Chapter 2

Designing Charm++ Programs

Laxmikant V. Kale

Department of Computer Science, University of Illinois at Urbana-Champaign

We learned about the design philosophy, and basic concepts in CHARM++, as well as its features and benefits. In this chapter, we will review the process of designing CHARM++-based applications, and discuss the design issues involved. Specifically, we will illustrate how to use individual features of CHARM++, through a series of examples.

2.1 Simple Stencil: Using Over-decomposition and Selecting Grain-size

Let us first consider the process of developing a parallel implementation of a simple stencil code. Through this example, we will illustrate how to specify over-decomposition in practice, and how to make practical grain size decisions. We will use the same example to show how to extract automatic communication-computation overlap (which is a benefit of the CHARM++ model), and show how easy it is to exploit features such as load balancing and fault tolerance.

Imagine that the data that we wish to deal with is represented by a three-dimensional grid. Further, the computation we wish to carry out for each cell involves using values from the neighboring cells in the grid. To make

it concrete, let us consider a 7-point stencil: calculating the new value of a cell located at location [x,y,z] in the grid requires values of the cells whose coordinates differ by just one in exactly one of the dimensions involved. Other than this somewhat abstract description of the pattern of data communication needs, we will omit the rest of the details of the computation to keep the description simple. Suffice it to say that this communication pattern arises in many science/engineering simulations, such as those involving solving the Poisson equation, or fluid dynamics computations.

How should the three-dimensional grid be decomposed among the processors? For simplicity of analysis, let us assume that the grid has a cubical aspect ratio. That is, the number of cells along each of the dimensions is the same. If one is designing the application with a traditional programming model, such as MPI, one can consider several options: for example, the data grid may be divided into horizontal slabs, with one slab assigned to each processor. However, with a sufficiently large number of processors, it is known that the communication volume (i.e. the amount of data being communicated in total) is smaller if one divides the grid into cubes, and assigns one cube to each processor.

Again, for the purpose of simplicity, let us assume that there is only one processor core on each node. So, when we say "processor" in the above paragraph, we simply mean a node. We will return to the issue of multicore nodes in the context of CHARM++ a little bit later.

Notice that the above strategy in case of the simple MPI programs requires us to request a cubic number of processors. In addition, often the stencil codes keep the number of cells on each processor exactly equal by requiring that the total grid size along each dimension be a power of 2. This also helps keep the code simple, in particular in the part where each processor needs to calculate which portion of the global grid it owns. However, this decision now has an unintended consequence: the number of processors has to be a cube of a power of 2! Alternativey, one can write clever MPI code that can utilize a MxNxK processor grid with some careful decomposition along each dimension, or even write a much more sophisticated (and complex) multi-block code in MPI.

2.1.1 Grainsize Decisions

When we think about the same problem in CHARM++, we should still choose a cubic decomposition. The grid is represented by a three-dimensional array of chare objects. Each object communicates with its neighboring objects so as to send the boundary data that they need to complete the stencil computation. However, we pretty much ignore the number of processors in deciding the decomposition. Below, we will describe several considerations one may use in deciding how large each cube should be. However, the important thing to note is that the *number of processors* is not a primary consideration.

A basic guideline in deciding how large each cube (and in general, each chare object) should be is: *make it as small as possible, but large enough*

that its computation is significantly larger than the overhead of scheduling all the method invocations it must handle. On current machines, typically, the overhead of scheduling a single method invocation is hundreds of nanoseconds. There may be memory allocation overheads that make it slightly larger. CPU overhead for remote communication is of the order of a microsecond at most in modern machines. In any case, since each one of the cubes must send and receive about 6 asynchronous method invocations each cycle, if we make the computation larger than, say, 100 microseconds, that should be adequate. As we will see below, exact determination of the optimal grain-size is not needed.

Other considerations such as memory usage may push the application developers to use a larger grain size than that. In any case, a common approach among CHARM++ application developers is to parameterize the choice of grain size, and to determine the size to use in production runs experimentally.

In applications that use a weak-scaling approach, where the amount of data used on each processor remains roughly constant, these processor-independent guidelines can be translated into processor dependent terms. For example, an application developer may say that this application behaves reasonably well when there are between 20 and 40 objects per processor. It should be understood that the basic consideration is still the size of each object, and the number of objects per processor (often called "the virtualization ratio") is a secondary, derived, metric.

So, the important point to remember: in most applications, it is not necessary to precisely decide the grain size. As long as it is sufficiently large to amortize the overheads, and small enough to make several chares on each processor available to the adaptive runtime system, there is a range of grain size values that yield similar, and close to optimal, performance.

If dynamic load balancing is necessary for the application, it helps to have more than 10 objects per processor. It also is important in that situation that no single object be too large. If a single object is larger than the average load per processor, for example, there is nothing a load balancer can do to reduce the execution time below a single object's time. It is usually easy to meet the constraints on the average grain size (described in the paragraphs above), as well as this maximum grain size constraint, because of the broad range of grain sizes that satisfy those constraints. In some rare applications, it may become necessary to split objects that become too large computationally as the application evolves. CHARM++ supports dynamic insertion of new objects for this purpose.

Returning to our simple stencil computation, we have now defined the three-dimensional array of objects that is being distributed to processors under the control of the runtime system. If we choose, we can describe a mapping (a user-defined assignment of objects to processors). We may specify the mapping to be immutable or just an initial mapping that the runtime system can change as the application evolves. Programmers control which type of mapping and dynamic reassignment they desire.

Even for this simple, regular, program we can see several benefits in a

CHARM++ based design. The program will run fine on any given number of processors. It will just distribute all objects on available processors (See Figure 2.1, with a 2-D example, for simplicity). Sure, some processors may get one more object than other processors, but this "quantization" error is certainly acceptable as it is typically less than 5-10 % penalty, and applies *only when* the number of processors does not evenly divide the number of objects. Typically, cache performance of the code improves because of the blocking effect of smaller objects, leading to improved performance compared with having just one chunk of each processor. Communication is naturally overlapped with computations; while some chunks are waiting for their communication, other chunks can compute, the interleaving occurring without programmer intervention, because of the message-driven execution model.

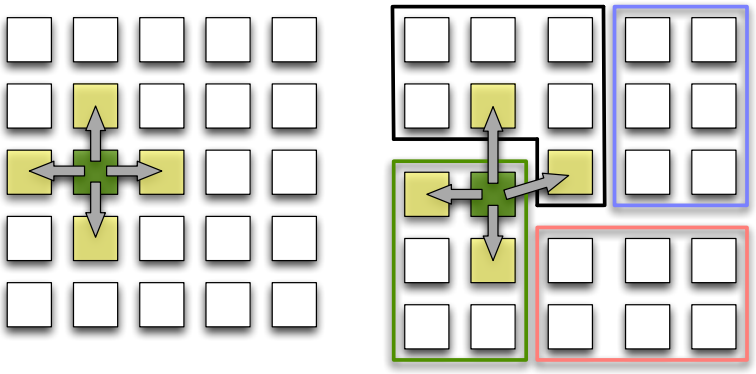

FIGURE 2.1: Chares for a 2D Stencil computation and their mapping to physical processors

How does our execution model work for this application? Looking a bit under the hood, on each processor we have a separate scheduler (See Figure 1.3 and the associated description in Chapter 1), working with a pool of method invocations waiting to be executed. It picks one of them, delivers it to the targeted object, and when (only when) the object returns control back to the scheduler, it repeats the cycle. Notice how this naturally interleaves execution of objects, adaptively and automatically overlapping communication and computation.

2.1.2 Multicore Nodes

Let us return to the issue of **multicore nodes** now. The programmer has multiple options for dealing with this. One is to assert that there is one "PE" associated with each core (or even, with each hardware thread). One still expects there to be multiple objects on each PE, and the model works as described above, except (1) communication within a physical node is fast

because the RTS uses shared memory for implementing it. (2) When needed, the application itself can use the shared memory in specific ways to optimize performance. As a simple example, CHARM++ supports read-only variables, for whom only one copy needs to be made on each node.

A second option is to treat a collection of cores (say a NUMA domain: i.e. a set of core sharing some level of cache and/or a memory controller) as a single PE. In this case, the decomposition may employ chunkier objects. Now, each PE may utilize multiple cores associated with it by using many possible constructs: for example, OpenMP, or CHARM++'s internal CkLoop construct that can use the cores via their associated hardware threads.

With the second mode, message sizes are larger, and the memory locked up in ghost regions (for our stencil example) is proportionately smaller. The former achieves better overlap of communication and computation, and *may* have cache performance benefits depending on patterns of data accesses. Which method is better depends on many application dependent factors, and the programmer can make the right choice. The essential characteristics of the program are still unchanged with either option.

2.1.3 Migrating Chares, Load Balancing and Fault Tolerance

For various reasons, including load balancing, we may want the chares to be able to migrate from one processor to another during the execution. To facilitate this, the programmer must provide the system a bit of additional information: how to serialize (i.e. pack and unpack) the data of the object. This is done in CHARM++ using a powerful and flexible PUP (pack-and-unpack) framework. For every chare class that you wish to make migratable, you must declare a PUP method. This method simply enumerates the variables of the object, with a few notations to indicate sizes where they cannot be inferred. The system will call the same PUP method of a chare for finding the size of the chare, for serializing its contents, and for reconstructing (deserializing) it from a packed message on the other processor.

Once the chares are made migratable, many new benefits follow. For example, one can migrate "work to data" at will. Better still, you can let the system do some load balancing for you. In this simple application, all chares have equal amount of work (except for quantization effect because the number of rows and columns may not be divisible by the corresponding dimensions of the chare array). Yet, suppose, this application is running on a cluster consisting of processors of differing speeds. [1] You can now just turn on load balancing, insert a call to balance load only after first few iterations (so the system has the load data to base its decisions on) and presto: the chares will be

[1] Years ago, we had this situation at the CSAR (Center for Simulation of Advanced Rockets): we had a cluster, and then new faster nodes were bought and added to the cluster. Of course, most MPI jobs confined themselves to either the new partition or the old partition entirely. However, CHARM++ jobs were able to run on the combined system, adjusting to the speed heterogeneity with its load balancers. [34]

allocated in proportion to the speed of the processors, with faster processors getting proportionately more work. This is achieved using chare migratability, and the RTS's ability to keep track of processor speeds and chare loads (as well as chare communication patterns).

In fact, suppose we add some kind of dynamic load imbalance to the problem. Say, by adding particles to our simulation. The only additional change you have to do is to make the call to balance load every few iterations, instead of doing it at the very beginning.

Now, suppose, you want to checkpoint the state of the program periodically, so that you can start from the last checkpoint if the program runs out of allocated time before it finishes. Well, since the system knows how to migrate your chares to other processors, it can migrate them to disk! The RTS has to perform some complicated maneuvering to ensure its own state is stored accurately when a checkpoint is take; but for the application programmer, the code involved is simply calling `CkStartCheckpoint(char * dir, const CkCallback& cb)` collectively from every chare, every certain number of iterations. Additionally, for most applications, you can restart using the saved files on a different number of processors. The set of original chare objects is just scattered on a different number of processors.

This kind of disk-based checkpoint is not quite true "fault-tolerance" because it does not include automatic restart. But CHARM++ also supports true fault tolerance strategy: it is called *in-memory double checkpointing.* In this case, you would have to provide simple link-time and compile-time options (the details of which can be found in the CHARM++ manual and tutorials). After that, in your code, you simply add a different call to checkpoint (instead of the disk-based checkpoint call mentioned above). To test this, suppose this program is running on a cluster of workstations. At any random point in time, try just killing the application process on one of the processors (in Linux, just `kill -9 PID`). The system will detect that one of the processes has died, retrieve the in-memory checkpoint, roll back to it, and continue execution from it in less than a second! Of course, on proprietary machines such as Cray and IBM installations, or clusters running under a job scheduler, there is a problem: the job schedulers running on today's supercomputers will simply kill a job if one node dies. However, as the schedulers get more sophisticated, they will support such fault tolerance strategies; this is starting to happen on some of the current supercomputers.

2.2 Multiphysics Modules Using Multiple Chare Arrays

One of the powerful techniques one can use in designing parallel CHARM++ programs is to use multiple chare arrays. This can be used to eliminate un-

necessary "coupling" [2] between software modules, and to promote modularity as well as facilitate collaborative development of software.

As a first simple example, consider a hypothetical simulation that involves

- a solid modeling component (e.g. structural dynamics) that requires the use of unstructured (say, tetrahedral) meshes, which are partitioned using a program such as ParMETIS or Scotch.
- a fluid modeling component that uses structured grids, partitioned using a different partitioner.

In a traditional MPI application, one either uses spatial decomposition, dividing the set of processors between the two components, and thus sacrificing efficiencies of using idle times of one module for advancing the computation of the other. Alternatively, one can decompose both computations on the same set of processors. However, with the latter approach, multiple artificial couplings develop. The number of partitions of solids must be the same as that for fluids (equal to the number of processors). Further, the partitions assigned to, say, the 200th processor for the fluids solids are simply brought together because their partitioners called them 200'th! Of course, more sophisticated means can be used, but at a significant cost to the programmer. With CHARM++, the solids can be partitioned into n partitions, while the fluid data is partitioned in m components (i.e. m and n are not necessarily equal). Further, the runtime controls the mapping and may bring together the pieces that communicate with each other the most.

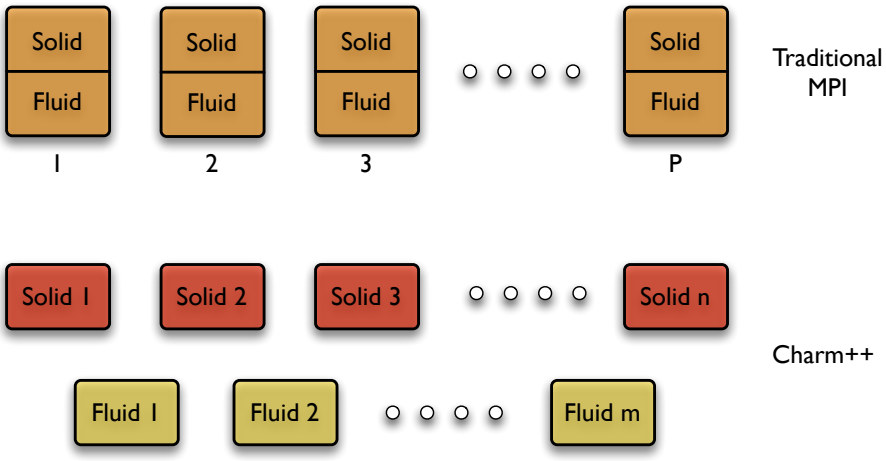

FIGURE 2.2: Decomposition in traditional MPI versus CHARM++

Another example of the use of multiple chare arrays is provided by NAMD

[2] In software engineering terminology, such coupling is considered detrimental to good software structure.

(See Chapter 4), which uses separate sets of chares for atoms, for pairwise explicit force calculations, for bonded force calculations, and several chare arrays for the particle-mesh Ewald sub-algorithm. Even more dramatic is the use of over a dozen Chare arrays in OpenAtom (Chapter 5) for representation of the electronic structure, as well as for intermediate parallel data structures.

Multiple chare arrays are also useful in building highly reusable libraries. For example, a sorting library can use a chare array, that can be "bound" to the application's chare array (so that the corresponding elements migrate together and are kept on the same common processor as each other), and can carry out sorting of the data supplied by the application chares.

2.2.1 LeanMD

To illustrate the process of using multiple chare arrays for improving modularity and increasing parallelism, let us consider *LeanMD*. LeanMD is a mini-app meant to mimic the structure of the dominant aspects of NAMD, and illustrates this usage of multiple arrays in a much simpler context with a few hundred lines of code [127, 126, 2].

The simulation consists of a set of particles (such as a set of inert gas atoms). For simplicity, we will assume that the particles are confined to a 2-dimensional periodic box. Each particle experiences a force due to every other particle (for example, Van der Waal's force); however, the force decays sharply over distance (we assume), and so we can ignore forces due to all atoms beyond a certain cut-off distance. In each time-step, we calculate and add up the forces experienced by each atom, and use them to calculate new accelerations, velocities, and positions of each particle using standard Newtonian physics.

The first design decision is how to decompose the data into objects (chares). A simple idea is linear decomposition: the first k particles go the 0'th chare, and so on. But then, every pair of chares will have to exchange particles to see any of them exert forces on the other. Instead, we decompose particles based on their coordinates into boxes. If we choose the size of each box to be slightly larger than the cutoff distance, each box will need particles from only the neighboring 8 boxes. With MPI, at this point in the design, you will start thinking about a multiblock code, and writing the coordination code for managing multiple boxes on a processor. But with Charm++, you just think of each box as a virtual processor (i.e. a chare) by itself, and describe its life cycle in a coherent code, without thinking about the physical processor. Also, at this point, you will do a simple grainsize analysis to decide if it is worthwhile using a finer decomposition (say, sizes of boxes being about half of the cutoff distance, and allowing interactions with boxes in a larger neighborhood). For the simplicity of this example let us stay with the basic decomposition. This immediately requires us to program for exchange of particles after every timestep so that each particle is in the correct box. (We can reduce that communication by increasing the box size slightly and doing particle exchange after multiple steps.).

The interesting issue is the scheme for calculating forces. Since neighboring boxes must interact, a straightforward idea is to send the particles of a box to all the neighboring boxes, get the particles from the neighboring boxes, and calculate and add up forces on your own particles from all the relevant particles (that you now have). However, because of Newton's third law, we need to calculate forces between each pair of atoms (and by extension, each pair of boxes) only once; so we are duplicating the force computation work. Since this work is known to be over 90% of the overall computation, we wish to avoid the duplication.

One could design complicated schemes for deciding which box calculates forces for which neighbor. But a simpler alternative is presented by our ability to use multiple chare arrays. For each pair of neighboring boxes, we postulate an *interaction* object (see Figure 2.3). These interaction objects can be organized into a chare array. A good index structure for this array is to use a 4-dimensional sparse array: The interactions between box (4,8) and (4,7) is calculated by an interaction object with index (4,7,4,8). We sequence the two indices in lexicographic order, so as to avoid duplication (so that we do not think of this interaction object as (4,8,4,7) as well). As another benefit, the interactions between two boxes can now be calculated on a third processor, different than the processors where the boxes live, if the load balancer so desires. This idea (of force computations on a different processor), which was later *proposed* independently by multiple researchers [69, 229, 30], was used naturally in the original NAMD design [128] because we were thinking in terms of object-based decomposition.

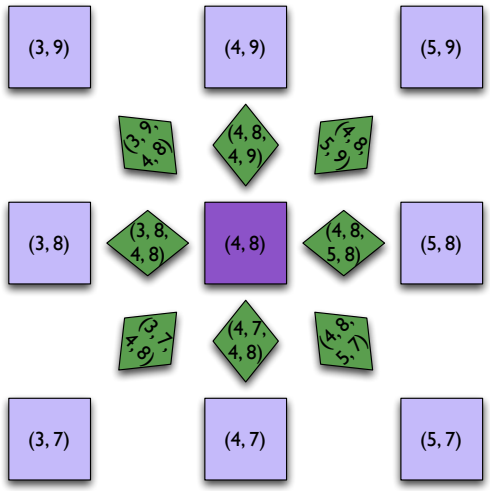

FIGURE 2.3: Use of interaction objects for force calculations in LeanMD

Now that the design is done, and especially with factoring of the code between the two types of objects, the code becomes relatively simple: The

```
array [2D] Box {
...
  entry void run() { // the sdag entry method of a box
    for (t=0; t<steps; t++) {
      myInteractions.coordinates(C);
      // broadcast coordinates to the section comprising
      // interaction objects that need my coordinates

      // forces received via a section reduction
      when forces(vector <Force> f)
        serial { integrate(f); } // and update coordinates C
      send particles to neighbors;
      for (i=0; i<numNeighbors; i++) {
        when moveAtoms(vector <Atoms> A)
          serial { mergein(A,C); }
      }
      if (t%M == 0) { BalanceLoad(); when doneBalancing() {} }
      if (t%N == 0) { CkStartMemCheckpoint(..); when ckptDone
          () {} }
    }
  };
...
}
```

FIGURE 2.4: Skeletal code for the Box class

life-cycle of each object is very cleanly expressed: a box object repeatedly broadcasts its coordinates to its associated interaction objects, receives forces from them, adds them up and updates positions of each of its particles. It then (periodically) sends particles that have moved out of its bounds to the neighboring boxes. No particles move so fast as to cross more than one box boundary; or else you are using too large a time step. The almost-real pseudocode in Figure 2.4 illustrates the box's code. We omit the detailed syntax, as well as details of initialization, to avoid getting into a tutorial discussion of syntax, which is out of scope for this book. The complete example code, with a 3-dimensional simulation, is available publicly [127].

The interaction object's life-cycle is even simpler: in each iteration it waits for particles form the boxes that are connected with it, calculates forces, and sends them back to the two boxes. We showed that code in Chapter 1. We just need to add the two lines for load balancing and fault-tolerance (based on in-memory checkpoint with automatic restart) to that code to match the box's code above.

Of course, the sequential details of interaction and integration, and some parallel initializations, must be coded by the user. The rest of the coordination of which boxes and interface objects live on which processor, how to sequence their execution to increase the overlap and so on, are left to the runtime system. By writing simple PUP routines as described above the code can

```
array [4D] Interaction {
// Each Interaction object is a member of a 4D chare array
...
  entry void run() {
    for (t=0; t<steps; t++) {
      when coordinates(vector <Atom> C1),
           coordinates(vector <Atom> C2)
        serial { calculateInteractions(C1, C2);
                 sendForcesBack();}
      if (t%M == 0) { BalanceLoad(); when doneBalancing() {} }
      if (t%N == 0) { CkStartMemCheckpoint(..); when ckDone()
          {} }
    }
  };
}
```

FIGURE 2.5: Skeletal code for the Interaction class

do load balancing, automatic fault tolerance, (and/or checkpointing to disk), with just a couple of additional lines of code, as shown in Figure 2.5.

2.3 SAMR: Chare Arrays with Dynamic insertion and Flexible Indices

The stencil example above showed that the chares can be organized into two-dimensional or three-dimensional arrays. The LeanMD example uses a two-dimensional and another four-dimensional array of chares. However, the chares can be organized into even more sophisticated and general indexing structures. In particular, one can use bit vectors as indices. In addition, one can insert and delete elements (i.e. individual chare objects) from arrays of chares, and this can be done dynamically as the computation evolves.

We will illustrate both these features using structured adaptive mesh refinement (SAMR) as an example. SAMR is used in physical simulations where the degree of spatial resolution needed varies significantly from region to region. In one particular formulation that we will be our focus, the region is organized as an octree. For the sake of simplicity, let us consider a 2-dimensional example, where we will use a quad-tree instead. The leaves of the tree represent regions that are being explicitly simulated, using a structured grid (i.e. a mesh). The internal nodes in the tree represent regions that have been adaptively refined.

In a simple scenario, computation may begin with a tree of uniform depth. Assume that each leaf has a 512x512 chunk of data. As the simulation evolves, the numerics might indicate that some regions represented by some of the leaves need higher resolution, and so need to be refined. With refinement, the 512x512 grid of the leaf needs to become a 1024x1024 grid. This is accomplished by creating 4 more leaves which become children of the leaf being refined. Although each region can only be refined once during a single time step, over a series of time steps some regions can get very deeply refined, so the maximum and minimum depth of the tree can be widely different. Also, as simulation progresses, some regions may not need the resolution we currently have. If all the 4 children (which happened to be leaves) of an internal node N wish to be coarsened, they can be absorbed in N, thus deleting the 4 leaves.

Charm++ provides a natural way of representing this computation. Each node of the quad-tree, including the leaves, can be implemented as a chare belonging to a single chare array. Each element of this chare array is represented by a bit vector index. In the current version of Charm++, array indices can be 128 bits long. We create a natural indexing scheme that assigns a unique index to each node of the tree as follows: We reserve 8 bits to encode the depth of the node (the depth of the root being 0), and the remaining bits to encode the branches one takes from the root to get to the node. Thus a node with index (d, b) has 4 children, (d+1, b00), (d+1, b01), (d+1, b10) (d+1, b11), encoded in the 128 bits in the obvious way, with the unused bits at the tail end being set to 0. Figure 2.6 illustrates this indexing scheme, using a smaller number of bits for ease of illustration. Note that the rightmost bits that are

not pertinent to the scheme (and are set to 0 for a canonical representation) are omitted by showing them as blanks in the figure.

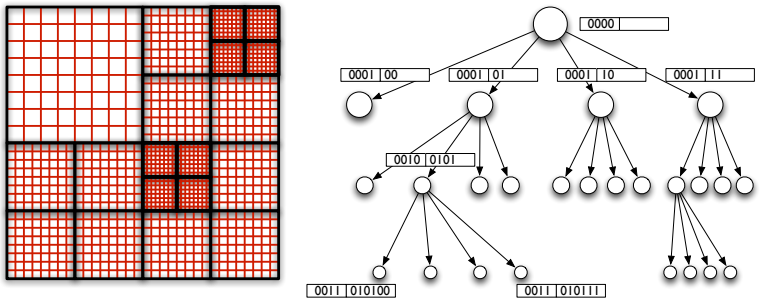

FIGURE 2.6: BitVector Indexing of Chares for AMR

With this indexing, a chare can find the index of its parent and the neighbors (for boundary-exchanges) by simple local operations based on the knowledge of its own index. Further, if a chare decides to refine, it can locally calculate indices of the 4 new chares it must create and insert into the chare array, as described in the above paragraphs. The CHARM++ load balancer decides on which processor each chare resides based on predicted loads, while a built-in scalable location manager handles delivery of messages directed at specific indices to the correct processor.

An additional benefit of the CHARM++ model is that the code is written from the point of view of each leaf, i.e. a block of uniformly refined mesh, along with much simpler code for the chares corresponding to the internal nodes of the tree. This is unlike typical codes for the same purpose in MPI, which must be written from the point of view of a *processor*, where multiple blocks are housed.

A specific implementation of the 2-D SAMR is described in a recent paper [148]. This paper also illustrates how SAMR codes can benefit from the support for asynchrony in CHARM++ by eliminating multiple synchronizations in typical re-meshing steps in such codes.

2.4 Combinatorial Search: Task Parallelism

The phrase "task parallelism" is used by different researchers in different senses. For example, some researchers have used it to simply connote any use of message-driven execution. A `when block` (see Chapter 1) in `Structured Dagger` is ready to execute as soon as the object has reached the statement, and the relevant messages have arrived. This *task* is then kept

in the *local* scheduler's queue of ready tasks from where tasks are picked up one at a time, non-preemptively. However, this is not what we mean here by task parallelism. The pattern we are describing has been called "agenda parallelism", and it occurs in combinatorial search and divide and conquer applications.

As an example, consider the problem of finding a k-coloring for a given graph. One can begin with a state in which no vertex of the graph is colored; when you consider each state, you select one of the vertices to color, color it with each of the possible colors that are consistent with the constraint (that no 2 neighboring vertices should have the same color), and thus create several child states. This then defines a search tree.

How can one parallelize this application using CHARM++? This is facilitated by CHARM++'s support for dynamic creation of chares. The graph itself does not change during the execution, and so can be represented as a read-only variable/structure. Each chare's state consists of a table of vertices that have already been colored, and the assigned colors for those vertices. When it starts execution, each chare heuristically selects an uncolored vertex, and fires off a new chare corresponding to each possible color that can be assigned to the selected vertex, sending the color-map as a constructor message. Sophisticated heuristics are used to ensure that the color assignment does not create an easy-to-infer infeasibility, and for selecting a good vertex to color next.

One must exercise reasonable *grain size* control in this application to avoid excessive overhead on one hand, and serialization on the other hand. Again, the guiding principle is that no single chare should be too large (i.e. its computational work should be substantially smaller than the average work per chare), and the average work for each chare should be significantly larger than the overhead of scheduling and load balancing it. A simple strategy is to decide to explore a state sequentially when the number of vertices that remain to be colored falls below a threshold. More sophisticated strategies can be used, as described in [125].

How do we stop such a computation? It is not adequate to specify that once a chare finds a solution, it will print the solution and call `CkExit()`. For one thing, you may be interested in all the solutions; in any case, you must also cater to the case when there are no solutions. Both of these challenges can be resolved by using CHARM++'s built-in quiescence detection library. It employs an algorithm that runs in the background, and reports via a callback when no computations are executing, and no messages are in transit.

Interestingly, this kind of dynamic creation of tasks can co-exist (in a single application) with more structured and iterative computations typically expressed with chare arrays. The message-driven execution combined with load balancing capabilities ensure that such an application is feasible and will work efficiently. At the most, depending on the context, it may require a more specialized set of load balancers, since CHARM++ uses separate balancers for such dynamically created tasks.

2.5 Other Features and Design Considerations

Let us consider some additional features and discuss how and under what conditions are they desirable.

Priorities: Recall that there are potentially multiple "messages" (i.e. asynchronous method invocations and ready threads) awaiting execution in the scheduler's pool on any PE (processor). By default, the system executes them in FIFO (first-in-first-out) order. However, in some situations the programmer may wish to influence this order. This can be accomplished by associating a priority with the method invocations (again, consult the manuals for the details of how to do this). One can also declare entry methods, and therefore all invocations of them, to be "expedited", in which case they bypass the priority queue. In effect, they are treated as the highest priority messages, and are executed as soon as they are picked from the network by the scheduler.

How and when to leverage priorities? As an example, consider a situation where the work consists of two types of messages: those that have only local clients and those that have remote clients (i.e. when the work is done, you have to send the result to a potentially remote chare). The latter work should have higher priority, because someone else is waiting for them, and the network latency will delay them. In general, work on the critical path of the computation can be assigned higher priority. Decisions like this are often taken after visualizing program performance via the *Projections* tool (See Chapter 3), and identifying possible bottlenecks.

Threads vs Structured Dagger: For truly reactive objects, which do not know which of their methods will be called and how many times, the generality provided by the baseline CHARM++ methods is adequate and appropriate. However, if an object's life cycle is statically described, then one should use either `Structured Dagger` or a threaded entry method to code it. Threaded entry methods should only be used if `Structured Dagger` is inadequate to express the control flow. This is because a `Structured Dagger` entry method typically captures the parallel life cycle of an object, including all its remote dependencies, in a simple script, separating parallel and sequential code naturally. However, for example, if the control is deep in a function call stack, and you need a remote value, it is more convenient to use a threaded method. Threaded methods, although very efficient, are still not as efficient as the `Structured Dagger` methods: their context switching time (which is typically less than a microsecond) involves switching user-level stacks, and the need to allocate and copy/serialize the stack also adds its own overhead. Scheduling overhead for `Structured Dagger` (as well as baseline CHARM++ methods) is consists of a few function calls, amounting to tens to a hundred nanoseconds on current machines.

2.6 Utility of Charm++ for Future Applications

Even though Charm++ is a mature system, its signature strengths, arising from an introspective and adaptive runtime system, make it a system well-suited for addressing the challenges of the upcoming era of increasingly sophisticated applications and increasingly complex parallel machines. This is true at both the extreme scale machines, beyond the current generation of petascale computers, as well as the much smaller department-size parallel machines that are expected to be ubiquitous.

Sophisticated applications, when given a larger computer, do not increase the resolution everywhere; instead, they tend to use dynamic and adaptive refinements to best exploit the extra compute power. Sophisticated applications also use multiple modules, typically for simulating different physical aspects of the phenomena being studied. Both of these trends are likely to strengthen in future. The dynamic load balancing capabilities, as well as the ability to support multiple modules are critical for these applications. Yet as illustrated in this chapter, with the solid-fluid example as well as the SAMR example, Charm++ is well-suited to express such applications with high programmer productivity.

Power and energy considerations make future machines more complex; some predictions for the future also include components failing more frequently than they do now. Again, the introspection and adaptivity, and the concomitant dynamic load balancing and fault tolerance capabilities in Charm++, help alleviate the programmer burden in dealing with such machines.

Thus, we expect the Charm++ programming model to serve the parallel applications community very well in the coming years. Of course, the runtime system itself will need improvements and modifications to cope with new hardware challenges it will surely face.

2.7 Summary

We discussed, through a series of examples, how to go about designing a Charm++ application. The stencil code illustrated how to keep the code processor-independent, and how to think about and control the grainsize of chares. Multi-physics codes such as the solid-fluid simulation, as well as molecular dynamics codes, illustrated how multiple collections of chares (Chare Arrays) can be used to cleanly express the logic of the program. Dynamic insertion/deletion and flexible indexing structures were illustrated via the structured AMR example. We also saw how to deal with task parallelism. Finally, we learned a bit more about how to use priorities and how to choose between

threaded, structured-dagger, and simple entry methods. Obviously, elaborating and describing the whole design process, with many additional features of CHARM++ and details of their use, is beyond the scope of this book. For this, we request the reader to consult the online and other upcoming tutorials on programming with CHARM++.

Acknowledgements

Work on the CHARM++ system was carried out over two decades by generations of graduate students and staff members, and each has left his or her stamp on the software and the design. Many research grants over those years from U.S. agencies including the National Science Foundation, Department of Energy, National Institutes of Health and National Aeronautics and Space Administration have directly or indirectly contributed to the development of CHARM++ or its applications. Some recent grants that directly funded this work are the NSF HECURA program (NSF 0833188) and the DOE FAST-OS program (DE-SC0001845). I also thank all those who helped in writing an preparing the first two chapters, including significant help from Abhinav Bhatele, especially for many nice figures he created.