The work/exchange model: a generalized approach
to dynamic load balancing

Milton Curtis Wikstrom
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/rtd
Part of the Computer Sciences Commons

Recommended Citation
http://lib.dr.iastate.edu/rtd/9787

This Dissertation is brought to you for free and open access by Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
The Work/Exchange Model: A generalized approach to dynamic load balancing

Wikstrom, Milton Curtis, Ph.D.

Iowa State University, 1991
The Work/Exchange Model:
A generalized approach to dynamic
load balancing

by

Milton Curtis Wikstrom

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Major: Computer Science

Approved:
Signature was redacted for privacy.

In Charge of Major Work
Signature was redacted for privacy.

For the Major Department
Signature was redacted for privacy.

For the Graduate College

Iowa State University
Ames, Iowa
1991
# TABLE OF CONTENTS

**DEDICATION** ................................................. ix

1. **INTRODUCTION** .......................................... 1

2. **LITERATURE SURVEY** .................................... 6
   2.1 Static Load Balancing ................................. 6
      2.1.1 Independent tasks ............................... 8
      2.1.2 Data dependencies ............................... 9
   2.2 Dynamic Load Balancing ............................. 10
      2.2.1 Algorithms ...................................... 11
      2.2.2 Theoretical foundations ....................... 17
   2.3 Real-Time Load Balancing ......................... 21

3. **THE WORK/EXCHANGE MODEL** .......................... 23
   3.1 Introduction .......................................... 23
   3.2 The Work/Exchange Model ........................... 24
      3.2.1 The original model ............................ 24
      3.2.2 Simplified model .............................. 26
   3.3 Derivation of Equations ............................ 29
      3.3.1 Case 1: $C > D$ ............................... 31
      3.3.2 Case 2.1: $\frac{D}{2} < C \leq D$ ............... 35
3.3.3 Case 2.2: \( 0 \leq C \leq \frac{D}{2} \) ............................................. 42
3.4 Verification of Equations .................................................. 44
   3.4.1 Simulation .............................................................. 44
   3.4.2 Implementation test .................................................. 46
3.5 Perfectly Balanced Case .................................................. 47
3.6 Speedup Equations ......................................................... 52
   3.6.1 Case 1 speedup equations ........................................... 52
   3.6.2 Case 2.1 speedup equations ....................................... 54
   3.6.3 Case 2.2 speedup equations ....................................... 55
3.7 Speedup Graphs ............................................................. 56
3.8 Conclusions ................................................................. 59

4. MYTHS OF LOAD BALANCING .............................................. 64
   4.1 Introduction .............................................................. 64
   4.2 Myth 1 ...................................................................... 66
      4.2.1 To balance or not to balance? ................................ 67
      4.2.2 Experimental results ........................................... 68
   4.3 Myth 2 ...................................................................... 71
   4.4 Myth 3 ...................................................................... 74
   4.5 Conclusions .............................................................. 76

5. AN ENHANCER FOR DYNAMIC LOAD BALANCING ...................... 77
   5.1 Introduction .............................................................. 77
   5.2 Extensions Using Temporal Locality ................................. 79
      5.2.1 Temporal locality ................................................ 80
      5.2.2 Enhancements to the load balancer ............................ 80
5.3 Experimental Results for Nondeterministic Load Imbalance 82
   5.3.1 Problem description 82
   5.3.2 Results 83
5.4 Conclusions 85
6. THE GENERALIZED WORK/EXCHANGE MODEL 87
   6.1 Introduction 87
   6.2 The Generalized Model 89
   6.3 Elapsed Time Equations 92
      6.3.1 Case 1: $C > D_{01}$ 93
      6.3.2 Case 2.1: $\frac{D_{01}}{2} < C \leq D_{01}$ 96
      6.3.3 Case 2.2: $0 \leq C \leq \frac{D_{01}}{2}$ 103
   6.4 Potential Speedup 106
      6.4.1 Perfectly balanced case 106
      6.4.2 Case 1 speedup 109
      6.4.3 Case 2.1 speedup 111
      6.4.4 Case 2.2 speedup 112
   6.5 Verification of Equations 115
   6.6 Summary 117
7. A FORMAL DEFINITION OF LOAD OPTIMIZATION 120
   7.1 De Facto Definition 120
   7.2 Alternative Definition 123
   7.3 Remarks 124
8. CONCLUSIONS AND FUTURE WORK 125
   8.1 Conclusions 125
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 3.1:</td>
<td>Summary of important equations</td>
<td>56</td>
</tr>
<tr>
<td>Table 6.1:</td>
<td>Summary of generalized equations</td>
<td>114</td>
</tr>
<tr>
<td>Table 6.2:</td>
<td>Steady-state values of speedup</td>
<td>115</td>
</tr>
<tr>
<td>Table 6.3:</td>
<td>Summary of variables</td>
<td>119</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

| Figure 2.1: | Overview of load balancing research | 7 |
| Figure 2.2: | Static load balancing research | 8 |
| Figure 2.3: | Dynamic load balancing algorithm research | 19 |
| Figure 2.4: | Dynamic load balancing theory | 21 |
| Figure 3.1: | Two processor Work/Exchange Model | 25 |
| Figure 3.2: | Simplified Work/Exchange Model | 27 |
| Figure 3.3: | A four cycle, case 1 program | 34 |
| Figure 3.4: | A generalized four cycle, case 2.1 program | 40 |
| Figure 3.5: | A four cycle, case 2.1 program | 41 |
| Figure 3.6: | A four cycle, case 2.2 program | 45 |
| Figure 3.7: | A case 1 program: balanced and imbalanced | 49 |
| Figure 3.8: | A case 2.1 program: balanced and imbalanced | 50 |
| Figure 3.9: | A case 2.2 program: balanced and imbalanced | 51 |
| Figure 3.10: | Speedup curves for $W_0 = 100$ and $W_1 = 100$ | 57 |
| Figure 3.11: | Speedup curves for $W_0 = 100$ and $W_1 = 75$ | 60 |
| Figure 3.12: | Speedup curves for $W_0 = 100$ and $W_1 = 50$ | 61 |
| Figure 3.13: | Speedup curves for $W_0 = 100$ and $W_1 = 25$ | 62 |
| Figure 3.14: | Speedup curves for $W_0 = 100$ and $W_1 = 0$ | 63 |
Figure 4.1: Equal time balanced and unbalanced trace; nonzero-cost bal-
ancer .................................................. 69
Figure 4.2: Comparison of execution times ................................. 71
Figure 4.3: Equal time balanced and unbalanced trace; zero-cost balancer 73
Figure 4.4: A faster unbalanced program using a half duplex channel . . 75
Figure 5.1: Comparison of execution times ................................. 84
Figure 6.1: Generalized Work/Exchange Model; first cycle, 3 processors . 91
Figure 6.2: Balanced counterpart to Figure 1 ................................. 107
Figure 6.3: Illustration of meanings of abstract variables ................... 108
Figure 6.4: Using a fixed reference for communication cost, $C$ ............ 109
DEDICATION

TO TERI,
my sweetheart, best friend, and wife.
Without her love, encouragement, and understanding, this work
might never have been completed.
1. INTRODUCTION

Consider the situation of ten people building a brick wall\(^1\). Suppose that the people are spread out in a line, each with their own pile of bricks, working on the fixed span closest to themselves. If each person starts at the appropriate position, works at the same rate as the others, has the same number of bricks as the others, and is able to successfully coordinate with his adjacent neighbors at the edge of his workspace, then the wall will be built in approximately a tenth of the time that it would take a single person to build the wall.

Now consider executing a parallel program on a multiprocessor computer with ten processors. We can think of the execution of a parallel program as analogous to building the brick wall. Within the program, we assume that there are ten subtasks that can be executed in parallel; each subtask is assigned to a different processor. Just as each bricklayer must occasionally coordinate with his neighbors, so must the parallel subtasks occasionally synchronize and pass information to other subtasks. If we assume that each subtask is of identical size to the other subtasks, each processor has the same clock speed, and interprocessor communication proceeds without network contention, then the parallel program will complete execution in approximately a tenth of the time that it will take a single processor to execute the program\(^2\).

\(^1\)Inspiration for this example comes from a similar example found in [19].
\(^2\)Of course, the communication overhead time will lessen the actual speedup.
In the latter example, since each subtask requires the same amount of time to execute as any other processor, we say that the problem has a balanced workload (or is **load balanced**). The wall building problem is also balanced since each worker has the same number of bricks and works at the same speed.

Now suppose that the subtasks in the multiprocessor example have different sizes. Then some subtasks will finish the parts of their work that can be done without prior synchronization before other subtasks finish similar work. The result is that some subtasks will have to wait for their slowest neighbor to finish before they can complete the synchronization and communication phase. Furthermore, that slowest neighbor may, in turn, be waiting on its other neighbor. Hence, the speed of program execution is driven by the execution time of the largest subtask. This is an example of an **unbalanced program**. Clearly, the unbalanced program takes longer to execute than the balanced counterpart.

In terms of our wall building example, this is similar to assigning some workers longer sections of the wall to build (and more bricks) than other workers. Let us assume that a worker must complete a given layer of bricks before proceeding to the next layer. Then the worker cannot work ahead of his neighbor at a boundary since the offset nature of brick positions will force some coordination. Therefore, if one bricklayer has less work to do than his neighbor, he must wait for his neighbor to complete a row before proceeding further. So by giving some workers more to do than other workers, the wall is going to take longer to build than it would have if each worker had been assigned the same amount of work.

The fallacy with the above examples are that they may cause one to assume that an unbalanced program can always be speeded up by rewriting the program to
have a more equitable distribution of workload. We began to wonder early on if there were other factors that influence the potential speedup that one would obtain from a redistribution of workload.

This led us to conduct a literature survey to determine the following:

1. What is load imbalance?

2. Are there any hidden factors besides workload that affect the performance obtainable from workload redistribution?

The results of the literature survey are discussed in detail in *Chapter 2: Literature Survey*. Surprisingly, we found that the former question has not been adequately defined in the literature and that the latter question has not been explored to any reasonable extent either.

We first decided to study the nature of load imbalance. Our approach was to define a mathematical model that could be used to model parallel programs distributed over two processors. This model is called the Work/Exchange Model and is presented in detail in *Chapter 3: The Work/Exchange Model*. Included are detailed derivations for equations giving exact values at any arbitrary cycle of execution for the following quantities:

1. the elapsed total time on a given processor,

2. the idle time on a given processor incurred during a given cycle,

3. the total ensemble time of both processors,

4. the total ensemble time of the perfectly balanced counterpart of the unbalanced program, and
the potential speedup that could theoretically be obtained by redistributing the workload of the unbalanced program into equal quantities of workload on each processor.

After having developed the equations of the two processor Work/Exchange Model and having pondered their significance, we realized that several of the commonly held beliefs expressed in the literature appear to be incorrect. In particular, we find that communication delay time does contribute a significant role in how workload should be redistributed to get minimal execution time. Furthermore, we find that there exist many cases of unbalanced programs that cannot be speeded up by redistribution of workload. This is significant because it means that further efforts on the part of applications programmers to speed up a distributed program are often futile. These and other issues dealing with load balancing myths are the topic of Chapter 4: Myths of Load Balancing.

Then in Chapter 5: An Enhancer for Dynamic Load Balancing, we take a more practical approach with the presentation of a useful application: an enhancer for dynamic load balancers. We assume that a program has a corresponding load balancer that is invoked at regular intervals. Essentially, an enhancer is a software tool that manages the load balancer. That is, the enhancer monitors the current state of a running parallel program and makes the following decisions regarding invocation of a load balancer by the application:

1. Supposing that the application is now requesting load balancing, should the load balancer be allowed to run at this time or is it more efficient to allow the program to run in its unbalanced state?
2. Should the period of invocation of the load balancer be modified?

The enhancer makes its decisions based on estimated running times of the program in both the balanced and unbalanced states. These estimates are obtained through application of the timing and speedup equations developed in conjunction with the Work/Exchange Model. Included are results of experiments run on a 128-node nCUBE 2 hypercube.

One of the problems with the model developed in Chapter 3 is its restriction to the two processor case. Fortunately, we were able to generalize the equations for an arbitrary number of processors. This work is summarized in Chapter 6: The Generalized Work/Exchange Model. Included in the chapter are derivations for the idle-time, elapsed total time, ensemble time, and speedup equations for arbitrary numbers of cycles and processors. A section verifying the equations by experiment is also included.

Then in Chapter 7: A Formal Definition of Load Optimization, we finally are able to use the results of the previous chapters to formally define the terms load imbalance and load optimization. We also show that the two concepts are distinct and that one should apply the latter term in conjunction with discussion of load balancers. We conclude with Chapter 8: Conclusions and Future Work.
2. LITERATURE SURVEY

In this chapter, we summarize the findings of an on-going literature survey. For an overview of categories of research conducted under the general category of load balancing, we refer the reader to Figure 2.1. As Figure 2.1 shows, the topic of load balancing can be subdivided into three basic categories:

1. static load balancing,
2. dynamic load balancing, and
3. real-time load balancing.

Actually, static load balancing is a special case of dynamic load balancing. However, from both historical and practical perspectives, static load balancing research is significantly different from the other subcategories of dynamic load balancing. Therefore, I have chosen to classify static load balancing as a distinct, basic category.

2.1 Static Load Balancing

In static load balancing, the even balancing of the load among processors is done prior to run time. This may be accomplished by an intelligent compiler or by the applications programmer. In fact, most programmers of parallel machines do static
load balancing during the design stage of a program, often without realizing that this is what they are doing.

Static load balancing can be divided into the subcategories of balancing independent tasks or balancing modules with data dependencies. In the literature, the term task is frequently used to denote a self-contained job that can be executed in any order with other tasks in the system. For this reason, static task balancing is often referred to as the task scheduling or the mapping problem. Some of the work in this area is found in [6, 15, 30, 31, 35, 36, 43]. Figure 2.2 summarizes previous work in static load balancing.
2.1.1 Independent tasks

In [6], the methods of hierarchical clustering, branch and bound, and diffusion are compared and discussed. Hierarchical clustering is the concept of grouping tasks together that have high communication requirements. The use of many job classes is used in [35] in a similar fashion, with the assumption of a heterogeneous, multiprocessor system.

In [15], migratory and non-migratory load sharing policies are compared. In non-migratory load sharing, jobs may not be transferred once they have started. In migratory load sharing, jobs may be transferred after starting. Results show that no performance is gained by allowing the transfer. In a related paper [31], the authors discuss the relationship of job scheduling and system partitioning. Another paper
that uses a migration protocol is [36]. In this paper, a load balancing algorithm for distributed tasks is introduced called the Drafting Algorithm.

A graphical, heuristic approach for task mapping is found in [43]. In this paper, the authors propose the use of a state space reduction technique called branch-and-bound-with-under-estimates. With this method, two underestimating functions are used to calculate task assignment costs based on minimal execution and communication values.

In [30], the authors take a dramatically different approach by claiming that no explicit balancing is required since large numbers of tasks and the central limit theorem will cause execution times to average to about the same load over all the processors. The areas of both static and dynamic load balancing of independent tasks strongly overlap with distributed systems research. For this reason, these areas are also referred to as distributed load balancing.

2.1.2 Data dependencies

The other subcategory of static load balancing research relates to balancing jobs having interdependencies of data. Work in this area includes the papers of [1, 4, 8, 29]. The first two sources take a graphical approach. In [29], a semi-automated partitioning algorithm is used for static balancing in shared memory systems. The simulated annealing method described in [1] is somewhat less applicable to static balancing since it focuses on approximating certain NP-complete problems through combinatorial optimizations.

In [4], a two-phase simulated annealing approach is used. Annealing is analogous to annealing in chemistry in that energy functions are used with the goal of
finding the base state of least energy. Least energy corresponds to mapping of sub-
tasks that results in minimal execution time. In the first phase, process assignment is 
established. In the second phase, communication links are established. Since depen-
dencies can be considered by communication link requirements, this approach may 
be extendable to dynamic load balancing as well.

Finally, an algorithm for optimal module to processor assignment for distributed, 
heterogeneous systems is given in [8]. Included in the paper is the development of the 
Computation Model that characterizes a distributed program with data dependencies. 
The problem with static load balancing is that it requires one to know a priori the 
run–time characteristics of the subtasks to be balanced. This often forces one to make 
timing estimates. As the estimates stray from observed behavior, the effectiveness of 
static balancers decreases. Another problem is that static balancers are not able to 
adjust to unexpected changes in load. Changes in load can occur from executing any 
branching form of code like IF–THEN–ELSE or WHILE–DO statements. In terms 
of weather forecasting programs, one can easily imagine the difficulties of trying 
to predict in advance when and where a computationally intensive event (like a 
thunderstorm) will arise. For this reason, many people prefer to study dynamic load 
balancing.

2.2 Dynamic Load Balancing

In dynamic load balancing, the balancing is performed at run–time in response 
to changing needs and conditions in the program. Like static load balancing, dynamic 
load balancing can be further divided into two distinct areas of research. I refer to 
these two areas as algorithm development and theoretical foundations.
2.2.1 Algorithms

The area of algorithm development concentrates on finding effective load balancing techniques and on the actual development of load balancers. This area abounds with a variety of papers in which authors expound their latest techniques. Usually included are simulations or implementation tests that show the superiority of one approach to another under the guise of restricted conditions or assumptions. Hence, a great deal of confusion exists as to which algorithms are good and why.

Currently, there are six basic types of dynamic load balancing heuristics:

- graphical methods,
- energy functions,
- nearest neighbor (or local) methods,
- central controller methods,
- scattered decomposition, and
- other methods

A general overview of these areas can be found in [18, 19]. For a more limited survey of dynamic load balancing algorithms from an "operating systems" point of view, we refer the reader to [21]. Note that the operating system point of view assumes the "embarrassingly parallel" case of no intertask dependencies. For a more complete summary, we refer the reader to Figure 2.3.

**Graphical methods** In graphical methods, either the work domain or the processor domain can be viewed in terms of a graph. Edges are generally used to
represent communication delays. Nodes represent resources and/or workloads. Grid partitioning is addressed in [19, 40]. One form of partitioning is called the **Recursive Bisection Method (RBM)**\(^1\). In this method, the domain is divided into two subparts with equal levels of workload. Then each subpart is divided again by recursive calls to the RBM algorithm. This continues until the number of subparts equals the number of processors. Early work in this method is found in [3, 18]. Later work includes [12] in which a scalable version with improved run-time complexity is developed, and [22] in which the “n4 partitioner” is presented. A hierarchical clustering and branch-and-bound method for independent tasks is presented in [6]. While the RBM method is very good at putting a domain into balance, it suffers from high communication cost and is difficult to implement.

Another graphical approach is presented in [45] in which both jobs and resources are allowed to migrate into “clusters”. One final graphical method is based on extending partial schedules [41]. Although this paper was written for real time load balancing, the technique could easily be adapted to unreal time situations. The last three sources [6, 41, 45] are applicable to independent tasks only.

**Energy functions** A second method is based on the minimizing of an energy or objective function of the form \( E = E_1 + E_2 \) where \( E_1 \) is a measure of imbalance and \( E_2 \) is a measure of total communication costs [18]. Two approaches exist for minimizing \( E \):

- Simulated Annealing
- Neural Networks

\(^1\) Also known as the **Orthogonal Recursive Bisection (ORB) method**.
Simulated Annealing (SA) has a physics analogy with $E$, $E_1$, and $E_2$ corresponding to potential energy of a system, a repulsive hard core, and an attractive long range potential, respectively. Simulated annealing corresponds to finding the ground state of the system. While it can be efficiently implemented and results in arbitrarily good solutions, it is only an approximate solution with an execution time that is difficult to estimate.

The second energy function approach is the Neural Network (NN) and was originally introduced by Hopfield and Tank. This method suffers from local minima traps. Also, task execution times must be known in advance.

**Nearest neighbors method** The third class of dynamic load balancing heuristics is called the nearest neighbors or local scheduling method. With this technique, each process keeps track of its own workload and the workload of its nearest neighbors. For broadcast buses, the *Random*, *Threshold*, and *Shortest* load sharing policies presented in [13] has been applied. Each policy consists of a transfer policy and a threshold policy. The threshold policies are based on local workload only; no communication costs are incurred in their calculations. The transfer policies have the same names as their load sharing corresponding policy names. In the Random Transfer policy, a node is selected at random and sent any extra load. In the Threshold policy, a node is also picked at random and probed to determine its state of workload. This will continue up to a constant number of times until a node is found that can take the extra load. If none is found, the originating node keeps the extra load. The third policy, Shortest, pools a fixed number of randomly selected processors and sends extra workload to the least loaded node if it isn’t busy; otherwise, it keeps the load.
Another local algorithm is the *Migratory and Non-Migratory Load Sharing* algorithms presented in [15]. Basically, this paper examines the benefits of moving jobs that have already been started. It is concluded in the paper that little gain results from moving jobs that have already started.

In [24], three algorithms are compared:

- **Random Selection of Remote Host (RSRH),**
- **Probabilistic Job Dispatching (PJD),** and
- **Probabilistic Dispatching Using Queue Length as Balancing Factor (PDQLBF)**

RSRH compares its load with the average of all other nodes. It then sends any extra load to a random location. PJD sends extra load to other processors in proportion to their load relative to the sender. PDQLBF is the same as PJD except that it adjusts its estimates of another node’s workload by the amount of work it has just sent.

In [44], three dynamic load sharing algorithms are presented and compared through simulation. In the first algorithm, a processor compares its load to the load of the least busy processor. In the second, a processor compares its load with all other processors and sends zero, one or two jobs to each processor depending on results of the comparison. The third algorithm is similar to the second, except that a time interval is started during which no further jobs may be sent. This is to prevent too many jobs that are in transit from being left out of load estimations. Results presented say that all three algorithms perform poorly when the system is heavily loaded or when they are invoked too often. Also, out-of-date information causes the algorithms to perform poorly.
Perhaps the best known local algorithms are the \emph{Gradient Method (GM)} and \emph{Contracting Within a Neighborhood (CWN)}. Both are presented in \cite{26, 27}. Additional sources for the Gradient Method are \cite{28, 32}. GM is the more elaborate of the two and uses parameters called the low-water-mark and high-water-mark. Furthermore, a processor may have a workload state of idle, neutral, or abundant. Each processor keeps a proximity count that essentially is an estimate of the distance to the nearest idle node. Idle nodes have a proximity count of zero; otherwise, the count is one plus the minimum proximity of all nearest neighbors. This information is then used to decide whether or not to send extra load to an idle processor. This is an example of keeping global information using only local communications.

In CWN, all jobs \textit{must} be sent to a neighbor when first created. Associated with each job is a hop count. Every job must travel a minimum number of hops, called the horizon, but must not be passed more than a maximum number of hops, called the radius. Even though the CWN is simpler than GM, the claim is that it performs better.

Additional nearest neighbor algorithms that are written from a queuing theory perspective, which assumes all tasks are independent, include the \emph{Diffusion Scheme} and \emph{Dimension Exchange} presented in \cite{9} and the \emph{Hypercube Nearest Neighbor, Nearest Balancing Strategy}, and \emph{Locally Averaging Neighbor (LAL)}\footnote{Yes, the second L in LAL is correct. Perhaps the authors' of \cite{23} wanted to avoid confusion with Local Area Networks (LAN)} algorithms presented in \cite{23}.

\textbf{Central controller} A fourth method of developing dynamic load balancing algorithms uses a central controller to determine when a job should be moved. One
such central controlling method is presented in [51]. This approach assumes the use of a local area network. Information is collected from the network regarding the load distribution of the system by a central controller. Each processor places its own load status onto the network by storing it in an extra field added to every network packet. The information is then placed into a Balancing Control Center (BCC) by the central controller and is then distributed to every node via a balancing vector. This method is only good for low and moderate traffic situations. An additional algorithm is presented in [5] that considers heterogeneous systems.

**Scattered decomposition** The fifth type of dynamic load balancing algorithm falls under the category of scattered decomposition. In scattered decomposition, the portion of the domain that a given processor is responsible for is scattered over the underlying data domain. This type of domain division has high associated communication cost but is highly effective for problems having no nearest neighbor structure. Problems of this type include matrix problems, Monte Carlo, and ray tracing algorithms. Since rapidly varying or dynamic problems will tend to accumulate work over an area covered by many processors, this method is considered to be "rigorously optimal" for these kinds of problems. For more information on scattered decomposition and its application to load balancing, we refer the reader to [18, 39].

**Other methods** There are some types of dynamic load balancing algorithms that do not fit nicely into the five previously discussed types. The first is from [14] and covers both sender-initiated and receiver-initiated algorithms. In the sender-initiated algorithms, a processor that has too much work to do attempts to find an idle processor that can share some of the load. In the receiver-initiated algorithms, an idle
processor attempts to find a processor with too much work. The authors of the paper concluded that the sender-initiated algorithms are better for light and moderate load situations since less communication costs are required. However, receiver-initiated algorithms appear to be better for heavy load situations.

In [34], an algorithm for preemptive scheduling of $n$ independent jobs of known time is presented. In the paper it is claimed that the algorithm has a complexity cost of $O(\log n + \log^3 m)$ time where $m$ is the number of uniform machines.

In [2], a load balancing algorithm for a broadcast bus is presented. This algorithm is called GAMMON and has been implemented to balance load between local area network connected SUN work stations. GAMMON, which stands for Global Allocation from Maximum to Minimum in cONstant time, was found to be good in low and moderate traffic situations and has a time complexity independent of the number of workstations.

Finally, there is the self-scheduling approach in which processors take and execute work on their own initiative. This approach is generally intended for shared memory architectures but is also applicable to message-passing architectures. For a brief discussion of self-scheduling, see [18]. Related work can also be found in [17, 46].

2.2.2 Theoretical foundations

The area of theoretical foundations is a fairly recent development that was begun in response to the confusion that abounds on the algorithm development side of dynamic load balancing. Under the heading of theory, researchers are studying the concept of load balancing itself. We refer the reader to Figure 2.4 for a summary of work done to date. Work includes analysis of ideal situations from which actual
balancers can be compared. The area of theory can be further divided into two subareas. Henceforth, I shall refer to these areas as analysis and approaches.

**Analysis** In the area of analysis, researchers are primarily interested in the definition, types, and contributing factors of dynamic load balancing. In [7], an excellent start has been made at determining the types of load imbalance and in developing timing equations for each of these types. Basically, the paper defines a type of load imbalance according to the kind of synchronization used within a program and according to the changing nature of the imbalance. Synchronization can be full, partial, or none. Furthermore, the amount of imbalance on a given processor can be fixed or can vary with time. Fixed imbalance and varying imbalance are often referred to as deterministic and nondeterministic load imbalance, respectively. Two primary drawbacks to the paper are that:

- no overlap between computation and synchronization is allowed,
- equations are only applicable to large numbers of processors

This paper is one of my primary sources and its drawbacks in part, have inspired much of my current research interests.

Additional work in the analysis area include [16] and [37]. In [16], the trade-off between speedup and efficiency in the context of independent tasks is presented. In [37], the authors examine the benefits of dynamically remapping workloads between phases of computation. During the course of each phase, imbalance is assumed to be varying in a gradual fashion. However, the state of imbalance is assumed to drastically change during the transition from one phase to the next.

---

3This is sometimes referred to as quasi-static imbalance.
Figure 2.3: Dynamic load balancing algorithm research
From the independent task viewpoint, [36] considers the tradeoffs of utilization and communication cost. Finally, a discussion of how to partition a set of tasks in order to balance communication costs and imbalance can be found in [40].

Theoretical approaches  One problem with trying to develop a theoretical framework for dynamic load balancing is that it encompasses too broad an area. Therefore, instead of trying to capture the "big picture" in their research, most researchers focus on a smaller area of research that is in line with some theme. Some of these themes are mentioned below.

One such theme has already been mentioned—the remapping of workloads between phases of relatively static imbalance [37]. Other themes focus on the type of synchronization that takes place in a program. For example, the area of static load balancing, task balancing, and distributed balancing can all be categorized under the theme of no synchronization. Most work in dynamic load balancing, however, lies under the category of full synchronization.

Another theme considers the consequences of trying to balance too often. When this happens, the extra communication costs slow a program down. This is a specific example for the need of throttling discussed in [10]. In response to the over balancing problem, a load balancer enhancer was proposed by [38]. In this work, a load balancer is invoked only when needed. The decision as to when to invoke the balancer is based on the detection of the first local minimum in a degradation function. Independent work has also been done in this area by Wikstrom and others in [49]. However, in our work, the decision to balance is based on generalizations of static timing equations.

A somewhat different approach to reducing communication penalties and idle
Dynamic Load Balancing) Theory

Analysis
- Equations for large P
  Types of imbalance
- Remapping / Phases
  performance vs. communication
- tradeoffs: speed vs. efficiency
- tasks: utilization vs. communication
- partitioning to balance
  communication and imbalance

Approaches
- throttling
- quasi-static
- enhancers
  first local minima
- task duplication
  Transparent Process Cloning
- cube partitioners

Figure 2.4: Dynamic load balancing theory

time in a processor is to duplicate tasks on several processors [20] so that duplicates of data are immediately available on several processors at once. The name given to the technique in [20] is Transparent Process Cloning (TPC). Finally, the problem of scheduling groups of processors to tasks is discussed in [25] in which a hypercube partitioner is presented.

2.3 Real-Time Load Balancing

The third main branch of load balancing is real-time load balancing. In real-time balancing, balancing is usually an added performance feature which is optional. The main emphasis of research in this area is determining how to get better performance results with a bounded amount of effort. The number of papers in this area is enormous. Since this area is not directly germane to my area of research, no attempt was made at a literature survey in this category.
However, one paper [41] in this area can be adapted into a dynamic load balancing technique. In this paper, partial schedules of future tasks that can be scheduled within the real-time constraints are developed. These partial schedules can be formed in one of two ways: First, all remaining tasks can be considered, or second, only a subset is considered. While the former is more likely to find a feasible schedule, it is at the expense of high computational cost. Therefore, the paper concludes that the second approach is better. This appears to be another form of the branch–and–bound graphical approach discussed earlier.
3. THE WORK/EXCHANGE MODEL

3.1 Introduction

In Chapter 2 we find that the term load imbalance is not formally defined in the literature. In fact, most people naively assume load imbalance measures only the degree of inequitable work assignment to different processors. Only a few papers recognize that load distribution and communication issues are interrelated. Therefore, we feel that it is appropriate, at this time, to introduce a theoretical model from which we are able to address the issues listed above. The model we introduce in this chapter is called the Work/Exchange Model.

In Section 3.2.2, we define the Work/Exchange Model for two processors. In this model, we are able to exactly characterize the behavior of a time-stepped program distributed on two processors. We assume the program has the form of a repeating cycle of two phases. The first phase entails performing useful work. The second phase is for communication exchange. Since the waiting for an incoming message causes idle time to accrue, this phase consists totally of idle waiting time. The amount of work on a given processor is held constant over all cycles. We refer to the type of load imbalance that results under this restriction as deterministic load imbalance. One additional restriction is that all communication delay times be constant.

Then in Section 3.3, we develop the exact timing and idle-time equations for
arbitrary programs that meet the requirements of the Work/Exchange Model. In Section 3.4, we establish the validity of these equations. Next, in Section 3.5, we consider a seemingly best-case situation of a program being redistributed into a perfectly balanced distribution. Then having established timing equations for an unbalanced program and its balanced counterpart, we are able to derive speedup equations in Section 3.6 that represent the maximum speedup that one could obtain from a program by redistributing the work in equal amounts. However, as we alluded to above, myths exist regarding speedup; this occurs in part due to the ignoring of communication costs and overlap with useful work. In Section 3.7 we display many of the interesting and unexpected results that we find in the speedup equations of Section 3.6. Finally, we make some concluding remarks in Section 3.8.

3.2 The Work/Exchange Model

3.2.1 The original model

Our first approach to studying load imbalance is to develop the two processor model pictured in Figure 3.1. In the model, we envision two processors in a cycle of performing work \(N_i\) followed by an exchange of messages. The message exchange consists of a nonblocking send \(S\) followed by a blocking read \(R\). Each send is composed of a time setting up and issuing the send, and the time for the actual transmission from the sender\(^1\). Also, each read is divided into the time of issuing and preparing for a read command \(I\), and the actual receiving time of the message.

\(^1\)Actual transmission time is measured from the time the first byte leaves the sender up to when the first byte arrives at the receiver. Additional network transmission time is not included.
off the network \(R - I\). There may also exist additional wait time \(W_i\) between completion of the read setup and the start of the actual receiving of data. A message transfer between processors is represented by a directed arc from one processor line to another. Associated with each message transfer is a communications cost \(C\). In general, all of the above values can differ for each cycle and each processor. However, we chose to restrict all values except work time and wait time as constant for both processors and for all cycles. Within each processor, work time is also held constant. Wait time is determined by the other values.

After working out several examples using the model, we began to suspect that
the following situations might hold:

- for some cases with uneven distribution of work, the total execution time is still minimized,

- the timing characteristics during the first few cycles often differs dramatically from successive cycles,

- for all combination of values, a steady state condition involving one or more cycles eventually develops, and

- when model restrictions are relaxed to allow differing values of communication transit time (to model network contention, for example), some cases with uneven distribution actually outperform their balanced counterparts.

where the balanced counterpart mentioned above is obtained by redistributing the work into equivalent amounts on each processor. This counterpart represents the perfectly balanced case and is discussed in detail in Section 3.5.

Having established the model, the next goal is to develop equations reflecting the actual execution time for \( j \) cycles in terms of the variables listed in Figure 3.1. Unfortunately, we were unable to derive these equations with the model in the aforementioned form. While we believe that these equations are derivable, we believe that the task is very difficult due to the vast number of cases that arise with the use of so many variables.

3.2.2 Simplified model

So in order to make the derivations easier, we choose to simplify the model to the form shown in Figure 3.2. Now instead of six variables per processor, there
are only two. One variable, $I_i$, represents the idle time that processor $i$ suffers while waiting for a message to arrive. The other variable, $W_i$, represents a conglomeration of the other five original variables.

To prevent the component of the graph representing the idle time for a given cycle from separating non-idle components of a cycle, we chose to redefine the cycle boundary such that the next cycle begins with the time to actually read a message. So the first component represents the reading of a message, 'number crunching', and the send of a message. The second component represents the idle wait time for the next message to arrive.

Now we describe the model in greater detail. In the context of the model, we
can describe an arbitrary, iterative looped program meeting the requirements below. First, the program is distributed on exactly two processors. Second, the distributed program consists of a sequence of tasks on each processor such that after the execution of a task on a processor, a message is sent to and read from the neighboring processor\(^2\). Third, on a given processor, the execution time of each task is identical from one cycle to the next. And finally, the time required for every message to transit the network is constant\(^3\). Programs that meet the above requirements include certain finite difference problems and the climate modeling problems. Additional problems meeting these requirements are discussed in [19]. Figure 3.2 illustrates the timing diagram for the first cycle of a program in the Work/Exchange Model. We define a cycle as the tuple \(\{i, T_i(j), T_i(j + 1), W_i, I_i(j)\}\) where \(T_i(j)\) and \(T_i(j + 1)\) are the starting time and completion time of the \(j^{th}\) iteration running on processor \(i\) (PE\(_i\)), respectively. We define an iteration as a two-phase process consisting of a non-idle work phase taking time \(W_i\) followed by a message exchange phase. The message exchange phase often entails idle waiting time and uses \(I_i(j)\) time.

In the model, we envision two processors in a cycle of performing work \(W_i\) followed by an exchange of messages. The message exchange consists of a nonblocking send followed by a blocking read. Sometimes, a processor completes execution of a task prior to receiving the expected, incoming message. When this occurs, the processor accrues idle wait time \(I_i(j) > 0\).

We use \(i\) to denote the processor and \(j\) to denote the cycle. The communication

\(^2\)It is this cyclic behavior of performing a task and then exchanging messages that led to the name “Work/Exchange Model,” for the model introduced below.

\(^3\)Since we are using only two processors, this is trivially true. It is also accurate for message-passing systems with cut-through routing, such as nCUBE and Intel hypercubes.
transit delay time, $C \geq 0$, is measured from the time the first byte leaves the sender up to when the first byte arrives at the receiver. A message transfer between processors is represented by a sloped line from one processor time line to the other. Wait time varies and is a function of all previous cycles, the current work time, and the communication time. Under the simplified model, the equations are derivable. A detailed derivation of each equation and examples of various cases are presented next.

### 3.3 Derivation of Equations

Before we derive the actual equations, an explanation of variables and notation is needed. For a processor $i$ (PE$_{i}$), the time required during cycle $j$ for idle time is given by $I_{i}(j)$. As the model contains two processors, $i$ may have the value of 0 or 1. Also $j$ may be any positive integer. Similarly, the time to do useful work on processor $i$ during the $j^{th}$ cycle for non-idle time is $W_{i}(j)$. Since the non-idle time is restricted to be constant for a given processor, we simplify the variable representation to $W_{i}$. To simplify the model further, we assume without loss of generality that $W_{0} \geq W_{1}$. Since the difference between the non-idle times is frequently used in the equations, we assign this nonnegative value to the constant $D$ where $D = W_{0} - W_{1}$. Notice this implies that $W_{0} = W_{1} + D$ and $W_{1} = W_{0} - D$.

The communication network transit time is given by $C$ and is held constant in the model. The value $T_{i}(j)$ represents the accumulated execution time of a program on PE$_{i}$ for the first $j$ cycles. Since a program does not complete until the slowest processor completes, we also introduce the accumulated processor time, $T(j) = \max_{i=0,1} T_{i}(j)$. 

Notice that the subscript denotes the processor number. The value in parenthesis denotes the current cycle or last cycle for the accumulating variables of $T(j)$ and $T_i(j)$. We use the subscript value of $A$ for the analogous values of the perfectly balanced or averaged case. Since behavior of the perfectly balanced case will be identical on both processors, the processor number need not be distinguished.

Now, the equations can be derived. First, consider the wait time, $I_1(1)$, on $\text{PE}_1$ for the first cycle. This value is the span of time between the completion of non-idle time on $\text{PE}_1$ and the arrival of a message from $\text{PE}_0$.

$$I_1(1) = W_0 + C - W_1$$
$$= D + C$$ \hfill (3.1)

Before we accept this value, we must guarantee that the wait time is nonnegative since the completion of the wait time establishes the starting time of the next cycle. Were we to allow negative wait times, we would be allowing overlap of non-idle components of different cycles on the same PE. Clearly, this is physically impossible. Since both $D$ and $C$ are guaranteed to be nonnegative, $I_1(1)$ must also be nonnegative.

The wait time on $\text{PE}_0$ for the first cycle, $I_0(1)$, is the amount of time between the completion of the non-idle time on $\text{PE}_0$ and the arrival of a message from $\text{PE}_1$. Since $W_1 \leq W_0$, this message could arrive at $\text{PE}_0$ before it is needed. Therefore, $I_0(1)$ is given by:

$$I_0(1) = \max(W_1 + C - W_0, 0)$$
$$= \max(C - D, 0)$$
$$= \begin{cases} 
C - D & \text{when } C > D \ (\text{Case 1}) \\
0 & \text{when } C \leq D \ (\text{Case 2}) 
\end{cases}$$ \hfill (3.2)
We now continue discussion on a case-by-case basis.

3.3.1 Case 1: $C > D$

Having calculated the idle times for the first cycle, we can now calculate the accumulated total times for each PE. In general, the accumulated time through $j$ cycles on PE$_i$, $T_i(j)$, will be the accumulated time through $j - 1$ cycles, $T_i(j - 1)$, plus the non-idle time for this cycle, $W_i$, plus the idle time for this cycle, $I_i(j)$. That is, the general equation for accumulated time on processor $i$ through $j$ cycles is:

$$T_i(j) = T_i(j - 1) + W_i + I_i(j)$$

where $T_i(j - 1)$ is defined to have the value of zero when $j = 1$. Hence, for cycle 1, the accumulated time equations are given by:

$$T_{0}(1) = W_0 + C - D \quad (3.3)$$

$$T_{1}(1) = W_1 + C + D = W_0 + C \quad (3.4)$$

Since $T_{1}(1) \geq T_{0}(1)$, the accumulated program time $T(1) = T_{1}(1)$.

Next, we derive the idle time equations of the second cycle:

$$I_{1}(2) = T_{0}(1) + W_0 + C - (T_{1}(1) + W_1)$$
$$= (W_0 + C - D) + W_0 + C - (W_0 + C) - (W_0 - D)$$
$$= C \quad (3.5)$$
\[ I_0(2) = T_1(1) + W_1 + C - (T_0(1) + W_0) \]
\[ = (W_0 + C) + (W_0 - D) + C - (W_0 + C - D) - W_0 \]
\[ = C \quad (3.6) \]

The derivations of the accumulated time equations of the second cycle are as follows:

\[ T_0(2) = T_0(1) + W_0 + I_0(2) \]
\[ = (W_0 + C - D) + W_0 + C \]
\[ = 2(W_0 + C) - D \quad (3.7) \]

\[ T_1(2) = T_1(2) + W_1 + I_1(2) \]
\[ = (W_0 + C) + (W_0 - D) + C \]
\[ = 2(W_0 + C) - D \quad (3.8) \]

Notice that \( T_0(2) = T_1(2) \). Since both PE's finish at the same time in the second cycle, we know that all successive cycles will repeat the pattern established in the first two cycles.

Therefore, the idle wait time equations are as follows:

\[ I_0(j) = C - D \quad \text{where } j = 1, 3, 5, \ldots \quad (3.9) \]
\[ I_0(j) = C \quad \text{where } j = 2, 4, 6, \ldots \quad (3.10) \]

\[ I_1(j) = D + C \quad \text{where } j = 1, 3, 5, \ldots \quad (3.11) \]
\[ I_1(j) = C \quad \text{where } j = 2, 4, 6, \ldots \quad (3.12) \]

In addition, we conclude that for any even cycle, the accumulated execution time
equations are:

\[ T_0(j) = \frac{j}{2} \times T_0(2) \]
\[ = j(W_0 + C) - \frac{j}{2}D \quad \text{where } j = 2, 4, 6, \ldots \quad (3.13) \]

\[ T_1(j) = \frac{j}{2} \times T_1(2) \]
\[ = j(W_0 + C) - \frac{j}{2}D \quad \text{where } j = 2, 4, 6, \ldots \quad (3.14) \]

For any odd cycle, we conclude the accumulated execution time equations are:

\[ T_0(j) = T_0(j - 1) + T_0(1) \]
\[ = (j - 1)(W_0 + C) - \frac{j - 1}{2}D + (W_0 + C - D) \]
\[ = j(W_0 + C) - \frac{(j + 1)}{2}D \quad \text{where } j = 1, 3, 5, \ldots \quad (3.15) \]

\[ T_1(j) = T_1(j - 1) + T_1(1) \]
\[ = (j - 1)(W_0 + C) - \frac{j - 1}{2}D + (W_0 + C) \]
\[ = j(W_0 + C) - \frac{j - 1}{2}D \quad \text{where } j = 1, 3, 5, \ldots \quad (3.16) \]

We condense the previous equations by using the 'floor' and 'ceiling' functions as follows:

\[ T_0(j) = j(W_0 + C) - \left\lceil \frac{j}{2} \right\rceil D \quad \text{for } j = 1, 2, 3, \ldots \quad (3.17) \]

\[ T_1(j) = j(W_0 + C) - \left\lfloor \frac{j}{2} \right\rfloor D \quad \text{for } j = 1, 2, 3, \ldots \quad (3.18) \]
Since $T_1(j) \geq T_0(j)$ for all $j$, $T(j) = T_1(j)$. Figure 3.3 illustrates an example of a four cycle, Case 1 program using the specific values of $W_0 = 20$, $W_1 = 12$, and $C = 15$.

**CASE 1) C > D**

- $W_0 = 20$
- $W_1 = 12$ (D = 8)
- $C = 15$

![Diagram](image)

**Figure 3.3: A four cycle, case 1 program**
3.3.2 Case 2.1: $\frac{D}{2} < C \leq D$

Recall that the derivation of Equation 3.2 forces a case–by–case derivation of the equations. Case 1 deals with instances of the model where $C > D$. Case 2 deals with model instances where $0 \leq C \leq D$. However, Case 2 divides into two subcases — Case 2.1: $\frac{D}{2} < C \leq D$; and Case 2.2: $0 \leq C \leq \frac{D}{2}$. In this section we examine the reason for the subcases and derive the equations of Subcase 2.1.

From the previous section we know that the following equations hold when $C \leq D$:

\[ I_1(1) = D + C \quad (3.19) \]

\[ I_0(1) = 0 \quad \text{when } C \leq D \quad (3.20) \]

Hence, the derivation for the accumulated time for the first cycle is as follows:

\[ T_0(1) = W_0 + I_0(1) \]
\[ = W_0 \quad (3.21) \]

\[ T_1(1) = W_1 + I_1(1) \]
\[ = (W_0 - D) + (D + C) \]
\[ = W_0 + C \quad (3.22) \]

So for the first cycle, $T(1) = T_1(1)$ and PE$_1$ finishes $C$ time units later than PE$_0$.

So for the second cycle, the idle wait time equations can be derived as follows:

\[ I_1(2) = T_0(1) + W_0 + C - (T_1(1) + W_1) \]
\[ I_0(2) = \max(T_1(1) + W_1 + C - (T_0(1) + W_0), 0) \]
\[ = \max((W_0 + C) + (W_0 - D) + C - W_0 - W_0, 0) \]
\[ = \begin{cases} 
2C - D & \text{if } \frac{D}{2} < C \leq D \quad (\text{Case 2.1}) \\
0 & \text{if } 0 \leq C \leq \frac{D}{2} \quad (\text{Case 2.2}) 
\end{cases} \tag{3.24} \]

From the derivation of \( I_0(2) \) we can see why Case 2 is broken into two subcases. As with all idle time equation derivations, idle time is never allowed to be negative.

So continuing with the derivations for the Case 2.1 in which \( \frac{D}{2} < C \leq D \), the equations for the accumulated time through the second cycle are derived as follows:

\[ T_0(2) = T_0(1) + W_0 + I_0(2) \]
\[ = W_0 + W_0 + 2C - D \]
\[ = 2(W_0 + C) - D \tag{3.25} \]

\[ T_1(2) = T_1(1) + W_1 + I_1(2) \]
\[ = (W_0 + C) + (W_0 - D) + D \]
\[ = 2W_0 + C \tag{3.26} \]

Notice that PE\(_1\) finishes after PE\(_0\) and \( T(2) = T_1(2) \) since the quantity \( C - D \) is less than or equal to zero. In addition, it is important to note that PE\(_1\) finishes exactly \( D - C \) time units later than PE\(_0\).
For the third cycle the idle wait time equations can be derived as follows:

\[ I_1(3) = T_0(2) + W_0 + C - (T_1(2) + W_1) \]
\[ = [2(W_0 + C) - D] + W_0 + C - (2W_0 + C) - (W_0 - D) \]
\[ = 2W_0 + 2C - D + W_0 + C - 2W_0 - C - W_0 + D \]
\[ = 2C \]

(3.27)

\[ I_0(3) = T_1(2) + W_1 + C - (T_0(2) + W_0) \]
\[ = (2W_0 + C) + (W_0 - D) + C - [2(W_0 + C) - D] - W_0 \]
\[ = 0 \]

(3.28)

Furthermore, the third cycle accumulated time equations are derivable as follows:

\[ T_0(3) = T_0(2) + W_0 + I_0(3) \]
\[ = [2(W_0 + C) - D] + W_0 + 0 \]
\[ = 3W_0 + 2C - D \]

(3.29)

\[ T_1(3) = T_1(2) + W_1 + I_1(3) \]
\[ = (2W_0 + C) + (W_0 - D) + 2C \]
\[ = 3(W_0 + C) - D \]

(3.30)

With cycle 3 we observe a repeat of the behavior of cycle 1. That is, both cycles end with \( PE_1 \) completing \( C \) time units after \( PE_0 \). Therefore, all cycles after cycle 2 exhibit the same behavior as established in the first two cycles.

So in terms of the accumulated execution time, we have sufficient information to derive the general equations without additional derivations. However, the information
needed to derive the general equations for idle wait time is still lacking. Therefore, we derive the idle time equations one additional step as follows:

\[ I_1(4) = T_0(3) + W_0 + C - (T_1(3) + W_1) \]
\[ = (3W_0 + 2C - D) + W_0 + C - [3(W_0 + C) - D] - (W_0 - D) \]
\[ = 3W_0 + 2C - D + W_0 + C - 3W_0 - 3C + D - W_0 + D \]
\[ = D \] (3.31)

\[ I_0(4) = T_1(3) + W_1 + C - (T_0(3) + W_0) \]
\[ = [3(W_0 + C) - D] + (W_0 - D) + C - (3W_0 + 2C - D) - W_0 \]
\[ = 3W_0 + 3C - D + W_0 - D + C - 3W_0 - 2C + D - W_0 \]
\[ = 2C - D \] (3.32)

Notice that in Equation 3.32, the value \( I_0(4) = 2C - D \) is nonnegative since \( \frac{D}{2} < C \leq D \).

With inclusion of the last set of equations, there exists sufficient information for deriving all the general equations for Case 2.1. However, for the sake of completeness and as a check against predicted results, we complete the derivations for cycle 4 by deriving the accumulated time equations:

\[ T_0(4) = T_0(3) + W_0 + I_0(4) \]
\[ = (3W_0 + 2C - D) + W_0 + (2C - D) \]
\[ = 4W_0 + 4C - 2D \] (3.33)

\[ T_1(4) = T_1(3) + W_1 + I_1(4) \]
\[ \begin{align*}
\[3W_0 + C) - D\] + (W_0 - D) + D \\
= 4W_0 + 3C - D
\end{align*} \tag{3.34} \]

As with cycle 2, in cycle 4 PE1 finishes \(D - C\) time units later than PE0 and behaves as expected.

As the behavior of Case 2.1 is somewhat harder to visualize than Case 1, we refer the reader to Figure 3.4 to examine a general case, four cycle example. Included are values for the accumulated execution time, idle times, non-idle times, and time differences between completions of the two PE's for each of the four shown cycles. Figure 3.4 is self-explanatory except to note that the quantity \(C - D \leq 0\) since \(\frac{D}{2} < C \leq D\) for Case 2.1.

From an examination of Figure 3.4, it is readily apparent that the following general equations are correct for the case of \(\frac{D}{2} < C \leq D\):

\[
I_0(j) = \begin{cases} 
0 & \text{for } j = 1, 3, 5, \ldots \\
2C - D & \text{for } j = 2, 4, 6, \ldots 
\end{cases} \tag{3.35}
\]

\[
I_1(j) = \begin{cases} 
D + C & \text{for } j = 1 \\
D & \text{for } j = 2, 4, 6, \ldots \\
2C & \text{for } j = 3, 5, 7, \ldots 
\end{cases} \tag{3.36}
\]

\[
T_0(j) = \begin{cases} 
W_0 + \frac{j-1}{2} [2W_0 + 2C - D] & \text{for } j = 1, 3, 5, \ldots \\
\frac{i}{2} [2W_0 + 2C - D] & \text{for } j = 2, 4, 6, \ldots 
\end{cases} \tag{3.37}
\]

\[
T_1(j) = \begin{cases} 
W_0 + C + \frac{j-1}{2} [2W_0 + 2C - D] & \text{for } j = 1, 3, 5, \ldots \\
D - C + \frac{i}{2} [2W_0 + 2C - D] & \text{for } j = 2, 4, 6, \ldots 
\end{cases} \tag{3.38}
\]

\[
T(j) = T_1(j) \text{ for } j = 1, 2, 3, \ldots \tag{3.39}
\]
CASE 2.1) \( D/2 < C \leq D \)

As with Case 1, we observe that the behavior of the first cycle varies substantially from accumulated times of latter cycles.

Figure 3.5 contains a four cycle example of Case 2.1 using the specific values of \( W_0 = 20 \), \( W_1 = 12 \), and \( C = 6 \).
CASE 2.1) $D/2 < C \leq D$

$W0 = 20 \ (D = 8)$
$W1 = 12$
$C = 6$

**Figure 3.5:** A four cycle, case 2.1 program
3.3.3 Case 2.2: $0 \leq C \leq \frac{D}{2}$

In this section we consider the second of the two subcases of Case 2. That is, we consider the equation derivations under the restriction that $0 \leq C \leq \frac{D}{2}$. From previous sections, we already know that for this case, the following equations hold:

\begin{align*}
I_0(1) &= 0 \\
I_1(1) &= C + D \\
T_0(1) &= W_0 \\
T_1(1) &= W_0 + C
\end{align*}

\begin{align*}
I_0(2) &= 0 \\
I_1(2) &= D
\end{align*}

From these we derive the accumulated time equations for cycle 2:

\begin{align*}
T_0(2) &= T_0(1) + W_0 + I_0(2) \\
&= W_0 + W_0 + 0 \\
&= 2W_0
\end{align*}

\begin{align*}
T_1(2) &= T_1(1) + W_1 + I_1(2) \\
&= (W_0 + C) + (W_0 - D) + D \\
&= 2W_0 + C
\end{align*}

Next, we derive the idle time equations for the third cycle:

\begin{align*}
I_1(3) &= T_0(2) + W_0 + C - (T_1(2) + W_1)
\end{align*}
\[ I_0(3) = \max(T_1(2) + W_1 + C - (T_0(2) + W_0), 0) \]
\[ = \max((2W_0 + C) + (W_0 - D) + C - 2W_0 - W_0, 0) \]
\[ = \max(2C - D, 0) \]
\[ = 0 \] (3.49)

\( I_0(3) = 0 \) because \( 2C - D \) is less than or equal to zero for this subcase.

Finally, we derive the accumulated time equations for the third cycle:

\[ T_0(3) = T_0(2) + W_0 + I_0(3) \]
\[ = 2W_0 + W_0 + 0 \]
\[ = 3W_0 \] (3.50)

\[ T_1(3) = T_1(3) + W_1 + I_1(3) \]
\[ = (2W_0 + C) + (W_0 - D) + D \]
\[ = 3W_0 + C \] (3.51)

For this case, a clear pattern exists. We conclude from observation that the following general equations are true:

\[ I_0(j) = 0 \] (3.52)  
\[ I_1(j) = \begin{cases} 
D + C & \text{for } j = 1 \\
D & \text{for } j = 2, 3, 4, \ldots 
\end{cases} \] (3.53)
\[ T_0(j) = jW_0 \] (3.54)
\[ T_1(j) = jW_0 + C \] (3.55)
\[ T(j) = T_1(j) \] (3.56)

Figure 3.6 contains a 4-cycle example of Case 2.2 using the specific values of \( W_0 = 20 \), \( W_1 = 12 \), and \( C = 3 \).

### 3.4 Verification of Equations

Having derived in Section 3.3 the timing equations for the three cases of the WEM, we now verify their accuracy. We do this in two ways. In the first approach, we present an execution trace simulation that compares the simulated times with the predicted times. In the second approach, we present an actual program written for the nCUBE 2.

#### 3.4.1 Simulation

To conduct the simulation of the equations from Section 3.3, we wrote a 'C' program that simulates execution of a WEM program on two processors. Included is a passage of time on each processor for useful work and for message exchanges. Every time an incoming message is expected for a given cycle on a given processor, the amount of elapsed time recorded is the larger of two possible cases:

- the time of arrival minus the elapsed time through the last execution of useful work (when the message arrives later than the last time of useful work), or
CASE 2.2) 0 <= C <= D/2

W0 = 20  \quad (D = 8)
W1 = 12
C = 3

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure3.6.png}
\caption{A four cycle, case 2.2 program}
\end{figure}

- zero (when the message arrives prior to the completion of the useful work phase)

The simulation was run with \( W_0 \) fixed at 100 time units. The value of \( W_1 \) was varied from the integers 0 through 99. The communication value \( C \) was varied from the integer values 0 to 199, and the number of cycles, \( j \) varied over the integers 1 through 100. In all, 2,000,000 work/exchange cycles were tested for a grand total of 4,000,000 equation verifications.
The results of the simulation show that the equations are correct within an accumulated error of 0.000000 time units. The error was calculated as the accumulation of the absolute value of the difference in reported values from expected values for $T_0(i)$, $T_1(i)$, $I_0(i)$, $I_1(i)$ for each of the combinations of variables discussed above and for each value of $i$ between 1 and $j$.

Both the code and the output for the simulation can be found in Appendix A.

3.4.2 Implementation test

During the beginning phase of the research on load balancing, we wrote a two processor nCUBE 2 program that models the movement of particles within a confined space. In the program, each processor is initially assigned the physical region representing half of the total problem domain space.

With the passage of time, some particles cross over from one processor's domain to the other's (and vice versa). The result is that after some time has passed, one processor will have more particles (and hence more work to do) than the other. Hence, load imbalance occurs.

In order to test the validity of the equations, the above program was modified to make several separate runs. Each run was set to have a fixed amount of unbalanced work on each processor, a set number of cycles, $j$, to run, and a fixed amount of communication delay time, $C$. A total of 12 different communication values were used. The number of cycles was varied as 1, 2, 3, 4, 5, 6, and 50 for each communication value. Finally, the workload on each processor was determined randomly for each run. Hence, a total of 84 separate runs were made for a total of 852 cycles.

Also, within the C code lines have been added that measure the actual time
taken for work, communication, and total ensemble time. Then the work and com-
munication values are placed into the WEM equations to get a predicted value. The
output of the program shows the normalized values used in each run, the WEM case
that applies, and the percentage of error of the predicted execution time to the actual
run time.

To summarize the results, the maximum percentage of error is 5.4447% with the
majority of runs averaging about 1%. We feel that this is a good confirmation of the
validity of the equations. We attribute the error obtained to have occurred due to
sharing the hypercube with others, clock inaccuracy, and clock delays due to context
switches in which the operating system would 'steal' cycles on the host.

Both the code and output from this program are found in Appendix B.

3.5 Perfectly Balanced Case

In the Section 3.3, equations are derived for the three cases of the WEM. Since
these equations state execution times for unbalanced parallel distributed programs,
the next logical step is to consider what the execution time would be for a given
program if it could be more evenly distributed over the processors.

In this section, we derive the equation that represents the perfectly balanced
case. That is, we assume in the equation that a program can be redistributed in such
a way that exactly equal quantities of useful work are distributed onto each processor.
We recognize, of course, that “real life” programs can rarely be balanced in such a
fashion. However, by assuming that a perfectly balanced state can be achievable, we
are finding the best case execution time with which we can make comparisons.

By using the best case equations and the equations derived in Section 3.3,
speedup equations can be derived that will indicate the maximum speedup that can be obtained from a program through redistribution of workload. These potential speedup equations are useful since that can indicate when further effort on the part of a programmer becomes useless.

Finally, one additional assumption must be made before the perfectly balanced equation can be derived. We assume that the communication transit delay time, $C$, is the same for both the balanced and unbalanced program distributions. Since the workload is balanced in the perfectly balanced case, each processor will reach the message exchange phase at the same time. Then each processor will sent its message to the other process and receive the incoming message at the same time as the other processor. Hence, each cycle on each processor will proceed in an identical fashion. So for a given cycle, the time to execute will be the average of the use work from the unbalanced program plus the communication delay time, $C$. For $j$ cycles, we get the following equation:

$$T_{avg}(j) = j(W_{avg} + C)$$

$$= j \left( \frac{W_0 + W_1}{2} + C \right)$$

$$= j \left( \frac{2W_0 - D}{2} + C \right)$$

$$= j(W_0 + C) - \frac{j}{2}D \quad (3.57)$$

Now that we have defined the perfectly balanced case and derived its timing equation, we will now reexamine the three examples from Section 3.3. Figure 3.7 shows the Case 1 program of Figure 3.3 with its corresponding balanced counterpart. Notice that no speedup exists for executions of even number of cycles. For odd cycle executions, a relatively small amount of speedup occurs.
CASE 1) C > D

\[ W_0 = 20 \quad (D = 8) \]
\[ W_1 = 12 \quad (D = 8) \]
\[ C = 15 \]

\[ W_{avg} = \frac{W_0 + W_1}{2} = 16 \]

**DISTRIBUTION**

UNEQUAL

EQUAL

**Time**

Figure 3.7: A case 1 program: balanced and imbalanced
CASE 2.1) $D/2 < C \leq D$

$W_0 = 20$ \hspace{1em} ($D = 8$)
$W_1 = 12$
$C = 6$

$W_{avg} = \frac{W_0 + W_1}{2} = 16$

**DISTRIBUTION**

**UNEQUAL**

**EQUAL**

**Time**

$T_1 = 22 = 2(W_{avg}C)$
$T_2 = 38 = 2(W_{avg}C)$
$T_3 = 58 = 3(W_{avg}C)$
$T_4 = 82 = 4(W_{avg}C)$

Figure 3.8: A case 2.1 program: balanced and imbalanced
CASE 2.2) $0 \leqslant C \leqslant D/2$

\[ W_0 = 20 \quad W_1 = 12 \quad C = 3 \]

\[ W_{\text{avg}} = \frac{W_0 + W_1}{2} = 16 \]

\[ \text{UNEQUAL DISTRIBUTION} \]

\[ \begin{array}{cccc}
\text{PE0} & & & \text{PE1} \\
12 & \rightarrow \rightarrow \rightarrow & 20 & \rightarrow \rightarrow \rightarrow \\
23 & \rightarrow \rightarrow \rightarrow & 35 & \rightarrow \rightarrow \rightarrow \\
35 & \rightarrow \rightarrow \rightarrow & 43 & \rightarrow \rightarrow \rightarrow \\
55 & \rightarrow \rightarrow \rightarrow & 63 & \rightarrow \rightarrow \rightarrow \\
75 & \rightarrow \rightarrow \rightarrow & 83 & \rightarrow \rightarrow \rightarrow \\
\end{array} \]

\[ \begin{array}{cccc}
\text{PE}_{\text{avg}} & & & \text{Time} \\
16 & \leftarrow \leftarrow \leftarrow & 19 = T_{\text{avg}} (1) = W_{\text{avg}} + I_{\text{avg}} \\
35 & \leftarrow \leftarrow \leftarrow & 38 = T_{\text{avg}} (2) = 2 (W_{\text{avg}} + I_{\text{avg}}) \\
54 & \leftarrow \leftarrow \leftarrow & 57 = T_{\text{avg}} (3) = 3 (W_{\text{avg}} + I_{\text{avg}}) \\
73 & \leftarrow \leftarrow \leftarrow & 76 = T_{\text{avg}} (4) = 4 (W_{\text{avg}} + I_{\text{avg}}) \\
\end{array} \]

\[ \text{EQUAL DISTRIBUTION} \]

Figure 3.9: A case 2.2 program: balanced and imbalanced
Figure 3.8 shows the Case 2.1 program of Figure 3.5 with its corresponding balanced counterpart. In this case, speedup exists for both the odd and even number of cycle executions. However, notice the odd cycles executions exhibit greater speedup than the even cycle executions.

Finally, Figure 3.9 contains the perfectly balanced counterpart to the Case 2.2 trace from Figure 3.6. Examination of the figure reveals that the greatest speedup occurs in the case compared to the figures discussed above.

At this point, the reader is probably wondering why the three WEM cases presented in Section 3.3 exhibit differing potentials for speedup. In the next section, we resolve this issue by deriving potential speedup equations for each of the three WEM cases.

3.6 Speedup Equations

Now that we have the derivation of the timing equation for the perfectly balanced case, Equation 3.57, and the timing equations for the three unbalanced cases of the WEM, Equations 3.18 – 3.56, we can now derive equations for speedup. Once again, these equations serve as an upper bound on the speedup that could potentially be obtained if a program is redistributed in a perfectly balanced manner.

3.6.1 Case 1 speedup equations

Equation derivations  So having derived the equations for both balanced and unbalanced accumulated execution time, we now derive the equation representing the
speedup, \( S(j) \), for a program that executes \( j \) cycles:

\[
S(j) = \frac{\text{Unbalanced execution time}}{\text{Balanced execution time}} = \frac{T(j)}{T_{\text{avg}}(j)} = \frac{j(W_0 + C) - \left\lfloor \frac{j}{2} \right\rfloor D}{j(W_0 + C) - \frac{j}{2}D}
\]

\[
= \begin{cases} 
1 & \text{for } j = 1, 3, 5, \ldots \\
\frac{j(W_0 + C) - \frac{j-1}{2} D}{j(W_0 + C) - \frac{j}{2}D} & \text{for } j = 2, 4, 6, \ldots 
\end{cases}
\]

\[
= \begin{cases} 
1 & \text{for } j = 1, 3, 5, \ldots \\
\frac{2(W_0 + C) - \left(1 - \frac{1}{j}\right) D}{2(W_0 + C) - D} & \text{for } j = 2, 4, 6, \ldots 
\end{cases}
\]

\[
= \begin{cases} 
1 & \text{for } j = 1, 3, 5, \ldots \\
1 + \frac{D}{j[2(W_0 + C) - D]} & \text{for } j = 2, 4, 6, \ldots 
\end{cases}
\]

\[(3.58)\]

**Upper bound on speedup** Since \( C > D \) and \( W_0 \geq D \), we can determine an upper bound for the odd cycle case of the speedup by maximizing the numerator and minimizing the denominator of the fractional part as follows:

\[
S(j) = 1 + \frac{D}{j[2(W_0 + C) - D]} \quad \text{for } \( J = 1, 3, 5, \ldots \)
\]

\[
< 1 + \frac{D}{j[2(D + D) - D]} \quad \text{for } J = 1, 3, 5, \ldots
\]

\[
= 1 + \frac{1}{3j} \quad \text{for } J = 1, 3, 5, \ldots
\]

\[(3.59)\]

This speedup upper bound shows how a person can be misled when he examines only the first cycle to determine load imbalance of a program that runs for multiple cycles.
For the case of an even number of cycles, no speedup is possible. So an unbalanced program will perform as efficiently as a perfectly balanced program when $C > D$. This is an example that demonstrates that the naive definition of load imbalance is misleading and in need of refinement. We define the measure of imbalance to be the reciprocal of the speedup.

### 3.6.2 Case 2.1 speedup equations

**Equation derivations**

Having derived the formula for $T(j)$ in Equations 3.38 and 3.39 and the formula for $T_{avg}(j)$ in Equation 3.57, we now derive the speedup, $S(j)$, for Case 2.1:

$$
S(j) = \frac{T(j)}{T_{avg}(j)}
$$

For $j = 1, 3, 5, \ldots$

$$
= \frac{w_0 + c + \left(\frac{i-1}{2}\right)[2w_0 + 2c - D]}{j(w_0 + c) - \frac{D}{2}}
$$

For $j = 2, 4, 6, \ldots$

$$
= \frac{D - c + \frac{1}{2}[2w_0 + 2c - D]}{j(w_0 + c) - \frac{D}{2}}
$$

**Upper bound on speedup**

In addition, we can find the upper bounds of the speedup by maximizing the numerator and minimizing the denominator of the
fractional component. This is done by taking \( C \) to its smallest possible value of \( D/2 \) and by taking \( D \) to its largest possible value of \( W_0 \):

\[
S(j) = 1 + \frac{D}{j(W_0 + 2C - D)} < 1 + \frac{1}{2j} \text{ for } j = 1, 3, 5, \ldots \quad (3.63)
\]

\[
S(j) = 1 + \frac{D - C}{j(W_0 + C - \frac{D}{2})} < 1 + \frac{1}{2j} \text{ for } j = 2, 4, 6, \ldots \quad (3.64)
\]

### 3.6.3 Case 2.2 speedup equations

**Equation derivations** The final equation left to derive is one of speedup for this case. So, when \( 0 < C < \frac{D}{2} \), the speedup through the \( j^{th} \) cycle, \( S(j) \), is given by:

\[
S(j) = \frac{T(j)}{T_{avg}(j)} = \frac{jW_0 + C}{j(W_0 + C) - \frac{1}{2} D} = \frac{2W_0 + 2C - D + \frac{2C}{j} - 2C + D}{2W_0 + 2C - D} = 1 + \frac{\frac{1-i}{j}2C + D}{2W_0 + 2C - D} - \frac{D - \frac{i-1}{j}2C}{2W_0 - (D - 2C)} \quad (3.65)
\]

where \( T_{avg}(j) \) is the accumulated time equation for the perfectly balanced case derived in Equation 3.57.
Upper bound on speedup. By taking \( D \) to its maximum value of \( W_0 \), and taking \( C \) to its minimum value of 0, we get the upper bound limit as follows:

\[
S(j) = 1 + \frac{i-1}{2} \frac{2C}{D - 2C} \leq 1 + \frac{W_0 - 0}{2W_0 - W_0 + 0} = 2
\]

This matches the maximum theoretical limit for a two processor machine.

Table 3.1 contains a summary of all important equations derived in Section 2.

<table>
<thead>
<tr>
<th>Case 1</th>
<th>Case 2.1</th>
<th>Case 2.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C \geq D )</td>
<td>( D/2 &lt; C \leq D )</td>
<td>( 0 \leq C \leq D/2 )</td>
</tr>
<tr>
<td>( S(j) )</td>
<td>( 1 + \frac{1}{2} \frac{D}{2W_0 + 2C - D} )</td>
<td>( 1 + \frac{D - i - 1}{2} \frac{C}{2W_0 - (D - 2C)} )</td>
</tr>
<tr>
<td>( S_{Max}(j) )</td>
<td>( 1 + \frac{1}{2} )</td>
<td>2</td>
</tr>
<tr>
<td>( T_0(j) )</td>
<td>( j(W_0 + C) - \left\lfloor \frac{j}{2} \right\rfloor D )</td>
<td>( jW_0 )</td>
</tr>
<tr>
<td>( T_1(j) )</td>
<td>( j(W_0 + C) - \left\lfloor \frac{j}{2} \right\rfloor D )</td>
<td>( jW_0 + C )</td>
</tr>
<tr>
<td>( T(j) )</td>
<td>( T_1(j) )</td>
<td>( T_1(j) )</td>
</tr>
<tr>
<td>( I_0(j) )</td>
<td>( C - D )</td>
<td>0</td>
</tr>
<tr>
<td>( I_1(j) )</td>
<td>( D + C )</td>
<td>( D + C )</td>
</tr>
</tbody>
</table>

3.7 Speedup Graphs

Graphs of potential speedup of unbalanced programs for various values are shown in Figures 3.14 through 3.10, respectively. The graphs clearly show how dramatically
the first few cycles can vary from the asymptotic value of imbalance. Since the naive definition of load imbalance corresponds to just the first cycle, it is clear how misleading and inaccurate the naive definition really is.

![SPEEDUP GRAPH](image)

Figure 3.10: Speedup curves for $W_0 = 100$ and $W_1 = 100$

One can take this observation a step further and consider the influence of both work distribution and communication factors. While both play a major role in the degree of imbalance, it is clear from the complexity of the timing equations and the overlap of cycles in the model diagrams that more factors are involved. However, the exact nature of these additional factors are is not clear from this work. The desire to determine what these additional factors is a primary motivation for a new approach. This new approach should not only clarify the various factors of load imbalance, but
should also be applicable to an arbitrary number of processors and to any arbitrary code structure on each processor. Such an approach is the topic of later sections of this report.

As a final note before leaving this section, when studying load imbalance we are interested in both MIMD and SIMD architectures. In the Work/Exchange Model, by allowing the amount of work to differ between processors and then calculating the total execution time by taking both processors through \( j \) cycles, we are modeling the MIMD approach. In a SIMD architecture, however, all processors execute exactly the same code with some processors being active while the remainder are inactive. We can model this behavior by forcing the work in each processor to be the same. In this case we view the work to include the stepping through code, even when inactive.

For a given cycle, if a processor is inactive through the entire cycle, then it does not process any incoming messages. Therefore, no idle time occurs. This is important since all processors must wait for the length of the maximum idle time in the multiprocessor for that cycle. One way to simplify the view is to consider a SIMD program in which each PE is processing a queue of jobs. Once a given PE has emptied its queue it goes idle for the remainder of the program. So in terms of the Work/Exchange Model, successive cycles for this processor need not be drawn. Hence we can generalize the distinction between MIMD and SIMD in Work/Exchange Model as varying work per cycle with fixed number of cycles, versus fixed work per cycle with varying number of cycles, respectively. Of course, hybrids may also be considered that use a combination of the above.
3.8 Conclusions

In this chapter, we have successfully demonstrated how time-stepped, deterministically load unbalanced programs can be modeled for the two processor case. By use of this model, we were also able to derive exact timing and speedup equations for these same programs. Input to the equations consists of the work per cycle for each processor, the communication delay, and the total number of cycles. Results show that for cases where the communication delay is less than the difference of work on each processor, that little or no speedup is possible. However, when communication delay is relatively large compared with the difference of work, speedup can be as large as the asymptotic limit of two.
Figure 3.11: Speedup curves for $W_0 = 100$ and $W_1 = 75$
Figure 3.12: Speedup curves for $W_0 = 100$ and $W_1 = 50$
Figure 3.13: Speedup curves for $W_0 = 100$ and $W_1 = 25$
Figure 3.14: Speedup curves for $W_0 = 100$ and $W_1 = 0$
4. MYTHS OF LOAD BALANCING

4.1 Introduction

Achieving good performance from a multiprocessor is a nontrivial task. Fortunately, there are performance enhancement tools at one's disposal such as dynamic schedulers, execution time profilers, and load balancers. The popularity of load balancing is increasing with the continuance of research and development of load balancing algorithms. Much of the recent research focuses on methods of obtaining uniform workload [2, 3, 7, 9, 12, 13, 14, 23, 24, 27, 32, 36, 37, 38, 41, 44, 45, 51]. Often these methods disregard factors such as communication costs and network contention. Hence, many myths\(^1\) have arisen regarding the performance improvement that may result from use of a load balancer.

In this chapter, we expose as incorrect, a hierarchy of three commonly held beliefs. For the first two myths, we assume a general computing model with full-duplex communication channels and ignore network contention issues. For the third myth, we assume that either communication channels are half-duplex or that network contention exists.

\(^1\)Lest we give the reader some misconceptions from reading the above, we hasten to point out that none of the myths discussed in this chapter are directly attributable to any particular paper or person; rather, they pervade the field of topic as commonly assumed beliefs.
The first myth is that load balancing is always useful for increasing performance. The fallacy of this belief is that the cost of the load balancer is usually not negligible. We have found that even for balancers whose invocation has been tuned, this cost can cause unbalanced programs to execute in less time than their balanced counterparts.

Now, for the sake of argument, assume that a load balancer can be found with negligible cost. Under this assumption, one might think that a balanced program is always faster than its unbalanced counterpart. This is the second myth. We find in practice that unbalanced programs often require no more time than their balanced counterparts. This observation follows from the interaction between load imbalance and communication delay times.

To get our strongest statement yet, we assume the presence of either half-duplex channels or network contention. Once again, let us assume that a zero-cost load balancer can be found. Then the third myth is that a balanced program is never slower than its unbalanced counterpart. In fact, unbalanced programs can execute more quickly than their balanced counterparts.

In Sections 3 and 4, we examine in greater detail, the first and second myths, respectively. Included are counterexamples and descriptions of the conditions that cause these unexpected results. To more easily facilitate discussion of these two myths, we first introduce, in Section 2, a simple computing model called the Generalized Work/Exchange Model (GWEM) [47]. In the GWEM, timing equations have been developed for both unbalanced programs and their balanced counterparts. From these timing equations, speedup equations have also been derived as the ratio of unbalanced execution time to balanced execution time.

Then in Section 5, we consider the third myth without use of the GWEM.
Instead, a two processor, half-duplex channel architecture is assumed for a specific counterexample. Finally, concluding remarks are given in Section 6.

4.2 Myth 1

In this section, the first of a hierarchy of three myths is examined. We assume in this section that full-duplex channels are used and we ignore network contention issues. The first myth is that load balancing is always useful. While load balancing does occasionally reduce the overall execution time of a distributed program, there are instances in which unbalanced programs outperform their dynamically balanced counterparts. The cause of this unexpected behavior is attributable to the cost of the load balancer. We have found that this cost is usually quite significant.

When most people use a load balancer, they try to tune the balancer to have an optimal or near-optimal period. For programs with deterministic load imbalance, this approach works well. However, due to the very nature of many scientific applications, the workload on each processor generally changes over time – often in an unpredictable manner. Such behavior is called nondeterministic load imbalance [7]. For such programs, tuning the load balancer is, at best, an art. Since the vast majority of load balancing algorithms do not have the ability to adjust their periods of invocation to adapt to changing workloads, programs that have been tuned by observation of the first few cycles will typically become untuned in very short time.

\footnote{By cost, we mean the additional execution time and idle waiting time overheads that result from using the balancer.}

\footnote{See [13, 48, 49] for two different solutions to the dynamic tuning problem for nondeterministic workloads.}
Therefore, a program state is often reached in which the load balancer is:

1. invoked too often causing excessive overhead, or

2. invoked too infrequently

The first state listed above is related to the first myth. In the remainder of this section, we will use the GWEM to more closely examine when a balancer should be used. Then we will give a specific example of a program with nondeterministic load imbalance that performs better without balancing.

4.2.1 To balance or not to balance?

Of those programs having nondeterministic load imbalance, many have the additional characteristic that the workload changes in a gradual, or quasi-static, manner. We have found that the equations developed in the GWEM are still useful for estimating execution times for programs both with and without load balancing. We denote the elapsed time for the former case by \( T_{LB} \) and the elapsed time for the latter case by \( T \). To apply the equations, one must first obtain estimates for \( W_0 \), \( W_1 \), and \( C \). For the first case, the values of \( W_0 \) and \( W_1 \) are the same since the load balancer is to be invoked. Therefore, both values are set to the average of the estimates of the next case. For the second case, we use the actual values of \( W_0 \) and \( W_1 \) as measured in the cycle that the load balancer would be invoked. In both cases, we use the actual value of \( C \) measured in the previous cycle as the estimate for the current value of \( C \).

Finally, we must also estimate the cost for invoking the load balancer, \( C_{LB} \). Once again, this value may be estimated or can be measured by using the actual
measured time that the load balancer required during its most recent, prior invocation.

Once we have obtained estimates for $T$, $T_{LB}$, and $C_{LB}$, we can decide whether or not to use a load balancer. The criteria for the decision are as follows:

- if $T_{LB} + \alpha + C_{LB} \leq T$, then use a load balancer
- if $T_{LB} + \alpha + C_{LB} > T$, then do not use a load balancer

We include a correction factor, $\alpha$, in the equations to account for the additional execution time that can be expected as a balanced program gradual becomes unbalanced. Figure 4.1 illustrates an example of a program trace in the GWEM that requires less time to execute in an unbalanced state than it does using a balancer. In the example, we show the workloads of the program with a balancer to be 82 time units and 68 time units, respectively, to illustrate that perfectly balanced workloads will become unbalanced over time. Notice that the unbalanced program required less time to execute than its balanced counterpart.

4.2.2 Experimental results

Next, we consider an actual program to further illustrate the point that an unbalanced program can outperform its balanced counterpart.

**Problem description** The problem we selected is a very simplified rendition of a weather modeling program that simulates the development and movement of "hurricanes". In our program, evenly spaced grid points are used to represent positions of the earth's surface. Each process is responsible for a contiguous set of points.
Figure 4.1: Equal time balanced and unbalanced trace; nonzero-cost balancer
Initially, each process is assigned to one-half of the surface domain; we use only two processors in this example. Storm velocities are determined at the program's start using a random number generator. The number of storms is held constant, but each storm may move around within the domain. The surface area that a storm covers also changes with time. Cloud formation and abatement is determined by the current state of a grid point and the state of that grid point's nearest neighbors.

The program was run multiple times. For each set of runs, a different assignment of maximum storm intensity was used. As maximum storm intensity increases, so does the likelihood of rapid cloud formation. Since storm intensity is related to cloudtop height, storm intensity affects the volume of atmosphere that a storm envelopes. Hence, storm intensity at a grid point is directly proportional to the computational work required for a grid point. Therefore, changes of imbalance increase as storm intensity increases. Areas with no cloud development are assigned a unit value of storm intensity. All runs were conducted using two nodes of a 64-node nCUBE 2.

**Results** Figure 4.2 illustrates typical performance results for our abstract hurricane modeling program. The $x$-axis of the graph is in units of maximum storm intensity assigned to the fixed number of "hurricanes". The $y$-axis is in units of speedup of runs using the balancer compared to times for runs without the balancer. (Note that we use "speedup" here to indicate relative speed of two parallel versions, not parallel to serial comparison.)

Notice in Figure 4.2 that unbalanced runs outperform the balanced runs for storm intensities of 0 to 135 units. This is explained by the low levels of imbalance.

---

A set of runs consists of running the program with and without a load balancer.
occuring in these program runs. Recall, that the amount of work needed to be performed for a given grid point is directly related to the storm intensity at that grid point. Clearly, when the load balancer itself requires more time than the time saved by using the balancer, a slowdown in program execution results.

4.3 Myth 2

From the previous section, we see that the cost of a load balancer can cause an unbalanced program to execute in less time than the dynamically balanced counterpart. However, one might argue that the first myth does not apply to their programs since they could have or will someday develop a load balancer with negligible cost. Then they might state that use of such a load balancer would cause their programs
to run in less time than if the balancer had not been used. Such an argument falls under the domain of the second myth.

The second myth states that if a zero-cost load balancer can be found, then a dynamically balanced program will always outperform the unbalanced counterpart. Instead, we have found that even with a hypothetical, zero-cost load balancer, unbalanced programs often require no more time to execute than their dynamically balanced counterparts. Once again we assume the presence of full-duplex channels and we ignore network contention issues.

As we saw in Section 3, the criteria of when to use a load balancer is as follows:

- if $T_{LB} + \alpha + C_{LB} \leq T$, then use a load balancer
- if $T_{LB} + \alpha + C_{LB} > T$, then do not use a load balancer

However, in this section, we are assuming that a zero-cost load balancer will someday be found. Therefore, we can simplify the above criteria into the following:

- when $T_{LB} + \alpha \leq T$, then use a load balancer
- when $T_{LB} + \alpha > T$, then do not use a load balancer

Hence, to illustrate the myth, it suffices to find examples in which $T_{LB} = T$. Examination of the GWEM speedup equations shows that such examples are easy to find. For instance, consider the Case 2.1 situation illustrated in Figure 4.3 where $W_0 = 100, W_1 = 50, C = 50$, and $j = 2$. In this example, both the unbalanced and dynamically balanced versions of the program finish at the same time. Such situations are even more common in Case 1 situations. The basic reason for this behavior lies with the ratio of communication transit delay cost compared to the difference in
Figure 4.3: Equal time balanced and unbalanced trace; zero-cost balancer
workloads on the two most heavily loaded processors. As the value of $C$ increases, a condition is eventually reached in which all processors become idle while waiting for incoming messages. In such cases, it doesn’t matter whether or not the workload is balanced since all the processors have idle time that can be applied to additional work. One significant conclusion that can be made from this observation is that it may not always be possible to speed up unbalanced programs that periodically synchronize in a networking or distributed environment.

4.4 Myth 3

Finally, in this section, we make the strongest of the three claims by showing that an unbalanced program can actually execute in less time than its dynamically balanced counterpart. This is a direct contradiction to the myth that dynamically balanced programs with zero-cost balancers will *always* perform at least as well as the unbalanced counterparts. Again, we give the benefit of the doubt to those that think a zero-cost balancer will someday be found.

In order to show that the myth is wrong, it suffices to provide a single counterexample that shows an unbalanced program outperforming its dynamically balanced counterpart. We choose to restrict ourselves to an architecture having 2 processors and a half-duplex channel. In Figure 4.4, a counterexample is shown. With some thought, one can also see why allowing network contention with full-duplex channels might cause similar results.
Figure 4.4: A faster unbalanced program using a half duplex channel
4.5 Conclusions

In the literature on load balancing, it is not uncommon to find variations of the following myths:

- load balancing is *always* useful for improving performance,
- given a zero-cost and tuned load balancer, a dynamically balanced program will *always be faster* than its unbalanced counterpart, and
- given a zero-cost and tuned load balancer, a dynamically balanced program will *never be slower* than its unbalanced counterpart.

In this paper, we have shown each of the myths outlined above to be incorrect.

However, it is not our intent to say that load balancing is not useful. On the contrary, we feel that load balancers are essential performance enhancement tools. Rather, our intention is to point out that situations of imbalance do exist that can not be remedied by load balancing. In fact, we have even found situations in which a tuned load balancer will increase the execution time of a program.

With the aid of the Generalized Work/Exchange Model, we were also able to show the exact conditions and causes for these unexpected results. The overall results can be summarized by saying that whenever communication transit times are relatively large compared to the difference in work on the two most heavily loaded processors, then load balancing will not be beneficial. But as the communication time decreases, the benefits of using a load balancer tends to increase.
5. AN ENHANCER FOR DYNAMIC LOAD BALANCING

5.1 Introduction

In many scientific realms there exist problems that are still beyond the reach of current computational abilities. While it is true that many such problems are being solved using vector computers and multiprocessors, there is no clear cut method of porting a problem on a given target architecture so as to effectively harness the full power of parallelism available on the machine. One of the main concerns, of course, is minimizing communication overhead on the parallel machine. Other concerns are determining the domain decomposition and performing load balancing.

In recent years, issues of load and scheduling have been analyzed carefully, and many creative algorithms have been developed to perform load balancing [7, 13, 14, 18, 19, 25, 42, 46, 49, 50, 48]. One common characteristic of these balancers is that they are invoked periodically after a set number of iterations. Such load balancers include those presented in [12, 27, 26, 28, 41, 51]. For problems that have deterministic load imbalance [7], this approach works well. However, for many problems, the amount of load imbalance varies with time. In these cases we find that the load balancer is sometimes invoked when it is not needed, thereby causing the program to take longer than it would have without the load balancer. Alternately, there are times when the load imbalance changes faster than expected so that the
load balancer is not invoked often enough.

For such problems, it is desirable to have a load balancer enhancement that can dynamically adjust to changing imbalance within a program. In this chapter, we present a method for enhancing load balancers which incorporates two features:

- the ability to decide, once invoked, whether to perform any load balancing for this invocation, and
- the ability to tune the period of invocation.

In order for the enhancement to have these decision-making abilities, there must be a way for it to intelligently estimate the time required for the program to run to the next load balancer invocation with and without balancing. It also needs to predict both the time required to perform the load balancing and the time required by the enhancement.

This brings us to the real crux of the problem: How can one make reliable timing predictions about unpredictable execution behavior? We start off by considering a simple model of two processors and developing the necessary framework for providing criteria that can be effectively used by the enhancer. For the two processor case, we develop equations for deterministic load unbalanced problems. Next we use these equations and the concept of temporal locality to get close approximations of execution time for programs containing nondeterministic load imbalance. To keep matters simple, we restrict ourselves to problems with a regular time stepping phase.

In this chapter, we use the timing equations developed in Chapter 2. Recall that these equations are applicable to the two processor case for programs with deterministic load imbalance and return exact results. They differ significantly from
the equations developed in [7] in that overlap is allowed between useful computation
on one cycle and the waiting time due to communications on the previous cycle. They also differ in that the equations presented here are for two processors while the
equations in [7] are applicable for large numbers of processors.

In Section 2, we show how these equations are adapted to the nondeterministic
load unbalanced case in which temporal locality holds. While more is said later about
temporal locality, let it suffice for now to say that temporal locality is the principle
that program behavior changes in a gradual enough fashion so that one cycle's timing
characteristics is similar to the timing characteristics of the cycle that follows.

Then in Section 3, we present experimental results of an abstract climate problem
with no load balancing, with load balancing, and with enhanced load balancing. The experiments are performed on the nCUBE 2 using two processors. Finally, in
Section 4 we present our conclusions.

5.2 Extensions Using Temporal Locality

While the equations derived above are good for certain programs with deter­
ministic load imbalance, they are not sufficient for programs with nondeterministic
load imbalance. Since the goal of this chapter is to show how load balancers can be
enhanced to speed up programs with nondeterministic load imbalance, we need to
show how the equations can be applied to these kinds of programs. We do this by
taking advantage of temporal locality.
5.2.1 Temporal locality

In [11, 33] the principle of locality is introduced with reference to working sets for page-faults within an operating system. We modify this into the principle of temporal locality. This property say that during any given cycle in our multiprocessor environment, there is a high probability that the time required to execute the next cycle's task on the same processor is $W_{j+1} = W_{j} \pm \phi$ for some $\phi$ such that $\phi \ll W_{j}$. In other words, the execution time for tasks on a given processor changes in a gradual fashion. Therefore, the load imbalance also changes gradually.

Since the program to be speeded up is assumed to exhibit temporal locality, we claim that the deterministic load imbalance timing equations can still be used providing that the number of cycles between load balancer invocations is sufficiently small as to assure near correctness from the equations.

5.2.2 Enhancements to the load balancer

To balance or not to balance? Under the assumption of temporal locality, the equations derived in the previous section are used to estimate the execution times of the program from the current cycle up to some arbitrary\(^1\) number of additional cycles for an arbitrary amount of imbalance during this cycle. Hence, we can now estimate the times the processors take from the current invocation of the enhancer until the next invocation of the enhancer. There are two cases to consider. First is the case that the enhancer decides to not invoke the load balancer. We denote the elapsed time for this case by $T$. The second case is that the enhancer does invoke the

\(^1\)Maximum number of additional cycles is dependent on how rapidly the program's load imbalance is changing.
load balancer. This elapsed time is denoted by $T_{LB}$. To apply the equations, one must first obtain estimates for $W_0$, $W_1$, and $C$. For the first case, we use the actual values of $W_0$ and $W_1$ as measured in the previous cycle. The principle of temporal locality assures us that these estimates are close approximations to the correct values. For the second case, the values of $W_0$ and $W_1$ are the same since the load balancer is to be invoked. Therefore, both values are set to the average of the estimates of case one. In both cases, we use the actual value of $C$ measured in the previous cycle as the estimate for the current value of $C$.

The enhancer must also be able to estimate the cost for invoking the load balancer, $C_{LB}$. Once again, this value may be estimated by using the actual measured time that the load balancer required during its most recent, prior invocation.

Finally, the enhancer must be capable of estimating the cost that the enhancer itself requires, $C_E$ for making its estimates. This is a fixed cost and is measured only once at the start of the program.

Once the enhancer obtains estimates for $T$, $T_{LB}$, and $C_{LB}$, it decides whether or not to invoke the load balancer. The criteria for the decision are as follows:

- if $T_{LB} + C_{LB} \leq T$, then invoke the load balancer
- if $T_{LB} + C_{LB} > T$, then do not invoke the load balancer

The enhancer does not use the value $C_E$ for deciding to invoke the balancer because this cost is always incurred by the enhancer. The value $C_E$ is used, however, in the decision of whether or not to change the period. More is said on changing the period in the next section.
Changing the period  The other function of the enhancer is to tune the period of the invocation of the load balancer. Clearly, if the program becomes unbalanced very slowly and the balancer is frequently invoked, its period should be increased. Similarly, if the imbalance is changing quickly and the balancer is infrequently invoked, then the period should be decreased.

Determination of when to change the period can be made by comparing the quantities $T_{LB} + C_{LB} + C_E$ and $T$. The rules for changing the period are given as follows:

- if $T_{LB} + C_{LB} + C_E + \alpha < T$, then decrease the period
- if $T < T_{LB} + C_{LB} + C_E - \beta$, then increase the period
- if $T + \beta \leq T_{LB} + C_{LB} + C_E \leq T - \alpha$, then do not change the period

The role of $\alpha$ and $\beta$ is to prevent changes in the period from occurring when both sides of the relations are approximately equal.

5.3 Experimental Results for Nondeterministic Load Imbalance

5.3.1 Problem description

The problem we selected is a very simplified version of a climate modeling program that simulates the development and movement of “thunderstorms”. Although most of the actual physics is absent in our program, the program is still very similar in structure to actual climate modeling programs. Therefore, we feel that it is a good representative test of what can be done with actual climate problems.

In our program, evenly spaced grid points are used to represent positions of the earth's surface. Each process is responsible for a contiguous set of points. Initially,
each process is assigned to one-half of the surface domain. Storm velocities are determined at the program's start using a random number generator. The number of storms is held constant, but each storm may move around within the domain. The surface area that a storm covers also changes with time. Cloud formation and abatement is determined by the current state of a grid point and the state of that grid point's nearest neighbors.

The program was run multiple times. For each set of runs\(^2\), a different assignment of maximum storm intensity was used. As maximum storm intensity increases, so does the likelihood of rapid cloud formation. Since storm intensity is related to cloudtop heights, storm intensity affects the volume of atmosphere that a storm envelopes. Hence, storm intensity at a grid point is directly proportional to the computational work required for a grid point. Therefore, changes of imbalance increase as storm intensity increases. Areas with no cloud development are assigned a unit value of storm intensity. All runs were conducted using two nodes of a 64-node nCUBE 2. The actual "thunderstorm model" code is in Appendix C.

5.3.2 Results

Figure 5.1 illustrates typical performance results for our abstract climate modeling program. The x-axis of the graph is in units of maximum storm intensity assigned to the fixed number of "thunderstorms". The y-axis is in units of speedup of runs using the balancer or balancer and enhancer compared to times for runs with neither the balancer nor enhancer. (Note that we use "speedup" here to indicate relative

\(^2\)A set of runs consists of running the program with no load balancer, with a balancer, and with both a balancer and enhancer.
speed of two parallel versions, not parallel to serial comparison.)

Figure 5.1 contains many interesting results. The first is that the enhanced runs generally outperform the load-balanced-only runs by about five to ten percent in terms of net program speedup.

![Comparison of execution times](image)

Figure 5.1: Comparison of execution times

Note that unbalanced runs outperform the balanced runs for storm intensities of 0 to 135 units. This is explained by the low levels of imbalance occurring in these program runs. Recall, that the amount of work needed to be performed for a given grid point is directly related to the storm intensity at that grid point. Clearly, when the load balancer itself requires more time than the time saved by using the balancer, a slowdown in program execution results.

To a significantly lesser degree, the same slowdown behavior occurs when the enhancer is used for storm intensities of 0 to 50. However, in these cases, the enhancer
is able to successfully determine that the load balancer is not needed. Therefore, the only additional cost incurred by these runs is the time required by the enhancer to make its decisions. This time penalty is too slight to be visible on the graph.

Another observation is that performance of load balanced runs, with and without the enhancer, begin to converge at the 350 level of storm intensity. We believe this to be the result of increasing load imbalance caused by more rapid cloud formation in one processor's domain than the other's. Since load imbalance is increasing, the need to invoke the load balancer also increases. Thus, as the optimal period for invoking the balancer approaches one, the calculations performed by the enhancer become unnecessary overhead. We observe in runs above the 380 storm level intensity that the enhanced runs and load-balanced-only runs do indeed converge and that the balanced-only and enhanced runs perform about equally well.

5.4 Conclusions

A common characteristic of time-stepping scientific problems ported to multiprocessor machines is that each exhibits inherent load imbalance, since these programs model actual physical processes. The load imbalance changes gradually as a function of time, but in an unpredictable manner. As the levels of imbalance change with time, so does the need for balancing. In this chapter, we have addressed this issue by introducing a load balancing enhancer. This enhancer has the capability to both decide if the balancer should be invoked when called upon and to change the period of the balancer as needed.

The decision-making ability of our enhancer is based on equations developed in the Work/Exchange Model for time-stepping problems with deterministic load
imbalance. By application of the temporal locality that many scientific programs naturally exhibit, we are able to extend use of these equations to situations where nondeterministic load imbalance arises.

It is the hope of the authors to extend our work to include enhanced balancing for programs distributed on more than two processors. However, due to the rich complexity of the equations for two processors, we will need carefully-chosen simplifications in order to proceed to the many-processor case. We believe that we have developed a firm foundation from which extensions to higher number of processors can be developed.

Results from our implementation are highly encouraging. For programs with only slight imbalance, the enhanced program performs marginally worse than original, unbalanced program. However, it still gives better results than when a load balancer is used alone. For programs with very rapidly changing imbalance, we found that it is better to use a balancer without an enhancer since such programs tend to need balanced every cycle anyway. However, for programs with moderately changing imbalance, the enhancer makes significant improvement in performance over the performance obtained when only a load balancer is used.
6. THE GENERALIZED WORK/EXCHANGE MODEL

6.1 Introduction

In many scientific realms, problems exist beyond the reach of current computational abilities. While it is true that many such problems are being solved using vector computers and multiprocessors, no clear cut method exists for porting a problem on a given target architecture so as to effectively harness the full power of parallelism available on the machine. One of the main concerns, of course, is minimizing communication overhead on the parallel machine. Other concerns include determining the domain decomposition and performing load balancing. Load balancing may be static, dynamic, or real-time.

This research focuses on dynamic load balancing and its relationship to communication costs. Load balancing is the process by which unequal workloads in the multiprocessor environment are redistributed among the processors in more equitable proportions. The basic idea is to reduce idle time during the execution of a parallel program. This in turn, should reduce the wall-clock time needed to run the program. The term "dynamic" means that the redistribution is performed zero or more times during the execution of the program.

From the literature survey discussed in Chapter 2, our attention was focused on two issues in particular. First, very little attention has been paid to the effects
of communication costs on load balancing and effects of overlapping communication costs with load balancing. In fact, most authors ignore the issue entirely [3, 6, 12, 14, 17, 18, 34, 41, 45, 46]. Of those authors that do consider communication cost, most either use statistical methods to estimate average performance for large numbers of processors [9, 13, 15, 16, 20, 23, 24, 25, 27, 28, 32, 36, 37, 38, 40], or they assume that communication delays and useful work cannot overlap [7].

Secondly, many papers present heuristic methods for load balancing [2, 3, 5, 6, 9, 12, 13, 14, 15, 17, 18, 23, 24, 26, 27, 28, 32, 34, 39, 41, 44, 45, 46, 51]. While several authors attempt to justify the choice of their methods through intuitive or comparative arguments, analytical arguments are usually not presented.

The observation of these two points was the primary motivation for focusing the author's research goals. The goals are as follows:

- examine the relationship of and tradeoffs between communication costs and load imbalance,

- develop a theoretical framework to aid in the assessment of heuristic load balancing methods, and

- utilize the theory to design a practical heuristic

In this chapter, we present the results of research to date regarding the second research goal. As of the time of this writing, a theoretical framework called the Generalized Work/Exchange Model has been developed. The Generalized Work/Exchange Model is a theoretical framework used to model iterative looped programs scheduled onto an arbitrary number of processors. The model applies only to those programs that conform to the restrictions outlined in the next section. This model extends the
two processor Work/Exchange Model\textsuperscript{1} presented in [49, 50] to include an arbitrary number of processors.

In the next section, we present the framework, definitions, and notation of the model. Then in Section 3, equations are derived that exactly describe the elapsed time that will occur on a given processor after execution for \( j \) cycles. Equations for idle time on a given processor for a given cycle are also derived. In Section 4, the concept of a perfectly balanced case is presented along with its corresponding elapsed ensemble time equations. Using the equations developed for both perfectly balanced and for unbalanced programs, we derive speedup equations. These speedup equations serve as an indication of the best-case speedup obtainable for infinitely divisible programs. Finally, a summary and discussion of future work is presented in Section 5.

\textbf{6.2 The Generalized Model}

In the Generalized Work/Exchange Model, several assumptions about the iterative looped program must be made. First, the program is assumed to be distributed on exactly \( n \) processors. Second, the distributed program consists of a sequence of tasks on each processor such that after the execution of a task on a processor, a message is sent to and read from all other processors\textsuperscript{2}. Third, on a given processor, the execution time of each task is identical from one cycle to the next. And finally, the

\textsuperscript{1}Note: For those familiar with [50], there has been a notational change. In this report, we use \( W_i \) and \( I_i \) to denote non-idle work time and idle time, respectively. Previously, these values were denoted by \( S_i \) and \( W_i \), respectively.

\textsuperscript{2}It is this cyclic behavior of performing a task and then exchanging messages that led to the name "Work/Exchange Model". The term 'Generalized' comes from the extension of the model from two to an arbitrary number of processors.
time required for every message to transit the network is constant\textsuperscript{3}. Programs that meet the above requirements include certain finite difference and climate modeling problems. Additional problems meeting these requirements are discussed in [19].

Figure 6.1 illustrates the timing diagram for the first cycle of a program in the Generalized Work/Exchange Model on 3 processors. Notice that PE\textsubscript{0} does not have any idle time during the first cycle. We define a cycle as the tuple \((i, T_i(j), T_i(j + 1), W_i, I_i(j))\) where \(T_i(j)\) and \(T_i(j + 1)\) are the starting time and completion time of the \(j^{th}\) iteration running on PE\textsubscript{\(i\)}, respectively. We define an iteration as a two-phase process consisting of a non-idle work phase taking time \(W_i\) followed by a message exchange phase. The message exchange phase often entails idle waiting time and uses \(I_i(j)\) time.

In the model, we envision \(n\) processors in a cycle of performing work \(W_i\) followed by an exchange of messages. The message exchange consists of a nonblocking send followed by a blocking read. Sometimes, a processor completes execution of a task prior to receiving the expected, incoming message. When this occurs, the processor accrues idle wait time \(I_i(j) > 0\).

Before we derive the actual equations, an explanation of variables and notation is needed. First, we use a subscript \(i\) to distinguish processors. Second, the value in parenthesis denotes the current cycle except in the case of time variables like \(T(j)\) and \(T_i(j)\). For time variables, the parenthesized value denotes the total number of elapsed cycles.

For the \(i^{th}\) processor, PE\textsubscript{\(i\)}, the time required during the \(j^{th}\) cycle for idle time

\textsuperscript{3}To facilitate derivation of the equations, we assume that each processor broadcasts it message to all other processors.
is denoted as $I_i^j$. Idle time varies and is a function of execution time of previous cycles, the work times on the different processors, and the communication time. As the model contains $n$ processors, $i$ may be any whole number in the range $0 \leq i < n$. Also $j$ may be any positive integer. Similarly, the time required on processor $i$ during the $j^{th}$ cycle for non-idle time is $W_i^j$. Since the non-idle time is restricted to be constant for a given processor, we simplify the notation to $W_i$. To simplify the model further, we assume without loss of generality that $W_i \geq W_{i+1}$ for all $i \geq 0$. As a notational shorthand, we define constant $D_{ij}$; where $D_{ij} = W_i - W_j$. Notice that this implies that $W_i = W_j + D_{ij}$.

The communication network transit time is given by $C$ and remains constant in the model. The value $T_i^j$ represents the accumulated execution time of a program.
on PE\textsubscript{i} for the first \( j \) cycles. Since a program does not finish until the completion of the final processor, an accumulated ensemble time is also needed. We denote this time as 
\[
T(j) = \max_{0 \leq i < n} T_i(j).
\]

As we derive timing and speedup equations in the two sections that follow, it will become apparent that certain combinations of terms will frequently occur. Therefore, in order to save space and to logically relate these different terms, we introduce two additional terms. The first term represents the average workload on the \textit{two most heavily loaded} processors and is denoted by \( W_{01} \). \( W_{01} \) denotes a shorthand way of expressing \((W_0 + W_1)/2\). \( H_{01} \) is the second term and it expresses half the difference in workload of the two largest workload values. Expressed mathematically, we mean that $H_{01} = D_{01}/2 = (W_0 - W_1)/2$. See Figure 6.3 for an illustration of the intuitive meanings of some of the more abstract variables discussed above.

### 6.3 Elapsed Time Equations

Now, the equations can be derived. First, consider the idle time, \( I_0(1) \), on PE\textsubscript{0} for the first cycle. This value is the maximum time between the completion of useful work time on PE\textsubscript{0} and the arrival of messages from each of the other processors. The derivation for \( I_0(1) \) is as follows:

\[
I_0(1) = \max_{i \geq 0} (W_i + C - W_0, 0)
= \max(W_1 + C - W_0, 0)
= \begin{cases} 
C - D_{01} & \text{when } C > D_{01} \\
0 & \text{when } 0 \leq C \leq D_{01}
\end{cases} \quad \text{(Case 1)}
\]

(6.1)
Similarly, the idle time for the first cycle for every processor other than PE0, \( I_{i>0}(1) \), is derived as follows:

\[
I_{i>0}(1) = \max\left(\sum_{j \neq i} W_j + C - W_i, 0\right) = \max(0 + C - W_i, 0) = C + D_{0i}
\]

(6.2)

Notice that the second step in both derivations above rely on the assumption that \( W_i \geq W_{i+1} \) for all \( i \geq 0 \).

6.3.1 Case 1: \( C > D_{01} \)

Having calculated the idle times for the first cycle, we can now calculate the total time accumulated on each processor. In general, the accumulated time through \( j \) cycles on PE\(_i\), \( T_i(j) \), will be the accumulated time through \( j - 1 \) cycles, \( T_i(j - 1) \), plus the non-idle time for this cycle, \( W_i \), plus the idle time for this cycle, \( I_i(j) \). That is, the general equation for accumulated time on processor \( i \) through \( j \) cycles is:

\[
T_i(j) = T_i(j - 1) + W_i + I_i(j)
\]

where we define \( T_i(0) = 0 \). Hence, for cycle 1, the accumulated time equations are given by:

\[
T_0(1) = 0 + W_0 + (C - D_{01}) = C + W_1
\]

(6.3)

\[
T_{i>0}(1) = 0 + W_i + C + D_{0i} = C + W_0
\]

(6.4)
Since $T_{i>0}(1) \geq T_0(1)$, the accumulated ensemble time $T(1) = T_{i>0}(1)$.

Next, we derive the idle time equations of the second cycle:

$$I_0(2) = \max_{i>0} (T_i(1) + W_i + C) - T_0(1) - W_0$$
$$= [(C + W_0) + W_1 + C] - (W_1 + C) - W_0$$
$$= C \quad (6.5)$$

$$I_{i>0}(2) = \max_{j \neq i, j>0} (T_j(1) + W_j + C) - T_{i}(1) - W_i$$
$$= \max_{j \neq i, j>0} (T_0(1) + W_0 + C, T_j(1) + W_j + C) - T_{i}(1) - W_i$$
$$= \max_{j \neq i, j>0} ((C + W_1) + W_0 + C, (C + W_0) + W_j + C) - T_{i}(1) - W_i$$
$$= (2C + W_0 + W_1) - (C + W_0) - W_i$$
$$= C + W_1 - W_i$$
$$= C + D_{1i} \quad (6.6)$$

The derivations of the accumulated time equations of the second cycle are as follows:

$$T_0(2) = T_0(1) + W_0 + I_0(2)$$
$$= (C + W_1) + W_0 + C$$
$$= 2(C + W_0) \quad (6.7)$$

$$T_{i>0}(2) = T_i(1) + W_i + I_i(2)$$
$$= (C + W_0) + W_i + (C + W_1 - W_i)$$
$$= 2(C + W_0) \quad (6.8)$$

Notice that $T_0(2) = T_{i>0}(2)$. This result leads to the following observations. First,
the accumulated ensemble time $T(2) = T_{i \geq 0}(2)$. Second, because all PEs finish at the same time in the second cycle, we know that all successive cycles will repeat the pattern established in the first two cycles.

Therefore, the idle wait time equations are as follows:

$$I_0(j) = \begin{cases} \frac{C - D_{01}}{2} & \text{for odd } j \\ C & \text{for even } j \end{cases}$$ (6.9)

$$I_{i>0}(j) = \begin{cases} \frac{C + D_{0i}}{2} & \text{for odd } j \\ C + D_{1i} & \text{for even } j \end{cases}$$ (6.10)

In addition, we conclude that for any even cycle, the accumulated execution time equations are:

$$T_i(j) = \frac{j}{2} * T_i(2) = \frac{j}{2} * 2(C + W_{01}) = j(C + W_{01}) \text{ for } i \geq 0 \text{ and even } j$$ (6.11)

Recall that $W_{01} = (W_0 + W_1)/2$ and is the average work load found on the two most heavily loaded processors.

For any odd cycle, we conclude the accumulated execution time equations are:

$$T_0(j) = T_0(j - 1) + T_0(1) = \frac{(j - 1)}{2} * 2(C + W_{01}) + (C + W_1) = (j - 1)(C + W_{01}) + C + W_1 = j(C + W_{01}) - H_{01} \text{ for odd } j$$ (6.12)
\[ T_{i>0}(j) = T_{i}(j-1) + T_{i}(1) \]
\[ = \frac{(j-1)}{2} \cdot 2(C + W_{01}) + (C + W_0) \]
\[ = (j-1)(C + W_{01}) + C + W_0 \]
\[ = j(C + W_{01}) + H_{01} \text{ for odd } j \]  
(6.13)

Since \( T_{i>0}(j) \geq T_{0}(j) \) for all \( j \), \( T(j) = T_{i>0}(j) \).

So the Case 1 \((C > D_{01})\) equations can be summarized as follows:

\[ I_{0}(j) = \begin{cases} 
C - D_{01} & \text{for odd } j \\
C & \text{for even } j 
\end{cases} \]  
(6.14)

\[ I_{i>0}(j) = \begin{cases} 
C + D_{0i} & \text{for odd } j \\
C + D_{1i} & \text{for even } j 
\end{cases} \]  
(6.15)

\[ T_{0}(j) = \begin{cases} 
j(C + W_{01}) - H_{01} & \text{for odd } j \\
j(C + W_{01}) & \text{for even } j 
\end{cases} \]  
(6.16)

\[ T_{i>0}(j) = \begin{cases} 
j(C + W_{01}) + H_{01} & \text{for odd } j \\
j(C + W_{01}) & \text{for even } j 
\end{cases} \]  
(6.17)

\[ T(j) = T_{i>0}(j) \]  
(6.18)

6.3.2 Case 2.1: \( \frac{D_{01}}{2} < C \leq D_{01} \)

From Equation 6.1, one can see that a case-by-case derivation is necessary. Case 1 reflects situations with \( C > D_{01} \); Case 2 reflects situations in which \( 0 \leq C \leq D_{01} \). However, Case 2 further divides into two subcases: Case 2.1 where \( \frac{D_{01}}{2} < C \leq D_{01} \) and Case 2.2 where \( 0 \leq C \leq \frac{D_{01}}{2} \). In this section we examine the reason for the subcases and derive the equations of Case 2.1.
From the previous section we know that the following equations hold when $C \leq D_{01}$:

$$I_0(1) = 0 \text{ when } C \leq D_{01}$$  \hspace{1cm} (6.19)

$$I_{i>0}(1) = C + D_{0i}$$  \hspace{1cm} (6.20)

Hence, the derivation for the accumulated time for the first cycle is as follows:

$$T_0(1) = W_0 + I_0(1)$$  
$$= W_0 + 0$$  
$$= W_0$$  \hspace{1cm} (6.21)

$$T_{i>0}(1) = W_i + I_i(1)$$  
$$= W_i + (C + D_{0i})$$  
$$= W_i + (C + W_0 - W_i)$$  
$$= C + W_0$$  \hspace{1cm} (6.22)

So for the first cycle, $T(1) = T_{i>0}(1)$ since all other processors finish $C$ time units after PE$_0$ finishes.

For the second cycle, the idle wait time equations are derived as follows:

$$I_0(2) = \max(\max_{i>0}(T_i(1) + W_i + C) - T_0(1) - W_0, 0)$$
$$= \max(\max_{i>0}((C + W_0) + W_i + C) - W_0 - W_0, 0)$$
$$= \max((C + W_0) + W_1 + C) - W_0 - W_0, 0)$$
$$= \max(2C - W_0 + W_1, 0)$$
\[ I_{i>0}(2) = \max_{j \neq i, j > 0} (T_j(1) + W_j + C - T_i(1) - W_i) \]
\[ = \max_{j \neq i, j > 0} (T_0(1) + W_1 + C, T_j(1) + W_j + C - T_i(1) - W_i) \]
\[ = \max_{j \neq i, j > 0} (W_0 + W_0 + C, (C + W_0) + W_j + C - T_i(1) - W_i) \]
\[ = \max_{j \neq i, j > 0} (2W_0 + C, W_0 + W_j + 2C) - (C + W_0) - W_i \]
\[ = W_0 - W_i \]
\[ = D_0i \]  

(6.24)

In the derivations of \( I_{i>0}(2) \), we choose the first term in the 'max' expression since by assumption we know \( C \leq D_{01} \leq D_{0j} \) for \( j > 0 \). Adding \( W_j \) to all terms of this inequality results in \( W_j + C \leq W_0 - W_1 + W_j \leq W_0 \). Hence, \( W_0 \geq W_j + C \)

From the derivation of \( I_0(2) \) we can see why Case 2 is broken into two subcases. As with all idle time equation derivations, idle time is never allowed to be negative.

Continuing with the derivations for the Case 2.1 in which \( \frac{D_{01}}{2} < C \leq D_{01} \), we derive the equations for the accumulated processor time through the second cycle as follows:

\[ T_0(2) = T_0(1) + W_0 + I_0(2) \]
\[ = W_0 + W_0 + (2C - D_{01}) \]
\[ = 2W_0 + 2C - W_0 + W_1 \]
\[ = 2(C + W_{01}) \]  

(6.25)
\[ T_{i>0}(2) = T_i(1) + W_i + I_i(2) \]
\[ = (C + W_0) + W_i + D_{0i} \]
\[ = (C + W_0) + W_i + (W_0 - W_i) \]
\[ = C + 2W_0 \]  

(6.26)

Since the quantity \( C - D_{01} \leq 0 \implies W_1 + C \leq W_0 \implies 2C + W_0 + W_1 \leq C + 2W_0 \), we know that all \( PE_i \)'s (except \( PE_0 \)) finish after \( PE_0 \). Hence \( T(2) = T_{i>0}(2) \) in addition, it is important to note that \( PE_{i>0} \) finish exactly \( D_{01} \) time units later than \( PE_0 \).

For the third cycle the idle wait time equations can be derived as follows:

\[ I_{0}(3) = \max_{i>0} (T_i(2) + W_i + C) - T_0(2) - W_0 \]
\[ = ((C + 2W_0) + W_1 + C) - 2(C + W_0) - W_0 \]
\[ = 0 \]  

(6.27)

\[ I_{i>0}(3) = \max_{j \neq i, j>0} (T_j(2) + W_j + C) - T_i(2) - W_i \]
\[ = \max_{j \neq i, j>0} (T_0(2) + W_0 + C, T_j(2) + W_j + C) - T_i(2) - W_i \]
\[ = \max_{j \neq i, j>0} (2(C + W_0) + W_0 + C, (C + 2W_0) + W_j + C) - T_i(2) - W_i \]
\[ = \max_{j \neq i, j>0} (2W_0 + W_1 + 3C, 2W_0 + W_j + 2C) - (2C + 2W_0) - W_i \]
\[ = \max_{j \neq i, j>0} (W_1 + C, W_j) + (2W_0 + 2C) - (2W_0 + C) - W_i \]
\[ = (2W_0 + W_1 + 3C) - (2W_0 + C) - W_i \]
\[ = 2C + W_1 - W_i \]
\[ = 2C + D_{1i} \]  

(6.28)
We choose the term on the left from the ‘max’ expression since \( W_i + C \geq W_j \) for \( j > 0 \).

Furthermore, the third cycle accumulated time equations are derived as follows:

\[
T_0(3) = T_0(2) + W_0 + I_0(3)
\]
\[
= 2(C + W_{01}) + W_0 + 0
\]
\[
= 2(C + W_{01}) + W_0
\]  \hspace{1cm} (6.29)

\[
T_i > 0(3) = T_i(2) + W_i + I_i(3)
\]
\[
= (C + 2W_0) + W_i + (2C + W_1 - W_i)
\]
\[
= 2(C + W_{01}) + C + W_0
\]  \hspace{1cm} (6.30)

Clearly, \( T(3) = T_i > 0(3) \). Also, at the end of the third cycle, all other processors finish exactly \( C \) time units after PE_0 finishes. This is the same behavior that occurs at the end of the first cycle and leads to the observation that the behavior of cycles two and three will be repeated for all successive pairs of cycles. Therefore, all cycles after cycle 3 exhibit the same behavior as established by cycles 2 and 3. Hence, the general equations are easily derived and are given as follows:

\[
I_0(j) = \begin{cases} 
0 & \text{for } j = 1 \\
0 & \text{for odd } j > 1 \\
2C - D_{01} & \text{for even } j
\end{cases}
\]
\[
= \begin{cases} 
0 & \text{for odd } j \\
2C - D_{01} & \text{for even } j
\end{cases}
\]  \hspace{1cm} (6.31)
Next we derive the general time equations for PEG with odd numbers of cycles:

$$T_0(j) = \begin{cases} 
W_0 & \text{for } j = 1 \\
T_0(1) + \frac{j-1}{2} \text{ (time for repeating period)} & \text{for odd } j > 1 
\end{cases}$$

$$= \begin{cases} 
W_0 & \text{for } j = 1 \\
T_0(1) + \frac{j-1}{2} (T_0(3) - T_0(1)) & \text{for odd } j > 1 
\end{cases}$$

$$= \begin{cases} 
W_0 & \text{for } j = 1 \\
W_0 + \left(\frac{j-1}{2}\right) 2(C + W_{01}) & \text{for odd } j > 1 
\end{cases}$$

$$= (j-1)(C + W_{01}) + W_0$$

$$= j(C + W_{01}) + H_{01} - C \text{ for odd } j \quad \quad (6.33)$$

Next we derive the general time equations for PEG with even numbers of cycles:

$$T_0(j) = \begin{cases} 
2(C + W_{01}) & \text{for } j = 2 \\
T_0(2) + \frac{j-2}{2} \text{ (time for repeating period)} & \text{for even } j > 2 
\end{cases}$$

$$= \begin{cases} 
2(C + W_{01}) & \text{for } j = 2 \\
2(C + W_{01}) + \frac{j-2}{2} (T_0(3) - T_0(1)) & \text{for even } j > 2 
\end{cases}$$

$$= 2(C + W_{01}) + \frac{j-2}{2} (2(C + W_{01})) \text{ for even } j$$

$$= j(C + W_{01}) \text{ for even } j \quad \quad (6.34)$$

Next we derive the general time equations for the remaining processors with odd
numbers of cycles:

\[
T_{i>0}(j) = \begin{cases} 
C + W_0 & \text{for } j = 1 \\
T_{i}(1) + \frac{j-1}{2} \text{ (time for repeating period)} & \text{for odd } j > 1
\end{cases}
\]

\[
= \begin{cases} 
C + W_0 & \text{for } j = 1 \\
C + W_0 + \frac{j-1}{2} (T_{i}(3) - T_{i}(1)) & \text{for odd } j > 1
\end{cases}
\]

\[
= C + W_0 + \frac{j-1}{2} (2(C + W_{01})) \text{ for odd } j
\]

\[
= (j - 1)(C + W_{01}) + C + W_0 \text{ for odd } j
\]

\[
= j(C + W_{01}) + H_{01} \text{ for odd } j
\]

(6.35)

Finally we derive the general time equations for the remaining processors with even numbers of cycles:

\[
T_{i>0}(j) = \begin{cases} 
C + 2W_0 & \text{for } j = 2 \\
T_{i}(2) + \frac{j-2}{2} \text{ (time for repeating period)} & \text{for even } j > 2
\end{cases}
\]

\[
= \begin{cases} 
C + 2W_0 & \text{for } j = 2 \\
(C + 2W_0) + \frac{j-2}{2} (T_{i}(3) - T_{i}(1)) & \text{for even } j > 2
\end{cases}
\]

\[
= C + 2W_0 + \frac{j-2}{2} (2(C + W_{01})) \text{ for even } j
\]

\[
= (j - 2)(C + W_{01}) + C + 2W_0 \text{ for even } j
\]

\[
= (j - 1)(C + W_{01}) + H_{01} + W_0 \text{ for even } j
\]

\[
= j(C + W_{01}) + 2H_{01} - C \text{ for even } j
\]

(6.36)

Since \(D_{01} - C \geq 0\), we get an accumulated ensemble time of \(T(j) = T_{i>0}(j)\).

So the Case 2.1 equations can be summarized as follows:

\[
I_0(j) = \begin{cases} 
0 & \text{for odd } j \\
2C - D_{01} & \text{for even } j
\end{cases}
\]

(6.37)
\[ I_i(j) = \begin{cases} 
C + D_{0i} & \text{for } j = 1 \\
D_{0i} & \text{for even } j \\
2C + D_{1i} & \text{for odd } j > 1 
\end{cases} \quad (6.38) \]

\[ T_0(j) = \begin{cases} 
j(C + W_{01}) + H_{01} - C & \text{for odd } j \\
j(C + W_{01}) & \text{for even } j 
\end{cases} \quad (6.39) \]

\[ T_{i>0}(j) = \begin{cases} 
j(C + W_{01}) + H_{01} & \text{for odd } j \\
j(C + W_{01}) + 2H_{01} - C & \text{for even } j 
\end{cases} \quad (6.40) \]

\[ T(j) = T_{i>0}(j) \quad (6.41) \]

### 6.3.3 Case 2.2: \(0 \leq C \leq \frac{D_{01}}{2}\)

In this section we consider the second subcase of Case 2. That is, we consider the equation derivations under the restriction that \(0 \leq C \leq \frac{D_{01}}{2}\). From previous sections, we already know that for this subcase, the following equations hold:

\[ I_0(1) = 0 \quad (6.42) \]

\[ I_{i>0}(1) = C + D_{0i} \quad (6.43) \]

\[ T_0(1) = W_0 \quad (6.44) \]

\[ T_{i>0}(1) = C + W_0 \quad (6.45) \]

\[ I_0(2) = 0 \quad (6.46) \]

\[ I_{i>0}(2) = D_{0i} \quad (6.47) \]
From these we derive the accumulated time equations for cycle 2:

\[ T_0(2) = T_0(1) + W_0 + I_0(2) \]
\[ = W_0 + W_0 + 0 \]
\[ = 2W_0 \quad (6.48) \]

\[ T_i>0(2) = T_i(1) + W_i + I_i(2) \]
\[ = (C + W_0) + W_i + D_{0i} \]
\[ = C + 2W_0 \quad (6.49) \]

Next, we derive the idle time equations for the third cycle:

\[ I_0(3) = \max \left( \max_{i>0} \left( T_i(2) + W_i + C \right) - T_0(2) - W_0, 0 \right) \]
\[ = \max \left( (T_i(2) + W_i + C) - 2W_0 - W_0, 0 \right) \]
\[ = \max \left( (C + 2W_0) + W_i + C) - 2W_0 - W_0, 0 \right) \]
\[ = \max \left( 2C - W_0 + W_i, 0 \right) \]
\[ = \max \left( 2C - D_{01}, 0 \right) \]
\[ = 0 \quad (6.50) \]

\[ I_{i>0}(3) = \max_{i\neq j} \left( T_j(2) + W_j + C \right) - T_i(2) - W_i \]
\[ = \max_{i\neq j, j>0} \left( (T_j(2) + W_j + W_j + C) - (C + 2W_0) - W_i \right) \]
\[ = \max_{i\neq j, j>0} \left( (2W_0) + W_0 + C, (C + 2W_0) + W_j + C) - (C + 2W_0) - W_i \right) \]
\[ = \max_{i\neq j, j>0} \left( C + 3W_0, 2C + 2W_0 + W_j) - (C + 2W_0) - W_i \right) \]
\[ = \max_{i\neq j, j>0} \left( W_0, C + W_j) + (C + 2W_0) - (C + 2W_0) - W_i \right) \]
\[ = W_0 + (C + 2W_0) - (C + 2W_0) - W_i \]
\[ = D_{0i} \quad (6.51) \]

Since \( C \leq D_{01}/2 \) for Case 2.2, the idle time on PE0 for the third cycle is zero. For \( I_{i>0}(3) \), the left hand term of the 'max' expression is chosen because the condition of Case 2.2: \( C \leq D_{01}/2 \) implies that \( 2C + W_1 \leq W_0 \). Since \( C + W_j \leq 2C + W_j \leq 2C + W_1 \) for \( j > 0 \), then we know \( C + W_j \leq W_0 \).

Finally, we derive the accumulated time equations for the third cycle:

\[
T_0(3) = T_0(2) + W_0 + I_0(3) \\
= (2W_0) + W_0 + 0 \\
= 3W_0 \quad (6.52)
\]

\[
T_{i>0}(3) = T_i(2) + W_i + I_{i}(3) \\
= (C + 2W_0) + W_i + D_{0i} \\
= C + 3W_0 \quad (6.53)
\]

In this case, a clear pattern exists. For any processor and cycle, the next cycle will complete \( W_0 \) time units later. Hence, we conclude from observation that the following general equations are true for Case 2.2:

\[
I_0(j) = 0 \quad (6.54)
\]

\[
I_{i>0}(j) = \begin{cases} 
C + D_{0i} & \text{for } j = 1 \\
D_{0i} & \text{for } j \geq 2 
\end{cases} \quad (6.55)
\]

\[
T_0(j) = jW_0 \quad (6.56)
\]
We refer the reader to [50] for a two processor, 4-cycle example of Case 2.2 using the specific values of $W_0 = 20$, $W_1 = 12$, and $C = 3$.

### 6.4 Potential Speedup

#### 6.4.1 Perfectly balanced case

Before we can use the equations from Section 6.3 to calculate speedup, we must first determine the ensemble time equation for the perfectly balanced case. In this ideal case, each processor executes an equal amount of non-idle work and then each broadcasts a message. For example, the perfectly balanced case corresponding to the program shown in Figure 6.1 is illustrated in Figure 6.2. Since the communication transit delay, $C$, is assumed to be constant, all messages will be received simultaneously after a delay of $C$. Therefore, the accumulated ensemble time equation for the perfectly balanced case for $n$ processors for $j$ cycles is given by:

$$T_{avg}(j) = j \left( \frac{\sum_{k=0}^{n-1} W_k}{n} + C \right) = j(W_{avg} + C) \quad (6.59)$$

Some additional notation must also be introduced. We use $D_{2-avg}$ to denote the average workload on the two busiest processors minus the average system work-
Figure 6.2: Balanced counterpart to Figure 1
load. Mathematically, we mean that $D_{2-avg} = W_{01} - W_{avg} = (W_0 + W_1)/2 - (\sum_{k=0}^{n-1} W_k)/n$. Figure 6.3 may be of help to the reader at obtaining an intuitive feel for the meaning of both $D_{2-avg}$ and some variables introduced in Section 2.

Figure 6.3: Illustration of meanings of abstract variables

A second notational remark has to do with communication cost, $C$. Because $C$ is restricted to a range of values for each of the three cases, it is difficult to make comparisons between the three cases. The solution is to select one case as a frame of reference, and then to describe $C$ in terms of this frame of reference. As Figure 6.4 shows, we have chosen to denote a communication value in the Case 2.2 range by $C_A$. 

Then communication with the same offset in Case 2.1 can be denoted as $C_A + H_{01}$. Similarly, communication in the Case 1 situation can be denoted as $C_A + kH_{01}$ for $k \geq 2$. Finally, as Figure 6.4 indicates, the value of $H_{01} - C_A$ is denoted $C_B$.

Having derived the ensemble time equations for both unbalanced programs and their balanced counterparts, and given all additional needed notation, we can now derive the theoretical speedup equations for the three cases described in Section 6.3. We denote this speedup for a program that executes $j$ cycles as $S(j)$.

### 6.4.2 Case 1 speedup

Recall, from Section 6.3.1 that the accumulated ensemble time for an unbalanced, Case 1 program is given by:

$$
T(j) = T_{i>0}(j) = \begin{cases} 
  j(C + W_{01}) + H_{01} & \text{for odd } j \\
  j(C + W_{01}) & \text{for even } j
\end{cases}
$$

(6.60)

Since the equations for odd and even cycles differ, we derive the speedup for
each separately. First we derive the speedup equation for odd numbers of cycles:

\[
S(j) = \frac{\text{Unbalanced execution time for odd cycles}}{\text{Balanced execution time}} = \frac{T(j)}{T_{avg}(j)}
\]

\[
= \frac{j(C + W_{01}) + H_{01}}{T_{avg}(j)}
\]

\[
= \frac{j(C + W_{01}) + H_{01} + T_{avg}(j) - T_{avg}(j)}{T_{avg}(j)}
\]

\[
= 1 + \frac{j(C + W_{01}) + H_{01} - T_{avg}(j)}{T_{avg}(j)}
\]

\[
= 1 + \frac{j(C + W_{01}) + H_{01} - j(W_{avg} + C)}{T_{avg}(j)}
\]

\[
= 1 + \frac{jC + jW_{01} + H_{01} - jW_{avg} - jC}{T_{avg}(j)}
\]

\[
= 1 + \frac{j(W_{01} - W_{avg}) + H_{01}}{T_{avg}(j)}
\]

\[
= 1 + \frac{jD_{2-avg} + H_{01}}{j(W_{avg} + C)}
\]

\[
= 1 + \frac{D_{2-avg} + \frac{1}{j}H_{01}}{W_{avg} + C} \text{for odd } j
\]

The derivation of the speedup for the even case is as follows:

\[
S(j) = \frac{\text{Unbalanced execution time for even case}}{\text{Balanced execution time}} = \frac{T(j)}{T_{avg}(j)}
\]

\[
= \frac{j(C + W_{01})}{T_{avg}(j)}
\]

\[
= \frac{j(C + W_{01}) + T_{avg}(j) - T_{avg}(j)}{T_{avg}(j)}
\]

\[
= 1 + \frac{j(C + W_{01}) - T_{avg}(j)}{T_{avg}(j)}
\]
6.4.3 Case 2.1 speedup

In Section 6.3.2, it is shown that the accumulated ensemble time for an unbalanced, Case 2.1 program is given by:

\[
T(j) = T_{i>0}(j) = \begin{cases} 
  j(C + W_{01}) + H_{01} & \text{for odd } j \\
  j(C + W_{01}) + 2H_{01} - C & \text{for even } j 
\end{cases}
\]  

(6.63)

Once again, both odd and even cycle cases exist. However, for odd numbers of cycles, the value of \(T(j)\) is the same for both Case 1 and Case 2.1. Therefore, the speedup will be the same as is derived in Section 6.4.2. So for the odd case, the speedup equation is as follows:

\[
S(j) = 1 + \frac{D_{2-\text{avg}} + \frac{1}{j}H_{01}}{W_{\text{avg}} + C} \quad \text{for odd } j
\]  

(6.64)

The derivation of the speedup for the even case is as follows:

\[
S(j) = \frac{\text{Unbalanced execution time for even cycles}}{\text{Balanced execution time}} = \frac{T(j)}{T_{\text{avg}}(j)} = \frac{j(C + W_{01}) + 2H_{01} - C}{T_{\text{avg}}(j)}
\]
\[ j(C + W_{01}) + 2H_{01} - C + T_{\text{avg}}(j) - T_{\text{avg}}(j) \]
\[ = 1 + \frac{j(C + W_{01}) + 2H_{01} - C - T_{\text{avg}}(j)}{T_{\text{avg}}(j)} \]
\[ = 1 + \frac{j(C + W_{01}) + 2H_{01} - C - j(W_{\text{avg}} + C)}{T_{\text{avg}}(j)} \]
\[ = 1 + \frac{jC + jW_{01} + 2H_{01} - C - jW_{\text{avg}} - jC}{T_{\text{avg}}(j)} \]
\[ = 1 + \frac{j(W_{01} - W_{\text{avg}}) + 2H_{01} - C}{T_{\text{avg}}(j)} \]
\[ = 1 + \frac{jD_{2-\text{avg}} + H_{01} - C_A}{j(W_{\text{avg}} + C)} \]
\[ = 1 + \frac{D_{2-\text{avg}} + \frac{1}{j}(H_{01} - C_A)}{W_{\text{avg}} + C} \]
\[ = 1 + \frac{D_{2-\text{avg}} + \frac{1}{j}C_B}{W_{\text{avg}} + C} \text{ for even } j \quad (6.65) \]

### 6.4.4 Case 2.2 speedup

In Section 6.3.3, it is shown that the accumulated ensemble time for an unbalanced, Case 2.2 program is given by:

\[ T(j) = T_{i>0}(j) \]
\[ = jW_0 + C \quad (6.66) \]

This is a very simple case to derive speedup since odd and even cycle cases do not need to be handled separately. The derivation for Case 2.2 speedup is as follows:

\[ S(j) = \frac{\text{Unbalanced execution time}}{\text{Balanced execution time}} = \frac{T(j)}{T_{\text{avg}}(j)} \]
Where \( C \) equals \( C_A \) in this case.

Table 6.1\(^5\) contains a summary of all important equations derived in Sections 3 and 4.

\( ^5\)Unless otherwise specified, entries with two equations have the 'odd \( j \)' case on top and the 'even \( j \)' case on the bottom.
Table 6.1: Summary of generalized equations

|       | Case 1  
|---|---
| $2H_{01} < C$ | $H_{01} < C \leq 2H_{01}$ | $0 \leq C \leq H_{01}$ |
| $S(j)$ | $1 + \frac{D_{2-avg} + \frac{1}{2} H_{01}}{W_{avg} + C}$ | $1 + \frac{D_{2-avg} + \frac{1}{2} H_{01}}{W_{avg} + C}$ | $1 + \frac{D_{2-avg} + C_B - \frac{1}{2} C}{W_{avg} + C}$ |
| $T_0(j)$ | $\frac{j(C + W_{01}) - H_{01}}{j(C + W_{01})}$ | $\frac{j(C + W_{01}) + H_{01} - C}{j(C + W_{01})}$ | $jW_0$ |
| $T_{1>0}(j)$ | $\frac{j(C + W_{01}) + H_{01}}{j(C + W_{01})}$ | $\frac{j(C + W_{01}) + H_{01}}{j(C + W_{01}) + 2H_{01} - C}$ | $jW_0 + C$ |
| $T_{avg}(j)$ | $j(W_{avg} + C)$ | | |
| $T(j)$ | | $T_{1>0}(j)$ | |
| $I_0(j)$ | $\frac{C - D_{0i}}{C}$ | $0$ | $2C - D_{0i}$ | $0$ |
| $I_{1>0}(j)$ | $\frac{C + D_{0i}}{C + D_{1i}}$ | $\frac{C + D_{0i}}{D_{0i}}$ (even $j$) | $\frac{C + D_{0i}}{2C + D_{1i}}$ (odd $j \geq 3$) | $\frac{C + D_{0i}}{D_{0i}}$ ($j = 1$) | $\frac{C + D_{0i}}{D_{0i}}$ ($j \geq 2$) |
By taking the limit of \( S(j) \) as \( j \) approaches infinity, one can obtain the steady-state equations for speedup as well. Table 6.2 contains the list of these values.

Table 6.2: Steady-state values of speedup

<table>
<thead>
<tr>
<th>Case 1 ( 2H_{01} &lt; C )</th>
<th>Case 2.1 ( H_{01} &lt; C \leq 2H_{01} )</th>
<th>Case 2.2 ( 0 \leq C \leq H_{01} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lim_{j \to \infty} S(j) )</td>
<td>( 1 + \frac{D_{2-avg}}{W_{avg} + C} )</td>
<td>( 1 + \frac{D_{2-avg} + C_B}{W_{avg} + C} )</td>
</tr>
</tbody>
</table>

Table 6.3 contains a summary of the variables used in this chapter. All entries apply to one cycle only unless otherwise noted.

6.5 Verification of Equations

Now that the generalized equations have been derive, we verify their accuracy with an execution trace simulation. Since there is a vast number of possible situations to check, we are forced to take certain liberties. Instead of running a program that checks all cases within a given set of ranges as was done in Section 3.4, we instead rely on the use of a random number generator.

We use the random number generator to select for a given run, the following values:

- the number of processors, \( n > 2 \)
- the workload on each of the remaining \( n - 2 \) processors
We assume $W_0$ is fixed at 100 time units. Once again, the value of $W_1$ is varied from the integers 0 through 100, inclusive. Also, the communication value $C$ is again varied from the integer values 0 to 199, inclusive. The number of cycles, $j$, is varied from 1 to 200, inclusive. However, to keep the number of test cases reasonable, we use the random number generator to select the remaining values as mentioned above. The maximum number of processors for any given run is limited at an upper bound of 100.

The results of the simulation show that the equations are correct within an accumulated error of 0.000000 time units. The error is calculated as the accumulation of the absolute value of the difference in experimentally determined values for the timing and idle time equations for each processor and each cycle.

A table is also included in the output. Each column of the table represents groups of 10 processors and the row is divided into the 5 different communication cases:

- Case 1 – odd number of cycles
- Case 1 – even number of cycles
- Case 2.1 – odd number of cycles
- Case 2.1 – even number of cycles
- Case 2.2

The Case 2.2 is not broken down into odd and even cycles since both use the same equations. Entries in the table indicate how many cycles were tested for a given range of processors and communication case.
The simulation was run for 4,040,000 different cycles which checks 404,000,000 instances of the predicted equation values against actual values calculated in the simulation. A check of the table shows that all major communication case and processor number ranges are well represented in the simulation. Both the code and the output for the simulation can be found in Appendix D.

6.6 Summary

In this chapter, we have successfully demonstrated how time-stepped, deterministically load unbalanced programs can be modeled with the Generalized Work/Exchange Model for arbitrary numbers of processors. By use of this model, we were also able to derive exact timing and speedup equations for these same programs. Input to the equations consists of the work per cycle for each processor, the communication delay, the total number of cycles, and the total number of processors.

In [50] we were able to show the clear influence of $C$ on the timing and speedup equations. To summarize those results, as $C$ approached zero in the Case 2.2 equations, $T(j)$ would decrease resulting in a possible speedup of $n$ on $n$ processors. However, as $C$ was forced to be larger as in the other two cases, potential speedup was significantly limited. This was particularly true for the Case 1 situation. The reason for this was that as $C$ became large, both processors would go idle while waiting for a message to arrive. Hence, it did not really matter how much the processors were unbalanced since they both had time to 'waste'.

We believe that similar results hold for the arbitrary $n$-valued equations. However, at the time of this writing, we have not completely worked out the speedup limit equations. The speedup limit equation for Case 2.2 is readily apparent, though.
By placing all the work on PE₀ and letting C = 0, it is easy to see that a speedup of n occurs for n processors. For Case 2.2 when C ≠ 0 and for the other cases, speedup is affected by the value of j and we believe that increasing j will rapidly decrease the maximum obtainable speedup. This observation shows the common practice in the literature of estimating speedup by examining only first cycle execution times generally causes one to drastically over estimate the potential speedup available through load balancing.

Another conclusion that we can clearly make is that overlap of work and communication does have significant effects on execution times of unbalanced programs. Therefore, this overlap should not be ignored when considering the effects of load balancing.

Our final conclusion is in regard to the complexity of the equations. When we first started this research, we assumed that the equations for such a simple model would be trivial. Instead, the equations turned out to be quite complex. Only by assuming that communication was performed as a broadcast, were we able to complete the derivation. However, in spite of this limitation, we feel that the development of these equations lays a firm foundation upon which further study can be made. In particular, we hope to utilize these results to further study both the nature of what load imbalance is and the surprising and sometimes counterintuitive affects that arise from the overlap of load imbalance and communication.
Table 6.3: Summary of variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Alternate Form</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_{2-avg}$</td>
<td>$W_{01} - W_{avg}$</td>
<td>Average workload on two busiest processors minus the average system workload</td>
</tr>
<tr>
<td>$D_{ij}$</td>
<td>$W_i - W_j$</td>
<td>Difference of workload on PE$_i$ and PE$_j$</td>
</tr>
<tr>
<td>$H_{01}$</td>
<td>$D_{01}/2$</td>
<td>Half the difference of work on the busiest 2 processors</td>
</tr>
<tr>
<td>$I_i(j)$</td>
<td></td>
<td>Idle time on proc. $i$ during the $j^{th}$ cycle</td>
</tr>
<tr>
<td>$j$</td>
<td></td>
<td>Number of cycles</td>
</tr>
<tr>
<td>$\lim_{j \to \infty} S(j)$</td>
<td></td>
<td>Asymptotic speedup</td>
</tr>
<tr>
<td>$n$</td>
<td></td>
<td>Number of processors</td>
</tr>
<tr>
<td>$S(j)$</td>
<td>$\frac{T(j)}{T_{avg}(j)}$</td>
<td>Speedup of balanced versus unbalanced programs after $j$ cycles</td>
</tr>
<tr>
<td>$T(j)$</td>
<td>$\max_{i \geq 0}(T_i(j))$</td>
<td>Exec. time for all proc. after $j$ cycles</td>
</tr>
<tr>
<td>$T_{avg}(j)$</td>
<td></td>
<td>Time after $j$ cycles for balanced prog.</td>
</tr>
<tr>
<td>$T_i(j)$</td>
<td></td>
<td>Execution time on PE$_i$ after $j$ cycles</td>
</tr>
<tr>
<td>$W_{01}$</td>
<td>$\frac{W_0 + W_1}{2}$</td>
<td>Average load on busiest two processors</td>
</tr>
<tr>
<td>$W_{avg}$</td>
<td>$\frac{1}{n} \sum_{k=0}^{n-1} W_k$</td>
<td>Average system workload</td>
</tr>
<tr>
<td>$W_i$</td>
<td></td>
<td>Workload on processor $i$</td>
</tr>
</tbody>
</table>
7. A FORMAL DEFINITION OF LOAD OPTIMIZATION

From the literature survey presented in Chapter 2, we see that no formal definition of load balance exists. However, many papers rely on a nebulous definition based on the viewpoint that a balanced program must have an equal distribution of workload on every utilized processor. In Chapters 3 through 6 we find that this naive definition is inadequate, and is the source of many myths regarding load imbalance.

In the next section, we formalize the naive, de facto definitions that many people assume are correct. Then in Section 7.2, we present an alternative definition that does not suffer from the problems of the de facto definition. We call our new term load optimization.

7.1 De Facto Definition

In the 46 referenced works of this dissertation of which I am not an author, no formal definitions exist to describe measures of balance or imbalance. However, there clearly exist de facto definitions based on the premise that a balanced program must have an equal distribution of workload on every processor that is being utilized by a parallel program. This definition is based on the idea that program execution time

When we discuss unbalanced programs and their balanced counterparts, we assume that the same number of processors are utilized. Hence, balancing in this context refers to the redistribution of existing subtasks. To do otherwise would permit
is equivalent to the time required to execute the workload on the most heavily loaded processor. So by equalizing workload on all processors, the execution is thought to be minimized.

Let us first consider a measure of load imbalance for some arbitrary program. Assume that the program has been divided into several subtasks and mapped or distributed onto the processors of a parallel architecture. First, we define workload as the *amount of time required for a quantity of useful work to be executed by the processor onto which it has been mapped*. We do not consider communication time or inter-task synchronization time to be included in a processor's workload.

Suppose a program decomposition and mapping can be found such that every processor has an identical workload. We refer to such a program decomposition and mapping as being *perfectly balanced*. Without loss of generality, let us normalize the workload of any perfectly balanced case program to 1 time unit. Furthermore, assume that a perfectly balance case can always be found for an arbitrary program.

Informally, to get a measure of imbalance, we take the ratio of the maximum workload of an arbitrary program distribution and mapping to the maximum workload of the same program's perfectly balanced counterpart.

To formalize this concept, we first must introduce some notation. Let $\mathcal{P}$ represent an arbitrary program and $D(\mathcal{P})$ represent a particular distribution and mapping of $\mathcal{P}$. Let $D_p(\mathcal{P})$ denote the perfectly balanced distribution and mapping of $\mathcal{P}$. Let $W_{max}(x)$ denote the maximum workload of some program distribution, $x$. Notice in the case of $D_p(\mathcal{P})$ that every processor will have the same workload so the notation $W_{max}(D_p(\mathcal{P}))$ can be simplified to $W(D_p(\mathcal{P}))$.
Next, we formalize the definition of the measure of imbalance as follows:

**Definition:** The measure of imbalance for $D(\mathcal{P})$ is the ratio of the maximum workload in $D(\mathcal{P})$ to the workload that would exist in $D_P(\mathcal{P})$.

So given the above restrictions, we can express this concept mathematically as:

$$\text{measure of imbalance of } D(\mathcal{P}) = \frac{W_{max}(D(\mathcal{P}))}{W(D_P(\mathcal{P}))}$$

where $P$ is some arbitrary program. Notice that the measure of imbalance is a unitless quantity.

For example, consider a program distributed onto 3 processors with workloads of 10 time units, 1 time unit, and 1 time unit, respectively. Then the perfectly balanced counterpart will have workloads of 4 time units, 4 time units, and 4 time units, respectively. So in this case, the measure of imbalance is $10$ time units / $4$ time units = 2.5. Also, if such issues as communication delay time are ignored, then this program will take 2.5 times longer to execute than its balanced counterpart.

So having defined the measure of imbalance, we now define the **measure of balance** as:

**Definition:** The measure of balance for $D(\mathcal{P})$ is the ratio of the workload that would exist in $D_P(\mathcal{P})$ to the maximum workload in $D(\mathcal{P})$.

In other words, the measure of balance is the reciprocal of the measure of imbalance.

Referring back to our example with a measure of imbalance of 2.5, the measure of balance is 1 time unit / 2.5 time units. That is, the unbalanced program is 40% balanced and can be speeded up 2.5 times.
7.2 Alternative Definition

The problem with the definitions given in the previous section is that they fail to take into account other factors that can affect execution time. For example, they ignore communication delay, network contention, and the idle times that arise due to inter-task data dependencies. Instead, they focus solely on reaching a perfectly balanced distribution and mapping of only the useful work (i.e. workload) of a program.

In this section, we propose a new formalism called load optimization. First, we give an informal definition by observing that an 'unbalanced' program distribution and mapping can be put into 'balance' through any arbitrary program decomposition and mapping that results in a minimized execution time. The focus of our new definition is on minimizing execution time. This contrasts with the naive definition that focuses on equal workload distribution. Under the naive definition of load imbalance, we can actually have unbalanced programs that execute in less time than their balanced counterparts. However, we cannot have unbalanced programs that execute in less time than their minimized counterparts.

We define a program’s minimized counterpart as the optimal program decomposition and mapping that leads to minimal execution time.

Again, let $\mathcal{P}$ represent an arbitrary program and $D(\mathcal{P})$ represent a particular distribution and mapping of $\mathcal{P}$. Let $D_{opt}(\mathcal{P})$ denote the minimized counterpart of $\mathcal{P}$. Let $T(x)$ denote the time required to execute some program distribution and mapping, $x$, on some arbitrary parallel architecture.
We formalize our definition of load optimization:

**Definition:** The measure of load optimization for $D(P)$ is the ratio of the time required to execute $D(P)$ on some arbitrary parallel architecture to the time required to execute the minimized counterpart of $P$ on the same parallel architecture.

Expressed mathematically we get the following:

$$\text{measure of load optimization of } D(P) = \frac{T(D(P))}{T(D\text{opt}(P))}$$

### 7.3 Remarks

An important point is that load balancing and load optimizing are fundamentally different concepts. Furthermore, when a person wants to reduce the execution time of a program, that person should strive for a program that is load optimized but not necessarily load balanced.

This point is clearly demonstrated in Chapter 4 in the culmination of three myths. In that chapter, various cases of unbalanced programs are compared with their balanced counterparts. In each case, we find that the unbalanced version executes in the same or less time than its balanced counterpart. If we were to compare the unbalanced versions to their minimized counterparts instead, we would find that the minimized versions would always execute in the same or less time. Therefore, we propose that henceforth, all discussion of load balance be done in the context of load optimization.
8. CONCLUSIONS AND FUTURE WORK

8.1 Conclusions

One of the primary goals stated at the outset of this dissertation was to learn more about the nature of load imbalance. A second goal was to apply what we had learned to speeding up parallel programs.

We began with the literature survey presented in Chapter 2. From the literature survey, we discovered several things:

- load imbalance had never been formally defined,
- most past work has focused on heuristic algorithms for dynamic load balancing,
- a premise behind the algorithms was that an equal redistribution of work was best,
- very little objective analysis was available to rate the algorithms, and
- even a purely random approach to load balancing seemed to do about as well as the proposed heuristics.

The lack of objective results and a poor theoretical foundation led us to develop the Work/Exchange Model as presented in Chapter 3. In this theoretical model, we were able to exactly characterize the behavior of a time-stepped program distributed...
on two processors. Of course, the model assumed that the program meant some rather stringent requirements. Nevertheless, by using the model, we were able to derive exact equations for elapsed time and idle time for each processor after an arbitrary number of cycles of conducting work and exchanging messages. We were also able to develop exact speedup equations were speedup was defined as the ratio of the programs execution time to the execution time of its perfectly balanced counterpart. Through use of simulation, we were able to verify the correctness of the equations.

One major benefit of having developed the Work/Exchange Model was that we then had a tool that allowed us greater insight into load imbalance. This led to the development of the three myths presented in Chapter 4. The essence of these myths was to discredit the premise that a program will always be speeded up by redistributing its workload into equal quantities. They also made it clear that such things as communication delay time, network contention, and the structure of the architecture itself, all played a role in determining the optimal load distribution.

Then in Chapter 5, we took a break from the theory by introducing a practical application derived from the Work/Exchange Model. We called this application an enhancer for dynamic load balancers. The function of the enhancer was twofold:

1. to decide if it would actually be beneficial to invoke a load balancer when a program requests balancing

2. to tune the period of invocation of a load balancer

In essence, the enhancer's role was to balance the load balancer. In an implementation of the nCUBE 2 hypercube, we found that the use of the enhancer did indeed result in additional speedup over use of a load balancer alone.
In the experiment we were limited to only two processors since the enhancer was based on the Work/Exchange Model. So our next goal was to generalize the model for an arbitrary number of processors. The fulfillment of this work is chronicled in Chapter 6. We chose to call the improved model, the Generalized Work/Exchange Model. Again, as with the original model, exact equations were developed for elapsed time, idle time, and potential speedup for each of an arbitrary number of processors running a program for an arbitrary number of cycles. The catalyst that allowed the generalization to take place was the introduction of one additional assumption. The assumption was that the communication phase could be performed as a broadcast message. As before, we also included an exhaustive simulation study to verify the correctness of the equations.

Finally, in Chapter 7 we were able to lay the capstone of the entire thesis. In this chapter, we first presented the de facto definition of load imbalance. This was then followed by our new and improved version which took into account the issues that contributed to the myths of Chapter 4 using the old definition. However, instead of calling the new definition a measure of imbalance, we instead called it a measure of load optimization. We felt that the distinction was necessary since imbalance had such a strong connotation. Chapter 7 then concludes with the observation that load balance and load optimization are distinctly different; a program can be balanced and still be not optimized. However, a load optimized program is always guaranteed to have minimal execution time.
8.2 Future Work

Next, we outline areas for future research. First, the author needs to complete derivation of the speedup limit equations discussed above. Second, additional thought is needed to determine why the equations behave as they do. Third, generalizing the model so that a communication broadcast need not be assumed would be desirable (but is probably intractable). Finally, implementation work is needed to show how these equations can be applied to less restricted, 'real-world' cases. In [48, 50], we have shown how static equations for two processors and the concept of temporal locality could be combined to develop an effective load balancer enhancer for two processors. In this report, we were able to derive the equations to model an arbitrary number of processors. However, questions remain as to whether or not the coupling of the equations and temporal locality will hold for more than two processors. Possible future work could include testing the feasibility of developing load balancer enhancers for arbitrary numbers of processors.
BIBLIOGRAPHY


ACKNOWLEDGMENTS

This work was performed in part at Ames Laboratory under contract number W-7405-eng-82 with the U. S. Department of Energy. The United States government has assigned the DOE Report number IS-T-1589 to this thesis.

I would like to thank my friend and advisor, Dr. G. M. Prabhu, for his guidance, patience, and encouragement. During some of the darker days of graduate school, he was one of the few that displayed an unwavering faith in my abilities.

Dr. John L. Gustafson also played a crucial role in the completion of this work. Not only did he provide outstanding guidance and prodding, he also stepped in with an effort equivalent to that of a full advisor. John also showed me the true meaning of the words “research excellence”. John’s better-half, Denise, was also helpful in keeping everything running smoothly. Her job search efforts on my behalf are also appreciated and allowed me more time to finish this dissertation when the “crunch” was on.

Dr. S. Kothari, Dr. Charlie Wright, and Dr. Gene Takle are also deserving of recognition for their suggestions and guidance and for their roles as members of my POS committee. Their contributions were certainly significant. It is a shame that committee members like these do not receive the recognition they deserve.

Next, I would like to thank the staff and my fellow students in the Computer Science Department and at Ames Laboratory including, but not limited to: David Coleman, Linda Null, and Susan Stolfus. Your friendship and assistance made the stay at Iowa State University much more bearable. Also, a special note of thanks is due to Michael Carter for his reading of drafts of a paper that later became Chapter 6 of this thesis.

Finally, I give a hearty thanks to my parents, Drs. Gunnar and Marilyn Wikstrom, and wife of six weeks, Teri Rolene Wikstrom, for their love, support, and encouragement.
Simulation Program: simver.c

#include <stdio.h>
#define MAXJ 100

main()
{
    float i0[MAXJ], i1[MAXJ], t0[MAXJ], t1[MAXJ];
    float error=0.0, pi0, pi1, pt0, pt1;
    int j, c, w0, w1;
    int i;
    j = 100;
    w0 = 100;
    for (w1=0; w1<w0; w1++)
    {
        for (c=0; c<2*w0; c++)
        {
            i0[0] = i1[0] = t0[0] = t1[0] = 0.0;
            for (i=1; i<=j; i++)
            {
                t0[i] = max(t0[i-1] + w1 + c, t0[i-1] + w0);
                t1[i] = max(t0[i-1] + w0 + c, t1[i-1] + w1);
                i0[i] = max(t0[i] - w0 - t0[i-1], 0.0);
                i1[i] = max(t1[i] - w1 - t1[i-1], 0.0);
            }
            predict(w0,w1,i,c,&pt0,&pt1,&pi0,&pi1);
            error += (t0[i] >= pt0 ? t0[i]-pt0 : pt0-t0[i] );
            error += (t1[i] >= pt1 ? t1[i]-pt1 : pt1-t1[i] );
            error += (i0[i] >= pi0 ? i0[i]-pi0 : pi0-i0[i] );
        }
    }
}

/*
    printf("w0 = %d, w1 = %d, and c = %d\n",w0,w1,c);
    printf(" j t0[i] t1[i] i0[i] i1[i] \n");
    printf(" --- -------- -------- -------- --------\n");
*/
    for (i=1; i<=j; i++)
    {
        t0[i] = max(t1[i-1] + w1 + c, t0[i-1] + w0);
        t1[i] = max(t0[i-1] + w0 + c, t1[i-1] + w1);
        i0[i] = max(t0[i] - w0 - t0[i-1], 0.0);
        i1[i] = max(t1[i] - w1 - t1[i-1], 0.0);
    }
    /*
    printf("%3d %7.2f %7.2f %6.2f %6.2f \n",i,t0[i],t1[i],i0[i],i1[i]);
*/
predict(w0,w1,i,c,&pt0,&pt1,&pi0,&pi1);
error += (t0[i] >= pt0 ? t0[i]-pt0 : pt0-t0[i] );
error += (t1[i] >= pt1 ? t1[i]-pt1 : pt1-t1[i] );
error += (i0[i] >= pi0 ? i0[i]-pi0 : pi0-i0[i] );
```c
error += (il[i] >= pil ? il[i]-pil : pil-il[i] );
/*
 * printf("%.3d %.2f %.2f %.2f %.2f

", i,pt0,pt1,pi0,pi1);
 */
}
}
printf("Sum |error| = %.9f w0=%.3d w1=%.3d c=%.3d\n", error,w0,w1,c);
}
max(a,b)
float a, b;
{ return( a >= b ? a : b);
}

predict(w0,w1,j,c,t0,t1,iO,i1)
int w0, w1, j, c;
float *t0, *t1, *i0, *i1;
{
    float d;
    d = w0 - w1;
    /****** Case 1: C > D ******/
    if (c > d) {
        *t0 = j*(w0+c) - ((j+l)/2) * d;
        *t1 = j*(w0+c) - (i/2) * d;
        if ((j/2) == ((j+l)/2)) { /* j is even */
            *i0 = c;
            *i1 = c;
        } else {
            *i0 = c - d;
            *i1 = c + d;
        }
    } /* Case 1: C > D ******/
    /****** Case 2.1: D/2 < C <= D ******/
    else if (c > d/2) {
        if ((j/2) == ((j+l)/2)) { /* j is even */
            *t0 = (j/2)*(2.0*w0+2.0*c-d);
            *t1 = d-c+(j/2)*(2.0*w0+2.0*c-d);
            *i0 = 2.0 * c - d;
            *i1 = d;
        } else {
            *t0 = w0+((j-1)/2)*(2*w0+2*c-d);
            *t1 = w0+c+((j-1)/2)*(2*w0+2*c-d);
            *i0 = 0.0;
            *i1 = (j == 1 ? d + c : 2.0 * c);
        }
    }
```
}  
*/***** Case 2.2: 0 <= C <= D/2  ******/
else {
    *t0 = j*w0;
    *t1 = j*w0+c;
    *i0 = 0.0;
    *i1 = (j == 1 ? d + c : d);
}  
Output

Sum |error| = 0.000000  w0=100 w1=100 c=200
APPENDIX B

nCUBE 2 equation verification program

#include<stdio.h>
#include <math.h>
#define MAXCYCLES 1024
#define MAXBUF 1024*8
#define TRUE 1
#define FALSE 0
#define MAXPARTICLES 1024

int NumParticles = 100, NumCycles = 100, DEBUG = FALSE, PERIOD = 1;
int DISTANCE_ON = FALSE;
don delayno;
double HEAT = 10.0;

struct node {
    double x, y, dx, dy;
} nodearray[MAXPARTICLES];
int iproc,mypid,myhost,lnproc,nproc;
int stillO, stilll;

char charbuf[MAXBUF], charbuf2[MAXBUF];

/*****************************/
/* data types needed by the load balancer */
struct listnode {
    double x,
    y,
    dx,
    dy;
    struct listnode *next;
    int arrayindex;
};
double oldmidb = -1.0, midb = 0.5;
/*****************************/
main(argc, argv)
int argc;
char **argv[];
{
    double mytotaltime, mycommtime, myworktime;
    double yourtotaltime, yourcommtime, yourworktime;
    double sec[2], When(), update(), dmax(), dmin3();
    int delayno, delaycount, commcount, jcount;
    int i, other, type = 1, flags = 0;
    double s0, s1, commtime, scale;
    double c,d,actual,predicted, correction;

    /*
    int nextload, i;
    nextload = PERIOD;
    /* When nextload reaches zero, invoke the loadbalancer */
    if (!nextload) {
        loadbalance();
        nextload = PERIOD;
    }
    */
    whoami(&iproc,&mypid,&myhost,&lnproc);
    nproc=(l<lnproc);
    if (iproc==1) {
        delayno = 10;
        other = 0;
    } else
        other = 1;
    init();
    /* get routine in main memory */ sec[0] = When();

    for (delaycount=0;delaycount<3;delaycount++){
        if (iproc==0) {
            if (delaycount==0)
                delayno = 10;
            else if (delaycount==1)
                delayno = 50;
            else
                delayno = 100;
        }
        printf("\n");
        for (commcount=1000;commcount<MAXBUF;commcount+=2000) {
            for (jcount=1;jcount<8;jcount++) {
                if (jcount==7) jcount=50;
                /* Synchronize processors for next timing run */
                nwrite(charbuf,0,other,type,flags);
                nread(charbuf2,0,&other,&type,NULL);
sec[0] = When();
for (NumCycles=0; NumCycles<jcount; NumCycles++) {
    /* calculate new positions for the particles */
    update();
    /* delay a fixed time */
    delay(delayno);
    /* send a message with commcount bytes */
    nwrite(charbuf, commcount, other, type, flags);
    nread(charbuf2, commcount, &other, &type, NULL);
} /* NumCycles */

sec[1] = When();
mytotaltime = sec[1] - sec[0];

/* estimate total comm transit delay (for one cycle). */
/* --- first, synchronize processors */
    nwrite(&mytotaltime, sizeof(double), other, type, flags);
    nread(&yourtotaltime, sizeof(double), &other, &type, NULL);
/* --- now, time a message exchange of desired length */
/* --- (this time will be for ONE cycle only!) */
sec[0] = When();
for (NumCycles=0; NumCycles<jcount; NumCycles++) {
    nwrite(charbuf, commcount, other, type, flags);
    nread(charbuf2, commcount, &other, &type, NULL);
}
sec[1] = When();
mycommtime = (sec[1] - sec[0])/(double)jcount;
/* --- next determine msg startup time to correct transit time */
sec[0] = When();

/*
for (NumCycles=0; NumCycles<jcount; NumCycles++) {
    nwrite(charbuf, 0, other, type, flags);
    nread(charbuf2, 0, &other, &type, NULL);
}
sec[1] = When();
correction = (sec[1] - sec[0])/(double)jcount;
*/
    nwrite(charbuf, commcount, other, type, flags);
sec[1] = When();
nread(charbuf2, commcount, &other, &type, NULL);
correction = 2.0 * (sec[1] - sec[0]);

/*
    nwrite(charbuf, 0, other, type, flags);
sec[0] = When();
nread(charbuf2, 0, &other, &type, NULL);
sec[1] = When();
correction = (sec[1] - sec[0]);
*/
    mycommtime -= correction;
/* --- exchange comm delay times and average */
write(&mycommtime,sizeof(double),other,type,flags);
read(&yourcommtime,sizeof(double),&other,&type,NULL);
commt ime = (mycommtime + yourcommtime) * 0.5;

/* estimate non-idle work time (for one cycle) */
sec[0] = When();
for (NumCycles=0;NumCycles<jcount;NumCycles++) {
    update();
    delay(delayno);
}
sec[1] = When();
myworktime = (sec[1] - sec[0])/(double)jcount;
myworktime += correction;
/* exchange non-idle times for one cycle */
write(&myworktime,sizeof(double),other,type,flags);
read(&yourworktime,sizeof(double),&other,&type,NULL);
scale = dmin3(myworktime,yourworktime,commt ime);
/* calculate predicted total time */
s0 = myworktime/scale;
s1 = yourworktime/scale;
c = commtime/scale;
d = s0 - s1;
/ Case 1: C > D **/
if (c > d) {
    predicted = jcount*(s0+c) - (jcount/2) * d;
    printf("Case 1: ");
}
/ Case 2.1: D/2 < C <= D **/
else if (c > d/2) {
    /* jcount is even */
    if ((jcount/2) == ((jcount+1)/2)) {
        predicted = d-c+(jcount/2)*(2*s0+2*c-d);
        printf("Case 2.1E: ");
    }
    else {
        predicted = s0+c+((jcount-1)/2)*(2*s0+2*c-d);
        printf("Case 2.10: ");
    }
}
/ Case 2.2: 0 <= C <= D/2 **/
else {
    predicted = jcount*s0+c;
    printf("Case 2.2: ");
}
if (iproc==0){
    actual = dmax(mytotaltime,yourtotaltime)/scale;
    printf("s0=%%7.4f s1=%%7.4f c=%%7.4f ",

"T(7.d)=%7.4f (T-P)/T : %7.4f%%
", myworktime/scale, yourworktime/scale, 
commtime/scale, jcount, actual, 
(actual-predicted)/actual*100.0);

} /* jcount */
} /* commmcnt */
} /* delaycount */

/*****************************/
initO{
int i, direction;
int srandO;
double MaxRandom = 1024.0 * 32.0;

for (i=0;i<MAXBUF;i++)
charbuf[i] = 'a';
srand((int)WhenO);
/* Alternate the random values between the 2 PEs */
if (iproc==0) randO;
for (i=0;i<NumParticles/nproc;i++){
nodearray[i].x = (double)randO/MaxRandom/2.0 + 0.5*iproc;
rando;
nodearray[i].y = (double)randO/MaxRandom;
rando;
direction = ( (double)randO/MaxRandom >= 0.5) ? 1 : -1 ;
rando;
nodearray[i].dx = direction *
(double)randO/MaxRandom/(101.0 - HEAT);
rando;
direction = ( (double)randO/MaxRandom >= 0.5) ? 1 : -1 ;
rando;
nodearray[i].dy = direction *
(double)randO/MaxRandom/(101.0 - HEAT);
rando;

/* for deterministic load unbalance,
don't allow particle to move */
nodearray[i].dx = 0.0;
nodearray[i].dy = 0.0;
}

/*****************************/
double
updateO{
int i, j;
double center, distance, distancesqr = 0;
double fabs(), nx, ny;

still0 = still1 = 0;
for (i=0; i<NumParticles/nproc; i++) {
    nx = nodearray[i].x + nodearray[i].dx;
    ny = nodearray[i].y + nodearray[i].dy;
    if (nx < 0.0) {
        nx = -nx;
        nodearray[i].dx = -nodearray[i].dx;
    } else if (nx >= 1.0) {
        nx = 2.0 - nx;
        nodearray[i].dx = -nodearray[i].dx;
    }
    if (ny < 0.0) {
        ny = -ny;
        nodearray[i].dy = -nodearray[i].dy;
    } else if (ny >= 1.0) {
        ny = 2.0 - ny;
        nodearray[i].dy = -nodearray[i].dy;
    }
    nodearray[i].x = nx;
    nodearray[i].y = ny;
    if (nx < 0.5) still0 += 1;
    else still1 += 1;
    center = (iproc == 0) ? midb/2 : (1.0 - midb)/2;
    distance = fabs(nx - center);
    distancesqr += distance * distance;
}
return(distancesqr);

$/)loadbalance()
{
    struct listnode *insertnode(), *makenode(), *makelist(),
    *header, *temp;
    struct node yournode;
    double yourx, dabs();
    int count = 0, myindex, min(), max(), other, msgtype = 10;

    /* create a linked list of particles sorted on the
     * x-coordinate ordered so that particles closest to the middle
     * boundary are first */
    header = makelist(NULL);
/* exchange info for most extreme particle */
other = (iproc + 1) % 2;
myindex = (*header).arrayindex;
nwrite(&nodearray[myindex], sizeof(struct node),
other, msgtype, 0);
nread(&yournode, sizeof(struct node), &other, &msgtype, NULL);
if (DEBUG && (iproc==0))
printf("PE%d sending %f and %f and receiving %f count = %d\n",
iproc, (*header).x, nodearray[myindex].x, yournode.x, count);

/* if the most extreme particle in each PE needs exchanged,
then loop */
while (stillfurther((*header).x, yournode.x)) {
  count += 1;

  /* update the array of particles */
  nodearray[myindex] = yournode;

  /* update the linked list */
  temp = header;
  if (DEBUG && (iproc==0))
  printlist(header);
  if (DEBUG && (iproc==0))
  header = (*header).next;
  if (DEBUG && (iproc==0))
  printlist(header);

  (*temp).x = yournode.x;
  (*temp).y = yournode.y;
  (*temp).dx = yournode.dx;
  (*temp).dy = yournode.dy;
  (*temp).next = NULL;
  /* (*temp).indexarray = same as old node */
  header = insertnode(temp, header);

  if (DEBUG && (iproc==0))
  printlist(header);
  /*
  insertnode(makenode(&yournode, myindex), header);
  free(temp);
  */

  /* exchange info for most extreme particle */
  myindex = (*header).arrayindex;
  nwrite(&nodearray[myindex], sizeof(struct node),
  other, msgtype, 0);
  nread(&yournode, sizeof(struct node), &other, &msgtype, NULL);
  if (DEBUG && (iproc==0))
  printf("PE%d sending %f and %f and receiving %f count = %d\n",
  iproc, (*header).x, nodearray[myindex].x, yournode.x, count);
} /* end while - particles have been exchanged */

/* calculate the new middle boundary */
get the x-coordinate of most extreme particle on other PE */

nwrite(&((*header).x),sizeof(double),other,msgtype,0);
nread(&yourx,sizeof(double),&other,&msgtype,NULL);

midb = ((*header).x + yourx) / 2.0;

/* print the new boundary value and the number of
 particles exchanged */
if (iproc==0)
 printf("PE%d: swapping %d particles, list size = %d midb = %f\n", 
iproc,count,getlistsize(header),midb);

/* free up nodes in the list */
while (header != NULL) {
 temp = (*header).next;
 free(header);
 header = temp;
}
}

/**/*************/
struct listnode
*makelist(header)
struct listnode *header;
{
 struct listnode *insertnode(), *makenode();
 int i;

 header = NULL;

 /* insert each node in the array into the linked list */
 for (i=0; i<(NumParticles/nproc); i++) {
  header = insertnode(makenode(&nodearray[i],i),header);
 }

 return(header);
}

/**/*************/
struct listnode
*makenode(particle,arrayindex)

struct node *particle;
 int arrayindex;
{
 struct listnode *parnode;

 /* create a listnode structure */
 parnode = (struct listnode *)malloc(sizeof(struct listnode));
/* copy the contents particle into parnode */
(*parnode).x = (*particle).x;
(*parnode).y = (*particle).y;
(*parnode).dx = (*particle).dx;
(*parnode).dy = (*particle).dy;
(*parnode).next = NULL;
(*parnode).arrayindex = arrayindex;

return(parnode);
}

/********************************************/
struct listnode
*insertnode(parnode,header)

struct listnode *parnode;
struct listnode *header;
{
    struct listnode *temp, *temp2;

    /* should have been done already, but do just in case */
    (*parnode).next = NULL;

    if (header == NULL)
        return(parnode);

    temp = header;
    temp2 = NULL;
    /* got to here ==> list has at least one node ==> at MOST one of 
     the temporary pointers can be NULL throughout following code */
    while ((temp != NULL) && stillfurther((*temp).x,(*parnode).x)) {
        temp2 = temp;
        temp = (*temp).next;
    }
    /* if at end of list then attach particle node to end of list */
    if (temp == NULL)
        (*temp2).next = parnode;

    /* if at start of list then attach list to particle node and point 
    header at particle node */
    else if (temp2 == NULL) {
        (*parnode).next = header;
        header = parnode;
    }
    /* else add particle into middle of the list */
    else {
        (*parnode).next = temp;
        (*temp2).next = parnode;
    }
    return(header);
}
/* PE0 : list sorted largest to smallest, return (x1 > x2) */
if ((iproc == 0) && (x1 >= x2))
    return(1);
else if ((iproc == 1) && (x1 <= x2))
    return(2);
return(0);

int printlist(struct listnode *header)
{
    struct listnode *temp;
    int count = 1;
    printf("printing list for PE%d\n\n",iproc);
    temp = header;
    while (temp != NULL) {
        printf("particle %d contains: %f\n",count++,(*temp).x);
        temp = (*temp).next;
    }
}

int getlistsize(struct listnode *header)
{
    struct listnode *temp;
    int count = 0;
    temp = header;
    while (temp != NULL) {
        count++;
        temp = (*temp).next;
    }
    return(count);
}

double dmin3(double a, double b, double c)
{
    if ((a<=b) && (a<=c))
        return(a);
    else if ((b<=a) && (b<=c))
        return(b);
    else
        return(c);
}
```c
{  
	/* loop delay */
	if (t, j,
	}

t = count(0);  
	/* count */

t = count(0);  
	/* count */

t = count(0);  
	/* count */

t = count(0);  
	/* count */

/******

/* count */

t = count(0);  
	/* count */

t = count(0);  
	/* count */

t = count(0);  
	/* count */

/******

/******

/******

/******

/******

/******

/******

/******

/******

/******
```
### Output

<table>
<thead>
<tr>
<th>Case</th>
<th>W0</th>
<th>W1</th>
<th>C</th>
<th>j</th>
<th>T(j)</th>
<th>(T-P)/T%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.1898</td>
<td>3.1229</td>
<td>1.0000</td>
<td>1</td>
<td>4.1624</td>
<td>-0.6585%</td>
</tr>
<tr>
<td>1</td>
<td>3.2942</td>
<td>3.1656</td>
<td>1.0000</td>
<td>2</td>
<td>8.4460</td>
<td>-0.1634%</td>
</tr>
<tr>
<td>1</td>
<td>3.2833</td>
<td>3.1834</td>
<td>1.0000</td>
<td>3</td>
<td>12.7022</td>
<td>-0.3762%</td>
</tr>
<tr>
<td>1</td>
<td>3.0577</td>
<td>2.8582</td>
<td>1.0000</td>
<td>4</td>
<td>15.7989</td>
<td>-0.2084%</td>
</tr>
<tr>
<td>1</td>
<td>2.9792</td>
<td>3.0204</td>
<td>1.0000</td>
<td>5</td>
<td>19.9174</td>
<td>-0.3066%</td>
</tr>
<tr>
<td>1</td>
<td>3.2405</td>
<td>3.1538</td>
<td>1.0000</td>
<td>6</td>
<td>25.1425</td>
<td>-0.1609%</td>
</tr>
<tr>
<td>1</td>
<td>3.0867</td>
<td>2.9665</td>
<td>1.0000</td>
<td>50</td>
<td>201.4623</td>
<td>0.0408%</td>
</tr>
<tr>
<td>1</td>
<td>1.1613</td>
<td>1.1793</td>
<td>1.0000</td>
<td>1</td>
<td>2.1747</td>
<td>0.6176%</td>
</tr>
<tr>
<td>1</td>
<td>1.2334</td>
<td>1.1920</td>
<td>1.0000</td>
<td>2</td>
<td>4.4113</td>
<td>-0.3217%</td>
</tr>
<tr>
<td>1</td>
<td>1.1718</td>
<td>1.1787</td>
<td>1.0000</td>
<td>3</td>
<td>6.5095</td>
<td>-0.1968%</td>
</tr>
<tr>
<td>1</td>
<td>1.2213</td>
<td>1.1900</td>
<td>1.0000</td>
<td>4</td>
<td>8.8122</td>
<td>-0.1173%</td>
</tr>
<tr>
<td>1</td>
<td>1.2235</td>
<td>1.1935</td>
<td>1.0000</td>
<td>5</td>
<td>11.0480</td>
<td>-0.0873%</td>
</tr>
<tr>
<td>1</td>
<td>1.2164</td>
<td>1.1902</td>
<td>1.0000</td>
<td>6</td>
<td>13.2194</td>
<td>-0.0039%</td>
</tr>
<tr>
<td>1</td>
<td>1.2220</td>
<td>1.1955</td>
<td>1.0000</td>
<td>50</td>
<td>110.3292</td>
<td>-0.0980%</td>
</tr>
<tr>
<td>1</td>
<td>1.0195</td>
<td>1.0000</td>
<td>1.2309</td>
<td>1</td>
<td>2.2465</td>
<td>-0.5327%</td>
</tr>
<tr>
<td>1</td>
<td>1.0273</td>
<td>1.0000</td>
<td>1.2382</td>
<td>2</td>
<td>4.4913</td>
<td>-0.2747%</td>
</tr>
<tr>
<td>1</td>
<td>1.0255</td>
<td>1.0000</td>
<td>1.2368</td>
<td>3</td>
<td>6.7396</td>
<td>-0.3205%</td>
</tr>
<tr>
<td>1</td>
<td>1.0204</td>
<td>1.0000</td>
<td>1.2312</td>
<td>4</td>
<td>8.9536</td>
<td>-0.1342%</td>
</tr>
<tr>
<td>1</td>
<td>1.0200</td>
<td>1.0000</td>
<td>1.2363</td>
<td>5</td>
<td>11.2169</td>
<td>-0.2213%</td>
</tr>
<tr>
<td>1</td>
<td>1.0000</td>
<td>1.0070</td>
<td>1.2573</td>
<td>6</td>
<td>13.5736</td>
<td>0.0639%</td>
</tr>
<tr>
<td>1</td>
<td>1.0204</td>
<td>1.0000</td>
<td>1.2331</td>
<td>50</td>
<td>111.9622</td>
<td>-0.1796%</td>
</tr>
<tr>
<td>1</td>
<td>1.0147</td>
<td>1.0000</td>
<td>1.5493</td>
<td>1</td>
<td>2.5555</td>
<td>-0.3320%</td>
</tr>
<tr>
<td>1</td>
<td>1.0263</td>
<td>1.0000</td>
<td>1.5469</td>
<td>2</td>
<td>5.1076</td>
<td>-0.2449%</td>
</tr>
<tr>
<td>1</td>
<td>1.0000</td>
<td>1.0031</td>
<td>1.5614</td>
<td>3</td>
<td>7.6799</td>
<td>-0.0968%</td>
</tr>
<tr>
<td>1</td>
<td>1.0524</td>
<td>1.0000</td>
<td>1.5968</td>
<td>4</td>
<td>10.4793</td>
<td>-0.1224%</td>
</tr>
<tr>
<td>1</td>
<td>1.0000</td>
<td>1.0088</td>
<td>1.5659</td>
<td>5</td>
<td>12.8402</td>
<td>-0.0545%</td>
</tr>
<tr>
<td>1</td>
<td>1.0189</td>
<td>1.0000</td>
<td>1.5463</td>
<td>6</td>
<td>15.3233</td>
<td>-0.0740%</td>
</tr>
<tr>
<td>1</td>
<td>1.0170</td>
<td>1.0000</td>
<td>1.5443</td>
<td>50</td>
<td>127.5217</td>
<td>-0.0900%</td>
</tr>
<tr>
<td>2.2</td>
<td>6.3378</td>
<td>3.1422</td>
<td>1.0000</td>
<td>1</td>
<td>7.3651</td>
<td>0.3712%</td>
</tr>
<tr>
<td>2.2</td>
<td>6.0045</td>
<td>2.8825</td>
<td>1.0000</td>
<td>2</td>
<td>13.2095</td>
<td>1.5175%</td>
</tr>
<tr>
<td>2.2</td>
<td>6.3344</td>
<td>3.1751</td>
<td>1.0000</td>
<td>3</td>
<td>20.5101</td>
<td>2.4720%</td>
</tr>
<tr>
<td>2.2</td>
<td>6.3002</td>
<td>3.1440</td>
<td>1.0000</td>
<td>4</td>
<td>26.8215</td>
<td>2.3144%</td>
</tr>
<tr>
<td>2.2</td>
<td>6.2957</td>
<td>3.1555</td>
<td>1.0000</td>
<td>5</td>
<td>33.3040</td>
<td>2.4779%</td>
</tr>
<tr>
<td>2.2</td>
<td>5.9359</td>
<td>2.8618</td>
<td>1.0000</td>
<td>6</td>
<td>37.5341</td>
<td>2.4479%</td>
</tr>
<tr>
<td>2.2</td>
<td>6.2546</td>
<td>3.1347</td>
<td>1.0000</td>
<td>50</td>
<td>323.3940</td>
<td>2.9886%</td>
</tr>
<tr>
<td>2.10</td>
<td>2.2374</td>
<td>1.1916</td>
<td>1.0000</td>
<td>1</td>
<td>3.3120</td>
<td>2.2 539%</td>
</tr>
<tr>
<td>2.1E</td>
<td>2.2201</td>
<td>1.1855</td>
<td>1.0000</td>
<td>2</td>
<td>5.6591</td>
<td>3.6677%</td>
</tr>
<tr>
<td>2.1E</td>
<td>2.2197</td>
<td>1.1943</td>
<td>1.0000</td>
<td>3</td>
<td>8.7786</td>
<td>1.6505%</td>
</tr>
<tr>
<td>2.1E</td>
<td>2.2282</td>
<td>1.1936</td>
<td>1.0000</td>
<td>4</td>
<td>11.1307</td>
<td>2.2 677%</td>
</tr>
<tr>
<td>2.10</td>
<td>2.2178</td>
<td>1.1913</td>
<td>1.0000</td>
<td>5</td>
<td>14.2039</td>
<td>1.2185%</td>
</tr>
<tr>
<td>2.1E</td>
<td>2.2209</td>
<td>1.1906</td>
<td>1.0000</td>
<td>6</td>
<td>16.5898</td>
<td>1.9608%</td>
</tr>
<tr>
<td>2.1E</td>
<td>2.2187</td>
<td>1.1964</td>
<td>1.0000</td>
<td>50</td>
<td>136.9104</td>
<td>1.1047%</td>
</tr>
<tr>
<td>Case</td>
<td>W0</td>
<td>W1</td>
<td>C</td>
<td>j</td>
<td>T(j)</td>
<td>(T-P)/T%</td>
</tr>
<tr>
<td>------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-------</td>
<td>---------</td>
</tr>
<tr>
<td>1</td>
<td>1.7722</td>
<td>1.0000</td>
<td>1.2364</td>
<td>1</td>
<td>3.0629</td>
<td>1.7733%</td>
</tr>
<tr>
<td>1</td>
<td>1.7682</td>
<td>1.0000</td>
<td>1.2381</td>
<td>2</td>
<td>5.2862</td>
<td>0.7909%</td>
</tr>
<tr>
<td>1</td>
<td>1.7545</td>
<td>1.0000</td>
<td>1.2324</td>
<td>3</td>
<td>8.3190</td>
<td>1.3548%</td>
</tr>
<tr>
<td>1</td>
<td>1.7270</td>
<td>1.0000</td>
<td>1.2419</td>
<td>4</td>
<td>10.5057</td>
<td>0.8034%</td>
</tr>
<tr>
<td>1</td>
<td>1.7557</td>
<td>1.0000</td>
<td>1.2318</td>
<td>5</td>
<td>13.5999</td>
<td>0.9845%</td>
</tr>
<tr>
<td>1</td>
<td>1.7266</td>
<td>1.0000</td>
<td>1.2428</td>
<td>6</td>
<td>15.8069</td>
<td>1.0759%</td>
</tr>
<tr>
<td>1</td>
<td>1.7500</td>
<td>1.0000</td>
<td>1.2319</td>
<td>50</td>
<td>132.1379</td>
<td>1.3558%</td>
</tr>
<tr>
<td>1</td>
<td>1.6624</td>
<td>1.0000</td>
<td>1.5518</td>
<td>1</td>
<td>3.2769</td>
<td>1.9168%</td>
</tr>
<tr>
<td>1</td>
<td>1.6811</td>
<td>1.0000</td>
<td>1.5493</td>
<td>2</td>
<td>5.8179</td>
<td>0.6548%</td>
</tr>
<tr>
<td>1</td>
<td>1.6782</td>
<td>1.0000</td>
<td>1.5429</td>
<td>3</td>
<td>9.0914</td>
<td>1.1702%</td>
</tr>
<tr>
<td>1</td>
<td>1.6833</td>
<td>1.0000</td>
<td>1.5469</td>
<td>4</td>
<td>11.6262</td>
<td>0.6199%</td>
</tr>
<tr>
<td>1</td>
<td>1.6837</td>
<td>1.0000</td>
<td>1.5507</td>
<td>5</td>
<td>14.9418</td>
<td>0.9182%</td>
</tr>
<tr>
<td>1</td>
<td>1.6773</td>
<td>1.0000</td>
<td>1.5454</td>
<td>6</td>
<td>17.4565</td>
<td>0.8729%</td>
</tr>
<tr>
<td>1</td>
<td>1.6443</td>
<td>1.0000</td>
<td>1.5522</td>
<td>50</td>
<td>145.9352</td>
<td>1.5187%</td>
</tr>
<tr>
<td>2.2</td>
<td>10.0729</td>
<td>3.1316</td>
<td>1.0000</td>
<td>1</td>
<td>11.1782</td>
<td>0.9425%</td>
</tr>
<tr>
<td>2.2</td>
<td>10.0669</td>
<td>3.1329</td>
<td>1.0000</td>
<td>2</td>
<td>21.4482</td>
<td>1.4662%</td>
</tr>
<tr>
<td>2.2</td>
<td>9.5217</td>
<td>3.0092</td>
<td>1.0000</td>
<td>3</td>
<td>30.2464</td>
<td>2.5225%</td>
</tr>
<tr>
<td>2.2</td>
<td>9.5600</td>
<td>2.8681</td>
<td>1.0000</td>
<td>4</td>
<td>39.7601</td>
<td>1.3083%</td>
</tr>
<tr>
<td>2.2</td>
<td>9.9814</td>
<td>3.1158</td>
<td>1.0000</td>
<td>5</td>
<td>51.7555</td>
<td>1.6394%</td>
</tr>
<tr>
<td>2.2</td>
<td>10.0797</td>
<td>3.1447</td>
<td>1.0000</td>
<td>6</td>
<td>62.5278</td>
<td>1.6782%</td>
</tr>
<tr>
<td>2.2</td>
<td>10.0802</td>
<td>3.1548</td>
<td>1.0000</td>
<td>50</td>
<td>514.4547</td>
<td>1.8359%</td>
</tr>
<tr>
<td>2.2</td>
<td>3.4914</td>
<td>1.1965</td>
<td>1.0000</td>
<td>1</td>
<td>4.5581</td>
<td>1.4644%</td>
</tr>
<tr>
<td>2.2</td>
<td>3.4762</td>
<td>1.1970</td>
<td>1.0000</td>
<td>2</td>
<td>8.2328</td>
<td>3.4061%</td>
</tr>
<tr>
<td>2.2</td>
<td>3.4807</td>
<td>1.1946</td>
<td>1.0000</td>
<td>3</td>
<td>11.8955</td>
<td>3.8111%</td>
</tr>
<tr>
<td>2.2</td>
<td>3.4620</td>
<td>1.1930</td>
<td>1.0000</td>
<td>4</td>
<td>15.5355</td>
<td>4.4723%</td>
</tr>
<tr>
<td>2.2</td>
<td>3.4619</td>
<td>1.1899</td>
<td>1.0000</td>
<td>5</td>
<td>19.1755</td>
<td>4.5150%</td>
</tr>
<tr>
<td>2.2</td>
<td>3.4649</td>
<td>1.1931</td>
<td>1.0000</td>
<td>6</td>
<td>22.8689</td>
<td>4.7214%</td>
</tr>
<tr>
<td>2.2</td>
<td>3.4663</td>
<td>1.1962</td>
<td>1.0000</td>
<td>50</td>
<td>184.3551</td>
<td>5.4447%</td>
</tr>
<tr>
<td>2.10</td>
<td>2.6600</td>
<td>1.0000</td>
<td>1.2519</td>
<td>1</td>
<td>3.9915</td>
<td>1.8445%</td>
</tr>
<tr>
<td>2.1E</td>
<td>2.6821</td>
<td>1.0000</td>
<td>1.2366</td>
<td>2</td>
<td>6.9025</td>
<td>4.3700%</td>
</tr>
<tr>
<td>2.10</td>
<td>2.6886</td>
<td>1.0000</td>
<td>1.2390</td>
<td>3</td>
<td>10.2418</td>
<td>1.4408%</td>
</tr>
<tr>
<td>2.1E</td>
<td>2.6754</td>
<td>1.0000</td>
<td>1.2346</td>
<td>4</td>
<td>13.1466</td>
<td>3.1686%</td>
</tr>
<tr>
<td>2.10</td>
<td>2.6783</td>
<td>1.0000</td>
<td>1.2363</td>
<td>5</td>
<td>16.4525</td>
<td>1.4359%</td>
</tr>
<tr>
<td>2.1E</td>
<td>2.6742</td>
<td>1.0000</td>
<td>1.2339</td>
<td>6</td>
<td>19.3839</td>
<td>2.6703%</td>
</tr>
<tr>
<td>2.1E</td>
<td>2.6756</td>
<td>1.0000</td>
<td>1.2336</td>
<td>50</td>
<td>156.5305</td>
<td>1.6087%</td>
</tr>
<tr>
<td>1</td>
<td>2.4838</td>
<td>1.0000</td>
<td>1.5582</td>
<td>1</td>
<td>4.1366</td>
<td>2.2 857%</td>
</tr>
<tr>
<td>1</td>
<td>2.5631</td>
<td>1.0000</td>
<td>1.5957</td>
<td>2</td>
<td>7.0544</td>
<td>4.2516%</td>
</tr>
<tr>
<td>1</td>
<td>2.5037</td>
<td>1.0000</td>
<td>1.5451</td>
<td>3</td>
<td>10.8208</td>
<td>1.6460%</td>
</tr>
<tr>
<td>1</td>
<td>2.5663</td>
<td>1.0000</td>
<td>1.5961</td>
<td>4</td>
<td>13.9217</td>
<td>2.9058%</td>
</tr>
<tr>
<td>1</td>
<td>2.4687</td>
<td>1.0000</td>
<td>1.5540</td>
<td>5</td>
<td>17.4365</td>
<td>1.4948%</td>
</tr>
<tr>
<td>1</td>
<td>2.5013</td>
<td>1.0000</td>
<td>1.5500</td>
<td>6</td>
<td>20.2753</td>
<td>2.3239%</td>
</tr>
<tr>
<td>1</td>
<td>2.4954</td>
<td>1.0000</td>
<td>1.5447</td>
<td>50</td>
<td>166.9013</td>
<td>1.3657%</td>
</tr>
</tbody>
</table>
APPENDIX C

Thunderstorm Program

#include<stdio.h>
#include <math.h>
#define TRUE 1
#define FALSE 0
#define MAXSTORMS 20
#define MAXBUF 1024
int numstorms, numcycles, PERIOD, HEAT, work, center=150;
int ENHANCE = FALSE;
int ChgPERIOD = FALSE;
int DEBUG = FALSE;
int DEBUG2 = FALSE;
int NEVERBALANCED = FALSE;
/* grid in [1-100][1-100]; [0] and [101] boundaries */
int grid[102][102];
int grid2[102][102];

int stormx[MAXSTORMS], stormy[MAXSTORMS];
int stormdx[MAXSTORMS], stormdy[MAXSTORMS];
int mystorms;
int oldmidb = 50, midb = 50, xmin, xmax;
int buf[MAXBUF], buf2[MAXBUF];
int bufcount;
int iproc,mypid,myhost,lnproc,nproc;
int msgtype=1,other;
double MaxRandom = 1024.0 * 32.0;
double sec[10],time;
double HhenO;
double lb = 0.0, c, correction, le;

main(argc,argv)
int argc;
char *argv[];
{
    int nextload, i, x, y;

int balance;

GetParameters(argc,argv);

whoami(iproc,&mypid,&myhost,&lnproc);
nproc=(i<<lnproc);

/*
if (iproc==0)
printf("got past init\n"); */

/* synchronize processors */
write(buf,0,other,msgtype,0);
read(buf2,0,&other,&msgtype,NULL);
/* zzzmain */

sec[0] = When();
nextload = PERIOD;
for (i=0; i<numcycles; i++) {
  msgtype = i;
  /* update thunderstorms */
  if (iproc==0)
    printf("calling update on cycle %d\n",i);
  /*
  /* update() */
  for (y=1; y<=100; y++)
    for (x=xmin; x<=xmax; x++)
      delay(work*grid[x][y]);
  /* printstorm(); */

  /* when nextload reaches zero, invoke the loadbalancer */
  --nextload;
  if (!nextload) {
    sec[9] = When();
    if (ENHANCE || ChgPERIOD) {
      balance = enhance(&nextload);
    }
    else {
      balance = FALSE;
      nextload = PERIOD;
    }
    if (!ENHANCE || balance) {
      sec[7] = When();
      loadbalance();
      sec[8] = When();
    }
  }
sec[1] = When();
time = sec[1] - sec[0];

/*
printf("\n\n******iproc = %d, time = %8.6f\n",iproc,time);
if (iproc==0)
printf("%d %8.6f\n", work, time);
if (iproc==0) printf("%d %8.6f\n", center, time);
*/
if (iproc==0)
printf("storms=y,d cycles=%d per=%d HEAT=%d work=%d
cen=%d time= %8.6f %d %d
", niimstorms,numcycles,PERIOD, HEAT, work, center, time, ENHANCE, ChgPERIOD);
}

initO
{int dummy, xxchg,i, x, y, direction;
 int srandO;
double mycorrection, yourcorrection, mycommtime,
yourcommtime, yourle;

/* zzzinit */
/* initial loadbalancing time to zero */
sec[7] = sec[8] = 0.0;

/* determine the other processor's number, xmin, and xmax */
if (iproc == 0) {
    other = 1;
xmin = 1;
xmax = midb;
}
else {
    other = 0;
xmin = midb + 1;
xmax = 100;
}

/* initialize storm grid to 1's */
for (y=0; y<102; y++)
    for (x=0; x<102; x++)
        grid[x][y] = 1;

/* all processors see each tstorm head */
for (i=0; i<numstorms; i++)
    stormx[i] = rand() % 100 + 1;
stormy[i] = rand() % 100 + 1;
direction = ( (double)rand() / MaxRandom >= 0.5) ? 1 : -1;
stormdx[i] = direction * (int)((double)rand() / MaxRandom*HEAT);
stormdy[i] = direction * (int)((double)rand() / MaxRandom*HEAT);
grid[stormx[i]][stormy[i]] += center;
}

if (DEBUG && (iproc==0)) printstorm();
srand(0);
/* Alternate the random values between the 2 PEs */
if (iproc==0) rand();

/* get measure of 'c' and 'correction' */
if (iproc==0) xxchg = xmax; else xxchg = xmin;
for (y=0; y<102; y++)
    buf[y] = grid[xxchg][y];
for (y=0; y<10; y++) {
    nwrite(&buf,102*sizeof(int),other,msgtype,0);
    nread(&buf2,102*sizeof(int),&other,&msgtype,NULL);
}
mycommtime = (sec[5] - sec[4]) / 10.0;

nwrite(&buf,102*sizeof(int),other,msgtype,0);
nread(&buf2,102*sizeof(int),&other,&msgtype,NULL);
mycorrection = sec[4] - sec[3];
nwrite(&buf,102*sizeof(int),other,msgtype,0);
nread(&buf2,102*sizeof(int),&other,&msgtype,NULL);
mycorrection += sec[4] - sec[3];

/* keep communication correction time - the time used to build up and tear down buffers should be counted as work, not transit time */
nwrite(&mycorrection,sizeof(double),other,msgtype,0);
nread(&yourcorrection,sizeof(double),&other,&msgtype,NULL);
correction = (mycorrection + yourcorrection) * 0.5;

/* estimate communication transit delay 'c' for no load balancing */
mycommtime -= correction;
nwrite(&mycommtime,sizeof(double),other,msgtype,0);
nread(&yourcommtime,sizeof(double),&other,&msgtype,NULL);
c = (mycommtime + yourcommtime) * 0.5;

/* estimate time to make estimations 'le' */
/* set to false so enh. doesn't xch lb msg */
NEVERBALANCED = FALSE;
sec[9] = When();
enhance(&dummy);
enhance(&dummy);
enhance(&dummy);
enhance(&dummy);
enhance(&dummy);

le = (When() - sec[9])/5.0;
fwrite(&le,sizeof(double),other,msgtype,0);
freed(&yourle,sizeof(double),&other,&msgtype,NULL);
le = (le + yourle) * 0.5;
NEVERBALANCED = TRUE;

/********************************************/
update()
{
    int x, y, direction, xxchg;

/* zzzupdate */
if (DEBUG) printStorm();

    /* exchange middle border values with other pe */
    if (iproc==0) xxchg = xmax; else xxchg = xmin;
    for (y=0; y<102; y++)
        buf[y] = grid[xxchg][y];
    fwrite(&buf,102*sizeof(int),other,msgtype,0);
    fread(&buf2,102*sizeof(int),&other,&msgtype,NULL);
    if (iproc==1) xxchg = xmin-1; else xxchg = xmax+1;
    for (y=0; y<102; y++)
        grid[xxchg][y] = buf2[y];

/* update storm grid by averaging nearest neighbors */
for (y=1; y<101; y++) {
    for (x=xmin; x<=xmax; x++) {
        grid2[x][y] =
            grid[x-1][y+1] + grid[x][y+1] + grid[x+1][y+1] +
            grid[x-1][y] + grid[x][y] + grid[x+1][y] +
            grid[x-1][y-1] + grid[x][y-1] + grid[x+1][y-1];
        grid2[x][y] = (int)(grid2[x][y]/9.+.5);
        if (grid2[x][y] > 1) {
            grid2[x][y] = (double)rand()/MaxRandom >= 0.40) ?
                grid2[x][y]-1 : grid2[x][y] ;
            rand();
        }
    }
}
for (y=1; y<101; y++)
    for (x=xmin; x<=xmax; x++)
        grid[x][y] = grid2[x][y];

/* update storm centers */
for (x=0; x<numstorms; x++) {

if (DEBUG) printf(" Storm (%d,%d) ",stormx[x],stormy[x]);
stormx[x] += stormdx[x];
if (stormx[x] < 1) {
    stormx[x] = -stormx[x];
    stormdx[x] = -stormdx[x];
} else if (stormx[x] > 100) {
    stormx[x] = 200 - stormx[x];
    stormdx[x] = -stormdx[x];
} stormy[x] += stormdy[x];
if (stormy[x] < 1) {
    stormy[x] = -stormy[x];
    stormdy[x] = -stormdy[x];
} else if (stormy[x] > 100) {
    stormy[x] = 200 - stormy[x];
    stormdy[x] = -stormdy[x];
} grid[stormx[x]][stormy[x]] += center;
if (DEBUG) printf(" moved to (%d,%d)\n",stormx[x],stormy[x]);
}

/******************************************************************/
loadbalance()
{
    int x, y, bufcount, i, xxchg;
    int temp;
    /* zzzload */
    /* buf[0] has worktotal; successive have columns going
       AWAY from midb */
    buf[0] = 0;
    bufcount = 1;
    if (iproc==0) {
        buf[bufcount] = 0;
        for (y=1; y<=100; y++)
            buf[bufcount] += grid[xmax][y];
        for (x=xmax-1; x>=xmin; x--) {
            buf[bufcount] = 0;
            for (y=1; y<=100; y++)
                buf[bufcount] += grid[x][y];
            buf[bufcount] += buf[bufcount-1];
            bufcount++;
        }
        buf[0] = buf[--bufcount];
    if (DEBUG) printf("worksum=\%d work(i)=\%d\n",buf[0],buf[1]);
else {
    buf[bufcount] = 0;
    for (y=1; y<=100; y++)
        buf[bufcount] += grid[xmin][y];
    for (x=xmin+1; x<=xmax; x++) {
        buf[bufcount] = 0;
        for (y=1; y<=100; y++)
            buf[bufcount] += grid[x][y];
        buf[bufcount] += buf[bufcount-i];
        bufcount++;
    }
    buf[0] = buf[--bufcount];
}
/* exchange worksum vector and worktotal */
nwrite(buf,(bufcount+1)*sizeof(int),other,msgtype,0);
nread(buf2,MAXBUF*sizeof(int),&other,&msgtype,NULL);

/* case 1) mytotal > yourtotal */
if (buf[0] > buf2[0]) {
    /* while ( mytotal-myclosest > yourtotal+myclosest) */
    bufcount = 1;
    while ((buf[0]-buf[bufcount]) > (buf2[0]+buf[bufcount])) {
        /* add one to the number of columns I xfer to you */
        bufcount++;
    }
    temp = (bufcount==1) ? 0 : buf[bufcount-1];
    if (DEBUG)
        printf("pe%d needs to transfer out %d col myo %d myu %d ...
", iproc,bufcount-1,buf[0],buf2[0],buf[0]-temp,
        buf2[0]+temp);
    /* xfer (bufcount - 1) columns */
    for (i=0; i<bufcount-1; i++) {
        if (iproc==0) xxchg = xmax; else xxchg = xmin;
        for (y=0; y<102; y++)
            buf[y] = grid[xxchg][y];
        nwrite(&buf,102*sizeof(int),other,msgtype,0);
        if (iproc==0) xmax--; else xmin++;
        if (DEBUG)
            printf("pe%d writing out... xmin=%d xmax=%d
", iproc,xmin,xmax);
    }
}
/* case 2) mytotal < yourtotal */
else if (buf[0] < buf2[0]) {
    /* while ( mytotal+yourclosest < yourtotal - yourclosest) */
    bufcount = 1;
    while (((buf[0]+buf2[bufcount]) < (buf2[0]-buf2[bufcount]))) {
        /* add one to the number of columns I receive FROM you */
        bufcount++;
    }
    /* read in all needed columns */
temp = (bufcount==1) ? 0 : buf[bufcount-1];
if (DEBUG)
printf("pe%d needs to transfer in %d col myo %d myu %d %d\n", 
iproc,bufcount-1,buf[0],buf2[0],buf[0]-temp,
buf2[0]+temp);
/* xfer in (bufcount - 1) columns */
for (i=0; i<bufcount-1; i++) {
    nread(&buf2,102*sizeof(int),&other,&msgtype,NULL);
    if (iproc==1) xxchg = xmin-1; else xxchg = xmax+1;
    for (y=0; y<102; y++)
        grid[xxchg][y] = buf2[y];
    if (iproc==1) xmin--; else xmax++;
    if (DEBUG)
printf("pe%d reading in... xmin=%d xmax=%d\n",iproc,xmin,xmax);
}
/*
--adjust xmin xmax
--exchange time for load balancing (this will synch them)
--average the time lb
*/

/***************************************************************************/
int stillfurther(x1, x2)
double x1, x2;
{
    /* PE0 : list sorted largest to smallest, return (x1 > x2)
    PE1 : list sorted smallest to largest, return (x1 < x2) */
    if ((iproc == 0) && (x1 > x2))
        return(1);
    else if ((iproc == 1) && (x1 <= x2))
        return(2);
    return(0);
}

/***************************************************************************/
int min(a,b)
int a, b;
{
    if (a >= b)
        return(b);
    else
        return(a);
}

/***************************************************************************/
int max(a,b)
int a, b;
{
return( a>=b ? a : b );
}

/***********************************************************************/
GetParameters(argc,argv)
int argc;
char *argv[];
{
    int i, nglobal(), nlocal();

    nglobal();
    if (argc < 7) {
        printf("ERROR: Usage: t numstorms numcycles period");
        printf(" HEAT work center [ENH] [CP]\n\n");
        exit(0);
    }

    numstorms = i = 0;
    while (argv[1][i] != '\0')
        numstorms = 10 * numstorms + argv[1][i++] - '0';
    if (numstorms > MAXSTORMS) {
        printf("ERROR: must use less than %d storms\n",MAXSTORMS);
        exit(0);
    }

    numcycles = i = 0;
    while (argv[2][i] != '\0')
        numcycles = 10 * numcycles + argv[2][i++] - '0';
    if (numcycles < 1) {
        printf("ERROR: must have a positive number of cycles\n");
        exit(0);
    }

    PERIOD = i = 0;
    while (argv[3][i] != '\0')
        PERIOD = 10 * PERIOD + argv[3][i++] - '0';
    if (PERIOD < 1) {
        printf("ERROR: must have a positive number for ");
        printf("the lb's period\n");
        exit(0);
    }

    HEAT = i = 0;
    while (argv[4][i] != '\0')
        HEAT = 10.0 * HEAT + argv[4][i++] - '0';
    if (HEAT > 100) {
        printf("ERROR: HEAT must be integer in range [0..100]\n");
        exit(0);
    }

    work = i = 0;
    while (argv[5][i] != '\0')

}
work = 10.0 * work + argv[5][i++] - '0';

center = i = 0;
while (argv[6][i] != '0')
    center = 10.0 * center + argv[6][i++] - '0';

ENHANCE = ((argc >7) && (argv[7][0]=='1')) ? TRUE : FALSE;
ChgPERIOD = ((argc >8) && (argv[8][0]=='1')) ? TRUE : FALSE;

/*
 printf("storms=%d cycles=%d period=%d HEAT=%d work=%d\n", 
    numstorms, numcycles, PERIOD, HEAT, work);
*/

nlocal();

/********************************************************************************
delay(count)
int count;
{
    int i, j;

    /* delay loop */
    for (i=0;i<count;i++) j = (i+1) - (i+1)/2 - (i+1)/2;
}

enhance(nextload)
int *nextload;
{
    double s0, s1, mytime, yourtime;
    double d, t, twlb, mylb, yourlb;
    double yourt, yourtwlb;
    int msgtype = 11;

    /* Determine usefulness of invoking load balancer...
  1) estimate T(nextload) without balancing (T) 
     by getting s0, s1, and c
     -- get the number of particles on each PE 
     -- get time per particle to do work in balanced case 
     -- get comm time per particle in balanced case 
     -- get max number particles now 
     -- use above to estimate s0, s1, and c via 
        linear interpolation 
     -- calculate T using these values 
     -- print above values and measure and print actual values 
        for comparison 
  2) estimate T(nextload) with balancing (TWLB)
     -- for fixed number of particles, just use measured value 
        from first cycle for s0, s1, and c and use equations
*/
-- for changing total work, use times from 1) and an
  estimation of number of particles/2 to get s0, s1,
  and c to estimate T
-- print above values and measure and print actual values
  for comparison

3) estimate cost to do balancing (LC)
  ***** this is the real wild card *****
- need to estimate number of particles expected to
go out of balance
-- for small j, this will be some percentage of number
  on previous cycle
-- at some number j, would expect a steady state to be reached
  - could simply use measurement of previous balancing (at
    least as long as j is keep constant
if TWLB + LC < T then loadbalance
*/
/* timing structure in main() is as follows...
  sec[2]  work
  sec[3]  nwrite
  sec[4]  nread
  sec[5]  work
  sec[6]
*/

/* estimate s0 and s1 for case of no load balancing */
nwrite(&mytime,sizeof(double),other,msgtype,0);
cread(&yourtime,sizeof(double),&other,&msgtype,NULL);
s0 = (mytime > yourtime) ? mytime : yourtime;
s1 = (mytime <= yourtime) ? mytime : yourtime;

/* estimate time 't' for no loadbalancing with period 'PERIOD' */
d = s0 - s1;
/****** Case 2.1: D/2 < C <= D ******/
if ((c > d*0.5) && (c <= d)) {
  if ((PERIOD/2) == ((PERIOD+1)/2)) { /* PERIOD is even */
    t = d-c+(PERIOD/2)*(2*s0+2*c-d);
  } else {
    t = s0+c+(PERIOD-1)/2)*(2*(s0+c)-d);
  }
}
/****** Case 2.2: 0 <= C <= D/2 ******/
else if (c <= d/2) {
  t = PERIOD*s0+c;
}
/****** Case 1: C > D ******/
else {
t = PERIOD*(s0+c) - (PERIOD/2) * d;

if (DEBUG2 && (iproc==0))
printf("ENH unb s0=\%f s1=\%f d=\%f c=\%f t=\%f
",s0,s1,d,c,t);

/* now estimate s0, s1, & c for the perfectly balanced case */
s0 = (s0 + s1) * 0.5;
s1 = s0;
d = 0;

/* estimate time 'twlb' with loadbalancing with period 'PERIOD' */
/* d = s0 - s1; d=0 */
twlb = PERIOD*(s0+c);

/* estimate time to do load balancing by using measured time
only once */
if (sec[8]>0.001) & NEVERBALANCED) {
    mylb = sec[8] - sec[7];
    nwrite(&mylb,sizeof(double),other,msgtype,0);
    nread(&yourlb,sizeof(double),&other,&msgtype,NULL);
    lb = (mylb + yourlb) * 0.5;
    NEVERBALANCED = FALSE;
}

if (DEBUG2 && (iproc==0))
printf("ENH b s0=\%f s1=\%f d=\%f c=\%f twlb=\%f lb=\%f le=\%f
", s0,s1,d,c,twlb,lb,le);

/* zzzenhance */
/* estimate time for enhancer from time measurements */
if (ChgPERIOD) {
    if ( (PERIOD > 1) && (t > twlb + lb + le + c) ) {
        PERIOD = PERIOD - 1;
    }
}

/* */
if (iproc==0) printf("period = \%d
",PERIOD);
if ( (PERIOD > 1) && (t > twlb + 2.0 * c + lb + le) ) {
    else if (t < twlb + lb + le)
    */
    else if (t < twlb + lb + le - c)
        PERIOD = PERIOD + 1;
}

/* need to change this to a calc. value in later version */
#nextload = PERIOD;

/* */
if (t < twlb + lb + 2. * c) {
    if (t < twlb + 0.5 * lb ) {
*/
    if (ENHANCE && (t < twlb + 0.5 * lb )) {
        if (DEBUG2) printf("ENHANCER INVOKING LB - NO\n");
return(FALSE);
}
if (DEBUG2) printf("ENHANCER INVOKING LB - YES\n");
return(TRUE);

int x,y;
if (iproc==0) {
    printf("vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv\n");
    for (y=0; y<60; y++) {
        for (x=0; x<xmax; x++) {
            if (grid[x][y]>=20) putchar('*');
            else if (grid[x][y]>10) putchar('+');
            else if (grid[x][y]>9) putchar('9');
            else if (grid[x][y]>8) putchar('8');
            else if (grid[x][y]>7) putchar('7');
            else if (grid[x][y]>6) putchar('6');
            else if (grid[x][y]>5) putchar('5');
            else if (grid[x][y]>4) putchar('4');
            else if (grid[x][y]>3) putchar('3');
            else if (grid[x][y]>2) putchar('2');
            else if (grid[x][y]==2) putchar('2');
            else putchar(' ');
        }
        putchar('
');
    }
    printf("\n
");
    printf("vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv\n");
}
printf("---------------------------------------------\n\n");
The following is the code used to test the generalized Work/Exchange equations for accuracy. The code calculates actual time that elapses in a program and compares this time with the predicted value calculated from the formulas. The program ranges over a wide variety of cases. For more details, see Chapter 6.

```c
#include <stdio.h>
#include <sys/time.h>
define MAXPE 100
#define MAXCYCLE 200
#define MAXWORK 100
#define TRUE 1
#define DEBUG 0

float idle[MAXPE], etime[MAXPE], idlep[MAXPE], etimep[MAXPE];
float temptime[MAXPE];
float error=0.0;
int comm, work[MAXPE];

int n; /* number of current processors (0..n-1) */
/* DS to collect stats on ranges of tested testcases */
int stats[5][MAXPE+1], testcase;
int numtests = 0;

unsigned short seed16v[3];
struct timeval tp;
struct timezone tzp;

main()
{
    int j; /* current cycle */
    int count1, count2, peindex;

    /* Initialize variables */
    work[0] = MAXWORK;
    gettimeofday(&tp,&tzp);
    seed16v[0] = getpid();
    seed16v[1] = getpid();
    seed16v[2] = tp.tv_sec;
```
seed48(seed16v);

for (count1=0; count1<5; count1++)
    for (count2=0; count2<MAXPE; count2++)
        stats[count1][count2] = 0;

    /* Loop over workload value range of PE 1 */
    for (work[1]=0; work[1]<=work[0]; work[1]++) {
        /* determine the number of PEs and set workload values */
        getwork();

        /* Loop over the communication delay value range */
        for (comm=0; comm<2*work[0]; comm++) {
            /* Clear elapsed time vector for next set of runs */
            for (peindex=0; peindex<n; peindex++)
                etime[peindex] = 0.0;

            if (DEBUG==TRUE) {
                printf("n--------------------------\n");
                printf("wo=%d w1=%d c=%d pe=%d\n", work[0], work[1], comm, n);
                printf("\n et0 ep0 et1 ep1 i0 ip0 i1 ip1 ");
                printf("e0-ep0 e1-ep1 i0-ip0 i1-ip1\n");
            }

            /* Loop over the range of cycles */
            for (j=1; j<=MAXCYCLE; j++) {
                /* Determine next set of time and idle time values */
                getvalues();
                /* Get predicted values using the equations */
                predict(j);
                /* Save accumulated error */
                geterror();

                stats[testcase][(n-2)/10] += 1;
                numtests += 1;
            }
        }
    }

    /* Print out results */
    printf("An accumulated sum of the absolute value");
    printf(" of the error is...\n");
    printf(" Sum |error| = %9.6f\n", error);
    printf("from a total of %d tested cases.\n", numtests);

    printf(" Processors\n");
    printf("Cases 2-9 10-19 20-29 30-39 40-49 ");
printf("50-59 60-69 70-79 80-89 90-99\n");
printf("==================================
");
for (count1=0; count1<5; count1++) {
    switch(count1) {
    case 0 : printf("1 odd "); break;
    case 1 : printf("1 even"); break;
    case 2 : printf("2.1 odd "); break;
    case 3 : printf("2.1 even"); break;
    case 4 : printf("2.2 "); break;
    }
    for (count2=0; count2<MAXPE/10; count2++) {
        printf("%6d", stats[count1][count2]);
    }
    putchar('\n');
}
} /* end main() */

gterror()
{
    int pe;
    /* Check for elapsed time error */
    error += etime[0] > etimep[0] ? etime[0]-etimep[0] :
        etimep[0]-etime[0];
    /* Predicted value for all pe=1 are stored only in etimep[1] */
    for (pe=1; pe<n; pe++)
            etimep[1]-etime[pe];
    /* Check idle times for error */
    for (pe=0; pe<n; pe++)
        error += idle[pe] > idlep[pe] ? idle[pe]-idlep[pe] :
            idlep[pe]-idle[pe];
}

gvalues()
{
    int pe; /* processor number (0,n-1) */
    int maxpe;
    float maxtime;
    for (pe=0; pe<n; pe++) {
        /* get max. elapsed time during previous cycle 
            for all PE's except for this one */
        maxoldtime(pe,&maxtime); /* (my pe, other's time) */
        /* calculate the new elapsed time and save 
            in temporary location */
/* new time equals larger of two possibilities:
  1) largest time for another pe to get message here
  2) my own time plus my own work this cycle */
temptime[pe] = max( maxtime + comm,
  etime[pe] + work[pe]);

/* calculate new idle time for this pe and this cycle */
idle[pe] = max( temptime[pe] - etime[pe] - work[pe], 0.0);
}

/* write new elapse time values in time array */
for (pe=0; pe<n; pe++)
etimeCpe[pe] = temptime[pe];

maxoldt ime (pe, maxtime)
int pe;
float *maxtime;
{
  int peindex;
  *maxtime = 0.0;

  /* Check for max times for smaller valued PE's */
  for (peindex=0; peindex<pe; peindex++)
    if ( (etimeCpeindex[pe] + work[peindex]) > *maxtime)
      *maxtime = etimeCpeindex[pe] + work[peindex];

  /* Check for max times for larger valued PE's */
  for (peindex=pe+1; peindex<n; peindex++)
    if ( (etimeCpeindex[pe] + work[peindex]) > *maxtime)
      *maxtime = etimeCpeindex[pe] + work[peindex];
}

getwork0
{
  int pe;
  if (work[1]==0)
    n = 2;
  else {
    /* get the number of processors, 2 <= n */
    getrandom(2,MAXPE,&n);

    for (pe=2; pe<n; pe++)
      getrandom(1,work[pe-1],&work[pe]);
  }
  if (DEBUG==TRUE) {
    printf("\n\n%d pes:",n);
    for (pe=0; pe<n; pe++)
      printf(" %d",work[pe]);
  }
}
printf(’\n’);
}

getrandom(lowerbound, upperbound, value)
int lowerbound, upperbound, *value;
{
    *value = (int)(lrand48()/(double)0x7fffffff
     * (upperbound-lowerbound+1) + lowerbound);
}

max(a, b)
float a, b;
{
    return( a >= b ? a : b);
}

predict(j)
int j;
{
    float d01, w01, h01;
    int peindex;

    d01 = work[0] - work[1];
    h01 = 0.5 * d01;
    w01 = 0.5 * (work[0] + work[1]);

    /******** Case 1: D01 < C ********/
    if (d01 < comm) {
        /* j is even */
        if ((j/2) == ((j+1)/2)) {
            testcase = 1;
            etimep[0] = j*(w01+comm);
            etimep[1] = j*(w01+comm);
            idlep[0] = comm;
            for (peindex=1; peindex<n; peindex++)
                idlep[peindex] = comm + work[1] - work[peindex];
        } /* j is even */
        else {
            testcase = 0;
            etimep[0] = j*(w01+comm) - h01;
            etimep[1] = j*(w01+comm) + h01;
        } /* j is even */
    } /* j is even */
} /* j is even */
idlep[0] = comm - d01;
for (peindex=1; peindex<n; peindex++)
  idlep[peindex] = comm + work[0] - work[peindex];
if (DEBUG==TRUE)
  printf("cl0 %4.0f %4.0f %4.0f %4.0f %4.0f \n", etime[0], etimep[0], etime[1], etimep[1],
    idle[0], idlep[0], idle[1], idlep[1],
    etime[0]-etimep[0], etime[1]-etimep[1],
    idle[0]-idlep[0], idle[1]-idlep[1]);
}

}*/ Case 2.1: D/2 < C <= D */
else if (comm > d01/2) {
  /* j is even */
  if ((j/2) == ((j+1)/2)) {
    testcase = 3;
    etimep[0] = j*(w01+comm);
    etimep[1] = j*(w01+comm) + 2.0 * h01 - comm;
    idlep[0] = 2 * comm - d01;
    for (peindex=1; peindex<n; peindex++)
      idlep[peindex] = work[0] - work[peindex];
    if (DEBUG==TRUE)
      printf("c2.1e%4.0f %4.0f %4.0f %4.0f %4.0f \n", etime[0], etimep[0], etime[1], etimep[1],
        idle[0], idlep[0], idle[1], idlep[1],
        etime[0]-etimep[0], etime[1]-etimep[1],
        idle[0]-idlep[0], idle[1]-idlep[1]);
  } /* j is even */
  else {
    testcase = 2;
    etimep[0] = j*(w01+comm) + h01 - comm;
    etimep[1] = j*(w01+comm) + h01;
    idlep[0] = 0;
    for (peindex=1; peindex<n; peindex++)
    if (DEBUG==TRUE)
      printf("c2.1o%4.0f %4.0f %4.0f %4.0f %4.0f \n", etime[0], etimep[0], etime[1], etimep[1],
        idle[0], idlep[0], idle[1], idlep[1],
        etime[0]-etimep[0], etime[1]-etimep[1],
        idle[0]-idlep[0], idle[1]-idlep[1]);
  }
  /* j is even */
  else {
    testcase = 4;
    etimep[0] = j*work[0];

}*/ Case 2.2: 0 <= C <= D/2 */
else {
  testcase = 4;
  etimep[0] = j*work[0];
etimep[1] = j*work[0] + comm;
idlep[0] = 0.0;
for (peindex=1; peindex<n; peindex++)
idlep[peindex] = j==1 ? comm + work[0]-work[peindex] :
work[0]-work[peindex];
if (DEBUG==TRUE)
printf("c2. %4.0f %4.0f %4.0f %4.0f %4.0f ");
printf("%4.0f %4.0f %4.0f %4.0f %4.0f %4.0f \n",
extime[0], etimep[0], etime[1], etimep[1],
idle[0], idlep[0], idle[1], idlep[1],
extime[0]-etimep[0], etime[1]-etimep[1],
idle[0]-idlep[0], idle[1]-idlep[1]);
}

Output

The program was executed for 4,040,000 different cycles. Since each cycle was
executed with an average of 50 processors and 2 equations were verified for each
processor-cycle, a total of 404,000,000 equation verifications took place. The total
accumulated error from all these equation comparisons is 0.000000. The output also
includes a table that indicates the tests cases were spread out over all the possible
cases. The actual program output follows:

An accumulated sum of the absolute value of the error is...

\[ |\text{error}| = 0.000000 \]
from a total of 4040000 tested cases.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Processors</td>
<td></td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>1 odd 94100102900174500125000 79700169800150200232500158500217700</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 even 94100102900174500125000 79700169800150200232500158500217700</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.1 odd 22800 18400 22400 27200 10100 24800 14700 43400 30400 40800</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.1 even 22800 18400 22400 27200 10100 24800 14700 43400 30400 40800</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.2 46200 37400 46200 55600 20400 50800 30200 88200 62200 83000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>