C**: A Large-Grain, Object-Oriented, Data-Parallel Programming Language

James R. Larus
Brad Richards
Guhan Viswanathan

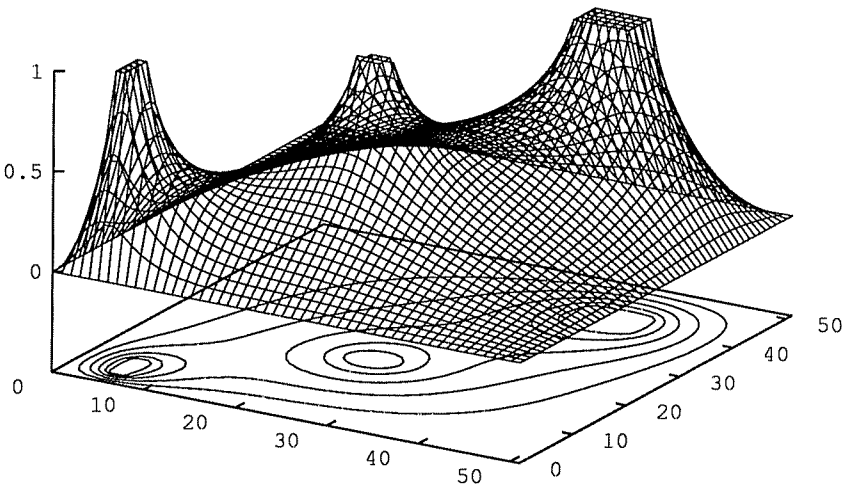
Technical Report #1126

November 1992

C^{**}: A Large-Grain, Object-Oriented, Data-Parallel Programming Language

James R. Larus, Brad Richards, and Guhan Viswanathan[1]
Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton Street
Madison, WI 53706 USA

November 24, 1992



[1]This work was supported by the National Science Foundation under grants CCR-9101035 and CDA-9024618.

Abstract

C^{**} is a new data-parallel programming language based on a new computation model called *large-grain data parallelism*. C^{**} overcomes many disadvantages of existing data-parallel languages, yet retains their distinctive and advantageous programming style and deterministic behavior. This style makes data parallelism well-suited for massively-parallel computation. Large-grain data parallelism enhances data parallelism by permitting a wider range of algorithms to be expressed naturally.

C^{**} is an object-oriented programming language that inherits data abstraction features from C^{++}. Existing scientific programming languages do not provide modern programming facilities such as operator extensibility, abstract datatypes, or object-oriented programming. C^{**} —and its sequential subset C^{++}—support modern programming practices and enable a single language to be used for all parts of large, complex programs and libraries.

This technical report consists of three parts. The body of the report is a copy of a paper that appeared in the 5^{th} *Workshop on Languages and Compilers for Parallel Computing,* Yale University, Aug. 1992 (to appear, *Lecture Notes in Computer Science,* Springer-Verlag). The first appendix is a detailed overview of C^{**}, which describes the language more thoroughly than the paper. The second appendix contains several sample C^{**} programs and some preliminary performance results.

Contents

Chapter 1

C^{**}: A Large-Grain, Object-Oriented, Data-Parallel Programming Language

James R. Larus
larus@cs.wisc.edu
Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton Street
Madison, WI 53706 USA
608-262-9519

Data-parallel programming is not SIMD programming, although this important distinction is rarely made. Lockstep SIMD behavior reflects the underlying hardware and is not an essential attribute of data-parallel programming. C^{**} is a new data-parallel programming language based on a new parallel programming model called large-grain data parallelism. Unlike many data-parallel models, large-grain data parallelism supports both data-parallel operations with independent threads of control and nearly-deterministic execution. In particular, unlike other non-SIMD data-parallel languages, C^{**} retains the SIMD data-parallel languages' advantages of a simple, explicable parallel semantics; machine-independence; and nearly-deterministic execution. In addition, large-grain data parallelism naturally accommodates nested parallelism.

C^{**}, in addition, is an object-oriented language that inherits C^{++}'s data abstraction features, which aid in constructing large systems. Existing scientific programming languages do not support operator extensibility, abstract datatypes, or object-oriented programming. C^{**} —and its sequential subset C^{++}—provide these facilities and enable a single language to be used for all parts of large, complex programs and libraries.

The CM-2 parallel computer is responsible for much of the confusion between data parallelism and SIMD programming. Hillis and Steele convincingly argued that data parallelism—which they defined as "simultaneous operations across large sets of data"—is a widely-applicable programming technique for massively parallel computers, by which they meant SIMD computers like the CM-2 [8]. Many data-parallel languages reflect the quirks of SIMD hardware [1, 12]. Nevertheless, data parallelism has grown beyond its SIMD origins because it offers the appealing advantages of a simple, explicable parallel semantics and nearly-deterministic, race-free

execution.[1] On the other hand, SIMD execution is a major limitation for executing programs that contain conditionals and loops with data-dependent behavior. SIMD execution steps all processors through both arms of a conditional and all loop iterations.

Data-parallel languages have not yet settled on standard terminology. The following terms are used throughout this paper. Wherever possible, the paper relates these terms to others' terminology. An *aggregate* is a collection of *elements*. Aggregates differ from other data structures because data-parallel *operations* can be applied simultaneously to their elements. *Simultaneity* is a logical rather than physical requirement. The computation on each element must appear to occur at the same time, even if physical constraints prevent the computations from executing concurrently.

This work shows that data parallelism and SIMD are not synonymous by describing a newly-implemented language *C*** that preserves the advantages of data parallelism—an ability to reason about program behavior and nearly-deterministic execution—without requiring parallel computations to proceed in lockstep. The rest of this paper primarily discusses existing data-parallel languages (Section 2.1) and *C*** (Section 2.2). It then briefly describes the status of the implementation and future work.

2.1 Related Work

This section critically discusses some widely-known data-parallel languages. By describing them in a common framework, we can compare language characteristics with the following taxonomy:

- Extensibility. Can a programmer define data-parallel functions and apply them to new data types?
- Naming scheme for aggregates. How are aggregates constructed and specified?
- Style of execution. Do data-parallel operators execute with SIMD semantics?
- Data races. What happens if data-parallel operators' data accesses conflict?
- Nested parallelism. Can a data-parallel operator invoke a data-parallel operator?

2.1.1 Fortran 90

Fortran 90 is not a data-parallel language. It is a sequential language with a (new) data-parallel sublanguage for arrays. Nevertheless, because of Fortran's prevalence, this sublanguage is likely to be the data-parallel language encountered by most programmers. Fortran's data-parallel sublanguage is composed of arithmetic operators (including intrinsic elemental functions) and 15 other functions, all of which accept array arguments and produce a scalar or array result. Each function computes a simultaneous, element-by-element result.

User-defined functions can accept array arguments and return array results but cannot specify data-parallel computation. The exception is that a composition of data-parallel operators is data parallel. However, functions containing these expressions cannot have this semantics if they contain control flow or side effects. In addition, data-parallel operators only work for

[1] On the CM-2, when multiple processors write a location, the write that succeeds is indeterminate. Data-parallel languages expose this indeterminacy, which is why their execution is not totally deterministic. This limited indeterminacy can be considered either a feature or a bug, but it clearly reflects an inherent property of interprocessor communication that is difficult to avoid.

arrays of numbers or strings. Fortran 90 also introduces data structures, but they are second class citizens for data parallelism. Since data parallelism is encapsulated in a predefined set of operations that does not include a higher-order apply function, issues of SIMD semantics and nested parallelism do not arise.

2.1.2 C^*

C^* is a data-parallel programming language based on C (with a few ideas from C^{++}) that was originally designed for the SIMD CM-2 parallel computer [12, 11, 14, 15]. The original language described by Rose was C^* version 5. Hatcher and Quinn demonstrated that C^*, despite its origins, could be effectively compiled for shared-memory and message-passing MIMD parallel computers [7]. Their dialect, although called Dataparallel C (DPC), is nearly identical to C^* v5. Thinking Machines subsequently changed C^* into a new language, called C^* version 6 [16], that bears little resemblance to original C^*.

C^*, Version 5

C^* v5 extends the C language with *domains*. A domain declaration, like a C *struct*, specifies the fields in each aggregate element and, optionally, the aggregate's size. A domain also names the aggregate so its elements can be operated on simultaneously. The latter aspect is unusual. Aggregates are not first class values (like arrays). Instead, a data-parallel operation is applied to all instances of a domain with a given name. For example, suppose a programmer wants to write a data-parallel matrix multiplication routine for a program that computes many matrices. Either the programmer must introduce a new domain for every pair of matrices (shades of Pascal!) or must devise a scheme to select pairs of matrices from the undifferentiated pool of matrices with the same domain type.

Data-parallel operations occur either in a domain select statement or a domain member function (the main borrowing from C^{++}). The data-parallel code is arbitrary C that is prohibited only from invoking another data-parallel operation. A *virtual processor* executes the data-parallel code simultaneously—in strict SIMD style—on each element of a domain. This semantics introduces the problems of SIMD execution discussed above, but eliminates data races and allows reasoning about a single thread of execution.

Dataparallel C

Hatcher and Quinn demonstrated that C^* was not limited to SIMD hardware and could be compiled for both message-passing and shared-memory MIMD computers. Their key observation was that SIMD-style operation-by-operation synchronization is unnecessary. Synchronization is only necessary between communicating statements running on distinct virtual processors. Even so, the overhead of this synchronization would be high, except that on MIMD machines, a few physical processors emulate each step of the many virtual processors. As problem size grows, the ratio of virtual to physical processors increases and the relative overhead of synchronization decreases.

Hatcher and Quinn extended C^* (and renamed their version to Dataparallel C) with lay-out directives for domains and nearest-neighbor communication primitives. These additions offer little benefit for shared memory, but were necessary to achieve reasonable performance on message-passing computers.

C^*, Version 6

Thinking Machines recently introduced a new version of C^* that dramatically revised the language. The *shape* abstraction replaced domains. A shape specifies the rank, dimension, and layout of an aggregate, but not the contents of each element. An aggregate element's contents is specified by prepending the aggregate's shape as a storage class modifier in variable declarations. In effect, the aggregate is declared an element field at a time. C^* version 6, unlike version 5, treats aggregates as first class values to which user-defined data-parallel operations can be applied. In addition, C^* version 6 introduces communication and storage layout primitives that expose the underlying message-passing orientation of the target hardware.

2.1.3 Paralation Lisp

Sabot's Paralation Lisp is a simple and general data-parallel language that avoids the SIMD bias. In Paralation Lisp, aggregates are called *paralations* and elements are called *fields*. Unlike the other languages, Paralation Lisp is based on a dynamically-typed language, Common Lisp, and paralations are not strongly typed, which makes them difficult to compile efficiently. A programmer applies an arbitrary Lisp function to a paralation. The function is evaluated element-wise on each element. The language loosely defines the semantics of parallel evaluation by declaring a program that depends on the evaluation order in a paralation to be "in error." This semantics eschews one of data-parallel programming's principal advantages by allowing races. Although Paralation Lisp does not permit nested parallelism, subsequent work lifted this restriction [3].

2.1.4 NESL

Blelloch's language NESL is a strongly-typed, applicative data-parallel language designed to support nested parallelism [2]. In NESL, a programmer applies a pure function (i.e., without side-effects) to a one-dimensional aggregate (vector) that contains arbitrary elements. The function application results are collected into a new aggregate. This model is semantically attractive, since it eliminates conflicts and data races due to imperative updates, but it shares implementation difficulties with functional languages.

2.1.5 PC++

Lee and Gannon described a new programming model called the *distributed collection model* and used it as the basis for PC++, their data parallel extension to C++ [9]. In their model, a *collection* is an aggregate that contains elements and is mapped to a set of processors by a *distribution*. Data parallel operations are invoked by sending a message to an aggregate, which redistributes the message to its elements, where the operation is actually applied. This model is similar to Chien's Concurrent Aggregates [6]. PC++ attempts to avoid conflicts by preventing an element from updating values in other elements, but read-write conflicts are still possible and may not be caught by the compiler[2]

2.1.6 Paragon

The Paragon programming environment is a collection of C++ classes that provide data-parallel array operations similar to those in Fortran 90 [5]. This approach is as limited as Fortran 90's

[2] Personal communication from Dennis Gannon, Nov. 18, 1992.

data-parallel sublanguage, but is a good demonstration of the extensibility and abstraction mechanisms in C++.

2.2 Design and Rationale for C^{**}

C^{**} is a new data-parallel language based on C^{++}. This section describes the small set of extensions to C^{++} that constitute C^{**} and explains the rationale for the design choices. C^{**} is a new language whose design will surely evolve. This section should be read as a snapshot of a work-in-progress rather than a final design.

C^{**} introduces a new type of object into C^{++}. These objects are Aggregates,[3] which collect elements into an entity that parallel functions can manipulate concurrently. C^{**} also provides slices so a program can manipulate portions of an Aggregate. Since these concepts are extensions to C^{++}, a C^{**} program can exploit that language's abstraction and object-oriented programming facilities.

2.2.1 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). Each element in an Aggregate object contains the member fields defined in the class. For example, the following declarations declare several 2-dimensional matrices of an indeterminate and two fixed sizes:

```
class matrix {float value;} [] [];
class small_matrix {float value;} [5] [5];
class large_matrix {float value;} [100] [100];
```

The bounds of the indeterminate-sized matrix, `matrix`, are specified when allocating an instance: `new matrix [100][100]`.

An Aggregate's rank is the number of dimensions listed in its class declaration. Rank is fixed by the declaration and cannot change. The cardinality of each dimension may be specified in the class declaration. If omitted, the cardinality must be supplied when an instance of the Aggregate is created. Each dimension is indexed from 0 to $N-1$, where N is the dimension's cardinality. For example, indices for both dimensions of a `small_matrix` run from $0 \ldots 4$.

An Aggregate object looks similar to—but differs fundamentally from—a conventional C^{++} array of objects:

- An Aggregate class declaration specifies the type of a collection, not individual elements. This is an important point: a `matrix` is an object containing a two-dimensional collection of floating point values, not a two-dimensional array of objects. The latter facility is still possible in C^{++}.
- Aggregate member functions are applied to an entire collection of elements, not individual elements.
- Elements in an Aggregate can be operated on concurrently, unlike objects in an array.

[3] The capital "A" is deliberate and distinguishes Aggregates from the unrelated data initialization lists called "aggregates" that are already part of C and C^{++}.

- Aggregates can be sliced (see Section 2.2.5).

However, individual Aggregate elements are referenced in the same manner as array elements, so, for example, if `A` is a `small_matrix` object, `A[0][0]` is its first element.

Discussion

Classes, because of their support for abstractions, are a natural basis for encapsulating aggregates and their operations. In C^{++}, however, classes and arrays are too distinct to directly specify aggregates. In C^{++}, either an array can be a class member or a array's elements can be objects. However, if an array is a class member, its bounds must be fixed in the class declaration. On the other hand, if we seek indeterminate bounds by making array elements into objects, the array itself is not an object. To be concrete, two possible C^{++} definitions of a matrix type require array bounds to be fully specified and do not permit a matrix abstract datatype:

```
class matrix {
  float value[10][10];
};

class matrix_element {float value;};
typedef matrix_element matrix[10][10];
```

C^{**} solves this problem by introducing a new type declaration that incorporates an Aggregate's dimension into its class declaration. The syntax is a combination of class and array declarations and clearly emphasizes that the declaration specifies an entire aggregate. Unfortunately, requiring static allocation for Aggregates—even if the bounds are unspecified—is awkward for aggregates whose membership changes with time, such as tree nodes. A program must preallocate enough tree nodes when the aggregate is created and cannot dynamically add new elements to the aggregate. C^{*}'s data-parallel operators, which are applied to all instances of a type, are better suited to dynamic allocation—as long as a program builds only one object (e.g., tree) from any aggregate.

2.2.2 Aggregate Constructors

C^{++} classes may define constructor and destructor functions that execute immediately after allocating space for an object and immediately before releasing the space. An allocation function has the same name as the class and the deallocation function's name is the class name preceded by a tilde ("~").

An Aggregate constructor initializes the entire collection of elements. This differs from an array of objects, in which each object's constructor must be invoked specifically. For example:

```
class matrix {
  float value;

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

defines a matrix constructor that initializes all elements to specified value, so:

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

creates a 100×100 matrix of 1's.

2.2.3 Aggregate Functions

In C^{++}, a class's *member functions* implement abstract datatype operations. These functions can access both public and private class fields. A member function (`f`) is applied to a class object (`o`) with a different syntax than a normal function call: `o.f()`. *Virtual member functions*, which can be overloaded and inherited, form the basis for object-oriented programming. A *friend function* is allowed access to private fields, however it is invoked with normal function call syntax and does not permit inheritance.

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

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

float matrix::sum () {
  float tot = 0.0;
  int i, j;
  for (i = 0; i < cardinality (0); i++)
    for (j = 0; j < cardinality (1); j++)
      tot += *this[i][j];
  return (tot);
}

ostream& operator<< (ostream &out, matrix &m) {
  int i, j;
  for (i = 0; i < cardinality (0); i++)
    for (j = 0; j < cardinality (1); j++)
      out << m[i][j].value << " ";
    out << '\n';
}
```

the function `sum` returns the sum of the values in a matrix and the operator `<<` is a friend function that prints a matrix.

All Aggregates have these two member functions predefined:

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

2.2.4 Parallel Functions

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

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

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

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

A parallel function is invoked simultaneously on all elements of its parallel argument. Each invocation can determine the coordinates of the element to which it is applied from the pseudo variables:

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

Semantics of Parallel Functions

A parallel function is a large-grain data-parallel operator that is applied *atomically and simultaneously* to each element in an Aggregate. To explain this 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 an individual Aggregate element. Hence, calling a parallel function starts many invocations, all of which execute simultaneously. An invocation's *state* is the collection of storage locations read or written during the invocation.

Applied atomically means that while an invocation is executing, its state is only modified by itself, not by other concurrently executing tasks. In other words, the call appears to execute instantaneously and the invocations are unaffected by concurrently executed tasks. In effect, the semantics are as if each invocation executes 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, `#1`, `#2`, etc., which differ in each invocation).

If invocations of a parallel function modify global state, the only guarantee is that when the call terminates, each modified location will contain a value written by some invocation. If an invocation modifies more than one location, only a portion of its modifications may be visible after the call. Other invocations' changes to a location may overwrite a particular invocation's modifications.

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

```
friend void stencil (parallel matrix A) parallel
 {
   A[#1][#2].value = (A[#1-1][#2].value + A[#1+1][#2].value
                    + A[#1][#2].value + A[#1][#2-1].value
                    + A[#1][#2+1]) / 5.0;
 }
```

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

Discussion

An analogy between large-grain data-parallel operations and SIMD operators may help motivate the semantics of parallel functions. Each processor in a SIMD computer applies an operation simultaneously, which means that no processor sees effects from an operation on another processor. The operations, however, execute independently. For example, when processors add numbers, each processor's hardware may pass through a different sequence of states.[4] User-level SIMD execution lacks this freedom to choose the most efficient computation for a particular datum.

Large-grain data parallelism permits computations on each element in an aggregate to proceed simultaneously, with independent threads of control. If a computation is a pure function—with no side effects—it suffices to specify the computation's initial state. However, an operation with side effects may affect the behavior of other operations. The interoperation data dependences must be ordered, or races will cause indeterminate execution.

These data dependences can be ordered in a surprisingly large variety of ways. Sequential execution imposes a canonical order on iterations over arrays (aggregates) that totally orders dependences. Parallelizing compilers respect this order. SIMD execution imposes a near-total order by advancing parallel computations in lockstep. Despite the disadvantages discussed above, SIMD execution is easy to understand because a SIMD program only has a single program counter, which can be followed like a sequential program counter. Another approach to preventing races is to prohibit conflicting dependences. Steele proposed a programming model in which operations that are not causally related (i.e., may execute in either order or simultaneously) must commute or are prohibited [13]. Violations of these semantics can only be detected during a program's execution. Steele described a complex memory system to detect these violations.

Large-grain data parallelism takes a different approach to preventing interoperation data dependences from affecting a program's behavior. While a parallel computation is running, its state is changed only by the computation itself, so it appears as if the computation executed sequentially. This suggests an analogy to database transactions, which use serializability as a correctness criterion [10]. Serializability, however, is not appropriate for data parallelism, because it requires equivalence to some sequential execution of the parallel tasks. For example, consider a stencil computation on a matrix. When a processor writes a location's average, serializability requires neighboring computations to start computing anew with this value. Instead of trying to make data parallelism equivalent to serial computation, large-grain data parallelism regards the computations for each element as equivalent and independent events that happen simultaneously.

Since parallel invocations cannot communicate, a typical programming style is to repeatedly invoke parallel operations. The communication occurs in the intervals between parallel calls, during which the results of the previous parallel computation can be distributed to other processors (see Figure 2.1).

Large-grain data parallelism allows data-parallel operations to invoke data-parallel operations. The semantics of nested operations are unaffected by the nesting. The side effects from one nested call are not visible to another nested call from a different invocation.

Large-grain data parallelism weakly guarantees the side-effect consistency in a data-parallel call. The problem, fortunately, arises only because of the undesirable situation of multiple, overlapping, concurrent writes to a location. If each invocation writes a single location, C^{**}'s

[4] Ironically, on the CM-2, where integer addition is implemented in microcode on top of bit-serial hardware, each processor passes through a different sequence of states.

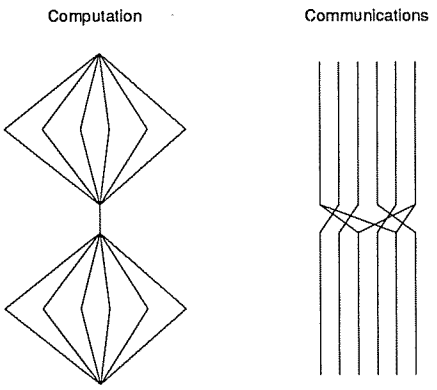


Figure 2.1: Programming style for large-grain data parallelism. In this model, parallel tasks cannot communicate, so the results of a parallel call are distributed after the call and before the next call.

semantics is similar to C^*'s: an arbitrary invocation's change is visible and other modifications are lost. However, if an invocation modifies multiple locations, we cannot ensure that all of the invocation's changes are consistently visible, unless we are ready to discard entirely other changes. For example, consider three invocations that modify locations A, B, C, and D in the following manner:

Invocation	Modified Locations
1	A, B, C
2	A, B, D
3	A, C, D

If we desire consistency and commit an invocation's entire result (say invocation 1's), all changes to location D must be discarded because of the overlapping modifications. Rather than discard changes (and incurring a heavy penalty to detect when this is necessary), C^{**} adopts the view that overlapping writes are fundamentally undesirable.

Results From Parallel Functions

Parallel functions produce either scalar or Aggregate results. If the underlying type is identical to the parallel argument's type, the parallel function allocates a new Aggregate of the same size as the parallel argument and initializes it with results returned from the corresponding parallel invocations of the function. The function invoked on the first element of the parallel argument computes the value for the first element of the result, and so on. The new Aggregate's constructor, if any, is not invoked.

On the other hand, if the underlying result type is a scalar, the values returned from the parallel function must be returned in reduction return statements (Section 2.2.4) that reduce values from the invocations into a single value.

For example, the two functions:

```
friend matrix operator* (parallel matrix A, matrix B) parallel
  {return (A[#1][] * B[][#2]);}

friend float operator* (parallel mrow R, mcol C) parallel
  {return%+ (R[#1].value * C[#1].value);}
```

comprise a matrix multiplication routine that creates and returns a new parallel matrix of the same size as the parallel matrix `A`. The first operator takes the two matrices as arguments and, for each position in the product matrix, computes the dot product of the appropriate row and column. The second operator computes this dot product.

If an Aggregate contains multiple fields, a return statement specifies the binding between returned values and fields with a construct similar to the one used to initialize `const` objects. For example:

```
class c {int a;  char *b;}[];

friend c f (parallel c A) parallel {
  {return {a(#1); b(int_to_str(#1));}
}
```

sets both fields in each element to a representation of the element's position in the Aggregate.

Discussion

An expression-oriented data-parallel programming style, such as the one used in the matrix multiplication example, is desirable for compilers. In this style, a computation is an expression built by composing operators. If the operators are pure—compute a new result rather than modify an existing one—a compiler has great freedom to rearrange and execute them in parallel. Techniques such as APL dragthrough can eliminate many, if not all, intermediate results [4]. C^{++}'s inlining facilities facilitate this process by making a function's code available at call sites.

Unfortunately, C^{++} does not provide a clean mechanism for a parallel function to create and return a composite result. In C^{++}, a function returns a composite result either by dynamically allocating it or modifying a parameter. Neither approach is convenient for parallel functions, and both complicate compiler analysis. C^{**}'s approach is only a slight extension to C^{++}, but permits a compiler to easily determine that the result of a parallel function is a new object built by independent, parallel computations.

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 sequence v can be reduced: $v_1 \circ v_2 \circ v_3 \ldots \circ v_n$. 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 and mediates conflicts between concurrent writes. The second combines values returned from the multiple invocations of a parallel function.

Combining Assignment Combining assignments are legal only within parallel functions. A combining assignment uses a specified operator to combine the set of values assigned to a location. Combining assignments are necessary because an ordinary assignment permits only one invocation to modify a location. For example:

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

matrix::sum_elements () parallel {
  sum =%+ value;
```

```
    if (value > 0)
      pos_sum =%+ value;
}
```

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 partial results from each invocation with a reduction return. This statement combines the value returned from each invocation of a parallel function with a specified operator. For example:

```
friend float sum (parallel mrow A) parallel
  {return%+ (A[#1].value);}
```

Discussion

*C** overloads C's += and similar operators for reductions when the left operand is a scalar and the right operand is a parallel expression. This choice leads to complex rules for determining when an expression is parallel and breaks the equivalence between `a = a + 1` and `a += 1`. Because of the semantics of large-grain data parallelism, overloading existing operators is confusing. The assignment in `a += 1` affects only local state until the surrounding parallel call finishes. Reduction operators appear different because their semantics—of combining multiple writes into a single, consistent value—is different.

2.2.5 Slices

A *slice* selects a subset of an Aggregate along the axis introduced by a subscript dimension. A slice is not a copy. It shares all selected elements with the full Aggregate. Slices are particularly valuable when they themselves are Aggregates and consequently can be manipulated in parallel. Slices permit effective specification of parallel computations on a portion 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. Trailing empty braces may be omitted, so:

```
A[1][]  <=>  A[1]
```

For example: the following three expressions are slices of a matrix:

```
matrix A;

A[i]      - i-th row of A
A[i][]    - or could be written this way
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 and declaration of new members. The slice's subclass declaration both names the new subclass and describes which indices must be specified to compute the slice. `#n` denotes the n^{th} index in a reference to the slice. For example, consider the following matrix slices:

```
class mrow :  matrix[#1];      - Row slice of matrix
class mcol :  matrix[][#1];    - Column slice of matrix
```

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

Slices are similar in many ways to function closures in that they capture an input parameter (index) and return a value that can be applied to another input to complete the computation of a result. This idea can be generalized by using the subclass facility to specify a complete remapping of indices. For example:

```
class matrixT : matrix[#2][#1];
```

is a class for a transposed matrix. In many cases, coercion between a matrix and its transpose need not allocate storage and physically transpose the array. Instead, the code to access the matrix can swap the row and column indices.

Discussion

Subclassing is appropriate for Aggregate slices in all respects except one. Any functions defined for a base class, say `matrix`, cannot be inherited by the derived class, say `mrow`, because the base class's member functions expect to operate on an entire aggregate, not a portion of it. Inheritance of other member fields is not a problem, since they are specified for every Aggregate element, both in the full Aggregate and the slice. However, a slice cannot introduce new data members.

2.3 Status

We have implemented a prototype translator for C^{**}, based on the GNU g++ compiler. The compiler produces naive code for a DECstation workstation and Sequent Symmetry shared-memory computer. We are investigating optimization techniques, language extensions, and compilation techniques for non-shared-memory machines such as the CM-5.

2.4 Conclusion

Parallel programming is a difficult task that demands a high level of support from the programming model and language. Data parallelism is a programming paradigm that reduces the complexity of parallel programming by permitting a programmer to focus on a single item, rather than worry about the interaction of many threads of control. This programming style results in parallel programs (i.e., programs that use parallel algorithms) that can be effectively compiled for a wide range of parallel computers because they contain large amounts of easily-identifiable parallelism.

Many data-parallel languages have two flaws. They either are too closely tied to SIMD hardware or they discarded the race-free properties of SIMD execution. The first approach imposes the execution inefficiences of SIMD hardware on all programs. This style of execution is not a burden in programs with simple control flow and evenly balanced work. However, many programs do not fit this mold and perform poorly under this model. The other approach, of viewing data parallelism as only mapping operations across aggregates, discards SIMD data parallelism's most attractive feature: nearly deterministic, nearly race-free program execution.

C^{**} is a new data-parallel programming language based on C^{++} that avoids these problems. It introduces a new model of data parallelism, called large-grain data parallelism, that offers

nearly race-free execution without the restrictions of SIMD execution. In addition, C^{**} is based on an object-oriented language with strong support for data abstraction and provides a good basis for building libraries of parallel data structures and operations. Finally, C^{**} should result in fast, efficient programs because it was designed to facilitate efficient compilation for a wide range of parallel computers.

Acknowledgements

Guhan Viswanathan and Brad Richards implemented the C^{**} compiler and runtime system. They and Satish Chandra provided many helpful comments on this paper.

Bibliography

[1] Guy E. Blelloch. *Vector Models for Data-Parallel Computing.* MIT Press, 1990.

[2] Guy E. Blelloch. NESL: A Nested Data-Parallel Language. Technical Report CMU-CS-92-103, Department of Computer Science, Carnegie Mellon University, January 1992.

[3] Guy E. Blelloch and Gary W. Sabot. Compiling Collection-Oriented Languages onto Massively Parallel Computers. *Journal of Parallel and Distributed Computing*, 8:119–134, 1990.

[4] Timothy A. Budd. An APL Compiler for a Vector Processor. *ACM Transactions on Programming Languages and Systems*, 6(3):297–313, July 1984.

[5] Craig M. Chase, Alex L. Cheung, Anthony P. Reeves, and Mark R. Smith. Paragon: A Parallel Programming Environment for Scientific Applications Using Communication Structures. In *Proceedings of the 1991 International Conference on Parallel Processing (Vol. II Software)*, pages II–211–218, August 1991.

[6] Andrew A. Chien and William J. Dally. Concurrent Aggregates (CA). In *Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 187–196, February 1990.

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

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

[9] Jenq Kuen Lee and Dennis Gannon. Object Oriented Parallel Programming Experiments and Results. In *Proceedings of Supercomputing 91*, pages 273–282, Albuquerque, NM, November 1991.

[10] Christos Papadimitriou. *The Theory of Database Concurrency Control.* Computer Science Press, 1986.

[11] John R. Rose. C*: A C++-like Language for Data-Parallel Computation. Technical Report PL-87-8, Thinking Machines Corporation, December 1987. Appeared in Usenix C++ Workshop Proceedings.

[12] 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*, pages 2–16, Santa Clara, California, May 1987.

[13] Guy L. Steele Jr. Making Asynchronous Parallelism Safe for the World. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 218–231, January 1990.

[14] Thinking Machines Corporation. C* Reference Manual, Version 4.0, August 1987.

[15] Thinking Machines Corporation. Supplement to C* Reference Manual, Version 4.3, May 1988.

[16] Thinking Machines Corporation. C* Reference Manual, Version 6.0 Pre-Beta, July 1990.

Appendix A

C^{**} Language Overview

This document describes C^{**}, an extensible, large-grain, value-oriented data parallel language based on C^{++}. The language is described in terms of its (upwardly compatible) changes to C^{++}. The section numbers (r.#) refer to the C^{++} Reference Manual in Bjarne Stroustrup, *The C^{++} Programming Language,* second edition, Addison-Wesley, 1991.

A.1 Aggregates

C^{++} (r.9) Syntax:

class-specifier:
  class-head *member-list*$_{opt}$

C^{**} Syntax:

class-specifier:
class-head *member-list*$_{opt}$ *agg-spec*$_{opt}$
agg-spec:
  [*expression*$_{opt}$]
  agg-spec [*expression*$_{opt}$]

In C^{**}, Aggregate objects are the basis for parallelism.[1] An Aggregate class (Aggregate, for short) declares an ordered collection of values, called Aggregate elements (element, for short), that can be operated on simultaneously by Aggregate parallel functions (parallel function, for short). Each element in an Aggregate object contains all member fields listed in the class declaration. For example, the following declarations define 2-dimensional matrices of various sizes:

```
class matrix {float value;} [] [];
class small_matrix {float value;} [5] [5];
class large_matrix {float value;} [100] [100];
```

The rank of an Aggregate is the number of dimensions specified in its class declaration. 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 dimension of a `small_matrix` run from $0 \ldots 4$.

An Aggregate object differs fundamentally from a conventional C^{++} array of objects:

[1] Note, Aggregates have no relation to aggregates (r.8.4.1) in base C or C^{++}.

- An Aggregate class declaration specifies the type of the collection, not the individual elements.
- Aggregate member functions operate on the entire collection of elements, not individual elements.
- Elements in an Aggregate can be operated in parallel, unlike objects in an array.
- Aggregates can be sliced (see Section A.3).

However, Aggregate elements are referenced in the same manner as objects in an array.

A.2 Aggregate Member Functions

Member functions in an Aggregate are similar, in most respects, to class member functions. The key difference is that Aggregate member functions are defined on an entire Aggregate, not an element. Consequently, the keyword `this` is a pointer to the entire Aggregate. For example, in:

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

ostream& operator<< (ostream &out, matrix &m) {
  int i, j;
  for (i = 0; i < cardinality (0); i++)
    for (j = 0; j < cardinality (1); j++)
      out << m[i][j].value << " ";
    out << '\n';
}
```

`<<` prints a matrix to a stream.

All Aggregates have the following two member functions predefined:

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

A.2.1 Aggregate Constructors

A constructor for an Aggregate class initializes the entire collection of elements, not the individual elements. This is unlike an array of objects, in which the array is initialized by each object's constructor, which is invoked in turn. For example,

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

defines a matrix constructor that initializes all elements to specified value, so

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

creates a 100×100 matrix of 1's.

Constructors may be parallel functions.

A.2.2 Parallel Functions

C++ (r.8) Syntax:
cv-qualifier:
  `const`
  `volatile`

*C*** Syntax:
cv-qualifier:
  `const`
  `volatile`
  `parallel`

C++ (r.7.1) Syntax:
decl-specifier:
  storage-class-specifier
  type-specifier
  fct-specifier
  template-specifier
  `friend`
  `typedef`

*C*** Syntax:
decl-specifier:
  storage-class-specifier
  type-specifier
  fct-specifier
  template-specifier
  `friend`
  `typedef`
  `parallel`

A *parallel function* is a member or friend function in an Aggregate class that is invoked simultaneously on each element of the Aggregate.

Aggregate member and friend functions are sequential by default. A parallel function (either a member or friend function) is identified by keyword `parallel` after its argument list. For example:

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

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

In a parallel friend function, exactly one argument, of the same type as the Aggregate, must be prefaced by `parallel` (for example, the first argument in `transpose`). This argument is the function's *parallel argument.* In a parallel member function, the parallel argument is the Aggregate to which the function is applied.

A parallel function is invoked simultaneously on all elements of its parallel argument. Each invocation can determine the coordinates of the element to which it is applied from the pseudo variables:

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

Semantics of Parallel Functions

A parallel function is a large-grain data-parallel operator that is applied atomically and simultaneously to each element in an Aggregate. To explain this 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 execute simultaneously. A function invocation's *state* is the collection of storage location 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 any other concurrently executing task. 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:

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

Since a parallel function is applied *simultaneously* to an Aggregate's elements, all invocations begin with identical state (except for the pseudo variables, `#1`, `#2`, etc., which differ in each invocation).

If invocations of a parallel function modify global state, the only guarantee is that each modified location will contain a value computed by some invocation. However, if an invocation modifies more than one location, only a portion of its modifications may be visible after the parallel call. Other invocations' changes to the other locations may overwrite the first invocation's modifications.

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

```
friend void stencil (parallel matrix A) parallel
 {
   A[#1][#2].value = (A[#1-1][#2].value + A[#1+1][#2].value
                      + A[#1][#2].value + A[#1][#2-1].value
                      + A[#1][#2+1].value) / 5.0;
 }
```

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

Implementation Note

Copying is unnecessary for locations not modified by another invocation or that are modified by only one invocation and not read by other invocations. A location modified by one invocation and read by other invocations can either be locked to let the readers finish or could be implemented with the copying scheme. Multiple writers present no complications since the writes must either be combined with an explicit combining assignment (Section A.2.2) or allow an arbitrary write to succeed.

Parallel Return Type

C^{++} (r.6.6) Syntax:

jump-statement:
  `break ;`
  `continue ;`
  `return` *expression*$_{opt}$ `;`

  `goto` *identifier*;

C^{**} Syntax:

jump-statement:
  `break ;`
  `continue ;`
  `return` *expression*$_{opt}$ `;`
  `return` *ctor-initializer*
  `goto` *identifier* ;

Parallel functions produce either scalar or Aggregate results. If the underlying type is the same as the parallel argument's type, the parallel function allocates a new Aggregate of the same size as the parallel argument and initializes it with results returned from the corresponding parallel invocations of the function. The Aggregate's constructor, if any, is not invoked. The function invoked on the first element of the parallel argument computes the value for the first element of the result, and so on. For example:

```
friend matrix operator* (parallel matrix A, matrix B) parallel
  {return (A[#1][] * B[][#2]);}
```

multiplies two matrices by setting each element of the result matrix to the inner product of the corresponding row and column (see A.3).

If an Aggregate contains multiple fields, a return statement can specify the binding between returned values and member fields with a construct similar to the one used to initialize `const` objects. For example,

```
class c {int a;  char *b;}[];

friend c f (parallel c A) parallel {
  {return {a(#1); b(int_to_str(#1));}
}
```

sets both fields in each element to an appropriate representation of the element's position in the Aggregate.

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 A.2.2) that combine values from the invocations into a single value of the specified type. For example, the inner product can be defined:

```
friend float operator* (parallel mrow R, mcol C) parallel
  {return%+ (R[#1].value * C[#1].value);}
```

Combining Assignment

C^{++} (r.5.17) Syntax:

assignment-operator: one of
  `= *= /= %= += -= >>= <<= &= ^= |=`

C^{**} Syntax:

assignment-operator: one of
  `= *= /= %= += -= >>= <<= &= ^= |=`
  `=%* =%% =%+ =%& =%^ =%| =%>? =%<?`

Combining assignments are legal only within parallel functions. When invocations on different elements modify a location with combining assignments, the values assigned to the location are combine by operator specified to the right of the %. The operators >? and <? are *min* and *max*, respectively. Combining assignments are necessary because an ordinary assignment permits only one invocation to modify the location.

Because of the semantics of parallel functions, within one of these functions, a combining assignment only appears to change the assigned variable by the value produced by the right-hand expression.

For example:

```
s =%+ A[#1].value;          - Parallel +-reduction

if (p) s =%+ A[#1].value;   - Parallel +-reduction on subset
```

Reduction Returns

C^{++} (r.6.6) Syntax:

jump-statement:
- `break ;`
- `continue ;`
- `return` *expression*$_{opt}$ `;`
- `goto identifier ;`

C^{**} Syntax:

jump-statement:
- `break ;`
- `continue ;`
- `return` *expression*$_{opt}$ `;`
- `return/`*red-op expression* `;`
- `return/`*choose-op expression* `:` *expression* `;`
- `goto identifier ;`

red-op: one of
- `%+ %* %& %| %^ %<? %>?`

choose-op: one of
- `%<? %>?`

A parallel function with a scalar result must combine the results from each element with a reduction return. This statement combines the value returned from each invocation of a parallel function using the operator specified to the right of the %. A parallel function with a non-void result type, must terminate with one or more reduction returns, all of which specify the same operator.

For example:

```
float sum (parallel mrow A) parallel
  {return%+ (A[#1].value);}
```

The second form of reduction return implements Quinn and Hatcher's tournament idiom: determine the value associated with the invocation that returns the minimum or maximum key value. The expression before the colon is the key used to compare returned values. The expression after the colon is the value associated with the invocation. The result of the return statement is the value from an invocation that returns the minimum (maximum) key. For example, the following function finds the row in a matrix containing the smallest element:

```
int smallest_row (parallel matrix M) parallel
  {return%<? (M[#1][#2].value : #1);}
```

To Be Considered:

User-supplied reduction operations.

A.3 Slices

C++ (r.5.2) Syntax:
postfix-expression:
...
postfix-expression [*expression*]
...

*C*** Syntax:
postfix-expression:
...
postfix-expression [$expression_{opt}$]
...

Omitting an index expression from a dimension of an Aggregate reference produces a slice that includes all elements along that dimension. A slice is *not* a copy. It shares all selected elements with the larger Aggregate. The type of the slice is a subaggregate of the larger Aggregate (see Section A.3.1).

Trailing empty braces may be omitted, so

```
A[1][]  <=>  A[1]
```

For example:

```
matrix A;

A[i]        - i-th row of A
A[i][]      - or could be written this way
A[][j]      - j-th column of A
```

Slicing is higher precedence than indexing, so given a choice between treating an index operation as being part of a slice or indexing an object, the former is always chosen. For example, in

```
class D1 {...} [][][];
class S1 : D1[#1];
class S2 : D1[#1][#2];

D1 X[100][100][100];

... X[1][2] ...
```

the reference to `X` is an `S2` slice, rather than an indexing operation on an `S1` class.

A.3.1 Subaggregates and Slices

C++ (r.10) Syntax:

base-spec:
: *base-list*
base-list:
base-specifier
base-list , *base-specifier*

*C*** Syntax:

base-specifier:
  complete-class-name
  `virtual` *access-specifier*$_{opt}$ *complete-class-name*
  access-specifier virtual$_{opt}$ *complete-class-name*

base-specifier:
  complete-class-name partial-agg-spec$_{opt}$
  `virtual` *access-specifier*$_{opt}$ *complete-class-name partial-agg-spe*
  access-specifier virtual$_{opt}$ *complete-class-name partial-agg-spec*

partial-agg-spec:
  [*expr-index*$_{opt}$]
  partial-agg-spec [ ]
  partial-agg-spec [*expr-index*$_{opt}$]

expr-index:
  expression
  # *integer*

A slice of an Aggregate produces a subaggregate, which is a new Aggregate class. Subaggregates allow logically contiguous subsets of an Aggregate to be manipulated as a single entity, including having parallel operations performed upon them.

A subaggregate declaration names the new class, describes the slicing operations that produces the subclass, and describes how to map references to the subaggregate into the underlying Aggregate. The pseudo variable `#n` denotes the n^{th} index in a reference to the subaggregate. Dimensions whose index is not specified in a references to the subaggregate must have been specified in the slice.

For example, consider the following slices of a matrix:

```
class mrow :  matrix[#1];     - Row slice of matrix
class mcol :  matrix[][#1];   - Column slice of matrix
```

A reference to an `mrow` provides the column coordinate. The row coordinate is determined when a matrix is sliced along its first axis and the `mrow` is created.

Subaggregates are *not* subclasses. All though they may define new member functions, they do not inherit their underlying Aggregate's member functions. Nor, many they define new data members. In addition, the type of a slicing operation on an Aggregate must be unique, the `partial-agg-spec`'s for all subaggregate of an Aggregate must be unique.

Appendix B

Examples

This section contains three sample C^{**} programs and some preliminary speedup numbers.

B.1 Matrix Multiplication

The first example is the complete program for multiplying matrices:

```
#include "css-rt.h"

struct matrix
{
    double val;

    matrix();
    matrix operator* (matrix &B) parallel;
    void print(int i);
} [512][512];


/* Initialize the matrix with pseudo random values: */
matrix::matrix()
{
#define ran(seed) ((seed=(seed*16397+14927)%29383), seed/10000.0)
    int i, j;
    long seed = 39293;

    for(i=0; i!=N; i++)
    {
        for(j=0; j!=N; j++)
            (*this)[i][j].val = ran(seed);
    }
#undef ran
}


/* A column slice of a matrix */
struct column : matrix[][.]
{
};
```

```
/* A row slice of a matrix */
struct row: matrix [.][]
{
    /* Dot product of a row by a column */
    double operator*(column &B) parallel
    {
        return %+ (val * B[#0].val);
    }
};


matrix matrix::operator* (matrix &B) parallel
{
    return { val((*this)[#0][] * B[][#1]) };
}


void matrix::print(int i)
{
    int j, k;

    if (i == 0)
        return;

    for (j = 0; j < cardinality(0); j ++)
    {
        printf("%d:\t", j);
        for (k = 0; k < cardinality(1); k ++)
            printf("%g ", (*this)[j][k].val);
        printf("\n");
    }
    printf("\n");
}



main()
{
    matrix A, B;

    (A * B).print(0);
}
```

We compiled this program with the preliminary version of our compiler and ran it on a 20 processor Sequent Symmetry. Figure B.1 compares the performance of the C^{**} program against a hand-coded C program. Both curves are normalized against the parallel, hand-coded C program running on one processor. As the graph shows, the absolute performance of the C^{**} program is less than the C program. There are two reasons for this difference. First, our compiler generates simple code and does not perform any optimizations. In addition, our implementation defers all scheduling to run-time. A better compiler would statically schedule this routine.

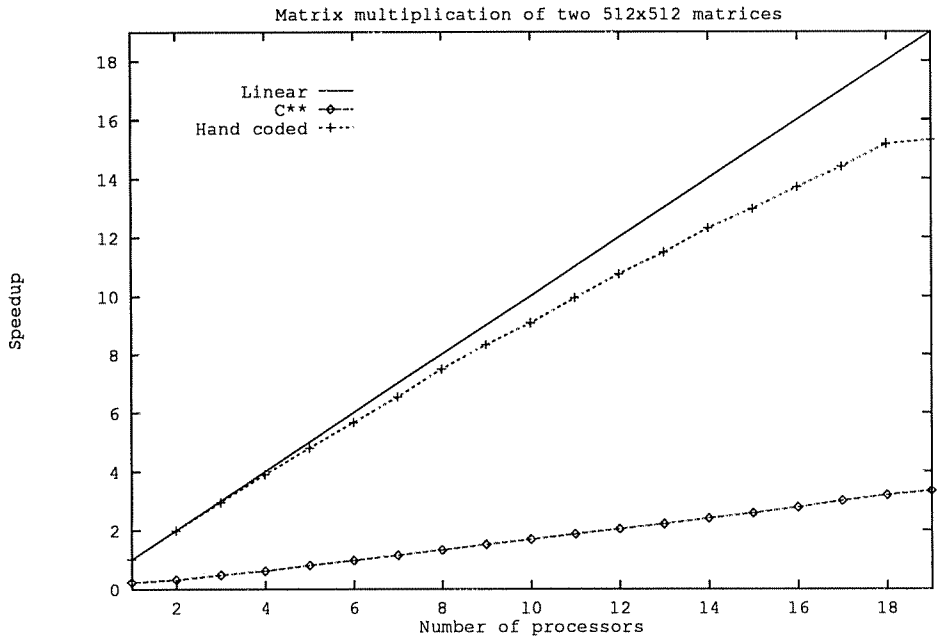


Figure B.1: Comparison of C^{**} and C matrix multiplication routines on a 20 processor Sequent Symmetry.

B.2 Pgrid

Pgrid is a program that computes potentials in a grounded box. The figure on the frontpiece is an example of its computation. We ran two versions of this program. The first version operated on each grid point individually:

```
/*========================================================================
     This program computes potentials within a grounded box.  There are
 "objects" within the box whose potential is set to "1".  (The locations
 of these objects are read in from a file at startup.)  A grid of data
 points is formed over the box, and the potential at each point is found
 by averaging the values of the four nearest neighbors.  This process is
 repeated until the grid "relaxes".  That is, until the largest change
 in value of any of the cells during an iteration becomes very small.
     Normally, one would keep track of this largest change, and iterate
 until it was less than some required epsilon.  This program is written
 such that it finds the largest change during each iteration, but the
 value is ignored and a fixed number of iterations are performed.  This
 is because the order in which the grid points are updated in the
 sequential version affects the speed at which the grid becomes relaxed.
 It would likely be the case that the parallel version required a dif-
 ferent number of iterations to reach a given state of relaxation.  For
 timing purposes, it was thought to be better if both the sequential and
 parallel version performed the same number of iterations over the grid.
 ========================================================================*/

#include <stdio.h>
#include <math.h>
#include "css-rt.h"
```

```
#define DIM     52
#define INF     100000


struct grid
{
    float val;          /* This grid point's value */
    float diff;         /* difference between old and new values */

    grid() parallel     /* set to all zeros initially */
    {
        val = 0;
    }

    float update() parallel;

} [DIM][DIM];

int obj[DIM][DIM];      /* array holding locations of "objects" */
FILE *infile;

/*
 *  This method updates each of the grid points in parallel.  The
 *  largest change (as a percentage) is returned and would ordinarily
 *  be used to decide when the computation was sufficiently precise.
 */
float grid::update() parallel
{
    float old;

        diff = 0;
        if ((#0==0) || (#0==DIM-1) || (#1==0) || (#1==DIM-1))   /* edges */
            return %>? diff;

        if (obj[#0][#1] == 1) {         /* on an object, value is 1 */
            val = 1;
            diff = 0;
        } else {                        /* otherwise, average neighbors */
            old = val;
            val = ((*this)[#0][#1-1].val + (*this)[#0][#1+1].val +
                        (*this)[#0-1][#1].val + (*this)[#0+1][#1].val)/4.0;
            if (val != 0)
                diff=fabs((double)(val - old))/val;
            else if (fabs((double)(val - old)) == 0)
                diff = 0;
            else
                diff = INF;
        }

        return %>? diff;                /* return largest change */
}


main()
{
    int i, j;
    float max=0;
    grid G;
```

```
        infile = fopen("setup", "r");   /* read object positions from file */
        if (infile==NULL) fprintf(stderr, "Didn't find setup file.\n");

        for (j=1; j<DIM-1; j++)
            for (i=1; i<DIM-1; i++)
                fscanf(infile, "%d", &obj[i][j]);

        for (i=0; i<500; i++) {         /* do 500 iterations */
            max = G.update();
        }

        for (i=0; i<DIM; i++) {
            for (j=0; j<DIM; j++)
                printf("%d  %d  %f\n", i, j, G[i][j].val);
            printf("\n");
        }
}
```

This version was inefficient because the amount of computation at each point is small and our simple compiler does not optimize the program by combining these small tasks. The second version of the program is similar, except that it performed parallel operations on each row in the matrix:

```
/*======================================================================
     This program computes potentials within a grounded box.  There are
 "objects" within the box whose potential is set to "1".  (The locations
 of these objects are read in from a file at startup.)  A grid of data
 points is formed over the box, and the potential at each point is found
 by averaging the values of the four nearest neighbors.  This process is
 repeated until the grid "relaxes".  That is, until the largest change
 in value of any of the cells during an iteration becomes very small.
     Normally, one would keep track of this largest change, and iterate
 until it was less than some required epsilon.  This program is written
 such that it finds the largest change during each iteration, but the
 value is ignored and a fixed number of iterations are performed.  This
 is because the order in which the grid points are updated in the
 sequential version affects the speed at which the grid becomes relaxed.
 It would likely be the case that the parallel version required a dif-
 ferent number of iterations to reach a given state of relaxation.  For
 timing purposes, it was thought to be better if both the sequential and
 parallel version performed the same number of iterations over the grid.
 ======================================================================*/

#include <stdio.h>
#include <math.h>
#include "css-rt.h"
#define DIM     52
#define INF     100000


struct rows
{
    float values[DIM];  /* This grid point's value */
    float max_diff;     /* largest difference between old and new values */
```

```
    rows() parallel     /* set to all zeros initially */
    {
        int i;

        for (i=0; i<DIM; i++)
            values[i] = 0;
    }

    float update() parallel;

} [DIM];

int obj[DIM][DIM];      /* array holding locations of "objects" */
FILE *infile;

/*
 *  This method updates each of the grid points in parallel.  The
 *  largest change (as a percentage) is returned and would ordinarily
 *  be used to decide when the computation was sufficiently precise.
 */
float rows::update() parallel
{
    int col;
    float old, diff;

        max_diff = 0;
        if ((#0==0) || (#0==DIM-1))     /* edges */
            return %>? max_diff;

        for (col=1; col<DIM-1; col++) {
            if (obj[#0][col] == 1) {    /* on an object, value is 1 */
                values[col] = 1;
            } else {                    /* otherwise, average neighbors */
                old = values[col];
                values[col] = (values[col-1] + values[col+1] +
                        (*this)[#0-1].values[col] +
                        (*this)[#0+1].values[col])/4.0;
                if (values[col] != 0)
                    diff=fabs((double)(values[col] - old))/values[col];
                else if (fabs((double)(values[col] - old)) == 0)
                    diff = 0;
                else
                    diff = INF;
                if (diff > max_diff) max_diff = diff;
            }

        }
        return %>? max_diff;            /* return largest change */
}


main()
{
    int i, j;
    float max=0;
    rows R;

        infile = fopen("setup", "r");   /* read object positions from file */
```

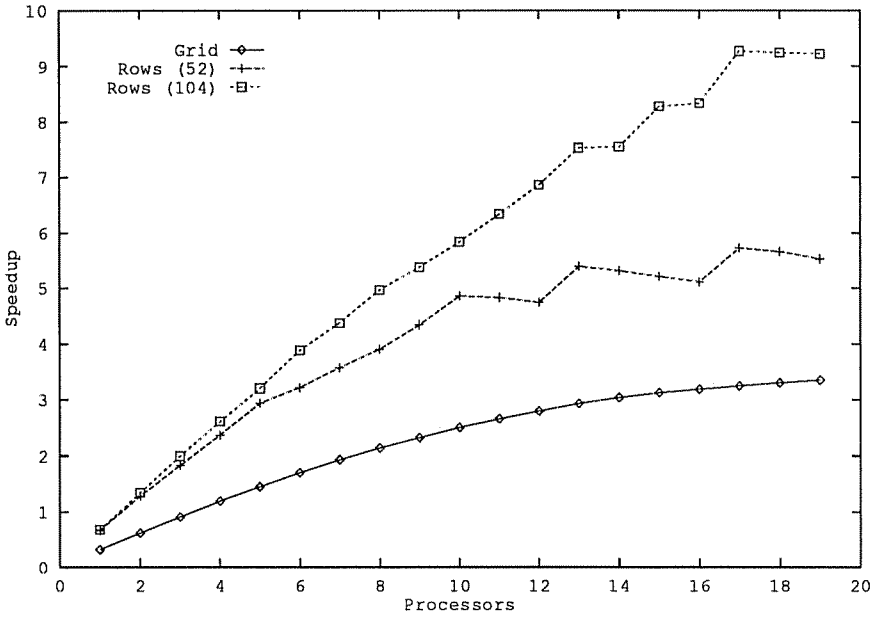


Figure B.2: Comparison of C^{**} and C grid routines on a 20 processor Sequent Symmetry.

```
        for (j=1; j<DIM-1; j++)
            for (i=1; i<DIM-1; i++)
                fscanf(infile, "%d", &obj[i][j]);

        for (i=0; i<500; i++) {          /* do 500 iterations */
            max = R.update();
        }

        for (i=0; i<DIM; i++) {
            for (j=0; j<DIM; j++)
                printf("%d  %d  %f\n", i, j, R[i].values[j]);
            printf("\n");
        }
}
```

Figure B.2 shows the speedup of the two version of this program. The row-blocked version of the program did not have enough parallelism on the original 52×52 matrix, so we also ran it on a larger matrix.

B.3 Quicksort

The final example is a C^{**} program for sorting a vector with quicksort. Although quicksort is not a very good parallel algorithm, this program shows several important features of C^{**}. The data parallel operations are performed on two-element descriptors that delimit the partition of the vector. These operations are recursive and so the data parallelism is nested. The current compiler does not schedule nested data parallelism properly, so the performance of this program does not improve with additional processors.

```
/* qs.c
   quicksort program in C**.

   Copyright (c) James R. Larus, June 1992.
   All rights reserved.
*/

#include <stdlib.h>
#include <stdio.h>

#include "css-rt.h"

class desc
{
 public:
  void qs () parallel;                  /* Quicksort */

  desc (int l, int h, int* v) parallel {low = l; high = h; vec = v;};

  desc (int l, int mid, int h, int* v)
  {
      (*this)[0].low = l;
      (*this)[0].high = mid;
      (*this)[1].low = mid;
      (*this)[1].high = h;
      (*this)[0].vec = v;
      (*this)[1].vec = v;
  };

 public:
  int low, high;
  int *vec;
} [] ;

int partition (int *vec, int low, int high);


int
main (int argc, char *argv[])
{
  int N;
  int *vec;
  int i;

  if (argc < 2)
    {
      printf("Usage : qsort size\n");
      exit(-1);
    }

  N = atoi(argv[1]);
  vec = new int[N];

  for (i = 0; i < N; i++)
    vec[i] = random ();
```

```
  desc d[1](0, N, vec);
  d.qs ();

  for (i = 0; i < N - 1; i++)
    if (vec[i] > vec[i+1])
      printf ("%d: %d > %d: %d\n", i, vec[i], i+1, vec[i+1]);
}


void desc::qs () parallel
{
  if (low < high - 1)
    {
      int mid = partition (vec, low, high);
      desc sd[2](low, mid, high, vec);

      sd.qs ();
    }
}


int partition (int *vec, int low, int high)
{
  int pivot = vec [(low + high) / 2];

  while (low < high)
    {
      while (vec[low] <= pivot && low < high) low ++;
      while (vec[high] > pivot && high > low) high --;
      if (low < high)
        {
          int tmp = vec[low];
          vec[low] = vec[high];
          vec[high] = tmp;
        }
    }
  vec[(low + high) / 2] = vec[low];
  vec[low] = pivot;
  return (low - 1);
}
```
