Controlling Concurrency and Expressing Synchronization in Charm++ Programs

Laxmikant V. Kale, Jonathan Lifflander
{kale, jliffl2}@illinois.edu

University of Illinois at Urbana-Champaign

Abstract. Charm++ is a parallel programming system that evolved over the past 20 years to become a well-established system for programming parallel science and engineering applications, in addition to the combinatorial search applications with which it started. At its earliest point, the precursor to Charm++, the Chare Kernel, was a purely reactive specification, similar to most actor languages. This paper describes the evolution of a series of concurrency control mechanisms that have been deployed in Charm++ to tame this unrestricted concurrency in order to improve code clarity and/or to improve performance.

1 Introduction

One of the challenges in parallel programming, especially in science and engineering applications, is resource management. This is especially true for dynamic and irregular applications, such as those involving dynamic adaptive mesh refinements. Newer machines, with issues of power and component failures, also create related challenges. A programming system supported by a smart adaptive runtime system that automates resource management is therefore desirable.

Charm++ is a concurrent-objects parallel programming system that has been used for programming science and engineering applications. With Charm++, one programs in C++, providing a few additional declarations to facilitate parallel mechanisms such as asynchronous method invocations. A Charm++ computation consists of a number of C++ objects that interact via asynchronous method invocations. An adaptive runtime system controls and dynamically changes assignments of objects to processors, and also chooses the sequence in which ready method invocations will execute on a given processor. These control mechanisms empowers the runtime to automate load balancing, as well as implement other resource management policies.

In this paper, we will focus on how concurrency and synchronization *within* an individual object is expressed. We present these concepts, which have been described in earlier literature [14, 28] going over 20 years by us, in a pedagogical and historical sequence, with illustrations from recent case studies.

We begin this paper with a brief history of the beginning of Charm++, to elucidate the evolution of its constructs, and to set the context for the description of concurrency control mechanisms in Section 3. From 1983 to 1985, a new parallel execution model for logic programming was developed, called the Reduce-Or

process model [26]. This allowed a relatively novel combination of AND and OR parallelism. In particular, the main innovation was something called *consumer-instance parallelism*. Given a Horn clause such as `p(X), q(Y), r(X,Y)`, this model was able to exploit independent-AND parallelism between p and q literals, the OR parallelism underneath p (as well as q), and also the parallelism between multiple instances of r created by incrementally joining solutions to p and q. This was accomplished by having the activation record for the clause as a persistent object. As each solution for p (or q) was returned to it, it combined them with the stored (already-received) solutions to q (or p), and fired a task for computing each instance (the so-called "consumer-instance") of r so created. Initially, as a part of a PhD thesis [25], this model was implemented in an interpreted mode, working with Prof. David Scott Warren. Later, byte-code compilation [41] and related optimizations were developed. However, for the theme of this paper, the interesting part is the runtime system itself. The runtime system needed to have a dynamic load balancer, to distribute all the goals across processors, especially as the distributed memory architectures (such as the "hypercubes", including NCUBE, and iPSC/2) were targeted. It also needed prioritization to focus the search on the most promising paths.

The main "applications" considered in the development of ROLOG (as the compiled implementation of Reduce-Or Process Model was called) involved combinatorial search, including N-queens, Knight's-tour, graph coloring, etc. [24, 31]. Our interest shifted to the applications themselves, rather than the logic programming language used to express them. Consequently, the speed of finding a solution became an important goal in itself. These developments led to extraction of the runtime system into a separate entity, called the *chare kernel* [32]. This was a C-based parallel programming system. ROLOG itself was implemented on top of the chare kernel.

The term *chare* was borrowed from an earlier project on parallel implementation of functional languages called *RediFlow* [33] by Keller, Lindstrom, et al.; *chare* means a small task or a *chore* in old English. The activation records for evaluation of a Rolog Clause mentioned above can each be implemented as a chare. In the chare kernel, a chare was an object with its own ID; it was load balanced by the system, and it was possible to send messages to a chare.

One will recognize an ABCL-style concurrent object [49, 50, 9], or an "actor" in this description immediately [1], although we came to it from the functional language implementations, and macro-dataflow ideas. The *reactive kernel* and *Cantor* [4] were other relevant contemporary systems. However, the chare was clearly very similar to the notion of a concurrent object or an actor developed earlier by Agha [2] and Yonezawa [49] et al., which built upon Hewitt's earlier work [17]. The main differences, in retrospect, were minor up to this point: a C-based implementation, reflecting an efficiency orientation, and a focus on combinatorial search applications. From the language point of view, one difference was that, unlike actors, chares did not have access to their mailboxes. They simply executed every method anyone invoked on them.

There were potentially multiple invocations that were ready on a processor, stored in a prioritized message queue. The system picked the next message from this queue, invoked it on the named object, and it selected another message *only when it returned*. Any guards or internal synchronization within a chare were the responsibility of user's code within the method. This typically led to a lot of buffering and flags indicating what is ready and what is not. The reactive notation also affected the expression of the overall flow of control. A series of solutions to this problem constitute the focus of this chapter. We return to this theme in section 3, after reviewing the somewhat orthogonal but important developments within the Charm++ model in the next section.

Note that the chare kernel was developed before C++ had really taken off. So the language (called Charm by 1991), while object-based, was translated to C by a simple translator. In 1992, with increasing popularity of C++, a C++-based version was created, and it was called Charm++.

2 Charm++ and CSE Applications

In the early 1990's, the attention of Charm++ developers shifted to applications in computational science and engineering (CSE), from the combinatorial search applications that were dominant earlier. In part because of the nature of these applications, and because of the pragmatic orientation that CSE applications necessitated, several new features and language constructs were developed that improved expressiveness of Charm++ in comparison with the plain Chare Kernel as well as the Actor languages of that time.

The first of this was the notion of organizing the chares into indexed collections. This followed naturally from the need to support domain decomposition methods used in CSE. Consider a two-dimensional decomposition of a 2D domain in fluid dynamics. A single chare is responsible for one chunk of this decomposition. It needed to communicate (its borders) with the four neighboring chares. But what does "neighboring" mean? In the plain Charm of that time, one would have to create a network of chares, and pass IDs from one to the other in complex manner to ensure that everyone had the IDs of the four chares they needed. The need for an indexed organization was anticipated and developed in early work by Sanjeev Krishnan and Joshua Yelon [42, 48]. These ideas were developed into the notion of a "chare array": an indexed collection of chares [37]. Although they were called "arrays", the index could be quite general, supporting sparse arrays as well as collections indexed by bit-vectors or even strings. A program (or more accurately, a computation) consisted of one or more chare array. These were typically created at the beginning of the computation by a "main" chare, but they could also be created dynamically in the middle of the computation.

Method invocation was directed to an individual member of the collection: `A[i].foo(x,y)` caused an asynchronous method invocation ("asynchronous" in that it returned immediately to the caller) being sent to the i'th member of the collection whose ID was represented by "A". The system took charge of global

location management via a scalable scheme [37], so that it could identify the processor on which the named chare lived, and deliver the message to it.

For plain chares, their "seeds" (the messages containing the constructor arguments) were moved around by the load balancers; but once they took root (i.e. were installed on a processor, and executed their constructor), they were not allowed to migrate. For combinatorial search applications, where new chares were created all the time, this was a reasonable strategy. In contrast, chare array members were allowed to migrate. This allowed CSE applications to be load balanced dynamically. Observing that these applications tended to exhibit the *principle of persistence* [30], a suite of measurement-based load balancing strategies [8] were developed that periodically re-examine the load and migrate chares to restore balance. Research on such adaptive load balancers continues to date, and Charm++ provides an excellent proving ground for new load balancing ideas.

The adaptive runtime system, of which the load balancers are a part, has continued to evolve. It now supports features such as automatic checkpointing [52, 40], communication optimizations [34], fault tolerance, and power-and-temperature optimizations [43].

The Charm++ model and all its constructs described so far do not have the notion of a "processor" in them. For the sake of practicality, processor-level constructs were added: the most basic of these mechanisms allowed specification on which processor to create a given chare. A more interesting example was a construct called *branch-office chare* [21] (later renamed *chare group*). A chare group consists of a set of chares such that there is exactly one chare (the "branch") on each processor. A regular chare, which does not know which processor it is on, can simply ask for a pointer to the local branch of a chare group, and invoke regular C++ methods on it. The members of the group can communicate with each other just as if they are chare array members—using the processor number as the index. This construct allowed development of many support libraries, including the load balancers mentioned above.

The base language described above does not have any global variables. Various types of global variables, based on specific modes of information sharing, were added early on to the language [44].

Several CSE applications have been developed using Charm++. NAMD for biomolecular simulations was developed in mid 1990s and has continued to evolve with Charm++. Other applications span topics such as computational astronomy [20], quantum chemistry and nanomaterials [35], agent-based simulation of contagion [5], etc. Also, several higher level languages have been developed using Charm++ [27].

2.1 Comments on the Charm++ Model

Fairness and Scheduling Strategy By default, Charm++ processes pending method invocations in FIFO order. Also, method invocations by an object on itself are explicitly specified as either in-line or asynchronous by the programmer. This thus pushes the onus on fairness to the programmer; with FIFO, all the

explicitly scheduled invocations will be executed fairly. However, Charm++ also supports prioritized queues, instead of (or in addition to) FIFO queues. In this case, the execution may not be fair. Again, the responsibility for ensuring non-starvation is mostly borne by the programmer. We have found this to be adequate for the applications we have developed so far. Further, the scheduler itself is pluggable component; so it is possible, for example, to replace it with one based on "lottery-scheduling" principles, where one selects between tasks randomly, with probabilities determined by the priority of the task.

Message Passing Semantics In Charm++, messages are passed by value by default. If the serialization methods are implemented correctly for a user-defined type, a deep copy will be made of the data being serialized. However, if Charm++ is used with shared memory, data within a node can be passed by pointer if the programmer indicates that the data should be *conditionally* packed: only packed into a message when the data leaves the node. If a method invocation is marked as conditional, the programmer must ensure that the semantics are correct (e.g. the data is only read in that method).

3 Concurrency Control within a Parallel Object

The earliest version of Charm supported a fairly flat and reactive control structure. A chare was defined by a series of "entry points" (later called "entry methods" in Charm++), in addition to a set of data members and private methods. Its behavior is specified as a set of reactions: *if* the chare gets an invocation for its entry method A, it will execute the body of method A, and so on. The concurrency in such chares is unrestrained. Such a reactive specification does not allow a clear description of the life cycle of a chare. Also, it leads to a cluttered program, with buffers, flags and counters for keeping track of where the chare is in its life-cycle. This section, which is the main topic of the paper, describes three notational mechanisms for constraining the concurrency — specifying which of the many possible actions a chare can execute will be allowed to execute — and simplifying the expression of the life-cycle of a chare.

3.1 Dagger

The Dagger notation, developed around 1993 [13], allows specification of dependencies between computational actions and messages within a chare. A dag-chare is a special type of chare that supports such specification. A chare definition consists of a set of computational blocks called *when-blocks*. Each when-block is preceded by a list of dependencies. There are two kinds of dependencies: entry-method names, and condition variables. A condition-variable is set by calling `ready(condition-variable-name)`. A message sent to an entry method is not eligible to be looked at until it is *expected*. An entry method (named, say, EP) is marked as expected by calling `expect(EP)`. A when-block is ready to execute

when all the condition variables in its dependency list are set and all the entry method invocations in its dependency list are both expected and received.

A when clause has the form: when g_0 , ... , g_k : { computation } where each g_i is either a condition variable or the name of an entry method.

The collection of when clauses defines a static dataflow graph. This graph both allows and constrains concurrency within an object. It is important to remember that all the actions within a chare take place on a single processor. So, when multiple actions within an object are described as "concurrent", it does not imply any parallel execution between them.

Consider the following code fragment, based on an example from the first paper about Dagger [12]. The data-flow graph corresponding to this chare definition is shown in Figure 1.

```
dag chare C {
    // ... declarations of local variables, condition variables,
    // ... entry methods and private methods

    when init: { ... Computation C0 ... ; expect(e1); expect(e2); }

    when e1: { ... Computation C1 ... ; ready(R); }

    when e2: {... Computation C2 ... ; expect(e3); }

    when R, e3 : { ... Computation C3 ... ; }
}
```

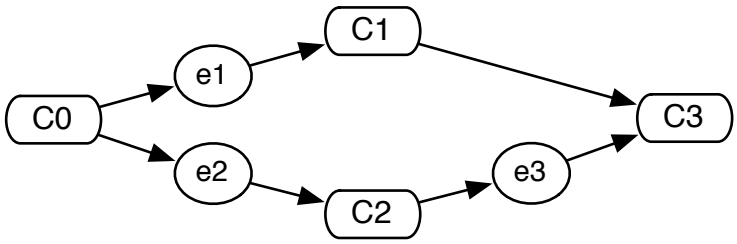

Fig. 1: Example dependency graph that could be expressed using the Dagger notation.

The final computational block `Computation3` is dependent on receipt of a message directed at entry method e3, but it also requires that condition variable R be set, and that the message for e3 be *expected*. `Computation1` and `Computation2` can be carried out in either order, depending on whether the message directed at e1 arrives before or after that directed at e2. Yet, they are not parallel computations: they both belong to the same chare, and therefore will be serialized in one of those two sequences. In general, the same behavior

can be specified with different but equivalent graphs: for example the *expect(e3)* and *ready(R)* statement can be swapped in this example.

To describe a more concrete example, consider a formulation of martix-matrix multiplication where the row-blocks of the left matrix (say A), and column blocks of the right matrix (say B) are distributed among processors, using a distributed hash table (indeed, the earliest version of Charm supported distributed hash tables, which were simply called "distributed tables" [44]). The job of a particular chare is to request one block of rows from A, one block of rows from B, multiply them out, and send the result to be stored in another distributed table. Such a formulation may be useful in a context where dynamic balancing of block-multiplication tasks is necessary.

The "reactive" code for this chare, in plain Charm, is shown below. Note the use of counters and buffers (to store the row or column block that arrived earlier).

```
chare multiplyBlock
  int count;
  float *row, *column;

  entry init: (message Work *msg) {
    count = 2;
    Find(A, msg->rowNum, getRow, myChareID());
    Find(B, msg->colNum, getCol, myChareID());
  }
  entry getRow: (message TBL_REPLY *m) {
    row = m->data;
    if (--count == 0) matmul_block(row, col);
  }
  entry getCol: (message TBL_REPLY *m) {
    col = m->data;
    if (--count == 0) matmul_block(row, col);
  }
...
```

In contrast, the same code is expressed using the Dagger notation as shown below:

```
dag chare multiplyBlock
  entry init: (message Work *msg);
  entry getRow: (message TBL_REPLY *row);
  entry getCol: (message TBL_REPLY *col);

  when init: {
    Find(A, msg->rowNum, getRow, myChareID());
    Find(B, msg->colNum, getCol, myChareID());
    expect(getRow); expect(getCol);
  }
  when getRow, getCol: { matmul_block(row->data, col->data); }
```

The Dagger code makes the dependencies clear, avoids the use of counters, and automates the buffering required. The entry declarations associate message variables (which must have distinct names) with each entry method, so the buffered data (row, col) can be accessed in the subsequent when block.

We selected these examples from the first Dagger paper, to be faithful to the original syntax. Note that at that time, a message pointer was the only parameter an entry method was allowed to have. Modern Charm++, as well as the *Structured Dagger* notation we describe next, allow more general parameters for entry methods.

Synchronization mechanisms and, in particular, the inheritance anomaly (following the phrase coined by the Rosette system [47]) in concurrent object languages have been well studied in the literature [10]. One of the most comprehensive study of the problem and possible solutions was presented by Matsuoka and Yonezawa [39]. Our approach was to simply disallow inheriting "dagger" methods (called the body methods in some of the literature, analogous to the "run" threads of Java). Other sequential methods can be inherited just as in C++, because Chares are, after all, C++ classes. In practice, this has not been a hindrance in using the Dagger or SDAG (see next section) notation. Further, the focus of much of the related work in concurrent objects community was on expressing semantic constraints on individual methods. For example, a popular example of such a constraint was: a `get` method should not be executed on a bounded buffer object if the buffer is empty. In contrast, Dagger is designed to support expression of dependence graphs between computations and messages, and the ability to better express the life-cycle of an object.

3.2 Structured Dagger

In Dagger, we allow arbitrary dependencies (a DAG) to be represented between the entry methods or message receptions for a given parallel object in the system. Although this is very powerful, we found for many real applications of Dagger that a full dependency graph is not needed. The disadvantage of a full dependency graph is that there is no natural flow to the application's code. This makes comprehending the application code and flow of the parallel application difficult and unintuitive.

In Structured Dagger (SDAG, for brevity) [29] we limit the graphs that can be expressed to those constructed with single-entry single-exit (structured) blocks. This restricts one to a set of dependencies that are either sequenced (the default) or explicitly defined to be overlappable (i.e. they do not depend on each other). Although this reduces the set of graphs that can be represented, all the real applications we have found can be represented cleanly even with this limitation. For the cases that cannot be expressed using SDAG, one can fall back to the original reactive specification method. An example of a graph that cannot be expressed in Structured Dagger, without losing concurrency is shown in Figure 2 below.

In SDAG, the fundamental construct is a `when` statement that specifies a dependency on a incoming message or set of messages. In Charm++, a message

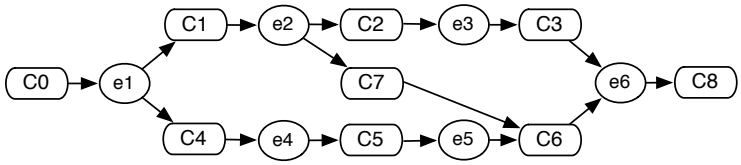

Fig. 2: An unstructured DAG that cannot be expressed in SDAG

is targeted toward a method on a certain parallel object or chare in the system. So if we want to wait for a method invocation of void foo(int param), we could specify the following:

```
when foo(int param) { /* block1 */ }
/* block2 */
```

In this case, block1 will not execute until foo arrives, and because SDAG defines a sequence (i.e. program order) by default, block2 will not execute until foo arrives and block1 executes. Note, that SDAG constructs can be nested, so block1 can specify more when constructs or other SDAG constructs.

We can also wait on more than one method to arrive by simply specifying a list of methods that we are expecting:

```
when foo(int param), bar(int size, char str[size]) { /* block1 */ }
/* block2 */
```

Since SDAG defines a sequence by default, the following code will wait for foo to arrive, execute block1, then wait for bar to arrive and execute block2:

```
when foo(int param) { /* block1 */ }
when bar(int size, char str[size]) { /* block2 */ }
```

SDAG works by buffering any messages that are not *ready* to be received. A message is ready if a when statement that matches the incoming message has been executed by the SDAG runtime. If a when statement is encountered and no message has arrived that matches that declaration, a trigger is created that acts as a continuation that can be activated when the appropriate message(s) arrive. Thus, the "ready" statement and condition variables of the plain "Dagger" described in the previous section are not needed with SDAG, there use is replaced by relying on program order, and restricting the description to structured graphs.

If we have a set of statements that are overlappable (i.e. they can be executed in any order) we can override the default sequence enforced by using the overlap statement. For example, if foo(...) and bar(...) can actually be executed in any order according to the semantics of the application, we can declare the following:

```
overlap {
  when foo(int param) { /* block1 */ }
  when bar(int size, char str[size]) { /* block2 */ }
}
```

In general, we can specify a set of SDAG constructs in an overlap that can be executed without regard to ordering. Each nested construct within the overlap will be ordered separately — so an overlap relaxes the ordering to a partial order between a set of statements.

For many applications, we have found that we need to wait on a number of messages, all of the same type. An example of this is a typical near-neighbor interaction, were we wait for some defined number of neighboring elements to send data to this object. SDAG provides a convenient syntax for declaring this interaction pattern: using a for loop when the messages constitute a sequence, or a forall when the incoming messages are allowed to be processed in any order.

The following is an example of using a SDAG for loop:

```
for (i = 0; i < 4; i++)
  when updateGhostRegion(int d, int size, double buf[size]) serial {
    updateBoundary(d, size, buf);
  }
```

Using this code, we wait for each of the neighbors' data to arrive, execute some code (possibly performing an update or saving a pointer) when each arrives and continue only when all of them have arrived.

If the application we are writing is iterative, one possible problem with the above code is that we may not explicitly synchronize between iterations. If this is the case, a neighbor message for a subsequent iteration might arrive out-of-order with the current iteration. Note the above code does not have any way of specifying which iteration we are waiting on: we only wait on some method updateGhostRegion(...) to arrive.

To make this common case much easier, we allow a when trigger to wait on a certain *reference number* that can be included with a message. In SDAG, the first integer specified in the parameter list is the reference number for that message — and can be used to make the dependency more specific:

```
when updateGhostRegion[iter](int i, int d, int size, double buf[size])
```

In the above code, we wait on a specific class of the updateGhostRegion messages — ones that are marked with the reference number iter. So in an iterative application without explicit synchronization between iterations, we would write the following code, which is an example of how a 5-point stencil computation (Jacobi relaxation, for instance) could be implemented in SDAG:

```
serial {
  prepareGhostRegions();
  thisProxy(wrapX(x + 1),y).updateGhostRegion(iter, TOP, size, topReg);
  thisProxy(wrapX(x − 1),y).updateGhostRegion(iter, BOTTOM, size, botReg);
  thisProxy(x,wrapY(y − 1)).updateGhostRegion(iter, LEFT, size, leftReg);
  thisProxy(x,wrapY(y + 1)).updateGhostRegion(iter, RIGHT, size, rightReg);
}
for (i = 0; i < 4; i++)
  when updateGhostRegion[iter](int i, int d, int size, double buf[size]) serial {
    updateBoundary(d, size, buf);
  }
serial {
  int c = doCalc() < targetDiff;
  CkCallback cb(CkReductionTarget(Tile, checkConverged), thisProxy);
  if (iter % 5 == 1) contribute(sizeof(int), &c, CkReduction::logical_and, cb);
}
if (++iter % 5 == 0) {
  when checkConverged(bool result) serial { converged = true; }
}
```

In this code segment, while the stencil computation has not converged, we prepare the ghost regions for sending, and then send a message to each neighbor with the corresponding region copied into a buffer. Then, in the following for loop, we wait to receive 4 neighboring ghost regions that have a reference number corresponding to the current iteration iter. After receiving all 4 ghost regions, we run a compute kernel doCalc and then asynchronously contribute to a reduction that logically ANDs all the local convergence decisions. We exploit asynchronous reductions in Charm++ by only contributing every 5 iterations and waiting for the result of the reduction 4 iterations later. In this way, the reduction is overlapped with the computation and we only block waiting to find out if the computation has converged every several iterations, instead of synchronizing every iteration. If the computation has converged, we set the local converged variable to true and stop executing the computation.

The serial construct simply specifies a sequential block of C++ code to be executed in sequence. The programmers have to explicitly mark these blocks of code due to the implementation details of how the SDAG code is parsed: our implementation does not actually parse all of C++ and serial allows us to mark which blocks the SDAG translator can safely ignore and pass to the C++ compiler directly.

If we want to wait on n method invocations (or n nested SDAG constructs, in general), but the order they are executed does not matter, we can use the forall construct in SDAG. The semantics are the same as overlap, but it is more convenient when we have n identical sequences that can be overlapped:

```
forall [iter] (0:10, 1)
  when recvData[iter](int param) { /* block1 */}
/* block2 */
```

In this case, we wait for 10 instances of recvData to arrive, each tagged with reference numbers from 0..9 (note the upper-bound on the range is exclusive: it defines a range $[0, 10)$ with a stride of 1). The receives can arrive in any order and block1 will be executed for each one as they arrive. When they all arrive, block2 will be executed.

Fibonacci Example using SDAG Using SDAG we can define a pedagogical Fibonacci using the (inefficient) recursive algorithm in the following way:

```
entry void calc(int n) {
  if (n < THRESHOLD) serial { respond(seqFib(n)); }
  else {
    serial {
      CProxy_Fib::ckNew(n − 1, false, thisProxy);
      CProxy_Fib::ckNew(n − 2, false, thisProxy);
    }
    when response(int val)
      when response(int val2)
        serial { respond(val + val2); }
  }
};
```

In this example, we define a calc(...) method that calculates the n'th Fibonacci number by either sequentially calculating the Fibonacci number if n is small enough (for efficiency reasons), or creating two children chares, waiting for both their responses, and then adding them up. In either case, the respond function sends the answer to the parent. Using SDAG, we explicitly define the dependency on waiting for both the responses from the two children and SDAG buffers one of the responses until they both arrive and we can add them. Although this is a simple example, it demonstrates the power of SDAG— without this we would have to manually buffer the first response and add them up later.

3.3 Threads

Often in the middle of executing sequential code, some remote data is required to proceed with the computation. In Charm++, this requires sending a message to a chare, waiting for a response, and then continuing execution. With SDAG this pattern can be expressed cleanly, but only if the waiting occurs at the top level entry method, because when blocks are allowed only in the entry methods).

```
entry void waitsForData(..) {
  serial {
    // some computation
    f(..);
    g(..);
  }
  when dataNeeded(...) serial {
    // continue execution with remote data
  }
}
```

Here, f and g may be regular methods or stand alone functions. This code works fine because the waiting happens at the top level. But if it is necessary to fetch remote data when the control is inside of (say) the function g, this is not supported by SDAG. Putting a when inside the body of g (g is regular C++ code), will just be flagged as a syntax error by the C++ compiler. In a SDAG entry, when a when statement is encountered, control typically *returns* to the Charm++ scheduler, with no trace of the ongoing work left on the stack itself. All the bookkeeping information about the pending when blocks and buffered messages is left in the SDAG data structures.

However, if we were to use a threaded model (assuming threads that are migratable) we can wait on remote data and then continue executing in the same context when the data arrives. CHARM++ supports this by allowing an entry method to be marked as threaded.

```
entry [threaded] void foo(..);
```

By declaring a method as such, the method will actually run in a user-level thread that is migratable. A thread is made migratable by allocating its stack using isomalloc, which allocates data with a globally-unique virtual address. The isomalloc function works by reserving the same virtual space on all processors [19, 3].

We can then declare a certain entry method to be sync, which allows it to actually return data:

```
entry [sync] ReturnMsg* bar(..);
```

Then inside the implementation of the foo entry method, we can make a call to bar, wait for the result, and seamlessly continue execution when the data arrives:

```
Worker::foo(..) {
  // do some computation
  ReturnMsg* msg = remoteChare.bar(...);
  // continue execution when msg arrives
}
```

Further, the call to remoteChare.bar() doesn't need to be at the top level entry method. In the earlier example, this call could be inside the body of the C++ function g(), which still works, because when the call is made, the user-level thread simply suspends, with its stack intact.

Futures: Threads are useful for describing this interaction pattern, but we may want to overlap the computation with the communication. In the above example, once the sync entry method is invoked, we wait for the message from bar to arrive before we proceed. However, although we know we will need the data from bar, we may not need it immediately. A Future is an abstraction that allows us to declare a container that will hold the data at some future time. The future will only block when we try to "open" the container and access the data. Using

futures, we can postpone waiting on the remote data until it is required for the computation.

The future construct was described, in the sense we use it, in the multiLisp system of Halstead [15], although multiple precursors existed before that. Taura, Matsuoka, and Yonezawa [46] extended ABCL to support the *future* construct as well.

The following code creates a CkFuture and passes it to an entry method:

```
Worker::foo(..) {
  // do some computation
  CkFuture ft = CkCreateFuture();
  remoteChare.bar(ft, ...); // call the remote chare with a future
  // continue execution
  ReturnMsg* msg = (ReturnMsg*)CkWaitFuture(ft); // wait on future
  // execute using the data from the remote chare
}
```

Here, we create a future that will hold the data that remoteChare.bar(...) will produce when it finishes execution. When we make the call to the remoteChare, we include the future, so it has a place to put its response. Then, when we actually need the data we can explicitly call CkWaitFuture(...), which will block if the remote data has not arrived.

Instead of using SDAG to express Fibonacci, we can express the same concurrency pattern using threaded methods and futures, as shown below. Note that since Charm++, as a C++ library, does not have a translator, except for parsing interface files and SDAG code, the method for accessing and setting futures is somewhat verbose.

```
void run(int n, CkFuture f) {
  if (n < THRESHOLD) result = seqFib(n);
  else {
    CkFuture f1 = CkCreateFuture();
    CkFuture f2 = CkCreateFuture();
    CProxy_Fib::ckNew(n-1, f1);
    CProxy_Fib::ckNew(n-2, f2);
    ValueMsg* m1 = (ValueMsg*)CkWaitFuture(f1);
    ValueMsg* m2 = (ValueMsg*)CkWaitFuture(f2);
    result = m1->value + m2->value;
    delete m1; delete m2;
  }
  ValueMsg *m = new ValueMsg();
  m->value = result;
  CkSendToFuture(f, m);
}
```

Synchronization Mechanisms Based on Threads The user level thread mechanism underlying Charm++ is designed to be used in a flexible manner. The API allows one to extract the (opaque) threadID of the currently running thread,

to suspend the current thread (and thereby transferring control to the scheduler which may resume another ready thread), and to "awaken" a thread (which puts the threadID in the scheduler's queue of ready threads). This API can be used to implement customized synchronization mechanisms. As an example, a counting semaphore can be implemented as shown in the pseudocode below:

```
wait(x) {
   while (x->value == 0) { enqueue(x->waitingQ, CthThread()); CthSuspend(); };
   x->value--; }
signal(x) { x->value++; tid = dequeue(x->waitingQ); CthAwaken(tid);}
```

Note that this works because Charm++'s threads are cooperative (not preemptive), and each thread is confined to one core at a given time, until it is migrated by the load balancer. Thread migration does not happen while a thread is waiting in a queue, by convention.

3.4 Comparing Concurrency Control Mechanisms

So, should one use Dagger, SDAG, or threads in a given situation? The Dagger mechanism is historically and empirically been subsumed by SDAG by the Charm++ user community. The reasons are easy to discern and were alluded to earlier: a structured graph is adequate for most real applications, and when it is too restrictive, one can use the flat, reactive, entry methods of Charm++ to restore full concurrency. Using threads efficiently is more complicated. Again, statistically, most Charm++ users tend to prefer SDAG. Avoiding the (admittedly small) extra overhead of threads, and the need to predict stack sizes, combined with environment-dependent challenges of migrating threads for load balancing are some of the reasons why. Also, the cleaner separation of parallel and sequential code that SDAG engenders (via the "serial" construct) is often seen as a beneficial feature. On the other hand, some programmers find it beneficial to *not* have that separation, and so prefer using threads. In particular, if you are calling a function f from a threaded entry method, you do not have to know if this function is completely local, or if it may request and block for some remote data. That way, a function that is sequential today, may be modified by the writer of that function to become parallel later, without requiring a code-change in the caller's code. Threads are also useful when you need to block for some specific remote data when you are deep in the function-call stack.

The need for abstraction, especially arising out of supporting other programming models, is another reason for using threads. For example, AMPI [19] implements the well-known MPI abstraction on top of Charm++. To benefit from Charm++'s load balancers, AMPI maps multiple MPI "ranks" on a single processor. When one rank issues a receive call, and the data is not available, the implementation needs to suspend the execution of the calling rank, and resume execution on any other rank that is ready on that processor at that point. This blocking receive can be implemented using Charm++ threads. AMPI implements each user "rank" (which the user thinks of as an MPI process) as a user-level thread embedded in a Charm++ chare.

The specific issues that come up when one is trying to migrate a chare, in which a user-level thread or a DAG is embedded, are discussed in our earlier paper [51], which also presents detailed performance comparisons of the alternative methods.

4 Controlling Concurrency across Parallel Objects

The control structures described so far: threads, futures, Structured Dagger, etc., can be used to control and manage the concurrency and control flow within a chare. CHARM++ also has several mechanisms to enable chares to work together in various ways to increase efficiency and/or programmability.

4.1 Asynchronous Collective Operations

A simple example of this is allowing the use of asynchronous broadcasts and reductions (as shown in the 5-point stencil example) over a chare array, which can be sparsely populated and can grow and shrink over time without explicit synchronization. In addition, CHARM++ has a very efficient built-in algorithm to detect termination across the entire system: the state when no messages are in flight and all processors are idle [45]. The termination detection mechanism in CHARM++ is very easy to use, and only requires a single call to the system:

```
CkStartQD(CkCallback(...));
```

In the above snippet, whenever this call is made, the CHARM++ runtime starts its termination detection algorithm, and when it confirms quiescence, it triggers the callback, which allows the user to define an arbitrary endpoint to be notified (for instance, a entry method on a chare, or broadcast to a chare array).

An example where these features are very beneficial is adaptive mesh refinement (AMR). In traditional MPI (and thus, in any bulk-synchronous) implementations of AMR, remeshing is an expensive operation that requires multiple collective operations to determine when all the remeshing decisions are finished propagating based on the mesh criteria. In the CHARM++ implementation [36], one abstracts the computation (structured as blocks in an oct-tree) as a dynamic collection of blocks indexed by their position in the tree using a chare array. During remeshing, instead of using $\mathcal{O}(d)$ (where d is the depth of the propagation) expensive collective operations over all the processors to determine when remeshing is finished, we use point-to-point messages to propagate decision messages and then wait for termination to be detected by the system. A recent paper [36] shows that this methodology is highly-scalable and has many beneficial properties.

4.2 Queuing Policies

CHARM++ allows a priority to be set for an entry method invocation; such priorities are used to schedule a message when it arrives on the destination processor. Under the hood, CHARM++ maintains a queue of outstanding messages

that execute in turn on each processor for the set of objects that live there. When a message arrives, it is placed in the queue to be executed in a certain order depending on its priority. Although message priorities are a heuristic, they can be very important for obtaining high performance.

Message priorities can be set very easily for an invocation by adding a single argument:

```
Worker::foo() {
  CkEntryOptions opts;
  opts.setPriority(100);
  remoteWorker.method(data, &opts);
}
```

Note that CHARM++ also allows priorities to be bit-vectors or other variable-sized fields, which is useful for state-space search applications [6].

In addition, CHARM++ allows the user to specify a queuing strategy that is used for the message when it arrives on the destination processor. By default, messages are enqueued in FIFO order, but this can be changed easily:

```
opts.setQueueing(CK_QUEUEING_LIFO);
```

An example where priorities make a high impact on application performance is dense LU factorization. In dense LU factorization, the matrix being factorized is decomposed into a 2D grid of blocks, which in the CHARM++ implementation [38] is encapsulated in a chare array. We can succinctly describe the parallel control flow of a non-pivoting LU in SDAG as follows:

```
1  entry void factor() {
2    for (step = 0; step < min(thisIndex.x, thisIndex.y); step++) {
3      overlap {
4        when recvL[step](blkMsg *mL) serial { L = mL; }
5        when recvU[step](blkMsg *mU) serial { U = mU; }
6      }
7      serial {
8        // Schedule the trailing update for sometime later with low priority
9        CkEntryOptions opts;
10       opts.setPriority(calcPrioDepOnLoc(x,y));
11       thisProxy(x,y).processTrailingUpdate(step, &opts);
12     }
13     when processTrailingUpdate[step](int step) atomic {
14       updateMatrix(L, U);
15     }
16   }
17   if (x == y) serial {
18     thisProxy(x,y).processLocalLU();
19   } else if (x < y)
20     when recvL[step](blkMsg *mL) serial { thisProxy(x,y).processComputeU(mL); }
21   else
22     when recvU[step](blkMsg *mU) serial { thisProxy(x,y).processComputeL(mU); }
23 };
```

Each block goes through various phases as it executes depending on its location in the matrix. The most critical operation for unleashing concurrency is performing the diagonal factorization (line 18), which only depends on a few of the trailing updates (matrix-matrix multiplies) to be executed (lines 2-16) (each diagonal enables all the trailing updates below and to the right of the diagonal, but only the ones above and to the left of the next diagonal are required to start that computation).

Note in the above example when recvL and recvU arrive (lines 4-5) instead of immediately executing the trailing update that is available, we delay the execution by enqueuing a message in the local queue with low priority that will start the trailing update (see lines 9-11). In this example, we exploit Charm++ prioritized scheduling to reduce the priority of an operation that might hamper work directly on the critical path from executing.

4.3 Memory-aware Scheduling in LU

Another example of across-chare concurrency control also comes from LU factorization. When LU is being weak-scaled, as it often is for obtaining the top-500 benchmark results, it needs to run very close to memory limits to obtain maximum performance and reach the FLOP limit of DGEMM (the matrix-matrix multiplies that the trailing updates execute). The typical Charm++ idiom is to send messages to a chare when the data is ready. However, for certain applications that are memory-sensitive, aggressively sending data when it is ready might exceed memory limits on the receiving end.

In our highly-scalable implementation of dense LU [38], we demonstrate how to exploit Charm++ groups to control incoming messages by explicitly scheduling when messages arrive. For LU, instead of sending the block of data when it is ready on the sender-side, we notify the receiver that the data is ready and allow the receiver to determine which blocks to request based on what is ready and the optimized schedule it has computed that adheres to the dependencies natural in an LU computation. With this methodology, we are able to achieve high performance without exceeding memory limits or treating processors as first-class entities.

4.4 Charisma: Controlling Concurrency across chares

Let us turn now, from the runtime schemes for across-chare concurrency control within a processor, to language-level mechanisms for controlling and expressing concurrency across chares, even when they are spread across multiple processors.

Note that Structured Dagger allowed clean expression of the life-cycle of a given chare, while still avoiding overly constraining the execution order, via the `overlap` and `forall` statements. However, the behavior of the program as a whole is not explicitly expressed; it remains an emergent property that needs to be inferred from the descriptions of behaviors many chares, possibly belonging to multiple chare arrays. Again, we built upon an empirical observation that a fixed, data-independent communication pattern among the chares is common

in most (but certainly not all) applications. For instance, in such applications, which tend to be iterative, the content of messages and even their sizes may change from iteration to iteration, but the basic pattern of message-exchanges (the dataflow among the objects) remains the same. *Charisma* [18] is a notation developed to facilitate elegant expression of such applications.

Charisma also supports multiple indexed collections of chares, as Charm++, but their behavior is expressed by a collective script (hence we call it an *orchestration* language). This script is written in a special notation, while the sequential code in the form of plain methods of chares is kept in separate C++ files. The main statement in Charisma is a foreach statement.

```
foreach i in stencil[i]
      stencil[i].foo();
end−foreach
```

The code above asks all members of a chare array (stencil) to execute their method foo. More interestingly:

```
foreach i in stencil[i]
    q[i] <− stencil[i].bar(p[i−1]);
end−foreach
```

tells each chare stencil[i] to consume the parameter p[i-1] and produce the parameter q[i]. The charisma compiler connects producers and consumers by generating appropriate message-passing (Charm++ method invocations) code. The concurrency in Charisma is only constrained by data dependencies and program order. Without going into technical details, and simplifying the example, the following code fragment for the 5-point stencil computation illustrates Charisma.

```
foreach [i,j] in stencil
(top[i,j], bottom[i,j], left[i,j], right[i,j]) <− stencil[i,j].publishboundaries();
repeat
      foreach [i,j] in stencil
          (+error, top[i,j], bottom[i,j], left[i,j], right[i,j]) <−
               stencil[i,j].publishboundaries(top[i+1,j], bottom[i−1,j],
               left[i,j+1], right[i,j−1]);
until (error < THRESHOLD)
```

Values for the boundaries generated in previous iteration are consumed by the neighbors in the next iteration. The + symbol preceding error specifies a reduction (i.e. a commutative-associative operation). The operator for the reduction (here, max) is specified in the declarations, not shown here.

Although some applications, such as Barnes-Hut, are not amenable to Charisma, because of the data-dependent data-flow they exhibit, a substantial class of applications are expressible using Charisma. Charisma scripts are compiled into Structured Dagger programs.

5 Case Studies and Performance

In this section, we summarize two case studies to demonstrate that the concurrency control mechanisms, and specifically SDAG, lead to high performance code. These case studies are taken from our 2011 HPC Challenge submission, which won the class 2 award for programming language productivity [22].

5.1 LeanMD

LeanMD [23] is a molecular dynamics simulation benchmark written in Charm++. It simulates the behavior of atoms using the Lennard-Jones potential to calculate the interaction between uncharged molecules. The benchmark is similar to the short-range non-bonded force calculation that NAMD calculates [7] and it also resembles the miniMD application found in the Mantevo benchmark suite [16] maintained by Sandia National Laboratory.

LeanMD is parallelized using a hybrid spatial and force decomposition. The three-dimensional space consisting of molecules is divided into equal-sized cells that hold a set of molecules using the cutoff distance r_c and a margin. During each iteration, the force calculation between a set of neighboring cells is assigned to another set of parallel objects called the *compute*s. Using the forces that are sent to the computes, they perform the force integration and update the properties of the atom — namely acceleration, velocity, and position.

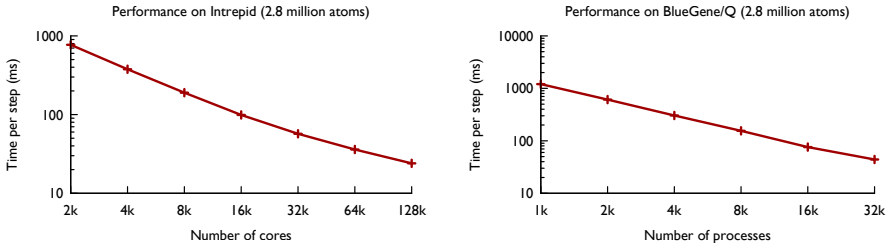

Fig. 3: Performance of LeanMD for the 2.8 million atoms system on Vesta (IBM BG/Q) and Intrepid (IBM BG/P)

Our code is very short (only 693 lines of code[1]) and it can be dynamically load balanced using many built-in strategies by Charm++, can be checkpointed to disk or in-memory for fault tolerance, and is not sensitive to different shapes of simulation domains nor to the number of processors.

The following is a snippet of SDAG code that shows the parallel flow of control that describes the Cell object for LeanMD:

[1] The line count was generated using David Wheeler's SLOCCount.

```
array [3D] Cell {
  entry Cell();
  entry void run() {
    for(stepCount = 1; stepCount <= finalStepCount; stepCount++) {
      // send current atom positions to my computes
      serial { sendPositions(); }
      // update properties of atoms using new force values
      when reduceForces(vec3 forces[n], int n) serial { updateProperties(forces); }

      if ((stepCount % MIGRATE_STEPCOUNT) == 0) {
        // send atoms that have moved beyond my cell to neighbors
        serial { migrateParticles(); }
        // receive particles from my neighbors
        for(updateCount = 0; updateCount < inbrs; updateCount++) {
          when receiveParticles(const std::vector<Particle> &updates) serial {
            for (int i = 0; i < updates.size(); ++i)
              particles.push_back(updates[i]);
          }
        }
      }

      if (stepCount >= firstLb && (stepCount - firstLb) % lbPeriod == 0) {
        serial { AtSync(); } // periodically call load balancer
        when ResumeFromSync() { }
      }

      if (stepCount % checkptFreq == 0) { // periodically checkpointing
        serial { contribute(CkCallback(CkReductionTarget(Cell,startCheckpoint),
            thisProxy(0,0,0))); }
        if (thisIndex.x == 0 && thisIndex.y == 0 && thisIndex.z == 0) {
          when startCheckpoint() serial {
            CkStartMemCheckpoint(CkCallback(CkIndex_Cell::cpDone(),thisProxy));
          }
        }
        when cpDone() { }
      }
    }
  };
};
```

Figure 3 shows the scaling of LeanMD on BG/P and BG/Q — two IBM supercomputers. We achieve near-linear scaling and demonstrate that load balancing is beneficial for obtaining high-efficiency. The checkpoint (milliseconds) and restart time (100-200 milliseconds) for LeanMD is very low. As seen in the above snippet, using all these features of Charm++ requires very little extra work by the programmer.

5.2 Dense LU Factorization

As described earlier, we have implemented a dense LU factorization library [38] in Charm++ that fully conforms to the HPC Challenge [11] specification. Our implementation is a fully-composable library (it can share space and time with another parallel Charm++ module) that allows for flexible data placement (by writing a simple block-to-processor function).

Our implementation has been scaled up to 8064 cores on Jaguar (Cray XT5 with 12 cores and 16GB per node) by increasing problem sizes to occupy a constant fraction of memory (75%) as we increased the number of cores used. We obtain a constant 67% of peak performance across this range. We also demonstrate strong scaling on Intrepid, an IBM Blue Gene/P machine, by running a fixed matrix size ($n = 96,000$) from 256 to 4096 cores. The results are shown in Figure 4.

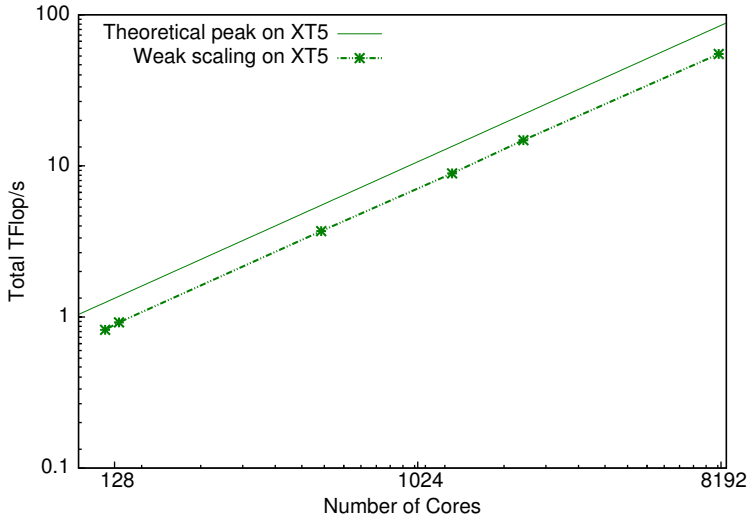

Fig. 4: Weak scaling (matrix occupies 75% of memory) from 120 to 8064 processors on Jaguar (Cray XT5).

6 Conclusion

We presented a historical overview of various mechanisms for controlling concurrency that have been developed for the Charm++ parallel programming system. Within a single chare (a message driven object), the mechanisms included Dagger, SDAG, and threaded entry methods based on migratable user-level threads. Of these, SDAG was seen to be the most beneficial and popular method, although threads combined with futures, and other synchronization mechanisms

are very useful in somewhat narrower contexts. Mechanisms for coordination and control of concurrency across chares, and indeed across processors, were also discussed. These ranged from priorities, quiescence detection, memory aware scheduling, as well as Charisma, an orchestration languages that specifies the behavior of a collection of chares, when they are known to exhibit static data-flow. We included two case studies to demonstrate the raw performance attained by Charm++ using these methods.

Charm++ has become one of the few parallel programming systems developed in academia that has been successful as a production-quality system for a significant number of highly scalable parallel applications in Science and Engineering, in regular use by scientists on supercomputers in USA and elsewhere. In addition to being a programming language in its own right, it also forms a substrate for development of other high level languages, of which Charisma is an example. We expect that it will be used as a back-end by new programming languages that will be developed by us and others; in this context, its support for interoperability is very important.

As the field moves to more complex machines and increasingly sophisticated adaptive applications, we think that Charm++ will play a larger role in the coming years. Its features for tolerating component failures and for managing power, energy and core temperatures will make it suitable for exascale computers. We expect the concurrency control abstractions described in this paper to evolve to meet the challenges of this future.

References

1. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.
2. G. A. Agha and W. Kim. Actors: a unifying model for parallel and distributed computing. *J. Syst. Archit.*, 45(15):1263–1277, 1999.
3. G. Antoniu, L. Bouge, and R. Namyst. An efficient and transparent thread migration scheme in the PM^2 runtime system. In *Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP) San Juan, Puerto Rico. Lecture Notes in Computer Science 1586*, pages 496–510. Springer-Verlag, April 1999.
4. W. C. Athas and C. L. Seitz. Multicomputers: Message passing concurrent computers. *IEEE Computer*, Aug. 1988.
5. K. Bisset, A. Aji, M. Marathe, and W. chun Feng. High-performance biocomputing for simulating the spread of contagion over large contact networks. In *Computational Advances in Bio and Medical Sciences (ICCABS), 2011 IEEE 1st International Conference on*, pages 26–32, Feb.
6. J. A. Booth. Balancing priorities and load for state space search on large parallel machines. Master's thesis, University of Illinois at Urbana-Champaign, 2003.
7. R. Brunner, J. Phillips, and L.V.Kalé. Scalable molecular dynamics for large biomolecular systems. In *Proceedings of SuperComputing 2000*, 2000.
8. R. K. Brunner and L. V. Kalé. Handling application-induced load imbalance using parallel objects. In *Parallel and Distributed Computing for Symbolic and Irregular Applications*, pages 167–181. World Scientific Publishing, 2000.

9. D. Caromel. Abstract Control Types for Concurrency (Position Statement for the panel : *How could object-oriented concepts and parallelism cohabit ?*). In L. O'Conner, editor, *International Conference on Computer Languages (IEEE ICCL'94)*, pages 205–214. IEEE Computer Society Press, August 1993.
10. C.Tomlinson and V.Singh. Inheritance and synchronization with enabled-sets. In *ACM OOPSLA*, pages 103–112, 1989.
11. J. Dongarra and P. Luszczek. Introduction to the HPC Challenge Benchmark Suite. Technical Report UT-CS-05-544, University of Tennessee, Dept. of Computer Science, 2005.
12. A. Gursoy and L. Kale. Tolerating latency with dagger. In *Proceedings of the Eigth International Symposium on Computer and Information Sciences*, Istanbul, Turkey, November 1993.
13. A. Gursoy and L. Kalé. Dagger: Combining the Benefits of Synchronous and Asynchron ous Communication Styles. In *Proceedings of the 8th International Parallel Processing Sy mposium*, April 1994.
14. A. Gursoy and L. Kalé. Dagger: Combining the Benefits of Synchronous and Asynchronous Communication Styles. In H. G. Siegel, editor, *Proceedings of the 8th International Parallel Processing Symposium*, pages 590–596, Cancun, Mexico, April 1994.
15. R. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, October 1985.
16. M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving performance via mini-applications. Technical report, Sandia National Laboratories, September 2009.
17. C. Hewitt, P. Bishop, and R. Steiger. A universal ACTOR formalism for artificial intelligence. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 235–245. SIAM, 1973.
18. C. Huang and L. V. Kale. Charisma: Orchestrating migratable parallel objects. In *Proceedings of IEEE International Symposium on High Performance Distributed Computing (HPDC)*, July 2007.
19. C. Huang, G. Zheng, S. Kumar, and L. V. Kalé. Performance Evaluation of Adaptive MPI. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006*, March 2006.
20. P. Jetley, F. Gioachin, C. Mendes, L. V. Kale, and T. R. Quinn. Massively parallel cosmological simulations with ChaNGa. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008.
21. L. Kale. The Chare Kernel parallel programming language and system. In *Proceedings of the International Conference on Parallel Processing*, volume II, pages 17–25, Aug. 1990.
22. L. Kale, A. Arya, A. Bhatele, A. Gupta, N. Jain, P. Jetley, J. Lifflander, P. Miller, Y. Sun, R. Venkataraman, L. Wesolowski, and G. Zheng. Charm++ for productivity and performance: A submission to the 2011 HPC class II challenge. Technical Report 11-49, Parallel Programming Laboratory, November 2011.
23. L. Kale, A. Arya, N. Jain, A. Langer, J. Lifflander, H. Menon, X. Ni, Y. Sun, E. Totoni, R. Venkataraman, and L. Wesolowski. Migratable objects + active messages + adaptive runtime = productivity + performance a submission to 2012 HPC class II challenge. Technical Report 12-47, Parallel Programming Laboratory, November 2012.

24. L. Kale, B. Ramkumar, V. Saletore, and A. B. Sinha. Prioritization in parallel symbolic computing. In T. Ito and R. Halstead, editors, *Lecture Notes in Computer Science*, volume 748, pages 12–41. Springer-Verlag, 1993.
25. L. V. Kalé. *Parallel architectures for problem solving.* PhD thesis, State Univ. of New York, Stony Brook (USA), 1985.
26. L. V. Kalé. Parallel execution of logic programs: the REDUCE-OR process model. In *Proceedings of Fourth International Conference on Logic Programming*, pages 616–632, May 1987.
27. L. V. Kale. Programming Models at Exascale: Adaptive Runtime Systems, Incomplete Simple Languages, and Interoperability. *The International Journal of High Performance Computing Applications*, 23(4):344–346, October 2009.
28. L. V. Kale and M. Bhandarkar. Structured Dagger: A Coordination Language for Message-Driven Programming. In *Proceedings of Second International Euro-Par Conference*, volume 1123-1124 of *Lecture Notes in Computer Science*, pages 646–653, September 1996.
29. L. V. Kale and M. Bhandarkar. Structured Dagger: A Coordination Language for Message-Driven Programming. In *Proceedings of Second International Euro-Par Conference*, volume 1123-1124 of *Lecture Notes in Computer Science*, pages 646–653, September 1996.
30. L. V. Kale, M. Bhandarkar, and R. Brunner. Run-time Support for Adaptive Load Balancing. In J. Rolim, editor, *Lecture Notes in Computer Science, Proceedings of 4th Workshop on Runtime Systems for Parallel Programming (RTSPP) Cancun - Mexico*, volume 1800, pages 1152–1159, March 2000.
31. L. V. Kale, B. H. Richards, and T. D. Allen. Efficient parallel graph coloring with prioritization. In *Lecture Notes in Computer Science*, volume 1068, pages 190–208. Springer-Verlag, August 1995.
32. L. V. Kalé and W. Shu. The Chare Kernel base language: Preliminary performance results. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages 118–121, St. Charles, IL, August 1989.
33. R. Keller, F. Lin, and J. Tanaka. Rediflow Multiprocessing. *Digest of Papers COMPCON, Spring'84*, pages 410–417, February 1984.
34. S. Kumar. *Optimizing Communication for Massively Parallel Processing.* PhD thesis, University of Illinois at Urbana-Champaign, May 2005.
35. S. Kumar, Y. Shi, E. Bohm, and L. V. Kale. Scalable, fine grain, parallelization of the car-parrinello ab initio molecular dynamics method. Technical report, UIUC, Dept. of Computer Science, 2005.
36. A. Langer, J. Lifflander, P. Miller, K.-C. Pan, , L. V. Kale, and P. Ricker. Scalable Algorithms for Distributed-Memory Adaptive Mesh Refinement. In *Proceedings of the 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2012). To Appear*, New York, USA, October 2012.
37. O. Lawlor and L. V. Kalé. Supporting dynamic parallel object arrays. In *Proceedings of ACM 2001 Java Grande/ISCOPE Conference*, pages 21–29, Stanford, CA, Jun 2001.
38. J. Lifflander, P. Miller, R. Venkataraman, A. Arya, T. Jones, and L. Kale. Mapping dense lu factorization on multicore supercomputer nodes. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2012*, May 2012.
39. S. Matsuoka and A. Yonezawa. Analysis of Inheritance Anomaly in Object-Oriented Concurrent Languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Object-Based Concurrency.* MIT Press, 1993.

40. X. Ni, E. Meneses, and L. V. Kalé. Hiding checkpoint overhead in hpc applications with a semi-blocking algorithm. In *IEEE Cluster 12*, Beijing, China, September 2012.
41. B. Ramkumar and L. V. Kalé. Compiled execution of the Reduce-Or process model on multiprocessors. In *The North American Conference on Logic Programming*, pages 313–331, Oct 1989.
42. Sanjeev Krishnan and L. V. Kale. A parallel array abstraction for data-driven objects. In *Proceedings of Parallel Object-Oriented Methods and Applications Conference*, Santa Fe, NM, February 1996.
43. O. Sarood and L. V. Kalé. A 'cool' load balancer for parallel applications. In *Proceedings of the 2011 ACM/IEEE conference on Supercomputing*, Seattle, WA, November 2011.
44. A. Sinha and L. Kalé. Information Sharing Mechanisms in Parallel Programs. In H. Siegel, editor, *Proceedings of the 8th International Parallel Processing Symposium*, pages 461–468, Cancun, Mexico, April 1994.
45. A. B. Sinha, L. V. Kale, and B. Ramkumar. A dynamic and adaptive quiescence detection algorithm. Technical Report 93-11, Parallel Programming Laboratory, Department of Computer Scie nce , University of Illinois, Urbana-Champaign, 1993.
46. K. Taura, S. Matsuoka, and A. Yonezawa. An efficient implementation scheme of concurrent object-oriented languages on stock multicomputers. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM SIGPLAN Notices*, June 1993.
47. C. Tomlinson, W. Kim, M. Scheevel, V. Singh, B. Will, and G. Agha. Rosette: An object-oriented concurrent systems architecture. In *Proceedings of the 1988 ACM SIGPLAN workshop on Object-based concurrent programming*, OOPSLA/ECOOP '88, pages 91–93, New York, NY, USA, 1988. ACM.
48. J. Yelon and L. V. Kale. Agents: An undistorted representation of problem structure. In *Lecture Notes in Computer Science*, volume 1033, pages 551–565. Springer-Verlag, August 1995.
49. A. Yonezawa. *ABCL: An Object Oriented Concurrent System.* MIT Press, 1990.
50. A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. *ACM SIGPLAN Notices, Proceedings OOPSLA '86*, 21(11):258–268, Nov 1986.
51. G. Zheng, O. S. Lawlor, and L. V. Kalé. Multiple flows of control in migratable parallel programs. In *2006 International Conference on Parallel Processing Workshops (ICPPW'06)*, pages 435–444, Columbus, Ohio, August 2006. IEEE Computer Society.
52. G. Zheng, X. Ni, and L. V. Kale. A Scalable Double In-memory Checkpoint and Restart Scheme towards Exascale. In *Proceedings of the 2nd Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS)*, Boston, USA, June 2012.