Dynamic Load Distributions for Adaptive Computations on MIMD Machines using hybrid Genetic Algorithms

by

TIMOTHY HAROLD KAISER

Doctor of Philosophy, Computer Science, The University of New Mexico, 1997
Masters of Science, Electrical Engineering (Applied Physics), The University of California, San Diego, 1987
Bachelor of Science, Physics, The University of Missouri, Rolla, 1981

Dissertation
submitted in partial fulfillment of the degree of

Doctor of Philosophy
Computer Science

December, 1996

© 1996, Timothy Harold Kaiser

To:
Dr. Alice M. Vargas,
my best friend

ACKNOWLEDGMENTS

Special thanks go to the people at the Albuquerque Resource Center
Bob Robey
Amy Shreve
Marlene Wallace
Tom Sahs

Thanks to my parents for giving me a good start,

Dr. Donald Prahlow for showing me it's never to late,

Mike Gittings, for his work on SAGE
Mike Mckay, for support while working on NEWPLOT

Thanks to my committee
Dr. Brian Smith for Fortran 90 and trying to keep me out of trouble
Dr. Frank Gilfeather for letting me play with his nice toys and the suggestion for the ASCI calculation
Dr. Bernard Moret for answering all of those questions
Dr. Arthur (Barney) Maccabe for staying calm and writing a really good book
Dr. Henry Shapiro for not giving up on me

Musical support:
Heart, Wilson Phillips, J. S. Bach, Styx

Co-pilot support:
Dream dreams. Set your sights on the farthest star and let
your dreams become reality. Only you can chart your
course with God as your co-pilot.

This work was supported in part by the High Performance Computing Education and Research Center at the University of New Mexico, The Albuquerque Resource Center, and The Maui High Performance Computing Center under U. S. Air Force contract No. F29601-93-2-0001.

Dynamic Load Distributions for Adaptive Computations on MIMD Machines using hybrid Genetic Algorithms

by

TIMOTHY HAROLD KAISER

Doctor of Philosophy, Computer Science, The University of New Mexico, 1997
Masters of Science, Electrical Engineering (Applied Physics), The University of California, San Diego, 1987
Bachelor of Science, Physics, The University of Missouri, Rolla, 1981

ABSTRACT

With the advent of readily available Multiple-Instruction, Multiple-Data (MIMD) machines, calculations are being performed which, until recently, were impractical. Physical scientific calculations such as weather forecasting, shock propagation, fluid migration and aerodynamics calculations are being performed at higher fidelity and over larger problem spaces. These types of calculations are often performed by mapping physical space onto a two dimensional grid. When using a MIMD computer, each processor is responsible for performing calculations for a portion of the grid.

To obtain optimal performance on a MIMD computer, the computational load for each processor must be nearly equal and the communication between processors must be minimized. Unfortunately, balancing load and simultaneously reducing communication is difficult. Many modern scientific grid calculations exacerbate the problem because grid points are dynamically created and deleted during the calculation. If a calculation is started with an optimal distribution of points, it will not remain optimal.

This dissertation addresses the issue of providing automatic, dynamic load balancing and communication reduction to achieve near optimum performance for dynamic grid calculations. It has resulted in a new hybrid genetic algorithm, run in parallel with a dynamic grid calculation to achieve near optimum dynamic load balance and communication cost.

The hybrid genetic algorithm was incorporated into a parallel adaptive grid program framework, developed for this effort. As a particular test, routines to solve Euler's equation of gas dynamics were incorporated into the framework. The resultant parallel adaptive grid application with dynamic rebalancing capabilities was used to solve these problems in two dimensions.

The hybrid genetic algorithm was shown to dramatically improve the run time of the simulations as compared to static or random dynamic reallocation of cells to processors. The algorithm was shown to improve performance when compared to recursive bisection techniques and standard genetic algorithms on the example calculations.

The hybrid genetic algorithm load balancing technique can be incorporated into a large class of grid calculations. The framework developed for this effort is robust, flexible and extensible, and can also be used to create a wide variety of grid simulations. An outline for future enhancements to the framework is provided.

Table of Contents

Chapter 1. Introduction, scope, and overview.......1

1.1. Introduction.......1

1.2. Scope of the research effort.......2

1.3. Requirements for the genetic algorithm to be effective.......5

1.4. Overview of the remaining sections of this thesis.......7

Chapter 2. Additional introducation and background information.......10

2.1. What is a grid program?.......10

2.1.1. Irregular grids.......11

2.1.2. Adaptive gridding.......12

2.2. What is parallel computing?.......12

2.3. Parallel architectures.......15

2.3.1. Flynn's taxonomy.......15

2.3.2. Memory architectures.......15

2.4. Parallel programming and grid programs.......16

2.4.1. Parallel programming for regular grids.......16

2.4.2. Parallel programming for irregular and adaptive grids.......17

2.5. Thesis statement revisited.......18

2.6. Basic genetic algorithms description.......19

2.7. Review of two other methods.......20

2.7.1. Bisection.......20

2.7.2. Wheat's tiling.......23

Chapter 3. Discussion of hardware and software tools.......26

3.1. The target architecture. 26
3.1.1. Description of the MHPCC SP2 hardware. 27
3.1.2. Description of the ARC SP1. 29
3.2. The software tools and numerical routines. 30
3.2.1. Brief discussion of the PLIFE framework. 31
3.2.1. Brief discussion of the parallel GA framework. 32
3.2.1. Euler's equations of gas dynamics. 34

Chapter 4. Development of heuristics. 37
4.1. Introduction. 37
4.2. Simulation description. 37
4.3. Description of information presented. 43
4.4. Results with no load balancing and random load balancing. 44
4.5. Introduction of the fitness function and the request vector. 46
4.6. Expansion on sine qua non one, use domain specific knowledge. 47
4.6.1. Larger contiguous blocks implies less communication. 48
4.6.2. Clusters of cells are generated at irregular rates. 51
4.6.3. Future load can be estimated. 52
4.7. Cells split in the region of the shock front. 56
4.8. Sine qua non two, improvement in the GA. 60
4.8.1. Fast mutation. 60
4.8.2. Discussion of the Mansour-Fox algorithm for static grids. 61
4.9. Expansion on sine qua non 2, improve the GA algorithm, and enhancements to the Mansour-Fox algorithm. 63
4.10. Initial test of the improved Mansour-Fox algorithm. 64
4.11. Fast GA yields fast simulation time and reinforcement of sine qua non one. ..

68
4.12. Expansion on sine qua non one & two, tuning of the GA using the improved Mansour-Fox algorithm. 70
4.13. What was learned from these runs? 71
4.13.1. Summary of sine qua non 1 and 2....71
4.13.2. GA was shown to be effective. 72

Chapter 5. Application of heuristics....74
5.1. Problem description, 16 node SP1 and SP2 runs. 74
5.2. Applying what was learned from the 4 node calculation to 16 node runs. ...79
5.3. Results with no load balancing and random balancing on the SP1. 80
5.4. Initial results and tuning for a particular architecture. 84
5.5. Verification of the thesis statement....89
5.6. Expansion on sine qua non 1 & 2, tuning the GA algorithm, tuning of the GA using the improved Mansour-Fox algorithm....90
5.7. Application of the results to a different architecture, SP2 runs. 97
5.8. Significance of the SP2 simulations. 98
5.9. Application of the results to a different problem. 100
5.10. Comparison to bisection results. 101
5.11. Summary. 103

Chapter 6. Summary....104
6.1. Introduction....104
6.2. Showed the GA to be effective. 104
6.3. Discovered sine qua nons....105
6.3.1. Improvements to the GA....105

6.3.1.1. Fast mutation. 105
6.3.1.2. Mansour-Fox algorithm and its improvement. 105
6.3.2. Use of domain specific knowledge. 107
6.4. Tuning of the GA. 107
6.5. Developed tools. 108
6.6. Recommendations. 108
6.6.1. Use the two sine qua nons. 108
6.6.2. Use a framework to test and develop applications. 108
6.6.3. Use the improved Mansour-Fox algorithm. 109
6.7. Areas of future work. 109
6.7.1. Apply Mansour-Fox algorithm to other problems. 109
6.7.2. Further development of framework. 109
6.7.3. Load prediction. 110
6.7.4. Work on configurations with fast communication. 110
6.7.5. Run tests on multilevel ASCI systems. 111
6.7.6. Use other numerical routines. 111
6.7.7. Run comparisons of GA with/without binomial probability distribution. 112

Appendix A. The PLIFE Framework. 113
A.1. Overview PLIFE framework. 113
A.2. Heritage of PLIFE. 113
A.3. Requirements of a tool for thesis verification. 114
A.4. Programming model for PLIFE. 114
A.4.1. Overview of the functional units of PLIFE. 115
A.4.2. Numerics. 116
A.4.3. Load Balancing. 117

A.4.4. Dynamic regridding and communication management. 117
A.5. PLIFE background information. 117
A.6. Overview of the grid structure. 119
A.6.1. Grid data structure...121
A.6.2. Each processor holds a complete description of the grid..........................124
A.6.3. Each processor only holds values which are required. 124
A.7. Boundary conditions. 125
A.8. Calculation cycle...125
A.9. Cell splitting, joining, and finding neighbors. 126
A.10. Other uses for PLIFE. 128
A.11. Customizing PLIFE for a particular application.......................................129
A.14. Lessons learned from the construction of PLIFE. 132
A.15. Future directions For PLIFE. 134

Appendix B. Numerical solution of Euler's equations of gas dynamics..................137
B.1. Introduction to Euler's equations...137
B.2. Conservation of energy and mass. 139
B.3. Numerical method for Euler's equations. 140

Appendix C. Genetic Algorithm description..145
C.1. Overview...145
C.2. The serial genetic algorithm...145
C.3. Mutations in the GA...147
C.4. The parallel operation of the GA. 149
C.5. Use of the genetic algorithm. 151
C.5.1. Pflagmove called at various times during a run..151

C.5.2. How the initial population is created. 152

C.5.3. Fitness function. 153

References. 157

List of Figures

Figure 2.1. Division of cells using bisections methods. 22

Figure 3.1. SP2 logical frame and switch board. 29

Figure 3.2. Diagram of a SP2 with 80 nodes and 5 frames where each line represents 4 connections. 29

Figure 4.1. Initial density for an example calculation. 39

Figure 4.2. Initial pressure for an example calculation. 40

Figure 4.3. Initial distribution of cells to processors. 42

Figure 4.4. Cluster of cells moved by the genetic algorithm. 49

Figure 4.5. Illustrates the calculation rate dependency on the movement of extra load for node 0. 54

Figure 4.6. A shock wave propagating through the grid. 57

Figure 4.7. Distribution of cells to processors at the end of the simulation with communication between split and unsplit cells counted yielding a communication time of 348 seconds. 59

Figure 4.8 Distribution of cells to processors near the ending time of the simulation with communication between split and unsplit cells not counted yielding a communication time of 228 seconds. 60

Figure 4.9. Plot of overall simulation run time and GA run time showing the trend that shorter GA run times leads to shorter overall simulation run times. 69

Figure 5.1. Initial distribution of cells to processors. 76

Figure 5.2. Initial location for regions of higher pressure. 77

Figure 5.3. Blow up of one of the regions of higher pressure. 78

Figure 5.4. Distribution of cells to processors for a portion of the grid when the random function is used to balance the load, independent of communication cost. 82

Figure 5.5. Finesse of the grid at the end of the calculation....83
Figure 5.6. Assignment of cells to processors when weight for communication is 0.4 at the end of the genetic algorithm run. 85
Figure 5.7. Assignment of cells to processors when weight for communication is 0.3 at the end of the genetic algorithm run. 86
Figure 5.8. Assignment of cells to processors when weight for communication is 0.2 at the end of the genetic algorithm run. 87
Figure 5.9. Assignment of cells to processors when weight for communication is 0.1 at the end of the genetic algorithm run. 88
Figure 5.10. Plot of run times for the simulation using the hybrid genetic algorithm. Data is grouped by population size, 480 down to 240 and each bar represents a different frequency of calling the hill climbing routine, 1-5, left to right. ...94
Figure 5.11. Plot of run times for the simulation using the hybrid genetic algorithm. Data is grouped by the frequency of calling the hill climbing routine, 1-5 and each bar represents a different population size, 480 down to 240, left to right. 95
Figure 5.12. Plot of run times for the simulation using the hybrid genetic algorithm. Data is grouped by number of generations, 480 down to 240 and each bar represents a different frequency of calling the hill climbing routine, 1-5, left to right..96
Figure 5.13. Plot of run times for the simulation using the hybrid genetic algorithm. Data is grouped by the frequency of calling the hill climbing routine, 1-5 and each bar represents a different number of generations, 480 down to 240, left to right. 97
Figure A.1. Diagram of an application derived from the PLIFE framework. 116
Figure A.2. The grid structure used within PLIFE. 123
Figure B.1. Edges between cells of the PLIFE grid. 141

List of Tables

Table 4.1. Base line simulation for which no movement of cells occurred and a simulation for which the movement occurred randomly but balanced the load. 45

Table 4.2. Application of the Mansour-Fox algorithm. 63

Table 4.3. Summary of simulation runs using the hybrid genetic algorithm with a fixed population size, 200. 65

Table 4.4. Summary of simulation runs using the hybrid genetic algorithm with a fixed population size, 150. 66

Table 4.5. Summary of simulation runs using the hybrid genetic algorithm with a fixed population size, 100. 66

Table 4.6. Average run times for simulation runs with a constant number of generations and frequency of calling the hill climbing routine. 67

Table 4.7. Average genetic algorithm run times for simulation runs with a given number of generations and frequency of calling the hill climbing routine. 68

Table 4.8. Summary of additional simulation runs with shorter genetic algorithm run times. 69

Table 5.1. Summary of three base line runs; with (1) no attempt to balance the load, (2) using a crude load balancing scheme, (3) a hybrid GA to balance the load. 81

Table 5.2. Summary of runs with varying weights for the importance of minimizing communication. 84

Table 5.3. Summary of simulation runs using the hybrid genetic algorithm with a population size of 480. 91

Table 5.4. Summary of simulation runs using the hybrid genetic algorithm with a population size of 320. 92

Table 5.5. Summary of simulation runs using the hybrid genetic algorithm with a

population size of 240. 92

Table 5.6. Summary of simulation runs using the hybrid genetic algorithm. Runs with the given population size and frequency are averaged together. 93

Table 5.7. Summary of simulation runs using the hybrid genetic algorithm. Runs with the given number of generations and frequency are averaged together........95

Table 5.8. Summary of SP2 runs. 98

Table 5.9. Summary of SP1 runs with a different scenario. 101

Table 5.10. Comparison of run times using the GA and bisection methods.................102

Chapter 1. Introduction, scope, and overview.

1.1. Introduction.

This thesis discusses heuristics used to decrease the run time of parallel adaptive grid programs by maintaining good load balance and maintaining good communication performance. These heuristics use, as their core, a hybrid genetic algorithm (GA). The purpose of the GA is to dynamically map portions of the grid to various processors to maintain good load balance and low communication. Created as a result of this research effort, this hybrid genetic algorithm provides a marked improvement over an algorithm developed by Mansour and Fox (1991). The improvements in the algorithm allow it to run faster than the Mansour and Fox algorithm yet enable it to still return high quality solutions, that is one that has good balance and low communication. The decrease in run time of the GA, along with the high quality of the solution returned, allows an example calculation to run 75% faster than the same simulation run as a parallel application without the GA. (This measurement of speedup and the example calculations are described in more detail below.)

Although the improved GA forms the core of the algorithm for decreasing the run time, other heuristics have been developed which help to decrease the run time of the adaptive grid program. These heuristics use domain specific knowledge to help the GA run effectively, that is, return a high quality solution. In this thesis the improved hybrid genetic algorithm is described, along with the heuristics which enable it to run effectively. A series of experiments are described that track the development of the genetic algorithm and other heuristics. This research thus shows that domain specific knowledge can be readily incorporated into the genetic algorithm to improve overall performance.

If the algorithms described in this thesis were only applicable to a single example calculation, they would be of little value. Although the example adaptive grid program used to test these new algorithms is fairly general, even more can be said about the algorithms applicability to other simulations. It is shown that what was learned from a early set of experiments using an adaptive grid program could be applied to different, but similar adaptive grid program runs. One of the features of the improved hybrid genetic algorithm is that its behavior is more tunable than a standard GA. It is shown that tuning the GA for a particular adaptive grid program run enables similar adaptive grid program runs to run effectively. In addition to showing the effectiveness of the GA for improving the run time of the adaptive grid program, it is expected that the information provided in this document will find use in other areas of optimization because of the generality of the methods derived as a result of this work.

1.2. Scope of the research effort.

The research effort described in this thesis deals with adaptive grid programs. Why is there an interest in improving the run time of parallel adaptive grid programs? Simply stated, they are used often and on many types of platforms. Adaptive grid programs are used for many applications including weather modeling, aerodynamics and the simulation of explosive dynamics. Such programs are run on many types of platforms including single processor PCs, and single processors of high end super computers such as the Cray T90.

Adaptive grid programs are also run as parallel applications, using two to thousands of processors. Relative to using thousands of processors, the research described herein concentrated on a smaller scale parallelism, using up to 16 processors at once. Dedicated large scale parallel processing machines, consisting of hundreds or even thousands of

dedicated processors, are not generally available to most researchers. What is available to many people is a smaller collection of processors on workstations, connected together with some type of network. It is often possible to use these workstations as nodes dedicated to a particular application until the application is completed. The SP2 running under loadleveler is such a system. Also collections of workstations that can be used to form a parallel architecture are generally available. An office that has 10 to 20 workstations can dedicate some time, say between 2 and 5 AM every morning, to run parallel simulations with no outside interruptions. This research applies to such systems also.

Because of their general availability, this research effort concentrates on such parallel architectures, using smaller numbers of processors connected together with a network. We also assume that the processors will only be required to run a single task while running the parallel application.

There have been recent efforts, such as Condor (1997), to connect hundreds or even thousands of workstations, spread over a campus or even between cities, together as a parallel processing system. Such systems use computer cycles that would otherwise go unused. If the owner of one of the workstations starts a task on a node which is part of the parallel processing system, the parallel application is evicted from the processor. The measurement of the performance of the heuristics discussed herein for architectures where there is a possibility of tasks being evicted from processors is left as an area for future research.

Although this research concentrated on small scale parallelism using dedicated processors communicating using a network, the algorithms and heuristics described herein could be modified as appropriate to be used on other architectures. In particular, the

fitness function could be modified to account for the change in communication performance for a slower network or faster shared memory communication. Modifications could be made to the algorithms to account for the eviction of tasks from processors. In fact, the GA could be used to assign the portion of the grid which was contained on terminated processors to other processors. Research into the effectiveness of the GA to reassign such portions of the grid to other processors could form the basis of another dissertation effort.

Clearly, not all adaptive grid programs can be tested as part of this research effort and those adaptive grid programs that are tested can not be run on every platform. To bound the research effort, a subset of all adaptive grid programs and all platforms was chosen to work with.

Because of their generality, it was decided that the research effort would concentrate on working with an adaptive grid program which solved Euler's equations of fluid flow. One of the reasons Euler's equations are a good choice is that they are very general and are used for many types of investigations. For example, even though there are equations that capture particular nuances of weather modeling, aerodynamics and explosive dynamics that are not captured by Euler's equations, Euler's equations are still used for simulations of these phenomena because of their generality.

There are many ways that can be used to maintain good load balance for grid programs. See the following: Walshaw and Berzins (1992), Walshaw and Berzins (1995), Williams, (1990), Ozturan, deCougny, Sheparard, and Flaherty (1994). The last reference categorizes the methods in three ways, recursive bisection, probabilistic methods, and iterative local migration. Wheat's (1992) method falls in the last category. He

describes a method based on tiling and local migration. His method is discussed in more detail in section 1.11.2. The genetic algorithm is considered a probabilistic method.

One of the purposes of the studies discussed herein, and the thesis statement for this research effort, is to show that:

A genetic algorithm can be effectively used to decrease the run time of adaptive grid programs by maintaining good load balance and maintaining good communication performance.

To show the relevance of this research in the of GAs on adaptive grid problems, comparisons are made to both Wheat's work and recursive bisection methods. The advantages of using the GA methodology is discussed.

1.3. Requirements for the genetic algorithm to be effective.

Following Wheat's approach, performance improvements are measured, relative to no attempt to balance the load. That is, for a given number of processors, the parallel grid program was run with no attempt to balance the load and the run time is measured. The same simulation is then run with the algorithms used to maintain balance and low communication turned on. The difference in the run times is used to determine the performance improvements.

As discussed by Wheat and summarized in his table 4.1, adaptive load balancing algorithms which run as part of a parallel program must run quickly to be effective. In fact in his "Scenario A" and using his tiling algorithm, he suffered an overall loss in performance due, as he says, "to the application terminating before the improvement

from tiling could recover the cost of tiling." He goes on to state that, if the application would have been run longer, the cost of tiling would have been recovered and an improvement in run time would have been seen. The goal and accomplishment of this research is to show that the algorithms used to balance load and reduce communication recover all of the their cost and reduce total run time.

This document discusses heuristics used to maintain good load balance and to maintain good communication performance for adaptive grid programs. For these heuristics to be effective, they must run quickly. Genetic algorithms have not been used for this application in the past because they have been viewed as being too slow to converge to a high quality solution. However, the research described in this document has shown that a genetic algorithm can be used to derive a reasonable solution and that solution can be effective in increasing the performance of adaptive grid programs, even when the run time for the GA is counted. How is this possible? It is imperative that additional heuristics be used to aid the genetic algorithm and that the GA be designed to run quickly. The purpose of these heuristics is to decrease the run time of the GA and increase the quality of the solution returned.

It is shown that what is learned from one run of an adaptive grid program using the GA to increase performance can be applied to other similar adaptive grid program runs on different problems and even different machines. Thus, many of the techniques described in this document may be applied to other adaptive grid applications.

It is hoped that the techniques described in this document will lead others to develop heuristics to use with and without genetic algorithms to improve the performance of applications. In particular, because this document describes optimization systems which

must run in a time critical manner, it is hoped that inspiration can be obtained for the development of additional optimization systems that run quickly.

1.4. Overview of the remaining sections of this thesis.

This document traces the development of the heuristics and the methods used to decrease the GA run time and ultimately the adaptive grid program run time. To this end, this document consists of this introductory chapter, five additional chapters and three appendices.

Chapter two contains additional introduction and background information. There is a description of adaptive grid programs, parallel programming, and the issues associated with load balancing and maintaining good communication performance. Then there is a discussion of dynamic allocation of processors for adaptive grid programs and an explanation of why it is a difficult task. The thesis statement is restated and a description of genetic algorithms is given. The genetic algorithm method is compared to other methods, including bisection and an iterative local migration method, in particular, Wheat's tiling method.

Chapter 3 briefly describes the tools developed for this research effort and the numerics used for solving Euler's equations. For additional information, the reader is referred to the appendices. This chapter also describes the SP1 and SP2 machines used in this research effort.

Chapter 4 describes the first tests of the genetic algorithm using 4 nodes of the Albuquerque Resource Center (ARC) SP1. As the result of these tests, several heuristics to increase the GA effectiveness were developed. A 12% improvement of the performance of the adaptive grid program was seen using these heuristics in these test cases. This

improvement can be contrasted to Wheat's performance degradation on his first example and a 6% improvement on his second example.

There are two sine qua nons which must be observed to allow the genetic algorithm to increase the performance of the adaptive grid program. One, domain specific knowledge must be used. Two, improvements must be made to the standard GA. Chapter 4 discusses the use of some domain specific knowledge. Also, Chapter 4 discusses the Mansour-Fox algorithm and improvements to their algorithm developed as part of this research effort. Mansour and Fox have described a hybrid GA which they used for static allocations of sections of grids to processors. Chapter 4 describes their algorithm and improvements made to it to decrease its run time. Chapter 4 also mentions a fast mutation procedure developed for this effort. The method is described in more detail in Appendix C.

Chapter 5 describes three other example calculations using 16 nodes of the SP1 and the Maui High Performance Computing Center's SP2. It is postulated that what was learned from the 4 node SP1 calculations is general enough to apply to other configurations or calculations. This is then demonstrated on the 16 node SP1 calculations, even though the calculations were significantly different. The benefit of tuning the GA to a specific problem is shown. That is, it is shown that there exists a set of GA control parameters which produce a better performance for the particular example calculation. Also what was learned from the SP1 16 node calculation, including the tuning, was applicable to the SP2 calculation and to different SP1 16 node calculations. A 75% performance improvement was seen on the SP2 calculations compared to the same parallel run without the GA.

Chapter 6 is a summary. Is discusses the discovery of the two sine quo nons that

improvements must be made to the GA and the use of domain specific knowledge. The improvements to the GA are discussed that enable faster arrival at high quality solutions. The use of domain specific knowledge to aid in finding high quality solutions is also summerized. This final chapter also discusses the heuristics which were developed as part of this research effort and gives recommendations for the use of these heuristics for other applications. Ideas are presented for future follow-on research efforts.

The three appendices give details of the programs used for this research.

Chapter 2. Additional introducation and background information.

2.1. What is a grid program?.

For the purposes of this thesis a grid program is a computer program which performs a time evolving simulation of a physical system using a collection of data. The data is an estimate of some physical quantity at various points in a two dimensional space. The points are arranged in a two dimensional grid. In general, the values of the variables change as the simulation progresses. The simulation is time stepped, that is, the values of all of the variables are calculated at time equal T_0, and then values are calculated for time equals T_1 with $T_1=T_0+\Delta T_1$ where ΔT_1 is some nonzero, usually positive, quantity. The time stepping is repeated for time equals $T_2=T_1+\Delta T_2$ and so on. To calculate the values of variables at time equals T_1, the values of the variables at time T_0 are used, that is the value of the variables at some time step are a function of the values of the variables at the previous time step and often the time step size.

Grid programs are often used to obtain numerical solutions to partial differential equations (PDEs). To find a numerical solution to a PDE, the region of interest is divided into a collection or grid of cells. In two dimensions the cells are often rectangles. A cell can be thought of a finite volume surrounded by a collection of edges. Two or more cells share an edge. Discrete approximations to the derivatives in the PDE are derived with the values of the variables assumed constant within each of the cells. This discretization replaces a continuous differential equation with a set of algebraic equations. The coefficients of the algebraic equations derived from the discretization are determined by the values of the cell variables. The algebraic equations are solved to advance the solution from T_0 to T_1. Depending on the method of discretization, the value of the cell variable at the next time step can be a function of the surrounding 4 or 9 neighbor cells. See Appendix A and

Celia and Gray (1992) for more information.

A cell is used to represent a small area, a subset of the entire area represented by the grid of cells. In general, the smaller the area represented by a single cell the more accurate the numerical approximation to the PDE becomes. In regions of low derivatives typical numerical techniques used for many grid problems have an accuracy of order h^2. The accuracies are also proportional to the values of the derivatives of the variables. In regions of discontinuity, such as shock fronts, methods of order h are used. See Sod (1976), Lapidus (1967), Berger (1982), Berger (1984), Berger (1985),Berger (1987) and Press, Flannery, Teukosky, and Vetterling (1986), Stoer and Bulirsch (1984). With accuracy of order h^2, the error in the numerical approximation is proportional to the dimensions of the cell to the second power. Thus, decreasing the size of a cell by 1/2 in a given direction increases the accuracy of the approximation in that direction by a factor of 4.

To increase the accuracy of a calculation it is desirable to have a grid with a large number of small cells. Having a larger number of cells requires more machine memory and causes the calculation time to increase. One way to decrease the running time for a simulation which uses a large number of cells is to use parallel computing, another way is to use irregular and adaptive grids.

2.1.1. Irregular grids.

For many grid programs, especially programs designed for computational fluid dynamics or solving Euler's equations, it is desirable to use irregular grids. Irregular grids are those which have cells of different sizes in different regions of the grid. That is, in some regions the cells may represent an area of 1m by 1m while in some other area of

the grid the cells may represent an area of 0.125 m by 0.125 m. Clearly, it takes 64 cells of size 0.125 m by 0.125 m to represent the same area as one cell of the size 1 m by 1 m.

Grid calculations are often used to estimate some quantity, such as pressure, as a function of space and often time. The spacing of the grid determines the resolution of the spatial gradients of pressure which can be resolved. To resolve large gradients, fine grids are required. Often a phenomenon being simulated will contain regions with large gradients and regions with small gradients. Large gradients increase the memory requirement for a calculation because large numbers of elements are required to resolve the gradients. To reduce memory and computation requirements and still resolve areas with large gradients, it is desirable to have fine grids where required and coarse grids where they can be tolerated.

2.1.2. Adaptive gridding.

For simulations that evolve as a function of time, the regions requiring fine gridding can move. Also, over time, regions develop that have high gradients, for example when two pressure fronts merge. It is desirable to increase the resolution of the grid in these areas. Adding, or removing cells is called dynamic regridding, adaptive gridding, or adaptive mesh refinement. When the resolution for an area is increased, additional computational elements are added. The computation time required to update the variables in the areas where cells are added is greater because additional work needs to be performed for each computational cycle. Adding cells to the calculation grid will cause the program to run slower. Again, one way to decrease the running time for a simulation which uses a large number of cells is to use parallel computing.

2.2. What is parallel computing?

Parallel computing is the use of multiple computers or processors working together on a common task. The task is divided among the various processors and each processor works on its section of the problem or subtask. In order to complete its task, a processor is allowed to exchange information with other processors working on the same task. Dividing the task among various processors is known as task partitioning and the exchange of information is simply known as communication. See Kumar, Grama, Gupta and Karypis (1994).

The purpose of parallel computing is to decrease the time required to complete a given task. By breaking a task into, say N subtasks, and allowing N processors to each work on a single subtask, the total time required to complete the task can often be reduced.

Task speedup, or just speedup, is defined as the execution time of some task and algorithm when run on a single processor divided by the execution time for the task and the same algorithm when it is run on multiple processors. Parallel efficiency is a measure of how efficiently the processors are being used to solve a problem. Parallel efficiency is determined by dividing speedup by the number of processors used for a problem, Maccabe (1993). With 100% parallel efficiency the execution time for a task run on N processors is 1/N times the execution time on a single processor.

The analysis of the performance of parallel systems is often done as a function of problem size. The concept of problem size can be formalized. Often the problem size is taken to be the length of the string used to encode an instance of the problem, assuming some reasonable encoding scheme. Here we use a different but related metric. The size of a problem, **W**, is the number of computational steps required to find the solution using

the best known sequential algorithm. For many common problems such as matrix inversion, addition, or multiplication, the size of the encoding is polynomially related to **W**.

Suppose it takes time T_p to solve some problem on p processors and it takes time W on a single processor. We define the overhead function T_0 as the difference of W and T_p times the number of processors used.

Equation 2.1 $$T_0 = pT_p - W$$

With 100% efficiency, $T_0 = 0$. T_0 is always non-negative, assuming the algorithm used on the single processor is the same algorithm used on P processors.

Amdahl (1967) gives an upper limit for the speedup of a task. For a given task, some fraction, F_P, of the task can be run in parallel. Call the fraction which can not be run in parallel, that is, the serial portion of the task, F_s. The serial portion of a task can not have its execution time reduced by running on multiple processors. If we make the optimistic assumption that by using N processors the parallel portion of the task can be run N times faster, Amdahl's law gives as the lower limit on the execution time of a task:

Equation 2.2 $$ExecutionTime_{\min} = ExecutionTime_{one}\left[F_S + \frac{F_P}{N}\right]$$

where $\text{ExecutionTime}_{\text{one}}$ is the time take to run the problem on a single processor. The maximum speedup is then

Equation 2.3 $$Speedup_{\max} = \frac{1}{F_S + \frac{F_P}{N}}$$

The maximum speedup is limited by the portion of a task which can not be parallelized. If 10% of a task cannot be parallelized then the maximum speedup is 10, even if an infinite number of processors are used.

2.3. Parallel architectures.

2.3.1. Flynn's taxonomy.

Flynn (1972) has devised a classification scheme for describing multiple processor computing systems. Flynn's taxonomy is based on the number of data streams and the number of instruction streams used by a parallel computing system. A single processor machine working alone has a single instruction stream and single data stream. Such a system is known as a SISD machine for single instruction and single data streams. SIMD stands for single instruction stream, multiple data stream. In this type of system, all processors or processing elements execute the same instruction in lock step but each processor works on different data. Vector processors are SIMD machines.

MIMD stands for multiple instruction streams with multiple data streams. The processors in a MIMD system operate more independently than in a SIMD system. In a MIMD system the various processors working on a common problem may execute the same program or each may run different programs. In either case the processors are not in lock step. When running the same program, the processors may be executing in different sections of the same program, possibly even different procedures. When the processors of a MIMD system execute the same program, such a system is often called SPMD for single program multiple data. Note that SPMD systems are a special case of MIMD systems. This dissertation primarily discusses SPMD systems.

2.3.2. Memory architectures.

Parallel computing systems can also be classified by their memory architecture. As mentioned, when working on a task, the various processors often need to communicate information to each other. The method of communication is often determined by the memory architecture. There are two primary memory architectures for parallel computers, shared memory and distributed memory. In shared memory multiprocessors, all processors in the system have equal capability to access all of the memory of the system. In a distributed memory multiprocessing machine, each processor has its own local memory which is not directly accessible by other processors. Information is passed from one processor to a second when the first processor sends a message to the second processor and the second processor receives the message. This dissertation primarily discusses distributed memory systems. For a survey of parallel architectures see Tanenbaum (1994) or Maccabe (1993). See van de Goor (1989) for a description of a taxonomy based on the orthogonal concepts of control and data.

2.4. Parallel programming and grid programs.

This section discusses parallel programming for grid programs. In particular it discusses mapping of cells to processors. Issues associated with load balancing and good communication performance are also discussed.

2.4.1. Parallel programming for regular grids.

The section on parallel programming was introduced by saying that parallel programming can be used to speed up grid programs. How does this work? Various portions of the grid are assigned to different processors. That is, different processors hold the data for different portions of the grid. The processors are responsible for performing the calculations for the portions of the grid for which they hold the data. As discussed briefly in Section 1.5, the value of cell variables at the next time step is a function of its

neighbors. If a processor holds a cell which has a neighbor on a different processor then the values from the neighbors must be communicated before the calculation can proceed to the next time step.

The speed up of the parallel grid program is determined by two factors, load balance and communication time. Speed up is greatest when the load is balanced and the communication is minimized. A parallel calculation is said to be in balance if each processor has the same amount of work to perform. If we assume, as is the case for some grid calculations, that the computation time for each cell is identical, then load balancing is trivial. Assign an equal number of elements to each processor. The manner in which the data is distributed across the processors can effect the communication time and thus the run time. If the grid is regular and rectangular, balancing load while minimizing communication is trivial. The grid is broken into a number of rectangles equal to the number of processors. The communication is amongst processors which hold data for adjoining rectangles.

2.4.2. Parallel programming for irregular and adaptive grids.

If the grid is irregular, that is, it has a finer resolution in some areas than others or the elements are different shapes, then the distribution of data cells in order to attain the minimum amount of communication with good load balance is not obvious. Given a set of processors and an irregular grid, the problem of finding if there is a distribution of cells which balances load and which meets a given requirement for communications is difficult. As discussed by Varadarajan and Hwang (1994), when this problem is stated in a formal way it can be shown to be NP-complete.

For simulations that evolve as a function of time, the regions requiring fine gridding

can move. For programs with dynamic calculation requirements, the initial allocation of data to processors can become less appropriate as time progresses. In addition, the communication between processors will rise above the optimal level. As discussed above, when performing load balancing for irregular grids, the communication patterns must be addressed to maximize performance. The irregular nature of the grid can evolve as a function of time so that the load balancing must also be done dynamically. Unfortunately, each time the grid is changed, to find the best distribution, an NP-hard problem must be solved. It is impractical to solve the problem optimally for each regridding. Heuristics must be used to find good solutions to the problem. This dissertation primarily deals with heuristics to find a partitioning of the cells to processors for irregular grids which have both good load balance and good communication patterns, but not necessarily the optimal partition.

2.5. Thesis statement revisited.

To date, genetic algorithms have not been used to perform fine grained load distribution for dynamic adaptive grid calculations. Related endeavors have been undertaken. Fine grained load balancing using other techniques has been done for adaptive grid programs. See Section 1.11 below for a brief discussion and references to additional methods. Genetic algorithms have been applied to load distribution for static irregular grids, Mansour and Fox (1991). Genetic algorithms have been used for dynamic course grained load balancing, where entire tasks are moved between processors. See for example Kidwell (1992) and Muenetome, Takai, and Sato (1994). Parallel genetic algorithms have been used for other related NP-hard problems and combinatorial optimization. See for example Levine (1994) and Bianchini and Brown (1993) and Watson (1995). For a discussion of NP-hard problems see Garey and Johnson (1979) or Moret and Shapiro *Algorithms from P to NP, Volume 2* to be published.

Genetic algorithms have not been used in the past for the purpose of mapping cells to processors for adaptive grid programs. As discussed by Williams (1990), probabilistic methods are viewed to require too many iterations and are too expensive to use for this problem. The purpose of this dissertation is to show that a genetic algorithm, a probabilistic method, can be used to derive a fine grained load distribution for dynamic adaptive grid calculations. To state this formally, the following thesis statement is offered.

Thesis Statement:

A genetic algorithm can be effectively used to decrease the run time of adaptive grid programs by maintaining good load balance and maintaining good communication performance.

2.6. Basic genetic algorithms description.

This section introduces genetic algorithms. More information can be found in Fogel (1996) and in Appendix C.

Genetic Algorithms, or GAs, simulate the process of evolution to optimize a function. The algorithm is simple. A potential solution for a particular optimization problem is represented by some string, usually a string of bits. There is a function which takes potential solutions as input and evaluates them. This function is called the fitness function. Good solutions to the problem are given a high fitness. The algorithm proceeds as follows:

```
(1) Generate a collection of potential problem solutions.  This is
    normally done in some random fashion.  This collection is called the
    population.  An individual member of the population is called a
```

```
    chromosome and each bit of the chromosome is a gene.

(2) Repeat until done.
    Find the fitness of the members of the population.

    Select half of the population to reproduce.  The members of the
    population with the higher fitnesses have a higher probability of
    reproducing.

    Discard the half of the population which does not reproduce.

    Allow the remaining half of the population to reproduce, replacing
    the old population.

    Allow potential mutations in the populations.
```

"Repeat until done" means repeat for a given number of iterations or repeat until there is a member of the population with a given fitness or until there is no change in the population for some given number of generations.

How do solutions reproduce? There are several methods possible and one is described next. Given two solution chromosomes, split them at some arbitrary and random location. Recombine the pieces. An example follows. P1= "abdefg", P2= "1234567". Split the strings after the third character. The new string produced, P3, has as its characters the beginning of P1 and the end of P2. P3= "abc4567". This method of reproduction is called crossover.

Mutation is the random flipping or setting of bits or genes. Each gene will mutate with some small probability. Fogel (1996) describes this process.

2.7. Review of two other methods.

2.7.1. Bisection.

Many of the heuristic techniques used to assign sections of grids to processors are based on recursive bisection. Given a problem domain, the algorithms recursively subdivide the domain into two parts. The various bisection techniques differ in the manner in which they divide the domain. Three of the techniques are recursive coordinate bisection, recursive graph bisection and recursive spectral bisection. Recursive coordinate bisection produces partitions which can lead to large amounts of communication. Recursive spectral bisection tends to yield well balanced, well connected partitions to minimize communication but it tends to be computationally intensive. For more information and a description of recursive coordinate bisection see Kumar, Grama, Gupta, and Karypis (1994).

This section describes a different bisection method for partitioning cells to processors. This method is probably not new but no references to it have been seen. The method is similar to recursive coordinate bisection and is called recursive centroid bisection. Some of the example calculations were run using this method to compare its effectiveness to that of the GA. The calculations for which the GA was used to assign cells to processors ran faster than the calculation for which the recursive centroid bisection method was used.

Assume there is a collection of cells to be distributed to N processors. N is a power of two. The x,y coordinates of the cells are known. Let Z be the collection of cells and P be a vector of length N, containing the processor numbers for the processors. The algorithm proceeds as follows:

```
dir=-1
call bisect(Z,P,dir)
```

Routine bisect is the following:

```
if(length P = 1) assign Z cells to processor P and return
```

```
if(dir=-1)then
       find x value of the centroid of Z
       split cells into two groups, Z1
       with x < centroid and Z2 with x > centroid.
else
       find y value of the centroid of Z
       split cells into two groups, Z1 with
       y < centroid and Z2 with y > centroid.
endif

split P into two halves P1 and P2
dir=dir*(-1)
call bisect(Z1,P1,dir)
call bisect(Z2,P2,dir)
```

This algorithm produces N partitions of cells. The partitions are not guaranteed to have the same number of cells. The algorithm will do a better job than recursive coordinate bisection of assigning clusters of cells which are nearly collocated to the same processor. This can be seen in the one dimensional example shown below.

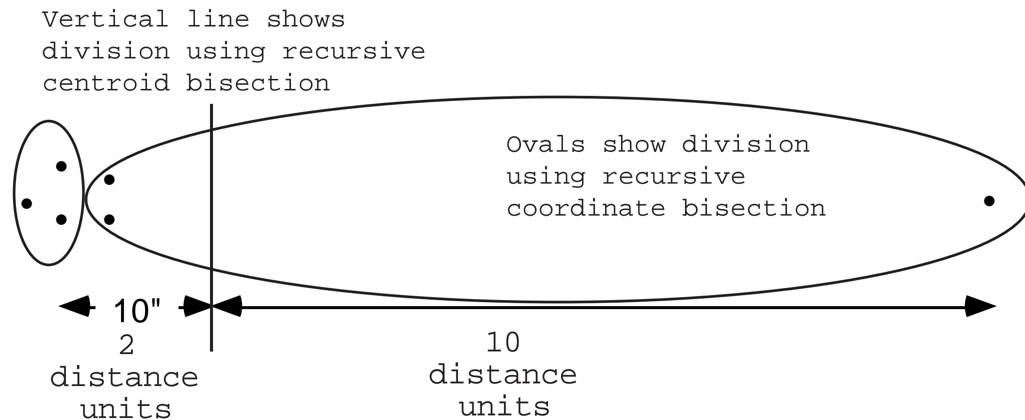


Figure 2.1. Division of cells using bisections methods.

As discussed, one of the criticisms of recursive coordinate bisection is that the partitions it produces can lead to large amounts of communication. Simulations were run using the method outlined above, recursive coordinate bisection, and a GA to distribute cells to processors. The communication times were longest using recursive coordinate bisection. For the example calculations, the use of the GA produced run times which

were 21% faster than the same simulation when the recursive coordinate bisection was used and 13% faster than when recursive centroid bisection was used. The increase in performance was because of a decrease in the communication time.

2.7.2. Wheat's tiling.

For some grid simulations, even with regular grids, the processing cost associated with each element may not be constant. Changes in processing costs may result from the dynamics of the physical systems being modeled. The processing time associated with a cell can be a function of both time and location in the grid.

Based on work by Leiss and Reddy (1989), Wheat (1992) developed a fine grained data migration scheme to perform load balancing for such applications. Initially an equal number of data elements may be assigned to each processor. Each processor is considered to be part of a neighborhood of processors. Processors in the same neighborhood hold data elements which are close to each other on the grid. A processor can be a part of more than one neighborhood. The load balancing phase consists of several steps.

Initially all processors determine their work load. The work load is simply a measure of the time required to complete a given number of iterations of the calculation. Next the average of the work load in each neighborhood is determined. Processors which are lightly loaded request data elements from the processor with the greatest load. A processor which receives requests for cells prioritizes work requests by size. Requests are satisfied until the new work load is below the neighborhood average. Finally the processor that off-loads work determines the elements to send to requesting processors and sends the data elements to the processors.

One key difference between Wheat's work and the work discussed herein is that Wheat restricted the movement of element to neighboring processors. Although a GA could be developed to enforce this restriction, no such restriction was used for this research effort. This task is left as a future research effort. It would be a worthwhile research effort because this restriction may produce partitions with less communication. With this restriction in effect, the GA would be doing nothing more than controlling migration.

One advantage to the genetic algorithm approach to distributing cells to processors its versatility. The restriction discussed above could be enforced by a simple modification to the fitness function. Thus by a simple change the algorithm is changed from a global balancing technique a local migration technique. This versatility was one of the main reasons a GA was chosen as the basis of this research effort. It was known that a traditional GA would not perform well in the effort of mapping cells to processors. It was postulated that because of its versatility, the GA could form the basis of an effective mapping algorithm.

There is another significant difference in this research and Wheat's work. The simulations which were used to test the mapping algorithms were much more dynamic in their memory and computational requirement. Wheat's grid was regular and was not adaptive, cells were not added or removed. For his first two example simulations, the calculation work per cell was constant in time as was the total memory requirement for the cells. Also the amount of calculation time required per cycle per cell was a known function of the spatial coordinates. Thus, he could have used a preprocessing routine to assign cells to processor as Mansour and Fox (1991) did in their work. For his final example, as he calls it, his "validation run", Wheat also used a fixed grid. For this example calculation, the per-cell processing costs were not known a priori. In his future

directions section, Wheat briefly discusses dynamic loading but does not give any data for performance improvement using his algorithm.

The research discussed herein goes beyond Wheat's efforts. As discussed above, the grids for this work are not only irregular but adaptive. The computational loads are a function of both time and location. Thus, there are an additional two levels of complexity in the simulations used to test the cell mapping algorithms. While Wheat's work addressed simulations that were relatively static, it is shown in this research effort that the GA could form the basis of an algorithm that was able to addapt to the extra levels of complexity and dynamics.

Chapter 3. Discussion of hardware and software tools.

3.1. The target architecture.

As discussed in Section 1.2, the research described herein concentrated on a smaller scale parallel system, using up to 16 distributed memory processors that are connected together by a network and communicate using message passing. This research effort concentrated on such a parallel distributed memory system because of its general availability. Common collections of workstations can be used to form a parallel system. An office that has 10 to 20 workstations, can dedicate some time, say between 2 and 5 AM every morning, to run parallel simulations with no outside interruptions. The IBM SP1 and SP2 can also be utilized as such a system, a collection of workstations connected together with a high speed network and using message passing to communicate.

The research discussed herein was performed using the Albuquerque Resource Center (ARC) IBM SP1 and the Maui High Performance Computing Center (MHPCC) IBM SP2. The SP1 and the SP2 are MIMD distributed memory machines. The IBM loadleveler software allows the nodes to be dedicated to a particular parallel task until the task is completed. The hardware for the machines is described in more detail below. A description of the loadleveler system can be found in the *IBM AIX Parallel Environment: Operation and Use* manual (1994). The nodes of the machines communicate using message passing. PVM and MPI are the two principal message passing systems used on the SP1 and SP2.

PVM is one of the most popular libraries, developed by Geist, Beguelin, Dongarra, Jiang, Mancek, and Sunderam, (1994). PVM stands for Parallel Virtual Machine. One criticism of PVM is its relatively low performance. Message latencies are high and

bandwidth is low. Because of its relatively low performance, PVM was not used for this research effort.

MPI was used as the message passing system for this research effort. MPI or message passing interface was developed by the Message Passing Interface Forum (1994). MPI is not a library but a standard for a message passing library. The standard dictates the routines which are to be available in a compliant library. Bindings are provided for C and Fortran 77. Routine invocation methodologies and the semantics of the library calls are dictated by the standard. The actual methodology for achieving the semantics of the routines is not dictated but left to the implementer. MPI's main advantage is that it is an industry wide agreed standard which can be adopted by vendors to promote portability of parallel programs. Other advantages of MPI are discussed in Appendix A.

3.1.1. Description of the MHPCC SP2 hardware.

Because the SP1 and SP2 were used extensively for this research effort, a slightly more detailed description of these machines will be given next.

The MHPCC IBM SP2 is a 400 node parallel processing machine. (The maximum number of nodes available on an SP2 is 512. Because of the way it is configured, the maximum number of nodes that can be used by one program on the MHPCC SP2 is 128 but most applications use between 8 and 32.) Each node of the machine contains a 66-MHz Power2 IBM processor. The nodes have a minimum of 64 Mbytes RAM and are capable of 264 MFLOPS.

The SP2 is partitioned into logical frames of 16 nodes. The individual nodes within the frame and the frames are connected by an IBM proprietary switching technology,

Stunkel, Shea, and Abali (1994). Each logical frame contains 16 processors and a switch board. See figure 3.1. Each switch board contains 8 logical switch elements. The elements are wired in a 4 way to 4 way bidirectional scheme.

Processors are connected to the "left" side of the switch board. There are 16 bidirectional lines on the "right" side of the board. These lines are used to connect to other frames. SP2 systems with up to 80 processors, 5 frames, are constructed by directly connecting the frames using the links on the right side of the switch board. From a given frame, 4 lines are routed to each of the other 4 frames.

For SP2 systems with more than 80 nodes, the configuration is different. Cascading switch boards are used. For an SP2 with 128 nodes, 8 frames and 4 intermediate switch boards are used. 16 lines are available for connection to the outside world from the switch boards on each frame. The 16 lines from each frame are connected to the 4 switches in sets of 4. For larger configurations an additional level of cascading is used.

For all configurations supported by IBM, the SP2 has a message latency of 30 microseconds and a bandwidth of 40 Mbytes/second, application to application. The hardware latency is much smaller, but depends on the number of cascade switch boards.

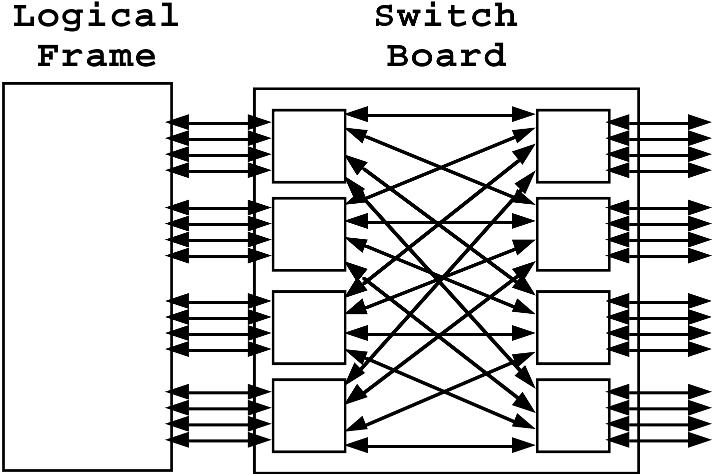


Figure 3.1. SP2 logical frame and switch board.

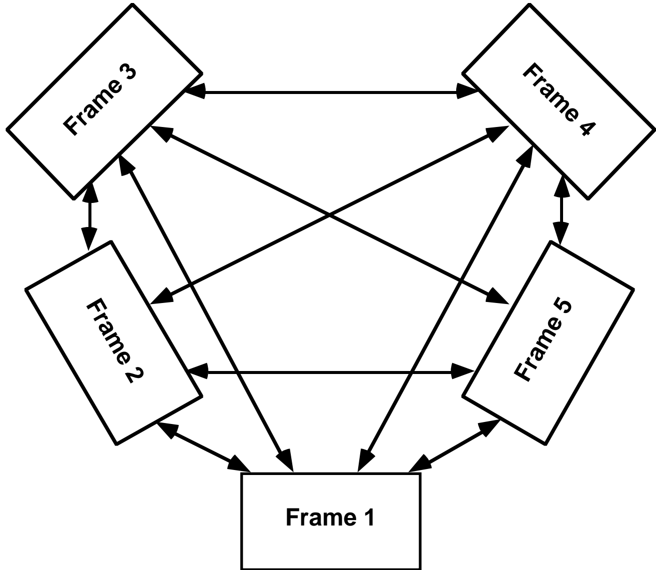


Figure 3.2. Diagram of a SP2 with 80 nodes and 5 frames where each line represents 4 connections.

3.1.2. Description of the ARC SP1.

The SP1 is the forerunner of the SP2 and it is very similar but has slower processors and message passing hardware. The ARC SP1 has 32 nodes, but at most 16 of the nodes can be used on any one problem. Each node contains an IBM POWER 1 processor and 64 Mbytes. SP1 nodes are networked together using IBM TB0 adapters instead of the TB2 adapters used in the SP2. The bandwidth of these adapters is about 8 Mbytes/second.

What is the performance relationship between the SP1, the SP2, and a collection of workstations? The importance of the SP1 and the SP2 machines is that they represent a level of performance comparable to a network of workstations available to organizations for about $3,000/node. A system could be constructed using commercial off the shelf (COTS) components. The floating point performance of the SP1 nodes is comparable to that of a system based on a 120 MHz Motorola 603e chip. Such systems are available from mail order catalogs with 16 Mbytes of RAM for $1700, including the monitor, and a 1 Gbyte disk drive. The additional memory would cost about $500. 100Base-T networking cards could be purchased for about $250. One or more hubs would be required at a cost of about $1000 for one with twelve ports. The Linux operating system and the other software that could be used to drive the system is available free. A public domain implementation of the MPI standard, MPICH, is available for almost all common Unix and Linux platforms, Gropp and Lusk (1996). MPICH could be used to provide the message passing for the COTS system.

3.2. The software tools and numerical routines.

As with any computer, the hardware and system software is only part of the story of the machine. The software tools used on the computer are equally important. As part of this research effort, a collection of software tools and numerical algorithms were developed. The program used to test the heuristics for assigning cells to processors was a combination of the PLIFE framework, the GA framework and the numerical routines, all created for this work.

The next subsection describes the PLIFE framework. The framework was used as the basis of the parallel adaptive grid program. The development of data structures for

adaptive grid programs is a subject of on-going research. See Gittings (1994) and Yiu, Greaves, Cruz, Saalehi, and Borthwick (1996).

Section 3.2.2 discusses a framework for a parallel GA. This framework was used to create the GA for this research effort. Parallel genetic algorithms are also the subject of recent research efforts, Bianchini and Brown (1993).

The final subsection in this chapter discusses Euler's equation. Numerical algorithms were developed to solve Euler's equations of gas dynamics. The numerical methods developed to solve Euler's equation on an irregular grid are discussed in Appendix B. The published data on this subject is extensive. Good sources for up to date information are the web pages maintained by Leveque (1997) and Larsson (1997).

3.2.1. Brief discussion of the PLIFE framework.

PLIFE is a framework for creating two dimensional, parallel, adaptive grid, or adaptive mesh programs. The primary purpose of the development of PLIFE was to have a tool for the study of load balancing techniques for adaptive grid programs.

A program tool used to evaluate the research for this dissertation must have several characteristics. First, it must perform some type of numerical calculation. The calculation must be a simulation of a physical system using a grid of data points or cells. The tool must operate as a parallel program, with sections of the grid assigned to various processors. The program must allow adaptive grids. That is, the program must have the capability to change the grid resolution in particular locations. This is done either by taking a particular cell and breaking it into a collection of cells, or by taking a collection of cells and combining them to create a larger cell. Finally, to test various dynamic load balancing

strategies, the program must be able to move data cells from one processor to another.

PLIFE was designed to meet these requirements to test dynamic load balancing strategies for adaptive grid programs. In addition to the basic requirements outlined above, PLIFE was designed to be flexible and portable. For additional information on PLIFE see Appendix A.

3.2.1. Brief discussion of the parallel GA framework.

CHUCK is a framework for developing a parallel genetic algorithm program. It was used to create the GA for this research effort. To use CHUCK to create a parallel GA, the user needs to add the fitness function and initialization routines. Two of the highlights of CHUCK include parallel operation and a fast mutation ability. The parallel modes of operation are discussed here. The fast mutation operation, which may be new as no references to it have been found, is discussed in Appendix C. The use of the parallel GA for this research effort is discussed in detail in Chapters 4 and 5.

There are two fundamental modes of operation of CHUCK. In the first mode the master processor holds the population and sends requests for fitness function evaluations to the slave processors. In the second mode, the population is distributed across all processors with each processor holding its own subpopulation. If the size of the total population is divisible by the number of processors, then each processor holds the same number of members of the population. Each processor does the fitness function evaluation for its subpopulation and each does its own reproduction. This is the mode which was used for this research effort.

There are many interesting variations on this parallel method of operation which

have been used in the past. See for example Bianchini and Brown (1993). CHUCK takes many of the concepts for the parallel operation of genetic algorithms and puts them into a single framework. The following are supported by the genetic algorithm used for this research.

In variation 1 each processor works independently for "I" generations. Then, a global parallel sort is done, followed by a redistribution of the top half of the population. This variation allows for a tightly to loosely coupled algorithm depending on the value of I. It can allow separate and independent evolutions to occur and in the end take the best of the independent populations. With tight coupling, (I=1), a sequential algorithm can be simulated.

In variation 2, a rectangular topology for the processors working on the problem is assumed. Each processor is allowed to work independently for "J" generations. Then an exchange is done between neighbors of some portion of the population, left-right and top-bottom. The amount of the population exchanged between neighboring processors is determined by an input parameter. There is a concern that this method is somewhat artificial, in that, it forces a topology onto a problem which does not have a topology of its own.

In variation 3, an aggression factor is assigned to each processor. Then the processors are allowed to work independently for "K" generations. Next, the aggressive processors force a portion of their population onto the other processors. It has been found that this allows a degree of randomization which helps prevent stagnation of the population.

Variation 4 is a combination of 1, 2, and 3. The threshold for the amount of each

parallel method is controlled by the input parameters.

For this research effort, a combination of methods 1 and 2 is used. The aggressive takeover algorithm is controlled by an input parameter. For this effort it was turned off. The effect of the aggressive takeover form of parallelism is left as an area for future research.

3.2.1. Euler's equations of gas dynamics.

Euler's equations of gas dynamics were used as the basis of the simulations for testing the load balancing and communication control algorithms. Euler's equations are a set of partial differential equations which predict the flow of fluids and gas. The equations predict energy, pressure, density, and velocity as a function of time. It is assumed that the gas or fluids are compressible and they flow without viscosity.

Euler's equations are used for the simulation of many types of physical phenomena. The simulation of weather is one area where Euler's equations are applicable. Calculations of flows around two and three dimensional objects, such as aircraft, can also be done. Simulations of explosions are often performed by assuming that the explosive device is a ball of hot gas. If an explosive device has sufficient energy, explosions in solids can also be simulated because the solids will act as gases under very high pressure.

Let ρ be the density of the gas or fluid, u be the velocity in the x direction, v be the velocity in the y direction, p is pressure and E is the total energy per unit volume for the gas. In Euler's equations all of these quantities can be dependent on t, x and y. In two dimensions and rectangular coordinates, Euler's equations are of the form:

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathbf{F}}{\partial x} + \frac{\partial \mathbf{G}}{\partial y} = 0$$

where

$$U = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ E \end{bmatrix}, \quad F = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ Eu + pu \end{bmatrix}, \quad G = \begin{bmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ Ev + pv \end{bmatrix}.$$

E, the total energy, is the sum of the kinetic energy and e, the internal energy of the fluid. It can be written as:

$$E = \frac{1}{2}\rho\left(u^2 + v^2\right) + \rho e.$$

Finally, it is assumed that the internal energy is a known function of the pressure and density:

$$e = e(p, \rho).$$

This relationship is known as the equation of state for the gas and it depends on the particular gas which is to be simulated. The equation of state for the internal energy of an ideal gas is of the form:

$$e = \frac{p}{(\gamma - 1)\rho}$$

with γ=1.4 for air.

The Euler system of equations is in a class known as conservation equations. Specific quantities involved in the simulation are conserved, that is, the sum over all space of a conserved quantity is a constant in time. For the Euler system, the conserved quantities

are energy, mass, and momentum.

Chapter 4. Development of heuristics.

4.1. Introduction.

This chapter describes the first tests using the genetic algorithm on 4 nodes of the ARC SP1. A 12% improvement of the performance of the adaptive grid program was seen in these test cases. This improvement can be contrasted to Wheat's (1992) performance degradation on his first example and a 6% improvement on his second example.

As the result of these tests, several heuristics to increase the GA effectiveness were developed. Also, in the course of these test cases the essential requirements and conditions for the GA to be effective became apparent. These two sine qua nons are that, (1) domain specific knowledge must be used to aid the GA and (2) improvements must be made to the genetic algorithm.

In the first part of this chapter we have a description of the example simulation, that is, the Euler's equation problem solved by the adaptive grid program used to test the heuristics. Next, we have a description of the information presented in the various charts and graphs in the rest of the chapter. Most of the chapter is devoted to a description of the results of the simulation, the heuristics which were the fruit of these simulations, and the relationship between the heuristics and the two sine qua nons.

4.2. Simulation description.

An important classic, one dimensional Euler problem is the simulation of a shock tube. It is assumed that a long tube is divided into two sections. The sections are separated by a membrane which can be instantaneously removed. The gases on each side of the membrane are at different temperatures, densities and pressures. At time equals zero the membrane is removed. Euler's equations predict the values of the variables as a

function of time along the length of the tube. This problem is often called Sod's (1978) problem after the author who used it to study various numerical techniques. Its importance is that it is often used to study numerical techniques.

There is a two dimensional extension to this problem which is often used for testing numerical routines. Assume, that at time equals zero, we have a cylinder or ball of high pressure, temperature, and density gas enclosed in a membrane. The membrane is removed and the evolution of the system is followed. This two dimensional extension of Sod's problem was the first one used to test the load balancing and communication minimization algorithms.

For the test case the calculation grid simulated a region of space from (x,y)=(0,0) to (x,y)=(1,1). In the region where $r = \sqrt{x^2 + y^2} < 0.1$ we have an initial pressure of 3.0 and a density also of 3.0. In the rest of the space the initial pressure and density was 1.0. The velocities of all of the materials at the starting time of the simulation were zero.

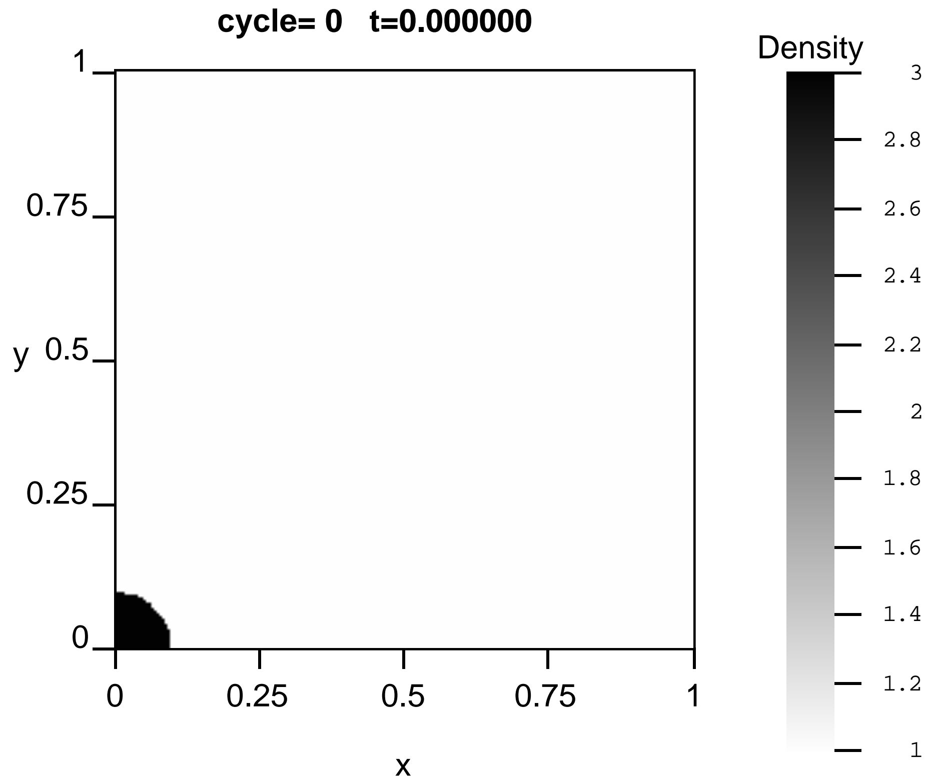


Figure 4.1. Initial density for an example calculation.

The boundary conditions along the x and y axes are reflective. This means that, in the region across the boundaries, the following conditions hold. The pressures and densities are the same on each side of the boundary. Across the top and bottom boundaries, the velocities of the materials in the y direction are opposite of each other. Across these boundaries the velocities in the x direction are the equal. Across the left and right boundaries, the velocities of the materials in the x direction are opposite of each other, while the velocities in the y direction are equal.

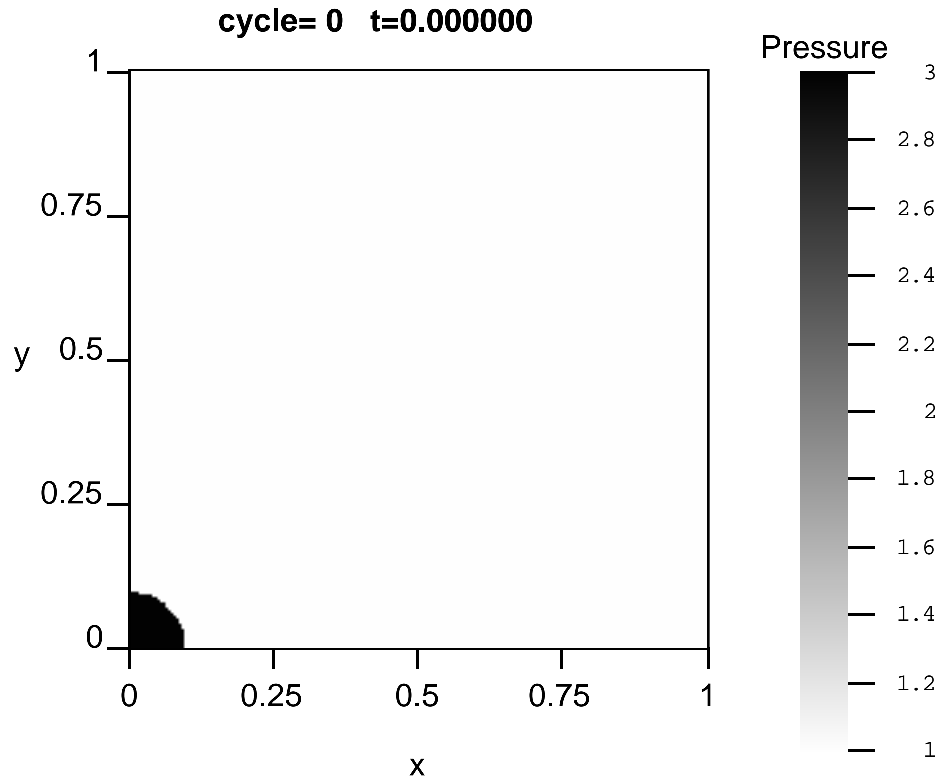


Figure 4.2. Initial pressure for an example calculation.

What is the implication of these boundary conditions? With initial conditions that are symmetric about the origin, and reflective boundary conditions, we can simulate what is occurring in the four quadrants of the plane by performing the calculation in only the first quadrant. For the problem outlined above, we simulate a cylinder by only performing the calculation for a quarter of the cylinder. The calculation time is cut by a factor of 4. This configuration was used because such types of calculations, simulating 4 quadrants using only one, are often performed in practice.

The system is allowed to evolve as the simulation time progresses. As time progresses, the material in the high pressure, high density region will spread to the region of lower pressure. Material will first start to move at the discontinuity at r=0.1. We can define two important radii, r_O and r_I. The outer radius, r_O is greater than 0.1 and r_I, the inner

radius, is less than 0.1. Only in the regions between r_O and r_I is material moving. The values of these two radii change with time, with r_O growing and r_I going to zero. These two radii are important in the heuristic used to estimate a lower bound for the communication cost for a distribution of cells. This estimation is used in the fitness function of the genetic algorithm.

Initially the calculation grid was 256x256. Grid adaptation, that is cell splitting, occurred for every cell where the difference in the velocity of the material in that cell and a neighboring cell was greater than 0.005. At a given simulation time, the cells roughly between r_O and r_I will have split.

For the first test of the genetic algorithm 4 processors were used. Each processor was initially responsible for a 128x128 section of the grid. Processor 0 was initially assigned the region of the grid in the bottom left hand corner, the corner with the region of higher pressure and density. Processor 1 held the bottom right hand corner of the grid. Processor 2 the top left corner and processor 3 the other corner. See figure 4.3.

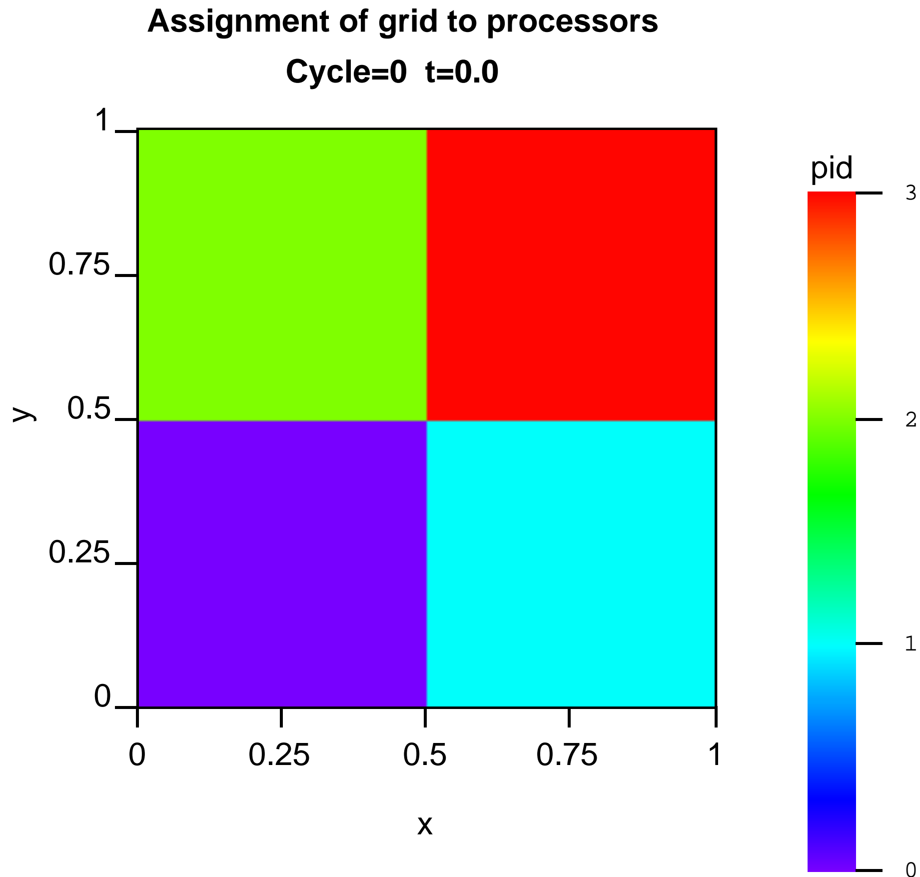


Figure 4.3. Initial distribution of cells to processors.

As cells split on processor 0 they were moved periodically from that processor. The genetic algorithm was given the responsibility to determine which of the split cells to move from processor 0. They were moved to maintain a good load balance but also in such a fashion as to maintain a small amount of communication.

The calculation was run for 1000 iterations for a simulated time of 0.233 seconds. At the last iteration 9167 of the 16,384 cells on processor 0 had split. The cells which spit are shown in figures 4.7 and 4.8.

The initial conditions describe a calculation for which most of the action occurs in

the bottom left hand corner of the grid. This section of the grid is held by processor 0. Only cells on processor 0 split. If cells were not moved to the other processors, their load would remain constant.

There was an additional restriction in effect for this calculation which was not present in later example calculations. Only one level of adaptation was allowed. For this calculation, once a cell split, its children were not allowed to split again.

The effect of this restriction is that, even if left alone, processor 0 would never get grossly out of balance with respect to the other processors. If no cells are moved, at the end of the simulation processor 0 spends 1.5 seconds per iteration in the *calculation* subroutine, while the other processors spend 0.55 seconds. The *calculation* routine is the routine that performs the numerical calculations of the simulation. The total time spent in the *calculation* subroutine is 999 seconds for 0 and 558 for the other processors. The average is about 668 seconds.

This gives the GA a difficult task. If there was no additional communication, and a perfect balance of load, the best possible decrease in run time would be 999-668=331 seconds. So to improve performance, the total of the GA run time and the increase in communication must be under 331 seconds.

4.3. Description of information presented.

Many runs of the simulation described above were performed to test the cell to processor distribution algorithms and to develop the heuristics. The charts on the following pages summarize the results of the test runs. The charts contain the following information.

We have the frequency at which the hill climbing routine was called, for example, every 3 generations. The significance of this piece of information will be explained in following sections. We have the size of the population for the genetic algorithm, followed by the number of generations for the routine. The next column contains the most important information, the run time of the simulation in seconds. The speed up is the percent increase in speed compared to the base line case, the run when no attempt was made to balance the load. Next we have the GA run time in seconds.

Return is an interesting parameter. It is a measurement of the return on the investment of running the GA. It is the ratio of the GA run time to the decrease in run time compared to the base line case. A return of 2 indicates that the overhead of GA was worthwhile. It saved twice as much time as it took to run. Any positive return indicates that running the GA saved time. A negative return indicates the run took longer than the base case.

Balance is a measure of the imbalance for a simulation. It is the average positive difference between the time spent in the calculation subroutine for each node, and the average of the calculation time for all nodes. A balance of 0 would imply that the calculation was perfectly balanced. The comm. time or communication time is the average time spent by all of the nodes in communication.

For some of the simulations the GA was not called. For these simulations where the genetic algorithm was not called, the appropriate columns are 0.

4.4. Results with no load balancing and random load balancing.

The first chart given below, table 4.1, contains the run time statistics from the base

line cases, those against which the genetic algorithm runs are compared. When there was no attempt to move load from processor 0, the simulation ran for 1451 seconds. The communication time was low, at 47 seconds. Processor 0 spent 999 seconds in the *calculation* subroutine while the others took 558 seconds in the *calculation* routine for the averagegiven above of about 668 seconds. Note that if we could balance the load with no increase in communication or any other overhead, we would have a run time of about 1451-999+668=1120 seconds.

For the second run, a probabilistic routine was called to balance the load. At specific intervals throughout the simulation, a routine was called to move cells between processors. The cells were moved to random processors but with the stipulation that the load would be balanced after the cells were moved. With this random distribution there was no attempt to move clusters of cells in patterns to bring about a low communication cost except that cells were moved in groups of 16 cells as described in section 4.6.1. This simulation ran almost 200 seconds longer, 1631 seconds. The balance parameter was in a range, between 9 and 10. Communication was high at 537 seconds. The communication cost was the chief reason for the long run time for this run.

Often	Size	Gen	Wall Time	Speed up	GA	Return	Balance	Comm. time
0	0	0	1451	0	0	0	82.7	47
0	0	0	1631	-11.0	0	0	9.9	537

Table 4.1. Base line simulation for which no movement of cells occurred and a simulation for which the movement occurred randomly but balanced the load.

In the sections that follow, the data from the simulation runs which were performed using the GA to distribute cells to processors will be presented. Before presenting this data, the fitness function for the GA will be introduced and the application of sine qua

non one, *use domain specific knowledge*, will be discussed. Intuitively, even before the first of the GA runs were performed, the importance of using domain specific knowledge was known.

4.5. Introduction of the fitness function and the request vector.

As has been stated, the purpose of calling the GA is to produce a distribution of cells to processors which has a good load balance and yet has a low communication requirement. As would be expected, the fitness function for the GA reflects the desire to return distributions with good balance and low communication. The fitness function was of the form $fitness = c_1 Fitness_{load} + c_2 Fitness_{communication}$, with c1 + c2 =1. For convenience, $\mathrm{Fitness}_{\mathrm{load}}$ and $\mathrm{Fitness}_{\mathrm{communication}}$ are in the range of 0 to 1. Thus the fitness function had a range of 0 to 1. (For more information about the fitness function, see Appendix C.)

$\mathrm{Fitness}_{\mathrm{load}}$ has a high value for distributions which have a good load balance. What number of cells should be moved between processors to produce a good load balance? The routine *Find_req* determines a goal of the number of cells to be moved to various processors to balance the load. *Find_req* returns a vector of length equal to the number of processors. The vector is called *request.* It contains the fraction of the total number of cells that each processor needs to balance the load. We want $\mathrm{Fitness}_{\mathrm{load}}$ to be a maximum for distributions of cells to processors which match the *request* vector.

To derive the request vector, assume we are working on 4 processors and then generalize to P processors.

Assume we start with equal load, N, on 4 processors. Processor 0 splits M of its

cells. Processor 0 now has 4M new cells to calculate, but calculations are no longer performed for the M cells that split. The load on processor 0 is thus N+4M-M=N+3M.

We have:

Processor	Load
0:	$N+3M$
1:	N
2:	N
3:	N
ave=	$\left(N+\frac{3}{4}M\right)$

What percentage, W, of the N+3M cells must leave processor 0 to balance the load?

$W = N+3M-\left(N+\frac{3}{4}M\right) = 2\frac{1}{4}M$. For an arbitrary number of processors, P, the average is $\left(N+\frac{3}{P}M\right)$ so $W = N+3M-\left(N+\frac{3}{P}M\right) = \left(3-\frac{3}{P}\right)M$.

The load that leaves processor 0 is distributed evenly over the other processors. Recall that the request vector must sum to 1. The request vector is thus request(0)=1-W and request(N>0)=W/(N-1).

4.6. Expansion on sine qua non one, use domain specific knowledge.

The first portions of domain specific knowledge used are very general, applicable to almost any adaptive grid program and independent of the method used to assign cells to processors. In fact, for the second simulation discussed above, even though the cells were moved to random processors, sine qua non one, use domain specific knowledge,

was important to this run. Domain specific knowledge was used to reduce the communication time and to improve the load balance. This may be surprising considering the run time for this simulation was longer than the simulation for which no attempt was made to balance the load. This will be explained below.

4.6.1. Larger contiguous blocks implies less communication.

As discussed in Chapter 1, for most grid programs, cells need data from their neighbors in order to update values for the next time step. If neighbors are on different processors, communication needs to occur to update the values of the cells.

> **The first piece of domain specific knowledge used was that if the blocks of contiguous cells on the same processor are large, communication will be less than if the block of contiguous cells on the same processor are small. This knowledge was exploited by moving cells in clusters.**

What is a cluster? A cluster of cells is a rectangular region containing at least 16 contiguous cells. All the cells of a cluster that are moved by the GA are at the same level in the tree structure that describes the grid. (For a description of the data structures used to store the grid, see Appendix A. For a description of the subroutines referenced in this section also, see Appendix A.)

A cluster of 16 cells all share the same grandparent. This is shown graphically in the figure below. A tree structure with the 16 leaves sharing a common ancestry is shown.

A cluster can contain more than 16 cells if one or more of the cells has children. For example, the second cluster shown below has 24 cells. When a cluster is moved, the 16

cells at the same level and their descendents are moved, not the parents or grandparent of the 16 cells.

The area covered by a cluster is the area of the grandparent or, 16 times the area of the smaller cells.

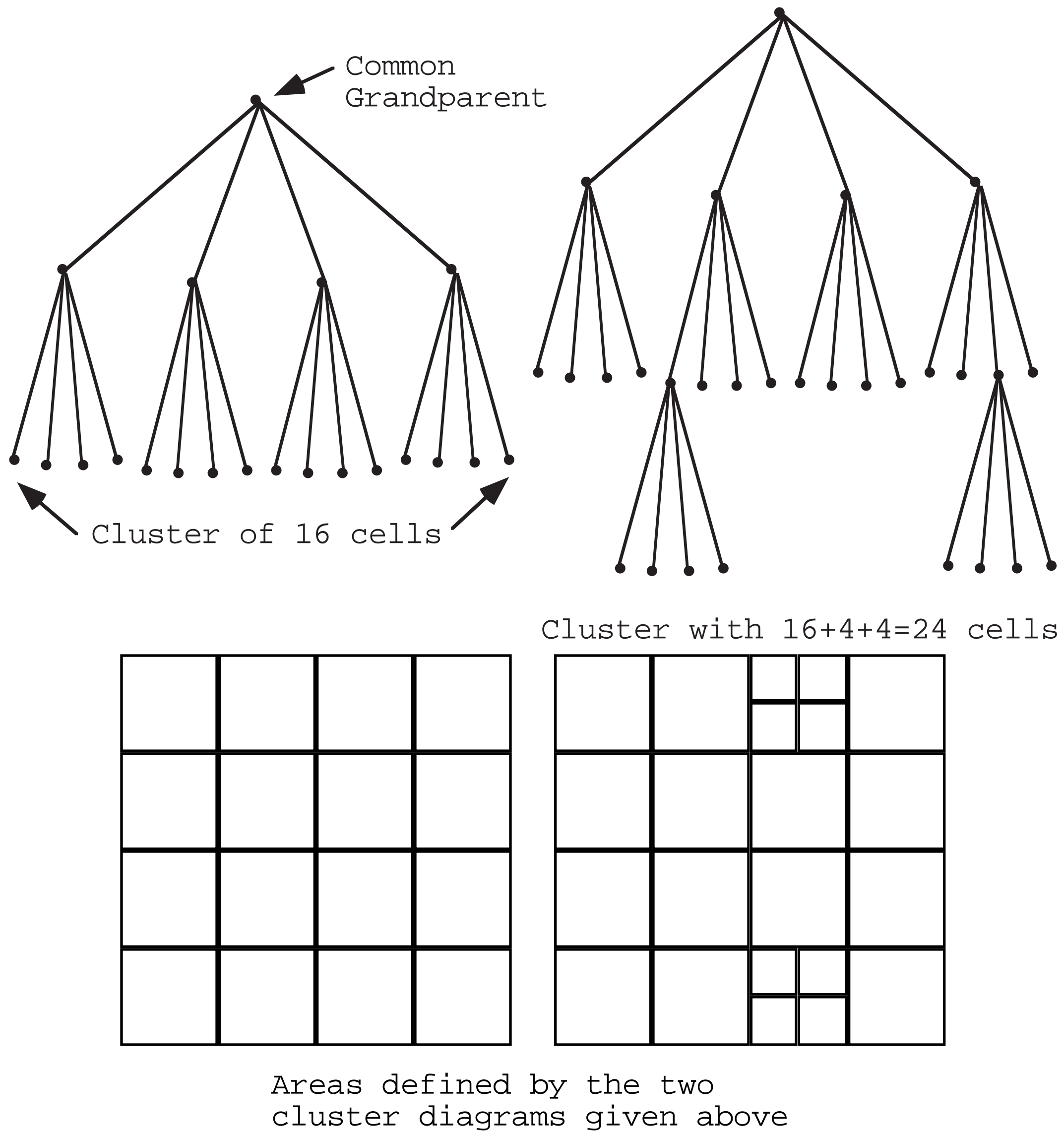


Figure 4.4. Cluster of cells moved by the genetic algorithm.

All of the clusters moved by the GA have grandparents at the same level in the tree

structure. All clusters represent the same sized area in the grid. The level in the tree structure at which the cells of a cluster are found is determined by a flag set in the program. Setting the parameter to different levels changes the level of finesse at which the GA can operate to balance the load. Setting the level closer to the root of the quad trees decreases the finesse. In this document finesse refers to the refinement and delicacy of performance, execution, or artisanship.

Why move cells in clusters? Moving in groups places a lower upper limit on the communication cost for a distribution of cells. When cells are moved in clusters, the only communication between processors will be for cells on the edge of the clusters. Even for the random distribution discussed above, the communication time was reduced by moving cells in clusters. (As will be discussed below, when the GA is used to assign clusters to processors, the size of the contiguous blocks of cells that are on the same processor tend to be larger than when the cells are distributed randomly. This enables a further lowering of the communication cost.)

Another advantage of moving cells in clusters is that the search space for the algorithm that is used to assign cells to processors is reduced. For most algorithms, this allows a faster run time. What does this mean, the search space is reduced? Consider 128 clusters of 16 cells, each cluster being assigned to one of four processors. There are 4^{128} or 2^{256} possible distributions of clusters to processors. If the cells are distributed one at a time there are $4^{(128*16)}$ or 2^{4096} possible distributions of cells to processors.

The GA uses as input a collection of clusters of cells and a description of the grid in the neighborhood of the clusters. This description includes the processor id for the cells surrounding the clusters. This information is used by the GA when calculating the cost of

communication for a particular mapping of cells to processors.

For the experiments described in this chapter and the next, the initial grid is defined with N levels in the quad trees. (N varies by experiment.) The level for cells in the clusters is set at N+1. Initially, there are no cells which are parts of clusters. The clusters are empty. Full clusters are generated as the simulation progresses. When the four cells below a grandparent for a cluster split, the cluster is full, that is, it has 16 cells.

Clusters are considered for movement by the GA only when they are full. For a full cluster the four leaves have split into at least 16. This represents a factor of at least 4 increase in the computation time for the area covered by a cluster. Before a cluster is full, its extra computational load is not considered to be significant.

When the GA completes its execution, it returns a vector that contains the assignment of clusters to processors. This list is passed back to the PLIFE framework. The framework moves the cells to other processors, finds neighbors as needed, and adjusts communications. If N is the number of clusters, then the number of cells moved is M >= 16N. The number exceeds 16 N when one or more of the 16 cells in a cluster have split.

For this example calcluation, the GA was called periodically to assign full clusters to other processors. That is, periodically, the genetic algorithm was given the responsibility to determine which of the split cells to move from processor zero.

4.6.2. Clusters of cells are generated at irregular rates.

How often should the GA be called? If the GA is called often then each time it is called it has few clusters to move. As discussed by Ozturan, deCougny, Shephard, and

Flaherty (1994) the run time improvement from moving a few cells may not outweigh the time required to move them. If the GA is given too many cells to move then the search space for the GA is larger and a good solution may not be found. Also, if the simulation is allowed to run for a long time between GA invocations the calculation will be out of balance for many cycles.

Based on the experience from a small set of parametric studies, the GA was called when there were 100 clusters to distribute to other processors. Thus, for these simulations, the GA was called at cycle numbers such that, each time it was called, it had about 100 clusters of cells, or 1600 cells to distribute. It was called at cycle numbers: 65, 141, 212, 279, 341, 399, 454, 505, 552, 597, 640, 680, 718, 754, 790, 824, 857, 890, 923, 956.

> **The second piece of domain specific knowledge used was that over the course of the calculation clusters of cells are generated at irregular rates. By measuring the rate of the generation of clusters, and using this measurement, the efficiency of the GA can be increased.**

Initially, when the GA was being tested it was run at regular intervals, every 25 calculation cycles. Running the GA at the cycle numbers given above produced better run times because the GA was not given excessive cells to distribute one time and only a few the next. Also, by giving the GA about the same amount of work each time it is called the same set of GA control parameters can be used from one invocation to the next. If a set of GA control parameters are found which work well for one invocation they will likely work well for the next invocation.

4.6.3. Future load can be estimated.

For this example calculation, since there is only a single level of adaptation and excess load is only generated on processor 0, we use a simple function for the request vector. Fortunately, the simple form of the function allows for load prediction. This can be used to increase speed up.

If a distribution of cells to processors is done according to the request vector given above, the loads will be balanced. Shortly thereafter, processor zero will again have excess load. Node 0's amount of excess load will continue to grow until the next redistribution. Also, node zero will always be the last to finish a calculation cycle.

Assume a redistribution of cells occurs at iteration number J, so the 4 nodes are balanced. Also assume a calculation cycle takes 1 second. Assume that node 0 is gaining load at a constant rate from cells splitting and at some later iteration, iteration K, node 0 has gained load enough load to cause the calculation rate to be 1.03 seconds/cycle. At a still later iteration number, L, node 0 has gained load and it is causing the calculation rate to be 1.12 seconds/cycle.

Now assume W, defined in section 4.5, is multiplied by a factor, $\Gamma>1$. When a redistribution is performed, processor 0 will have less load than the other processors. Until its load increases to that of the other processors, it will run faster.

Assume enough extra load is off-loaded to the other processors so that processor 0 runs 0.09 seconds faster per iteration, that is at a rate of 0.91 seconds/cycle. The other processors will run 0.03 seconds slower. They will determine the overall calculation rate with times of 1.03 seconds/cycle. The rate will remain constant at 1.03 seconds/cycle until enough additional load is produced on node 0 to bring its rate over 1.03. This

occurs at iteration L. See figure 4.5.

If we go back to the case where no additional load is off loaded from processor 0, at iteration L, the rate is 1.12 seconds/cycle.

Between iteration J and K, the calculation rate is faster if no additional load is removed from node 0. After the Kth cycle, the calculation rate is faster for the case when the additional load is moved from 0.

The total run time is the integral of the calculation rate. Sometime after iteration K, the run for which the additional load is moved from processor 0 will pass the base line case. The rate will continue to be faster. Until the load is redistributed, it will continue to gain an advantage over the run for which no additional load is moved.

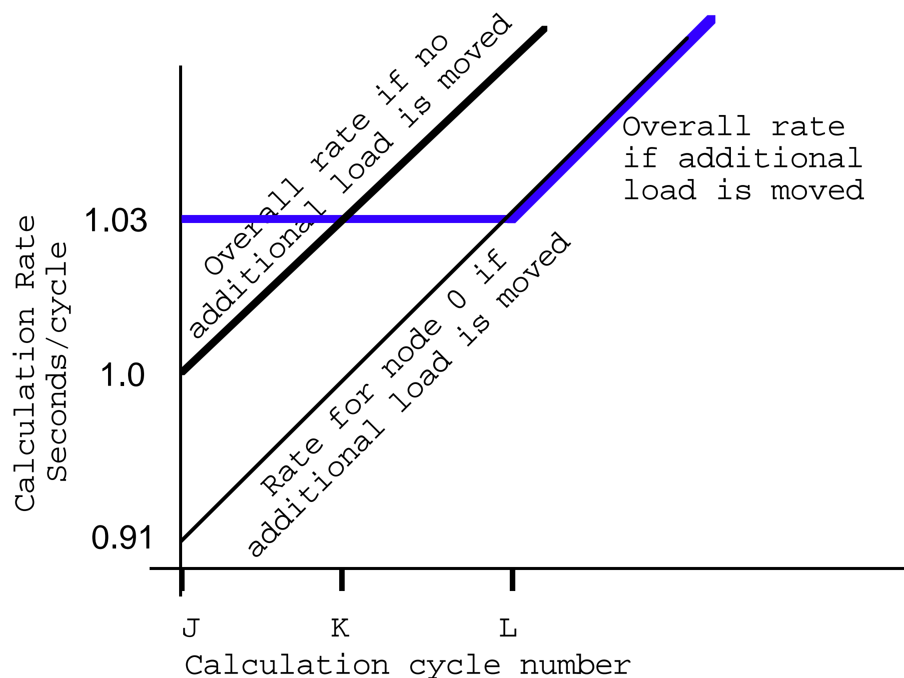


Figure 4.5. Illustrates the calculation rate dependency on the movement of extra load for node 0.

When the additional load is moved from processor 0, the overall speedup for a run is dependent on the rate at which new load is generated on 0, the frequency at which the load is redistributed, and the parameter Γ. For this particular set of simulations, it has been found that a value of Γ of about 1.3 will produce overall run times that are about 15 seconds faster.

> **Thus we have the third use of domain specific knowledge. The knowledge is that, at some time extra load is being generated on a particular processor and also for the near future, extra load will continue to be generated on that processor. This knowledge is exploited by the movement of extra load. The movement of extra load decreased the run time.**

Although the value of Γ=1.3 produced good run times, the balance parameter was higher than expected, about 9.5. What this signifies is that the calculation would have been balanced if processors 1-3 each gave processor 0 about 9.5 seconds of work.

Why such a bad balance? Γ=1.3 was a good value at the start of the calculation, a value even a little higher may have been better. At the end of the calculation Γ=1.3 was too high. This caused processor 0 to be under loaded. A better strategy would have been to have a variable Γ. The further study of strategies to determine the amount of load to be moved between processors is left as an area of future work. However, these experements show that improvements are possible when problem specific information is used. Further work needs to be done to determine methods to find dynamic values for Γ.

4.7. Cells split in the region of the shock front.

Using the domain specific knowledge discussed so far did not lead to very good results. Simulations were run using this knowledge and various implementations of the fitness function and GA control parameters. The improvements in the run time were small, a few seconds. Analysis of the results indicated two problems. The run times for the GA were long and the communication costs were high. In order for the GA to be effective these issues needed to be addressed. We will next discuss the use of domain specific knowledge that made a significant improvement in the communication time for the simulation and thus made a significant improvement in the run time. The communication time was reduced from 350 seconds to 230 seconds.

The domain specific knowledge deals with the characteristics of the propagation of shock waves. With the incorporation of this knowledge the way the fitness function counts communication cost was changed.

Recall that only cells which have split are considered for movement by the GA. The load balancing algorithm was designed to move these split cells to maintain a good balance.

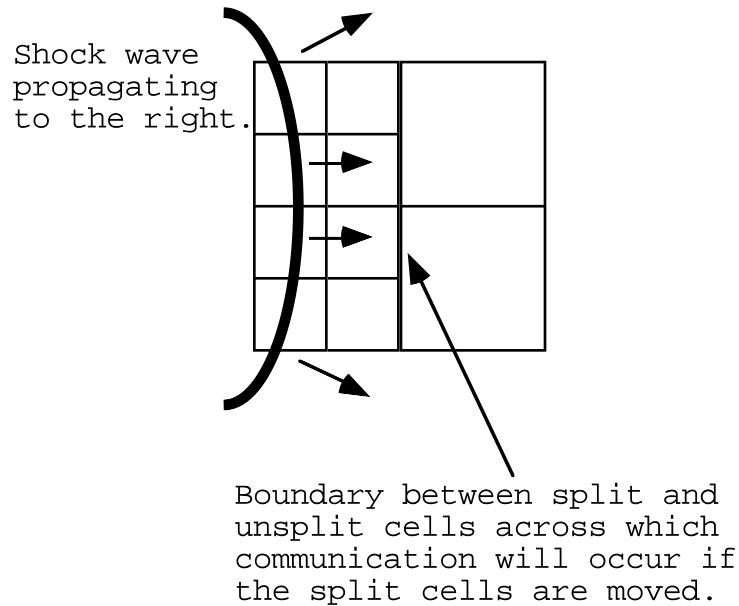


Figure 4.6. A shock wave propagating through the grid.

As a shock wave propagates the cells just in front of it tend to split. Assume a shock wave is moving left to right through the grid. Cells directly to the right of the wave will have split and can be moved by the load balancing algorithm. Cells a little further to the right will not have split and are not moved. Thus there is a boarder between cells which can not be moved by the GA and those that can be moved.

What happens on this boarder is critical. There is a conflict. To balance the load, the GA wants to move the cells which have split. At the same time it does not want to move the cells because this would cause communication. If the GA decides not to move the split cells on the boarder, then it will not want to move the cells behind them either, and so on.

Assume the GA decides not to move the cells. The next time the GA is called, the

shock wave will have propagated further to the right. The previously unsplit cells will have split and will be ready to be moved. To their left there are cells which are still on the original processor. The GA will not want to move these new cells because doing so will cause communication with the cells to the left. The bottom line is that the GA will not want to move cells.

Consider what happens if, in the fitness function, we do not count the communication between the split cells and the unsplit cells. First, the GA is "free" to place the cells on the boarder on any processor without penalty to the fitness function. In doing so it can do a better job of balancing the load.

But won't this cause an increase in communication and slow down the run time? Yes, but only for a short while. This is where the domain specific knowledge comes in. We know the shock wave will continue to propagate and we know the cells in front of it will soon split. The next time the GA is called these cells will be moved. Most likely they will be moved to the processor which holds the cells behind them. We only pay a communication penalty for these cells until the next time the GA is called.

> **Thus we have the fourth use of domain specific knowledge that, as a shock wave propagates, the cells just in front of it tend to split. We exploit this knowledge by not counting the communication between the split cells and the unsplit cells in the fitness function. The use of the domain specific knowledge greatly increased the ability of the genetic algorithm to improve performance by decreasing the communication cost of the distributions of cells to processors.**

This modification to the fitness function produced interesting results for the distribution of cells. Distributions tend to form in clumps, with all cells in a clump on a single processor. These clumps grow after each invocation of the GA. They tend to grow along the direction of the propagation of the shock wave. This tends to reduce communication and produce better run times. It is interesting to note that for a spherically expanding shock wave, the clumps tend to form wedges. The differences in the distribution of the cells when communication between split and unsplit cells is counted and when it is not counted can be seen by comparing figures 4.7 and 4.8. Again, the improvement in communication time between these two runs was 120 seconds with no increase in the run time for the GA.

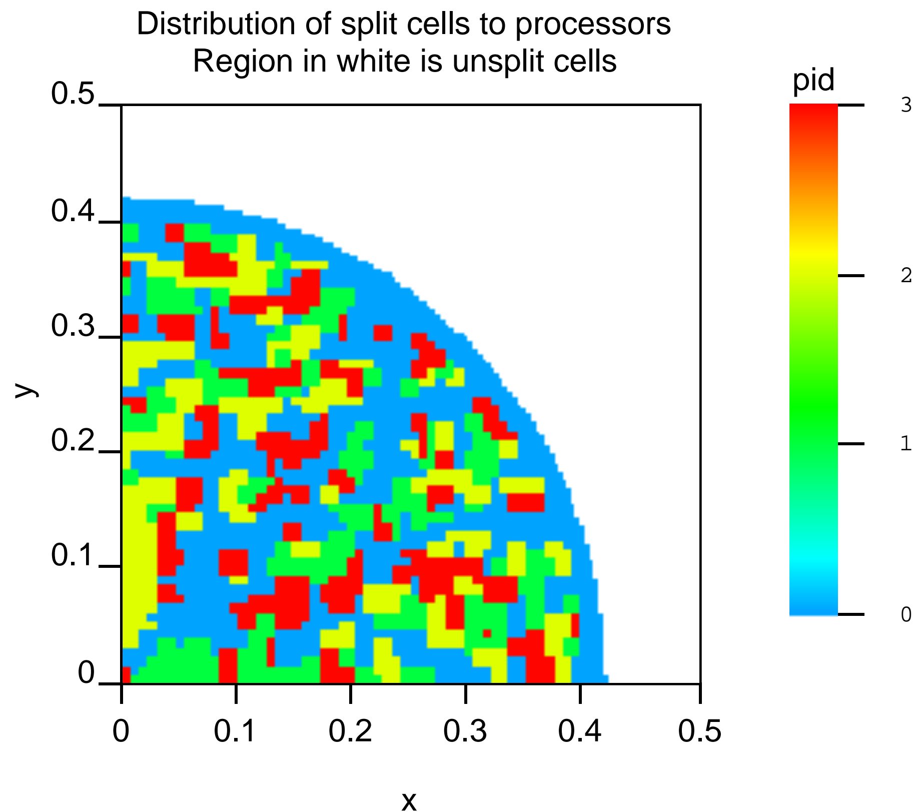


Figure 4.7. Distribution of cells to processors at the end of the simulation with communication between split and unsplit cells counted yielding a communication time of 348 seconds.

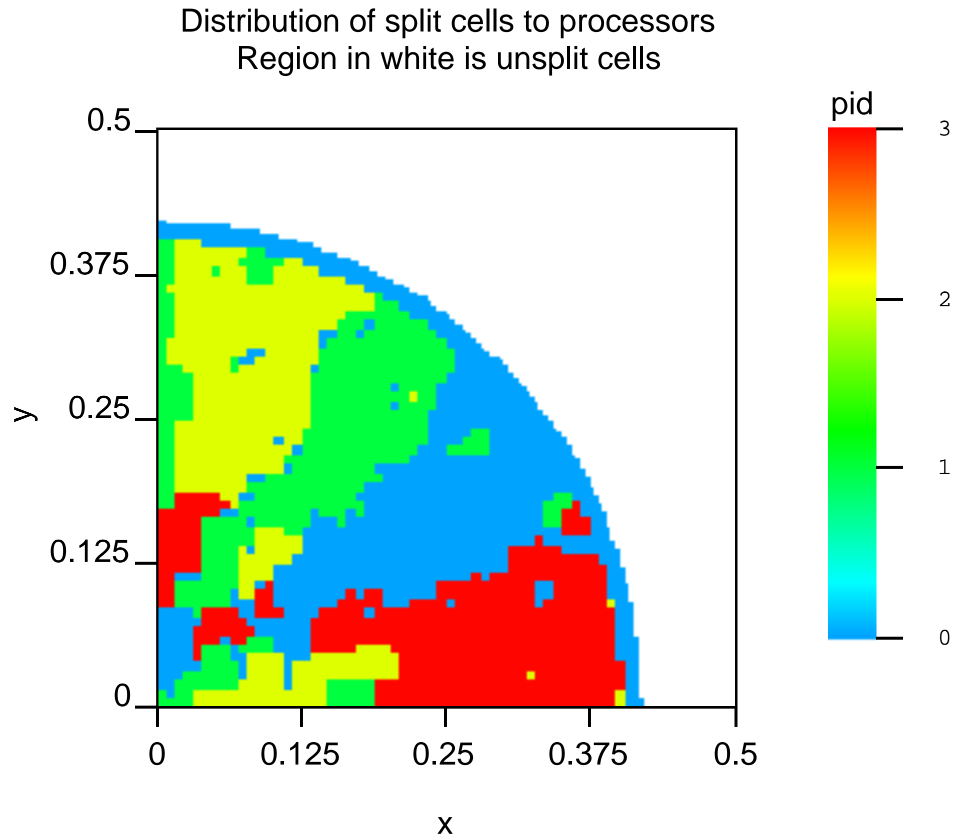


Figure 4.8 Distribution of cells to processors near the ending time of the simulation with communication between split and unsplit cells not counted yielding a communication time of 228 seconds.

As discussed above, there are two sine qua nons. Domain specific knowledge must be used and improvements must be made to the algorithm of the GA. We now discuss the second sine qua non: improvements to the GA.

4.8. Sine qua non two, improvement in the GA.

4.8.1. Fast mutation.

The standard method of mutation in a GA runs in time proportional to the population size times the size of the vector describing the members of the population. This method is described by Fogel (1996). A mutation methodology was developed for use in this research effort which runs in time proportional to the population size times the number of

genes, times the mutation probability P. For small P this is method is much faster. Typical values for P are small, on the order of 0.01. This was the value used for this research. For a description of standard and fast mutation procedure see Appendix C. The fast mutation was the first modification to the basic structure of the genetic algorithm.

4.8.2. Discussion of the Mansour-Fox algorithm for static grids.

Mansour and Fox suggested a collection of modifications to the basic genetic algorithm to improve the quality of solutions returned. Their modifications to the standard GA were incorporated into the load distribution routines. Their algorithm will be discussed next. Then the improvements made to their algorithm as a result of this research effort will be discussed.

Mansour and Fox (1991) used a genetic algorithm to perform load balancing and communication reduction for a static computation problem. This GA was used to allocate grid elements to processors for a finite element calculation on a hypercube. The GA seeks an element-to-processor mapping which minimizes communication and maximizes load balance. Tests were performed on two irregular grids. In the first case, a distribution was derived which allowed a run time efficiency of 93% on 8 nodes of a hypercube. In the second example an efficiency of 97% was obtained running on 16 hypercube nodes. Their efficiency was a comparison of the measured run time to an estimate of the best possible run time.

The Mansour-Fox algorithm starts off like a standard GA. Initially though, they do give a higher weight to minimizing communication in the fitness function. When 50% to 75% of the generations are completed, changes are made. The weighting becomes a function of iteration number, with the weighting for the final generation being 50-50.

Also, after 50% to 75% of the generations are completed, a greedy hill climbing algorithm is called for every member of the population. After this routine is first called, it is called every generation. This greedy algorithm first removes isolated cells. That is, if a cell is surrounded by four cells which are on the same processor and the center cell is assigned to a different processor then the center cell is assigned to the same processor as the surrounding cells. Also a check is performed for every cell which has neighbors on different processors to see if the cell should be assigned to one of these processors. This hill climbing algorithm is called in addition to the standard GA operations of finding fitness values and reproduction.

Finally, they suppress normal mutation after the 50% to 75% of the generations are completed. They replace the normal mutation, where an assignment of any cells to any processors can take place, with a more selective mutation. Mutations are restricted to cells at the boundaries of groups of cells which are on the same processor.

Mansour and Fox found that the quality of the distributions returned by this algorithm was better than a standard GA. They had a better match to the fitness function. They found that their algorithm was less likely to converge prematurely. Also, the hybrid algorithm returned better solutions than a straight hill climbing routine. Although the quality of the solutions returned by their algorithm was high, the run time was long. For a grid with 301 cells the run time on a SPARC workstation was 1416 seconds.

When the Mansour-Fox algorithm was applied to this adaptive grid problem the algorithm returned good distribution of cells to processors, with good balance and low communication times. These times are shown in table 4.2. As discussed in detail in the next section even with the good balance and low communication the overall calculation

speed ups were not very good. In fact one of the simulations ran slower than the base line case.

Often	Size	Gen	Wall Time	Speed up	GA	Return	Balance	Comm. time
1	200	200	1480	-2.0	191	-0.2	9.8	246
1	200	150	1429	1.5	145	0.2	9.4	240
1	200	100	1376	5.4	99	0.8	9.8	236
1	150	200	1429	1.6	147	0.2	9.9	243
1	150	150	1390	4.4	109	0.6	10.6	231
1	150	100	1337	8.5	74	1.5	10.0	225
1	100	200	1390	4.4	101	0.6	9.5	239
1	100	150	1369	6.0	76	1.1	10.1	247
1	100	100	1316	10.3	50	2.7	10.0	223

Table 4.2. Application of the Mansour-Fox algorithm.

4.9. Expansion on sine qua non 2, improve the GA algorithm, and enhancements to the Mansour-Fox algorithm.

Why are the speed up values so poor? The GA ran too long. For an adaptive grid program, if the GA runs too long then the quality of the distribution returned is irrelevant. As discussed, the chief goal is to minimize the overall program run time. We cannot afford a high overhead for the load balancing routine.

There are two problems with the Mansour-Fox algorithm. First, the greedy hill climbing routine is costly. Secondly, it can not be terminated early because of the dynamic fitness function.

Addressing these problems in this research effort led to improved algorithms. First, the GA is run as a parallel algorithm. Secondly, the greedy algorithm is called every N generations instead of every generation. N is an input parameter. Finally we have the

option of using a static fitness function.

The addition of the ability to call the greedy algorithm every N generations and to do early termination enables a trade off of distribution quality and GA run time. It was hoped that calling the greedy algorithm every N generations and allowing early termination would decrease the run time of the GA, but not greatly decrease the quality of the distribution returned by the algorithm.

4.10. Initial test of the improved Mansour-Fox algorithm.

The hope was borne out by experiment. A series of test cases or parameter studies were run to see if calling the greedy algorithm less often would decrease the run time of the GA, but not greatly decrease the quality of the distribution returned by the algorithm. For the test cases performed for this research effort, it was found that by calling the greedy algorithm every N generations, where N was in the range 2 to 5, the algorithm did run faster and the quality of the solution returned was still good. For the test cases, as with the runs with the regular Mansour-Fox algorithm, the size of the population for the genetic algorithm was in the set {200, 150, 100}. The same values were used for the number of generations the GA was run. The frequency of calling the hill climbing routine varied from 1 to 5.

Next, the results of the parameter studies are presented. We compare the runs of the simulation when there was no effort to balance the load, to runs of the simulation for which the genetic algorithm was called with various parameter settings.

The next 3 tables show the data from the parameter studies. The simulation ran slowest when the population, number of generations and frequency of calling the hill

climbing routine were the largest. When the GA ran faster, the run times are below the base line case.

The best run time was 1315 seconds for a speed up of 10%. For this run the population size and number of generations were 100 and the frequency was 1. The next best run was 1318 seconds with a frequency of 2 and the same size and number of generations.

Often	Size	Gen	Wall Time	Speed up	GA	Return	Balance	Comm. time
1	200	200	1480	-2.0	191	-0.2	9.8	246
1	200	150	1429	1.5	145	0.2	9.4	240
1	200	100	1376	5.4	99	0.8	9.8	236
2	200	200	1434	1.2	133	0.1	10.3	244
2	200	150	1381	5.1	97	0.7	9.9	241
2	200	100	1356	7.0	66	1.4	9.9	241
3	200	200	1434	1.2	116	0.1	10.0	263
3	200	150	1374	5.6	83	0.9	9.8	247
3	200	100	1340	8.3	58	1.9	9.1	240
4	200	200	1386	4.7	103	0.6	9.9	242
4	200	150	1380	5.1	77	0.9	9.9	259
4	200	100	1369	6.0	53	1.6	9.5	270
5	200	200	1405	3.3	98	0.5	9.7	265
5	200	150	1369	6.0	70	1.2	9.9	261
5	200	100	1334	8.8	49	2.4	9.9	244

Table 4.3. Summary of simulation runs using the hybrid genetic algorithm with a fixed population size, 200.

Often	Size	Gen	Wall time	Speed up	GA	Return	Balance	Comm. time
1	150	200	1429	1.6	147	0.2	9.9	243
1	150	150	1390	4.4	109	0.6	10.6	231
1	150	100	1337	8.5	74	1.5	10.0	225
2	150	200	1371	5.8	101	0.8	9.9	227
2	150	150	1351	7.4	75	1.3	9.9	231
2	150	100	1318	10.1	51	2.6	10.0	226
3	150	200	1355	7.1	84	1.1	9.9	228
3	150	150	1370	5.9	64	1.3	10.0	257
3	150	100	1346	7.8	46	2.3	9.9	258
4	150	200	1376	5.4	78	1.0	9.8	257
4	150	150	1371	5.8	60	1.3	9.9	258
4	150	100	1353	7.2	40	2.4	9.7	272
5	150	200	1358	6.8	74	1.3	9.4	241
5	150	150	1348	7.7	54	1.9	10.1	246
5	150	100	1340	8.3	38	2.9	9.8	254

Table 4.4. Summary of simulation runs using the hybrid genetic algorithm with a fixed population size, 150.

Often	Size	Gen	Wall time	Speed up	GA	Return	Balance	Comm. time
1	100	200	1390	4.4	101	0.6	9.5	239
1	100	150	1369	6.0	76	1.1	10.1	247
1	100	100	1316	10.3	50	2.7	10.0	223
2	100	200	1369	6.0	70	1.2	9.6	244
2	100	150	1340	8.3	52	2.1	9.7	241
2	100	100	1334	8.8	38	3.1	10.2	244
3	100	200	1366	6.2	60	1.4	9.8	259
3	100	150	1340	8.3	44	2.5	9.8	251
3	100	100	1347	7.7	32	3.3	9.7	270
4	100	200	1354	7.1	54	1.8	9.2	255
4	100	150	1345	7.9	40	2.6	10.2	255
4	100	100	1339	8.4	28	3.9	10.2	261
5	100	200	1355	7.1	51	1.9	10.1	258
5	100	150	1352	7.4	39	2.6	10.4	266
5	100	100	1332	8.9	27	4.5	10.0	251

Table 4.5. Summary of simulation runs using the hybrid genetic algorithm with a fixed population size, 100.

	200 Generations	150 Geneartions	100 Generations
Every 1	1433.1	1396.0	1342.9
Every 2	1391.6	1357.4	1336.1
Every 3	1385.2	1361.3	1344.2
Every 4	1372.1	1365.5	1353.7
Every 5	1372.8	1356.0	1335.5

Table 4.6. Average run times for simulation runs with a constant number of generations and frequency of calling the hill climbing routine.

Are there trends that indicate a particular frequency of calling the hill climbing routine is more effective? To see the trends, we average the run times for a fixed frequency and fixed number of generations. This averaging yields Table 4.7. One general trend is that as the frequency of calling the hill climbing routine decreases, the overall run time for the simulation also decreases.

The modifications to the Mansour-Fox algorithm are useful in decreasing the simulation run time. That is, there is value in calling the hill climbing routine less often than every generation.

The value of calling the hill climbing routine less often will become even more apparent when the second set of simulation runs are discussed in the next chapter.

A second trend can be seen in the chart below. As the number of generations decreases, the overall run time for the simulation also decreases. Note that, as the number of generations decrease and the frequency decreases, the run time for the GA also decreases.

	200 Generations	150 Geneartions	100 Generations
Every 1	146.4	109.9	74.6
Every 2	101.4	74.8	51.6
Every 3	86.6	63.9	45.4
Every 4	77.9	59.0	40.4
Every 5	74.1	54.2	37.9

Table 4.7. Average genetic algorithm run times for simulation runs with a given number of generations and frequency of calling the hill climbing routine.

4.11. Fast GA yields fast simulation time and reinforcement of sine qua non one.

We can combine all of these observations and say as a general trend. As the run time for the GA decreases, the run time for the simulation also decreases. This trend can be seen if we do a plot of the run time for the GA versus the overall simulation run time. See figure 4.9.

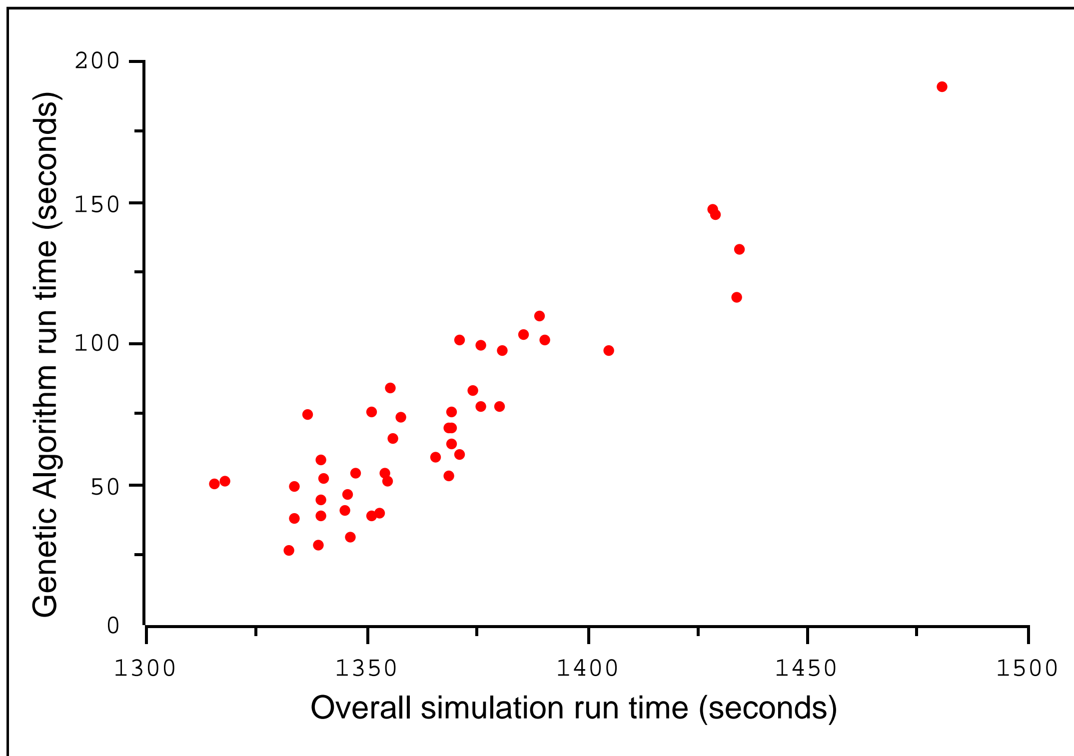


Figure 4.9. Plot of overall simulation run time and GA run time showing the trend that shorter GA run times leads to shorter overall simulation run times.

With this plot the trend is obvious. This plot reinforces sine qua non one, that improvements must be made to the GA to decrease its run time but still have it return a high quality solution. How much further can the run time for the GA be decreased until the overall run time for the simulation starts to increase? A short series of test runs were performed to examine this question. The simulation was run using population sizes and numbers of generations of 80 and 60.

Often	Size	Gen	Wall Time	Speed up	GA	Return	Balance	Comm. time
1	80	80	1305	11.2	33	4.4	10.0	224
2	80	80	1297	11.8	24	6.4	9.7	232
3	80	80	1340	8.3	21	5.3	10.0	266
4	80	80	1340	8.3	19	5.9	9.9	274
1	60	60	1301	11.5	20	7.3	9.6	240
2	60	60	1309	10.8	14	10.1	9.7	248
3	60	60	1323	9.7	13	10.2	10.2	254
4	60	60	1379	5.2	11	6.6	9.5	324

Table 4.8. Summary of additional simulation runs with shorter genetic algorithm run times.

A minimum run time of 1297 seconds occurs when the population size is 80, the frequency is 2, and the GA time is 24 seconds. For a shorter GA run time, the overall run time starts to increase. The major contributor to the increased run time is communication. With the shorter run time for the GA, the resulting mapping of cells to processors has a higher communication time.

When the population size is 60, the minimum run time is 1301 seconds with a GA time of 20 seconds. For a shorter GA run time, the overall run time increases rapidly.

Again, the major contributor to the increased run time is communication.

Why does communication cost go up? The initial population given to the GA, on average, represents a balanced distribution of cells to processors. That is, if an average of the number of cells going to each processor was taken over all of the population, the values obtained for the numbers of cells going to each processor would represent a balanced load. The goal of the GA is to take these initially balanced distributions and create a distribution of cells to processors which also has a low communication cost. With a very short run time the GA can not do this. In the limit of 0 run time, the distribution of cells to processors would be balanced but random. This case was discussed above and it had a run time of 1631 seconds.

There is an amazing number in this chart which is worth pointing out. With a population size of 60 and a frequency of 2, we have a run time of 1309 seconds and a GA run time of 14 seconds. The amazing number is the return on investment, 10.1. The GA improved the run time over 10 times the time it took to run.

4.12. Expansion on sine qua non one & two, tuning of the GA using the improved Mansour-Fox algorithm.

Clearly, there are some sets of GA control parameters that work better than others. Finding a set that works well for a particular problem can be viewed as tuning the GA. Tuning the GA for a particular problem decreases the simulation run time for that problem. In tuning for a particular simulation, a combination of domain specific knowledge and the improvements of the GA algorithm are being used. Allowing a variable frequency of calling the hill climbing routine was an improvement in the GA and adjusting the frequency is tuning. The use of domain specific knowledge is more subtle and indirect. Different

simulations will have different characteristics as to the degree of imbalance as a function calculation cycle. Also, the improvements that are enabled by allowing the GA to run for some time are a function of the particular simulation.

In the next chapter it will be shown that

> **tuning the GA for a particular problem can aid in finding GA control parameters to increase the performance for a related problem. This reflects the use of domain specific knowledge and the improved genetic algorithm.**

What is a related problem? A related problem is one with different but similar initial conditions. A problem with the same initial conditions but run on a different machine is also considered a related problem.

4.13. What was learned from these runs?

4.13.1. Summary of sine qua non 1 and 2.

The running of the simulations discussed in this chapter produced knowledge about what must be done to enable a GA to reduce the run time of an adaptive grid program. Two essential principles or sine qua nons were obeyed to enable the speed up. These sine qua nons were that (1) domain specific knowledge must be used and (2) improvements of the basic algorithm of the GA must be implemented.

Several examples of the use of domain specific knowledge were given. The knowledge that the blocks of contiguous cells on the same processor are large yields reduced communication, is exploited by moving cells in clusters. The knowledge that over the course of the calculation clusters of cells are generated at irregular rates is exploited by

moving cells at irregular times. The third piece of domain specific knowledge that, if at a particular time extra load is being generated on a particular processor then for the near future extra load will continue to be generated on that processor. This is exploited by the movement of extra load. We have a fourth use of domain specific knowledge. As a shock wave propagates the cells just in front of it tend to split. We exploit this knowledge by not counting the communication between the split cells and the unsplit cells in the fitness function. This use of the domain specific knowledge greatly increased the ability of the genetic algorithm to improve performance by decreasing the communication cost of the distributions of cells to processors.

Several techniques were used to improve the algorithm of the GA. The first discussed was the use of a fast mutation algorithm. The Mansour-Fox algorithm was applied to this problem but the results were not particularly good. Next, improvements were made to the Mansour-Fox algorithm to decrease its run time yet still enable a high quality solution to be returned.

Finally, this chapter discussed tuning of the GA. Tuning of the GA is done by adjusting the size of the populations, the number of generations and the frequency of calling the hill climbing routine. Giving the GA the ability to be tuned by adjusting the frequency of calling the hill climbing routine is application of sine qua non two. Performing the tuning of the GA is an application of sine qua non one, applying domain specific knowledge.

4.13.2. GA was shown to be effective.

When sine qua non one and two were applied, the GA was shown to be effective. For this particular example calculation, improvements in the run time of 12% were seen

when domain specific knowledge was used along with the improvements to the GA.

The next chapter discusses a simulation for which the improvements were much higher. Also, another type of tuning is shown to be effective, tuning the fitness function by adjusting the relative importance of communication and load balance. The next chapter also shows that tuning for one simulation enables the GA to run different simulations to run effectively by using the same or similar GA control parameters.

Chapter 5. Application of heuristics.

The heuristics and knowledge developed running the four node calculations described in Chapter 4 were applied to 16 node calculations on the ARC SP1 and MHPCC SP2.

This chapter describes the results of those simulations and the additional insight that was gained from these runs. This chapter starts with a description of the basic problem that was run on the SP1 and SP2. Section 5.2 describes how what was learned from the four node calculations was applied to the new simulations. As in the preceding chapter, the results of using no load balancing and random load balancing are presented. These results are presented to have a base line to which to compare the results of the simulation when the GA was used. The next section presents the initial GA results. This section also describes tuning the fitness function to a particular architecture, the SP1. A statistical verification is given that, the GA was effective in reducing the run time for the example calculation by 120 seconds. Section 5.5 describes a set of runs of the simulation using various GA control parameters. Again, this is tuning the GA for a particular problem. Using the tuning information from this problem as a gauge for GA control parameters, the same simulation was run on the MHPCC SP2. As discussed is section 5.7, the improvement in run time for these runs was up to 75%. This SP2 simulation shows that what was learned from one simulation could be applied to a similar but different architecture. Next, again using the SP1 tuning information, a different simulation was run on the SP1. This run, discussed in section 5.8, shows that tuning information from one run can be used effectively to reduce the run time for a different but similar simulation.

5.1. Problem description, 16 node SP1 and SP2 runs.

For this second test case, the calculation was designed to be much more irregular, without symmetry. Seven regions of higher pressure and density were laid down on top

of a background in pseudo random locations. The background conditions were the same as the 4 node test case, with an initial pressure and density of 1.0. Each region of higher pressure and density was circular but the values of the variables within the regions were not uniform or circularly symmetric. See figure 5.2 for the location of the regions and figure 5.3 for a blow-up of one of the regions to show the irregularity in pressure. Similar irregularities are present for density and pressure for the other blobs.

The values of the variables within the regions were made irregular to simulate irregularities which can be produced in some physical systems, such as the variability in the amount of ink a printer deposits on paper. Also these irregularities provide a different test for the load balancing routines, because they will cause irregular concentrations of heavy load. In the previous test case the concentrations of heavy load were symmetric about the origin.

The calculation grid simulated a region of space from (x,y)=(0,0) to (x,y)=(2,2). Initially, the grid was uniform with 256x256 cells. Four levels of adaptation were allowed. If all of the cells had split four times, the grid would have become 4096x4096. The calculation was run for 750 cycles with a final simulation time of 0.9489E-02 seconds.

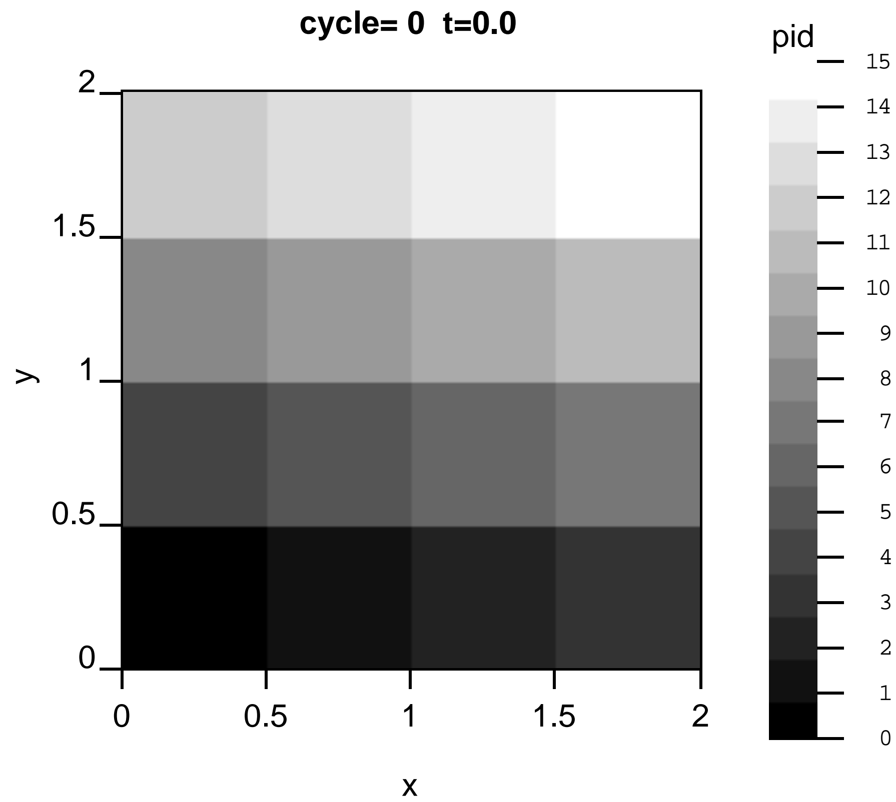


Figure 5.1. Initial distribution of cells to processors.

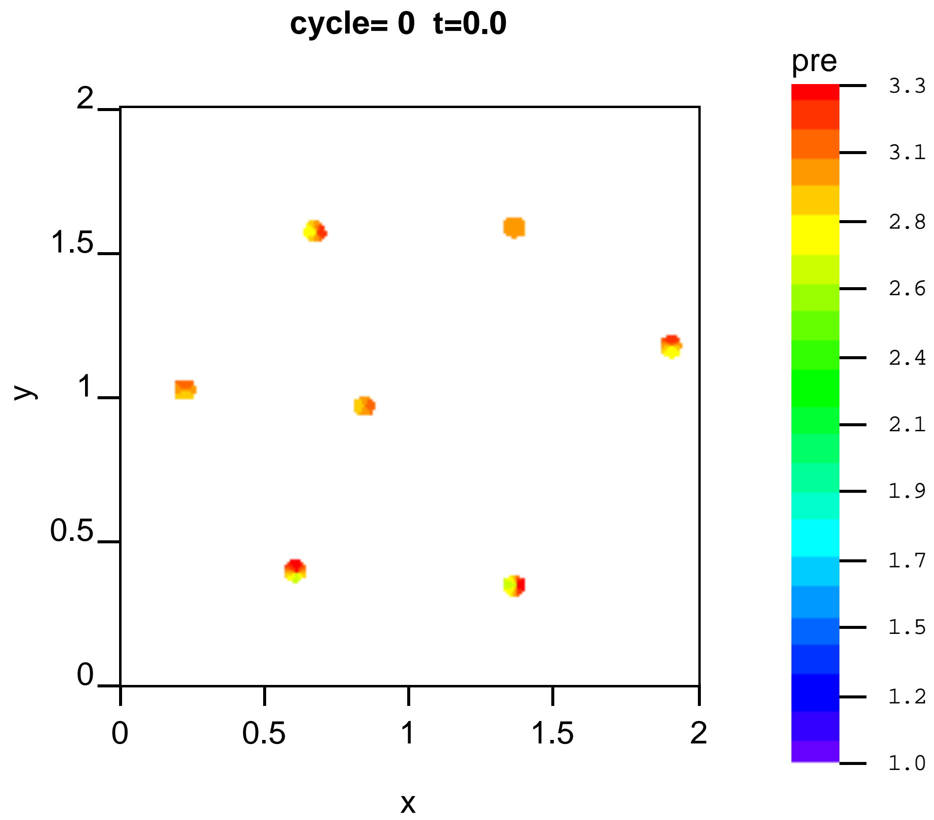


Figure 5.2. Initial location for regions of higher pressure.

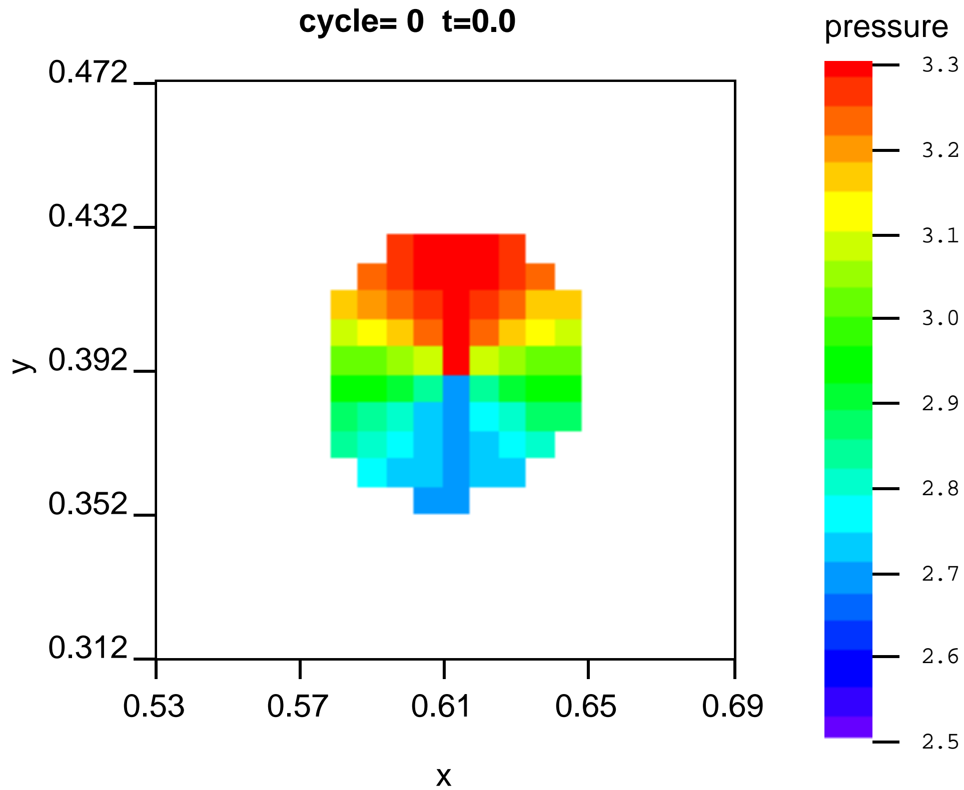


Figure 5.3. Blow up of one of the regions of higher pressure.

As the simulation progresses, the mass and energy flows outward from the blobs shown in figure 5.2. The first time this simulation was run, it was discovered that much of the splitting of cells and thus the need for redistribution occurred early in the simulation, and all over the grid at about the same time. The GA was called a few times to map cells to processors and the calculation continued. Unfortunately, after these first few distributions of cells, very few additional cells were moved. As far as the load balancing algorithm was concerned, the problem had become static. We are not interested in static, but dynamic gridding.

To make the problem more interesting, it was decided to deposit the blobs over time. This was like an inkjet printer depositing blobs of ink on paper as a function of time. This gave the GA work to do throughout the run instead of just at the beginning.

5.2. Applying what was learned from the 4 node calculation to 16 node runs.

The efficiency of the GA to maintain good load balance and low communication was improved by using the information gained from the 4 node calculations described in the previous chapter. In particular, the two sine qua nons of using domain specific knowledge and using the improved GA were applied.

As previously done, to reduce the long term communication cost, the GA ignored communication between split and unsplit cells. This again tended to cause the cells to be distributed in such a manner as to form large clusters assigned to the same processor. This was true even though the initial distribution of energy in the regions of high energy was not uniform. An example of this clustering can be seen in figure 5.8.

Also, the cells were again distributed in clusters. There was a significant difference in this calculation. Because four levels of adaptation were allowed, clusters usually contained many more that 16 cells. With more cells in the cluster the reduction in the search space for the GA was more significant than with 16 in the cluster. As before, by distributing in clusters the communication cost was also reduced.

For this calculation the GA was again called at irregular intervals. When cells were moved between processors by the GA, they were moved at cycle numbers:21, 141, 162, 262, 311, 396, 441, 484, 526, 573, 626, 669, 715. These cycle numbers were again chosen so that each time the GA was called, it was expected to distribute about the same number of clusters of cells, about 100. Although these cycle numbers were stored in an array at the beginning of the simulation by counting the numbers of clusters of cells the cycle numbers at which the GA was called could have been determined dynamically by

the program.

The improvements in the GA that were discussed in the previous chapter were used for this example calculation. In particular, fast mutation, the Mansour-Fox algorithm and the improvements to the Mansour-Fox algorithm were used.

5.3. Results with no load balancing and random balancing on the SP1.

To have a base line to which to compare results, the simulation was run with the GA turned off. There was no attempt to balance the load, so that no cells were moved between processors.

As a second comparison for the GA results, PLIFE was run with the GA again turned off, but with a crude load balancing algorithm turned on. For this case the cells were moved to the various processors pseudo randomly. The probabilistic function that determined the distribution was designed to, on average, produce distributions which matched the *request* vector. Recall that the *request* vector contains an estimate of the number of cells each processor should receive to balance the load. See chapter 4 for additional information on the *request* vector.

The simulation was also run using the Mansour-Fox hybrid GA but with the fitness function adjusted so that the communication cost was ignored. The goal of the GA was thus to find a distribution of cells which balanced the load, independent of the cost of communication. For these runs the GA was run for 100, 150, and 200 generations with population sizes of 240, 320, and 480. The hill climbing and parameter adjustment routines were called starting after 75% of the generations had completed, and every generation thereafter.

Table 5.1 summarizes the base line runs, the runs against which the further GA runs, the runs after appling sine qua non one, will be compared.

Often	Size	Gen	Wall time	Speed up	GA	Return	Balance	Comm. Time
0	0	0	1115	0	0	0	31.6	41
0	0	0	1240	-10.1	0	0	2.2	638
1	320	100	1207	-7.7	60	-1.5	2.2	555

Table 5.1. Summary of three base line runs; with (1) no attempt to balance the load, (2) using a crude load balancing scheme, (3) a hybrid GA to balance the load.

In the first run of table 5.1, no attempt was made to balance the load. This run was characterized by a large load imbalance between the various processors. Processor 2 had a calculation time of 697 seconds and processor 10 had a calculation time of 104 seconds. These values, along with the calculation times for the other processors, yielded a balance of 31.6. The cost of communication was low, 41.4 seconds. The overall wall clock run time was 1115 seconds.

The second run was the one where a probabilistic function was used to assign clusters to processors. The function did a good job of balancing the load. It had a balance value of 2.22. This run was characterized by a large amount of communication, about 600 seconds. This large amount of communication caused a long run time of 1240 seconds. The distribution of cells to processors for a portion of the grid at the end of the run is shown in figure 5.4. Figure 5.5 shows the finesse of the grid at the end of the simulation for part of the grid. Although the distribution of cells to processors changes for each simulation, the finesse of the grid does not change because it is determined by the numerics of the calculation.

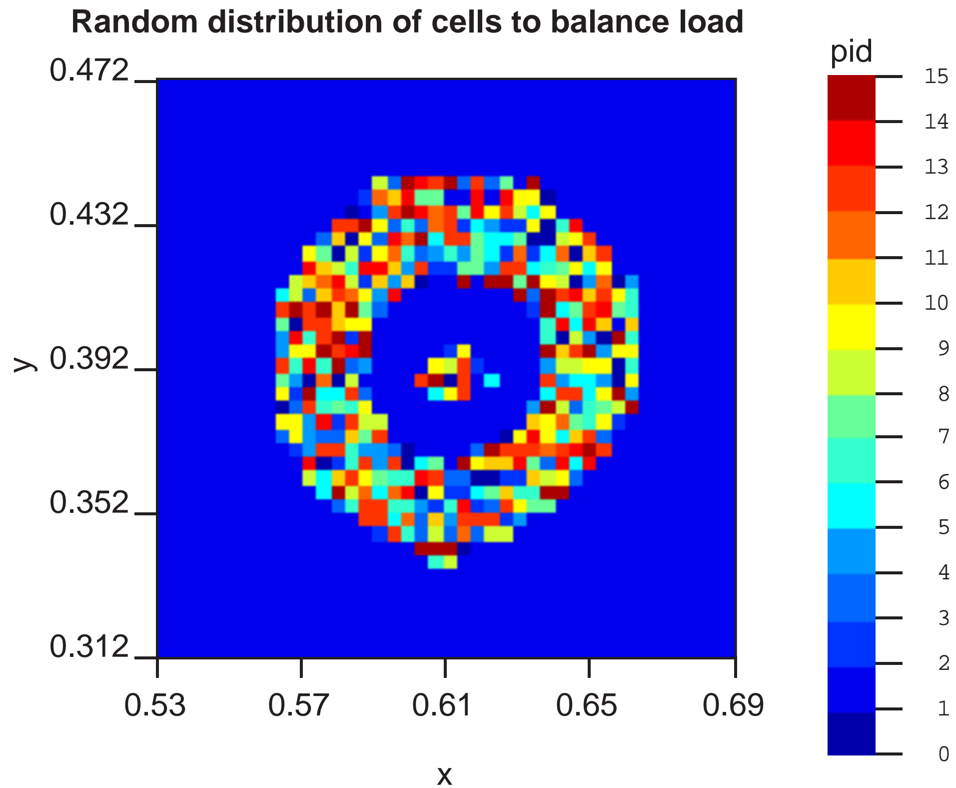


Figure 5.4. Distribution of cells to processors for a portion of the grid when the random function is used to balance the load, independent of communication cost.

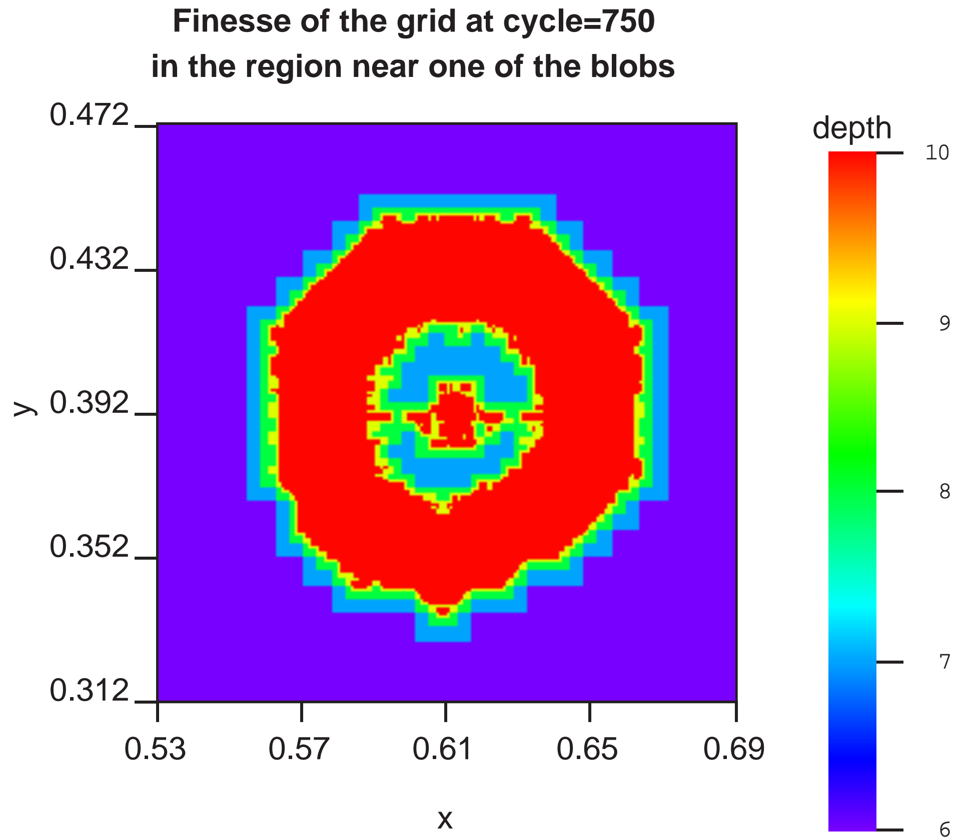


Figure 5.5. Finesse of the grid at the end of the calculation.

For the third run the GA was used to assign clusters to processors. For this run, the fitness function was adjusted so that it returned distributions of clusters to processors that did a good job at balancing the load, independent of communication cost. Even though the GA did a good job of balancing the load, the run time was longer than making no effort to balance load. The run time was 1207 seconds, or about 100 seconds longer than the first base line case. Note that we have a negative return on investment for running the GA, -1.5, and a negative speed up. Even if the overhead of running the GA, about 60 seconds, is removed, this run is still slower than making no effort to speed up the computation.

5.4. Initial results and tuning for a particular architecture.

With this last run there was good load balance but the communication cost was high. If there is a hope of getting improved performance over the base line case, some effort must be made to lower communication while trying to get a good load balance. It was conjectured that adjusting the importance of minimizing communication in the GA fitness function would produce faster run times. To test this conjecture, a small set of runs were performed with decreasing starting and ending weights for the importance of communication minimization. The sets of starting and ending weights which were tried were {{0.5 to 0.4}, {0.4 to 0.3}, {0.3 to 0.2}, {0.2 to 0.1}}. Trying the various parameters to find one that works best is tuning the GA for a particular simulation and machine.

The GA was run using the Mansour-Fox algorithm with a population size of 320 and it was run for 100 generations. For the first 75 GA generations, the weighting of the importance of minimizing communication and load balance were held constant. Starting at generation 75 and continuing to generation 100, these values were slowly changed to their final values. The results of these runs are summarized in table 5.2. The final distribution of cells to processors for the these runs is shown in figures 5.6 through 5.9.

Often	Size	Gen	Wall time	Speed up	GA	Return	Balance	Comm. Time	Weight
1	320	100	975	14.3	40	3.5	10.6	257	0.6-0.5
1	320	100	928	20.2	44	4.3	5.8	303	0.5-0.4
1	320	100	952	17.2	48	3.4	3.5	337	0.4-0.3
1	320	100	942	18.3	46	3.7	2.6	358	0.3-0.2
1	320	100	955	16.8	47	3.4	2.3	367	0.2-0.1

Table 5.2. Summary of runs with varying weights for the importance of minimizing communication.

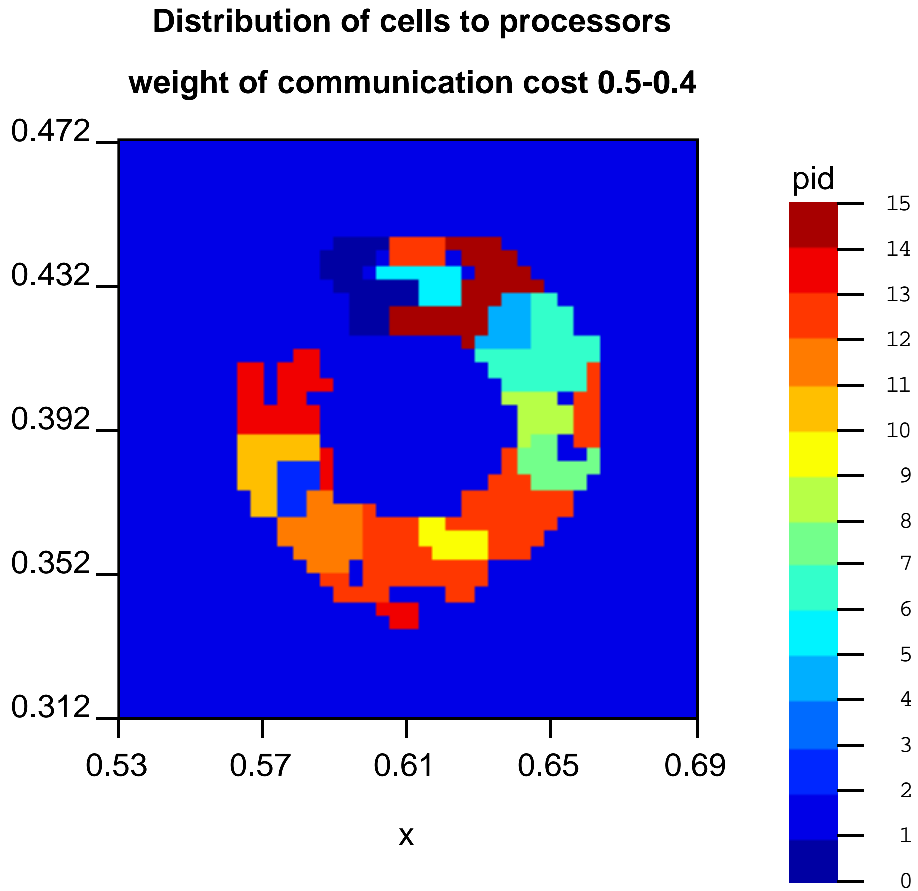


Figure 5.6. Assignment of cells to processors when weight for communication is 0.4 at the end of the genetic algorithm run.

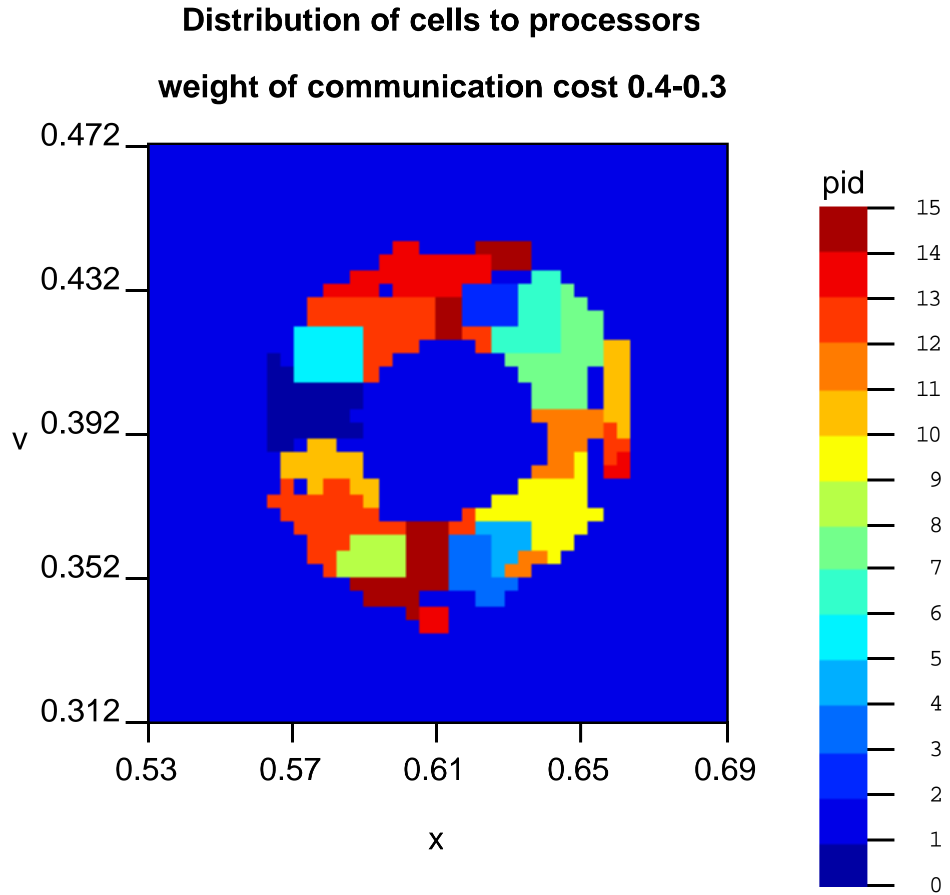


Figure 5.7. Assignment of cells to processors when weight for communication is 0.3 at the end of the genetic algorithm run.

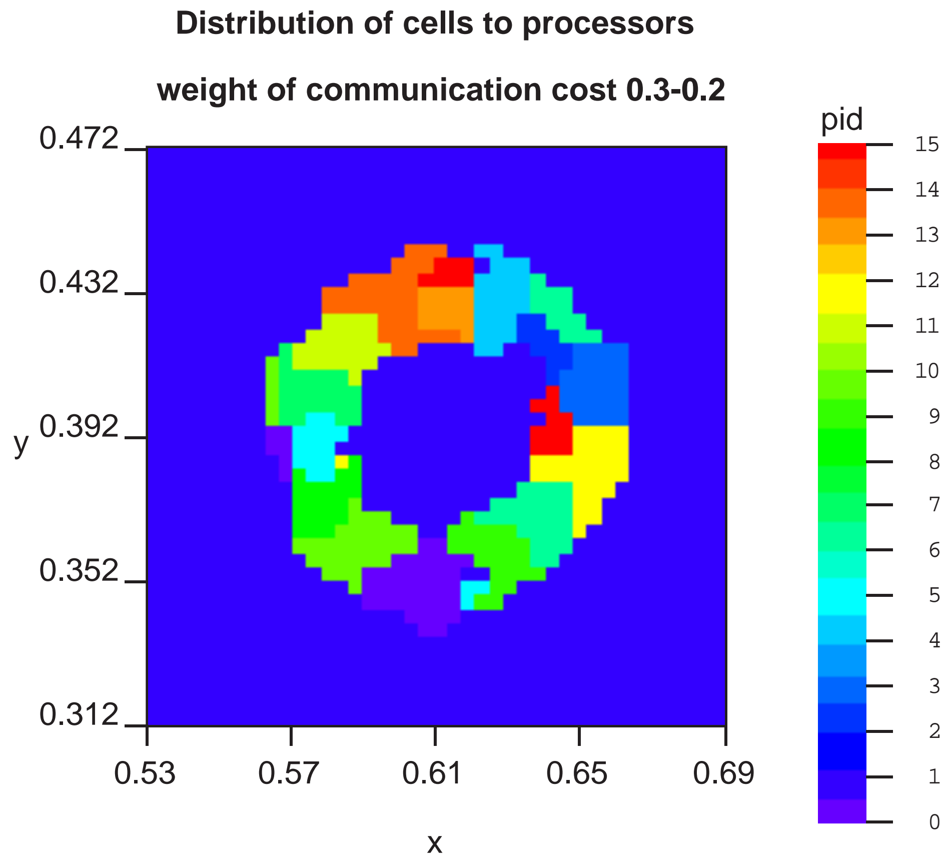


Figure 5.8. Assignment of cells to processors when weight for communication is 0.2 at the end of the genetic algorithm run.

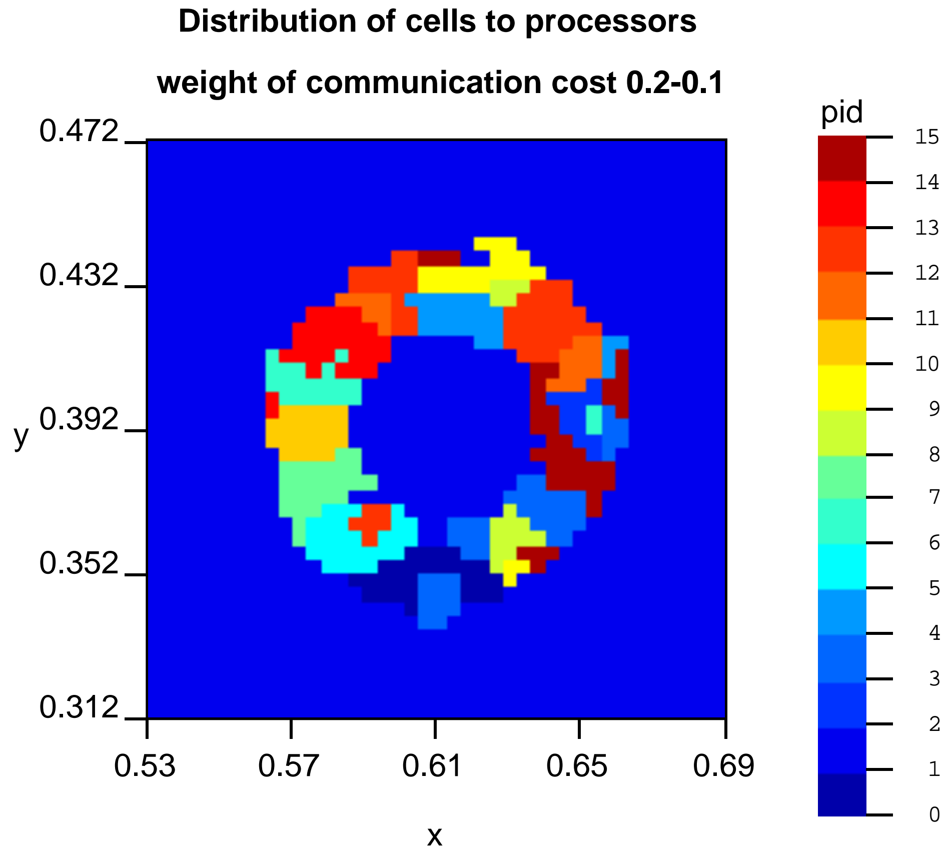


Figure 5.9. Assignment of cells to processors when weight for communication is 0.1 at the end of the genetic algorithm run.

All of these runs were successful. All of them ran faster than all base line simulations. The fastest of these runs was 928 seconds, for a speed up of 187 seconds or 20%. The weightings for this run varied between 0.3 and 0.2.

As the importance of minimizing communication in the fitness function is reduced, the average time spent for communication increases. There was a significant variance in the communication time, from 257 to 367 seconds.

The weighting in the fitness function for load balance is 1 - weighting for communication. As the importance of good load balance goes up in the fitness function,

we would expect to see a better load balance. This can also be seen in the data. From the data we see that as the importance of load balance increases the balance parameter decreases, indicating better load balance. For the first calculation, the final weighting for balance was 1-0.5=0.5, with a value for the balance parameter of 10.6. The calculation was not well balanced. For the last calculation in this series, the final weighting for balance was 1-0.1=0.9 with a value for the balance parameter of 2.3, near what was obtained when communication cost was ignored.

The average run time for this series of calculations is 950.34 seconds, considerably less than the run time of 1115 seconds for the base line case, 165 seconds. Thus it appears that for this particular simulation, the thesis statement of this dissertation has been verified. The genetic algorithm can indeed be beneficial in reducing the run time for adaptive grid programs.

5.5. Verification of the thesis statement.

A further analysis can be performed to determine the statistical significance of the improvement in run times. We propose the hypothesis that the average difference between the run times is greater than 120 seconds, or two minutes. To test this hypothesis an estimate of the variance in run times for the base line case is needed. (Small variances are present because of interrupts that the nodes of the calculation can receive during computation.) Running the same base line calculation 5 times produced a standard deviation for the run times of 6.78 seconds. The average was 1117 seconds. There is a standard deviation of 17.34 seconds for the 5 calculations discussed above. Using the standard test for the difference of two means when only the population means and

deviations are known, Walpole and Myers (1978), we have $T' = \frac{\left(\overline{X_1} - \overline{X_2}\right) - d_0}{\sqrt{\frac{S_1^2}{n_1} + \frac{S_1^2}{n_1}}}$ with

$$v' = \frac{\left(\frac{S_1^2}{n_1} + \frac{S_1^2}{n_1}\right)^2}{\frac{\left(\frac{S_1^2}{n_1}\right)^2}{n_1 - 1} + \frac{\left(\frac{S_1^2}{n_1}\right)^2}{n_2 - 1}}$$

degrees of freedom. We have a value of T'=5.4 and v'=5.3. We

test at the 99% level of confidence so t_a=3.3. With T' > t_a with a 99% level of confidence we do not reject the hypothesis that the average improvement in run time is greater than 120 seconds.

Although this analysis showed the statistical significance of the improvement in the average run time, by further tuning of the GA, results are obtained that are much better than the average improvement discussed above.

5.6. Expansion on sine qua non 1 & 2, tuning the GA algorithm, tuning of the GA using the improved Mansour-Fox algorithm.

Further tuning of the GA was done by running additional parameter studies. For this series of simulations the population sizes and number of generations for the GA varied along with the frequency of calling the hill climbing routine. Also the results of the parameter studies demonstrate trends which attest to the importance of reducing the

frequency of calling the hill climbing routine. That is, there exists a frequency of calling the hill climbing routine, which produces better run times than calling the hill climbing routine every generation. Thus, the importance of the improvement of the Mansour-Fox algorithm of allowing a variable frequency of calling the hill climbing routine is documented. The importance of sine qua non two, improving the GA, is reinforced.

Tables 5.3 through 5.5 summarize the run times for the simulation with the various parameter settings. The size of the population was varied from 240 to 480 members. The number of generations for the GA ranged from 100 to 200. The frequency of calling the hill climbing routine varied from 1 to 5 generations.

Often	Size	Gen	Wall time	Speed up	GA	Return	Balance	Comm. Time
1	480	100	956	16.7	60	2.6	5.5	321
1	480	150	1004	11.0	89	1.2	6.1	302
1	480	200	999	11.6	117	1.0	5.7	308
2	480	100	930	19.9	42	4.4	6.3	304
2	480	150	934	19.4	60	3.0	5.7	299
2	480	200	974	14.4	83	1.7	5.5	319
3	480	100	924	20.7	38	5.1	5.5	307
3	480	150	928	20.2	52	3.6	5.2	305
3	480	200	971	14.9	70	2.0	5.2	309
4	480	100	918	21.4	33	6.0	5.8	308
4	480	150	922	20.9	47	4.1	6.1	293
4	480	200	968	15.2	76	1.9	5.1	313
5	480	100	943	18.2	32	5.3	5.8	320
5	480	150	921	21.1	45	4.3	6.1	297
5	480	200	955	16.8	60	2.7	6.0	314

Table 5.3. Summary of simulation runs using the hybrid genetic algorithm with a population size of 480.

Often	Size	Gen	Wall time	Speed up	GA	Return	Balance	Comm. Time
1	320	100	898	24.2	42	5.2	5.9	283
1	320	150	964	15.7	61	2.5	6.5	314
1	320	200	951	17.3	81	2.0	5.6	299
2	320	100	915	21.8	30	6.8	6.4	297
2	320	150	923	20.7	42	4.5	5.2	307
2	320	200	945	18.0	56	3.0	6.2	304
3	320	100	889	25.4	27	8.4	5.7	293
3	320	150	920	21.2	37	5.3	5.9	308
3	320	200	952	17.1	50	3.3	6.7	299
4	320	100	886	25.8	24	9.6	5.0	288
4	320	150	907	22.9	34	6.2	5.0	299
4	320	200	926	20.5	44	4.3	5.5	319
5	320	100	908	22.8	23	9.2	5.9	301
5	320	150	909	22.7	32	6.4	4.9	310
5	320	200	947	17.7	44	3.8	5.1	325

Table 5.4. Summary of simulation runs using the hybrid genetic algorithm with a population size of 320.

Often	Size	Gen	Wall time	Speed up	GA	Return	Balance	Comm. Time
1	240	100	926	20.5	32	5.8	6.4	312
1	240	150	959	16.3	49	3.2	5.7	332
1	240	200	953	17.0	63	2.6	6.3	309
2	240	100	907	23.0	24	8.8	5.9	307
2	240	150	927	20.2	35	5.3	6.7	296
2	240	200	955	16.8	45	3.5	6.6	320
3	240	100	901	23.8	22	9.9	6.0	300
3	240	150	921	21.1	29	6.7	6.3	306
3	240	200	943	18.3	44	3.9	5.3	323
4	240	100	900	23.9	19	11.2	5.0	312
4	240	150	912	22.2	27	7.4	6.1	298
4	240	200	910	22.6	35	5.9	5.9	301
5	240	100	908	22.8	18	11.2	5.6	317
5	240	150	940	18.6	26	6.9	6.3	317
5	240	200	904	23.3	34	6.2	6.0	293

Table 5.5. Summary of simulation runs using the hybrid genetic algorithm with a population size of 240.

The average run time over all simulations is 932 seconds, for a speed up of over 19%. The worst speed up is 11% and the best is over 25%, with a run time of 886 seconds. The average communication time is 307 seconds, compared to only 41 seconds when no cells were moved.

With the data in three tables it is difficult to discern any general trends. There is one that can be seen. For a fixed population size and fixed frequency of calling the hill climbing algorithm, the return on the investment of calling the GA goes down as the number of generations goes up. This type of behavior is not unusual for genetic algorithms. In the early evolutionary cycles they tend to make large amounts of progress in finding a good match to the fitness function. In later generations the rate of progress slows.

By using different groupings of the data, some interesting trends can be seen. The three tables can be reduced to a single one with 15 entries. Table 5.6 shows the average run times for all simulations with a given population size and a given frequency of calling the hill climbing routine. (The averages were taken over runs with 100, 150 and 200 generations.)

Frequency/Pop. Size	480	320	240
every 1	986.4	937.5	945.6
every 2	946.0	927.9	929.7
every 3	940.7	920.3	921.5
every 4	936.3	906.3	907.2
every 5	939.5	921.2	917.5

Table 5.6. Summary of simulation runs using the hybrid genetic algorithm. Runs with the given population size and frequency are averaged together.

The data from this table was used to create bar charts. A trend can be seen in the bar charts below. For the first chart, figure 5.10, we have three groupings of five bars, the three generation sizes, and the five frequencies. The frequencies increase left to right. Figure 5.10 shows that for a fixed population size, the longest run times for this series occur when the hill climbing routine is called the most often. As the frequency of calling the routine decreases, so does the overall run time, that is, until the frequency reaches 5. At this point the run time starts to go up. For this particular simulation, there is a benefit in not calling the hill climbing routine every iteration. (Recall that for all simulations the hill climbing routine is first called only after 75% of the generations are completed.)

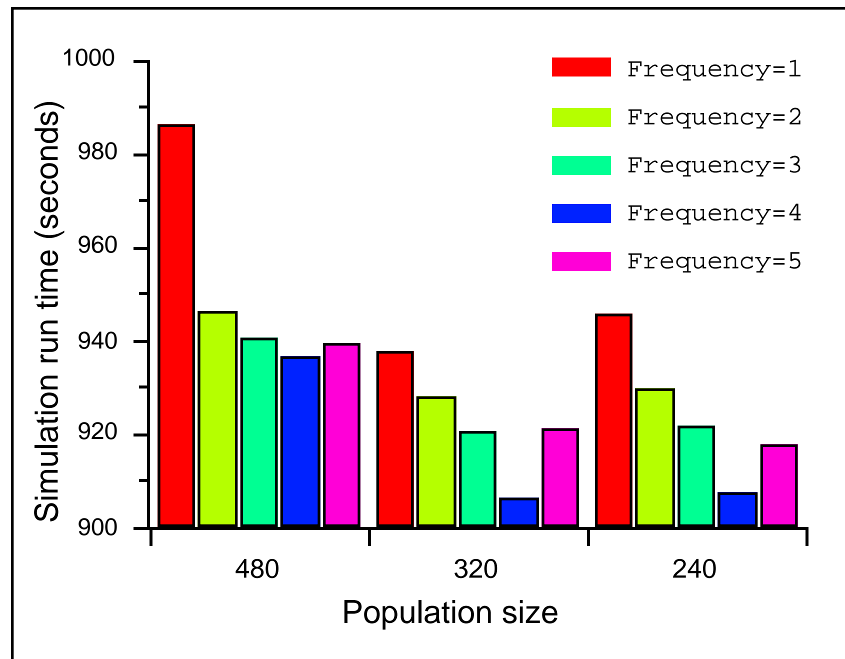


Figure 5.10. Plot of run times for the simulation using the hybrid genetic algorithm. Data is grouped by population size, 480 down to 240 and each bar represents a different frequency of calling the hill climbing routine, 1-5, left to right.

The next chart, figure 5.11, shows the same data but in a different format. Here we have groupings of frequency with the different bars representing population sizes of 480, 320, and 240. It can be seen that the same trend holds for all three sizes of the population.

The maximum run time occurs when the frequency is 1 and the minimum occurs when the frequency of calling the hill climbing routine is 4.

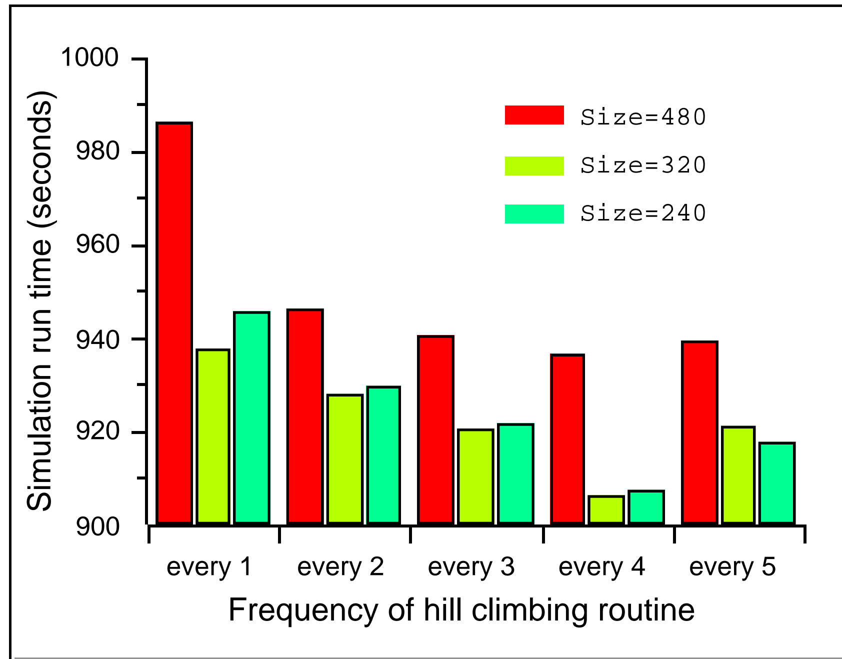


Figure 5.11. Plot of run times for the simulation using the hybrid genetic algorithm. Data is grouped by the frequency of calling the hill climbing routine, 1-5 and each bar represents a different population size, 480 down to 240, left to right.

Instead of averaging the data over number of generations we can average over population size. This again gives us a 15 element chart with variances in frequency and the number of generations for the GA.

Frequency/Generations	200	150	100
every 1	967.5	975.6	926.3
every 2	958.2	928.1	917.3
every 3	955.2	922.8	904.5
every 4	934.5	913.9	901.4
every 5	935.4	923.1	919.7

Table 5.7. Summary of simulation runs using the hybrid genetic algorithm. Runs with the given number of generations and frequency are averaged together.

We see similar trends. The longest run times occur with the highest frequency of calling the hill climbing routine, the best when the frequency was 4.

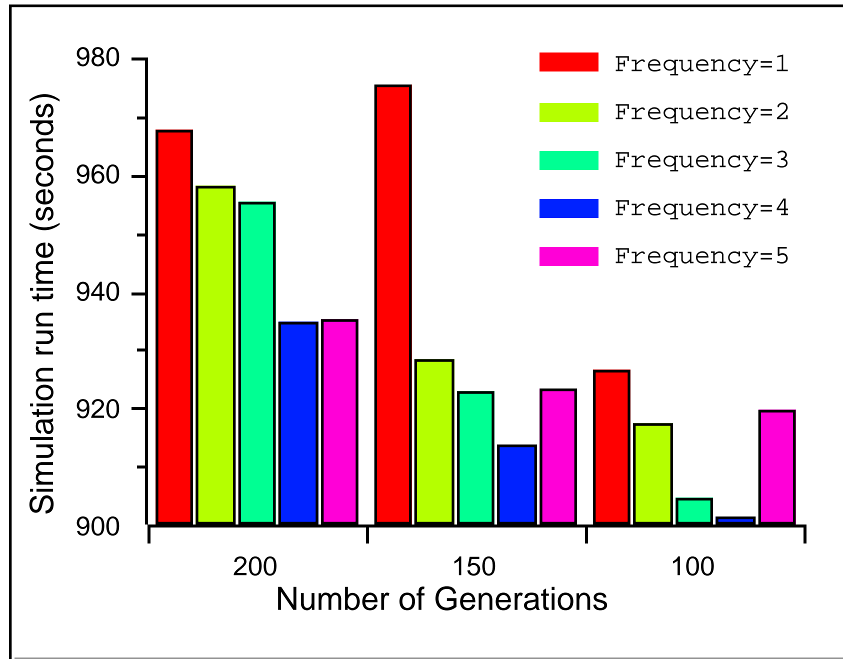


Figure 5.12. Plot of run times for the simulation using the hybrid genetic algorithm. Data is grouped by number of generations, 480 down to 240 and each bar represents a different frequency of calling the hill climbing routine, 1-5, left to right.

For the cases when the GA was run for 200 generations, there is little difference between running with a frequency of 4 and 5. If the frequency was decreased further, the run time would be expected to go up. The limiting case of decreasing frequency is that the hill climbing routine is never called. We would be back to the standard GA which had an average simulation run time of 986 seconds when run for 200 generations.

In summary, for this particular simulation the average best frequency of calling the hill climbing routine was 4. The run times were much faster than when the hill climbing routine was called every generation. Thus the importance of the improvement in the Mansour-Fox algorithm

of allowing a variable frequency of calling the hill climbing routine was confirmed.

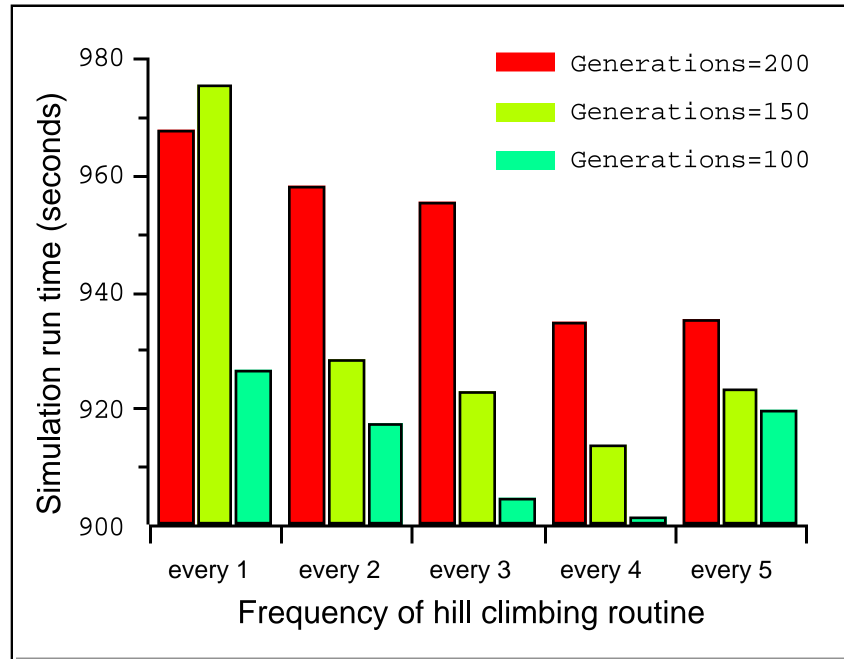


Figure 5.13. Plot of run times for the simulation using the hybrid genetic algorithm. Data is grouped by the frequency of calling the hill climbing routine, 1-5 and each bar represents a different number of generations, 480 down to 240, left to right.

5.7. Application of the results to a different architecture, SP2 runs.

A short series of runs were performed on the MHPCC SP2. The GA control parameters used for these runs were the same as the parameters used for the twelfth run of table 5.4, except the weighting for the importance of communication was changed for each run. The GA was called with a population size of 480 for 150 generations and the hill climbing routine was called with a frequency of 4.

The floating point performance on the SP2 is higher than the SP1. The average time spent in the calculation subroutine for these runs was 148 seconds. For the similar SP1 runs, the time was 193 seconds.

The significant difference between the SP1 and the SP2 is the communication performance. The bandwidth of the communication hardware on the SP2 is 40 Mbytes/second as opposed to 8 Mbytes/second.

It is conjectured that the best performance on the SP2 would be found when the weighting for the importance of communication in the fitness function is lower than on SP1. Five runs were performed with varying weights for the importance of communication in the fitness function. These runs, along with a base case, where the GA was not called, are summarized in the table below.

Wall time	Speed up	GA	Return	Balance	Comm. Time	Weight
892	0	0	0	24.6	13	0
551	61.9	27	12.8	4.8	75	0.5-0.4
511	74.7	26	14.8	2.2	80	0.3-0.2
512	74.5	26	14.8	1.9	81	0.2-0.1
541	65.0	35	10.1	1.8	93	0.1-0.0
547	63.1	30	11.6	1.8	102	0.0-0.0

Table 5.8. Summary of SP2 runs.

5.8. Significance of the SP2 simulations.

We have in this chart the starting and ending weights for the importance of minimizing communication. Notice the communication times for these runs are much lower, by about 200 seconds. This allows for a much higher return on investment for the GA, up from 4 for the SP1 run. Even with the faster communication on the SP2, there is still some benefit in having the GA to try to simultaneously balance load and keep communication low. The best run time was with a weighting of the importance of communication at 0.2. This is in contrast to the SP1 runs where the best runtimes were found when the weighting

was in the range 0.5 to 0.4 as shown in table 5.2. The conjecture that the best performance on the SP2 would be found when the weighting for the importance of communication in the fitness function is lower than on SP1 is confirmed.

Using the SP1 to determine GA control parameters for the SP2 was effective. That is, tuning for one architecture enables the calculation to run effectively on another. Also, by taking into account the difference in the communication performance of the two machines we know that adjustments in the weighting for communication in the fitness function should be done. We are using domain specific knowledge from one machine to predict the GA control parameters for another machine.

> **The domain specific knowledge is the set of GA control parameters that work well for one simulation. This knowledge is exploited by applying similar parameters to the same problem on the different machine.**

We have a more significant result from this set of SP2 runs. The GA included the improvement of fast mutation, Mansour-Fox algorithm, the improvements to the Mansour-Fox algorithms, the use of domain specific knowledge such as including moving cells in clusters, calling the GA at irregular intervals, not counting comunication between split and unsplit cells, tuning the GA using a related simulation, tuning the GA for a particular machine. The techniques included in the GA are the fruit of the two sine qua nons, use domain specific knowledge and improve the algorithm. With the application of the two sine qua nons, the significant result for these runs is that the speed up in the calculation was 75%. We can say:

> **the application of the the two sine qua nons, domain specific knowledge**

must be used and improvements must be made to the genetic algorithm enabled a speed up in the run time of an adaptive grid program of 75% for this problem.

5.9. Application of the results to a different problem.

The SP1 was used to perform another simulation, with initial conditions different from the ones described in section 5.1. The set of control parameters, population size, number of generations, and hill climbing routine frequency that gave the best results for the previous simulation were used for a new simulation. The purpose of this new simulation was to show that the best control parameters found by tuning the GA were useful for a simulation with different initial conditions. This is a further example of the use of domain specific knowledge and also shows the value of tuning the GA for a particular problem.

The domain specific knowledge is again the set of GA control parameters that work well for one simulation. This knowledge is exploited by applying these parameters to a similar problem on the same machine.

The difference in the new simulation was the characteristics of the blobs of energy and mass. For this new calculation the blobs were uniform. Also, they were deposited at slightly different simulation times. The GA was again called at cycle numbers such that it had about 100 clusters of cells to move but the cycle numbers were different than for the previous calculation.

The simulation was run with the GA turned off, with the GA running, and finally using the recursive spectral bisection method to distribute the load. For the run with the GA turned on, the population size was 320 and the number of generations was 100. The

frequency of calling the hill climbing routine was once every 4 generations. The results of the simulation are given in table 5.9. With the load balancing routine turned off, the simulation ran for 1489 seconds with a maximum communication time of 66 seconds. Using the bisection method to assign cells to processors produced a distribution which had a very large amount of communication and thus it ran slower. Using the GA, the simulation ran about 14% faster. With this improvement, it is shown that with the use of the domain specific knowledge, GA control parameters that worked well for a similar run, the GA was able to produce a faster run time.

Often	Size	Gen	Wall time	Speed up	GA	Return	Balance	Comm. Time
0	0	0	1489	0	0	0	41.4	66
0	0	0	1664	-10.5	0	0	4.4	908
4	320	100	1309	13.8	43	4.2	7.5	539

Table 5.9. Summary of SP1 runs with a different scenario.

5.10. Comparison to bisection results.

The chart above compares the GA to a bisection technique. To expand on this comparison, the original example calculations discussed in this chapter were run on the SP1 and SP2 using both the recursive coordinate bisection method and the recursive centroid bisection method to assign cells to processors. These routines were called at the same iteration number as the GA and were used to move the same clusters of cells. The run times are reported in table 5.10.

Machine	Algorithm	Run time	Comm. Time	Balance
SP1	Best GA	886	288	5
SP1	Centroid Bisection	1014	452	4.7
SP1	Coordinate Bisection	1122	544	4.8
SP2	Best GA	511	80	2.2
SP2	Centroid Bisection	526	104	3.6
SP2	Coordinate Bisection	571	119	3.8

Table 5.10. Comparison of run times using the GA and bisection methods.

Both bisection routines did a good job of balancing the load. The communication costs required by the distributions produced by these algorithms were higher than those produced by the GA.

On the SP1, the run times were considerably longer than the run times using the GA. The primary culprit for the increase in run times was the communication cost. As expected, the communication cost using the recursive coordinate bisection method was the highest.

One of the reasons why the communication cost was lower for the GA gets back to the tendency for the GA to form wedge distributions. There is a correlation of where cells are placed from one invocation of the GA to the next. Once a wedge is started, it tends to grow on each invocation of the GA, keeping the communication cost down. The "force" causing these wedges to grow is the penalty in the fitness function for communication. For the bisection methods, there is no force tending to cause clusters of cells on a single processor to grow.

The runs on the SP2 using the bisection methods are slower than the best runs of the

GA, but the difference between the recursive centroid bisection runs and the GA are not as great. Again, this is because the communication is much faster on the SP2. Even with the faster communication hardware, the run times using recursive coordinate bisection method are slower than the GA runs, again because of communication time.

5.11. Summary.

For this particular simulation the average best frequency of calling the hill climbing routine was 4, with run times much faster than when the hill climbing routine was called every generation. The importance of the improvement in the Mansour-Fox algorithm of allowing a variable frequency of calling the hill climbing routine was confirmed.

A set of GA control parameters that worked well for one simulation was applied to a different machine with modification to account for the difference in communication speeds. This use of domain specific knowledge allowed the GA to significantly reduce the run time for the simulation on the new machine.

Also, the domain specific knowledge, the set of GA control parameters that work well for one simulation, were used on a second similar simulation. This enabled the second calculation to see speed up without any additional tuning of the GA.

With the application of the two sine qua nons, domain specific knowledge must be used and improvements must be made to the algorithm; the genetic algorithm enabled a speed up in the run time of an adaptive grid program of 75%.

Chapter 6. Summary.

6.1. Introduction.

This chapter summarizes the accomplishments of this research effort. Recommendations are given about how what was learned in this research effort could be applied to other areas. Ideas for future work are also presented.

6.2. Showed the GA to be effective.

The ability of genetic algorithms to reduce run times of adaptive grid programs by performing dynamic load balancing and communication reduction was tested. Simulations were run to determine the effectiveness of the GA using the ARC SP1 and the MPHCC SP2.

The significance of the SP1 and SP2 tests is that these machines have a level of performance which is available to most universities for about $3,000 per node. A similar machine could be created using commercial off the shelf workstations and 100 BaseT ethernet connectors.

Using the SP1, a hybrid genetic algorithm has been shown to be effective in reducing run times for an adaptive grid program by maintaining good load balancing and maintaining low communication. Running on 4 nodes of the SP1, the simulation time for the example calculation was reduced from 1451 seconds to 1297 seconds, for a speed up of 12%. Running on 16 nodes, the run time for the example calculation was reduced from 1115 seconds to 886 seconds for a speed up of 26%.

The hybrid genetic algorithm has also been shown to be effective in reducing run

times on 16 nodes of the SP2. For the example calculation on the SP2 the speed up was 75%.

6.3. Discovered sine qua nons.

Genetic algorithms have not been used for this application in the past because they have been viewed as being too slow to converge to a high quality solution. However, the research described in this document has shown that a genetic algorithm can be used to derive a reasonable solution and that solution can be effective in increasing the performance of adaptive grid programs, even when the run time for the GA is counted. How is this possible? It is imperative that additional heuristics be used to aid the genetic algorithm and that the GA be designed to run quickly. The heuristics and algorithm changes are related to two sine qua nons which must be observed to allow the genetic algorithm to increase the performance of the adaptive grid program. One, domain specific knowledge must be used. Two, improvements must be made to the GA. The use of domain specific knowledge aids the GA in finding a high quality solution. The improvements in the GA are designed to decrease its run time yet still allow it to find good solutions.

6.3.1. Improvements to the GA.

6.3.1.1. Fast mutation.

The standard method of mutation in a GA runs in time proportional to the population size times the size of the vector describing the members of the population. A mutation methodology was developed for use in this research effort which runs in time proportional to the population size times the number of genes, times the mutation probability P. For small P this method is much faster.

6.3.1.2. Mansour-Fox algorithm and its improvement.

Mansour and Fox suggested a collection of modifications to the basic genetic algorithm to improve the quality of solutions returned. Their modifications to the standard GA were incorporated into the load distribution routines. When the Mansour-Fox algorithm was applied to this adaptive grid problem the algorithm returned good distributions of cells to processors, with good balance and low communication times. Unfortunately, the Mansour-Fox algorithm ran too slow. Much of the improvement in the run time from moving cells to other processors was hidden by the long run time of the Mansour-Fox algorithm.

There are two problems with the Mansour-Fox algorithm. First, the greedy hill climbing routine is costly. Secondly, it can not be terminated early because of its dynamic fitness function.

Addressing these problems in this research effort led to improved algorithms. First, the GA is run as a parallel algorithm. Secondly, the greedy algorithm is called every N generations instead of every generation. N is an input parameter. Finally we have the option of using a static fitness function.

The addition of the ability to call the greedy algorithm every N generations and to do early termination enables a trade off of distribution quality and GA run time. It was found that calling the greedy algorithm every N generations decrease the run time of the GA, but does not greatly decrease the quality of the distribution returned by the algorithm. This reduction in run time while still returning a good solution allowed the GA to be effective in reducing the run time of the adaptive grid program.

The best value for “N” for a particular simulation is a function of the calculation

being performed. For the 16 node example calculation a value of N=4 on average produced the best simulation run times.

6.3.2. Use of domain specific knowledge.

The first piece of domain specific knowledge used was that if the blocks of contiguous cells on the same processor are large, communication will be less than if the blocks of contiguous cells on the same processor are small. This knowledge was exploited by moving cells in clusters.

The second piece of domain specific knowledge used was that over the course of the calculation, clusters of cells are generated at irregular rates. By measuring the rate of the generation of clusters, and using this measurement to project load distributions, the efficiency of the GA can be increased.

We have a fourth use of domain specific knowledge that, as a shock wave propagates, the cells just in front of it tend to split. We exploit this knowledge by not counting the communication between the split cells and the unsplit cells in the fitness function. The use of this domain specific knowledge greatly increased the ability of the genetic algorithm to improve performance by decreasing the communication cost of the distributions of cells to processors.

6.4. Tuning of the GA.

The GA can be tuned to produce better results, that is, enable the simulation to run faster. It is tuned by adjusting the number of generations, the population size, and frequency of calling the hill climbing routine. The GA can be tuned to produce good results for a particular simulation. More importantly, once the GA is tuned for a particular

simulation the parameters can be used to produce good run times for related problems. A related problem is one with different but similar initial conditions. A problem with the same initial conditions but run on a different machine is also considered a related problem.

6.5. Developed tools.

As part of this research effort, a collection of software tools and numerical algorithms were developed. The program used to test the heuristics for assigning cells to processors was a combination of the PLIFE framework, the GA framework and the numerical routines, all created for this work. PLIFE is a framework for creating two dimensional, parallel, adaptive grid, or adaptive mesh programs. The GA framework, CHUCK, can be used to create parallel genetic algorithms. It includes a fast mutation methodology and supports several modes of parallel operation.

6.6. Recommendations.

6.6.1. Use the two sine qua nons.

The two principles that enabled the GA to be effective were: that domain specific knowledge was used, and improvements were made in the genetic algorithm. These two sine qua nons can be applied to other applications, with or without GAs. Heuristics that are developed to solve a particular problem should take advantage of problem specific data. The GA is a "classic" algorithm that has been used in its original form for many problems. When tackling a problem, even using a classic algorithm, methodologies should be sought to improve it for the particular problem at hand.

6.6.2. Use a framework to test and develop applications.

The PLIFE framework can be used to develop parallel adaptive grid applications. Other new numerical and load balancing routines are easy to incorporate into the framework.

Thus, PLIFE can also be used as a test platform for these routines, independent of whether the final target application for these routines is based on the PLIFE framework.

6.6.3. Use the improved Mansour-Fox algorithm.

Mansour and Fox have shown that their hybrid genetic algorithm is more efficient than a standard genetic algorithm for creating grid to processor assignments. With this work, it has been shown that reducing the frequency of calling the hill climbing routine does not greatly decrease the quality of the solution returned by the algorithm but the run time can be significantly decreased. Thus, for real time or time critical optimization problems where speed is critical, the modified Mansour-Fox algorithm should be used.

6.7. Areas of future work.

6.7.1. Apply Mansour-Fox algorithm to other problems.

Mansour and Fox have developed an algorithm which finds good solutions. The improved Mansour and Fox runs quickly. It could be applied to other optimization problems. One potential area of application is image enhancement. As an optical signal propagates through the atmosphere, phase distortions can be introduced in the wave front by turbulence or gradients in the optical properties of the atmosphere. These distortions cause the recorded image to be distorted. Mathematically, the distortions are variances in the Fourier components of the image. If these phase distortions are known then they can be removed and the image becomes sharper. A genetic algorithm could be used to find the variances in the Fourier components of the image. The fitness function for the GA would be a measure of the image sharpness. See Muller and Buffington (1974) for a discussion of this problem.

6.7.2. Further development of framework.

Several areas have been discussed for future development of the framework. The most important of these deal with increasing the generality of the data structures describing the grid.

Presently it is assumed that cells are rectangular and each cell has four neighbors. The data structures could be generalized so that cells are polygons with an arbitrary number of sides, and cells could have an arbitrary number of neighbors. This would allow numerical schemes to be used which are not based on a five point stencil.

One of the problems encountered in running the application based on PLIFE is the use of large amounts of memory to store the description of the grid. PLIFE could be modified so that each processor does not hold a description of the complete grid. This would decrease the memory usage and promote the scalability of applications based on the framework.

6.7.3. Load prediction.

Off loading more work than required to balance a calculation from processors that are creating additional work faster than the others was found to be useful. An exploration into load prediction to optimize this process may be useful. This will be especially useful for calculations for which load is being created rapidly on a few of the many nodes of a calculation.

6.7.4. Work on configurations with fast communication.

If a simulation is being performed on a processor with very rapid communication, then it is more important to balance the load than to worry about both balancing load and maintaining low communication. Tests could be performed on machines such as the

Cray T3E. (See Cray T3D , http :// www.cray.com /PUBLIC /product-info/ mpp/ T3D_overview.html). These tests could be done to see if it is beneficial to totally ignore communication or if some consideration must still be given in the fitness function to having reasonable communication.

6.7.5. Run tests on multilevel ASCI systems.

Simulations could be performed on a hybrid machine, a machine which had a collection of shared memory nodes connected together by a high speed network. Machines have been proposed which take this concept one or more steps further. A parallel machine would be comprised of shared memory nodes. The shared memory nodes would be connected together in clusters by a high speed network. Groups of the clusters would be connected together by another network and so on. The key aspect of this architecture is multiple levels of communication bandwidth and latency between nodes. The hybrid genetic algorithm could be tested to see its applicability for this architecture. Also, it may be useful to run the GA to balance the load at several levels of the communication hierarchy. Thus the GA could balance the load amongst the clusters. Then the GA could balance the load of the individual nodes of the clusters. You could also use one of the nodes of the calculation to balance the load while the others continue to perform the numerical calculation.

6.7.6. Use other numerical routines.

The numerical routines which drove the adaptive grid program were based on the Euler equations of gas dynamics. Additional tests of the GA could be run using the same equations but different initial conditions. Also, different routines could be used, such as the Naiver-Stokes, or the shallow water equations. The GA could also be used to perform optimization for three dimensional simulations.

6.7.7. Run comparisons of GA with/without binomial probability distribution.

The realization that the mutation process was a binomial experiment was used to speed up the genetic algorithm. Stand alone genetic algorithm runs could be performed comparing the run time, and quality of solutions returned, using the algorithm described in this dissertation and the standard method of performing mutations. In the base line case, genes in the chromosome would be tested one at a time to see if they are to be mutated. In the second case a binomial random variable would be used to determine which genes would be mutated so that each would not need to be tested.

Appendix A. The PLIFE Framework.

A.1. Overview PLIFE framework.

PLIFE is a framework for creating two dimensional, parallel, adaptive grid, or adaptive mesh programs. PLIFE is the application framework which was used as a test platform to evaluate, and validate strategies described in the thesis of this dissertation. This section, which describes PLIFE, contains several subsections. We discuss PLIFE's heritage, that is, we mention the previously written programs that most directly influenced its development. Next, we have a discussion of the requirements which drove the design of PLIFE. Then we discuss the overall programming model of the framework. A description of the data structures used to implement the programming model of PLIFE is given. The communication setup routines are discussed. Some ideas are given of how PLIFE maybe used by others. The techniques for modifying PLIFE to simulate a particular physical system are outlined. The lessons learned from the construction and use of PLIFE are also discussed.

A.2. Heritage of PLIFE.

The development of the overall structure of PLIFE had two primary influences. These influences were the hydrodynamics simulation, SAGE, and the graphics package NEWPLOT.

SAGE is a hydrodynamics simulation developed by Mike Gittings of Science Applications International Corporation, Gittings (1992). It is used primarily by the scientists at Los Alamos National Laboratory to perform calculations to estimate shock wave propagation.

NEWPLOT was developed by this author at Science Applications International Corporation, Kaiser (1993). NEWPLOT is an application for visualizing data stored in a grid. It has extensive data analysis and post processing capabilities. It can be used to visualize and analyze regular grid data sets, and irregular grid data sets produced by both SAGE and PLIFE.

A.3. Requirements of a tool for thesis verification.

The primary purpose of the development of PLIFE was to have a tool for the study of load balancing techniques for adaptive grid programs. A program tool used to evaluate the research for this dissertation must have several characteristics. First, it must perform some type of numerical calculation. The calculation must be a simulation of a physical system using a grid of data points or cells. The tool must operate as a parallel program, with sections of the grid assigned to various processors. The program must allow adaptive grids. That is, the program must have the capability to change the grid resolution in particular locations. This is done either by taking a particular cell and breaking it into a collection of cells, or by taking a collection of cells and combining them to create a larger cell. Finally, to test various dynamic load balancing strategies, the program must be able to move data cells from one processor to another.

PLIFE was designed to meet these requirements to test dynamic load balancing strategies for adaptive grid programs. In addition to the basic requirements outlined above, PLIFE was designed to be flexible and portable.

A.4. Programming model for PLIFE.

First, we give a quick summary of the programming model of PLIFE. Additional

details will be given throughout this section.

A.4.1. Overview of the functional units of PLIFE.

Programs written using the PLIFE framework consists of three logically and physically distinct units; numerics, load balancing, and dynamic regridding and communication management. The user is responsible for adding the numerics and load balancing routines. The dynamic regridding and communication management section is the core of the PLIFE framework and is not changed by the user. The figure below shows a block diagram of the relationship between the various parts of an application derived from the PLIFE framework. The program developed for this research effort using the PLIFE framework is about 14,000 lines. Of this, 2,000 lines are user supplied and the rest is the basic framework.

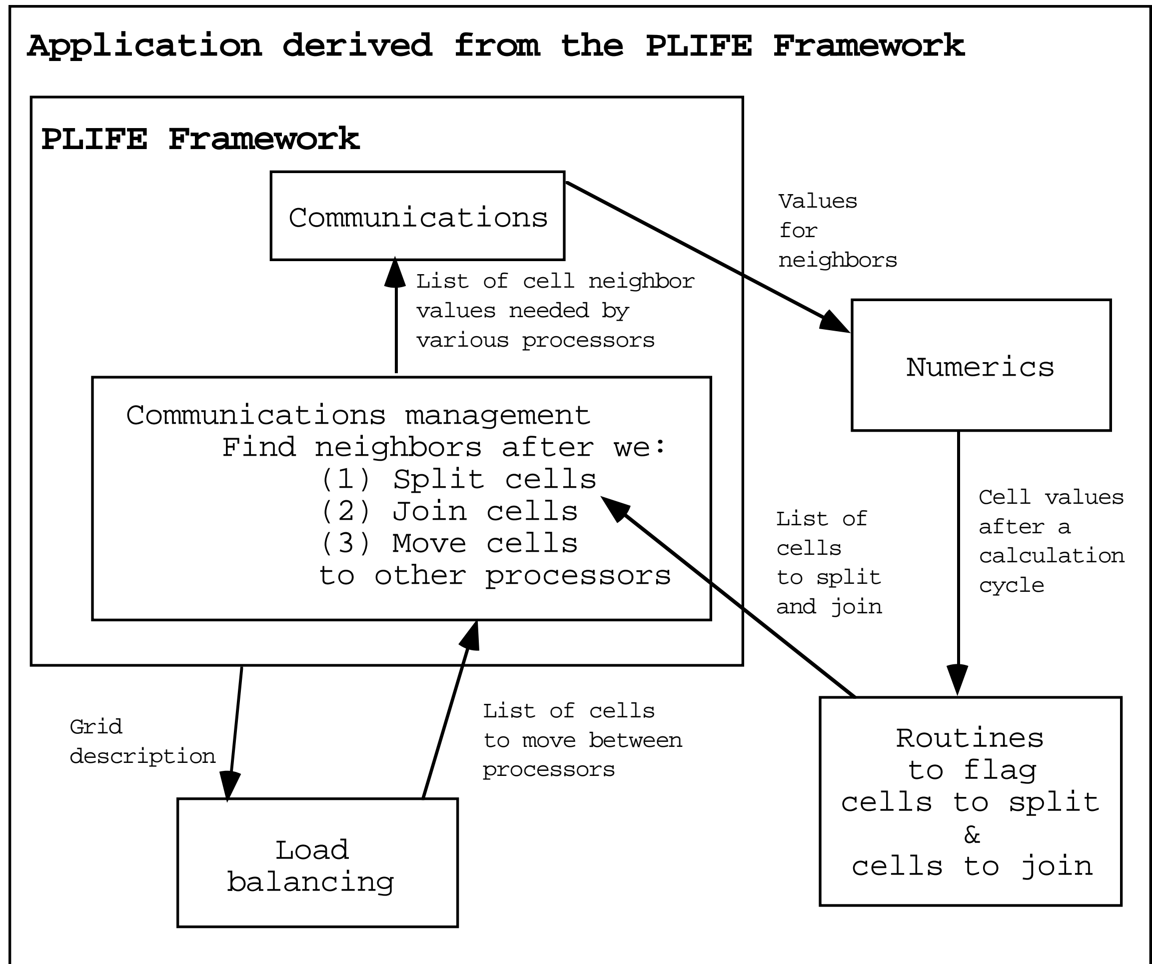


Figure A.1. Diagram of an application derived from the PLIFE framework.

A.4.2. Numerics.

The numerics of PLIFE are the routines which perform the calculation for the simulation of a particular physical system. Users of PLIFE are able to add to this framework routines which model a particular physical system, as long as that model is based on a five point stencil. The class of problems which can be modeled using a five point stencil is extremely large. The communications routines with PLIFE ensure that the data required from the surrounding four cells is available when needed, even if neighboring cells are on different processors.

For this research effort, numerics were added to solve Euler's equations of hydrodynamic flow. The routines are described in Appendix B.

A.4.3. Load Balancing.

Users are free to implement their own dynamic load balancing strategies. PLIFE makes a complete grid description available to the load balancing routines. The grid description contains sufficient information to determine, the location for all cells, their neighbors, and the processor which holds a particular cell. PLIFE requires, as output from the user supplied routines, a list of cells to move between processors. PLIFE moves the values for the cells as requested, assigns responsibility for the moved cells to the new processor, reassign neighbors, and adjusts communications. For this research effort, the dynamic load balancing routines were based on a genetic algorithm.

A.4.4. Dynamic regridding and communication management.

The core of PLIFE is the dynamic regridding and communication management section. PLIFE "takes care of" the bookkeeping. Dynamic regridding is the action of cells being created or a collections of cells being joined into fewer cells. PLIFE determines neighbors for the new cells and sets up pointers from cells to their neighbors. If a processor has cells which have neighbors on a different processor, PLIFE will set up the communication so that values are passed when they are needed.

A.5. PLIFE background information.

PLIFE is designed using the single program, multiple data or SPMD paradigm. PLIFE was designed as a SPMD application because each processor performs the same

type of calculation but on different sections of the calculation grid, that is, different data.

Like SAGE, PLIFE was designed to perform a simulation of some physical phenomena occurring in a two dimensional region and evolving in time. The region is divided into a grid of cells. Each cell represents a small rectangular section of the total area. Cells do not need to be square. The values of the variables for each cell change as a function of time. Values for a cell, at time T_{n+1}, are a function of the cell variables at the previous time step, and the values of the four surrounding cells at the previous time step.

For both SAGE and PLIFE, cells do not need to be all the same size. Regions were a higher resolution is desired can have a finer grid, and thus smaller cells.

It is possible for both programs to adapt the grid. This means that the grid resolution can be a function of both time and space. If, at some time, a higher resolution is desired in some region, then the cells in that region can be split into four rectangular cells of equal area. A region which is represented by a single cell is then represented by four cells. One or more of these four cells can be split again to obtain an even higher resolution.

PLIFE handles much of the bookkeeping associated with a parallel adaptive grid program. PLIFE handles communication and communication setup, the splitting and joining of cells for adaptive gridding, and moving cells between processors.

Many grid calculation algorithms are used for scientific simulations such as weather forecasting or hydrodynamics. The traditional language of these scientific simulations

has been Fortran 77, or even earlier versions of Fortran. SAGE, a hydrodynamics simulation, is written in Fortran 77 with some Cray extensions to handle dynamic memory allocation.

PLIFE is not a hydrodynamic simulation, but it can be used as a framework to write such simulations. It is desirable for PLIFE to have the ability to dynamically allocate memory. As described below, it is also desirable to have derived data types. Although some compiler writers have added extensions to Fortran 77 to allow for both of these features, programs written using these extensions are not portable.

It is also desirable to have an operator overloading capability. This aids the customizing of PLIFE for particular numerical routines.

Two common languages which provide for dynamic memory allocation, derived data types, and operator overloading are C++ and Fortran 90. PLIFE was written in Fortran 90. Fortran 90 was chosen primarily in the hope that the framework would be customized and used by researchers performing large scale scientific calculations. Many of these users are already familiar with Fortran. See Adams., Brainerd, Martin, Smith, and Wagener (1992). for a description of Fortran 90

As a side note, a conversion of PLIFE to C++ would be nearly mechanical because the Fortran 90 constructs used in PLIFE are available in C++.

A.6. Overview of the grid structure.

One of the features which PLIFE inherited from NEWPLOT is the method used to

store the grid. NEWPLOT stores the grid using a forest of quad trees. The individual trees of the forest are held in a two dimensional allocatable array. The array is an array of data structures containing many pointers. We have pointers to parents in the tree. If a node in the tree is not a leaf, then we have pointers to four children allocated. This gives the basic tree structure. We also have pointers for the neighbors to a cell. One additional pointer is used to reference the data values for the cells. Two additional pointers, two logicals, and two integers are used for various bookkeeping functions. With four bytes for each pointer and integer, each cell occupies a minimum of 58 bytes of storage. Additional space is required if the data value pointer is allocated. Additional details on this storage method are given below.

Each processor holds a complete view of the grid. But, each processor only holds data for cells for which it is responsible for updating and the neighbors for these cells. Before values for cells are updated, communication occurs so that every cell knows the values for its neighbors.

When values for a cell are updated, the program must know where to obtain the values for the surrounding cells. This is why each cell has pointers which give the location of its neighbors, and in turn, its neighbor's values.

PLIFE maintains on each processor, a list of the processor's cells whose values are needed by other processors. These cells are needed by the other processors because they are neighbors to cells held on the other processors. PLIFE sends the values to the other processors as needed at every calculation cycle.

The numerical routines which deal with updating cell values work the same way even if the grid is distributed amongst various processors. This makes it easier to write routines which update cell values; that is, the code is independent of where a neighbor cell is located.

Communication in PLIFE is handled using the message passing interface (MPI) message passing system. MPI was used primarily to support the portability of the application. MPI has other features which were useful in creating PLIFE. In particular, the call MPI_ALLTOALLV was very useful. Using a single call to MPI_ALLTOALLV, PLIFE can perform a global exchange of data with each processor sending and receiving different amounts of data. MPI also has the ability to pass user defined data types in addition to the standard data types. This ability is useful when customizing PLIFE to perform a particular type of physical calculation.

Responsibility for a cell can be shifted between processors. That is, a cell can be moved from one processor to another. Again, when this occurs, PLIFE will find the neighbors and adjust communications appropriately. This feature is used in dynamic load balancing.

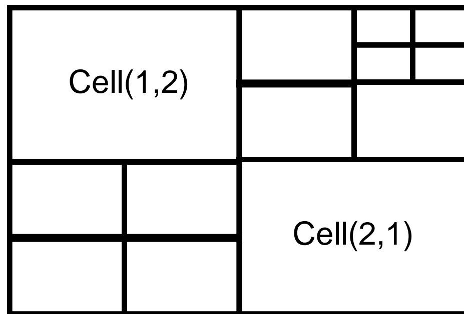
A.6.1. Grid data structure.

Traditionally, for calculations which are being performed to model a two dimensional physical area, the data cells have been distributed evenly over a grid. More recently irregular grids have been used. This section discusses in more detail the data structures used by PLIFE to accomplish the irregular gridding.

We first discuss the forest of quad trees. Every cell is a node or a leaf in a quad tree. If a cell is not a leaf in the quad tree then it has four children. Cells have parents, and can have a southwest child, a southeast child, a northwest child, and a northeast child. Of course, cells at the root of a tree do not have parents and leaves do not have children. When finer zoning is desired in a given region, cells are split. Only cells which are leaves in a quad tree can be split. When a cell is split, it is split into four subcells of equal size.

The collection of base cells, which are the roots of the trees in the forest, are stored in a two dimensional array. The number of base cells in the two dimensional array is determined by input parameters. In addition, the physical size represented by the base cells in the horizontal and vertical directions is also determined by input parameters. Thus, the area being simulated in a PLIFE calculation is determined by the number of base cells in each direction and the size of the base cells. The bottom left base cell is given the designation (1,1). The base cell to the immediate right is (2,1) and the one above (1,1) is (1,2).

Consider the grid shown below with two base cells in the horizontal direction and two in the vertical direction. Cell (1,1) and (2,2) have been split into four cells. In addition the northeast child of cell (2,2) has also been split. The traditional forest of trees representation for this collection of cells is also shown.



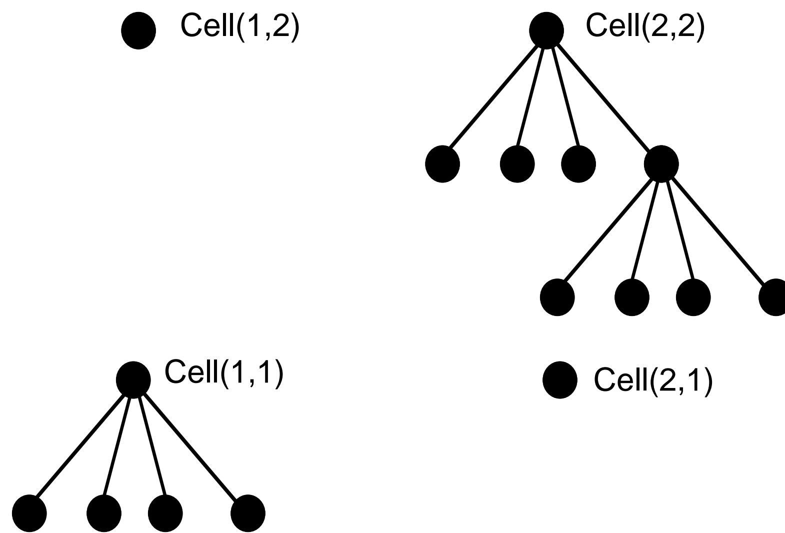


Figure A.2. The grid structure used within PLIFE.

There are other fields in the data structure for a cell. Each cell has a *value_type* data structure and data structures which are used to point to neighbors. We discuss the *value_type* derived data type next.

The *value_type* data structure contains the information which represents the physical quantities being calculated as part of the simulation. If PLIFE is being used for a hydrodynamics simulation then *value_type* may contain fields for pressure, temperature, density, and material velocities. In the game of life, *value_type* would contain a single

field which would be the population density for a cell.

PLIFE performs calculations using a five point stencil. To update a value of a particular cell, values for the four surrounding cells must be known. If a cell is on the edge of a grid, then the pointer is to a ghost cell. If a cell in not on an edge, a neighbor is either the neighbor of a cell's parent or a child of the neighbor of the parent.

A.6.2. Each processor holds a complete description of the grid.

The primary motivation for creating PLIFE, that is, to have a tool for use in studies of load balancing strategies, influenced key aspects of the design. It is desirable to have the potential for the load balancing algorithms to work in parallel, with portions running on all processors. Also, it is desirable to have the ability to test algorithms which require a global picture of the distribution of cells. There is a desire to have the ability to test algorithms in which each processor has the same view of the forest of trees which describe the grid of cells. If each processor has only a limited knowledge of the structure of the grid, flexibility in designing load balancing strategies is limited. It was decided to give each processor a complete collection of the forest of trees which describe the grid layout.

A.6.3. Each processor only holds values which are required.

Even though each processor has a complete description of the layout of the calculation grid, this does not mean that each processor holds data for all cells of the calculation. Each processor only holds data for cells for which it updates values and the data for neighbors of its cells.

The principal disadvantage of each processor holding a complete grid description is the use of extra memory. Allocating the value data structure only for cells which are used on each processor reduces the extra memory usage. Depending on the system which is being simulated, the value data structure can contain any number, possibly a large number, of fields. Thus, the memory savings can be significant.

A.7. Boundary conditions.

Specific boundary condition calculations may be required to update values for cells which are on the edge of the grid. These are different from calculations for interior cells. One common method of facilitating the boundary condition calculations is to create ghost cells as neighbors to cells on the edge of the grid. Neighbors for a cell are first determined when a cell is created. If the cell is on the edge of a grid, a ghost cell is created as a neighbor. This ghost cell is added to a list of ghost cells which are updated as needed at the start of a calculation cycle.

A.8. Calculation cycle.

This section describes what occurs during a PLIFE calculation cycle. A cycle is the advancement of the simulation time from T_n to T_{n+1} or, from T to T+ΔT, and the corresponding updating of all cell values.

At the beginning of the cycle, values for neighbors which are held on other processors are sent and received. Ghost cells are also updated.

After ghost cells are updated and values for neighbors are received from remote processors all of the data required to update a cell is available. Cell values are updated

using values from the surrounding cells and a user supplied calculation subroutine.

Other auxiliary calculations may be performed. For routines which use an adaptive time step, the new time step may be calculated. Also grid adaptation, that is, cells splitting or joining, may occur.

A.9. Cell splitting, joining, and finding neighbors.

Because cell splitting, joining, and the process of coordinating the communication for neighbor cells across processors are intimately related, they will be discussed together.

Every processor is responsible for determining if one of its cells is to be split or if four cells are to be joined into a single cell. An identifier for the cell to be split is added to a list. This list is then broadcast to all processors so the data structure describing the grid can be updated.

When a cell is split into four new cells, the processor holding the new cells is responsible for finding the neighbors for the cells. The neighbors for a new cell may be on a different processor. A cell maybe the neighbor to more than one cell on another processor.

If a processor holds a cell which is a neighbor of a cell which splits or joins, then that cell's neighbors can also change. In this case, the processor which has its neighbor change is responsible for finding the new neighbor. It is possible that if a collection of cells are joined to form a single cell then the processor holding that joined cell may no longer need a neighbor from a different processor.

Every processor holds a list of its cells which are needed as neighbors by other processors. Along with this list, a count of how many times the cells are used as neighbors on another processor is maintained.

When cells are created or deleted, a new list is created of new neighbors needed from other processors. Again, along with the list is the number of times a particular cell is needed as a neighbor. The new list does not contain all cells which will be sent from other processors. It only contains a change in the list of cells to be sent. On this list, the number of times a cell is needed as a neighbor can be negative. This indicates that the cell is no longer needed as a neighbor by as many cells.

The lists of new neighbors are exchanged amongst the processors. The newly received lists are merged with a master list of cells which are already being sent to other processors. If the count of the times a cell is needed goes to zero then it is removed from the master list.

Each cell has a unique identifier consisting of two integers. The first value is determined by the path to the cell in a particular tree in the forest. The second is determined by the processor which holds the cell and also, the tree in the forest to which the cell is a member. This identifier is used as a tag in the list of cells. Each entry in the list of cells to be sent has the tag and the number of times it is needed by a remote processor. If a cell is needed by two remote processors, it will be on the list twice.

Next we describe what happens when a processor sends a collection of cell values to

other processors. The values being sent are copied into an array. The values being sent to processor zero are first in the array and the values for processor one are next and so on. Values are sent using the MPI call MPI_ALLTOALLV. MPI_ALLTOALLV is a collective broadcast and receive routine. A single call will cause values to be sent and received from all processors. The number of values sent and received from the various processors does not need to be the same. The received values are stored in an array. The values received from processor zero are stored first in the array.

When values are received using the MPI_ALLTOALLV call, they do not need to be copied to cells. When a cell is to receive its value from a remote processor, the pointer associated with the value data structure is not given an arbitrary memory location. It points to the location in the array of received values which holds the correct data.

Therefore, when transferring data from one processor to another, a copy of the data is performed on the transmitting end but a copy is not required on the receiving end. Ideally, a copy would not be required on either end. Unfortunately, all algorithms considered in designing PLIFE for transferring data without at least one copy caused deficiencies in other areas of the program design and so were rejected.

A.10. Other uses for PLIFE.

PLIFE is a framework which can be used by other researchers interested in several areas of study. The study of numerical techniques for solving two dimensional problems is an area of active research. Different numerical techniques are easy to test in PLIFE because they are easy to incorporate. PLIFE can be used to study numerical techniques for regular, irregular, and dynamic grids.

The primary purpose of the development of PLIFE, was to have a tool for the study of load balancing techniques for adaptive grid programs. The routine which moves cells to different processors has a simple interface. The only required input to the routine is a list of cells to move. Once the cells are moved, PLIFE finds neighbors as required and sets up the communications.

Another area of active research is the development of three dimensional adaptive grid programs, especially hydrodynamics applications. Three dimensional adaptive grid programs use very large amounts of data and are very time consuming to run. Thus, it is desirable to run these programs as parallel applications.

PLIFE can be modified to run as a three dimensional adaptive grid program. Now, each cell represents a single data value. PLIFE can easily be modified so that each cell would contain a column of data. Grid adaptation in the first two dimensions would remain unchanged. In the third dimension, PLIFE could use an allocatable array to represent the data. The modifications to the framework would be small. The communications setup routines would need to be modified slightly. The length of the allocatable array would need to be considered when moving data between processors.

A.11. Customizing PLIFE for a particular application.

PLIFE was designed to be easily modified to enable the design of custom adaptive grid programs based on a five point stencil. Three principal modifications need to be made to PLIFE to customize it for a particular application. The data structure *value_type* may be modified. The operators +, -, *, and, / may need to be overloaded to define what

it means to apply these operators to a *value_type*. Finally, the mathematics of the simulation must be added as a collection of one or more subroutines.

The first two modifications to PLIFE occur in the Fortran 90 module *numz.* Among other things, *numz* contains the definition for the data types *cell_type* and *value_type*. All cells are of type *cell_type*. Cells contain a field, *value*, which is of type *value_type*. Type *value_type* is a sequence of real values. A typical definition of *value_type*, given in Fortran 90 syntax is:

```
type value_type
   sequence
   real(b8) pres_temp, temp_val, input
end type value_type
```

where b8 is defined as

```
integer, parameter:: b8 = selected_real_kind(p=12).
```

This requires the integer to have at least 12 decimal digits of precision and implies real(b8) is a 64 bit real on most machines.

The definition of *value_type* given above was used for a simulation of a heat flow problem. *Pres_temp* is the temperature, *temp_val* is a temporary value and *input* is the value for the heat source at a particular location.

The module *numz* also contains the routines which overload the operators for *value_type*. These operators may be used as needed in the routines which perform the

mathematics of the simulation.

The main mathematics of the simulation is contained in the subroutine *calculate*. *Calculate* must be written to accept, through the module *global*, a linked list of cells that a given processor has the responsibility of updating. When *calculate* is called, cells already know the values of their neighbors. *Calculate* also knows the size of cells, that is, Δx and Δy for cells at a given level of the tree structure. The data structure for each cell also contains the depth of the cell in the tree structure. Based on this available information, the subroutine *calculate*, updates the values of the cells held on a particular processor.

The subroutine *bc* updates the ghost cells, which contain boundary conditions. *Bc* receives as input the x and y location of a cell, which boundary it is on, (top, bottom, left, right), and a pointer to the cell for which it is a neighbor. This last piece of information is useful if reflective or transmissive boundary conditions are being used.

To set initial conditions, the routine *set_force* is called. *Set_force* takes as input a cell and sets the fields of the cell *value_type* data structure. *Set_force* calls the routine *centers* which finds the x and y coordinates for the cell. It then calls the routine *force* to set the fields of value based on cell location. To customize PLIFE, the subroutine *force* would be modified.

The subroutines *pflagsplit* and *pflagjoin* are used to determine which cells are to be split into four cells and which cells are to be joined to form larger cells. These routines

receive a linked list of cells to test. The criterion for splitting or joining cells is model dependent. If the criterion is met then a pointer to the cell is added to a linked list.

The load balancing routine, *pflagmove*, uses as input, a complete description of the grid tree structure. It must produce an array of cells to be moved, and the processor to which they are moved. After pflagmove is called, routines are called to move the cells and adjust the communication.

A.14. Lessons learned from the construction of PLIFE.

In implementing various numerical techniques within PLIFE, the robustness of the framework for adaptive grid programs was demonstrated. Significantly different algorithms were implemented without changing the underlying framework. Specifically, both a finite difference technique and a finite volume algorithm were used without restructuring the framework.

Although it is meaningful to show the robustness of the framework, this was not the most important accomplishment to come out of the change to new numerical techniques. Important lessons were learned about the framework construction. These are described next.

The lessons learned deal primarily with the basic data structures of the framework and in turn the data structures which represent the calculation grid.

Although it is possible to emulate a eight or nine point stencil using a basic five point stencil, a more general scheme would be beneficial. An N point stencil maybe

useful with a five point stencil as a special case. This removes the concept of a single top/bottom/left/right neighbor for a cell and replaces it with the concept of a collection of surrounding cells. This allows for, among other things, two or more neighbors to a side. Right now this can be emulated with a computation and communication penalty. The fundamental change of the framework data structure would be from a quad tree to an arbitrary tree structure.

Currently, the fundamental "entity" of the framework is a cell. It may be useful to have as the fundamental entity of the framework, an edge instead of a cell. An edge would then have a pointer to two or more cells and a square cell would be pointed to by four edges. Again this appears to be a more general scheme than the rectangular cell based scheme. Also, many of the newer numerical techniques, including surprisingly, the finite volume techniques, are fundamentally edge based. They may be easier to implement from the edge based framework.

Purging of the quad tree is useful. In the calculations done for this work, some cells in the forest of quad trees will never be changed or referenced after the initial grid setup. The *value_type* data structures associated with these cells are nullified. This decreases calculation and communication time by nearly 50%.

One of the reasons for using Fortran 90 as the source language for PLIFE is more of a language research related issue. It was hoped to press the language to see if and where it fell short of C and C++ in abilities to create dynamic memory, arbitrary data structure programs. In comparing to these languages, the only feature which is glaringly absent from Fortran 90 is the C++ template. The addition of a template feature would ease the

creation of functions for arbitrary data types.

A.15. Future directions For PLIFE.

There are two possible directions of development for PLIFE in the future. The first direction is related to where PLIFE can be taken without major changes to the framework. The second direction concerns changes to the data structures used in the framework.

Without any major changes, other numerical techniques can be incorporated into PLIFE. These techniques may include the full Navier-Stokes equations for fluid flow. Multimaterial, radiation hydrodynamics could be done. Heat flow problems have been simulated.

Other load balancing techniques could be tested within PLIFE. The diffusion methods of Wheat (1992) could be implemented. Various graph partitioning methods would fit well within the framework.

As discussed above, PLIFE could be used to solve three dimensional problems. In this case each cell would contain a vector of data. This would require a minor modification. The communication routines would need to be changed to allow for variable length data items because different columns could contain a different number of points. We note that for the three dimensional calculations, the quad tree structure of the grid would not need to be changed. In particular, it would not grow. This is a nice feature. For the three dimensional simulation, the relative memory overhead of representing the grid decreases.

Many improvements could be made in PLIFE if the data structures were changed.

This is another area of future work.

PLIFE could be rewritten to allow an arbitrary number of sides for a cell. This would allow for triangular cells which are used in some hydrodynamics simulations. One of the popular numerical techniques for hydrodynamics assumes octagonal cells.

Even with square cells, allowing an arbitrary number of sides would be useful. Assume we had a square cell. On one side we have a collection, say, N cells as neighbors. The side of the cell could be divided into N segments or pseudo sides. Each pseudo side would have its own single neighbor. This is a very clean and efficient way to handle cells with an arbitrary number of neighbors to a side.

There is another closely related way of handling neighbors and edges which maybe worth while exploring. We could allow an arbitrary number of neighbors to an edge. How would this be useful? We have three examples.

Assume we have a cell which borders on one side, two cells of half its size. Then that edge could have two neighbors.

This would eliminate an inefficiency in the present version of PLIFE. To handle the condition mentioned above, we do the following. If a cell has on its border two cells of half its size, we pass the two values up the grid structure. The parent of the two cells now has their values. The cell to the side then receives the values from the larger cell. The act of passing up values to parents is time consuming. If we allowed two or more neighbors to a side, the pass up procedure could be avoided.

Assume we wanted to do a calculation with square cells but a nine point stencil. That is, a cell wants values from the top, bottom, left, right cells, and the four cells on its corners. We could have two neighbors to a side, the ones directly adjacent, and each side would have a corner as a neighbor.

Finally, assume we want to do a calculation with a highly accurate finite difference algorithm. These algorithms use a larger number of points to perform a calculation. We may want to use in one direction 5 cells, the center cell, the cells on each side, and the cells one step further away.

It maybe useful to remove the feature that each processor holds an image of the entire grid. The grid description can use a large amount of memory. The data structure describing a single cell, without the real data, takes 12 pointers, 2 integers, and 2 logicals or 60 bytes. Assuming 16 base cells and 7 levels of splitting, the grid description uses over 31 Mbytes. For virtual memory systems most of the description will be put to disk. If there is an operation which requires each processor to view all of the grid, then there can be a significant time penalty to bring this description in from disk.

Appendix B. Numerical solution of Euler's equations of gas dynamics.

B.1. Introduction to Euler's equations.

Euler's equations of gas dynamics were used as the basis of the simulations for testing the load balancing and communication minimization algorithms. Euler's equations are a set of partial differential equations which predict the flow of fluids and gas. The equations predict energy, pressure, density, and velocity as a function of time. It is assumed that the gas or fluids are compressible and they flow without viscosity.

Euler's equations are used for the simulation of many types of physical phenomena. The simulation of weather is one area where Euler's equations are applicable. Calculations of flows around two and three dimensional objects, such as aircraft, can also be done. Simulations of explosions are often performed by assuming that the explosive device is a ball of hot gas. If an explosive device has sufficient energy, explosions in solids can also be simulated because the solids will act as gases under very high pressure.

An important classic, one dimensional problem, is the simulation of a shock tube. It is assumed that a long tube is divided into two sections. The sections are separated by a membrane which can be instantaneously removed. The gasses on each side of the membrane are at different temperatures, densities and pressures. At time equals zero the membrane is removed. Euler's equations predict the values of the variables as a function of time along the length of the tube. This problem is often called Sod's (1978) problem after the author who used it to study various numerical techniques. Its importance is that it is often used to study numerical techniques.

There is a two dimensional extension to this problem which is often used for testing numerical routines. Assume, that at time equals zero, we have a cylinder or ball of high pressure, temperature, and density gas enclosed in a membrane. The membrane is removed and the evolution of the system is followed. This two dimensional extension of Sod's problem is used to test the load balancing and communication minimization algorithms in this thesis.

Let ρ be density of the gas or fluid, u be the velocity in the x direction, v be the velocity in the y direction. P is pressure and E is the total energy per unit volume for the gas. In Euler's equations all of these quantities can be dependent on t, x and y. In two dimensions and rectangular coordinates, Euler's equations are of the form:

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathbf{F}}{\partial x} + \frac{\partial \mathbf{G}}{\partial y} = 0$$

where

$$U = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ E \end{bmatrix}, \quad F = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ Eu + pu \end{bmatrix}, \quad G = \begin{bmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ Ev + pv \end{bmatrix}.$$

E, the total energy, is the sum of the kinetic energy and e, the internal energy of the fluid. It can be written as:

$$E = \frac{1}{2}\rho\left(u^2 + v^2\right) + \rho e.$$

Finally, it is assumed that the internal energy is a known function of the pressure and density:

$$e = e(p, \rho).$$

This relationship is known as the equation of state for the gas and it depends on the particular gas which is to be simulated. The equation of state for the internal energy of an ideal gas is of the form:

$$e = \frac{p}{(\gamma - 1)\rho}$$

with γ=1.4 for air.

B.2. Conservation of energy and mass.

The Euler system of equations is in a class known as conservation equations. Specific quantities involved in the simulation are conserved, that is, the sum over all space of the conserved quantities is a constant in time. For the Euler system, the conserved quantities are energy, mass, and momentum.

It is possible to simulate sources and sinks with conservation equations. A energy source is a location where energy is added to the system being simulated. A sink is where it is removed. If there are sources and sinks, then the conservation requirement is modified to include their contribution.

The requirement that the quantities are conserved is useful in checking the correctness of numerical algorithms for solving the system of equations. Numerical solutions to Euler's equations must be conservative. As time progresses in a simulation, the sum of all of the conserved quantities over the region of calculation must be a constant. If they are not, there is an error. Again sources and sinks are allowed but must be considered in

the accounting.

B.3. Numerical method for Euler’s equations.

Often, the first step in the numerical solution of Euler's equations is to divide the region of interest into a collection of cells. In two dimensions the cells are often rectangles or triangles. A cell can be thought of a finite volume surrounded by a collection of edges. Two or more cells share an edge.

One way to ensure conservation in numerical simulations is to use an edge based scheme. We calculate the flux of the quantities across the edges as a function of time. Assume two cells share an edge. At $t=t_n$ we are given the values of variables in the cells. The values are used to calculate the flow rate across the edge shared by the two cells. These flow rates are then used to calculate the values of the variables at $t=t_{n+1}$. The values at $t=t_{n+1}$ are equal to the values at $t=t_n$ plus the flow rates across the edges times the time step size. Flow rates across an edge can be negative. Specifically, for two cells that share an edge, the flow rate across that edge into one cell is the negative of that for the other cell. Conservation is ensured because quantities, such as mass, which leave one cell are sent to a neighboring cell. Numerical methodologies which treat the cells as a volume surrounded by a collection of edges are often called finite volume schemes.

The algorithms which were used to numerically solve Euler’s equations are described next. The algorithms are an adaptation of the three dimensional finite volume scheme developed by Vijayan and Kallinderis (1994).

These algorithms must calculate the values of density, momentum, and energy at

$T=t_{n+1}$ from values at $T=t_n$. This is a multistep process. Estimates are made of $U=[\rho \quad \rho u \quad \rho v \quad E]$ along the edges of the cells at $T=t_{n+1/2}$, at a half time step. With the updated values of U, we calculate the fluxes across the cell boundaries. Finally, using the fluxes we calculate U at $T=t_n$. As discussed in the reference given above, performing the calculation using half time steps is similar to the methodology used in the two step Lax-Wendroff finite element scheme. Half time stepping is done to ensure second order accuracy in time.

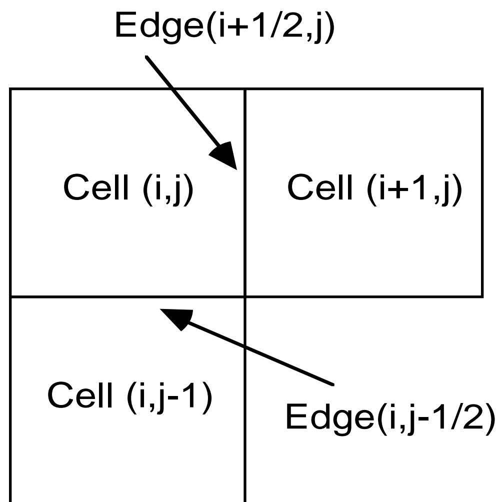


Figure B.1. Edges between cells of the PLIFE grid.

Consider the edge between cells (i,j) and (i+1,j). We designate values on this edge between the two cells, at time $t=t_{n+1/2}$, using the subscripts (n+1/2,i+1/2,j).

Let $\mu = \rho u$ be the momentum in the x direction and $\eta=\rho v$ be the momentum in the other direction.

The values of U at the right edge of cell (i,j) are determined from :

$$\rho_{n+1/2,i+1/2,j} = \frac{1}{2}\left(\rho_{n,i,j} + \rho_{n,i+1,j}\right) - \frac{\Delta t}{2\Delta x}\left(\mu_{n,i+1,j} - \mu_{n,i,j}\right)$$
$$\mu_{n+1/2,i+1/2,j} = \frac{1}{2}\left(\mu_{n,i,j} + \mu_{n,i+1,j}\right) - \frac{\Delta t}{2\Delta x}\left(\left[\mu_{n,i+1,j}u_{n,i+1,j} + p_{n,i+1,j}\right] - \left[\mu_{n,i,j}u_{n,i,j} + p_{n,i,j}\right]\right)$$
$$\eta_{n+1/2,i+1/2,j} = \frac{1}{2}\left(\eta_{n,i,j} + \eta_{n,i+1,j}\right) - \frac{\Delta t}{2\Delta x}\left(\left[\eta_{n,i+1,j}u_{n,i+1,j}\right] - \left[\eta_{n,i,j}u_{n,i,j}\right]\right)$$
$$E_{n+1/2,i+1/2,j} = \frac{1}{2}\left(E_{n,i,j} + E_{n,i+1,j}\right) - \frac{\Delta t}{2\Delta x}\left(\left[E_{n,i+1,j} + p_{n,i+1,j}\right]u_{n,i+1,j} - \left[E_{n,i,j} + p_{n,i,j}\right]u_{n,i,j}\right)$$

Values at the left edge of cell (i,j) are defined similarly but with i replaced by i-1.

For the top and bottom edges the equations are of the form:

$$\rho_{n+1/2,i,j+1/2} = \frac{1}{2}\left(\rho_{n,i,j} + \rho_{n,i,j+1}\right) - \frac{\Delta t}{2\Delta x}\left(\eta_{n,i,j} - \eta_{n,i,j+1}\right)$$
$$\mu_{n+1/2,i,j+1/2} = \frac{1}{2}\left(\mu_{n,i,j} + \mu_{n,i,j+1}\right) - \frac{\Delta t}{2\Delta x}\left(\left[\mu_{n,i,j+1}v_{n,i,j+1}\right] - \left[\mu_{n,i,j}v_{n,i,j}\right]\right)$$
$$\eta_{n+1/2,i,j+1/2} = \frac{1}{2}\left(\eta_{n,i,j} + \eta_{n,i,j+1}\right) - \frac{\Delta t}{2\Delta x}\left(\left[\eta_{n,i,j+1}v_{n,i,j+1} + p_{n,i,j+1}\right] - \left[\eta_{n,i,j}v_{n,i,j} + p_{n,i,j}\right]\right)$$
$$E_{n+1/2,i,j+1/2} = \frac{1}{2}\left(E_{n,i,j} + E_{n,i,j+1}\right) - \frac{\Delta t}{2\Delta x}\left(\left[E_{n,i,j+1} + p_{n,i,j+1}\right]v_{n,i,j+1} - \left[E_{n,i,j} + p_{n,i,j}\right]v_{n,i,j}\right)$$

Next, from the values of U at the edges, we find velocities by dividing momentum by mass. We use these values to find the fluxes as follows:

$$F_{n+1/2} = \begin{bmatrix} \rho_{n+1/2}u_{n+1/2} \\ \rho_{n+1/2}u_{n+1/2}{}^2 + p_{n+1/2} \\ \rho_{n+1/2}u_{n+1/2}v_{n+1/2} \\ E_{n+1/2}u_{n+1/2} + p_{n+1/2}u_{n+1/2} \end{bmatrix}, \quad G_{n+1/2} = \begin{bmatrix} \rho_{n+1/2}v_{n+1/2} \\ \rho_{n+1/2}u_{n+1/2}v_{n+1/2} \\ \rho_{n+1/2}v_{n+1/2}{}^2 + p_{n+1/2} \\ E_{n+1/2}v_{n+1/2} + p_{n+1/2}v_{n+1/2} \end{bmatrix}.$$

Let ω be the volume of a cell. The density at $T=t_{n+1}$ for the a cell is given by:

$$\rho_{n+1,i,j} = \rho_{n,i,j} - \frac{\Delta t \Delta x}{\omega}\left(\rho_{n+1/2,i,j+1/2}u_{n+1/2,i,j+1/2} - \rho_{n+1/2,i,j-1/2}u_{n+1/2,i,j-1/2}\right) - \frac{\Delta t \Delta y}{\omega}\left(\rho_{n+1/2,i+1/2,j}u_{n+1/2,i+1/2,j} - \rho_{n+1/2,i-1/2,j}u_{n+1/2,i-1/2,j}\right)$$

The first line of this expression is the value from the previous calculation cycle. The second line is the update due to the flux across the top and bottom of the cell and the third line is the flux across the vertical edges of the cell. Δx is the length of the segment across which material is flowing in the y direction and Δy is the length of the segment across which material is flowing in the x direction.

It is possible for a cell to have as its neighbors on any side two cells of half its size. When this happens, the flux across the edge is the sum of the fluxes from the two cells. The fluxes are calculated in nearly the same way as given above. What differs is the estimate of the values of U at the half time step at the edge between the two cells.

If Δx is the size for the bigger cell then the values of U at the edge of a cell are:

$$\rho_{n+1/2,i+1/2,j} = \frac{1}{2}\left(\rho_{n,i,j} + \rho_{n,i+1,j}\right) - \frac{\Delta t}{\Delta x}\left(\mu_{n,i+1,j} - \mu_{n,i,j}\right)$$

$$\mu_{n+1/2,i+1/2,j} = \frac{1}{2}\left(\mu_{n,i,j} + \mu_{n,i+1,j}\right) - \frac{\Delta t}{\Delta x}\left(\left[\mu_{n,i+1,j}u_{n,i+1,j} + p_{n,i+1,j}\right] - \left[\mu_{n,i,j}u_{n,i,j} + p_{n,i,j}\right]\right)$$

$$\eta_{n+1/2,i+1/2,j} = \frac{1}{2}\left(\eta_{n,i,j} + \eta_{n,i+1,j}\right) - \frac{\Delta t}{\Delta x}\left(\left[\eta_{n,i+1,j}u_{n,i+1,j}\right] - \left[\eta_{n,i,j}u_{n,i,j}\right]\right)$$

$$E_{n+1/2,i+1/2,j} = \frac{1}{2}\left(E_{n,i,j} + E_{n,i+1,j}\right) - \frac{\Delta t}{\Delta x}\left(\left[E_{n,i+1,j} + p_{n,i+1,j}\right]u_{n,i+1,j} - \left[E_{n,i,j} + p_{n,i,j}\right]u_{n,i,j}\right)$$

The difference between these equations and the previous set is that the term $\frac{\Delta t}{2\Delta x}$ has been replaced with $\frac{\Delta t}{\Delta x}$. This has the effect of cutting Δx in half. From the physical point of view, the effect is as if the larger cell had been bisected into four identical cells of the half the size.

There is a difficulty solving Euler's equations numerically. In the presence of large shocks, some numerical solutions, including the one outlined above, tend to show nonphysical oscillations. A standard method for dealing with these oscillations is to introduce an artificial viscosity into the problem. This tends to reduce the sharpness of the peaks in the calculated solution but removes the oscillations. Following the methodology described by Lapidus (1967), artificial viscosity was added to the routines described above. The development of numerical routines which more accurately capture shock peaks and do not have nonphysical oscillations is an area of active research, and beyond the purpose of this research. However, once better solution techniques are further developed, they can be readily incorporated into this research. They are conjectured to not impact the control aspect of this work, namely, the effectiveness of the genetic algorithm to seek out load balanced configurations.

Appendix C. Genetic Algorithm description.

C.1. Overview.

This chapter describes the genetic algorithm framework, CHUCK. CHUCK was used within the PLIFE framework to find mappings of cells to processors. The goal of the mapping is to find distributions of cells which produce good load balance and have a low communication cost.

We first have a review of the basic algorithm for a serial GA. Next we have a description of the parallel implementation of CHUCK. A section describes how the GA was used within PLIFE. A description is given of how the initial population for the GA is created. An introduction is given to the fitness function used within the GA.

C.2. The serial genetic algorithm.

The serial algorithm for a GA is straight forward. A potential solution for a particular optimization problem is represented by some string or vector. The vectors are called chromosomes and the individual entries in the vector are called genes. For this work, the vector contains integers. There is a fitness function which evaluates the potential solutions. Good solutions to the problem are given a high fitness. The algorithm proceeds as follows.

(1) Generate a collection of potential problem solutions. This is normally done in some random fashion. This collection is called the population.

(2) Repeat until done.

Find the fitness of the members of the population.

Sort the population so that the best members come to the top.

Discard the bottom half of the population.

Allow the top half of the population to reproduce replacing the old population.

Allow potential mutations in the population.

"Repeat until done" normally means repeat for a given number of iterations, repeat until there is a member of the population which is of a given fitness, or repeat until the population stagnates. We can have early termination of the algorithm if a member of the population reaches the desired fitness.

How do solutions reproduce? There are several methods possible. In this GA we do the following. Given two solution strings, split them at some arbitrary and random location. Recombine the pieces. An example follows.

P1= "abdefg", P2= "1234567". Select an integer between 1 and the length of the string, say 3. Split the strings after the third character. The new string produced, P3, has as its characters the beginning of P1 and the end of P2. P3= "abc4567".

There are many variations on the algorithm given above. One of the more popular is to use a roulette method of determining which members of the population reproduce. See Fogel (1996). With this methodology, the probability that a member of the population will reproduce is proportional to its relative fitness, that is, relative to the rest of the population. The significant difference in this methodology is that it allows the possibility for a member of the population to reproduce even if its fitness value is not in the top half. The roulette method of population selection was used for this research effort.

C.3. Mutations in the GA.

Traditionally, there has been two methods used for doing mutations of members of the population. CHUCK can use the first method described below. The second method, which is not used by this GA because it is too time consuming, is also described. A third, alternate method, which is used by CHUCK is also described.

For the first method of mutation, a probability that a member's chromosome will suffer a mutation is given as an input parameter. The mutation algorithm proceeds as follows.

Let P be the probability that a member suffers a mutation.

```
DO for every member of the population
  Call a random number generator
  IF (the random number is < P) THEN
          Pick at random, a gene from the chromosome
          Change that gene to a random number
  ENDIF
ENDDO
```

This method runs in time proportional to the size of the population.

For the second method the input probability is used differently. It describes the probability that an individual gene from the chromosome will "successfully" mutate. The mutation algorithm proceeds as follows.

```
Let P be the probability that a gene successfully mutates.
DO for every member of the population
  DO for every gene in the chromosome of the member
          Call a random number generator
          IF (the random number is < P) THEN
```

Mutate the gene
ENDIF
ENDDO
ENDDO

The difference between these two methods is that for the second method a random number generator is called for every gene of every chromosome while in the first it is called only twice for every chromosome. This second method is time consuming because it runs in time proportional to the population size times the size of the vector describing the members. This method is used by most GA developers and described by Fogel (1996).

There is a third method, developed for this research effort, which approximates this second method but runs in time proportional to the population size times the number of genes, times the probability P. For small P this is much faster. We look at the method described above in more detail to derive this third method.

Assume there are N genes in a chromosome. We can view the mutation of a gene as a success of a trial of an experiment. The probability, P, that a gene will be "successfully" mutated is a constant value for all genes. The probability that a gene will mutate is independent of a neighboring gene mutating.

The paragraph given above describes a binomial experiment. See Helstrom (1984). The probability that X number of genes out of N will mutate is given by the binomial distribution. With q=p-1, the probability distribution for X of $b(x;n,p) = \binom{n}{x} p^x q^{n-x}$.

Functions are available which produce binomial distributed random deviates as described in Press, Flannery, Teukosky, and Vetterling (1986). Using such a function, we have the third methodology for mutation.

```
DO for every member of the population
  Find X, a binomial distributed random number in the range n   X   0
  DO for i =1 to X
          Pick a gene from the chromosome at random
          Mutate that gene
  ENDDO
ENDDO
```

What is the average run time for this algorithm? This can be found by knowing the average value for X. For the binomial distribution, we have $\overline{X} = np$. The average run time is proportional to population size times the chromosome size times the probability that a gene mutates.

C.4. The parallel operation of the GA.

CHUCK is a framework for a parallel genetic algorithm. There are two fundemental modes of operation of chuck. In the first mode the master processor holds the population and sends requests for fitness function evaluations to the slave processors. In the second mode, the population is distributed across all processors with each processor holding its own subpopulation. Each processor does the fitness function evaluation for its subpopulation and each does its own reproduction. This is the mode which was used for this research effort. If the size of the total population is divisible by the number of processors, then each processor holds the same number of members of the population.

There are many interesting variations on this parallel method of operation which

have been used in the past. See for example Levine (1994). CHUCK takes many of the concepts for the parallel operation of genetic algorithms and puts them into a single framework. CHUCK also adds a fast mutation methodology, which may not be new but no references to it have been seen. The following are supported by the genetic algorithm used for this research.

In variation 1 we allow each processor to work independently for "I" generations. Then, we do a global parallel sort and then redistribute the top half of the population and continue. This variation allows for tightly to loosely coupled algorithms depending on the value of I. We can allow separate and independent evolutions to occur and in the end take the best of the independent populations. With tight coupling, I=1, we can simulate a sequential algorithm.

In variation 2 we assume a rectangular topology for the processors working on the the porblem. We allow each processor to work independently for "J" generations. Then we do an exchange between neighbnors of some portion of the population left-right and top-bottom. The amount of the population exchanged between neighboring processors is determined by an input parameter. There is a concern that this method is somewhat artificial, in that, we are forcing a topology onto a problem which does not have a topology of its own.

In variation 3 we assign an aggression factor to each processor. We then allow the processors to work independently for "K" generations. Next, the aggressive processors force a portion of their population onto the other processors. It has been found that this allows a degree of randomization which helps prevent stagnation of the population.

Variation 4 is a combination of 1, 2, and 3. The threshold for the amount amount of each parallel method is controlled by the input parameters.

For this research effort, a combination of methods 1 and 2 is used. The aggressive takeover algorithm is controled by an input parameter. For this effort it was turned off. The effect of the aggressive takeover form of parallelism is left as an area for future research.

C.5. Use of the genetic algorithm.

The PLIFE framework can be used to create adaptive grid simulations. During a run of a program using the PLIFE framework, regions of the grid can become finer zoned. Processors that hold regions of the grid which become finer zoned become more heavily loaded. PLIFE has the capability to move such cells to different processors in order to maintain a good load balance. For this research effort, a GA was used to determine what cells are moved.

The GA runs in parallel on all nodes used for the calculation. The GA is given, as input, a collection of cells which can be moved between processors. The GA outputs a mapping of cells to processors. As would be expected, the fitness function for the GA has a maximum value for distributions of cells to processors which have good load balance and low communication. The fitness function is described below.

C.5.1. Pflagmove called at various times during a run.

Pflagmove is the routine that PLIFE calls to flag cells to be moved between processors.

For this research effort, Pflagmove, in turn, calls the genetic algorithm, CHUCK.

Initially, for the run of the PLIFE program, the load is balanced. After running a while, an imbalance is created. The GA is called. Hopefully, after the cells are moved, the program is in balance again. The PDE solver within PLIFE continues to run, becomes unbalanced, and the GA is called. This cycle continues until the program finishes.

For the experiments described in chapters 4 and 5, the GA was called at irregular numbers of cycles. It was called at calculation cycle numbers, such that, every time it was called it had approximately 100, (100 ± 10), clusters of cells to distribute.

C.5.2. How the initial population is created.

The routine *pflagmove* is used to flag the cells to move between processors. *Pflagmove*, in turn, calls the genetic algorithm routine CHUCK. Before CHUCK is called, the routine *find_req* is called. *Find_req* determines a goal of the number of cells to be moved to various processors to balance the load. *Find_req* is discussed in more detail in section 4.5.

Find_req has, as one of its inputs, the total number of clusters which are available to be moved between processors. *Find_req* returns a vector of length equal to the number of processors. The vector is called *request.* It contains the fraction of the total number of clusters that each processor needs to balance the load. This vector is used to create the initial population for the GA.

The chromosome for the members of the population are vectors of integers. In

particular, the integers are processor numbers for the mapping of cells to processors. The length of the chromosome, the number of genes, is equal to the total number of cells which are available for redistribution.

The initial population is created using a pseudo random probabilistic function. The pseudo random probabilistic function is designed such that, averaged over a large population, the percentage of cells which are initially assigned to processor N is equal to the Nth entry in the vector *request*, the vector produced by *Find_req*. That is, the number of genes equal to N is, on average, equal to the Nth entry in the vector *request*.

C.5.3. Fitness function.

Mansour and Fox have used a hybrid GA to statically assign sections of a grid to processors to obtain good load balance and low communication. See Mansour and Fox (1991). Their work did not address adaptive grid refinement.

They defined an objective function of the form $c_1 \sum_p N^2(p) + c_2 \sum_p d(p)$, where N(p) is the load on processor P and d(p) is the communication required by the processor. This fitness is high when the objective function is low.

Following their lead, the fitness function used for this research was similar. It was quadratic in load distribution and linear in communication.

The fitness function was of the form $fitness = c_1 Fitness_{load} + c_2 Fitness_{communication}$, with c1 + c2 =1. For convenience, $\text{Fitness}_{\text{load}}$ and $\text{Fitness}_{\text{communication}}$ are in the range of 0 to 1.

This gives a fitness function also in the range of 0 to 1. c1 and c2 are weighting parameters that assign a relative importance in the fitness function of getting a good load balance and low communication.

We want $Fitness_{load}$ to be a maximum for distributions of cells to processors which match the *request* vector described above.

Let Z be the total number of cells which are being considered for movement, R be the request vector, and N be the number of processors. The fitness function is defined in terms of a cost function. The cost function for a particular load distribution, L, is

$C = \sum_{n=1}^{N} \left(\frac{Z}{N} + \left| L_n - R_n * Z \right| \right)^2$, where L_n is the load on a processor.

This function has two of the desired properties. It is quadratic in load distribution and it has a minimum for a load distribution which matches the requested distribution. However, it is not necessarily in the range 0 to 1. The minimum cost of a load distribution is $C_{\min} = \sum_{N} \left(\frac{Z}{N} \right)^2 = \frac{Z^2}{N}$.

To obtain a value between 0 and 1, we scale the function given above so that when we have $C=C_{min}$, the $Fitness_{load} = 1$ and when $C=C_{max}$, the $Fitness_{load} = 0$.

To do this we need an estimate of the maximum cost of the worst load distribution. The maximum cost is a function of the request vector and could be found by solving a

optimization problem of the following form. With R_n known and $\sum_{n=1}^{N} R_n = 1$, find the

maximum value of $C = \sum_{n=1}^{N} \left(\frac{Z}{N} + |L_n - R_n * Z| \right)^2$ subject to the constraints that $\sum_{n=1}^{N} L_n = Z$.

Instead of solving this optimization problem for every new *request* vector, we make the following assumptions and approximation. Assume that R(1)=0 and R(N>1)=Z/(N-1). A bad distribution for this particular *request* vector would be L(1)=Z and L(N>1)=0. This distribution gives a cost of $C = Z^2 \frac{3N^2 - N - 1}{N(N-1)}$ or about $3Z^2$. As described above, we start the GA with mappings that are, on average, balanced. We do not expect to see distributions as bad as the one just described. So we use as an approximation of the maximum cost of a distribution Z^2 instead of $3Z^2$.

This gives the result for $\text{fitness}_{\text{load}}$ of

$$\text{Fitness}_{\text{load}} = \frac{1}{C_{\min} - C_{\max}} \sum_{n=1}^{N} \left(\frac{Z}{N} + |L_n - R_n * Z| \right)^2 - \frac{C_{\max}}{C_{\min} - C_{\max}}$$

or

$$\text{Fitness}_{\text{load}} = \frac{N}{Z^2(1-N)} \sum_{n=1}^{N} \left(\frac{Z}{N} + |L_n - R_n * Z| \right)^2 - \frac{N}{1-N}.$$

This fitness function has the properties we desire, including values in the range of 0

to 1 for C in the range Z^2 to Z^2/N.

The communication cost, $Cost_{communication}$, is simply a mapping of the amount of communication required for a distribution to the range 0 to 1. Again, if the communication is a minimum then $Cost_{communication}$ is 0 and $Fitness_{communication}$ is 1. If the communication is a maximum then $Cost_{communication}$ is 1.

References

Adams, J., Brainerd, W., Martin, J., Smith, B., Wagener, J. (1992). Fortran 90 Handbook. New York, N.Y. : McGraw Hill.

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. Proceedings of AFIPS Spring Joint Computer Conference 30, (pp. 483-485). Atlantic City, NJ: Paril.

Berger, M. (1982). Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations (Technical Report No. STAN-CS-82-924). Stanford, CA: Stanford University, Department of Computer Science.

Berger, M. and Jameson, A. (1985) Automatic Adaptive Grid Refinement for the Euler Equations. American Institute of Aeronautics and Astronautics 23(4), p. 561.

Berger, M. and Oliger J. (1984). Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations. Journal of Computational Physics. 53, 484-512.

Berger, M. On Conservation at Grid Interfaces. Journal of Numerical Analysis 24(5), p 967.

Bianchini, R. and Brown, C. (1993). Parallel Genetic Algorithms on Distributed-Memory Architectures (Technical Report No. 436). Rochester, NY: University of Rochester, Computer Science Department.

Celia, M. and Gray, W. (1992). Numerical methods for differential equations : fundamental concepts for scientific and engineering applications. Englewood Cliffs, N.J. : Prentice Hall.

Condor-admin (1997). Condor. Available: http://www.cs.wisc.edu/condor

Cray Research (1996). T3D Overview. Available: http://www.cray.com/PUBLIC/product-info/mpp/T3D_overview.html

Flynn, M. J. (1972). Some computer organizations and their effectiveness. IEEE Transactions on Computers, C-21, 948-960.

Fogel, D. (1996). Evolutionary computation, toward a new philosophy of machine intelligence. Piscataway, NJ: IEEE Press.

Garey, M., & Johnson, D. (1979). Computers and intractability, A guide to the theory of NP-completeness. New York, NY: W. H. Freeman.

Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Mancek, R., & Sunderam, V. (1994). PVM 3 users guide and reference manual. Oak Ridge, TN: Oak Ridge National Laboratory.

Gittings, M. (1992). SAIC's adaptive grie eulerian hydrocode. Numerical Methods Symposium. (p. 368). Menlo Park, California, USA: Defense Nuclear Agency.

Gropp, B., and Lusk, R. (1996). Portable MPI model implementation, Argon National Laboratory: by anonymous ftp from info.mcs.anl.gov in the directory pub/mpi.

Helstrom, C. (1984). Probability and Stochastic Processes for Engineers. New York, NY: Macmillan Publishing.

IBM *(1994).* IBM AIX Parallel Environment: Operation and Use Kingston, NY: IBM Corporation.

Kaiser, T. (1993, June). Scientific visulaization for adaptive hydrodynamic grids. Scientific Visualization. Symposium conducted at the Computers in Physics 93 Conference, Albuquerque.

Kidwell, M. (1993). Using genetic algorithms to schedule distributed tasks on a Bus-Based System. Proceedings of the 5th International Conference on Genetic Algorithms. (p. 368). Urbana, Illinois, USA: Morgan Kaufmann Publishers.

Kumar, V., Grama, A, Gupta, A., & Karypis, G. (1994). Introduction to parallel computing design and analysis of algorithms. Redwood City, CA: Benjamin/Cummings.

Lapidus A. (1967). A detached shock calculation by second-order finite differences. Journal of Computational Physics. 2, 154-177.

Lapidus, A. and Oliger J. (1967). A Detached Shock Calculation by Second-Order Finite

Differences. Journal of Computational Physics. 2, 154-177.

Larsson, J. (1997). CFD Online. Available: http://www.tfd.chalmers.se/CFD_Online

Leiss, E. and Reddy, H. Distributed Load Balancing: Design and Performance Analysis. W. M. Keck Research Computation Laboratory. Vol 5, 1989, pp. 205-270.

Leveque, R. (1997). Home Page. Available: http://www.amath.washington.edu/~rjl

Levine, D. (1994). A parallel genetic algorithm for the set partitioning problem (Technical report ANL-94-23). Argonne, IL: Argonne National Laboratory.

Maccabe, A. (1993). Computer systems architecture, organization, and programming. Homewood, IL: Irwin.

Mansour, M., & Fox, G, (1991). A hybrid genetic algorithm for task allocation in multicomputers. Proceedings of the Fourth International Conference on Genetic Algorithms, (p. 466). San Diego, California, USA: Morgan Kaufmann Publishers.

Message Passing Interface Forum. (1994). MPI: A message-passing interface standard (Technical report CS-94-230). Knoxville, TN: University of Tennessee, Computer Science Department.

Moret, B. and Shapiro, H. Algorithms from P to NP. Volume 2. To be published.

Muenetome, M., Takai, Y., & Sato, Y., (1994) A genetic approach to dynamic load

balancing in a distributed computing system. Proceedings of The First IEEE Conference On Evolutionary Computation. (p. 418). Orlando, Florida, USA: IEEE.

Muller, R., & Buffington, A. (1974). Real-time correction of atmospherically degraded telescope image through image sharpening. Journal of the Optical Society of America. 64(9), p. 1204.

Ozturan C., deCougny, H., Sheparard, M., and Flaherty, J. (1994). Parallel Adaptive Mesh Refinement and Redistribution on Distributed Memory Computers (Technical Report). Troy, NY: Rensselaer Polytechnic Institute, Scientific Computation Research Center.

Press, W., Flannery, B., Teukosky, S., & Vetterling, W. (1986). Numerical recipes: the art of scientific computing. Cambridge, England: Cambridge University Press.

Sod, G. A. (1978). A survey of several finite difference methods for systems of nonlinear hyperbolic conservation laws. Journal of Computational Physics. 27, 1-31.

Stoer J., & Bulirsch, R. (1984). Introduction to numerical analysis (2nd ed.). New York, NY: Springer-Verlag.

Stunkel , C., Shea, D., & Abali, B. (1994). The SP2 communication subsystem. Yorktown Heights, NY. IBM Thomas J. Watson Research Center.

Stunkel, C. , Shea, D., & Abali, B. (1994). Architecture and implementation of Vulcan. Proceedings of the 8th international Parallel Processing Symposium, Cancun, Mexico.

Tanenbaum, A. S. (1994). Distributed operating systems. Englewood Cliffs, NJ: Prentice Hall.

van de Goor, A. (1989). Computer Architecture and design. Workingham, England : Addison Wesley.

Varadarajan, R. and Hwang, I. (1994). An efficient dynamic load balancing algorithm for adaptive mesh refinement. (Technical Report). Rio Piedras, PR: University of Puerto Rico, Department of Mathematics.

Varadarajan, R., & Hwang, I. (1994). An efficient dynamic load balancing algorithm for adaptive mesh refinement (Technical Report). University of Puerto Rico & University of Florida.

Vijayan P. and Kallinderis Y. (1994). A 3D finite-volume scheme for the Euler equations on adaptive tetrahedral grids. Journal of Computational Physics. 113, 249-267.

Walpole, R. & Myers, R. (1978). Probability and Statistics for Engineers and Scientists, 2Ed. New York, NY: Macmillan Publishing.

Walshaw, C. and Berzins, M. (1992). Dynamic load balancing for PDE solvers on

Adaptive Unstructured Meshes. (Technical Report No. 92.32). Leeds, England: University of Leeds, School of Computer Science.

Walshaw, C. and Berzins, M. (1995). Dynamic load balancing for PDE solvers on Adaptive Unstructured Meshes. Journal of Numerical Analysis 7(1), p 17.

Watson, M. (1991). Common LISP modules, artificial intelligence in the era of neural networks and chaos theory. New York, NY: Springer-Verlag.

Wheat, S. (1992). A fine gained data migration approach to application load balancing on MP MIMD machines (Technical Report No.C S92-18). Albuquerque, NM: University of New Mexico, Department of Computer Science.

Williams, R. (1990). Performance of Dynamic Load Balancing Algorithms for Unstructured Mesh Calculations. (Technical Report No. C3p913). Pasadena, C: California Institute of Technology, Concurrent Supercomputing Facility.

Yiu, K., Greaves, D., Cruz, S., Saalehi, A., and Borthwick, A. Dynamic load balancing for PDE solvers on Adaptive Unstructured Meshes. Computers and Fluids 23(8), p 759.