# ICC++ – A C++ Dialect for High Performance Parallel Computing*

A. A. Chien, U. S. Reddy, J. Plevyak and J. Dolby**

Department of Computer Science
1304 W. Springfield Avenue
Urbana, IL 61801

**Abstract.** ICC++ is a new concurrent C++ dialect which supports a single source code for sequential and parallel program versions, the construction of concurrent data abstractions, convenient expression of irregular and fine-grained concurrency, and high performance implementations. ICC++ programs are annotated with potential concurrency, facilitating both sharing source with sequential programs and automatic grain size tuning for efficient execution. Concurrency control is at the object level; each object ensures the consistency of its own state. This consistency can be extended over larger data abstractions. Finally, ICC++ integrates arrays into the object system and the concurrency model. In short, ICC++ addresses concurrency and its relation to abstractions – whether they are implemented by single objects, several objects, or object collections. The design of the language, its rationale, and current status are all described.

**Keywords**
concurrent languages, parallelism, object-parallel programming, memory consistency, concurrent object-oriented programming.

## 1 Introduction

Object-oriented approaches to writing parallel programs support modularity, polymorphism, and code reuse which provide crucial leverage for managing the complexity of concurrency and distribution. This leverage aids in writing high performance programs which exploit complex irregular and adaptive computational methods as well as intricate distributed data structures. However, concurrency interacts subtly with abstractions, so a concurrent object-oriented

* The research described in this paper was supported in part by NSF grants CCR-9209336 and MIP-92-23732, ONR grants N00014-92-J-1961 and N00014-93-1-1086 and NASA grant NAG 1-613. Andrew Chien is supported in part by NSF Young Investigator Award CCR-94-57809. Uday S. Reddy is supported by the National Science Foundation, grant NSF-CCR-93-03043

** The authors can be contacted at (217) 333-6844 (phone) and (217) 333-3501 (fax), and by e-mail at {achien, reddy, jplevyak, dolby}@cs.uiuc.edu.

language must be carefully designed to preserve the benefits of object-oriented techniques. Furthermore, language design impacts achievable efficiency, so such languages must also be designed with modern compiler optimization techniques in mind.

ICC++ is based upon C++ because, since C++ has been widely adopted, leveraging it exploits both existing software development tools and the training of many programmers. However, C++ introduces a number of complications, most of which derive from the exposure of low-level implementation information. Examples include pointers, especially pointer arithmetic within objects and arrays, and a restrictive but unsafe type system. In the design of ICC++, we have attempted to preserve C++, modifying or augmenting the language only where it was absolutely necessary for concurrency and for effective compiler analysis.

We believe that for a parallel object-oriented language to be widely accepted, it must support the following capabilities:

- construction of *concurrent data abstractions*,
- convenient expression of *irregular and fine-grained concurrency*,
- high *sequential and parallel* performance, and
- *single source* code for sequential and parallel program versions

Data abstractions are the key element of object-oriented programming; it must be possible to build a data abstraction and assure its correctness in a number of concurrency environments. These data abstractions must be able to support high levels of concurrency to provide modularity for highly concurrent program structures. Object-oriented programs are by their nature fine-grained (many objects and procedures are small), and the natural expression of concurrency is often in these units. Performance is important, as it is often the *raison d'être* of parallelism. Parallelizing a program must not preclude its efficient execution on one or a small number of processors. If data locality is high, parallel C++ codes must produce C-like efficiency to be competitive. A single source code is important because the vast majority of software is developed for uniprocessors; single source allows many programs to be parallelized without radical rewriting. ICC++ supports these four capabilities with three key features:

*Extensional Concurrency Constructs* ICC++ provides language constructs for specifying *available concurrency* as a partial order of execution. These language constructs encourage the specification of irregular parallelism because concurrency can be specified by annotating blocks or loops without disturbing the surrounding program structure. Specification of available concurrency also supports efficient implementation, allowing the system to serialize execution as necessary for efficiency.

*Flexible Object Concurrency Model* ICC++ specifies an object consistency model which allows programmers to reason about a data abstraction's correctness. In addition, ICC++ also specifies concurrency guarantees for objects, enabling programmers to reason about progress. Because it may be preferable to implement

a data abstraction with several objects, the consistency model can be extended over multiple objects structurally and procedurally.

*Integrated Arrays and Objects* ICC++ integrates arrays and objects, providing concurrent arrays with an object interface. These object collections can be used to build concurrent abstractions, providing modularity for a wide range of concurrent program structures.

ICC++ was designed as part of the Illinois Concert project, and is described fully in [15, 23]. We have researched the design of concurrent object oriented programs [35, 13, 17, 16], building numerous application programs totaling over 40,000 lines. In addition, we have studied the design of concurrent object-oriented languages and their implementation [38, 37, 25, 14, 12], exploring a variety of aggressive compiler and runtime techniques. The design of ICC++ was based on this experience, and an extensive survey of parallel object-oriented approaches. An implementation based on the Illinois Concert system has been in progress since February 1995, and has recently become operational. Performance results are not yet available, but we expect them to be in line with previous published studies using the Concert system [14, 19, 37, 38, 36].

The remainder of the paper is organized as follows. The three key features enumerated above are described in Sections 2, 3 and 4 respectively. An extended example ICC++ program is examined in Section 5. In Section 6, the most pertinent related work is briefly described and related issues are discussed. Finally, Section 7 closes by summarizing the paper.

## 2 Concurrency

The specification of concurrency should be as flexible as possible, while supporting a single source with sequential program versions and efficient sequential and parallel implementations. Therefore concurrency should be introduced as an extension to existing syntactic structures and the concurrent constructs should leave implementations maximal flexibility. ICC++ provides concurrent blocks and concurrent loops as extensions to standard C++ syntax so as to enable incremental parallelization of existing code. Furthermore, both the blocks and loops introduce *available concurrency*, not guaranteed concurrency, providing implementation freedom.[3]

### 2.1 `conc` Blocks

The primary construct for introducing available concurrency is the `conc` block, an extension of the C++ compound statement, which specifies that the contained statements are only partially ordered based upon local data dependences. `conc` blocks are defined as follows:

[3] ICC++ also includes primitives which guarantee concurrency for situations where a guarantee is required [23]; however, these primitives are expensive as they dictate an implementation to the system, and should be used with care.

$$\texttt{conc } \{ \; S_1 \texttt{;} \; \ldots \; \texttt{;} \; S_n \texttt{;} \; \}$$

is a compound statement which defines the following partial order $\prec$ on its constituents:

$$S_i \prec S_j \Longleftrightarrow i < j \; and \; S_i \Rightarrow S_j$$

where $S_i \Rightarrow S_j$ indicates that statement $S_j$ depends upon $S_i$. Statements in a `conc` block are executed such that if $S_i \prec S_j$, then $S_i$ will be executed completely before $S_j$ is begun. $S_j$ depends upon $S_i$ if

1. any identifier assigned in $S_i$ is read in $S_j$, or
2. if $S_i$ contains a jump statement.

The first rule allows concurrent operations upon objects via pointers, while preserving sequential semantics for local variables, including the pointers themselves. Thus `conc { a->b(); a->c();}` would be concurrent but `conc { a = new Foo(); a->b(); }` would be sequential. Jump statements are `goto`, `break` and `continue`; the second rule gives them natural semantics, serializing the `conc` block around them. A `conc` block exits after all statements within it have completed. As in C++, nested blocks are treated as single statements.

These rules are designed to expose concurrency upon objects, while preserving sequential semantics where natural; this enables the introduction of concurrency with small perturbation to program structure. Sequentializing for local variables allows preexisting compound statements that declare and use local variables to be transformed into conc blocks, exposing concurrency for calls upon objects within them. Similarly, permitting control flow within conc blocks, and providing a natural semantics for it, allows `conc` to be applied to preexisting code with such irregular control structures.

By exposing *available* concurrency rather than *guaranteed* concurrency, the `conc` block provides crucial implementation latitude. Since there is no requirement for fair scheduling of the statements within the block, they can be sequentialized and calls may be inlined where appropriate. Furthermore, the implementation can choose where to exploit concurrency depending upon the grain size and scale of the target machine.

### 2.2 `conc` Loops

Each of the C++ looping constructs can be modified by `conc` producing `conc for`, `conc while`, and `conc do while`. C++ is unusual in that no loop construct has a distinguished loop variable, as does `for` in Pascal and `do` in FORTRAN. Thus, the semantics of the concurrent loop forms must be designed carefully to expose cross-iteration concurrency while retaining reasonable behavior for the local variables. Furthermore, the concurrent loops must be compatible extensions: since all C++ loops allow control flow operations, the concurrent loops must support them as well. `conc` loops are dynamically unfolding `conc` blocks. Thus, loop carried (read after write) dependences are respected only for scalar variables, but not for others such as array dependences and those through pointer structures.

```
conc while (i < 5) {                if (i < 5) conc {
  a->foo( i);                         a->foo(i);
  i++;                                i++;
}                          ==>        if (i < 5) conc {
                                        a->foo(i);
                                        i++;
                                        ...
```

The above code fragment exemplifies how the dynamically unfolding `conc` block works. This code executes as follows. First the `if` test is evaluated, and then the outer `conc` block starts. The call on `foo` and the `i++` both start immediately. As soon as the `i++` finishes, the nested `if` statement starts, and this cycle repeats until the `if` test fails. Note that the nested `if` must wait for the `i++`, but not for the call to `foo`. Thus, the calls to `foo` operate concurrently, but the index variable `i` is sequentialized properly.

The motivation of this design parallels that of `conc` blocks. Permitting control flow and respecting scalar variable dependences within concurrent loops simplifies adding concurrency to preexisting sequential loops. As with `conc` blocks, concurrent loops specify *available* concurrency and make no guarantees about actual concurrency. This allows the implementation considerable latitude in scheduling iterations, such as running groups of iterations sequentially on different nodes.

### 2.3 Examples

In our experience with the Concurrent Aggregates language, we have found the semantics of `conc` blocks and loops to be widely useful. Some example `conc` structures are shown below.

```
void qsort(int A[], int p, int r) {   void qsort(int A[], int p, int r) {
  if (p < r) {                          if (p < r) conc {
      int q = partition(A,p,r);             int q = partition(A,p,r);
      qsort(A,p,q);                         qsort(A,p,q);
      qsort(A,q+1,r);                       qsort(A,q+1,r);
    }                                     }
}                                     }
```

The canonical `qsort` program fragment above illustrates adding a `conc` to incrementally parallelize a program; the sequential code is on the left, the parallel code on the right. `conc` blocks respect dependences for local variables, and dependences are created only by assignments. Thus, the two calls to `qsort` will be constrained to occur after `partition` because the call to `partition` assigns `q` and the calls to `qsort` use it. So adding a `conc` to this function introduces concurrency trivially.

```
Particle particles[] = new Particles[particle_count];

conc for(int i = 0; i < particle_count; i++)
  conc for(int j = 0; j < particle_count; j++)
    particle[i]->check_collision(particle[j]);
```

The above loops illustrate how sequential behavior is preserved while still exposing concurrency. Since loop carried dependences are respected for `i` and `j`, the loop tests and index increments all behave as in sequential `for` loops, but the calls to `check_collision` proceed in parallel.

## 3 Objects, Data Abstraction, and Concurrency

The core of object-oriented programming is building abstractions – encapsulated data and operations upon it which define a well-specified interface. These operations perform logically atomic updates to the abstraction's state; and each operation must maintain the consistency of that state. Support for such abstractions in sequential languages is well understood [30, 34], but the situation is more complex for concurrent languages [1, 43, 17, 2, 4, 9, 24, 22]. Concurrency allows only a partial order on state updates, complicating the notion of consistency. Any concurrent model must preserve the notion of logically atomic operations upon an abstraction in a concurrent setting.

Concurrency models must also be designed with programmability and single-source maintenance in mind. For instance, a simple model in which every member function had exclusive access to the object for its duration would, in a naive implementation, make nested calls deadlock. This would naturally entail much programming around the language and drastic changes to any conceivable sequential source. Even if directly nested calls were allowed, all concurrent calls upon a single object would be sequentialized; this is, in our experience, a very burdensome restriction and a fruitful source of deadlock.

The ICC++ object concurrency model thus aims to maintain logically atomic operations, while being as permissive as possible. The model has three elements: an object consistency model and mechanisms for extending that model over multiple objects, for building larger data abstractions, and across multiple abstractions.[4]

### 3.1 Object Data Abstractions

To support object abstractions, ICC++ defines an *object consistency model* and *object concurrency guarantees*. The consistency model preserves the notion of logically atomic operations by ensuring that method calls do not disrupt each other. Since this model is defined by the language, it does not depend on usage conventions for correctness as in [8, 24, 9, 40, 6]. Concurrency guarantees define which member calls will run concurrently, allowing programmers to reason about progress and deadlock.

**Object Consistency** In ICC++, concurrent method invocations on an object are constrained such that intermediate object states created within a member

[4] This concurrency model has been designed with the *Inheritance Anomaly* in mind, but we defer discussion of that issue to Section 6.

function are not visible. In essence, this means that their effect on the member variables is as if the member functions operated one after another. This preserves the same notion of consistency as for sequential objects: a series of member calls each leaving the object in a consistent state. Nested calls (i.e. calls on `this`) are an extension of the caller for concurrency control purposes. Finally, direct access to object state from outside member functions (e.g. `a->field_name`) are subjected to the same consistency model through implicit accessor members.

**Object Concurrency Guarantees** Concurrency guarantees enable a programmer to reason about concurrency to ensure progress. ICC++ guarantees that all member function calls for which the order of execution explicitly cannot affect final object state will execute concurrently. This is determined by syntactic examination: each member function is examined to determine which member variables it might read and write, either in itself or transitively through calls upon `this`. A given pair of member functions are guaranteed to be concurrent if neither can read any member variable the other may write. Specific examples for which concurrency is guaranteed include two methods which share no member variables and those that employ read-only sharing of member variables.

**Examples** The following simple class provides an example of how the concurrency model works.

```
class Particle {
  double mass;
  double x_vel, y_vel, x_pos, y_pos;
  double x_vel_1, y_vel_1;
  PListElt *my_neighbors;

 public:
  void check_collision(Particle *other) {
    if (collision_test(x_pos, other->x_pos, y_pos, other->y_pos))
      x_vel_1 += other.x_vel * (other.mass/mass);
      y_vel_1 += other.y_vel * (other.mass/mass);
    }
  }

  void update(void) {
    x_vel = x_vel_1;
    y_vel = y_vel_1;
    x_pos += x_vel;
    y_pos += y_vel;
  }
};
```

The idea behind `check_collision` is that two particles collide if they are closer than `epsilon` apart, and so affect each others' velocity. `check_collision` will be called for each pair of particles, and then `update` will be called to change each particle's velocity and position.

```
Particle particles[PCount];

conc for(int i = 0; i < PCount; i++)
  conc for(int j = 0; j < PCount; j++)
    particles[i].check_collision(particles[j]);

conc for(int i = 0; i < PCount; i++)
  particles[i].update();
```

The combination of consistency and concurrency guarantees assures that concurrent calls to `check_collision` for each of a particle's neighbors do not create race conditions updating `x_vel_1` and `y_vel_1`, and that there is no deadlock when two particles call `check_collision` on each other.

### 3.2 Object Ensemble Abstractions – `integral`

Dynamic data abstractions are often most conveniently implemented as ensembles of objects such as trees, networks and other pointer-based structures. Concurrency control must allow consistency to be maintained upon such multi-object abstractions. The `integral` type specifier, when applied to a member variable, extends object abstraction consistency to include all references to that member variable. In short, all references to it will be considered read/write operations on that field, providing local serialization. Note that this does not prevent interference, since the object whose reference is declared `integral` could be shared.

### 3.3 Composing Abstractions Procedurally – `friend`

A fundamental aspect of coordinating concurrent activities is the need to perform coordinated updates across several distinct abstractions [21, 31], involving some form of transactions. `friend` functions in C++ are considered member functions upon all arguments for which they are friends, and thus `friend` functions in ICC++ can be used to procedurally compose operations on several objects into a single consistent operation subject to the same object consistency and concurrency guarantees as above. That is, the `friend` function will be consistent with respect to all of its arguments for which it operates as a friend.

### 3.4 Examples

A queue abstraction illustrates composing multiple objects into an abstraction, requiring the use of `integral`.

```
class Queue {
  integral queueElt *head;
  integral queueElt *tail;

 public:
  Queue(void) {
```

```
class queueElt {
  friend class Queue;
  queueElt *next;
  integral queueElt *prev;

  void set_prior(queueElt *e) {
```

```
    head = new queueElt;
    tail = new queueElt;
    head->next = tail;
    tail->prev = head;
  }

  void enqueue(queueElt *elt) {
    head->next->set_prior(elt);
    head->next = elt;
    elt->prev = head;
  }

  queueElt *dequeue(void) {
    return tail->prev->unsnap();
  }

  friend bool operator<(Queue&, Queue&);  };
};
```

```
    prev = e;
    prev->next = this;
  }

  queueElt *unsnap(void) {
    if (!prev || !next)
      return NULL;
    else {
      prev->next = next;
      next->prev = prev;
      next = NULL;
      prev = NULL;
      return this;
    }
  }
```

Since `enqueue` and `dequeue` work only with the `head` and `tail` member respectively, items can be inserted and removed from the queue concurrently. In fact, `enqueue` and `dequeue` are *guaranteed* to be concurrent. `dequeue` returns `NULL` when the queue is empty. When the queue is empty or nearly so, `enqueue` and `dequeue` operate upon the same list elements, and care must be taken to avoid race conditions. When the queue is empty (having only head and tail dummies) `enqueue` and `dequeue` both access the head object, and `dequeue` will always return `NULL`. When there is one item, `unsnap` and `set_prior` will not interfere on that object, preserving the queue's consistency.[5] When there are two items, `set_prior` and `unsnap` access the same element concurrently, but touch different parts of its state. The `integral` declarations on `head` and `tail` prevent multiple `enqueue`s or `dequeue`s from interleaving.

```
bool operator <(Queue &left, Queue &right) {
  int length = 0;

  for(QueueElt *ptr = left.head->next; ptr != left.tail) {
    length++; ptr = ptr->next; }

  for(ptr = right.head->next; ptr != right.tail) {
    length--; ptr = ptr->next; }

  return (length < 0);
}
```

This function takes and compares the lengths of the two queues; taking the length of a `Queue` object requires exclusive access to the queue to prevent insertions and deletions while the length is being taken, which could result in

[5] the `integral` declaration on `queueElt::tail` prevents `set_prior` and `unsnap` from interleaving their updates to `prev`.

wrong answers. However, a comparison requires the lengths of two queues, and `friend` allows the `operator <(Queue *, Queue *)` to prevent disruptions to either queue while the lengths are being measured.

### 3.5 Discussion

The concurrency control model for objects is critical not only for programmability, but for execution efficiency as well. Our concurrency control model is described in terms of visible state changes, rather than locking or exclusivity, to allow the compiler to optimize concurrency control. In the absence of an operational view of locking (and the lock granularity), concurrency guarantees are also necessary. Declaring intermediate states to be invisible naturally makes objects thread-safe – allowing object state to be safely cached in registers under compiler control. Our consistency model could be implemented by monitors or many variants of read/write locking.

Composing objects into larger abstractions structurally and procedurally is a difficult problem that recurs in virtually all concurrent systems. While transactions and nested transactions provide an elegant, flexible model, they are far too expensive for object-level concurrency. For fine-grained concurrency, overhead of a few instructions is all that can be tolerated for the common case. Our `integral` mechanism meets this cost constraint. The `friend` mechanism is more expensive, potentially requiring remote locking, and is included as a building block for when such expensive structures are really essential.

## 4 Arrays, Objects, and Collections

Arrays are important both for concurrency and data distribution in many concurrent programs. ICC++ provides *collections* which compatibly extend C++ arrays, integrating them into the object model. This allows array-level functionality to be expressed as members of an array class. These collection classes support a wide variety of concurrency patterns, from a data-parallel array model to more complex concurrent abstractions. They are related to collections in pC++ [29] when used for data parallelism, but each element can access the entire collection, allowing them to implement more complex composite behavior as well. Finally, collections allow distributions to be explicitly specified (see [23]).

### 4.1 Defining Object Collections

Collections are defined with standard class declarations, with the addition of `[]` to the class name and are declared just as arrays. This declaration creates separate classes for the elements, called `type`, and for the collection itself, called `type[]`. Member variables and functions can be defined for both classes; declarations for the entire collection use explicit type qualification. Each element in a collection has a private set of the element members, and the collection members form a separate object which is shared across all elements. Collections can be

nested, and intermediate levels are both collections themselves and elements of the enclosing level.

```
// implicit definition
double grid_cell_size;
class GridCell {
  Particle particles[];
  int particle_count;
} grid[50];
```

```
// explicit definition
class Grid_1D[] {
  Particle particles[];
  int particle_count;
  double Grid_1D[]::cell_size;
} grid[50];
```

The above code shows two ways to create a grid collection. The implicit definitions correspond to traditional arrays, but the explicit definition exposes the two classes `Grid_1D[ ]` and `Grid_1D`, allowing the variable `cell_size` to be declared as a collection member. Note that collection objects are declared just like arrays.

Collections may be nested just like arrays, such classes being declared with multiple sets of [ ]. The inner levels of such nested collections are both collections themselves and elements of their enclosing collection. Also, since collections are classes, they support derivation. It works class-wise so that `derived` inherits from `base`, `derived[ ]` inherits from `base[ ]` and so forth.

### 4.2 Data-Parallel Collections

The simplest use of collections is to express data parallelism, by which we mean synchronous application of the same method to every element of the collection. This construes data parallelism as object-parallel as in pC++[29], rather than as the vector operations common in data parallel FORTRAN. The ability to declare collection-level functionality as collection members allows collections to encapsulate data parallelism beneath a collection interface.

```
class Grid[][] {
  Particle particles[];
  int particle_count;
  double Grid[][]::cell_size;

 public:
  void update(void) {
    conc for(int i = 0; i < particle_count; i++)
      particles[i]->update();
  }

  void check_collisions(void) {
    conc for(int i = 0; i < particle_count; i++)
      conc for(int j = 0; j < particle_count; j++)
        particle[i]->check_collision(particle[j]);
  }

  void Grid[][]::do_all(void (Grid::*op)(void)) {
```

```
    conc for(int i = 0; i < size(); i++)
      conc for(int j = 0; j < (*this)[0].size(); j++)
        (*this)[i][j].*op();
  }
};
```

The `Grid` elements `check_collisions` member calls `check_collision` for each particle pair in the grid cell. The `Grid::update` member calls `update` for each particle in the cell. The `size()` member functions is predefined for all collection classes, and returns the number of elements in the collection. Data parallel calls across the collection would use the `Grid[][]` member function `do_all` to fan a given `Grid` member function out across each element. Such collection members allow vector operation syntax like that in pC++ or Dome[5] to be used, as shown below.

```
Grid grid[50][50];

for(int i = 0; i < NUM_ITERATIONS; i++) {
  grid.do_all(Grid::check_collisions);
  grid.do_all(Grid::update);
}
```

### 4.3 Concurrent Abstractions

Collections are a convenient way of expressing distributed abstractions which present a concurrent interface. Collections have predefined members which give elements access to the entire collection: `index()` yields an element's position in the collection, `size()` returns the collection's size and `<type>[]::this` refers to the collection object itself.[6]

```
template<class Element>
class MultiSet[] {
  Element elts[];
  int elt_count;

 public:
  Element add_elt(Element elt) {
   return elts[elt_count++] = elt;
  }

  int find_elt(Element elt) {
   return MultiSet[]::this->find_elt(elt);
  }

  int find_elt_internal(Element elt) {
   int count = 0;
   for(int i = 0; i < elt_count; i++)
```

[6] There are other predefined collection members [23]

```
    if (elts[i] == elt) count++;
   return count;
  }

  int MultiSet[]::find_elt(Element elt) {
   int count = 0;
   conc for(int i = 0; i < size(); i++)
    count += (*this)[i].find_elt_internal(elt);
  }
};
```

The `MultiSet` abstraction is a distributed multi-set in which different elements are stored in each collection element. Elements are inserted into specific elements and looking up an element therefore must look across the entire collection. The ability of elements to access the entire collection allows the `MultiSet` elements to cooperatively implement a concurrent interface to the abstraction: multiple calls to `add_elt` can proceed simultaneously when called upon different elements of the collection, as shown below.

```
MultiSet<int> set[17];

conc for(int i = 0; i < 100; i++)
 set[i%17].add_elt(i);
```

Note that the use of templates and collections allows the `MultiSet` to be a reusable abstraction that presents a concurrent interface. This combination supports reusable libraries of concurrent abstractions.

### 4.4 Discussion

Collections in ICC++ represent a unification of collections as distributed arrays of objects as in [29, 10] and the aggregate approach as in [18]. The array approach is more compatible with the preexisting C++ notion of arrays and offers the advantage of separating the collection and constituent types. This can allow distinct members to be defined upon each type. A drawback to the independence of the types is that the element members have no primitive mechanism to refer to the whole collection, making it harder to implement concurrent abstractions like the `MultiSet` above. The aggregate approach supports cooperation amongst the constituents, but by combining the array and constituent types, it complicates deriving collections from classes. By creating two *distinct* but *related* types, ICC++ collections combine the advantages of both approaches.

This approach to collections, by integrating arrays into the object model, also divorces arrays and pointers[7], which has two benefits. First, it increases type safety by preventing the confusion of arrays of objects and pointers to single object allowed by using pointers for arrays. Second, and more importantly, it eliminates one difficulty for program analysis: pointer arithmetic.

[7] This necessitates some changes to C++ syntax [23]

## 5 Extended Example

The constructs described so far are designed to allow flexible expression of concurrency and incremental parallelization of existing programs. We illustrate how these constructs are used in a simple distinct element example, in which particles are moving about in space, colliding with one another. The overall program moves the particles around the grid for a succession of time steps. Each iteration consists of three phases: checking for collisions, updating the particles' positions and finally moving the particles between grid cells.

```
Grid grid[][];

for(int i = 0; i < TIME_STEPS; i++) {
  grid->do_all(Grid::check_collisions);
  grid->do_all(Grid::update);
  grid->do_all(Grid::regrid);
}
```

The top-level loop is simple because each phase has been encapsulated as one data-parallel operation across the entire collection `grid`. The three phases proceed as follows.

**handling collisions** applies a test for contact among particles. All forces imparted by collisions are also calculated. The code for this phase was presented in Section 4.2; recall that parallelism was exposed both across particles, and across collisions. It was noted in [13] that an auxiliary contact list had to be generated for each particle to vectorize this loop, but ICC++'s more flexible concurrency model makes this unnecessary.

**updating particles velocities and positions** merely involves updating the velocities of the particles with the forces calculated from the collisions and changing their positions based upon their velocities. The code for this phase was also presented in Section 4.2, and is a straightforward data-parallel operation across the particles.

**regriding** involves moving particles from one grid cell to another as their positions change over time.

```
void Particle::regrid(Grid *cell) {
  int old_col = cell.index();
  int old_row = (*cell.Grid[]::this).index();
  double size = cells.size;
  int new_row = x_pos / size;
  int new_col = y_pos / size
  if (new_row != old_row || new_col != old_col) {
    cell.remove(this);
    (*cell.Grid[][]::this)[new_row][new_col].add(this);
  }
}
```

```
void Grid::regrid(void) {
  for(int i = 0; i < particle_count; i++)
    particle[i].regrid(this);
}
```

This code is similar in structure to the `update` code, being a data parallel operation across the particles. However, the need to move particles between grid cells causes more complex patterns of concurrency control; multiple `add` and `remove` calls will be made to grid cells, and the consistency model ensures that they will not result in race conditions.

Thus all phases can be parallelized with ICC++. Contrast this with [13], where part of the particle interaction phase (the contact list generation) was completely sequential in the data-parallel version, limiting the potential speedup.

# 6 Discussion and Related Work

## 6.1 C++ Compatibility

In the design of ICC++, our intention was to avoid any gratuitous incompatibilities with sequential C++ programs. As a result, large sections or even entire C++ programs can be incorporated directly as ICC++ programs. However, there are two important differences. First, ICC++ eliminates pointer arithmetic, requiring explicit array type declarations for collections (encapsulated arrays). This change ensures that pointers are not used to point into arbitrary locations, which would reduce the effectiveness of aggressive compiler analysis techniques. Second, to support concurrent abstractions, objects must have well-defined concurrency control semantics. Our semantics ensures that concurrency control overhead is low (avoiding expensive callback checking), but can deadlock in cases of mutual recursion. External C++ functions can be called easily from ICC++, and bidirectional interoperability will be achieved with CORBA IDL bindings of ICC++ and C++. In summary, we believe ICC++ will allow many programs to be migrated from C++ with with modest effort, and the resulting ICC++ programs can be converted into legal C++ programs with purely mechanical transformations.

While our experience with ICC++ so far confirms that most C++ programs can be simply adapted, the standard template library poses particular problems. The draft C++ standard requires that container classes provide references to elements; since ICC++ does not allow pointers or references to built in types, this means that containers for such types cannot be constructed in ICC++. Forward, bidirectional and random access iterators have the same constraint. Thus, classes such as `list<int>` or `forward_iterator<int>` require special treatment. We are exploring alternate implementations for such classes, using helper classes.

## 6.2 Derivation and Concurrency

The *Inheritance Anomaly* threatens to undermine the concurrent object-oriented approach by turning two basic mechanisms against each other: derivation and

concurrency. The problems described by [32] involve the *sequencing* of messages; that is, when certain messages may be handled by a given object. In a sequential object-oriented language, when one tries to dequeue from an empty queue, the dequeue generates an error rather than being *delayed* until such time as there is something to get, as in the examples in [32]. Due to its C++ heritage, ICC++ takes the same approach, and there is no notion of messages being delayed until they can be handled. All synchronization must be ensured through control structures, such as sequential blocks, provided by the language.

### 6.3 Parallel C++ Efforts

The many approaches to parallel C++ can be divided into two categories: data-parallel and task-parallel extensions. Data parallel extensions of C++ [29, 28] employ collections or aggregates [17, 39] to describe parallelism, using objects to increase the flexibility of the data parallel model. However, data parallel languages cannot easily express more irregular and client-server forms of concurrency, limiting their domain of applications. Rewriting sequential programs as efficient data parallel programs often requires significant reorganization, as efficient alignment into parallel collections can cause major program structure disruptions.

The diversity of task-parallel extensions of C++ is much greater and can be loosely categorized based on their treatment of objects and concurrency. First, there are languages (or libraries) that introduce concurrency without changing the object model [6, 8, 40, 24]. These systems require the programmer to build concurrency control by convention, providing no language support for object consistency or for building abstractions from larger collections of objects. Second, many languages (or libraries) use objects to encapsulate concurrency, exploiting objects to represent data parallel collections or coarse-grained tasks [5, 22]. In these languages concurrency control may be expressed explicitly in a library, or implicitly via data flow dependences [22]. Concurrency in these models is generally expensive, and used only sparingly for coarse-grained abstractions. Finally, Compositional C++ provides `atomic` functions, but these are only useful for individual objects – they are not allowed to access another object – and hence can only be used to build single object data abstractions.[8] In contrast to these language designs, ICC++ provides an object model that integrates both concurrency control and concurrency guarantees, and is extensible, supporting concurrent data abstractions built with several objects.

Another important distinction amongst parallel C++'s is the scheduling or concurrency guarantees provided by the language. Data parallel languages have sequential semantics, so the data parallel C++'s provide no concurrency guarantees. Of the task parallel C++ dialects, Charm++ provides explicit control over scheduling [24], and Compositional C++ [9] provides guaranteed fair thread

[8] MPC++ is another parallel C++ dialect worthy of mention, but since MPC++ provides a programmable language syntax and semantics it is difficult to make specific comparisons.

scheduling for all `par` constructs. In contrast, ICC++ emphasizes the annotation of *potential concurrency*, and gives concurrency guarantees in an data-oriented form. This gives the implementation freedom to select an execution granularity (thread sizes) for efficiency, facilitating efficient sequential execution.

### 6.4 Other Concurrent Object-Oriented Languages

Though there are a wide variety of non-C++ concurrent object-oriented languages [43, 2, 17, 33, 27, 3], we focus on Actor-based languages [1] because they closely integrate the notion of actors (objects) and concurrency. This allows programmers to reason at the level of object-operation. However, the actor model provides no clear basis for building data abstractions from collections of objects, and the actor model provides no concurrency guarantees. In contrast, ICC++ includes both concurrency guarantees and language support for building abstractions from ensembles (structures or collections) of objects. In addition, to date most of the Actor based languages have been inefficient in implementation. Recent work in our group [37, 38, 36] and others [41, 26] demonstrates that actor languages need not be inefficient.

### 6.5 Illinois Concert Project

ICC++ is the second language supported by the Concert project (the first is *Concurrent Aggregates* [17, 11]). The Illinois Concert system is a complete development environment for irregular parallel applications [19]. It supports a concurrent object-oriented programming model and includes a globally optimizing compiler, efficient runtime, symbolic debugger, and an emulator for program development. This system employs novel compiler techniques [37, 38, 36] and runtime techniques [38, 25] to achieve efficient execution of fine-grained programs on both sequential and parallel platforms. The Concert system has demonstrated sequential performance matching C and surpassing C++ on demanding numerical benchmarks such as the Livermore Kernels [38], and superior speedups and high absolute performance on a parallel molecular dynamics application (CEDAR [7]) on the the Cray T3D [20] and Thinking Machines CM-5 [42]. The implementation of ICC++ which has just become operational exploits the same aggressive compiler analysis and code optimization, so we expect similar performance in the near future.

## 7 Summary

ICC++ is a new C++ dialect designed to support both efficient sequential and parallel execution. By allowing concurrency to be introduced incrementally, ICC++ allows sequential and parallel program versions to be maintained with single source and permits convenient expression of irregular and fine-grained concurrency. By defining a simple object consistency model and a flexible set of extensions, ICC++ supports the construction of concurrent data abstractions.

Distributed data abstractions are further supported with the notion of collections – a compatible extension of arrays. Finally, by focusing on programmer annotation for *potential* concurrency, not actual concurrency, ICC++ allow the system to optimize execution granularity to match the underlying machine, providing both high performance sequential and parallel execution (on both distributed memory and shared memory systems).

ICC++ has been implemented based on the compiler and runtime technology extant in the Concert system, and future work will include not only extensive performance benchmarking, but also evaluation of the language and system through building several large-scale applications.

## References


1. G. Agha. Concurrent object-oriented programming. *Communications of the Association for Computing Machinery*, 33(9):125–41, September 1990.
2. Pierre America. A parallel object-oriented language with inheritance and subtyping. In *Proceedings of ECOOP/OOPSLA '90*, pages 161–8, 1990.
3. Birger Andersen. A genereal, grain-size adaptable, object-oriented programming language for distributed computers. Technical report, Department of Computer Science, University of Copenhagen, Copenhagen, Denmark, June 1992. Ph.D. thesis (partial).
4. Henri E. Bal. *The Shared Data-Object Model as a Paradigm for Programming Distributed Systems*. PhD thesis, Vrije Universiteit Te Amsterdam, Amsterdam, 1989.
5. Adam Beguelin, Erik Seligman, and Micheal Starkey. Dome: Distributed object migration environment. Technical Report CMU-CS-94-153, School of Computer Science, Carnegie-Mellon University, May 1994.
6. B.N. Bershad, E.D. Lazowska, and H.M. Levy. Presto: A system for object-oriented parallel programming. *Software — Practice and Experience*, 18(8):713–732, August 1988.
7. M. Carson and J. Hermans. *Molecular Dynamics and Protein Structure*, chapter The Molecular Dynamics Workshop Laboratory, pages 165–6. University of North Carolina, Chapel Hill, 1985.
8. Rohit Chandra, Anoop Gupta, and John L. Hennessy. Data locality and load balancing in COOL. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993.
9. K. Mani Chandy and Carl Kesselman. Compositional C++: Compositional parallel programming. In *Proceedings of the Fifth Workshop on Compilers and Languages for Parallel Computing*, New Haven, Connecticut, 1992. YALEU/DCS/RR-915, Springer-Verlag Lecture Notes in Computer Science, 1993.
10. S. Chatterjee. Compiling nested data parallel programs for shared memory multiprocessors. *ACM Transactions of Programming Languages and Systems*, 15(3), 1993.
11. A. A. Chien and W. J. Dally. Concurrent Aggregates (CA). In *Proceedings of Second Symposium on Principles and Practice of Parallel Programming*. ACM, March 1990.
12. A. A. Chien, W. Feng, V. Karamcheti, and J. Plevyak. Techniques for efficient execution of fine-grained concurrent programs. In *Proceedings of the Fifth Workshop*

*on Compilers and Languages for Parallel Computing*, pages 103–13, New Haven, Connecticut, 1992. YALEU/DCS/RR-915, Springer-Verlag Lecture Notes in Computer Science, 1993.
13. A. A. Chien, M. Straka, J. Dolby, V. Karamcheti, J. Plevyak, and X. Zhang. A case study in irregular parallel programming. In *DIMACS Workshop on Specification of Parallel Algorithms*, May 1994. Also available as Springer-Verlag LNCS.
14. Andrew Chien, Vijay Karamcheti, and John Plevyak. The Concert system – compiler and runtime support for efficient fine-grained concurrent object-oriented programs. Technical Report UIUCDCS-R-93-1815, Department of Computer Science, University of Illinois, Urbana, Illinois, June 1993.
15. Andrew Chien and Uday Reddy. ICC++ language definition. Concurrent Systems Architecture Group Memo, Also available from http://www-csag.cs.uiuc.edu/, February 1995.
16. Andrew A. Chien. Application studies for concurrent aggregates. Technical report, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Cambridge, Massachusetts, 1990.
17. Andrew A. Chien. *Concurrent Aggregates: Supporting Modularity in Massively-Parallel Programs*. MIT Press, Cambridge, MA, 1993.
18. Andrew A. Chien and William J. Dally. Experience with concurrent aggregates (ca): Implementation and programming. In *Proceedings of the Fifth Distributed Memory Computers Conference*, Charleston, South Carolina, April 8-12 1990. SIAM.
19. Andrew A. Chien and Julian Dolby. The Illinois Concert system: A problem-solving environment for irregular applications. In *Proceedings of DAGS'94, The Symposium on Parallel Computation and Problem Solving Environments.*, 1994.
20. Cray Research, Inc., Eagan, Minnesota 55121. *CRAY T3D Software Overview Technical Note*, 1992.
21. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, California, 1993.
22. A. Grimshaw. Easy-to-use object-oriented parallel processing with Mentat. *IEEE Computer*, 5(26):39–51, May 1993.
23. Concurrent Systems Architecture Group. The ICC++ reference manual. Concurrent Systems Architecture Group Memo, June 1995.
24. L. V. Kale and Sanjeev Krishnan. CHARM++: A portable concurrent object oriented system based on C++. In *Proceedings of OOPSLA'93*, 1993.
25. Vijay Karamcheti and Andrew Chien. Concert – efficient runtime support for concurrent object-oriented programming languages on stock hardware. In *Proceedings of Supercomputing'93*, 1993.
26. Woo Young Kim and Gul Agha. Efficient support for location transparency in concurrent object-oriented programming languages. In *Proceedings of the Supercomputing '95 Conference*, San Diego, CA, December 1995.
27. H. Konaka. An overview of ocore: A massively parallel object-based language. Technical Report TR-P-93-002, Tsukuba Research Center, Real World Computing Partnership, Tsukuba Mitsui Building 16F, 1-6-1 Takezono, Tsukuba-shi, Ibaraki 305, JAPAN, 1993.
28. James Larus. C**: a large-grain, object-oriented, data parallel programming language. In *Proceedings of the Fifth Workshop for Languages and Compilers for Parallel Machines*, pages 326–341. Springer-Verlag, August 1992.
29. J. Lee and D. Gannon. Object oriented parallel programming. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. IEEE Computer Society Press,

1991.
30. Barbara Liskov. Data abstraction and hierarchy. *ACM SIGPLAN Notices*, 23(5):17–34, May 1988.
31. Barbara Liskov. Distributed programming in argus. *Communications of the ACM*, 31(3):300–313, March 1988.
32. S. Matsuoka and A. Yonezawa. *Research Directions in Object-Based Concurrency*, chapter "Analysis of Inheritance Anomaly in Object-Oriented Concurrent Languages". MIT Press, 1993.
33. Stephan Murer, Jerome A. Feldman, Chu-Cheow Lim, and Martina-Maria Seidel. pSather: Layered extensions to an object-oriented language for efficient parallel computation. Technical Report TR-93-028, International Computer Science Institute, Berkeley, CA, June 1993 November 1993.
34. N. Wirth and M. Reiser. *Programming in Oberon – Steps beyond Pascal and Modula*. Addison Wesley, 1992.
35. T. Ng, X. Zhang, V. Karamcheti, and A. A. Chien. Parallel macromolecular dynamics on the Concert System. In *Submitted for publication*, 1995.
36. John Plevyak and Andrew Chien. Efficient cloning to eliminate dynamic dispatch in object-oriented languages. Submitted for Publication, 1995.
37. John Plevyak and Andrew A. Chien. Precise concrete type inference of object-oriented programs. In *Proceedings of OOPSLA'94, Object-Oriented Programming Systems, Languages and Architectures*, pages 324–340, 1994.
38. John Plevyak, Xingbin Zhang, and Andrew A. Chien. Obtaining sequential efficiency in concurrent object-oriented programs. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 311–321, January 1995.
39. G. Sabot. *The Paralation Model*. MIT Press, Cambridge, Massachusetts, 1988.
40. K. Smith and A. Chatterjee. A C++ environment for distributed application execution. Technical Report ACT-ESP-015-91, Microelectronics and Computer Technology Corporation (MCC), November 1990.
41. K. Taura, S. Matsuoka, and A. Yonezawa. An efficient implementation scheme of concurrent object-oriented languages on stock multicomputers. In *Proceedings of the Fifth ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*, 1993.
42. Thinking Machines Corporation, 245 First Street, Cambridge, MA 02154-1264. *The Connection Machine CM-5 Technical Summary*, October 1991.
43. Akinori Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System*. MIT Press, 1990. ISBN 0-262-24029-7.

This article was processed using the LaTeX macro package with LLNCS style