_2 _TEMP_3 0 4
            0 4 0 4 2 1 2))))
      (TOUCH T.184) (TOUCH T.183)
      (LET ((T.185 (FUTURE
        (LET ((_TEMP_1 T.183) (_TEMP_2 T.184)
          (_TEMP_3 (MAKE-MATRIX 4 4 0)))
          (MATMUL_BLOCK _TEMP_1 _TEMP_2 _TEMP_3 0
            4 0 4 0 4 2 2 2))))
        (TOUCH T.185))))))

```

Figure 3.5: Sample output program using *Selectall* with eight processors

requires: graph is a graph created by make-graph, nproc

is the number of available processors.

effects: returns Mul-T code that will execute the desired graph.

```

(DEFINE (RUN3) (BLOCK
  (LET ((TEMP.186 (FUTURE
    (LET ((TEMP.188 (FUTURE
      (LET ((_TEMP_1 'MATRIX-A) (_TEMP_2 'MATRIX-A))
        (MATADD _TEMP_1 _TEMP_2 1 1))))
      (TEMP.189 (FUTURE
        (LET ((_TEMP_1 'MATRIX-B) (_TEMP_2 'MATRIX-B))
          (MATADD _TEMP_1 _TEMP_2 1 1))))))
        (TOUCH TEMP.188) (TOUCH TEMP.189)
        (LET ((_TEMP_1 TEMP.188) (_TEMP_2 TEMP.189))
          (MATADD _TEMP_1 _TEMP_2 1 2))))))
    (TEMP.187
      (FUTURE (LET ((TEMP.190 (FUTURE
        (LET ((_TEMP_1 'MATRIX-C) (_TEMP_2 'MATRIX-C))
          (MATADD _TEMP_1 _TEMP_2 1 2))))
        (TEMP.191 (FUTURE
          (LET ((_TEMP_1 'MATRIX-D) (_TEMP_2 'MATRIX-D))
            (MATADD _TEMP_1 _TEMP_2 1 2))))))
          (TOUCH TEMP.190) (TOUCH TEMP.191)
          (LET ((_TEMP_1 TEMP.190) (_TEMP_2 TEMP.191)
            (_TEMP_3 (MAKE-MATRIX 4 4 0))
            (PROC_WORK1 NIL) (PROC_WORK2 NIL)
            (PROC_WORK3 NIL))
            (SET PROC_WORK1 (FUTURE (MATMUL_BLOCK _TEMP_1
              _TEMP_2 _TEMP_3 0 3 0 4 0 4 2 1 2)))
            (SET PROC_WORK2 (FUTURE (MATMUL_BLOCK _TEMP_1
              _TEMP_2 _TEMP_3 3 1 0 4 0 4 1 1 1)))
            (SET PROC_WORK3 (FUTURE (MATMUL_BLOCK _TEMP_1
              _TEMP_2 _TEMP_3 4 0 0 4 0 4 1 1 1)))
            (TOUCH PROC_WORK1) (TOUCH PROC_WORK2)
            (TOUCH PROC_WORK3) _TEMP_3))))))
      (TOUCH TEMP.186) (TOUCH TEMP.187)
      (LET ((_TEMP_1 TEMP.186) (_TEMP_2 TEMP.187)
        (_TEMP_3 (MAKE-MATRIX 4 4 0)))
        (MATMUL_BLOCK _TEMP_1 _TEMP_2 _TEMP_3 0 4 0 4 0 4 2 2 2))))))

```

Figure 3.6: Sample output program using *Tree Heuristic* with eight processors

Chapter 4

Testing

This section covers the different tests performed on the system and the results that were obtained from these tests. There are several reasons for performing the tests. We would like, first of all, to know if the programs actually work and if they come up with suitable answers. This was proven, the programs did compile and produce executable, correct code. The second reason for performing the tests was to investigate which one of the different scheduling techniques worked better and with which set of tests cases. In other words, the tests determined which type of scheduling produced faster code given a certain type of expression. The expressions were characterized by how much inherent parallelism it contained (i.e. an very parallel expression is one with a very wide graph and many leaves), and how big the actual expression was.

To find out which scheduling program produced the fastest code the programs had to be timed. The timings were obtained from both `tmult` and `mult`. `Tmult`

is a simulated version of mult which runs on one processor. Mult is the parallel language used, which can actually run the program in parallel giving the desired tasks to the available processors. The problem with using mult and the reason we used tmult is that the time provided by mult is actual physical time. It doesn't take into account unix scheduling so that the times it returns are never completely correct. Another problem with running mult is that the Encore machine never seemed to have more than 8 processors running so the times acquired for the simulations had to be restricted to this range. It will be noticed that the results obtained from mult seem to contain more noise. These mult results should, therefore, be taken in context these disclaimers. On the other hand it must be also be noted that tmult does not take into account the communication time between processors. This was not a very big factor, as can be seen by comparing the timings of tmult and mult graphs, but it did change the results a little, and, it can be argued, in some important fashion.

4.1 Results

The expressions used can be seen in table 4.1. They represent both common and extreme cases. Each example tries to illustrate an interesting point about the different scheduling techniques. All matrices can be assumed different since no algebraic simplifications are performed.

The results of running expression A through the compiler and executing it can be seen in Figures 4.1 and 4.2. This expression is a basic expression. It is also

Test Expressions	
A:	$((x + y) \cdot (a + b))$
B:	$(a \cdot (a \cdot (c \cdot ((x \cdot y) \cdot (y \cdot z))))))$
C:	$(((((a \cdot b) \cdot (c \cdot d)) \cdot ((a \cdot b) \cdot (c \cdot d))) \cdot (((a \cdot b) \cdot (c \cdot d)) \cdot ((a \cdot b) \cdot (c \cdot d))))))$
D:	$(((((a + b) + (c + d)) + ((a + b) + (c + d))) + (((a \cdot b) \cdot (c + d)) \cdot ((a + b) + (c + d))))))$
E:	$((a^{-1} \cdot a^{-1}) + (b^{-1} \cdot b^{-1}))$
F:	$(((((a \cdot a) + (a \cdot a))^{-1}) \cdot ((a \cdot a) + (a \cdot a))))$

Table 4.1: Table of test functions. All variables are 20 by 20 matrices.

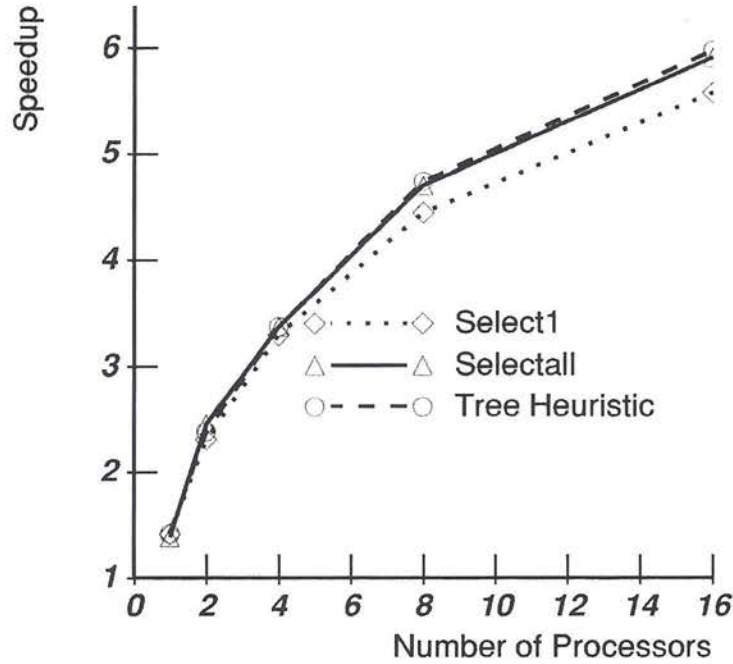


Figure 4.1: Timings for expression A as seen in the simulator.

short and simple and contains only additions and multiplications. By looking at the figures it is hard to find any discrepancies between the different compiling techniques. This similarity is probably due to the fact that it is such a small expression and the advantages of one method over the other can not be easily seen at this scale. This means that to make the results clearer and the advantages significant more complex expressions have to be used.

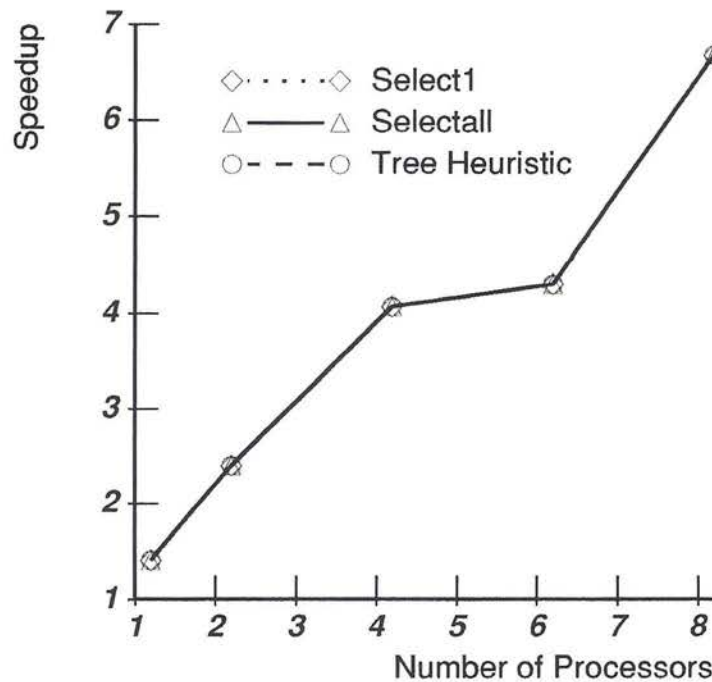


Figure 4.2: Timings for expression A as seen in mult.

Expression B is a very sequential expression which makes it into an unbalanced graph. The results of the tests with this expression can be seen in Figures 4.3 and 4.4. The operations in this expression have to be performed, for the most part, one after the other since the data dependencies require this to be so. The tests again are very similar for all the scheduling techniques. The only noticeable aspect that can be gathered is that *select1* lags a little behind the others in the simulator results. This can be attributed to that part of the expression that can be computed in parallel. The other two techniques probably ran these operations in parallel and therefore gained the small speedup that is seen. Notice how this disadvantage of *select1* is not seen in the mult simulations. What probably happened is that the communication overhead time made that slight advantage disappear. The fact that the mult times are identical

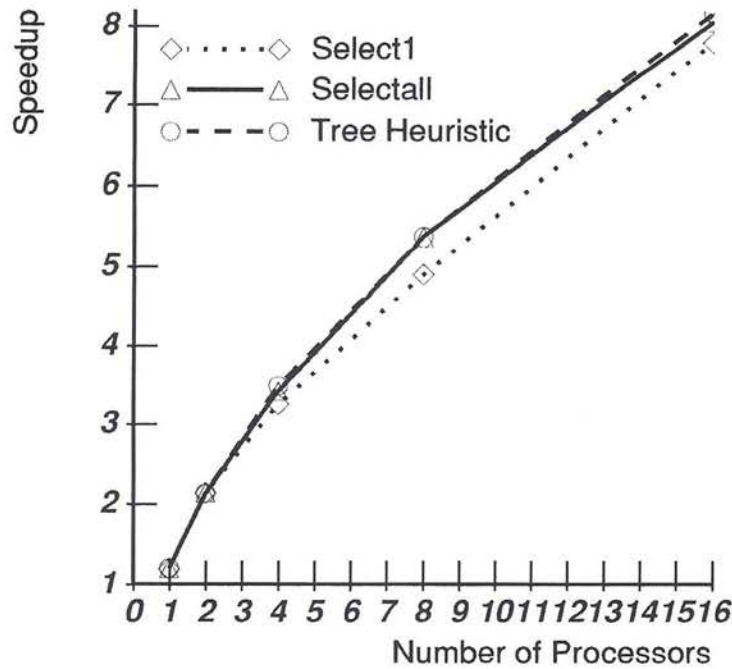


Figure 4.3: Timings for expression B as seen in the simulator.

for all scheduling techniques seems to imply that we do not lose anything by applying parallel techniques to a serial problem. The solution as computed in parallel takes just as much time as if it was computed serially by applying all processors to each task in sequence.

The first expression to show some actual discrepancy between the timing for the different schedulers is expression C. These results are in Figures 4.5 and 4.6. This expression produces a graph that possesses a considerable amount of parallelism. There are up to eight multiplications in this graph that can be computed in parallel. It is also, however, a very well distributed graph in that all the operations are multiplications. We see from the results of the simulator that *selectall* and *tree heuristic* are somewhat faster than *select1*. The difference even increases as the number of

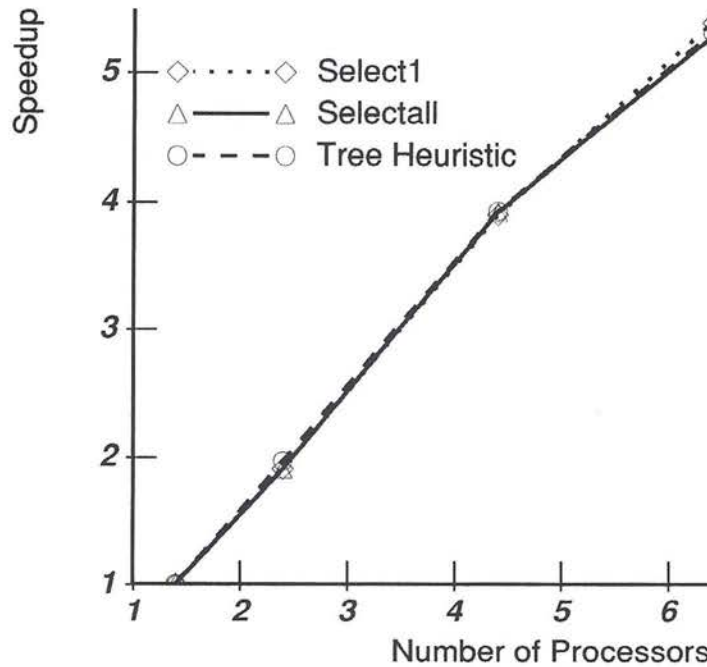


Figure 4.4: Timings for expression B as seen in mult.

processors is increased. This proves that scheduling the operations serially, when we have some parallelism available, is not a wise idea. However, the overhead costs that result from the creation of *futures* lower the speed up that is achieved by giving more processors to the task. Moreover, this overhead will only increase with the addition of more processors, which is the reason why we do not get a linear speedup. These results demonstrate to us that a parallel approach is indeed better than a purely serial one (around 40% better). It must be noted, however, that these same results do not appear when the programs are run on mult. In mult all the scheduling heuristics perform more or less the same. There is no clear advantage to compiling the expression in parallel. This is probably due to the communication costs. Later on, we will see other types of expression that do execute faster in parallel, even with the

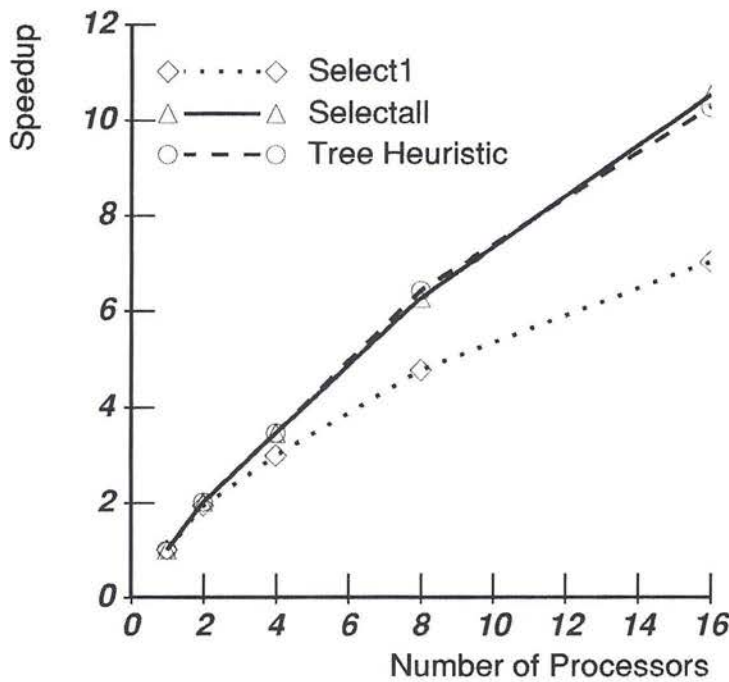


Figure 4.5: Timings for expression C as seen in the simulator.

communication costs.

But which one of the parallelizing methods is better. To answer this question we look at Figures 4.7 and 4.8. These are the timings for expression D. This expression has the same form as expression C except that the operators are different. If we imagine it's graph form we would see that all the operation nodes are addition except those on one branch that goes straight from the root to one of the leaves. *Selectall* is the worse possible scheduling in this case. At each level it gives all the operation nodes the same number of processors and waits for all of them to finish. This means that, at every level, the multiplication node will get the same number of processors that all the other addition nodes get. The multiplication will, of course, take a lot longer to execute while, in the meantime, all the processors allocated to the additions will be left

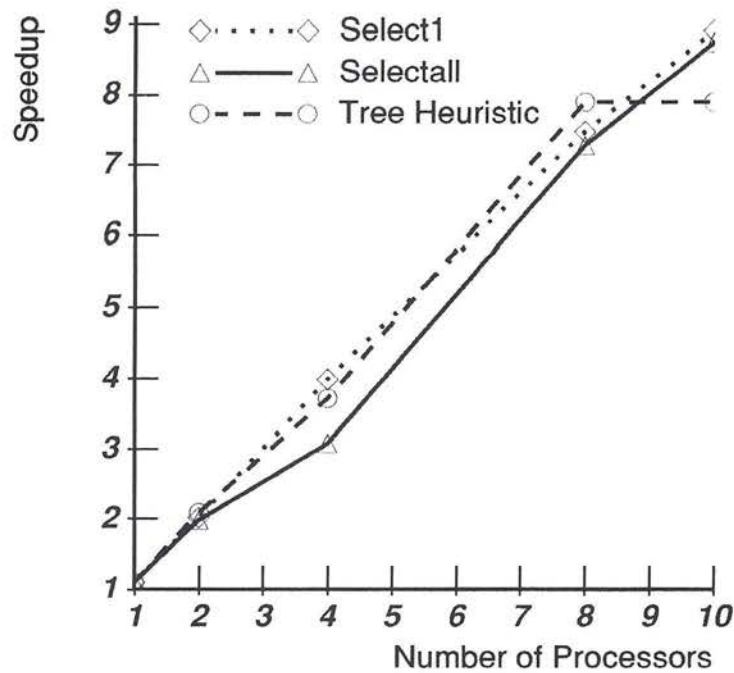


Figure 4.6: Timings for expression C as seen in mult.

idle. This is the reason why *selectall* is much slower than the two other techniques. Between *select1* and *tree heuristic* there is little difference. This difference makes *select1* faster than *tree heuristic*. The tests essentially say that it is faster to implement the expression sequentially than it is to use a parallel method. My explanation for this phenomena is that there are a great number of operations to perform and not enough processors. This results in *tree heuristic* giving fractional processors to the task. Fractional processors are, currently, resolved by sharing processors, which is what we were trying to prevent, or by creating more tasks.¹ This problem will only be solved if we can give more processors to the problem. Notice how the speedups of *select1* and *tree heuristic* seem to match as the number of processors reaches it's

¹Refer to S. Prasanna. Ph.D. thesis.

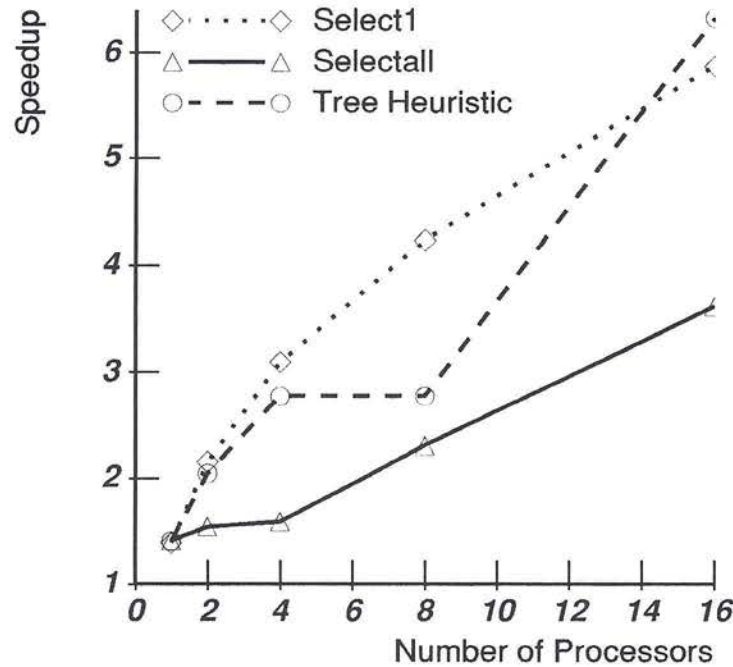


Figure 4.7: Timings for expression D as seen in the simulator.

maximum. This can be taken as an indication that the proposed solution is correct.

The timings for graph E can be seen in Figures 4.9 and 4.10. These look very similar to graph A. This is due to the fact that the expression being calculated is indeed very similar. This test shows that the incorporation of the inverse function does not, in any dramatic way, change the behavior of the programs. Here, like in the first test, we can appreciate how parallelism has a clear advantage over a purely sequential approach. This advantage is even more clear on the mult results. The inverse operation, therefore, seems to be a good candidate for parallelization within an expression.

Finally Figures 4.11 and 4.12 show the timings of expression F, which also uses the inverse operation. The result of this operation is the identity matrix. This way the

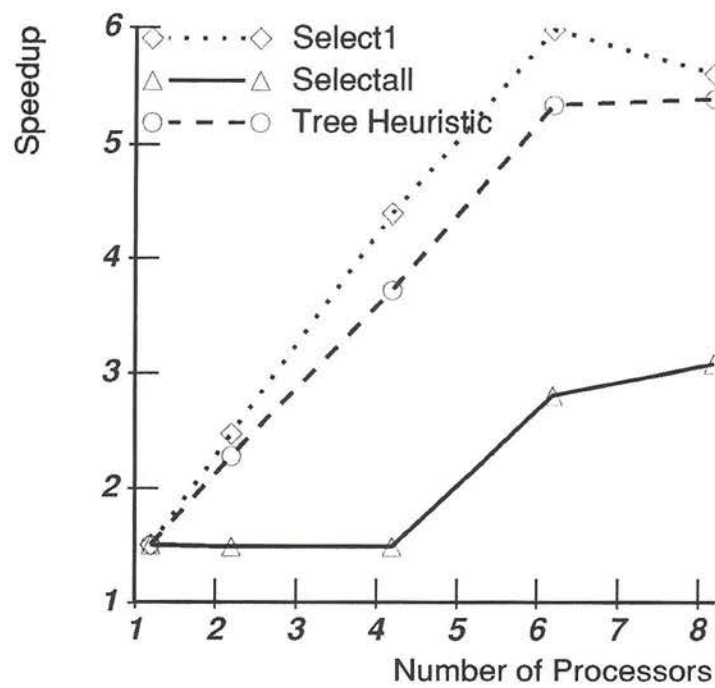


Figure 4.8: Timings for expression D as seen in mult.

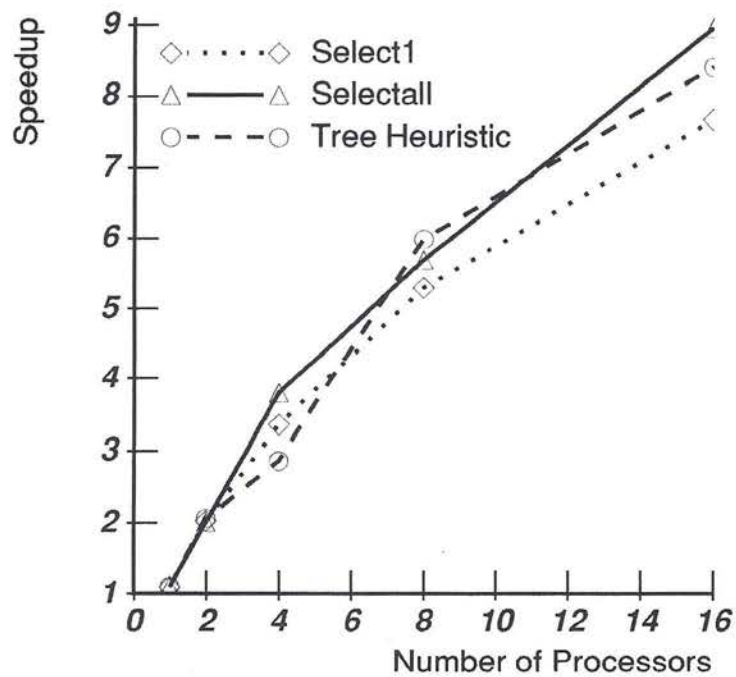


Figure 4.9: Timings for expression E as seen in the simulator.

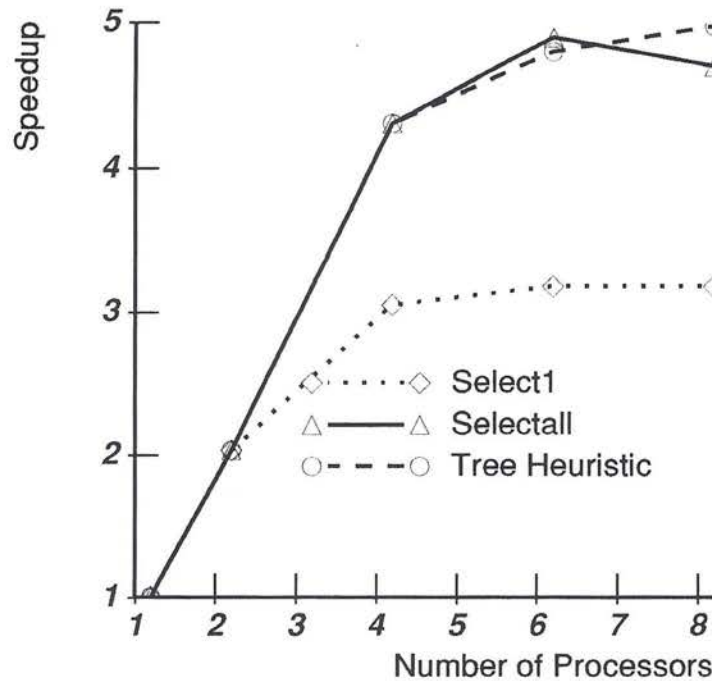


Figure 4.10: Timings for expression E as seen in mult.

validity of the result can be easily checked. The graphs show how *selectall* executes faster in the simulator. The reason for this is that *selectall* allocates all the processors available to the computation of the inverse which is the most time consuming operation in the expression. This fact can be checked if we follow the algorithm as it is applied to the graph. Unfortunately this also means that we will have all these processors communicating with each other. This situation is made even worse by the fact that the inverse operation requires a lot of communication between processors. The more processors we have the greater the volume of the communication gets. This could explain why we get a drop in the speedup when doing the tests on mult and not in the simulator, but this is far from certain. Since the simulator ignores communication it does not reduce its speed up by much. Mult, however, might be slowed

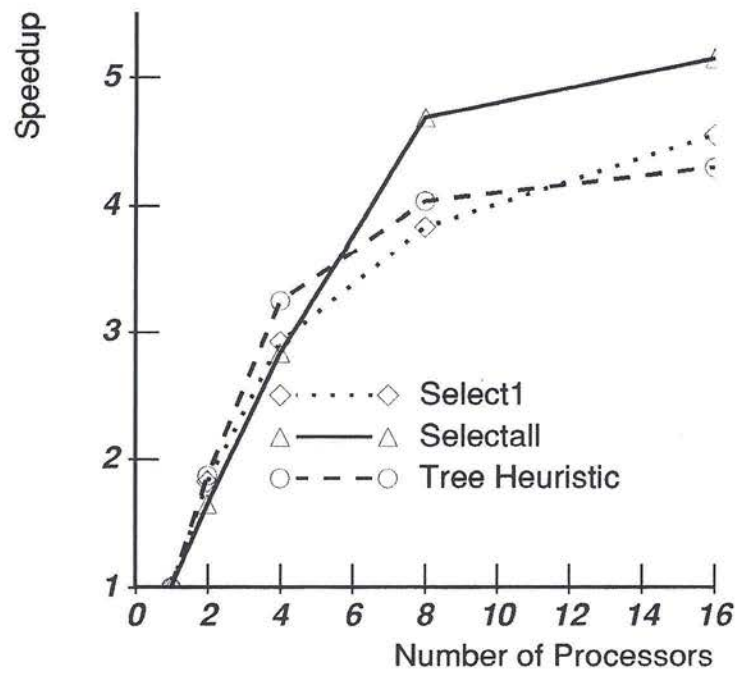


Figure 4.11: Timings for expression F as seen in the simulator.

down by communication delays.

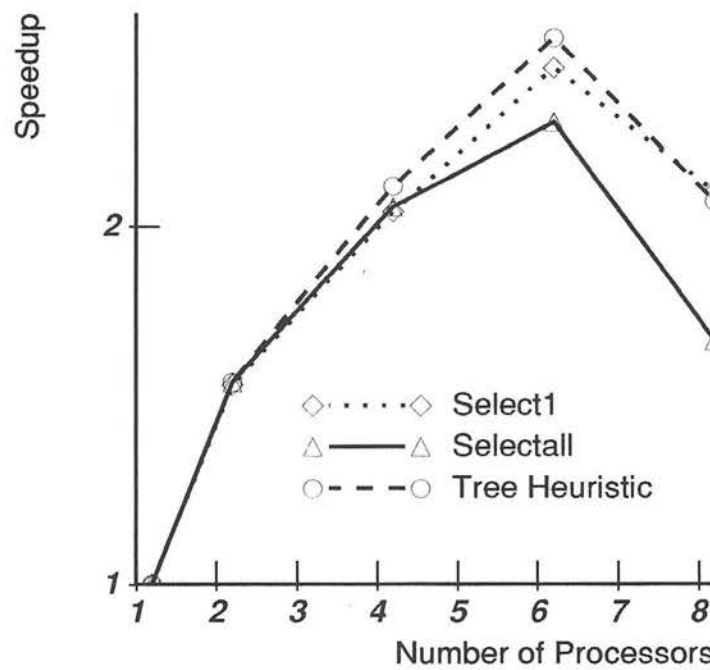


Figure 4.12: Timings for expression F as seen in mult.

Chapter 5

Conclusion

After seeing all the tests and comparing the results we must conclude that the parallelizing scheduling heuristics increase the speed of the evaluation of an expression. There are however some doubts as to which scheduling heuristic is better. I believe that the *tree heuristic* provides a better speedup. The fact that this was not seen in all the tests is probably due to an error in the actual heuristic. After some fine-tuning of the work heuristic, I think that this scheduler will achieve a much better speed than what we can now see. It should also be modified to take into account the communication overhead. In designing the different heuristics this overhead was not taken into account. This assumption, however, did not have any major effect on any of the test cases except with expression F. Here these delays might have possibly caused some problems.

In general, the individual tests serve as indications of what scheduling technique

works better in a specific case. The software is reliable and can be used as a tool for computing the value of matrix expressions.

Further work would include the creation and addition of DAGs, which I have mentioned throughout the document. The addition of more matrix operations would also make it more usable. Some simple ones include multiplication by a constant and addition with a constant. Some new and different scheduling heuristics could be added and their effectiveness tested against the ones already implemented. Finally a new matrix inverse might be implemented that does not do as much redundant work as this one does.

Appendix A

Description of the files

demo This files loads all the other files and contains the timing mechanism used for testing.

comp.t Contains the parser and the *Select1* and *Selectall* compilers. The functions *make-graph* and *compile-graph* can be found here.

comp2.t Contains the *tree heuristic* code generator.

matmul.t Code for the execution of a matrix multiplication.

compmul.t Interface between the code for matrix multiply and the compiler.

matadd.t Code for matrix addition and subtraction.

matother.t Code for LU decomposition and matrix inverse.

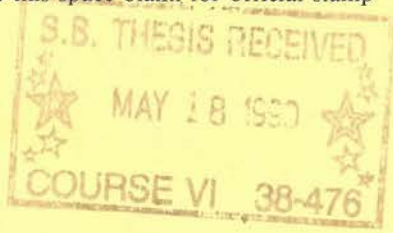
util.t Various utilities used by the other programs.

ex.t Example graphs.

simplmul.t A simpler multiplication subroutine.

THESIS RECEIPT AND GRADE SHEET

Not Valid Without Official Stamp

VIDAL JOSE M Print Name (Last) (First) (Middle Initial)		This thesis, as required for the S.B. Degree, has been received by the Department of Electrical Engineering & Computer Science, M.I.T.	
Leave this space blank for official stamp 		ANANT AGARWAL Print Supervisor's Name	SHAFI GOLDWASSER Print Academic Advisor's Name
		Today's Date: _____ Course: 6-1 <u>6-3</u> (circle one)	
		Expected Graduate Date: <u>June 4 1990</u>	
		MIT ID Number <u>596-01-7375</u>	

Thesis registered for as: ☒ 6ThU ☐ 6.929
 (check one)

Thesis Title: Parallelizing Compiler for Matrix Expressions
 (absolutely no more than 80 characters/spaces)

12 Enter the total number of 6ThU units for which you have registered in ALL terms, but NOT those units which were cancelled and not reinstated. (Minimum 12, maximum 30)

AREA BELOW TO BE FILLED OUT BY SUPERVISOR COMPLETELY

A Enter the final Thesis Grade. Should this thesis be placed in the Library? YES NO
 (circle one)

☒ An oral presentation has been conducted. AA
 (Please initial.)

Should the thesis be considered for the Electrical Engineering Thesis Prizes? YES NO
 (circle one)

To be considered, both the thesis and a nominated note (use the space below, or the reverse of the White Grade Sheet) must be received in 38-476 by the May INSTITUTE Deadline. The supervisor's note should explain the work's significance and the student's accomplishment.

For information about the Computer Science Thesis Prizes, the supervisor (not the student) should contact Prof. R. H. Halstead's office (NE43-205, 3-3537).

Please provide the student with both oral and written evaluation of all aspects of the thesis project. Use the space below for consideration of both the research and the write-up.

Excellent piece of work. Jose understood some of the complex theory involved in optimal partitioning of parallel programs and implement a matrix expression compiler. His thesis also includes some solid measurements of the performance of his algorithm on a real parallel machine.

Supervisor's Name (please print): ANANT AGARWAL Signature: [Signature] Date: _____

The supervisor should detach the back (GOLD) copy, and give ALL other copies to the student. Copies 1-3 must accompany the signed thesis to 38-476, where they will be stamped, before being distributed by the student, as shown below.

#1, WHITE (Grade Sheet): 38-476
 #3, PINK: Academic Advisor

#2, YELLOW (Thesis Receipt): Student
 #4, GOLD: Thesis Supervisor