Parallel Programming Using C++
A Survey of Current Systems

Gregory V. Wilson (editor)

Chapter 1

C**

PLEASE IGNORE THE UGLY LAYOUT OF THIS INTRODUCTORY MATERIAL; IT WILL BE FIXED UP DURING PRODUCTION.

Parallel Programming in C**: A Large-Grain Data-Parallel Programming Language

James R. Larus, Brad Richards, and Guhan Viswanathan
Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton Street
Madison, WI 53706 USA
{larus,richards,gviswana}@cs.wisc.edu

This work is supported in part by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant #F33615-94-1-1525 and ARPA order no. B550, an NSF NYI Award CCR-9357779, NSF Grants CCR-9101035 and MIP-9225097, DOE Grant DE-FG02-93ER25176, and donations from Digital Equipment Corporation and Sun Microsystems. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Wright Laboratory Avionics Directorate or the U.S. Government.

1.1 Introduction

C** is a large-grain data-parallel programming language. It preserves the principal advantages of SIMD data parallelism—comprehensible and near-determinate parallel execution—while relaxing SIMD's constricted execution model [14]. We have used C** as a vehicle for experimenting with parallel language features and with implementation techniques that exploit program-level control of a parallel computer's memory system [16]. This paper both describes the language and summarizes progress in language design and implementation since the previous C** paper [14].

Data-parallel programming languages originally evolved on fine-grain SIMD computers [6], which execute individual instructions in lockstep on a collection of processing units. Early data-parallel languages, such as C* [24], mimicked this execution model by executing each parallel operation in lockstep. Both the machines and languages benefited from the simplicity of a single thread of control and the absence of data races. However, both suffered from SIMD's intrinsically inefficient conditional statements—in which each processing unit steps through both arms of a condition—and inefficiencies introduced by the synchronization necessary to run SIMD programs on the more common MIMD processors.

Section 1.2 outlines several alternatives to SIMD execution of data-parallel languages. Most of these languages take a pragmatic approach and run data-parallel operations in MIMD style, i.e. asynchronously. This approach causes no problems for simple operators, such as whole-array arithmetic, and offers the notational convenience of structuring a program with data-parallel operators. However, uncontrolled sharing allows data races, and these data-parallel languages provide few, if any, mechanisms for serializing conflicts. In effect, these languages trade a higher-level programming model for implementation ease and the siren's lure of high performance.

C** follows a different approach. It defines a clear semantics for conflicting memory references in asynchronously executed data-parallel operations (Section 1.3). In C**, invoking a data-parallel operation on a data aggregate asynchronously executes the operation on each element in the collection. However, C**'s semantics require that invocations appear to execute simultaneously and instantaneously, so that their memory references cannot conflict. This semantics is similar to the copy-in-copy-out semantics of primitive operations in other data-parallel languages. However, unlike other languages, C**'s semantics is not limited to a few arithmetic operations on dense matrices. Instead, C** defines and implements this semantics for arbitrary C++ code.

C** prevents conflicts within a data-parallel operation by deferring the delivery of values until after the operation completes and by providing a rich and extensible collection of reduction operators to combine conflicting values. A data-parallel operation can modify memory, but changes do not become globally visible until the operation completes. At that point, modifications from different invocations are reconciled into a globally consistent state for the next data-parallel operation. This mechanism works well for one-to-one and one-to-many communication, but is insufficient for many-to-one or many-to-many communication since providing a semantics for conflicting writes to the same location is difficult. Instead, C** supports the latter forms of communication with a rich variety of reductions, including reduction assignments and user-defined reduction functions (Section 1.4).

Of course, a clear semantics is no substitute for high performance. Our C** implementation exploits Tempest [9, 16], an interface which provides user-level code with the mechanisms to implement a shared-address space and a custom coherence policy. LCM, C**'s memory system, allows shared memory to become inconsistent during data-parallel operations. When a data-parallel operation modifies a shared location, LCM uses a fine-grain, copy-on-write coherence policy that matches C** semantics to copy the location's cache block (Section 1.5). When the data-parallel operation finishes, LCM reconciles copies to create a consistent global state.

As an extended example, Section 1.6 contains our solution to the polygon overlay problem Appendix **??**. The first version is an inefficient, but concise, data-parallel program, which is greatly improved by using a better algorithm. This algorithm's performance is in turn improved by using high-level C** mechanisms—in particular, user-level reductions—to improve communication. Other benchmarks also show that, although high-level and concise, C** programs can run as fast, or faster than low-level, carefully-written and tuned programs.

It is a commonplace that parallel programming is difficult, and that parallel machines will not be widely used until this complexity is brought under control. If true, the programming languages community bears responsibility for this failure. It has invested more effort in packaging hardware-level features, such as message passing, than in exploring new languages that raise the level of programming abstraction, such as HPF.

Parallel languages with a higher-level semantics may not please all programmers or solve all problems, but they do make parallel programming easier. The architecture community draws a clear distinction between mechanisms, which hardware should provide, and policy, which software should implement [31, 32]. The languages community should look at hardware mechanisms as a means to an end, not an end in itself.

1.2 Data-Parallel Languages

A *data-parallel* programming language expresses parallelism by evaluating, in parallel, operations on collections of data [11]. The key features of such a language are: a means for aggregating data into a single entity, which we will call a *data aggregate*; a way to specify an operation on each element in an aggregate; and a semantics for the parallel execution of these operations. Data-parallel languages should be distinguished from the data-parallel programming style [7], which can be used even in languages that do not claim to support data parallelism.

Unfortunately, the definition of data parallelism is a bit fuzzy around the edges. For example, languages such as Fortran-90 [1] and HPF [12] mix other programming models with a limited collection of data-parallel operations. A sub-language can be data parallel, even though its parent language is not. In addition, functional languages provide data aggregates and operations [2], but typically do not consider parallel execution. However, since these languages do not permit side effects, extending their semantics to permit parallel execution is straightforward.

Data-parallel languages differ widely in the operations and semantics that they provide. We choose to classify data-parallel operations into four categories: fine grain, coarse grain, large grain, and functional. The discussion below is organized around this classification, since these categories strongly affect semantics. Techniques for specifying data aggregates differ mainly in syntactic details and are not discussed further.

1.2.1 Fine-Grain Languages

As discussed earlier, *fine-grain* data-parallel languages originally evolved from the model of fine-grain SIMD machines, such as the ICL DAP and Thinking Machines CM-1 Connection Machine [10]. Beyond hardware simplicity, a fine-grained SIMD model offers several programming advantages. Since each SIMD processor executes the same instruction simultaneously, a parallel program has a single thread of control and is easier to understand. In addition, read-write and write-read data races cannot occur since a parallel instruction reads its input before computing and writing its output. The only possible conflicts are output dependencies in which two instructions write to a memory location. Some machines (e.g., CM-2) provide an elaborate collection of mechanisms for combining values written to a memory location.

Unfortunately, SIMD execution has a fatal disadvantage for many programs. In particular, lockstep execution is extremely costly for programs with conditionals, since each processor must step through both arms of a conditional, although it only executes code from one alternative.

Several languages, such as C*[1] [24], directly implement a SIMD model

[1] Version 5 of C*. Version 6 changed the language significantly; this section considers

and consequently inherit its advantages and disadvantages. In C*, a *domain* is a collection of data instances, each of which is associated with a virtual processor. The virtual processors for a domain execute operations on their instance in lockstep. The granularity of a lockstep operation is a language operator, rather than a machine instruction (a distinction which makes little difference in C). Although it was designed for SIMD machines, Quinn and Hatcher successfully compiled C* for MIMD machines by eliminating unnecessary synchronization and asynchronously executing sequences of non-conflicting instructions [8, 28].

Fine-grain data parallelism has other manifestations as well. Lin and Snyder distinguish point-based data-parallel languages, such as C*, from array-based ones [18]. Array-based languages, such as ZPL [18] or parts of Fortran-90 [1] and HPF [12], overload operators to apply to data aggregates—for example, add arrays by adding their respective elements—and provide array shift and permuatation operations. This approach expresses parallelism through compositions of the initial parallel operators. For some application domains, such as matrix arithmetic, point-based languages produce clear and short programs.

To contrast these approaches, compare Program 1.1, which contains a point-based stencil written in C* and Program 1.2, which contains an array-based stencil written in HPF.

```
domain point {
  float x;
} A [N][N];

[domain point].{
  int offset = (this - &A[0][0]);
  int i = offset / N;              /* Compute row and column */
  int j = offset % N;
  if ((i > 0) && (j > 0) && (i < N) && (j < N))
      x = (A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1]) / 4;
}
```

Program 1.1: Point-based stencil written in C* (version 5).

```
A(1:N, 1:N) = (A(0:N-1, 1:N) + A(2:N+1, 1:N) + A(1:N, 0:N-1) + A(1:N, 2:N+1)) / 4
```

Program 1.2: Array-based stencil written in HPF.

Both types of fine-grain languages communicate by reading and writing memory. SIMD and whole-array operations share a read-compute-write

only the older version.

semantics in which a parallel operation reads its input values before modifying a program's state. For example, in both stencils, the computations average neighboring values from the previous iteration.

These semantics prevent read-write, but not write-write, conflicts. Some fine-grain languages allow reduction functions to combine colliding values into a single value. We refer to this process as a *reduction assignment*. An alternate view of a parallel assignment operator treats it as a *mapping* [25, 30], which is a restricted many-to-many communication operator.

1.2.2 Coarse-Grain Languages

Fine-grain data parallelism either shares SIMD's inefficient execution model, which requires excessive synchronization on non-SIMD hardware. An obvious generalization is to execute data-parallel operations asynchronously, so that each invocation of an operation runs independently of other ones. Although this change eliminates inefficient conditional statements, it also raises new problems with memory conflicts between parallel tasks. The original SIMD data-parallel model requires no locks, barriers, or other explicit synchronization. Asynchronous data-parallel languages, for the most part, ignore the possibility of conflict and allow these error-prone features of MIMD programming.

Coarse-grain data-parallel languages allow data-parallel operations to execute asynchronously. For example, HPF's `INDEPENDENT DO` loops [12] or pC++'s parallel member functions [17] execute arbitrary code as a data-parallel operation. These languages provide no guarantees about conflicting memory accesses, so a programmer must ensure that parallel operations are independent. For example, an HPF stencil operation (Program 1.3) needs two copies of an array to ensure that updates do not interfere with reads. The real cost of coarse-grain data parallelism is the time and effort required to write, understand, and debug the complex code and not the storage or time overheads.

```
!HPF$ INDEPENDENT
DO 10 I=1,N
  !HPF$ INDEPENDENT
  DO 10 J=1,N
    A1(I, J) = (A2(I-1, J) + A2(I+1, J) + A2(I, J-1) + A2(I, J+1)) / 4
10 CONTINUE

!HPF$ INDEPENDENT
DO 20 I=1,N
  !HPF$ INDEPENDENT
  DO 20 J=1,N
```

```
    A2(I, J) = (A1(I-1, J) + A1(I+1, J) + A1(I, J-1) + A1(I, J+1)) / 4
20 CONTINUE
```

Program 1.3: Coarse-grain stencil in HPF.

Communication in coarse-grain data-parallel languages again occurs through assignment to memory. Assignments may cause conflicts, and a programmer must ensure that parallel operations are data race-free by avoiding conflicting data accesses or by resolving collisions with reductions. Most coarse-grain languages limit reductions to a predefined set of operators, but some, such as HPF, are considering adding user-defined reductions.

1.2.3 Large-Grain Languages

Large-grain data-parallel languages allow coarse-grain parallelism, but provide a clearly defined semantics for conflicting memory accesses. For example, C** [14] specifies that each invocation of a data-parallel operation runs as if executed simultaneously and instantaneously, so that all invocations start from the same memory state and incur no conflicts. When an invocation updates a global datum, only that invocation sees a change to the memory state until the data-parallel operation completes. At that point, all changes are merged into a single consistent view of memory. Program 1.4 shows how a stencil operation can be written in C**. The pseudo-variables, `#0` and `#1`, are bound to each invocation's i^{th} and j^{th} coordinates, respectively. Since it is written with only one copy of the array, this code is similar to the point-based stencils described earlier (Program 1.1).

```
A[#0][#1] = (A[#0-1][#1] + A[#0+1][#1] + A[#0][#1-1] + A[#0][#1+1]) / 4;
```

Program 1.4: Large-grain stencil in C.**

Large-grain languages permit conflict-free execution of coarse-grain programs, at the expense of considerable compiler analysis or run-time complexity. However, as discussed below, this complexity is manageable and the semantic clarity is beneficial to programmers.

1.2.4 Purely Functional Languages

The final data-parallel languages are purely functional (e.g., NESL [5]) and offer advantages of data parallelism and functional programming. Conflicts do not occur because these languages do not permit imperative updates. As a result, the languages need not limit grain size or define new memory semantics to guarantee deterministic execution. On the other hand, these languages present all of the implementation difficulties of conventional functional languages [22].

Data communication in purely functional languages occurs through function arguments and return values. Functional languages heavily use reductions to combine values returned from parallel functions, thereby providing powerful many-to-one communication mechanisms. Reductions, unfortunately, do not extend easily to many-to-many communication, so programmers must build and decompose intermediate structures.

1.3 C** Overview[2]

C** is a large-grain data-parallel language (to use the taxonomy and concepts introduced in Section 1.2). It was designed to investigate whether large-grain data parallelism is both useful as a programming paradigm and implementable with reasonable efficiency. After several years of effort, the answer to both questions appears to be “yes”.

C** introduces a new type of object into C++. These objects are *Aggregates*, which collect data into an entity that can be operated on concurrently by parallel functions. C** also introduces *slices*, so that a program can manipulate portions of an Aggregate. These concepts are extensions to C++, so a C** program can exploit that language's abstraction and object-oriented programming facilities.

1.3.1 Aggregates

In C**, Aggregate objects are the basis for parallelism. An Aggregate class (Aggregate, for short) declares an ordered collection of values, called Aggregate elements (elements, for short), that can be operated on concurrently by an Aggregate parallel function (parallel function, for short). To declare Aggregates, C**extends the class definition syntax of C++in two ways. First, the programmer specifies the type of the Aggregate element following the name of the Aggregate. Second, the number of dimensions and their sizes follow the element type. For example, the following declarations define 2-dimensional matrices of floating point elements of an indeterminate and two determinate sizes:

```
class matrix(float) [] [] {···};
struct small_matrix(float) [5] [5]  {···};
class large_matrix(float) [100] [100] {···};
```

[2]This section is a revised version of the C** language description, which appeared elsewhere [14]. The language syntax has evolved slightly, but the basic concepts have not changed.

Like C++classes, Aggregates use either the keyword `class` or `struct` to declare a new type of object. Unlike C++classes, Aggregates have a rank and cardinality that is specified by their declaration. An Aggregate's data members can be either basic C++types or structures or classes defined by the programmer. An Aggregate's *rank* is the number of dimensions specified in its class declaration. Rank is defined by the declaration and cannot be changed. The *cardinality* of each dimension may be specified in the class declaration. If omitted from the class, the cardinality must be supplied when the Aggregate is created. Each dimension is indexed from 0 to $N-1$, where N is the cardinality of the dimension. For example, indices for both dimensions of a `small_matrix` run from 0 to 4.

An Aggregate object looks similar, but differs fundamentally, from a conventional C++ array of objects:

- An Aggregate class declaration specifies the type of the collection, not of the individual elements. This is an important point: `matrix`, which is an Aggregate, is an object consisting of a two-dimensional collection of floating point values, not a two-dimensional array of objects.

- Aggregate member functions operate on the entire collection of elements, not individual elements (Section 1.3.2).

- Elements in an Aggregate can be operated on in parallel, unlike objects in an array.

- Aggregates can be sliced (Section 1.3.5).

However, Aggregate elements can be referenced in the same manner as objects in an array. For example, if `A` is a `small_matrix` object, `A[0][0]` is its first element.

1.3.2 Aggregate Functions

Aggregate member functions are similar to class member functions in most respects. A key difference, however, is that Aggregate member functions are applied to an entire Aggregate, not just an element, and that the keyword `this` is a pointer to the entire Aggregate. For example, in:

```
class matrix (float) [] []{
  friend ostream& operator<< (ostream&, matrix&);
};

ostream& operator<< (ostream &out, matrix &m) {
  for (int i = 0; i < cardinality (0); i++) {
    for (int j = 0; j < cardinality (1); j++)
```

```
      out << m[i][j] << " ";
    out << '\';
  }
}
```

the operator `<<` is a friend function that prints a matrix to a stream.

All Aggregates automatically have the following two member functions:

`int rank ()`	– *Return number of dimensions in Aggregate*
`int cardinality (int dim)`	– *Return cardinality of dimension* `dim`

1.3.3 Aggregate Constructors

As in C++, Aggregates may define constructors and destructors. An Aggregate constructor initializes the entire collection of elements, not the individual elements. By contrast, each element in an array is initialized by a call to the type's default constructor. For example:

```
class matrix (float) [] [] {
  matrix (float initial_value) {
    int i, j;
    for (i = 0; i < cardinality (0); i++)
      for (j = 0; j < cardinality (1); j++)
        (*this)[i][j] = initial_value;
    };
};
```

defines an Aggregate matrix of floating point values whose constructor initializes all matrix elements to a specified value, so that

```
new matrix [100][100] (1);
```

creates a 100×100 matrix of 1's.

1.3.4 Parallel Functions

Aggregate member and friend functions are sequential by default. A *parallel function* is a member or friend function in an Aggregate class that can be invoked simultaneously on each element of the Aggregate. A parallel function (either a member or friend function) is identified by keyword `parallel` after its argument list. For example:

```
class matrix (float) [] [] {
  float checksum () parallel;
  friend transpose (parallel matrix) parallel;
};
```

declares `checksum` and `transpose` to be parallel functions.

In a parallel member function, the *parallel argument* is the Aggregate object to which the function is applied. In a parallel friend function, the parallel argument must be prefaced by `parallel` (for example, the first argument in `transpose`).

Analogous to the variable `this`, which points to the entire Aggregate, parallel functions may also use the `self` pointer, which points to the element the invocation operates on.

A parallel function behaves as if it were invoked simultaneously on all elements of its parallel argument. An invocation of a function on an Aggregate element can determine the coordinates of its element from the pseudo variables:

`#0`	1^{st} coordinate
`#1`	2^{nd} coordinate
...	
`#`$n-1$	n^{th} coordinate

Semantics of Parallel Functions

To explain parallel functions more precisely, we must define a few terms. A parallel function *call* is the application of a parallel member or friend function to an Aggregate. A parallel function *invocation* is the execution of the function on one Aggregate element. Hence, calling a parallel function starts many function invocations, all of which appear to execute simultaneously. A function invocation's *state* is the collection of memory locations read or written during the invocation.

"Applied atomically" means that while a parallel function is executing, its state is only modified by the function itself, not by other concurrently executing tasks. In other words, the function appears to execute instantaneously and is unaffected by anything else running at the same time. In effect, the semantics are as if each invocation executed as follows:

- Atomically copy all referenced locations into a purely local copy.
- Compute using local copies.
- Write all modified copies back to global locations.

Since a parallel function is applied *simultaneously* to an Aggregate's elements, all invocations begin with identical state (except for the pseudo variables, `#0`, `#1`, etc., which differ in each invocation).

As an example, consider a stencil computation on a matrix:

```
friend void stencil (parallel matrix A) parallel
{
  A[#0][#1] = (A[#0-1][#1] + A[#0+1][#1]
              + A[#0][#1-1] + A[#0][#1+1]) / 4.0;
}
```

The computation is applied simultaneously to each element of the matrix, so the new values are entirely a function of the old.

If invocations of a parallel function modify global state, C** only guarantees that each modified location will contain a value computed by some invocation. If an invocation modifies more than one location, only a portion of its modifications may be visible after the parallel call. Part of a multi-location modification can be overwritten by other invocations.

Results From Parallel Functions

If a parallel function's result type is the same as its parallel argument's type, the parallel function allocates a new Aggregate of the same size and type as the parallel argument and initializes it with results returned from the corresponding invocations. The function invoked on the first element of the parallel argument computes the value for the first element of the result, and so on. Since the parallel function initializes the result Aggregate, the Aggregate's constructor, if any, is not invoked.

On the other hand, if the result type is a scalar, the values returned by the parallel function must be returned in reduction return statements (Section 1.3.4) that combine results from the invocations into a single value of the specified type.

For example:

```
friend matrix operator* (parallel matrix A, matrix B) parallel
  {return (A[#0][] * B[][#1]);}

friend float operator* (parallel mrow R, mcol C) parallel
  {return%+ (R[#0] * C[#0]);}
```

is a matrix multiplication routine that creates and returns a new parallel matrix of the same size as the parallel matrix `A`. The invocation associated with element (i, j) of `A` computes the dot product $A(i, *) \cdot B(*, j)$.

Reductions

Reductions are a basic operation in data-parallel programming because they provide a conflict-free and efficient way of combining results from independent computations. A reduction applies an associative binary operator, pair-wise, to a sequence of values. For example, if $\circ$ is the operator, the reduction of the sequence v is $v_1 \circ v_2 \circ v_3 \ldots \circ v_n$. Given n processors, this reduction can be applied in parallel in $\lg n$ steps.

C** supplies two types of reductions. The first combines values assigned to a location to avoid conflicts between concurrent writes. The second combines values returned from the multiple invocations of a parallel functions.

Reduction Assignment

Reduction assignments are legal only within parallel functions. A reduction assignment uses the operator specified to the right of the % to combine the values assigned to the location in different invocations. Reduction assignments are necessary because an ordinary assignment permits only one invocation to modify the location. Changes to the location are not visible until the parallel function call completes.

For example:

```
float sum = 0.0, pos_sum = 0.0;

matrix::sum_elements () parallel {
  sum =%+ *self;
  if (*self > 0)
    pos_sum =%+ *self;
}
```

computes two sums. The first is the sum of all elements in a matrix. The second (`pos_sum`) is the sum of the positive elements in the matrix.

Reduction Returns

A parallel function with a scalar result must combine the results from each invocation with a reduction return. This return combines multiple values using the operator specified to the right of the %.

For example:

```
friend float sum (parallel mrow A) parallel
  {return%+ (A[#0]);}
```

1.3.5 Slices

A *slice* selects a subset of an Aggregate along an axis. A slice is not a copy. It shares all selected elements with the larger Aggregate. Slices are particularly valuable when they themselves are also Aggregates and consequently can be manipulated in parallel. Slices permit effective specification of parallel computations on pieces of an Aggregate. For example, many matrix computations are naturally described in terms of operations on rows and columns. If the row and column slices are Aggregates, the operations can be data parallel.

In C**, omitting an index expression from a dimension of an Aggregate reference produces a slice that includes all elements along that dimension. This slice differs from the pointer to an array element that results from omitting a subscript in a C++ array reference. Trailing empty braces may be omitted, so that `A[1][]` is equivalent to `A[1]`. For example, the following three expressions are slices of a matrix:

```
matrix A;

A[i]    – i-th row of A
A[i][]  – equivalent to the above
A[][j]  – j-th column of A
```

Subclassing and Slices

In C**, the type of a slice is a subclass of the Aggregate's class, in all respects except inheritance of functions. The declaration of a slice's subclass both names the new subclass and describes which indices must be specified in computing the slice. `#n` denotes the n^{th} index in a reference to the slice. For example, consider the following matrix slices:

```
class mrow : matrix[.];    //  row slice of matrix
class mcol : matrix[][.];  //  column slice of matrix
```

`mrow` is a row from a matrix that is computed by omitting the column index.

1.4 User-Defined Reductions

Since C** was first defined, the principal change to it has been the addition of user-defined reductions. This feature permits efficient many-to-one communication by providing a well-defined semantics for conflicting writes. Existing data-parallel languages typically resolve collisions with a limited collection of reduction functions. C** provides *user-defined reductions*, which are arbitrary binary functions that combine colliding values. This feature extends reduction functions to user-defined data types and enables programmers to combine values in many more situations.

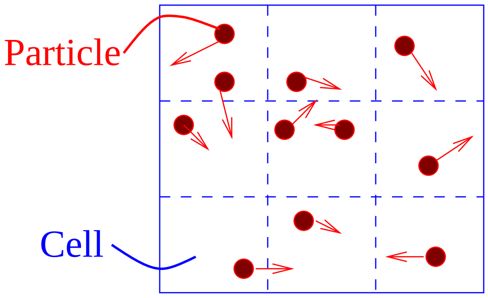

Figure 1.1: DSMC Schematic Diagram

1.4.1 DSMC Example

To motivate user-defined reductions, consider DSMC, a particle-in-cell code that simulates particle movement and collision using discrete Monte Carlo simulation [29]. DSMC divides space into cells in a static cartesian grid and distributes molecules among cells (Figure 1.1). Molecular interactions create forces that move molecules. DSMC simulates the movement and collision of molecules over many time steps, each of which comprises three phases:

move: molecules change position based on their velocities.

enter: new molecules enter the domain from a jet stream.

collide: molecules collide with others in the same cell.

Parallelism in DSMC can be expressed as data-parallel operations on either cells or molecules. The latter technique is more complex as the collide phase requires an extra mapping to associate molecules with cells. The discussion below is based on the cell approach, which is used in practice [29, 21].

The move phase requires many-to-many communication since it updates the positions of all molecules. In the process, molecules may move from one cell to another (usually neighboring) cell. Inter-cell molecule transfer results in many-to-many communication, since each cell exports molecules to different destinations and receives molecules from many other cells.

Consider expressing this pattern in a fine-grain data-parallel language. A fine-grain language can transfer molecules between cells through parallel assignment. Let *Leave* and *Enter* be the maximum number of molecules leaving and entering a cell in a time step, respectively. A parallel assignment can transfer only one molecule at a time. Therefore, a fine-grain language requires *Leave* repetitions of the assignment to transfer all molecules. Furthermore, unless the language provides reductions capable of resolving conflicts, these parallel assignments would fail when two molecules are simultaneously transferred to the same cell. To handle this case, the transfer must become a two step process. In the first step, senders to a cell choose a winner. In the second step, the winner transfers a molecule. In the worst case, this algorithm requires $Leave \times Enter$ parallel steps, each of which involves two synchronizations. User-defined reductions or a parallel prefix operation to enumerate incoming molecules reduces the cost of combining colliding molecules, however, the algorithm still require *Leave* steps in a fine-grain language.

In a coarse-grain language, this algorithm is typically implemented using low-level synchronization primitives, such as locks or monitors. These constructions are undesirable since they introduce explicit synchronization, which destroys the data-parallel abstraction. In addition, their high cost, and the serialization they induce, can prevent parallel speedup [21].

By contrast, user-defined reductions capture the many-to-many communication pattern of molecule transfer. In a language that supports them, such as C**, DSMC can use an `append` function to resolve conflicts between molecules entering a cell by merging lists of them. This approach leads to good performance. For example, on a 32 processor CM-5, a version of DSMC written in C** using user-defined reductions ran 1.1 times faster than the hand-coded C version of the program.

1.4.2 Syntax of User-Defined Reductions

User-defined reductions combine colliding values in a parallel assignment. These reductions are specified by a function name in a reduction operation. For example, the polygon overlay C** code (Program 1.7) uses the combining function `merge` in the statement `return %merge theList : nullList;`. The `nullList` argument is initial value of the accumulator.

In general, the reduction function for values of type α has type $\alpha \times \alpha \rightarrow \alpha$. For example, the `merge` function in Program 1.7 has type `polyList_s` $\times$ `polyList_s` $\rightarrow$ `polyList_s`. The current C** compiler uses a variant of this typing rule in which the first parameter serves as both input and output (e.g., type `(polyList_s *)` $\times$ `polyList_s` $\rightarrow$ `void`).

1.4.3 Semantics of User-Defined Reductions

Although the concept of user-defined reductions is not new, they have not been implemented in another parallel programming language (to the best of our knowledge). However, the MPI message-passing library does provide for user-defined reductions [20]. In the process of implementing user-defined reductions, several questions arose about their semantics. The discussion below describes these choices and how we resolved them in C**.

Conflicts

If used in reduction assignments, user-defined reduction functions are invoked during execution of a parallel function, which, in languages such as C**, has a well-defined semantics for conflicting memory accesses. What restrictions (if any) should be placed on memory accesses in reduction functions?

The most severe restriction is to require that reduction functions be side-effect free, so their output is a pure function of their input.[6] This approach has many advantages (and is used by SIMD hardware and fine-grain languages) since pure functions do not introduce new conflicts. Although this restriction may be reasonable for atomic data, such as numbers, it can impede reductions on structured data, since it demands a functional style that may introduce considerable overhead. Consider, for example, using user-defined reductions to add new nodes to an oct-tree in a Barnes-Hut algorithm [4]. Without side-effects, each insertion requires copying all nodes along the path from the root to the leaf. Even this approach is problematic in languages like C, that separate memory allocation from initialization since, without an effect system [19], assignments will appear as side effects.

Alternatively, a language may permit compiler directives, such as HPF's INTENT directive, to declare safety properties that a compiler cannot prove. This approach allows reduction functions that have side effects but do not cause conflicts. However, erroneous directives may lead to subtle conflict errors.

A third alternative requires a run-time system to identify data access conflicts due to unsafe user-defined reductions, as in Steele's Parallel Scheme [?]. This approach identifies all errors at runtime, but is complex and expensive.

In the end, we decided not to restrict user-defined reduction functions and to trust programmers to use this powerful feature carefully, since it could easily subvert the language semantics. This decision was in part

[6] Note that side effects and associativity are independent properties since a routine with side effects can easily be associative.

necessary because of the difficulty of determining if an arbitrary piece of C++ code is side-effect free and in part desirable so programmers had the freedom to experiment with this new language feature.

Reduction Assignment Result Availability

If a parallel function references the target location of a reduction assignment, what value does it see after the assignment? With the original C** semantics, a parallel function sees only its update, as if it runs sequentially. Although this provides a simple semantic model, it can be difficult to implement, so we considered several alternatives.

The first is to prohibit non-reduction accesses to a location used for reduction assignments, as in Fortran D [**?**]. This method's advantage is that it provides a run-time system with the flexibility to determine when and how a reduction should be carried out. However, this method has two major disadvantages. First, it burdens a programmer unnecessarily. For example, in DSMC's move phase, the parallel function for a cell reads its molecule list and moves molecules to other cells' lists. The restriction requires a programmer to maintain a separate "incoming" list for each cell, and merge it with the cell's molecule list in a separate phase. Second, static analysis cannot identify all erroneous accesses. More complex and expensive run-time disambiguation is necessary to detect all errors.

The second approach is to retain the original value of a location, even after a reduction assignment. When a data-parallel operation completes, the run-time system combines colliding values and updates locations. In effect, this approach performs reductions after the data-parallel operation (although a system has the freedom to perform the two phases concurrently, so long as a reduction does not conflict with a data-parallel operation).

The final approach combines the two and is suitable for a language like C**. Like the second alternative, it defers reductions, but it also updates a local copy of a location to merge the local contribution. This approach ties reductions closely to C**'s semantics and may not be appropriate for coarse-grain languages in general.

Combining Order

Since primitive reduction functions are associative and commutative, the order in which values are combined does not affect the result. This freedom provides a run-time system with flexibility to implement reductions efficiently. For example, a run-time system can use a combining tree to implement reductions in logarithmic parallel time. User-defined reductions, on the other hand, are not necessarily associative or commutative, but are usually effectively associative. *Effectively associative functions* are reduction functions whose lack of associativity does not affect a program's results [25]. For example, consider using an `append` function as a set union operator for a set in which the order of items is semantically unimportant.

Arbitrary reduction order is a reasonable choice for user-defined reductions for two reasons. First, a programmer can implement a specific ordering by collecting all values into a list (e.g., using `append`), sorting the list, and combining values explicitly. More importantly, the absence of ordering allows optimizations that improve the performance of user-defined reductions.

1.4.4 Extensions to User-Defined Reduction Assignments

A reduction assignment is actually two actions, combining conflicting values and updating the target with the value. By default, the update operation is simple assignment. C** generalizes the update with a user-defined *update* function. An update function can store the combined value in the target in non-trivial ways. For example, in DSMC (Section 1.4.1), an update function merges the list of incoming molecule with a cell's list of molecules. A user-defined update is only syntactic sugar and adds no power to the language. To continue the DSMC example, without user-defined updates, the programmer must store the incoming list in a temporary and append it to the cell's list later. Semantically, an update function satisfies the same restrictions as a combining function (Section 1.4.3). The `insertPoly` function Program 1.5 (reproduced below from the polygon overlay code in Program 1.10) is a example of an update function. `insertPoly` adds incoming polygons to a grid cell's polygon list.

```
void insertPoly(polyNode_p *result, poly_s thePoly)
{
  polyNode_p ptr = new polyNode_s;//  allocate new node
  ptr->poly = thePoly;            //  fill it in
  ptr->next = *result;            //  link node into list
  *result = ptr;                  //  return result
}
```

Program 1.5: Code to Partition a Polygon Vector.

Another option is to omit a combining function and only specify an update function. The C**runtime system uses the update function to insert colliding values, one by one, in the target location. The function `polyVec::partitionVec` in Program 1.10 uses the update function `insertPoly` (in Program 1.5), in this manner, to add incoming polygons one by one. Updates without combining are analogous to integrated reductions [33].

1.4.5 Implementing User-Level Reductions

User-defined reductions can be implemented on a message-passing machine or by using the Tempest interface with minimal run-time system support. Furthermore, the semantics of reductions provide a compiler with opportunities for optimization. This section describes implementation of user-defined reductions in the C** compiler. In C**, reductions follow the deferred reduction model that retains old values. Currently, C** does not specify methods to identify or prevent conflicts between independent reductions. All mechanisms described in this section extend easily to other data-parallel languages. The description has three stages. The first describes basic mechanisms that the compiler uses to implement reductions. The second shows how the compiler can vectorize messages to improve performance. The third describes how local combining allows the compiler to reduce the amount of data communicated between processors.

Basic Reductions

A reduction assignment updates its target with the combined value of colliding right-hand-side values. The C**implementation involves two processors: the processor that executes the reduction assignment (processor A) and the processor that owns the target location (processor B). Processor A, which executes the reduction, sends processor B a message containing three items: the right-hand-side value, the combining function descriptor, and the target location pointer. At the end of the parallel phase, Processor B collects incoming reduction messages, combines colliding values and updates target locations. To implement update reductions (Section 1.4.4), processor B replaces the combining function with an update function.

The "owner-updates" model is simple to implement and requires minimal runtime system support. It depends on the runtime system to support target location queries, which is usually available in languages that provide a global name space.

Bulk Reductions

During a data-parallel operation, a processor may execute multiple reductions for two reasons. First, the number of elements in a collection is typically much larger than the number of processors, so each processor executes multiple invocations. Second, each invocation may itself execute

Program	Scientific Domain	Compared against	C**	SPMD	(SPMD/C**)
DSMC	Particle-in-cell	Hybrid SM-MP	74.2	82.7	0.90
EM3D	Electromagnetics	Hybrid SM-MP	10.7	5.0	2.14
Water	Molecular dynamics	Shared memory (SM)	13.0	13.6	0.96
Moldyn	Molecular dynamics	Hybrid SM-MP	27.0	26.7	1.01
FFT	Signal processing	Shared memory (SM)	2.0	8.5	0.11

Table 1.1: Comparative benchmark execution times (in seconds on 32-processor CM-5)

multiple reduction assignments. The deferred reduction model allows the compiler to defer sending reduction messages until the end of the parallel phase. This permits several messages to the same destination processor to be bundled into a single message, which is typically far more efficient to send and receive.

Local Combining

Local combining further enables the compiler to reduce the amount of data communicated in messages. If different invocations perform reductions on the same target, values can be combined locally, before global combining. C**'s run-time system uses a per-processor hash table to track common targets. Probing this table increases the overhead of the reduction, but allows for a decrease in communications cost. This is a good example of an optimization that trades off worse sequential performance for better communication (and therefore parallel) performance.

1.4.6 User-defined Reduction Performance

We evaluated our reduction implementation by comparing C** versions of five benchmarks (one small and four medium-size applications) against highly optimized hand-coded SPMD versions of the programs on a 32 processor CM-5. All of these programs' communication patterns were fully captured by reduction assignments (both user-defined and primitive). Table 1.1 summarizes our results. In the worst case, the C** version was 2x slower than the message-passing EM3D code. It was between 1% slower and 10% faster on DSMC, Moldyn and Water, and 4.25x faster on the communication intensive FFT code. Given the complexity and effort in tuning the SPMD codes, the C** programs are far more attractive.

1.5 C** Implementation[7]

Efficiently compiling parallel languages for parallel computers is difficult. Most languages assume a shared address space in which any part of a computation can reference any data. Parallel machines provide either too little or too much support for many languages [15]. On one hand, message-passing machines require a compiler to statically analyze and handle all details of data placement and access, or pay a large cost to defer decisions to run time. On the other hand, shared-memory machines provide more dynamic mechanisms, but generally use them to implement a fixed cache-coherence policy that may not meet a language's needs.

In C**, the compiler exploits control over the memory system of a parallel computer to construct a language-specific address space for a high-level parallel language. Because the semantics of memory match the semantics of a language, a compiler can generate efficient code, with assurance that the memory system will detect unusual cases and errors so that a run-time system can handle them.

The hardware base is a parallel computer with a Tempest-like interface, which provides mechanisms that permit user-level software to implement shared-memory policies [13, 23]. A Tempest memory system is possible on a wide range of parallel systems, including those without shared-memory hardware [27]. Tempest offers a program control over both communications and data placement, as is possible with message passing, and the dynamic fine-grain policies possible with shared memory.

Reconcilable Shared Memory (RSM) provides a global address space and basic coherence policy whose two key policies governing memory system behavior are under program control. The first is the system's response when a processor requests a copy of a cache block. The second is the system's response when a processor returns a cache block in response to a request. Unlike most shared-memory systems, RSM places no restrictions on multiple outstanding writable copies of a block and permits non-sequentially consistent memory models. A language-specific coherence protocol uses RSM mechanisms to support a language's semantics directly. Custom coherence policies can also improve the performance of shared-memory programs written in any language. For example, global reductions and stale data fit naturally into the RSM model. Finally, RSM can help detect unsynchronized data accesses (data races).

[7] This section is a shorter version of the full LCM paper [16].

1.5.1 LCM

RSM systems can aid the implementation of parallel programming languages, particularly higher-level languages such as C**. A natural way to implement C**'s semantics is a copy-on-write scheme, in which each parallel invocation obtains and modifies its own copy of shared data. We implemented this policy in an RSM system called Loosely Coherent Memory (LCM). LCM and the C** compiler cooperate to detect the need for shared data and to copy it, instead of the conventional approach in which a compiler generates conservative code to copy shared data. LCM's copies, although they share the address of the original, are private to a processor and remain inconsistent until a global reconciliation returns memory to a consistent state.

RSM offers several advantages over explicit copying. A compiler can produce code optimized for cases in which no copying is necessary; these predominate in many programs. Compiler-produced copying code is conservative and incurs unnecessary overhead either by copying too much data or by testing to avoid unnecessary copying. The LCM copy-on-write scheme defers copying until a location is actually accessed, which reduces the quantity of data that must be copied. The Myrias machine [**?**] implemented, in hardware, a similar copy-on-write mechanism for parallel DO loops. It, however, used a fixed reconciliation policy and copied entire hardware pages.

A compiler's control of LCM permits optimizations when analysis is possible. For example, not all modifications to shared data need cause a copy. Only items which might be shared between processes must be copied. If compiler analysis determines that no other process will access a location, it need not be copied, which avoids the overhead of making and reconciling a copy. However, this approach requires close cooperation between the compiler and memory system to select—at a fine grain—policies governing portions of data structures.

In C**, computation alternates between parallel and sequential phases. Memory becomes coherent at the end of a parallel phase as processors reconcile their modified memory locations. C**'s semantics dictate how copies are reconciled. In general, C** requires only that the coherent value left in a memory location modified by a parallel function call be a value produced by some invocation of the call. LCM discards all but one of the modified copies. However, values written by C**'s reduction assignments require different reconciliation functions that combine values.

1.5.2 LCM Implementation

We built an LCM system on the Blizzard implementation of Tempest. Blizzard is a fine-grain distributed shared memory system that runs at near shared-memory hardware speeds on a Thinking Machines CM-5 [27]. We compared the performance of four C** programs running under both the unmodified Stache protocol [23] and LCM (implemented using the Tempest mechanisms provided by Blizzard). We found that the LCM memory system improved performance by up to a factor of 4 for applications that used dynamic data structures. LCM's performance was slightly slower than transparent shared memory for applications with static data and sharing patterns, which a compiler can analyze and optimize directly without using LCM.

LCM is a user-level Tempest protocol that runs on a CM-5. We started with the user-level Stache protocol [23], which provides cache-coherent shared memory and uses a processor's local memory as a large, fully associative cache. This cache is essential to ensure that a processor's locally modified (inconsistent) blocks are not lost by being flushed to their home node. When a modified cache block is selected for replacement (either because of a capacity or conflict miss), the block is moved to the Stache in local memory. On a cache miss to the block, its value comes from the Stache, rather than the block's home processor.

LCM provides the C** compiler with three directives. The first of these, `mark_modification(addr)`, creates an inconsistent, writable copy of the cache block containing `addr`. If the block is not already in the processor's cache, it is brought in. The second, `reconcile_copies()` appears as a global barrier executed by every processor. When it finishes and releases the processors, the memory has been reconciled across all processors and is again in a coherent state. This directive flushes all modified blocks back to their home processors to be reconciled. Outstanding read-only copies of these blocks are then invalidated throughout the system. The third, `flush_copies()`, performs a partial reconciliation by flushing a processor's modified cache blocks back to their home processors. The next section illustrates how the C** compiler uses these directives.

C** parallel function invocations start execution with the original (pre-parallel call) global state. LCM retains an unmodified copy of global data throughout a parallel call. At the first write to a cache block managed by the copy-on-write policy, the block's home node creates a *clean copy* of the block in main memory. The node uses a clean copy to satisfy subsequent requests for unmodified global data.

Another complication is that each processor typically runs many distinct invocations of a parallel function. The system must ensure that a new invocation does not access local cache blocks modified by a previous invocation. To avoid this error, LCM's `flush_copies()` directive removes modified copies of global data from the Stache. If a compiler cannot ensure that invocations access distinct locations, it issues this directive between invocations. This directive flushes cache blocks to their home processor, where they are reconciled. A subsequent read of one of these blocks returns its original value from the clean copy. Cache flushing, although semantically correct, performs poorly for applications that re-use data in flushed blocks. In another approach, each processor keeps a clean copy of every block it modifies. In this case, the `flush_copies` directive returns modified values to their home node and replaces the cached value with the clean copy, so it remains local for a subsequent reference.

LCM's memory usage depends on the number of potentially modified locations. At a location's first `mark_modification` directive, LCM creates a clean copy in memory. Cached copies resulting from this directive require slightly more state information than ordinary cached blocks. Clean copies exist only during a parallel function call and are reclaimed at the `reconcile_copies()` directive.

1.5.3 Compiling C**

Compiling a C** program to run under LCM is straightforward. To ensure the correct semantics for parallel functions, the C** compiler inserts memory system directives, described above, in parallel functions. Alternatively, the compiler could guarantee these semantics with run-time code that explicitly copies data potentially modified in a parallel function invocation. Explicit copying works well for functions with static and analyzable data access patterns. However, it becomes complicated and expensive for programs with dynamic behavior, since the generated code must either perform run-time checks or copy a conservative superset of the modified locations. This section illustrates both approaches with a static parallel function (the stencil function) and a dynamic parallel function (an adaptive mesh) and compares the performance of LCM against the explicit copying strategy.

Stencil Example

As a first example, consider a simplified version of the code generated by the C** compiler for a `stencil` function to run under LCM:

```
void stencil_SPMD(matrix &A)
{
  for( all invocations assigned to me)
  {
```

```
    set variables #0 and #1;

    // Function body:
    mark_modification(A[#0][#1]); // LCM directive

    A[#0][#1] = (A[#0-1][#1] + A[#0+1][#1] + A[#0][y-1] + A[#0][#1+1]) / 4.0;
    flush_copies();               // LCM directive
  }
  reconcile_copies();             // LCM directive
}
```

Each invocation writes to `A[#0][#1]`, which is also read by its four neighboring invocations. Compiler analysis easily detects this potential conflict, which the C** compiler rectifies with `mark_modification` directives. The `flush_copies` directive removes modified copies from a processor's cache before another invocation starts. The `reconcile_copies` directive causes the memory system to reconcile modified locations and update global state to a consistent value.

Because compiler analysis reveals that `stencil` accesses the entire array, the C** compiler could also preserve C** semantics by maintaining two copies of `A`—all reads come from the old copy of `A` and all writes go to the new copy of `A`. After each iteration, the code exchanges the two arrays with a pointer swap. This simple technique preserves the C** semantics with little overhead beyond the cost of twice the memory and cache usage.

1.5.4 Dynamic C** Program

LCM offers greater benefits for programs with dynamic behavior that is difficult or impossible to analyze. These programs require extensive (and expensive) run-time operations to run in parallel [26]. For example, consider an adaptive mesh version of `stencil`, which selectively subdivides some mesh points into finer detail. It is part of a program that computes electric potentials in a box. The program imposes a mesh over the box and computes the potential at each point by averaging its four neighbors. At points where the gradient is steep, finer detail is necessary and the program subdivides the cell into four child cells. This process iterates until the mesh relaxes. Initially, points on the mesh are represented in a two-dimensional matrix, but dynamically allocated quad-trees capture cell subdivision:

```
// Update my quad-tree in the mesh
double Mesh::update_mesh() parallel
{
  // What part of tree changed?
  *self = update_quad_tree(self, neighbors);
```

```
  // Return maximum of local values
  return %> local_epsilon;
}

// Main program - do the iterations
main()
{
  ⋮
  create_mesh();
  while (difference >= epsilon)
    difference = update_mesh();
  ⋮
}
```

In this program, the mesh changes dynamically so a compiler cannot determine which parts will be modified. Without an LCM system, a compiler must conservatively copy the entire mesh between iterations to ensure C**'s semantics. With LCM, the memory system detects modifications and copies only data that is modified.

1.6 Polygon Overlay Example

As in the other chapters in this book, we illustrate our language with a C** program that computes polygon overlays. This problem starts with two maps, A and B, each covering the same geographic area and each composed of a collection of non-overlapping polygons. This calculation computes the intersection of the two maps by computing the geometric intersection of polygons from each map. As described in Chapter **??**, we assume that polygons are non-empty rectangles and that the entire collection fits in memory.

This section outlines two implementations of the polygon overlay calculation in C**. The first is simple and inefficient (Section 1.6.1), but fits the data-parallel style well. However, as in life, style is no substitute for thought, and the second version uses an auxiliary data structure to greatly reduce the cost of computation (Section 1.6.2).

1.6.1 Naïve C** Implementation

The naïve program directly applies data parallelism to the problem. Each polygon in one map executes a data-parallel operation that computes its intersection with every polygon in the second map. The non-empty intersections form the result of the computation. This method is simple, but extremely inefficient, since most intersections are empty.

```
struct poly_s {                  //  a polygon
    short xl, yl;                //  low corner
    short xh, yh;                //  high corner
};

struct polyVec (poly_s) []       //  polygon Aggregate
{
  ··· member functions omitted···
};

polyVec *leftVec, *rightVec;     //  input vectors
```

Program 1.6: C declarations for naïve algorithm.**

Program 1.6 shows the relevant C** declarations for this program. Each polygon is represented by the coordinates of its lower left and upper right corners. The Aggregate class, `polyVec`, holds polygons from an input map.

The parallel C** function `polyVec::computeVecVecOverlay` (in Program 1.7) computes the intersection of polygon `self` (which, analogous to `this`, points to the polygon a invocation is responsible for) with the second vector of polygons `vec`. Each non-empty intersection is added to a local list `theList`, and independent local lists are combined with the user-defined reduction `merge` in Program 1.7. The data-parallel overlay operation is applied to the first vector of polygons by the statement:

```
  results = (leftVec->computeVecVecOverlay(rightVec)).head;
```

```
polyList_s polyVec::computeVecVecOverlay(polyVec *vec) parallel
{
    polyList_s theList = {NULL, NULL};      //  ptrs to head and tail

    for (int i=0; i<vec->cardinality(0); i++) {
        polyNode_p tmp = polyOverlay(self, &((*vec)[i]));
        if (tmp != NULL)
            theList.insert(tmp);
    }
    return %merge theList : nullList;       //  user-defined reduction
}

void merge(polyList_s *result, polyList_s theList)
{
    if (result->head==NULL && result->tail==NULL) { //  Is result NULL?
```

```
        *result = theList;
    } else if (!(theList.head==NULL && theList.tail==NULL)) {
        result->tail->next = theList.head;
        result->tail = theList.tail;
    }
}
```

Program 1.7: C code for naïve overlay**

For efficiency reasons, the data-parallel operation in Program 1.7 returns a structure containing pointers to the head and tail of its list of polygons. The `merge` routine destructively concatenates two lists by changing the tail of the first to point to the head of the second one.

1.6.2 Grid Partitioning a Map

We greatly improved the performance of the computation by exploiting locality—both geographic, in the problem, and spatial and temporal, in the computer. Instead of comparing every polygon against every other polygon, the revised program compares a polygon against the far smaller collection of polygons that are spatially adjacent. This program *partitions* the space in the second polygon map into a rectilinear grid and uses this grid to reduce the number of polygons that must be examined. This change requires a new, two-dimensional, `partition` class (in Program 1.8) to maintain the decomposed polygon map. Each cell in the partition contains a list of polygons that are partially or entirely within the cell. The second list in a cell is used in the double partition approach (Section 1.6.2).

```
struct polyNode_s {              // polygon list cell
    poly_s      poly;             // the polygon
    polyNode_s *next;             // link to next
};
typedef polyNode_s *polyNode_p;

struct partition_s {
    polyNode_p  lists[2];         // pair of lists
};

struct partition(partition_s) [][]
{
  ··· other member functions omitted···
};
```

Program 1.8: Declarations for partition algorithm.

The partitioning approach requires a new overlay routine (Program 1.9). It only needs to compare a polygon against polygons in the partition cells that it overlaps. These cells can be quickly identified from the two endpoints that define the polygon. A polygon overlaps all partition cells between the partition that contains its lower left and upper right corners.

With the naïve program, computing the overlay of two datasets containing approximately 60K polygons each resulted in over 3.6 billion polygon comparisons. By contrast, the partitioning version, using a partition of 45 by 45 cells, required only 3.6 million comparisons—an improvement of three orders of magnitude.

```
#define ownPoly(x,y,p)\
        ((findCell(p->poly.xl)==x) && (findCell(p->poly.yl)==y))

#define findCell(x)  ((int)(((x)-1) / (cellSize)))

polyList_s polyVec::computeVecPartOverlay(partition *p) parallel
{
    polyList_s theList = {NULL, NULL};

    int xStart = findCell(self->xl);     // find appropriate cells
    int xStop  = findCell(self->xh);
    int yStart = findCell(self->yl);
    int yStop  = findCell(self->yh);

    for (int x=xStart; x<=xStop; x++)    // step through cells
        for (int y=yStart; y<=yStop; y++)
            for (polyNode_p ptr=((*p)[x][y].lists[0]); ptr!=NULL; ptr=ptr->next) {
                polyNode_p tmp = polyOverlay(self, &(ptr->poly));
                if ((tmp != NULL) && (ownPoly(x,y,tmp)))
                    theList.insert(tmp); // link in overlap
                else if (tmp != NULL)
                    delete tmp;          // not "owned", delete
            }
    return %merge theList : nullList;
}
```

Program 1.9: Overlay Routine for Partition Algorithm.

Since a pair of polygons may overlap in several partitions, the code must be careful to avoid recording duplicate intersections. The C** program, in the routine `ownPoly`, records the intersection of two polygons only when the lower corner of the overlap falls within the current partition cell.

The distribution of polygons in partition cells affects load balancing and hence the program's performance. We use a simple heuristic to partition the polygons. The program first calculates the number of cells in each partition from the area of an input polygon map and the number of polygons it contains. The program then computes the average polygon area and sets the partition cell size to some multiple of the average polygon area, called the *granularity*. More will be said about choices of granularity in Section 1.6.4.

The code in Program 1.10 partitions the Aggregate of polygons. The data parallel function invocation on a polygon copies the polygon into the appropriate partition cells. Since this process is many-to-many communication, with the potential for write conflicts, a user-defined reduction (`insertPoly`) links polygons into a partition's cell list.

```
void insertPoly(polyNode_p *result, poly_s thePoly)
{
    polyNode_p ptr = new polyNode_s;//  allocate new node
    ptr->poly = thePoly;            //  fill it in
    ptr->next = *result;            //  link node into list
    *result = ptr;                  //  return result
}

void polyVec::partitionVec(partition *p, int n) parallel
{
    int xStart = findCell(self->xl);  //  find dest cells
    int xStop  = findCell(self->xh);
    int yStart = findCell(self->yl);
    int yStop  = findCell(self->yh);

    for (int x=xStart; x<=xStop; x++) //  do combining writes
        for (int y=yStart; y<=yStop; y++)
            &((*p)[x][y].lists[n]) =%insertPoly *self;
}
```

Program 1.10: Code to Partition a Polygon Vector.

Note that the reduction assignment passes a polygon structure, not a pointer to a polygon. This is for efficiency. Passing a pointer as an argument to the user-defined reduction operation means that `insertPoly` must dereference the pointer to obtain a copy of the polygon to link into the list. This dereference requires communication if the reduction occurs on a processor other than the one that allocated the polygon. Passing polygons, instead of pointers, causes the polygon data to be sent directly to the processor that performs the reduction and eliminates a potential extra round of communication.

The same reasoning, in reverse, applies to the overlay routine in Program 1.7. The solution is formed by a reduction that references pointers to the head and tail of each invocation's polygons list. Thus, only a pair of pointers are passed between processors. If the code used a combining assignment, similar to the one in Program 1.10, it would transmit all polygons in the list, instead of just the list's head and tail. Of course, sending the polygons might be more efficient if the processor that invoked the data-parallel operation later read the entire result.

Partitioning Both Maps

A natural extension of the partitioning approach is to partition *both* polygon vectors and overlay the partitions cell by cell. Although this double partitioning approach performs the same number of comparisons as the single partition version, it exploits memory locality more effectively. Each grid cell now contains two lists of polygons, one list for each map. Since both of a cell's polygon lists are allocated on the same processor, the polygon comparison phase requires no communication. Program 1.11 lists the parallel overlay function.

```
polyList_s  partition::computePartPartOverlay() parallel
{
    polyList_s theList = {NULL, NULL};

    for (polyNode_p p1=self->lists[0]; p1!=NULL; p1=p1->next) {
        for (polyNode_p p2=self->lists[1]; p2!=NULL; p2=p2->next) {
            polyNode_p tmp = polyOverlay(&(p1->poly), &(p2->poly));
            if ((tmp != NULL) && (ownPoly(#0,#1,tmp)))
                theList.insert(tmp);
            else if (tmp != NULL)
                delete tmp;
     }
  }
  return %merge theList : nullList;
}
```

Program 1.11: Overlay Routine for Double Partition Version.

Although the extra partitioning step increases execution time for small numbers of processors, the double partition code has better locality, and therefore incurs fewer misses, which increases scalability. For example, a run with 60K polygon datasets on 32 processors resulted in 2151 coherence misses during the overlay phase of the single partition code. The double partition overlay phase incurred only 46 misses. Section 1.6.4 shows that this change results in more nearly linear scaled speedups for the double partition version.

1.6.3 Performance Tuning

The initial version of the partitioned code ran reasonably well, but several changes improved both the speed of the C** code and its scalability. Although C** is a high-level language, we were able to perform these optimizations within the language.

Memory Management

The program creates and destroys many polygon list nodes during the partitioning and overlay phases. Measurements showed that it spent considerable time in calls to `malloc` and `free`. Furthermore, malloc's 8-byte memory overhead for each allocation greatly increased the total memory requirement. We therefore implemented our own list of free polygon list node on each processor. When the list is exhausted, the program requests four pages of memory (16K bytes) and carves it into pieces of the appropriate size.

Communication Reduction

In order to obtain good speedups on a distributed shared memory machine such as the CM-5, it is important to reduce communication when possible. This section describes three optimizations that decrease communication.

The first improvement changed polygon coordinates to `shorts`, instead of `ints`. This cut both the memory and communication requirements in half. Also, padding the partition cells to 32 bytes reduced false sharing.

Since polygon maps are accessed sequentially, our second optimization uses a large unit of coherence to exploit temporal locality. Our memory coherence protocol, implemented in software, allows us to maintain coherence on larger 1024-byte blocks, rather than the default 32-byte blocks [23]. This change reduced the number of cache faults on shared data significantly. For example, on datasets containing approximately 60K polygons each, and running on 32 processors, the number of cache misses dropped from 60,876 to 4,962.

The final optimization further reduced communication in the double partition code by distributing the grid data and grid computation identically. The shared-memory substrate distributes global data using round-robin page placement, while C**'s run-time system divides invocations in equal blocks. By adjusting the number of partition cells to occupy a power-of-two number of pages,this optimization ensures that a processor accesses exactly those grid cells that are allocated to it. Thus, the entire overlay computation requires no communication other than the reduction to combine pieces of the solution.

Area Optimizations

Both the single and double partitioning approaches can benefit from an area-based optimization. Once a polygon's entire area has overlapped other polygons, it can be removed from consideration. Discarding polygons reduces the number of comparisons and therefore the cost of calculating the intersection.

For the single partition version, each polygon from the first map records its unused area. When the area is consumed, the polygon is discarded. A more aggressive approach could record unused areas of polygons in the second map as well. The second optimization is inappropriate for the single-partition version since it requires changes from one invocation to be visible to all others. The results in the next section only use the simple optimization for the single-partition code.

The double-partition approach requires more complex analysis since a polygon in the first map cannot be discarded unless it is completely enclosed in a partition cell. However, a slightly more complex area calculation, which starts with the portion of a polygon enclosed by a cell, works well. Furthermore, the double-partition version can exploit the second optimization because both lists of polygons within a cell are accessed by the same parallel function invocation. Unfortunately, measurements showed that the cost of building and maintaining a second data structure to record polygon areas outweighed the benefits of the more aggressive optimization. As the granularity of a partition increases, the optimization becomes more attractive. However, our measurements showed that it was better to use smaller granularities and the first area optimization than larger granularities and both optimizations. Thus, the results for the double-partition code use only the first area optimization.

1.6.4 Results

All measurements were run on a 32-processor partition of a Thinking Machines CM-5. Our timings do not include the time to read the data from disk. On the CM-5, the time to read the input files is larger than the time to actually perform the computation. For example, on the 60K polygon datasets and 32 processors, it takes more than 20 times longer to read the data than to compute the overlay. Since the CM-5 does not support parallel I/O, the input files are read by a front-end process and sent to a single CM-5 processor, which creates a serial bottleneck that would artificially limit the program's speedup.

For the same reason, we also did not include the time to distribute the input data across the processors. Since all data initially resides on the processor that read the file, distributing the data takes a large amount of time and forms a bottleneck. On 32 processors, the data distribution time is roughly equal to the time for two partitioning steps and the overlay computation. For reference, the graph in Figure 1.7 shows how including the distribution time would reduce the application's speedup. This bottleneck is clearly an area for future work.

Granularity

Selecting a granularity for partition cells (Section 1.6.2), as typical, requires tradeoffs. At first glance, small granularities appear best since they require fewer comparisons during the overlay phase. However, as the granularity decreases, polygons span partition cells more frequently. This increases memory requirements, since each of these polygons is duplicated. Partitioning with a granularity of 5 doubles the number of polygons that must be represented. Even with a granularity of 20, approximately 50% more polygons must be represented.

The granularity also affects the program's speedup. The partitioning phase runs best with large granularities, since few cell-spanning polygons require less communication. However, the overlay computation performs less work with small granularities. The best granularity is therefore a balance between the partition and overlay phases' needs. This balance is different for the single and double partition approaches, since the double partition code spends more time partitioning.

As was mentioned in Section 1.6.3, we adjusted the number of cells in the partition to ensure that the memory underlying a partition is power-of-two number of pages. This limits the choice of granularity, but some latitude still remains. In the experiments below, we used a granularity of 30 for the smallest dataset and a granularity of 36 for the other two datasets. These values were compromises that produced consistently high performance for varying numbers of processors.

The k100 dataset

The k100 series of polygon maps were the largest ones provided. Each contains approximately 60,000 polygons. Figure 1.2 shows the execution

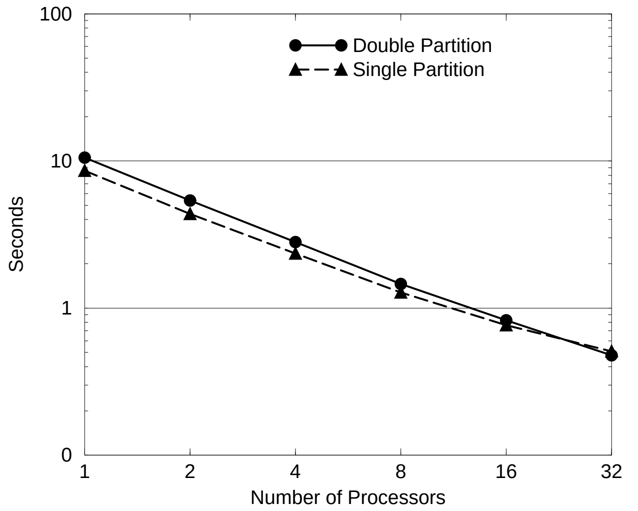

Figure 1.2: Execution times of the overlay phase for the k100 data set (partition granularity 30).

times to compute the overlay for the first two datasets of this series. The runtime of the naïve program is not shown, since it is far larger than the partitioning versions. For example, on one processor, it requires more than 4500 seconds for the naïve computation, versus roughly 10 seconds for the partitioned program. Figure 1.3 shows the scaled speedups (speedups with respect to one-processor runs) for all three versions.

The naïve code has the best scaled speedup: 25.8 on 32 processors (an efficiency of 80.6%). The partitioning versions have scaled speedups of 16.0 and 20.7, for efficiencies of 52.0% and 64.7%. Detailed analysis of the partitioning versions showed that load imbalances reduced the speedup. The average time spent idle due to load imbalance is roughly constant with respect to the number of processors. On one processor, it amounts to just over 1% of the computation time. At 32 processors it makes up about 25%. A good portion of the problem is that the total computation time on 32 processors is only half of a second, which exacerbates the load imbalance.

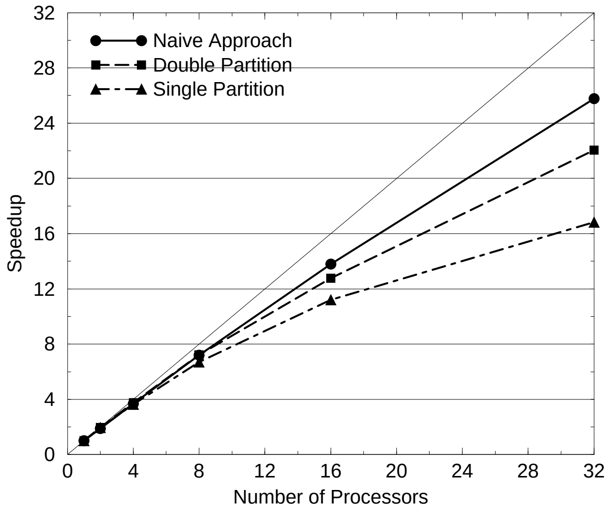

Figure 1.3: Scaled speedups of the overlay phase for the k100 data set (partition granularity 30).

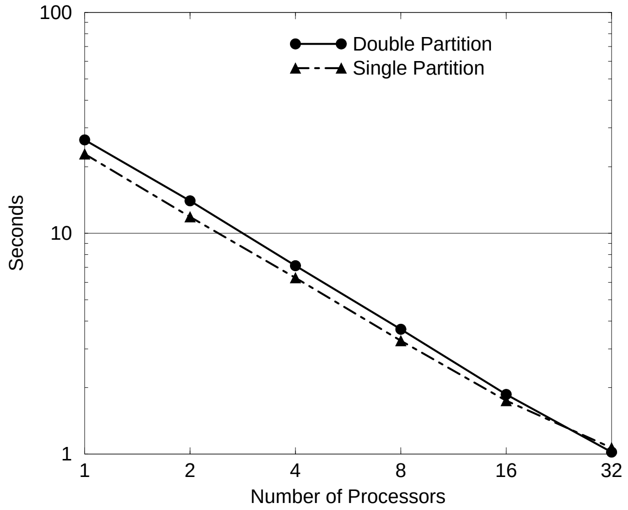

Figure 1.4: Execution times of the overlay phase for the 150K data set (partition granularity 36).

Larger Datasets

To study the program's performance, we generated a new dataset with roughly 150,000 polygons per input file. Datasets of this size used nearly all memory on a single processor. Figure 1.4 shows the execution times and Figure 1.5 the scaled speedups for the overlay phase. All three versions of the code produced better scaled speedups on the larger dataset, with double partitioning reaching a scaled speedup of 25.7, for an efficiency of 80.4%. The naïve code reached 28.6 for an efficiency of 89.4%.

To obtain better scaled speedups, we generated a dataset with approximately 300,000 polygons per input file. This problem could not run on one processor due to memory limitations, so we measured performance on 2–32 processors. Figure 1.6 shows the execution times and Figure 1.7 the scaled speedups. The double partition code achieved a scaled speedup of 13.6, for an efficiency of 85.0%. We could not measure the performance of the naïve code for this dataset because it took too long to run (over 20 hours on 2 processors). Figure 1.7 also shows scaled speedups for the partition and overlay phases.

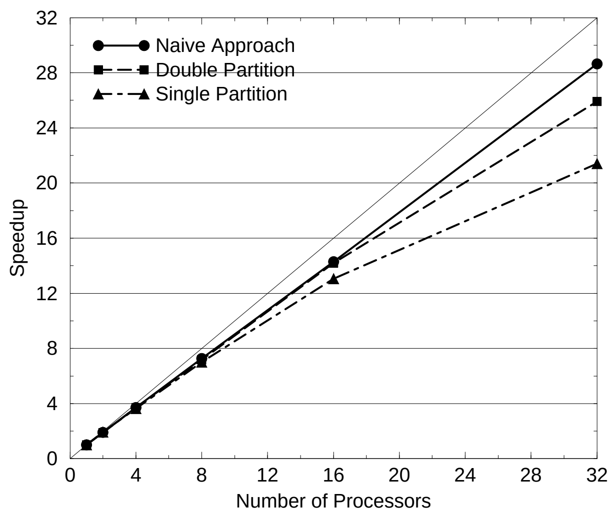

Figure 1.5: Scaled speedups of the overlay phase for the 150K data set (partition granularity 36).

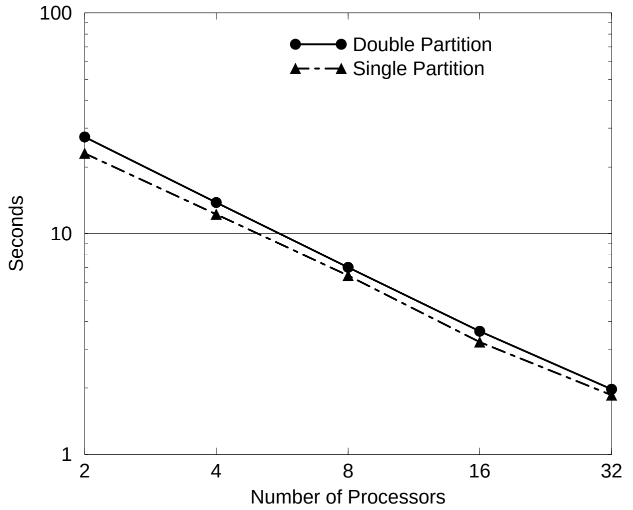

Figure 1.6: Execution times for the 300K data set (partition granularity 36).

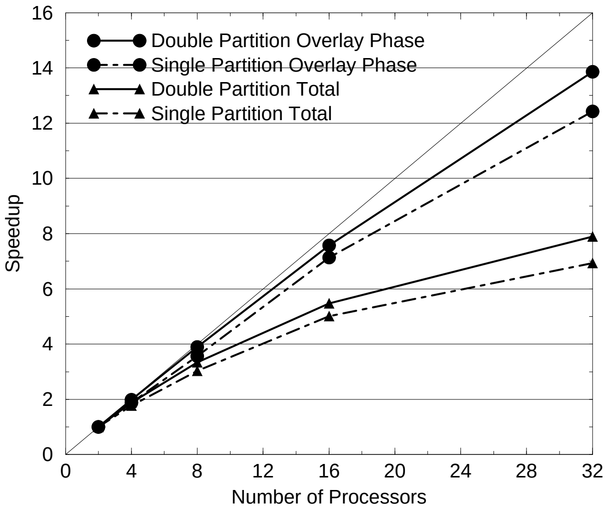

Figure 1.7: Scaled speedups for the 300K data set (partition granularity 36).

1.7 Conclusion

Designing a new programming language is in many ways like writing the Great American Novel—except that the rewards, both on the average and at the margin, are more lucrative for authors than language designers. Both occupations require overwhelming confidence that your wonderful new idea will succeed and prosper where the vast majority of your predecessors sank into oblivion.

C**'s conceit was that a parallel language could offer some benefits of SIMD programming, without some of its disadvantages, by restricting parallel execution semantics. C**research has focused on techniques for efficiently implementing these semantics and language extensions for increasing the semantics' generality. In both respects, the research is successful. By exploiting Tempest user-level shared memory, the C**compiler and system implement a large-grain data-parallel language with a simple, clear semantics and little unnecessary overhead. Although the many polygon overlay examples are difficult to compare because of algorithm and processor differences, C**'s speedup curves are close to linear. In addition, our experience with C**demonstrates that user-level reductions are a powerful language feature that permits concise, efficient implementation of many parallel algorithms, even within the constraints of large-grain data parallelism.

The course that we followed is, unfortunately, unlikely to lead to wide popularity and use of C**. The recent languages on the best seller lists—C, C++, Perl, TCL—are an amalgamation of ideas from earlier research languages and all began with a low-cost, widely-available implementation. pC++(Chapter **??**) is consciously following this model. It is unclear, however, if this model will succeed for high-performance computing since portability is often the enemy of performance. Seeing little hope of writing a best seller, our efforts followed a different approach and concentrated on implementing, developing, and demonstrating a few ideas.

During this research, several more general observations have become clear:

- The programming language community is too introspective. In recent years, it has begun a dialog with the applications community. However, it still does not interact with computer architects, to understand and influence future machines. Machines are still designed and built to fit perceived needs of users, without much consideration to the requirements of languages or compilers [**?**]. In the end, the users, compiler writers, and architects all suffer.

  The C**research has been conducted as part of the Wisconsin Wind Tunnel Project [**?**], which is investigating the hardware/software tradeoffs in parallel shared memory machines. A key design tenet of the Wind Tunnel research is that hardware should provide mechanisms and software should implement policy. C**has both exploited this approach by using the Tempest mechanisms to implement its language semantics in the memory system. Hardware provides low-costs tests that detect exceptional conditions without slowing normal execution.

- Following a moving target is hard. Four years ago, we introduced parallelism into C**by extending the class mechanism. Today, we would likely specify Aggregates with Templates (like the Amelia Vector Template Library Chapter **??**). When we started, Templates were a new language feature that compilers partially and poorly implemented. Today, Templates are a widely used and important language feature that most compilers still implement poorly.

  Our primary problem was that C++ is a very complex language and the continual standardization process increased the number and complexity of its features. As a consequence, C++ compilers are complex, poorly written, and constantly changing. Adding our minor changes to C++ required considerable time and effort better spent on research. Templates, when properly implemented, may permit language extension without language modification.

- Too many languages, not enough evaluation; or, what makes a good language? In the absence of objective criteria, everyone will continue using and teaching familiar languages whose compilers are at hand. Without quantitative comparisons against alternative languages and implementations, it will remain impossible to write a programming languages paper that excites more than a small handful of readers. In a better world, new language features (and implementations) would be compared by measures that matter, such as performance, conciseness, and readability.

  This book takes a first step in that direction. By comparing the many implementations of the polygon overlay code, a reader can begin to evaluate the many parallel C++ languages. Of course, all of the usual disclaimers ("its only one program running on an experimental implementation on a slow, old machine") apply if our C**code does not fare well.

Bibliography

[1] Jeanne C. Adams, Walter S. Brainerd, Jeanne T. Martin, Brian T. Smith, and Jerrold L. Wagener. *Fortran 90 Handbook*. McGraw-Hill, 1992.

[2] John W. Backus. Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs. *Communications of the ACM*, 21(8):613–41, August 1978.

[3] David H. Bailey. Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers. *Supercomputing Review*, 4(8), August 1991.

[4] J. E. Barnes and P. Hut. A Hierarchical O(N log N) Force Calculation Algorithm. *Nature*, 324(4):446–9, December 1986.

[5] Guy E. Blelloch. NESL: A Nested Data-Parallel Language. Technical Report CMU-CS-92-103, School of Computer Science, Carnegie-Mellon University, April 1993.

[6] Michael J. Flynn. Very High-Speed Computing Systems. *Proceedings of the IEEE*, 54(12):1901–09, 1966.

[7] Geoffrey C. Fox. What Have We Learnt from Using Real Parallel Machines to Solve Real Problems. In Geoffrey C. Fox, editor, *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, volume 2, pages 897–955, January 1988.

[8] Philip J. Hatcher and Michael J. Quinn. *Data-Parallel Programming on MIMD Computers*. MIT Press, 1991.

[9] Mark D. Hill, James R. Larus, and David A. Wood. Tempest: A Substrate for Portable Parallel Programs. In *COMPCON '95*,

pages 327–32, San Francisco, CA, March 1995. IEEE Computer Society.

[10] W. Daniel Hillis. *The Connection Machine.* MIT Press, 1985.

[11] W. Daniel Hillis and Guy L. Steele, Jr. Data Parallel Algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[12] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran Handbook.* MIT Press, 1994.

[13] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating Message-Passing and Shared-Memory: Early Experience. In *Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 54–63, May 1993.

[14] James R. Larus. C**: a Large-Grain, Object-Oriented, Data-Parallel Programming Language. In Utpal Banerjee, David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages And Compilers for Parallel Computing (5th International Workshop)*, pages 326–41. Springer-Verlag, August 1993.

[15] James R. Larus. Compiling for Shared-Memory and Message-Passing Computers. *ACM Letters on Programming Languages and Systems*, 2(1-4):165–80, March-December 1994.

[16] James R. Larus, Brad Richards, and Guhan Viswanathan. LCM: Memory System Support for Parallel Language Implementation. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 208–18, October 1994.

[17] J. K. Lee and D. Gannon. Object Oriented Parallel Programming: Experiments and Results. In *Proceedings of Supercomputing '91.* IEEE Computer Society and ACM SIGARCH, 1991.

[18] Calvin Lin and Lawrence Snyder. ZPL: An Array Sublanguage. In *Languages and Compilers for Parallel Computing (Proceedings of the Sixth International Workshop)*, pages 96–114. Springer-Verlag, 1994.

[19] John M. Lucassen and David K. Gifford. Polymorphic Effect Systems. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 47–57, January 1988.

[20] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, May 1994.

[21] Shubhendu S. Mukherjee, Shamik D. Sharma, Mark D. Hill, James R. Larus, Anne Rogers, and Joel Saltz. Efficient Support for Irregular Applications on Distributed-Memory Machines. In *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, 1995.

[22] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall International (UK), 1987.

[23] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–337, April 1994.

[24] John R. Rose and Guy L. Steele Jr. C*: An Extended C Language for Data Parallel Programming. In *Proceedings of the Second International Conference on Supercomputing*, volume III, pages 2–16, May 1987.

[25] Gary W. Sabot. *The Paralation Model: Architecture-Independent Parallel Programming.* MIT Press, 1988.

[26] Joel H. Saltz, Ravi Mirchandaney, and Kay Crowley. Run-Time Parallelization and Scheduling of Loops. *IEEE Transactions on Computers*, 40(5):603–12, May 1991.

[27] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 297–307, October 1994.

[28] Bradley K. Seevers, Michael J. Quinn, and Philip J. Hatcher. A Parallel Programming Environment Supporting Multiple Data-Parallel Modules. *International Journal of Parallel Programming*, 21(5), 1992.

[29] Shamik D. Sharma, Ravi Ponnusamy, Bongki Moon, Yuan-Shin Hwang, Raja Das, and Joel Saltz. Run-time and Compile-time Support for Adaptive Irregular Problems. In *Proceedings of Supercomputing '94*, pages 97–106, November 1994.

[30] Guy L. Steele Jr. and W. Daniel Hillis. Connection Machine LISP: Fine-Grained Parallel Symbolic Processing. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 279–97, August 1986.

[31] David A. Wood, Satish Chandra, Babak Falsafi, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, Shubhendu S. Mukherjee, Subbarao Palacharla, and Steven K. Reinhardt. Mechanisms for cooperative shared memory. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 156–68, May 1993.

[32] William A. Wulf. Compilers and Computer Architecture. *IEEE Computer*, 14(7):41–7, July 1981.

[33] Bwolen Yang, Jon Webb, James M. Stichnoth, David R. O'Hallaron, and Thomas Gross. Do&Merge: Integrating Parallel Loops and Reductions. In *Languages and Compilers for Parallel Computing (Proceedings of the Sixth International Workshop)*, pages 169–83. Springer Verlag, 1994.

Contents