ADAPTIVE MESH REFINEMENT FOR DISTRIBUTED PARALLEL ARCHITECTURES

by

Daniel James Quinlan

B. A., University of Colorado at Denver, 1987

A thesis submitted to the Faculty of the Graduate School of the University of Colorado at Denver in partial fulfillment of the requirements for the degree of Doctor of Philosophy Applied Mathematics 1993
This thesis for the Doctor of Philosophy degree by

Daniel James Quinlan

has been approved for the

Department of

Mathematics

by

[Signature]

Stephen F. McCormick

[Signature]

John Ruge

Date June 6th 1993
Quinlan, Daniel James (Ph.D., Applied Mathematics)

Adaptive Mesh Refinement for Distributed Parallel Architectures

Thesis directed by Professor Stephen F. McCormick

The purpose of adaptive mesh refinement is to match the computational demands to an application's activity. In a fluid flow problem, this means that only regions of high local activity (shocks, boundary layers, etc.) can demand increased computational effort, while regions of little flow activity (or interest) are more easily solved using only relatively little computational effort. A thorough exploitation of these techniques is crucial to the efficient solution of more general problems arising in large scale computation.

The fast adaptive composite grid method (FAC) is an algorithm that uses uniform grids, both global and local, to solve principally elliptic partial differential equations. However, FAC suffers in the parallel environment from the way in which the levels of refinement are treated sequentially. The asynchronous fast adaptive composite method, AFAC, and a new method, AFACx, eliminate this bottleneck of parallelism. In both AFAC and AFACx, individual refinement levels are processed in parallel. AFACx both generalizes AFAC and permits the use of more complex block structured mesh refinement required for self-adaptive mesh refinement. Although each level's processing may be parallelized in FAC, AFAC and AFACx may be much more efficiently parallelized. It is shown that, under most circumstances, AFAC is superior to FAC in a parallel environment. The theory for AFACx and an evaluation of its performance, including FAC and AFAC, is a significant part of this thesis; the remainder of the thesis details the object-oriented development of parallel adaptive mesh refinement software.

The development of parallel adaptive mesh refinement software is divided into three parts: 1) the abstraction of parallelism using the C++ parallel array class library P++; 2) the abstraction of adaptive mesh refinement using the C++ serial adaptive mesh refinement class library AMR++; 3) the serial application, specifically the single grid application, defined by
the user. Thus, we present a greatly simplified environment for the development of adaptive mesh refinement software in both the serial and parallel environment. More specifically, such work provides an architecture independent environment to support the development of more general complex software, which might be targeted for a parallel environment.

This abstract accurately represents the content of the candidate's thesis. I recommend its publication.

Signed [Signature]

Stephen F. McCormick
DEDICATION

To my wife Kirsten and son Carsten, without whose patient support the decade of undergraduate and graduate school would not have been possible.
ACKNOWLEDGEMENTS

Many people have helped me throughout my student years at CU Denver, and I want to thank a select few. I owe my most sincere thanks to Steve McCormick for his advice, support, and collaboration throughout my undergraduate and graduate education. Additional thanks to: Max Lemke for his collaboration on the software development, his thorough testing, and his insight into the singular perturbation example; Kristian Witsch, Max’s advisor, for advice on details of the singular perturbation example problem and for supporting Max in our the joint work; and Dinshaw Balsara for forcing me into the block structured problems, which eventually led to the AFACx algorithm. Additional thanks go to James Peery and Allen Robinson at Sandia National Laboratory for the loan of a SUN Sparc Station, which allowed for significant extension of the software development of P++ and AMR++, and my mother Pat Quinlan, a graphics artist, who helped prepare the figures. My deep appreciation goes to AFOSR for its early support of my work by way of a graduate research assistantship and a later summer fellowship.

Finally, I sincerely thank Tom Zang at NASA Langley for his support, which led to a three-year NASA fellowship that allowed me to pursue my doctoral research work. The motivation for much of this work was the problem that he proposed for my fellowship, which originally came from an SBIR grant that was funded by NASA Langley and for which I was the Principal Investigator.
CONTENTS

CHAPTER

1 INTRODUCTION ......................................................... 1
   1.1 Algorithm Development ........................................ 1
   1.2 Software Development .......................................... 5
   1.3 Problems and Future Work ..................................... 10
2 PARALLEL ALGORITHM DESIGN (THEORY) .......................... 12
   2.1 Overview of the FAC and AFAC Composite Grid Algorithms .. 12
   2.2 Notation and Definition of FAC, AFAC, and AFACx Algorithms 13
       2.2.1 FAC Algorithm ........................................... 15
       2.2.2 AFAC Algorithm ......................................... 16
       2.2.3 AFACx Algorithm ........................................ 18
           2.2.3.1 AFACx Motivation .................................. 18
           2.2.3.2 AFACx Definition .................................. 21
       2.3 AFACx Convergence Theory .................................. 22
3 PARALLEL SOFTWARE DESIGN ...................................... 33
   3.1 Introduction ..................................................... 33
   3.2 C Language Implementation of Parallel AFAC/FAC ............... 34
       3.2.1 Program Structure ...................................... 34
       3.2.2 Data Structures ......................................... 35
       3.2.3 Multigrid Solver ........................................ 35
       3.2.4 AFAC/FAC Scheduler ..................................... 36
       3.2.5 Grid Manager ............................................ 36
       3.2.6 Multilevel Load Balancer (MLB) ........................ 36
3.6.2 Block Structured Grids — Features and Restrictions .......... 78
3.6.3 Some Implementation Issues ................................ 81
3.6.4 Object-Oriented Design and User Interface .................. 83
3.6.5 Static and Dynamic Adaptivity, Grid Generation .............. 85
3.6.6 Current State and Related Work ................................ 87
3.7 Object-Oriented Design for Parallel Adaptive Refinement ........ 88

4 PARALLEL ADAPTIVE MESH REFINEMENT: RESULTS ............... 90
4.1 Introduction ...................................................... 90
4.2 Comparison of Convergence Factors ............................. 93
  4.2.1 Composite Grid Convergence Factors for Poisson Equation .. 93
  4.2.2 Convergence Factors for Singular Perturbation Problem .... 94
4.3 Performance of AFAC ............................................. 96
  4.3.1 Simple Example on iPSC/2: .................................. 97
  4.3.2 Simple Example on iPSC/1: .................................. 99
  4.3.3 Complex Example on iPSC/1: ................................ 99
4.4 Performance Comparison of FAC and AFAC Algorithms .......... 101
  4.4.1 Parallelization for Distributed Architectures ............... 101
    4.4.1.1 Parallel Multigrid Solver — Standard Grid Partitioning . 101
    4.4.1.2 Parallelization of FAC — Linear Single Level Grid Partitioning 103
    4.4.1.3 Parallelization of AFAC — Linear Multilevel Grid Partitioning . 103
  4.4.2 Comparison of Interprocessor Communication in FAC and AFAC 104
  4.4.3 Relative Performance Results ............................... 107
  4.4.4 Conclusions of FAC versus AFAC ............................ 108
4.5 Dynamic Adaptivity in Parallel Environment ..................... 110
  4.5.1 Multilevel Load Balancing .................................. 111
  4.5.2 Dynamic Movement of Local Refinement (Grid Tracking) .... 111
# FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Example composite grid with five levels.</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>Effect of overlapping communication with computation.</td>
<td>39</td>
</tr>
<tr>
<td>3</td>
<td>FORTRAN and C of C++ example code fragments</td>
<td>41</td>
</tr>
<tr>
<td>4</td>
<td>C Language example for block by block cache based execution</td>
<td>42</td>
</tr>
<tr>
<td>5</td>
<td>Distributed memory example code</td>
<td>44</td>
</tr>
<tr>
<td>6</td>
<td>Equivalent P++, object-oriented example code</td>
<td>45</td>
</tr>
<tr>
<td>7</td>
<td>Equivalent P++, object-oriented example code using explicit indexing</td>
<td>45</td>
</tr>
<tr>
<td>8</td>
<td>Equivalent P++, object-oriented example code</td>
<td>47</td>
</tr>
<tr>
<td>9</td>
<td>An example for VSG Update based on the Owner computes rule: A = B + C on 3 processors</td>
<td>59</td>
</tr>
<tr>
<td>10</td>
<td>The standard method of grid partitioning with overlap.</td>
<td>59</td>
</tr>
<tr>
<td>11</td>
<td>The object-oriented design of the P++ environment</td>
<td>63</td>
</tr>
<tr>
<td>12</td>
<td>C++ / M++ / P++ example code: McCormack (Hyperbolic) Scheme</td>
<td>66</td>
</tr>
<tr>
<td>13</td>
<td>The interaction the Overlap Update and VSG Update concepts for standard multigrid partitioning with coarse level agglomeration</td>
<td>73</td>
</tr>
<tr>
<td>14</td>
<td>The interaction the Overlap Update, VSG Update, and BSG Interface Update concepts for FAC and AFAC partitioning of a block structured locally refined grid</td>
<td>74</td>
</tr>
<tr>
<td>15</td>
<td>Example of a composite grid, its composite grid tree, and a cutout of two blocks with their extended boundaries and their interface</td>
<td>89</td>
</tr>
<tr>
<td>16</td>
<td>Simple composite grid</td>
<td>98</td>
</tr>
</tbody>
</table>
17 Regular single level strip partitioning of a 3-level composite grid structure onto 4 processors (FAC). ........................................ 121

18 Irregular multilevel strip partitioning of a 7-level composite grid structure onto 16 processors (AFAC). ........................................ 121

19 Complex composite grid problem with 40 patches. ........................... 122

20 Timings in milliseconds regarding with respect to patch and parallel processor system size (AFAC: black bars, FAC: white bars). ...................... 123

21 Timings for Load Balancing using MLB ........................................... 124

22 Results for a singular perturbation problem: Plots of the solution, the error and the composite grid with two different choices of the accuracy $\eta$ in the self-adaptive refinement process. .............................................. 125
TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Convergence factors for Poisson's equation on simple composite grid.</td>
<td>94</td>
</tr>
<tr>
<td>2</td>
<td>Convergence factors for Poisson's equation on block structured composite grid.</td>
<td>94</td>
</tr>
<tr>
<td>3</td>
<td>Convergence factors for a singular perturbation problem and, for comparison,</td>
<td>96</td>
</tr>
<tr>
<td></td>
<td>Poisson's equation.</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Timings for simple AFAC on iPSC/2.</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>Timings for Complex AFAC on iPSC/1.</td>
<td>102</td>
</tr>
<tr>
<td>6</td>
<td>Communication structure analysis of FAC and AFAC for the simple 7 patch</td>
<td>105</td>
</tr>
<tr>
<td></td>
<td>model problem.</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Communication structure analysis of FAC and AFAC for the complex 30 patch</td>
<td>106</td>
</tr>
<tr>
<td></td>
<td>model problem.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Timings for repositioning grids.</td>
<td>114</td>
</tr>
<tr>
<td>9</td>
<td>Timings for addition of new refinement.</td>
<td>115</td>
</tr>
<tr>
<td>10</td>
<td>Relative timings of AFAC for Moving, Adding, and Load Balancing (MLB).</td>
<td>116</td>
</tr>
<tr>
<td>11</td>
<td>Accuracy (L1-norm ε) vs. number of grid points (n) and number of blocks (b)</td>
<td>119</td>
</tr>
<tr>
<td></td>
<td>for MG-V on a uniform grid and FAC on self-adaptively refined composite grids.</td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

One of the aims in the development of efficient algorithms to solve partial differential equations (PDEs) is to allow the computational intensity to be proportional to the activity that the solution resolves. What this means practically is that solutions are then obtained with minimum computation, which in turn allows for investigation of larger even more complicated problems, which in turn we want solved with minimum computational effort, and so on. It is the finite resources of even today’s most modern computers that ensure termination of this recursion. Since, in a realistic application, this activity is non-uniformly distributed and localized in the problem space, the use of local refinement reduces computational complexity away from these localized regions and reduces the global computational work. More specifically, this use of local refinement allows for greater accuracy in the computation of large scale problems, and so the solution is obtained more efficiently. The resulting computational mesh is called the composite grid (see figure 1). Since the requirement for such refinement is often only seen at runtime as the solution evolves, such refinement often must be added self-adaptively. The use of self-adaptive refinement is most important with the use of more than just a few local refinement regions, since then the error of not adding refinement where required jeopardizes the effectiveness of additional levels of refinement.

1.1 Algorithm Development

A complicating feature in the development of these local refinement techniques is their introduction onto parallel computers. The nonuniform nature of the computational workload for a composite grid is in direct conflict with the general goal of efficient processor
utilization. Classical methods of computing with local refinement grids, aside from exhibiting slow convergence to the solution, require substantial synchronous intergrid transfers and processing of the solution between the composite grid levels. This means that, with a composite grid load balanced onto multiple processors, there is substantial inefficiency due to this synchronous processing of grid levels and the nonuniform use of local refinement throughout the problem domain. The alternative of partitioning each level across all processors is problematic since such partitionings greatly reduce the composite grid level's representation (size) in each processor on massively parallel architectures. Indeed, new algorithms for solving equations posed on composite grids have an opportunity to improve the existing performance of the parallelized serial algorithms. Thus, meaningful work on parallel adaptive mesh refinement is not just a computer science issue, but one which combines the design of parallel algorithms with the development of better mathematical algorithms, clearly crossing both disciplines almost equally.

The existing fast adaptive composite grid method, FAC (see [40] and [39]) is a discretization and solution method for partial differential equations designed to achieve efficient local resolution by systematically constructing the discretization based on various regular grids and using them as a basis for fast solution. Using multigrid as the individual grid solver, FAC has been applied to a variety of fluid flow problems, including incompressible Navier-Stokes equations [36] in both two and three dimensions. A more recently developed variation of FAC designed for parallel computers, the asynchronous fast adaptive composite grid method (AFAC), allows for independent processing of the individual regular grids that constitute the composite grid. This means that the cost of solving the independent grid equations may be fully shared by independent processing units.

AFAC is a method for adaptive grid discretization and solution of partial differential equations. Where FAC is forced to process the composite grid level synchronously, AFAC
eliminates this bottleneck of parallelism. Through a simple mechanism used to reduce inter-level dependence, individual refinement levels can be processed by AFAC in parallel. Coupled with multigrid, MG (see [10], [22], and [41]) for processing each constituent grid level, AFAC is usually able to solve the composite grid problem in a time proportional to what it would take to solve the global grid alone, with no added refinement. See Hart and McCormick [1] for further details.

Because of the way local grids add computational load irregularly to processors, an efficient load balancer is an essential ingredient for implementing adaptive methods on distributed memory machines. The complexity of this process, as well as the overall algorithm itself, depends intimately on how the processors are assigned to the computational tasks. The independence of the various refinement levels in the AFAC process allows these assignments to be made by level (in contrast to the usual domain decomposition approach), which greatly simplifies the associated load balancing of the composite grid. To balance loads between these levels, in this thesis we develop a new load balancing algorithm, Multilevel Load Balancing (MLB), which borrows heavily on the multilevel philosophy that guides most of the work presented. Specifically, MLB is a load balancing strategy that addresses the different levels of disparity in the loads that are spread spatially across the multiprocessor system. The algorithm is intended for use with applications that change dynamically, as is the case in self-adaptive mesh refinement and time dependent applications. Other load balancers have been previously developed, but most often exhibit slow performance that limits their usefulness on anything but predominantly static applications.

But even with the capability to solve composite grid problems efficiently in a parallel environment, there is still a large class of practical problems that are not sufficiently well addressed. The problems in this class are dynamic in nature: the movement and time evolution and decay of regions of activity force continual and substantial manipulation of the resulting partitioned composite grid. Local refinement around, or upstream of, shocks,
for example, often must be allowed to move (track the shock) so that the time dependent problem may be solved properly. Using conventional methods for assignment of work to processors, the result is an inefficient handling of this large class of problems because of how and where the data is localized in distributed memory. However, analysis of these inefficiencies leads to the sort of unconventional methods for assignment of work that can be processed efficiently in a parallel environment using AFAC and the partitioning strategy used in MLB. Clearly, the resulting movement of regions of local refinement that occurs in dynamic problems is not handled efficiently using conventional techniques (e.g., partitioning based on the division of the problem domain). In contrast, the resulting new methods for assignment of work allow for more efficient handling of this important class of dynamic problems in parallel environments.

Our experiments with a variety of local refinement algorithms for the solution of the simple potential flow equation on parallel distributed memory architectures demonstrates that, with the correct choice of solvers, performance of local refinement codes shows no significant sign of degradation as more processors are used. Contrary to conventional wisdom, the fundamental techniques used in our adaptive mesh refinement methods do not oppose the requirements for efficient vectorization and parallelization. In fact, this research has shown that algorithms that are expensive on serial and vector architectures, but are highly parallelizable, can be superior on parallel architectures. Thus, parallelization capabilities play an important role in the choice of the best suited algorithm for a specific application.

Chapter 2 details the development of AFAC and AFACx, including computational results of their use in a parallel environment (more results are found in sections 4.2 and 4.3). The results comparing FAC and AFAC appear in chapter 4, while results on the use of dynamic adaptivity for time dependent equations are presented in section 4.5. These results compare the relative computational costs of the two iterative algorithms, including the computational costs of adding and removing refinement regions adaptively and load
balancing the resulting composite grid using MLB. Section 3.2.6 describes MLB, a new multilevel load balancing algorithm for parallel adaptive refinement algorithms. Its use is a central part of good performance on parallel architectures. Section 4.4 contains a comparison of the parallel performances of FAC and AFAC for two sample composite grids. Section 2.2.3 introduces a new algorithm, AFACx, which improves on the AFAC algorithm by expanding applicability to more complex block structured local refinement grids. The use of AFACx both simplifies the use of parallel adaptive mesh refinement and is more efficient than AFAC. There we present the motivating theory for AFACx as well as a convergence proof. The next subsection illustrates the use of self-adaptivity by presenting an example problem where it is used. The final subsection addresses the complexity analyses of FAC, AFAC, and AFACx. In particular, we answer some problematic questions about how composite grids can be optimally partitioned for these three algorithms.

1.2 Software Development

The work introduced in the algorithm design section was the product of two separate parallel adaptive refinement implementations. The purpose of this section is to introduce the chapters that detail some of the more practical aspects of these separate implementations.

The implementation of parallel adaptive refinement, required for meaningful algorithm development, is a non-trivial issue that has historically limited further research in this important field. For the work in this thesis to be accomplished, the problem of practical development of parallel adaptive refinement software was addressed. The results form a significant part of this thesis because they so successfully resolve the requirements of self-adaptive mesh refinement for complex applications, even though the problems that are presented are simplistic in nature. Specifically, this thesis presents an object-oriented set of libraries in C++ (class libraries) that abstract the details of parallel adaptive refinement by separating the abstractions of parallelism and adaptive refinement from the user’s application code, which is permitted to define the application on only a uniform domain. It
is hoped that additional collaborative work will make the set of C++ class libraries more generally useful and available in the near future.

Parallel adaptive mesh refinement is a truly sophisticated level of software development, which is important since it strains the limits of what can be accomplished in software without massive support. Its use requires the development of many interrelated data structures to permit the flow of control between the solvers that are required for the implementation of the composite grid algorithms. Additionally, adaptive refinement is necessarily dynamic since the composite grid is most often required to evolve at runtime. ¹ Because of the dynamic requirements of adaptive mesh refinement, FORTRAN, a static language, was not considered to be an option in the development of the final codes. Note that even static refinement under explicit user control would require dynamic memory management since in the parallel environment data would have to be shuffled between processors in any load balancing scheme. The experience in this project was that FORTRAN is too outdated for use in the development of sophisticated parallel adaptive refinement software because of its inability to support the abstractions required to define algorithms independent of the location of the data that is manipulated. The concept of abstractions, and a presentation of how and why an algorithm should be expressed independent of the organization of its data, forms a basis for the P++ and AMR++ work that this thesis presents. We recognize that special FORTRAN memory management could as well have been developed to allow for some or most of these problems, but such work would have layered already complex code on top of yet another layer of software with its own inherent restrictions.

The first implementation was done in the language C, which permitted dynamic management of memory, a requirement of both static and adaptive refinement. The experience with the working C version of the local refinement code is detailed in section 3.2. Although written modularly and using the best techniques in software design known at the

¹In our experience, the ratio of code supporting the details of adaptive refinement to code defining the single grid application is approximately 20:1 for serial and parallel environments (due to the proportionally increased complexity of the parallel environment).
time, the complexity of the parallelism in the adaptive refinement code could not be hidden, thus greatly complicating the resulting code. The principal complications were in the partitioning of the composite grid, the management of memory for data shuffled between processors in the load balancing of the composite grid, the complex data structures which required access from all levels, etc.; all of these features were necessarily handled explicitly within the parallel adaptive refinement implementation. The C language version was completed with two separate implementations, one for serial computers and a second for parallel computers. Though acceptable for a research code, the requirement of separate serial and parallel codes limits the ability of parallel computing to address even more complex applications and, just as important, their maintenance in a commercial setting.

The C language version of parallel adaptive mesh refinement was run on the iPSC/1, iPSC/2, iPSC/860, SUPRENUM, and nCUBE parallel machines using precisely the same code, so a degree of parallel machine independence was actually achieved. Due to the conventional way the code was developed, with a procedural language (FORTRAN is also a procedural language), these complexities combined, and in effect multiplied, to limit the ability to develop the sort of sophisticated application codes we need for solving complicated flow problems efficiently. A more sophisticated self-adaptive mesh refinement code would have made work even more difficult. Later work in C++, however, better allowed for the management of these difficult issues and permitted even greater sophistication. The consistent goal throughout was to present a greater degree of architecture independence in order to simplify the implementation further.

The experiences with parallel adaptive mesh refinement in the original implementation in C, and those with a much more complex local refinement hypersonic application developed (with Dinshaw Balsara) specifically for the serial environment, have motivated additional work to simplify the development of such numerical software for parallel distributed
memory computers. The second generation of the parallel adaptive refinement implementation has been done much differently. The problems of combined parallelism, adaptive refinement, and application specific code are too interrelated to permit significant advancement to more complex forms of adaptivity and applications.

The solution to this software difficulty presents abstractions as a means of handling the combined complexities of adaptivity, mesh refinement, the application specific algorithm, and parallelism. The abstraction for adaptive refinement is represented by the definition of adaptive refinement in an object-oriented way that is independent of both explicit parallel details and specific application details. The abstraction of parallelism is to represent the lower level array operations in a way that is independent of parallelism (and adaptive refinement). These abstractions greatly simplify the development of algorithms and codes for complex applications. As an example, the abstraction of parallelism permits the development of application codes (necessarily based on parallel algorithms as opposed to serial algorithms, whose data and computation structures do not allow parallelization) in the simplified serial environment, and the same code can be executed in a massively parallel distributed memory environment. Since the codes require only a serial C++ compiler, we avoid the machine dependent restrictions of research projects involving parallel compilers. ²

We attack the details of parallel adaptive mesh refinement software development by dividing the problem into three large parts: 1) the abstraction of parallelism using the C++ parallel array class library P++; 2) the abstraction of adaptive mesh refinement using the C++ serial adaptive mesh refinement class library AMR++; 3) the serial application specific single grid application, defined by the user. The division into these parts serves to make the project smaller than it would otherwise be since the development of large codes is inherently nonlinear. Additionally, each part is sufficiently separate to form the basis of other large software, so the pieces are substantially reusable. This sort of code reuse is a common feature of the C++ object-oriented language. Thus, we present a greatly simplified

²However, such work toward parallel C++ compilers (notably CC++ and pC++) is important.
environment for the development of adaptive mesh refinement software in both the serial and parallel environment. More specifically, such work provides an architecture independent environment to support the development of more general complex software, which might be targeted for the parallel environment. We now summarize the individual parts.

P++ is a C++ parallel array class library that simplifies the development of efficient parallel programs for large scale scientific applications, while providing portability across the widest variety of computer architectures. The interface for P++ matches that of M++\(^3\), a commercially available array class library for serial machines, so numerical applications developed in the serial environment may be recompiled, unchanged, to run in the parallel distributed memory environment. Although general in scope, P++ supports current research in parallel self-adaptive mesh refinement methods by providing parallel support for AMR++, a serial class library specific to self-adaptive mesh refinement. The P++ environment supports parallelism using a standard language, C++, with absolutely no modification of the compiler. For parallel communication, it employs existing widely portable communications libraries. Such an environment allows existing C++ language compilers to be used to develop software in the preferred serial environment, and such software to be efficiently run, unchanged, in all target environments.

AMR++ is a C++ adaptive mesh refinement class library that abstracts details of adaptive mesh refinement independent of the users specific application code, which is used much like an input parameter to AMR++. AMR++ is written using the M++ serial array class library interface, and thus can be recompiled using P++ to execute in either the serial or parallel distributed memory environment. Thus, AMR++ is a serial application written at a higher level than P++, and specific to self-adaptive refinement. Forming a serial adaptive mesh refinement code requires only a uniform grid solver specific to the users' application. If the users' application code uses the M++/P++ array interface, then the users' AMR++/application code can be recompiled to execute in the parallel environment.

\(^3\text{M++ is a product of Dyad Software.}\)
Thus, such abstractions as AMR++ and P++ greatly simplify the development of complex serial, and especially parallel, adaptive mesh refinement software.

Chapter 3 presents the details of the first generation of the parallel adaptive refinement code and the problems that were solved and introduced by the combined complexity of adaptive refinement in the parallel environment. The work in this chapter motivated further work that simplified the object-oriented development of the later more complex parallel codes. Additionally, it motivates the requirement for a superior language for development of the adaptive mesh refinement application, whether targeted for the serial or parallel environments. Chapter 3 also presents the object-oriented design of the second generation parallel adaptive mesh refinement implementation. Section 3.5 presents the P++ parallel array class library and section 3.6 presents the AMR++ class library, including details of the self-adaptive mesh refinement strategy.

1.3 Problems and Future Work

Problems with the current work are discussed in chapter 5, since it is these problems that the future work will attempt to address. Finally, chapter 6 presents some of the future work that might be done to expand the usefulness of adaptive refinement methods for parallel architectures. These include improvements and ideas for the composite grid algorithms and the object-oriented strategies that guide their practical implementation. Additional detail focuses on possible improvements to the parallel array class library P++ and the adaptive mesh refinement class library AMR++.
Figure 1: Example composite grid with five levels.
CHAPTER 2

PARALLEL ALGORITHM DESIGN (THEORY)

2.1 Overview of the FAC and AFAC Composite Grid Algorithms

Reasons for basing the solution process on a composite grid include: 1) uniform solvers and their discretizations are more easily defined for complex equations; 2) multigrid solvers, which are appropriate to use, are simpler and more efficient on uniform grids; and 3) iterative processes are most efficiently implemented for the case of structured uniform grids. In section 2.2, we introduce the existing fast adaptive composite grid method, FAC, and the asynchronous fast adaptive composite grid method, AFAC, and uses them to motivate and define the details of AFACx, the new algorithm that this thesis presents. Convergence for the AFACx algorithm is proved in section 2.3. A different and more restricted analysis is contained in [43].

Both FAC and AFAC are multilevel methods for adaptive grid discretization and solution of partial differential equations. FAC has a sequential limitation to the processing of the individual levels of refinement, whereas AFAC has much better complexity in a parallel computing environment because it allows for simultaneous processing of all levels in the computationally dominate solution phase. Coupled with multigrid (MG) processing of each level and nested iteration on the composite grids, AFAC is usually able to solve the composite grid equations in a time proportional to what it would take to solve the global grid alone. See [22] and [41] for further details.

Both FAC and AFAC consist of two basic steps that are described loosely as follows. 
Step 1. Given the solution approximation and composite grid residuals on each level, use MG to compute a correction local to that level (solving the error equation). Step 2. Combine
the local corrections with the global solution approximation, compute the global composite grid residual, and transfer the local components of the approximation and residual to each level. The difference between FAC and AFAC is in the order in which the levels are processed and in the details of how they are combined.

Convergence theory in [40] shows that the FAC and AFAC asymptotic convergence factors, satisfy the relation $|||AFAC||| = |||FAC|||^k$, where AFAC and FAC are the error propagation operators for AFAC and FAC, respectively, and $||| \cdot |||$ is the composite grid energy norm $(|||u^r||| = (L^r u^r, u^r)^{1/2}$, where $(\cdot, \cdot)$ denotes the $L^2$ inner product and $L^r$ the composite grid operator). Although the theory is restricted to the two-level composite grid, experimental results have verified this relation even on a very large number of levels (a specific test verified this property on a 50-level composite grid\textsuperscript{1}). Though the algorithmic components in our code are chosen slightly differently than for the convergence analysis, experiences show that very similar behavior is obtained. This implies that two cycles of AFAC are roughly equivalent to one cycle of FAC.

2.2 Notation and Definition of FAC, AFAC, and AFACx Algorithms

To define these algorithms, we present the following problem and notation. We begin by posing the weak form of the continuous equation and its discretization. Let $\mathcal{H}^1_0(\Omega^0)$ be the usual Sobolev space of real-valued functions on the problem domain $\Omega^0$ that have zero values (in the usual sense) on the boundary, $\partial \Omega^0$, of $\Omega^0$. Assume that $a(\cdot, \cdot)$ is a generic, real-valued, bilinear, symmetric, bounded, and coercive form on $\mathcal{H}^1_0(\Omega^0) \times \mathcal{H}^1_0(\Omega^0)$, and that $f(\cdot)$ is a generic real valued functional on $\mathcal{H}^1_0(\Omega^0)$. The weak form is then to find $u \in \mathcal{H}^1_0(\Omega^0)$ such that

\[ a(u, v) = f(v), \quad \forall v \in \mathcal{H}^1_0(\Omega^0). \]

\textsuperscript{1}The actual solution of such a 50-level composite grid problem implies greater precision than double precision grid values permit, but this is an issue of machine accuracy.
To discretize this equation, let $T^0$ be a regular finite element partition of $\Omega^0$ (e.g., triangles or cells) and let $V^0 \subset H_0^1(\Omega^0)$ be the associated finite element space (e.g., piecewise linear functions on triangles or piecewise bilinear functions for cells). Let $\ell \geq 1$ denote the number of subregions of $\Omega$ (in the discrete problem, which we will develop, $\ell$ will define the number of composite grid levels), which we assume are nested:

$$\Omega^\ell \subset \Omega^{\ell-1} \subset \cdots \subset \Omega^1 \subset \Omega^0.$$  

For each $k \in \{1, 2, \ldots, \ell\}$, let $T^k$ be a regular finite element partition of $\Omega^k$ constructed by appropriately refining the elements of $T^{k-1}$, and define $V^k \subset H_0^1(\Omega^k)$ as the finite element space associated with $T^k$. Note that functions in $V^k$ are zero on $\partial \Omega^k$. Define $W^k = V^{k-1} \cap V^k$, which is a coarse grid subspace of $V^k$. Note that $W^k$ is similar to $V^{k-1}$ except that it is local (if $\Omega^k$ is a proper subregion of $\Omega^{k-1}$). For convenience, let $W^0 = \{0\}$.

Note that the refinement has been constructed so that it is conforming in the sense that $W^k = V^{k-1} \cap H_0^1(\Omega^k)$. We will refer to $W^k$ as the restricted local refinement space for the local refinement region $\Omega^k$. Define $\tilde{I}^k : W^k \rightarrow V^k$ as the natural embedding (interpolation) operator.

Now define the composite grid by

$$\Omega^c = \bigcup_{i=0}^p \Omega^i,$$

and its associated space by

$$V^c = \sum_{i=0}^p V^i.$$  

Then the discrete problem we solve is the composite grid Galerkin discretization, using the composite grid space $V^c$: find $u^c \in V^c$ such that

$$a(u^c, v) = f(v), \quad \forall v \in V^c.$$  

Let $L^k : V^k \rightarrow V^k$ be the discrete operator on grid space $V^k$ determined by

$$a(u^k, v^k) = (L^k u^k, v^k), \quad \forall u^k \in V^k,$$

where $(\cdot, \cdot)$ denotes the $L^2(\Omega^0)$ inner product. Note
that $L^0$ is an approximation to the differential operator on the global region $\Omega^0$. Let $I^k_\ell : V^k \rightarrow V^\ell$ and $I^\ell_k : V^\ell \rightarrow V^k$ be given interlevel transfers operators (interpolation and restriction, respectively, defined by the finite element formulation). Note that $I^k_\ell$ is the natural imbedding operator and $I^\ell_k$ is its adjoint. Finally, consider the restricted grids $\tilde{\Omega}^k = \Omega^{k-1} \cap \Omega^k$ and their associated spaces $\tilde{V}^k = \tilde{W}^k$ and operators $\tilde{L}^k$, $\tilde{I}^k_\ell$, and $\tilde{I}^\ell_k$, and let $\tilde{n}^k$ denote the restricted grid solution, $1 \leq k \leq \ell$. (For AFAC and AFACx, these restricted (intermediate) grids and their operators are designed to remove error components that are common to both fine and coarser levels and that would otherwise prevent simultaneous processing of the levels.)

### 2.2.1 FAC Algorithm

FAC (see [40] and [39]) is a discretization and solution method for partial differential equations designed to achieve efficient local resolution by systematically constructing the discretization based on various regular grids and using them as a basis for fast solution. Using multigrid as the individual grid solver, FAC has been applied to a variety of fluid flow problems, including incompressible Navier-Stokes equations [36] in both two and three dimensions.

Loosely speaking, one FAC iteration consists of the following basic steps:

- **Step 1.** For all $k \in \{0, 1, \ldots, \ell\}$, compute $f^k$ by transferring the composite grid residual to $\tilde{\Omega}^k$.

- **Step 2.** Set $k = 0$ (so that we start the computation on $\tilde{\Omega}^0$) and the initial guess on $\tilde{\Omega}^k$ to zero.

- **Step 3.** Given the initial guess and composite grid residuals on level $k$, use multigrid (or, alternatively, any direct or iterative solver) to compute a correction local to that level, that is, "solve" the error equation that results from the use of the residual assigned to $f^k$ on $\tilde{\Omega}^k$.

- **Step 4.** If $k < \ell$, then: interpolate the "solution" (resulting from step 3) at the interface of levels $\tilde{\Omega}^k$ and $\tilde{\Omega}^{k+1}$ to supply $\tilde{\Omega}^{k+1}$ with complete boundary conditions, so that its correction equation is properly posed; interpolate it also to (the interior of)
\( \Omega^{k+1} \) to act as the initial guess; set \( k \leftarrow k+1 \); and go to step 3. If \( k = \ell \), interpolate all corrections (i.e., "solutions" of each level's projected composite grid residual equations) from the finest level in each region (i.e., \( \Omega^k / \Omega^{k+1} \)) to the composite grid.

To be more specific, in addition to the above notation, let \( J^{k+1}_k : V^k \rightarrow V^{k+1} \) denote the mapping that interpolates values from level \( k \) on the interface (i.e., the boundary of \( \Omega^{k+1} \) that does not coincide with the boundary of \( \Omega^0 \)). Note that the computation of the composite grid residual equations at the interface is covered in detail in [39]. Given the composite grid right-hand side \( f^c \) and initial approximation \( u^c \), then one iteration of FAC (see McCormick [39] for motivation and further detail) is defined more concretely as follows (we show here the direct solver version for simplicity, which requires no initial guess on \( \Omega^k \)):

- **Step 1.** For all \( k \in \{0, 1, \ldots, \ell\} \), set \( f^k = I^k_c(f^c - L^c u^c) \).
- **Step 2.** Set \( k = 0 \).
- **Step 3.** Compute \( u^k = (L^k)^{-1} f^k \).
- **Step 4.** If \( k < \ell \), set \( u^{k+1} = J^{k+1}_k u^k \) on \( \Omega^{k+1} \) and go to step 3. If \( k = \ell \), form \( u^* = I^\ell_c u^k \) on \( \Omega^k / \Omega^{k+1} \) for each \( k \) (\( \Omega^{k+1} = \emptyset \)).

### 2.2.2 AFAC Algorithm

AFAC ([22] and [41]) is a multilevel method for adaptive grid discretization and solution of partial differential equations. AFAC appears to have near optimal complexity in a parallel computing environment because it allows for simultaneous processing of all levels of refinement. This is important because the solution process on each grid, even with the best solvers, dominates the computational intensity. This is especially true for systems of equations where the solution process is even much more computationally intensive than the evaluation of the residuals. Coupled with multigrid processing of each level and nested iteration [39] on the composite grids, AFAC is usually able to solve the composite grid equations in a time proportional to what it would take to solve the global grid alone. See Hart and McCormick [22] and McCormick [40] for further details.
The principal step is the computation of an approximation to the oscillatory component of the solution on each composite grid level. To simplify the explanation, we define $u^k = f^k u^k - \bar{f}^k \bar{u}^k$ to be the "oscillatory component" of the solution $u^k$.

Loosely speaking, one AFAC iteration consists of the following basic steps:

- **Step 1.** Compute $f^k$ for all $k \in \{0, 1, \ldots, \ell\}$ by transferring the composite grid residual to $\Omega^k$, and similarly for $\tilde{f}^k$ for all $k \in \{1, 2, \ldots, \ell\}$.

- **Step 2.** Set the initial guess to zero on $\Omega^k$ for all $k \in \{0 \ldots \ell\}$, and similarly on $\tilde{\Omega}^k$ for all $k \in \{1, 2, \ldots, \ell\}$.

- **Step 3.** For all grid levels $\Omega^k (k \in \{0, 1, \ldots, \ell\})$:

  - **Substep 3a.** Use multigrid (or, alternatively, any direct or fast iterative solver) to compute a correction local to that level, that is, "solve" the equation that results from the use of $f^k$ on $\Omega^k$ and $\tilde{f}^k$ on $\tilde{\Omega}^k$ ($k > 0$).

  - **Substep 3b.** Subtract the restricted grid "solution" from the local grid "solution."

This forms the "oscillatory components."

- **Step 4.** Interpolate and add the "oscillatory components" on all of $\Omega^k$ for all $k \in \{0, 1, \ldots, \ell\}$ to all finer composite levels.

To be more specific, given the composite grid right-hand side $f^c$ and initial approximation $u^c$, then one iteration of FAC (see McCormick [35] for motivation and further detail), is defined more concretely as follows (we show here the direct solver version for simplicity, which again needs no initial guesses on $\Omega^k$ or $\tilde{\Omega}^k$):

- **Step 1.** For all $k \in \{0, 1, \ldots, \ell\}$, set $f^k = I^k_c (f^c - L^c u^c)$ and (for $k > 0$) $\tilde{f}^k = \bar{I}^k_c (f^c - L^c u^c)$.

- **Step 2.** For all $k \in \{0, 1, \ldots, \ell\}$, set $u^k = 0$ and (for $k > 0$) $\bar{u}^k = 0$.

- **Step 3.** For all $k \in \{0, 1, \ldots, \ell\}$:

  - **Substep 3a.** Compute $u^k = (L^k)^{-1} f^k$ and (for $k > 0$) $\bar{u}^k = (\bar{L}^k)^{-1} \bar{f}^k$.

  - **Substep 3b.** Set $u^k = I^k_c u^k - \bar{I}^k_c \bar{u}^k$ for $k \in \{1, 2, \ldots, \ell\}$. 
• Step 4. Set \( u^c = u^0 + I_0^c u^0 + \sum_{k=1}^{c} I_k u^k \).

The processing of each step of AFAC is fully parallelizable. For example, the levels in step 3 can be processed simultaneously by a MG solver, which is itself parallelizable [10]. The present version of the code uses synchronization between steps 3 and 4, although asynchronous processing would be allowed here. (With an efficient load balancing scheme, asynchronous processing of the levels provides little real advantage.) A more complete derivation of AFAC, along with a convergence proof and related theory, can be found in McCormick [40].

2.2.3 AFACx Algorithm  AFACx is a new algorithm, which this thesis presents and analyzes as its principal mathematical development. The motivation for AFACx is in the use of adaptive refinement for problems with complex internal regions demanding additional, but local, resolution. Such complex internal regions are found around, and upstream of, shocks and complex shock structures.\(^2\)

In this section, since the AFACx algorithm is new, we present an expanded description, which includes the principal motivation for its development and use.

2.2.3.1 AFACx Motivation  The use of adaptive refinement for problems with geometrically complex regions of activity requires more than simple rectangular refinement. In such problems, local refinement strategies must cover the target regions with non-regular meshes, or collections of regular rectangular grids that combine to conform to the non-regular regions. In the latter approach, using collections of regular rectangular grids, the efficiency and simplified implementation of complex application codes can be restricted to the more conventional setting\(^3\), where good efficiency on the structured rectangular grids is assured.

For explicit problems, the details of handling the resulting block structured local refinement regions are an issue only at the interfaces; the properties of the algorithm (stability,
convergence, etc.) are not as much an issue as they are for implicit problems. With implicit equations, the handling of block structured grids is more problematic since the solution of the individual blocks is not sufficient for the solution of the block structured grid. More complex methods are typically required, but rarely applied, to also resolve the smoother components that span the collection of block structured grids, since if only the blocks are processed with iterative solvers, the smooth errors (across blocks) are poorly damped and result in poor convergence factors for the block structured solution.

It would be sufficient to solve the block structured local refinement grid problems directly, but this is prohibitively expensive. Alternatively, we could define a multigrid set of coarsenings (of the blocks and the structure of blocks), which would permit fast multigrid solution of such block structured refinement regions. But the automated coarsening of the block structure (beyond that of simply the coarsened blocks themselves) is difficult to implement and the solvers abstracted to work on such coarsenings are inefficient.

For implicit solvers, the block structured solution is required, on the block structured grid, if we intend to use AFAC, since the formulation requires an approximate solution on the refinement patch. FAC could be simplified to use relaxation on the block structured regions starting from the solution interpolated from the coarser level (the global grid in the two level composite grid case). The use of relaxation avoids the complication of constructing the block structured coarsenings that a multigrid solver would require for the block structured refinement region. We seek a similar efficiency, but with AFAC, so that the composite grid levels can be processed asynchronously. AFACx is just such an algorithm, since it requires no definition of the coarsened block structured local refinement regions and uses only relaxation on the predefined block structured refinement region.

\footnote{However, the details of these explicit algorithms at each grid point can be more complex., e.g., for PPM and ENO methods for Euler equations, the Riemann solvers are more sophisticated than most relaxation methods used in the implicit MG solvers.}

\footnote{A substantial amount of work was done on this approach, and its failure motivated the object-oriented approach that was taken and that led to AMR++ (see section 3.6).}

\footnote{This version (variation) of FAC is introduced as FACx in section 4.2.
AFACx uses only the predefined block structured region and requires no construction of coarsening, even for the individual blocks, but still preserves the multilevel efficiency of AFAC\textsuperscript{7} and processes all levels asynchronously. The predefined block structured grid includes the finest level of the local refinement region and the grid built consistent with the flagged points on the coarser composite grid level, which were used to generate the block structured local refinement region. In practice it is easier, and equivalent, to let each block define a single coarsening. This coarsening of each block is guaranteed to exist because the local refinement block was derived from flagging points on the coarser composite grid level (the coarse grid points of the block structured refinement patch).

Thus, AFACx uses the finest level of the block structured refinement grid (the block structured refinement grid itself), and a single coarser level (which we have shown is predefined since it corresponds to the flagged points that were used initially to build the finer local refinement level). Because AFACx avoids processing the coarser levels on each block, it is cheaper than AFAC, though the difference is only in the processing of the coarser levels and so it is not very significant in a serial environment.\textsuperscript{8} However, in a parallel environment, the avoidance of processing coarser levels means substantially less message traffic in the multiprocessor network, and a higher parallel efficiency for the overall method. Since, in the context of adaptive mesh refinement, the local regions are sized according to demand, it is likely that such refinement blocks would not be sufficiently large to adequately hide the communication overhead of processing the coarsest levels. Thus, by avoiding the coarsening altogether, we avoid a potentially significant overhead in the parallel environment, as well as the complicated construction of the block structured coarsening that would be required of a completely general block structured grid.

Finally, the use of relaxation on the block structured grids is what makes AFACx, and the analogous variant of FAC, attractive. This is because the user defined relaxation

\textsuperscript{7}The convergence factors are observed to be within 3\% of that of AFAC.

\textsuperscript{8}The relatively inefficient processing of short vectors is also avoided in the vector environment.
(which is assumed to be parallelizable) is easily supported on the block structured grids. Then the process of exploiting the parallelism across blocks, on the block structured grid, is equivalent to that of exploiting the parallelism across multiple processors, on the partitioned grid. Thus, in the support for the block structured grid, interface boundaries are copied in much the same way that messages are passed between processors. These details are hidden from the user in the AMR++ class library in the same way that the message passing is hidden from the user in the P++ class library; see chapter 3, sections 3.6 and 3.5, respectively.

2.2.3.2 AFACx Definition In the case of AFAC, we can attempt to understand it by considering the use of exact solvers on the composite, local refinement, and restricted grids. An important step in AFAC is the elimination of the common components between the coarser composite grid level and the finer local refinement grid patch. This step allows us to avoid the amplification of these components (inherently smooth components, since only they are represented on, or shared by, both the local refinement and coarser levels) in the interpolation and addition of the solution from the coarser levels up through the finer composite grid levels, finally forming the composite grid approximation. The result in AFAC, after this important step, on each level, is an approximation to the oscillatory contribution to the composite grid solution, which is unique to that level. AFAC and AFACx differ only in the way that this oscillatory contribution is computed: AFAC uses exact or fast approximate solvers, and AFACx uses only relaxation (typically one or two sweeps).

In order to differentiate between the individual relaxation iterates, we will use subscripts: $u_n^k$ is the $n$th iterate approximating the solution $u^k$ on the $k$th grid level $\Omega^k$. Actually, we use only one iteration on $\Omega^k$, but allow for more on $\tilde{\Omega}^k$. Let $\rho(L^k)$ be the spectral radius of the discrete differential operator $L^k$. Then Richardson's iteration, which we will use throughout our analysis, is given by

$$u^k_{n+1} = u^k_n - \frac{1}{\rho(L^k)} (L^k u^k_n - f^k) \equiv R_k (u^k_n; f^k),$$

and similarly for $\tilde{u}^k_{n+1}$ and $\tilde{R}_k (\tilde{u}_n; \tilde{f}^k)$. Denote $\tilde{R}_k (\tilde{u}^k_n; \tilde{f}^k) = \tilde{R}_k (\tilde{R}_k (\tilde{u}^k_n; \tilde{f}^k))$ and so on for
\( \hat{R}_n^k \) so that \( \bar{u}_n^k = \hat{R}_n^k(\bar{u}_n^k, \bar{f}^k) \). To simplify the explanation below, we define \( u_n^k = I_k^c u_l^k - I_k^c \bar{u}_n^k \) to be the "oscillatory component" of the first iterate \( u_l^k \). An important aspect of AFACx is that the relaxation iterate \( \overline{u}_n^k \) is computed first and the initial guess for \( \Omega^k \) is interpolated from the iterate \( \hat{u}_n^k \) on \( \hat{\Omega}^k \) (i.e., \( u_0^k = \hat{I}_k^c \hat{u}_n^k \)).

We can now define AFACx more precisely. Given the composite grid right-hand side \( f^c \) and initial approximation \( u^c \), then one cycle of AFACx based on one relaxation sweep per level \( \Omega^k \) and \( n \) sweeps per level \( \hat{\Omega}^k \) is given by the following:

- Step 1. For all \( k \in \{0, 1, \ldots, \ell\} \), set \( f^k = I_k^c(f^c - L^c u^c) \) and \( (k > 0) \bar{f}^k = \hat{I}_k^c(f^c - L^c u^c) \).

- Step 2. For all \( k \in \{1, 2, \ldots, \ell\} \), set \( \bar{u}^k = 0 \).

- Step 3. For all \( k \in \{0, 1, \ldots, \ell \} \):
  
  Substep 3a. Set \( \bar{u}_n^k = R_n^k(\bar{u}_n^k, \bar{f}^k) \) for \( k > 0 \) and \( \bar{u}_n^k = 0 \) for \( k = 0 \), then \( u_0^k = \hat{I}_k^c \bar{u}_n^k \), and \( u_1^k = R_k(u_0^k, f^k) \) for \( k > 0 \) and \( u_1^k = (L^0)^{-1} f^0 \) for \( k = 0 \).

  Substep 3b. Set \( \bar{u}_1^k = I_k^c u_1^k - \hat{I}_k^c \bar{u}_n^k \).

- Step 4. \( u^c = u^c + I_0^c u^0 + \sum_{k=1}^{\ell} u_1^k \).

Notice step 3 uses only relaxation, and that the fine grid initial guess is the restricted grid approximation interpolated to \( \Omega^k \), namely, \( \hat{I}_k^c \bar{u}_n^k \). All steps except step 3 are the same as in AFAC.

In the next section, the connections between AFACx and AFAC are further clarified since the convergence proof of AFACx relies on convergence theory developed for AFAC.

### 2.3 AFACx Convergence Theory

We prove the convergence of AFACx in several steps. The basic idea for the development is to use existing AFAC theory [40] and establish that the AFACx convergence factor is at most only slightly larger than that of AFAC. As with the AFAC theory (McCormick [40]), we develop this in the restricted setting of a two-level composite grid problem.

To simplify notation in the case of a two-level composite grid, we introduce the
fine grid patch $\Omega_h = \Omega^1$ and its restricted grid $\Omega_{2h} = \tilde{\Omega}^1$. In a similar way, we define the following:

- the exact discrete solution on $\Omega_h$ is denoted by $u^h = u^1$, and on $\Omega_{2h}$ by $u^{2h} = \tilde{u}^1$.

Further, we use the subscript $n$ to denote the $n$th iterate of a relaxation step. Thus, $\tilde{u}^{2h}_n$ is the $n$th iterate of the relaxation operator on $\Omega_{2h}$, starting from $u^{2h}_0$. 

- the interpolation operator from $\Omega_{2h}$ to $\Omega^c$ is denoted by $I^c_{2h} \equiv I^c_1$. 

- the restriction operator from $\Omega^c$ to $\Omega_h$, is denoted by $I^c_{2h} \equiv I^c_1$.

- similarly, interpolation and restriction between $\Omega_{2h}$ and $\Omega_h$ are denoted by $I^h_{2h}$ and $I^h$, respectively.

- the fine grid discrete differential operator on $\Omega_h$ is denoted by $L^h \equiv L^1$.

- the restricted grid discrete differential operator on $\Omega_{2h}$ is denoted by $L^{2h} \equiv \tilde{L}^1$.

To support the comparison of AFACx and AFAC, we require a specific version of AFAC that uses a combination of exact and iterative solvers. To this end, we define $\overline{AFAC}$ as the usual AFAC method defined in section 2.2.2, except that the exact or multigrid solver on $\Omega_h$ is replaced by one relaxation sweep starting from the initial guess $u^{2h}_0 = I^c_{2h} u^{2h}$. Note that $u^{2h}$ is the exact solution of the discrete problem on the restricted grid $\Omega_{2h}$. We will first relate $\overline{AFAC}$ to the approximate-solver version of AFAC introduced in [40], which is denoted by $AFAC_{\xi}$, where $\xi = \begin{pmatrix} \xi^h \\ \xi^{2h} \end{pmatrix}$. It is defined by replacing the exact solvers on $\Omega_{2h}$ and $\Omega_h$ by approximate solvers based on operators $M^{2h} \approx (L^{2h})^{-1}$ and $M^h \approx (L^h)^{-1}$ that satisfy

$$(1 - \xi^h) (L^h)^{-1} \leq M^h \leq (L^h)^{-1},$$

and

$$(1 - \xi^{2h}) (L^{2h})^{-1} \leq M^{2h} \leq (L^{2h})^{-1},$$

respectively. That is, the exact solutions on $\Omega_{2h}$ and on $\Omega_h$ are replaced by the iterates $u^{2h} = M^{2h} I^c_{2h} (f^c - L^c u^c)$ and $u^h = I^h_{2h} u^{2h} + M^h (L^h I^h_{2h} u^{2h} - I^h (f^c - L^c u^c))$, respectively.
Note that, in the case of Richardson’s iteration, \( M^{2h} = \frac{1}{\rho(L^{2h})} I \) and \( M^h = \frac{1}{\rho(L^h)} I \).

Finally, let \( \delta > 0 \) and \( \bar{\delta} > 0 \) be the quantities defined on pages 110 and 118 of [40] that are typically bounded uniformly in \( h \), depending on the application. In this section, we assume that \( L^c \) in symmetric and positive definite and that the following variational conditions hold:

\[
L^{2h} = L_{2h}^c L_{2h}^c , \quad L^h = L_h^c L_h^c
\]

and

\[
I_{2h}^c = c_{2h} (I_{2h}^c)^T , \quad I_h^c = c_h (I_h^c)^T
\]

where \( c_{2h} \) and \( c_h \) are positive constants. Assume further that \( I_{2h}^c \) and \( I_h^c \) are full rank, so that \( L^{2h} \) and \( L^h \) are (symmetric) positive definite.

**Theorem 1** The spectral radii of the two-level exact solver versions of \( AFAC \) and \( FAC \) satisfy the relation

\[
\rho(AFAC) = \rho(FAC)^{\frac{1}{2}}.
\]

Thus, with \( ||| \cdot ||| \) denoting the composite grid energy norm \( (|||u^e||| = (L^c u^e , u^e)^{\frac{1}{2}}) \), we have

\[
|||AFAC||| \leq \left( \frac{\delta}{1 + \delta} \right)^{\frac{1}{2}}.
\]

The convergence factor for the approximate solver version, \( AFAC_\epsilon \), satisfies

\[
|||AFAC_\epsilon||| \leq (1 - \epsilon) |||AFAC||| + \epsilon.
\]

where \( \epsilon = \max(\epsilon^{2h}, \epsilon^h) \).

**Proof:** See McCormick [40], page 144.

**Lemma 1** \( AFAC \) converges with factor bounded according to

\[
|||AFAC||| \leq (1 - \epsilon) |||AFAC||| + \epsilon < 1,
\]

where \( \epsilon = \left(1 - \frac{1}{\delta} \right)^{\frac{1}{2}} \).
Proof: First consider the case of $n = 1$. The inequality on page 118 of [40] shows that

$$\| (I - \frac{1}{\rho(L^h)} L^h) e^h \| \leq \left( 1 - \frac{1}{\delta} \right)^2 \| e^h \|$$

(1)

for the initial error $e^h$ on $\Omega^h$. Since $M^h = \frac{1}{\rho(L^h)} I > 0$ for Richardson iteration, we clearly have

$$\langle r^h, (1 - c^h) (L^h)^{-1} r^h \rangle \leq \langle r^h, M^h r^h \rangle \leq \langle r^h, (L^h)^{-1} r^h \rangle$$

for $r^h \equiv L^h e^h$ and $c^h \equiv \left( 1 - \frac{1}{\delta} \right)^{\frac{1}{2}}$. The lemma now follows from the estimate for $AFAC_x$ in Theorem 1 with $\xi = \frac{c^h}{0}$. The lemma now also follows for general $n \geq 1$ because additional relaxations on level $h$ in $AFAC$ cannot increase the energy norm of the error.

Q.E.D.

$AFAC$ and $AFAC^c$ differ by a perturbation term that can be expressed in terms of the operator

$$P_n^h = -\frac{1}{\rho(L^h)} L^h f^h I_{2h} \left[ I - \frac{1}{\rho(L^{2h})} L^{2h} \right]^n.$$  

(2)

Lemma 2 The error propagation operators $AFAC_x$ and $AFAC^c$ are related according to

$$AFAC_x e^c = AFAC^c e^c + I^c P_n^h e_0^{2h},$$

where $e_0^{2h}$ is the initial error on level $2h$.

Proof: First consider the case $n = 1$. Following the definition of $AFAC_x$, the iterate $u_1^{2h}$ on $\Omega_{2h}$ using the initial guess $u_0^{2h}$ is given by

$$u_1^{2h} = u_0^{2h} - \frac{1}{\rho(L^h)} (L^h u_0^{2h} - f^h).$$

Then, as defined, the initial guess for the subsequent relaxation step on $\Omega_h$ is just $u_0^h = I_{2h}^h u_1^{2h}$. So the iterate on $\Omega_h$ is computed as

$$u_1^h = u_0^h - \frac{1}{\rho(L^h)} (L^h u_0^h - f^h).$$
Then, by substitution,

\[ u^h_1 = I^h_{2h} \left[ u^{2h}_0 - \frac{1}{\rho(L^{2h})} (L^{2h}u^{2h}_0 - f^{2h}) \right] - \frac{1}{\rho(L^h)} \left( L^h I^h_{2h} \left[ u^{2h}_0 - \frac{1}{\rho(L^{2h})} (L^{2h}u^{2h}_0 - f^{2h}) \right] - f^h \right). \]

Consider the splitting of \( u^h \) into its energy orthogonal components: \( u^h = I^h_{2h} u^{2h} + t^h \), where \( u^{2h} \) is a level \( 2^h \) component and \( I^h_{2h} L^h t^h = 0 \). Then,

\[ u^h_1 = I^h_{2h} \left[ u^{2h}_0 - \frac{1}{\rho(L^{2h})} L^{2h} \varepsilon^{2h}_0 \right] - \frac{1}{\rho(L^h)} \left( L^h I^h_{2h} \left[ u^{2h}_0 - \frac{1}{\rho(L^{2h})} L^{2h} \varepsilon^{2h}_0 \right] - L^h \left( I^h_{2h} u^{2h} + t^h \right) \right), \]

where \( \varepsilon^{2h}_0 = u^{2h}_0 - u^{2h} \). We thus have

\[ u^h_1 = I^h_{2h} \left[ u^{2h}_0 - \frac{1}{\rho(L^{2h})} L^{2h} \varepsilon^{2h}_0 \right] - \frac{1}{\rho(L^h)} \left( L^h I^h_{2h} \left[ \varepsilon^{2h}_0 - \frac{1}{\rho(L^{2h})} L^{2h} \varepsilon^{2h}_0 \right] \right) + \frac{1}{\rho(L^h)} L^h t^h. \] \( (3) \)

Now, following the definition of \( \overline{AFAC} \), its iterate \( \overline{u}^h_1 \), which uses as its initial guess the exact solution from grid \( \Omega_{2h} \) (namely, \( I^h_{2h} u^{2h} \)), is given by \( ^9 \)

\[ \overline{u}^h_1 = I^h_{2h} u^{2h} + \frac{1}{\rho(L^h)} L^h t^h. \] \( (4) \)

The final processing step on \( \Omega^* \) subtracts from the final iterate the interpolated \( 2^h \) approximation, which is just

\[ I^h_{2h} \left[ u^{2h}_0 - \frac{1}{\rho(L^{2h})} L^{2h} \varepsilon^{2h}_0 \right] \]

for the case of \( AFACx \) and \( I^h_{2h} u^{2h} \) for the case of \( \overline{AFAC} \). We represent the solution after the subtraction of the restricted grid solution for \( AFACx \) and \( \overline{AFAC} \) by \( u^*_h \) and \( \overline{u}^*_h \), respectively:

\[ u^*_h = u^h_1 - I^h_{2h} \left[ u^{2h}_0 - \frac{1}{\rho(L^{2h})} L^{2h} \varepsilon^{2h}_0 \right] \]

\[ = - \frac{1}{\rho(L^h)} \left( L^h I^h_{2h} \left[ \varepsilon^{2h}_0 - \frac{1}{\rho(L^{2h})} L^{2h} \varepsilon^{2h}_0 \right] \right) + \frac{1}{\rho(L^h)} L^h t^h \] \( (5) \)

\( ^9 \)Here, the initial guess on \( \Omega^* \) is given by \( u^*_0 = I^h_{2h} u^{2h} \) on the fine grid patch.
\[ u^h = u_1^h - I_{2h}^h u_2^h \]
\[ = \frac{1}{\rho(L_h)} (L_h u^h). \]  \hfill (6)

The definition of \( P_n^h \) for \( n = 1 \) then shows that
\[ P_1^h e_0^{2h} = -\frac{1}{\rho(L_h)} L_h I_{2h}^h \left[ I - \frac{1}{\rho(L_{2h})} L_{2h} \right] e_0^{2h} \]
\[ = u_2^h - \bar{u}^h, \]  \hfill (7)

from which the desired equality follows.

Assuming now the more general case of \( n \geq 1 \) relaxation steps, it is easy to verify that the approximations after the final processing step satisfy

\[ u_n^h = -\frac{1}{\rho(L_h)} L_h I_{2h}^h \left[ I - \frac{1}{\rho(L_{2h})} L_{2h} \right] e_0^{2h} + \frac{1}{\rho(L_h)} L_h q^h \]  \hfill (8)

and

\[ \bar{u}^h = \frac{1}{\rho(L_h)} L_h q^h. \]  \hfill (9)

Again, the desired inequality follows from the definition of \( P_n^h \).

Q.E.D.

\( \| \cdot \| \) was defined as the composite grid energy norm. But for functions \( v^h \in V_h \), we note that \( \| I_h^c v^h \| = \langle L_c I_h^c v^h, I_h^c v^h \rangle^{\frac{1}{2}} = \langle L_h v^h, v^h \rangle^{\frac{1}{2}} \), which we write simply as the fine grid energy norm \( \| v^h \| = \langle L_h v^h, v^h \rangle^{\frac{1}{2}} \). Similarly, for \( v^{2h} \in V^{2h} \) we can define \( \| v^{2h} \| = \langle L^{2h} v^{2h}, v^{2h} \rangle^{\frac{1}{2}} = \| I_{2h}^c v^{2h} \| \).

**Lemma 3** The perturbation term \( P_n^h e_0^{2h} \) is bounded according to

\[ \| P_n^h \| \leq \rho \left( (L_{2h})^{-\frac{1}{2}} \left( I - \frac{1}{\rho(L_{2h})} L_{2h} \right) \right)^n \left( I - \frac{1}{\rho(L_{2h})} L_{2h} \right)^{\frac{1}{2}} \left( I - \frac{1}{\rho(L_{2h})} L_{2h} \right)^{\frac{n}{2}} \left( L_{2h} \right)^{-\frac{1}{2}}. \]
Proof: We can simplify the evaluation of the energy norm $||| \cdot |||$ since it is related to the $L_1$ norm $|| \cdot ||$:

$$||| P_n^h e_0^{2h} ||| = \langle L^h P_n^h e_0^{2h}, P_n^h e_0^{2h} \rangle^\frac{1}{2} = \langle (L^h)^\frac{1}{2} P_n^h e_0^{2h}, (L^h)^\frac{1}{2} P_n^h e_0^{2h} \rangle^\frac{1}{2} = ||(L^h)^\frac{1}{2} P_n^h e_0^{2h}||.$$  

First note that

$$||| e_0^{2h} ||| = ||| P_n^h e_0^{2h} |||$$

$$= \langle L^h P_n^h e_0^{2h}, P_n^h e_0^{2h} \rangle^\frac{1}{2}$$

$$= \langle e_0^{2h}, I_{2h}^T L^h I_{2h} e_0^{2h} \rangle^\frac{1}{2}$$

$$= \langle e_0^{2h}, L_{2h} e_0^{2h} \rangle^\frac{1}{2}$$

$$= ||| (L_{2h})^\frac{1}{2} e_0^{2h} |||.$$  \hspace{1cm} (10)

By defining $w_{2h} = (L_{2h})^\frac{1}{2} e_0^{2h}$, $G_h = -\frac{1}{\rho(L_\infty)} L^h$, and $F_{2h} = (I - \frac{1}{\rho(L_\infty)} L_{2h})^n$, we thus have

$$\frac{||| P_n^h e_0^{2h} |||}{||| e_0^{2h} |||} = \frac{||(L^h)^\frac{1}{2} G_h I_{2h}^h F_{2h} e_0^{2h}||}{||(L_{2h})^\frac{1}{2} e_0^{2h}||}$$

$$\leq \sup_{|w_{2h}| = 1} \left[ \frac{||(L^h)^\frac{1}{2} G_h I_{2h}^h F_{2h} (L_{2h})^\frac{1}{2} w_{2h}||}{||w_{2h}||} \right]$$

$$= \left| (L^h)^\frac{1}{2} G_h I_{2h}^h F_{2h} (L_{2h})^\frac{1}{2} \right|$$

$$= \rho \left( (L_{2h})^\frac{1}{2} F_{2h} \left( I_{2h}^h G_h L_{2h} I_{2h}^h \right) F_{2h} (L_{2h})^\frac{1}{2} \right)^\frac{1}{2}$$

$$= \rho \left( (L_{2h})^\frac{1}{2} F_{2h} \left( I_{2h}^h \frac{1}{\rho(L^h)^2} (L^h)^\frac{1}{2} I_{2h}^h \right) F_{2h} (L_{2h})^\frac{1}{2} \right)^\frac{1}{2}.$$  \hspace{1cm} (11)

The last lines follow because, for any $n \times n$ matrix $A$, $||A|| = \sqrt{\rho(A^T A)}$.

Q.E.D.

Lemma 4 The fine and restricted coarse grid operators satisfy the inverse relation

$$(L^h)^{-1} \geq I_{2h}^h (L_{2h})^{-1} I_{2h}^h.$$  

Proof: For any matrix $A$, we have $A^T A \preceq I \iff AA^T \preceq I$. Hence,

$$I = (L_{2h})^\frac{1}{2} L_{2h} (L_{2h})^\frac{1}{2}$$
\[
= (L^{2h})^{-\frac{1}{2}} I_{2h}^h I_{2h}^h (L^{2h})^{-\frac{1}{2}} \\
\Rightarrow I \geq (L^h)^{\frac{1}{2}} I_{2h}^h (L^{2h})^{-\frac{1}{2}} (L^{2h})^{-\frac{1}{2}} I_{2h}^h \leq (L^h)^{\frac{1}{2}} I_{2h}^h (L^{2h})^{-\frac{1}{2}} I_{2h}^h L^h \frac{1}{2} \\
\Rightarrow (L^h)^{-1} \geq I_{2h}^h (L^{2h})^{-1} I_{2h}^h. \tag{12}
\]

Q.E.D.

Lemma 5 There exists a constant \( c \in \mathbb{R}^+ \) such that

\[
I_{2h}^h \frac{1}{\rho(L^h)^2} (L^h)^2 I_{2h}^h \leq c \frac{1}{\rho(L^{2h})} (L^{2h})^2.
\]

Proof: First we observe that \( \rho(L^h) \) scales \( L^h \) so that \( \frac{1}{\rho(L^h)} L^h \leq I \). Thus,

\[
I_{2h}^h \frac{1}{\rho(L^h)^2} (L^h)^2 I_{2h}^h \leq I_{2h}^h \frac{1}{\rho(L^h)} (L^h)^2 I_{2h}^h.
\]

Then, to establish the bound we seek, we need only prove that

\[
I_{2h}^h \frac{1}{\rho(L^h)} (L^h)^2 I_{2h}^h \leq c \frac{1}{\rho(L^{2h})} (L^{2h})^2.
\]

For some constant \( c \in \mathbb{R}^+ \). First notice that,

using Lemma 4, we can choose a constant \( b \in \mathbb{R}^+ \) such that

\[
I \leq b I_{2h}^h T I_{2h}^h \\
\Rightarrow (L^{2h})^{-\frac{1}{2}} \leq b (L^{2h})^{-1} I_{2h}^h T I_{2h}^h (L^{2h})^{-1} \\
\Rightarrow I_{2h}^h (L^{2h})^{-\frac{1}{2}} I_{2h}^h T \leq b \left( I_{2h}^h (L^{2h})^{-1} I_{2h}^h T I_{2h}^h (L^{2h})^{-1} I_{2h}^h T \right) \leq b (L^h)^{\frac{1}{2}} \\
\Rightarrow L^h I_{2h}^h (L^{2h})^{-\frac{1}{2}} I_{2h}^h T L^h \leq b L^h (L^h)^{\frac{1}{2}} = b I \\
\Rightarrow L^h I_{2h}^h (L^{2h})^{-\frac{1}{2}} I_{2h}^h T L^h \leq b I \\
\Rightarrow L^h I_{2h}^h (L^{2h})^{-\frac{1}{2}} I_{2h}^h T L^h \leq b I \\
\Rightarrow (L^h I_{2h}^h (L^{2h})^{-1}) (L^h I_{2h}^h (L^{2h})^{-1})^T \leq b I \\
\Rightarrow (L^h I_{2h}^h (L^{2h})^{-1})^T (L^h I_{2h}^h (L^{2h})^{-1}) \leq b I
\]
\[ L^h I_{2h} (L^h)^{-1} (L^{2h})^{-1} I_{2h}^T L^h \leq b I \]
\[ (L^{2h})^{-1} I_{2h}^h L^h T I_{2h}^h (L^{2h})^{-1} \leq b I \]
\[ I_{2h}^h L^h T I_{2h}^h \leq b (L^{2h})^2. \]  

(13)

Dividing by \( \rho(L^h) \), we have shown that
\[ I_{2h}^h \frac{1}{\rho(L^h)} (L^h)^2 I_{2h}^h \leq b \frac{1}{\rho(L^h)} (L^{2h})^2. \]

We want the smallest value for \( b \) such that
\[ I \leq b I_{2h}^h T I_{2h}^h, \text{ so we let} \]
\[ b = \frac{1}{\lambda_{\min} \left( I_{2h}^h T I_{2h}^h \right)}. \]

(14)

Now we need to find a minimum value for \( c \) so that
\[ \frac{b}{\rho(L^h)} \leq \frac{c}{\rho(L^{2h})}. \]

Letting \( \lambda_{\min}(A) \) denote the smallest positive eigenvalue of a symmetric positive definite matrix \( A \), then the minimum value for \( c \) is evidently
\[ c = \frac{1}{\lambda_{\min} \left( I_{2h}^h T I_{2h}^h \right)} \frac{\rho(L^{2h})}{\rho(L^h)} \]
\[ = \frac{1}{\lambda_{\min} \left( I_{2h}^h T I_{2h}^h \right)} \frac{\rho(I_{2h}^h T L^h I_{2h}^h)}{\rho(L^h)} \]
\[ \leq \frac{1}{\lambda_{\min} \left( I_{2h}^h T I_{2h}^h \right)} \rho(I_{2h}^h T I_{2h}^h) \frac{\rho(L^h)}{\rho(L^h)} \]
\[ = \frac{1}{\lambda_{\min} \left( I_{2h}^h T I_{2h}^h \right)} \rho(I_{2h}^h T I_{2h}^h) \]
\[ = \sigma \left( I_{2h}^h T I_{2h}^h \right). \]  

(15)

where \( \sigma \) denotes the condition number. Hence,

So that,
\[ I_{2h}^h \frac{1}{\rho(L^h)^2} (L^h)^3 I_{2h} \leq \sigma \left( I_{2h}^h T I_{2h}^h \right) \frac{1}{\rho(L^{2h})} (L^{2h})^2, \]
and the lemma is proved.

Q.E.D.

Lemma 6 If $I_{2h}^h$ is based on piecewise bilinear elements on a uniform rectangular grid, then

$$
\sigma \left( \begin{bmatrix} I_{2h}^h & I_{2h}^h \\ I_{2h}^h & I_{2h}^h \end{bmatrix} \right) \leq 4.0.
$$

Proof: The stencil for the bilinear interpolation operator $I_{2h}^h$ is given by

$$
\begin{array}{ccc}
\frac{1}{16} & \frac{1}{8} & \frac{1}{16} \\
\frac{1}{8} & \frac{1}{4} & \frac{1}{8} \\
\frac{1}{16} & \frac{1}{8} & \frac{1}{16}
\end{array}
$$

The stencil for the product $I_{2h}^h I_{2h}^h$ is therefore

$$
\begin{array}{ccc}
\frac{1}{16} & \frac{1}{8} & \frac{1}{16} \\
\frac{3}{8} & \frac{3}{4} & \frac{3}{8} \\
\frac{1}{16} & \frac{1}{8} & \frac{1}{16}
\end{array}
$$

the lemma now follows from a straight-forward mode analysis to estimate the eigenvalues of this stencil operator.

Q.E.D.

The following theorem establishes the AFACx converge in the usual "optimal" multilevel sense whenever AFAC does.

Theorem 1 The spectral radius of AFACx is bounded below one uniformly in $h$, assuming this is true of AFAC and that $n$ is sufficiently large.

Proof: Lemmas 3 and 5 combine to prove

$$
\begin{align*}
\| \mathcal{P}^h_n \| & \leq \rho \left( \left( I - \frac{1}{\rho(L^{2h})} L^{2h} \right) \right)^n \left( I - \frac{1}{\rho(L^{2h})} L^{2h} \right)^{n/2} \\
& \leq \rho \left( \left( I - \frac{1}{\rho(L^{2h})} L^{2h} \right) \left( I - \frac{1}{\rho(L^{2h})} L^{2h} \right)^{n/2} \right)^{1/2}
\end{align*}
$$
\[
\begin{align*}
&= \left\| \frac{\sqrt{c}}{\rho(L^{2h})^{\frac{1}{2}}} (L^{2h})^{\frac{1}{2}} \left( I - \frac{1}{\rho(L^{2h})} L^{2h} \right)^n \right\| \\
&\leq \sqrt{c} \max_{0 \leq \beta \leq 1} \left| \beta^\frac{1}{2} (1 - \beta)^n \right| \text{ where } \beta \text{ is any eigenvalue of } \frac{1}{\rho(L^{2h})} L^{2h} \\
&= \sqrt{\frac{c}{2n + 1}}. 
\end{align*}
\]  

From Lemma 2, we thus have

\[
\|[AFACxe^e]\| = \|[AFACe^e + I_n^h P_n e^2h]\| \leq \|[AFACe^e]\| + \|[I_n^h P_n e^2h]\| \\
\leq \|[AFAC]\| \cdot \|e^e\| + \|[P_n e^2h]\| \\
\leq \left( \|[AFAC]\| + \sqrt{\frac{c}{2n + 1}} \right) \|e^e\|. 
\]  

Q.E.D.
CHAPTER 3
PARALLEL SOFTWARE DESIGN

3.1 Introduction

The design and implementation of parallel software is a hindrance to obtaining feedback on the design of parallel numerical algorithms. An important part of this design involves implementation of the proposed algorithms on the complex target architectures used on today's computers and the feedback of these results into the design of the algorithm. An understanding of the numerical properties of the parallel algorithm can be obtained from a serial implementation, but issues of parallel performance show up only in the much more complex parallel environment, though often some details of performance on a proposed parallel architecture can be estimated with a good understanding of the algorithm.

Based on the design and development of much parallel software, a recurring set of problems was recognized as fundamental to the expanded development of software as complex as the parallel adaptive mesh refinement (AMR) codes. The initial development of a parallel adaptive refinement code was complex enough to prevent its expanded use on much more realistic fluid applications. As a result, a portion of the thesis research effort was spent in the analysis and resolution of these road blocks to efficient and productive software design.

This thesis implements two separate parallel adaptive refinement codes, one in C and one in C++ (using the object-oriented design features of C++). The C language version, which was completed first and served as motivation for the C++ version, and earlier FORTRAN work showed the substantial difficulties involved in the use of FORTRAN or
any other procedural language for parallel adaptive refinement. Motivated by these observations a new way to develop general parallel software, and specifically parallel adaptive mesh refinement software, was designed and is presented in what follows.

3.2 C Language Implementation of Parallel AFAC/FAC

The initial implementation of the general problem design was mostly carried into the second object-oriented AMR++/P++ version, though the AMR++/P++ implementation was substantially more robust and feature laden (see section 3.6).

3.2.1 Program Structure  A decomposition of the composite grid problem domain is commonly used to partition work across multiple processors. However, since AFAC requires minimal intergrid transfers, additional solver efficiency (the dominant cost) is obtained by partitioning the composite grid by level. A partition of the problem domain might cut across many grids and add substantially to the total communication cost involved in updating internal boundary values, but a partition by level means that the grids will be shared across a minimal number of processors. This reduces and simplifies necessary communication between processors sharing a given grid, which is especially effective since most message traffic occurs during the MG solves. In addition, level partitioning allows for a more simplified load balancing strategy. An even greater advantage is that it allows for the movement of grids, as required by shock tracking, with no movement or rebalancing of the distributed composite grid. Further, level partitioning allows for a greater amount of each grid to be stored in the processors that share it. This results in longer vectors to be formed by the solvers, which is expected to better utilize the vector and pipeline hardware on machines with these features.

With this reduction in total communication requirements, the existing communication costs can be more easily hidden by the computational work. This sort of message latency hiding would be expected to appear best when there is special message passing hardware designed to relieve the main CPU of the message passing overhead. Due to the
unbalanced communication-to-computation costs associated with the iPSC/1 and its lack of special communication hardware, this message latency hiding was, however, difficult to measure.

3.2.2 Data Structures  The relationship between the levels of the composite grid is expressed in a tree of arbitrary branching factor at each vertex. Each grid exists as a data structure at one node of this tree. This composite grid tree is replicated in each node. In the case of a change to the composite grid (adding, deleting, or moving a grid), this change is communicated globally to all processors so that the representation of the composite grid in each processor is consistent. The partitioning of all grids is also recorded and is consistent in all processors.

Storage of the matrix values uses a one-dimensional array of pointers to row values. These rows are allocated and deallocated dynamically, allowing the partitioned matrices to be repartitioned without recopying the entire matrix. This is important to the efficiency of the dynamic load balancer, MLB. The effect of noncontiguous matrices is not felt when using this data structure since subscript computation is done by pointer indirection for all but the last subscript and, in this case, subscript computation is done by the addition of an offset. This organization is particularly effective for partitioning along one dimension.

In the case of a 3D problem space, 2D planes would be dynamically allocated. Multiple rows or planes could also be allocated, allowing choices of vector lengths to optimize the implementation on vector machines.

The newer implementation using P++ does not have to address this level of detail since such issues as storage and data layout are a part of P++ and the abstraction that it represents.

3.2.3 Multigrid Solver  A significant change to the code described in Briggs et al. [10] is the restriction to one-dimensional decomposition ("strips") and the allowance of irregular partitioning. The experiments documented here use (2,1) V-cycles that recurse to
the coarsest grid, which consists of one interior point. The AFAC scheduler is responsible for ordering the execution of the necessary multigrid and intergrid transfer subroutines based on the receipt of messages. For example, multigrid is executed for all grids asynchronously and is driven by the order in which the messages are received.

3.2.4 AFAC/FAC Scheduler The AFAC scheduler handles the asynchronous scheduling of the steps needed to perform an AFAC iteration. An AFAC iteration is divided into operations between communications. The scheduler orders these operations on each of the grids contained in each processor. Thus, it is intended that much communication would be hidden by the computation that is scheduled while waiting for messages (message latency hiding).

3.2.5 Grid Manager The grid manager is responsible for the creation of the grid data structures and the update of the composite grid tree in all processors. Calls to the grid manager allow for the passing of internal boundary values between processors and the adjustment of the partitions as called for by MLB. Additional services provided are modification of the composite grid by the addition of a new refinement, as required of an adaptive method, and movement of any grid or stack of grids, as required in shock tracking.

3.2.6 Multilevel Load Balancer (MLB) MLB is responsible for the dynamic readjustment of the evolving composite grid. As new grids are built, the composite grid is modified, its tree is updated in all processors, and the partitions are adjusted to balance the loads. Given the data structures used for the storage of the matrices (outlined previously), MLB can adjust a partition at a cost commensurate with the amount of data transferred between processors. Additionally, MLB assures that the data transferred between processors during partitioning follows a minimum length path. Further, since the cost of determining if a multiprocessor system requires balancing (a global communication) is approximately the cost of MLB when no partitions are changed, there is a negligible penalty for the frequent load balancing required in dynamic refinement.
3.2.7 Data Flow  The design of the solver allows for progress on a given grid to be divided into 27 computationally equal parts. After each part is finished, all shared grids are checked for the receipt of boundary values (messages) from co-owning processors. All shared grids are serviced in a round-robin fashion, but are checked for receipt of boundary values before any wholly owned grid is processed. This gives the shared grids a higher priority than the wholly owned grids.

Using the solver in this way allows for good processor utilization. When used with the load balancer, the total-solve times vary only a few percent between processors. Thus, processor utilization during the most costly part of APAC is quite high. Further, the order of execution is both dynamic and nearly optimal, since the work done on each processor is driven by the availability (receipt) of messages.

A significant improvement in this context would be to increase the fineness of grain in the parallelism available. The current graininess in the parallelism of the solver in each processor depends on the size of the grids owned. With a very coarse grain of parallelism, the receipt of messages during the solve of a large grid does not trigger the servicing of the grid whose boundary values were just received. Thus, the execution (servicing) of the shared grids is not handled optimally. The remedy is to partition these large grids into smaller pieces and thus reduce the time a shared grid waits for service while processors finish the larger grids.

A commonly suggested optimization for the organization of message-passing in the parallel environment allows relaxation on the overlap (ghost) boundary and then triggers message passing on that overlap while relaxation is done on the interior. The motivation is to trigger the message passing as soon possible and then use the interior relaxation to hide the latency associated with the communications. Contrary to common understanding, the effect of this optimization was only a few percent improvement on the larger problems run and about 10% additional overhead on the smaller problems. However, using the iPSC/2
asynchronous communication calls means that the results are inconclusive since the iPSC/2 hardware only supports very limited overlap in computation and communication. The results are detailed in figure 2. Note that this sort of message latency hiding could be handled transparently to the user in the object-oriented C++, though it is not done currently.

3.3 Problems with Computer Languages

This thesis explores an object-oriented design for complex parallel numerical software. This line of research was discovered after having made several attempts at the design of parallel adaptive mesh refinement codes for realistic applications using complicated non-rectangular refinement grids. The experience was useful in discovering just a few of the very wrong ways to implement adaptive refinement software. Initial work on block structured grids was unsuccessful mostly because of the lack of the C language's ability to support encapsulation.

3.3.1 Problems with FORTRAN

FORTRAN is a static procedural language, and as a result has limited flexibility to handle the necessarily dynamic requirements of adaptive mesh refinement. In a parallel environment, even static refinement would be problematic since, without special support, the shuffling of data between processors as new refinement is added would require recopying large amounts of data. Memory management in general becomes a practical limitation of FORTRAN for such complex software. Additionally, the details of adaptive mesh refinement, application requirements, and parallelism can become unnecessarily mixed because of the lack of the data hiding (the ability to hide the organization and access of internal data) and encapsulation (the ability to hide the internal manipulation). FORTRAN 90 addresses many of these issues, but is mostly unavailable. High Performance FORTRAN (HPF) ignores most of the features of FORTRAN 90 that might simplify adaptive mesh refinement, specifically the object-oriented-type features of FORTRAN 90 (e.g., operator overloading).

Additional problems with FORTRAN:
Pass 1st or Not

- Prepass Boundaries
- Don't Prepass

Figure 2: Effect of overlapping communication with computation.
• Type checking is an important requirement in the development of large complex software. The use of user defined types in C and C++ significantly reduces the debugging time required for adaptive refinement codes because the complexity of the different data structures can be isolated, separated, and represented as different types (by the use of user defined structures). In the resulting implementation, type checking verifies a consistent expression of the algorithm implementation. This type checking is stronger in C++, and provides more advanced capabilities in the object-oriented design since there is greater expressiveness in the definition of objects that combine the organization of data, as in the C language "structures" and the method functions that manipulate the object's data. For a more complete definition of the C++ language, see [47].

• Dynamic Memory Management is another important requirement for parallel adaptive mesh refinement. Alternatively, the use of a memory management layer between the parallel adaptive mesh refinement and the FORTRAN language can provide the required support. The advantage of using existing FORTRAN code and the efficiency that FORTRAN presents at the lowest levels can make FORTRAN deceptively attractive. More common approaches have mixed C++ and FORTRAN so that the advantages leveraged from the use of the object-oriented design can be exploited. This thesis has not taken a mixed language approach since the object-oriented design is required at a high level to implement the AMR code and at a low level (the level of the looping constructs) to provide the architecture independence.

• The use of common blocks in FORTRAN does not adequately isolate the data, or its internal structure (partitioned or contiguous), away from the statements that manipulate the data. The effect complicates the development and maintenance of complex codes and especially complicates the porting of codes designed for the serial environment to the parallel environment. Many codes that solve explicit equations
are sufficiently simple that they do not have such problems. Similarly, the definition of standards for libraries is sometimes reduced to the awkward standardization of common block variable orderings, unnecessary in more advanced languages.

3.3.2 Problems with Procedural Languages  The fundamental problem that was experienced in the implementation of the C language parallel adaptive refinement codes was the overwhelming complexity of combining the application specific problem with the adaptive mesh refinement and the explicit parallelism for the distributed memory environment. Each would have been tractable individually, but these software engineering problems combine nonlinearly. This is nothing more than the statement that the time requirements of software development in general are nonlinear in the number of lines of code.

The solution to this problem starts with the recognition that an algorithm can be expressed independent of the organization of its data. For example, the addition of the arrays could be expressed in either FORTRAN or C (or C++) as in figure 3.

FORTRAN
DO i=0, Size
DO j=0, Size
   A(i,j) = B(i-1,j) + B(i+1,j) + B(i,j-1) + B(i,j+1)
END DO
END DO

C or C++
for (int i=0; i < Size; i++)
   for (int j=0; j < Size; j++)

Figure 3: FORTRAN and C of C++ example code fragments.

Both the FORTRAN and C or C++ versions of this statement implicitly rely on the contiguous ordering of the data in the arrays A, B, and C. This is due to the definition of the indexing operators, ( ) in FORTRAN and [ ] in C and C++. The reliance on the contiguous ordering of the data means that the algorithm’s expression is NOT independent of the organization of the data.

A result of this dependence of the algorithm’s implementation on the organization of its data is that a change in the layout of the data affects the expression and implementation
of the algorithm. In the case of a vector architecture, the data should be organized into contiguous vectors of constant stride so that the vector hardware can efficiently evaluate the algorithm. This is the trivial case of the example implementation, and the traditional style of implementation maps well the low level efficient vector processing. In the case of a cache-based RISC microprocessor architecture, the use of the consecutively ordered multidimensional arrays $A$ and $B$ and the sequential looping force continued flushing of the microprocessor cache. In this case, the implementation gets the correct result but efficiency is sacrificed because the sequential loop processing flushing the cache’s record of the element $B[i][j-1]$ (among others). The solution that enables efficient processing is a block by block processing of the two-dimensional grid, but we clearly see how this modification affects the implementation in figure 4. Further, this block by block processing of the 2D grid is in conflict

```c
for (int Block_i=0; i < Size / Block_Size; i++)
  for (int Block_j=0; j < Size / Block_Size; j++)
    for (int Element_i=1; Element_i < Block_Size; Element_i++)
      for (int Element_j=1; Element_j < Block_Size; Element_j++)
        {
          int i = Element_i + (Block_i + Block_Size);
          int j = Element_j + (Block_j + Block_Size);
        }
```

Figure 4: C Language example for block by block cache based execution.

with the efficient vector processing since utilization of the cache requires many very short vectors to be processed, and efficient vector processing requires longer vectors. Attempts to have such issues be addressed at compile time have been relatively unsuccessful.

The case of the equivalent distributed memory code changes the processing even more drastically since the data is partitioned across multiple processors, and so explicit message passing must be introduced. The resulting implementation is greatly expanded (in lines of code) and the implementation is far from clear because of the required additional parallel logic. Figure 5 shows an example code (showing a simpler case where global addressing is
used at the expense of the whole array's storage on each processor\(^1\) with the equivalent parallel implementation of the previous code fragment.

In all three examples, the implementation of the algorithm is affected by the details of the target architecture. The effect of more sophisticated parallel architectures is most extreme. The use of complex algorithms on such architectures greatly complicates the software development. More specifically, the more complex parallel implementation hides the details of the algorithm's definition and thus precludes the normal development evolution of the software as increasingly complex applications are attempted (e.g., more physics). The effect on software development is to force dual implementations, one serial and simple to modify, extend, and the second parallel and difficult to extend. The practical effect of multiple implementations makes the development of realistic parallel applications expensive and slow, because the algorithm cannot economically be modified for each of several, or perhaps many, different architectures. The fundamental reason for this problem is the dependence of the implementation of the algorithm on the organization of the data. The ability to express algorithms independent of the architecture is thus a principal feature of the software for parallel adaptive refinement work.

### 3.4 Motivation for Object-Oriented Design

Having seen, by example of the Jacobi relaxation code fragment, that the implementation is affected by the target architecture, we want representations of the algorithms that sufficiently abstract the details of each possible architecture. The work on array languages during the late seventies provides just such an abstraction. Here the arrays are manipulated using array operators that internally understand the details of the target architecture, but which by their use permit the implementation of the algorithm independent of the organization of the data. Specifically, we do not know how the arrays are represented internally, but this detail is unimportant to the definition of the algorithm. Thus, the algorithm may be

---

\(^1\)This simplification is done only for clarity since the use of non-global indexing would be less clear.
int SIZE = 100;
int Node_Number_Table[16] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
double Solution[SIZE][SIZE];
double Old_Solution[SIZE][SIZE];

void main()
{
    int i, j;
    int Iterate;
    double Sum = 0.0;
    int Max_Number_of_Iterations = 10;
    int Right_Message_Type = 100; /* Arbitrary values */
    int Left_Message_Type = 101; /* Arbitrary values */

    int Process_ID = myid(); /* Current Process ID */
    long Processor_Number = mynode(); /* The Node ID of this Processor */
    int Number_of_Processors = numnodes(); /* Total number of Processors */

    /* Compute our partition of the distributed Solution array */
    int Start = (SIZE / Number_of_Processors) * Processor_Number;
    int End = (SIZE / Number_of_Processors) * (Processor_Number+1) - 1;

    /* Modify ends of the partition */
    if (Processor_Number == Node_Number_Table[Number_of_Processors-1])
        End--;

    if (Processor_Number == Node_Number_Table[0])
        Start++;

    /* Initialize whole array to zero! */
    for (i=0; i < SIZE; i++)
        for (j=0; j < SIZE; j++)
            {
                Solution[i][j] = 0.0;
                Old_Solution[i][j] = 0.0;
            }

    /* Initialize Interior */
    for (i=1; i < SIZE-1; i++)
        for (j=1; j < SIZE-1; j++)
            Solution[i][j] = 1.0;

    for (Iterate=1; Iterate < Max_Number_of_Iterations; Iterate++)
    {
        /* Assign current processors partition of Solution to Old_Solution.
        We do this to setup the Jacobi iteration */
        for (i=Start-1; i <= End+1; i++)
            for (j=1; j < SIZE-1; j++)
                Old_Solution[i][j] = Solution[i][j];

        /* Do Jacobi relaxation for Laplace Equation */
        for (i=Start; i <= End; i++)
            for (j=1; j < SIZE-1; j++)
                Solution[i][j] = (Old_Solution[i-1][j] + Old_Solution[i+1][j] +
                                  Old_Solution[i][j-1] + Old_Solution[i][j+1]) / 4.0;

        /* Send the new solution on our boundary to the Right and Left Processors! */
        if (Processor_Number < Node_Number_Table[Number_of_Processors-1])
            send (Right_Message_Type, &(Solution[End][0]),
                  SIZE * sizeof(double),
                  Node_Number_Table[Processor_Number+1], Process_ID);
        if (Processor_Number > Node_Number_Table[0])
            send (Left_Message_Type, &(Solution[Start][0]),
                  SIZE * sizeof(double),
                  Node_Number_Table[Processor_Number-1], Process_ID);

        /* Receive the new solution on our boundary from the Right and Left Processors! */
        if (Processor_Number < Node_Number_Table[Number_of_Processors-1])
            recv (Left_Message_Type, &(Solution[Start-1][0]),
                  SIZE * sizeof(double));
        if (Processor_Number > Node_Number_Table[0])
            recv (Right_Message_Type, &(Solution[Start-1][0]),
                  SIZE * sizeof(double));
    }
    printf("Program Terminated Normally!\n");
}

Figure 5: Distributed memory example code.
defined using simple array operations as in figure 6, regardless of the target architecture. In the case of a vector architecture, the array operators process the vectors one at a time\(^2\). In RISC architectures, the internal representation can be processed by block type operations where each block fits into the cache \(^3\).

\begin{verbatim}
Index I (1,Size,1);
Index J (1,Size,1);
A(I,J) = B(I-1,J) + B(I+1,J) + B(I,J-1) + B(I,J+1);
\end{verbatim}

Figure 6: Equivalent P++, object-oriented example code.

Many algorithms are not well suited to the array language implementation, for example, the usual tridiagonal solve which would require explicit scalar indexing\(^4\). However, such algorithms are not parallel or vectorizable, so the array language only encourages the use of algorithms that are better suited to the more complex architectures available today, without limiting other algorithms. Clearly, the use of a serial algorithm in a parallel environment only results in poor performance, not in an incorrect result. Figure 7 shows an example code fragment of the object-oriented code fragment using the explicit looping and indexing, which is also easily provided.

\begin{verbatim}
for (int i=0; i < Size; i++)
    for (int j=0; j < Size; j++)
        A(i,j) = B(i-1,j) + B(i+1,j) + B(i,j-1) + B(i,j+1);
\end{verbatim}

Figure 7: Equivalent P++, object-oriented example code using explicit indexing.

The use of the Index type specifies a FORTRAN 90 like triplet of integers specifying the initial indexed position, the number of elements to be indexed, and the associated index stride. Then the arrays \(A\) and \(B\) are manipulated using the array operators + and =, which treat only those portions of \(A\) and \(B\) that are indexed using the Index variables \(I\) and \(J\). In this case, the Index variables are themselves manipulated using the + and - operators, so that \(B\) is indexed using combinations of \(I + 1, I - 1, J + 1, \text{and} J - 1\).

\(^2\)In complex array expressions, this has special sorts of inefficiencies that will be discussed later.

\(^3\)This has been worked out by Roldan Pozo at the University of Tennessee as part of the distributed LAPACK++ class library.

\(^4\)Such algorithms must use explicit indexing (using looping and scalar index array operators), which can be easily provided within the definition of the array language, but are not considered array operations.
But the use of an array language would require the construction of a specialized compiler. Such compilers are not commonly available. Though FORTRAN 90 array extensions have been added to many implementations of FORTRAN 77, only a few compilers use them to simplify development of parallel software. The use of a specialized compiler would limit the use of the resulting codes to the few architectures where such compilers might be available. This effectively contradicts the goal of architecture independence. An alternative that would permit the use of array constructs is to build such an array language into an existing language.

A procedural language would not permit the extension of the language without modifying the compiler, since the language defines a static set of types that may not be extended. But an object-oriented language permits the definition of user defined types (objects), called an extensible type system, and such user defined types (objects) behave similar to the static set of types defined in procedural languages. The practical effect is to permit extension of the language to include new user defined types and to have them function as if they were an original part of the language definition.

The C++ language is a relatively new object-oriented language that is widely available for nearly every serial and parallel computer. It is increasingly being used in the development of numerical software across a broad range of computer architectures. Although developed originally in 1984, C++ has stabilized sufficiently to support large implementation projects and is increasingly being used for the development of sophisticated numerical software.

Figure 8 shows an example of an array and index object, illustrating how the data and method functions which manipulate the object's data are combined. The example definition of the Index and Intarray C++ classes define new types to the compiler and code that uses them. The details of the implementation can be specific to a given architecture,

---

5A substantial portion of the engineering codes developed at Sandia National Labs are using C++.
6Several of the largest production codes at Sandia National Laboratory are developed using C++.
class Index
{
    private:
        int Position;
        int Count;
        int Stride;
    public:
        Index & operator + ( const Index & X , int i );
        Index & operator - ( const Index & X , int i );
        Index & operator = ( const Index & X );
}

class intarray
{
    private:
        Array_Descriptor_Type *Array_Descriptor;
        int *Data;
    public:
        Array & operator () ( const int & i );
        Array & operator () ( const int & i , const int & j );
        Array & operator () ( const Index & I );
        Array & operator () ( const Index & I , const Index & J );
        Array & operator + ( const Array & Lhs , const Array & Rhs );
        Array & operator - ( const Array & Lhs , const Array & Rhs );
        Array & operator * ( const Array & Lhs , const Array & Rhs );
        Array & operator / ( const Array & Lhs , const Array & Rhs );
        Array & operator = ( const Array & Rhs );
}

Figure 8: Equivalent P++, object-oriented example code.
but the interface can be constant and is architecture independent. This is the principal way this thesis proposes architecture independence for general numerical software.

This thesis uses C++ to define an array class library. Specifically, an array object is defined in C++ and all possible operators between arrays are provided, so that the array objects appear as built-in types (though user defined). The actual interface is copied from the commercial M++ \(^7\) array language. The motivation for this work is to simplify the use of software implemented using the array interface for distributed memory architectures. In the serial environment, it is sufficient to use the M++ class library. The purpose of the P++ class library is to extend the identical interface for use on distributed memory parallel computers with absolutely no change to the original serial code. The parallel array class library P++ uses the serial array class library internally. The result is a separation between the details of the parallel array object and the serial array object. Such separation divides the complexity of the combined design of the single processor array object and its extended use in the parallel environment. Since the interface is identical, the code using the interface (the original serial source code) does not change. The computational model is to run the same code on each processor, which is called a Single Processor Multiple Data model (SPMD model).

The P++ array class library, which provides general support for data parallelism, is part of the strategy to support parallel adaptive refinement. Its complete support requires support for the adaptive refinement details of adaptive mesh refinement so that different applications can reuse the complex data structures that adaptive mesh refinement codes contain. The common AMR code, independent of the application and parallel issues, is presented in the AMR++ class libraries. In this way, we expect substantial reuse of code across many adaptive refinement applications. More details are contained in [5] and in section 3.6.

\(^7\)M++ is a product of Dyad Software (1-800-366-1573).
3.4.1 Problems with the Object-Oriented C++ Languages  The use of an object-oriented language is no panacea. There are often dozens of ways to implement a give application, which is the strength of the language. Yet, as studies at AT&T have shown, the learning time for C++ is approximately 12-18 months. It is important to note, however, that most people are more productive in C++ after the first few months than they were previously.

The principal problems experienced with C++ have to do with its ability to optimize the resulting code. The use of simple arrays of primitive types (such as int, float, and double) and the compiler's knowledge of these primitive types permit common optimization. For numerical software, such optimization centers around loop optimization, including removal of constant subexpressions and use of registers for temporary storage and accumulation. Such optimization greatly improves the efficiency of the compiled code. The use of non-primitive types, such as C++ objects (effectively user defined types), greatly limits optimization. Within the current line of C++ compilers, optimizers are typically turned off internally if non-primitive types are discovered. The result is a mass of function calls and the loss of register storage for accumulation within inner loops. Such code is inefficient at runtime. The to obtain performance emphasis is on the class library programmer (who are forced to work within C++ at a deeper level than typical users, who generally use such class libraries, in conjunction with C++, in more simple ways). Details of these problems have been discussed in [1], [45], and [46].

3.5 P++, a Parallel Array Class Library

3.5.1 Introduction and Motivation  The current trend in the development of parallel architectures is clearly toward distributed memory designs. This is evident from current product announcements from Intel, Cray, and Thinking Machines, even though the latter two originally had successful machines of competing design. Experience has shown

8This was the conclusion of discussions within the C++ conference on BIX (Byte Information Exchange).
that shared memory machines are easier to program than distributed memory ones and also have autoparallelizing tools that are not available for distributed memory architectures. This is due to the fact that the shared memory programming model permits parallelization independent of the position of data between processors. However, memory access in shared memory machines becomes increasingly problematic as more processors are added; this non-scalability of the shared memory design limits its potential to support the greater computational power demanded of future applications. Within the last few years, approaches of adding local caches to the global memory of shared memory architectures have slightly improved memory access. The problem of efficiently optimizing cache usage, however, is very similar to the data partitioning problem on distributed memory machines, and is not yet satisfactorily resolved. Distributed memory machines require the explicit control of data, for example, data partitioning, in order to obtain the same or better parallel performance as shared memory machines. This explicit control of parallelism, through Message Passing, is difficult to achieve and the result is a dramatically more complicated programming environment.

The lack of a shared memory programming model on distributed memory machines is the fundamental disadvantage of its programming. The availability of portable communication interfaces, and similar programming tools, implemented on many distributed memory machines significantly simplifies the portability among distributed architectures, but does not address the difficulties of the explicit control of parallelism done through the use of message passing. Although even a shared memory parallel programming model in a distributed memory environment, called Virtual Shared Memory, would be advantageous from the point of view of software development, such general attempts have resulted in poor performance, except for an overly restricted set of problems. Newer advances in this area have not yet been evaluated for feasibility and efficiency [15]. Because the general data distribution problem remains unsolved, such Virtual Shared Memory environments attempt to
support parallelism without explicit knowledge of the algorithm's structure and requirements to off-processor data. The resulting voluminous amount of accesses to off-processor memory degrades performance. This is mostly due to unnecessary and often replicated communication start-up-times and/or data transfer of irrelevant data in proximity to relevant data. It seems clear that the support of parallelism requires at least an interpretation of algorithm requirements, so that accesses to off-processor data can be managed efficiently [7].

P++ is a user environment that simplifies the development of efficient parallel programs for large-scale scientific applications. It permits portability across the widest variety of computer architectures. The target machines are distributed memory computers with different types of node architectures (scalar, vector, or superscalar), but the requirements of shared memory computers are also addressed. Such a simplifying environment for the development of software is sorely needed to take advantage of the current and future developments in advanced computational hardware. The P++ environment does this using a standard language, C++, with absolutely no modification of the compiler. For parallel communication, it employs existing widely portable communications libraries. Such an environment allows existing C++ language compilers to be used to develop software in the preferred serial environment, and such software to be efficiently run unchanged in all target environments.

The explicit goal of P++ is the support of advanced computational methods for the solution of large-scale computational problems. For efficiency reasons and simplification, the use of P++ is restricted to the large class of structured grid methods for partial differential equations. Applications using P++ can selectively exploit the added degree of freedom presented by parallel processing by use of an optimization module within the array language interface. A collection of defaults ensures deterministic behavior. In this way, complicated optimization issues of parallel execution may be easily introduced by setting switches within the user interface, which are then interpreted at runtime. Through the use of Virtual Shared
Memory, restricted to array variables (Virtual Shared Grids), issues such as partitioning become matters of optimization, and not criteria for correct execution. Due to the restriction and optimization for structured grids, we expect the same parallel performance as for codes based on the traditionally used explicit Message Passing model. To speed the development of the P++ environment, we use a reasonably efficient, commercially available array language library, M++, developed in C++; the M++ array interface is also used as the P++ array interface.

The internal implementation of P++ is based on the "Single Program Multiple Data stream" (SPMD) programming model. This model is coupled with the "Virtual Shared Grids" (VSG) programming model (section 3.5.5), which is a restriction of general operating system controlled Virtual Shared Memory to all types of structured grids, controlling communication at runtime. The user interface and programming model guarantee that serial codes developed in the P++ environment can be run on parallel distributed memory machines without modification. Because the array syntax is more compact than explicit looping, in the description of the algorithm, the resulting code is easier to implement and debug, even for strictly serial codes, in the serial environment. Moreover, since the SPMD and VSG programming models permit the serial code to be used in a parallel environment, the development of parallel codes using P++ is simpler than the development of serial codes when the explicit looping model is used. This is primarily due to the fact that the VSG model allows the specification of data partitioning to be an optimization, in contrast to most other developments in this area where appropriate data partitioning is essential for the correct execution of the code. P++ employs a default grid partitioning strategy, which can be overridden in several ways specific to the users application.

Recognizing that the acceptance of advanced parallel computers depends on their use by scientists, and not parallel researchers, it is hoped that P++, as a technological advance, will simplify access to these parallel machines. While significantly simplifying our
own work on parallel self-adaptive local refinement methods, it is hoped that P++ will more generally simplify the introduction of structured grid methods for large scale scientific computing onto advanced parallel machines.

3.5.2 Goals of the P++ Development The general goal of the P++ development is to provide a simplified parallel programming environment. In this section, some ideal requirements for a user interface and programming model for distributed memory architectures are stated. These are fulfilled with the P++ environment for a large, but restricted, class of problems (detailed in subsection 3.5.3):

- Algorithm and code development should take place in a serial environment.
- Serial source codes should be able to be compiled and recompiled to run in parallel on distributed architectures without modification.
- Codes should be portable between different serial and parallel architectures (shared and distributed memory machines).
- Vectorization, parallelization, and data partitioning should be hidden from the user, except for optimization switches to which the user has full access and that have meaning only on vector or parallel environments.
- The debugging of parallel software should be greatly simplified. First, this is done if the software is debugged in the serial environment and data parallelism is exploited by recompilation. Second, the object-oriented treatment of the array operations avoids the respecification of index bounds, so one of the most common errors in the implementation of numerical software is eliminated because Index objects can be computed once and reused. Third, the object-oriented design used to build application codes from the P++ data parallel arrays further abstract low level implementation details of the solvers and separate them from the remaining application code.
3.5.3 The P++ Applications Class

The restriction to a simplified class of applications for P++ has allowed focus on a general solution and evaluation of the functionality and performance of the P++ environment by using realistic application codes. Additionally, the restriction to a large but reasonable class of problems assures that P++ is not exaggerated to overly general or particularly pathological cases. In addition, extending the generality more toward general Virtual Shared Memory would cause the performance limiting factors to apply also to P++. The use of P++ is restricted to all different kinds of structured grid oriented problems. Domain decomposition and applications using adaptive block structured local refinement belong to this target class, and the problems introduced by their use in parallel environments has motivated this work. Grids can be constructed with all levels of freedom (e.g., overlapping blocks, adaptively added, deleted, resized, ...), as long as they fulfil the above restrictions. In particular, P++ is dimensionally independent, which allows for its use in one-to-four dimensional problems. Although not yet applied to many applications, the target applications are large-scale computational problems in fluid dynamics and combustion. Specific algorithms with which this work is intended to be tested include:

- Explicit time stepping algorithms (Piecewise Parabolic Method (PPM) for hypersonic flow [4], [5]);
- Standard multigrid algorithms on rectangular domains ([10], [26], [34], [38]);
- Multilevel local refinement algorithms on simple grids ([22], [28], [29], [40]);
- Multilevel adaptive local refinement algorithms on simple grids ([41], [42]);
- Multilevel adaptive local refinement algorithms on block structured grids.

3.5.4 P++ Implementation in Standard C++

C++, an object-oriented language, was chosen as a basis for P++ because of the language's ability to support abstractions, which is fundamental in the development of the user interface that is expected to abstract issues of data parallelism. Some important features of C++ follow (for a definition
and description the reader is referred to [47]); each is obtained from C++ and carries over
directly to the P++ environment, including any combination of C++/P++ use:

- Easy design, development and maintenance of large codes;
- ANSI standard under development;
- Object-oriented features like inheritance, data hiding, and encapsulation;
- Dynamic memory management;
- Strong typing of variables;
- Guaranteed user defined initialization through user defined constructors;
- Operator and function overloading;
- Templates;
- Same efficiency allowed as for C (currently a research area).

Most of these features are not unique to C++, so we do not preclude the use of any other
object-oriented language, such as Smalltalk. But C++ is currently available on a wider
number of machines than any other object-oriented language that suits numerical needs.
Additionally, the C++ language is a superset of the C language, and so all C software is
also available for use with C++. Specifically, this allows for use of distributed memory
communication libraries, like PVM, to be used as easily with C++ as with C.

P++, as currently developed, uses the AT&T C++ Cfront compiler and the Intel
iPSC communications library. In the near future, however, it is planned to make the code
portable through use of EXPRESS or PVM and later also PARMACS (see section 3.5.3).
Since C++ is a superset of C and the communication library is designed for use with C,
these libraries can be easily used from within C++.

3.5.5 The Programming Model of P++ Use of the Single Program Mul-
tiple Data (SPMD) programming model combined with the VSG programming model is
important, since without the combined programming models, the simplified representation
of the parallel program from the serial program would not be practical. Their combined
effect is the principal means by which P++ allows serially developed codes to be efficiently run on distributed memory machines.

3.5.5.1 Single Program Multiple Data (SPMD) Model  In contrast to the explicit host/node programming model, which requires both a host and one or more node programs, the SPMD programming model consists of executing one single program source on all nodes of the parallel system. For example, the suggested programming model of the Intel Delta machine is SPMD. This is becoming increasingly common in new generation parallel machines from Intel, Cray, and TMC.

Implementation of the SPMD model requires that commonly available functionality in the serial environment be provided in the parallel environment in such a way that the serial source code can be used on the distributed memory machine. One of the most important functionalities that is provided in the parallel programming model to support basic functionality of the serial code is a parallel I/O system. This can then be used in place of the serial I/O system, to support the required functionality of the parallel SPMD programming environment.

Currently, only basic functionality of the SPMD programming model (I/O system: printf, scanf; initialization and termination of processes) is available. Implementation details are abstracted from the user. The SPMD programming model replicates the functionality of the traditional parallel host/node programming model. For example, the standard function scanf for reading from standard input is implemented in such a way that an arbitrarily chosen master node reads the data from standard input and distributes it to all other nodes (slave nodes). This master/slave relationship is only present within the Parallel I/O System and not used, or intended, elsewhere in P++.

3.5.5.2 Virtual Shared Grids (VSG) Model  The concept of Virtual Shared Grids gives the appearance of Virtual Shared Memory restricted to array variables. Computations are based on global indexing. Communication patterns are derived at runtime,
and the appropriate send and receive messages are automatically generated by P++. In contrast to Virtual Shared Memory, where the operating system does the communication without having information about the algorithm's data and structure, the array syntax of P++ provides the means for the user to express details of the algorithm and data structure to the compiler and runtime system. This guarantees that the number of communications and the amount of communicated data are minimized. Through the restriction to structured grids, the same kind and amount of communication as with the explicit Message Passing model is sent/received and, therefore, also approximately the same efficiency is achieved. This is a tremendous advantage over the more general Virtual Shared Memory model.

The amount and size of communication are further minimized by the ability of the user to override the default partitioning. Specifically, Virtual Shared Grids allow the treatment of partitioning as a parameter in the optimization. This is an important feature of the VSG model since it permits serial applications to be run correctly and to exploit data parallelism inherent in their array expressions, and so still permits the auxiliary description of data organization after the code is running. This greatly simplifies decisions about data partitioning, which is the singular additional degree of freedom in the designing of the data parallel implementation. Note that the data parallel implementation might not be sufficient, but is a common component in the design of numerical software. For example, data parallelism is the principal part of the model for a single grid iterative solvers (including multigrid solvers), but is not sufficient for optimal parallel performance using composite grid solvers (including FAC, AFAC, and AFACx) because additional functional parallelism can be exploited in these more sophisticated solvers.

There are two basic communication models that are currently implemented in P++ (how these models interact is described in more detail in the examples in section 3.5.10):

- VSG Update:

  The Owner Computes Rule is a common rule for the evaluation of expressions in
the parallel environment. It actually has several conflicting definitions, but generally means that an owner is defined (usually by the processor owning the Lhs of a \( \text{Lhs} = \text{Rhs} \) expression) and the required \( \text{Rhs} \) arguments are sent to the "owning" processor. Finally, the relevant part of the expression is evaluated on the owning processor (no communication is required for the evaluation step, but there is potentially more communication required to send the relevant parts of the \( \text{Rhs} \) to the owner). In the implementation of the communication model for the general Virtual Shared Grids concept, this classical Owner Computes rule is restricted. Instead, what might be applied to the whole expression is applied instead to the binary subexpression, where the Owner is defined arbitrarily to be the left operand. This simple rule handles the communication required in a parallel environment; specifically, the designated owner of the left operand receives all parts of the distributed array necessary to perform the given binary operation (without further communication). Thus, the temporary result and the left operand are partitioned similarly (see figure 9).

- **Overlap Update:**

  In order to provide high performance for a broad range of iterative methods that would use the VSG programming model, as a specific optimization there is an implemented nearest neighbor access to array elements through the widely used technique of grid partitioning with overlap (currently restricted to width one; see figure 10). In this way, the most common type of array accesses can be handled by complicated expressions where communication in the parallel environment is limited to one overlap update that occurs in the definition of the \( '=' \) sign defined (overloaded) for the arrays. The necessity of updating the overlapping boundaries, based on whether the overlap has been modified after the preceding update, is detected at runtime. Thus, communication in the parallel environment is minimized.

  Virtual Shared Grids are constructed in a distributed fashion across the processors
* P++ user level:
  A = B + C
* P++ internal execution:
  1. \( T = B + C \)

\[ \begin{array}{c}
  P1: T11 = B11 + C1  \\
  \quad \text{receive C21 from P2}  \\
  \quad T12 = B12 + C21  \\
  P2: T2 = B2 + C22  \\
  \quad \text{send C21 to P1}  \\
  P3: \text{idle} \\
\end{array} \]

2. \( A = T \)

\[ \begin{array}{c}
  P1: \text{send T1 to P3}  \\
  P2: \text{send T2 to P3}  \\
  P3: \text{receive T1 from P1}  \\
  \quad \text{receive T2 from P2}  \\
  \quad A = T \\
\end{array} \]

Figure 9. An example for VSG Update based on the Owner computes rule: \( A = B + C \) on 3 processors.

Figure 10: The standard method of grid partitioning with overlap.
of the parallel system. Partitioning information and processor mapping are stored in a partitioning table (part of the DataManager object). This partitioning table basically contains processor numbers that own the Virtual Shared Grids, and the local and global positions and sizes of the partitions. Functions are made available that abstract the details of access queries of the table's data.

All information required for evaluating expressions using VSG is expressed through the array syntax and the use of the partitioning table. The numbers of entries in the table is reduced by grouping associated grids and providing applications support for storage of only the required data on a processor by processor basis. This is necessary due to the large sizes that these tables can be in massively parallel systems. The table is efficiently implemented through a combination of arrays and linked lists. Thus, all necessary global information can be made locally available (even on massively parallel systems of 1000+ processors). Thus, access to global information about the partitioned arrays requires no communication and contributes insignificantly to memory overhead.

A simple mechanism is provided to interrogate the communication pattern between pairs of VSGs at runtime. This mechanism looks at the availability of data in the adjacent processors that are required to satisfy the specific instance of the distributed binary operation. If necessary, it triggers communication to send or receive the required pieces of the array operands (VSG Update) on the basis of the Owner Computes rule. In this way, whole buffers of data are known at runtime to be required, and no sophisticated loop analysis is needed to recover this information. Thus, there is no need for costly element by element communication. Compiler analysis would be required to do such analysis since it would be prohibitively expensive at runtime.

In addition to the two basic communication models of VSG Update and Overlap Update, possible enhancements include:
- **Communication Pattern Caching (CPC):** This would permit the communication patterns, once computed for an array statement, to be cached and recovered for further use. Thus, the determination of the array partition relationships (communication pattern) specific to a particular array statement's indexing could be handled with minimal overhead. Note that CPC could be used across multiple array statements since, within an application, we expect that many different array statements would require, and thus could reuse, the same communication patterns.

- **Deferred Evaluation:** This more complicated evaluation rule allows for significant optimization of the given expression so that communication associated with correct evaluation in the parallel environment can be minimized. In a vector environment, the given operations that form the array expression are optionally collapsed to form aggregate operators and provide for the aggregate operators' implementation in FORTRAN, so that the expression can fully exploit the vector hardware (e.g., through chaining). The use of deferred evaluation of the array statements (also called lazy evaluation) permits the full expression to be know at runtime before evaluation. The evaluation can even be deferred across many array statements (problems are encountered across conditional statements that have dependencies on the results of the deferred array statements, though this might be solved through some compiler support for deferred evaluation). Currently, this principle has been implemented but not yet evaluated for single nodes of vector machines (e.g., a Cray Y-MP in collaboration with Sandia National Laboratories). In particular, the efficient use of chaining and the optimization of memory access are addressed. It is planned to further pursue this approach and fully exploit it into the VSG programming model of P++.

While there are other complicated reasons for the use of Deferred Evaluation, those mentioned above are only some uses specific to the vector environment. Other uses include the deferred evaluation over large numbers of statements and the grouping
of blocks of these statements, based on runtime dependency analysis, so that each block can be executed independently. Such a system would permit the recognition of functional levels of parallelism at runtime. Note that runtime dependency analysis would be based on the use of hash tables and collision detection within these separate hash tables for the Lhs and Rhs operands in each array statement.

3.5.6 The Object-Oriented Design of P++

The basic structure of the P++ code consists of multiple interconnected classes. The overall structure and the interconnections are illustrated in figure 11. The following types of objects are common to the M++ array interface (see also section 3.5.7) and within P++ form a similar user interface:

- `<type>._VSG_Array`:

  `<type>` specifies the type of array elements (currently restricted to float (64 bit) and integer). Dimensional independence up to four dimensions is realized. 1D default partitioning information is stored in the Data_Manager.

- `VSG_Index`:

  Only simple index objects are provided. Each stores: Position, Count, and Stride. Member functions for index objects include set and arithmetic operations. Some typical examples for the use of indexes are addressing grid boundaries or interiors.

  The `<type>._VSG_Array` uses internally the M++ array object `<type>._Array` and copies most of the interface member functions of the M++ array object `<type>._Array`. In this way, the `<type>._VSG_Array` uses the same member functions of `<type>._Array`. So the interface is the same and the numerical software developed in the serial environment executes in a data parallel mode in the multiprocessor environment.

  The following object is specific to the P++ interface (see also section 3.5.7) and is also seen by the user:

- `Optimization_Manager`:

  User control for details of parallel execution.
Figure 11: The object-oriented design of the P++ environment.
The following objects are hidden from the user and represent the notable features of the underlying organization:

- **Data_Manager**:
  All partitioning data is stored in tables. Member functions allow interrogation of the Data_Manager to find the processor associated with any part of any object of type `<type>\_VSG\_Array`, etc.

- **Communication_Manager**:
  These member functions are the only functions that allow access to send/receive and other primitives of the communications library and diagnostics environment (Intel, EXPRESS, PARMACS; see section 3.5.8). Access to constants relative to parallel execution, e.g., number of processors, is also available.

- **Diagnostic_Manager**:
  This class has restricted flow trace capabilities (number of constructor/destructor calls) and also gathers statistical properties of the background communication (e.g., number of messages sent/received).

- **Parallel_I/O_Manager**:
  Standard I/O functions overloaded with versions for use in the parallel environment (e.g. `printf`, `scanf`). Currently, all I/O is processed through a single processor in a master slave relationship. File I/O is not handled, though it is a critical requirement of the large scale computational target applications. We hope to use existing file I/O packages for simplification.

**3.5.7 The P++ User Interface**

The P++ interface consists of a combination of the commercially available M++ array language interface and a set of functions for parallel optimization, the Optimization Manager. The P++ user interface provides for switching between M++ in the serial environment to P++ in the serial or parallel environment.
3.5.7.1 The M++ array class library  The commercially available M++ array class library (from Dyad Software Corp.) is used to simplify the software development. The M++ interface is modified only slightly; we consider the modifications to be bug fixes. The array interface provides a natural way to express the problem and data structure for structured grid applications; additionally, the syntax of the interface permits a natural exploitation of the parallelism represented within expressions used for structured grid applications (because single no execution ordering is assumed). By using M++ within P++, the details of serial vs. parallel interpretation of the array statements are separated. It is hoped that since the internal restrictions to the structured grid work are mostly contained in M++, the move to support unstructured grids, in the future, will be separable and simplified.

The serial M++ interface allows dimensionally independent arrays to be dynamically created and manipulated with standard operators, and subarrays to be defined by indexing. In addition, it has optional array bounds checking. At runtime, the explicit loop hides the data's organization and operation structure, whereas the equivalent array expression may have its data's organization and operation structure interpreted. Importantly, within an array expression, there is no data dependency by definition. In fact, the array language represents a simplification in the design, implementation, and maintenance of structured grid codes. The functionality of the M++ interface is similar to the array features of FORTRAN 90 (see figure 12). In the current version of P++, only a restricted set of data types (integer and float arrays) is implemented. However, complete sets of operators are provided.

We feel that the choice of C++ as the programming language and M++ as an array interface, made to provide as much information about the problem and structure of the data as possible, is strategic in providing a solid base for the parallel array language support for the target numerical problem class. It is especially strategic for support of parallel adaptive mesh refinement, since the adaptive nature of the application means insufficient information
#include "header.h"
#ifndef PPP
#define doubleArray double_VSG_Array
#define Index VSG_Index
#endif

void MacCormack (Index I, double Time_Step, doubleArray & F, doubleArray & U_NEW, doubleArray &U_OLD)
{
    // array expression:
    F = (U_OLD * U_OLD) / 2;
    // scalar expression:
    U_NEW (0) = U_OLD (0) - Time_Step * (F (1) - F (0));
    // indexed array expression:
    U_NEW (I) = U_OLD (I) - Time_Step * (F (I+1) - F (I));
    // array expression:
    F = (U_NEW * U_NEW) / 2;
    // indexed array expression:
    U_NEW (I) = 0.5 * (U_OLD (I) + U_NEW (I)) - 0.5 * Time_Step * (F (I) - F (I-1));
}

void main()
{
    int N;
    double Time_Step;
    scanf ("N", &Time_Step);
    doubleArray U_OLD (N,N),U_NEW (N,N), F(N,N);
    // Setup data
    ...
    int Interior_Start_Position = 1;
    int Interior_Count = N-2;
    int Interior_Stride = 1;

    Index Interior (Interior_Start_Position, Interior_Count, Interior_Stride);
    MacCormack (Interior, Time_Step, F, U_NEW, U_OLD);
}

Figure 12: C++ / M++ / P++ example code: MacCormack (Hyperbolic) Scheme.
at compile time about the partitioning of the data (the composite grid). Such partitioning is defined only at runtime, so the communication patterns must be interpreted then. Similar runtime support is an accepted and required part of any attempt to provide parallel compiler support for complex applications, though, with compiler support, additional efficiency might be possible even for the runtime support (such issues are under investigation; see [2], [7], [11], [12], [24], [25]).

3.5.7.2 The P++ Optimization Manager
The Optimization Manager allows for override of defaults for user control of partitioning, communication, array to processor mappings, communication models of Virtual Shared Grids, parallel I/O system, etc. Optimizations of this kind only have significance in a parallel environment. The Optimization Manager is the only means by which the user can affect the parallel behavior of the code. The Optimization Manager provides a consistent means of tailoring the parallel execution and performance. It provides the user with four types of partitioning for the array (grid variable) data:

- **Default partitioning**: This involves even partitioning of each grid variable across all nodes, based on 1D partitioning of the last array dimension (see figure 2).

- **Associated partitioning**: Grid variables are partitioned consistent with others. This strategy provides for same size or coarser grid construction in multigrid algorithms, but also has general use.

- **User defined partitioning**: A mapping structure is used to construct user defined partitioning.

- **Application based partitioning**: This allows for the introduction of user specified load balancing algorithms to handle the partitioning of one or more specified grid variables.

Currently, the functionality of the Optimization Manager is restricted in its support for above partitioning strategies as required for the examples in section 3.5.10.
3.5.8 Portability and Target Architectures of P++

The target architectures of P++ are all existing and evolving distributed memory multiprocessor architectures, including the Intel PARAGON and iPSC/860, the Connection Machine 5, the nCUBE 2, the coming distributed memory Cray machine, and all kinds of networks of workstations (such as of Suns or IBM RS-6000s). P++ requires only a C++ compiler or C++ to C translator, which have begun to become widely available (e.g., AT&T C front compiler), and a portable parallel communications library, such as PVM or Express. The current P++ implementation uses the Intel iPSC communications library. For the near future, however, it is planned to base P++ independently on the EXPRESS and PARMACS parallel communications environments, guaranteeing portability of P++ across all of the architectures for which these environments are available. Experiences in the past have shown that one or the other will be implemented on all machines of this type shortly after they become available. Due to implementations of PARMACS and EXPRESS for several shared memory architectures, P++ will also be available for this class of machine which significantly simplifies support for shared memory machines.

Since C++ is a superset of C and PVM, EXPRESS, and PARMACS support C, each can be used within P++. PVM, PARMACS, and EXPRESS are described in more detail:

- **PVM**

PVM is a public domain programming environment for the development of parallel applications and provides a vendor independent layer for communication support. Its latest release is similar to the proposed Message Passing Interface standard (MPI).

- **EXPRESS**

EXPRESS from ParaSoft Corp. is a programming environment for the development of parallel FORTRAN and C programs. EXPRESS is available for a variety of different distributed memory architectures (Intel iPSC, nCUBE, ...) and for networks...
of workstations (Sun, IBM RS 6000, Silicon Graphics, ...). In addition to allowing distributed memory codes to also run in a shared memory environment, it is also available for some shared memory multiprocessor architectures (Cray Y-MP, IBM 3090/AIX). Besides a communications library, it contains modules for parallel I/O, a graphics system, performance analysis tools, etc.

- **PARMACS**

PARMACS (ANL/GMD Macros), which is a joint development of the German National Laboratory for Computer Science and Applied Mathematics (GMD) and Argonne National Laboratory [8], is marketed through PALLAS GmbH. PARMACS is a process programming model for FORTRAN, based on macros (expanded by a standard Unix Macro expander). A C version is planned for the near future. PARMACS basically contains macros for process initialization, communication, etc., and is available for the Intel iPSC, nCUBE, Meiko, SUPRENUM, Parsytec Megaframe, and Sun and IBM RS-6000 networks of workstations. In addition, implementations for the shared memory architectures Alliant FX 28, Sequent Balance 2000, and Encore Multimax exist. As the PALLAS Performance Analysis tools (PA-Tools) are based on PARMACS, they also become available for use within P++.

Additional work must be done to support the new distributed memory machines with vector or superscalar nodes based on a vector processing model. This work requires incorporation of the P++ design with the recent work on vectorization of the C++ array language done in collaboration with Sandia National Laboratories. Additional optimization could be done and is planned to eventually support the shared memory class of machine by development of a shared memory specific version of P++.

**3.5.9 Performance of P++** To date, the only running versions of P++ are on the iPSC/860, the iPSC parallel simulator (running on SUN workstations), and, in serial mode, the SUN and IBM PC. The performance of P++ on the actual hardware is dominated
by the single node performance, because no additional communication is added over the serial implementation, though for specific applications communication in the explicit Message Passing programming model for distributed memory architectures could be better optimized than that which P++ provides automatically. Such optimizations would involve the restriction of message passing, using knowledge about how several array expressions might access memory; or the timing (scheduling) of messages using knowledge of dependencies across several array expressions.

The current implementation could be optimized by analysis over multiple loops, though such multiple array statement analysis is not presently a part of P++. The use of deferred evaluation is required before such work can be done. Then we could expect similar performance as that obtained with the explicit Message Passing programming model for distributed memory architectures, even in the case of highly optimized hand coded communication. Current message passing comparisons are relative to a more naive, not highly optimized, explicit message passing model that does not consider optimization across multiple array expressions. Additional multiple array expression optimization is possible using the P++ Optimization Manager's message passing control, but it is not automated within P++.

The M++ array library serves as a base to provide FORTRAN like performance to P++, but its current performance is about 20%-100% of FORTRAN performance and degrades the single node performance of P++. It is the VSG principle in combination with the Optimization Manager's capability of allowing the user to define efficient partitioning that guarantees amounts of communication nearly identical to the explicit Message Passing programming model. For many applications, better performance may be available, since the simplified P++ development environment allows for greater implementation effort to be spent on the selection of more advanced and faster computational algorithms. It is hoped that this will additionally offset the ability of future P++ work to compete directly with
FORTRAN, though this is a current area of research.

Performance is important, since without efficient execution of the application source code, the effects of the parallel or vector architecture are lessened or lost:

- **Single node performance:** Steps have been taken to optimize the single node performance so that the P++ environment can be accurately assessed. For vector nodes and nodes that are most efficiently programmed through a vector programming model, this work has included application of the C++ array language class libraries on the Cray (through collaboration with Sandia National Laboratories). First results by Sandia National Laboratory concerning performance comparisons with FORTRAN are very promising. With an optimized C++ array class library, about 50% - 90% of the FORTRAN vector performance was achieved for complete codes. The comparison is difficult because the realistic codes could not be readily implemented in FORTRAN to include the more rich physics available in the C++ versions. In some cases, the C++ compiler optimization switches had to be turned off. Such problems are examples of incomplete and often immature C++ compilers, thereby hampering the comparison of FORTRAN and C++ on large realistic software. Similar performance has been demonstrated by P++ on the SUN Sparc, but only on those select P++ array statements chosen for initial evaluation and testing of P++, not complete codes.

- **Parallel system performance:** Secondary to single node performance, parallel performance is mostly affected by the amount of communication introduced. The P++ VSG model optimizes this and introduces no more communication than the explicit Message Passing model. However, currently, no runtime analysis is done across multiple array statements, which might better schedule communication and possibly further minimize the required communication. Such further work would
require deferred evaluation (lazy evaluation). Additional performance could be obtained by caching of the communication patterns and their reuse in multiple array expressions. Such work has not been included in the P++ implementation, but in a part of the P++ research and has been a part of several codes to test these ideas. Work done specific to optimized parallel evaluation of array expressions has been carried out for a number of relevant architectures in [14].

3.5.10 P++ Support for Specific Examples Although P++ is dimensionally independent, most example applications have been 2D; however, a 3D multigrid code has been demonstrated. In figure 13, P++ is demonstrated with a partitioning developed to support multigrid.

3.5.10.1 Standard Multigrid Algorithms on Rectangular Domains Multigrid is a commonly used computational algorithm especially suited to the solution of elliptic partial differential equations. In contrast to single grid methods, multigrid uses several grids of different scale to accelerate the solution process.

The usual way to implement regular multigrid algorithms on a distributed memory system is based on the method of grid partitioning ([10], [34], [38]). The computational domain (grid) is divided into several subgrids that are assigned to parallel processors (see figure 2). The subgrids of the fine grids and the associated subgrids of the coarse grid are assigned to the same processor. Each multigrid component (e.g., relaxation, restriction, and interpolation) can be performed on a subset of the interior points of the subgrid independently (in parallel). Calculation of values at interior boundary points, however, needs the values from neighboring subgrids. Since distributed memory machines have no global address space, these values somehow must be made available. Instead of transferring the values individually at the time they are needed, it is more efficient to have copies of neighboring grid points in the local memory of each processor. Hence, each process contains an overlap area, which has to be updated after each algorithmic step. Because the details of the algorithm on a small
number of points per processor are problematic, agglomeration is one of the strategies that can be used to consolidate the distributed application to a smaller number of processors.

Figure 13. The interaction the Overlap Update and VSG Update concepts for standard multigrid partitioning with coarse level agglomeration.

Figure 14 shows the runtime support from P++ for the interpreted communication patterns of the solver and for the agglomeration strategy used in the parallelization of multigrid. Several variant strategies are possible, but the use of VSG reduces the details of their implementation to defining the fine and coarse grid partitioning. The particular VSG communication models, VSG Update (based on the Owner Computes rule) or Overlap Update, is chosen on the basis of data availability within the partitioned grid variables.

3.5.10.2 Multilevel local refinement algorithms on block-structured grids

As a more complicated example of the flexibility of P++, we demonstrate some of the support within P++ for block structured local refinement ([22], [28], [29], [41], [42]). During the solution of partial differential equations on structured grids, local refinement allows for the solution complexity to depend directly on the complexity of the evolving activity. Specifically, regions local to problem activity are refined. The effect is to provide a discretization specifically tailored to an application's requirements.

Local refinement composite grid methods typically use uniform grids, both global and local, to solve partial differential equations. These methods are known to be highly
Figure 14. The interaction the Overlap Update, VSG Update, and BSG Interface Update concepts for FAC and AFAC partitioning of a block structured locally refined grid.
efficient on scalar or single processor vector computers, due to their effective use of uniform grids and multiple levels of resolution of the solution. On distributed memory multiproce-
sors, such methods benefit from their tendency to create multiple isolated refinement regions, which may be effectively treated in parallel. However, they suffer from the way in which the levels of refinement are treated sequentially in each region. Specifically, in FAC, the finer levels must wait to be processed until the coarse-level approximations have been computed and passed to them; conversely, the coarser levels must wait until the finer level approximations have been computed and used to correct their equations. In contrast, AFAC eliminates this bottleneck of parallelism. Through a simple mechanism used to reduce inter-level dependence, individual refinement levels can be processed by AFAC in parallel. The result is that the convergence factor bound for AFAC is the square root of that for FAC. Therefore, since both AFAC and FAC have roughly the same number of floating point operations, AFAC requires twice the serial computational time as FAC, but AFAC may be much more efficiently parallelized.

Specifically, the local refinement of geometries within regions of activity is not easily done using a single rectangular local refinement patch. In order to better represent geomet-
tries within local activity, we introduce block structured local refinement. The flexibility introduced by block structured local refinement equally applies to block structured global grids. Though elliptic solvers for block structured grids (beyond that of relaxation methods) are not provided for in the current work nor considered in the thesis, it is an important and interesting area of current work by the GMD and others.

For example, in figure 6, the composite grid shows a rectangular domain with a curved front embedded within. Such problems could represent oil reservoir simulation models with simple oil/water fronts or more complicated fluid flow problems with shock fronts. In this specific example, the front is refined with two levels; the first level is represented by grids 2 and 3, the second by grids 4, 5, and 6.
For the parallel environment using FAC, because of the sequential processing of the local refinement levels, the composite grid is partitioned as shown in figure 6. Note that solvers used on the individual grids make use of Overlap Updates provided automatically by P++. The intergrid transfers between local refinement levels rely on VSG Updates, also provided automatically by the P++ environment. Note that P++ support of the block structured local refinement is limited and does not include the block structured grid (BSG) Interface Update, which must be handled within the block structured grid code or library. Underlying support in the parallel environment for the BSG Interface Update is provided by either Overlap Update or VSG Update, or by a combination of the two.

Support from P++ for a partitioning specific to AFAC is similarly provided. The different application specific partitioning (shown in figure 14) naturally invokes automated support from the P++ environment in a slightly different manner. The use of an environment such as P++, which permits the implementation of the algorithms (in this case FAC and AFAC) in the parallel environment independent of the organization of their data, greatly simplifies the software development process since it may be developed in a serial workstation environment where productivity is high and since each algorithm can reuse similar code. This is important because 99.9% of the two algorithm's implementations are similar, even though they are optimized using distinctly different organizations (partitionings) of the composite grids.

3.5.11 Related Research To our knowledge, there is currently no study of architecture-independent programming environments that permit software developed specifically for serial machines to be run on parallel distributed memory architectures, and that use existing commonly available compilers. The most important work done in related areas is as follows (apologies to anyone we might have missed):

- Los Alamos National Laboratories' work on C++ for Scientific Computation [18]: Initial work was done relative to the use of C++ for large scale scientific
computation on vector computers. The work on WAVE++, a CFD code, details the advantages and disadvantages of the use of C++ for scientific computation in general and for vector environments in particular. More recent work has been done combining C++ object-oriented design with adaptive refinement hypersonic applications.

- **Sandia National Laboratories' work on C++ vectorization [45]:** this effort on array languages for C++ shows that 50-90% of the equivalent FORTRAN performance can be attained. Some of the largest laboratory applications codes there are being developed using C++, e.g., RALE++.

- **Paragon [13]:** This is a parallel programming environment for scientific applications (mostly image processing) using Communication Structures. It is also based on C++ and contains concepts similar to P++, but is much more restrictive (though it has been demonstrated on a larger number of computers than P++). This is primarily due to the fact that the concept of communication structures is not as general and powerful as the concept of Virtual Shared Grids in allowing and expressing the view of the distributed memory as a global address space, restricted to specific objects. Additionally, indexing is cumbersome and generally restrictive compared to that of P++, which is borrowed from M++ (whose origins are in Gauss).

- **FORTRAN D [19]:** This work develops the extensive list of different array partitionings done in FORTRAN D (based on FORTRAN 77) for use with FORTRAN 77 (and FORTRAN 90 (FORTRAN 90D)). It does not yet employ any concept similar to Virtual Shared Grids. Use of PARTI (see below) within FORTRAN 90 D, however, is expected. This currently is a point of research.

- **PARTI** (Parallel Runtime Tools at ICASE [7]): It provides runtime support for use with FORTRAN and contains clever means by which FORTRAN loops can be interrogated and existing data parallelism discovered and exploited at runtime.
PARTI is primarily focused on unstructured grids. As opposed to P++, the seamless introduction of the PARTI primitives requires compiler modifications.

- **SUPERNUM** [48]: This is a semiautomatic parallelization tool for distributed architectures on FORTRAN compiler level. The prototype developed within SUPERNUM is restricted to a very specific class of applications.

### 3.6 AMR++, an Adaptive Mesh Refinement Class Library

#### 3.6.1 Introduction

AMR++ is a C++ class library that simplifies the details of building self-adaptive mesh refinement applications. The use of this class library significantly simplifies the construction of local refinement codes for both serial and parallel architectures. AMR++ has been developed in a serial environment using C++ and the M++ array class interface. It runs in a parallel environment because M++ and P++ share the same array interface. Therefore, AMR++ inherits the machine targets of P++ and, thus, has a broad base of architectures on which to execute. The efficiency and performance of AMR++ is mostly dependent on the efficiency of P++ (and, thus, M++) and M++ in the serial and parallel environments, respectively. Together, the P++ and AMR++ class libraries separate the abstractions of local refinement and parallelism to significantly ease the development of parallel adaptive mesh refinement applications in an architecture independent manner.

The AMR++ class library represents work that combines complex numerical, computer science, and engineering application requirements. Therefore, the work naturally involves compromises in its initial development. In the following sections, the features and current restrictions of the AMR++ class library are summarized.

#### 3.6.2 Block Structured Grids — Features and Restrictions

The target grid types of AMR++ are 2D and 3D block structured grids with rectangular or logically rectangular grid blocks. On the one hand, they allow for a very good representation of complex internal geometries that are introduced through local refinement in regions with
increased local activity. This flexibility of local refinement block structured grids equally applies to global block structured grids that allow matching complex external geometries. On the other hand, the restriction to structures of rectangular blocks, as opposed to fully unstructured grids, allows for the application of the VSG programming model of P++ and, therefore, is the foundation for good efficiency and performance in distributed environments, which is one of the major goals of the P++/AMR++ development. Thus, we believe that block structured grids are the best compromise between full generality of the grid structure and efficiency in a distributed parallel environment. The application class forms a broad cross section of important scientific applications.

In figure 15, the global grid is the finest uniformly discretized grid that covers the whole physical domain. Local refinement grids (level \( i + 1 \)) are formed from the global grid (level \( i = 0 \)) or recursively from refinement grids (discretization level \( i \)) by standard refinement with \( h_{i+1} = \frac{1}{2} h_i \) (refinement factor of two) in each coordinate direction. Thus, boundary lines of block structured refinement grids always match grid lines on the underlying discretization level.

The construction of block structured grids in AMR++ has some practical limitations that simplify the design and use of the class libraries. Specifically, grid blocks at the same level of discretization cannot overlap. Block structures are formed by distinct or connected rectangular blocks that share their boundary points (block interfaces) at those places where they adjoin. Thus, a connected region of blocks forms a block structured refinement grid. It is possible that one refinement level consists of more than one disjoint block structured refinement grid. In the dynamic adaptive refinement procedure, refinement grids can be automatically merged if they adjoin each other.

In figure 15(a), an example for a composite grid is illustrated: The composite grid shows a rectangular domain within which we center a curved front and a corner singularity. The grid blocks are ordered lexicographically; the first digit represents the level, the second
digit the connected block structured refinement grid, and the third digit the grid block. Such problems could represent the structure of shock fronts or multi-fluid interfaces in fluid flow applications: In oil reservoir simulations, for example, the front could be an oil-water contact front moving with time and the corner singularity could be a production well. In this specific example, the front is refined with two block structured refinement grids; the first grid on refinement level two is represented by grid blocks 2.1.1 and 2.1.2, and the second grid on level three by grid blocks 3.1.1, 3.1.2, and 3.1.3. In the corner on each of the levels, a single refinement block is introduced.

For ease of implementation, in the AMR++ prototype, the global grid must be uniform. This simplification of the global geometry was necessary in order to be able to concentrate on the major issues of this work, namely, to implement local refinement and self-adaptivity in an object-oriented environment. The restriction is no general constraint and can be more or less easily raised in a future version of the prototype. Aside from implementation issues, some additional functionality has to be made available:

- For implicit solvers, the resulting domain decomposition of the global grid may require special capabilities within the single grid solvers (e.g., multigrid solvers for block structured grids with adequate smoothers, such as inter block line or plane relaxation methods).

- The block structures in the current AMR++ prototype are defined only by the needs of local refinement of a uniform global grid. This restriction allows them to be cartesian. More complicated structures as they result from difficult non-cartesian external geometries (e.g., holes or split points; see [37]) currently are not taken into consideration. An extension of AMR++, however, is principally possible. The wide experience for general 2D block structured grids that has been gained at the GMD [37] could form a basis for these extensions. Whereas our work is comparably simple in 2D because no explicit communication is required, extending the GMD work to
3D problems is very complex, if not intractable.

3.6.3 Some Implementation Issues In the following, some implementation issues are detailed. They also demonstrate the complexity of a proper and efficient treatment of block structured grids and adaptive refinement. AMR++ takes care of all of these issues, which would have to be handled explicitly if everything had to be handled at the application level.

- **Dimensional Independence and Dimensional Independence Indexing (DII):**
  The implementation of most features of AMR++ and its user interface is dimensionally independent. Being derived from user requirements, on the lowest level, the AMR++ prototype is restricted to 2D and 3D applications. This restriction can, however, be easily removed.

  One important means by which dimensional independence is reached is dimensionally independent indices (DII), which contain index information for each coordinate direction. On top of these DII indices are index variants defined for each type of sub-block region (interior, interior and boundary, boundary only, ...). Convex regions only require a single DII, but non-convex regions require multiple DII. For example, for addressing the boundary of a 3D block (non-convex), one DII index is needed for each of the six planes. In order to avoid special treatment of physical boundaries, all index variants are defined twice, including and excluding the physical boundary, respectively. All index variants, several of them also including extended boundaries (see below), are precomputed at the time when a grid block is allocated. A possible enhancement, for efficiency, would permit them to be shared (cached), since they are independent of the array data. In the AMR++ user interface and in the top level classes, only index variants or indicators are used, thereby allowing a dimensionally independent formulation, except for the very low level implementation. Many low level operations, such as interpolation, for example, are necessarily dependent on the
problem dimension.

- **Implementation of block structured grids:** The AMR++ grid block objects consist of the interior, the boundary, and an extended boundary of a grid block, as well as interface objects (links) that are formed between adjacent pairs of grid block objects. The links contain P++ array objects that do not consist of actual data but serve as views (subarrays) of the overlapping parts of the extended boundary between adjacent grid block objects. The actual boundaries that are shared between different blocks (block interfaces) are complex structures that are represented in a list within each grid block object. Block interface objects are formally derived from the grid block objects (so that interfaces of interfaces are possible (and required for corners where grid blocks meet in two-dimensional problems), and interfaces of interfaces of interfaces are possible (where three-dimension blocks meet at corners)).

For example, in 3D, interfaces between blocks are 2D planes, those between plane-interfaces are 1D-line interfaces, and, one step further, those between line-interfaces are points (zero-dimensional). In figure 15(b), grid blocks 2.1.1 and 2.1.2 of the composite grid in figure 15(a) are depicted, including their block interface and their extended boundary. The regular lines denote the outermost line of grid points of each block. Thus, with an extended boundary of two, there is one line of points between the block boundary line and the dashed line for the extended boundary. In its extended boundary, each grid block has views of the values of the original grid points of its adjoining neighboring block. This way it is possible to evaluate stencils on the interface and, with an extended boundary width of two, to also define a coarse level of the block structured refinement grid in the multigrid sense.

- **Data structures and iterators:** In AMR++, the composite grid is stored as a tree of all refinement grids, with the global grid being the root. Block structured grids are stored as lists of blocks (for ease of implementation; collections of blocks would
be sufficient in most cases). In figure 15(c), the composite grid tree for the example composite grid in figure 15(a) is illustrated. The user interface for doing operations on these data structures is a set of iterators. For example, for an operation on the composite grid (e.g., zeroing each level or interpolating a grid function to a finer level), an iterator is called that traverses the tree in the correct order (preorder, postorder, no order). This iterator takes the function to be executed and two indicators that specify the physical boundary treatment and the type of sub-grid to be treated as an argument. The iteration starts at the root and recursively traverses the tree. For doing an operation (e.g., Jacobi relaxation) on a block structured grid, iterators are available that process the list of blocks and all block interface lists on each grid calling the object member function input as a parameter to the iterator. Iterators are provided for all the relevant AMR++ objects, and so allow simplified internal processing as required for the composite grid solvers. The use of the iterators is not specific to the algorithms currently implemented in AMR++ (FAC, AFAC, and AFACx), and is intended to be a general interface for other algorithms as well.

### 3.6.4 Object-Oriented Design and User Interface

The AMR++ class libraries are customizable by using the object-oriented features of C++. For example, in order to obtain efficiency in a parallel environment, it may be necessary to introduce alternate iterators that traverse the composite grid tree or the blocks of a refinement region in a special order. However, the use of alternate iterators does not change the serial code that uses them, but allows the P++ operations on different composite grid levels to run concurrently. In this way, the data parallel model of P++ is mixed with the tasking parallel model, which can be either supported through C++ tasking libraries\(^5\) or compiler supported extensions\(^6\). The same is true for alternate composite grid cycling strategies, as for example needed in AFAC as opposed to FAC algorithms (section 2.2).

\(^5\)Dirk Grunwald at the University of Colorado at Boulder has developed such parallel tasking class libraries.
\(^6\)Carl Kesselman at Caltech has developed CC++, which provides tasking support as part of a simple C++ language extension.
Application specific parts of AMR++, such as the single grid solvers or criteria for adaptivity, which have to be supplied by the user, are also simply specified through substitution of alternate base classes: A preexisting application (e.g., problem setup and uniform grid solver) uses AMR++ to extend its functionality and to build an adaptive mesh refinement application. Thus, the user supplies a solver class and some additional required functionality (refinement criteria, ...) and uses the functionality of the highest level AMR++ ((Self-)Adaptive-)Composite_Grid class to formulate his special algorithm or to use one of the supplied PDE solvers. In the current prototype of AMR++, FAC and AFAC based solvers (section 2.2) are supplied. If the single grid application is written using P++, then the resulting adaptive mesh refinement application is architecture independent, and so can be run efficiently in a parallel environment.

The design of AMR++ is object-oriented and the implementation of our prototype extensively uses features like encapsulation and inheritance: The abstraction of self-adaptive local refinement, which involves the handling of many issues, including memory management, interface for application specific control, dynamic adaptivity, and efficiency, is reached through grouping these different functionalities into several interconnected classes. For example, memory management is greatly simplified by the object-oriented organization of the AMR++ library: Issues such as lifetime of variables are handled automatically by the scoping rules for C++, so memory management is automatic and predictable.

As the AMR++ interface is object oriented, the control over construction of the composite grid is intuitive and natural: The creation of composite grid objects is similar to the declaration of floating point or integer variables in procedural languages like FORTRAN and C. Users basically formulate their solvers by allocating one of the predefined composite grid solver objects or by formulating their own solvers on the basis of the composite grid objects and the associate iterators, and by supplying the single grid solver class (object).

Although not a part of the current implementation of AMR++, C++ introduces a
template mechanism in the latest standardization of the language (AT&T version 3.0), which only now starts to be part of commercial products. The general purpose of this template language feature is to permit class libraries to use user specified base types. For AMR++, for example, the template feature could be used to allow the specification of the base solver and adaptive criteria for the parallel adaptive local refinement implementation. In this way, the construction of an adaptive local refinement code from the single grid application on the basis of the AMR++ class library can become even simpler and more cleanly implemented.

There is no shortage of obscure details associated with adaptive mesh refinement and its implementation, but this will not be discussed further. The interested reader is referred to [27] and [31].

3.6.5 Static and Dynamic Adaptivity, Grid Generation  In the current AMR++ prototype, static adaptivity is fully implemented. Users can specify their composite grid either interactively or by some input file: For each grid block, AMR++ needs its global coordinates and the parent grid block. Block structured local refinement regions are formed automatically by investigating neighboring relationships. In addition, the functionalities for adding and deleting grid blocks under user control are available within the Adaptive_Composite_Grid object of AMR++.

Recently, dynamic adaptivity has been a subject of intensive research. First results are very promising and some basic functionality has been included in the AMR++ prototype: Given a global grid, a flagging criteria function, and some stopping criteria, the Self_Adaptive_Composite_Grid object contains the functionality for iteratively solving on the actual composite grid and generating a new discretization level on top of the respective finest level. Building a new composite grid level works as follows:

(1) The flagging criteria delivers an unstructured collection of flagged points in each grid block.

(2) For representing grid block boundaries, all neighboring points of flagged points are
also flagged.

(3) The new set of grid blocks to contribute to the refinement level (gridding) is built by applying a smart recursive bisection algorithm similar to the one developed in [6]: If building a rectangle around all flagged points of the given grid block is too inefficient, it is bisected in the longer coordinate direction and new enclosing rectangles are computed. The efficiency of the respective fraction is measured by the ratio of flagged points to all points of the new grid block. In the subsequent tests, 75% is used. This procedure is repeated recursively if any of the new rectangles is also inefficient. Having the goal of building the rectangles as large as possible within the given efficiency constraint, the choice of the bisection point (splitting in halves is too inefficient because it results in very many small rectangles) is done by a combination of signatures and edge detection. The reader is referred to [6] or [27] for more details.

(4) Finally, the new grid blocks are added to the composite grid to form the new refinement level. Grouping these blocks into connected block structured grids is done the same way as it is done in the static case.

This flagging and gridding algorithm has the potential for further optimization: The bisection method can be further improved, and a clustering and merging algorithm could be applied. This is especially true for refinement blocks of different parent blocks that could form one single block with more than one parent. Internal to AMR++, this kind of parent/child relationship is supported. The results in section 4.6, however, show that the gridding already is quite good. The number of blocks that are constructed automatically is only slightly larger (< 10%) than a manual construction could deliver.

A next step in self-adaptive refinement would be to support time dependent problems whose composite grid structure changes dynamically with time (e.g., moving fronts). In this case, in addition to adding and deleting blocks, enlarging and diminishing blocks must be supported. Though some basic functionality and the implementation of the general
concept are already available, this problem has not yet been tackled further.

3.6.6 Current State and Related Work  The AMR++ prototype is implemented using M++ and the AT&T Standard Components II class library to provide standardized classes (e.g., linked list classes). Through the shared interface of M++ and P++, AMR++ inherits all target architectures of P++. AMR++ has been successfully tested on Sun workstations and on the Intel iPSC/860.

Taking into account the large functionality of AMR++, there are still several insufficient aspects and restrictions, and a large potential for optimization in the current prototype (as already pointed out in the preceding description). Until now, AMR++ has been successfully used as a research tool for the algorithms and model problems described in the next two sections. However, AMR++ provides the functionality to implement much more complicated application problems.

Concerning parallelization, running AMR++ under P++ on the Intel iPSC/860 has proven its full functionality. Intensive optimization, however, has only been done within P++. AMR++ itself offers a large potential for optimization. For example, efficiently implementing self-adaptivity, including load (re)balancing in a parallel environment, requires further research. In addition, the iterators that are currently available in AMR++, though working in a parallel environment, are best suited for serial environments. Special parallel iterators that, for example, support functional parallelism on the internal AMR++ level would have to be provided.

To our knowledge, the combined P++/AMR++ approach is unique. There are several other developments in this area (e.g., [37]), but they either address a more restricted class of problems or they are still restricted to serial environments. However, important work at Los Alamos National Laboratory and Lawrence Livermore National Laboratory has addressed adaptive mesh refinement for SIMD and MIMD, respectively, explicit equation solvers.
3.7 Object-Oriented Design for Parallel Adaptive Refinement

Based on experiences with the first C language version of parallel adaptive refinement, the combined complexities of the application, adaptive mesh refinement, and parallelism were seen to limit additional features of the implementation. The requirements of block structured local refinement, self-adaptivity, and more complex applications were considered out of range. The relatively simplistic adaptive refinement code was roughly 16,000 lines and only solved the simple potential flow (principally, the Poisson equation) problem.
Figure 15. Example of a composite grid, its composite grid tree, and a cutout of two blocks with their extended boundaries and their interface.
CHAPTER 4

PARALLEL ADAPTIVE MESH REFINEMENT: RESULTS

4.1 Introduction

This chapter reports on comparative results of several facets of this thesis work. Specifically, we compare the two composite grid algorithms FAC and AFAC. These comparisons are a little incomplete since they exclude the newer AFACx, which would correspond to AFAC formulated using only relaxation on the refinement level. A comparison of FAC and AFAC is nevertheless representative of the general case, since both FAC and AFAC can be formulated using relaxation of the composite grid refinement levels; we refer below to such schemes as FACx and AFACx, respectively.

We introduce the details of parallel performance of FAC and AFAC on several parallel computers. Some of these machines are no longer available, and the codes that were used to obtain these results can, in most cases, no longer be run. This awkward level of software reliance on specific hardware, even from a common computer vendor, was one dominant motivation for the development of P++ and AMR++. Such issues as architecture independence, even among the restricted class of parallel computers of the same basic architectures, were not common when these codes were developed, but they will become increasingly important in larger software development projects and magnified on projects involving machines of increasingly different architecture. Such code might be required to have a lifetime spanning that of several parallel computers (which seem to have lifetimes limited to about 5 years).

Several different parallel machines are used for the accumulated results. The dominant factor in predetermination of the results is the computation/communication ratio,
since it factors out the relative computational advantage of some of the newer machines that have increased computational speed (MFLOPs) but maintain the software layer within the communication network that connects processors.

We expect good vectorization capabilities of algorithms to be important now and in the future since, in many species of the current and coming generation of distributed memory multiprocessor architectures, high performance can only be reached if the programming model is based on vector operations. We have in mind that the i860 is no vector processor, but one only achieves high performance if its compiler is based on a vector model, which most of the currently available ones are not. With this in mind, we note that some parallel algorithms perform differently across many machines because of the different computation/communication cost ratios. Thus, to fully test the AFAC algorithm, it is important to test FAC and AFAC in a controlled way, under a variety of computation/communication cost ratios. Therefore, we test and compare AFAC and FAC on the SUPRENUM in both scalar and vector modes and on the earlier generations of Intel (iPSC/1, iPSC/2, and iPSC/860). It is interesting, and important, that in vector mode we observe distinctly better efficiency for AFAC than for FAC. It is likely that many algorithms are sensitive to the computation/communication cost ratio. The results show that, for large problems, FAC is sensitive to this ratio and that, for these same problems, AFAC is not. This is a distinct advantage of AFAC, which should make it appropriate across a wide class of parallel architectures, especially on parallel machines with fast vector floating point performance, or correspondingly slow communication (message startup and transfer rate). The latter architectures with relatively slow communication rates are increasingly common in more recently introduced parallel computers because message passing has consistently involved a software layer in its implementation. This software layer is an especially inherent property of portable communication libraries, which are required for advanced parallel software (in order to amortize the software development across multiple architectures).
To simplify the code that implements the adaptive AFAC and FAC algorithms in both scalar and vector modes, we solve a relatively simple Poisson problem. This permits the addition of complexity in the evaluation of the code and exploration of alternate parallel partitioning strategies. This thesis details these alternate parallelization strategies and compares AFAC and FAC under a variety of situations. We choose two composite grids of different structural complexity to show the effects of the composite problem domain on these two algorithms when run on different numbers of processors and grid sizes. For simplicity and focus, we restrict each level of refinement to the same size, although this is not a restriction in the code. Under these varying parameters (composite grid complexity, number of processors, and size of the uniform composite grid levels), we compare the effects of different computation/communication cost ratios of FAC and AFAC on different distributed memory multiprocessor architectures.

Section 2.2 describes the FAC and AFAC algorithms, section 3.2 details the FAC and AFAC codes, section 4.4 reports on the comparison of FAC and AFAC on the target architectures, and section 4.4.7 summarizes the conclusions of these comparisons.

Additional results indicate the effects of dynamic adaptivity of the composite grid in the parallel environment. These results are compared to the relative computational time of a single AFAC iteration. They are specific to the Intel iPSC/1, but also present performance relative to an AFAC iteration. Thus, the results are representative of a machine with the iPSC/1's communication/computation ratio, which is mostly unchanged even in more recent computer designs that use both faster processors and communication hardware. The performance of manipulating the composite grid is investigated. This manipulation includes the addition and deletion of local refinement regions, the required load balancing after addition or deletion of local refinement, and the repositioning (moving) of local refinement within the composite grid.

\footnote{Communication is still buried in a software layer and this greatly affect its performance, except notably on the Transputer.}
4.2 Comparison of Convergence Factors

The principal result shows the difference between the convergence factor bounds of FAC and AFAC. Convergence theory in [40] shows that the convergence rates of FAC and AFAC have the relation \( ||\rho_{AFAC}|| = ||\rho_{FAC}||^2 \). Although the theory is restricted to a 2-level composite grid, these results have been experimentally verified to hold even on very large numbers of levels (a specific test verified this property of the convergence rate on a 50-level composite grid). Though the algorithmic components in our code are chosen slightly differently than for the convergence analysis, experience shows that very similar behavior is obtained. This implies that two cycles of AFAC are roughly equivalent to one cycle of FAC.

A second interesting result is that AFACx obtains nearly the same convergence rates as AFAC (within a few percent). This result is experimentally verified on the same 50-level problem that was used to demonstrate the independence of the convergence rate on the number of levels.

4.2.1 Composite Grid Convergence Factors for Poisson Equation

We present results of convergence tests on several example composite grid problems. The first is a simple composite grid problem using a small stack of rectangular local refinement regions positioned in a corner. The convergence factors are independent of the position of the local refinement regions and are experimentally shown to be asymptotically independent of the number of composite grid levels. The convergence factors for the Poisson equation on the simple composite grid are shown in table 1 we include results for a (2,1) V-cycle on the global grid for comparison). These factors were determined by performing 20-30 cycles, beginning with a random initial guess and using a zero Rhs, and observing the worst-case ratio of successive residual error norms.

Using block structured local refinement, the same equation is solved on a composite grid build using static adaptivity. Additional refinement is added to specific positions in the global domain, and new regions are added to each iteration after the composite grid is solved
Table 1: Convergence factors for Poisson’s equation on simple composite grid.

<table>
<thead>
<tr>
<th>Convergence Factor $\rho$</th>
<th>$h$</th>
<th>5-point stencil</th>
<th>9-point stencil</th>
</tr>
</thead>
<tbody>
<tr>
<td>Poisson</td>
<td></td>
<td>1/64</td>
<td>1/512</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1/64</td>
<td>1/512</td>
</tr>
<tr>
<td>MG-V</td>
<td>0.08</td>
<td>0.08</td>
<td>0.02</td>
</tr>
<tr>
<td>FAC</td>
<td>0.11</td>
<td>0.11</td>
<td>0.10</td>
</tr>
<tr>
<td>FACx</td>
<td>0.13</td>
<td>0.13</td>
<td>0.11</td>
</tr>
<tr>
<td>AFAC</td>
<td>0.35</td>
<td>0.35</td>
<td>0.33</td>
</tr>
<tr>
<td>AFACx</td>
<td>0.35</td>
<td>0.35</td>
<td>0.33</td>
</tr>
</tbody>
</table>

Table 2: Convergence factors for Poisson’s equation on block structured composite grid.

<table>
<thead>
<tr>
<th>Convergence Factor $\rho$</th>
<th>$h$</th>
<th>5-point stencil</th>
<th>9-point stencil</th>
</tr>
</thead>
<tbody>
<tr>
<td>Poisson</td>
<td></td>
<td>1/64</td>
<td>1/512</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1/64</td>
<td>1/512</td>
</tr>
<tr>
<td>MG-V</td>
<td>0.08</td>
<td>0.08</td>
<td>0.02</td>
</tr>
<tr>
<td>FACx</td>
<td>0.17</td>
<td>0.18</td>
<td>0.11</td>
</tr>
<tr>
<td>AFACx</td>
<td>0.40</td>
<td>0.41</td>
<td>0.33</td>
</tr>
</tbody>
</table>

2. Table 2 shows the convergence rates of AFACx and FACx (compared to MG-V on the global grid) using the more complex block structured local grids (figure 22). Note that the use of the block structured grid inhibits the use of FAC and AFAC since the block structured coarsenings required for their definition are not easily constructed.

4.2.2 Convergence Factors for Singular Perturbation Problem

Numerical results (details are found in [33]) have been obtained for the model problem

$$-\varepsilon \Delta u + au_x + bu_y = f \quad \text{on} \quad \Omega = (0,1)^2$$

with homogeneous Dirichlet boundary conditions on $\partial\Omega$ (except on the boundary where we force a boundary layer) and $\varepsilon = 0.00001$. This problem serves as a good model problem for complex fluid flow applications, because several of the properties that are related to self-adaptive mesh refinement are already present in this simple problem. The equation is discretized using isotropic artificial viscosity (diffusion):

$$L_h := -\varepsilon_h \Delta_h + a D_{h,x} u + b D_{h,y} u \quad \text{with} \quad \Delta_h = D_{h,x}^2 + D_{h,y}^2,$$

$$\varepsilon_h := \max\{\varepsilon, \beta h \max\{|a|,|b|\}/2\}.$$
For comparison, some results with a nine point stencil discretization corresponding to bilinear finite elements for the Laplace operator have also been obtained.

The discrete system is solved by multilevel methods: MG-V on the finest global grid and FAC or AFAC on composite grids with refinement. For standard multigrid methods, it is known that, with artificial viscosity, the two-grid convergence factor (spectral radius of the corresponding iteration matrix) is bounded below by 0.5 (for \( h \to 0 \)). This leads to multilevel convergence factors that tend to 1.0 with increasing number of levels (e.g., [20]). For many more details, see [27]. In [16], a multigrid variant that shows surprisingly good convergence behavior has been developed: convergence factors stay far below 0.5 (with three relaxations on each level). Here, essentially this method is used, which has the following components:

- Discretization with isotropic artificial viscosity using \( \beta = 3 \) on the finest grid \( m \) and \( \beta_{l-1} = 1/2 (\beta_l + 1/\beta_l) \) for coarser grids \( l = m - 1, m - 2, \ldots \)
- Standard MG components (odd even relaxation, full weighting, and bilinear interpolation).

Anisotropic artificial viscosity may also be used, but generally requires (parallel) zebra line relaxation, which has not yet been fully implemented. For FAC and AFAC, the above MG method with V(2,1) cycling is used as a global grid solver. On the refinement levels, three relaxations are performed.

In table 3, several convergence rates for FACx, AFACx, and MG-V are shown for the example equation. The finest grids have mesh sizes of \( h = 1/64 \) and \( h = 1/512 \). For FAC and AFAC, the global grid has the mesh size \( h = 1/32 \), and the (predetermined) fine block always covers 1/4 of the parent coarse block along the boundary layer.

The following observations were made:

- In the case of FAC and AFAC, both for the 5 point and the 9 point Laplacian, the choice of \( \beta \) must be further investigated.
<table>
<thead>
<tr>
<th>Convergence Rate $\rho$</th>
<th>$h$</th>
<th>5-point stencil</th>
<th>9-point stencil</th>
</tr>
</thead>
<tbody>
<tr>
<td>Poisson</td>
<td>$a = b = 0$</td>
<td>MG-V</td>
<td>0.08</td>
</tr>
<tr>
<td>$\beta = 1, \varepsilon = 1$</td>
<td>FACx</td>
<td>0.33</td>
<td>0.10</td>
</tr>
<tr>
<td>SPP</td>
<td>$a = b = 1$</td>
<td>MG-V</td>
<td>0.14</td>
</tr>
<tr>
<td>$\beta = 3, \varepsilon = 0.00001$</td>
<td>FACx</td>
<td>0.65</td>
<td>0.60</td>
</tr>
<tr>
<td>SPP</td>
<td>$a = b = 1$</td>
<td>AFACx</td>
<td>0.67</td>
</tr>
<tr>
<td>$\beta = 1.1, \varepsilon = 0.00001$</td>
<td>AFACx</td>
<td>0.85</td>
<td>0.55</td>
</tr>
</tbody>
</table>

Table 3. Convergence factors for a singular perturbation problem and, for comparison, for Poisson's equation.

- For MG-V and the 5 point stencil, the results are as expected, while the 9 point stencil gives better but also deteriorating results.
- V cycles are used; W or F cycles would yield better convergence rates but worse parallel efficiency.
- The 9 point stencil for the Laplacian fulfills the Galerkin condition with respect to the level transfers used and shows better convergence rates than the 5 point stencil.
- Results for the composite grid problem are not as good as expected, but only the MG-V scheme problem is specially treated for the singular perturbation equation.

4.3 Performance of AFAC

To study various properties of the AFAC code, two composite grid examples were used. The two examples were chosen to show different aspects of the implementation on both the 32-node iPSC/1 and the 16-node iPSC/2. One example is a simple seven-level composite grid with all constituent uniform grids of the same size (as in figure 16). The second example is a much more complicated composite grid with 40 constituent uniform grids, all of the same size (figure 19). Although this composite grid is nonphysical, its purpose is to show the flexibility of the algorithm and prototype code and to simulate in 2D the potential complexity and number of grids that could be present in 3D applications. This example is not actually a limiting case since the composite grid for a problem posed in
3D could have even more grids and more levels of refinement. Also, the simulation of the 3D problem in 2D is not accurate since the computational loads of the grids posed in 3D are much larger than those posed in 2D. However, the larger loads would be expected to parallelize much better than the smaller loads since overhead could be more easily hidden.

Although two examples were chosen, because of the limited availability of the iPSC/2 and the test machine's limited memory, only the much smaller composite grid example was run on the iPSC/2. The tests on the iPSC/2, however, show the much better computation-to-communication balance over that of the iPSC/1. Both examples were run on the 32-node iPSC/1.

Each level was an nxn grid, counting boundary grid lines, with n varying from 17 to 129. Results of timing these runs on the iPSC/2 for the simple example are depicted in table 4. Similarly, results for the complex example run on the iPSC/1 are shown in table 5. Additionally, we provide results, on the iPSC/1, for the grid manager functions necessary for adaptive work in parallel environments. Specifically, we detail the results of MLB (table 21), moving grids (table 8), and adding further refinement (table 9). Finally, we show the relative costs of these functions to that of solving the complex composite grid (table 10).

4.3.1 Simple Example on iPSC/2: In table 4, the displayed times for the iPSC/2 are the worst-case times for one AFAC cycle (including the individual times for the MG-V solve, step 1, and the interlevel transfers, step 2) and for a D-dimensional cube with D ranging from zero to four. While the data for n = 65 and D = 4 is missing because of a hardware anomaly, for the case n = 129 only D = 4 data is depicted because the iPSC/2's memory could not support the cases D < 4.

Note the reasonably good speed-up exhibited by AFAC, especially for the larger problems. In particular, from a single node to an eight-node cube, speed-ups for n = 17,33, and 65 were about five, six, and seven, respectively. A related observation is that cases with the same ratio of grid points to processors have nearly the same total times. Note also that
Figure 16: Simple Composite Grid
the sum of the solve and transfer times is often greater than the total time. This is because the times documented here are each worst-case over all processors.

4.3.2 Simple Example on iPSC/1: In table 5, the displayed times for the iPSC/1 are the worst-case times for one AFAC cycle and for a $D$ dimensional cube with $D$ ranging from zero to five. No data is available for the case $n = 129$ since messages passed during intergrid transfers can exceed 16K in length, which is the maximum message size allowed on the iPSC/1.

Here we see the effect of the more expensive communication costs on the iPSC/1. The result is that speed-up is not as good as it is on the more balanced iPSC/2, though it is still quite good. While the transfer times show less improvement when parallelized than the solve times, the solvers are much more expensive computationally.

4.3.3 Complex Example on iPSC/1: In table 5, the displayed times for the iPSC/1 are the worst-case times for one AFAC cycle and for a $D$-dimensional cube with $D$ ranging from zero to five. There was not enough memory or allowable message passing length for the case $n = 129$.

This more computationally intensive example shows the greater potential of AFAC in a parallel environment with complex grids. In this problem, speed-up for the solves is almost perfect. In particular, from a single node to an eight-node cube, speed-ups for $n = 17,33$, and 65 were about 7.7, 7.9, and 7.8, respectively. Since the $n = 65$ case was too large for a single node, the time was estimated by noticing that this grid contains about four times the computational load of the $n = 33$ grid. This relationship can be noticed in other examples when run on a single node.

On larger cubes ($D = 4,5$), substantial degradation in speed-up is noticed. This results only partially from the additional communication during the solves compared to the lessened computation available on each processor. Particularly noticeable is the effect of problem size: speed-up is degraded on the larger grids, contrary to earlier results. Notice that
Table 4: Timings for simple AFAC on iPSC/2.
on a D5 the speed-up for the $n = 17$ grid is 15, while the speed-up on the $n = 65$ grid is seven. This is due to the larger granularity of the parallelism present on each node. Since the shared grids are serviced only after the completion of access of the entire grid, the larger grids result in longer delays in processing the more expensive shared grids (more expensive because of the required communication of boundary values). The remedy is straightforward. As discussed in the section on data flow, a reduction of the grain size, presently tied to the grid size, would force the servicing of the shared grids more frequently. Thus, further partitioning of the large grids within a processor would make the parallelism finer and eliminate the waiting done by shared grids. With a finer granularity of parallelism in each processor, speed-up should improve significantly through the larger cubes, particularly on larger problems.

4.4 Performance Comparison of FAC and AFAC Algorithms

The asynchronous processing of the composite grid in the parallel environment is only advantageous if doing so offsets the poorer numerical convergence factor of AFAC compared to FAC. In this section, we study this issue by examining and comparing the performance of AFAC and FAC in the parallel environment.

4.4.1 Parallelization for Distributed Architectures

A basic AFAC code for distributed memory multiprocessor architectures was developed to solve Poisson's equation on the unit square, using the finite volume element technique ([46]) for composite grid discretization and MG-V for solution of each grid level problem. The code was later modified in a straightforward manner to permit FAC as well as AFAC implementations the only change was the introduction of sequential processing of the composite grid levels through a reordering of the basic operations that originally formed the AFAC algorithm. Both algorithms are so closely related that this in fact resulted in an efficient implementation of FAC.

4.4.1.1 Parallel Multigrid Solver — Standard Grid Partitioning

In the AFAC/FAC code, the Dirichlet problems on all patches and the global grid are solved by a
Table 5: Timings for Complex AFAC on iPSC/1.
parallel full multigrid (FMG) solver that is truncated to the two finest levels. For simplicity, we call this truncated FMG solver a 2-step FMG cycle. Although the grid sequence for the standard multigrid solver must be processed sequentially, processing on a given grid can be done simultaneously across its points. This naturally leads us to consider standard grid partitioning [34], where each processor works on the fine grid and the associated parts of the coarse grid related to a specific subdomain.

Inter-processor communication, as necessary during residual computation or relaxation on a given grid, is done by exchange of data in so-called process overlap areas. Inter-grid data movement, as necessary within the grid transfer operations such as interpolation, can be done totally inside the processors, and therefore does not require any inter-processor communication.

4.4.1.2 Parallelization of FAC — Linear Single Level Grid Partitioning

Due to the necessity of sequential treatment of the sequence of levels, the parallelization strategy that we chose for FAC involves regular linear strip partitioning of each level independently over all processors. All processors share an equal sized subpatch (strip) of each patch [23], which automatically leads to very good load balancing. If $n$ is the number of processors and $p$ the number of patches, then this partitioning leads to $n \cdot p$ subpatches and $(n - 1) \cdot p$ process overlap areas (where internal boundaries are shared between processors). Inter-process communication is done in the solver part as described in the previous section. Inter-level communication has to be done nearly exclusively over process boundaries.

Concerning adaptivity, it is possible to add, move, or delete one or more patches at any time without redistribution or rebalancing of the workload with respect to other patches.

4.4.1.3 Parallelization of AFAC — Linear Multilevel Grid Partitioning

Because all levels can be processed simultaneously, global irregular multilevel strip partitioning of all patches across all processors based on a workload function is used. Each processor
shares an equal part of the rows contained in the sum of all patches. A contiguous number of rows belonging to the same patch and the same processors denotes a subpatch. The maximum number of subpatches is \( n + p - 1 \); the maximum number of overlap areas is \( n - 1 \).

The implementation of AFAC allows asynchronous scheduling of multiple levels. Priority is given to shared grids that naturally require more communication. Additionally, boundary processing can be done before processing interior data so that exchange of boundary data can be overlapped with computation as much as the architecture allows (message latency hiding is not supported on SUPRENUM, but is on the Intel machines). As in FAC, inter-processor communication is done within the solver, but inter-level communication is done without respect to process boundaries.

Initial load balancing can be done in a nearly optimal way based on a work function for the initial composite grid. Adaptively adding or deleting a patch, however, requires load rebalancing. This is done by the multilevel load balancer (MLB). Experience shows that this load balancer is highly efficient and requires only a small fraction of the time needed for the arithmetic phases of the algorithm even in the case of Poisson's equation [41]. For a detailed discussion of adaptive aspects, see [42].

4.4.2 Comparison of Interprocessor Communication in FAC and AFAC

In addition to the rough factor of two in convergence factor in favor of FAC, the influential parallel performance factors are the vectorization capabilities and the communication structure. Both algorithms are very vectorizable because standard vectorization and parallelization do not interact: vectorization is done along the rows of the levels (programming language C), while parallelization is done along the columns (thus partitioning along rows).

Whereas the amount of arithmetic and degree of vectorization for FAC and AFAC are very similar, the number and size of the subpatches, i.e., the distribution of the patches on a parallel computer, differ significantly: For FAC, \( n \cdot p \) subpatches are needed, while for AFAC, \( n + p - 1 \) subpatches are needed. As a result, the communication structure, the
number of messages, and the message lengths of the algorithms differ significantly.

Tables 6 and 7 give an overview of the communication structure of both algorithms for three examples each. \( p \) denotes the number of patches, \( n \) the number of processors, \( nz \) the number of grid points in each direction on each patch, and \( ps \) the maximum number of subpatches. Distinguishing between the solution and level transfer phases, the number of synchronization steps (communication steps, \# steps), the maximum number of communications (\# comms.), and the maximum amount of communicated data (in Kwords, \# Kwords) are listed.

<table>
<thead>
<tr>
<th>Simple composite grid</th>
<th>( p = 7 )</th>
<th>FAC</th>
<th>AFAC</th>
<th>( p = 7 )</th>
<th>FAC</th>
<th>AFAC</th>
<th>( p = 7 )</th>
<th>FAC</th>
<th>AFAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n = 4 ) ( nz = 127 )</td>
<td>28</td>
<td>10</td>
<td>112</td>
<td>22</td>
<td>112</td>
<td>22</td>
<td>112</td>
<td>22</td>
<td>112</td>
</tr>
<tr>
<td>( \max_{i=1,n} p_s )</td>
<td>4</td>
<td>2</td>
<td>16</td>
<td>3</td>
<td>16</td>
<td>3</td>
<td>16</td>
<td>3</td>
<td>16</td>
</tr>
</tbody>
</table>

**Solution**

<table>
<thead>
<tr>
<th># steps</th>
<th>FAC</th>
<th>AFAC</th>
<th># steps</th>
<th>FAC</th>
<th>AFAC</th>
<th># steps</th>
<th>FAC</th>
<th>AFAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 728 )</td>
<td>104</td>
<td>4,368</td>
<td>( 728 )</td>
<td>104</td>
<td>624</td>
<td>( 840 )</td>
<td>120</td>
<td>4,368</td>
</tr>
<tr>
<td># comms.</td>
<td>48</td>
<td>61</td>
<td># comms.</td>
<td>80</td>
<td>61</td>
<td># comms.</td>
<td>200</td>
<td>61</td>
</tr>
<tr>
<td># Kwords</td>
<td>192</td>
<td>306</td>
<td># Kwords</td>
<td>3,120</td>
<td>44</td>
<td># Kwords</td>
<td>3,600</td>
<td>44</td>
</tr>
</tbody>
</table>

**Level transfer**

<table>
<thead>
<tr>
<th># steps</th>
<th>FAC</th>
<th>AFAC</th>
<th># steps</th>
<th>FAC</th>
<th>AFAC</th>
<th># steps</th>
<th>FAC</th>
<th>AFAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 12 )</td>
<td>12</td>
<td>12</td>
<td>( 12 )</td>
<td>12</td>
<td>12</td>
<td>( 12 )</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td># comms.</td>
<td>48</td>
<td>48</td>
<td># comms.</td>
<td>192</td>
<td>192</td>
<td># comms.</td>
<td>192</td>
<td>192</td>
</tr>
<tr>
<td># Kwords</td>
<td>48</td>
<td>48</td>
<td># Kwords</td>
<td>48</td>
<td>48</td>
<td># Kwords</td>
<td>48</td>
<td>48</td>
</tr>
</tbody>
</table>

Table 6. Communication structure analysis of FAC and AFAC for the simple 7 patch model problem.

In the solver phase, the number of communication steps, the number of communications, and the amount of communicated data are much larger for FAC than for AFAC. This is because the numbers linearly depend on the number of subpatches, which are more numerous due to each patch being partitioned over all processors in FAC. The number and amount of communications per step, per processor, and per processor and step, and the average message length can be easily computed from the given numbers.

In the grid transfer phase (between levels), the amount of communicated data in
<table>
<thead>
<tr>
<th>Complex composite grid</th>
<th>( p = 30 )</th>
<th>( n = 4 )</th>
<th>( n_x = 63 )</th>
<th>( p = 30 )</th>
<th>( n = 16 )</th>
<th>( n_x = 63 )</th>
<th>( p = 30 )</th>
<th>( n = 16 )</th>
<th>( n_x = 127 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FAC</td>
<td>AFAC</td>
<td>FAC</td>
<td>AFAC</td>
<td>FAC</td>
<td>AFAC</td>
<td>FAC</td>
<td>AFAC</td>
<td></td>
</tr>
<tr>
<td>( p_x )</td>
<td>120</td>
<td>33</td>
<td>480</td>
<td>45</td>
<td>480</td>
<td>45</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \max_{i=1,n} p_x )</td>
<td>4</td>
<td>2</td>
<td>16</td>
<td>2</td>
<td>16</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Solution</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># steps</td>
<td>2,640</td>
<td>88</td>
<td>2,640</td>
<td>88</td>
<td>3,120</td>
<td>104</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># comms.</td>
<td>15,840</td>
<td>528</td>
<td>79,200</td>
<td>2,640</td>
<td>93,600</td>
<td>3,120</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># Kwords</td>
<td>127</td>
<td>4</td>
<td>634</td>
<td>21</td>
<td>1,310</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Level transfer</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># steps</td>
<td>58</td>
<td>58</td>
<td>58</td>
<td>58</td>
<td>58</td>
<td>58</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># comms.</td>
<td>232</td>
<td>16</td>
<td>928</td>
<td>64</td>
<td>928</td>
<td>64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># Kwords</td>
<td>58</td>
<td>58</td>
<td>58</td>
<td>58</td>
<td>234</td>
<td>234</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7. Communication structure analysis of FAC and AFAC for the complex 30 patch model problem.

both algorithms is roughly equal, but the number of messages to be exchanged is higher in FAC. In FAC, at each step, all processors are involved and exchange messages in parallel (since every patch is partitioned across all processors). In AFAC, only those processors that contain subpatches linked by a father/son relationship, in the composite grid tree, are involved. Therefore, the number of communications has a different impact on performance, because the degree of parallelism, in the transfer between levels, is higher in FAC.

Under the assumptions that communication between distinct pairs of processors does not have mutual influence and that communication and computation cannot be overlapped, the communication structure of this transfer between patch levels in FAC is superior to that of AFAC. But since the solver phase dominates the time spent in an FAC/AFac iteration (solve/transfer), the superiority of FAC in the transfer phase is not significant in most cases. Specifically, the number of communications per processor per step is much smaller in FAC. This is the significant number if one considers the parallelism in the execution of communication between distinct pairs in the transfer phase. As a result, the amount of communicated data per message executed in parallel is smaller in FAC. Also, all messages
have about the same length. In AFAC, however, the message passing associated with the
transfer phase is less parallel, although alternative mappings of the composite grid to the
processors could alleviate this at an increased cost to the solver phase. As a result, the
messages are longer and often of different lengths. Thus, in AFAC, the time required for
one communication step is as large as the time required to exchange the longest message.
The impact of this disadvantage of AFAC is significantly lessened if the transfer rate of each
pair of processors is dependent on the number of communicating pairs. This is the case on
hierarchical bus-coupled architectures such as SUPRENUM, where the total transfer rate
is restricted by the bus bandwidth. On hypercubes, this is only the case for non-nearest
neighbor communication of processor pairs that use the same links. Also, the transfer rate
depends on whether both processors, of each pair, are within one cluster / nearest neigh-
bor (faster) or in distinct clusters / non-nearest neighbor (slower). Also, the possibility of
overlapping communication and computation on the Intel machines (currently not possible
on SUPRENUM) makes the influence of larger messages in AFAC significantly smaller. The
latter is due to the greater parallelism of the AFAC algorithm. Since, for the message length
associated with FAC and AFAC, message start-up time in most cases is the dominating fac-
tor in overall communication time, the smaller number of messages, even longer messages,
associated with AFAC over FAC is a perceived advantage.

4.4.3 Relative Performance Results In this section, results of numerical
experiments that compare FAC and AFAC on SUPRENUM and the Intel IPSC/2 and
iPSC/860 are presented.

- The code has been run on SUPRENUM in scalar and vector mode in order to provide
  for very different computation/communication cost ratios; although the communi-
cation time stays nearly constant (same start-up costs, same transfer rate, slightly
  faster buffering in vector mode), the computation time differs significantly. For a
  more detailed discussion of the distributed architectures, see [34]. Factors of up to
20 in favor of vector computations for pure arithmetic processing are typical. Thus, the impact of communication on overall performance differs radically in both modes. Though it is obvious that on SUPRENUM the vector unit should be used, the scalar results give some idea of how the percentage of time spent in communication changes if basic architectural parameters are changed. All results were obtained on one cluster of a 4-cluster (64 nodes) SUPRENUM machine at GMD in Bonn. For results on the 64 node SUPRENUM machine, the reader is referred to [30].

- The code has been run in scalar mode on a 4D subcube of the 32 node Intel IPSC/2-VX of the GMD. The vector floating point processor has not been used due to vector memory restrictions (1 MB/node, cf. section 3.2). Additional results on the 32 node IPSC/2 can be found in [41]. Results specific to the use of adaptive AFAC on the iPSC are in [42].

- Finally, the code has been run on a 4D subcube of the 32 node Intel iPSC/860 of GSF based on the Portland Group C Compiler. Special vector libraries have not yet been available, so that the possibility of overlapping functional units, of the i860, has not been optimally used. The performance is therefore restricted by the "scalar" speed (cf., section 3.2). For future compiler releases, significant speed-up in pure computation time has to be expected, whereas communication time will stay rather constant. This leads to a further decrease of the computation/communication cost ratio.

Timings are in milliseconds for 1 cycle. D0, D2, and D4 denote systems consisting of \(2^0\), \(2^2\) and \(2^4\) processors, respectively.

4.4.4 Conclusions of FAC versus AFAC To compare AFAC and FAC, we restrict ourselves to two test problems. The first is a simple 7-level composite grid with all refinement at a corner, similar to figure 17. The second is a complex composite grid with

---

\(3^{rd}\) The iPSC/860 results are not complete due to system problems and restricted access time to the system.
a total of 30 local refinement patches, as in figure 19. In order to give a fair comparison of both methods, in the following bar charts (see figure 20), two cycles of AFAC (black bars) are always compared to one cycle of FAC (white bars), which takes the convergence factors of both methods into account.

Referring to figure 20, note that while FAC out performs AFAC on a single processor, in most cases AFAC out performs FAC even on a few processors (16). This is amplified by the inclusion of vectorization. The case where FAC is better than AFAC is with small numbers of large patches in scalar mode on a few processors. In this case, the effect of communication is minimized and computation dominates the AFAC/FAC algorithm, that is, the reasons that FAC is superior on a single processor extends to this restrictive class of problems on small numbers of processors.

Specific to the 7-level composite grid (formed as in figure 17, but with 7 levels instead of the 3 pictured), we see that FAC is better in the case of the scalar results, at least for up to 16 processors. But for the vectorized code, AFAC is superior. This is due to the increased relative communication costs brought about by the much faster vectorized computation. Thus, since AFAC (as shown in tables 2 and 3) has reduced communication in the parallel environment, it has better performance.

In the case of the complex composite grid (with 30 patches, similar to figure 19), AFAC performs better than FAC in both scalar and vector nodes. This is due to the poor communication properties (shown in tables 2 and 3) of FAC on composite grids with greater complexity. As a result of these poor communication properties, AFAC out performs FAC even on the relatively small (16) processor parallel computers.

The effect of the number of processors in a parallel system is not so clear on the small number of processors that we have presented, so in figure 20 we include some results on 32 and 64 processors. What these results show (on both the 7-level composite grid and the 30 patch composite grid in vector mode) is that the increase in the numbers of processors
has a dramatic effect on the comparison of AFAC and FAC, with AFAC performing very efficiently, especially compared to FAC.

Thus, we see that the superiority of AFAC over FAC is dependent on both the number of processors and the complexity of the grid. The effect of vectorization just amplifies the poor communication properties of FAC on even small numbers of processors. If we make the assumption that these results will extrapolate to larger multiprocessor computers, then we expect the marked superiority of AFAC over FAC and, likely, all other local refinement algorithms because of their synchronous treatment of the composite grid levels.

For the 30 patch model problem, one should be aware that the current FAC implementation is not optimal, because the problem actually only contains 6 levels. Patches of different refinement areas are treated in parallel, but could be partitioned more efficiently. The alternative partitioning would force all patches at the same level to be put into a list, which would then be partitioned across all processors, similar to the way that AFAC treats all the patches. As a result, the subpatches in each processor would be larger and there would be fewer total messages associated with FAC. However, this partitioning would have the disadvantage of added complexity, especially for use in an evolving, dynamic, composite grid.

4.5 Dynamic Adaptivity in Parallel Environment

An important and potentially disabling requirement of parallel adaptive mesh refinement for time dependent problems is the dynamic changes required of the evolving composite grid distributed across the multiprocessor environment. The dynamic changes take the form of addition and deletion of refinement regions, re-load-balancing after modifications of the composite grid, and repositioning of existing refinement as a part of tracking moving regions where additional refinement is required. An important issue in the parallel environment is how the composite grid can be partitioned. Depending on this partitioning, the work associated with adding/deleting new refinement or moving existing refinement can
be prohibitive. The partitioning available with AFAC permits any ordering of the composite grid, so moving grids move only logically in relative position but need not be reshuffled in the multiprocessor system. Alternatively, a partitioning of the physical domain forces significant reshuffling when a grid is moved within the physical domain. Also, adding/deleting of grids is more expensive since the partitioned domain limits the ability to cache the added and deleted grids (deleted grids need not be deallocated and could be added trivially).

4.5.1 Multilevel Load Balancing Figure 21 shows the times required to load balance the composite grid after addition of local refinement grids of increasing size. Notice that these times depend only slightly on cube size since most work is spent in the transfer of grid data between nodes. The cost of computing the target processors of the unbalanced data is very inexpensive. In this test of MLB, a grid is added to the existing complex composite grid and the resulting unbalanced composite grid is then rebalanced. Times are unavailable for the case $n = 65$ since message sizes are greater than 16K for the grid data transfer between nodes, which is the maximum message size on the iPSC/1. To compensate for this omission, the case $n = 9$ is presented. Note that these times are small compared to the costs of solving the composite grid problem.

4.5.2 Dynamic Movement of Local Refinement (Grid Tracking) To test the potential performance of AFAC for handling moving grids, we computed the times required to make the movement of the largest stack of grids shown in figure 19. In the complex grid example solved and timed previously, the 30 grids stacked on the coarsest grid are moved, and the time required for this move and the update of the movement in all processors is recorded. Specifically, both the relative and global positions of all 30 grids are communicated and updated in all processors. The time required is identical for all processors and is shown in table 8. Note that these times are small compared to the costs of solving the composite grid problem.
4.5.3 Dynamic Addition of Local Refinement   To estimate the cost associated with building further refinement on the existing composite grid, we computed the times for one node to both build a grid and communicate the composite grid modification to the composite grid tree present in all other processors; see table 9. The times are different between the node that owns the new grid and those that merely update their record of the composite grid. This is because the owning node must initialize the new grid, update the right-hand side, and perform other functions, so the times depend in the size of the new grid. Nodes not owning the new grids must update their copy of the composite grid, which is inexpensive, so their times reflect the cost of a global communication and are therefore dependent on the cube size. If necessary, the cost of building the new grid could be parallelized by letting several processors each build a piece of the new grid. Thus, the new grid would be shared rather than wholly owned after construction. In any case, the costs of adding a grid are still small compared to the costs of solving the composite grid problem.

4.5.4 Relative Costs of Dynamic Adaptivity   In table 10, we compare (in logarithmic scale) the times required of AFAC on the complicated example to move a stack of 30 grids, add further refinement, and load balance the additional refinement. We compare these times to show that these grid manager functions have little impact on the overall cost of solving the composite grid problems, even in a parallel environment.

4.5.5 Conclusions about Dynamic Adaptivity   This section has detailed the use of dynamic adaptivity combined with the composite grid solver. The results make it clear that the use of time-dependent problems with parallel adaptive mesh refinement does not degrade the performance of these adaptive composite grid solvers. This is an important result because time-dependent applications are critical and computationally expensive. A motivating use for massively parallel machines is this problem class: that they can be solved using parallel adaptive mesh refinement, and that the solution methods do not degrade by use of adaptive techniques, are significant results. Such results are in some respects
counter-intuitive due to the non-uniform nature of the composite grid, but this is countered by the design of the composite grid solver AFAC, since AFAC allows a decoupling of the composite grid levels. It is the design of AFAC (and we expect the even better properties of AFACx) that allow the decoupling and, as a result, the much greater freedom to partition the composite grid more efficiently in the multiprocessor system.

4.6 Self-Adaptive Refinement Using P++/AMR++

By example, we demonstrate some of the features of AMR++ and examples for the support of P++ for the design of parallel self-adaptive block structured local refinement applications on the basis of FACx and AFACx algorithms. The singular perturbation equation is an interesting example problem since it is not as simple as the Poisson equation and provides a mechanism to introduce more realistic singularities for which we can better justify the use of self-adaptive mesh refinement.

In a parallel environment, partitioning the composite grid levels becomes a central issue in the performance of composite grid solvers. In figure 14, two different partitioning strategies that are supported within P++/AMR++ are illustrated for the composite grid. For ease of illustration, grid blocks 2.2 and 2.3 are not included. The so-called FAC partitioning in figure 14(b) is typical for implicit and explicit algorithms, where the local refinement levels have to be treated in a hierarchical manner (FAC, MLAT,...). The so-called AFAC partitioning in figure 14(a) can be optimal for implicit algorithms that allow an independent and asynchronous treatment of the refinement levels. In case of AFAC, however, we must consider that this partitioning is only optimal for the solution phase, which dominates the computational work of the algorithm. The efficiency of the level transition phase, which is based on the same hierarchical structure as FAC and which can eventually dominate the aggregate communication work of the algorithm, highly depends on the architecture and the application (communication / computation ratio, single node (vector) performance, message latency, transfer rate, congestion, ...). For this reason, additional work should be done to
Table 8: Timings for repositioning grids.
Table 9: Timings for addition of new refinement.
Table 10: Relative timings of AFAC for Moving, Adding, and Load Balancing (MLB).
eliminate this final coupling between the composite grid levels (see chapter 5). For determining whether AFAC is better than FAC in a parallel environment, the aggregate efficiency and performance of both phases and the relation of the convergence factors must be properly evaluated. For more details, see [31] and [27]. Both types of partitioning are supported in the P++/AMR++ environment.

Solvers used on the individually partitioned composite grid levels make use of overlap updates within P++ array expressions that automatically provide communication, as needed. The intergrid transfers between local refinement levels, typically located on different processors, rely on VSG updates. The VSG updates are also provided automatically by the P++ environment. Thus, the underlying support of parallelism is isolated in P++ through either overlap update or VSG update, or a combination of both, and the details of parallelism are isolated away from the AMR++ application. The block structured interface update is handled in AMR++. However, communication is hidden in P++ (mostly VSG update). See section 3.6 for more detail on AMR++, and section 3.5 for more detail on P++.

The use of the tools described above is now demonstrated with preliminary examples. The adaptivity provided by AMR++ is necessary in case of large gradients or singularities in the solution of the PDE. They may be due to rapid changes in the right-hand side or the coefficients of the PDE, corners in the domain, or boundary layers, for example. The details of the self-adaptive regridding are contained in section 3.6.5. Here, the first and last cases will be examined on the basis of model problems. This work is not intended to represent current research specific to the singular perturbation problem. The reader is referred to [33] for more detail.

Singularly perturbed PDEs represent the modelling of physical processes with relatively small diffusion (viscosity) and dominating convection. They may occur as a single equation or within systems of complex equations, e.g., as the momentum equations within
the Navier-Stokes or as supplementary transport equations in the Boussinesque system of equations. Here, we merely treat a single equation, but we only use methods that generalize directly to the more complex situations. Therefore, we do not rely on the direct solution methods provided by downstream or ILU relaxations for simple problems with pure upstream discretization. The latter are not direct solution methods for systems of equations; cf. [20]. Moreover, these types of flow direction dependent relaxations are not efficiently parallelizable in the case of only a few relaxations, which is what is usually used in multilevel methods. This, in particular, holds on massively parallel systems.

As opposed to the Poisson equation, the convergence factors do not only depend on the PDE, but also on the particular solution. The results in this thesis have been obtained for the exact solution

\[ u(x) = \frac{e^{(x-1)/\varepsilon} - e^{-1/\varepsilon}}{1 - e^{-1/\varepsilon}} + \frac{1}{2} e^{-100(x^2+(y-1)^2)}. \]

This solution exhibits a boundary layer at \( x = 1, 0 \leq y \leq 1 \) and a steep hill around \( x = 0, y = 1 \) (see figure 14(c)). In order to measure the error of the approximate solution, a discrete approximation to the \( L_1 \) error norm is used. This is appropriate for this kind of problem: For solutions with discontinuities of the above type, one can observe first order convergence only with respect to a norm of this type (no convergence in the \( L_{\infty} \) norm, order 0.5 in the \( L_2 \) norm).

The results have been obtained with the flagging criteria

\[ h^f \left[ \beta h \max(|a|, |b|) \left( |D_{h,x} u| + |D_{h,y} u| \right) \right] \geq \eta \]

with a given value of \( \eta \). For \( \varepsilon < \varepsilon_h \), the second factor is an approximation to the lowest order error term of the discretization. Based on experiments, \( f = 1 \) is a good choice. Starting with the global grid, the composite grid is self-adaptively built on the basis of the flagging and gridding algorithm described in section 3.6.5.

In table 11, the convergence factors for MG and FAC are presented for three values
<table>
<thead>
<tr>
<th>k</th>
<th>( e )</th>
<th>( n )</th>
<th>( e )</th>
<th>( n )</th>
<th>( b )</th>
<th>( e )</th>
<th>( n )</th>
<th>( b )</th>
<th>( e )</th>
<th>( n )</th>
<th>( b )</th>
<th>( e )</th>
<th>( n )</th>
<th>( b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/32</td>
<td>0.0293</td>
<td>961</td>
<td>0.0293</td>
<td>961</td>
<td>1</td>
<td>0.0293</td>
<td>961</td>
<td>1</td>
<td>0.0293</td>
<td>961</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1/64</td>
<td>0.0159</td>
<td>3969</td>
<td>0.0160</td>
<td>1806</td>
<td>4</td>
<td>0.0160</td>
<td>1967</td>
<td>4</td>
<td>0.0159</td>
<td>2757</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1/128</td>
<td>0.0083</td>
<td>16129</td>
<td>0.0089</td>
<td>3430</td>
<td>10</td>
<td>0.0087</td>
<td>3871</td>
<td>10</td>
<td>0.0083</td>
<td>6212</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1/256</td>
<td>0.0043</td>
<td>65925</td>
<td>0.0056</td>
<td>6378</td>
<td>19</td>
<td>0.0051</td>
<td>7943</td>
<td>16</td>
<td>0.0043</td>
<td>13473</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1/512</td>
<td>0.0023</td>
<td>261121</td>
<td>0.0073</td>
<td>12306</td>
<td>34</td>
<td>0.0044</td>
<td>15909</td>
<td>30</td>
<td>0.0023</td>
<td>27410</td>
<td>22</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 11. Accuracy (L1-norm \( e \)) vs. number of grid points \( n \) and number of blocks \( b \) for MG-V on a uniform grid and FAC on self-adaptively refined composite grids.

of \( \eta \), with two of the corresponding block structured grids displayed. The corresponding error plots give an impression of the error distribution restricted from the composite grid to the global uniform grid. Thus, larger errors near the boundary layer are not visible. These results allow the following conclusions:

- In spite of the well-known difficulties in error control of convection-dominated problems, the grids that are constructed self-adaptively are reasonably well suited to the numerical problem.

- As long as the accuracy of the finest level is not reached, the error norm is approximately proportional to \( \eta \). As usual in error control by residuals, with the norm of the inverse operator being unknown, the constant factor is not known.

- If the refinement grid does not properly match the local activity, convergence factors significantly degrade and the error norm may even increase.

- Additional tests have shown that, if the boundary layer is fully resolved with an increased number of refinement levels, the discretization order, as expected, changes from one to two.

- The gridding algorithm is able to treat very complicated refinement structures efficiently: The number of blocks that are created is nearly minimal (compared to hand coding).

- Though this example needs relatively large refinement regions, the overall gain by using adaptive grids is more than 3.5 (taking into account the different number of
points and the different convergence factors). For pure boundary layer problems, gain factors larger than 10 have been observed.

- These results have been obtained in a serial environment. AMR++, however, has been successfully tested in parallel. For performance and efficiency considerations, see sections 3.5 and 3.6.
Figure 17. Regular single level strip partitioning of a 3-level composite grid structure onto 4 processors (FAC).

Figure 18. Irregular multilevel strip partitioning of a 7-level composite grid structure onto 16 processors (APAC)
Figure 19: Complex composite grid problem with 40 patches.
Figure 20. Timings in milliseconds regarded with respect to patch and parallel processor system size (AFAC: black bars, FAC: white bars).
Figure 21: Timings for Load Balancing using MLB
Figure 22. Results for a singular perturbation problem: Plots of the solution, the error and the composite grid with two different choices of the accuracy $\eta$ in the self-adaptive refinement process.
CHAPTER 5

CONCLUSIONS

Adaptive mesh refinement and its use on parallel architectures is both practical and worthwhile. The parallelization of traditionally serial algorithms in the parallel environment, while commonly accepted and often reasonably efficient, is shown to be satisfactory in many specific circumstances on small numbers of processors. However, the advantages more specifically in the design of new mathematical algorithms are demonstrated through the development of AFAC and, in particular within this thesis, AFACx. AFACx addresses the requirements of more general self-adaptive mesh refinement and, at the same time, uncovers a generalization of the original AFAC algorithm and improves on the efficiency of solving composite grid problems in parallel.

This thesis combines both the development of a new numerical algorithm to the adaptive solution of PDEs with developments in the software engineering of such complex parallel codes. The more computer science oriented developments have resulted in a greatly simplified environment for the construction of parallel numerical software in general. The resulting P++ environment, a C++ parallel array class library, is demonstrated on the target application area of this thesis: parallel adaptive mesh refinement software.

Initial results detailing performance with AFAC and FAC were implemented on a variety of available distributed architectures in an initial code that showed the necessity of a better development environment. In particular, it is shown that the good performance of AFAC and its superiority over FAC on many examples in a parallel environment is a property of the algorithm and not dependent on peculiarities of any one machine. The results show that algorithms that are more expensive on single processor architectures, but are very well
parallelizable, can be much superior on parallel architectures in many cases. A factor of roughly two in favor of FAC in serial mode gives way in favor of AFAC in parallel mode, especially on larger numbers of processors.

An additional and unexpected result of the difficult implementation work that was done to investigate the parallel properties of FAC, AFAC, and AFACx was the development of a superior architecture independent environment for programming. The use of C++ has permitted an expanded scope of the work that was originally attempted, much more so than what would have been possible in FORTRAN. The work motivated by the requirement to build such complex parallel adaptive mesh refinement codes has led to the development of a runtime interpretation of parallelism and a radically new programming environment. The array language environment protects the user from the difficulties of developing even complex software. Additionally, it eliminates the requirement of parallel debugging, a particularly difficult task that has historically limited the complexity of numerical software on distributed memory environments. More than anything else, this thesis concludes that adaptive mesh refinement for parallel architectures requires support from both improved mathematical algorithms and attention to advances in computer science.
CHAPTER 6

FUTURE WORK

We expect that the development of this work will continue into the foreseeable future. More specifically, the design of AFAC that has evolved into AFACx will continue to be improved. Similarly, the software that is used to develop the parallel adaptive mesh refinement work presented in this thesis will continue to evolve (P++ and AMR++) research. There are several problems that will likely be addressed through continued research and they separate into future work on algorithmic design of AFAC and future work on the design of P++ and AMR++.

The development of improved algorithms for the parallel processing of the composite grid, derived from the self-adaptive mesh refinement process, will continue to be an important research area. It is hoped that the software developments in this thesis will improve the accessibility of this research area. The development of AFACx greatly improves the ability of the fundamental concepts developed in AFAC to address more realistic problems with even greater efficiency. We expect further research specific to the adaptive algorithms to be done, which might include:

- Decoupling of Intergrid Transfers. Current work on AFACx simplifies the solvers on each composite grid level, but nothing has been done to simplify the final interpolation and projection processes that occur at each iteration. Additional work to decouple these processes, partially or completely, will be done so that this step can be more fully parallelized.

- Improved theoretical understanding of AFAC and AFACx, particularly the relationship of AFACx to BPX [43].
Development of the programming environment that was used to support the research into AFAC and AFACx is the second area of research. Additional work on P++ and AMR++ might include:

- Efficiency for P++. Comparable to that of FORTRAN. Current work is being done to make the P++ array class library as efficient as possible. Current serial C++ array classes in use at Sandia National Laboratories perform at 50-80% the level of FORTRAN on the Cray. Ongoing collaboration with James Peery and Allen Robinson at Sandia will allow for some improvements to be made to P++ in order to obtain similar results on the Cray 2, and the single node architectures of distributed memory architectures (scalar, vector, or superscalar). With respect to parallel efficiency and the amount and cost of communication, P++ already performs nearly as well as codes based on explicit message passing, though more analysis will be required to account for optimizations across multiple array statements. Such multiple expression optimizations are possible by hand, but are often tedious.

- Interpretation of Parallel Indexed Array Operations. An important core of P++ is the interpreted parallel array operations, specifically the message passing that each operation generates. Currently, the interpretation of the message passing between processors owning the array data is based on the use of a fixed overlap along each edge of the partition. The use of a dynamically sized overlap is required to handle more complex array statements efficiently, which requires greater overlap to avoid the more expensive VSG update (e.g., 4th-order operators). Future work should isolate this required functionality and parameterize it, by the overlap width, so that it can be made more broadly useful. Such work could predict the optimal overlap size at runtime and so dynamically provide increased optimization to the runtime interpretation of parallelism for other runtime environments, more than just P++.

- Mixing of Data Parallel Model with Task Parallel Model. Currently, P++ provides
a simplified access to data parallel efficiency. However, for efficiency in parallel adaptive mesh refinement applications, we require additional task parallel support. Thus, it is important to mix the two types of parallel support together. Language support at Caltech, CC++, provides access to task parallelism and so is a target for additional work to combine with P++ for simplified support of numerical software in the parallel environment. However, other work at University of Colorado at Boulder has focused library support of task parallelism. Additional work should be done to evaluate both approaches and discover which is best suited for use with parallel adaptive mesh refinement.

- Use with Portable Communication Libraries. P++ is currently implemented using the Intel NX communication library and an EXPRESS\textsuperscript{TM} \footnote{EXPRESS is a trademark of ParaSoft Corp.} -like portable communication library from Caltech. Current versions are running on the Intel iPSC/860 Hypercube, the Intel Simulator, Sun workstations, IBM PC, and the Cray 2. Future work might use the standardized message passing interface (MPI) within P++ or, alternatively, P++ might use PVM internally.

- Use of Portable Class Libraries. AMR++ is implemented using M++ and P++, but uses some standard class libraries that limit its use on some of these. Current work uses the AT&T Standard Components II class library to provide standardized helper classes (e.g., linked list classes), which would then be available on all the target machines. Further investigation will be required to see if such standard class libraries are portable and widely available enough to allow P++ and AMR++ to be used on even obscure machines. The alternative would be to directly support the similar interface of the linked list, serial array, and other class libraries within P++.

- Replacement of M++. It is not clear that further work should continue to use the M++ class library, since it is currently a bottleneck to the performance of P++ and AMR++.

M++ was originally used because it simplified the development of P++
and allowed only parallel issues to be researched. These issues were the original focus of our investigation. The success of P++ will strongly depend on its ability to provide near FORTRAN performance. Support of a similar, or improved, interface, but restricted to the requirements of P++ (and optimized for use with P++), could easily be substituted. Such an intermediate serial array class library could more easily be made portable, and heavily optimized, across the proposed target architectures.

- Optimized Message Passing Subsystem. The scheduling of message in P++ is currently handled in a simple statement by statement processing. A much more highly optimized message passing subsystem of P++ could dynamically treat collections of statements and, in doing so, schedule the message passing with overlapping computation. Additional combining of messages could greatly reduce the number of messages sent within the multiprocessor system. This is especially efficient in modern parallel computers that have improved communication hardware allowing up to 160+ Megabyte/sec transfer rates, but interject a software based communication layer, which universally leads to high message startup costs.

- Improved Control of Partitioning. This is a critical issue in the development of parallel software, since within P++ the algorithm definition separated from the partitioning, and so any partitioning of the data is considered valid. However, the control of partitioning of data is not currently easy to define or manipulate (unless the default partitionings are used). For most applications, the manipulation that algorithms require is restricted to the load balancer, which must then have explicit control. Such explicit control is available and sufficient in P++. However, the user manipulation is awkward since explicit positions must be computed and defined within the P++ interface to the partitioning. The object-oriented language provides a much simpler solution through the use of mapping objects. The mapping object would abstractly define a partitioning independent of the number of processors (thus
defining a portable mechanism for the definition of a partitioning). Each P++ array would naturally define a partitioning and thus would contain a stack of mapping objects that could be pushed on or popped off the stack. These mapping objects would define a partitioning and could be defined with scope so that, after an array's partitioning was manipulated within local scope, the previous partitioning would be reset after the new mapping object that changed the array went out of scope. In this way, scoping rules can be used to simplify the partitioning of the distributed arrays. Alternate manipulation of the mapping objects could permanently change the partitioning. Maintenance of very large parallel software projects could even require a system of permissions within the definition of the mapping objects. This is an extension of work done on scoping of Vienna FORTRAN partitions. However, it could be easily implemented in C++ for the P++ objects without manipulation of the compiler. P++ could be a substantial advantage in productively prototyping even such a FORTRAN partitioning control subsystem.

- **Performance Tool Interface for P++.** Since performance is an important goal of the P++ work, additional work should combine performance evaluation tools in a way that permits the user to interpret relative performance and so allow the user to optimize the P++ implementation. Such optimizations might be specific to a given machine architecture and would involve tuning of the partitioning and other controls available from in the P++ Optimization Manager. There are several parallel performance monitoring packages (some are available with each of the communication libraries, such as EXPRESS and PARMACS); it is not currently clear which should be used.

- **Time Dependent Explicit Hyperbolic Solvers with AMR++.** As a research tool,
AMR++ is currently being used to support study of the AFAC algorithm and variants of this algorithm designed for specific support of block-structured local refinement for potential flow problems. Additionally, one of the PPM codes has been ported to the P++ environment and is running on one processor of the Cray Y-MP, but more work specific to efficient use of P++ on the Cray is required to make it competitive with the FORTRAN versions. This work is mostly dependent on M++ working well on the Cray. The P++ PPM application has been designed to work with the AMR++ class library, using the PPM application as the single grid solver to build a P++/AMR++ application for the parallel environment. However, tests of these combined applications will require enhancements of AMR++ in order to handle nonuniform grids. This work has yet to be finished.

- **Unstructured Grids.** One of our goals is to provide abstractions for parallelism, local refinement, and adaptivity for the development of unstructured grid codes. However, direct work on unstructured grids seems ill-advised without substantial experience on the simpler case of structured grids. Thus, we have focused our initial work on block structured logically rectangular grids and will build on the abstractions that P++ and AMR++ provide for these more modest types of applications. This will provide the foundation for the unstructured grid cases to be supported later.

- **Flexibility of AMR++ for Larger Classes of Applications.** As already stated above, there is still work to be done to make AMR++ sufficiently general to run the test and other complex fluid flow applications. Currently, AMR++ is used to simplify research on adaptive local refinement for elliptic problems using the AFAC algorithm for solution of the composite grid problem (see the example in the previous section). To date, only simpler model problems have been tested. Nevertheless, the resulting work is already more sophisticated and much simpler to use than previous work done on local refinement methods for parallel environments (see [28], [29]).
• Simplify the Use of AMR++. Although not a part of the current implementation of AMR++, C++ introduces a template mechanism in the latest standardization of the compiler. The general purpose of this template language feature is to permit class libraries to use user-specified base types. Thus, a class library for linked lists could be supplied by defining the link data type. For AMR++, the template feature will be used to allow specification of the base solver and adaptive criteria for the parallel adaptive local refinement implementation. In this way, we hope that the AMR++ class library will provide a simple tool for the construction of adaptive mesh refinement codes from single grid applications. Similarly, we hope to show that the combination of AMR++ and P++ allow the construction of parallel adaptive mesh refinement codes from single grid (serial) applications.

• Disk Storage of AMR++ Grid Blocks. AMR++ has no provision for its grid blocks to be stored off line on disk. Such capability would be relatively simple to implement and would be required for large systems of equations such as are found in combustion codes modeling multiple species. Its use would help control the virtual paging that might otherwise be required to support the large memory requirements of such code. This feature would copy a similar feature in CMPGRD (a grid generation/discretization package for the use of structured grids with complex boundaries). Current work on C++ persistent objects might greatly simplify this proposed feature.
BIBLIOGRAPHY


