INFORMATION TO USERS

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

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

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

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®
Software Components for Simulation and Optimization

by

Shannon D. Scott

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree Master of Arts

Approved, Thesis Committee:

William W. Symes, Chairman
Professor of Computational and Applied Mathematics

Danny C. Sorensen
Professor of Computational and Applied Mathematics

Steven J. Cox
Professor of Computational and Applied Mathematics

Houston, Texas

April, 2001
Abstract

Software Components for Simulation and Optimization

by

Shannon D. Scott

Explicit-time finite-difference approximations to the acoustic wave equation admit data parallelism, in which the problem domain (a grid) is sub-divided among several processors. Communication must take place between the processors to update the inner-domain boundaries after each time step. Overlapped subdomains can defer communication until several time-steps have been taken. We show nonetheless that minimal overlapping produces the fastest run-times.

Complex simulation-driven optimization requires integration of subprograms with widely varying natural data structures and levels of abstraction. Component architectures provide an integration method that also alleviates platform and programming incompatibilities. A component architecture built on commodity software packages provided the necessary integration for our acoustic control application.

C++ Expression Templates allow a finite-difference equation, or most any other mathematical operation, to be represented in a more natural form than hand-coded loops, ideally without a loss of efficiency, by deferring the cost of evaluating the expression until assignment. In principle compilers can optimize the expression as a
whole. In practice, our experience suggests that compiler optimization must advance further before expression templates can reach their potential.
Acknowledgments

I would like to acknowledge a whole host of people who made this thesis possible.

First, and foremost, I'd like to think my advisor Bill Symes. I would have trouble listing all the things Bill did to make this possible. He was a sounding board for new ideas, a filter for bad ideas, and a general source of encouragement.

I would like to think my committee, Danny Sorensen and Steve Cox, for their input and feedback. They are, with Bill Symes, among the professors from whom I have learned the most.

I would like to thank Nate Winslow for keeping me sane by a constant stream of encouragement. Nate was key in helping me address all the small issues that can drive one crazy, which is a subject Nate knows a great deal about (the issues, that is). Nate made this whole process seem much more possible.

I would like to thank all my fellow graduate students in the department for listening to my early explanations of what I was doing, and offering advice on better ways of saying it.

I would also like to thank Daria Lawrence for everything she did, even though she told me not to. Daria really is a key to making this all possible, she just claims she's not. There would be many more "running with scissors" type accidents without her here to keep us all in line.
Contents

Abstract ii
Acknowledgments iv
List of Illustrations vii
List of Tables viii

1 Introduction 1
  1.1 Domain Overlapping for Finite-Difference Approximations 1
  1.2 Component Architectures for Scientific Computing 1
  1.3 Expression Templates for Finite Difference Approximations 3

2 Grid Overlapping for Finite-Difference

  Domain Decomposition 4
  2.1 Introduction 4
  2.2 Finite-Difference Approximation of the Acoustic Wave Equation 5
  2.3 Run-Time Model 7
  2.4 Measurement of Model Parameters 10
  2.5 Experiments 11
  2.6 Discussion 14
  2.7 Conclusion 19

3 Component Architectures for Scientific Computing 20
  3.1 Introduction 20
3.2 Component Background ........................................... 25
3.3 Experiment .......................................................... 27
  3.3.1 Mathematical Background ................................. 27
  3.3.2 Code ............................................................ 32
  3.3.3 Component Framework ..................................... 35
  3.3.4 Wave Simulators ........................................... 49
3.4 Results ............................................................. 51
3.5 Discussion .......................................................... 54
3.6 Conclusion .......................................................... 59

4 High-Level Encoding of

Finite-Difference Approximations ............................. 61
  4.1 Introduction .................................................... 61
  4.2 Deferred Evaluation ........................................ 62
  4.3 Expression Templates ..................................... 67
  4.4 Results .......................................................... 73
  4.5 Conclusion ........................................................ 74

5 Conclusion .............................................................. 76

Bibliography ............................................................ 79
## Illustrations

2.1 Finite Difference Stencil ........................................... 7
2.2 Strip Decomposition of Domain with Overlap .................. 7
2.3 Example of computation and update of point values in 1-D slice of the domain .......................................................... 8
2.4 Experimental and Estimated Run-Times as a Function of Overlap 13
2.5 Block Decomposition of Domain ..................................... 16

3.1 Client/Server Model ..................................................... 33
3.2 Client/Server Model with data local to server .................. 34
3.3 Client/Server Interaction with the ORB ......................... 38
3.4 Client/Server Communication over IIOP ......................... 39
3.5 Remote Object Controller ............................................. 43
3.6 Strip Decomposition of Domain ...................................... 49
3.7 Master/Slave Server ................................................... 51
3.8 Percentage of total time consumed by CORBA method call .... 54
3.9 Initial $400 \times 400$ wave field sampled at left boundary for 1000 time-steps 54
3.10 Final wave field sampled at left boundary ...................... 55
3.11 Abstraction and the Connection of Concrete Classes ........... 57

4.1 Operation Parse Tree .................................................. 67
Tables

2.1 Measured Values of Processor Parameter $\alpha$ ........................................... 12
2.2 Measured Values of Connection Latency in Microseconds ..................... 12
2.3 Measured Values of Connection Bandwidth in $\mu$sec / double-precision grid point ................................................................. 13

4.1 Run times in seconds for first expression (without difference operators) 73
4.2 Run times in seconds for second expression (with difference operators) 74
Chapter 1

Introduction

Simulation-driven control and optimization problems combine simulation and optimization codes. These problems often involve the connecting several different code libraries, each to solve a different portion of the problem. In this thesis, we study three related aspects of writing simulation-driven control codes: making the system more flexible, making the system more efficient, and making the code easier to write.

1.1 Domain Overlapping for Finite-Difference Approximations

Finite-difference approximations to the wave equation can be parallelized by dividing the domain among the available processors, resulting in $p$ sub-domains, where $p$ is the number of processes. Overlapping the sub-domains allows several time steps to be taken before the processors must communicate. The assumption is that, since communication is relatively expensive, performing several iterations before communicating will increase efficiency. However, we will show that for a reasonable set of assumptions, domain overlapping does not produce a faster code.

1.2 Component Architectures for Scientific Computing

The need to combine simulation and optimization codes arises when developing a simulator-driven control or optimization application. The simulation and optimization packages are often separate codes, that must be connected to form the overall application.
However, these code packages, or libraries, may not be directly compatible. This can be true when libraries are written in languages such as C++, in which new data types can be defined by the user. These data types are used to express the data structures used by the library, and so might be peculiar to the library. However, other libraries might use different types to express the same concept, which would complicate connecting the libraries, as the data types would either have to be translated between libraries, or one of the libraries would have to be modified to use the other's types. This is further complicated by using libraries written in different programming languages, which might have different concepts of user-defined types, or no user-defined types at all.

Components are a software concept whereby applications are divided into self-contained blocks of code. Each component provides an interface through which other components communicate. Dividing the system into more manageable pieces can greatly reduce developer time, and increase code reuse. Components can be tested as a single piece of code that performs one function. Components allow code to be more easily reused, as a potential user only has to familiarize themselves with the component's interface, and not with the internal workings of the component. This hiding of the internal workings allows issues such as parallelism, or complex algorithms, to be hidden from the end-user.

We studied a boundary-control problem for the acoustic-wave equation, which required connecting simulation and optimization codes. We simplified this connection by dividing the code into components. The component system was built using off-the-shelf software packages, such as the Hilbert Class Library for optimization, and CORBA (Common Object Request Broker Architecture) for inter-component communication. Two versions of the acoustic-wave simulator where written, a serial version, and a distributed-memory parallel version. The componentization hide the
issues of parallelism of the simulator, so that a non-parallel optimization library could be used. The resulting system is more flexible than if the code had been written as a single piece. For example, the simulators, serial and parallel, can be changed at run-time. Also, the system can be easily expanded by adding new components.

1.3 Expression Templates for Finite Difference Approximations

C++ allows the basic set of operators +, -, *, etc. to be over-loaded, that is applied to user-defined data types. We could define operators on finite-difference grids, but this tends to be inefficient, as the operators produce temporary grids. Expression templates overcome this by deferring the evaluation of any single operation in an expression to the evaluation of the whole expression. An entire operation is expressed as a structure of objects, which upon being assigned to the final target grid is evaluated. This also allows the entire expression to be optimized as a whole, which can result in faster running code.
Chapter 2

Grid Overlapping for Finite-Difference Domain Decomposition

2.1 Introduction

Large-scale finite-difference codes can benefit greatly from parallelization. Large problems may be parallelized in several ways. Parallelization by division of data, or domain decomposition, is an example method. Domain decomposition makes each processor responsible for a sub-domain of the total domain. In finite-difference codes, this is achieved by dividing the problem grid among the processors.

During a simulation, each processor will need to communicate with one or more of its neighbors to update inner boundary values. This communication can be slow in relation to processing speed, as is the case with the PC cluster used in this experiment. Overlapping of the sub-domains allows communication to be deferred until several iterations have been made, at which time a larger set of data is then communicated. This results in a trade-off between reduction of communication costs, and an increase in computational costs. This paper will explore this trade-off.

Domain overlapping for finite-difference approximations was explored by Meza and Symes [18]. The work reported here is an extension of that paper. An example of a parallel finite-difference code without domain overlapping is described in Minkoff [19].

In this experiment, domain overlapping was applied to an explicit-time-step finite-difference algorithm for hyperbolic equations. However, domain overlapping should be applicable to other parallel implementations of iterative finite-difference algorithms,
and possibly other iterative methods in general. We assumed a static division of the grid, as opposed to a dynamic load-balanced division.

We will derive a run-time model of the experiment from simpler models of computation and communication. Parameters in these simpler models will be measured from a cluster of PCs. We will verify the model experimentally. We will then show that for most problems, of the type investigated here, domain overlapping is not beneficial. We will also address issues of scalability of the method.

2.2 Finite-Difference Approximation of the Acoustic Wave Equation

The acoustic wave equation in 2-D can be written

\[ \frac{\partial^2 p}{\partial t^2} = c^2 \left( \frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} \right) \]

We can approximate the partial derivatives in space and time with a second-order difference approximation

\[ \frac{\partial^2 p}{\partial x^2} \approx \frac{2p(x) - p(x - \Delta x) - p(x + \Delta x)}{\Delta x^2} \]

However, the entire approximation can be more easily written with forward and reverse first-order difference, which have the form (for a function \( f(\tau) \))

\[ D_\tau^+ f(\tau) = \frac{1}{\Delta \tau} (f(\tau + \Delta \tau) - f(\tau)) \]

and

\[ D_\tau^- f(\tau) = \frac{1}{\Delta \tau} (f(\tau - \Delta \tau) - f(\tau)) \]

This results in the (2,2) finite-difference approximation to the wave equation

\[ D_t^+ D_t^- p = c^2 \left( D_x^+ D_x^- p + D_y^+ D_y^- p \right) = 0 \]
The approximation produces a sequence of pressure grids in time. Let \( t = t_0 + i\Delta t \), \( i = -1, 0, 1, 2, \ldots, N \), then let \( p^n \) denote the pressure grid at time \( t = t_n \), i.e. \( p(x, y, t_n) \). Assume the initial pressure grids \( p^{-1} \) and \( p^0 \) are given, then we wish to compute the pressure grids \( p^1, \ldots, p^N \). The discrete equation can be rearranged to show this more clearly

\[
\frac{p^{n+1} - 2p^n + p^{n-1}}{\Delta t^2} - c^2 \left( D_x^+ D_x^- p^n + D_y^+ D_y^- p^n \right) = 0
\]

Let the discrete Laplace operator be represented by

\[
\left( D_x^+ D_x^- p^n + D_y^+ D_y^- p^n \right) = Lp^n
\]

then

\[
p^{n+1} = 2p^n - p^{n-1} + c^2 \Delta t^2 Lp^n
\]

The boundary conditions are hidden in the discrete Laplace operator.

It is useful to separate the boundary conditions from the system at this point. An element of the grid \( p^n \) can be denoted \( p^n_{i,j} \). Let \((Lp^n)_{1,j}\) denote the first column of grid points adjacent to the left boundary, then

\[
(Lp^n)_{1,j} = \frac{-p^n_{0,j} + 2p^n_{1,j} - p^n_{2,j}}{\Delta x^2} + \frac{-p^n_{i,j-1} + 2p^n_{i,j} - p^n_{i,j+1}}{\Delta y^2}
\]

but, \( p^n_{0,j} = f(t_n) = f^n \). So, the discrete Laplace operator can be re-written to yield the discrete system

\[
p^{n+1} = 2p^n - p^{n-1} + c^2 \Delta t^2 Lp^n - c^2 \frac{\Delta t^2}{\Delta x^2} Jf^n
\]

where \( J \) is an injection operator.

The approximation can be conveniently coded as a stencil applied to the pressure/time grids. We must store two grids: the current time grid, and the last. The next grid overwrites the last. Application of the stencil only requires access to a few points around each grid point updated. Figure 2.1 shows the structure of the (2,2) stencil.
2.3 Run-Time Model

A run-time model is produced by adding the time necessary for a single processor to update its domain to the time needed to communicate internal boundaries amongst the processors.

Let's assume the problem domain is a grid of size $n \times m$. For this experiment, the problem domain is subdivided into strips, each of size $n \times \frac{m}{p}$, where $p$ is the number of processors.

![Strip Decomposition of Domain with Overlap](image)

**Figure 2.1** Finite Difference Stencil

**Figure 2.2** Strip Decomposition of Domain with Overlap

Boundaries between sub-domains are overlapped by a fixed number of grid points. The number of points overlapped determines the number of time steps that may be taken before communicating. If the finite-difference stencil has a half width of $w$, then $w$ grid points are required on each side of a given grid point to be updated.
Overlapping the grid $w \times q$ points allows $q$ steps to be taken without communication. This overlap is $w \times q$ grid points beyond the processor boundary. Each processor at a boundary will overlap $w \times q$ points with its neighbor, for a total of $2 \times w \times q$ grid points overlapped in total, per processor.

Figure 2.3 shows a cross-section of the boundary between processors 0 and 1. Circles represent grid points. Filled circles contain invalid data. The stencil's half-width, $w$, is one, and the overlap, $q$, is two. This allows two time steps to be taken before communicating.

Note that invalid grid point values only occur past the overlap boundary, and at least one processor always has a valid value for a given point.

\begin{align*}
  t = t_0 \\
  &\circ \circ \circ \circ \circ \circ \circ \circ \circ P0 \\
  &P1 \circ \circ \circ \circ \circ \circ \circ \circ \\
  \end{align*}

\begin{align*}
  t = t_1 \\
  &\circ \circ \circ \circ \circ \bullet \circ \circ \circ P0 \\
  &P1 \bullet \circ \circ \circ \circ \circ \circ \\
  \end{align*}

\begin{align*}
  t = t_2 \\
  &\circ \circ \circ \circ \circ \bullet \bullet \circ \circ \circ P0 \\
  &P1 \bullet \bullet \circ \circ \circ \circ \circ \\
  \end{align*}

**Figure 2.3** Example of computation and update of point values in 1-D slice of the domain

Each processor is responsible for updating a non-overlapped sub-domain of size $n \times \frac{m}{p}$ plus two overlapped regions of size $n \times (q - 1) \ w$, for a total of $n \times \frac{m}{p} + 2(q - 1) \ w$, at each time step. Each processor will update its region simultaneously with every other processor.
A simple linear model for the time necessary to update k grid points is

\[ t_{\text{comp}} = \alpha k \]

where \( \alpha \) is the time a processor must spend to update a single grid point.

Then the time necessary for a single processor to update its sub-domain is

\[ t_{\text{comp}} = \alpha \left( \frac{nm}{p} + 2(q - 1)wn \right) \]

A model for the time to communicate k grid points from a single processor to another processor is

\[ t_{\text{comm}} = \lambda + \beta k \]

where \( \lambda \) is the latency, and \( \beta \) is the bandwidth, of the inter-process connection.

After q time steps, the overlapping regions must be updated. This requires sending \( n \times qw \) data points to each neighbor, then receiving the same amount from each neighbor. Experience has shown that messages sent to multiple nodes, from a single node, overlap well. Experiments indicate that this is commonly the case, as much of the communication time is spent with the data in transit, at which time the processor can be sending another message. This means that sending a message to two processors, takes approximately the same time as sending the message to only one. So, exchanging \( n \times qw \) grid-points between processes will take time

\[ t_{\text{comm}} = \lambda + \beta qwn \]

Then, the total time (computation plus communication) necessary for q time steps is

\[ t_{\text{total}} = qt_{\text{comp}} + t_{\text{comm}} = \alpha q \left( \frac{nm}{p} + 2(q - 1)wn \right) + \lambda + \beta qwn \]

The average time per q time steps is

\[ t_{\text{ave}} = \frac{t_{\text{total}}}{q} = \alpha \left( \frac{nm}{p} + 2(q - 1)wn \right) + \frac{\lambda}{q} + \beta wn \]

(2.1)
We would like to know the optimal amount of overlap $q$, given $w$, $n$, $\alpha$, $\beta$, and $\lambda$. The first derivative of the average iteration time with respect to $q$ is:

$$\frac{dt_{ave}}{dq} = 2\alpha wn - \lambda q^{-2}$$  \hspace{1cm} (2.2)

The second derivative is:

$$\frac{d^2t_{ave}}{dq^2} = \lambda q^{-3}$$

which is always greater than 0.

Setting Equation 2.2 equal to zero

$$\frac{dt_{ave}}{dq} = 2\alpha wn - \lambda q^{-2} = 0$$

gives,

$$q^2 = \frac{\lambda}{2\alpha wn}$$

or,

$$q = \pm \sqrt{\frac{\lambda}{2\alpha wn}}$$  \hspace{1cm} (2.3)

### 2.4 Measurement of Model Parameters

Model parameter $\alpha$ is a measure of the algorithm's serial run time. It is measured by timing the code's execution in serial for several grid sizes. This produces a set of run-time/grid-size data pairs that are used to fit a linear model by the least-squares method. The coefficient of the linear model is taken to be $\alpha$. Note that grid-size is the total number of grid points $(n \times m)$.

Estimation of the parameters $\beta$ (bandwidth), and $\lambda$ (latency) are slightly more complicated. The time necessary to pass a message from one processor to another, and then back, is measured for messages of various sizes. This is repeated a large number of times for each pair of processors available. These measurements correspond
to round-trip time, so halving the time values approximates one-way time. An affine model is fitted to the data by least-squares. The linear term is taken to be $\beta$, while $\lambda$ is the constant term.

2.5 Experiments

The second-order stress-velocity formulation of the acoustic wave equation given earlier was used for experimentation. The half-width, w, of the scheme is 1. That is, updating a grid point requires knowing the value of one grid point to the right, and one grid point to the left.

The finite-difference code was written in C++, though few C++ extensions beyond C where used. The data grids where allocated as single blocks of memory, and accessed in a column-oriented fashion by index arithmetic, which best suits the strip-decomposition of the domain.

The parallel message-passing library MPI was was used for inter-process communication (LAM/MPI for Linux) [3] [4].

The code was run on a Linux cluster comprised of four PC nodes, which we call the Scyld cluster. Each node consists of a single Intel 300 MHz Pentium II processor, 250 Mbytes of SDRAM, and 4 Gbytes of local disk space. These are connected by 100 base T ethernet, without its own subnet.

An estimation of the computation-time model parameter $\alpha$ was made by timing the running of the serial finite-difference code for grid sizes of $50 \times 50$ to $500 \times 500$ double-precision grid points. A total of 100 iterations were run per processor on the Scyld cluster. The resulting estimates for $\alpha$ are (in $\mu$sec per grid point) found in table 2.1.

Latency ($\lambda$) and bandwidth ($\beta$) were measured as described above. A simple MPI application was written that passed a message of arbitrary size between two arbitrary
nodes. This was run for 10,000 iterations with pseudo-randomly chosen message sizes, originating nodes, and receiving nodes. The resulting data was processed to provide communication models for each pair of processors. Table 2.2 shows the latency measurements, where each row corresponds to a sending node, and each column corresponds to a receiving node. The average latency was \(198.0\ \mu\text{seconds}\).

Bandwidth measurements (\(\beta\)) (in \(\mu\text{sec} / \text{double-precision grid point}\)) are given in table 2.3, where each row corresponds to a sending node, and each column corresponds to a receiving node. The average bandwidth was 0.904 \(\mu\text{seconds per double-precision grid-point}\).

For experimentation, a grid size of \(256 \times 256\) was chosen. Inserting this, and the values above, into Equation 2.1 yields an average iteration time, in microseconds, of

\[
t_{\text{ave}} = 14826 + 474q + \frac{198}{q}
\]

<table>
<thead>
<tr>
<th></th>
<th>Scyld0</th>
<th>Scyld1</th>
<th>Scyld2</th>
<th>Scyld3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scyld0</td>
<td>-</td>
<td>198.9</td>
<td>188.2</td>
<td>194.8</td>
</tr>
<tr>
<td>Scyld1</td>
<td>229.9</td>
<td>-</td>
<td>182.0</td>
<td>178.5</td>
</tr>
<tr>
<td>Scyld2</td>
<td>207.3</td>
<td>201.8</td>
<td>-</td>
<td>186.1</td>
</tr>
<tr>
<td>Scyld3</td>
<td>237.4</td>
<td>192.2</td>
<td>178.6</td>
<td>-</td>
</tr>
</tbody>
</table>

**Table 2.2** Measured Values of Connection Latency in Microseconds
Table 2.3  Measured Values of Connection
Bandwidth in $\mu$sec \(\text{ }/\text{ double-precision grid point}

<table>
<thead>
<tr>
<th>Scyld0</th>
<th>Scyld1</th>
<th>Scyld2</th>
<th>Scyld3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scyld0</td>
<td>0.9056</td>
<td>0.9016</td>
<td>0.9015</td>
</tr>
<tr>
<td>Scyld1</td>
<td>0.9061</td>
<td>—</td>
<td>0.9020</td>
</tr>
<tr>
<td>Scyld2</td>
<td>0.9077</td>
<td>0.9046</td>
<td>—</td>
</tr>
<tr>
<td>Scyld3</td>
<td>0.9042</td>
<td>0.9031</td>
<td>0.9044</td>
</tr>
</tbody>
</table>

An estimation of the optimal value of overlap, $q$, from Equation 2.3 is

$$q = B + \sqrt{\frac{\lambda}{2\alpha \omega n}} = \sqrt{\frac{198}{2 \times 0.925 \times 1 \times 256}} = 0.65$$

or, rounded to the nearest whole number, $q = 1$.

This is essentially no overlap, as $q = 1$ is the minimum overlap allowed by the finite-difference algorithm. That is, for a finite-difference scheme with a stencil half-width of 1, each application of the finite-difference stencil will require the neighboring point on either side of the point being processed.

![Image](image-url)

Figure 2.4  Experimental and Estimated Run-Times as a Function of Overlap
Figure 2.4 shows the run-times of 100 experiments with varying overlaps. The solid line is the predicted run-time, as a function of overlap.

2.6 Discussion

Domain overlapping is a method by which latency costs can be traded for extra processing costs. Therefore, this method would seem most applicable to fast processors connected by a slow network. The latency savings comes by saving information to be sent only after several time steps. This requires overlapping domains, to allow extra time steps to be taken without communicating. These larger domains require extra processing, which in the overlapped regions is redundant. This trade-off must be balanced to allow the fastest possible run-times.

Note that connection bandwidth does not affect the trade-off. This is because communicating $n$ grid points per cycle is approximately equal in time-cost to communicating $q \times n$ grid points every $q$ time steps.

Experimental date from the Scyld cluster shows that no overlapping of the subdomains ($q = 1$) is optimal. This is in agreement with the predicted value from Equation 2.3.

Equation 2.3 shows that the optimal amount of overlap increases with increasing latency. The Scyld clusters' ethernet connection is known for high latency; many other connection methods show a much reduced connection latency [9]. Therefore, domain overlapping would be even less useful for faster connections.

Equation 2.3 also shows optimal overlap decreasing with an increase in $\alpha$, a measure of the algorithm's serial run time. This implies that a sufficiently fast problem will benefit from domain overlapping. This may become the case as processor speeds improve. However, many finite-difference schemes are more costly than the acoustic-wave model chosen here, and so will benefit even less from domain overlapping [21].
A quick calculation shows that a speed improvement of almost a factor of 6 would be necessary before domain overlapping would be useful for this experimental problem.

Finally, Equation 2.3 also implies that domain overlapping will be beneficial for a sufficiently small problem. As a result of the domain decomposition chosen for this experiment, benefits from overlapping will not change with increasing numbers of processors. As the number of processors is increased, the area assigned to each processor decreases. However, the size of the boundaries between the sub-domains does not decrease, since the domain is decomposed into strips. Therefore, the size of the overlapped region does not change. Since domain overlapping trades communication latency for computation time in the overlapped boundaries, increasing the number of processors has no effect. For this decomposition, the problem itself must be small enough to benefit from domain overlapping.

If we assume that the value given by Equation 2.3 is rounded to the nearest whole number, than a value of $q = 1.5$ defines the break-point beyond which domain overlapping is useful. A quick rearrangement of Equation 2.3 gives, for the Scyld cluster

$$n = \frac{\lambda}{4.5aw} = 47.6$$

So, a problem with a height of 47 grid points, or less, will benefit from domain overlapping. Note that this says nothing of the width of the problem domain.

This oddity can be circumvented by choosing another domain decomposition. A strip decomposition, where only one dimension is sub-divided, was chosen for this experiment for its simplicity. Each processor communicates a fixed-sized boundary region with its two neighbors. Another option is to sub-divide both dimensions of a two-dimensional domain. This results in each processor being responsible for a rectangular region that may border other processors on all four sides. Inter-processor communication will likely scale better with this method, as the communicated bound-
aries are made smaller by increasing the number of processors. Figure 2.5 shows an example of the two-dimensional blocks domain decomposition method.

However, the block decomposition is most beneficial for a large number of processors. Our cluster of PCs is comprised of only four processors, and so does not represent a sufficiently large number of processors to see the benefits of the block decomposition method experimentally.

<table>
<thead>
<tr>
<th></th>
<th>P0</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td></td>
<td>P3</td>
</tr>
<tr>
<td>m</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 2.5** Block Decomposition of Domain

A run-time model for the block decomposition of figure 2.5 can be easily derived using the tools we developed earlier. Assume that a domain of size $n \times m$ is subdivided among $p$ processors. Also assume that $\sqrt{p}$ is a whole number, for simplicity. Than each sub-domain will be of size $\frac{n}{\sqrt{p}} \times \frac{m}{\sqrt{p}}$.

If periodic boundary conditions are assumed, than each processor will have to communicate with four neighbors. The amount of data communicated with each will be determined by the amount of overlap shared with that neighbor. If we wish to take $q$ steps without communicating, and the half-width of our finite-difference stencil is $w$, than a processor will overlap $\frac{w+1}{\sqrt{p}}$ grid points with a horizontal neighbor, and $\frac{w}{\sqrt{p}}$ grid points with a vertical neighbor.

Each processor's sub-domain will also overlap its diagonal-neighbor's domains. We will consider these regions negligible, and not count them in the model.
In total, each processor is responsible for a sub-domain of size \( \frac{mn}{p} + 2wq\frac{n}{\sqrt{p}} + 2wq\frac{m}{\sqrt{p}} \).

Processing this area will take time

\[
t_{comp} = \alpha \left( \frac{mn}{p} + 2wq\frac{m + n}{\sqrt{p}} \right)
\]

per iteration.

Every \( q \) time steps, the processors must communicate. We will assume that these communications will overlap, and so only count the time of one. Let \( l = \max(m, n) \), then the time necessary to communicate with all neighbors will be

\[
t_{comm} = \lambda + \frac{wq\beta l}{\sqrt{p}}
\]

The time-cost for \( q \) time steps will be

\[
t_q = qt_{comp} + t_{comm}
\]

and the average time-cost per iteration will be

\[
t_{average} = \alpha \left( \frac{mn}{p} + 2wq\frac{m + n}{\sqrt{p}} \right) + \frac{\lambda}{q} + \frac{w\beta l}{\sqrt{p}}
\]

(2.4)

The first derivative of Equation 2.4 is

\[
\frac{dt_{average}}{dq} = 2\alpha w\frac{m + n}{\sqrt{p}} - \frac{\lambda}{q^2}
\]

(2.5)

The second derivative is

\[
\frac{d^2t_{average}}{dq^2} = \frac{\lambda}{q^3} > 0
\]

Setting Equation 2.5 equal to zero yields

\[
q^2 = \frac{\lambda\sqrt{p}}{2\alpha w(m + n)}
\]

(2.6)

this equation differs from it previous analog by the additional factor of \( \sqrt{p} \). This shows that it will scale with the number of processors, but only by a factor of \( p^{\frac{1}{2}} \). In
fact, domain overlapping becomes more useful as the number of processors increases. This can be seen by noting that as the number of processors increases, each is given a smaller amount of overlapped region to process. This causes the ratio of time each processor spends communicating, as compared to computing, to increase. This makes the connection-latency costs more noticeable. However, the second domain-decomposition method also decreases the inner-boundary size with increasing numbers of processors, so overlapping becomes less costly. This produces a new, optimal trade-off between overlapping and latency, which is a function of the number of processors.

Inserting the constants from the earlier experiment ($\alpha = 0.925\mu sec / data\ point$, $\lambda = 198\mu sec$, $w = 1$, $p = 4$, $m = n = 256$) yields an optimal overlap of $q = 0.64$, or, rounded to the nearest whole number, $q = 1$. Therefore, domain overlapping is not beneficial for the second domain-decomposition method applied to the problem on the Scyld cluster, either.

More than four processors is needed for the effects of domain overlapping to be useful. Using values from the Scyld cluster, Equation 2.6 can be extrapolated, to determine at which number of processors overlapping might be beneficially employed. Inserting the Scyld clusters’ values for this experiment ($\alpha = 0.925\mu sec / data\ point$, $\lambda = 198\mu sec$, $w = 1$, $p = 4$, $m = n = 256$) gives

$$q^2 = 0.209\sqrt{p}$$

Since any useful value of $q$ must be a whole number, an optimal $q = 1.5$ is the smallest number that rounds to $q = 2$. Inserting this value of $q$ gives $p = 116$ processors.

This implies that domain overlapping can be useful when a sufficiently large number of processors are available. However, we can not be sure that the simple timing models developed in this experiment will hold for larger networks of processors. It
seems reasonable that a larger system will use a more complicated connection hierarchy, and therefore require a more complex communication model.

2.7 Conclusion

Domain overlapping trades communication latency costs for computational costs. For many applications, the cost of computation is higher than the cost in communication latency. For these problems, domain overlapping is not beneficial. Only very simple problems, run on systems with very high communication latency, will benefit from domain overlapping.

A relatively simple two-dimensional problem was explored here. Many problems, especially three-dimensional problems, are more computationally costly. Also, the cluster of PC's used were connected by ethernet, which has a relatively high communication latency. However, the problem did not benefit from domain overlapping for problems of a realistic size. This is nice from the perspective of code complexity, as domain overlapping is more complicated than the equivalent code without overlapping.
Chapter 3

Component Architectures for Scientific Computing

3.1 Introduction

Scientific software libraries are often designed to solve a set of related problems. Scientific applications often require the use of more than a single library to solve an entire problem. However, combining libraries in a single application can lead to unwanted complexities.

The software libraries may not be directly compatible. For example, they may not be written in the same, or even compatible, languages, for example Fortran 90 and C++. One library may require a model of programming that is not appropriate for the others. For example, a library intended for use on parallel computers may place requirements on serial libraries that can not be easily met.

Object-oriented libraries can also add complication in that the key concept of object-orientation involves the definition of new data types. These data types have to be handled by any code interfacing with the library. Using two libraries in a single code can require that types be converted for passing between libraries, or that one library be modified to accommodate another's defined types.

This was not an issue with earlier non-object-oriented languages popular for scientific coding such as Fortran77; there where no user-defined types to cause problems. Fortran 77 libraries written by different individuals could be connected without the issues raised by object-oriented methods.

However, even non-object oriented code libraries can cause inter-penetration of defined types and concepts. An example of this is parallel message passing libraries,
such as MPI, which is used for distributed-memory parallel programming. A code using MPI must use the set of types MPI has defined, which are intended to ease the programming of parallel codes. However, these types are pervasive, in that they must be accounted for throughout the code, often in places where no parallelism is used. Also, adapting the code to work with another parallel library can be made difficult by the dependence of the code on the library's defined types. Some projects have decided to simply define the library's types as base types for the entire project, such as the TAO project, which is built on PETSc, an MPI-based library [7] [5]. This binds TAO to the use of MPI.

Our instance of this problem of combining different code libraries is the need to combine software libraries to solve simulation based control and optimization problems. This often requires connecting one or more optimization libraries with one or more physical simulators. The simulation and optimization libraries are often written by different sets of experts. The difficulty is making the libraries interact.

In this experiment, we present a problem of optimal control of an acoustic wave field. The problem requires finding a time-series of control vectors that, by way of the structure of the problem, dampen a given wave field. Solving the problem requires simulating the wave field, and finding improved control series, which involves the coupling of a simulation package to an optimization package.

The optimization package used was the Hilbert Class Library (HCL), an object-oriented C++ package that abstracts the concepts common to optimization problems [2]. HCL provides a hierarchy of C++ classes in which optimization problems are more easily framed. This made writing the optimization component very simple.

Two acoustic-wave simulator where developed for the experiment. The first was a serial application; the second was a distributed-memory parallel application which used MPI. The parallel simulator followed the form described in the first chapter of
this paper. Both simulators where written in C++, but could have been written in any other appropriate language, such as Fortran.

We solved the problem of connecting the optimization and simulation libraries by applying a software concept called components. We divide the problem into a HCL-based optimization component, and a simulation component.

Components are a means of structuring software by separating functionality into logical units. Each component is a self-contained entity that is capable of providing some set of services, which are defined in a stated interface. This allows the separation of the implementation of an algorithm, with its related details and complexities, from the application of the algorithm, or essentially a separation of implementation from interface.

Components can be used to separate a monolithic code so that the resulting portions are easier to develop and maintain. This separation also promotes code reuse, as each separate piece, since it has a more well-defined function, can be more easily applied to other problems.

In many ways, this is similar to the focus of object-oriented methods, which divide a code into smaller pieces by encapsulation of data and operations on the data, and dividing the encapsulated data into a hierarchy of related types. Components, however, stress the point of separation over abstraction, in fact, in his book, Szyperski claims that object-oriented and component techniques can be though of as being orthogonal [24]. The result of using components is code with clearer distinctions between logical sub-units. However, this does not mean that object-oriented and component methods can not be used together, in fact their combination can be a valuable tool.

Components can also be seen as a means of bringing separate codes together to solve a common problem. Components provide a mechanism by which the libraries
can be included in an application without undue complication. Components provide an interface, behind which all complications of the library, such as language, and other details of implementation, are hidden. Only types agreed upon, and stated in the interface, may be communicated to and by the component.

We liked the following conceptual model: components are similar to the workings of the United Nations. A large number of representatives, who may speak different languages, expect different protocols of communication, etc., are allowed to interact by the use of language interpreters and a defined structure for the proceedings. Note that ideally the translation is a matter of translating languages, and not meanings.

The component model has several advantages. The components are self-contained; there is nothing inside available to change. This means a user does not have to familiarize themselves with the internal workings of the component before they use it. The user simply has to know what the component does, and how it communicates with other components.

The hiding of the component's interior allows the component to be written in any language capable of solving the problem, and implementing the communication methods necessary. This means that components written in different languages can easily be connected to perform a task. Keeping the internal workings hidden also allows parallelism to be hidden in a way that allows parallel codes to interact with non-parallel codes.

Validation and testing of code is simplified since each component can be tested in isolation. Each component can be treated as a black-box, that is we are not concerned with the internal workings, we are only concerned that the correct outputs are returned for a given set of inputs. Since the internal workings can not be changed by the user, we do not have to worry that their modifications will invalidate the
component. Also, since the internal workings of the component can not be casually modified, there is no fear of someone making poorly conceived modifications.

Components simplify the writing of applications by groups of people, since each component can be developed separately.

A project related to the use of components in a more sophisticated way is the NetSolve project[8]. NetSolve uses the component concept to allow users to construct codes that will eventually be executed on several machines. The NetSolve architecture handles the issues of finding available computing resources, and portioning the tasks to the available computers. We are not as concerned with the ability to execute code on multiple widely separated machines, but simply to divide applications in to portions that can be more easily written.

Another related project is the Common Component Architecture group, which is developing a standard for scientific computing with components [6].

We needed a framework in which the components could communicate. For this we chose the Common Object Request Broker Architecture (CORBA), a "vendor-independent architecture and infrastructure that computer applications use to work together over networks" [20]. CORBA is well supported, with implementations for most every computer architecture and operating system [14].

CORBA supports a client-server model. That is, one application, the server, provides services to another application, the client. For our experiment, we found it most natural to code the optimization component as the client, and the wave simulator as the server.

The optimization client and parallel wave simulators worked together to solve the boundary control problem. The key point, however, was that neither sub-unit knew of the internal details of the other. The HCL optimization client was not a parallel
application, yet could be used to control the parallel wave simulator. Likewise, the simulation server knew nothing of HCL, yet could be controlled by the client.

3.2 Component Background

What is a software component? While several definitions of components exist [6] [24], we have found the following requirements to be useful

Components are building-blocks of code

Components are connection oriented

Components are binary composable

Components communicate in a known manner

Components are self-contained building-blocks of code from which software applications are constructed. A component is a deployable entity [24]; a unit of software that can be used without modification. The internal workings of the component are hidden from the user, which allows the developer to hide such issues as parallelism, and allows components to be written in different languages.

Components are often provided in a compiled form which deters modification. This promotes reuse since, if nothing can be modified, and the component fits the task, it is simply reused.

A component provides a known set of services to the application. The services are listed in a defined interface, which is specified in a language-neutral format. The defined interface is often in the form of a list of named operations that can be performed by the component [24]. Any data types, other than the most basic concrete types, that are communicated by the component are defined in the interface.
A component's interface forms a signature for that component, with the intention that any component that implements the interface can be used in place of any other component implementing the interface. An example of this is the serial and parallel simulators developed for this project. They support the same simulation interface, and so can be used interchangeably. Changing simulation servers does not require changing, or modifying, the optimization client.

Applications are constructed by connecting components which provide the needed functionality. The components perform the desired task by communicating messages amongst themselves. These communications are in the form of concrete streams of data, which results in a message that can be easily transported between applications. The messages can also be easily transported over a network connection, which allows applications to be constructed that execute on more than one computer.

The fact that the messages must be representable as concrete streams of data, and that components may be written in more than one language, implies that the types communicated between the components must be completely specified in the interface. This limits the use of objects as messages, unless the components support the same set of object definitions. In general, it is simpler to avoid communicating non-concrete types.

The actual communication is commonly handled by a "middle-ware" layer: the component framework. The framework provides communication protocols for the components, as well as a means of defining the interfaces. The framework hides the actual mechanism by which the messages are relayed, so the application looks the same whether the components are connected by inter-process communication, network sockets, or some other means.

Components are binary composable. That is, components are connected without the need for recompiling; no binary modification is made to the components when
connected. This allows components to be connected to the application while the application is running, which allows an application to dynamically structure itself. The ability to link at run-time would allow, in more complicated applications, the ability to locate and use the best resources available at run time [8]. In general, run-time composure results in a more flexible application.

Are there examples of components in widespread use today? Surprisingly, a Fortran 77 library is an example of a component. The library is "connected" by linking it into the application. The linking does not require that the library be re-compiled, and could be done at run-time. The interface is a list of procedures that can be used by the application. The interface is simple, since Fortran 77 does not support user-defined types; only basic types, such as integers and floating point numbers, are communicated through the interface. Fortran 77 libraries can be called from other programming languages, such as C/C++ and Java.

Modern operating systems are also an example of component systems [24]. Each application, as well as lower sub-system services such as device drivers, can be thought of as a component. The operating system provides the framework in which the components communicate, which is often a system with even wider applicability, such as Microsoft's component object model (COM), or CORBA as used by the Gnome desktop project for Linux [12].

3.3 Experiment

3.3.1 Mathematical Background

We performed a simple experiment to test the applicability of the component concept to scientific programming. The goal was to minimize the energy in an acoustic wave field by controlling one boundary of the domain. More precisely, we started with the
2-D acoustic wave equation

\[
\frac{\partial^2 p}{\partial t^2} = c^2 \left( \frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} \right)
\]

with inhomogeneous initial conditions \( p(x, y, 0) = p_0(x, y) \). Where \( p(x, y, t) \) is the pressure at point \((x,y,t)\), and \( c \) is the wave velocity. The wave velocity was constant for this experiment.

We measure the pressure as a function of time near the left-most boundary: \( p(x_0, y, t) \), where \( x_0 \) is near \( x = 0 \). We use this information to control the left-most boundary: \( p(0, y, t) = f(y, t) \). The other boundaries are set to zero (Dirichlet condition).

We wish to chose \( f(y, t) \) to minimize the flux through \( x_0 \), ie. roughly the mean square of \( p(x_0, y, t) \).

We form the finite-difference approximation, as described in Chapter 1. This gives us a system of the form

\[
p^{n+1} = 2p^n - p^{n-1} + c^2 \Delta t^2 Lp^n - c^2 \frac{\Delta t^2}{\Delta x^2} J f^n
\]

If we assume the pressure grids are stored in vectors, where each vector is stored in a column-oriented manner (column one, then column two, ...), then the injection operator \( J \) is the \( N_z N_y \times N_z \) block matrix

\[
J = \begin{pmatrix}
I_{N_z} & 0 \\
0 & 0
\end{pmatrix}
\]

where \( I_{N_z} \) is the \( N_z \times N_z \) identity matrix.

Then the finite-difference approximation can be represented as a linear system

\[
\begin{pmatrix}
I \\
\hat{A} & I \\
I & \hat{A} & I \\
\vdots & \vdots & \vdots \\
I & \hat{A} & I
\end{pmatrix}
\begin{pmatrix}
p^1 \\
p^2 \\
p^N
\end{pmatrix}
+
\begin{pmatrix}
I & \hat{B} \\
0 & I \\
0 & 0 \\
\vdots & \vdots \\
0 & 0
\end{pmatrix}
\begin{pmatrix}
p^{n-1} \\
p^0 \\
p^0
\end{pmatrix}
- c^2 \frac{\Delta t^2}{\Delta x^2}
\begin{pmatrix}
J \\
J \\
\vdots \\
J
\end{pmatrix}
\begin{pmatrix}
f^1 \\
f^2 \\
\vdots \\
f^N
\end{pmatrix}
\]
where

\[ \hat{A} = \hat{B} = -c^2 \Delta t^2 L - 2I \]

We can now write the system in the form

\[ AP + BP_0 = \lambda^2 Jf \]

where

\[ \lambda^2 = c^2 \frac{\Delta t^2}{\Delta x^2} \]

Let \( F \) be an operator representing the operation of the wave equation. Then the system can be written

\[ F(P_0, f) = P \]

where \( f \) is the control sequence, \( P \) is the pressure grid sequence, and \( P_0 \) is the initial configuration, i.e. \( p^{-1} \) and \( p^0 \).

We will need a sampling operator, since we are only monitoring the left-most column of grid points in each pressure grid. The sampling operator must remove the first column of each pressure grid. Let \( Q \) be this sampling operator, it can be written

\[ Q = \begin{pmatrix} I_{N_z} & 0 & \cdots \\ & I_{N_z} & 0 & \cdots \\ & & \ddots & \cdots \\ & & & I_{N_z} & 0 & \cdots \end{pmatrix} \]

where \( I_{N_z} \) is the \( N_x \times N_z \) identity matrix.

Then, the optimization problem we wish to solve can be written

\[ \min_f \| QF(P_0, f) \|^2 \]

which, since \( F \) and \( S \) are linear, can be written

\[ \min_f \| QF(P_0, 0) + QF(0, f) \|^2 \]
Let the solution with initial data and zero control be

\[ QF(P_0, 0) = d \]

Then we wish to solve

\[ \min_f \|QF(0, f) + d\|^2 \]

This is an interesting statement. It implies that we should simulate the system with the initial data, but with no control, and then look for the control sequence that, with zero initial conditions, gives the same solution with opposite sign. This makes intuitive sense; we are looking for the wave field that will destructively interfere with the given field.

Let the total operator be

\[ Sf = QF(0, f) \]

Returning to the linear system

\[ AP + BP_0 = \lambda^2 Jf \]

and setting \( P_0 = 0 \) to give us \( F(0, f) \) gives

\[ AP = \lambda^2 Jf \]

or, solving for \( P \)

\[ P = \lambda^2 A^{-1} Jf \]

Add the action of the sampling operator to get the final output

\[ Sf = QP = \lambda^2 QA^{-1} Jf \]

We chose to use the conjugate-gradient method to solve the least-squares problem [13]. The conjugate gradient method is a good choice for solving the linear least-squares problem, as it only requires the action of the operator, and its adjoint. We
now need to derive the adjoint of the operator. For arbitrary \( u \) in the domain of \( S \), and \( v \) in the range of \( S \), we want an operator \( S^* \) such that

\[
\langle S(u), v \rangle = \langle u, S^*(v) \rangle
\]

So, if we have a vector \( v \) in the range of \( S \), then the adjoint would be

\[
S^*v = \lambda^2 (QA^{-1}J)^Tv = u
\]

or

\[
\lambda^2 JT A^{-T}QTv = u
\]

Let \( u = JT \hat{u} \), then

\[
u = JT \hat{u} = \lambda^2 JT A^{-T}QTv
\]

Canceling \( J \) gives

\[
\hat{u} = \lambda^2 A^{-T}QTv
\]

Redistributing terms gives

\[
A^T \hat{u} = \lambda^2 QTv
\]

Note that \( QT = J \). This gives

\[
A^T \hat{u} = \lambda^2 Jv
\]

So, to calculate the adjoint we must solve a system related to the forward system in that the system matrix is transposed. The forward map comes from a discretization that is explicit in time. This results in a system matrix that is lower triangular and can be solved by forward substitution. The adjoint system then is upper triangular and can be solved by backward substitution. This is similar to solving the discrete system backwards in time.
3.3.2 Code

We assumed the optimization and simulation components would be running in different memory spaces, which is standard practice when using CORBA. In fact, this was necessary as one version of the simulation component would be a distributed-memory parallel application, which would be composed of several processes running on separate machines. To keep the optimization client separate and non-parallel, it had to run in its own process.

A result of the decision to allow the components to run in separate memory spaces was that we had to be careful of efficiency losses due to excessive data movement. If an operation requires moving a piece of data, and the movement is more costly in time than the operation, then it is probably best to operate on the data locally.

A simple model of the system might have the optimization and simulation components passing entire data vectors between themselves; which would be extremely inefficient. This is especially the case for large data sets, such as the wave-pressure fields used by the simulator. The majority of the data is operated upon in the simulation server; the optimization client only has to perform a few simple operations per iteration. Therefore, a more efficient model is to have the optimization client instruct the simulation server on performing the optimization, without the client handling any of the data directly.

NetSolve does not enforce this minimization of data movement; it moves data to a machine that can operate upon the data. NetSolve makes a model of the time necessary to transport the given data, and the computational time on the available machines. This model is used to find the most efficient use of resources [8]. Of course, NetSolve is intended to be used to solve more general problems, while we have developed a method applicable to specific problems.
The requirement to minimize data movement impacts the server, as well. We could, for example, make a component for each object we wish to control. Interaction of the two server components would require a CORBA connection between them. For example, assume we have two data vectors which we wish to add. Each vector is represented by a separate component, in a separate memory space. Adding the vectors would require that the data from one vector be moved to the memory space of the other. Since all connections to the components are made through CORBA, moving the data would take place through CORBA. This would be extremely inefficient, in that we must move the data to perform the simple operation, and that we must move the data through CORBA, which is not intended to be used as a high-speed data conduit.

This implies that we must keep the vectors in a single memory space, which implies the need for a single server process which contains all the data needed to perform the simulation. In the multiple-process case, we would have had to manage multiple CORBA connections; now we only have one.

The code we developed has the optimization client "pulling the strings" of the simulator. For most any optimization method we choose to apply, the client only needs the result of a few simple linear algebra operations performed on the data.
and control vectors, for example inner products and "axpy" operations. However, the data and control vectors are stored on the server, so the server must be able to perform the operations. In other words, the client instructs the server to apply the vector operations to the data as needed. The client refers to the data by a unique identifier, which the server assigns to each data vector. This allows the client to create, and operate upon, data vectors as needed. Any scalar results of the operations are returned to the client.

![Diagram of Client/Server Model](image)

**Figure 3.2** Client/Server Model with data local to server

Therefore, to make a server that can be optimized by a given optimization client, we have to implement in the server every operation the client will need. However, this should be a small set of operations. We could produce a number of simulators that provide these operations, and a number of optimization clients that call them, and applications could be built by "mixing and matching" the components.

This "remote control" of the problem data is natural for an object-oriented library, such as HCL. HCL abstracts the issues of optimization to a level above the actual representation of the data. It does this by describing the data, and operations on the data, as part of a class hierarchy; algorithms in HCL are written in terms of base classes. For example, our data grids where presented as child classes of the HCL_Vector class; HCL operates on our data class as if it were a HCL_Vector. HCL is not concerned that the data is actually stored remotely.
Given the assumptions above (data stored in simulation server), the majority of the computational work will be done by the server. Therefore, the optimization client can be run most anywhere that is convenient, while the simulation server is run on a system providing the needed computational power. For example, the simulator could be run on a cluster of PCs, while the optimization component could be run on a desktop machine connected to the simulator by a network.

The system is not limited to only one optimization client and one simulation server component; other components can be added as needed. An example would be adding a component to provide visualization capabilities. We can also imagine more complicated problems that require the simulation of multiple phenomena in a single application, for which we could have one component for each simulator [22]. However, to continue the guarantee of efficiency, any added components must minimize their data movement, as well.

3.3.3 Component Framework

The optimization and simulation components must be connected by some mechanism so that they may communicate. The connection could be implemented in several ways. For example, if the components reside on the same physical machine, then we could connect them through some form of interprocess communication, such as named pipes. If the components were located on separate machines, then we might connect them by writing special network socket code. In either case, we would be required to write much of the code to make the connections.

Developing our own component communication system could be complicated. The details of the communication libraries might vary from machine to machine, for which our code would have to compensate, resulting in specialized code for each machine with limited portability. There is also the issue of variations in the representation of
data on the separate machines. Examples of these are differences in byte-ordering, and bit widths of "standard" types such as integers. Our component connecting code would have to compensate for these, and other, issues.

Also, we would like the inter-component communication to take the form of method calls, by which data would be passed. Neither network sockets nor interprocess communication support remote method calls in any direct form; both communication methods simply pass messages. We can extend interprocess communication and network socket communications to emulate the calling of methods by encoding the method calls in the messages passed between components. However, encoding the methods represents extra work on the part of the developer, and can be quite overwhelming when there are a large number of methods to be accounted for.

We could instead use a remote procedure call library (RPC), but in general these have the following deficiencies. A particular library is often intended to support one programming language, but we would like to be able to support multiple languages. Also, libraries may not be available for every computer architecture we wish to run on, and these libraries may not be able to communicate between one another. Finally, we would like to limit our dependence on any one library, and instead use a standardized method of inter-component communication.

We would like some mechanism to connect our components without having to address the issues above. These mechanisms for the connection of components are commonly called component frameworks. Component frameworks allow us to connect our components without concern for how the components will communicate. The framework can be thought of as providing an adapter, which we use to "plug" our components into the application.

We chose the popular component framework CORBA (Common Object Request Broker Architecture). CORBA provides the necessary code to "adapt" our compo-
ments to each other, and the communication methods to allow them to communicate. CORBA's communication model allows us to connect components, whether they are located on the same machine or different machines, without any changes in our code. CORBA also addresses the problems of data representation by converting when necessary.

CORBA supports a client-server model. The server component provides a set of services in the form of a list of methods it can perform. The client controls the server by calling methods from the server's list of methods. Data is passed as the arguments and return values of the method calls. The server only performs the operations it is instructed to perform by a client.

CORBA provides an Interface Definition Language (IDL) for defining the interface to a component. IDL is intended to be implementation-language independent, and provide a consistent form for the interface. Also, IDL provides support for exception handling, and various hierarchical mechanisms such as modules of methods.

CORBA is intended to address the problem of connecting codes written in different languages running on different machines [24]. CORBA can not only be used to hide language issues, but also issues of software libraries. Software libraries, especially object-oriented libraries, tend to be pervasive. That is, they often define new data types, and these data types must be supported by any code using the library. This can be a problem when the library defines types that are quite different, or even incompatible, with the structures used in the host code. Of course, the issue becomes even greater if the library was written in a different, possibly object-oriented, language from the host code. CORBA, and the component concept, can be used to hide these issues quite effectively, which proved very useful for this problem.

CORBA itself is simply a specification of requirements to which an implementation must conform. All implementations provide an Object Request Broker (ORB) which
provides the mechanism for communication between client and server. The ORB also provides several other services, such as the means of locating and controlling servers.

A CORBA implementation provides a compiler to compile the IDL interface into code in the target language. IDL compilers exist for many popular languages including C and C++. In the case of C being the target language, the compiler will produce a header file defining all types and methods used by the interface. The compiler also produces "stub" code for the client, and "skeleton" code for the server. The stub code provides functions for the client to call, while the skeleton provides a means for the server to register functions to be called. Both the header, stub, and skeleton codes are produced from the IDL interface.

![Diagram of Client/Server Interaction with the ORB]

**Figure 3.3** Client/Server Interaction with the ORB

If the client and server are running on separate machines, then each will be connected to a separate ORB running on that machine. Communication between the physically separated client and server is handled by communication between the ORBs. CORBA 2 [20] provides the Internet Inter-ORB Protocol (IIOP), a standard for communication between two CORBA ORBs over a TCP/IP connection. The format guarantees that different CORBA implementation ORBs can communicate, even if they are running on different platforms.
We chose to use the CORBA implementation ORBit, which is used by the Gnome desktop project, and is shipped with RedHat Linux. The computers in our PC cluster are running Linux, and so ORBit was already available. ORBit currently supports CORBA bindings to C, with a preliminary binding to C++. The C++ binding seemed to be only in the earliest stages of development when we started our project, so we chose to use the C binding. This impacted our project, as most of the code was already written in C++. Other CORBA implementations may have been more suitable to our needs, but we stayed with ORBit because it was so readily available.

**Optimization Client**

The optimization client was written in HCL. This project started with a conversation on the merits of parallelizing HCL. We realized that parallelizing HCL would require re-writing a large portion of the code. We also realized that there where several other unsatisfactory issues with parallelizing HCL. First, the optimization algorithms do not explicitly deal with parallelism, nor should they. Adding new optimization algorithms might be hampered if the developer had to deal with issues of parallelism, as well as any issues of the algorithm. Secondly, methods of parallelism vary according to details of the problem to be solved, and computer, or computers, on which the code
will be run. An example of this is the differences in programming for shared memory and distributed memory. Which would HCL support? Lastly, methods for parallelism tend to change, and choosing one method tends to tie the application to the method. An example of this is code written with PVM, which has been largely replaced by MPI. The message library tends to be rather pervasive in the code, and converting to a newer, more supported, library is not trivial. This is however, as mentioned earlier, contrasted by the TAO project's choice to choose one form of parallelism [7].

The desire to use HCL as part of parallel applications led us to the component concept. With this approach, we could parallelize the simulator, where most of the time-consuming work is performed, and leave HCL as is. This approach came with an issue of its own; HCL is very object-oriented, and any interface to it must be object-oriented. However, forcing the component interface to deal with the object issues would have caused problems. Supporting objects outside the component would require the component framework, and any component attached to this component to deal with objects.

The server might not be written in an object-oriented language, nor be able to provide any abstractions commonly found in object-oriented programming. For example, it is common for scientific codes to be written in Fortran 77, which is not object-oriented. CORBA can connect components written in different languages, so the component framework would not be an issue, but the F77 server would not know how to deal with the objects. It is often possible to simulate objects in a non-object-oriented language, but this should not be necessary, not to mention that the fundamental concept of components would be violated. That is we should be able to connect components without having to deal with issues internal to either component, including language issues. Allowing the optimization client to force the simulation
server to support objects, because the client was written in an object-oriented lan-
guage, would defeat the separation that components provide.

As mentioned earlier, for efficiency, the server must provide a single process to
which a client connects (through CORBA). The interface to the server provides a
method for every operation that might be performed on the data-objects the server
contains. However, the client was written with the HCL library, which is object-
oriented. HCL is intended to manipulate objects, and so the client interface had
to emulate the objects stored in the server component. Our solution to the issue
was the use of proxy classes/objects in the client component to represent remote
classes/objects stored in the server component. This follows the popular design pat-
tern of the same name [11]. Each remote object is represented by a proxy object on
the client.

An example is, the server houses a vector of type `RemoteVector`. The server
interface provides methods to operate on `RemoteVectors`. The client makes an object
of type `RemoteVectorProxy` to represent each `RemoteVector` stored on the server.
The client can then simply manipulate the proxies as if they were the actual data
containers.

The proxy classes are very simple. A proxy class provides a method corresponding
to each method in the component interface to which it connects. Calling a method
results in the appropriate CORBA method being called. Operations on the proxy
result in operations on the remote object.

The proxy stores a unique identifier for the remote object it represents; the iden-
tifier was assigned when the remote object was created. The identifier is passed with
each method call, so that the server knows to which remote object it should apply
the method.
Allocation and deallocation of remote objects are tied respectively to the constructor and destructor of the proxy object. Therefore, allocating a proxy object results in a remote object being allocated on the server. Likewise, deallocation of a proxy results in deallocation of the remote object the proxy represents.

For this experiment, the proxy classes were inherited from the appropriate HCL classes, thereby making them part of the HCL class hierarchy. For example, a proxy class for a remote vector would be made a child of the HCL class hclVector. This allows the proxy vector to be used in any HCL algorithm which operates on HCL vectors.

**Simulation Server**

The simulation server components were also written in C++, though in only a loosely object-oriented fashion. The servers could have been written in another language, such as Fortran.

As mentioned earlier, for efficiency it is necessary to store all data on the server, and have the server perform any operations on the data. These operations include the forward wave equation, adjoint wave equation, and any linear algebra necessary. The client requests an operation to be performed by passing, through a CORBA method call, identifiers to the vectors involved (input and output).

The server maintains a collection of data and operator objects. Operations are constructed by combining data and operator objects. The objects are controlled through a single interface, which we have come to call the Remote Object Controller (ROC). The ROC provides the interface for the server to the outside world. The ROC is the object that the CORBA interface supports; connecting to the simulation server simply implies connecting to the server's ROC.
The ROC provides a list of methods, one for each method of each object it controls. The method names are encoded to allow easier human interaction. For example, a method of a class named `RemoteVector` might have a method `Add`. This method would be listed in the ROC's method list as `RemoteVector/Add`. This a essentially an implementation of a name space, such as found in C++; the `Add` method is in the `RemoteVector` name space.

ROC methods are called through the CORBA interface. Arguments to the methods are all concrete types, or light-weight defined types, such as integers and floating-point values. Entire data vectors can not be passed through the interface. Though this would be useful in some cases, we did not implement the ability to make a point; moving large data sets is time consuming.

Objects stored in the server are referenced by a unique identifier. The identifier is assigned by the server at the time of allocation, and passed to the client requesting the allocation. These are the identifiers that are stored in the client's proxy classes. The ROC uses the identifier to locate arguments to a remote method call.

For this experiment, the identifiers where defined as 32-bit unsigned integers. This representation has the advantage of being small, and so easy to communicate and store. We could have implemented a more sophisticated identifier scheme, such as an identifier that is a human-readable text string, or a structure that contains
extra data relevant to the remote object, but we did not feel it was necessary for this project.

The identifiers are chosen by a simple method; they are the pointer to the remote object on the server, type-cast to an unsigned long integer. This has the advantage of simplicity. The server simply dereferences the identifier, after type-casting back to a pointer, to "locate" the referenced object. However, since the identifier is defined in the component interface as a 32-bit integer, 64-bit architectures would not be able to directly support this approach (their pointers would not fit into a 32-bit integer). A more elegant approach would be to keep a table on the server that relates identifiers to objects. Also, note that the identifiers are in no way modified by the client. They are simply stored when the object is allocated, and used later, as is, to reference it.

Once the arguments of a method call are located, by their identifiers, they can be operated on as expected. In this experiment, the remote objects are in fact C++ objects, and the ROC only has to call the appropriate method of for the base object. For example, assume we wish to add a remote vector \( x \) to another remote vector \( y \). Both are objects stored on the server, which where instantiated from a RemoteVector class. This class has a method of the form

\[
\text{RemoteVector}::\text{add}( \text{double scale, const RemoteVector} \, \& y )
\]

where \( \text{RemoteVector} \, \& y \) is a reference to another remote vector.

Then, the ROC will include a line in its method list

\[
\text{RemoteVector}_{...}\text{add}( \text{in RemoteVectorID thisVector,} \\
\text{in double scale,} \\
\text{in RemoteVectorID y) }
\]

where the "in" tags found before type definitions in the method's argument list specify that the values are to be sent, one-way, from the client to the server. There are also "out" and "inout" tags that perform one-way communication in the opposite direction
(a return value), and two-way communication (a passed value that can be modified).
The type RemoteVectorID represents the identifier used to reference the objects.
Note that we passed the identifier for both the vectors $x$ and $y$. We had to pass the
identifier for the vector $x$, the subject of the operation, since the ROC must find the
vector object to be able to call its associated Add method. From the perspective of
the $x$ vector, this seems to be more of a function call than a method call. In reality, it
is a method call to the ROC, the remote object to which other components connect.
The ROC forwards the call to the remote vector object.

The client initiates the addition by calling the add method of the $x$ vector's proxy
object

$$x.\text{Add}(3.4,y)$$

where $x$ and $y$ are of type RemoteVectorProxy. This results in the CORBA method
above being called with the identifiers for the two vectors, which where stored in the
proxies. The CORBA ORB forwards this call to the server, by way of the ROC. The
ROC's code for this method might be as simple as

RemoteVectorLookup( this ).Add( scale, RemoteVectorLookup( y ) )

where RemoteVectorLookup is a helper function that returns a reference to the remote
vector identified by the given ID.

Note that the actual RemoteVector method is called directly; the remote vector
is unaware that it is being controlled remotely. This greatly simplifies the job of
coding the server. In our experiment, the server was first written as a stand-alone set
of classes capable of simulating the problem. The ROC was then written to control
these classes.

Let us add another method to the ROC to further illustrate the structure of the
ROC and the server. Assume that the RemoteVector class also has a method
double RemoteVector::InnerProduct( RemoteVector &y )
that performs an inner product with another vector \( y \) and returns the result as a double.

We would first add the method to the ROC's interface. The interface would already contain our `add` method. So, with the addition of our new method, the interface might look like

```c
interface ROC
{
  typedef unsigned long RemoteVectorID;

  RemoteVector_add( in RemoteVectorID thisVector,
                   in double scale,
                   in RemoteVectorID y );

  double RemoteVector_inner( in RemoteVectorID thisVector,
                              in RemoteVectorID y );
}
```

The interface compiler produces a header file containing C language declarations of the types and methods defined by the interface. The interface compiler also produces C code, called a "skeleton", to be compiled with our server. The skeleton provides the code needed to implement the communication interface.

The body of the skeleton is a message loop, which directs incoming method requests to functions of our making. We must tell the skeleton which of our C functions are to be called when a specific method call is made through the interface. For this purpose, the interface compiler defines a C `struct` of pointers to functions; one
pointer for each method in the interface. We must provide a function implementation with the correct signature, which will be called by the skeleton.

For example, the entry in the structure corresponding to our new inner product function might look something like

```c
CORBA_double (*RemoteGrid_inner)

( PortableServer_Servant servant,
  const ROC_RemoteGridID thisVector,
  const ROC_RemoteGridID y,
  CORBA_Environment *ev );
```

This is a standard C-style definition of a pointer to a function that takes the listed arguments and returns a double. Note that the identifiers for the two vectors to be operated upon are accompanied by two other arguments. The first is a CORBA-specific structure that allows our code to interact with the CORBA subsystem; Servant is the CORBA term for the code that is actually doing the work of calling the function. In this experiment we did not need the servant in the ROC methods, and so it can be ignored. The final argument is a pointer to a structure that allows our code to monitor the state of the CORBA environment. We use this to report any errors. Also, note that the return type is actually a CORBA_double, which in this case is simply type cast to a standard C double.

Now, we must make a new function, to be added to the ROC, which will be called when an inner product is requested. This function must be able to find the two vectors identified in the method call, and call their inner-product method. The new function must have a definition that conforms to the pointer definition given in the structure found in the interface header file. Note that the name of the function is not important, though it is of course best to name it something sensible (it is not important because we will access it as a pointer).
recall that `lookUpRemoteVector()` is a helper function that returns a reference to the identified object.

Our new ROC function might then look like

```c++
double RemoteVector_inner( PortableServer_Servant servant,
    const ROC_RemoteGridID thisVector,
    const ROC_RemoteGridID y,
    CORBA_Environment *ev )
{
    return lookUpRemoteVector( thisVector ).InnerProduct(
        lookUpRemoteVector( y ));
}
```

We have ignored any error checking for clarity.

When the ROC is initialized, on starting the server application, the structure of pointers to ROC functions mentioned earlier must be initialized. For example, if the structure is declared as type `POA_ROC_funcs` and is named `ROC_funcs`, then we would initialize it for this simple example as follows

```c++
POA_ROC_funcs ROC_funcs = { &ROC_add, &ROC_inner };  
```

The structure would then be passed to the ORB.

We have glossed over many of the underlying details of the CORBA implementation. The bibliography lists several references to information on the CORBA details.

After writing the ROC code, we found that newer versions of ORBit (the CORBA ORB we used) can produce a file, given the ROC's interface, that is a framework for the ROC. This is selected by an optional flag passed to the IDL interface compiler. This would have made writing the ROC much less mechanical. Also, interface compil-
ers that target C++ produce base classes that are inherited to implement the server. This would have also made developing the ROC less mechanical.

3.3.4 Wave Simulators

As stated earlier, two versions of the wave simulator were made, a serial, and a parallel version. The wave simulators produce the finite-difference solution to the acoustic wave equation as described in chapter 1. The serial simulator was a straightforward application of the algorithm.

The parallel version was much more complicated. An initial distributed-memory parallel simulator was written in a single-program multiple-data (SPMD) form using MPI for message passing [10]. The simulator was then tested as a stand-alone application.

For the parallel simulator, the problem domain was divided into \( P \) sub-grids, where \( P \) is the number of processors available. Each sub-grid was located on the corresponding processor. Each sub-domain included boundary, or ghost, cells, to simplify boundary calculations. Along external boundaries the ghost-cells are set to the global boundary conditions. Internal ghost-cells overlap for adjoining sub-domains. A single time-step proceeds as outlined in chapter 1.

![Figure 3.6 Strip Decomposition of Domain](image-url)
Figure 3.6 demonstrates this sub-grid division for \( P = 4 \) processors. Recall that we control the left-most boundary, and monitor field values at the column adjacent to the boundary. Thus, the control and data vectors live entirely within the first process. This was a simplification that allowed us to use a non-parallel vector class to store and process the data. Otherwise, we would have had to use distributed vectors to represent any slice of the domain that crossed processor boundaries.

The structure of the component system imposes an interesting restriction on the server. As mentioned earlier, the client-server model, as supported directly by CORBA, requires a one-to-one connection: one client application to one server application. However, the parallel simulator is composed of multiple processes, and we do not want to use CORBA to connect them.

The simulator processes are already connected by an MPI network; we can use the MPI broadcast function to broadcast messages, passed through the CORBA interface, to the processes. This requires a single master process to which the client connects through CORBA. The master process houses the ROC. The master process receives the incoming method requests, packs them into an MPI buffer along with a tag encoding the method type, and broadcasts them to the processes in the simulator. The simulator processes unpack, and process, the message.

This is, however, no longer a true SPMD application, in fact, it has become a master/slave application. A master/slave application is an application in which a single process controls one or more drone processes. This complicates programming. In the SPMD model, the process execute the identical code. In the master/slave, two different code sets must be maintained. The master/slave model adds the overhead of first making the request to the master, which then forwards the request to the drones. Also, in this case, the CORBA method must be encoded, and packed into an
MPI buffer, before being transmitted. In general, the SPMD programming model is a simpler approach.

![Diagram of Master/Slave Server]

**Figure 3.7** Master/Slave Server

### 3.4 Results

The focus of this project was to test the viability of component techniques applied to scientific programming. In the process of testing the concept, we performed an experiment in control of an acoustic wave system.

The Remote Object Controller (ROC) has the capability of timing its own operations. The timing of a method call works as follows: A method call is made, through CORBA, to the ROC. The ROC starts a high-precision timer before processing the method call. The ROC then locates any remote objects needed for the call, and forwards the call to the appropriate object. The ROC waits for the method call to
return, takes care of any error handling, and then stops the timer. The elapsed time is accumulated to a buffer, which can be accessed through the CORBA interface.

The following is the result of a run for a 200 x 200 point grid, and 1000 time steps per optimization iteration. The simulation server was run in parallel on our cluster of four Intel/Linux machines. The optimization client was run on a separate Intel/Linux work-station. The initial wave field was generated separately by smoothing a field of uniformly-distributed random numbers.

dh211ipc.caam.rice.edu% time HCLClient.x init_p4_100x100.dat

===================== HCL(CG): Solve =====================
HCL.CG::Solve: iteration = 0 rel. error = 1
HCL.CG::Solve: iteration = 1 rel. error = 0.027905
HCL.CG::Solve: iteration = 2 rel. error = 0.0140198
HCL.CG::Solve: iteration = 3 rel. error = 0.00851566
HCL.CG_d::Solve: Solve SUCCEEDED - rnorm = 4.87712

================= HCL(CG)::Solve Summary =================
Residual norm: 4.87712
Total number of iterations: 3
Total number of user A*x calls: 4
Total time in CG iteration: 0.03
Total time in user A*x routine: 0.01

=================================

Server Timing Summary
Total time in server: 39.4646
Time in vector operations: 0.179262 0.454235%
Time in wave operator: 39.2649 99.4939%
Time in vector overhead: 0.0200762 0.0508714%
Time in wave operator overhead: 0.000405054 0.00102637%
Finished.
0.050u 0.010s 0:43.89 0.1% 0+0k 0+0io 316pf+0w

Note that the total wall-time was 43.89 seconds. The server consumed 39.46 seconds of this, or 89.9 percent, while the client accounted for 4.43 seconds, or 10.1 percent.

The largest portion of the total time was spent simulating the wave field, as we would expect. A smaller amount of time was consumed by the client. Was this time spent by the HCL algorithms doing useful work, or time spent making CORBA method calls?

To answer this question we performed another simple experiment. We made a client that simply called the linear and adjoint wave operators for several grid sizes and numbers of time steps. We timed a number of calls, with a fixed grid size and number of time steps, and compared this to the time the ROC reported as being taken by the server. The difference was the amount of time spent by the new client, which did very little work, and the time spent in the CORBA call. Figure 3.8 shows the average time used by CORBA as a percentage of the total time, averaged over several iterations. The total time spent by CORBA is very small in relation to the time spent by the simulator.

The results of the optimization problem where also interesting. Recall that the goal was to find a time-series of pressure values along the left-most boundary that would cancel the wave field resulting from non-zero initial conditions. The problem converged in three iterations, and gave the following results for a grid of size 400 × 400,
Figure 3.8 Percentage of total time consumed by CORBA method call

and 1000 time steps. Figure 3.9 is the initial data without cancellation, and figure 3.10 is with cancellation.

Figure 3.9 Initial $400 \times 400$ wave field sampled at left boundary for 1000 time-steps

3.5 Discussion

We explored the applicability of components by applying them to a simple control problem. We found that, in many ways, the code did benefit from being written in a component form.
Figure 3.10 Final wave field sampled at left boundary

Coding the simulation server was straightforward, if somewhat mechanical. We started with a working distributed-memory parallel wave simulator, and a serial wave simulator, and produced a CORBA interface for them. We now have a set of simulators that are easier to use than they were before. Using them in a code simply requires calling the CORBA interface to the servers. Since the servers implement the same interface, they are interchangeable. Issues of parallelism, and data structures are hidden behind the interface. The interface has made the simulators easier for others to use as well, since they only have to deal with a small set of methods to use the servers.

The Hilbert Class Library optimization clients where extremely easy to code. By using proxy objects to represent the remote objects, we were able to directly apply the class hierarchy of HCL. This allowed use to make a powerful optimization application with only a few lines of code. In general, we have developed a new means of adding complex data structures and operations to HCL, or to any other object-oriented library. This, in itself, is a very useful tool.

Our tests show that the run-time overhead of CORBA is acceptably small when the method calls are course-grained. Keeping the remote method calls course-grained
was easily accomplished. The steps in a single optimization iteration where sufficiently computationally costly that the time taken by the CORBA call to initiate them was insignificant.

CORBA is relatively simple to use, and provides functionality that greatly simplifies the connecting of components. We only used the most basic features of CORBA. Several other features might be beneficial to scientific coding.

It would be relatively simple to have other people make components that could be added to the system. They would not even need to have access to the existing components; they would only need a statement of the interface.

We should investigate other CORBA ORB's. We chose ORBit because it was readily available. Other ORB's provide functionality that ORBit does not, such as bindings to languages other than C, and a working name server. An ORB with a C++ binding would allow use to continue the use of C++ throughout the system.

Several observations where made in the process of developing the code for this project. We are not simply connecting object-oriented libraries as a whole, though this was the initial reason for pursuing this project. We are connecting the lowest level of the class hierarchy; we are connecting the concrete classes through a defined interface. The classes must be concrete, as we can only communicate concrete-types through the interface. The interface specifies connection-points, at the most concrete level, at which we can connect object-oriented libraries.

This is more than a complicated form of remote procedure calls (RPC), though, at the lowest level it is basically RPC. Conceptually, however, it is much more. We have a defined interface that allows us to easily add other components. We have a more sophisticated run-time linking system, thanks to CORBA. We also have the ability to easily hide programming language issues, parallelism, and hardware architecture
issues. CORBA also allows use to connect components on multiple machines without any extra work.

Making a good interface is critically important to the success of the component. The interface should ideally be designed with only the server component in mind, as the interface must specify exactly what services the component provides. To maximize reusability, the interface should be made general enough to state the concept of the components services. The interface should not be designed with a specific application in mind, but rather the interface should be designed to describe the services provided by a component that would provide the widest usability. But, of course, what services the component should provide is a larger question related to the entire application, and so the component can not be created in a vacuum.

However, does the interface really do a sufficient job of describing the services of the component, or do you still have to have some knowledge of the internal workings of the component? We feel that the interface does not always describe what the component does, especially if the component performs complex operations. There is also the issue of: is it possible to exactly specify every option necessary to tell the component what it should be doing? Perhaps a solution is to make sure that the
component provides it's services in small enough units that complex operations can be specified exactly. Although, this might make using the component very tedious.

We are making building-blocks of code, that can be reused in other applications. However, it could be argued that the only component in the current system is the simulation component. The optimization client is simply a code that makes use of the component. In many ways, the client is a script that controls the simulation.

We could turn the client-server relationship around, and make the optimization code the server. One could imagine a system in which a simulator contacts an optimization server for "help" with optimizing its problem. The optimization server could be made to service multiple simulation-problems simultaneously. However, we do not see a benefit from this inversion of the client-server relationship. The client-server relation simply specifies which portion controls the application, and which provides services. It seems more natural to think of the optimization code as controlling the simulation problem, and so it should be the client.

However, can we extend the component idea even further for this simple problem? We could make an optimization server, as described above, and connect it to our existing simulation server. An "agent" code would coordinate the interaction of the components. The agent would alternate between asking the simulation server to simulate a problem, and asking the optimization server to find an improved iterate. We have gained greater flexibility by making the optimization code a true component. Other optimization components can now be simple "plugged in". However, we still gain very little for this problem by taking control away from the optimization code. The Hilbert Class Library, on which the optimization client is built, provides sufficient flexibility in choice of optimization routines, without being a component.

Still, if we could extend the component concept sufficiently far, we could encapsulate all complex operations in components. We would have components for
optimization, components for simulation, components for linear algebra, and for any other operations we might want to perform. Then, writing an application would be as simple as scripting the interaction of the components. This would be especially simple if the component framework is well integrated into the operating system, then using a component would be as simple as a system call.

Adding other servers to the system would simply require that they implement the existing interface, which is fairly simple. The new simulators would not have to be wave simulators, either. The interface simply requires that they be able to provide an operator and its adjoint. The servers could also be written in most any language convenient, or could be adapted from existing codes.

3.6 Conclusion

We believe the component method can be used to structure scientific codes so that development effort can be more easily divided between a team of people, and the results can be reused in other scientific codes.

Components proved to be an efficient means of structuring code consisting of optimization and simulation elements. Separating the code into components allowed each portion to be developed and tested separately, which greatly simplified the design process.

Components did impose at least one unwanted restriction on this project. The parallel wave server had to be written as a master/slave code, rather than a code in the simpler SPMD style.

Creating the interface code, the body of the ROC, was rather mechanical, but we have reason to believe that this could be automated, and new CORBA tools have already done much to minimize the work done by the developer.
Whether the component method is seen as a new concept in software design, or simply another way to link otherwise incompatible software libraries, the method is a useful tool.
Chapter 4

High-Level Encoding of Finite-Difference Approximations

4.1 Introduction

Finite difference operations can be written in a very natural form, such as our example from chapter 1,

\[ D_t^+ D_t^- p = c^2 (D_x^+ D_x^- p + D_y^+ D_y^- p) = 0 \]

which is a finite-difference approximation to the acoustic wave equation. However, this equation can not be directly used as computer code; it would have to be expanded to a series of nested-loop operations, or function calls.

Some programming languages, such as C++, allow overloading of basic operators. This allows us to assign meaning to common notational elements for user-defined types. For example, assume we have a vector class \texttt{Vector}, for which we have overloaded the "+" operator to perform vector addition, and the "=" operator to perform assignment. Then we could write \( \mathbf{z} = \mathbf{x} + \mathbf{y} \), where \( \mathbf{z}, \mathbf{x}, \mathbf{y} \) are objects of type \texttt{Vector}.

We could, using operator overloading, code our finite difference approximation in a form very similar to the one above. However, this will generally result in poor performance, since overloaded operators generally return a temporary object as the result, which is passed by value. For example, adding two vectors would result in a temporary vector representing the sum, which would be discarded once the values where assigned to the target vector. This can be very time consuming for objects that contain large amounts of data, as memory must be allocated, filled by copying, and then deallocated. The result is an inefficient operation.
We can avoid creating the temporary objects if we defer the evaluation of the operation until such time as we can evaluate the expression as a whole. Ideally, we would like the compiler to do this for use.

Expression templates are an efficient means of deferring evaluation of the operations to avoid temporary objects [25]. Several software packages make use of expression templates. Blitz++ is a linear algebra library that "uses template techniques to achieve high performance" [1]. POOMA (Parallel Object-Oriented Methods and Applications), and the package it is built on, PETE (Portable Expression Template Engine) make use of template expressions for high performance scientific coding [16] [15].

Expression templates require the use of C++ templates, a feature of the language that some have been hesitant to use, since many compilers where slow to support them fully. However, we can explore the concept without using templates. We can approximate the use of templates while only using standard C++ references. The result is a system of deferred-evaluation operators that work conceptually similar to templated expressions. Unfortunately, the reference version is not as efficient in time as the templated version, but does have the advantage of simplicity.

4.2 Deferred Evaluation

Let's analyze our example operation on our vector class

\[
\begin{align*}
    z &= x + y; \\
    \text{What is happening here? Let's begin by first looking at a typical form of operator overloading in C++.
}
\end{align*}
\]

The "+" operator would have a definition of the form

\[
\begin{align*}
    \text{Vector Vector::operator+ ( Vector &y );}
\end{align*}
\]

In C++, the operation

\[
\begin{align*}
    x + y
\end{align*}
\]
would be equivalent to the method call

```cpp
x.operator+(y)
```

Note that the operator returns the vector by value. That is, a temporary vector is created inside the operator method, and then returned by copying. This can be very inefficient. The temporary vector must be allocated, and filled, and then will be discarded after the next operation. This would especially be inefficient if the vectors are large, or if several operations must be performed.

For example, if we have several vectors to sum

```cpp
z = a + b + c + d
```

each addition results in a temporary vector. We must find some way to return the result of the operation without creating temporary vectors, as the vector returned by value is the main cause of inefficiency.

What if we could instead return a place holder: an object that represents the operation, but can be more efficiently returned? We could return a small (in size) object that stores only the information needed to perform the operation at a later time. In this case, that would be a reference to each of the operands.

How would these place-holder objects interact with the vector operations? Note that conceptually the sum of two vectors is still a vector. This hints at a possible object-oriented approach.

We will implement a hierarchy of vector classes to represent the operations. The base class is a nearly abstract class that provides a method to “add” two vectors, and an abstract method to allow element-wise access to a vectors data. The class might look something like

```cpp
class VectorBase
{
  public:
```
// Abstract element-access method
virtual double get( int index ) const = 0;

// Over-loaded addition operator
VectorAdd operator+( const VectorBase &y ) const;
}

Note that the addition operator returns an object of type VectorAdd, which we will define shortly.

From this base class we derive (by inheritance) our concrete-data vector class. The concrete vector class must overload the assignment operator, which takes a reference to an object of type VectorBase. Values are assigned element-by-element by calling the argument's element-wise access method. The vector class would then look like class Vector : public VectorBase
{
  double *data;
  int len;

  public:
  Vector( int length );
  double get( int index ) const;

  // Over-loaded assignment operator
  Vector& operator=( const VectorBase &y );
};

We need a class to represent the addition operation. It too should be a child of the base class. We will make its element-wise access method inlined for simplicity.
class VectorAdd : public VectorBase
{
    VectorBase &x;
    VectorBase &y;

    public:
    VectorAdd( const VectorBase &x, const VectorBase &y );

    inline double get( int index ) const
    {
        return x.get(index) + y.get(index);  
    }
};

The VectorAdd class stores references to two vectors, which are the vectors to be added in the end. Note that the element access method simply calls the element access methods of the vectors referenced, and returns the sum.

It is important to note that the addition operator is defined in the base class, but the assignment operator is only defined in the concrete vector class. The addition operator allows any two vectors, of type Vector or VectorAdd, to be added, which allows us to chain addition operations. The assignment operation only allows us to assign values to a concrete vector, of type Vector. This makes sense, as assigning values requires actual memory space to which to assign. However, the argument of the assignment operator is of type VectorBase, so any of the concrete vector types can be assigned to the concrete vector.

Now, we can implement the addition operator for the base class

VectorAdd VectorBase::operator+( const VectorBase &y ) const
{
    return AddVector( this, y );
We are returning a vector by value, but it is only an object of the light-weight `VectorAdd` type, which is too small to cause a large efficiency penalty. Note that since the addition operator was implemented in the base class, it can operate on any binary combination of `VectorAdd` and concrete `Vector` classes.

We can also implement the assignment operator for the `Vector` class

```cpp
Vector& Vector::operator=( const VectorBase &y )
{
    for( int i = 0; i < len; ++i )
        data[i] = y.get(i);
    return this;
}
```

Now, parsing the statement

```
z = a + b + c + d
```

where `z`, `a`, `b`, `c`, `d` are of type `Vector`, would result in the following. First, the addition operator would be called for `c + d`, which would result in a temporary `VectorAdd` object. The temporary would be added to `b`, resulting in a second temporary of type `VectorAdd` (which would now encode `b + c + d`). Similarly, vector `a` would be added to the last temporary, resulting in a final `VectorAdd` object which encodes the entire operation.

A diagram should clarify this concept. The statement

```
z = a + b + c + d
```

will produce a structure of objects as in figure 4.1.

It is this tree structure that is passed to vector `z`'s assignment operator. Specifically, the object marked `temporary #3` (figure 4.1), which is the root of the parse tree, is passed to the assignment operation. The assignment operator iterates over each el-
4.3 Expression Templates

Evaluation of the parse tree in 4.1 requires a series of virtual method calls: each element accessed requires a virtual method call for each object in the structure. The method calls are virtual because they are defined in the base class, and bound polymorphically. This is a result of treating the partial results of an operation as a valid vector, and so making the objects representing the operation part of the class hierar-
chy. The cost in virtual method calls makes this a very expensive operation, though it may be faster than a non-deferred evaluation implementation.

We would prefer the method calls to be inlined. Inlining allows the compiler to optimize across function boundaries, resulting in a much more efficient code [17]. Templates are a feature of C++ that allow data types to be determined at compile time [23]. This gives us a means of binding the method calls at compile-time, rather than run-time. We can then discard the class hierarchy, and code the operations in a more generic form.

Making a flexible library of data containers and templated-expressions can be very complex, and in general the code can become very complicated. However, a simple example similar to the example above can be given. Our data container, the class Vector, needs little change. We must template the assignment operator, and add a templated addition operator.

class Vector
{
    double *data;
    int len;

public:
    Vector( int length );

    inline double get( int index ) const { return data[index]; }

    template<class V>
    Vector& operator=( const V y )
VectorAdd<Vector&, Vector&> operator+( const Vector& y );
VectorAdd<Vector&, VectorAdd> operator+( VectorAdd y );
}

Note that the templating of the assignment operator simply means that we can assign any object to the Vector that provides the necessary methods: the methods called in the body of the assignment operator. The definition of the assignment operator would be

template<class V>
Vector& Vector::operator=( const V y )
{
    for( int i = 0; i < len; ++i )
        data[i] = y.get( i );
}

The definitions of the addition operators would be

VectorAdd<Vector&, Vector&> Vector::operator+( const Vector& y )
{
    return VectorAdd<Vector&, Vector&>( this, y );
}

VectorAdd<Vector&, VectorAdd> Vector::operator+( VectorAdd y )
{
    return VectorAdd<Vector&, VectorAdd>( this, y );
}

The templated type V must provide a get() method, that takes an integer as an argument, and returns a double. This is the element-access method of the vector
classes we have made so far. Thus, the assignment method can assign, to itself, any object which provides an element-access method.

Note that there should be some form of error checking taking place in the assignment operator. We have omitted the error checking to keep the examples simple.

We can also re-write our addition-operation place holder class, since it no longer needs to be derived from a base class. Note that it too needs an addition operator, since we can add a vector to a VectorAdd object.

template<class V1, class V2>

class VectorAdd
{
  V1 &x;
  V2 &y;

  public:
    VectorAdd( const V1 &_x, const V2 &_y );

  inline double get( int index ) const
  {
    return x.get(index) + y.get(index);
  }

  VectorAdd<VectorAdd,Vector&> operator+( const Vector& y )
  {
    return VectorAdd<VectorAdd,Vector&>( *this, y );
  }

  VectorAdd<VectorAdd,VectorAdd> operator+( VectorAdd y )
  {
    return VectorAdd<VectorAdd,VectorAdd>( *this, y );
  }
};
Now, for our example operation \( z = a + b + c + d \) the following will occur when the assignment operator is finally called. The assignment operator for object \( z \) loops over each element in its vector, calling the element-access method of the object passed to it (the parse tree). For example, the assignment operator is called with a \textit{VectorAdd} object, which represents the top node in the expression parse tree. The assignment operator calls the \texttt{get()} method with the index \( i \). This call results in a call to each object referenced by the top \textit{VectorAdd} object. Each referenced object calls the element-access method of its referenced objects. This continues until the access method of a concrete container class is called, at which time actual values are returned. This is most easily thought of as a chain of recursive calls, where at each branch either actual values are returned, or the access method of the referenced objects is called, and the result is summed and returned.

Our hope for efficiency is that the total call chain to the access methods (for a single index) is made of inline method calls. Then, the methods can be optimized as a single expression at compile time.

Note that the types stored by \textit{VectorAdd} are templated separately. This is so that \textit{Vectors} are stored by reference, and \textit{VectorAdd}s are stored by value. Objects of type \textit{Vector} persist through the execution of the operation, so we can keep a reference to them. The objects of type \textit{VectorAdd}, however, are created as temporary objects which are return by value from addition operations, and so must be stored by value.

Note that each class requires two addition operators, for a total of four addition operators for the two classes. This is because we can add any combination of two objects, of type \textit{Vector}, or \textit{VectorAdd}. Each operator ensures the proper storage of the referenced objects in the resulting \textit{VectorAdd} object.

Extending the template expression to handle operations other than addition can cause problems for our simple example. Adding operations will cause the number of
operator methods to grow. In general, if we have \( n \) operator and container classes, then we must have \( n^2 \) operator methods: one for each combination of operands.

This can be prevented by adding another class: an expression class that behaves as a wrapper for operator classes. The expression class stores one operator class, such as `VectorAdd`, by value. Then, the expression class is used in place of the operator class in further processing. In this model, only the container, and the expression, classes must have overloaded operators. The result is that we must still write \( n^2 \) overloaded operators, but now \( n \) is only the number of container classes, plus the expression class.

An interesting point is that operations performed with template expressions are encoded as the type of the objects in the parse tree. For example, the operation \( a + b + c + d \), using the simple classes above, would result in an object of type `VectorAdd<Vector&, VectorAdd<Vector&, VectorAdd<Vector&, Vector&>> >)`.

This expression is equivalent to the parse tree in figure 4.1 encoded with templates.

Obviously, the encoded types can be come quite complicated. Adding the expression class to the system, as described above, would result in an even longer expression. The example expression is a rather simple one; as we add other operators, the expressions will become even more complicated. Other common practices, such as templating the type stored by the vector class (such as `float` or `double`), will add to the complexity of the expression type.

Ideally, the actual expression type is never seen by the user. However, compiler and debugger error messages use the full type definition, which results in messages which are very difficult to read, and commonly fill several pages of text per error.
4.4 Results

We tested our template-expression system with the following problems. The first is a simple test of the templated-expressions without any shift or difference operators. The expression used was

\[ A = 3.4 \times (-B + 5.6) + \text{abs}(C + 1.2 \times D); \]

where A, B, C, D were two-dimensional grids of doubles. A reference hand-coded version was made for comparison. The result of 1000 iterations is given in Table 4.1.

The second code tested the difference operators needed to code finite differences. The expression was

\[ A = \text{Dxp}(\text{Dxn}(B)) + \text{Dyp}(\text{Dyn}(B)); \]

The result of 1000 iterations is given in Table 4.2.

Note that in both cases the expression template code is slower than the hand-coded reference, but the time difference for the difference expression is larger than the difference for the non-difference expression. In general, we found any expression with a difference operator to be noticeably slower than the reference code.

We coded the expression-template difference operator as follows. A difference operator can be made by subtracting two shift operators:

\[ \text{Dxp}(A) = A - \text{shift}_x(A,1) \]

<table>
<thead>
<tr>
<th>GridSize</th>
<th>ExpressionTemplates</th>
<th>ReferenceCode</th>
</tr>
</thead>
<tbody>
<tr>
<td>128 x 128</td>
<td>2.67</td>
<td>2.43</td>
</tr>
<tr>
<td>256 x 256</td>
<td>13.53</td>
<td>12.59</td>
</tr>
<tr>
<td>512 x 256</td>
<td>54.18</td>
<td>50.74</td>
</tr>
</tbody>
</table>

Table 4.1 Run times in seconds for first expression (without difference operators)
<table>
<thead>
<tr>
<th>GridSize</th>
<th>ExpressionTemplates</th>
<th>ReferenceCode</th>
</tr>
</thead>
<tbody>
<tr>
<td>128 × 128</td>
<td>2.09</td>
<td>1.21</td>
</tr>
<tr>
<td>256 × 256</td>
<td>12.35</td>
<td>9.74</td>
</tr>
<tr>
<td>512 × 256</td>
<td>50.99</td>
<td>41.81</td>
</tr>
</tbody>
</table>

**Table 4.2** Run times in seconds for second expression (with difference operators)

where \( \text{Dxp} \) is the difference operator in the positive x direction, \( \text{shift}_x \) is a shift operator, and \( A \) is a grid. The shift operator does not actually shift the data stored in \( A \). Recall that template expressions eventually result in a series of indexed element accesses for each node in the parse tree. The shift operator simply offsets the index by the appropriate amount before forwarding the element-access call. This should result in an efficient difference operator.

When we found that the difference operator was not as efficient as hoped, we inspected the assembly code produced by the compiler. We found a large amount of integer arithmetic in the inner loop of the operation. Inspection of the hand-coded reference operator showed much fewer integer operations. The floating-point operations were basically identical. We believe the extra integer operations where the result of the index arithmetic of the shift operators not being optimized away, which resulted in slower run-times.

### 4.5 Conclusion

Expression templates can be applied to finite-difference codes. The resulting system allows the finite-difference approximation to be stated in a form that is very familiar; the common operations needed to express finite differences, such as scale, addition, and difference operators, can be efficiently coded at a high level.
Unfortunately, our experiments proved to be slower than expected. The difference operator, which is necessary to the finite-difference approximation, seemed to be the least efficient of the coded operators. The other operators coded, which included addition, negation, scaling, and absolute value, where nearly as efficient as a hand-optimized reference code. However, we had hoped that the expression-template operators would be more efficient than the hand-coded version, as the compiler is given more opportunity to optimize the expression template code.

We assume that the lack of efficiency of the difference operator stems from the index operator. The index operator, for the difference operator, produced the index for a shifted element in the global data structure. The complexity of the index function left much room for a less-than-optimal implementation. Examination of the assembly code produced by the compiler showed that the expression-template version was executing a larger amount of integer arithmetic than the hand-coded version. We assume this was the result of the index function, which is where the majority of the integer arithmetic is found in each iteration. It might also be helpful determine if other C++ compilers could better optimize the code.

While our expression template system for finite differences needs further development, we believe that the use of expression templates for finite differences could reduce the amount of time spent by the user writing the code.
Chapter 5

Conclusion

We studied a method of overlapping the domains of a parallel finite difference approximation to the wave equation, with the intention of producing a faster running code when run on a cluster of workstations. We actually found that, for a realistic set of assumptions on processor and communication speed, domain overlapping produces a slower code than a non-overlapped domain. In the process of studying the method, we developed a way to analyze parallel algorithms, without actually writing the code. These model where then used to study an alternate version of the domain-overlapped finite-difference wave equation algorithm.

We developed a system for connecting simulation and optimization codes. The system is based on the method of dividing larger applications into distinct, self-contained blocks called components. Each component provides a set of services to the rest of the code; the services are requested by messaging.

Dividing the boundary-control problem into components produced a simpler, and more flexible, system. The client was able to control a simulation server, whether the server was a serial or parallel application, as the issues of parallelism where completely hidden in the simulation component. We also successfully hide the Hilbert Class Libraries class hierarchy from the server; the server was able to work with HCL without having any direct knowledge of its defined data types.

The application was built with off-the-self packages, such as HCL, and CORBA. HCL greatly simplified coding the optimization portion of the application. Using CORBA required that certain concessions be made, such as requiring the parallel simulator be written as a master/slave application. However, while CORBA may not
have fit the problem exactly, the benefits of using it where great. We were able to completely decouple the optimization and simulation codes. This allowed us to develop, and test, the simulator server and optimization client separately.

Extending the system to include other problems, or simply reusing any portion of the code in another application, is simplified by the component approach. For example, we might add a visualization package by wrapping it as a CORBA enabled component, and adding it to the system.

Finally, we studied a novel method for accelerating the development finite difference codes. This method, expression templates, allows a user to describe mathematical operations on grids at a very high level. Expression templates have been applied to more general problems, such as linear algebra packages; we wished to apply them to finite differences.

We developed a set of C++ grid, and related, classes using expression templates. However, our code did not perform as we had hoped. The performance of the templated expressions, without shift or difference operators was acceptable, though slightly slower than hand-optimized code. Adding difference operators slowed the code significantly. However, difference operators are necessary for the coding of finite difference approximations.

We are not sure why the difference operators are not as efficient as hoped; more time would have to be spent to pin-point the problem. Our investigation pointed to the index operator, the method that returns an element given an index. The index operator for the difference operator is rather complex, as it must be able to efficiently produce index-shifted elements in a multi-dimensional grid. Simplification of the index function would be a logical first step, but this would most likely mean that the grids would have to be restricted to a single number of dimensions, for example two
dimensions. It is also possible that other compilers would be able to better optimize away the index method, and so produce more efficient code.

A further problem is that expression templates produce simpler end-user code, but at the cost of rather complicated developer code. The developer code, which is hidden inside the class definitions, can be extremely complicated, and errors in this code will often produce compiler errors that are almost unreadable.

However, we still feel that expression templates could be used effectively for finite-difference codes, given more development.
Bibliography


