Parallel Array Classes and Lightweight Sharing Mechanisms

Steve W. Otto

Oregon Graduate Institute
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 93-009

April 1993

To appear in proceedings, Object Oriented Numerics Conference, April 1993, sponsored by SIAM and Rogue Wave Software.

Parallel Array Classes and Lightweight Sharing Mechanisms*

Steve W. Otto
Dept of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology
19600 NW von Neumann Dr
Beaverton, Oregon 97006-1999
otto@cse.ogi.edu

April 7, 1993

Abstract

We discuss a set of parallel array classes, MetaMP, for distributed-memory architectures. The classes are implemented in C++ and interface to the PVM or Intel NX message-passing systems. An array class implements a partitioned array as a set of objects distributed across the nodes – a "collective" object. Object methods hide the low-level message-passing and implement meaningful array operations. These include transparent guard strips (or sharing regions) that support finite-difference stencils, reductions and multi-broadcasts for support of pivoting and row operations, and interpolation/contraction operations for support of multigrid algorithms.

We generalize the concept of guard strips to an object implementation of lightweight sharing mechanisms for FEM and PIC algorithms. The sharing is accomplished through the mechanism of weak memory coherence and can be efficiently implemented. The price of the efficient implementation is memory usage and the need to explicitly specify the coherence operations. An intriguing feature of this programming model is that it maps well to both distributed-memory and shared-memory architectures.

1 Introduction

This paper describes MetaMP, a programming environment for distributed-memory computers. MetaMP stands for "meta-message-passing" and this should give the reader the clue that it has evolved from the standpoint of message-passing programming. The environment consists of a pre-processor, a set of C++ classes, and a runtime system that map onto a few well-known message-passing systems: PVM, NX, and Express. The attempt is to abstract SPMD, message-passing programming to a more usable level, without sacrificing much performance. The focus is on scalable, data-parallel applications, and only a subset of such applications is currently well-supported.

The fundamental idea of the system is to support important, parallel data structures such as (partitioned) multi-dimensional arrays and (partitioned) unstructured meshes. Beyond just the data structure, important mechanisms associated with these data structures are implemented as object methods. The mechanisms provided are those that seem to be natural for each data structure. A few examples of these primitives are (for arrays):

- update of guard strips (object method),
- abstract iterators (compiler level),

*To appear in proceedings, Object Oriented Numerics Conference, April 1993, sponsored by SIAM and Rogue Wave Software.

- row, column broadcasts (object method),
- bulk I/O primitives (object method).

For the unstructured mesh data structure we have the primitives:

- set-up of alias or sharing regions (object method),
- merge alias values (object method),
- abstract iterators (compiler level).

Most of these are provided as object methods but a few, notably iterators, are implemented by our preprocessor. This is done so that we can perform the same efficient access of arrays within loops as done by good Fortran compilers [1].

Hopefully, the primitive object methods give an efficient implementation of important constructs and idioms for parallel programs. If they are heavily re-used, we can justify a large effort to write very efficient methods and to port them to many platforms.

Before getting to the specifics of MetaMP, let us make a few comments about what we have found useful in basing our efforts on C++. What we say is perhaps not new, but it is interesting to say what works and what doesn't in the context of parallel applications. We have found that the use of object-oriented techniques:

- promotes data abstraction. No surprise here, but more specifically we achieve data abstraction through the use of:
  - function overloading. An object method such as **`elem()`** (single element access of an array) is consistently called this even though there are many different kinds of element access in the system.
  - self-describing data structures. All the information that describes the shapes, sizes, strides, etc., of a particular instance of a data structure is stored within that data structure itself. This makes it very easy to pass around things like a "2-D, (block, block-cyclic)" array and let the methods themselves inquire as to what particular type of partitioned array we have.
  - parameterized types. "Templates" in C++. This language feature allows one to create generic container classes independent of the particular data types that are to be put inside the structure. For example, once we have implemented 3-D partitioned arrays as a generic container class, we can simply re-use that same class to store a 3-D array of `int`s or a 3-D array of triples of **`double`**, etc.

Data abstraction, in turn, promotes parallel programming that is independent of specific implementation choices such as:

- partitioning choice (e.g., block versus block-cyclic). Abstracted iterators and object methods allow one to conveniently write programs that continue to function correctly when the partitioning choice is changed.
- data types. The "all-meets-all" parallel algorithm looks the same whether the elements are triples of floating-point numbers (N-body gravity) or row, column combinations of a matrix (Jacobi eigensolver).
- details of I/O. The array "read" method knows how to read from a striped, parallel file. Since the file was originally constructed by "write," information was written into the file which makes the file self-describing. For instance, if the file was written with a block-cyclic partitioned array on 8 processors, and is being read into a block partitioned array on 32 processors, the read method can know this and load in the data correctly.

Conversely, we have not found all C++ features to be useful. For example, we do *not* use operator overloading at all. The allowable operators and their syntax is very limited; I'm also not sure a parallel APL-like language would be very easy to read.

In some parts of the system, we have implemented constructs in the pre-processor even though one could do it via methods. A good example of this is iterators. In order to make sure that array element access within loops is efficient, we take control with the pre-processor and implement the loop and then re-write the body of the loop for efficient element access.

Finally, deep inheritance trees can cause problems. It is certainly useful to add a feature to a MetaMP class by using it as a base class and then inserting an extra method or two. Heavy use of inheritance can easily cause one to violate some assumption made somewhere in the hierarchy. For example, many MetaMP methods must be called "loosely synchronously" – that is, all processors must eventually get around to calling the method. If this is not satisfied the likely result is deadlock.

2 MetaMP Programming Model

We will discuss the details of the MetaMP array class library. The C++ classes of the library implement the partitioned array as a "collective" [2, 3] object within an SPMD programming model. This means that there is a separate instance of an array object on each processor and that these objects work together to give a consistent view of a single, global data structure such as an array. Parameters such as those that describe the global shape of the array are replicated across the object instances; those parameters that describe the local features of the sub-data structure (e.g., the sub-array on this processor) are unique to the local object instance. The data of the data structure is also not replicated unless it is explicitly aliased by a `guardStrip` or `alias` modifier at construction time.

The object methods provide important operations that access or modify the distributed data structure. Many methods contain communication or synchronization operations, implying that they must be called in a loosely synchronous fashion. A compiler could (often) check whether or not one is satisfying this requirement and this is an enhancement to the system we are considering.

We now give an overview of the MetaMP array class features and object methods.

Global Indexing of Arrays

Array elements are accessed via global indexing. There is no local index of an array element on a processor provided to the user. The choice of global indexing provides a consistent picture of what each array element is named and is implemented without additional runtime cost. If array elements are accessed from within loops governed by the abstract iterators, then one is guaranteed to access only on-processor elements of the array.

In the applications we have developed, we have achieved the access of off-processor array elements by the mechanisms of guard strips, aliasing regions, and bulk rolls or shifts of the array. That is, we employ usual message-passing techniques, but they are expressed at the level of the array and not in terms of explicit communications buffers.

The question arises as to whether one can easily provide for the simple access of off-processor array elements. In the DPC system of Quinn and Hatcher [4], off-processor references are handled by sending the requests and receiving the replies at the synchronization points implied by the DPC language. We are trying to do this within the more asynchronous framework of SPMD programs. In this case however, it seems that one needs access to an active-message layer in the message-passing system [5]. Baber's Hypertasking system [6] provided an off-processor capability. Due to the high degree of non-portability among message-passing systems in their support of active-messages, we do not have off-processor access in the current MetaMP array classes. Another comment to be made is that one can often achieve the desired effect in a more efficient manner through the use of aliasing. We will return to this point later in the paper.

Partitioning Syntax

For describing the partitioning of multi-dimensional arrays, we have borrowed from the syntax of HPF [7]. The pre-processor for MetaMP understands a syntax for the creation of "templates" (an abstract array of virtual processors to which real arrays align to), a distribute statement for templates (map the template onto the processors in block, block-cyclic fashion), and an align statement for the real array to the template (arrays are aligned with templates; the templates are mapped onto the physical processors).

This was done so as to provide a rich syntax, allowing the creation of many different mappings of arrays onto processors, but also a syntax that is becoming standardized.

Iterators

Iterators provide for each processor looping over all the array elements it owns. As such, the iterator takes an array class instance as an argument. There can be multiple statements within an iteration. All elements on a processor need not be looped over; the global index domain can be restricted and the strides need not be 1. We again specify this by borrowing from Fortran 90 notation:

```
[ lower : upper : step ],
```

i.e., the lower bound, the upper bound, and the step between successive members of the set. These numbers are in terms of the global dimensions of the array. The iterator will convert these to the starting and ending values for each processor.

As was mentioned previously, iterators are implemented by our pre-processor. We do this in the pre-processor (rather than as a method) so as to ensure that we get the same efficient array access within loops as that done by Fortran compilers. The pre-processor re-writes the body of the loop (in C++), performing induction variable creation and strength reduction optimizations.

Locality-based Access

Potential off-processor access of a local nature (limited distance in terms of the array indices, but not necessarily limited distance in memory) is provided by integrated guard strips. One can think of guard elements as being read-only copies of array elements on neighboring processors. There are object methods to update the guard values (all directions are done at once) and "corner values" are gotten without additional runtime cost. That is, separate messages do not need to be sent to a processor in a diagonal direction.

Bulk Data Movement

For writing efficient pipelining algorithms such as matrix multiplication and N-body gravity direct, there are object methods that transfer the entire sub-array from one processor to the next processor along some dimension of the array. All processors do this in parallel and the top processor wraps, leading to a rotated version of the partitioned array. The communications are performed efficiently and also the object instances are updated by the `roll()` method so that they "know" that they now contain a different section of the array. This is essential so that other methods, such as iterators, continue to function correctly. Another example of a bulk data-movement operation is matrix transpose, although the current array classes don't have this.

Broadcasting Primitives

These are primitives that capture the idea of broadcasting along some direction of an array. They are clearly useful for linear algebra, where row and column broadcasts are common. Actually specifying a row-broadcast from a dynamic origin (occurs in partial-pivoting) is quite involved and uses the `owner()` method so that processors can independently compute the processor that owns a certain row. The actual

broadcast is done with the `pointToSubArray()` method (so that a vector can alias to a row of a matrix) and the `copy()` method.

I/O – Bulk read/write Operations

Dumping an entire array to a (possibly parallel) file is provided as an object method. Consider a `read()`. The method knows the shape, partitioning, number of processors, and so on of the array desired in memory; it also reads similar information written in the file (the file is self-describing). Using this, it is easy to convert layouts, and to efficiently read the file.

Visualization – Drawing 2-D Arrays

A set of methods that give a convenient interface to draw a 2-D array in an X-window are being developed.

Aliasing Mechanisms for Unstructured Meshes

This can be thought of as a generalization of guard strip ideas to that of a "writable" guard strip. The writable guard can also be described as a weakly coherent, shared-memory. The "weak coherence" means that the programmer must specify when the shared memory is to be made consistent. This will be discussed in more detail later in the paper. There are object methods for set-up of the alias groups, and for the coherence operations.

Other Features

To make a programming environment usable, there are some other features which are helpful to have. These are not tied to any particular data structure and are not object methods. They include:

- portable timing primitives – a simple interface to a clock
- global broadcast, global reduction, barrier functions
- coordination primitives for writing asynchronous, master-slave parallel programs
- load balance display as an X-window
- X-window library for simple drawing

3 Example Applications

We show usage and explain more points about the array classes through some example parallel programs. The examples used in this section are a Laplace solver which is then extended to a Multigrid solver. The next section will discuss aliasing for FEM and PIC applications.

Laplace Solver

This is a program that solves a 2-D Laplace or Poisson problem via simple Jacobi relaxation. Though this is a far from optimal elliptic solver, we will turn it into a powerful one in the next section. Dirichlet boundary conditions can be set at any point on the regular 2-D mesh that describes the space. We won't show all the code, but merely the interesting features for parallelism.

We begin by declaring a two dimensional mesh of processors, and a 2-D template that is block distributed across the processor mesh. Our HPF-like notation for this is:

```
$ ProcMesh p1 p2 $
$ Template Two(M,N) $
$ Distribute Two(Block-p1,Block-p2) $
```

The $'s alert the pre-processor that these are MetaMP statements. These don't actually produce executable code, but instead cause the pre-processor to treat the set of processors as a $p1 \times p2$ mesh and to map the template array (regard this as a virtual set of $M \times N$ memory cells) `Two` in a block fashion onto the processor mesh.

The actual 2-D arrays are made by C++ constructors, modified with an HPF-like align statement:

```
array2D<float> phi(M,N);      $ Align phi(i,j) with Two(i,j) $
```

The align lines the actual array `phi` up with the template array `Two`. It has the effect of adding several arguments to the C++ constructor which must appear on the same line. All of this is nothing but a convenient syntax which follows HPF.

To get an array with a guard strip of width 1, we do:

```
$ Template TwoG(M,N) $
$ Distribute TwoG(Block-p1,Block-p2) GuardStrip 1 $

array2D<float> phio(M,N);      $ Align phio(i,j) with TwoG(i,j) $
```

The Jacobi iteration consists of an update of the guard strip values for the array `phio` (update the read-only shared memory cells), followed by an iteration through all the memory cells of `phi`:

```
phio.updateGuard();
$ phi.iterate i over [1:M-1:1] j over [1:N-1:1] $ {
        ... body of loop
}
```

`updateGuard()` is an array method, while `iterate` is expanded by the pre-processor. The loop body is written in terms of operations on individual array elements, using the `elem()` method:

```
phi.elem(i,j) = 0.25*( phio.elem(i-1,j) + phio.elem(i+1,j)
                       phio.elem(i,j-1) + phio.elem(i,j+1) );
```

Note that the guard strips are integrated into the arrays themselves. That is, the shared points are located at seemingly "illegal" points such as:

```
phio.elem( phio.start(1)-1, j)
```

where `phio.start(1)` is where the array `phio` begins on this processor (for dimension 1).

To make things a bit more concrete, we can say that the iterator is expanded out to:

```
for (i=phi.start(1); i<phi.end(1); ++i) {
    for (j=phi.start(0); j<phi.end(0); ++j) {
        ...body
    }
}
```

for block decompositions. Actually, it's a bit more complicated than this due to the fact that the iterator allows general `[l:u:s]` iteration sets. The starting and ending points on a given processor may not be simply given by the array starting location. For block distributions, the `elem()` method accesses a contiguous set of memory cells in the usual C fashion and is inlined for efficiency:

```
float& elem( int i, int j ) { return *(basePtr+i*stride[1]+j) }
```

Multi-Grid Extension, Graphics

The 2-D Laplace solver of the previous extension can almost trivially be converted to a 3-D solver by switching to the **array3D** class. Another extension is to extend the Jacobi relaxation solver to a Multi-Grid solver [8]. Fundamental to multi-grid techniques are array refinement and coarsening operations. These are illustrated in figure 1.

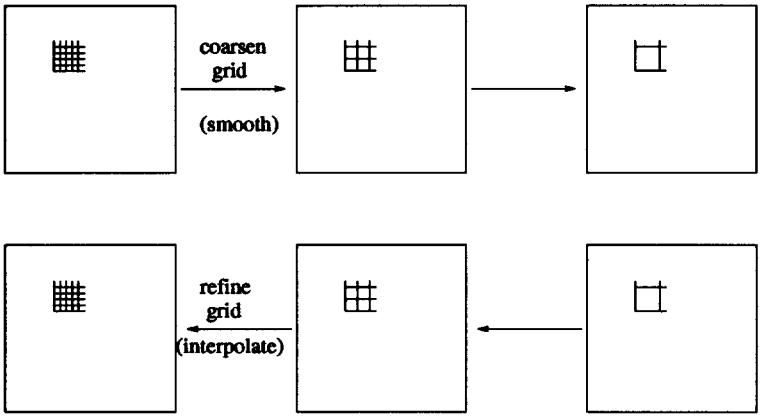

Figure 1: Coarsening and refinement operations for multi-grid. In coarsening, values from a fine grid are averaged and written to a coarse grid of fewer cells. In refinement, values from a coarse grid are interpolated to a fine grid of more cells.

The multi-grid program can be expressed with just the classes and methods presented in the previous section. We could statically define arrays of size **(M,N)**, **(M/2,N/2)**, **(M/4,N/4)**, ... and then copy values from one array to another. We have taken the slightly further step and defined **resize()** object methods which are used to re-interpret the array mapping and effectively change the size of the array. The same could be done by destructing and constructing the array, but **resize()** is light weight in the sense that memory is not re-allocated. **resize()** re-maps the array – this means that as the array shrinks it remains properly distributed across the processors, it doesn't, for example, all shrink into processor 0. Finally, as in a conventional program for multi-grid, the programmer still explicitly copies values, element by element, from an array of size **(M,N)** to one of size **(M/2,N/2)**.

Figure 2 shows the multi-grid solver running on a PVM installation. The solution is shown as a contour plot. This plot is gotten by simply opening a window with the **openPlot()** method:

```
phio.openPlot("phi field");
```

and drawing the field into the window via:

```
phio.draw();
```

Also shown in the figure are a picture of the residual field and a usage of the load balance display utilities. The residual field display was easily added by creating another partitioned array of the same alignment as **phi**, setting it proportional to the residual during the convergence check phase of the program, and drawing using the **draw()** method. The load balance display is used by calling it with each processor's estimate of its Mflops.

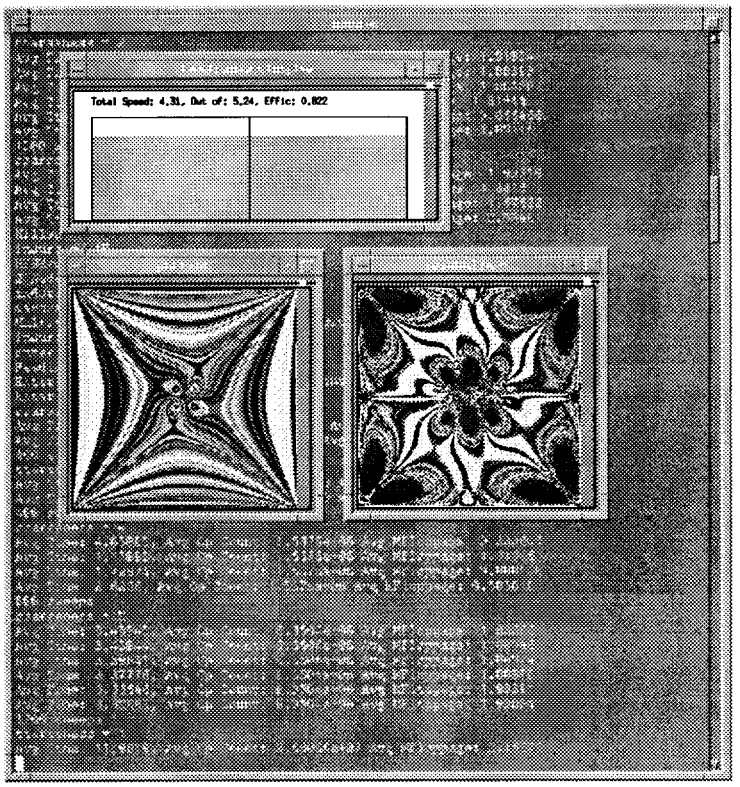

Figure 2: The Multi-Grid application running on PVM. The two large windows show the solution (on the left) and a picture of the residual field. The third window shows the load balance display. This function is called loosely synchronously and is given an estimate of the achieved Mflops on that processor. In the situation shown we are running on two processors and achieving 4.3 Mflops out of an estimated max of 5.24 for an efficiency of .82

4 Lightweight Sharing

In this section we will discuss our second effort at a useful set of classes for parallel programming. We generalize the idea of guard strips to the notion of a *writable guard strip*. When this idea is pursued, the guard region can be thought of as forming sets of memory cells which are shared between processors. In keeping with our philosophy of explicitly stating when communications occur, this shared memory is a *weakly coherent shared-memory*. This means that the program explicitly states when the various copies of the shared-memory are to be made coherent. Finally, in contrast to usual shared-memory models, no single copy of the memory "wins" and overwrites the other copies. Instead, the multiple copies are merged with some operator (usually a "+") in a symmetric way and all copies are replaced with the merged values.

The ideas are explained using two important scientific applications: unstructured mesh finite element methods, and particle-in-cell (PIC) algorithms.

Unstructured Meshes, FEM

Central to iterative solvers for finite element problems are matrix-vector products. If the operator or matrix is stored in a sparse form, we can think of such computations as being represented by the uni-processor code:

```
do i = 1,N
        do j = 1, n[i]
                y[i] += A[i][j] * x[nbrs[i][j]]
```

Here, the vectors `y[]` and `x[]` represent fields which live on the nodes of the unstructured mesh; the index `i` is the node label. `n[i]` is the number of neighbors of node `i`, and `nbrs[i][j]` is the label of the `j`th neighbor of node `i`. The matrix is stored such that `A[i][j]` is the matrix element between node `i` and the `j`th neighbor of `i`. The computation is illustrated in figure 3.

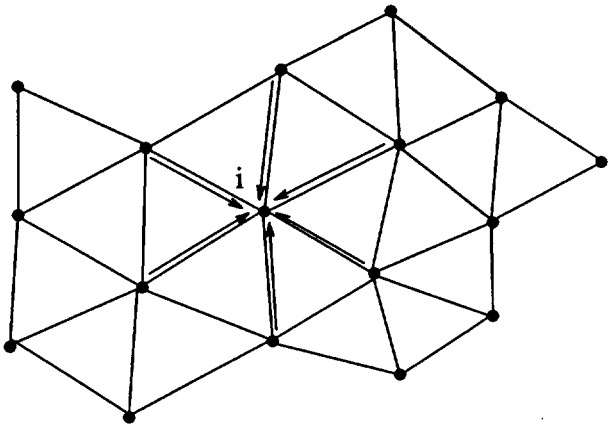

Figure 3: The sparse matrix-vector product. The matrix is non-zero only on nodes connected by edges, the arrows denote the neighbors of node `i` that are "pulled in" and summed to give a contribution to the vector element at `i`.

One suggested approach for parallelizing this type of computation is discussed by Koelbel, Saltz, Mehrotra, Berryman [9]. In this method, we partition the mesh nodes among the processors in a unique manner, that is, each node is stored in only one processor. The computation of the matrix-vector product then proceeds in an "owner-computes" fashion: the sum that will be stored at node `i` is done by the processor that owns node `i`. This is shown in figure 4.

The code for the matrix-vector product gets broken into two parts: one set of loops sums up contributions from on-processor vector elements, the second set of loops sums contributions to all the nodes connected to an off-processor node. The pseudo-code becomes something like:

```
loopOverAllMyPoints i
        do j = 1, n_local[i]
                y[i] += A[i][j] * x[nbrs_local[i][j]]
Communications -- bring non-local x values into local buffer
loopOverBndyPoints i
        do j = n_local[i]+1, n_non_local[i]
                y[i] += A[i][j] * buff[buff_index[i][j]]
```

`n_local[i]` is the number of on-processor neighbors of node `i`, `nbrs_local[][]` enumerates them. Similarly, for each node `i`, `n_non_local[i]` is the number of off-processor neighbors of node `i`, and `buff_index[][]` enumerates their locations in the communication buffer.

The owner-computes method has lead us into an unnatural split into a local and a non-local computation. A similar thing occurs in regular (structured mesh) computations. We have seen earlier how the integration of the "guard strip" into the fundamental data structure itself can lead to a cleaner parallel program that is still efficient. We pursue a similar idea here, with the extension that one can define the guard strip to be writable, as opposed to the read-only guard strips used in the Laplace and Multi-Grid solvers.

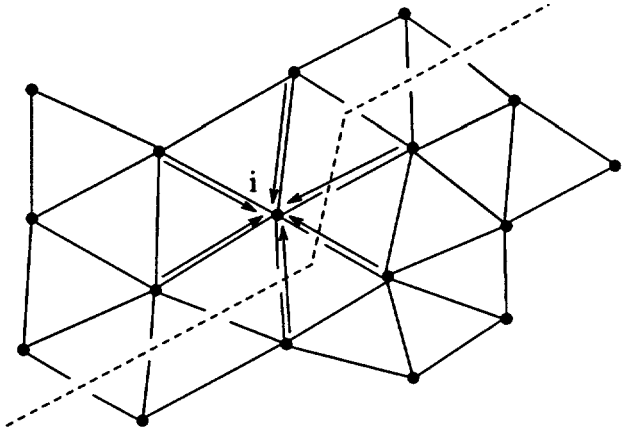

Figure 4: Owner computes for the parallel sparse matrix-vector product. The processor boundary is shown by a dotted line, arrows crossing the boundary represent inter-processor communications.

When integrating the guard strip into the unstructured mesh, we follow some of the ideas of Williams [10] and create *multiple aliases* of mesh nodes that are on processor boundaries. A set of alias groups for the previous unstructured mesh partition is drawn in figure 5.

Again we partition the unstructured mesh, but with overlap: the nodes along the processor boundary are replicated onto two (or more) processors. The copies of a single point form an alias group and should be thought of as a weakly coherent, shared-memory. The aliases are *all* writable, and the results of write operations are merged at some point specified by the program. This is where aliases differ from conventional shared-memory. Instead of one copy being the current "true" copy which will overwrite the other copies, we take the symmetric choice and say that the values coming from the separate writes must be merged with a generalized "+" operator (i.e., a commutative, associative binary function).

The pseudo-code now takes on the pleasing structure:

```
loopOverAllMyPoints i
        do j = 1, n[i]
                y[i] += A[i][j] * x[nbrs[i][j]]
Communications -- merge aliased y values with + function
```

Aliasing has formally promoted the communication buffer of the owner computes approach to portions of the vectors `x[]` and `y[]` that are shared. As is the case for the owner computes technique [9], the alias approach has a useful inspector / executor optimization strategy. The alias groups are created only once at the beginning of the program, and tables are constructed so as to efficiently perform the message passing – that is, we can coalesce all the alias communications between any pair of processors into one, large message.

The status of our unstructured mesh work is that the author and T. Kubaska of Intel SSD have working C codes for both the case of assembled and un-assembled matrices [11]. We are currently measuring speeds on an iPSC 860 and are re-working the software to provide a C++ interface. This interface will have the following features:

- vectors and sparse matrices are automatically partitioned collective objects,

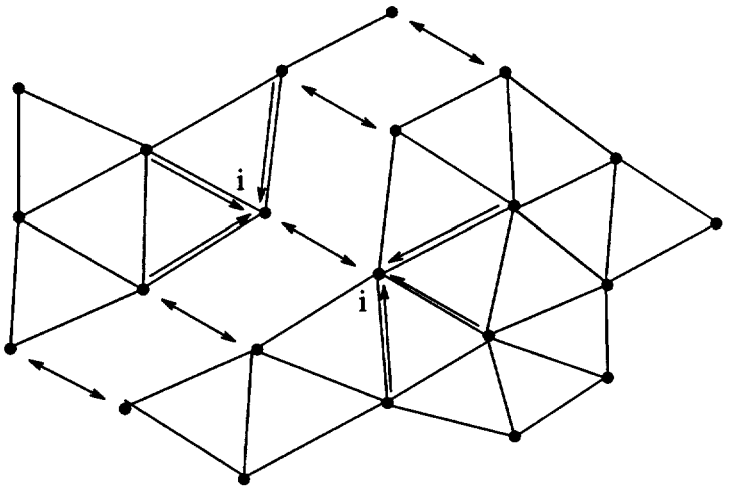

Figure 5: Alias groups for unstructured mesh computation. The double-ended arrows show each alias group. Mesh points along a processor boundary are replicated. The copies of a single point form an alias group and should be thought of as a weakly coherent, shared-memory. The aliases are *all* writable, and the results of write operations are merged at some point specified by the program.

- methods are called to define the alias points and to do the initial set-up (or inspector) portion of the computation,
- abstract loop over this processor's mesh nodes is provided,
- merging of aliases is again an object method.

Aliases can be thought of as an extension of guard strip ideas, but in the next section we will give an example of a computation in which the aliases are not merely along the processor boundaries. In addition, we will make stronger use of the fact that the aliases are writable. This will be essential for the efficiency of the calculation.

Particle-In-Cell Algorithms

In this section we discuss some aspects of particle-in-cell algorithms [12] and how they might be implemented cleanly and efficiently using the framework of lightweight aliases. The applications of interest are plasma simulations, where one has charged particles moving about in space, and electro-magnetic fields defined on a regular mesh.

The four major phases for each time step of a PIC algorithm are:

- Scatter: accumulate current densities ($\mathbf{j}$) at mesh points from the particles in neighboring cells. See figure 6.
- Field solve: advance the electro-magnetic fields ($\mathbf{E}$, $\mathbf{B}$) one time step. Like $\mathbf{j}$, these fields are located on the mesh sites.
- Gather: accumulate and interpolate field values to particle locations. See figure 7.
- Push particles: compute the force on each particle, and move it forward one time step.

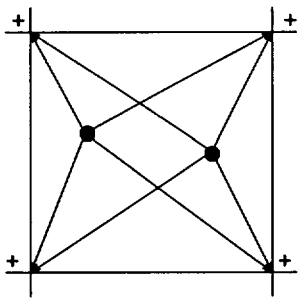

Figure 6: The particles in a cell "scatter" their position, velocity, and charge information to the **j** field located on the mesh sites. The "+" indicates that contributions from multiple particles are (vector) summed.

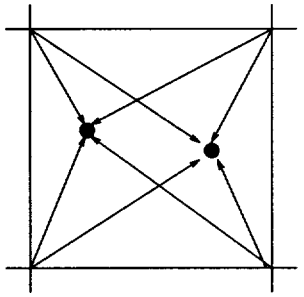

Figure 7: Field values are accumulated and interpolated to the particle positions.

Parallel PIC, Version 1

Aliasing techniques make it easy to arrive at efficient parallel algorithms. In a first version of a parallel PIC program, we can alias the **j, E, B** fields along processor boundaries as shown in figure 8. As before, the aliases have the effect of integrating guard strips or communication buffers into the fundamental data structure. A strong case can be made that the alias approach is necessary for an efficient implementation. Suppose there are several particles located in a cell along the processor boundary. Then, with the owner computes approach, all of the particle data would have to cross the processor boundary. With aliasing, only the cumulated values (the partial sums) need cross.

Parallel PIC, Version 2

A second version of parallel PIC is motivated by observations of D. Walker [12]. One of the features of real PIC simulations is that particles tend to clump up and so, in order to load balance, we actually want to employ different distributions of the particles and mesh. Figure 9 shows such a "non-conformant" distribution – for the mesh we choose a normal, block distribution, but for the particles we choose a hierarchical (in dimensions), adaptive distribution. Each distribution comes into play at different phases of the PIC algorithm, and each is effective at balancing the load during its phase.

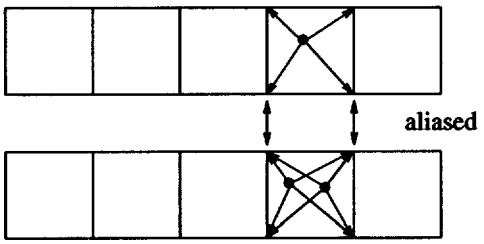

Figure 8: PIC version 1. If we alias boundary mesh sites, then instead of all the particle data crossing the processor boundary, only the cumulated values need cross.

We define an entire *region* to be aliased between processors. Figure 10 shows the two distributions superimposed, and a region of aliasing is shaded. This region is written to and read from by both processors and as before, a merge operation with a + operator makes the shared-memory coherent. Only one of the sharing regions is shown in figure 10. There are similar sharing regions between other pairs of processors in the figure.

Structure of PIC Algorithm

Aliasing seems to give a clean way of expressing efficient parallelizations for both conformant (version 1) and non-conformant (version 2) PIC algorithms. For each version, the basic structure of the parallel algorithm is as follows.

- Scatter from particles.

```
loopOver particles
        scatter from particle to local copy of j field
make aliased j field coherent
```

- Field solve.

```
run conventional, parallel field solve using field regions
make aliased E,B fields coherent
```

- Gather to particles.

```
loopOver particles
        gather, interpolate field values to particle location
```

- Push particles.

```
loopOver particles
        move particle one time step
need to migrate some particles
```

Gaussian Elimination

We have written a parallel Gaussian elimination program using the array classes. The program performs partial pivoting. The search for the next pivot and the multi-cast of the pivot row are done through well-defined object methods. As is well known, Gaussian elimination has potential load imbalance due

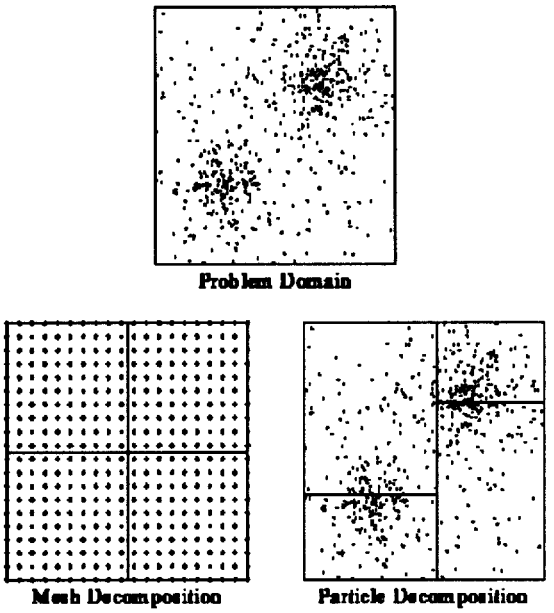

Figure 9: Non-conformant distributions. Due to particle clumping, we use an adaptive distribution that balances the (particle) load amongst the processors. During the field solve phase, we still want a conventional block distribution.

to the effect shown in figure 11. As the computation proceeds, the active region of the matrix shrinks towards the lower right corner.

One known attack on the load imbalance is to use a cyclic or block-cyclic distribution of the matrix. Then, even as the active region shrinks, many processors stay involved in the computation. We wish to support cyclic or block-cyclic distributions transparently through our abstracted iterators. To keep efficient access of the array elements, one again needs compiler support of the iterator. Mike Sharp is producing a C++ to C++ translator that will provide this efficient access [1].

N-Body Gravity, Jacobi Eigensolver

Parallel versions of N-body gravity and the cyclic Jacobi eigensolver both rely on an efficient all-meet-all primitive. When we say N-body gravity we mean the direct, N^2 algorithm, and the Jacobi method for eigenvalues is within a constant factor of the best known algorithms in the sequential case, and is scalable (in contrast to the others). The all-meet-all primitive can be represented by the sequential, triangular, double loop:

```
for ( i=0; i<N; ++i )
        for ( j=0; j<N; ++j )
                if ( i < j ) interact( &obj[i], &obj[j] );
```

All-meet-all is a parallel version of this, using algorithms such as those found in [13]. It is implemented as an object method, where the object is a one dimensional array of elements that can be an arbitrary datatype, and it also takes the `interact()` function as an argument. `interact()` is user-defined, and must take pointers to a pair of elements. For N-body gravity, it is the function that computes the force between the `i`th and `j`th particles, for Jacobi, it takes rows and columns of a pair of matrices and does

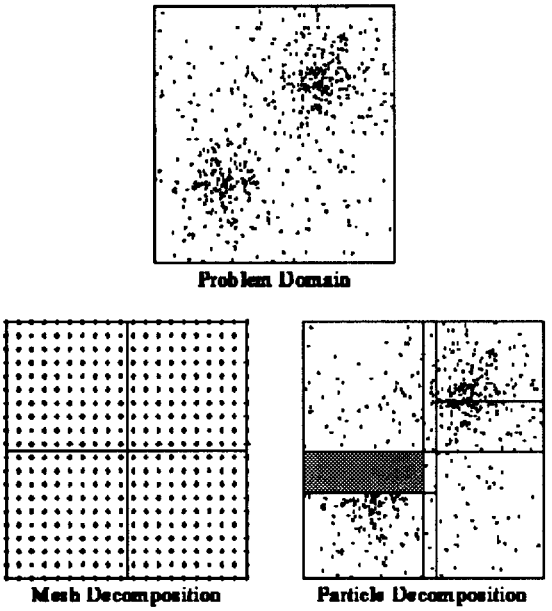

Figure 10: Alias region of **j**, **E**, **B** fields between two processors. We show the two distributions superimposed. The shaded area is shared by the processors above and below it.

a transformation on them. All-meet-all is an efficient and easy to use abstraction for the situation of a distributed list of items, and where each item must "meet" every other item.

5 Performance Results

The following table shows speed-up results for the Laplace and multi-grid programs on an iPSC/860. The absolute speed for one processor for all three programs was approximately 1.8 Mflops.

Program	N=1	2	4	8	16
laplace2, 512 × 512	1	1.9	3.8	7.4	14.7
laplace3, 64 × 64 × 64	1	1.9	3.8	7.3	14.5
mg2, 512 × 512	1	1.8	3.6	7.1	13.0

We are currently measuring speeds for our FEM / aliasing program, also on an iPSC/860 machine.

For our PVM implementation, we measure a speed of 2.2 Mflops per HP-720 processor for the multi-grid program. Figure 2 showed this program running on two HP-720 workstations for a total speed of 4.3 Mflops. Relatively tightly coupled applications such as multi-grid do not scale up well on this LAN-based system. Future networks, or PVM running on scalable parallel machines, will alleviate this problem.

6 Asynchronous Search Programs

Before closing, we would like to mention our asynchronous, loosely coupled parallel search programs. These run on PVM and scale-up quite successfully. They are a parallel implementation of a powerful search heuristic for solving combinatoric optimization problems [14, 15]. Figure 12 shows the parallel

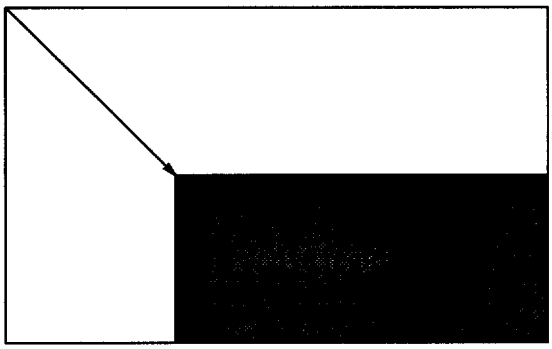

Figure 11: Gaussian elimination. The active region of the matrix shrinks as the algorithm proceeds down the diagonal.

graph partitioning (GP) solver running on 7 workstations, achieving an efficiency of 91%. Note that in a heterogeneous situation such as this, speed-up is difficult to define, but efficiency (what fraction of all the possible cycles was captured?) is still a valid concept. A related program for solving the traveling salesman problem (TSP) shows similar behavior.

7 Conclusions

The partitioned array support that is provided by our array classes certainly seem useful. The basis of C++ has made it easier to write re-usable software and we have successfully applied the fundamental object methods to several distinct parallel programs.

Unstructured mesh applications and parallel programs such as PIC seem to be naturally expressed in terms of a weakly-coherent, shared memory system. We are finding that these map well down to message-passing systems. Equally intriguing is the fact that these methods may map to large-scale shared memory architectures in quite a nice way. Aliased variables are precisely those which should be replicated in a shared memory implementation so as to avoid effects such as cache-line thrashing. The coherence calls are those places at which the replicated variables must be brought together.

Elaborate parallel applications demand carefully constructed parallel data structures and mechanism. Though it is true that some of the ideas discussed here are finding their way into HPF compilers, this seems to us to stretch the idea of what a compiler should do. Compilers preclude extension; a mixed approach of compiling and class definitions may be more workable.

We are in the process of making the MetaMP pre-processor and array classes and runtime system available. Application programs are also available. For information send email to Steve Otto at: `otto@cse.ogi.edu`.

Acknowledgments

I thank Jon Inouye, Ravi Konuru, Ted Kubaska, Robert Prouty, Mike Sharp, David Walker, Jonathan Walpole, and Roy Williams for useful discussions. I would like to acknowledge Intel Supercomputing Systems Division and OACIS, the Oregon Advanced Computing Institute, for financial support.

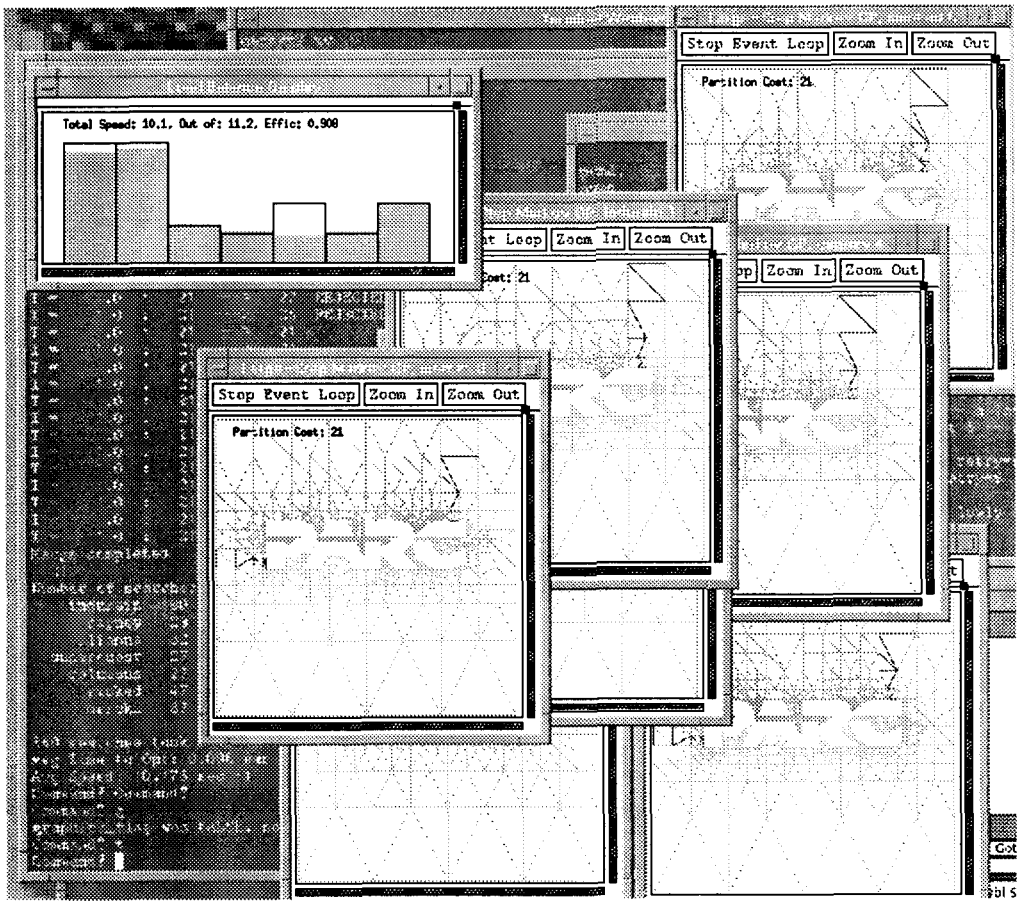

Figure 12: The parallel GP solver running with MetaMP/PVM. The many windows show the current solution that is being searched on each processor. This run executed on 7 workstations, a mix of Hewlett Packard and SUN Microsystems machines. The load balance display on the upper left shows that at this point, the parallel search is running at an efficiency of .91. The program is computing the min-cut bi-partition of a FEM mesh.

References

[1] M. Sharp and S. Otto. A class specific optimizing compiler. In *Object Oriented Numerics Conference '93*, April 1993.

[2] F. Bodin, D. Gannon, S. Narayana, P. Beckman, and S. Yang. Distributed pc++: Basic ideas for an object parallel language. In *Object Oriented Numerics Conference '93*, April 1993.

[3] M. Lemke and D. Quinlan. P++, a C++ virtual shared grids based programming environment for architecture-independent development of structured grid applications. Technical report, GMD: Gessellschaft fur Mathematik und Datanverarbeitung MBH, 1992.

[4] P. Hatcher and M. Quinn. *Data-Parallel Programming on MIMD Computers.* The MIT Press, 1991.

[5] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: a mechanism for integrated communication and computation. In *19th Annual International Symposium on Computer Architecture*, pages 256–66. ACM Press, 1992.

[6] M. Baber. Hypertasking support for dynamically redistributable and resizable arrays on the iPSC. In *The Sixth Conference on Hypercube Concurrent Computers and Applications*, pages 59–66. IEEE Computer Society Press, 1991.

[7] The High Performance Fortran Forum. Draft 1.0 of HPF. Technical report. Send electronic mail to netlib@ornl.gov with "send hpf-v10.ps from hpf" in the message body. The report is sent as a Postscript file. This site also has the LaTeX source of the draft; use "send index from hpf" to see the file names.

[8] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes in C, Second Edition.* Cambridge University Press, 1992.

[9] C. Koelbel, P. Mehrotra, J. Saltz, and H. Berryman. Parallel loops on distributed machines. In D. Walker and Q. Stout, editors, *The Fifth Distributed Memory Computing Conference.* IEEE, 1990.

[10] R. Williams. Voxel databases: A paradigm for parallelism with spatial structure. Technical Report CCSF-19-92, Caltech Concurrent Supercomputing Facility, 1992.

[11] T. Kubaska. Light weight sharing mechanisms for sparse matrix computations. Master's thesis, Oregon Graduate Institute of Science & Technology, 1993. In preparation.

[12] D. Walker. Particle-in-cell plasma simulation codes on the connection machine. In *Computer Systems in Engineering.* 1991.

[13] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume 1. Prentice Hall, Englewood Cliffs, NJ, 1988.

[14] O. Martin, S.W. Otto, and E.W. Felten. Large-step Markov chains for the traveling salesman problem. *J. Complex Syst.*, 5:3:299, 1991.

[15] O. Martin, S.W. Otto, and E.W. Felten. Large-step Markov chains for the TSP incorporating local search heuristics. *Oper. Res. Lett.*, 11:219–24, 1992.