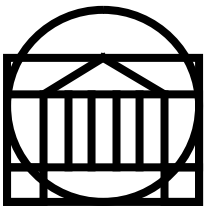
A Parallel Object-Oriented Framework for Stencil Algorithms

John F. Karpovich
Matthew Judd
W. Timothy Strayer
Andrew S. Grimshaw

January 27, 1993

Appeared in *Proceedings of the Second International Symposium on High Performance Distributed Computing,* pp. 34-41, July 1993.
Also available as University of Virginia, Department of Computer Science Technical Report CS-93-13 via the web @ ftp://ftp.cs.virginia.edu/pub/techreports/README.html.

Work partially sponsored by NSF and DOE.



DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF VIRGINIA
THORNTON HALL
CHARLOTTESVILLE, VIRGINIA 22903-2442
(804) 982-2200 FAX: (804) 982-2214

A Parallel Object-Oriented Framework for Stencil Algorithms

John F. Karpovich, Matthew Judd, W. Timothy Strayer, and Andrew S. Grimshaw

Department of Computer Science, University of Virginia

{jfk3w | mrj2p | wts4x | grimshaw} @virginia.edu

Abstract

We present an object-oriented framework for constructing parallel implementations of stencil algorithms. This framework simplifies the development process by encapsulating the common aspects of stencil algorithms in a base stencil class so that application-specific derived classes can be easily defined via inheritance and overloading. In addition, the stencil base class contains mechanisms for parallel execution. The result is a high-performance, parallel, application-specific stencil class. We present the design rationale for the base class and illustrate the derivation process by defining two sub-classes, an image convolution class and a PDE solver. The classes have been implemented in Mentat, an object-oriented parallel programming system that is available on a variety of platforms. Performance results are given for a network of Sun SPARCstation IPCs[1].

1. Introduction

There is a class of applications whose implementation is realized using *stencil algorithms*. In a stencil algorithm, the value of a multi-dimensional array element is a function of the values of other elements within a neighborhood of that element. Exactly which neighboring elements are used is defined by a *stencil*. For example, the four connected neighbors are used to find the next value of an element in an iterative partial differential equation (PDE) solver.

Stencil algorithms are generally computationally expensive and, as a consequence, they are good candidates for parallel execution. Indeed, parallel implementations of stencil algorithms abound. Furthermore, stencil algorithms are structurally very similar.

Stencil algorithms have been ported widely to parallel architectures. Each port represents a considerable investment of time and energy, much of which is spent implementing the same code structures that have already been done by others. This wasted effort is particularly galling when one considers the basic, underlying similarity of these algorithms.

The object-oriented programming paradigm offers an elegant solution to this problem: design a base class that encapsulates the essence of stencil algorithm behavior, implement that class for parallel execution, then derive application-specific stencil classes that implement the details that distinguish the application. This approach allows the programmer to concentrate on the specifics of the problem at hand, and to leverage off the work of others. Ideally, this should result in both reduced implementation time and better performing parallel code (because parallel programming experts can implement the base class). Additionally, if the underlying base class is portable to a wide variety of architectures, then the derived classes should be as well.

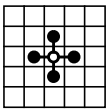
This paper reports on our work on a stencil base class with the properties described above. We have implemented a base class, `Stenciler`, and two application-specific derived classes in the Mentat Programming Language (MPL). Mentat [1-3] is an object-oriented parallel programming system that has been implemented on a variety of platforms including networks of workstations. The implementation of the stencil base class has been a success: the stencil class provides the basis for application-specific stencil algorithms, and the parallel implementation of the base class member functions afford the application good performance.

We present the stencil problem in more detail, concentrating on the data decomposition and data communication implications. We then provide a high-level overview of the implementation of the base class. We next present the implementations of two different derived classes, an image convolution class and a PDE solver, followed by the performance of both implementations on a network of Sun IPC workstations. We conclude with a discussion of our future plans.

2. Stencil Algorithms

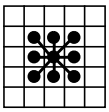
Stencil algorithms are used in a wide range of scientific applications, such as image convolution, solving partial

[1.] This work is partially supported by NSF grants ASC-9201822 and CDA-8922545-01, and DOE grant DE-F605-88ER25063.



$$F_{i,j} = \frac{I_{i-1,j} + I_{i+1,j} + I_{i,j-1} + I_{i,j+1}}{4}$$

(*a*) 2D 3×3 NEWS stencil and sample function



$$F_{i,j} = \sum_{k=0}^{2} \sum_{l=0}^{2} (I_{i+k-1,j+l-1} \times M_{k,l})$$

(*b*) 2D 3×3 eight-connected stencil and sample function

F - final matrix, I - input matrix, M - convolution mask

0	1	0
1	0	1
0	1	0

(c) 2D NEWS Stencil

1	2	1
2	4	2
1	2	1

(d) 2D 3×3 Gaussian filter

Figure 1 - Typical 2 dimensional stencils.

differential equations (PDE's), and correlation algorithms. Stencil algorithms are a class of algorithms that have several features in common: (1) the input data set is an array of arbitrary dimension and size, (2) there is a *stencil* that defines a local neighborhood around a data point, (3) some function is applied to the neighborhood of points that are "covered" when the stencil is centered on a particular point, and (4) this function is applied to all points in the data set to obtain a new data set.

Figure 1*a* shows a two dimensional 3×3 stencil that indicates that each output value will depend *only* on the "north," "east," "west," and "south" (called NEWS) neighboring points of the corresponding point in the input array. The associated function is an example of a stencil function that uses NEWS neighbors. Figure 1*c* shows the representation of a NEWS stencil in our implementation.

Figure 1*b* shows the stencil pattern needed for a 2D image convolution. In two dimensional image convolution, a small matrix, called a *mask*, is applied to the input array of image data. One such mask is shown in Figure 1*d*. Each point in the result is calculated by multiplying the value of each point in the mask by the appropriate neighbor of the corresponding point in the input image and summing. The equation in Figure 1*b* shows how to calculate the value of the filtered image at point (i, j) when using a 3×3 mask.

2.1. Parallel Solution

The fact that the same function is applied to all points in the input data set in any order and the calculation of each output point uses a regular pattern of neighborhood points around the corresponding input point makes parallelizing stencil algorithms a straight-forward process. Since all calculations logically occur in any order, there is no data-dependence between the output data points, so the problem

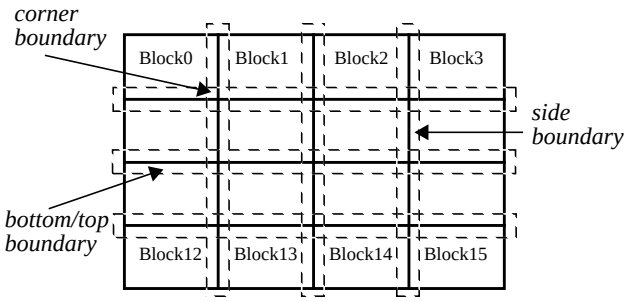


Figure 2 - A 4x4 rectangular decomposition

may be decomposed into several smaller pieces, each computed in parallel. The stencil defines the region of the input data set needed to calculate a piece of the output data set and, as a consequence, also defines the pattern of communication between the decomposed pieces. As shown in Figure 2, if the 2D image convolution problem is decomposed into a 4×4 matrix of rectangular "blocks," each of the 16 processors needs its block of input data points as well as data points from the boundary regions of processors above, below, and to the sides of the block. Each piece must therefore exchange data with its eight connected neighbors. In a parallel implementation this implies communication between worker objects.

Different stencils require different communications patterns. Stencil algorithms that don't require data from a neighbor eliminate the need to communicate with that neighbor. The problem decomposition pattern also effects the communication pattern needed. In a 2D problem there are 3 common decompositions: groups of rows, groups of columns, or rectangular blocks. For a given number of pieces, decomposing into rows or columns requires fewer communications, but also requires more overall data volume to be exchanged than a rectangular decomposition. This is a classic trade-off and determining which decomposition is best depends on the specific communications costs of each parallel system.

2.2. Parallel Object-Oriented Solution

The objective of all object-oriented solutions is to provide a framework that exploits the commonality among stencil algorithms and provides an environment for developing efficient parallel code to take advantage of the inherent data parallelism. The object-oriented programming paradigm, through object inheritance, supports exploiting the common attributes of objects while allowing the user to redefine or augment specific details. Our solution is the creation of a 2D parallel stencil class using the object-oriented parallel programming system Mentat.

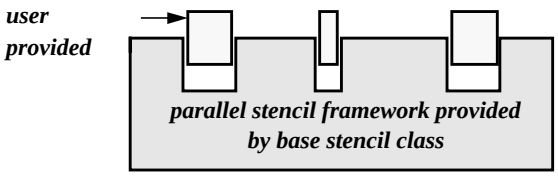


Figure 3 - Object-Oriented Stenciler Framework

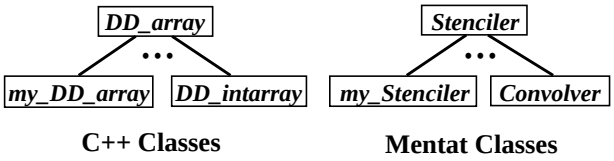


Figure 4 - Class Hierarchies

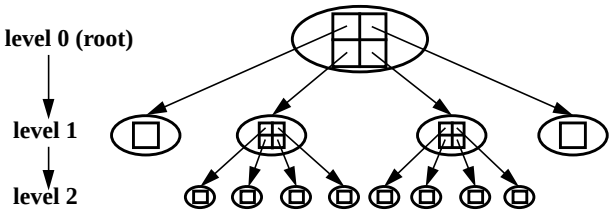


Figure 5 - Sample tree of Stenciler instances

We have defined a base stencil class that is designed to manage those areas that are common to all stencil algorithms while providing a framework for the user to create derived classes that can be tailored to specific applications (Figure 3). The base class contains built-in member functions to perform common tasks, such as managing data communication between pieces. The base class also contains well-commented stubs for member functions that the user must define, such as the stencil function. This approach minimizes the effort needed to create new stencil applications through reuse of common code while supporting flexibility in creating parallel stencil applications.

Our approach is not unique. In fact, there has recently been a movement towards using the object-oriented paradigm in traditionally FORTRAN-dominated scientific computation and numeric analysis applications. An annual conference has been organized for this growing community and the first one was held in April 1993 [4]. This approach has also been used in developing parallel gas combustion code [5].

3. Implementation Sketch

Our implementation consists of two class hierarchies, a C++ *DD_array* class hierarchy and a Mentat *Stenciler* hierarchy (Figure 4). Mentat classes are very similar to C++ classes [6] except that member functions of instances of Mentat classes may be executed in parallel. The Mentat system manages the communication and synchronization of Mentat object member function invocations, exploiting both data and functional parallelism.

DD_array Class Hierarchy

Stencil definitions and input and output data sets of 2D stencil applications are simply 2D arrays. In order to facilitate using arrays in the Mentat environment, a hierarchy of two dimensional array classes has been developed (the 2D array classes should really be a polymorphic class, but we do not currently have a compiler that supports C++ templates). The base class, `DD_array`, defines commonly used functions including creating arrays, extracting or overlaying sub-arrays, row and point access, etc. This base class has been extended through the creation of derived sub-classes for most of the primitive C data types, including `char`, `int`, `float`, and `double`. Two features of the `DD_array` class hierarchy are important: (1) memory space for the matrix is contiguous, and (2) the hierarchy can easily be extended to include a richer set of base data types, including classes and structures, such as a `DD_complexarray` class. The contiguous allocation of memory allows Mentat to pass `DD_array` data between process objects easily.

Stenciler Class Hierarchy

The `Stenciler` class provides the framework for creating 2D stencil applications. The user creates a new class derived from `Stenciler`. This derived class inherits all of the member functions of the base class, so instances of this new class have all of the built-in common functions provided with the `Stenciler` class. The user then supplies the application-specific code by overloading certain virtual member functions.

An instance of a `Stenciler` or derived class is designed to handle one piece of the total array. Each `Stenciler` instance can create additional workers to split the work-load into smaller pieces. These pieces, in turn, may be further divided, creating a general tree structure of pieces as shown in Figure 5. Each new level of the tree has a “contained-in” relationship to the previous higher level. The pieces at leaves of this tree structure are the workers who perform the stencil function. The interior instances are managers for the workers below them; the managers distribute and synchronize the work of their sub-piece and collect the results. This hierarchical tree structure of processes is a powerful and flexible tool for decomposing a stencil problem, especially when running on different hardware platforms.

```
persistent mentat class Stenciler {
  public:
  // ********** BUILT-IN FUNCTIONS **********
    int addStencil(stencil *sten)
    int doStencil();
    int getNumRows();
    int getNumCols();
    MATRIX_TYPE *getRegion(int ulr, int ulc, int lrr, int lrc);
    void init();
    int putRegion(int ulr, int ulc, MATRIX_TYPE *matrix);
    int setDest(string *dNm); // set destination file name
    int setGoal(float convGoal); // set convergence test goal
    int setIterations(int numIterations);// set number of iterations
    int setPieces(int xPieces); // set number of worker pieces
    int setRowsAndCols(int rows, int cols); // set size of data set
    int setRowPieces(int rowPieces); // set number of vertical pieces
    int setSource(string *sNm); // set source file name
    int setWindow(int w_ulrow, int w_ulCol,
                  int w_lrRow, int w_lrCol);
    float checkConvergence();
    int checkConvergence();
  // ********** USER DEFINED FUNCTIONS**********
    float checkConvergencePiece();
    int doStencilPiece(stencil *sten);
    int getMatrixPiece();
    stencil *getNextStencil();
    int prepareDest();
    int putMatrixPiece(); };
```

Figure 6 - Stenciler class interface

3.1. User-Defined Functions

The `Stenciler` class contains two kinds of member functions: built-in functions to handle common tasks, and user-defined functions for application-specific code (Figure 6 shows the `Stenciler` class interface). Stencil applications are tailored by overloading several provided function stubs with user-defined functions. The `doStencilPiece()`, `getNextStencil()` and `checkConvergencePiece()` functions are the heart of the stencil application, defining the work to be done for each data point and controlling which stencil to apply and when processing is complete. In addition to these functions, three user-defined functions, `getMatrixPiece()`, `putMatrixPiece()` and `prepareDest()`, are needed to handle application-specific file I/O operations.

Stencil Function

The `doStencilPiece()` function (Figure 7) defines the stencil function. This is where the actual work of calculating each worker's piece of the result is accomplished. For example, in image convolution this function is a 4-nested loop for performing the multiply and sum necessary to calculate each output value. The stub provided for this function contains information for how a typical stencil function may be coded. Often, the user only needs to copy the stub and provide a small amount of additional code to implement the stencil function.

```
int Convolver::doStencilPiece(stencil* sten) {
// variable declarations (omitted)

// The following functionality is IN THE STUB (omitted here)
// - calc rows and cols in current stencil (stenRows, stenCols)
// - calc coords of this piece's working window (ulr, ulc, lrr, lrc)

// calculate divisor - USER PROVIDED
  stensum = 0;
  for(i=0; i < stenRows; i++)
     for (j=0; stenCols; j++)
        if ((*sten)[i][j] < 0) stenSum -= (*sten)[i][j];
        else stenSum += (*sten)[i][j];
  if (stenSum != 0) divisor = stenSum;
  else divisor = 1;

// outer two loops PROVIDED IN STUB
  for (i=ulr; i <= lrr; i++)
     for (j=ulc; j <= lrc; j++) {
        // begin USER-DEFINED code
        (*destArray)[i][j] = 0;
        for (k=0; k < stenRows; k++)
           for (l=0; l < stenCols; l++)
           (*destArray)[i][j] += (*sten)[k][l] *
              (*srcArray)[i + k - stenRows/2][j + l - stenCols/2];
        (*destArray)[i][j] /= divisor;
        // end USER-DEFINED code
     }
}
```

Figure 7 - Convolver stencil function

Control Functions

Stencil applications are often iterative or, in the case of image convolution, several stencils may be applied in succession. To control the application of the stencil function, the user-defined `getNextStencil()` function (Figure 8) is called before each iteration to determine which stencil to apply next. In the case of successive image convolutions, this function simply returns the next stencil in the stencil list. For the PDE problem, `getNextStencil()` determines whether the computation has converged and, if not, continues applying a NEWS stencil.

Iterative stencil algorithms often require a calculation to determine when the convergence criteria has been met. The overloaded `checkConvergencePiece()` function (Figure 9) is defined when necessary to calculate some convergence data for each worker's piece. This data is collected by the built-in `checkConvergence()` function and can be used to determine the next stencil to apply and when the algorithm is completed.

File I/O Functions

Since file formats differ from application to application, the user must define how to read and write the data sets. The `getMatrixPiece()` and `putMatrixPiece()` functions must be defined to read in/write out the data of a worker piece. These functions are called respectively at the beginning and end of the `doStencil()` function (discussed below) for each leaf worker in the `Stenciler`

```
stencil *PDE_Solver::getNextStencil() {
// currentStencil points to the next stencil in the stencil list.
// convVal is the value set by the setGoal() function.
// currIter is the current iteration number - declared and set to 0

// local variable declarations - PROVIDED IN STUB
  stencil* next;
// local variable declarations - USER-DEFINED
  float testVal; // convergence value

  // begin USER-PROVIDED code
  testVal = checkConvergence();
  if (currIter == 0) next = currentStencil->st;
  else
     if (testVal <= convVal) next = NULL; // we're done
     else next = currentStencil->st; // keep plugging away
  currIter++;
  // end USER-PROVIDED code
  return(next); }
```

Figure 8 - PDE_Solver getNextStencil()

worker tree. The user may also redefine the `prepareDest()` function to set up the destination file by, for example, copying the header of the input file to the output file.

3.2. Built-in Stenciler Member Functions

The built-in functions are designed to perform certain tasks without the user writing any new code. Most of these functions allow the user to tailor a `Stenciler` object by defining such attributes as the number of sub-pieces, the shape of the decomposition, the names of the source and destination files, if applicable, the stencil(s) to be used, etc. Other built-in functions perform common tasks, for example, retrieving or overlaying a region of the array. These functions can be called from the user-defined code. Once the user-defined code has been provided and the `Stenciler` object has been initialized, the `doStencil()` function is called to perform the stencil operation.

Set-up Functions

The functions with the `set` prefix allow the user to provide information about the current application to a `Stenciler` object, and the functions with the `get` prefix allow the user to retrieve current values. The `Stenciler` implementation contains the notion of a *working window* within the data set where the stencil function is applied. Values outside of this window will not be changed, but will be used as input to the calculation of neighboring values that are within the working window. The window is set up by specifying the upper left and lower right corners in the `setWindow()` function.

To establish the communication pattern for the application, the stencil(s) must be defined. In our implementation, stencils are represented by small

```
float PDE_Solver::checkConvergencePiece() {
// local variable declarations - USER-DEFINED
  float total; // convergence value to be returned
  float temp;
  int i,j;

  // USER-PROVIDED code
  total = 0.0;
  if (prevArray == NULL) // i.e. if first iteration
     prevArray = new MATRIX_TYPE(myRows+2,myCols+2);
  else {
     // calc working window (ulr, ulc, lrr, lrc) - omitted
     // calc convergence test value for my piece
     for (i=ulr; i <= lrr; i++)
        for(j=ulc; j <= lrc; j++) {
           temp = (*srcArray)[i][j] - (*prevArray)[i][j];
           total += temp * temp;
        }
  }
  // set prevArray to current solution
  prevArray->overlay(ulr,ulc,srcArray->extract(ulr,ulc,lrr,lrc));
  return(total); }
```

Figure 9 - PDE_Solver checkConvergencePiece()

`DD_array`s. The non-zero values in the array indicate that a particular neighbor is needed in the calculation. The location of non-zero points and the size of the stencil determine the communication pattern and volume of data communicated. Stencils are kept in a list so that a series of stencils can be applied in succession. New stencils are added to this list via the `addStencil()` function.

DoStencil Function

Once a `Stenciler` object is created and initialized, the `doStencil()` function is called to execute the stencil. This function does the following:

(1) creates and initializes all additional worker objects needed;

(2) calls each worker to read its portion of the input array (via `getMatrixPiece()`);

(3) while there is more work to be done (via `getNextStencil()`), loops performing the following tasks:

  (a) exchanges necessary boundary data;

  (b) executes the stencil function (via `doStencilPiece()`);

(4) prepares the destination file and calls each worker to write its piece (via `prepareDest()` and `putMatrixPiece()`);

It is important to note that the user need only create the root-level `Stenciler` object and `doStencil()` does the rest. Also, the user does not need to modify this function, the user need only provide several user-defined functions described above in Section 3.1.

Other Functions

The `checkConvergence()` function is provided to control calling each leaf worker's user-defined

checkConvergencePiece() function and collecting the results. When called, this function determines whether the worker is a leaf node or an interior node. If it is an interior node, it calls checkConvergence() for each of its immediate subordinates, aggregates the partial test values, and sends the result to the next higher level in the worker tree. If the worker is a leaf, then checkConvergencePiece() is called to calculate the piece's partial result. The goal of this function is to abstract out the details of the worker hierarchy whenever possible.

The final two built-in functions, getRegion() and putRegion(), are provided to retrieve or overlay a region of the overall array. These functions are useful, for example, in checking convergence criteria.

4. Sample Stencil Implementations

To illustrate the use of the stencil framework we describe our experience with two sample implementations: an image convolver and a PDE solver using Jacobi iteration.

Image Convolution

Image convolution is a common application in digital image processing and computer vision [7]. In two dimensional image convolution, a small 2D stencil, also called a *filter* or *mask*, defines a region surrounding each picture element (pixel) whose values will be used in calculating the corresponding point in the convolved image. Each element of the filter is multiplied by the corresponding neighbor of the current pixel, and the results are summed and normalized. Figure 1*b* shows the stencil function for a 3×3 mask and 1*d* shows a common smoothing filter.

To implement the convolution application, the following steps were necessary:

(1) define the type of the input and output data sets to be DD_chararray;

(2) create a Convolver class derived from Stenciler;

(3) overload doStencilPiece() to define the stencil function;

(4) overload the control function getNextStencil();

(5) redefine the file I/O functions to input and output the matrix and prepare the destination file;

(6) create a main program to create, initialize and execute an instance of the Convolver class.

Creating the Convolver class was straight-forward; in fact, there were no additional variables or functions needed. The stencil function for convolution fits the framework provided in the stub for doStencilPiece(). The stub provides code to calculate the bounds of the working window and to loop through each point in the worker's piece. The only new code needed was the inner doubly nested loop to multiply and accumulate the value of a point and the code to normalize the result. An abbreviated version of the overloaded doStencilPiece() function is shown in Figure 7. Note that the code shown is used to clearly illustrate the use of the function and that standard C optimization techniques were used in the final code. The getNextStencil() function simply chains through the list of stencils supplied during initialization, returning each in turn. Therefore, a single Convolver instance can apply a series of filters to an image in succession. The getMatrixPiece() function was redefined to read from the input file the block of data "owned" by each leaf worker instance. Similarly, the putMatrixPiece() function was redefined to write to file the result of each leaf worker instance. Finally, the prepareDest() function was overloaded to prepare the header of the output file.

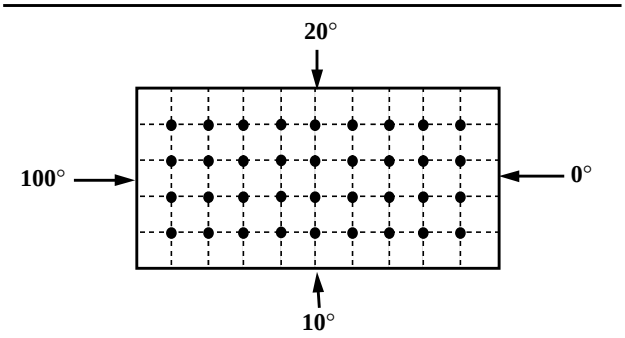


Figure 10 - Grid approximation for heated plate

The main program has the following structure: (1) declare, create and initialize an instance of the Mentat class Convolver, (2) tailor the instance by setting the source and destination file names, the number of worker pieces, the number of rows and columns in the data set, the number of row pieces (optional), and the list of convolution stencils (filters) to apply, (3) call doStencil() to execute the stencil algorithm, and (4) destroy the Mentat object.

PDE Solver Using Jacobi Iteration

Another common and important class of stencil algorithms are iterative methods. Jacobi iteration is a method for solving certain systems of equations of the form $A\grave{x} = \grave{b}$, where A is a matrix of coefficients, $\grave{x}$ is a vector of variables, and $\grave{b}$ is a vector of constants. The general procedure for using Jacobi iteration is to first guess the solution for all variables, and then to iteratively refine the solution until the difference between successive answers is below some pre-determined threshold [8].

The specific application of Jacobi iteration implemented and discussed here is the "heated plate" problem. The heated plate problem consists of a plate or sheet of material that has constant temperatures applied around the boundaries, and the goal is to determine the steady-state temperatures in the interior of the plate (Figure 10). The

temperature in the interior region is approximated by dividing the plate into a regular 2D grid pattern and solving for each of the grid points. The values at each point are approximated by the average of the values in the NEWS neighboring points. This transforms the problem into a system of linear equations which can be solved using Jacobi iteration. The form of the stencil function needed for Jacobi iteration is shown in Figure 1*a*.

To implement the PDE solver application, the following steps were necessary:

(1) define the type of the input and output data sets to be `DD_floatarray`;
(2) create a `PDE_Solver` class derived from `Stenciler`;
(3) overload `doStencilPiece()` to define the stencil function;
(4) overload `checkConvergencePiece()` to calculate the value of the convergence test for each leaf worker's piece;
(5) overload the control function `getNextStencil()`;
(6) redefine the file I/O functions to input and output the matrix and prepare the destination file;
(7) create a main program to create, initialize and execute an instance of the `PDE_Solver` class.

Creating the `PDE_Solver` class was slightly different from the `Convolver` example. The difference is that a new variable, `prevArray`, was added to store the source array of the previous iteration. This value is used in `checkConvergencePiece()` to determine whether the solution has converged. The stencil function for this problem is very simple and uses the framework provided in the `doStencilPiece()` stub with few modifications. The only user-defined code in `doStencilPiece()` is to calculate the average of the 4-connected neighbors for each matrix point.

The control structure of the PDE_Solver requires both an overloaded `getNextStencil()` function and an overloaded `checkConvergencePiece()` function. The `getNextStencil()` function simply calls the provided `checkConvergence()` function using the result to test if the iteration should continue (Figure 8). As discussed in Section 3.1, the `checkConvergence()` function handles calling `checkConvergencePiece()` for each leaf worker and collects and aggregates the results. The `checkConvergencePiece()` function was overloaded to calculate the convergence value of each matrix piece (Figure 9). The test value needed for the PDE solver problem is the sum of the squares of the difference between each point in the current and previous solutions.

The file I/O functions, `getMatrixPiece()`, `putMatrixPiece()`, and `prepareDest()`, were redefined exactly as in the `Convolver` example and the main program follows the same general outline with a few additions. Since the PDE problem defines constant temperature values around the matrix borders, the working window had to be set to include all of the matrix *except* the border region. This prevented the border from being changed during the stencil calculations. The convergence goal was set to an appropriate value to stop the iteration when the solution was within acceptable limits.

# Pieces	Convolver		PDE_Solver	
	Best Execution Time (min:secs)	Speedup	Best Execution Time (min:secs)	Speedup
1	49:33	N/A	43:39	N/A
2	24:33	2.0	23:01	1.9
4	12:37	4.0	11:37	3.8
6	8:47	5.6	7:53	5.5
8	7:07	7.0	7:08	6.1
10	5:53	8.4	6:47	6.4
12	5:08	9.7	6:23	6.8
14	4:57	10.0	6:10	7.1

Table 1: Performance Results

5. Performance

Our goal was to create a framework for stencil algorithms where the user can easily produce parallel code. The reason for wanting parallel code is, of course, speed. In order for our approach to be successful, the framework must provide a user with a program that is fast and exploits the inherent data parallelism of stencil applications. To test our performance we created two versions of each class, a sequential version written strictly in C++, and a parallel version written in C++/MPL. We executed the sequential versions on a Sun SPARCstation IPC and recorded the best of the wall-clock execution times. Similarly, we ran the parallel versions on a network of 16 IPCs connected via ethernet. The parallel versions were executed decomposing the problem into from two to fourteen row pieces. Each decomposition was run several times and the best time for each decomposition was recorded.

For the `Convolver` tests, both the sequential and parallel versions were executed using identical problems: a 2000×2000 8-bit grey scale image convolved with three successive 9×9 filters. The `PDE_Solver` problem used a 1024×1024 grid of floating point numbers to estimate the interior temperatures of the heated plate problem. Table 1 shows the raw best execution times and Figure 11 shows the speedup curve for the problems.

These results show good performance for the parallel implementation. Both speedup curves follow a classic pattern with nearly linear speedup at low numbers of

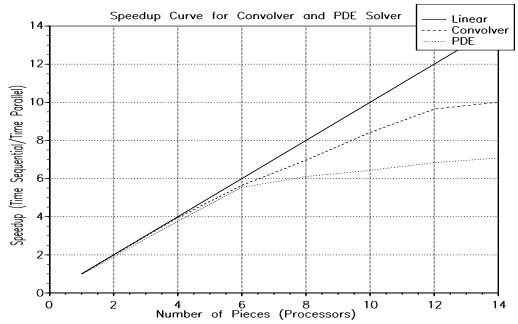


Figure 11 - Stenciler Speedup Curve

processors and then declining marginal return from the addition of more processors. This phenomenon is caused by two factors. First, computation granularity decreases as the problem is broken into smaller and smaller pieces, causing synchronization and communication to become a larger percentage of execution time. Second, time spent executing code that is inherently sequential, such as file I/O, increases as a percentage of total run time. We would expect that this decline in performance would be less pronounced with larger problem sizes or a more computationally expensive stencil function. Overall, the convolution example performed better because its computation granularity is significantly greater than for the PDE solver.

6. Summary and Future Work

When we began developing our framework for stencil algorithms, we had several goals in mind: (1) the environment should be easy to use for the application developer, facilitating rapid development of new stencil applications, (2) the framework should provide as much built-in functionality as possible to exploit the similarities among stencil algorithms and to reduce the effort needed by the programmer, (3) the framework should encourage code re-use wherever possible, (4) the programs generated must exploit the parallelism inherent in stencil algorithms and perform adequately, and (5) the code developed should be portable. The `Stenciler` class hierarchy meets these goals. The object-oriented framework and especially the built-in functions provided in the base class support the goals of ease of use, built-in code for common tasks, and code re-use. The parallel environment of the Mentat system provides the basis for exploiting the data parallelism natural to stencil algorithms. Performance results show that the programs produced using the `Stenciler` framework are capable of high performance. Finally, the Mentat environment provides an environment for easily porting user code from one environment to another.

The results are encouraging, but there are a number of areas where our approach can be improved and extended. The class interface and hierarchy still needs some clarification and revision. These improvements will become apparent as our experience with using this approach increases. We plan to port the `Stenciler` class and sample implementation code to a broader range of hardware platforms. We also would like to extend the domain of problems that can be supported using our general approach by creating higher dimensional stencil class hierarchies and applying the same technique to other classes of algorithms. Finally, we would like to extend the model of our base classes to include encapsulation of scheduling and problem decomposition decision information to facilitate better performance while keeping the details away from the user.

Availability

Mentat is available via anonymous FTP for Sun 3, Sun 4, SGI, iPSC/2, and iPSC/860. Paragon, CM-5 and RS6000 versions are expected in Summer 1993. The `Stenciler` code and related documentation will also be available by the time of the conference (July 1993). For further information, send email to *mentat@Virginia.edu*.

Acknowledgments

We would like to thank Mike DeLong for his help with the mathematics behind the Jacobi iteration PDE solver.

7. References

[1] A. S. Grimshaw, “Easy to Use Object-Oriented Parallel Programming with Mentat,” *IEEE Computer*, pp. 39-51, May, 1993.

[2] A. S. Grimshaw, E. Loyot Jr., and J. Weissman, “Mentat Programming Language (MPL) Reference Manual,” University of Virginia, Computer Science TR 91-32, 1991.

[3] A. S. Grimshaw, W. Timothy Strayer, and Padmini Narayan, “Dynamic, Object-Oriented Parallel Processing,” to appear in *IEEE Parallel and Distributed Technology: Systems and Applications*, May 1993.

[4] *Proceedings of the First Annual Object-Oriented Numerics Conference*, April 1993.

[5] J.F. Macfarlane, et al, “Application of Parallel Object-Oriented Environment and Toolkit (POET) to Combustion Problems,” Sandia report, September 1992.

[6] B. Stroustrup, *The C++ Programming Language*, 2nd ed. Addison-Wesley, Reading, Mass., 1991.

[7] Rafael C. Gonzales and Paul Wintz, *Digital Image Processing*, 2nd ed. Addison-Wesley, Reading, Mass., 1987.

[8] M. J. Quinn, *Designing Efficient Algorithms For Parallel Computers*, McGraw-Hill Book Company, New York, 1987.