Chapter 11

Charm++

Laxmikant Kale, University of Illinois, Urbana-Champaign

11.1 Introduction

Parallel programming is more difficult than sequential programming, despite occasional claims to the contrary by some. One needs to worry about additional issues such as race conditions, load imbalance, communication, locality, etc. How can one go about simplifying parallel programming? This is the question that motivated development of Charm++ in the 1990s.

The design of Charm++ was guided by following three design principles.

- The first principle, which we sometimes paraphrase as *"no reliance on magic, at least until you have learnt the trick"*, is to avoid an approach that depends on a component whose feasibility is not proven yet. This ruled out ambitious projects based on unrealized technologies, such as parallelizing compilers. The basis of this principle was in part pragmatic: we wanted our abstractions to be useful for developing applications *now*, rather than in future when the world's dream system has been built. But in part, it was based on a skepticism about whether pie-in-the-sky technologies such as parallelizing compilers can extract all the parallelism that a computational problem has (in contrast to the parallelism evident in the corresponding sequential code).
- The second, related, principle was based on the belief that what one needs is not full automation by "the system" but a good partnership between the parallel programmer and the system. We sought *an optimal division of labor between the programmer and the system.* Of course, as we developed and tested lower-level abstractions, we recognized that the optimal shifted upwards in the direction of more automation. But we always wanted the development of abstractions to be grounded in efficient implementations.
- The third principle was that the features and abstractions we design should be motivated by use cases coming from full-fledged "real" applications. Sometimes, there is a tendency among computer scientists to design abstractions based on the intrinsic beauty of an idea. This "platonic" view of design often leads to failure when the idea does not get adopted in the real world, since it does not address the needs of application developers. Of course, basing ideas and abstractions on just one application — an application-centered approach — does not lead to lasting contributions either, because they may be too specialized. This principle of "application oriented but computer-science centered research" was enunciated in a position paper [90].

The resultant programming model was characterized by its innovative use of overdecomposition, and its then unique execution model. We begin with a description of these foundational features.

11.2 The Charm Programming Paradigm and the Execution Model

11.2.1 Overdecomposition as a central Idea

Examining "Parallel Programming" as an activity in light of the principle of optimal division of labor, it becomes clear that parallel decomposition (*what* to do in parallel) is something that the programmers can do reasonably well, whereas the tedious details of which processor executes what at which time are good candidates for automation by the system. Especially in science and engineering applications, the parallelism (which typically arises from the physical world being simulated) is relatively easy to identify. This is true even for more abstract mathematical linear algebra algorithms. Tolerating latency and balancing load are programming tasks that add to the tedium.

Leveraging this observation requires an additional conceptual step: if the programmers can decompose, they can *overdecompose* into a large number of chunks (i.e. logical work-units and data units), rather than decomposing to the physical processors. This idea of *Overdecomposition* into natural work units and data units of reasonable granularity was the key to the entire programming paradigm that is at the base of Charm++.

It is useful to recall that this idea arose from our early work in the world of tree-structured computations in parallel logic programming: the divide-and-conquer and state-space-search world. There, it was natural to see that if we let every recursive call become a unit of decomposition and scheduling, the overhead will be overwhelming. This was true even with work-stealing as pioneered by Vipin Kumar et al. (and popularized later by Cilk): you still had the programmer deciding what to make a task. At the same time, the processor oriented decompositions appeared unnatural, because you could not easily decompose work to processors a-priori. The right approach seemed to be to let the programmer decide a reasonable grainsize, and turn over the placement and scheduling of the nodes of the tree to the runtime system. [93]

As we turned our attention to applications in science and engineering, the same principle of overdecomposition was found to be extremely fruitful: the basic idea is to have the programmer divide the computation into multiple collections of chunks of the right granularity. The right granularity is defined as “as small as possible, as long as it adequately amortizes the overhead”, with the overhead coming from scheduling, messaging and from other runtime system “tax”, based on number of chunks. The overhead also comes in the form of extra memory (for example, the increased memory needed for “halos” in stencil computations, when units are over-decomposed). Fortunately, these overheads are relatively small, allowing for a range of grainsizes to be suitable, *and* the grainsizes could easily be set parametrically, often without recompiling the program. Further, the degree of decomposition does not have to be influenced by the number of processors.

The chunks,which can be work-units or data-units or amalgams of the two, can take many forms in different languages within this paradigm. They can be objects, migratable user-level threads, functional “thunks”, continuations, and so on. Overdecomposition into “chunks” still leaves the question of how these chunks interact with each other. The overall Charm paradigm is neutral to that: many possible interaction mechanisms are possible, and we will see the possibilities in a later section on languages within the Charm family. Charm++, which is one of the models within this paradigm (the major one, to be sure), uses C++ objects as its “chunks”: an object is an amalgam of work and data unit. It uses *asynchronous method invocation* as the mechanism by which chunks interact with each other. Thus, Charm++ objects are units of mapping (what gets assigned to a processor) while the asynchronous method invocations are units of scheduling (what to execute next).

11.2.2 Message Driven Execution

A natural consequence of overdecomposition is that there are multiple work and data units mapped to a single processor core. Some arbitration mechanism is needed to decide how control transfers among the computations related to these units. Here we made another important decision: the control will transfer among these units implicitly and cooperatively, under the control of a data-driven user-level scheduler. There is one scheduler on each processor core. The scheduler works with a pool of asynchronous method invocations targeted at objects on that core that are awaiting execution. It picks one and transfers control to scheduling unit associated with it by invoking the method on the object named in the invocation. Control transfers back to the scheduler only when the method returns. This requires that the unit of scheduling must not contain any blocking calls.

Of course, any particular programming language that is an instance of this programming paradigm may include blocking constructs; we only expect that the *implementation* of such a language break the computation up in such a way that every scheduling unit it offers to the runtime system is a nonblocking one. If the scheduling unit were to be a user-level thread, this requirement amounts to requiring cooperative multi-threading.

(In the broader paradigm, beyond Charm++, a scheduling unit involves exactly one work-unit or data-unit, and possibly some other data. Thus, it may represent a continuation, or a message, in addition to an asynchronous method invocation.)

11.2.3 Empowering Adaptive Runtime Systems

Overdecomposition, and having the programmer express the parallel computation in terms of the logical work-units and data-units, begs another question: who is going to assign the units to processors? Much of the literature in parallel programming that examines this sort of question assumes that the alternatives are to let the *compiler* do the mapping or to let the *programmer* provide it. Some go further, and say that the compiler should provide a mapping but the programmer should be able to override it. But there can be a third player in this game: the runtime system. A runtime system can assign objects to processors initially, and, more importantly, can *change* the assignment in the middle of execution if it so wishes. This becomes possible because the programmer has not been allowed to use the knowledge of which processor an object is housed on directly, in any way that will interfere with such migration. Of course, you can find which processor you are on during execution of a scheduling unit and print it. But the only interactions are with other objects via asynchronous method invocation, and that continues to work whether the other object is local or somewhere else.

This creates a degree of freedom for the runtime system that turns out to be tremendously powerful. An *adaptive* runtime system can use its knowledge of the machine, and the current state of the computation to optimize execution in a variety of ways. We will see some of those ways in a later section; but for now we just note how consequential and significant these relatively simple design decisions are! Our research program resulting from trying to exploit this degree of freedom has kept us occupied for the past decade or more, while yielding a lot of low-hanging fruit for multiple CSE applications.

11.3 Basic Language

In Charm++, the programmer decomposes an application into objects with methods that can be invoked asynchronously by other objects. Each object is mapped to a processor by the runtime, and has only one method executing at a time, and after it finishes, control is passed back to the Charm++ runtime. The Charm++ runtime determines the next method to execute on that processor depending on the availability of data.

A Charm++ program is composed of C++ code and an interface file (called a .ci file), which describes the parallel objects and their corresponding method signatures to Charm++. This allows easy integration with previously written sequential C++ code.

11.3.1 Chares: the basic unit of decomposition

C++ objects that are units of parallelism (or units of mapping) are specially designated by the user, and are called *chares*. Chares are managed by the runtime system (RTS): they are placed on a processor by the RTS, scheduled for execution by it, load balanced with other objects, etc. Communication occurs by invoking a method on a chare, which does not require the user to know its location. During the course of the execution, chares can be created and destroyed as the computation progresses.

Execution of a Charm++ application starts with the creation of a special chare, called the *main chare*. This chare typically performs setup and the creation of other chares. In an interface file required by Charm++, the user defines the main chare and a constructor for it, where the execution begins.

```
mainmodule foo {
  mainchare Main {
    entry Main(CkArgMsg* m);
  };
};
```

In the corresponding C++ file, the same object needs to be described with the code that should be executed when the constructor is called from the runtime:

```
#include "foo.decl.h"
struct Main : public CBase_Main {
  Main(CkArgMsg* m) { CkPrintf("Running foo on %d processors\n", CkNumPes()); }
};
#include "foo.def.h"
```

The files *foo.decl.h* and *foo.def.h* are generated by the Charm++ compiler from the interface file, and contain declarations and definitions of the generated classes, respectively.. *CBase_Main* is the underlying class for the main chare that is generated by the Charm++ runtime. The user simply extends this class and gains the parallel functionality of the class described in the interface file. *CkArgMsg** includes two fields: *argc* and *argv* that can be accessed by dereferencing *m* in this case, e.g. *m->argc*.

11.3.2 Entry Methods: The basic unit of scheduling

Any C++ object that is described as a chare can have an entry method, which is a normal C++ method that can be invoked remotely and asynchronously by another chare. The parameters passed to the method (by the caller) are serialized, packed into a message, and sent to the called chare. The parameters of an entry method must be serializable so that they can be efficiently packed and unpacked. The Charm++ runtime provides efficient serialization for many common data types in C++ (arrays, vectors, sets, maps, etc.), along with common data types. To enable generation of the serialization code, the entry method signatures must be declared in the interface file.

To remotely communicate among chares, a proxy, which is returned when a chare is created, can be used by any object to invoke entry methods on it. An entry method is invoked by instigating a method call on the remote proxy much like a traditional method call in C++. Under the hood, Charm++ packs the parameters and sends a message to the processor where the chare lives.

11.3.3 Asynchronous Method Invocation

Remote invocations on a proxy in Charm++ are asynchronous and non-blocking. Asynchronous method calls cause the description of a chare to be reactive: each method is a description of the action to perform when a message arrives (i.e., an entry method is invoked).

11.3.4 Indexed Collections of Chares: Chare Arrays

In many applications, a pattern arises where each chare created must obtain proxies to a set of other chares, depending on the application's communication. While this approach allows arbitrary networks of chares to be created, the structural setup and bookkeeping requires substantial effort from the programmer. To reduce this effort, Charm++ supports collections of homogeneous chares (i.e. with the same C++ type) that are indexed uniquely within the collection. Every chare can then be addressed using the array proxy and the index within that array. Chare arrays provide further semantic information about the structure of an application, which allows the Charm++ runtime to optimize collective operations between sets of elements in a chare array.

Today's Charm++ applications often create multiple multi-dimensional chare arrays. If desired, the arrays can be sparse i.e. only some of the indices within its range are actually populated by a chare. A chare that is a member of a chare array can communicate with other members of the same chare array, as well as members of other chare arrays. Invoking a method on the array proxy or a section of the array, is then a broadcast or multicast, which is mapped efficiently by the runtime to the hardware, considering topology and other architectural parameters.

In the Charm++ interface file, a 1-dimensional chare array of N chares is described with a constructor and one entry method as follows:

```
array [1D] Foo {
  entry Foo(void);
  entry void someMethod(int arg);
};
```

This chare array can then be constructed by calling the Charm++ parallel new operator *ckNew*, which instantiates the elements and invokes *Foo*'s constructor on each one. The array proxy that is returned from this creation *CProxy_Foo* can then be used internally or externally to invoke an entry method on the entire array, a subset, or one element.

```
  int numChares = 100;
  CProxy_Foo array = CProxy_Foo::ckNew(numChares);
```

The corresponding C++ code provides the actual code to execute when the runtime calls the constructor or the entry method is invoked by another object.

```
class Foo: public CBase_Foo {
  Foo() { /* array constructor */ }
  void someMethod(int arg) { /* do something when someMethod is invoked */ }
};
```

The array proxy can then be used to broadcast to the entire array a certain entry method, which causes it to be invoked on every array element.

```
  array.someMethod(100);
```

By indexing the array proxy, a specific element in the array can be addressed and an entry method executed on just that element.

```
  int element = 5;
  array[element].someMethod(100);
```

In contrast to MPI, many-to-one commutative-associative operations (i.e. reductions) among chare array elements are asynchronous, each element depositing its contribution to the reduction when it is ready. Once the reduction is finished, a entry method is invoked by the runtime with the result from the reduction. The *contribute* call can be augmented with parameters when the reduction has an operator and payload.

```
  contribute(CkCallback(CkReductionTarget(X, finished), proxyToX));
```

In the Charm++ interface file, a entry method can be marked as a reduction target, indicating that it can be used to return the final value once the reduction has completed.

```
  chare Foo {
    entry [reductiontarget] void finished();
  }
```

11.3.5 Readonly Variables

Charm++ supports declaring *readonly* variables that can be read by any parallel object in a certain module. Like a sequential const variable in a C++ class, readonlys can only be set in the main chare's constructor and then are propagated by the Charm++ runtime to all the processors. Readonly variables are declared like this in interface file:

```
mainmodule foo {
  readonly int bar;
  mainchare Main { ... }
};
```

In the C++ file, they are declared and can be written only in the main chare's constructor:

```
/* readonly */ int bar;
class Main : public CBase_Main {
public:
  Main(CkArgMsg* m) {
    bar = 1;
  }
};
```

The variable *bar* is then readable by any parallel object in the *foo* module.

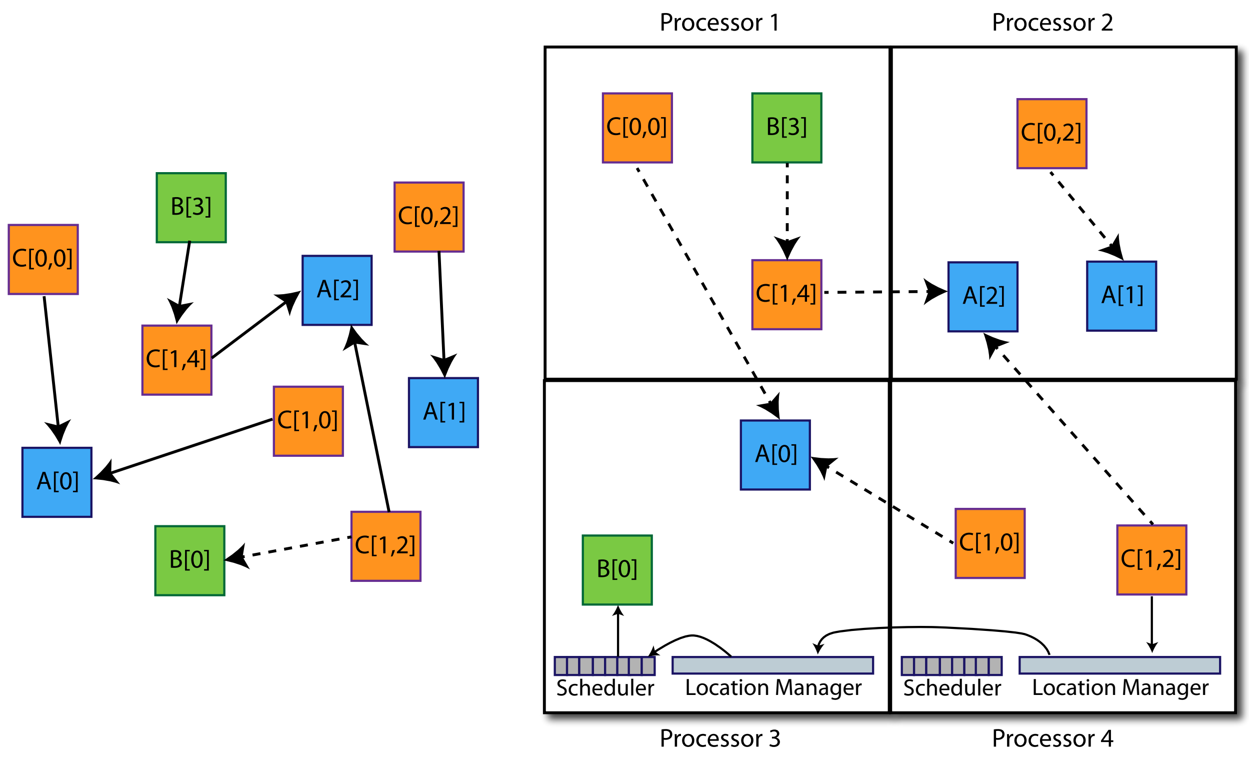

Figure 11.1:

11.3.6 Charm++ Objects: User and System view

The left part of Figure 11.1 illustrates the view of a Charm++ application as seen by the programmer: a collection of chare objects, interacting via asynchronous method invocations (shown via arrows). In this picture, we show three *chare arrays*: A is a simple 1D dense array, B is a spare 1D array (sparse because many elements in the range B[0..3] do not exist), while C is a 2D array. Note that each element of the chare arrays is a *chare*, a coarse-grained entity visible to the runtime system (in particular, it is *not* an array of scalars).

The diagram on the right shows how the runtime system views the same setup: in this case, we depict a machine with 4 processor cores (on same or different nodes). Objects are anchored to specific cores. The communication between chares, i.e. the asynchronous method invocation, is mediated by the runtime system. As an example, we expand the anatomy of the call from C[1,2] to B[0]. There is a location manager on each processor that finds the processor where the target chare (here, B[0]) lives. Scalable techniques for location management used by the Charm Runtime system are described in [100]. The location manager finds that B[0] lives on processor 3, and routes the message to it. After confirming that the object is indeed present on its processor, the location manager on processor 3 enqueues the message in the local scheduler's queue.

In Charm++, a rank is associated with each scheduler which is (typically) assigned to a core, and is called a PE (from the old term "processing element"). The design decision to anchor a chare to a core was taken to enhance locality, and to avoid locking overheads. Locality is enhanced because an object always executes on the same core, unless migrated away by the runtime. No locking is necessary for accessing elements of a chare because no two entry methods can be simultaneously active on the same chare (since only one processor executes it). However, there are advanced features in the language, typically used by library writers, that can break these assumptions in a well-defined manner. Further, it is possible to run Charm++ applications in a mode such that there is only one "PE" (akin to rank) associated each node, and the (multicore) parallelism within a node is expressed using pthreads or openMP. Accelerator support is also provided (see section **??**).

11.3.7 Example: 1-D 5-point Stencil Code

The following describes a 5-point stencil with a simple 1-D decomposition in Charm++. In this application, there is a main chare that begins the computation by creating a chare array of parallel objects, each one responsible for an equal portion of rows. Each element in the chare array sends the two boundary rows to its two neighboring chare array elements. For the edge cases, the chare array elements are wrapped.

After sending the boundaries, each array element waits to receive its two neighboring elements and then executes the computation kernel that computes the 5-point stencil. The kernel returns a boolean value signifying whether it has locally converged, depending on the error tolerance. These booleans are then ANDed together in a reduction between all the chare array elements, and the target of the reduction is a callback that broadcasts back to all the chare array elements the result. Depending on the result, the elements either all continue or send a message to the main chare that convergence has been reached, causing the main chare to print and call *CkExit()*.

11.3.8 The Structured Dagger Notation

In this application, an array element does not explicitly wait to receive the neighboring elements, instead it describes reactively the action to perform when it receives a boundary element. For each neighboring element that arrives, it will update its local boundary based on those values, and then if all boundaries have arrived it can start the computation kernel. This requires the object to keep the state of how many neighbors have arrived and then perform a different action (the last will spawn the kernel) upon receiving a boundary, depending on the state.

The advantage of this reactive approach is its generality: it allows arbitrary dynamic dependencies to be described. However, when a chare's lifecycle can actually be described statically to the runtime, the buffering and counting of messages can be automatically performed by the runtime, and the appropriate block of C++ code can be executed upon message arrival depending on the state of the object.

Structured Dagger (SDAG) is the scripting language in Charm++ that describes a chare's static lifecycle. This script allows an object to describe sequentially the flow of message arrival that it expects (all others are buffered) and the action to perform given its current state.

SDAG has many of the same sequential constructs (*while*, *if*, *for*, etc.) with a few parallel additions. SDAG control flows sequentially, making ordering explicit. The *when* keyword in SDAG tells the runtime to wait for a certain entry method and buffers any other messages that arrive for other entry methods. The message can differentiated by the entry method name and a reference number, which can be set when sending a message. The SDAG code will block until the *when* is satisfied and then the corresponding block of code is executed. If multiple entry methods can be executed in any order, the *overlap* keyword can be used, which allows multiple *when* clauses to be fulfilled in arrival order. The keyword *atomic* indicates a block of sequential C++ code to be executed.

For the stencil, the Charm++ interface file declares a main chare and a 1-D chare array of Jacobi objects with a SDAG description of the lifecycle of the Jacobi object. Each object while it has not converged, asynchronously sends messages updating its ghosts cells. After the sends, it waits for two updates to arrive, calling the sequential `updateBoundary` code for each boundary arrival. After both have arrived, the compute kernel is executed and the array element deposits its convergence value to a reduction, whose target is a broadcast to the entire array that checks convergence and possibly continues or sends a message to the main chare that computation is finished.

```
mainmodule jacobi1d {
  readonly CProxy_Main mainProxy;
  readonly int dimX; readonly int dimY; readonly int blockSz;
  readonly int numChares;
  mainchare Main {
    entry Main(CkArgMsg *m);
    entry void done();
  };
  array [1D] Jacobi {
    entry Jacobi(void);
    entry void updateGhosts(int dir, int size, double gh[size]);
    entry [reductiontarget] void checkConverged(bool result);
    entry void run() {
      while (!converged) {
        atomic {
          thisProxy(wrapY(thisIndex-1)).updateGhosts(BOTTOM, dimY, t[1]);
          thisProxy(wrapY(thisIndex+1)).updateGhosts(TOP, dimY, t[blockSz]);
        }
        for (remoteCount = 0; remoteCount < 2; remoteCount++) {
          when updateGhosts(int dir, int size, double buf[size])
          atomic { updateBoundary(dir, buf); }
        }
        atomic {
          int conv = computeKernel() < DELTA;
          CkCallback cb(CkReductionTarget(Jacobi, checkConverged), thisProxy);
          contribute(sizeof(int), &conv, CkReduction::logical_and, cb);
        }
        when checkConverged(bool result) atomic {
          if (result) { mainProxy.done(); converged = true; }
        }
      }
    };
  };
};
```

In the main chare, the Jacobi array is created and a call to the entry method `run` is broadcast to the entire array, causing each element to execute the above run method with has the SDAG code.

```
#include "jacobi1d.decl.h"

/*readonly*/ CProxy_Main mainProxy;
/*readonly*/ int dimX, dimY, blockSz, numChares;

class Main : public CBase_Main {
public:
  CProxy_Jacobi array;
  Main(CkArgMsg* m) {
    // read arguments from m, into dimX, dimY, blockSz, numChares
    mainProxy = thisProxy;
    array = CProxy_Jacobi::ckNew(numChares); // Chare array creation
    array.run(); // Start the computation
  }
  void done() { CkExit(); }
};
```

The rest of the C++ code, has the Jacobi constructor and the sequential routines to update the boundary and compute the 4-point stencil.

```
class Jacobi: public CBase_Jacobi {
public:
  double** t;
  Jacobi() { /* initialize and allocate regions */ }
  /* updateBoundary(), computeKernel(), etc. */
};
#include "jacobi1d.def.h"
```

11.3.9 Threaded Methods, Blocking invocations and Futures

You can us a regular, blocking, method invocation to get a response form a remote object, instead of the asynchronous method invocation we have described so far. This is supported by a method specifically tagged as "threaded" in the interface file; this indicates that such a method may suspend execution in the middle. The methods that have a non-void return value are tagged as "sync" methods. The system creates a user-level lighweight thread when it starts executing a threaded method. The system creates a user-level lightweight thread (much more lighter-weight than a pthread, and one that is not visible to the native OS) when it starts executing a threaded method. Notice that it is still a cooperative rather than pre-emptive multi-threading, and it is still true that only a single method is actually running at a time on a give chare; but it is now possible for multiple method invocations to be "active" with suspended threads.

In addition to waiting for sync methods to return, you can suspend a threaded entry method in other ways. For example, it can wait for a "future" to be evaluated. An explicit way of suspending and awakening threads is also supported.

Threaded methods are used in (relatively rare) situations when the blocking wait happens deep from nested function calls, since structured dagger notation requires such waiting to be lifted to the top level of the control flow. Structured dagger based methods have a slightly smaller overhead than threaded methods, don't need allocation of a separate stack, and are typically perceived as clearer to understand by Charm++ programmers.

11.3.10 Other Features:

Among other features, Charm++ supports assigning priorities to method invocations, and a flexible callback architecture that is especially useful for library writers. It supports dynamic insertion and deletion of new elements (at new indices) in an existing chare array, and support for specifying initial placement of chares, overriding Charm++'s default decisions. Sections of chare arrays (anologoues to sub-communicators in MPI) are also supported.

There are situations when a programmer needs to be aware of the notion of processors and nodes. Library writers, and certainly those writing low level libraries such as communicaiton optimizers or load balancers, may find this useful. For this purpose, Charm++ supports *Groups* and *NodeGroups*, which can be thought of as Chare arrays with exactly one member on each processor or node respectively.

We refer the interested reader to an extensive manual at the Charm++ website (http://charm.cs.illinois.edu/help) to explore these concepts.

11.4 Benefits of Overdecomposition and Message-Driven Execution

11.4.1 Independence from number of processors

Although the code for stencil (Jacobi relaxation) looks superficially similar to the corresponding MPI code, there are critical differences. The number of chares `numChares` is not the same as the number of cores (or nodes, for that matter). It is a number independent of the processor count, chosen by the programmer to amortize the overhead. A 3D version of the code, with 3D data array decomposed into 3D array of chares, illustrates this better. Assuming an equal decomposition along each dimension, the number of processors will need to be a cube, in a plain MPI implementation. In Charm+, the number of *chares* (i.e. objects) needs to be a cube, but the number of processors is unrestricted. One can run them on 173 processors, if one wants…

The table below shows the performance of a Charm++ program running on various number of processors in the range from (I suppose there is no point in showing a table. A plot would be nice, but figure limit will get us... Anecdotal will have to do).

This ability to decouple decomposition from processors is useful to exploit any machine (or, part of the machine) that is available for a job to its fullest extent. Even more importantly, it enables other runtime benefits, such as fault tolerance strategies as we will see later.

Of course, a programmer can write a multi-block style MPI code to achieve the same thing. However, the life-cycle of each block is not separately expressed in a processor-centric code. Further, the programmer is then assuming responsibility for managing placements of blocks on their own.

11.4.2 Asynchronous reductions

Reductions in Charm++ are non-blocking operations. Neither the processor is blocked, nor is the chare object participating in it is blocked. Firstly, the contribute operations extends over one chare array. If the application has other chare arrays, they can continue their execution as usual. Secondly, after calling "contribute" to deposit its data into a reduction, a chare object is free to execute other entry methods if it so chooses. In the code we saw above, the chare chooses to wait for the callback to `checkConverged`; But consider an alternative design: we decide that convergence checking every iteration is an overhead, and we should do it every 5 iterations (say). Further, we want to use the result of convergence testing done in iteration 5k+1 to be tested 4 iterations later (in iteration 5(k+1)). This is easy to accomplish by adding a conditional `if (i%5 == 1)` before line 26, and adding another conditional `if (++i % 5 == 0)` before line 28. Essentially, the reduction started in iteration 5k+1 will continue in the background, overlapped with the computation and communication of the next 4 iterations of the stencil algorithm.

Asynchronous collectives, in general, can improve performance significantly, because although the elapsed time for such collectives is often high, the processor-time occupied by them is relatively small [92].

```
1  while (!converged) {
2    atomic {
3      int x = thisIndex.x, y = thisIndex.y, z = thisIndex.z;
4      copyToBoundaries();
```

```
    thisProxy(wrapX(x-1),y,z).updateGhosts(i, RIGHT, dimY, dimZ, right);
    /* ...similar calls to send the 6 boundaries... */
    thisProxy(x,y,wrapZ(z+1)).updateGhosts(i, FRONT, dimX, dimY, front);
  }
  for (remoteCount = 0; remoteCount < 6; remoteCount++) {
    when updateGhosts[i](int i, int d, int w, int h, double b[w*h])
    atomic { updateBoundary(d, w, h, b); }
  }
  atomic {
    int c = computeKernel() < DELTA;
    CkCallback cb(CkReductionTarget(Jacobi, checkConverged), thisProxy);
    if (i%5 == 1) contribute(sizeof(int), &c, CkReduction::logical_and, cb);
  }
  if (i % lbPeriod == 0) { atomic { AtSync(); } when ResumeFromSync() {} }
  if (i % checkpointPeriod == 0) {
    atomic {CkStartMemCheckpoint(CkCallback(CkIndex_Jacobi::d(), thisProxy));}
    when d() { }
  }
  if (++i % 5 == 0) {
    when checkConverged(bool result) atomic {
      if (result) { mainProxy.done(); converged = true; }
    }
  }
}
```

11.4.3 Adaptive Overlap of Communication and Computation

Since multiple chunks are assigned to one processor, no single chunk can block the execution on a processor (unlike, say, a blocking receive in MPI). If one chunk is waiting for its data, another chunk can execute *if* its data (i.e. method invocation) is available. This message-driven execution adaptively overlaps communication and computation, without having to do any extra programming for it.

One way of optimizing traditional MPI applications is measure the communication time, and then work hard in programming to reduce the time spent in communication, so you can say "after these optimizations, the application now spends only 8% (for example) time in communication", and declare victory. However, another way of looking at what was achieved, is: the communication network is not utilized for 92% of the time! This spurt of communication now needs to be supported by the network. It is this sort of application behavior that necessitates an overengineered communication network, expensive and power hungry to boot.

In contrast, overdecompostion in the Charm model leads to spreading of the communication over an iteration or time-step, thus utilizing the communication network better. You can live with a "cheaper" lower bandwidth network better, or reduce the chance of communication contention impacting performance.

11.4.4 Compositionality

The kind of compositionality we are talking about here is a somewhat subtle point, but one very important for modularity. With traditional models such as MPI, it is *very hard* to get two *separately developed independent modules* to interleave their execution on an overlapping set of processors. "separately developed" here means that each module cannot reference entities in the other module directly. "independent module" here means that there is no data dependency between them, and their work can be carried out in any order. Typically, on any individual processor, one of the modules will complete its step, and then the other module is given control explicitly. Of course, you could use wild-card receives to interleave the execution of the two modules, but this gets "very hard", especially if each module's code has a complex life cycle, with long chain of message receipts.

Why is such interleaving important? Execution of any parallel module may lead to idle time on individual processors. This could be due to communication latencies or load imbalances. With interleaved execution, idle time in one module can be be overlapped with useful computation in the other. With Charm++, such interleaving occurs completely automatically, as a natural consequence of its data-driven execution model, without any programming on user's part.

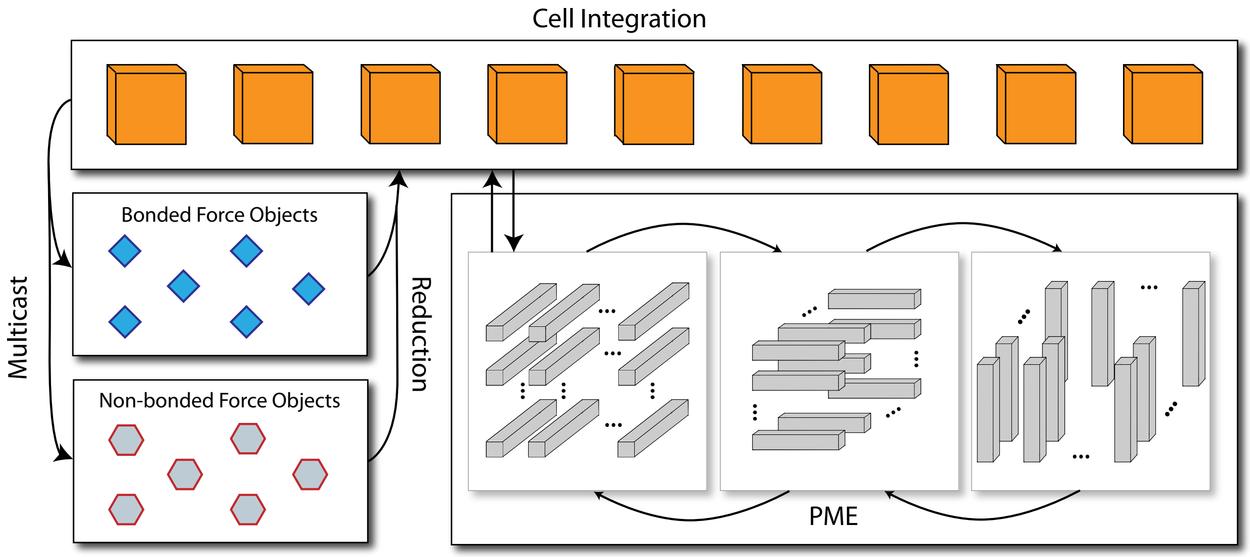

Figure 11.2:

11.4.5 Software Engineering Benefits: Separation of Logical entities

In traditional (say, MPI) programs, the processor-centric view leads to unnecessary coupling between modules. Domains of different kinds in the same simulation are typically divided into the same number of chunks, and i'th chunk of one module is placed with i'th chunk of the other module, simply because they both need to be on i'th processor. Such undesirable coupling arising out of processor-centric programming models is completely eliminated by Charm++: you can decompose different domains into different number of pieces, and leave it to the runtime to decide which chunks co-exist on a processor (say, so as to minimize communication).

11.5 Larger Example: simpleMD

As a larger example that illustrates some of the features of Charm++, consider the program to simulate biomolecules. In such an application, atoms in the simulation box are simulated in femtosecond time steps. In the particular algorithm we discuss, atoms are partitioned into cubic cells. There are three categories of forces experienced by each atom, and correspondingly, we create multiple chare arrays to calculate them. Each non-bonded force calculation object receives coordinates of atoms in 2 cells within a pre-specified cutoff distance from each other, calculates the forces (electrostatic and Van der Waal's) on atoms in each set due to those in the other set, and sends the resulting forces back to the 2 cells via reductions. The bonded force calculation objects calculate forces due to atoms connected by bonds. The more interesting and complex operation involves calculation of long-range forces using the so-called particle-mesh Ewald algorithm, which involves a forward and backward 3-D FFT. Since this is a fine-grained yet communication intensive operation, a pencil decomposition as shown in figure 11.2 is used. This involves calculating line FFTs sequentially along each dimension, permuting data between phases in order to bring the lines over which FFT needs to be calculated together on a single processor. This can be naturally expressed using 3 arrays of chares, consisting of pencils parallel to the x-axis, y-axis and z-axis respectively.

Thus, the natural decomposition of this application involves six separate arrays of chares. A Charm++ program expresses the lifecycle of each type of the element of these chares cleanly and separately, without worrying about where (as in, on which processor) each chare object lives. The runtime system decides the placements of various objects onto processors so as to minimize communication, balance load, handle interconnection topology related optimizations, etc.

11.6 Adaptive Runtime Features

A simple reference implementation of Charm++ as described so far can be created relatively easily with a static scheme (e.g. cyclic or block) assignments of chares to processors, a correspondingly simple location manager, a message-

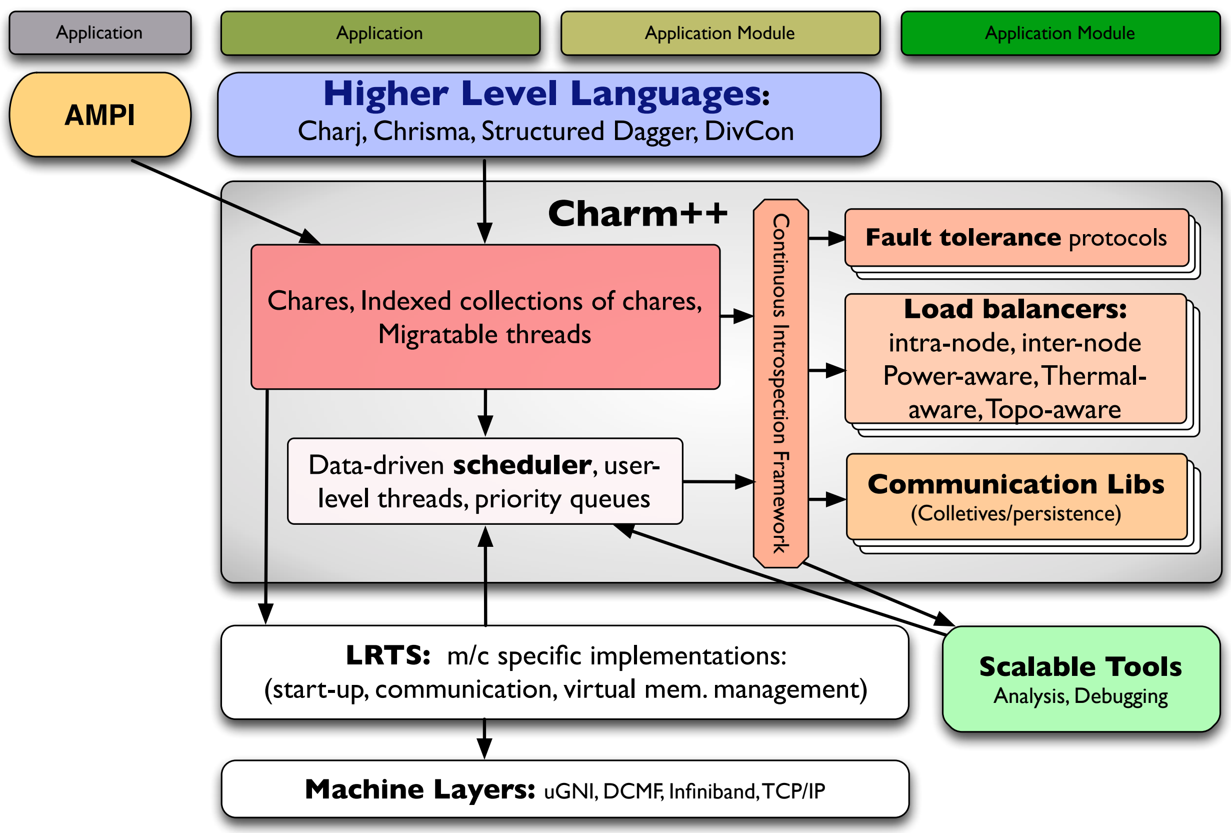
Application
Application
Application Module
Application Module
AMPI
Higher Level Languages:
Charj, Chrisma, Structured Dagger, DivCon
Charm++
Chares, Indexed collections of chares, Migratable threads
Data-driven scheduler, user-level threads, priority queues
Continuous Introspection Framework
Fault tolerance protocols
Load balancers: intra-node, inter-node Power-aware, Thermal-aware, Topo-aware
Communication Libs (Colletives/persistence)
Scalable Tools Analysis, Debugging
LRTS: m/c specific implementations: (start-up, communication, virtual mem. management)
Machine Layers: uGNI, DCMF, Infiniband, TCP/IP

Figure 11.3:

driven scheduler, with no migration of chares across processors during execution. The basic benefits of the model, listed in section 11.4, such adaptive overlap of communication and computation, and no dependence of processor counts, are still realized with this simple implementation. However, only an implementation that incorporates a more sophisticated *adaptive* runtime system can unleash the full power of the separation of concerns created by the model by ensuring that user's view is not processor-centric. An adaptive runtime can observe and monitor the behavior of the application and the hardware as the computation evolves. Such monitoring is enabled by the fact that the runtime system schedules each chare object, and mediates each communication between them. It can then take actions that improve the execution efficiency or reliability of the application. We next describe a few of the capabilities that the current Charm++ implementation supports.

11.6.1 Load balancing Capabilities in Charm++

Load imbalance, especially when it arises dynamically as the application evolves, has the potential to be biggest impediment to high performance and strong scaling of CSE applications. Even for applications with little or no dynamic change, making initial assignment of work to processors can be challenging. Charm++ has several schemes for automatically balancing load.

A large class of schemes in actual use in Charm++ rely on a heuristic principle we call "principle of persistence": the computational load and communication patterns exhibited by naturally decomposed objects *tend to* persist over time. This is true for a wide variety of applications, including those with dynamic refinements, or slow change in balance (as in the case of particle migrations). When user specifies use of such strategies, Charm++ RTS maintains a database of object loads and, optionally, communication graphs. It then uses a (user-specified) strategy from the Charm++ suite of load balancers to create a new assignment of objects to processors, and then migrates the objects to their new destinations before resuming execution.

The suite of strategies include those that ignore communication, as well as those that focus on reducing communication costs. Some are simple heuristics, while others are based on complex graph algorithms. Some only aim to *refine* an existing assignment, reducing the migration costs, while others seek to rebalance better by ignoring the existing assignments. An application programmer can also write an application-specific (or general purpose) strategy, and plug it in using a standard interface provided by Charm++. Strategies that utilize graph partitioners, such as METIS, Scotch and Trilinos, are also available. Here, it is important to remember that the graph we are talking about has the chare objects as its vertices. As such, compared with the normal application domain of these partitioners, where they may partition billion element graphs, the target here is much smaller graphs. This is one of the second order benefits of the Charm++ approach. The domain typically remains chunked in the same set of partitions, but the assignments of chunks to processors changes. This makes the execution time for coming up with a new assignment much faster compared with full-fledged domain decomposition. For the rare situations, one can also merge or split existing chares by deleting or inserting new elements in a chare array, although this feature is rarely used.

Leveraging this small size of graph, many strategies collect the object-graph one one node, and apply the decision-making algorithm sequentially. For very large machines, hierarchical strategies and distributed strategies are also available. A new meta-balancer scheme is being developed that monitors execution characteristics and their evolution, and selects an appropriate strategy, and decides how often to apply it, at runtime. This will be (IS by the time the book is out..) the default load balancer, unless the user overrides it with a more specialized strategy.

Automatic Load balancing is a signature strength of Charm++. Many of our applications have shown benefits of this feature on machines sizes up to 298,000 cores.

Using load balancing in the Stencil code

Using load balancing is relatively easy. The dominant mode in which it is used is synchronous: all chares call a special (collective-like) function called `AtSync()`. The RTS decides new placement of objects, migrates them, and calls a pre-designated entry method signifying resumption of execution. The object may find itself at another processor at this time; this is inconsequential from the programmer's point of view. To facilitate migration, object's data must be serialized. Charm++ provides a convenient "PUP" (pack-unpack) interface for chare objects to simplify specification of serialization code. In addition to this, the programmer must choose the load balancing strategy on the command line. With this simple change the code is ready to be automatically load balanced.

11.6.2 Fault Tolerance

Charm++ supports several alternative fault tolerance schemes, some in experimental mode and others in fully production versions. The migratable objects model is an excellent fit for fault tolerance strategies: if we can migrate an object to another processor for load-balancing, we can use the same mechanisms to checkpoint (migrate) it to the disk, or to another processor's memory. The baseline scheme leverages this ability and allows the application code to create a globally synchronous checkpoint with a single call. The runtime make sure all its data structures are faithfully stored. An interesting consequence of the migratable objects model is that this checkpoint can be used to resume execution on a different number of processors, as long as the original program was carefully written to avoid references to actual processors.

A similar scheme is also available to create checkpoints in other processor's memory, instead of the disk. In this case, the system can detect failure and automatically recover the application without killing the job. Each object creates a checkpoint on its current processor, as well as on a buddy processor. On detecting failure, the runtime system restores objects on the non-failed processors from their own checkpoints; the objects on the failed processor are restored on a spare processor using the checkpoint stored on its buddy.

Both of these schemes require that the job scheduler not kill the job if one of the nodes crashes. The basic scheme, as it is, can be used on a dedicated cluster of workstations. On large-scale machines, we have demonstrated the scheme using a fault injection mechanism that ensures that the job is not killed by the scheduler. The following table shows the performance of our double in-memory checkpoint scheme. The checkpoint overhead can be as low as a few milliseconds for low-memory applications such as molecular dynamics, and only a few seconds for large-memory applications. More interestingly, the recovery time measured from the time that the failure was detected to the point application resumed execution, is quite low as well.

We have recently extended this work to utilize local storage such as flash memories, and to reduce the impact of checkpoint time even further by overlapping checkpoint traffic with application execution.

More experimental strategies Charm++ provides are :

(1) proactive fault tolerance, where it utilizes signals from some external environmental monitoring system that warn it about impending failure of a node; it reacts by migrating chares away from that node and adjusting runtime structures such as spanning trees to eliminate the dependence on the (potentially) doomed node.

(2) A message-logging scheme that avoids having to rollback all the processors when one node fails. Only the failed node's objects are restored from a checkpoint, and they re-execute their messages (saved at their sender). This saves energy during recovery. Further, the recovering objects are migrated to multiple processors during recovery, thus parallelizing recovery. This feature allows the system to tolerate faults that happen at a faster frequency than the checkpoints!

Making SimpleMD Tolerate Faults

Utilizing the above strategies is easy. Leveraging the same PUP interface created for load balancing, all that the programmer has to do is make a call to the runtime at some periodicity to checkpoint their data, as shown in lines 19-21 of the code in section 11.4.2. The rest is handled by the system!

11.6.3 Shrinking or Expanding the sets of processors?

A feature of Charm++ explored extensively in the past, and one that may become important again due to "HPC-in-the-cloud" scenario is its ability to change the set of processors assigned to a job at runtime. This is accomplished by migrating objects away from (or in to) the processors that the job scheduler requests from (or assigns to) a running Charm++ job. So, a job running on 10,000 processors can be made to give up 1000 processors, continuing to run on 9000 processors, and later can be given 3000 more processors, as long as the job scheduler is capable of such behavior. We have demonstrated this using a job scheduler of our own design, and we hope that over time mainstream schedulers will start supporting such moldable jobs. The utility of such features to maximize system utilization is obvious, and the Charm runtime ensures that it utilizes the available processors with the best possible efficiency.

11.6.4 Experimental Features: Power, Temperature, heterogeneity and accelerators

Experimental: Power and Thermal adaptations

A recent experimental strategy is aimed at reducing cooling energy. It allows the machine operator to use a warmer A/C setting. This creates a risk that some chips will overheat. The runtime periodically monitors temperature of each core; for the cores that get too hot, it reduces their frequency a notch using DVFS available on many processors today. If they get cold enough, it bumps up their frequency again. These actions, of course, create a load imbalance in HPC applications that will normally make such a scheme impractical. However, Charm++ load balancers can handle different speed ratios between cores, and thus can restore balance after every exercise of DVFS frequency change. Other strategies for minimizing and constraining some user-specified combination of power, energy, and execution time are also being developed currently.

Experimental: Support for Heterogeneity and Accelerators

Accelerator chips of various kinds are now quite popular in high-performance computing, starting with the cell processor, GPGPUs and Intel's upcoming MIC chips. New features being added to Charm++, based on ongoing research, include support for accelerated entry methods. Specifically, the user can tag entry methods as accelerated, and specify whether the method should run only on the host, only on the accelerator or possibly both. The runtime system balances the load between the host and the accelerator device. It also does heterogeneous load-balancing among accelerated and non-accelerated nodes, such as those found on the blue waters system which houses accelerators on 10% of its nodes.

11.7 A quick look under the hood

11.7.1 System architecture

Figure 11.3 shows the overall architecture of the runtime system. A low-level runtime system abstracts machine details, including particularly the native communication primitives. The next layer (called "Converse") provides data-driven scheduling and user-level threads. The Charm runtime provides migratable threads, and scalable location managers. It also interfaces to plug-in strategies for load balancing, fault tolerance, and strategies for optimizing specific communication patterns.

11.8 Charm++ family of languages

Charm++ improves programming productivity because all the features mentioned above, including modularity, compose rationality, and automation off resource management. Yet, it's interaction mechanism is fairly primitive: asynchronous method invocation among objects is akin to message passing, and is the only mechanism in Charm++. But, with those features, Charm++ is also an excellent substrate for designing higher-level languages. Any such language will inherit these features from Charm++, And modules written in it will interoperate well with each other. At Illinois, we have developed several such languages.

Adaptive MPI (AMPI): Charm empowers users with MPI legacy code to take advantage of processor virtualization and over decomposition with minimal effort. AMPI, Charm's interface to MPI, enables developers to run any MPI program on top of Charm. The key advantage of using AMPI is the freedom to choose the number of MPI ranks independent of the system size. Choice of MPI ranks can thus be made from the application perspective which leads to substantial performance gains. Under the hood, MPI ranks are mapped to Charm user level threads. Multiple user level threads may reside on the same processor and are managed by Charm scheduler. Presence of multiple threads on a processor enables overlap of communication with computation without programming effort. Each time a thread is blocked on communication, the scheduler picks other threads and executes them. In addition, AMPI users can take advantage of other inbuilt Charm functionalities such as object migration, dynamic load balancing, fault tolerance and communication libraries.

Improved performance has been demonstrated for many MPI applications using AMPI. Execution time of BRAMS, which is a MPI based weather forecasting code, decreases by 25% when run on AMPI [137]. The application primarily benefits from computation-communication overlap and dynamic load balancing. A comprehensive study of advantages of AMPI for micro benchmarks such as jacobi3D, NAS's BT-MZ and fractography3D (crack propagation simulation) has also been done [84]. AMPI has also been used to execute, fine tune and debug programs which require large number of MPI ranks using small systems. Rocstar, which is a solid rocket motor simulation code, takes advantage of these features on a small system [88].

Adaptive MPI, like Charm++, is a general purpose language. It is possible to raise the level of abstraction further by specializing the language. Here, we do not mean domain-specific languages, although we have their place as well. Instead, we mean specializing languages by restricting the types of interaction between parallel entities they support.

Charisma only allows static data flow among Charm++ objects. To understand its motivation, consider a large Charm++ program. The chare classes express the behavior of each type of object in the program. However, the behavior of the program as a whole is not directly visible, and can be thought of as an emergent property of the behavior of individual objects. This detracts from the expressiveness of the application code. For a significant class of applications, the data flow among the objects does not change across application iterations. The same set of objects exchange the same pattern of messages with each other in each step. Of course the content and even the size of such messages will be different across iterations. For such applications, Charisma provides an elegant expression: the global flow of control and data among multiple chare arrays is expressed in a script-like language. The sequential methods of each object are expressed in C++ as before. The individual objects themselves only consume and publish values, whereas the charisma code connects the publishers and consumers.

Multiphase Shared Arrays

Task Parallelism, ParSSE, and DivCon

Future Plans: CharJ

11.9 Applications developed using Charm++

Several highly scalable applications have been developed using Charm++. We will summarize them here, and how they exploit Charm++ capabilitues.

One the earliers, and possibly most well-known application developed using Charm++ is NAMD. This is an appliation that was co-developed with many recent features of Charm++, by a team consisting of Klaus Schulten, Robert Skeel, Laxmikant Kale, and co-workers. NAMD is designed for simulations of biomolecular systems. The simulations trace the trajectories of all individual atoms that include water, proteins, and possibly lipid bilayers and other atoms, with a timestep of a femtosecond. Forces on atoms arise due to electrostatics, van der Waals forces, and bonds.

The electrostatic forces are computationally dominant, and are broken down further into short-range forces calculated pair-wise explicitly, and long range forces calculated using a FFT-based algorithm. There are six categories of chare objects in the natural decomposition expressed in Charm++, which are load balanced using a custom load balancer. The main challenge in biolmolecular simulation is to do millions (or even billions) of timesteps, where the work available for parallelzing in each timestep is relatively small. NAMD is able to achieve close to a milisecond–per-step rate, and has scaled to machines with 224,000+ cores. It shared the Gordon-Bell awatrd in 2002, and was a finalist in 2000. It is a widely used program with 50,000+ users and occupies a significant fraction of cycles on most national supercomputers.

OpenAtom (previously called LeanCP) is an application based for simulating elctronic structures of materials using the Car-Parinello algorithm. Implemented from scratch using Charm++, as a successor to the Piny code developed earlier, in a collaboration led by Dr. Glenn Martyna of IBM, OpenAtom strives for strong scaling with a finer decomposition of this problem than before. Although the technical details of the propblem or its parallel aspects are beyond the scope here, it is worth noting that the logical expression in Charm++ required over a dozen different chare arrays (distinct collections of chare objects). Although the computation is relatively static in terms of load patterns, it still requires a strong initial load balancing. Probably most importantly, this application allowed us to exhibit effects of interconnection topology induced contention on communication performance on modern machines. The topology aware mapping we developed for it often reduced execution times by 40% or more. The ability to do this mapping was failitated by degrees of freedom reated by the object-based decomposition.

ChaNGa is a computational astronomy program developed in a collaboration led by Prof. Thomas Quinn of University of Washington.

11.10 Charm++ as a research vehicle

In addition to its use for developing adaptive, scalable parallel applications, Charm++ is a excellent vehicle for facilitating research on various issues of critical importance in high performance computing, as we move towards exascale.

The load balancing framework in Charm++ provides an easily accessible way to plug-in different load balancing strategies. Load balancing in the general setting, as well as in specialized setting suitable for different applications, is still an open area of research. Researcher can test their ideas with relatively little effort on various benchmarks and applications in the Charm++ suite.

Similarly, communication between chares can be *delegated* to a user-defined plug-in library. This allows researchers to develop and test alternative algorithms for optimizing communication operations, such as collectives, or message-combining strategies.

With a somewhat larger effort, researcher can leverage the charm++ infrastructure for developing fault tolerance strategies.

Another area where we hope to have a significant participation by the HPC research community is in developing new languages and frameworks on top of Charm++. The interoperability and compositionality features of Charm++ ensure that such languages will have automatic support for dynamic load balancing, and can interleave execution of modules in novel languages with more traditional modules in Charm++ and MPI. The main benefit of Charm++ here is to provide a powerful backend that makes development of new languages quite easy (of course, beyond any compilation effort that is needed by the language).

For load balancers, for higher level languages, for fault tolerance techniques, for automatic communication optimizations, ..

11.11 Charm++: History and current Stete

Charm++ system has evolved over the years, in response to the types of applications we considered and the abstraction needs that we perceived in them. Its precursor, the Chare Kernel, and the early C-based Charm system, were used for combinatorial search applications, and parallel logic programming. This had the notion of chares, a word we borrowed from the RediFlow project of Bob Keller. By 1993 [91, 94] we developed a C++ based version called Charm++. As we increasingly focused on CSE applications, many of the modern concepts were added to it, including Chare Arrays, and threaded methods. Although the base language has remained the same since about 1998, much effort has been spent on its adaptive runtime system, and refinement of features via application experience.