An Abstract of the Thesis of

Youfeng Wu for the degree of Doctor of Philosophy in Computer Science presented on November 10, 1988

Title: Parallel Simplex Algorithms and Loop Spreading

Abstract approved: Redacted for Privacy

Ted. G. Lewis

Parallel solutions for two classes of linear programs are presented. First we parallelized the two-phase revised simplex algorithm and showed that it is possible to get linear improvement in performance. The simplex algorithm is the best known algorithm for solving linear programs, and we claim our result is the best one which can be achieved.

Next we study the parallelization of the decomposed simplex algorithm. One of our new parallel algorithms has achieved 2*P time of performance improvement over the decomposed simplex algorithm using P processors. Meanwhile, we discovered a particular variation of the decomposed simplex algorithm which can run 2 times faster than the original one. The new parallel algorithm linearly speedups the fast sequential algorithm.

As in any parallel program, unbalanced processor load causes the performance of the parallel decomposed simplex algorithm to
drop significantly when the size of the input data is not a multiple of the number of available processors. To remove this limitation, we invented a load balance technique called Loop Spreading that evenly distributes parallel tasks on multiple processors without a drop in performance even when the size of the input data is not a multiple of the number of processors. Loop Spreading is a general technique that can be used automatically by a compiler to balance processor load in any language that supports parallel loop constructs.
Parallel Simplex Algorithms 
and Loop Spreading

by

Youfeng Wu

A Thesis
submitted to
Oregon State University

in partial fulfillment of the requirements for the
degree of

Doctor of Philosophy

Completed November 10, 1988
Commencement June 1989
Approved:

Redacted for Privacy

Ted G. Lewis, Professor of Computer Science in Charge of Major

Redacted for Privacy

Chairman of Department of Computer Science

Redacted for Privacy

Dean of Graduate School

Date thesis is presented November 10, 1988
Typed by Youfeng Wu for Youfeng Wu
Acknowledgements

I would like to thank my thesis advisor Dr. Ted Lewis for directing me through the thesis work. Without his patient guidance, strong support, and many inspiring suggestions and discussions, this work could not be possible.

Also, many thanks are to the other members of my thesis committee: Dr. Bella Bose, Dr. Dave Birkes, Dr. Curtis Cook, and Dr. Joe Minne, for their invaluable comments and suggestions, and to Anita Osterhaug at Sequent for her professional proof-reading of the thesis.
# Table of Contents

Chapter 1. Introduction  
1.1. Parallel Program Design.................................................2  
1.2. Processor Load Balance.................................................4  
1.3. Thesis........................................................................5  

Chapter 2. Performance of Parallel Simplex Algorithm  
2.1. Introduction..................................................................9  
2.2. Two-phase Revised Simplex Algorithm and Its  
Parallelization......................................................................11  
2.3. Performance Analysis...................................................20  
2.4. Performance Experiment...............................................25  
2.5. Performance Comparisons..............................................30  
2.6. Conclusions..................................................................33  

Chapter 3. Parallel Algorithms for Decomposed Linear Programs  
3.1. Introduction..................................................................37  
3.2. Background...................................................................39  
3.3. Computational Procedure for Decomposed Simplex  
Algorithm............................................................................44  
3.4. Parallelizing Decomposed Simplex Algorithm.................48  
3.5. Parallel Algorithms for Decomposed Linear Programs.......57  
3.5.1. First Finished First (FFF) Algorithm............................57  
3.5.2. Tightly Synchronous (TS) Algorithm............................61  
3.5.3. Lookahead First Finish First (FFFL) Algorithm.............65  
3.5.4. Wypior's Approach....................................................68
3.5.5. Lookahead Tightly Synchronous (TSL) Algorithm ... 70
3.5.6. Processor Assignments for Lookahead Algorithms ... 71
3.5.7. Performance Comparison ........................................ 73
3.5.8. Fast Sequential Algorithm .................................... 75
3.5.9. Performance vs. Number of Processors ..................... 77
3.5.10. Removing Last Round Effects ............................... 79
3.5.11. Why TS algorithm is Good .................................. 82
3.6. Conclusions .............................................................. 87

Chapter 4. Parallel Processor Balance Through Loop Spreading

4.1. Introduction ............................................................. 88
4.2. Data Dependence and Process Scheduling ..................... 93
4.3. Loop Spreading and Problems .................................. 100
4.4. Loop Spreading When Substatements are Independent .... 103
4.5. Loop Spreading in Nested Loops and Matrix Multiplication Example ....................................................... 106
4.6. Loop Spreading When Substatements are Dependent ...... 109
4.7. Reducing Synchronization Overhead of Loop Spreading ... 120
4.8. Experiments With Dependent Substatements on a Shared Memory Machine ...................................................... 158
4.9. Related Work ............................................................ 164
4.10. Conclusions .............................................................. 166

Chapter 5. Implementation of Synchronization Primitives for Loop Spreading

5.1. Introduction ............................................................. 169
5.2. Straightforward Implementation of SYNC and WAIT ...... 178
5.3. Implementation of SYNC/WAIT for Column-major Spreading Scheme ......................................................... 181
5.4. Implementation of SYNC/WAIT for Row-major Spreading Schemes

5.5. Summary

Chapter 6. Conclusions and Future Work
6.1. Conclusions
6.2. Limitations of the Study
6.3. Future Work

Bibliography

Appendices
Appendix A. Two-Phase Revised Simplex Algorithm
A.1. Simplex Algorithm
A.2. Revised Simplex Algorithm
A.3. Two-phase Revised Simplex Algorithm
A.4. Computational Procedure for Two-phase Revised Simplex Algorithm

Appendix B. Sequential Algorithm for Decomposed Linear Programs
B.1. Decomposed Linear Programs
B.2. Dantzig and Wolfe's Decomposition Principle
B.3. Decomposed Simplex Algorithm
B.4. Retaining Sub-solutions Cross Central Iterations

Appendix C. Pascal Program for Parallel Matrix Multiplication With Loop Spreading

Appendix D. Pascal Program for Parallel Modified Matrix Multiplication With 1/2/3/4-Sequence and Column-major Loop Spreading
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1.</td>
<td>Performance of Parallel Simplex Algorithm</td>
<td>6</td>
</tr>
<tr>
<td>1.2.</td>
<td>Performance of TS Parallel Decomposed Simplex Algorithm</td>
<td>6</td>
</tr>
<tr>
<td>1.3.</td>
<td>Effect of Processor Load Balance on Performance of Parallel Simplex Algorithm Using Eight Processors</td>
<td>7</td>
</tr>
<tr>
<td>2.1.</td>
<td>Data Dependency Graph of the Two-phase Revised Simplex Algorithm</td>
<td>15</td>
</tr>
<tr>
<td>2.2.</td>
<td>Modified Data Dependency Graph For the Parallel Algorithm One</td>
<td>16</td>
</tr>
<tr>
<td>2.3.</td>
<td>Modified Data Dependency Graph For the Parallel Algorithm Two</td>
<td>16</td>
</tr>
<tr>
<td>2.4.</td>
<td>Task Graph of the Parallel Algorithm One</td>
<td>19</td>
</tr>
<tr>
<td>2.5.</td>
<td>Task Graph of the Parallel Algorithm Two</td>
<td>19</td>
</tr>
<tr>
<td>2.6.</td>
<td>Analytical Speedup of Algorithms One and Two</td>
<td>22</td>
</tr>
<tr>
<td>2.7.</td>
<td>Task Allocation of Algorithms One and Two</td>
<td>23</td>
</tr>
<tr>
<td>2.8.</td>
<td>Average Speedup of Algorithm One vs Number of Processors</td>
<td>27</td>
</tr>
<tr>
<td>2.9.</td>
<td>Average Speedup vs Input Size Using Seven Processors</td>
<td>29</td>
</tr>
<tr>
<td>2.10.</td>
<td>Performance Comparison With [FINKEL-87]</td>
<td>31</td>
</tr>
<tr>
<td>2.11.</td>
<td>Performance Comparison With [CHOI-88]</td>
<td>32</td>
</tr>
<tr>
<td>3.1.</td>
<td>Convex Polyhedron in 3-space Showing Region of Feasible Solutions to Sample Problem</td>
<td>40</td>
</tr>
<tr>
<td>3.2.</td>
<td>Pattern of a Decomposed Linear Program</td>
<td>41</td>
</tr>
</tbody>
</table>
3.3. Data Dependency Graph of The Decomposed Simplex Algorithm

3.4. Straightforward Parallelization Algorithm

3.5. Timing of The Straightforward Parallel Algorithm

3.6. Speedup of Straightforward Algorithm

3.7. SLAM II Code for Determining Expected Efficiency of Straightforward Algorithm

3.8. Timing of Subsolver Parallelizing Algorithm

3.9. Speedup of Subsolver Parallelizing Algorithm

3.10. Parallel FFF Algorithm

3.11. Timing of Parallel FFF Algorithm

3.12. Performance of Two FFF Implementations

3.13. Parallel TS Algorithm

3.14. Timing of Parallel TS Algorithm

3.15. Lookahead FFF Algorithm

3.16. Timing of Lookahead FFF Algorithm

3.17. Wypior's Algorithm

3.18. Timing of Wypior's Algorithm

3.19. Lookahead TS Algorithm

3.20. Timing of Lookahead TS Algorithm

3.21. Speedup of The Parallel Algorithms Over The Sequential Algorithm

3.22. Execution Times of SF/TS Sequential Algorithms

3.23. Speedup of Parallel TS Over Sequential TS and SF

3.24. Speedup of Parallel TS Over Sequential TS and SF When Number of Processors Changes From Three to Eight

3.25. Balanced Performance of Parallel TS Algorithm
3.26. Speedup of Spread TS over sequential TS and SF When Number of Processors Changes From Three to Eight

3.27. Total Numbers of Subiterations

3.28. Total Numbers of Central Iterations

4.1. Effect of Processor Load Balance on Performance of Parallel Simplex Algorithm Using Eight Processors

4.2. Effect of Processor Load Balance on Performance of Parallel Matrix Multiplication Algorithm

4.3. Loop Spreading for Parallel Load Balance

4.4. Static Scheduling Method 1 (a) and 2 (b)

4.5. Loop Spreading for Independent Substatements

4.6. Performance of Parallel Matrix Multiplication Algorithm With Loop Spreading

4.7. Inconsistent Loop Spreading With Dependent Substatements

4.8. Example of Evenly Distributed Dependent Substatements

4.9. Spreading Dependent Substatements

4.10. Synchronization Pattern for Spreading Dependent Substatements

4.11. Row-major Spreading

4.12. Column-major Spreading

4.13. Example Row-major Spreading Using Minimum Number of SYNCs

4.14. Example of 2-sequence Spreading

4.15. A 2-sequence Spreading That Uses Less Than (N MOD P) * (K-1) SYNCs/WAITs

4.16. A Non-optimal 2-sequence Spreading
5.6. Synchronization Pattern of 3-sequence Spreading 176
5.7. Synchronization Pattern of 4-sequence Spreading 177
5.8. Example of a Row-major Spreading in Which Multiple SYNCs Must Be Buffered 181
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1. Data Dependency in Two-phase Revised Simplex Algorithm</td>
<td>14</td>
</tr>
<tr>
<td>2.2. Complexity of Parallel Algorithms One and Two</td>
<td>20</td>
</tr>
<tr>
<td>2.3. Analytical Speedup of Algorithms One and Two</td>
<td>22</td>
</tr>
<tr>
<td>2.4. Execution Times of Sequential Algorithm and The Parallel Algorithm One</td>
<td>26</td>
</tr>
<tr>
<td>2.5. Execution Times of Sequential Algorithm and The Parallel Algorithm Two</td>
<td>26</td>
</tr>
<tr>
<td>2.6. Speedup of the Parallel Algorithm One</td>
<td>27</td>
</tr>
<tr>
<td>2.7. Speedup of the Parallel Algorithm Two</td>
<td>27</td>
</tr>
<tr>
<td>2.8. Execution Times of Sequential (a) and Parallel (b) Algorithms</td>
<td>28</td>
</tr>
<tr>
<td>2.9. Speedup of the Parallel Algorithms</td>
<td>29</td>
</tr>
<tr>
<td>2.10. Performance of [FINKEL-87]'s Parallel Simplex Algorithm</td>
<td>30</td>
</tr>
<tr>
<td>2.11. Performance Comparison With [FINKEL-87]</td>
<td>31</td>
</tr>
<tr>
<td>2.12. Performance of Choi's and Our Parallel Simplex Algorithms</td>
<td>32</td>
</tr>
<tr>
<td>3.1. Speedup of Straightforward Algorithm</td>
<td>51</td>
</tr>
<tr>
<td>3.2. Speedup of Subsolver Parallelizing Algorithm</td>
<td>55</td>
</tr>
<tr>
<td>3.3. Synchronization Intervals of TS algorithm</td>
<td>64</td>
</tr>
<tr>
<td>3.4. Execution Times of Parallel and Sequential Algorithms</td>
<td>73</td>
</tr>
<tr>
<td>3.5. Execution Time of Sequential SF, TS and Parallel TS Algorithms</td>
<td>76</td>
</tr>
<tr>
<td>3.6. Execution Time of Sequential TS, SF, and Parallel TS and The Speedup of Parallel TS Over Sequential TS and Sequential SF</td>
<td>78</td>
</tr>
</tbody>
</table>
3.7. Execution Time of Sequential TS, SF, and Spread TS and the Speedup of Spread TS Over Sequential TS and Sequential SF
3.8. Local Optimizability of SF Criterion
3.9. Local Optimizability of FFF Criterion
3.10. Local Optimizability of TS Criterion
3.11. Global Optimizability
Parallel Simplex Algorithms
and Loop Spreading

Chapter 1

Introduction

The demand for more powerful computers is rapidly growing in both numerical computation and symbolic processing (e.g. artificial intelligence) areas. The performance increases achieved through increases in components density are becoming more and more costly ([RIGEOUT-80], [SEITZ-84]), and designers are trying instead to achieve cost effective performance through logical structuring of these components, namely multiple processor architectures.

In recent years, several cost effective multiprocessor computers, such as Sequent/Balance and Transputers, have entered the applications world. Using these computers efficiently, however, requires solving many new issues, such as parallel program design and processor load balance.
1.1. Parallel Program Design

To solve a problem using a computer can be thought of as performing a partially ordered set (poset) of operations on a computer. For example, to calculate the sum of a, b, c, and d, the poset of operations can be \[ (((a+b)+(c+d)) (a+b)) \] or, represented graphically:

```
(a+b) + (c+d)
```

```
    /
   /|
  /  |
(a+b) (c+d)
```

Usually, the number of posets for a problem is tremendous. To design a computer program for the problem involves two tasks: 1) select one of the posets for the problem, and 2) represent the poset in a programming language.

In the early days of sequential program design, people underwent the difficulty of writing "good" programs. Since there are so many posets to choose from and so many ways to put the chosen poset to a sequence, writing a good program was in the domain of art that demands talent until the advent of the techniques such as structured programming. Using the structured method, a unique poset is selected systematically and this poset is purposively sorted.
out to a total sequence. Any partial relations are hidden from the programmer.

Parallel programming is in its early stage. Since independent operations should be performed on multiple processors to achieve high performance, partial relations among the operations must be clearly understood. Thus, the methods for sequential programming are no longer valid for designing parallel programs.

Many researchers tried to invent new methods for designing parallel programs ([ACKE-79], [ARVI-78], [CAMP-78], [DENN-74], [GAJSKI-82], and [IIZAWA-84]). However, very few of them are successful because most of their attentions focus on how to represent the partial relations of a program, and ignore the question of how to choose a poset for a given problem.

We studied parallel program design issues by examining two real problems: the parallel Simplex algorithm and parallel decomposed Simplex algorithm. It is our understanding that methods for designing sequential programs are still helpful in selecting a poset for a given problem. We may start designing a parallel program from a given sequential program. The sequential program should be analysed to reveal the partial relations existing inside the problem and then be rewritten in a parallel representation.
Since the choice and representation of the partial relations inside the problem are critical to a parallel solution, the analysing of the sequential program should focus on the data dependency existing in the program (which is part of the partial relations inherent to the problem). Then the performance of the parallel execution of the program is evaluated through analysis, simulation, or experimentation, by allowing tasks that are not data dependent to be performed by different processors. Based on the performance analysis, the choice of the poset and the representation of the poset is revised and the analysis process is repeated until an acceptable performance is obtained.

1.2. Processor Load Balance

When running a parallel program on a multiprocessor machine, either the programmer or the system has to distribute the parallel program on multiple processors. The traditional approach is that to represent a parallel program as a partially ordered graph (task graph) with each node and edge weighted by execution cost and to use some partitioning algorithm to allocate the tasks to processors.

There are several drawbacks to the traditional approach. First, for general graph, the optimal allocation is proven to be NP complete even for only three processors ([COFFMAN-76], [KASAHARA-84]). Numerous heuristics have been proposed to achieve reasonably good processor load balance ([ADAM-74], [HU-74], [KAUFMAN-74], [KOHLER-75], [GONZALEZ-77], [OUSTERHOUT-80, -82], [LO-87], [COFFMAN-76], [KASAHARA-84]).
Secondly, the approach tries to derive allocation by statical analysis of the source program. This is often inefficient, as the number of parallel tasks may not be known and the execution time of a task usually is not a constant.

A better approach, in our opinion, is to restrict the structure of a parallel program to a "simple" structure, for example, the one that results from any parallel program using only the PAR s1, ...sk ENDPAR and PAR i:=1 TO n DO s(i) ENDPAR statements. For this restricted class of parallel programs, we invented a new scheduling method (a mixture of static analysis and dynamic decision) that can allocate parallel tasks to multiple processors optimally with negligible runtime overhead.

1.3. Thesis

In Chapter 2, we parallelize the simplex algorithm through data dependency analysis and performance analysis. The performance of the parallel algorithm on the Sequent/Balance shared memory machine shows very close to linear speedup (Figure 1.1). This result is much better than those of [FINKEL-87] and [CHOI-88]. Furthermore, we discover several facts that suggest that shared-memory machines are better than distributed memory machines for solving linear programs (LPs). In particular, we suggest that any problem characterized by dynamic data partitioning can be parallelized on a shared memory machine more efficiently than on message-passing machines.
Figure 1.1. Performance of Parallel Simplex Algorithm.

Figure 1.2. Performance of TS Parallel Decomposed Simplex Algorithm.
In Chapter 3, we study six different ways to parallelize the sequential decomposed simplex algorithm (Appendix B). Although the decomposed simplex algorithm appears to have good parallelism, we demonstrate that straightforward parallelization results in very inefficient performance. The parallel solution proposed in literature ([WYPIOR-77]) is shown to have only half of the efficiency of one of our solutions, which speeds up the sequential algorithm by 2*P using P processors (Figure 1.2).

The load balance problem shown in the parallel decomposed simplex algorithm is that the performance of the algorithm drops significantly when the input data size is not a multiple of the number of processors available. Figure 1.3 shows the speedup of the best parallel algorithm over the sequential algorithm using 8 processors to solve decomposed LPs consisting of 3 to 20 subproblems.

Figure 1.3. Effect of Processor Load Balance on Performance of Parallel Simplex Algorithm Using Eight Processors.
In Figure 1.3, when the input data changes from 8 to 9 subproblems, the speedup drops from 6+ to around 4. The reason for such a drop is that when using 8 processors, the parallel execution of 9 tasks must be done in two rounds. The last round uses only one processor, leaving the remaining 7 processors idle.

In Chapters 4 and 5, we study **Loop Spreading**, a new technique to automatically restructure parallel loops so as to balance parallel tasks on multiple processors. Experimental results show that our new method gives excellent performance improvements. Using this method, we succeeded in keeping the performance of the parallel simplex algorithm always at its peak value.
Chapter 2

Performance of Parallel Simplex Algorithm

abstract

The performance of new parallel simplex algorithms for the Sequent/Balance shared memory machine is studied. The speedup is close to linear with the number of processors used in the parallel computation.

2.1. Introduction

In [FINKEL-87], parallel simplex algorithms for a loosely coupled message-passing machine are described. The observed performance "was seldom above 0.5", meaning that only 50% of the available processors were effectively used. By studying Finkel's algorithm we found that the simplex algorithm is by nature more suitable for implementation on a shared memory machine than on a loosely coupled machine because the kernel of the simplex algorithm is the pivot operation, which performs operations on both the rows and columns of the base matrix. For a loosely coupled machine, the parallel algorithm can either partition the matrix by row or by column, but not both, and distribute portions of the matrix to the local memories of individual processors. If the matrix is distributed by row (or column), expensive message passing is the only way for a processor to obtain the column (or row) data that is not stored in its
local memory. But, in a shared memory machine, the matrix is shared by all of the processors. A shared-memory processor can access all portions of the data equally well, thus resulting in better performance.

Arguably, contention for access to a shared memory can negate performance gains due to global access. In addition, lock/unlock operations incur overhead in the form of blocking. The question addressed by this research is whether the overhead of contention and locking is less than the overhead incurred by message passing. In what follows, we show that a shared memory solution to simplex algorithm is the "best" one in terms of scalability.

Performance Metrics

For a parallel algorithm (PA) obtained from a sequential algorithm (SA), the performance improvement of PA over SA can be measured by the speedup of PA over SA and the goodness of PA can be measured using the efficiency of PA. Formally, if SA takes $T_s$ time units to execute and PA takes $T_p(i)$ time units to execute using $i$ parallel processors then speedup of PA over SA using $i$ processors is defined as:

$$S_p(i) = \frac{T_s}{T_p(i)},$$

and the efficiency of the parallel program using $i$ processors is defined as:

$$E_p(i) = \frac{T_p(1)}{(i*T_p(i))}.$$
2.2. Two-phase Revised Simplex Algorithm and Its Parallelization

A linear Program (LP) is a system that finds vector $x$ that

minimizes $z = c^T x,$
subject to $Ax = b, x \geq 0$

where $A$ is a matrix of $m \times n$ $(n > m)$, $c$ is an $n$ element cost vector, $b$ is an $m$ vector, and $x$ is an $n$ unknown vector. The superscript $T$ denotes vector transposition. The equations $Ax = b$ stand for $m$ constraints on the unknowns. An example of an LP is:

Find $(x_1, x_2, x_3)$ that
minimizes $z = 2x_1 + x_2 + x_3$
subject to $2x_1 + x_2 - 3x_3 = 0$
$x_1 + x_2 + x_3 = 1$
$x_1, x_2, x_3 \geq 0.$

For this example, $c^T = (2, 1, 1), b^T = (0, 1),$ and $A = \begin{bmatrix} 2 & 1 & -3 \\ 1 & 1 & 1 \end{bmatrix}.$

The two-phase revised simplex algorithm ([DANTZIG-63], [ORCHARD-54], [SYSLO-83]) is the well known solution to this linear system and can be described as follows, where we assume $a_k$ is the $k$'th column of the matrix $A$, and $u_k$ is the $k$'th row of matrix $U$. 
Input. A, an m* n matrix; b, an m vector; c, an n vector.

Initialization. Extend A to an (m+2)* n matrix by adding c^T to be its m+1'th row and all zeros to be its m+2' th row;
the inverse of initial base \( U = B^{-1} = \begin{bmatrix} I_{m+1} & 0 \\ -e & 1 \end{bmatrix} \);
the initial base feasible solution \( x = (x_1, x_2, ..., x_{m+1}, x_{m+2}) = (b, 0, -\sum_{i=1}^{m} b_i) \);
the numbers of the base columns corresponding to the components of the optimal solution are remembered in vector w, with initial values of \( w_1 = n+1, w_2 = n+2, ..., w_{m+2} = n+m+2 \); phase = 1; q = m+2.

Iteration.

step 1. If \( x_q = 0 \) and phase = 1, set phase = 2, q = m+1.
For \( j = 1, ..., n \), calculate \( \delta_j = u_q * a_j \).

step 2. Calculate \( \delta_k = \min(\delta_j | j = 1, ..., n) \).
If \( \delta_k \geq 0 \) and phase = 1, then the original LP is infeasible and stop. If \( \delta_k \geq 0 \) and phase = 2, then \( x_q \) is the maximal and \( -x_q \) is the minimal value of the original LP, exit the repetition.
Otherwise, \( a_k \) is the new column to enter the base.

step 3. Compute \( y_i = u_i * a_k, i = 1, ..., q \), and \( \theta_i = x_i / y_i \) if \( y_i > 0 \),
i=1,...,m.
step 4. If all \( y_i \leq 0 \), \( i = 1, \ldots, m \), and phase = 1, then the original LP is infeasible, stop. If all \( y_i \leq 0 \), \( i = 1, \ldots, m \), and phase = 2, then the original LP is unbounded. Otherwise, calculate

\[
\theta = \theta_t = \min_{1 \leq i < m \& y_i > 0} (\theta_i)
\]

\( a_t \) is the column to be removed from the base.

step 5. Calculate the new values of the variables in the base solution and update \( U \):

\[
w_t = k, \quad x_t = \theta
\]

\[
x_i = x_i - \theta y_i \quad (i \neq t, \ i = 1, \ldots, q),
\]

\[
u_{ij} = u_{ij} - y_t u_{ij} / y_t \quad (i \neq t, \ i = 1, \ldots, q, \ j = 1, \ldots, m)
\]

\[
u_{tj} = u_{tj} / y_t
\]

Output. The optimal objective value is \(-x_q\), and the components of the optimal base feasible solution are \( x_1, \ldots, x_m \), where \( x_i \) corresponds to \( a_{w_i} \).

In the above algorithm, a normal iteration needs \( n(m+2) + q(m+2) + q(q+1) \) multiplications and additions, \( 2q \) divisions and \( 2n \) comparisons. If we assume a relation of 1 : 5 : 10 for the execution times of addition/comparison : multiplication : division, and \( q = m+1, n = 2m \), an iteration takes \( 24m^2 + 84m + 44 \) time units to execute.
In an iteration, we can see that $\delta_j$'s are modified in step 1 and used in step 2; $k$ is determined in step 2 and used in step 3; $y_i$'s and $\theta_i$'s are computed in step 3 and used in both step 4 and step 5; $t$ is found in step 4 and is used in step 5; $u_q$ is updated in step 5 and used in step 1 of the next iteration; and $u_i$'s are modified in step 5 and used in step 3 of the next iteration. These data dependency relations are summarized in Table 2.1, and the data dependency graph is shown in Figure 2.1.

<table>
<thead>
<tr>
<th>data items</th>
<th>modified</th>
<th>used</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\delta_{j,j=1...n}$</td>
<td>step 1</td>
<td>step 2</td>
</tr>
<tr>
<td>$k$</td>
<td>step 2</td>
<td>step 3</td>
</tr>
<tr>
<td>$y_i$</td>
<td>step 3</td>
<td>step 4,5</td>
</tr>
<tr>
<td>$\theta_{i,i=1...m+2}$</td>
<td>step 3</td>
<td>step 4,5</td>
</tr>
<tr>
<td>$t$</td>
<td>step 4</td>
<td>step 5</td>
</tr>
<tr>
<td>$u_q$</td>
<td>step 5</td>
<td>step 1</td>
</tr>
<tr>
<td>$u_{i,i=1...m+2}$</td>
<td>step 5</td>
<td>step 3</td>
</tr>
</tbody>
</table>

Table 2.1. Data Dependency in Two-phase Revised Simplex Algorithm
Figure 2.1. Data Dependency Graph of the Two-phase Revised Simplex Algorithm.

We have several ways to parallelize the two-phase revised simplex algorithm, all of which preserve the data dependency relations shown in Figure 2.1. In method one, we simply let the data dependency edges that go from step i to step i+k, k > 1, go to step i+1. The modified data dependency graph for this method is shown in Figure 2.2.

In method two, step 5 only updates $u_q$, and the updates of $u_i$'s, i<>q, are postponed to step 1 of the next iteration, as they are not used in the next iteration until step 3. The modified data dependency graph for method two is shown in Figure 2.3.
Figure 2.2. Modified Data Dependency Graph For the Parallel Algorithm One.

Figure 2.3. Modified Data Dependency Graph For the Parallel Algorithm Two.
The iterations of the parallel algorithm 1 can be described as follows (We use \( \text{PAR i:=1 TO n DO s(i) ENDPAR;} \) to denote the parallel execution of \( n \) statements \( s(i), \ i=1,2,...n \)). The task graph for the iterations is shown in Figure 2.4.

**Iteration of parallel algorithm 1.**

step 1. PAR \( j:=1 \) TO \( n \) DO \( s_1(j); \)

where \( s_1(j) \) is the following code:

\[
\delta_j = u_q \ast a_j;
\]

step 2. the same as before.

step 3. PAR \( i:=1 \) TO \( q \) DO \( s_3(i); \)

where \( s_3(i) \) is the following code:

\[
y_i = u_i \ast a_k;
\]

\( \text{IF } y_i >0 \text{ then } \theta_i = x_i/y_i; \)

step 4. the same as before.

step 5. PAR \( j:=0 \) TO \( q \) DO \( s_5(j); \)

where \( s_5(0) \) is the following code:

\[
w_t = k, \ x_t = \theta;
\]

FOR \( i:=1 \) TO \( m+2 \) DO

\( \text{IF } i <> t \text{ THEN } x_i = x_i - \theta y_i; \)

and \( s_5(j), j \geq 1, \) is the following code:

\[
u_{ij} = u_{ij}/y_t;
\]

FOR \( i:=1 \) TO \( m+2 \) DO

\( \text{IF } i <> t \text{ THEN } u_{ij} = u_{ij} - y_i u_{tj}; \)
Similarly, the iterations of the parallel algorithm 2 can be described as follows. Its task graph is shown in Figure 2.5.

**Iteration of parallel algorithm 2.**

step 1. PAR j:=1 TO n+q-1 DO $s_1(j)$;

where $s_1(j)$, $j<=n$, is the following code:

$$\delta_j = u_q \ast a_j;$$

$s_1(n+t)$ is the following code:

$$w_t = k, x_t = \theta;$$

FOR $i:=1$ TO $q$ DO

IF $i <> t$ THEN $x_i = x_i - \theta y_i$

$s_1(j)$, $n+q-1>j>n$, $j\neq t$, is the following code:

$$k := j-n;$$

FOR $i:=1$ TO $m+2$ DO

IF $i <> t$ THEN $u_{ik} = u_{ik} - y_i u_{tk}$

step 2. Omitted.

step 3. PAR $i:=1$ TO $q$ DO $s_3(i)$;

where $s_3(i)$ is the following code:

$$y_i = u_i \ast a_k;$$

IF $y_i >0$ then $\theta_i = x_i/y_i$;

step 4. Omitted.

step 5. PAR $j:=0$ TO $m$ DO $s_5(j)$;

where $s_5(j)$ is the following code:

$$u_{tj} = u_{tj}/y_t$$

$$u_{qj} = u_{qj} - y_q u_{tj}$$
Figure 2.4. Task Graph of the Parallel Algorithm One.

Figure 2.5. Task Graph of the Parallel Algorithm Two.
Note that, in both of the parallel algorithms, matrix $U$ is partitioned by rows among the processors in some steps and also are partitioned by columns in some other steps. For example, in the parallel algorithm 1, different processors in step 3 use different rows of matrix $U$ and in step 5 they modify different columns of the matrix. Similarly, in algorithm 2, different processors in step 1 modify different columns of $U$ and in step 3 they use different columns of the matrix.

2.3. Performance Analysis

Table 2.2 summarizes the complexity of the operations in parallel algorithms 1 and 2. For example, the first row of the table says that $s_1$ of algorithm 1 needs to be done $n$ times and each time needs $m+2$ multiplications and $m+2$ additions. $s_1$ of algorithm 2 needs to be done $n+q-1$ times, and each time needs $m+2$ multiplications and $m+2$ additions.

<table>
<thead>
<tr>
<th>task</th>
<th>algorithm 1</th>
<th>algorithm 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>*</td>
<td>+</td>
</tr>
<tr>
<td>s1</td>
<td>m+2</td>
<td>m+2</td>
</tr>
<tr>
<td>s2</td>
<td>n</td>
<td>1</td>
</tr>
<tr>
<td>s3</td>
<td>m+2</td>
<td>m+2</td>
</tr>
<tr>
<td>s4</td>
<td>m</td>
<td>1</td>
</tr>
<tr>
<td>s5</td>
<td>q+1</td>
<td>q+1</td>
</tr>
</tbody>
</table>

Table 2.2. Complexity of Parallel Algorithms One and Two.
We assume a relation of 1:5:10 for the execution times of addition/comparison:multiplication:division, and \( q=m+1, n=2m \). Further, we assume that we have \( p \) processors available. Then, one iteration of parallel algorithm 1 takes

\[
\left\lceil \frac{n}{p} \right\rceil s_1 + s_2 + \left\lceil \frac{q}{p} \right\rceil s_3 + s_4 + \left\lceil \frac{q}{p} \right\rceil s_5
= \left\lceil \frac{n}{p} \right\rceil (6(m+2)) + n + \left\lceil \frac{q}{p} \right\rceil (6(m+2)+10) + m + \left\lceil \frac{q}{p} \right\rceil (6(m+2)+10)
= \left\lceil \frac{2m}{p} \right\rceil (6m+12) + \left\lceil \frac{m+1}{p} \right\rceil (12m+44) + 3m
\]

One iteration of the parallel algorithm 2 takes

\[
\left\lceil \frac{(n+q-1)}{p} \right\rceil s_1 + s_2 + \left\lceil \frac{q}{p} \right\rceil s_3 + s_4 + \left\lceil \frac{q}{p} \right\rceil s_5
= \left\lceil \frac{(n+q-1)}{p} \right\rceil (6(m+2)) + n + \left\lceil \frac{q}{p} \right\rceil (6(m+2)+10) + m + \left\lceil \frac{q}{p} \right\rceil (6+10)
= \left\lceil \frac{3m}{p} \right\rceil (6m+12) + \left\lceil \frac{m+1}{p} \right\rceil (6m+38) + 3m.
\]

Remember that one iteration of the sequential algorithm takes \( 24m^2 + 84m + 44 \) time units to execute. Using these formulas, we have the following speedup estimates, shown in tabular form in Table 2.3 and graphically in Figure 2.6.
Although algorithm 2 performs better when $p$ is three times $m$, in the real world $m$ is usually very big compared to $p$. We will encounter mostly the case where $p \leq m/2$. In this case, both parallel algorithm 1 and 2 have the same performance, and nearly 100% processor utilization can be achieved.

Figure 2.7 illustrates the task allocation of the two algorithms. Figure 2.7.(1) shows that when $p$ is large, algorithm 1 performs
better than algorithm 2. Figure 2.7.(2) shows that when \( p \) is not large, algorithm 1 has the same performance as algorithm 2 (see Figure 2.7.3)) because the \( s_1 \)'s of algorithm 1 have to be fit into the limited processor resource.

![Diagram](image)

Figure 2.7. Task Allocation of Algorithms One and Two.

In the above analysis, we ignored the overhead associated with parallelization, such as process creation/termination, scheduling, locking, etc. For example, each parallel operation \( \text{PAR i:=1 TO n DO} \ s(i) \) will not only take time to execute \( s(i) \), but also time to create and terminate \( n \) processes. Since both parallel algorithms 1 and 2 have exactly three parallel statements, we can consider that the two
algorithms have the same overhead caused by parallelization.

In summary, when $p$, the number of processors is no more than $m$, the number of constraints of the input LP, the two parallel algorithms have the same performance and the speedup is nearly linear in the number of processors used.

**Loosely Coupled Processor Solution.**

If we are to implement the parallel algorithms on a loosely coupled message-passing machine, there will be significant communication overhead associated with the parallel algorithms 1 and 2. Take the parallel algorithm 1 as example. First, the $\delta_j$'s found in step 1 must be passed through all of the processors to find the minimum and determine the $k$. Then the $k$ has to be broadcasted to all of the processors to calculate $y_i$'s. In step 3, each process independently works on an individual row of $U$ and calculates an individual $y_i$ and $\theta_i$. The $\theta_i$'s found in step 3 must be passed through all of the processors to find the minimum and determine the $t$ in step 4. In step 5, it is no longer practical to partition matrix $U$ by columns since $U$ has been partitioned by rows in step 3. However, to update a row of $U$ in step 5 requires another row $u_t$ be available. Because the value of $t$ changes constantly, $u_t$ has to be obtained from other processors at runtime through message passing. These expensive communication costs make it hard to implement the parallel algorithms on a loosely coupled parallel machine.
The message passing overhead can be alleviated somewhat by overlapping computation with message passing, such as approached in [FINKEL-87] and [CHOI-88]. We will compare our result with the results of these two approaches implemented on message-passing systems.

2.4. Performance Experiment

The parallel algorithms have been implemented on the Sequent/Balance shared memory machine [THAKKAR-85]. The parallel statement

\[
\text{PAR } i:=1 \text{ TO } n \text{ DO } s(i) \text{ ENDPAR;}
\]

is directly implemented by using the parallel library routine m_pfork(n, s(i)) ([OSTERHAUG-86]), which creates n processes to execute the n statements s(i) in parallel. If the number of processors p is less than n, the parallel statement is implemented as follows:

\[
m\text{_pfork(p, s'(i));}
\]

where s'(i) is FOR j:=i STEP p TO n do s(j).

In the first experiment, five randomly generated linear programs (100 constraints and 200 variables) are solved using eight processors. Because p << m, both algorithms 1 and 2 should show the
same performance improvement over the sequential algorithm. The
execution times (in milliseconds) of the sequential and parallel
algorithms are summarized in Tables 2.4 and 2.5. The speedup of
the parallel algorithms over the sequential one is summarized in
Tables 2.6 and 2.7. From these tables, we can see that the two
parallel algorithms do perform roughly the same. The average
speedup for parallel algorithm 1 is plotted in Figure 2.8.

<table>
<thead>
<tr>
<th># procs</th>
<th>l1p</th>
<th>l2p</th>
<th>l3p</th>
<th>l4p</th>
<th>l5p</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq. alg.</td>
<td>1282310</td>
<td>1330890</td>
<td>1263030</td>
<td>1424590</td>
<td>1398720</td>
</tr>
<tr>
<td>1</td>
<td>1291800</td>
<td>1339000</td>
<td>1269510</td>
<td>1433340</td>
<td>1407060</td>
</tr>
<tr>
<td>2</td>
<td>656270</td>
<td>681100</td>
<td>644550</td>
<td>727290</td>
<td>715140</td>
</tr>
<tr>
<td>3</td>
<td>441840</td>
<td>458390</td>
<td>434460</td>
<td>490400</td>
<td>481540</td>
</tr>
<tr>
<td>4</td>
<td>334700</td>
<td>347490</td>
<td>329210</td>
<td>371810</td>
<td>364860</td>
</tr>
<tr>
<td>5</td>
<td>270770</td>
<td>280850</td>
<td>266430</td>
<td>300270</td>
<td>294620</td>
</tr>
<tr>
<td>6</td>
<td>225970</td>
<td>234470</td>
<td>222260</td>
<td>250860</td>
<td>245950</td>
</tr>
<tr>
<td>7</td>
<td>197030</td>
<td>204600</td>
<td>193910</td>
<td>218710</td>
<td>214670</td>
</tr>
<tr>
<td>8</td>
<td>173240</td>
<td>179800</td>
<td>168940</td>
<td>190280</td>
<td>188550</td>
</tr>
</tbody>
</table>

Table 2.4. Execution Times of Sequential Algorithm and
The Parallel Algorithm One.

<table>
<thead>
<tr>
<th># procs</th>
<th>l1p</th>
<th>l2p</th>
<th>l3p</th>
<th>l4p</th>
<th>l5p</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq. alg.</td>
<td>1282310</td>
<td>1330890</td>
<td>1263030</td>
<td>1424590</td>
<td>1398720</td>
</tr>
<tr>
<td>1</td>
<td>1257520</td>
<td>1303080</td>
<td>1235050</td>
<td>1397000</td>
<td>1367520</td>
</tr>
<tr>
<td>2</td>
<td>638060</td>
<td>662010</td>
<td>627610</td>
<td>707670</td>
<td>695020</td>
</tr>
<tr>
<td>3</td>
<td>428510</td>
<td>444470</td>
<td>421280</td>
<td>475440</td>
<td>466740</td>
</tr>
<tr>
<td>4</td>
<td>325090</td>
<td>337380</td>
<td>320020</td>
<td>360160</td>
<td>354250</td>
</tr>
<tr>
<td>5</td>
<td>262180</td>
<td>271910</td>
<td>257670</td>
<td>290660</td>
<td>285620</td>
</tr>
<tr>
<td>6</td>
<td>219000</td>
<td>227040</td>
<td>215280</td>
<td>243000</td>
<td>238280</td>
</tr>
<tr>
<td>7</td>
<td>189940</td>
<td>196990</td>
<td>186500</td>
<td>210200</td>
<td>206220</td>
</tr>
<tr>
<td>8</td>
<td>167460</td>
<td>173750</td>
<td>164770</td>
<td>187050</td>
<td>183270</td>
</tr>
</tbody>
</table>

Table 2.5. Execution Times of Sequential Algorithm and
The Parallel Algorithm Two.
Table 2.6. Speedup of the Parallel Algorithm One.

<table>
<thead>
<tr>
<th># procs</th>
<th>lp1</th>
<th>lp2</th>
<th>lp3</th>
<th>lp4</th>
<th>lp5</th>
<th>average</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.993</td>
<td>0.994</td>
<td>0.995</td>
<td>0.994</td>
<td>0.994</td>
<td>0.994</td>
</tr>
<tr>
<td>2</td>
<td>1.954</td>
<td>1.954</td>
<td>1.960</td>
<td>1.959</td>
<td>1.956</td>
<td>1.956</td>
</tr>
<tr>
<td>3</td>
<td>2.902</td>
<td>2.903</td>
<td>2.907</td>
<td>2.905</td>
<td>2.905</td>
<td>2.904</td>
</tr>
<tr>
<td>4</td>
<td>3.831</td>
<td>3.830</td>
<td>3.837</td>
<td>3.832</td>
<td>3.834</td>
<td>3.833</td>
</tr>
<tr>
<td>5</td>
<td>4.736</td>
<td>4.739</td>
<td>4.741</td>
<td>4.744</td>
<td>4.748</td>
<td>4.741</td>
</tr>
<tr>
<td>6</td>
<td>5.675</td>
<td>5.676</td>
<td>5.683</td>
<td>5.679</td>
<td>5.687</td>
<td>5.680</td>
</tr>
<tr>
<td>8</td>
<td>7.402</td>
<td>7.402</td>
<td>7.476</td>
<td>7.487</td>
<td>7.418</td>
<td>7.437</td>
</tr>
</tbody>
</table>

Table 2.7. Speedup of the Parallel Algorithm Two.

<table>
<thead>
<tr>
<th># procs</th>
<th>lp1</th>
<th>lp2</th>
<th>lp3</th>
<th>lp4</th>
<th>lp5</th>
<th>average</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.020</td>
<td>1.021</td>
<td>1.023</td>
<td>1.020</td>
<td>1.023</td>
<td>1.021</td>
</tr>
<tr>
<td>2</td>
<td>2.010</td>
<td>2.010</td>
<td>2.012</td>
<td>2.013</td>
<td>2.012</td>
<td>2.012</td>
</tr>
<tr>
<td>3</td>
<td>2.992</td>
<td>2.994</td>
<td>2.998</td>
<td>2.996</td>
<td>2.997</td>
<td>2.996</td>
</tr>
<tr>
<td>4</td>
<td>3.944</td>
<td>3.945</td>
<td>3.947</td>
<td>3.955</td>
<td>3.948</td>
<td>3.948</td>
</tr>
<tr>
<td>5</td>
<td>4.891</td>
<td>4.895</td>
<td>4.902</td>
<td>4.901</td>
<td>4.897</td>
<td>4.897</td>
</tr>
<tr>
<td>6</td>
<td>5.855</td>
<td>5.862</td>
<td>5.867</td>
<td>5.863</td>
<td>5.870</td>
<td>5.863</td>
</tr>
<tr>
<td>8</td>
<td>7.657</td>
<td>7.660</td>
<td>7.665</td>
<td>7.616</td>
<td>7.632</td>
<td>7.646</td>
</tr>
</tbody>
</table>

Figure 2.8. Average Speedup of Algorithm One vs. Number of Processors.
Note from Tables 2.4 and 2.5 that the absolute execution times of all of the sample problems are not very fast, for the following reasons: 1) the Pascal compiler used does not generate very efficient code; 2) Sequent/Balance processors are not very fast, and 3) the algorithm uses double precision real arithmetic. However, the average speedup shown in Figure 2.8 increases almost linearly with the increase in the number of processors used.

<table>
<thead>
<tr>
<th>input size</th>
<th>lp1</th>
<th>lp2</th>
<th>lp3</th>
<th>lp4</th>
<th>lp5</th>
</tr>
</thead>
<tbody>
<tr>
<td>60*120</td>
<td>a 264940</td>
<td>243460</td>
<td>236750</td>
<td>272640</td>
<td>266620</td>
</tr>
<tr>
<td></td>
<td>b 43190</td>
<td>39910</td>
<td>38740</td>
<td>44430</td>
<td>43500</td>
</tr>
<tr>
<td>80*160</td>
<td>a 624280</td>
<td>561990</td>
<td>622510</td>
<td>681440</td>
<td>649820</td>
</tr>
<tr>
<td></td>
<td>b 96960</td>
<td>87580</td>
<td>97120</td>
<td>105740</td>
<td>100990</td>
</tr>
<tr>
<td>100*200</td>
<td>a 1282310</td>
<td>1330890</td>
<td>1263030</td>
<td>1424590</td>
<td>1398720</td>
</tr>
<tr>
<td></td>
<td>b 197030</td>
<td>204600</td>
<td>193910</td>
<td>218710</td>
<td>214670</td>
</tr>
<tr>
<td>120*240</td>
<td>a 2314850</td>
<td>2367340</td>
<td>2380870</td>
<td>2333450</td>
<td>2542770</td>
</tr>
<tr>
<td></td>
<td>b 355010</td>
<td>362470</td>
<td>365260</td>
<td>357170</td>
<td>390970</td>
</tr>
<tr>
<td>140*280</td>
<td>a 4140750</td>
<td>3892170</td>
<td>3941780</td>
<td>3719990</td>
<td>3941920</td>
</tr>
<tr>
<td></td>
<td>b 625570</td>
<td>586760</td>
<td>593910</td>
<td>561360</td>
<td>594920</td>
</tr>
<tr>
<td>160*320</td>
<td>a 6837310</td>
<td>6866120</td>
<td>7743190</td>
<td>6967900</td>
<td>6917390</td>
</tr>
<tr>
<td></td>
<td>b 1020870</td>
<td>1024710</td>
<td>1155470</td>
<td>1039290</td>
<td>1032910</td>
</tr>
<tr>
<td>180*360</td>
<td>a 9408680</td>
<td>9610250</td>
<td>1014190</td>
<td>10298910</td>
<td>10080230</td>
</tr>
<tr>
<td></td>
<td>b 1395800</td>
<td>1427840</td>
<td>1506650</td>
<td>1528320</td>
<td>1496910</td>
</tr>
<tr>
<td>200*400</td>
<td>a 13861780</td>
<td>13368460</td>
<td>13736320</td>
<td>14327070</td>
<td>13599180</td>
</tr>
<tr>
<td></td>
<td>b 2051120</td>
<td>19816400</td>
<td>2029910</td>
<td>2112910</td>
<td>2013680</td>
</tr>
</tbody>
</table>

Table 2.8. Execution Times of Sequential (a) and Parallel (b) Algorithms.

A second experiment was carried out to study how parallel algorithm 1 improves performance over the sequential algorithm when the size of the input changes. Seven processors were used. The input LPs have 60, 80, 100, 120, 140, 160, 180, and 200 constraints. The number of variables is always taken as two times
the number of constraints. For each input size, five randomly generated linear programs were solved. The execution times (in milliseconds) of the sequential and parallel algorithms are summarized in Table 2.8. The speedup of the parallel algorithm over the sequential one is summarized in Table 2.9. The average speedup is plotted in Figure 2.9.

<table>
<thead>
<tr>
<th>Input size</th>
<th>lp1</th>
<th>lp2</th>
<th>lp3</th>
<th>lp4</th>
<th>lp5</th>
<th>average</th>
</tr>
</thead>
</table>

Table 2.9. Speedup of the Parallel Algorithms.

Figure 2.9. Average Speedup vs. Input Size Using Seven Processors.
Figure 2.9 tells us that the parallel algorithm's speedup over the sequential algorithm increases as the input size increases. Even for a relatively small input size, the parallel algorithm comes close to linear improvement in computation time.

2.5. Performance Comparisons

Comparison With Finkel's Results

In [FINKEL-87], the simplex algorithm is implemented using Lynx [SCOTT-84] on Crystal [DEWITT-84], a message-passing machine. Two processors are used, and the results are summarized in Table 2.10 (according to the data in Table 1 of [FINKEL-87]).

<table>
<thead>
<tr>
<th>m</th>
<th>speedup</th>
<th>efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0.03</td>
<td>0.01</td>
</tr>
<tr>
<td>5</td>
<td>0.08</td>
<td>0.04</td>
</tr>
<tr>
<td>10</td>
<td>0.34</td>
<td>0.17</td>
</tr>
<tr>
<td>15</td>
<td>0.65</td>
<td>0.32</td>
</tr>
<tr>
<td>20</td>
<td>0.69</td>
<td>0.34</td>
</tr>
<tr>
<td>25</td>
<td>0.85</td>
<td>0.42</td>
</tr>
<tr>
<td>30</td>
<td>1.00</td>
<td>0.50</td>
</tr>
<tr>
<td>35</td>
<td>0.84</td>
<td>0.42</td>
</tr>
<tr>
<td>40</td>
<td>1.11</td>
<td>0.55</td>
</tr>
<tr>
<td>45</td>
<td>1.01</td>
<td>0.50</td>
</tr>
<tr>
<td>48</td>
<td>0.92</td>
<td>0.46</td>
</tr>
</tbody>
</table>

Table 2.10. Performance of [FINKEL-87]'s Parallel Simplex Algorithm.

Running our parallel simplex algorithm on the same input data, we obtained the performance data shown in Table 2.11. In Figure 2.10, we plot the efficiency of the two results.
Table 2.11. Performance Comparison With [FINKEL-87].

<table>
<thead>
<tr>
<th>m</th>
<th>seq time</th>
<th>para time</th>
<th>speedup</th>
<th>efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>118</td>
<td>380</td>
<td>0.31</td>
<td>0.15</td>
</tr>
<tr>
<td>5</td>
<td>244</td>
<td>458</td>
<td>0.53</td>
<td>0.26</td>
</tr>
<tr>
<td>10</td>
<td>1026</td>
<td>888</td>
<td>1.16</td>
<td>0.58</td>
</tr>
<tr>
<td>15</td>
<td>4000</td>
<td>2472</td>
<td>1.62</td>
<td>0.81</td>
</tr>
<tr>
<td>20</td>
<td>7348</td>
<td>4224</td>
<td>1.74</td>
<td>0.87</td>
</tr>
<tr>
<td>25</td>
<td>16716</td>
<td>9132</td>
<td>1.83</td>
<td>0.91</td>
</tr>
<tr>
<td>30</td>
<td>25926</td>
<td>13896</td>
<td>1.87</td>
<td>0.93</td>
</tr>
<tr>
<td>35</td>
<td>46268</td>
<td>24418</td>
<td>1.89</td>
<td>0.94</td>
</tr>
<tr>
<td>40</td>
<td>66050</td>
<td>34580</td>
<td>1.91</td>
<td>0.95</td>
</tr>
<tr>
<td>45</td>
<td>92154</td>
<td>48094</td>
<td>1.92</td>
<td>0.96</td>
</tr>
<tr>
<td>48</td>
<td>125802</td>
<td>65376</td>
<td>1.92</td>
<td>0.96</td>
</tr>
</tbody>
</table>

From Figure 2.10, we can see that our parallel algorithm is about twice as efficient as Finkel's algorithm using two processors.
Comparison With Choi's Results.

In [CHOI-88], an experimental simplex algorithm is implemented in OCCAM on Transputers ([MAY-84]). Table 2.12. summarizes the performance of Choi's algorithm (From Figure 3-6 of [CHOI-88]) and our result on the same randomly generated LPs of 100 constraints and 200 variables. Figure 2.11. plots the respective speedup.

<table>
<thead>
<tr>
<th>#processors</th>
<th>speedup/Choi</th>
<th>efficiency/Choi</th>
<th>speedup/ours</th>
<th>efficiency/ours</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>4.61</td>
<td>0.92</td>
<td>4.90</td>
<td>0.98</td>
</tr>
<tr>
<td>6</td>
<td>5.58</td>
<td>0.93</td>
<td>5.86</td>
<td>0.98</td>
</tr>
<tr>
<td>7</td>
<td>6.25</td>
<td>0.89</td>
<td>6.77</td>
<td>0.97</td>
</tr>
<tr>
<td>8</td>
<td>6.85</td>
<td>0.86</td>
<td>7.75</td>
<td>0.97</td>
</tr>
</tbody>
</table>

Table 2.12. Performance of Choi's and Our Parallel Simplex Algorithms.

Figure 2.11. Performance Comparison With [CHOI-88].
Even though the Transputer is a specially designed parallel processor that supports message passing at the hardware level, our result is again better than that on a message-passing machine. More importantly, Choi's performance shows a clear tendency to drop when the number of processors increases, while our result linearly increases as the number of processors increases.

2.6. Conclusions

The simplex algorithm can be quite easily parallelized on the Sequent/Balance shared memory machine. Unlike the result in [FINKEL-87] and [CHOI-88], this parallelization can achieve nearly 100% efficiency. This is due to the fact that the simplex iteration can be parallelized by partitioning both the rows and the columns of the base matrix. In general, we argue that a shared memory machine is a better choice for implementing the simplex algorithm, based on the following observations.
1) A shared memory machine allows dynamic data partitioning. Consider the following problem: for an input matrix $A(n \times n)$, calculate $A(i, j) = A(i, j) + \sum_{k=1}^{i} A(i, k) \sum_{k=1}^{j} A(k, j)$. If we solve this problem on a message-passing system, we encounter the problem of how to distribute matrix $A$ to multiple processors. If we partition $A$ by row, then it is relatively easy to calculate $\sum_{k=1}^{i} A(i, k)$ as all of the $A(i, k)$'s are stored on the same processor, but it is very expensive to calculate $\sum_{k=1}^{j} A(k, j)$ as each $A(k, j)$ is on a different processor for $k = 1, ..., i$. If we solve the problem on a shared memory system, both row partitioning and column partitioning are equally cheap, as no explicit data distribution is necessary. The Simplex algorithm requires dynamic data partitioning.

2) Processor load balance is cheaper to achieve in a shared memory system. As shown in Chapter 4, loop spreading is a powerful technique used to balance processor load for parallel programs. However, loop spreading converts certain data that are local to a single processor to global data among multiple processors. As we know that data sharing in a message passing system is more costly than in a shared memory system, loop spreading on a message passing system is much more costly than on a shared memory system.

3) A shared memory system permits more parallelism than a message passing system. To see this, consider the example of multiple processors accessing common data. Assume value $x$ is to be
used by process p1, p2, ..., pn.

The code to do this in a shared memory system can be the following lock-access operations:

on processor pi (i=1.. n)

lock(x);

\[ x' := x; \]

unlock(x);

the code that uses \( x' \)

The code to do this in the message passing system can be the following broadcasting operation:

\[ p_0 \]
\[ p_i \ (i=1.. n) \]

seq i:=1 to n do send(x, c[i]);

receive(x, c[i]);

code that uses \( x \);

The sender in the message passing system has to make multiple copies of the same data by itself and send the copies to other processes. This tends to make the sender process the bottleneck of the program. But in the shared memory system, the copy operations can be distributively done by the processes that use the data. Also in this example, the receiving processes must receive data in the same order it is sent (assuming the message passing takes constant time), while in the shared memory system, the readers can compete to read the shared data and the actual sequence of the reads is
dependent on the processes' relative execution speeds. The latter case obviously has better performance because the faster processes also use the data sooner and don't have to wait their turn.

4) The message-passing model is more restricted than the shared memory model, so some well known techniques to obtain parallelism cannot be applied. Consider the reader/writer problem. In the shared memory system, a reader/writer lock can be used to allow multiple reader processes to access the common data simultaneously. However, in the message passing system, the only way to permit one process to modify a value while other processes read the value is to implement the modification/broadcast solution. Concurrent accesses by the reader processes are impossible.
Chapter 3

Parallel Algorithms for
Decomposed Linear Programs

Abstract

New parallel algorithms for the decomposed linear programs are developed. Our experiment on a shared memory machine shows that one of the new algorithms achieves more than 2*P times performance improvement over the sequential algorithm using P processors. Furthermore, we discovered a particular variation of the sequential algorithm which runs more than 2 times faster than the normal sequential algorithm on the shared memory machine.

3.1. Introduction

People have been looking for fast Linear Program solvers for a long time because linear programs model many real world applications and solving linear programs is computationally intensive ([DANTZIG-63], [BEN-68], [CHARNES-80], [GROTSCHL-81]). New sequential linear program solvers such as Karmarkar's algorithm ([KARMARKAR-84]) reduce the worst-case time complexity to a polynomial bound. But results of recent computational study ([GILL-85]) cast doubt on Karmarkar's claim that his algorithm will replace the simplex algorithm.
An alternate approach to solving computationally difficult linear programs is to devise parallel solutions that run on fast parallel machines ([WYPIOR-77], [FINKEL-87], [THOMPSON-87], [PANG-87], [CHOI-88]).

In Chapter 2, we parallelized the revised two-phase simplex algorithm with linear performance improvement in terms of number of processors used. In this paper, we study parallel algorithms for solving decomposed linear programs.

Direct parallelization of the sequential algorithm often results in very limited performance improvement using multiple processors because a sequential algorithm is designed without parallel consideration in mind. When we were parallelizing the decomposed simplex algorithm, we found that, without changing the algorithm itself, the sequential decomposed simplex algorithm can be improved by only half of the number of processors used. But by redesigning the algorithm, we achieved more than 2*P times performance improvement over the sequential algorithm, where P is the number of processors used in parallel computation. Furthermore, a particular variation of the sequential algorithm runs more than 2 times faster than the original sequential algorithm. The new parallel algorithm linearly speedups the new sequential algorithm.
3.2. Background

A Linear Program (LP) is a system that finds vector \( \mathbf{x} \) which

minimizes \[ z = \mathbf{c}^\top \mathbf{x}, \]
subject to \[ A\mathbf{x} = \mathbf{b}, \mathbf{x} \geq 0, \]

where \( A \) is an \( m \) by \( n \) matrix (\( n > m \)), \( \mathbf{c} \) is an \( n \) element cost vector, \( \mathbf{b} \) is a vector of length \( m \), and \( \mathbf{x} \) is an unknown vector of length \( n \). The superscript \( \top \) denotes vector transposition. The equation \( A\mathbf{x} = \mathbf{b} \) stands for \( m \) constraints on the unknowns. An example of an LP is:

Find \( (x_1, x_2, x_3) \) that

minimizes \[ z = 2x_1 + x_2 + x_3 \]
subject to \[ 2x_1 + x_2 - 3x_3 = 0 \]
\[ x_1 + x_2 + x_3 = 1 \]
\[ x_1, x_2, x_3 \geq 0 \]

For this example, \( \mathbf{c}^\top = (2, 1, 1), \mathbf{b}^\top = (0, 1), m = 2, n = 3, \) and

\[
A = \begin{bmatrix}
2 & 1 & -3 \\
1 & 1 & 1
\end{bmatrix}
\]

Geometrically, the constraints \( A\mathbf{x} = \mathbf{b} \) and \( \mathbf{x} \geq 0 \) define a convex polyhedron of dimension \( m \) in an \( n \) dimensional space. The polyhedron of the above LP is shown in Figure 3.1.
Figure 3.1. Convex Polyhedron in 3-space Showing Region of Feasible Solutions to Sample Problem.

The end points $p_1$ and $p_2$ in Figure 3.1 are the extreme points of the system. An extreme point is a solution that has no more than $m$ non-zero components. It can be shown that if an LP has a minimal solution, then one of its extreme points must be a minimal solution. In the above example, $p_1$ is the minimal solution.

The simplex algorithm [DANTZIG-63] solves an LP by starting from an extreme point and repeatedly going to the next adjacent extreme point that decreases the $z$ value, until it goes to an extreme point where the $z$ value can not be further decreased.

The decomposed linear programs are a special class of LPs in which the coefficient array $A$ contains all zeros except in the first few rows and along diagonal blocks according to the pattern shown
in Figure 3.2, where for j=1..n, $A_j$ is an m by $n_j$ matrix; $D_j$ is an $m_j$ by $n_j$ matrix; $b_j$ is an $m_j$ vector; and $b$ is an m vector.

![Figure 3.2. Pattern of a Decomposed Linear Program.](image)

Figure 3.2. Pattern of a Decomposed Linear Program.

An example of a decomposed LP is:

Find $(x_1, x_2, x_3, x_4)$ that

minimizes $z = 2x_1 + x_2 + x_3 + 5x_4$

subject to

\[
\begin{align*}
2x_1 + x_2 - 3x_3 + x_4 &= 0 \\
x_1 + x_2 &= 1 \\
3x_1 - x_2 &= 0 \\
x_3 + 5x_4 &= 1
\end{align*}
\]

In this example,

\[
A_1 = (2, 1), \quad A_2 = (-3, 1)
\]

\[
D_1 = \begin{bmatrix} 1 & 1 \\ 3 & -1 \end{bmatrix}, \quad D_2 = (1, 5)
\]

\[
b = (0), \quad b_1 = (1, 0)^T, \quad b_2 = (1).
\]
The decomposition principle of Dantzig and Wolfe [DANTZIG-60, 61] is an elegant method for solving decomposed LPs. According to this principle, an input decomposed linear program is treated as the central program, and the diagonal blocks are treated as the coefficient matrices of sublinear programs. Each central iteration first determines the solutions of the sublinear programs and then uses the solutions to determine its own pivot operation. Although there are many sequential implementations of the principle ([ADLER-73], [BEALE-65], [KUTCHER-73], [HO-81]), there are very few discussions on the parallelization of the algorithm ([WYPIER-77]).

Performance Evaluation Metrics

There are two ways that a program can be parallelized: 1) implicit parallelization ([KUCK-72, 76, 81, 84]), in which the parallel algorithm is the same as the sequential algorithm except that certain statements of the sequential algorithm are allowed to be executed in parallel; 2) explicit parallelization, in which the parallel algorithm employs a different approach to the problem than the sequential algorithm.

Implicit parallelization is limited because the original program was designed with sequential semantics in mind. Only very few programs can be implicitly parallelized with nearly linear speedup, and in most cases an efficiency of 10% is considered quite satisfactory ([LEE-85]), as stated in [WOLFE-87], "users rarely achieve
the peak speed of the machine unless they are willing to rewrite their programs."

Explicit parallelization requires that a programmer redesign the sequential algorithm to exploit parallelism in both the problem and the underlying parallel machine. However, explicit parallelization is a difficult job because the programmer has to think in terms of multiple threads of program execution and take care of many possible interactions among the parallel processes.

For a parallel algorithm (PA) obtained through implicit parallelization from a sequential algorithm (SA), the performance improvement of PA over SA can be measured by the speedup of PA over SA and the goodness of PA can be measured using the efficiency of PA. Formally, if SA takes $T_s$ time units to execute and PA takes $T_p(i)$ time units to execute using $i$ parallel processors, then the speedup of PA over SA is defined as:

$$S_p(i) = \frac{T_s}{T_p(i)}$$

and the efficiency of the parallel program using $i$ processors is defined as:

$$E_p(i) = \frac{T_p(1)}{(i \times T_p(i))}.$$
We note that $S_p(i) \leq i$ in the implicit parallelization case. This may not be true in the case of explicit parallelization. Assume $SA$ is parallelized explicitly through algorithm changes to $PA$. Then $PA$ can be executed sequentially by using only one processor. Clearly, we can rewrite $PA$ as another sequential program ($SA'$) using sequential constructs as if the program is executed by only one processor. Now suppose $PA$ is "obtained" from $SA'$ through implicit parallelization, and we compute the speedup of $PA$ over $SA'$ using $i$ processors. The speedup of $PA$ over $SA'$ will be $\leq i$. However, $SA'$ can run many times faster than $SA$. If $SA'$ runs $F (> 1)$ times faster than $SA$, then the speedup of $PA$ over $SA$ can be as high as $F*i$.

For both implicit and explicit parallelization, $E_p(i) \leq 1$. The efficiency of a parallel program is a variable of the number of processors used and is independent of which sequential program it corresponds to.

3.3. Computational Procedure for Decomposed Simplex Algorithm

Based on Dantzig-Wolfe's decomposition principle, we developed a decomposed simplex algorithm for solving decomposed linear programs, as follows (see appendix B for derivation):

**Input.** $A_i$, $m*ni$ matrices and $D_i$, $mi*ni$ matrices, $i = 1, \ldots n$; $b_0$, an $m$ vectors; $b_i$, $mi$ vectors, $i = 1, \ldots n$; $c_i$, $ni$ vectors, $i = 1, \ldots n$. 
Initialization. Assume \( e \) be a vector of all 1's.

The inverse of initial base

\[
U = (u_1, u_2, ..., u_{m+n+2}) = B^{-1} = \begin{bmatrix} I_{m+n+1} & 0 \\ -e & 1 \end{bmatrix};
\]

the initial base feasible solution

\[
s = (s_1, s_2, ..., s_{m+n+2}) = (b_0, 1, 1, 0, n+\sum_{i=1}^{m} b_{0i});
\]

the central left hand side vector \( b = (b_0, 1, 1, 0, 0) \);

the initial subsolutions \( e_x = (e_{x1}, e_{x2}, ..., e_{x_{m+n}}) = (0, 0, ..., 0) \)

and the corresponding indices of the subproblems that lead to

the subsolutions \( w = (w_1, w_2, ..., w_{m+n}) = (0, 0, ..., 0) \), meaning

that the initial subsolutions are not solutions of any

subproblems (sub-problems range from 1 to \( n \)); phase = 1; \( q = m+n+2 \).

Iteration.

Step 1. If \( s_q = 0 \) and phase = 1, then set phase = 2, \( q = m+n+1 \), and

redo step 1. If \( s_q < 0 \) or phase = 2, then

a) calculate \( c_j = (u_{q,1..m}A_j + u_{q,m+n+1}c_j) \), for \( j = 1, ..., n \).
b) using the two-phase revised simplex algorithm to solve sublinear problems $S_j$, for $j=1, \ldots, n$,

\[
S_j: \text{ minimize } c_jx_j, \\
\text{subject to } A_jx_j = b_j \text{ and } x_j \geq 0
\]

for optimal solutions or extreme homogeneous solutions (if $S_j$ is unbound) $x_j$, $j = 1, \ldots, n$. If any of the subproblems is infeasible, the original problem is infeasible, stop.

c) If $x_j$ is an optimal solution of $S_j$, make $a_j = (A_jx_j, 0, \ldots, 0, 1, 0, \ldots, 0, c_jx_j, 0)$, otherwise, make $a_j = (A_jx_j, 0, \ldots, 0, 0, 0, \ldots, 0, c_jx_j, 0)$.

d) For $j = 1, \ldots, n$, calculate $\delta_j = u_q \cdot a_j$. If phase = 2 then for $j = 1, \ldots, n$, calculate $\lambda_j = u_{m+n+2} \cdot a_j$ and if $\lambda_j \neq 0$ then set $\delta_j = 0$.

Step 2. Calculate $\delta_k = \min(\delta_j \mid j = 1, \ldots, n)$. If $\delta_k \geq 0$ and phase = 1, then the original LP is infeasible, stop. If $\delta_k \geq 0$ and phase = 2, then $s_q$ is the optimal solution and $-s_q$ is the minimal value of the original LP, exit. Otherwise, $a_k$ is the new column to enter the base.

Step 3. Compute $y_i = u_i \cdot a_k$, $i = 1, \ldots, q$.

Step 4. If all $y_i \leq 0$ and phase =1, then the original LP is infeasible, stop. If all $y_i \leq 0$ and phase =2, then the original LP is
unbounded, stop. Otherwise, calculate

$$\theta = \frac{s_i}{y_t} = \min_{1 \leq i \leq m+n \land y_t > 0} \left[ \frac{s_i}{y_t} \right]$$

and \( a_t \) is the column to be removed from the base.

**Step 5.** Calculate the new values of the variables in the base solution:

- \( w_t = k, s_k = \theta \)
- \( s_i = s_i - \theta y_i \quad (i \neq k, \ i = 1, \ldots, m+n+2) \)
- \( \mathbf{e}x_k = x_k, \)

and update \( U \), the inverse of the base:

- \( u_{ij} = u_{ij} - y_t u_{ij}/y_t \quad (i \neq t, \ i = 1, \ldots, m+n+2, j = 1, \ldots, m+n+2) \)
- \( u_{tj} = u_{tj}/y_t. \)

**Output.** The optimal objective value is \(-s_q\), and the optimal feasible solution (may not be basic) is \( x = (x_1, \ldots, x_n) \), where \( x_j \) is obtained from:

$$x_j = \sum_{i=1}^{\theta} s_i \mathbf{e}x_i$$
3.4. Parallelizing Decomposed Simplex Algorithm

The kernel of the procedure is the iteration of the steps 1 to 5. The data dependency graph of the iteration is shown in Figure 3.3, from which the parallelism can be easily seen as each iteration (the central iteration) requires the solutions from the subLPs, which can be solved independently. In addition, the calculation of $y_1$, ..., $y_q$ and $u_1$, ..., $u_q$ can be done in parallel.

Figure 3.3. Data Dependency Graph of The Decomposed Simplex Algorithm.
A straightforward parallelization is to invoke the subLP solvers in parallel and continue the central iteration when all of the subLP solvers finish. In this algorithm (call it SF algorithm), n processes, $p_1, p_2, ..., p_n$, are used and $p_i$ is assigned to solve the subproblem $i$ in step 1, as in Figure 3.4. When all of the subproblems find their solutions, the subprocesses send $\delta_i$'s to one of the processes (say $p_n$) and this process determines $\delta_k = \min(\delta_i)$. After $k$ is determined, it is broadcasted to all of the other processes, and the process $p_i$ calculates $y_i$ ($i=1,...,m$). Then, $y_i$'s are sent to $p_n$. $p_n$ determines $\theta_i = \min(s_i/y_i)$ and broadcasts $t$ to all of the other processes. Finally, the process $p_i$ updates $u_i$ ($i=1,...,q$), and the next iteration begins.
A timing chart of the algorithm is sketched in Figure 3.5, where $p_i$ represents process $i$, $i=1,...,n$, and the circle indicates the point in time when the optimal $x_k$ is produced.

![Timing Chart](image)

**Figure 3.5.** Timing of the Straightforward Parallel Algorithm.

The straightforward algorithm seems to exploit the inherent parallelism fairly well. However, step 2 cannot proceed until all of the subproblems finish. As indicated by [KUNG-76], the efficiency of this kind of synchronous algorithm is heavily affected by the structures of the subproblems. A synchronous algorithm performs best when the subproblems are of equal size and take the same amount of time to finish. Even with the assumption that all subproblems have the same number of constraints and same number of variables, the subproblems will take a very different number of iterations to finish, depending on the starting bases and the cost functions. It is even possible for one subproblem to find its solution
in one iteration, while another takes exponential number of
iterations ([KLEE-76]). Each central iteration has to wait until the
slowest subproblem finishes.

The algorithm has been implemented on the Sequent/Balance
shared memory machine. We ran the algorithm using 8 processors
on randomly generated decomposed LPs of 3 to 20 subproblems.
Table 3.1 summarizes the speedup of the algorithms over the
sequential algorithm. Figure 3.6. plots the speedup.

<table>
<thead>
<tr>
<th># subs</th>
<th>speedup</th>
<th>time: seqSF</th>
<th>time : paraSF</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1.97</td>
<td>3070</td>
<td>1562</td>
</tr>
<tr>
<td>4</td>
<td>2.47</td>
<td>4358</td>
<td>1764</td>
</tr>
<tr>
<td>5</td>
<td>3.10</td>
<td>12160</td>
<td>3928</td>
</tr>
<tr>
<td>6</td>
<td>3.69</td>
<td>29170</td>
<td>7900</td>
</tr>
<tr>
<td>7</td>
<td>4.06</td>
<td>41034</td>
<td>10106</td>
</tr>
<tr>
<td>8</td>
<td>3.41</td>
<td>87406</td>
<td>25662</td>
</tr>
<tr>
<td>9</td>
<td>3.01</td>
<td>152902</td>
<td>50812</td>
</tr>
<tr>
<td>10</td>
<td>2.95</td>
<td>189146</td>
<td>64158</td>
</tr>
<tr>
<td>11</td>
<td>3.20</td>
<td>352422</td>
<td>110170</td>
</tr>
<tr>
<td>12</td>
<td>4.24</td>
<td>643048</td>
<td>151668</td>
</tr>
<tr>
<td>13</td>
<td>4.63</td>
<td>721400</td>
<td>155776</td>
</tr>
<tr>
<td>14</td>
<td>5.04</td>
<td>1009842</td>
<td>200524</td>
</tr>
<tr>
<td>15</td>
<td>5.31</td>
<td>1868304</td>
<td>352124</td>
</tr>
<tr>
<td>16</td>
<td>5.33</td>
<td>1972218</td>
<td>370326</td>
</tr>
<tr>
<td>17</td>
<td>5.20</td>
<td>2971756</td>
<td>571646</td>
</tr>
<tr>
<td>18</td>
<td>5.04</td>
<td>4414118</td>
<td>876038</td>
</tr>
<tr>
<td>19</td>
<td>5.14</td>
<td>5051584</td>
<td>982986</td>
</tr>
<tr>
<td>20</td>
<td>5.27</td>
<td>6984076</td>
<td>1324426</td>
</tr>
</tbody>
</table>

Table 3.1. Speedup of Straightforward Algorithm.
From Figure 3.6, we see that the speedup is much less than the number of processors used. The speedups is at most 5 times using 8 processors (with an efficiency of less than 65%).

To elaborate on this inefficiency further, we cite the experimental data of [LINDBERG-84], which shows that, to solve the assignment problem of d dimensions (2d-1 constraints) using the standard Simplex algorithm, the number of simplex iterations follows the normal distribution with mean $\mu_d = 1.10d^{1.57}$, and standard derivation $\sigma_d = 0.33d^{1.51}$. According to this distribution, an assignment problem of d dimensions needs in the average of $1.10d^{1.57}$ iterations to solve. Assume the number of iterations for N assignment problems of d dimensions are $T_1, T_2, \ldots, T_N$, respectively.
Solving them sequentially requires a total of \( T_s = T_1 + T_2 + \ldots + T_N = N \mu_d \) iterations, and solving them in parallel using \( N \) processors needs \( T_p = \max(T_1, T_2, \ldots, T_N) \) iterations. The efficiency using \( N \) processors is:

\[
E(N) = \frac{T_s}{N \cdot T_p} = \frac{N \mu_d}{N \cdot T_p} = \frac{\mu_d}{T_p}
\]

To evaluate \( E(N) \), instead of determining the mean value of \( T_p \) analytically, we used simulation to estimate \( E(N) \). A SLAM II program ([PRITSKER-86]) is shown in Figure 3.7, and the expected efficiency for \( N = 10, d = 10 \) is 65.3%. This low efficiency matches our experimental result well.

```
INTLC, N=10, MEAN=40.86, STD=10.67;
NETWORK;
   CREATE, I,, 1000, 1;
   ASSIGN, II=1, TS=0, TP=0;
   LOOP ASSGN, TI = RNORM(MEAN, STD);
      ASSIGN, TS = TS + TI, 1;
      ACT, TI .LT. TP, SKIP;
   ACT;
   ASSIGN, TP = TI;
   SKIP ASSIGN, II = II + 1, 1;
      ACT, II .LT. N, LOOP;
   ACT;
   ASSIGN, TS = TS / N, EFF = TS / TP;
   COLCT, EFF, Efficiency;
   TERM;
ENDNETWORK;
FIN;
```

Figure 3.7. SLAM II Code for Determining Expected Efficiency of Straightforward Algorithm.
Because the big variance (\(\sigma_d\)) on the number of simplex iterations does not allow all subproblems to find their minimal solutions at the same time, new approaches or algorithm changes are needed to achieve better performance.

**Parallelizing Subproblem Solvers.**

An alternative way to parallelize the decomposed simplex algorithm is to parallelize individual subproblem solvers (see Chapter 2 or [WU-88a]). In this algorithm, step 1) can be solved as follows:

1. Perform step 1 a) in parallel;
2. Solve the n subproblems in b) one after the other in sequence and each subproblem is solved by multiple processors in parallel;
3. Execute step 1 c) and d) in parallel.

In this algorithm, no subproblem needs to wait for the other subproblems to finish. A timing chart for the algorithm is shown in Figure 3.8.

We ran the algorithm using 8 processors on randomly generated decomposed LPs of 3 to 20 subproblems. Table 3.2. summarizes the speedup of this algorithm over the sequential algorithm. Figure 3.9. plots the speedup.
Figure 3.8. Timing of Subsolver Parallelizing Algorithm.

<table>
<thead>
<tr>
<th>nbrSubs</th>
<th>SFPS/seqSF</th>
<th>seqSF</th>
<th>SFPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1.53</td>
<td>3070</td>
<td>2004</td>
</tr>
<tr>
<td>4</td>
<td>1.90</td>
<td>4358</td>
<td>2294</td>
</tr>
<tr>
<td>5</td>
<td>2.44</td>
<td>12160</td>
<td>4984</td>
</tr>
<tr>
<td>6</td>
<td>2.98</td>
<td>29170</td>
<td>9790</td>
</tr>
<tr>
<td>7</td>
<td>3.33</td>
<td>41034</td>
<td>12324</td>
</tr>
<tr>
<td>8</td>
<td>3.91</td>
<td>87406</td>
<td>22368</td>
</tr>
<tr>
<td>9</td>
<td>3.73</td>
<td>152902</td>
<td>40960</td>
</tr>
<tr>
<td>10</td>
<td>3.81</td>
<td>189146</td>
<td>49608</td>
</tr>
<tr>
<td>11</td>
<td>3.54</td>
<td>352422</td>
<td>99602</td>
</tr>
<tr>
<td>12</td>
<td>3.91</td>
<td>643048</td>
<td>164602</td>
</tr>
<tr>
<td>13</td>
<td>3.93</td>
<td>721400</td>
<td>183334</td>
</tr>
<tr>
<td>14</td>
<td>4.29</td>
<td>1009842</td>
<td>235622</td>
</tr>
<tr>
<td>15</td>
<td>4.65</td>
<td>1868304</td>
<td>401532</td>
</tr>
<tr>
<td>16</td>
<td>4.67</td>
<td>1972218</td>
<td>422358</td>
</tr>
<tr>
<td>17</td>
<td>4.29</td>
<td>2971756</td>
<td>692824</td>
</tr>
<tr>
<td>18</td>
<td>4.60</td>
<td>4414118</td>
<td>959350</td>
</tr>
<tr>
<td>19</td>
<td>4.64</td>
<td>5051584</td>
<td>1088842</td>
</tr>
<tr>
<td>20</td>
<td>4.95</td>
<td>6984076</td>
<td>1411766</td>
</tr>
</tbody>
</table>

Table 3.2. Speedup of Subsolver Parallelizing Algorithm.
The performance of the algorithm is even worse than the straightforward algorithm (see Figure 3.9). The reasons for the inefficiency are that 1) the subproblems are usually of relatively small size compared to the original problem and we know from Chapter 2 the performance drops when the input data size decreases; 2) in this algorithm, each subiteration needs a fork-join of processes (an invocation of the PAR construct, which costs CPU time), while the straightforward algorithm requires a fork-join only for each central iteration; 3) When the sizes of the subproblems are not the multiple of the number of processors, lots of last round effects are encountered, leaving several processors idle at the end of solving each subproblem. From these we conclude that parallelizing the subsolvers is not an appropriate approach to improve performance.
These two parallelizing approaches have one thing in common, that is, they both extract the parallelism in the sequential algorithm without modification to the algorithm itself. Better performance may result if we adapt the algorithm to parallel execution.

3.5. Parallel Algorithms for Decomposed Linear Programs

We note that in step 1, finding the best \( x_k \) among all of the solutions of the subproblems after waiting for all subproblems to finish is equivalent to moving from the current extreme point to an adjacent extreme point such that the objective function is improved by the greatest amount. Statistics shows that moving to the best adjacent extreme point performs only moderately better than moving to any of the adjacent extreme points which improves the objective function ([DANTZIG-63]). Thus, we can use any solution \( x_k \) that makes \( \delta_k < 0 \). In this way, there is less chance that a fast subproblem waits for a slow subproblem. We have several ways to implement this strategy.

3.5.1. First Finished First (FFF) Algorithm

Instead of finding the best solution among the subsolutions of all of the subproblems that make \( \delta_j < 0 \), we use the solution of the first finished subproblem that satisfies the condition.
Figure 3.10. Parallel FFF Algorithm (assume \( p_j \) finds \( \delta_j < 0 \)).

Figure 3.11. Timing of Parallel FFF Algorithm.
In this method, every time a subproblem $S_j$ finds its optimal solution $x_j$, it checks to see whether or not this solution makes $\delta_j = u_qa_j < 0$. When its solution satisfies the condition, it proceeds to step 2 and signals all other subprocesses to stop searching. When step 5 finishes, a new cycle starts. The modified algorithm is illustrated in Figure 3.10, and the timing chart of the algorithm is sketched in Figure 3.11 (where the circle indicates when the qualified solution $x_k$ is found).

The above discussion assumes that there are unlimited number of processors available for the parallel computation. However, in reality, the number of processors, $P$ may be less than the number of subproblems, $n$. In this case, $n$ subproblems will be solved in $n/P$ rounds, and in each round $P$ subproblems are solved. Notice that, the subproblem that could finish first when there are unlimited processors may be scheduled in a later round, so it may actually finish later than several other subproblems.

When we have only limited processor resource, we have no way to implement the pure FFF algorithm, since at least one of the subproblems in the first round have to be finished before going to the next round. If the fastest subproblem in not in the first round, the finished subproblem in the first round has be executed more iterations than it should be in the pure FFF algorithm.

Two approximations can be used to implement the FFF
algorithm. In the first implementation, whenever a subproblem finds a qualified solution in a round, this subproblem is considered the first finished subproblem, and all the subproblems in the remaining rounds will not be executed and a new central iteration is pursued. In the second implementation, the subproblems in all rounds are always executed, and the best subproblem in all of the finished ones is chosen to continue the central iteration. Although the first approach finds a qualified solution faster in a few early central iterations, it actually results in slower convergence. Figure 12. shows our experimental result of the two implementations on LPs of 3 to 16 subproblems using 8 processors. When number of processors is less than the number of the subproblems, the two implementations show the similar performance, and when the number of the subproblems is more than the number of the processors, the second implementation clean outperforms the first approach.

![Graph showing performance of two FFF implementations.](image)

Figure 3.12. Performance of Two FFF Implementations.
3.5.2. Tightly Synchronous (TS) Algorithm

We further notice that the minimal solution of one subproblem may not be as good as the non-minimal solutions of the other subproblems. For example, it is possible that a subproblem that takes a very long time to find a minimal solution may have already found a non-minimal solution that is better than the minimal solutions of the other subproblems. When this happens, the FFF algorithm will ignore these good solutions.

In the TS algorithm described here, instead of determining optimal solutions, each subproblem sends its current solution (not necessarily optimal) to the central process after some number of iterations. The central process selects from the n solutions the one that makes \( \delta_j = u_q a_j \) negative if one exists, or else repeatedly invokes the subproblems. If we assume that the subproblems are the same size, then all subproblems will take equal time to finish a single iteration, and this algorithm can synchronize all subproblems after they perform an equal amount of computation. The TS algorithm is shown in Figure 3.13. The timing of the algorithm can be sketched as in Figure 3.14 (where the circle indicates when the qualified solution \( x_k \) is found).
If the subproblems have different sizes, it is not difficult to assign subproblem $S_j$ a constant $I_{ij}$ as the number of iterations it should perform in central iteration $i$ before participating in the synchronization, so that all subproblems take the same amount of
time to finish. Assume it is known that in the average $\text{SI}_i$ subiterations should be performed before a qualified subsolution can be found in central iteration $i$, and subproblem $j$, $1 \leq j \leq n$, takes $C_j$ time units to execute one subiteration. In order to find the number of iterations that the subproblem $j$ should perform before participating in synchronization, we need to find $I_{ij}$, $1 \leq j \leq n$, such that

$$\frac{1}{n} \sum_{j=1}^{n} I_{ij} = \text{SI}_i \text{ and } C_j I_{ij} = C_j n I_{ij}^j, j' \neq j''.$$  

An approximation is to let $I_{ij} = \frac{\text{SI}_i}{n} \sum_{j=1}^{n} C_j$.

There is synchronization overhead associated with the TS algorithm, if $\text{SI}_i$ (call it the synchronization interval in central iteration $i$) is not chosen properly. For example, if $R_i$ rounds of synchronization take place before a qualified solution is found in central iteration $i$, the TS algorithm has additional overhead of $(R_i - 1) T(\text{step 1 c and d})$. On the one hand, we may want to reduce $R_i$, by increasing the number of subiterations that the subproblems should execute before participating in synchronization. In the extreme, we may let every subproblem execute until it finds the minimal subsolution. In this case, $R_i$ is always 1, and the TS algorithm becomes the straightforward algorithm. On the other hand, we may want to let the subproblems participate in synchronization more frequently, which may cause $R_i >> 1$. We want the synchronization interval $\text{SI}_i$ to be the minimal number of subiterations that the subproblems have to perform in central iteration $i$ before finding out
a qualified subsolution.

In order to determine $SI_i$, we did experiment to see how many central iterations take 1, 2, 3, or 4 and more subiterations. The result is shown in Table 3.3, obtained by experiment on the random LPs of $P=3$ to 15 subproblems, each subproblem has $m = P/2 - 1$, $P/2$, and $P/2 + 1$ constraints and $2*m$ variables. In Table 3.3, we see that all but one central iterations takes only one subiteration (for example, when $\#$ subs = 10 and $m = 4$, 47 out of 48 central iterations take only one subiteration).

<table>
<thead>
<tr>
<th>m</th>
<th>#sub</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>19</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>17</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>21</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>128</td>
</tr>
<tr>
<td>7</td>
<td>19</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>23</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>133</td>
</tr>
<tr>
<td>8</td>
<td>28</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>35</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>146</td>
</tr>
<tr>
<td>9</td>
<td>37</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>39</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>148</td>
</tr>
<tr>
<td>10</td>
<td>47</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>52</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>166</td>
</tr>
<tr>
<td>11</td>
<td>41</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>63</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>164</td>
</tr>
<tr>
<td>12</td>
<td>55</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>70</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>189</td>
</tr>
<tr>
<td>13</td>
<td>56</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>82</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>195</td>
</tr>
<tr>
<td>14</td>
<td>88</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>124</td>
</tr>
<tr>
<td>15</td>
<td>82</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>197</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>124</td>
</tr>
</tbody>
</table>

Table 3.3. Synchronization Intervals of TS algorithm.

We did further experiments to determine which central iteration is the one that takes more than one subiterations. The answer is always the first central iteration. More importantly, we found that the first central iteration always takes exactly $m+1$ subiterations when the subproblems consist of $m$ constraints.
An explanation to the above fact is that, in the first central iteration, every subsolvers must run through a complete phase 1 to determine an initial feasible solution. A complete phase 1 usually takes more than m iterations for an LP consists of m constrains. Among the multiple subproblems, the probability that at least one subproblem has a qualified solution right after the m'th iteration is very high (our experiment shows that this probability is 1). On the other hand, in the later central iterations, the probability that at least one subproblem has a qualified solution right after one iteration is close to one.

According to this experiment result, we choose the synchronization interval as

\[ S_{i} = \begin{cases} 
  m + 1, & i = 1 \\
  1, & i > 1 
\end{cases} \]

### 3.5.3. Lookahead First Finish First (FFFL) Algorithm

The consideration that leads to the lookahead algorithms is the observation that the optimal solution of a subproblem constantly changes as the objective vector of the subLPs changes. A non-minimal solution for one objective vector may be optimal or very close to optimal for another objective vector. This suggests that, during steps 2 to 5 of the central iteration, the subproblems should not wait for the next iteration to start. Instead, the subprocesses can
keep optimizing on the current objective functions, and when the next iteration starts, the solutions of some subproblems may already be good enough to satisfy the condition \( \delta_j < 0 \). So step 2 of the next iteration of the central problem can start immediately.

The lookahead algorithms use \( n \) processes (the subprocesses) for the \( n \) subproblems. An \( n+1 \)'th process (the central process) controls the central problem. These processes all share a global value \( u_q \), and each subprocess \( S_j \) maintains the values \( \delta_j \) and \( x_j \) which can be are accessed by the central process.

Each subprocess calculates its own objective vector from the global \( u_q \) that is updated by the central process. When it finds a solution with \( \delta_j < 0 \), it recommends the solution (\( x_j \) and \( \delta_j \)) to the central process. The central process periodically checks whether or not any of the \( \delta_j \) is negative, and once it finds such a \( \delta_j \), it starts steps 2 to 5. Meanwhile, the subproblems continue optimizing on the current objective vectors. When the central process finishes updating \( u_q \), the subproblems update their objective vectors.

The Lookahead FFF algorithm is based on the FFF algorithm. When a subproblem finds its optimal solution that makes \( \delta_j < 0 \), it will not signal the other subproblems to stop, so the other subproblems continue running as the central problem proceeds. The algorithm is shown in Figure 3.15. The timing of the algorithm is shown in Figure 3.16.
Figure 3.15. Lookahead FFF Algorithm (assume $p_i$ finds $\delta_i < 0$).

Figure 3.16. Timing of Lookahead FFF Algorithm.
3.5.4. **Wypior's Approach**

Wypior's approach is a variation of the lookahead FFF algorithm. In this approach, $n$ processes, $p_1, p_2, ..., p_n$, are assigned to solve the $n$ subproblems, and these processes continually perform steps 1 a) to c), send the result $a_j$'s to another process $p_A$, and wait for the $u_q$ before performing the next iteration. Process $p_A$ continually collects $a_j$'s and performs steps 1 d) and 2, and if it finds an $a_k$ that makes $\delta_k < 0$, it sends the $a_k$ to yet another process $p_B$. Process $p_B$ continually asks for $a_k$ from $p_A$ and performs steps 3, 4, and 5. This situation can be described in Figure 3.17. The timing of the algorithm is shown in Figure 3.18.

![Figure 3.17. Wypior's Algorithm.](image)
The reason for using process PA is that passing the solutions from p_i, i=1,...,n, to PB may take some non-trivial time, especially in a message passing system. Using PA, the collection of the solutions from p_1,...,p_n can be overlapped with the update of u done by PB.

One drawback of this method is that PA can be the bottleneck of the algorithm, as all of \delta_j's are calculated in PA sequentially without overlapping with any other processes and each \delta_j needs an inner product operation. An improvement is to let each subproblem solver calculate \delta_j and send \delta_j and a_j to PA. In this way, PA only needs to determine \delta_k = \min(\delta_j) and send a_k to PB. Even with this modification, we see that among all of the vector a_j's sent to PA, only a_k is used in later computation. We can further modify the algorithm so that each process p_j only sends \delta_j to PA, and when PA finds \delta_k, it sends k to process p_k, asking for the vector a_k; and p_k directly sends
With these two modifications, only very few data are shared among pj's and PA and PB. With the decreased data sharing, however, the consideration that leads to the necessity of PA is no longer valid, and we can merge PA to PB, thus resulting in the lookahead FFF algorithm. For this reason, we consider Wypior's algorithm less efficient than the lookahead FFF algorithm and will not evaluate Wypior's algorithm further.

3.5.5. Lookahead Tightly Synchronous (TSL) Algorithm

![Lookahead TS Algorithm Diagram]

Figure 3.19. Lookahead TS Algorithm.
Another lookahead algorithm is based on the tightly synchronized algorithm. In this algorithm, each subproblem checks to see whether or not its current solution makes $\delta_j < 0$ after a constant number of iterations. When one such solution is found, the subproblems continue running as the central problem proceeds. The algorithm is shown in Figure 3.19. The timing of the algorithm is shown in Figure 3.20.

![Figure 3.20. Timing of Lookahead TS Algorithm.](image)

3.5.6. **Processor Assignments for Lookahead Algorithms**

In SF, FFF, and TS parallel algorithms, all processors can be used to parallelize the central pivot operation after a qualified subsolution is found in step 1. In the lookahead algorithms (FFFL, TSL), after a qualified solution is found, the processors assigned to the subproblems continue to lookahead for new subsolutions. Thus the central pivot operation requires additional processors. For a system that has limited number of processors, we have to be careful
about how many processors should be given to the subproblem solvers and how many should be given to the central solver. For this reason, we take a close look into the lookahead algorithms.

To overlap central pivot operation with the subiterations, we partition the solving of subproblems into two stages: Stage I) for basic subiterations that determine the qualified solution and Stage II) for lookahead subiterations. As the basic stage can not begin until new \( U_q \) is updated by the central pivot operation, we also partitioned a central pivot operation into two stages: Stage III) for updating \( U_q \) and Stage IV) for updating \( U_1, ..., U_{q-1} \). Using these partitions, we can overlap the subiterations and the central pivot operations as follows:

<table>
<thead>
<tr>
<th>subiterations</th>
<th>central pivots</th>
</tr>
</thead>
<tbody>
<tr>
<td>I) basic</td>
<td>III) update ( U_q )</td>
</tr>
<tr>
<td>II) lookahead</td>
<td>IV) update ( U_1, ..., U_{q-1} )</td>
</tr>
<tr>
<td>I) basic</td>
<td>III) update ( U_q )</td>
</tr>
<tr>
<td>II) lookahead</td>
<td>IV) update ( U_1, ..., U_{q-1} )</td>
</tr>
</tbody>
</table>

The overlapping has the best efficiency if stages I and IV, II and III take the same amount of time to complete. Assume that we have a total of \( P \) processors, and \( X \) of them are for the subiterations and \( Y \) for central pivot operations. The complexity of the stages can be summarized as follows:
<table>
<thead>
<tr>
<th>Stage</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage I)</td>
<td>$48m^2 \cdot n / X;$</td>
</tr>
<tr>
<td>Stage II)</td>
<td>$48m^2 \cdot n / X;$</td>
</tr>
<tr>
<td>Stage III)</td>
<td>$24m^2 / Y;$</td>
</tr>
<tr>
<td>Stage IV)</td>
<td>$64m^2 / Y.$</td>
</tr>
</tbody>
</table>

Based on this data, we find that when $X = P - 1$ and $Y = 1$ the stages I and IV, and stages II and III roughly take the same time to complete, for $n > P$. In other words, we need to specify only one processor for the central pivot operation.

### 3.5.7. Performance Comparison

<table>
<thead>
<tr>
<th>nbrSubs</th>
<th>seqSF</th>
<th>paraSF</th>
<th>paraFFF</th>
<th>paraTS</th>
<th>paraFFFL</th>
<th>paraTSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3070</td>
<td>1562</td>
<td>1436</td>
<td>1108</td>
<td>1492</td>
<td>1600</td>
</tr>
<tr>
<td>4</td>
<td>4358</td>
<td>1764</td>
<td>1808</td>
<td>1228</td>
<td>2160</td>
<td>1982</td>
</tr>
<tr>
<td>5</td>
<td>12160</td>
<td>3928</td>
<td>3708</td>
<td>2420</td>
<td>4308</td>
<td>4542</td>
</tr>
<tr>
<td>6</td>
<td>29170</td>
<td>7900</td>
<td>6534</td>
<td>3808</td>
<td>8658</td>
<td>8114</td>
</tr>
<tr>
<td>7</td>
<td>41034</td>
<td>10106</td>
<td>7992</td>
<td>4830</td>
<td>10884</td>
<td>10034</td>
</tr>
<tr>
<td>8</td>
<td>87406</td>
<td>25662</td>
<td>21756</td>
<td>7446</td>
<td>19146</td>
<td>18326</td>
</tr>
<tr>
<td>9</td>
<td>152902</td>
<td>50812</td>
<td>44418</td>
<td>22048</td>
<td>31360</td>
<td>28514</td>
</tr>
<tr>
<td>10</td>
<td>189146</td>
<td>64158</td>
<td>63802</td>
<td>21952</td>
<td>35280</td>
<td>36406</td>
</tr>
<tr>
<td>11</td>
<td>352422</td>
<td>110170</td>
<td>142150</td>
<td>34570</td>
<td>63430</td>
<td>50584</td>
</tr>
<tr>
<td>12</td>
<td>643048</td>
<td>151668</td>
<td>219028</td>
<td>54724</td>
<td>94854</td>
<td>87684</td>
</tr>
<tr>
<td>13</td>
<td>721400</td>
<td>155776</td>
<td>263860</td>
<td>54888</td>
<td>93038</td>
<td>89708</td>
</tr>
<tr>
<td>14</td>
<td>1009842</td>
<td>200524</td>
<td>382862</td>
<td>75660</td>
<td>144098</td>
<td>113630</td>
</tr>
<tr>
<td>15</td>
<td>1866304</td>
<td>352124</td>
<td>670166</td>
<td>107174</td>
<td>213986</td>
<td>216058</td>
</tr>
<tr>
<td>16</td>
<td>1972218</td>
<td>370326</td>
<td>751294</td>
<td>114248</td>
<td>216034</td>
<td>213598</td>
</tr>
<tr>
<td>17</td>
<td>2971756</td>
<td>571646</td>
<td>1189230</td>
<td>226332</td>
<td>313058</td>
<td>310242</td>
</tr>
<tr>
<td>18</td>
<td>4414118</td>
<td>876038</td>
<td>805628</td>
<td>296108</td>
<td>463522</td>
<td>425990</td>
</tr>
<tr>
<td>19</td>
<td>5051584</td>
<td>982986</td>
<td>896552</td>
<td>314346</td>
<td>473002</td>
<td>486510</td>
</tr>
<tr>
<td>20</td>
<td>6984076</td>
<td>1324426</td>
<td>1059104</td>
<td>389814</td>
<td>661032</td>
<td>631356</td>
</tr>
</tbody>
</table>

Table 3.4. Execution Times of Parallel and Sequential Algorithms.
Figure 3.21. Speedup of Parallel Algorithms Over the Sequential Algorithm.

Table 3.4. summarizes the execution times (milliseconds) of the parallel algorithms and the sequential algorithm on the Sequent/Balance machine using 8 processors. The input LPs consist of $n=3$ to 20 subproblems, each with $m=2+2n/3$ constraints and $v=2m$.
variables. For each fixed triple (n,m,v), 5 different cases are run and averaged. Figure 3.21. plots the speedup of the parallel algorithms over the sequential algorithm.

The parallel TS algorithm has a peak speedup of more than twice the number of processors used (see Figure 3.21). The parallel FFF algorithm has a speedup a little less than the linear speedup. The Lookahead algorithms (FFFL and TSL) have nearly linear speedup in the number of processors used. All of the algorithms here perform much better than the parallel SF algorithm, which has only half of linear speedup.

3.5.8. Fast Sequential Algorithm

In the above, we showed that the parallel TS algorithm achieves more than 2*P speedup over the sequential algorithm. This implies that the TS criterion can reduce the execution time of the sequential algorithm by half as well. In order to see whether or not this is the case, we implemented a particular version of the sequential algorithm which uses the TS criterion. Table 3.5. summarizes the execution time of the new sequential algorithm together with that of the normal sequential algorithm and TS algorithm. Figure 3.22. plots the execution times of the two sequential algorithms, and Figure 3.23. plots the speedup of the TS algorithm over the two sequential algorithms.
Table 3.5. Execution Time of Sequential SF, TS and Parallel TS Algorithms.

<table>
<thead>
<tr>
<th>#subs</th>
<th>normal seq</th>
<th>seq TS</th>
<th>parallel TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3070</td>
<td>2518</td>
<td>1108</td>
</tr>
<tr>
<td>4</td>
<td>4358</td>
<td>3408</td>
<td>1228</td>
</tr>
<tr>
<td>5</td>
<td>12160</td>
<td>9190</td>
<td>2420</td>
</tr>
<tr>
<td>6</td>
<td>29170</td>
<td>17914</td>
<td>3808</td>
</tr>
<tr>
<td>7</td>
<td>41034</td>
<td>27220</td>
<td>4830</td>
</tr>
<tr>
<td>8</td>
<td>87406</td>
<td>49434</td>
<td>7446</td>
</tr>
<tr>
<td>9</td>
<td>152902</td>
<td>92560</td>
<td>22048</td>
</tr>
<tr>
<td>10</td>
<td>189146</td>
<td>100020</td>
<td>21952</td>
</tr>
<tr>
<td>11</td>
<td>352422</td>
<td>174810</td>
<td>34570</td>
</tr>
<tr>
<td>12</td>
<td>643048</td>
<td>302282</td>
<td>54724</td>
</tr>
<tr>
<td>13</td>
<td>721400</td>
<td>327004</td>
<td>54888</td>
</tr>
<tr>
<td>14</td>
<td>1009842</td>
<td>483288</td>
<td>75660</td>
</tr>
<tr>
<td>15</td>
<td>1868304</td>
<td>734916</td>
<td>107174</td>
</tr>
<tr>
<td>16</td>
<td>1972218</td>
<td>834966</td>
<td>114248</td>
</tr>
<tr>
<td>17</td>
<td>2971756</td>
<td>1224076</td>
<td>226332</td>
</tr>
<tr>
<td>18</td>
<td>4414118</td>
<td>1668260</td>
<td>296108</td>
</tr>
<tr>
<td>19</td>
<td>5051584</td>
<td>1868262</td>
<td>314346</td>
</tr>
<tr>
<td>20</td>
<td>6984076</td>
<td>2432204</td>
<td>389814</td>
</tr>
</tbody>
</table>

Figure 3.22. Execution Times of SF/TS Sequential Algorithms.
3.5.9. Performance vs. Number of Processors

In order to observe the behavior of the parallel TS algorithm when the number of processors changes, we run the TS parallel algorithm using 3 to 8 processors. The 5 input LPs are fixed to have 16 subproblems, each with 12 constraints and 24 variables. Table 3.6. summarizes the execution time of the parallel TS algorithm and
speedup of the parallel TS algorithm over the sequential algorithms. Figure 3.24. plots the corresponding speedup.

<table>
<thead>
<tr>
<th>alg/#procs</th>
<th>paraTS(time)</th>
<th>seqTS/paraTS</th>
<th>seqTS/paraSF</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq SF</td>
<td>1972218</td>
<td></td>
<td></td>
</tr>
<tr>
<td>seq TS</td>
<td>834966</td>
<td></td>
<td></td>
</tr>
<tr>
<td>paraTS/3</td>
<td>329774</td>
<td>2.53</td>
<td>5.98</td>
</tr>
<tr>
<td>paraTS/4</td>
<td>224636</td>
<td>3.72</td>
<td>8.78</td>
</tr>
<tr>
<td>paraTS/5</td>
<td>219304</td>
<td>3.81</td>
<td>8.99</td>
</tr>
<tr>
<td>paraTS/6</td>
<td>167860</td>
<td>4.97</td>
<td>11.75</td>
</tr>
<tr>
<td>paraTS/7</td>
<td>167120</td>
<td>5.00</td>
<td>11.80</td>
</tr>
<tr>
<td>paraTS/8</td>
<td>114640</td>
<td>7.28</td>
<td>17.20</td>
</tr>
</tbody>
</table>

Table 3.6. Execution Time of Sequential TS, SF, and Parallel TS and The Speedup of Parallel TS Over Sequential TS and Sequential SF.

Figure 3.24. Speedup of Parallel TS Over Sequential TS and SF When Number of Processors Changes From Three to Eight.
From Figure 3.24, we see that, as the number of processors increases from 3 to 8, the parallel TS algorithm has nearly linear speedup over the sequential TS algorithm, and the parallel TS algorithm has improved the performance of the normal sequential algorithm by twice the linear speedup.

3.5.10. Removing Last Round Effects

In Figure 3.23, a drop-off in speedup occurs when the number of subproblems goes from P to P + 1, where P is the number of processors used. This because when the N subproblems are executed by P processors in parallel, they are solved in ⌈N/P⌉ rounds, and in the last round only N MOD P subproblems are solved by N MOD P processors and the remaining processors are left idle.

We also observe last round effects in Figure 3.24. For the input LPs of 16 subproblems, the parallel TS algorithm has the best performance when the number of processors used is a divisor of 16. For example, when 4 (a divisor of 16) processors are used, the parallel TS algorithm has a speedup of 3.72 over the sequential TS algorithm. But when the number of processors increases from 4 to 7, the speedup only increases from 3.72 to 5.0. When the number of processors changes from 7 to 8 (a divisor of 16), the speedup jumps from 5.0 to 7.28.
Figure 3.25. Balanced Performance of Parallel TS Algorithm.

This is the typical processor load balance problem ([COFFMAN-76]). The performance drop-off can be prevented using the Loop Spreading technique described in Chapter 4. Figure 3.25. shows the
balanced speedup of the parallel TS algorithm over sequential TS algorithm and the efficiency of the parallel TS algorithm when loop spreading is used.

Comparing the results in Figure 3.25. to those in Figure 3.23, we see that the performance of the parallel TS algorithm is quite stable, showing an efficiency of around 90%, without drop-off when the number of subproblems changes.

Table 3.7. and Figure 3.26. show the balanced speedup of the parallel TS algorithm with loop spreading over sequential SF and TS algorithms when the number of processors changes from 3 to 8. From Figure 3.26, we see that the performance of the parallel TS algorithm is always more than two times the linear speedup when compared to the sequential algorithm and is very close to linear speedup over the fast sequential algorithm.

<table>
<thead>
<tr>
<th>alg/#procs</th>
<th>spreadTS(time)</th>
<th>spreadTS\seqTS</th>
<th>spreadTS\seqSF</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq SF</td>
<td>1972218</td>
<td>218</td>
<td>2.86</td>
</tr>
<tr>
<td>seq TS</td>
<td>834966</td>
<td>834966</td>
<td>2.86</td>
</tr>
<tr>
<td>3</td>
<td>291816</td>
<td>6.76</td>
<td>2.86</td>
</tr>
<tr>
<td>4</td>
<td>215034</td>
<td>9.17</td>
<td>3.88</td>
</tr>
<tr>
<td>5</td>
<td>184358</td>
<td>10.70</td>
<td>4.53</td>
</tr>
<tr>
<td>6</td>
<td>158664</td>
<td>12.43</td>
<td>5.26</td>
</tr>
<tr>
<td>7</td>
<td>132008</td>
<td>14.94</td>
<td>6.33</td>
</tr>
<tr>
<td>8</td>
<td>116706</td>
<td>16.90</td>
<td>7.15</td>
</tr>
</tbody>
</table>

Table 3.7. Execution Time of Sequential TS, SF, and Spread TS and the Speedup of Spread TS Over Sequential TS and Sequential SF.
Figure 3.26. Speedup of Spread TS over sequential TS and SF
When Number of Processors Changes From Three to Eight.

3.5.11. Why TS algorithm is Good

The evolution from SF to FFF and then to the TS algorithm aims at speeding up the step 1 of the decomposed simplex algorithm. We call this optimization local optimization. The side effect of this optimization is that the time saving in step 1 might increase the execution time of the step 1 in the next central iteration. Also, it might increase the total number of central iterations (and also the total number of subiterations) required to solve the given LP. We call the minimization of the total number of sub/central iterations the global optimization.
To see the relative goodness of the three algorithms in local optimization, we collected the number of subiterations in central iterations for each of the algorithms, shown in Tables 8 to 10. These data are the results of the experiment on the random LPs of $P=3$ to $15$ subproblems, each subproblem with $m = P/2 - 1$, $P/2$, and $P/2+1$ constraints and $2m$ variables. For each $(P, m, v)$, we collected the number of central iterations that requires 1, 2, 3, and 4 or more subiterations.

From Tables 8 to 10, the number of central iterations that require more than one subiteration decreases. In Table 3.8, the SF algorithm takes lots of central iterations that require 4 or more subiterations. For example, when $P=14$, $m=6$, 53 out of 87 central iterations take 4 or more subiterations. In Table 3.9, the situation gets a little better: only 26 out of 86 central iterations take 4 or more subiterations. From Table 3.10 we see that for the TS algorithm, all but one of the central iterations take 1 subiteration. We conclude that the TS algorithm does have the best local optimizability.
Table 3.8. Local Optimizability of SF Criterion.

<table>
<thead>
<tr>
<th>m</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>#subs</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>8 2 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>9 3 0 1 9 5 2 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>12 4 0 1 6 4 4 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>13 4 1 1 8 7 3 1 8 5 11 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>18 6 0 1 12 8 5 2 15 7 6 7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>11 9 5 2 14 12 9 7 10 10 14 16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>15 7 7 2 12 4 6 11 13 10 9 15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>16 6 7 14 14 5 7 30 11 10 17 33</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>10 4 11 16 14 10 21 23 13 2 9 38</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>18 7 9 24 13 8 9 43 12 3 9 56</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>19 7 13 30 17 11 14 42 16 11 11 52</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>17 9 6 53 18 6 10 56 16 13 7 82</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>18 11 15 57 17 5 11 67 20 3 13 81</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.9. Local Optimizability of FFF Criterion.

<table>
<thead>
<tr>
<th>m</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>#subs</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>8 2 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>11 2 0 1 9 5 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>13 3 0 1 8 4 3 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>13 5 0 1 11 6 1 11 9 7 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>14 6 0 1 13 10 3 119 8 4 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>15 12 3 1 18 10 6 4 19 11 12 7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>17 6 4 16 7 7 5 18 12 7 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>22 5 8 7 20 14 12 10 26 14 11 18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>20 10 6 3 20 16 10 10 22 14 10 19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>28 13 10 9 24 11 14 21 23 17 12 32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>21 17 11 16 28 16 8 21 25 19 9 34</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>28 23 13 26 26 18 19 30 41 14 17 48</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>27 23 12 25 33 22 16 36 30 15 22 46</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.10. Local Optimizability of TS Criterion.
<table>
<thead>
<tr>
<th># subs</th>
<th>SF Alg.</th>
<th>FFF Alg.</th>
<th>TS Alg.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>total subs</td>
<td>total cens</td>
<td>total subs</td>
</tr>
<tr>
<td>3</td>
<td>61</td>
<td>10</td>
<td>61</td>
</tr>
<tr>
<td>4</td>
<td>111</td>
<td>13</td>
<td>110</td>
</tr>
<tr>
<td>5</td>
<td>176</td>
<td>16</td>
<td>177</td>
</tr>
<tr>
<td>6</td>
<td>346</td>
<td>25</td>
<td>323</td>
</tr>
<tr>
<td>7</td>
<td>408</td>
<td>27</td>
<td>405</td>
</tr>
<tr>
<td>8</td>
<td>700</td>
<td>36</td>
<td>670</td>
</tr>
<tr>
<td>9</td>
<td>1053</td>
<td>46</td>
<td>1037</td>
</tr>
<tr>
<td>10</td>
<td>1540</td>
<td>57</td>
<td>1328</td>
</tr>
<tr>
<td>11</td>
<td>2088</td>
<td>66</td>
<td>2062</td>
</tr>
<tr>
<td>12</td>
<td>2639</td>
<td>74</td>
<td>2594</td>
</tr>
<tr>
<td>13</td>
<td>3412</td>
<td>84</td>
<td>3256</td>
</tr>
<tr>
<td>14</td>
<td>4734</td>
<td>103</td>
<td>4259</td>
</tr>
<tr>
<td>15</td>
<td>5991</td>
<td>114</td>
<td>5356</td>
</tr>
<tr>
<td>16</td>
<td>7732</td>
<td>136</td>
<td>7016</td>
</tr>
<tr>
<td>17</td>
<td>9226</td>
<td>141</td>
<td>8126</td>
</tr>
<tr>
<td>18</td>
<td>12412</td>
<td>183</td>
<td>10788</td>
</tr>
<tr>
<td>19</td>
<td>13388</td>
<td>177</td>
<td>11999</td>
</tr>
<tr>
<td>20</td>
<td>16389</td>
<td>208</td>
<td>17133</td>
</tr>
</tbody>
</table>

Table 3.11. Global Optimizability.

---

![Graph](image)

**Figure 3.27.** Total Numbers of Subiterations.
In order to compare the parallel algorithms in global optimization, we collected in Table 3.11 the total number of subiterations and the total number of central iterations taken to solve random LPs. These LPs range from \( P = 3 \) to \( P = 20 \) subproblems, each with \( m = P/2 - 1, \ P/2, \ \text{and} \ P/2+1 \) constraints and \( 2m \) variables. Figure 3.27. plots the total number of subiterations, and Figure 3.28. shows the total number of central iterations.

From Figure 3.27., we see that the TS algorithm has only about half of the total number subiterations used by the SF algorithm, and the SF and FFF algorithms have similar total numbers of subiterations. From Figure 3.28., we further see that all three algorithms have similar numbers of central iterations. It is clear that algorithm TS is not only the best in local optimization but also
the best in global optimization. This conforms our experimental resul

3.6. Conclusions

Direct parallelization of the sequential algorithm yields very limited performance improvement using multiple processors. For example, without changing the algorithm itself, the performance of the sequential decomposed simplex algorithm can be improved by only half the number of processors used.

We discovered four new ways to parallelize the decomposed simplex algorithm. The parallel TS algorithm can achieve more than 2*P times performance improvement over the sequential algorithm using P processors. Furthermore, sequential execution of the TS algorithm runs more than 2 times faster than the original sequential algorithm.
Chapter 4

Parallel Processor Balance Through Loop Spreading

Abstract

When the number of processors $P$ is less than the number of tasks $N$ in a parallel loop construct, the tasks have to be executed in $\lceil N/P \rceil$ rounds and the last round executes only $(N \mod P)$ tasks. In many cases, $N \mod P$ is close to one, so in the last round all but one processor is idle, which causes a significant drop in performance. This performance drop becomes more and more detrimental as the number of processors increases. Loop spreading is a technique for restructuring parallel loops so as to balance parallel tasks on multiple processors. A spread loop runs at least as fast as the nonspread loop even when $N \mod P = 0$, and shows no performance drop when $N$ changes. We show how the method keeps the performance of the matrix multiplication and simplex algorithm from decreasing as the input data changes size.

4.1. Introduction

Parallelizing a sequential program with an automatic compiler is desirable not only because of the need to restructure large amounts of existing sequential software, but also because programmers become less competent at optimization as computer complexity increases ([LAMPORT-75], [APPELBE-85], [LUBECK-85]). It has been long recognized that designing parallel programs imposes
a heavy burden on programmers -- it is sometime difficult for a programmer to ensure that the components of a parallel program do not interact unexpectedly. For example, when a programmer has to decide whether to use a PAR $i := 1 \text{ TO } n \text{ DO } S \text{ ENDPAR}$ parallel construct or a FOR $i := 1 \text{ TO } n \text{ DO } S \text{ ENDFOR}$ sequential construct, he/she has to find out whether or not the executions of $S$ in different iterations will interfere with one another. To discover such an interference, the programmer may have to mentally unwind the loop and imagine the many possibilities of interactions between different iterations. People tend to ignore such considerations, while a computer tool can analyze such loops faster and with greater precision.

There are rather sophisticated techniques for parallelizing FOR loops (for example, see [PADUA-86]). However, one problem that has been ignored in parallelizing FOR loops is the "processor balance problem". When the number of processors $P$ is less than the number of tasks $N$ in a parallel loop construct, the tasks are usually executed in $\lceil N/P \rceil$ rounds with the last round executing only $(N \mod P)$ tasks. In the worst case when $N \mod P$ is one, all but one processor are idle in the last round, and the performance of the parallelized program degrades sharply. This performance drop becomes more and more pronounced as the number of processors increases. For example, a chart describing the performance of the parallel decomposed simplex algorithm using 8 processors is shown in Figure 4.1 ([WU-88b]), in which, when the input data changes from 8 to 9 subproblems, the
speedup drops from 6+ to around 4. The reason for such a drop is that when using 8 processors, a parallel loop of 9 tasks must be executed in two rounds. The second round uses only one processor leaving the other 7 processors idle.

![Speedup Chart](image_url)

Figure 4.1. Effect of Processor Load Balance on Performance of parallel simplex algorithm using Eight Processors.

The problem of unbalanced processor load when executing parallel loops on multiple processors is not peculiar to the parallel simplex algorithm. This phenomenon can be observed even more clearly in our experiment with the parallel matrix multiplication algorithm. Figure 4.2. shows the speedups of the parallel algorithm over the sequential algorithm using 6 to 9 processors with input matrix sizes ranging from 20 to 50.
From Figure 4.2, we see that not only the performance drops when the input size is not a multiple of the number of processors, but also that the drop becomes more significant when the number of processors increases. It is desirable to devise a strategy that keeps performance at the peak value.

**Loop Spreading**

Loop spreading is a technique for automatically restructuring parallel loops so as to balance the parallel tasks better on multiple processors. We describe loop spreading by first studying an example.
EXAMPLE 1. Assume in the following loop

```
FOR i:=1 TO 4 DO
    s1(i);
    s2(i);
    s3(i);
END FOR;
```

sj(i)'s are independent, and they all take T time units to execute on a single processor. Also assume that we have 3 processors to run the loop in parallel. The usual way to execute a loop in parallel is to let each processor execute one iteration and if more iterations remain then let each processor execute one more iteration and so on. In this way, at least one processor needs to execute two iterations of the above loop and the total execution time will be 6 * T. However, if we are allowed to run s1(4), s2(4), and s3(4) on multiple processors, as shown in Figure 4.3, we can reduce the total execution time to 4*T, which is a significant performance improvement.

Figure. 4.3. Loop Spreading for Parallel Load Balance.
In the following, we study ways to restructure parallel loops that balance processor loads with minimal additional overhead. In Section 2 we first present the theoretical background. In Section 3 we introduce loop spreading and the problems we want to solve. In Section 4 we study the case when the statements inside a loop are independent, and in Section 6 we discuss how loop spreading works when the statements are not independent. In Section 7, we study in depth how to reduce the number of synchronization operations that are required for restructuring the loops whose substatements are dependent. In Sections 5 and 8 we present the experimental results on matrix multiplication and Simplex algorithm using our new methods. In Section 9, we compare our method with a similar method called "loop collapsing". Finally in Section 10 we summarize the results.

4.2. Data Dependence and Process Scheduling

We first introduce the concepts of iteration vector, data dependence, and loop carried dependence ([ALLEN-83]).

Iteration vector. For a k-level nested loop, a vector \((i_1,i_2,...,i_k)\) is called an iteration vector if the loop body can be executed when the \(j\)'th level loop is in the \(i_j\)'th iteration for all \(j = 1,...,k\).
We use \( S(I) \) to denote the execution of statement \( S \) during the iteration \( I = (i_1, i_2, \ldots, i_k) \).

**Data dependence.** Statements \( S_1 \) and \( S_2 \) have data dependence if and only if (iff) \( S_1 \) and \( S_2 \) have either read/write or write/write conflicts.

**Loop carried dependence.** Statements \( S_1 \) and \( S_2 \) have loop carried dependence iff there exist \( I = (i_1, i_2, \ldots, i_k) \) and \( \tau = (i_1, i_2, \ldots, i_k) \), \( I \neq \tau \), \( S_1(I) \) and \( S_2(\tau) \) have either read/write or write/write conflicts. Further, \( S_1 \) and \( S_2 \) have level \( j \) loop carried dependence iff \( i_1 = i_1, i_2 = i_2, \ldots, i_{j-1} = i_{j-1}, i_j \neq i_j \).

**EXAMPLE 2.** In the following FOR loop,

\[
\begin{align*}
\text{FOR } &i:=1 \text{ TO } 100 \text{ DO} \\
&\text{FOR } j:=1 \text{ TO } 100 \text{ DO} \\
&\quad S: x[i, j] := \text{comp}(x[i, j-1]); \\
&\quad \text{END FOR;} \\
&\text{END FOR;}
\end{align*}
\]

\( S((i, j)) \) reads \( x[i, j-1] \) and \( S((i, j-1)) \) writes \( x[i, j-1] \). Since iteration vectors \((i, j) \neq (i, j-1)\), we conclude that \( S \) has (level 2) loop carried dependence on itself.

There are various techniques ([WOLFE-82] and [ALLEN-83]) to break up loop carried dependencies (especially those false
dependencies: anti-dependence and output-dependence [KUCK-81]). In this paper, we assume that these techniques will be applied automatically whenever appropriate. Also, our context of loop parallelizability is constrained to the possibility of parallel execution of the loop iterations on multiple processors, and we assume that any partial parallelism of a loop has been properly handled by other techniques such as expression-tree height compression and loop splitting etc (see [PADUA-86]).

Based on these definitions and assumptions, we can state the following as a theorem.

**Theorem 4.1.** A FOR loop at level \( j \) is parallelizable if the loop body does not have any loop carried dependence at level \( j \).

**Proof.** Conflict will occur only if there exist iteration vectors \( I = (i_1, i_2, \ldots, i_k) \) and \( I' = (i'_1, i'_2, \ldots, i'_k) \), \( I \neq I' \), \( S_1(I) \) and \( S_2(I') \) are executed by two processes and the two processes simultaneously access a common data value. No loop carried dependence at level \( j \) says that for all such \( I \) and \( I' \), \( i_j = i'_j \). So, if we parallelize the \( j \)'th loop which lets a single process execute all statements with the same \( ij \) indexes sequentially, conflict can never occur.

Q.E.D.

**EXAMPLE 3.** Even though the loop in EXAMPLE 2 has level 2 loop carried dependence, it has no level 1 loop carried dependence
and can be safely parallelized as follows, since each level 2 loop is executed sequentially on a single processor:

```
PAR i:=1 TO 100 DO
  FOR j:=1 TO 100 DO
    S: x[i,j] := comp (x[i, j-1]);
    END FOR;
ENDPAR;
```

**Process Scheduling**

A parallelizable FOR loop of form [I] can be parallelized using the parallel construct of form [I'].

```
[I] FOR i:=1 TO N DO
  s(i);
ENDFOR;
```

```
[I'] PAR i:=1 TO N DO
  s(i);
ENDPAR;
```

The parallel construct PAR uses N processes to execute the N iterations in parallel. To reduce the process creation and context switch overheads, most systems (e.g., the Dynix system [THAKKAR-85]) use only P (or less than P) processes, where P is the number of processors in the system. For simplicity, we assume the program that contains the loop being discussed is the only job running on the P processors. To run the N iterations of loop [I] on the P processors, either static or dynamic scheduling strategies are used ([OSTERHAUG-86]).
In static scheduling, the N iterations are partitioned into \( \lceil N/P \rceil \) rounds and in each round except the last one, \( P \) adjacent iterations are solved by the \( P \) processors. In the last round \( N \mod P \) iterations are executed. A parallel code with static scheduling for loop [I] is given in [II]. Figure 4.4 (a) illustrates how the static scheduling works for the loop in EXAMPLE 1 (assume \( P = 3 \)).

[II] PAR g:=1 TO P DO
    FOR i:=g TO N STEP P DO
        s(i);
    ENDFOR;
ENDPAR;
In the parallel code above, each processor executes code that are \( P \) iterations apart. For example, in Figure 4.4 (a), iterations 1 and 4 are executed by the first processor.

In another static scheduling method, the first \( N \mod P \) processors each execute \( \lfloor N/P \rfloor \) iterations and the remaining \( (P - N \mod P) \) processors each execute \( \lfloor N/P \rfloor \) iterations. In this method, each processor executes adjacent iterations. [II'] shows parallel code for this schedule. Figure 4.4 (b) illustrates this static scheduling method. The code for this method is much more complex than the code for method 1 because each processor has to know how much work it needs to do.

\[
[II'] \quad \text{shares} := (N+P-1) \div P; \\
S := \text{shares}; \\
B := (N*K + 1) \mod (P+1); \\
\text{PAR } g:=1 \text{ TO } P \text{ DO} \\
\quad \text{IF } g > B \text{ THEN} \\
\quad \quad S := \text{shares} - 1; \\
\quad \text{ENDIF;} \\
\quad \text{FOR } j:=1 \text{ TO } S \text{ DO} \\
\quad \quad i := (g-1) * S + j; \\
\quad \quad s(i); \\
\quad \text{ENDFOR;} \\
\text{ENDPAR;}
\]

Static scheduling is frequently used when the iterations take roughly the same time to execute. However, even when the iterations are of the same size, the multiple processors may be left
unbalanced, as shown in Figure 4.4.

In dynamic scheduling, the N iterations are managed in a common queue, and are taken by processors as they finish previous iterations, until all iterations finish. Parallel code with dynamic scheduling for loop [I] is given as follows.

```
PAR g:=1 TO P DO
  i := TopOfQueue();
  WHILE NOT QueueEmpty() DO
    s(i);
    i := TopOfQueue();
  ENDWHILE;
ENDPAR;
```

There is overhead associated with the management of the common queue in dynamic scheduling (see [FANG-87] and [TANG-85]). In this paper, we assume the iterations are the same size (take the same time to execute) and study Loop Spreading, an efficient method for scheduling the iterations on multiple processors optimally.
4.3. Loop Spreading and Problems

During the execution of [II], unbalanced processor loading will result if \( N \mod P \) is not zero (see Figure 4.4). The reason is that in the last round, all but \( N \mod P \) processors are idle. Among the \( N \) processors, \( N \mod P \) processors take \( \lceil N/P \rceil \times T(s) \) time to execute the loop and \( (P - N \mod P) \) processors take \( (\lceil N/P \rceil - 1) \times T(s) \) time. If \( N \mod P \neq 0 \), the time difference between the heavily loaded processors and the lightly loaded processors is \( T(s) \).

Notice that when we break statements \( s(i) \) in [I] into \( K \) pieces of equal size, as in [III], and the \( K \) pieces are independent, we have \( N \times K \) iterations to schedule on \( P \) processors, as in [IV].

\[
\text{[III]} \quad \text{FOR } i:=1 \text{ TO } N \text{ DO} \\
\quad s1(i); \\
\quad s2(i); \\
\quad ....... \\
\quad sk(i); \\
\quad \text{ENDFOR;}
\]

\[
\text{[IV]} \quad \text{PAR } g := 1 \text{ TO } P \text{ DO} \\
\quad \text{FOR } i=g \text{ TO } N \times K \text{ STEP } P \text{ DO} \\
\quad \quad \text{IF } i <= N \text{ THEN} \\
\quad \quad \quad s1(i); \\
\quad \quad \text{ELSE IF } i <= 2 \times N \text{ THEN} \\
\quad \quad \quad s2(i-N); \\
\quad \quad \quad ....... \\
\quad \quad \text{ELSE IF } i <= (K-1) \times N \text{ THEN} \\
\quad \quad \quad sk-1(i-(K-2)\times N); \\
\quad \quad \text{ELSE} \\
\quad \quad \quad sk(i-(K-1)\times N); \\
\quad \quad \text{END IF;} \\
\quad \quad \text{END FOR;} \\
\quad \text{ENDPAR;}
\]

If each substatement \( s_j(i) \) takes execution time of \( T(sk) = T(s)/K \), then the time difference between the heavily loaded
processors and the lightly loaded processors will be at most $T(sk)$. This gives a much better load balance, especially when $K$ is large. For example, if we can break statements $s(i)$ into $\text{opt}K$ pieces, where

$$\text{opt}K = \min(k \mid k*N \mod P = 0),$$

then we can perfectly balance the loop on $P$ processors.

The strategy of scheduling $N*K$ subtasks instead of $N$ tasks is called Loop Spreading. The time saved through loop spreading is

$$[N/P]*K*T(sk) - [N*K/P]*T(sk), \text{ or}$$

$$K*T(sk) - [(N \mod P) * K / P] *T(sk).$$

Assume the probability that $N \mod P$ takes the value of 0, 1, ..., $P-1$ is uniformly $1/P$, then the expected value of $[(N \mod P) * K / P]$ is

$$E([(N \mod P) * K / P]) = \frac{1}{P} \sum_{i=1}^{P} \left[ ((i-1)*K)/P \right] \approx K/2.$$ 

So the average time saved through the loop spreading is $K*T(sk)/2$.

While loop [IV] can balance the $N$ iterations evenly on $P$ processors, we introduced $K$ IF checks in each of the $N*K$ iterations. Assuming on the average $K/2$ IF checks are executed in each of the
N*K iterations, loop [IV] introduces additional overhead of N*K*K*T(\text{IF})/2P in each processor. Compared to the time saved through loop spreading, the code in [IV] can speedup the execution of the loop only if

\[ K*T(s_k)/2 > N*K*K*T(\text{IF})/2P. \]

This condition is not easy to check because the loop bound N is usually unknown. Secondly, this condition is hard to satisfy as N is usually quite big compared to P. Third, [IV] will run slower than [II] when N mod P is zero. Finally, when s_1(i), .., s_k(i) are data dependent, the method above does not work.

We want the loop spreading method to have the following properties:

1) Applicability can be checked without knowing N.
2) The method can be applied to most loops.
3) When N mod P = 0, the spread loop should not run slower than the nonspread loop.
4) Can be applied when s_j(1), .., s_j(N) are data dependent.
4.4. Loop Spreading When Substatements are Independent

Assume statements $s_1(i)$, $s_2(i)$, ..., $s_k(i)$ in [III] are independent and assume $M = N \mod P$. Then the first $N - M$ iterations can be evenly distributed on $P$ processors, and we only need to spread the remaining $M$ iterations. This approach is illustrated in Figure 4.5, and is based on the parallel code in [V].

![Figure 4.5. Loop Spreading For Independent Substatements.](image-url)
PAR g := 1 TO P DO
  M := N mod P;
  U := N - M;
  FOR i:=g TO U STEP P DO
    sl(i); s2(i), ..., sk(i);
  ENDFOR;
  FOR i:=g TO M*K STEP P DO
    j := (i-1) DIV M + 1;
    ii := U + (i-1) MOD M + 1;
    CASE j OF
      1: sl(ii);
      2: s2(ii);
      ....
      k: sk(ii);
    END CASE;
  END FOR;
ENDPAR;

The CASE switch is usually implemented using a jump table so we can ignore the overhead in the CASE statement. The overhead due to spreading in [V] is $K*M*T(DIV, \ast)/P$. The code in [V] can speedup the execution of loop [II] when $K*M*T(DIV, \ast)/P$ is less than $K*T(sk)/2$, or (assuming $M = P/2$)

$$T(DIV, \ast) < T(sk).$$

This condition can be easily checked, as $T(DIV, \ast)$ is a constant known to the compiler.

To further reduce the overhead of loop spreading, we can replace the division and multiplication operations in [V] by iterative additions and subtractions. This is a well-known compiler technique (strength reduction, [AHO-77]), and it is always possible that operations like:
j := (i-1) \div M + 1, or
i := (i-1) \mod M + 1,

performed in each iteration are replaced by one or a few additions or
subtractions in each iteration, such as in [VI].

[VI] PAR g := 1 TO P DO
    M := N \mod P;
    U := N - M;
    FOR i:=g TO U STEP P DO
        s1(i);s2(i); ... , sk(i);
    ENDFOR;
    a := g; j := 1;
    FOR ij:=g TO M*K STEP P DO
        WHILE a > M DO
            a := a - M; j := j + 1;
        ENDWHILE;
        i := a + U; a := a + P;
        CASE j OF
            1: s1(i);
            2: s2(i);
            ....
            k: sk(i);
        END CASE;
    ENDFOR;
ENDPAR;

In [VI], for each ij in the second FOR loop, there are two
additions for updating a and i, and the WHILE loop inside the FOR
loop iterates at most P/M times. Thus the overhead of the loop
spreading is (assuming M = P/2),

\[ 2*M*K*T(+)/P + 2*(M*K*P/M)*T(+)/P = 3K*T(+). \]
So, when $3K \cdot T(+) < K \cdot T(sk)/2$, or

$$6 \cdot T(+) < T(sk),$$

the code in [VI] can speedup the execution of loop [II]. This condition is quite easy to satisfy as $6T(+) \text{ is a very small overhead, usually smaller than a multiplication or a division.}$ So, with the scheme in [VI], loop spreading can be applied to independent parallel loops with the properties:

1) Applicability can be checked using $6 \cdot T(+) < T(sk)$, without knowing $N$.

2) The condition $6 \cdot T(+) < T(sk)$ can be easily satisfied by most loops.

3) When $N \mod P = 0$, the spread loop will run as fast as the nonspread loop.

4.5. Loop Spreading in Nested Loops and Matrix Multiplication Example

A special case, and probably the most useful case, of loop [III] is when the loop body is itself a loop, as in [VII].

[VII] FOR i:=1 TO N DO
      FOR j:=1 TO K DO
          s(j,i);
      ENDFOR;
ENDFOR;

In this case, each iteration of the outer loop is naturally broken into K substatements \( s(1,i), s(2,i), \ldots, s(K,i) \). If these substatements are independent, the scheme in [VI] can be used to spread the loop, as in [VIII]. [VIII] will run no slower than [II] even if \( N \mod P = 0 \). The condition for this restructuring to be beneficial is that \( T(s(j,i)) > 6T(+) \).

```
[VIII] PAR g := 1 TO P DO
    M := N mod P;
    U := N - M;
    FOR i:=1 TO U STEP P DO
        FOR j := 1 TO k DO
            sj(i);
        ENDFOR;
    ENDFOR;
    a := g; j := 1;
    FOR i:=g TO M*K STEP P DO
        WHILE a > M DO
            a := a - M; j := j + 1;
        ENDDO;
        ii := a + U; a := a + P;
        s(j, ii);
    ENDFOR;
ENDPAR;
```

As an example, we apply this technique to matrix multiplication. The algorithm for calculating the product of two matrices \( X \) and \( Y \) of size \( N*K \) is as follows.

```
FOR i:=1 TO N DO
    FOR j:=1 TO K DO
        Z[i,j] := innerProd(X[i], Y[j]);
    ENDFOR;
ENDFOR;
```
It is not difficult to check that this loop has no level 1 (and level 2) loop carried dependence. The parallelization of the loop without loop spreading is the following:

```
PAR g := 1 TO P DO
    FOR i:=g TO N STEP P DO
        FOR j:=1 TO K DO
            Z[i,j] := innerProd(X[i], Y[j]);
        ENDFOR;
    ENDFOR;
ENDPAR;
```

The speedup of the parallelized loop over the sequential loop was shown in Figure 4.2. From Figure 4.2., we see that the speedup drops significantly when N increases q*P to q*P + 1, for q = 1, ..., ⌊N/P⌋. In order to remove the performance drop when N is not a multiple of P, we apply the scheme in [VIII] to the above parallelized matrix multiplication algorithm as in [IX].

```
[IX] PAR g := 1 TO P DO
    M := N mod P; U := N - M;
    FOR i:=g TO U STEP P DO
        FOR j:=1 TO K DO
            Z[i,j] := innerProd(X[i], Y[j]);
        ENDFOR;
    ENDFOR;
    a := g; j := 1;
    FOR i:=g TO M*K STEP P DO
        WHILE a > M DO
            a := a - M; j := j + 1;
        ENDFOR;
        ii := a + U; a := a + P;
        Z[ii,j] := innerProd(X[ii], Y[j]);
    ENDFOR;
ENDPAR;
```
We measured the speedup of the parallel algorithm with and without loop spreading over the sequential algorithm. In Figure 4.6, the algorithm with loop spreading shows stable speedup as the input size changes, regardless of whether or not N is a multiple of P. Furthermore, for all N the spread loop shows a speedup of no less than the nonspread loop.

![Figure 4.6. Performance of Parallel Matrix Multiplication Algorithm With Loop Spreading.](image)

### 4.6. Loop Spreading When Substatements are Dependent

Now we consider the case when the k substatements s1(i), s2(i), ..., sk(i) in [III] are dependent. We first assume that s1(i) ... sk(i) are totally dependent, meaning that sj+1(i) depends on sj(i) for j =1, ... k-1. Later we will extend the result to partially dependent cases.
When $s_1(i)$, ..., $s_k(i)$ are dependent, the spreading method discussed in Section 4 needs modification, or else inconsistent results may be produced by the spread loop. For example, if we spread the loop in Example 1, when $s_3(i)$ depends on the result of $s_2(i)$ and $s_2(i)$ depends on $s_1(i)$, using scheme [VI], we get the result shown in Figure 4.7 (a).

![Diagram](image)

Figure 4.7. Inconsistent Loop Spreading With Dependent Substatements.

Depending on the relative speed of the processors, $s_1(4)$ can finish either before or after $s_2(4)$ starts. If $s_1(4)$ finishes after $s_2(4)$ starts, $s_2(4)$ will not be able to use the result of $s_1(4)$, and the result will be incorrect. On the other hand, if $s_2(4)$ waits until $s_1(4)$ finishes and $s_3(4)$ waits until $s_2(4)$ finishes, the spreading will not be able to improve execution time (Figure 4.7 (b)).
Without a mechanism to enforce the ordering that \(s_1(4)\) finishes before \(s_2(4)\) starts, the only way to guarantee a correct result is to let \(s_1(4)\) and \(s_2(4)\) run on a single processor. In general, if \(s_j(i), j = 1, \ldots, k,\) are totally data dependent, we can guarantee correct results without using explicit synchronization only if \(s_j(i), j = 1, \ldots, k\) run on a single processor.

However, if all \(s_j(i), j = 1, \ldots, k\) are run on a single processor, we will have no way to improve the execution time using loop spreading over that the nonspread loop, no matter how we arrange the statements on the multiple processors. To see this, notice that there must be at least one processor to execute \([N/P]\) of \(s_j(i), i = 1, \ldots, N\). If all \(s_j(i), j = 1, \ldots, k\), run on a single processor, this processor will need \(K \cdot [N/P] \cdot T(s_k)\) time to execute the loop. The nonspread loop takes \(K \cdot [N/P] \cdot T(s_k)\) time to execute the loop as well.

**Synchronization**

Sync/wait primitives (Chapter 5, [MIDKILL-86], and [WOLFE-87]) can be used to enforce sequential execution when dependent statements are executed on multiple processors. \(SYNC(i, j)\) is used to indicate that statement \(s_j(i)\) has finished execution, and \(WAIT(i, j)\) is used to force a statement that is dependent on statement \(s_j(i)\) to delay execution until statement \(s_j(i)\) finishes.

As long as proper SYNC/WAITs are used, loop spreading is safe
in producing a correct result. Our objective is to improve the total execution time as much as possible through loop spreading, and meanwhile, to keep the number of SYNC/WAITs needed to conserve data dependency at a minimum.

We cannot simply insert SYNC/WAITs in [VI] or [VIII] to achieve the objective of loop spreading while conserving data dependence. This will become clear after we prove Theorem 4.2.

**Theorem 4.2.** If $N<P$ and $s_1(i), s_2(i), ..., s_k(i)$ are totally data dependent, loop spreading can not improve performance over [II].

**Proof.** As $s_{j+1}(i)$ must run after $s_j(i)$ for $j=1, ..., k-1$, using SYNC/WAITs or not, at least $K \cdot T(s_k)$ time is needed to run the loop. $K \cdot T(s_k)$ is also the time needed by [II] when $N<P$.

Q.E.D.

In [VI], we only spread the last $N \mod P$ iterations of [III]. As $(N \mod P) < P$, it is immediate from Theorem 4.2 that [VI] can not improve performance over [II] by inserting SYNC/WAITs.

Although we can not spread only the last $(N \mod P)$ iterations of [III] when $s_1(i), s_2(i), ..., s_k(i)$ are dependent to achieve the same time saving as when $s_1(i), s_2(i), ..., s_k(i)$ are independent, we are able to spread the last $M$, for any $M \geq P$, iterations of [III] to achieve the desired time saving. For example, the loop in Example 1 can be
spread as in Figure 4.8.

In Figure 4.8., the spread loop achieved the same time saving as when \( s_1(i), s_2(i), \ldots, s_k(i) \) are independent because the dependent statements are executed in different rounds so that when a WAIT is issued the corresponding SYNC would have already been issued. In this case, SYNC/WAITs are used only for safe-guard purpose. This observation leads to the following theorems.

**Theorem 4.3.** If \( N \geq P \), we can spread the \( N \) iterations of loop [III] so that the total execution time of the spread loop is minimized.

**Proof.** Assume \( N = qP + R, 1 \leq R < P \), and \( m_j = (jN) \text{ MOD } P \). First, \( s_1(1), \ldots, s_1(N) \) are placed on the \( P \) processors so that \( s_1(i), 1 \leq i \)
\[ \leq P \text{ are placed on processor } i \text{ in the first round, } s_1(P+i), 1 \leq i \leq P \text{ are placed on processor } i \text{ in the second round, ..., and } s_1(q*P+i'), 1 \leq i' \leq m1 \text{ are placed on processor } i' \text{ in the } (q+1)'th \text{ round.} \]

In general, assume \( s_j(1),...,s_j(N) \) have been placed in \( \lceil (j*N)/P \rceil *P \) rounds and the last round has only used the first \( m_j \) processors. We assign \( s_{j+1}(1),...,s_{j+1}(P-m_j) \) to the remaining processors on that round, \( s_{j+1}(P-m_j+1), s_{j+1}(P-m_j+2), ..., s_{j+1}(P-m_j+P) \) to the next round, ..., and so on.

An example of the above spreading scheme for loop

\[
\text{FOR } i := 1 \text{ TO } 8 \text{ DO } s_1(i); s_2(i); s_3(i) \text{ ENDFOR;}
\]

is shown in Figure 4.9.

![Figure 4.9. Spreading Dependent Substatements.](image)

From the above arrangement, we can observe the following facts:

1) \( s_j(i) \) must be at least one round before \( s_{j+1}(i) \). This is
because $s_j(i)$ is executed in round

$$r_j = \lceil (j-1)N + i/P \rceil = \lceil (j-1)(N \mod P + q \cdot P) + i/P \rceil$$

$$= (j-1) \cdot q + \lceil (j-1)(N \mod P) + i/P \rceil$$

and $s_{j+1}(i)$ is executed in round

$$r_{j+1} = \lceil jN + i/P \rceil = \lceil j(N \mod P + P) + i/P \rceil$$

$$= j \cdot q + \lceil j(N \mod P) + i/P \rceil$$

and

$$r_{j+1} - r_j = q + \lceil j(N \mod P) + i/P \rceil - \lceil (j-1)(N \mod P) + i/P \rceil \geq q.$$

Since $N \geq P$, we know $q \geq 1$.

2) The data dependency relation can be preserved by issuing a
SYNC($i, j$) after every statement $s_j(i)$, and issuing a WAIT($i, j$) before every $s_{j+1}(i)$ for $j = 1$ to $k-1$. From 1) we know that $s_{j+1}(i)$ is in a later round than $s_j(i)$. This implies that when the WAIT($i, j$) from $s_{j+1}(i)$ is issued, the corresponding SYNC($i, j$) from $s_j(i)$ must has already been issued (remember that we assumed that $s_j(i)$ all have the same size).

Since the arrangement above leaves no unused processor in the first \(\lceil K \cdot N/P \rceil \) rounds and no actual waiting takes place, the total execution time of the loop is \(\lceil K \cdot N/P \rceil \cdot T(s_k)\).

Q.E.D.
It can be seen that in Figure 4.9, every statement $s_j(i)$ is in a different processor as $s_{j+1}(i)$. Thus we need explicit synchronization between each pair of $s_j(i)$ and $s_{j+1}(i)$, as in Figure 4.10. Theorem 4.4 shows that this observation is true in general.

**Figure 4.10. Synchronization Pattern for Spreading Dependent Substatements.**

**Theorem 4.4.** When using the arrangement of Theorem 4.3 to spread the $N*K$ iterations of loop [III] to achieve optimal time saving, we need at least $N*(K-1)$ SYNC/WAITs whenever $N \mod P \neq 0$.

**Proof.** With the arrangement of Theorem 4.3, it is not difficult to see that $s_j(i)$ is placed on processor

\[ p(s_j(i)) = ((j-1)*N + i - 1) \mod P + 1, \]

and $s_{j+1}(i)$ is on processor
\[ p(s_{j+1}(i)) = (j^*N + i - 1) \mod P + 1. \]

\[ p(s_j(i)) - p(s_{j+1}(i)) = N \mod P. \] Since \( N \mod P \neq 0 \), we know \( p(s_j(i)) - p(s_{j+1}(i)) \neq 0 \), meaning that \( s_j(i) \) and \( s_{j+1}(i) \) run on different processors. So, all statements \( s_j(i) \) need SYNCs and all statements \( s_{j+1}(i) \) need WAITs, for \( j = 1, \ldots, k-1, i=1,\ldots,N \), resulting in a total of \( N^*(K+1) \) SYNC/WAITs.

Q.E.D.

The above two theorems state that we can always achieve optimal time saving through loop spreading using \( N^*(K-1) \) SYNC/WAITs as long as \( N > P \). This suggests that, for any \( N > P \), we can always separate the \( N \) iterations into two parts: iteration 1 to iteration \( U \), and iteration \( U+1 \) to iteration \( N \), where the value of \( U \) is chosen such that \( U \) is a multiple of \( P \) and minimizes \((N-U) > P\). Once we find \( U \), we can evenly execute the first \( U \) iterations without using SYNC/WAITs and spread the remaining \( N-U \) iterations using our method. Obviously, \( U = N - (P + N \mod P) \). So, for any given \( N \), we only need to spread the last \( P + N \mod P \) iterations. In general, the value of \( U \) can be determined as follows:

\[
\begin{align*}
\text{IF } N &\leq P \text{ THEN } U := N \\
\text{ELSE } U &\equiv N - (P + N \mod P);
\end{align*}
\]

We code the spreading scheme in Theorem 4.3 as follows.
PAR g := 1 TO P DO
  IF N <= P THEN M := 0
  ELSE M := N mod P + P;
  U := N - M;
  FOR i := g TO U STEP P DO
    s1(i); s2(i), ..., sk(i);
  ENDFOR;
  FOR ij := g TO M*K STEP P DO
    i := (ij-1) MOD M + 1;
    j := (ij-1) DIV M + 1.
    CASE j OF
      1: s1(i + U); SYNC(1,i);
      2: WAIT(1, i); s2(i + U); SYNC(2,i);
      ....
      K: WAIT(k-1, i); sk(i + U)
    END CASE;
    i := i + P;
  ENDFOR;
ENDPAR;

We can replace the MOD and DIV operations in [X] by additions and subtractions as in [VI]. Compared to the overhead of the spreading in [VI], the code above needs one additional SYNC/WAIT in each of the M*(K-1) iterations. Thus, the overhead of loop spreading [X] is

\[(2*M*K*T(+)/P) + 2*(M*K*P/M)*T(+)/P\]
\[\quad + M*(K-1)*T(SYNC/WAIT)/P\]
\[= 2*K*T(+)*(M/P + 1) + M*(K-1)*T(SYNC/WAIT)/P.\]

Since P ≤ M < 2*P, we can assume M = 3P/2. The loop spreading is beneficial if this value is less than K*T(sk)/2, or,
\[2^*K*T(\text{+})*(3/2+1) + 3*(K-1)^*T(\text{SYNC/WAIT})/2 \leq K^*T(\text{sk})/2, \text{ or}
10^*T(\text{+}) + 3^*T(\text{SYNC/WAIT}) \leq T(\text{sk}).\] 

(6.1)

The condition above can be easily tested since \(T(\text{+})\) and \(T(\text{SYNC/WAIT})\) are constants known to the system. For a shared memory machine, \(T(\text{SYNC/WAIT})\) can be as cheap as an addition (see Chapter 5). However, this cost can be very high in a message passing based system. We will study the methods to reduce the number of \(\text{SYNC/WAITs}\) in the later sections.

If \(s_1(i), s_2(i), ..., s_k(i)\) in [III] are only partially dependent, then we can use the strategy in Theorem 4.3 with less than \((P + N \ mod \ P) * (K-1)\) \(\text{SYNC/WAITs}\) to conserve data dependency. First, only the substatements in the last \((P + N \ mod \ P)\) iterations need \(\text{SYNC/WAITs}\). We can use the following guidelines to issue \(\text{SYNCs}\) and \(\text{WAITs}\).

1) Transitive dependency can be ignored. This guarantees that the total number of \(\text{SYNC/WAITs}\) for partially dependent \(s_1(i), s_2(i), ..., s_k(i)\) will be no more than for totally ordered \(s_1(i), s_2(i), ..., s_k(i)\).

2) Only one \(\text{SYNC}\) is needed for multiple dependency from the same source statement.

3) If \(s_j(i)\) depends on \(s (0 \leq s \leq k-1)\) statements, it should issue \(s\) \(\text{WAITs}\) before starting execution. Since \(s\) can be zero, if \(s_j(i)\) does not dependent on any other \(s_j'(i)\), it does not need to issue a \(\text{WAIT}\)
operation.

4.7. Reducing Synchronization Overhead of Loop Spreading

In the above, we showed that we can always spread the last $P + N \text{ MOD } P$ iterations using $(P + N \text{ MOD } P) \times (K-1)$ SYNC/WAITs to achieve optimal time saving. Since SYNC/WAITs may become expensive in a message passing system, we should reduce the number of SYNC/WAITs when possible.

![Diagram of processors and spreading]

Figure 4.11. Row-major Spreading.

We first focus our attention on the loop spreading schemes that are similar to [X]. That is, we place statements $\rho(s_1(1)), ..., \rho(s_1(N)), ..., \rho(s_j(1)), ..., \rho(s_j(N)), ..., \rho(s_k(1)), ..., \rho(s_k(N))$ on the $P$ processors consecutively and wrap them into rounds without leaving free
processors in between, where \( p(s_j(1)), \ldots, p(s_j(N)) \) stands for a permutation of \( s_j(1), \ldots, s_j(N) \). We call this kind of spreading scheme row-major spreading (Figure 4.11). The other type of spreading scheme, called column-major spreading (Figure 4.12), in which \( p(s_1(1)), \ldots, p(s_k(1)), \ldots, p(s_1(i)), \ldots, p(s_k(i)), \ldots, p(s_1(N)), \ldots, p(s_k(N)) \) are placed on a single processor and continue to the next processor, is studied later.

![Diagram of processors and spreading](image)

Figure 4.12. Column-major Spreading.

The following theorem establishes the lower bound on the number of SYNC/WAITs for row-major spreading schemes.

**Theorem 4.5.** For any row-major spreading scheme, the tight lower bound on the number of SYNC/WAITs needed to spread loop [III] to \( P \) processors is
min(N MOD P, (P - N MOD P)) * (K-1).

Proof. For N MOD P = 0 the theorem holds trivially. Assume N MOD P > 0.

We say that statements sj(i) and sj'(i) are a pair of statements if j' = j + 1. Obviously, a SYNC/WAIT is needed between two statements if and only if they are a pair of statements and they are not placed in the same processor. Assume \(\lfloor N/P \rfloor = t\). Then for any row-major spreading scheme, each processor has either 2*t, 2*t + 1, or 2*t + 2 statements from sj's and sj+1's. There are two cases to consider:

1) N MOD P < P/2. Then each processor can have at most 2*t + 1 statements from sj's and sj+1's. Since only one sj and one sj+1 can be a pair, among the 2*t + 1 statements on a processor, there are at most 2*t statements in pairs. Thus the P processors can have at most t*P pairs and the remaining N - t*P = N MOD P pairs need SYNC/WAITs.

2) N MOD P > P/2. Then in the best case 2*(N MOD P) - P processors have 2*(t+1) statements from sj's and sj+1's and the others have only 2*t + 1 statements. Thus the P processors can have at most (t*P + 2*(N MOD P) - P) pairs and the remaining N - (t*P + 2*(N MOD P) - P) = P - N MOD P pairs need SYNC/WAITs.
From 1) and 2), we need \( \min(N \mod P, P - N \mod P) \) SYNC/WAITs between \( s_j(1), ..., s_j(N) \) and \( s_{j+1}(1), ..., s_{j+1}(N) \). Thus we need a total of \( \min(N \mod P, (P - N \mod P)) \times (K-1) \) SYNC/WAITs, for all \( j = 1, ..., K-1 \).

To show that this lower bound is tight, we construct a row-major spreading scheme to achieve the lower bound as follows.

Initially, \( p(s_1(1)), ..., p(s_1(N)) \) are spread consecutively in the \( P \) processors. Assume \( p(s_j(1)), ..., p(s_j(N)) \) have been placed consecutively. \( N \mod P \) of the \( P \) processor will have \( t + 1 \) \( s_j \)'s and \( P - (N \mod P) \) will have only \( t \) of the \( s_j \)'s. For \( p(s_{j+1}(1)), ..., p(s_{j+1}(N)) \), we can always pick out \( t \times P \) of them and place them in \( t \) rounds so that none of them needs SYNC/WAIT. For the remaining \( N \mod P \) statements, if \( N \mod P \leq P / 2 \), we proved the theorem. We need only to consider the case when \( N \mod P > P/2 \).

Since \( N \mod P > P/2 \), any of \( (P - N \mod P) \) statements of \( s_{j+1}(1), ..., s_{j+1}(N) \) can form a full round with the remaining \( N \mod P \) of \( s_j(1), ..., s_j(N) \). We can pick out \( (N \mod P) - (P - N \mod P) \) of the remaining \( (N \mod P) \) of \( s_{j+1}(1), ..., s_{j+1}(N) \) to place on the processors without SYNCs. So, we need at most \( (P - N \mod P) \) SYNCs.

QED.
EXAMPLE 4. Assume we have \( P = 5 \) processors, and we want to spread the following loop \( (N = 8, K = 3) \) to the \( P \) processors with minimal execution time. Figure 4.13 shows a row-major spreading arrangement that uses 4 \( (4 = \min(N \mod P, (P - N \mod P)) \times (K-1)) \) SYNC/WAITs.

\[
\text{FOR } i := 1 \text{ TO 8 DO} \\
\hspace{1cm} s1(i), s2(i), s2(i); \\
\text{END FOR;}
\]

![Figure 4.13. Example Row-major Spreading Using Minimum Number of SYNCS.]

The row-major spreading scheme in Theorem 4.3 uses \( (P + N \mod P) \times (K-1) \) SYNC/WAITs, which is far from the lower bound established in Theorem 4.5. We would like to categorize the class of row-major spreading so that we can distinguish a good spreading scheme from a bad one. Notice that the 8 statements \( s3(1),...,s3(8) \) in the above example are broken into five sequences \( (7 \ 8), (2 \ 3), (1), (5 \ 6), \) and \( (4) \) when placing them on the 5 processors. In the loop spreading arrangement of Theorem 4.3, the \( N \) substatements \( sj(1), sj(2), ..., sj(N) \) are always kept as one sequence: \( (sj(1), ..., sj(N)) \). Since
the number of sequences that the statements $s_j(1),...,s_j(N)$ are broken into stands for the number of conditions the spread loop must check in each iteration in order for a processor to pick up the right statement to execute, we should try to keep that number as small as possible. On the other hand, using more sequences might reduce the number of SYNC/WAITs. So, we want a spreading scheme that uses less SYNC/WAITs and at the same time separates $s_j(1), s_j(2), ..., s_j(N)$ to fewer subsequences. Hopefully this is better than a spreading scheme that uses more SYNC/WAITs and/or separates $s_j(1), s_j(2), ..., s_j(N)$ into more subsequences.

We qualify the concept of "Number of Spreading Sequences" as follows.

DEFINITIONS. The number of Spreading Sequences is the maximum number of out-of-order elements which occur when placing $s_j(1), ..., s_j(N)$ onto processors in a row-major spreading scheme. If we use $i$ sequences in a spreading scheme, we call the spreading an $i$-sequence spreading scheme.

**Theorem 4.6.** Assume $N \geq P$. Using only one sequence to spread the $N*K$ statements of loop [III] to achieve optimal time saving requires at least $(P + N \mod P) * (K-1)$ SYNC/WAITs.

**Proof.** This is the direct consequence of Theorems 3 and 4. Q.E.D.
To simplify the discussion that follows, we assume $N < 2^*P$, since for any $N \geq 2^*P$ we can always spread the first $N - (P + N \mod P)$ iterations without using SYNC/WAITs.

**Theorem 4.7.** If $N \geq P$, we can use a 2-sequence spreading scheme to spread the $N$ iterations of loop [III] using only $(N \mod P)*(K-1)$ SYNC/WAITs to achieve the optimal time saving.

**Proof.** $N \geq P$ implies $N = P + N \mod P$. Assume $m_j = (j*N) \mod P$ and $r_j = (j*(N \mod P)) \mod N$. The $N*K$ statements can be spread as follows.

First, $s_1(1), ..., s_1(N)$ are spread on the $P$ processors so that $s_1(i), 1 \leq i \leq P$, are placed on processor $i$ in the first round, and $s_1(P+i')$, $1 \leq i' \leq r_1$ are placed on processor $i'$ in the second round. Then, $s_2(r_1+1), ..., s_2(P)$, $s_2(P+1)$, $s_2(P+2)$, ..., $s_2(N)$, $s_2(1)$, $s_2(2)$, ..., $s_2(r_1)$ are assigned to the remaining processors in the second round and extended to later rounds. Note that, no SYNC/WAIT is needed to force data dependency between $s_1(i)$ and $s_2(i)$, for $i = r_1+1$, ..., $N$, because $s_2(i)$ is in the same processor as $s_1(i)$, $i=r_1+1$, ..., $N$. Only $s_2(i')$, $i'=1,...,r_1$ may need to WAIT. In general, $s_j(1)$, ..., $s_j(N)$ are placed as two sequences $s_j(r_j+1), ..., s_j(N)$, $s_j(1)$, ..., $s_1(r_j-1)$. After that, we can assign $s_{j+1}(r_j+1), ..., s_{j+1}(N), s_{j+1}(1), ..., s_{j+1}(r_j)$ to the remaining processors on that round and extend to the following rounds. This arrangement is shown below.
From this arrangement, we establish the following facts:

1) There must be at most \((K-1)*(N \mod P)\) SYNC/WAITs. To show this we need only show that for any two statements \(s_j(i)\) and \(s_{j+1}(i)\), they must be on the same processor for all \(i = 1, ..., N\), except for \((N \mod P)\) of them. From the above arrangement we can see that \(s_j(1), ..., s_j(N)\) are in the two sequences \((s_j(r_j-i+1) ..., s_j(N))\) and \((s_j(1) ..., s_j(ri))\), and \(s_{j+1}(1), ..., s_{j+1}(N)\) are in the two sequences \((s_{j+1}(r_j+1) ..., s_{j+1}(N))\) and \((s_{j+1}(1) ..., s_{j+1}(r_j))\). It is not difficult to see that the statements are placed on processors according to the following criteria:

<table>
<thead>
<tr>
<th>statement</th>
<th>processor</th>
<th>ranges</th>
</tr>
</thead>
<tbody>
<tr>
<td>(s_j(i))</td>
<td>((m_j-1+i-r_j-1) \mod P + 1)</td>
<td>(i = r_j-1 + 1, ..., N;)</td>
</tr>
<tr>
<td>(s_j(i))</td>
<td>((m_j-1+N+i-r_j-1) \mod P + 1)</td>
<td>(i = 1, ..., r_j-1;)</td>
</tr>
<tr>
<td>(s_{j+1}(i))</td>
<td>((m_j+i-r_j-1) \mod P + 1)</td>
<td>(i = r_j + 1, ..., N;)</td>
</tr>
<tr>
<td>(s_{j+1}(i))</td>
<td>((m_j+N+i-r_j-1) \mod P + 1)</td>
<td>(i = 1, ..., r_j.)</td>
</tr>
</tbody>
</table>

To see whether or not we need SYNC/WAIT for statement \(s_j(i)\)
and $s_{j+1}(i)$, we need to determine whether or not $s_j(i)$ is on the same processor as $s_{j+1}(i)$. We consider two cases, $r_j \geq r_{j-1}$, and $r_j < r_{j-1}$.

Assume $r_j \geq r_{j-1}$. Then,

$$r_j - r_{j-1} = ((j^*(N \mod P)) \mod N - ((j-1)^*(N \mod P)) \mod N$$

$$= ((j^*(N \mod P) - (j-1)^*(N \mod P)) \mod N$$

$$= (N \mod P) \mod N$$

For $i \in (r_j + 1, \ldots, N)$, $s_{j+1}(i)$ is on processor $(m_j + i - r_{j-1}) \mod P + 1$. Since $r_j \geq r_{j-1}$, we also have $i \in (r_{j-1} + 1, \ldots, N)$ and the corresponding $s_j(i)$ is on processor $(m_{j-1} + i - r_{j-1} - 1) \mod P + 1$. The following calculation establishes that $(m_j + i - r_j) \mod P = (m_{j-1} + i - r_{j-1} - 1) \mod P$, which says that $s_{j+1}(i)$ and $s_j(i)$ are on the same processor.

$$= (m_j + i - r_j - m_{j-1} + i + r_{j-1}) \mod P$$

$$= (m_j - m_{j-1} - (r_j - r_{j-1})) \mod P$$

$$= ((j^*(N) \mod P - ((j-1)^*(N) \mod P) - (N \mod P)) \mod P$$

$$= (N \mod P - N \mod P) \mod P$$

$$= 0.$$

For $i \in (1, \ldots, r_j - (N \mod P))$, $s_{j+1}(i)$ is on processor $(m_j + N + i - r_{j-1}) \mod P + 1$. Since $r_j - r_{j-1} = N \mod P$, we also have $i \in (1, \ldots, r_{j-1})$, and $s_j(i)$ is on processor $(m_{j-1} + N + i - r_{j-1} - 1) \mod P + 1$. The following calculation establishes the equality of $(m_j + N + i - r_j) \mod P$ and $(m_{j-1} + N + i - r_{j-1})$, which shows that $s_{j+1}(i)$ and $s_j(i)$ are on the same processor.
\[(m_j + N - r_j + i) \mod P - (m_{j-1} + N - r_{j-1} + i) \mod P\]
\[= (m_j + N + i - r_j - m_{j-1} - N - i + r_{j-1}) \mod P\]
\[= (m_j - m_{j-1} - (r_j - r_{j-1})) \mod P\]
\[= 0.\]

Thus for both \(i \in (r_j + 1, ..., N)\) and \(i \in (1, ..., r_j - (N \mod P))\), \(s_{j+1}(i)\) and \(s_j(i)\) are on the same processor. We only need SYNC/WAIT for the remaining \(N \mod P\) \(s_{j+1}(i)\), \(i \in (r_j - (N \mod P)+1, ..., r_j)\). Note that, when \(i \in (r_j - (N \mod P)+1, ..., r_j)\), we also have \(i \in (r_{j-1} +1, ..., r_j)\). Thus the \(N \mod P\) SYNC/WAITs are used by the last \(N \mod P\) statements of \(s_{j+1}(i)\), \(i \in (r_j - (N \mod P)+1, ..., r_j)\), to wait for the SYNCs from the first \((N \mod P)\) statements of \(s_j(r_{j-1}+1) ... s_j(N)s_j(1)\)
\(s_j(2) \ldots s_j(r_{j-1})\). This fact is useful when implementing the spreading scheme.

Assume \(r_j < r_{j-1}\). Then,

\[r_{j-1} - r_j\]
\[= (((j-1)*(N \mod P)) \mod N - (j*(N \mod P))) \mod N\]
\[= (N - (N \mod P) \mod N - (N \mod P)) \mod N\]
\[= (N - N \mod P) \mod N\]
\[= N - N \mod P\]

For \(i \in (r_j +1, ..., r_{j-1})\), \(s_{j+1}(i)\) is on processor \((m_j + i - r_{j-1}) \mod P +1\). Since \(r_j < r_{j-1}\), the corresponding \(s_j(i)\), \(i \in (r_j +1, ..., r_{j-1})\), is on processor \((m_{j-1}+N + i-r_{j-1}-1) \mod P +1\). The following calculation establishes that \((m_j + i - r_j) \mod P = (m_{j-1} + N + i - r_{j-1})\), which shows that \(s_{j+1}(i)\) and \(s_j(i)\) are on the same processor.
(mj + i - rj) MOD P - (mj-1+N + i-rj-1) MOD P
= (mj + i - rj - mj-1 - N - i + rj-1) MOD P
= (mj - mj-1 -N + (rj-1 - rj)) MOD P
= ((j*N) MOD P - ((j-1)*N) MOD P - N + (N - (N MOD P))) MOD P
= (N MOD P - N MOD P) MOD P
= 0.

Note that, sj+1(i) and sj(i), i2 ∈ (rj-1 +1, ..., N), will not be on the same processor. Neither will sj+1(i) and sj(i), for i ∈ (1, ..., rj).
However, as rj-1 - rj = N - N MOD P, the number of i's in the range of (rj +1, ..., rj-1) is already N - N MOD P. Therefore, we only need SYNC/WAITs for the remaining N MOD P pairs of statements sj(i), i ∈ (rj-1 +1, ..., N) ⊆ (1, ..., rj), and sj+1(i), i ∈ (rj-1 +1, ..., N) ⊆ (1, ..., rj).
Again, the N MOD P SYNCS/WAITs are used by the last N MOD P statements of sj+1(i) to wait for the SYNCS from the first (N MOD P)
statements of sj(i), because sj(i), i ∈ (rj-1 +1, ..., N) ∪ (1, ..., rj), are the first N MOD P statements of sj(rj-1+1) ... sj(N)sj(1)sj(2) ... sj(rj-1), and
sj+1(i), i ∈ (rj-1 +1, ..., N) ∪ (1, ..., rj) are the last N MOD P statements of sj+1(rj+1) ... sj+1(N) sj+1(1) ... sj+1(rj).

2) If a SYNC/WAIT is needed between sj+1(i) and sj(i), then sj(i) must be at least 2 rounds before sj+1(i).

From the above discussion, we know that only the first N MOD P statements of sj(rj-1+1) ... sj(N) sj(1) ... sj(rj-1) need to issue SYNCS to the last N MOD P statements of sj+1(rj+1) ... sj+1(N) sj+1(1) ... sj+1(rj).
Thus, if a SYNC/WAIT is needed between \( s_{j+1}(i) \) and \( s_j(i) \), there must be exactly \( 2N - N \mod P \) statements between them. Since \((2N - N \mod P) = 2P + N \mod P\), \( s_j(i) \) must be at least 2 rounds before \( s_{j+1}(i) \).

From 2) we know that \( s_{j+1}(i) \) is scheduled two rounds later than \( s_j(i) \). This implies that when the WAIT from \( s_{j+1}(i) \) is issued, the corresponding SYNC from \( s_j(i) \) has already been issued (remember that we have assumed that \( s_j(i) \) all have the same size). Since the arrangement above leaves no unused processor in the first \( \lceil KN/P \rceil \) rounds and no waiting actually takes place, the total execution time of the loop is \( \lceil KN/P \rceil T(s_k) \).

Q.E.D.

Informally, the 2-sequence spreading scheme is that: after \( s_j(1),..., s_j(N) \) have been placed in two sequences, such as:

\[
\begin{align*}
&... s_j(t)... s_j(u-1)s_j(u) ... s_j(N)s_j(1) s_i(2) .... \\
&... ... s_j(t-1)
\end{align*}
\]

we place \( s_{j+1}(1) ... s_{j+1}(N) \) to the P processors in two sequences \( s_{j+1}(u) \)
\[
... s_{j+1}(N) s_{j+1}(1) ... s_{j+1}(u-1).
\]
EXAMPLE 5. Assume we have a loop

\[
\text{FOR } i:=1 \text{ TO } 7 \text{ DO } s1(i), s2(i), s3(i), s4(i) \text{ END FOR;}
\]

and we want to place it on 5 processors. In the 2-sequence spreading of Figure 4.14, only 6 instead of 21 SYNC/WAITs are used.

![Diagram of 2-sequence Spreading]

**Figure 4.14.** Example of 2-sequence Spreading.

It is also shown in the proof of Theorem 4.7 that 2-sequence spreading ensures that any SYNC is 2 rounds before the corresponding WAIT. Compare this with the result of Theorem 4.3, where any SYNC is only 1 round before the corresponding WAIT. This new spreading scheme allows more variation on the time difference between the sub-statements. Thus when a WAIT is issued the corresponding SYNC is more likely to have been issued.
The code using the 2-sequence spreading scheme is shown in [XI].

[XI] PAR g := 1 TO P DO
    R := N mod P;
    M := R + P;
    U := N - M;
    FOR i:=g TO U STEP P DO
        s1(i); s2(i), ..., sk(i);
    ENDFOR;
    FOR i:=g TO M*K STEP P DO
        j := (i-1) DIV M + 1;
        rj1 := ((j-1) * R) MOD M;
        b := (j-1)*M;
        a := b + P - rj1;
        CASE j OF
            1: .... ;
            .... ;
            j: IF (i <= a) THEN
                ii := i + rj1 + U - (j-1)*M;
            ELSE
                ii := i + U - a;
            END IF;
            IF (j > 1) AND ((i-b) > P) THEN WAIT(j-1, ii);
            sj(ii)
            IF (j < K) AND ((i-b) <= R) THEN SYNC(j, ii);
            ....
            K: .... ;
            .... ;
        END CASE;
    END FOR;
ENDPAR;

Compared to the overhead of the spreading in [X], this code needs two more additions, three more IF checks in each of M*K iterations. Each processor performs a total of R*(K-1)/P SYNC/WAITs. The overhead of loop spreading [XI] is
\[ 2K \cdot (M/P + 1) \cdot T(+) + 2M \cdot K \cdot T(+/P) + 3M \cdot K \cdot T(IF)/P + R \cdot (K-1) \cdot T(SYNC/WAIT)/P = 2K \cdot (2M/P + 1) \cdot T(+) + 3M \cdot K \cdot T(IF)/P + R \cdot (K-1) \cdot T(SYNC/WAIT)/P, \]

Assume \( R = P/2 \) and \( M = 3P/2 \). The additional overhead of loop spreading is

\[ 8K \cdot T(+) + 9K \cdot T(IF)/2 + (K-1) \cdot T(SYNC/WAIT)/2, \]

Thus, the 2-sequence loop spreading is beneficial if this value is less than \( K \cdot T(sk)/2 \), or

\[ 16T(+) + 9T(IF) + T(SYNC/WAIT) < T(sk). \quad (7.1) \]

Comparing (7.1) to (6.2), we see that 2-sequence spreading is better than 1-sequence spreading when \( 6T(+) + 9T(IF) < 2T(SYNC/WAIT) \).

Although the lower bound on the number of SYNC/WAITs for any row-major spreading is \( \min(N \mod P, (P - N \mod P)) \cdot (K-1) \) (see Theorem 4.5), no 2-sequence row-major loop spreading scheme can achieve this lower bound. In fact, only in one case when there is a 2-sequence loop spreading which can use one less than \( (N \mod P) \cdot (K - 1) \) SYNC/WAITs. One such example is shown in Figure 4.15, where we can use 14 instead 15 SYNC/WAITs the given loop. Note that in
Figure 4.15, $s_j(1), \ldots, s_j(8)$, $j=1, \ldots, 5$, occupy 8 full rounds of five processors and $s_5(1), \ldots, s_5(8)$ are placed as a single sequence. In general, for a loop of $N$ iterations of $K$ substatements to be spread on $P$ processors, we may use one less SYNC/WAITs only if there is a $j < K$ such that $j \times N \mod P = 0$ (and $s_j(1), \ldots, s_j(N)$ are placed in a single sequence). Except this, we can show that $(N \mod P) \times (K-1)$ is the minimum number of SYNC/WAITs we must use in any 2-sequence loop spreading scheme.

Figure 4.15. A 2-sequence Spreading That Uses Less Than $(N \mod P) \times (K-1)$ SYNC/WAITs.
Theorem 4.8. If $N \mod P > 0$, and there is no such a $j < K$ that $(j*N) \mod P = 0$, then $(N \mod P) * (K-1)$ is the minimum number of SYNC/WAITs in any 2-sequence loop spreading scheme that achieves optimal time saving.

Proof. Note that the lower bound is only for the 2-sequence spreading that achieves optimal time saving. A 2-sequence spreading that does not achieve optimal time saving can use less SYNC/WAITs, bounded only by the lower bound for general row-major spreading. For example, we can spread the following loop on 4 processors, as shown in Figure 4.16 with a 2-sequence spreading using only 1 SYNC/WAIT, instead of 3.

FOR $i:=1$ TO 7 DO s1(i); s2(i) ENDFOR;

![Figure 4.16. A Non-optimal 2-sequence Spreading.](image)
Let \( N = P + r \), where \( P \) is the number of processors and \( r = N \) \( \mod \) \( P > 0 \). Assume \( s_j \)'s are placed as two sequences \( s_j(a+1), \ldots, s_j(N) \) and \( s_j(1), \ldots, s_j(a) \), \( s_j+1 \)'s are placed as two sequences \( s_j+1(b+1), \ldots, s_j+1(N) \) and \( s_j+1(1), \ldots, s_j+1(b) \). Assume \( s_j+1(b+1) \) is placed on processor \( g+1 \). Assume the statement in \( s_j \)'s that is placed in the same processor as \( s_j+1(b+1) \) is \( s_j(c+1) \). There are four cases to consider, and we show that for each of the four cases, we need at least \( r \) \SYNC\,\,/\WAITs among the \( s_j+1 \)'s and \( s_j+1 \)'s.

Case 1: \( c = b \). In this case \( s_j+1(a+1), \ldots, s_j+1(c) \) must be placed in the processors starting at \( g+1 \) (see Figure 4.17). Therefore, we need at least \( r \) \SYNC\,\,/\WAITs since \( s_j(t) \) is \( r \) processors away from \( s_j+1(t) \), for all \( t = a+1, \ldots, c \).

![Figure 4.17. Illustration for Case \( c = b \).](image)

Case 2: \( c < b \). Consider the first statement (assume it is \( s_j(x) \) in Figure 4.18) in the round where \( s_j(b+1) \) is placed. Since \( s_j(c+1) \) is on processor \( g+1 \), \( s_j(x) \) is \( P-g \) statements later than \( s_j(c+1) \) in the sequence \( s_j(a+1), \ldots, s_j(N)s_j(1), \ldots, s_j(a) \). If \( x \geq b + 1 \) (Figure 4.18.a), then \( s_j+1(x) \) is no more than \( P-g-1 \) statements after \( s_j+1(b+1) \) according to \( c < b \). Since \( s_j+1(b+1) \) is only \( g \) processors apart from
si(x), we know that sj(x) and sj+1(x) are at most g+P-g-1 = P-1 processors apart. Taking into account the fact that sj(x) is the first statement in a round, we know that sj+1(x) will be in the same round as sj(x). Thus when c < b, x ≥ b + 1 is not allowed.

If x < b + 1, and b+1 ≤ a (Figure 4.18.b), then sj(b+1) is among sj(x), ..., sj(a) and thus sj+1(b+1) is in the same round as sj(b+1). This is not allowed either.

Figure 4.18. Illustration for Case c < b.
If \( x < b + 1 \), and \( b + 1 > a \) (Figure 4.18.c), then \( s_j(b+1) \) is \( d \) processors away from \( s_j(c+1) \), where \( 1 \leq d \leq r \). Since \( s_j(c+1) \) is in the same processor as \( s_{j+1}(b+1) \), \( s_j(b+1) \) and \( s_{j+1}(b+1) \) is also \( d \) processors away. Consequently, for each statement in \( s_j(b+1), ..., s_j(c)s_j(c+1), ..., s_j(a) \), its pair statement must be in the same relative position in \( s_{j+1}(b+1), ..., s_{j+1}(c)s_{j+1}(c+1), ..., s_{j+1}(a) \), and therefore they are also \( d \) processors away. So we need at least \( P \) SYNC/WAITs in this case (note that there are at least \( P \) statements in the sequence \( s_j(b+1), ..., s_j(c)s_j(c+1), ..., s_j(a) \)).

Case 3: \( c > b \geq a \). Assume \( c = b + y \) (\( y > 0 \)). If \( s_j(c+1) \) in the first sequence (see Figure 4.17.a), then \( c = a + r - 1 \) and \( y < r \) (Figure 4.19). Note that when \( c > a \), it is impossible for \( s_j(c+1) \) to be in the second sequence (see Figure 4.17.b), since when \( s_j(c+1) \) is in the second sequence, \( a \geq P \) and \( c \leq r \) must be true, which implies \( c < a \), a contradiction. When \( s_j(c+1) \) is in the first sequence, for \( t = c+1, ..., N \), \( s_{j+1}(t) \) has to be placed on processor

\[
p_{j+1}(t) = (g + y + t - c - 1) \mod P + 1,
\]

and \( s_j(t) \) has to be placed on processor

\[
p_j(t) = (g + t - c - 1) \mod P + 1.
\]

Since \( p_{j+1}(t) - p_j(t) = y \mod P = y \neq 0 \), \( s_j(t) \) and \( s_{j+1}(t) \) are in different
processors. Similarly, for \( t = 1, \ldots, a \), \( s_{j+1}(t) \) has to be placed on processor

\[
p_{j+1}(t) = (g + N - b + t - 1) \mod P + 1,
\]

and \( s_j(t) \) has to be placed on processor

\[
p_j(t) = (g + N - c + t - 1) \mod P + 1.
\]

Since \( p_{j+1}(t) - p_j(t) = (c - b) \mod P = y \neq 0 \), \( s_j(t) \) and \( s_{j+1}(t) \) are in different processors. So, the \( P \) pairs of statements \( s_j(t) \) and \( s_{j+1}(t) \), \( t = 1, \ldots, a, c+1, \ldots, N \), all are not in the same processor and thus need SYNC/WAITs between them.

Case 4: \( c > b \) and \( b < a \). By assumption \( s_j(a) \) must be in the same round as \( s_{j+1}(b) \), or else \( s_{j+1}(b+1) \) is the first statement in a round and \( (j*N) \mod P = 0 \). If \( s_j(1), \ldots, s_j(a) \) are all in the same round as \( s_{j+1}(b+1) \), \( s_{j+1}(b+1) \) then is in the same round as \( s_j(b+1) \) since \( b < a \).

This violates the optimal time saving requirement. If only some of
$s_j(1), \ldots, s_j(a)$ are in that round, say they are $s_j(z), \ldots, s_j(a)$, $s_j(z)$ must be the first statement in that round. If $z < b + 1$, $s_j(b+1)$ is among $s_j(z), \ldots, s_j(a)$ and $s_j+1(b+1)$ and $s_j(b+1)$ is in the same round and the optimal time saving requirement is violated. If $z \geq b+1$, we consider two possibilities: either $s_j(c+1)$ is in the first sequence of $s_j$'s, or $s_j(c+1)$ is in the second sequence of $s_j$'s.

![Diagram](image)

Figure 4.20. Illustration for Case $c > b$ & $b < a$.

When $s_j(c+1)$ is in the first sequence of $s_j$'s (see Figure 4.20.a), $a < P$ since $s_j(1)$ is placed after $s_j(c+1)$. $s_j(z)$ and $s_j(c)$ are $a-z+1$ processors apart and $s_j+1(b+1)$ and $s_j+1(z)$ are at most $z-1$ processors apart. This implies that $s_j(z)$ and $s_j+1(z)$ can be at most $a - z + z = a < P$ processors apart. So $s_j(z)$ and $s_j+1(z)$ will be in the same round. Again a violation to the optimal time saving requirement.
When $s_j(c+1)$ is in the second sequence of $s_j$'s (see Figure 4.20.b), $s_j(b+1)$ is $d$ processors away from $s_j(c+1)$, where $1 \leq d \leq r$.

Since $s_j(c+1)$ is in the same processor as $s_{j+1}(b+1)$, $s_j(b+1)$ and $s_{j+1}(b+1)$ is also $d$ processors away. Consequently, for each of statement in $s_j(b+1)$, $s_j(c)$, $s_j(c+1)$, ..., $s_j(a)$, its pair statement must be in the same position in $s_{j+1}(b+1)$, $s_{j+1}(c)$, $s_{j+1}(c+1)$, ..., $s_{j+1}(a)$, and therefore they are also $d$ processors away. So we need at least $P$ \texttt{SYNC/WAIT}s in this case (note that there are at least $P$ statements in the sequence $s_j(b+1)$, $s_j(c)$, $s_j(c+1)$, ..., $s_j(a)$).

Q.E.D.

We can devise a 3-sequence row-major spreading scheme similar to the 2-sequence spreading scheme that uses no more than $(N \mod P) \times (K-1)$ \texttt{SYNC/WAIT}s.

**Theorem 4.9.** If $N \geq P$, we can use three sequences to spread the $N$ iterations of loop [III] with optimal time saving using $(N \mod P) \times (K-1)$ \texttt{SYNC/WAIT}s.

**Proof.** Assume $m_i = (N \times i) \mod P$, $i=1$, ..., $K$. The $N \times K$ statements can be spread as follows.

First, $s_1(1)$, ..., $s_1(N)$ are spread on the $P$ processors so that $s_1(i)$, $1 \leq i \leq P$ are placed on processor $i$ in the first round, and $s_1(P+i')$, $1 < i' \leq m_1$ are placed on processor $i'$ in the second round.

The remaining $P-m_1$ processors in that round are assigned to
s2(m1+1), s2(m1+2), ..., s2(P). Then, if all of the remaining s2(i)'s can be placed in the third round, s2(1), s2(2), ..., s2(m1), s2(P+1), ..., s2(N) are assigned to the third round, otherwise, only s2(1), s2(2), ..., s2(m1), s2(P+1), ..., s2(2P-m1) are assigned to that round and s2(2P-m1+1), ..., s2(N) are assigned to the beginning m2 processors of the next round. Note that no WAIT is needed to force s2(i), i=1..P to be executed after s1(i), as s2(i) is in the same processor as s1(i), i=1..P. Only s2(P+i'), i'=1,...,N-P) may need to WAIT because they may not be in the same processor as s1(P+i'). In general, assume that s1(1),...,s1(N) have been placed in ⌊(j*N)/P⌋*P rounds and the last round has only used the beginning mj processors. We can assign s1+m1(mj+1),...,s1+m1(P) to the remaining processors on that round. If 2P - mj ≥ N, we place s1+m1(1), s1+m1(2), ..., s1+m1(mj), s1+m1(P+1), ..., s1+m1(N) in the next round, or if 2P-mj < N, we place s1+m1(1), s1+m1(2), ..., s1+m1(m1), s1+m1(P+1), ..., s1+m1(2P - mj) in the next round, and s1+m1(2P-mj+1), ..., s1+m1(N) in the beginning mj+1 processors of yet another round... This arrangement is shown below.

```
s1(1) s1(2) ... s1(P)
s1(P+1) ... s1(N)s2(mi+1)...
s2(1) s2(2) ... s2(mi)s2(P+1) ...
... s2(N)s3(m2+1) ...
s3(1)s3(2) ... s3(m2)s3(P+1) ...
... ...
sj(1) si(2) ... sj(mj-1-1) sj(P+1) ...
... sj(N)sj+1 mj+1) ...
... ...
```
From this arrangement, we can observe the following facts:

1) we need at most \((K-1)*(N \mod P)\) SYNC/WAITs. That is because \(s_j(1), s_j(2), ..., s_j(P)\) are always in processors 1, 2, ..., P and thus no SYNC/WAITs are needed for the statements. Only \(s_j(P+1)\) to \(s_j(N)\) need SYNC, for \(j = 1\) to \(K-1\), and \(s_j(P+1)\) to \(s_j(N)\) need WAITs for \(j = 2\) to \(K\).

2) \(s_j(i), \ i \in (P+1, ..., N)\), must be at least 1 round before \(s_{j+1}(i)\). This is because \(s_j(i)\) is executed in round

\[
    r_j = \lceil ((j-1)N + i)/P \rceil = \lceil ((j-1)(N \mod P + P) + i)/P \rceil
\]

\[= j - 1 + \lceil (j-1)(N \mod P) + i)/P \rceil\]

and \(s_{j+1}(i)\) is executed in round

\[
    r_{j+1} = \lceil (jN + i)/P \rceil = \lceil (j(N \mod P + P) + i)/P \rceil
\]

\[= j + \lceil (j(N \mod P) + i)/P \rceil\]

and

\[r_{j+1} - r_j = 1 + \lceil (j(N \mod P) + i)/P \rceil - \lceil (j-1)(N \mod P) + i)/P \rceil \geq 1.\]

3) The arranged loop above takes time \(\lceil K*N/P \rceil*T(s_k)\). From 2) we know that \(s_{j+1}(i)\) is in a later round than \(s_j(i)\). This implies that when the WAIT from \(s_{j+1}(i)\) is issued, the corresponding SYNC from
sj(i) has already been issued (remember that we have assumed that sj(i) all have the same size). Since the arrangement above leaves no unused processor in the first \( \lceil K*N/P \rceil \) rounds and no waiting actually takes place, the total execution time of the loop is \( \lceil K*N/P \rceil * T(s_k) \).

Q.E.D.

Compared to the 2-sequence spreading scheme, the 3-sequence spreading scheme needs one more IF check in each of M*K iterations. Thus the overhead of the 3-sequence spreading is

\[
2*K*(2*M/P + 1)*T(+) + 4*M*K*T(IF)/P + R*(K-1)*T(SYNC/WAIT)/P,
\]

and the 3-sequence spreading is beneficial if this value is less than \( K*T(s_k)/2 \), or

\[
16*T(+) + 12*T(IF) + T(SYNC/WAIT) < T(s_k).
\]  \hspace{1cm} (7.2)

Note that \((N \mod P)*(K-1)\) is not the lower bound for 3-sequence spreading. This can be observed from Figure 4.21. In Figure 4.21, \( N = 8, P = 5, \) and \( K = 2 \). The 3-sequence spreading places the 16 statements on 5 processors using only 2 SYNC/WAITs, instead of \( 3 = (N \mod P) \times (K-1) \) SYNC/WAITs.
Figure 4.21. Example of 3-sequence Spreading Using Less Than (N MOD P) * (K-1) SYNC/WAITs.

Also notice that, in general, no 3-sequence spreading scheme can achieve the minimum number of SYNC/WAITs for row-major spreading. To see this, consider the following loop:

FOR i := 1 TO 4 DO s1(i); ...; s8(i) ENDFOR;

For this loop, K=5, N = 4. If we were to place them on 5 processors using only three sequences, we have no way to use only 4 (min(2, 1)*4) SYNC/WAITs.

Even though the 3-sequence spreading in Theorem 4.9 does not improve the number of SYNC/WAITs over the 2-sequence spreading in Theorem 4.7, it is the basis of the next theorem, which shows that there is a 4-sequence spreading scheme that uses the minimum number of SYNC/WAITs.
Theorem 4.10. If $N \geq P$, we can use a 4-sequence spreading to spread the $N$ iterations of loop [III] with optimal time saving using only $\min((N \mod P)*(K-1), (P - N \mod P)*(K-1))$ SYNC/WAITs.

Proof. When $2*(N \mod P) \leq P$, the theorem is true from Theorem 4.9. So we assume $2*(N \mod P) > P$.

In the proof of Theorem 4.9, we showed that using 3-sequence spreading, statements $s_j(1)$, ..., $s_1(N)$ are in three sequences $(s_j(m_j+1), ..., s_j(P))$, $(s_1)$, ..., $(s(m_j-1))$, and $(s_j(P+1), ..., s_j(N))$, and only statements in the last sequence $s_j(P+1)$, ..., $s_{j+1}(N)$ need SYNC/WAITs.

We use induction to show that we can always separate the last sequence $(s_{j+1}(P+1), ..., s_{j+1}(N))$ into two sequences: $(s_{j+1}(c_j+P+1), ..., s_{j+1}(P+1), ..., s_{j+1}(P+c_j))$, where

$$c_j = (N \mod P) - (j*(P - N \mod P)) \mod (N \mod P)$$
$$= (N \mod P) - (j*P) \mod (N \mod P)$$

and only the first $(P - N \mod P)$ statements of $s_{j+1}(c_j+P+1)$, ..., $s_{j+1}(N)$, $s_{j+1}(P+1)$, ..., $s_{j+1}(P+c_j)$ need to wait for the SYNCs from the last $(P - N \mod P)$ statements of $s_j(c_j-1+P+1)$, ..., $s_j(N)$, $s_j(P+1)$, ..., $s_j(P+c_j-1)$.

For $j=1$. Using 3-sequence spreading, the statements $s_1(i)$ and $s_2(i)$, $i = 1$, ..., $N$ are placed on $P$ processors as follows.
where $t = N - (2\times(N \mod P) - P)$. Clearly, if we exchange $s_2(P+1)$, with a statement in $s_2(t+1)$, ..., $s_2(N)$, we can use fewer SYNC/WAITs, since doing so will bring $s_2(P+1)$ to the same processor as $s_1(P+1)$. In fact, all of the $N - t$ statements $s_2(P+1)$, ..., $s_2(P+N - t)$ can be brought to the same processors as $s_1(P+1)$, .... We can rearrange $s_2(P+1)$, ..., $s_2(N)$ as $s_2(P+N - t)$, as shown in the following.

$$s_2(2\times(N \mod P)+1), ..., s_2(N)$$

For the rearrangement, only $s_2(2\times(N \mod P)+1)$, ..., $s_2(N)$ need SYNCs from $s_1(2\times(N \mod P)+1)$, ..., $s_1(N)$, respectively. Since $2\times(N \mod P)+1 = P + (N \mod P) - (P - (N \mod P)) +1 = P + c_1 + 1$, we proved the case for $j = 1$.

Assume that we have separated $s_j(P+1)$, ..., $s_j(N)$ into two sequences: $s_j(c_{j-1}+P+1)$, ..., $s_j(N)$ and $s_j(P+1)$, ..., $s_j(P+c_{j-1})$, and only the first $(P - N \mod P)$ statements of $s_j(c_{j-1}+P+1)$, ..., $s_j(N)s_j(P+1)$, ..., $s_j(P+c_{j-1})$ need to wait for the SYNCs from the last $(P - N \mod P)$ statements of $s_{j-1}(c_{j-2}+P+1)$, ..., $s_{j-1}(N)$ and $s_{j-1}(P+1)$, ..., $s_{j-1}(P+c_{j-2})$. Without loss of generality, we can assume $s_j(c_{j-1}+P+1)$ is placed on processor 1. Then if we use the 3-sequence spreading to place $s_{j+1}(P+1)$, ..., $s_{j+1}(N)$, we end up with the placement:
\[ s_j(c_j-1+P+1), \ldots, s_j(N)s_j(P+1), \ldots, s_j(P+c_j-1)s_j+1(m_j-1+1), \ldots, s_j+1(P) \\
\]
\[ s_j+1(1), \ldots, s_j+1(m_j-1)s_j+1(P+1), \ldots, s_j+1(t) \\
\]
\[ s_j+1(t+1), \ldots, s_j+1(N). \]

By Theorem 4.9, we know that only \( s_{j+1}(P+1), \ldots, s_{j+1}(N) \) need SYNC/WAITs. However, we can further rearrange \( s_{j+1}(P+1), \ldots, s_{j+1}(N) \) to reduce the number of required SYNC/WAITs. We try to rearrange \( s_{j+1}(P+1), \ldots, s_{j+1}(N) \) as in the following,

\[ s_{j+1}(c_j+P+1), \ldots, s_{j+1}(c_j+P) \\
\]
\[ s_j+1(c_j-1+P+1), \ldots, s_j+1(N)s_j+1(P+1), \ldots, s_j+1(c_j+P). \]

1) There is no free processor unused in between the \( s_{j+1} \)'s. That is because

\[
c_j-1+P - ((c_j + P) + 1) + 1 = c_j - 1 - c_j \\
= (N \mod P) - (j*P) \mod (N \mod P) - (N \mod P) - ((j-1)*P) \mod (N \mod P) \\
= P \mod (N \mod P) \\
= P - (N \mod P).
\]

2) Only \( s_{j+1}(c_j+P+1), \ldots, s_{j+1}(c_j-1+P) \) need to issue WAITs. This is because \( s_{j+1}(c_j-1+P+1), \ldots, s_j+1(N)s_j+1(P+1), \ldots, s_j+1(c_j+P) \) are in the same processors as \( s_j(c_j-1+P+1), \ldots, s_j(N)s_j(P+1), \ldots, s_j(c_j+P) \) respectively.

3) The SYNCs that the \( (P - N \mod P) \) statements \( s_{j+1}(c_j+P+1), \ldots, s_{j+1}(c_j-1+P) \) need to wait for are from the last \( (P - N \mod P) \) statements of \( s_j(c_j-1+P+1), \ldots, s_j(N)s_j(P+1), \ldots, s_j(P+c_j-1) \). This is because \( c_j+P+1, \ldots, N,P+1, \ldots, P+cj \) is the result of \( P+1, \ldots, N \) circular right shift \( j*(P - N \mod P) \) locations, and the last \( P-N \mod P \) locations always correspond to the first \( P - N \mod P \) locations of the sequence after
circular shifting $P - N \mod P$ locations.

Q.E.D.

The code for the 4-sequence spreading scheme is shown in [XII].

[XII] PAR $g := 1$ TO $P$ DO
R := $N \mod P$;
M := R + P;
U := $N - M$;
FOR $i := g$ TO $U$ STEP $P$ DO
    s1(i);s2(i), ..., sk(i);
ENDFOR;
FOR $i := g$ TO $M*K$ STEP $P$ DO
    $j := (i-1) \div M + 1$;
    mj1 := ((j-1)*M) mod P;
    a := (j-1) * M + P - mj1;
    cj1 := R - ((j-1)*P) mod R;
    b := j * M - cj1;
    CASE $j$ OF
        1: ....
        j: IF $i <= a$ THEN
            sj(mj1 + i + U - (j-1)*M)
        ELSE IF $i <= (a + mj1)$
            sj(i + U - a)
        ELSE
            IF $i <= b$
                ii := i - (a + mj1) + U + P + cj1
            ELSE
                ii := i + U - b + P;
            IF ($j > 1$) AND ($1 \leq (i - (a + mj1)) \leq (P-R)$)
                THEN WAIT(j-1, ii);
            sj(ii);
            IF ($j < k$) AND ($2*R-P \leq (i - (a + mj1)) \leq R$)
                THEN SYNC(j,ii);
        ....
    k: ....
END CASE;
END FOR;
ENDPAR;

Compared to the overhead of the spreading in Theorem 4.9, this code takes one more addition and one more IF check in each of
M*K iterations, and a total of G*(K-1) SYNC/WAITs, where G = min((N MOD P), (P - N MOD P)). The overhead of loop spreading [XII] is

\[ 2*K*(3*M/P + 1)*T(+) + 5*M*K*T(IF)/P \]
\[ + G*(K-1)*T(SYNC/WAIT)/P. \]

If we assume the possibility that G = 0, 1, ..., P/2 are equal, then in the average G = P/4. The overhead of the 4-sequence spreading is

\[ 11*K*T(+) + 15*K*T(IF)/2 + (K-1)*T(SYNC/WAIT)/4, \]

Thus, the 4-sequence loop spreading is beneficial if this value is less than K*T(sk)/2, or

\[ 22*T(+) + 15*T(IF) + T(SYNC/WAIT)/2 < T(sk). \]  \hspace{1cm} (7.3)

Comparing (7.3) to (7.2), we see that 4-sequence spreading is better than 3-sequence spreading when 3*T(+) + 6*T(IF) < T(SYNC/WAIT)/2.
Column-major Spreading

Theorem 4.11. If \( N \geq P \), we can use a column-major spreading scheme to spread the \( N \) iterations of loop [III] with optimal time saving and use no more than \( (P - 1) \) SYNC/WAITs.

Proof. Assume \( r = (K*N) \mod P \). Then \( r \) processors have \( \lceil (N*K)/P \rceil \) statements and \( P - r \) processors have \( \lfloor (N*K)/P \rfloor \) statements.

Without loss of generality, we let the first \( r \) processors have \( r(N*K)/P \) statements and the remaining processors have \( \lfloor (N*K)/P \rfloor \) statements. We can place the \( N*K \) statements

\[
s_1(1), \ldots, s_1(N), \\
\ldots \\
s_j(1), \ldots, s_j(N), \\
\ldots \\
s_k(1), \ldots, s_k(N),
\]

in the \( P \) processors column by column and when the quota of the current processor is reached we continue to place the statements in the next processor. This placement is shown in Figure 4.22.

From this placement we see that we only need one SYNC/WAIT between the last statement of a processor \( t \) and the first statement of the next processor \( t+1 \), for \( t = 1, \ldots, P-1 \), if these two statements have the same index. For any other statements, we don't need SYNC/WAITs. This suggests that we need at most \( P - 1 \) SYNC/WAITs.
Furthermore, if these two statements have different subscripts, there is no need to place a SYNC/WAIT between the two processors. This can happen if \( (t \cdot \lfloor(N*K)/P \rfloor) \mod K = 0 \) for \( t = 1 \ldots ((N*K) \mod P) \), or \( t \cdot \lfloor(N*K)/P \rfloor \mod K = 0 \) for \( t = ((N*K) \mod P) + 1, \ldots, P - 1 \). Thus the total number of SYNC/WAITs we need is:

\[
P - 1 - \sum \{ t \mid (t \in (1 \ldots ((N*K) \mod P)) \land (t \cdot \lfloor(N*K)/P \rfloor \mod K = 0) \}
\]

\[
\lor (i \in (((N*K) \mod P) + 1, \ldots, P - 1) \land (t \cdot \lfloor(N*K)/P \rfloor \mod K = 0)).
\]

![Figure 4.22. Column-major Spreading with Minimum Number of SYNC/WAITs.](image)

However, we can not have optimal time saving with this placement since a SYNC always goes from the last statement of one processor to the first statement of the next processor. For example, if \( s_1(i), \ldots, s_j(i) \) are the last \( j \) statements of processor \( t \) and \( s_{j+1}(i), \ldots, s_k(i) \) are the first \( k-j \) statements of the next processor \( t+1 \), then
processor $t+1$ can not start execution until processor $t$ finishes executing all of the $s_1(i)$, ..., $s_j(i)$.

We modify the above placement as follows. If $s_1(i)$, ..., $s_j(i)$ are the last $j$ statements of processor $t$ and $s_{j+1}(i)$, ..., $s_k(i)$ are the first $k-j$ statements of the next processor $t+1$, then we let $s_{k-j+1}(i)$, ..., $s_k(i)$ be the last $j$ statements of processor $t$ and $s_1(i)$, ..., $s_{k-j}(i)$ be the first $k-j$ statements of processor $t+1$. This modification is shown in Figure 4.23.

![Figure 4.23. Column-major Spreading With Minimum Number of SYNC/WAITs and Optimal Time Saving.](image)

First, this modification will not affect the total number of SYNC/WAITs, because both the modified placement and original placement need exactly one SYNC/WAIT between processor $t$ and $t+1$.

Secondly, in the modified placement, $s_j(i)$ must be executed in
an earlier round than $s_{j_2}(i)$ for all $j_1 < j_2$, because $N \geq P$ and each processor has at least $K$ statements.

Q.E.D.

Code for the above spreading scheme is presented in [XIII], where we assume $2P > N > P$, and we have spread the first $U = N-(P + N \mod P)$ iterations evenly.

[XIII] shares := \(\frac{N*K + P - 1}{P}\);
busyOnes := \(\frac{N*K + 1}{P+1}\);
PAR g:=1 TO P DO
\[\begin{array}{l}
\text{IF } g > \text{busyOnes THEN} \\
\quad \text{quota} := \text{shares} - 1; \\
\quad \text{first} := \text{busyOnes} + (g-1)*\text{quota} + 1; \\
\text{ELSE} \\
\quad \text{quota} := \text{shares}; \\
\quad \text{first} := (g - 1)*\text{quota} + 1;
\end{array}\]
ENDIF;
i := (first - 1) DIV K + 1;
firstPart := i*K - first + 1;
lastPart := (quota - firstPart) MOD K;
lastStart := quota - lastPart + 1;
jK := firstPart;
BEGIN := 1;
FOR q :=1 TO quota DO
\[\begin{array}{l}
\text{IF } (q = \text{lastStart}) \text{ AND (lastStart < K) THEN} \\
\quad j := n - \text{lastStart} + 1; \\
\quad \text{WAIT}(j-1, i);
\end{array}\]
ENDIF
CASE j OF
1: $s1(i)$;
2: $s2(i)$;
.....
k: $sk(i)$;
END CASE;
IF (q = firstPart) AND (j < K) THEN \text{SYNC}(j, i);
j := j + 1;
IF j > jK THEN
\[\begin{array}{l}
\quad jK := K; j := 1; i := i + 1;
\end{array}\]
ENDIF
ENDFOR;
ENDPAR;
Since each processor has to execute statements $s_1(i), \ldots, s_x(i), s_1(i+1), \ldots, s_1(i+i'), s_{i+1}(i+1), \ldots, s_{k(i+i')}, s_y(i+i'+1), \ldots, s_{k(i+i')+1}$, it has to know how many statements it should execute and the values of $x$ and $y$. This requires an initial startup time of:

$$T_{\text{init}} = 4T(*) + 2T(/) + 2T(\text{MOD}) + 15T(+) .$$

Further, there is an overhead of three IF checks per statement executed and one SYNC/WAIT per processor. The last IF condition will be true for $\lceil(N*K)/K \rceil$ times, and whenever the condition is true three more additions are required. So, the total overhead per processor is (assume $N = 3*P/2$):

$$T_{\text{init}} + T(\text{SYNC/WAIT}) + 3*(N*K)*T(\text{IF})/P + 3*T(\text{+})*(N*K)/(P*K)$$

$$= T_{\text{init}} + T(\text{SYNC/WAIT}) + 9*K*T(\text{IF})/2 + 9*T(\text{+})/2 .$$

When this cost is less than $K*T(s_k)/2$, or,

$$(2*T_{\text{init}} + 2*T(\text{SYNC/WAIT}) + 9*T(\text{+}))/K + 9*T(\text{IF}) < T(s_k), \quad (7.4)$$

the spread loop will run faster than the unspread loop. Comparing (7.4) to (6.2), we can see that when

$$(2*T_{\text{init}} + 2*T(\text{SYNC/WAIT})+9*T(\text{+}))/K + 9*T(\text{IF})$$

$$< 7T(\text{+}) + 3T(\text{SYNC/WAIT})$$
this spreading is better than the spreading in Theorem 4.3. But, for small loops, $T_{\text{init}}$ may dominate the cost and this spreading may not be better than the spreading in Theorem 4.3.

In some sense, the row-major scheme is like static scheduling method 1 (see Figure 4.4 (a)), and the column-major scheme is similar to the static scheduling method 2 (see Figure 4.4 (b)). We know from Section 3 that method 2 is much more difficult to code than method 1, and it is not surprising that the spreading method above is also much more difficult to code than the row-major spreading schemes.
4.8. Experiments with Dependent Substatements on a Shared Memory Machine

We first modify the matrix multiplication algorithm to make the iterations of the second level FOR loop totally data dependent. This algorithm (call it MMM) can be specified as follows, where innerProd(i, j, X, Y) finds the inner product of i'th row of X and j'th column of Y, and Z[i,0] are zeros:

\[
\text{FOR } i:=1 \text{ TO } N \text{ DO}
\begin{align*}
\text{FOR } j:=1 \text{ TO } K \text{ DO} \\
Z[i,j] &:= \text{innerProd}(i, j, X, Y) + Z[i,j-1]; \\
\text{ENDFOR;} \\
\text{ENDFOR;}
\end{align*}
\]

It is not difficult to check that the loop has no level 1 dependence, and the iterations in level 2 FOR loop are totally dependent, since \(Z[i,j]\) uses the value of \(Z[i,j-1]\). The parallelization of the loop without loop spreading is:

\[
\text{PAR } g := 1 \text{ TO } P \text{ DO}
\begin{align*}
\text{FOR } i:=g \text{ TO } N \text{ STEP } P \text{ DO} \\
\text{FOR } j:=1 \text{ TO } K \text{ DO} \\
Z[i,j] &:= \text{innerProd}(i, j, X, Y) + Z[i,j-1]; \\
\text{ENDFOR;} \\
\text{ENDFOR;}
\end{align*}
\]

\[\text{ENDPAR;}\]

We want to experiment with the four spreading schemes: 1/2/4-sequence spreading and the column-major spreading. Four versions of the MMM algorithm described in the Sequent/Balance shared memory system Pascal can be found in Appendix D.
The performance of the spread and nonspread loops using 8 processors are plotted in Figure 4.24.

From Figure 4.24, it is clear that all of the spreading schemes remove the drop-off effect of the nonspread loop. This is because the T(sk) in the MMM algorithm is more than 20*T(*), and all of the spreading schemes are beneficial. Also from Figure 4.24, we can that all of the spreading schemes are nearly identical in performance (except that the 1-sequence spreading is a little worse). This is not surprising, because the overhead of the spreading schemes differ only by a small number of SYNC/WAITs and additions. On a shared memory machine, a SYNC/WAIT costs about the same as an addition (Chapter 5), thus all of the spreading schemes have similar overhead.
In Figure 4.25, we compared the column-major spreading with the 2-sequence row-major spreading. Again they show similar improvement over the nonspread loop.

In order to see the difference between the spreading schemes, we need to find a loop in which $T(sk)$ is small enough so that we can tell which spreading scheme is (or is not) beneficial. For this, we further modify the MMM algorithm as follows (call it MMM'), where function doSum(a, b) finds the sum of a and b:

```c
FOR i=1 TO N DO
    FOR j=1 TO K DO
        Z[i,j] := doSum(X[i, i], Y[j, j]) + Z[i,j-1];
    ENDFOR;
ENDFOR;
```
In MMM' algorithm, $T(sk)$ has been reduced from more than $20*T(*)$ to $T(proc\ call) + 2*T(+)$. 

Figure 4.26. Performance of 1/2-sequence Spreading on MMM' Algorithm.

Figure 4.26. shows the performance of 1/2-sequence spreading schemes on MMM' using eight processors. We can see that 1-sequence spreading is not beneficial for the MMM' algorithm because in most cases the spread loop runs slower than the nonspread loop. On the other hand, 2-sequence spreading shows outstanding performance since the spread code is almost always better than the nonspread code.

We next experimented with the decomposed simplex algorithm because the statements inside the parallel loops are totally
dependent. When applying the loop spreading techniques, we encountered a more complex situation that requires flexible application of the techniques. First, the loop in the algorithm to be spread is of the form:

```plaintext
PAR g:=1 TO N DO
    s1; ...; sk1;
    FOR i:=1 TO M DO
        t1(g,i); ...; tk2(g,i);
    END FOR;
    u1; ...; uk3;
END PAR;
```

For this loop, we first need to consider it as a loop that has \(k_1 + M \times k_2 + k_3\) substatements, and then apply the loop spreading techniques.

The second problem we encountered is that, when we spread \(s_1, \ldots, s_k\), which were originally run on a single processor, to multiple processors, the variables that cause the dependency among the statements must be made global to multiple processors. On a shared memory system, this poses no problem. However, on a message passing system, the spread loop may have to frequently pass large amounts of data from processor to processor, making the spreading technique hard to use.
Figure 4.27. Performance of Decomposed Simplex Algorithm With 2-sequence Row-major Spreading (a) and Column-major Spreading (b).

Figure 4.27. summarizes the performance of the algorithm after applying the 2-sequence row-major and column-major spreading methods to the decomposed simplex algorithm on the Sequent/Balance machine (Chapter 3). Eight processors are used in
the experiment. Clearly, the loop spreading technique significantly reduced the last round effect.

4.9. Related Work

Notice that loop spreading resembles loop collapsing of [Padua-86]. Loop collapsing was proposed to improve processor utilization when the number of iterations $N$ of a given loop is less than the number of processors and the second level iterations of the loop are independent. For example, loop [VII] is converted to the following using loop collapsing:

```
FOR $ij := 1$ TO $N \times K$ DO
  $j := (ij-1) \div N + 1$;
  $i := (ij-1) \mod N + 1$;
  $s(j,i)$;
END FOR;
```

and parallelized as:

```
PAR $g := 1$ TO $P$ DO
  FOR $ij := g$ TO $N \times K$ STEP $P$ DO
    $i := (ij-1) \mod N + 1$; $j := (ij-1) \div N + 1$;
    $s(j,i)$;
  END FOR;
END PAR;
```

Although when $N < P$, loop collapsing is the same as our loop spreading scheme [VI], in general, loop collapsing will perform much worse than our loop spreading scheme. First, in loop collapsing, the additional overhead is proportional to $N$, which makes it hard to
check the applicability of the scheme when $N$ is unknown. Secondly, when $N \mod P = 0$, the collapsed loop will run slower than the noncollapsed loop. Third, the additional overhead with loop collapsing is much higher than that for loop spreading (when $N > P$). Finally, loop collapsing can not be used on loops with dependent substatements. Even if we insert SYNC/WAITs for loop collapsing when substatements are dependent, in general, we have to use $N*(K-1)$ SYNC/WAITs. This is because loop collapsing keeps $s(j,1), \ldots, s(j,N)$ in only one sequence (see Theorem 4.4).

To experimentally show the relative performance of loop spreading over loop collapsing, we applied the following code for collapsing the loop of the matrix multiplication algorithm (which replaces DIV/MOD with repetitive addition and subtraction):

```plaintext
PAR g := 1 TO P DO
  i := g; j := 1;
  FOR ij := g TO N*K STEP P DO
    WHILE i > N DO
      i := i - N; j := j + 1;
    END WHILE;
    s(j,i);
    i := i + P;
  END FOR;
END PAR;
```

The performance of the collapsed matrix multiplication algorithm and the spread matrix multiplication algorithm is plotted in Figure 4.28. From this example, it is evident that loop spreading is superior to loop collapsing in avoiding performance drop-off when
the loop bound is not a multiple of the number of processors.

![Graph showing performance comparison between spread and collapsed matrix multiplication algorithms.](image)

**Figure 4.28. Performance of Collapsed and Spread Matrix Multiplication Algorithm.**

### 4.10. Conclusions

General processor load balancing is a very hard problem, even \textit{NP} complete if the parallel tasks are not well structured ([COFFMAN-76], [GAREY-79, p.239], [KASAHARA-84], [HU-74], [ADAM-74], [KAUFMAN-74], [KOHLER-75], [GONZALEZ-77], [OUSTERHOUT-80], [LO-87], [KRUATRA-88]). We propose loop spreading as a means to balance parallelized loops. Our experiment shows that loop spreading schemes are very efficient solutions to the restricted processor balance problem. Furthermore, loop spreading can be completely performed by an automatic tool when analysing the input.
program. Thus it is a technique which can be used by a parallelizing compiler.

For a loop whose body $s(i)$ can be partitioned into $K$ independent substatements, $s_1(i), \ldots, s_k(i)$, we can spread it evenly to multiple processors with improved performance as long as

$$T(s_k) \geq 6T(+)$$

When the $K$ substatements are data dependent, the spreading has to use synchronization primitives (SYNC/WAIT) to conserve data dependency.

Column-major spreading uses less SYNC/WAITs (only $P-1$) but introduces high startup overhead and complex code. Several row-major spreading schemes were shown to be as efficient as the column-major spreading on shared memory machines and easier to code. The spreading sequences, and the corresponding number of SYNC/WAITs of row-major spreading schemes are summarized as follows.

<table>
<thead>
<tr>
<th># sequences</th>
<th># SYNC/WAITs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$N*(K-1)$</td>
</tr>
<tr>
<td>2</td>
<td>$(N \mod P) *(K-1)$</td>
</tr>
<tr>
<td>3</td>
<td>$\leq (N \mod P) *(K-1)$</td>
</tr>
<tr>
<td>4</td>
<td>$\min((N \mod P), (P - (N \mod P))) *(K-1)$</td>
</tr>
</tbody>
</table>

The tight lower bound on the number of SYNC/WAITs for any row-major spreading scheme is $\min(N \mod P, P - N \mod P)$. This
lower bound is achieved by 4-sequence spreading.

In considering the trade-off between the cost of SYNC/WAITs and the computational overhead, we qualified the conditions when we can apply loop spreading with improved performance as follows:

1-sequence row-major spreading:
\[10 \cdot T(+) + 3 \cdot T(\text{SYNC/WAIT}) < T(\text{sk});\] (6.1)

2-sequence row-major spreading:
\[16 \cdot T(+) + 9 \cdot T(\text{IF}) + T(\text{SYNC/WAIT}) < T(\text{sk});\] (7.1)

3-sequence row-major spreading:
\[16 \cdot T(+) + 12 \cdot T(\text{IF}) + T(\text{SYNC/WAIT}) < T(\text{sk});\] (7.2)

4-sequence row-major spreading:
\[22 \cdot T(+) + 15 \cdot T(\text{IF}) + T(\text{SYNC/WAIT})/2 < T(\text{sk});\] (7.3)

Column-major spreading:
\[(2 \cdot T_{\text{init}} + 2 \cdot T(\text{SYNC/WAIT}) + 9 \cdot T(+))/K + 9 \cdot T(\text{IF}) < T(\text{sk}).\] (7.4)

Three facts make the 2-sequence row-major spreading attractive: 1) it is easy to code and the additional overhead is small, 2) in 50% of cases (when \(N \mod P \leq P - N \mod P\)) it uses the minimum number of SYNC/WAITs, 3) a SYNC and the corresponding WAIT are at least 2 rounds apart, which allows the technique to be applied to loops containing substatements that are not of the same size.
Chapter 5

Implementation of Synchronization Primitives for Loop Spreading

Abstract

Loop spreading is a technique to restructure parallel loops so as to balance parallel tasks on multiple processors. The implementation efficiency of the synchronization primitives (SYNC/WAIT) decides whether or not the loop spreading technique is practically useful. We present an implementation strategy in which a SYNC operation takes only a shared memory access (or sends only a message), and a WAIT operation takes only one conditional check (receives a message). Most importantly, the implementation of the primitives require only a working space of size $P$, where $P$ is the number of processors used in parallel computation.

5.1. Introduction

When the number of processors $P$ is less than the number of tasks $N$ in a parallel loop construct, the tasks are usually executed in $\lceil N/P \rceil$ rounds, with the last round executing only $N \mod P$ tasks. In the worst case when $N \mod P$ is one, all but one processor are idle in the last round, and the performance of the parallelized program degrades sharply. This performance drop becomes more and more significant as the number of processors increases.

Loop spreading (Chapter 4) is a technique to automatically restructure parallel loops so as to balance the parallel tasks on multiple processors. For example, without using loop spreading, the
following loop usually is allocated to 3 processors as in Figure 5.1.

\[
\text{PAR } i:=1 \text{ TO } 4 \text{ DO } s1(i); s2(i); s3(i); \text{ END PAR;}
\]

Figure 5.1. Example of Unbalanced Processor Load.

However, using loop spreading, the above loop can be allocated to 3 processors evenly as in Figure 5.2.

Figure 5.2. Load Balance Through Loop Spreading.
The diagonal arrows in Figure 5.2 represent the synchronization operations. Since s1(4), s2(4), and s3(4) are originally run on the same processor, synchronization operations (SYNC/WAIT [MIDKIFF-86] [WOLFE-87]) are required to guarantee correct results when spreading them to multiple processors. More specifically, the arrow from statements sj(i) to sj+1(i) represents a pair of SYNC(i, j) and WAIT(i, j) operations, where the SYNC(i, j) operation is issued from the processor which executes sj(i) after sj(i) finishes execution, and the WAIT(i, j) operation is issued at the other processor which executes sj+1(i) before sj+1(i) starts execution. The semantics of SYNC(i, j) indicate that statement sj(i) has finished execution, and those of WAIT(i, j) force statement sj+1(i) to delay execution until the corresponding SYNC(i, j) has been issued.

In Chapter 4, four row-major spreading schemes and one column-major spreading scheme are described. All of the spreading schemes spread a loop to multiple processors evenly, and each of the schemes uses a different number of SYNC/WAITs to spread the loop. The reader should study Chapter 4 for details of the Loop Spreading technique, we only briefly survey the methods, here. Assume loop PAR i:=1 TO N DO s1(i), ..., sk(i) ENDPAR is to be spread on P processors, and P ≤ N < 2*P. These spreading schemes can be summarized as follows.
Column-major spreading

Assume \( r = (K \cdot N) \mod P \). Then \( r \) processors have \( \left\lceil (N \cdot K) / P \right\rceil \) statements and \( P - r \) processors have \( \left\lfloor (N \cdot K) / P \right\rfloor \) statements. Let the first \( r \) processors have \( \left\lceil (N \cdot K) / P \right\rceil \) statements and the remaining processors have \( \left\lfloor (N \cdot K) / P \right\rfloor \) statements. We place the \( N \cdot K \) statements \( s_j(i), i=1,\ldots,N, j=1,\ldots,k \), in the \( P \) processors column by column. When only \( j_i \) statements of \( s_1(i), \ldots, s_k(i) \) can be placed on the current processor before the processor reaches its quota, we separate \( s_1(i), \ldots, s_k(i) \) into two sequences \( (s_k-j_i+1(i), \ldots, s_k(i)) \) and \( (s_1(i), \ldots, s_k-j_i(i)) \) and place the first sequence on that processor and the second sequence at the beginning of the next processor. This placement is shown in Figure 5.3.

![Figure 5.3. Synchronization Pattern of Column-major Spreading.](image)

For this spreading scheme, we need SYNC/WAITs only for the pairs of statements \( s_j(i) \) and \( s_{j+1}(i) \) that are on two different
processors, thus yielding a total of \((P - 1)\) SYNC/WAITs.

**1-sequence Row-major Spreading**

Assume \(N = P + R\) and \(m_j = (j*N) \text{ MOD } P\). First, \(s_1(1), \ldots, s_1(N)\) are placed on the \(P\) processors so that \(s_1(i), 1 \leq i \leq P\) are placed on processor \(i\) in the first round, \(s_1(P+i), 1 \leq i \leq R\) are placed on processor \(i\) in the second round. Assume that \(s_j(1), \ldots, s_j(N)\) have been placed in \([j*N/P]*P\) rounds and the last round has only used the first \(m_j\) processors. We can assign \(s_{j+1}(1), \ldots, s_{j+1}(P-m_j)\) to the remaining processors on that round, \(s_{j+1}(P-m_j+1), s_{j+1}(P-m_j+2), \ldots, s_{j+1}(P-m_j+P)\) to the next round, \(\ldots\), and so on. An example of 1-sequence spreading is shown in Figure 5.4.

PAR i:=1 TO 8 DO s1(i), s2(i), s3(i) ENDPAR;

![Diagram showing the synchronization pattern of 1-sequence spreading]

Figure 5.4. Synchronization Pattern of 1-sequence Spreading.
For this spreading scheme, we need one SYNC/WAIT for every pair of statements $s_j(i)$ and $s_{j+1}(i)$, thus yielding a total of $N^*(K-1)$ SYNC/WAITs.

2-sequence Row-major Spreading

Assume $m_j = (j*N) \mod P$ and $r_j = (j*(N \mod P)) \mod N$. First, $s_1(1), ..., s_1(N)$ are spread on the $P$ processors so that $s_1(i), 1 \leq i \leq P$ are placed on processor $i$ in the first round, and $s_1(P+i), 1 < i \leq r_1$ are placed on processor $i$ in the second round. Then, $s_2(r_1+1), ..., s_2(P)$, $s_2(P+1), s_2(P+2), ..., s_2(N)$, $s_2(1), s_2(2), ..., s_2(r_1)$ are assigned to the remaining processors in the second round and extended to later rounds. Note that, no SYNC/WAIT is needed to force data dependency between $s_1(i)$ and $s_2(i)$, for $i=r_1+1, ..., N$, because $s_2(i)$ is in the same processor as $s_1(i), i=r_1+1, ..., N$. Only $s_2(i')$, $i'=1, ..., r_1$ may need to WAIT. In general, $s_j(1), ..., s_j(N)$ are placed as two sequences $s_j(r_j-1+1), ..., s_j(N), s_j(1), ..., s_1(r_j-1)$, and after they are placed on the $P$ processors the last round uses only the first $m_j$ processors. After that, we can assign $s_{j+1}(r_j+1), ..., s_{j+1}(N)$, $s_{j+1}(1), ..., s_{j+1}(r_j)$ to the remaining processors on that round and extend to the following rounds. An example of 2-sequence spreading is shown in Figure 5.5.

For this spreading scheme, we need one SYNC/WAIT from every statement $s_j(i)$ in the first $N \mod P$ statements of $s_j(r_j-1+1), ..., s_j(N), s_j(1), ..., s_1(r_j-1)$ to their successor statements, which must be among the last $N \mod P$ statements of $s_{j+1}(r_j+1), ..., s_{j+1}(N), s_{j+1}(1), ...$.
si+j(rj), thus yielding a total of \((N \text{ MOD } P) \times (K-1)\) SYNC/WAITs.

\[
\text{PAR } i:=1 \text{ TO 7 DO } s1(i), \ldots, s5(i) \text{ ENDPAR;}
\]

Figure 5.5. Synchronization Pattern of 2-sequence Spreading.

**3-sequence Row-major Spreading**

Assume \(m_i = (N \times i) \text{ mod } P, i=1 \ldots K\). The \(N \times K\) statements can be spread as follows. First, \(s1(1), \ldots, s1(N)\) are spread on the \(P\) processors so that \(s1(i), 1 \leq i \leq P\) are placed on processor \(i\) in the first round, and \(s1(P+i), 1 < i \leq m_1\) are placed on processor \(i\) in the second round. The remaining \(P-m_1\) processors in that round are assigned to \(s2(m_1+1), s2(m_1+2), \ldots, s2(P)\). Then, if all of the remaining \(s2(i)\)'s can be placed in the third round, \(s2(1), s2(2), \ldots, s2(m_1), s2(P+1), \ldots, s2(N)\) are assigned to the third round, otherwise, only \(s2(1), s2(2), \ldots, s2(m_1), s2(P+1), \ldots, s2(2P-m_1+1)\) are assigned to that round and \(s2(2P-m_1+1), \ldots, s2(N)\) are assigned to the beginning \(m_2\) processors of the
next round. Note that no WAIT is needed to force $s2(i)$, $i=1..P$ to be executed after $s1(i)$, $i = 1, \ldots, p$ respectively, as $s2(i)$ is in the same processor as $s1(i)$, $i=1..P$. Only $s2(P+i')$, $i'=1,\ldots,(N-P)$ may need WAITs because they may not be in the same processor as $s1(P+i')$. In general, assume $s_j(1),\ldots,s_j(N)$ have been placed in $\lceil j*N/P \rceil$ rounds and the last round has only used the first $m_j$ processors. We can assign $s_{j+1}(m_j+1),\ldots,s_{j+1}(P)$ to the remaining processors on that round. If $2P - m_j \geq N$, we place $s_{j+1}(1), s_{j+1}(2), \ldots, s_{j+1}(m_j), s_{j+1}(P+1), \ldots, s_{j+1}(N)$ in the next round, or if $2P-m_j < N$, we place $s_{j+1}(1), s_{j+1}(2), \ldots, s_{j+1}(m_1), s_{j+1}(P+1), \ldots, s_{j+1}(2P-m_j)$ in the next round, and $s_{j+1}(2P-m_j+1), \ldots, s_{j+1}(N)$ in the first $m_j+1$ processors of yet another round. An example of 3-sequence spreading is shown in Figure 5.6.

Figure 5.6. Synchronization Pattern of 3-sequence Spreading.

For this spreading scheme, we need one SYNC/WAIT for every
pair of statements \( s_j(i) \) and \( s_{j+1}(i) \), for \( i > \lfloor N/P \rfloor \times P \), thus yielding a total of \((N \mod P) \times (K-1)\) SYNC/WAITs.

**4-sequence Row-major Spreading**

This is the same as 3-sequence spreading except that the last sequence \( s_j(P+1)s_j(P+2) \ldots s_j(N) \) of the 3-sequence spreading scheme is separated into two sequences \((s_{j+1}(c_j+P+1), \ldots, s_j(N))\) and \((s_{j+1}(P+1), \ldots s_{j+1}(c_j+P))\), where \( c_j = (N \mod P) - (j*P) \mod (N \mod P) \) if \( N \mod P > P - N \mod P \), otherwise \( c_j = 0 \). An example of 4-sequence spreading is shown in Figure 5.7.

```
PAR i:=1 TO 8 DO s1(i), ..., s5(i) ENDPAR;
```

![Figure 5.7. Synchronization Pattern of 4-sequence Spreading.](image)
For this spreading scheme, only the first \( \min(N \mod P, (P - N \mod P)) \) statements \( s_j+1(c_j+P+1), ..., s_j+1(c_{j-1}+P) \) of \( s_j+1(c_j+P+1), ..., s_j+1(P+c_j) \) need to wait for the SYNCs from the last \( \min(N \mod P, (P - N \mod P)) \) statements of \( s_j(c_{j-1}+P+1), ..., s_j(P+c_j-1) \), thus yielding a total of \( \min(N \mod P, (P - N \mod P)) \times (K-1) \) SYNC/WAITs.

In Section 2, we present a straightforward implementation for the SYNC/WAITs operations, and raise the problem we want to solve. Section 3 describes the SYNC/WAITs for column-major spreading scheme, and in Section 4, efficient implementation of SYNC/WAITs for row-major spreadings are derived.

5.2. Straightforward Implementation of SYNC/WAITs

In a shared memory system, the SYNC(i,j) and WAIT(i,j) operations in the spread programs can be implemented by setting and busy-waiting on the shared array element syncBuffer[i,j].

SYNC(i,j) sets syncBuffer[i,j] to a special number:

\[
syncBuffer[i,j] := \text{special number};
\]

and WAIT(i,j) busy-waits for syncBuffer[i,j] to become the special number, using the following code:

\[
\text{WHILE (syncBuffer[i,j] <> special number) DO;}
\]
Initially, we set the special number to 0. Before the L'th parallel loop is executed, we let the special number be L. In this way, no matter how many parallel loops are in a program, we need to initialize the array syncBuffer only once:

\[
\text{syncBuffer}[1..\text{MAXROW}, 1..\text{MAXCOLUMN}] := 0;
\]

In a message passing system that has no shared memory, SYNC and WAIT are implemented as sending and receiving messages. Therefore, the sending processor must know which processor to send and the receiving processor must know from which processor to ask for the signal. Assume each processor has a local copy of syncBuffer. Then, SYNC(i, j) and WAIT(i,j) operations can be implemented as follows:

\[
\text{SYNC}(i,j): \quad \text{determine receiving processor } g'; \\
\quad \text{send}(g', i, j);
\]

\[
\text{WAIT}(i,j): \quad \text{determine sending processor } g''; \\
\quad \text{WHILE } \text{syncBuffer}[i, j] <> \text{special number DO} \\
\quad \quad \text{receive}(g'', x, y); \\
\quad \quad \text{syncBuffer}[x, y] := \text{special number;} \\
\quad \text{ENDWHILE;}
\]

These implementations of SYNC/WAIT are fast, especially when we have carefully arranged the code so that when a WAIT is issued the corresponding SYNC most probably has been issued (this is true for all of the above spreading schemes). In this case, a SYNC
operation needs only a single shared memory access (sending a message) and the WAIT operation needs merely one conditional check (receiving a message).

However, the size of the array syncBuffer poses a storage problem and introduces additional system overhead. More importantly, if the size of the array can not be decided from the program source, loop spreading cannot be applied automatically, since a potential runtime error is introduced. For example, in the 1-sequence spreading scheme, implementation of syncBuffer requires a row size equal to \((P + N \mod P)\), and a column size equal to \((K-1)\). The array has to be kept very large to prevent runtime error when the value of \(K\) is unknown.

In the most general case, the above implementations are the only choices, because multiple SYNCs required by the statements in one processor may come from several different processors in nondeterministic order, requiring each processor to buffer the SYNCs and match up the buffered SYNCs with the corresponding WAITs. For example, in the spread loop in Figure 5.8, when processor p3 issues WAIT(1,2), it is not impossible that SYNC(1,1), SYNC(2,7), SYNC(3,4), and SYNC(3,1) all have been issued. Since all of these SYNCs are used by processor p3, they have to be buffered.

However, for the kind of spreading schemes summarized in Section 1, we can use less memory to implement the SYNC/WAITs.
PAR i:=1 TO 7 DO s1(i), ..., s4(i) ENDPAR;

Figure 5.8. Example of a Row-major Spreading In Which Multiple SYNCs Must Be Buffered.

5.3. Implementation of SYNC/WAIT for Column-major Spreading Scheme

We already know that column-major spreading needs at most one SYNC per processor. Further, we know that the processor $g$ which issues a SYNC(i, j) knows that only the previous processor

$$g' = (g + P - 2) \mod P + 1$$

may wait for the signal, and the processor $g$ which issues a WAIT(i, j) knows that only the next processor

$$g'' = g \mod P + 1$$
will issue the corresponding \texttt{SYNC}(i, j) operation. Based on these facts, only one buffer element per processor is needed. For a shared memory machine, \texttt{SYNC}(i, j) and \texttt{WAIT}(i, j) issued at processor \( g \) can be implemented as follows:

\begin{verbatim}
SYNC(i, j): syncBuffer[\( g' \)] := true;
WAIT(i, j): WHILE NOT syncBuffer[\( g \)] DO.
\end{verbatim}

For a message passing machine, the \texttt{SYNC}(i, j) issued from processor \( g \) can be implemented as

\begin{verbatim}
send(\( g' \));
\end{verbatim}

and the \texttt{WAIT}(i, j) operation issued in processor \( g \) can be implemented as

\begin{verbatim}
receive(\( g'' \));
\end{verbatim}
5.4. Implementation of SYNC/WAIT for Row-major Spreading Schemes

For row-major spreading schemes, each processor may issue many SYNC/WAITs. For example, for the 1-sequence spreading scheme, at least one processor must issue $N*(K-1)/P$ SYNC/WAITs, as there is a total of $N*(K-1)$ SYNC/WAITs among the $P$ processors. Thus the implementation in Section 3 for column-major spreading does not apply to the row-major spreading schemes.

However, if we can find a mapping

$$(i, j) \rightarrow o(i, j),$$

such that for any two $SYNC(i, j)$ and $SYNC(i', j')$ received by processor $g$, $o(i, j) < o(i', j')$ if and only if $SYNC(i, j)$ is received before $SYNC(i', j')$, then each processor only needs to store the number $o(i, j)$ for the most recently received $SYNC(i, j)$, and when it issues a $WAIT(i', j')$ operation, it only needs to check $o(i', j')$ against the stored $o(i, j)$. If $o(i', j') \leq o(i, j)$, then the corresponding $SYNC(i', j')$ must have been issued. Otherwise, it must wait for $SYNC(i', j')$.

In order to determine the mapping $o(i, j)$, we first establish the following results.
Theorem 5.1. For any of the four row-major spreading schemes, all of the SYNCs received at one processor are issued from the same processor.

Proof. Let \( p(s_j(i)) \) denote the processor that executes statement \( s_j(i) \). It is sufficient to show that if \( s_{j+1}(i') \) and \( s_{j''+1}(i'') \) are on the same processor, then statements \( s_j(i') \) and \( s_j(i'') \) must be on the same processor. We prove this for each of the row-major spreading schemes.

1-sequence spreading scheme. For 1-sequence spreading scheme, statements \( s_j(i) \) is executed on processor

\[
p(s_j(i)) = ((j-1)N + i - 1) \mod P + 1.
\]

If \( s_{j+1}(i') \) and \( s_{j''+1}(i'') \) are on the same processor, namely

\[
p(s_{j+1}(i')) - p(s_{j''+1}(i'')) = (j'N + i') \mod P - (j''N + i'') \mod P = 0,
\]

then

\[
p(s_j(i')) - p(s_j(i'')) = ((j'-1)N + i') \mod P - ((j''-1)N + i'') \mod P = 0,
\]
So, statements $s_j(i')$ and $s_j''(i'')$ must be on the same processor.

2-sequence spreading scheme. For 2-sequence spreading scheme, it is not difficult to see that statements $s_j(i)$ and $s_{j+1}(i)$ are executed on processors according to the following:

<table>
<thead>
<tr>
<th>statements</th>
<th>$p(s)$</th>
<th>ranges</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_j(i)$</td>
<td>$(mj-1+i-rj-1-1) \mod P +1$</td>
<td>$i = r_j-1 +1, \ldots, N$;</td>
</tr>
<tr>
<td>$s_j(i)$</td>
<td>$(mj-1+N + i-rj-1-1) \mod P +1$</td>
<td>$i = 1, \ldots, r_j-1$;</td>
</tr>
<tr>
<td>$s_{j+1}(i)$</td>
<td>$(mj+i-rj-1) \mod P +1$</td>
<td>$i = r_j + 1, \ldots, N$;</td>
</tr>
<tr>
<td>$s_{j+1}(i)$</td>
<td>$(mj+N + i-rj-1) \mod P +1$</td>
<td>$i = 1, \ldots, r_j$.</td>
</tr>
</tbody>
</table>

We only prove the case when $r_j-1 < r_j$. The proof is similar for $r_j \geq r_j$. We consider two cases.

1) Assume $i' \in (1, \ldots, r_j-1)$ and $i'' \in (1, \ldots, r_j-1)$. Since $s_{j+1}(i')$ is on processor $(mj'+N + i'-rj'-1) \mod P + 1$, and $s_{j+1}(i'')$ is on processor $(mj'' + N + i''-rj''-1) \mod P + 1$, $s_{j+1}(i')$ and $s_{j+1}(i'')$ being on the same processor means that

$$p(s_{j+1}(i')) - p(s_{j+1}(i'')) = (mj' + N + i'-rj') \mod P - (mj'' + N + i''-rj'') \mod P$$

$$= (mj' - mj'' + i' - i'' - rj' + rj'') \mod P = 0.$$  

Then, $s_j'(i')$ and $s_j''(i'')$ must be on the same processor because:
\[
p(s_j'(i')) - p(s_j''(i'')) \\
= (m_{j'-1} + N + i' - r_{j'-1}) \mod P - (m_{j''-1} + N + i'' - r_{j''-1}) \mod P \\
= (m_{j'-1} - m_{j''-1} + i' - i'' - r_{j'} - r_{j''}) \mod P \\
= (m_j' - m_j'' + i' - i'' - r_{j'} + r_{j''}) \mod P = 0.
\]

2) Assume \(i' \in (r_{j-1} + 1, \ldots, N)\) and \(i'' \in (1, \ldots, r_j - 1)\). Since \(s_j'+1(i')\) is on processor \((m_j' + i' - r_j - 1) \mod P + 1\), and \(s_j''+1(i'')\) is on processor \((m_j'' + N + i'' - r_j' - 1) \mod P + 1\), \(s_j'+1(i')\) and \(s_j''+1(i'')\) being on the same processor means that

\[
p(s_j'+1(i')) - p(s_j''+1(i'')) \\
= (m_j' + i' - r_j') \mod P - (m_j'' + N + i'' - r_j'') \mod P \\
= (m_j' - m_j'' + i' - i'' - r_j' + r_j'' - N) \mod P = 0.
\]

Then,

\[
p(s_j'+1(i')) - p(s_j''+1(i'')) \\
= (m_{j'-1} + i' - r_{j'-1}) \mod P - (m_{j''-1} + N + i'' - r_{j''-1}) \mod P \\
= (m_{j'-1} - m_{j''-1} + i' - i'' - r_{j'} - r_{j''} + N) \mod P \\
= (m_j' - m_j'' + i' - i'' - r_{j'} + r_{j''} - N) \mod P = 0.
\]

So, \(s_j'(i')\) and \(s_j''(i'')\) must be on the same processor.
3-sequence spreading scheme. For 3-sequence spreading scheme, statement $s_j(i)$ is executed on processor:

$$p(s_j(i)) = (j \cdot N - P + i - 1) \mod P + 1.$$ 

If $s_{j'+1}(i')$ and $s_{j''+1}(i'')$ are on the same processor, namely

$$p(s_{j'+1}(i')) - p(s_{j''+1}(i''))$$

$$= ((j'+1) \cdot N - P + i') \mod P - ((j''+1) \cdot N - P + i'') \mod P$$

$$= ((j' - j'') \cdot N + (i' - i'')) \mod P = 0,$$

then,

$$p(s_j'(i')) - p(s_j''(i''))$$

$$= (j' \cdot N - P + i') \mod P - (j'' \cdot N - P + i'') \mod P$$

$$= ((j' - j'') \cdot N + (i' - i'')) \mod P = 0.$$

So, statements $s_j'(i')$ and $s_j''(i'')$ must be on the same processor.

4-sequence spreading scheme. The proof is similar to that for the 2-sequence spreading, so we will omit it, here. Q.E.D.

The fact that all the SYNC's received at one processor will be sent from the same processor tells us that $o(i, j)$ is also the ordinal value of SYNC(i, j) within the SYNC(i, j)'s that are sent from the sending processor. Therefore, to determine $o(i, j)$, we need only to
know the order in which the SYNCs are issued from the same processors. The following theorem determines the value of \( o(i, j) \) for the row-major spreading sequences.

**Theorem 5.2.** 1) For 1-sequence row-major spreading scheme, \( o(i, j) \) can be the lexical order of \((j, i)\). 2) For all of the other row-major spreading schemes, we can use \( o(i, j) = j \).

**Proof.** The first result is immediate from the fact that for 1-sequence row-major spreading scheme, \( s_j(1), ..., s_j(N) \) are placed in that order before any of the \( s_{j+1}(i) \) are placed, and each of the \( s_j(i) \) needs to issue a SYNC to \( s_{j+1}(i) \).

The second result is immediate from the fact that for 2/3/4-sequence row-major spreading schemes, fewer than \( P \) of the \( s_j(1), ..., s_j(N) \) need to issue SYNCs, these SYNCs are sent to different processors, and \( s_j(i)'s \) are placed before any of the \( s_{j+1}(i) \) are placed. Q.E.D.

The following theorem determines the sending processor \( g' \) and receiving processor \( g'' \) for a processor \( g \).

**Theorem 5.3.** For the 1/2/3-sequence row-major spreading schemes and for the 4-sequence spreading scheme when \( N \mod P \leq (P - N \mod P) \),
\[ g' = (g - N - 1) \mod P + 1, \]
\[ g'' = (g - 1 + N) \mod P + 1, \]

and for the four sequence spreading scheme when \( N \mod P \geq (P - N \mod P), \)
\[ g' = (g + N-1) \mod P + 1. \]
\[ g'' = (g - N - 1) \mod P + 1. \]

**Proof.** This is based on the following three facts:

1. For the 1/3-sequence spreading schemes, when \( s_j(i) \) sends \( \text{SYNC}(i, j) \) to \( s_{j+1}(i) \), then \( s_{j+1}(i) \) is \( N \) statements following \( s_j(i) \).

2. For the 2-sequence spreading schemes, when \( s_j(i) \) sends \( \text{SYNC}(i, j) \) to \( s_{j+1}(i) \), then \( s_{j+1}(i) \) is \( N + P \) statements after \( s_j(i) \).

3. If \( N \mod P \leq (P - N \mod P) \), the 4-sequence spreading scheme is the same as the 3-sequence spreading scheme. If \( N \mod P > (P - N \mod P) \), when \( s_j(i) \) sends \( \text{SYNC}(i, j) \) to \( s_{j+1}(i) \), then \( s_{j+1}(i) \) is \( (P - N \mod P) + P \) statements after \( s_j(i) \).

Q.E.D.

Based on Theorems 4.1 to 4.3, for all of the four row-major spreading schemes we have studied, we can implement the \( \text{SYNC}/\text{WAIT} \) on a shared memory machine as follows:
Initialize: \[ \text{syncBuffer}[1..P] := 0; \]
\[ \text{SYNC}(i, j) \text{ from } g \text{ to } g': \quad \text{syncBuffer}[g'] := o(i, j); \]
\[ \text{WAIT}(i, j) \text{ at } g: \quad \text{WHILE (syncBuffer}[g] < o(i, j)) \text{ DO.} \]

In a message passing system, the SYNC and WAIT operations can be implemented as

\[ \text{SYNC}(i, j) \text{ from } g \text{ to } g': \quad \text{send}(g'); \]
\[ \text{WAIT}(i, j) \text{ for } g": \quad \text{receive}(g''). \]

For the above shared memory and message passing implementations, each processor needs to calculate \( g' \) and/or \( g'' \) only once.

5.5. Summary

For a parallel loop of form:

\[ \text{PAR } i:=1 \text{ TO } N \text{ DO } s1(i), ..., sk(i) \text{ ENDPAR;} \]
or

\[ \text{PAR } i:=1 \text{ TO } N \text{ DO} \]
\[ \quad \text{FOR } j := 1 \text{ TO } K \text{ DO} \]
\[ \quad \text{s}(i, j); \]
\[ \quad \text{ENDFOR;} \]
\[ \text{ENDPAR;} \]
the straightforward implementation of the synchronization primitives for loop spreading introduces potential runtime error, because the implementation has to use an array of size \((P + N \text{ MOD } P) \times (K - 1)\), in which the value of \(K\) has to be estimated if \(K\) is a variable.

By looking more closely at the five spreading schemes, we have succeeded in reducing the array from size \((P + N \text{ MOD } P) \times (K - 1)\) to a vector of size \(P\) without sacrificing performance. Since \(P\) is the number of processors on line, it is a constant known to the system.
Chapter 6

Summary and Future Work

6.1. Summary

The major results of this research can be summarized as follows.

. The revised two-phase simplex algorithm has been parallelized with linear improvement in performance. This result is much better than [FINKEL-87] and [CHOI-88].

. One of our new parallel decomposed simplex algorithms (TS) has achieved more than 2*P performance improvement over the sequential algorithm using P processors, which is nearly twice as efficient as the algorithm proposed in literature ([WYPIOR-77]).

. A new processor load balance technique named Loop Spreading is developed. Our experiment shows that this technique can significantly improve the performance of matrix multiplication and parallel decomposed simplex algorithms.

. As a general processor load balance technique, we showed that the loop spreading technique has the following properties:
1) Applicability can be checked without knowing loop bound (We supplied with the formulas) and appropriate version can be chosen in different situation.

2) The methods introduce very small overhead and can be applied to most loops.

3) When N mod P = 0, the spread loop will not run slower than the non-spread loop.

4) The method can be applied when the substatements inside the loop are data dependent.

6.2. Limitations of the Study

1) Many techniques that take advantage of the pattern of the input data are not incorporated in our simplex algorithms. Also, the strategies for controlling convergence, round-off error, etc. are not the emphasis of the research.

2) The test data we used for obtaining the performance results are generated randomly. This may restrict the validity of our result to "theoretical" linear programs.

3) Our experiments were done on a parallel machine with only 10 processors. Whether or not we can scale up the result here when more processors are added remains open.
6.3. Future Work

The following work is worthy of future research efforts:

1) Incorporating the techniques that take advantage of the pattern of the input data and the strategies for controlling convergence, round-off error etc. into our algorithms. One way to achieve this is to use the parallelization strategies studied here on some sequential program that has already been designed with these considerations.

2) Implementing our TS parallel decomposed simplex algorithm on a message passing based machine. We expect that the implementation on a message passing based machine will present even more challenges.

3) Automating the loop spreading techniques. Even though we showed that loop spreading can be done automatically, it is not trivial to implement such an automatic tool. For example, to decide whether or not loop spreading should be used, the tool must estimate the execution time of program segments. Further, it may turn out to be very hard to convert local variables to global variables that must span several processors.
Bibliography


[PRITSKER-86] Britsker, A. Alan, Introduction to Simulation and

[RIDEOUT-80] Rideout, V. L., "Limits to Improvement of Silicon
Integrated Circuits," Digest of Papers, IEEE Compcon (San

[SCOTT-84] Scott, M. L. and R. A. Finkel, "LYNX: A dynamic distributed
programming language," 1984 International Conference on

[SEITZ-84] Seitz, Charles, and Juri Matisoo, "Engineering Limits on

[SYSLO-83] Syslo, Maciej M., Deo, Narsiugh, and Kowalik, Janusz S.,
"Discrete Optimization Algorithms--with PASCAL Programs,"
Prentice-Hall, 1983.

Self-scheduling in Large Multiprocessor Systems," Proceeding
of 2nd SIAM Conference on Parallel Processing for Scientific

Supercomputing, Institute for Supercomputing, St. Petersburg.

the Linear Complementarity Problem," ORSA/TIMS - St. Louis,
Oct. 87.

[WOLFE-82] Wolfe, M.J., Optimizing Supercompiler for
Supercomputers, Ph.D. Thesis, University of Illinois, Urbana-
Champaign, 1982.


Appendices
Appendix A

Two-phase Revised Simplex Algorithm

In this appendix we give background on the two-phase revised simplex algorithm for solving linear programs.

A.1. Simplex Algorithm

A linear Program (LP) is a system that determines $x$ which

minimizes $z = \mathbf{c}^T \mathbf{x}$,

subject to $A \mathbf{x} = \mathbf{b}$, $\mathbf{x} \geq 0$

where $A$ is an $m$ by $n$ matrix ($n > m$), $\mathbf{c}$ is an $n$ element cost vector, $\mathbf{b}$ is a vector of length $m$, and $\mathbf{x}$ is an unknown vector of length $n$. The superscript $T$ denotes vector transposition.

Let $\mathbf{B}$ be any $m$ columns of $A$. If $\mathbf{B}$ is nonsingular and $\mathbf{x} = \mathbf{B}^{-1} \mathbf{b}$ $\geq 0$, then $\mathbf{B}$ is called a feasible base and $\mathbf{x}$ is called a base feasible solution ($\mathbf{x}$ is also an extreme point).
Assume $B$ is a feasible base of $A$; $N$ the non-base columns of $A$; $c_B$ and $x_B$ the vectors of $c$ and $x$ corresponding to $B$; $c_N$ and $x_N$ the vectors of $c$ and $x$ corresponding to $N$. That is

$$A = (B, N), \quad x=(x_B, x_N)^T, \quad c=(c_B, c_N)^T.$$ 

Then, we have:

$$Ax = b, \text{ or } Bx_B + Nx_N = b$$

$$x_B = B^{-1}b - B^{-1}N x_N$$

$$z = c^T x = c_B^T x_B + c_N^T x_N$$

$$= c_B^T B^{-1}b + (c_N^T - c_B^T B^{-1}N) x_N$$

$$= c_B^T x_0 + (c_N^T - c_B^T B^{-1}N) x_N$$

$$= z_0 + p^T x_N$$

where

$$x_0 = B^{-1}b,$$

$$z_0 = c_B^T x_0,$$

$$p^T = (c_N^T - c_B^T B^{-1}N).$$
If $B$ is the current feasible base, then, we can let $x_N = 0$, and $x_0 = x_B = B^{-1}b \geq 0$ is the current base feasible solution, $z_0$ is the current objective value. We want to find a non-base column $N_k = a_k$ to replace a column of $B$ so that the new base corresponds to a solution that decreases $z$. We can achieve this by selecting $a_k$ that satisfies

$$p_k = (c_{N_k} - c_B B^{-1} a_k) < 0$$

to replace a column of $B$ such that the resulting base $B'$ is still a feasible base.

Assume $x_k = B^{-1}a_k$, $z_k = c_B^T x_k$, $k = 1..n$, and $\theta$ which is a non-negative scalar value. Then

$$b - \theta a_k = Bx_0 - \theta Bx_k$$
$$b = \theta a_k + B(x_0 - \theta x_k)$$

Since $x_0 \geq 0$, if at least one component of $x_k$ is larger than zero, then we can always find $\theta = x_{oi}/x_{ki} = \min(x_{oj}/x_{kj} | x_{kj} > 0) > 0$ such that $x_0' = (x_0 - \theta x_k, \theta) = (x_{o1} - \theta x_{k1}, \ldots, x_{oi-1} - \theta x_{ki-1}, \theta, x_{oi+1} - \theta x_{ki+1}, \ldots, x_{om} - \theta x_{km}) \geq 0$.

Let $B_i$ be the column of $B$ corresponding to $x_{oi}$ and $a_k$ be a column of $N$ corresponding to $x_{ki}$. If we let $B'$ be the new base obtained by replacing $B_i$ with $a_k$, then $x_0'$ is the base feasible solution corresponding to $B'$. 
Similarly, we have

\[ z_0 - \theta \mathbf{z}_k = \mathbf{c}_B^T (\mathbf{x}_0 - \theta \mathbf{x}_k) \]

\[ z_0 - \theta \mathbf{z}_k^+ \mathbf{c}_{N_k} = \mathbf{c}_B^T (\mathbf{x}_0 - \theta \mathbf{x}_k) + \theta \mathbf{c}_{N_k} \]

\[ z_0 + \theta (\mathbf{c}_{N_k} - \mathbf{z}_k) = \mathbf{c}_B^T (\mathbf{x}_0 - \theta \mathbf{x}_k) + \theta \mathbf{c}_{N_k} \]

For the base \( B' \) and base feasible solution \( \mathbf{x}_0' \), the corresponding objective value is \( z_0' = z_0 + \theta (\mathbf{z}_k - \mathbf{c}_{N_k}) \). As

\[ \mathbf{p}_k = (\mathbf{c}_{N_k} - \mathbf{z}_k) < 0, \]

we know that \( z_0' < z_0 \). Notice that if we start with a linear program that maximizes \( z \), we can use the same method as above except requiring that \( p_k > 0 \).

Note that if \( x_k < 0 \) then \( z_0' \) can be made arbitrarily large, and the linear program is unbounded (-\( \infty \)). In this case, for any \( \theta \), \( \mathbf{x}_0 + \theta (1, -x_k) \) is a feasible solution (may not be base feasible) of the linear program. For the purpose of describing the decomposed algorithm later, we point out here that, for any \( \theta \), \( \theta (1, -x_k) \) is called a homogeneous solution of the LP, and the normalized homogeneous solution

\[ \frac{1}{1 - \sum_{i=1}^{\infty} x_{k_i}} * (1, -x_k) \]
is called an extreme homogeneous solution of the LP.

A.2. Revised Simplex Algorithm

To simplify the testing of \( p_k < (\text{or} >) 0 \), the revised simplex algorithm converts the LP

\[
\begin{align*}
\text{minimizes} & \quad z = c^T x \\
\text{subject to} & \quad Ax = b, \; x \geq 0
\end{align*}
\]

into the following equivalent problem:

\[
\begin{align*}
\text{maximizes} & \quad z \\
\text{subject to} & \quad Ax = b, \; x \geq 0, \; \text{and} \; c^T x + z = 0
\end{align*}
\]

This is equivalent to solving the following problem:

\[
\begin{align*}
\text{minimizes} & \quad z = c'^T x \\
\text{subject to} & \quad A'x = b', \; x \geq 0
\end{align*}
\]

where \( A' \) is obtained by augmenting the coefficient matrix \( A \) to

\[
A' = \begin{bmatrix} A & 0 \\ c^T & 1 \end{bmatrix}
\]

\( b' = (b \; 0)^T \), and \( c' = (0,0,\ldots,0,1)^T \).
If the current base of the argumented problem is $B$, $u_1, u_2, \ldots, u_m, u_{m+1}$ are the rows of $B^{-1}$, and $a_1, a_2, \ldots, a_{m+1}$ are the columns of $A'$, as $z$ must always be in the current base feasible solution, $c_{N_k} = 0$, and

$$p_k = c_{N_k}^T - c_B^T B^{-1} a_k = c_B^T B^{-1} a_k = -u_{m+1} a_k.$$  

So, to determine whether or not $a_k$ can be brought into the base, we only need to check that $u_{m+1} a_k < 0$.

To find an initial feasible base to start the simplex iterations, usually $m$ artificial variables $x_{n+1}, x_{n+2}, \ldots, x_{n+m}$ are added, and the coefficient matrix $A'$ is further augmented to $A''$

$$A'' = \begin{bmatrix} A & I & 0 \\ c^T & 0 & 1 \end{bmatrix}$$

With this coefficient matrix, the initial base is trivially $I_{m+1}$. When the simplex iteration stops, all of the artificial variables must be zero, if the given LP is feasible (there is at least one feasible solution).
A.3. Two-phase Revised Simplex Algorithm

In reality, a given LP may have no feasible solution at all. The two-phase revised simplex [ORCHARD-54] algorithm adds another constraint $x_{n+1} + x_{n+2} + \ldots + x_{n+m} + x_{n+m+2} = 0$ to determine whether or not a given LP is feasible and to ensure that all of the artificial variables converge to zero in the final base feasible solution. With the new constraint added, the coefficient matrix $A''$ is augmented to $A'''$,

$$A''' = \begin{bmatrix} A & I & 0 & 0 \\ c^T & 0 & 1 & 0 \\ 0 & e & 0 & 1 \end{bmatrix}$$

The initial base is

$$B = \begin{bmatrix} I_{m+1} & 0 \\ e & 1 \end{bmatrix} \quad \text{and} \quad B^{-1} = \begin{bmatrix} I_{m+1} & 0 \\ -e & 1 \end{bmatrix}$$

where $e$ stands for a row vector of all one's. Assume $u_1, u_2, \ldots, u_{m+2}$ are the rows of $B$, and $a_1, a_2, \ldots, a_{m+2}$ are the columns of $A'''$.

Phase I. The first phase of the algorithm is to maximize $x_{n+m+2}$ (with maximal value zero). Since the artificial variables $x_{n+i} \geq 0$, $i=1,\ldots,m$, the constraint $x_{n+1} + x_{n+2} + \ldots + x_{n+m} + x_{n+m+2} = 0$ ensures that $x_{n+m+2} \leq 0$. If the maximal value of $x_{n+m+2} < 0$, then the original LP has no feasible solution, since it means that $x_{n+1}, x_{n+2}, \ldots, x_{n+m}$ cannot be all zeros in any of the feasible solutions. In the first phase, the
condition to determine that the current objective value can be improved is $p_k = u_{m+2} a_k < 0$.

**Phase II.** When the first phase finishes, if $x_{m+n+2}$ is zero, the given LP is feasible and phase two can be started to maximize $z$. The last base during phase one can be used as the starting base of phase two. During phase two, $x_{m+n+2}$ must be kept zero to make all artificial variables remain zero when the simplex iteration finishes. To keep $x_{m+n+2} = 0$, we always keep it a component of the current base feasible solution (say it is $x_{Bm+2}$). Notice that after each iteration, $x_B' = (x_B - \theta x_k, \theta) = (x_B1-qx_k1,..,x_Bi-1-qx_ki-1,\theta, x_{Bi+1}-qx_ki+1,..,x_{Bm}-qx_km,z,x_{Bm+2}-qx_km+2)$ and $x_k = B^{-1} a_k$. As $\theta > 0$, to keep $x_{Bm+2}$ unchanged, we must make sure that $x_{km+2} = 0$. In other words, the column $a_k$ to enter the base must satisfy $u_{m+2} a_k = 0$ (two other methods to keep artificial variables zero in phase two can be found in [Murty-78, p72] and [Simmons-75, p140]). In the second phase, the condition to determine that the current objective value can be improved is $p_k = u_{m+1} a_k < 0$.

**A.4. Computational Procedure for Two-phase Revised Simplex Algorithm**

The two-phase revised simplex algorithm can be described procedurally as follows [SYSLO-83] (we assume $a_k$ is the k'th column of the matrix A, and $u_k$ is the k'th row of matrix U).
**Input.** A, an m*n matrix; b, an m vector; c, an n vector.

**Initialization.** Extend A to an (m+2)*n matrix by adding $c^T$ to be its m+1'th row and all zeros to be its m+2'th row;

the inverse of initial base $U = B^{-1} = \begin{bmatrix} I_{m+1} & 0 \\ -e & 1 \end{bmatrix}$;

the initial base feasible solution $x = (x_1, x_2, \ldots, x_{m+1}, x_{m+2}) = (b, 0, -\sum_{i=1}^{m} b_i)$;

the numbers of the base columns corresponding to the components of the optimal solution are remembered in vector $w$, with initial values of $w_1 = n+1$, $w_2 = n+2$, \ldots, and $w_{m+2} = n+m+2$;

phase = 1; q = m+2.

**Iteration.**

**step 1.** If $x_q = 0$ and phase = 1, set phase = 2, q = m+ 1.

For $j = 1, \ldots, n$, calculate $\delta_j = u_q * a_j$.

**step 2.** Calculate $\delta_k = \min(\delta_j \mid j = 1, \ldots, n)$.

If $\delta_k \geq 0$ and phase = 1, then the original LP is infeasible and stop. If $\delta_k \geq 0$ and phase = 2, then $x_q$ is the maximal and $-x_q$ is the minimal value of the original LP, exit the repetition.

Otherwise, $a_k$ is the new column to enter the base.

**step 3.** Compute $y_i = u_i * a_k$, $i = 1, \ldots, q$, and $\theta_i = x_i/y_i$ if $y_i > 0$, $i=1,\ldots,m$. 
step 4. If all $y_i \leq 0$, $i=1,...,m$, and phase =1, then the original LP is infeasible, stop. If all $y_i \leq 0$, $i=1,...,m$, and phase =2, then the original LP is unbounded. Otherwise, calculate

$$\theta = \theta_t = \min_{1 \leq i \leq m \& y_i > 0} (\theta_i)$$

$a_t$ is the column to be removed from the base.

step 5. Calculate the new values of the variables in the base solution and update $U$:

- $w_t = k$, $x_t = \theta$
- $x_i = x_i - \theta y_i$ (if $i \neq t$, $i = 1,...,q$),
- $u_{ij} = u_{ij} - y_i u_{tj}/y_t$ (if $i \neq t$, $i = 1,...,q$, $j = 1,...,m$)
- $u_{tj} = u_{tj}/y_t$

Output. The optimal objective value is $-x_q$, and the components of the optimal base feasible solution are $x_1, ..., x_m$, where $x_i$ corresponds to $a_{wi}$.
Appendix B

Sequential Algorithm for Decomposed Linear Programs

B.1. Decomposed Linear Programs

A decomposed linear program is to find \( x = (x_1, x_2, \ldots, x_n)^T \geq 0 \) which

\[
\text{minimizes } \sum_{j=1}^{n} c_j x_j
\]

subject to \( \sum_{j=1}^{n} A_j x_j = b, \ D_j x_j = b_j, \ x_j \geq 0, \) for \( j = 1, \ldots, n, \)

where for \( j=1..n, \ A_j \) is an \( m \) by \( n_j \) matrix; \( D_j \) an \( m_j \) by \( n_j \) matrix; \( b_j \) an \( m_j \) vector; \( c_j \) an \( n_j \) vector; \( x_j \) an \( n_j \) vector; and \( b \) an \( m \) vector.

The decomposed linear program in (I) has \( \sum_{j=1}^{n} n_j \) variables and \( m + \sum_{j=1}^{n} m_j \) constraints. The coefficient matrix of (I) has \( \sum_{j=1}^{n} n_j \) columns and \( m + \sum_{j=1}^{n} m_j \) rows. By directly applying the revised two-phase simplex algorithm to solve this problem, each iteration will operate
on a base of size \((m + \sum_{j=1}^{n} m_j) \times (m + \sum_{j=1}^{n} m_j)\), and the algorithm will take in the average 2 \((m + \sum_{j=1}^{n} m_j)\) iterations. When \((m + \sum_{j=1}^{n} m_j)\) is large, this program become intractable both in storage and execution time.

B.2. Dantzig and Wolfe's Decomposition Principle

The decomposition principle of Dantzig and Wolfe [DANTZIG-60] converts the decomposed program into an equivalent extremal problem of smaller size. To derive the equivalent extremal problem, assume that \(w_j = \{x_{j1}, x_{j2}, \ldots, x_{je}\}\) and \(v_j = \{y_{j1}, y_{j2}, \ldots, y_{jh}\}\) are the extreme points and the extreme homogeneous solutions of the subproblem \(S_j = \{D_j x_j = b_j, x_j \geq 0\}\). According to the resolution theorem of [MURTY-76], every solution \(x_j\) of \(S_j\) can be expressed as a convex combination of the extreme points in \(w_j\) plus a nonnegative combination of the extreme homogeneous solutions in \(v_j\). Namely, we can find \(s_{jk}\)'s and \(t_{jg}\)'s such that,

\[
x_j = \sum_{k=1}^{e_i} s_{jk} x_{jk} + \sum_{g=1}^{h_i} t_{jg} y_{jg}, \quad \text{and} \quad \sum_{k=1}^{e_i} s_{jk} = 1, \; s_{jk} \geq 0, \; t_{jg} \geq 0.
\]
For a solution \( \mathbf{x} = (x_1, x_2, \ldots, x_n) \geq 0 \) of (I), \( x_j \) must be a solution of \( S_j \), \( j = 1 \ldots n \), and thus should be able to be represented as above.

So, the constraints \( \sum_{j=1}^{n} A_j x_j = b \) can be rewritten as:

\[
\sum_{j=1}^{n} A_j x_j = \sum_{j=1}^{n} \left( \sum_{k=1}^{e_i} S_{jk} x_j + \sum_{g=1}^{h_i} t_{jg} y_{jg} \right) = b, \quad \text{and}
\]

\[
\sum_{k=1}^{e_i} S_{jk} = 1, \quad s_{jk} \geq 0, \quad t_{jg} \geq 0.
\]

and the objective function becomes

\[
\sum_{j=1}^{n} c_j x_j = \sum_{j=1}^{n} \left( \sum_{k=1}^{e_i} c_{jk} x_j + \sum_{g=1}^{h_i} c_{jg} t_{jg} \right)
\]

If we define the extremal problem as:

minimize \( \sum_{j=1}^{n} \sum_{k=1}^{e_i} c_{jk} s_{jk} + \sum_{j=1}^{n} \sum_{g=1}^{h_i} d_{jg} t_{jg} \)

subject to \( \sum_{j=1}^{n} \sum_{k=1}^{e_i} p_{jk} s_{jk} + \sum_{j=1}^{n} \sum_{g=1}^{h_i} q_{jg} t_{jg} = b, \) and

\[
\sum_{k=1}^{e_i} S_{jk} = 1, \quad s_{jk} \geq 0, \quad t_{jg} \geq 0.
\]

where \( c_{jk} = c_j x_{jk}, \) \( d_{jg} = c_j y_{jg}, \) \( p_{jk} = A_j x_{jk}, \) \( q_{jg} = A_j y_{jk}, \) and \( A_j \) and \( c_j \) are as defined in (I), then for every solution \( \mathbf{x} = (x_1, x_2, \ldots, x_n) \) of (I), there is a solution of (II): \( (s_{11}, s_{12}, \ldots, s_{e_1}, s_{21}, \ldots, s_{e_2}) \).
..., t_{11}, t_{12}, ..., t_{1h1}, t_{21}, ..., t_{ljh}, ...), \quad w = (x_{11}, x_{12}, ..., x_{1e1}, x_{21}, ..., x_{jej}, ...), \quad v = (y_{11}, y_{12}, ..., y_{1h1}, y_{21}, ..., y_{jh1}, ...), \quad \sum_{k=1}^{e_h} s_{jk} = 1, \quad \sum_{j=1}^{h_j} t_{jg} \geq 0, \quad t_{jg} \geq 0, \quad \text{such that}
\begin{align*}
x_j &= \sum_{k=1}^{e_h} S_{jk} x_{jk} + \sum_{g=1}^{h_j} t_{jg} y_{jg}. \quad \text{On the other hand, for every solution of the extremal problem: (s_{i1}, s_{i2}, ..., s_{iei}, s_{21}, ..., s_{jej}, ...),}
\end{align*}
\begin{align*}
w &= (x_{11}, x_{12}, ..., x_{1k}, x_{21}, ..., x_{jk}, ...), \quad v = (y_{11}, y_{12}, ..., y_{1h1}, y_{21}, ..., y_{jh1}, ...),
\end{align*}
x_j = \sum_{k=1}^{e_h} S_{jk} x_{jk} + \sum_{g=1}^{h_j} t_{jg} y_{jg} \quad \text{is also a solution of (I). Especially, the optimal solution of (II) corresponds to an optimal solution of (I). In other words, to solve the decomposed LP is equivalent to solve the extreme problem.}

In matrix form, the extremal problem is to minimize \( c'x \), subject to \( A'x = b' \) and \( x \geq 0 \), where,
\begin{align*}
c' &= (c_{i1}, ..., c_{iei}, d_{i1}, ..., d_{1h1}, ..., c_{n1}, ..., c_{nen}, d_{n1}, ..., d_{nhn}),
A' &= \begin{bmatrix}
p_{11}, ..., p_{1e1}, q_{11}, ..., q_{1h1}, ..., p_{n1}, ..., p_{nen}, q_{n1}, ..., q_{nh1} \\
1, ..., 1, 0, ..., 0, ..., 0, ..., 0, 0, ..., 0 \\
0, ..., 0, 0, ..., 0, ..., 0, ..., 0, ..., 0, ..., 0
\end{bmatrix},
\end{align*}
b' = (b, 1, ..., 1)^T.
To solve the extreme problem, we need a central solver to control the extreme problem and a subproblem solver for each of the subproblems for providing subsolutions to the central solver. We can use the two-phase revised simplex algorithm to be both the central problem solver and the subproblem solvers.

The extremal problem has as many variables as the total number of extreme points of all of the subproblems. In addition, it requires that the extreme points of $S_j$, $j = 1...n$, be known when needed. Since the number of extreme points of an LP is very large, we can not know all of them beforehand.

However, each iteration of the central solver operates on the current base which requires only $m+n$ of the extreme points to be known. To go to the next feasible base, we need only obtain another non-base column by obtaining a new subsolution of one of the subproblems.
B.3. Decomposed Simplex Algorithm

We use the two-phase revised simplex algorithm to be both the central problem solver and the subproblem solvers. Assume that the rows of the inverse of the central base are \( u_1, u_2, \ldots, u_{m+n+2} \). In phase one, column \( A'_j \) of \( A' \) can be brought into the current base to improve the objective value if \( u_{m+n+2} A_j < 0 \) (for phase two, using \( u_{m+n+1} A_j < 0 \)). As \( A'_j = (A_j x_{jk}, 0, \ldots, 0, d, 0, \ldots, 0, c_j x_{jk}, 0) \), where \( d \) is 1 if \( x_{jk} \) is an extreme point of subproblem \( S_j \) or 0 if \( x_{jk} \) is an extreme homogeneous solution of subproblem \( S_j \), \( u_{m+n+2} A'_j = u_{m+n+2,1..m} A_j x_{jk} + d u_{m+n+2,m+1} + u_{m+n+2,m+n+1} c_j x_{jk} = (u_{m+n+2,1..m} A_j + u_{m+n+2,m+n+1} c_j) x_{jk} + d u_{m+n+2,m+1} \). To find the column \( A'_j \) that has the smallest value of \( u_{m+n+2} A'_j \) as possible, the subproblems should try to find the \( x_{jk} \) that minimizes \( (u_{m+n+2,1..m} A_j + u_{m+n+2,m+n+1} c_j) x_{jk} \), namely, using \( (u_{m+n+2,1..m} A_j + u_{m+n+2,m+n+1} c_j) \) as the objective vector of all of the subproblems. If a subproblem is unbounded (when it finds a halfline \( x_e - \theta x_h \)), the subproblem has a choice to either return the extreme point \( x_e \) (if \( (u_{m+n+2,1..m} A_j + u_{m+n+2,m+n+1} c_j) x_e + u_{m+n+2,m+1} \) is smaller) or the extreme homogeneous solution \( x_h \) (if \( (u_{m+n+2,1..m} A_j + u_{m+n+2,m+n+1} c_j) x_h \) is smaller).

Corresponding to the two-phase revised simplex algorithm for solving normal LP's, we describe the algorithm for solving the decomposed linear program as follows (we assume \( a_k \) is the k'th column of the matrix \( A' \), and \( u_k \) is the k'th row of matrix \( U \)).
**Input.** $A_i$, $m \times n_i$ matrices and $D_i$, $m_i \times n_i$ matrices, $i = 1, \ldots, n$; $b_0$, an $m$ vectors; $b_i$, $m_i$ vectors, $i = 1, \ldots, n$; $c_i$, $n_i$ vectors, $i = 1, \ldots, n$.

**Initialization.** Assume $e$ be a vector of all $1$'s.

The inverse of initial base

$$U = (u_1, u_2, \ldots, u_{m+n+2}) = B^{-1} = \begin{bmatrix} I_{m+n+1} & 0 \\ -e & 1 \end{bmatrix};$$

the initial base feasible solution

$$s = (s_1, s_2, \ldots, s_{m+n+2}) = (b_0, 1, \ldots, 1, 0, \sum_{i=1}^{m} b_0 i);$$

the central left hand side vector $b = (b_0, 1, \ldots, 1, 0, 0);$ the initial subsolutions $e_x = (e_{x1}, e_{x2}, \ldots, e_{x_{m+n}}) = (0, 0, \ldots, 0)$ and the corresponding indices of the subproblems that lead to the subsolutions $w = (w_1, w_2, \ldots, w_{m+n}) = (0, 0, \ldots, 0)$, meaning that the initial subsolutions are not solutions of any subproblems (subproblems range from $1$ to $n$);

$$\text{phase} = 1; q = m+n+2.$$

**Iteration.**

**Step 1.** If $s_q = 0$ and phase = 1, then set phase = 2, $q = m+n+1$, and redo step 1. If $s_q < 0$ or phase = 2, then
a) calculate $c_j = (u_{q,1..m}A_j + u_{q,m+n+1}c_j)$, for $j = 1,...,n$.

b) using the two-phase revised simplex algorithm to solve sublinear problems $S_j$, for $j=1,...,n$,

\[ S_j: \begin{align*}
\text{minimize} & \quad c_j x_j, \\
\text{subject to} & \quad A_j x_j = b_j \quad \text{and} \quad x_j \geq 0
\end{align*} \]

for optimal solutions or extreme homogeneous solutions (if $S_j$ is unbound) $x_j, j = 1,...,n$. If any of the subproblems is infeasible, the original problem is infeasible, stop.

c) If $x_j$ is an optimal solution of $S_j$, make $a_j = (A_j x_j, 0, ..., 0, 1, 0, ..., 0, c_j x_j, 0)$, otherwise, make $a_j = (A_j x_j, 0, ..., 0, 0, 0, ..., 0, c_j x_j, 0)$.

d) For $j = 1,...,n$, calculate $\delta_j = u_q \ast a_j$. If phase = 2 then for $j = 1, ... , n$, calculate $\lambda_j = u_{m+n+2} \ast a_j$ and if $\lambda_j \neq 0$ then set $\delta_j = 0$.

Step 2. Calculate $\delta_k = \min(\delta_j \mid j = 1, ..., n)$. If $\delta_k \geq 0$ and phase = 1, then the original LP is infeasible, stop. If $\delta_k \geq 0$ and phase = 2, then $s_q$ is the optimal solution and $-s_q$ is the minimal value of the original LP, exit. Otherwise, $a_k$ is the new column to enter the base.
Step 3. Compute \( y_i = u_i \cdot a_k, i = 1, \ldots, q \).

Step 4. If all \( y_i \leq 0 \) and phase = 1, then the original LP is infeasible, stop. If all \( y_i \leq 0 \) and phase = 2, then the original LP is unbounded, stop. Otherwise, calculate

\[
\theta = \frac{1}{y_t} = \min_{1 \leq i \leq m+n, \gamma_i > 0} \begin{bmatrix} s_i \\ \gamma_i \end{bmatrix}
\]

and \( a_t \) is the column to be removed from the base.

Step 5. Calculate the new values of the variables in the base solution:

\[
w_t = k, \quad s_k = \theta \\
s_i = s_i - qy_i \quad (i \neq k, \quad i = 1, \ldots, m+n+2) \\
ex_k = x_k,
\]

and update \( U \), the inverse of the base:

\[
u_{ij} = u_{ij} - y_i \cdot u_{ij}/y_t \quad (i \neq t, \quad i = 1, \ldots, m+n+2, j = 1, \ldots, m+n+2) \\
\frac{u_{ij}}{y_t}.
\]

Output. The optimal objective value is \(-s_0\), and the optimal feasible solution (may not be basic) is \( x = (x_1, \ldots, x_n) \), where \( x_j \) is obtained from:

\[
x_j = \sum_{\forall \ w_i = j} s_i \cdot e_x_i
\]
B.4. Retaining Subsolutions Across Central Iterations

One performance consideration to the algorithm above is that, in different central iterations, the sublinear program that

\[
\begin{align*}
\text{minimize} & \quad c_j x_j, \\
\text{subject to} & \quad A_j x_j = b_j \text{ and } x_j \geq 0,
\end{align*}
\]

all have the same \( A_j \) and \( b_j \), and only \( c_j \) is changed from central iteration to central iteration. If an LP is feasible it will remain feasible when its object vector changes. Similarly a feasible solution will remain feasible to the LP even when its object vector changes. Thus a subproblem needs to run phase one only once. Later calls to the subproblem should use the optimal solution of the previous call as the starting point in phase two. Several complications result from this implementation.

First, the two-phase revised simplex algorithm adds the objective vector \( c_j \) to be the \( m+1 \)'th row of the coefficient matrix \( A_j \). Consequently, when \( c_j \) changes, the \( m+1 \)'th row of the last base should also be changed. Thus when provided with a new \( c_j \), the subproblem solver not only needs to assign \( c \) to the \( m+1 \)'th row of \( A_j \), but also needs to update the \( m+1 \)'th row of \( U \).

A straightforward way to update \( U \) is to first determine \( B \) and then invert it to get \( U \). But we can do better than this. Assume the
previous $U = B^{-1}$, and we want to get $U' = B'^{-1}$ where $B'$ is $B$ except that its $m+1$'th row is changed to $c' = (c_B, 1, 0)$, where $c_B$ is an $m$ vector.

Since $BB^{-1} = I$,

$$B'B^{-1} = J = \begin{bmatrix}
I_m & 0 & 0 \\
y_1 & y_2 & \cdots & y_{m+2} \\
0 & 0 & \cdots & 1
\end{bmatrix}$$

$$J^{-1} = \begin{bmatrix}
I_m & 0 & 0 \\
y_1 & y_2 & \cdots & y_{m+2} \\
0 & 0 & \cdots & 1
\end{bmatrix}^{-1} = \begin{bmatrix}
I_m & 0 & 0 \\
y_{m+1} & y_{m+2} & \cdots & 1 \\
0 & 0 & \cdots & 1
\end{bmatrix}$$

where $y = (y_1, y_2, \ldots, y_{m+2}) = c'U$. So, $U' = B'^{-1} = B^{-1}J^{-1}$. The computational procedure to find $U'$ is:

$$y = (y_1, y_2, \ldots, y_{m+2}) = c'U,$$

$$u'_{ij} = u_{ij} - y_j u_{i,m+1}/y_{m+1} \quad (i = 1, \ldots, m+2, j = 1, \ldots, m).$$

This operation is similar to a pivot operation in step 5 of the above algorithm.

Secondly, the $m+1$'th component of the last base feasible solution is the objective value of the last LP. With the new objective vector $c$, the $x_{m+1}$ should be assigned to $x_{1,m}c_{w1,wm}$.

Finally, when a subproblem finds out that the current LP is unbounded and finds a half line of feasible solutions in the form $x_0 + \theta(1, -x_k)$, it returns to the central problem an extreme homogeneous solution. However, when the next call to the subproblem is made with a new objective vector, the initial base feasible solution should still be $x_0$. 
Appendix C

Pascal Program for
Parallel Matrix Multiplication With Loop Spreading

(* *****************************************
 THIS PROGRAM EXPERIMENTS THE EFFECTS OF LOOP SPREAD TO BALANCE
 PROCESSOR LOADS ON MATRIX MULTIPLICATION, WHEN SUB-STATEMENTS ARE
 INDEPENDENT.
 (* *****************************************)

( MODULES:
  ( MATMUL -- SEQUENTIAL VERSION
  ( MMPARA -- PARALLEL BUT DOES NOT SPREAD
  ( MMSpread -- PARALLEL AND SPREAD THE LAST N MOD P ITERATIONS
  ( MMCollapse -- PARALLEL AND COLLAPSE N ITERATIONS

PROGRAM matrix_mul;
LABEL 99;
CONST
  SIZE = 50;  { size of matrices }
check = true;

TYPE
  str20 = string[20];
  matrix = array[1..SIZE, 0..SIZE] of real;
  integer = longint;

VAR
  a : matrix;  { first array }
  b : matrix;  { second array }
  c0, c1 : matrix;  { result arrays }
  m, n : integer;  { actual size of a matrix }
  nprocs, ts, tp, tl : longint;
  ret_val : longint;

FUNCTION cpus_online: longint; cexternal;
PROCEDURE m_lock; cexternal;
PROCEDURE m_unlock; cexternal;
FUNCTION m_set_procs(i : longint): longint; cexternal;
FUNCTION m_pfork(PROCEDURE a): longint; cexternal;
FUNCTION m_get_numprocs : longint; cexternal;
FUNCTION m_get_myid : longint; cexternal;
PROCEDURE m_kill_procs; cexternal;
PROCEDURE m_multi; cexternal;
PROCEDURE m_single; cexternal;
FUNCTION sTime(maxSecs: longInt): longint; cexternal;
FUNCTION gTime: longint; cexternal;  { return time in miniseconds }
PROCEDURE init_matrix;
VAR
i, j : integer;
nprocs : integer;
BEGIN
nprocs := m_get_numprocs; { number of processes }
i := 1 + m_get_myid; { start at i'th iteration }
while (i <= n) do BEGIN
  for j := 1 to n do BEGIN
    a[i, j] := i + j;
    b[i, j] := n - j;
  END;
i := i + nprocs;
END;
END; { init_matrix }

PROCEDURE dummy;
BEGIN END;

PROCEDURE innerProd(i,j : integer; VAR c : matrix);
VAR
k : integer;
ct : real;
BEGIN
  ct := 0;
  for k := 1 to n do
    ct := ct + a[i, k] * b[k, j];
  c[i,j] := ct;
END;

PROCEDURE matmul; { sequential procedure }
VAR
i, j, k : integer;
BEGIN
  for i:= 1 to n do
    for j := 1 to n do
      innerProd(i,j, c0);
END; { procedure matmul }

PROCEDURE mmpara ;
VAR
i, j, k : integer;
nprocs : integer; { number of processes }
BEGIN
  nprocs := m_get_numprocs;
i := 1 + m_get_myid; { start at i'th iteration }
while (i <= n) do BEGIN
  for j := 1 to n do
    innerProd(i,j, c1);
i := i + nprocs;
END;
END; { mmpara}

{ spreading the last n mod p iterations }

PROCEDURE mmspread;
VAR
ij : integer;
lu, start, m, u, i, j, k : integer;
nprocs : integer; { number of processes }
BEGIN
nprocs := m_get_numprocs;
start := 1 + m_get_myid; { start at i'th iteration }
m := n mod nprocs;
u := n - m;
i := start;
while (i <= u) do BEGIN
for j := 1 to n do
innerProd(i,j, cl);
i := i + nprocs;
END;
lu := m * n;
i := start + U;
ij := start;
j := 1;
while (ij <= lu) do BEGIN
if i > n do BEGIN
i := i - M; j := j + 1;
END;
innerProd(i,j, cl);
i := i + nprocs;
ij := ij + nprocs;
END;
END; { mmspread }

PROCEDURE mmcollapse;
VAR
ij : integer;
lu, start, u, i, j, k : integer; { local loop indices }
nprocs : integer; { number of processes }
BEGIN
nprocs := m_get_numprocs;
start := 1 + m_get_myid; { start at i'th iteration }
i := start;
ij := start;
j := 1;
lu := n*n;
while (ij <= lu) do BEGIN
if i > n do BEGIN
i := i - M; j := j + 1;
END;
innerProd(i,j, cl);
i := i + nprocs;
ij := ij + nprocs;
END;
PROCEDURE check_mats (algnmae: str20; VAR c1, c2 : matrix);
VAR
  i, j : integer;
  error : boolean;
BEGIN
  error := false;
  for i := 1 to n do
    for j := 1 to n do BEGIN
      if c1[i,j] <> c2[i,j] then BEGIN
        writeln(id:2, i:5,j:5,cl[i,j] :8:2,'< >', c2[i,j]:4:2);
        error := true;
      END;
    END;
  END;
  if error then BEGIN
    writeln;
    writeln('Panic! result is incorrect in ', algnmae);
  END;
END; { check_mats }

PROCEDURE abortIt(str: str20);
BEGIN
  writeln(str, ' error');
  goto 99;
END;

BEGIN { main program starts here }

write('Enter size of matrix:'); readln(n); nprocs := 0;
while (nprocs < 1) OR (nprocs > cpus_online) DO BEGIN
  write('Enter number of processes:'); readln(nprocs);
END;
IF (m_set_procs(nprocs) < 0) then abortIt('msetproc');

{ fork dummy processes to increase timing accuracy }
if (m_pfork(dummy) <> 0) then abortIt('fork dummy');

{ initialize data arrays }
if (m_pfork(init_matrix) <> 0) then abortIt('fork init_matrix');
{ timer counting down from 360000 (secs) }
if (sTime(360000) < 0) then abortIt('sTime');

writeln('Timing result in milliseconds:');
writeln('nbr-procs sequential parallel para+spread collapse');

{ sequential run }
tl := gTime;
for i := 1 to 10 do
  matmul;
  ts := gTime - tl;
  write(nprocs:8, ', ', ts:8, ', ');

{ sequential run }
tl := gTime;
for i := 1 to 10 do
  if (m_pfork(mmpara) <> 0) then abortIt('fork mmpara ');
  tp := gTime - tl;
  write(tp:8, ' ');

if check then check_mats('mmpara', c0, c1); { check results }

{ sequential run }
tl := gTime;
for i := 1 to 10 do
  if (m_pfork(mmspread) <> 0) then abortIt('fork mmspread ');
  tp := gTime - tl;
  write(tp:8, ' ');

if check then check_mats('mmspread', c0, c1); { check results }

{ sequential run }
tl := gTime;
for i := 1 to 10 do
  if (m_pfork(mmcollapse) <> 0) then abortIt('fork mmcollapse ');
  tp := gTime - tl;
  write(tp:8, ' ');

if check then check_mats('mmcollapse', c0, c1); { check results }

writeln;
m_kill_procs; { terminate child processes }

99:
END. { main program }
Appendix D

Pascal Program for Parallel Modified Matrix Multiplication With 1/2/3/4-sequence Row-major and Column-major Loop Spreading

(*-----------------------------------------------*)
{ this program experiments the effects of loop spreading to balance }
{ processor loads on Modified Matrix Multiplication (MMM), when }
{ sub-statements are dependent. }
(*-----------------------------------------------*)
{ MMM is like matrix multiplication but modified such that the second }
{ loop iterations are totally data dependent. }
{ Modules: }
{ Mmatmul -- the sequential version }
{ MMMpara -- the parallel one that does not spread }
{ MMM1SS -- row-major 1 sequence spreading }
{ MMM2SS -- row-major 2 sequence spreading }
{ MMM3SS -- row-major 3 sequence spreading }
{ MMM4SS -- row-major 4 sequence spreading }
{ MMMColS -- column-major spreading }
(*-----------------------------------------------*)

PROGRAM modified_matrix_mul ; { MMM algorithm }
LABEL 99;

CONST
  SIZE = 50; { maximum size of matrices }
  NBRPROCS = 9; { max number of processors }

TYPE
  str20 = string[20];
  matrix = array[1..SIZE, 0..SIZE] of real;
  INTEGER = LONGINT;

VAR
  a : matrix ; { first array }
  b : matrix ; { second array }
  c0, c1 : matrix ; { result array }
  i, j, n : INTEGER;
  nprocs, t1, t2: INTEGER;
  syncBuffer : array[1..NBRPROCS] of INTEGER;

FUNCTION cpus_online: INTEGER; cexternal;
PROCEDURE m_lock; cexternal;
PROCEDURE m_unlock; cexternal;
FUNCTION m_set_procs(i : INTEGER) : INTEGER; cexternal;

(*-----------------------------------------------*)

FUNCTION cpus_online: INTEGER; cexternal;
PROCEDURE m_lock; cexternal;
PROCEDURE m_unlock; cexternal;
FUNCTION m_set_procs(i : INTEGER) : INTEGER; cexternal;
FUNCTION m_pfork(PROCEDURE a): INTEGER; cexternal;
FUNCTION m_get_numprocs : INTEGER; cexternal;
FUNCTION m_get_myid : INTEGER; cexternal;
PROCEDURE m_kill_procs; cexternal;
PROCEDURE m_multi; cexternal;
PROCEDURE m_single; cexternal;
FUNCTION sTime(maxSecs: INTEGER): INTEGER;cexternal;
FUNCTION gTime: INTEGER;cexternal; { milliseconds }

{ initialize matrix FUNCTION }
PROCEDURE init_matrix :
VAR
  i, j : INTEGER;
  nprocs : INTEGER;
BEGIN
  nprocs := m_get_numprocs; { number of processes }
  i := 1 + m_get_myid; { start at i'th iteration }
  WHILE (i <= n) DO BEGIN
    FOR j := 0 to n DO BEGIN
      a[i, j] := i + j;
      b[i, j] := n - j;
    END;
    i := i + nprocs;
  END;
END; { init_matrix }

PROCEDURE dummy;
{ remove the time diff between the first m_pfork and later m_pforks. }
BEGIN END;

PROCEDURE innerProd(i,j : INTEGER; VAR c : matrix);
VAR
  k :INTEGER;
  ct :real;
BEGIN
  ct := 0;
  FOR k := 1 to n DO
    ct := ct + a[i, k] * b[k, j];
    c[i,j] := c[i,j-1] + ct;
END;

PROCEDURE Mmatmul; { sequential PROCEDURE }
VAR
  i, j : INTEGER;
BEGIN
  FOR i := 1 to n DO BEGIN
    FOR j := 1 to n DO BEGIN
      innerProd(i,j, c0);
    END;
  END;
END; { PROCEDURE Mmatmul }
PROCEDURE MMMpara;
VAR
  i, j : INTEGER;
  nprocs : INTEGER;  { number of processes }
BEGIN
  nprocs := m_get_numprocs;  { number of processes }
  i := 1 + m_get_myid;  { start at i'th iteration }
  WHILE (i <= n) DO BEGIN
    FOR j := 1 to n DO BEGIN
      innerProd(i,j,c1);
    END;
    i := i + nprocs;
  END;
END;  { MMMpara }

{ Sync/Wait procedures for 1-sequence spreading }
PROCEDURE WAIT1(sendProc, j, i: INTEGER);
VAR oij : INTEGER;
BEGIN
  oij := j*m+i;
  WHILE (syncBuffer[sendProc] < oij) DO;
END;

PROCEDURE SYNC1(currProc, j, i: INTEGER);
VAR oij : INTEGER;
BEGIN
  oij := j*m+i;
  syncBuffer[currProc] := oij;
END;

{ Sync/Wait procedures for 2/3/4-sequence and column-major spreading }
PROCEDURE WAIT234C(sendProc, j, i: INTEGER);
BEGIN
  WHILE (syncBuffer[sendProc] < j) DO;
END;

PROCEDURE SYNC234C(currProc, j, i: INTEGER);
BEGIN
  syncBuffer[currProc] := j;
END;

{ use 1-sequence spreading: needs M*(k-1) syncs }
PROCEDURE MMM1SS;
VAR
  sendProc : INTEGER;
  ij, r : INTEGER;
  nprocs : INTEGER;  { number of processes }
  lu, currPid, m, u, ii, jj, i, j : INTEGER;
BEGIN
  nprocs := m_get_numprocs;
  currPid := 1 + m_get_myid;
  IF n <= nprocs THEN
m := 0
else BEGIN
    r := n mod nprocs;
    m := nprocs + r;
END;
u := n - m;
i := currPid;
sendProc := (currPid - r);
IF sendProc <= 0 THEN sendProc := sendProc + nprocs;
WHILE (i <= u) DO BEGIN
    FOR j := 1 to n DO innerProd(i, j, cl);
i := i + nprocs;
END;
lh := m * n;
i := currPid;
ij := currPid;
j := 1;
WHILE (ij <= lh) DO BEGIN
    IF i > M THEN BEGIN { we know: M >= nprocs }
        i := i - M; j := j + 1;
    END;
    IF j > 1 THEN WAIT1(sendProc, j-1, i);
    ii := i + j;
    innerProd(ii, j, cl);
    IF j < n THEN SYNC1(currPid, j, i);
i := i + nprocs;
ij := ij + nprocs;
END;
END; { MMM1SS }

{ use 2-sequence spreading: needs (M MOD P)*(k-1) syncs }
PROCEDURE MMM2SS ;
VAR
    sendProc : INTEGER;
    rj, g, r : INTEGER;
nprocs : INTEGER; { number of processes }
lh, currPid, m, u, j, i, j : INTEGER;
BEGIN
    nprocs := m_get_numprocs;
currPid := 1 + m_get_myid;
    IF n <= nprocs THEN
        m := 0
    else BEGIN
        r := n mod nprocs;
        m := nprocs + r;
    END;
u := n - m;
i := currPid;
    sendProc := (currPid - r);
    IF sendProc <= 0 THEN sendProc := sendProc + nprocs;
    WHILE (i <= u) DO BEGIN
        FOR j := 1 to n DO innerProd(i, j, cl);
i := i + nprocs;
    END;
END;

lu := m * n;
i := currPid;
g := currPid;
j := 1;
Mrj := m;
WHILE (g <= lu) DO BEGIN
  IF i > M THEN BEGIN
    i := i - M; j := j + 1;
    Mrj := Mrj - r; IF Mrj <= 0 THEN Mrj := Mrj + M;
  END;
  IF (i <= Mrj) THEN
    ii := i + M - Mrj + U
  ELSE
    ii := i - Mrj + U;
  IF (j > 1) AND (i > nprocs) THEN WAIT234C(sendProc, j-1, ii);
  innerProd(ii, j, cl);
  IF (j < n) AND (i <= r) THEN SYNC234C(currPid, j, ii);
  i := i + nprocs;
g := g + nprocs;
END;
END; { MMM2SS }

{ use 3-sequence spreading: needs (M MOD P)*(k-1) syncs }
PROCEDURE MMM3SS;
VAR
  sendProc : INTEGER;
r, x, y, z, j1M, Uj1M : INTEGER;
lu, currPid, s,t, m, u, n1, ii, jj, i, j : INTEGER;
nprocs : INTEGER; { number of processes }
BEGIN
  nprocs := m_get_numprocs;
currPid := 1 + m_get_myid;
  IF n <= nprocs THEN
    m := 0
  else BEGIN
    r := n mod nprocs;
    m := nprocs + r;
  END;
  u := n - m;
i := currPid;
sendProc := (currPid - r);
  IF sendProc <= 0 THEN sendProc := sendProc + nprocs;
  WHILE (i <= u) DO BEGIN
    FOR j := 1 to n DO innerProd(i, j, c1);
    i := i + nprocs;
END;

  x := currPid; { kept as (a+g) mod M }
  Uj1M := U; { kept as U-(j-1)*M }
j := 1;
  j1M := 0; { kept as (j-1)*M }
y := 0; { kept as (j-1)*M mod P }
\[ z := \text{nprocs}; \]
\[ lu := m*n; \]
\[ i := \text{currPid}; \]

WHILE (\[i \leq lu\]) DO BEGIN
  IF \(x > M\) THEN \{ M always >= P \} BEGIN
    \[x := x - M;\]
    \[Uj1M := Uj1M - M; j := j + 1;\]
    \[j1M := j1M + M;\]
    \[y := y + r;\]
    IF \(y >= nprocs\) THEN \(y := y - nprocs;\)
    \[z := j1M - y + nprocs;\]
  END;
  \[x := x + nprocs;\]
  IF \(i <= z\) THEN BEGIN
    \[ii := y + i + Uj1M;\]
    \[\text{innerProd}(ii, j, c1);\]
  END ELSE IF \(i <= (z + y)\) THEN BEGIN
    \[ii := i + Uj1M;\]
    \[\text{innerProd}(ii, j, c1);\]
  END ELSE BEGIN
    IF \(j > 1\) THEN \text{WAIT234C}(sendProc, j-1, 0);
    \[ii := i + Uj1M;\]
    \[\text{innerProd}(ii, j, c1);\]
    IF \(j < n\) THEN \text{SYNC234C}(currPid, j, 0);
  END; \{ IF \}
  \[i := i + nprocs;\]
END; \{ WHILE \}
END; \{ MMM3SS \}

\{ use 4-sequence spreading: needs MIN ((M MOD P), (P - (M MOD P))*(k-1)) syncs \}

PROCEDURE MMM4SS;
VAR
  sendProc : INTEGER;
  jm, jlpMODr, pMODr, cj1, r, x, mj1, a0, a, b, j1M, Uj1M : INTEGER;
  lu, currPid, s,t, m, u, n1, ii, jj, i, j : INTEGER;
  offset, nprocs : INTEGER; \{ number of processes \}
BEGIN
  \[nprocs := \text{m_get_numprocs};\]
  \[currPid := 1 + \text{m_get_myid};\]
  IF \(n <= \text{nprocs}\) THEN
    \[m := 0;\]
  else BEGIN
    \[r := n \mod \text{nprocs};\]
    \[m := \text{nprocs} + r;\]
  END;
  \[u := n - m;\]
  \[i := \text{currPid};\]
  WHILE (\[i \leq u\]) DO BEGIN
    FOR \(j := 1\) to \(n\) DO BEGIN
      \[\text{innerProd}(i,j, c1);\]
    END;
  END;
  \[i := i + \text{nprocs};\]
END;

\[x := \text{currPid};\] \{ kept as (a+g) mod M \}
Uj1M := U;  \{ kept as U-(j-1)*M \}
j1M := 0;  \{ kept as (j-1)*M \}
mj1 := 0;  \{ kept as (j-1)*M mod P \}

jm := M;
b := nprocs;
j1pMODr := 0;
a := nprocs;
a0 := nprocs;
lu := m*n;
i := currPid;

IF R > (nprocs - R) THEN BEGIN
  cj1 := R;
pMODr := nprocs MOD R;
sendProc := (currPid + R);
  IF sendProc > nprocs THEN sendProc := sendProc - nprocs;
END ELSE BEGIN
  cj1 := 0;
sendProc := (currPid - r);
  IF sendProc <= 0 THEN sendProc := sendProc + nprocs;
END;

WHILE (i <= lu) DO BEGIN
  IF x > M THEN \{ M always >= nprocs \} BEGIN
    x := x - M; Uj1M := Uj1M - M; j := j + 1;
j1M := j1M + M;
jM := jM + M;
  IF R > (nprocs - R) THEN BEGIN
    j1pMODr := j1pMODr + pMODr;
    IF j1pMODr >= R THEN BEGIN
      j1pMODr := j1pMODr - R;
    END;
cj1 := R - j1pMODr;
  END ELSE
    cj1 := 0;
  b := jM - cj1;
mj1 := mj1 + r;
  IF mj1 >= nprocs THEN mj1 := mj1 - nprocs;
a0 := j1M + nprocs;
a := a0 - mj1;
END;
x := x + nprocs;
IF i <= a THEN BEGIN
  ii := mj1 + i + Uj1M;
  innerProd(ii, j, c1);
END ELSE IF i <= a0 THEN BEGIN
  ii := i + U - a;
  innerProd(ii, j, c1);
END ELSE BEGIN
  offset := i - a0;
  IF i <= b THEN
    ii := offset + U + nprocs + cj1
  ELSE
    ii := i + U - b + nprocs;
  IF (j > 1) AND (offset <=(nprocs-R)) THEN
    WAIT234C(sendProc, j-1, 0);
innerProd(i,j, c1);

IF (j < n) AND (offset > (r + R - nprocs)) AND ((i- a0) <=(R)) THEN
  SYNC234C(currPid), j, 0);
END; { IF }
i := i + nprocs;
END; { WHILE }
END; { MMM4SS }

{ use column-major spreading. }
PROCEDURE MMMColS;
VAR
  sendProc : INTEGER;
  q, shares, quote, busyOnes : INTEGER;
  first, firstPart, lastPart, lastStartq, jn : INTEGER;
  i, j : INTEGER;
  currPid, nprocs: INTEGER;
BEGIN
  nprocs := m_get_numprocs;
  currPid := 1 + m_get_myid;
  shares:= (N*n + nprocs - 1) DIV nprocs;
  quote := shares;
  busyOnes := (N*n) MOD nprocs;
  IF busyOnes = 0 THEN
    busyOnes := nprocs;
  IF currPid > busyOnes THEN BEGIN
    quote := shares - 1;
    first := busyOnes + (currPid - 1)*quote + 1;
  END ELSE
    first := (currPid - 1)*quote + 1;
  i := 1 + (first - 1) DIV n;
  firstPart := i*n - first + 1;
  jn := firstPart;
  lastPart := (quote - firstPart) MOD n;
  lastStartq := quote - lastPart + 1;
  j := 1;
  sendProc := (currPid + 1);
FOR q :=1 TO quote DO BEGIN
  IF (q = lastStartq) AND (lastPart < n) THEN BEGIN
    j := n - lastPart + 1;
    WAIT234C(sendProc, j-1, 0);
  END;
  innerProd(i,j, c1);
  IF (q = firstPart) AND (j < n) THEN
    SYNC234C(currPid, j);
  j := j + 1;
  IF j > jn THEN BEGIN
    jn := n;
    j := 1; i := i + 1;
  END { IF }
END { FOR }
END; { MMMColS }
PROCEDURE check_mats (alg: INTEGER; VAR cl, c2 : matrix);
VAR
   i, j : INTEGER;
   error : BOOLEAN;
BEGIN
   error := false;
   FOR i := 1 to n DO BEGIN
      FOR j := 1 to n DO BEGIN
         IF cl[i,j] <> c2[i,j] THEN BEGIN
            error := true;
         END;
      END;
   END;
   IF error THEN BEGIN
      writeln('Panic!! ', alg, ' result incorrect!');
   END;
END; {check_mats}

PROCEDURE abortIt(str: str20);
BEGIN
   writeln(str, ' error');
   goto 99;
END;

BEGIN { main program start here}

write('Enter size of matrix:');
readln(n); { matrix size n*n }
write('Enter number of processes:');
readln(nprocs);
IF (m_set_procs(nprocs) < 0) THEN abortIt('msetproc');
IF (m_pfork(dummy) <> 0) THEN abortIt('fork dummy');

{ initialize data arrays }
IF (m_pfork(init_matrix) <> 0) THEN abortIt('fork init_matrix');

{ timer counting down from 360000 (secs) }
if (sTime(360000) < 0) then abortlt('sTime');

writeln('Timing result in miliseconds:');
writeln('procs sequential parrale 1-spread 2-spread 3-spread 4-spread c-spread');
FOR i := -1 TO 5 DO BEGIN
   tl := gTime;
   FOR j := 1 to 10 DO BEGIN { run 10 times on the same data }
      CASE i OF
         -1: { sequential run result in c0 }
            Mmatmul;
         0: { parallel run w/o spreading result in c1 }
            IF (m_pfork(MMMpara) <> 0) THEN abortIt('fork MMMpara');
         1: { parallel run with 1-spreading result in c1 }
            IF (m_pfork(MMM1SS) <> 0) THEN abortIt('fork MMM1SS');
         2: { parallel run with 2-spreading result in c1 }
            IF (m_pfork(MMM2SS) <> 0) THEN abortIt('fork MMM2SS');
      END;
   END;
END;
3: { parallel run with 3-spreading result in c1 }
   IF (m_pfork(MMM3SS) <> 0) THEN abortIt('fork MMM3SS');
4: { parallel run with 4-spreading result in c1 }
   IF (m_pfork(MMM4SS) <> 0) THEN abortIt('fork MMM4SS');
5: { parallel run with col-spreading result in c1 }
   IF (m_pfork(MMMMColS) <> 0) THEN abortIt('fork MMMColS');
END; { CASE }
END; { FOR }
t2 := gTime - t1;
IF i = -1 THEN
   write(nprocs:5);
ELSE
   check_mats(i, c0, c1) { check results c0 and c1 }
   write(t2:5, ' ');
END; { FOR }
writeln;
99: m_kill_procs; { terminate child processes }
END. { main program }