TITLE

Charm++

BYLINE

Laxmikant V. Kalé

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
kale@illinois.edu

SYNONYMS

DEFINITION

Charm++ is a C++-based parallel programming system that implements a *message-driven migratable objects* programming model, supported by an adaptive runtime system.

DISCUSSION

Charm++ [1] is a parallel programming system developed at the University of Illinois at Urbana-Champaign. It is based on a message-driven migratable objects programming model, and consists of a C++-based parallel notation, an adaptive runtime system (RTS) that automates resource management, collection of debugging and performance analysis tools, and an associated family of higher level languages. It has been used to program several highly scalable parallel applications.

Motivation and Design Philosophy

One of the main motivations behind Charm++ is the desire to create an optimal division of labor between the programmer and the system: i.e., to design a programming system so that the programmers do what they can do best, while leaving to the "system" what it can automate best. It was observed that deciding what to do in parallel is relatively easy for the application developer to specify; conversely, it has been very difficult for a compiler (for example) to automatically parallelize a given sequential program. On the other hand, automating resource management — which subcomputation to carry out on what processor and which data to store on a particular processor — is something that the system may be able to do better than a human programmer, especially as the complexity of the resource management task increases. Another motivation is to emphasize the importance of data locality in the language, so that the programmer is made aware of the cost of non-local data references.

Programming Model in Abstract

A Charm++ computation consists of collections of objects that interact via asynchronous method invocations. Each object is called a *chare*. The chares are assigned to processors by an adaptive runtime system, with an optional override by the programmer. A chare is a special kind of C++ object. Its behavior is specified by a C++ class that is "special" only in the sense that it must have at least one method designated as an "entry" method. Designating a method as an entry method signifies that it can be invoked from a remote processor. The signatures of the entry methods (i.e., the type and structure of its parameters) are specified in a separate interface file, to allow the system to generate code for packing and unpacking the parameters into messages. (This is also called "serializing" or "marshalling" the parameters.) Other than the existence of the interface files, a Charm++ program is written in a manner very similar to standard C++ programs, and thus will feel very familiar to C++ programmers.

The chare objects interact with each other through asynchronous method invocations. Such a method invocation does not return any value to the caller, and the caller can continue with its own execution. Of course, the called chare may choose to send a value back by invoking another method in the caller object. Each chare has a globally valid ID (its *proxy*), which can be passed around via method invocations. Note that the programmer refers to only the target chare by its global ID, and not by the processor on which it resides. Thus, in the baseline Charm++ model, the processor is not a part of the ontology of the programmer.

Chares can also create other chares. Such creations are also asynchronous in that the caller does not wait until the new object is created. Programmers typically do not specify the processor on which the new chare is to be created; the system makes this decision at runtime. The number of chares may vary over time, and is typically much larger than the number of processors.

Message Driven Scheduler

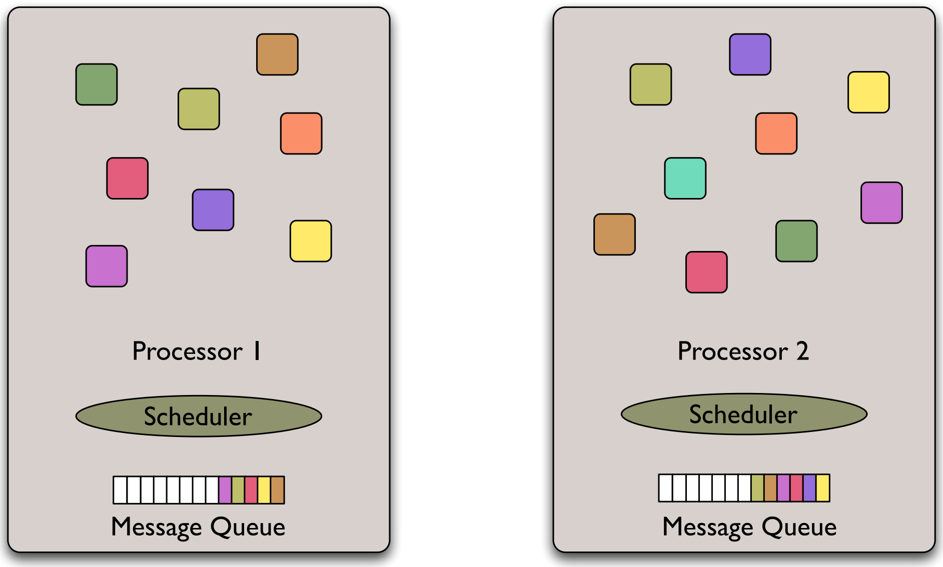

Figure 1: Message driven scheduler

Since there are many chare objects per processor, with potentially multiple asynchronous method invocations pending for each, their execution requires a scheduler. The Charm++ runtime system employs a simple user-level scheduler for this purpose, as shown in Figure 1. The scheduler is user-level in the sense that the operating system is not aware of the scheduler. Normal Charm++ methods are non-preemptive: once a method begins execution, it returns control to the scheduler only after it has completed execution. The scheduler works with a queue of pending entry methods. There is one scheduler on each processor. Note that this queue may include asynchronous method invocations for chares located on this processor, as well as "seeds" for new chares: these can be thought of as invocations of the constructor entry method. The scheduler repeatedly selects a message (a method invocation) along with the proxy of the chare to which the message is targeted, identifies the object targeted, creating an object if necessary, unpacks the parameters from the message if necessary, and then invokes the specified method with the parameters. Only when the method returns does it select the next message and repeats the process.

Chare-Arrays and Iterative Computations

The model described so far, with its support for dynamic creation of work, is well-suited for expressing the divide-and-conquer as well as divide-and-divide computations. The latter occur in the state-space search. Charm++ (and its C-based precursor, Charm, and Chare Kernel [2]) were used in the late 1980s for implementing parallel Prolog [3] as well as several combinatorial search [4] applications.

Science and engineering computations typically involve domain decomposition. The simple chares above *can* be used to create networks of chares. For example, one can organize chares in a two-dimensional mesh like network, and through some additional message passing, ensure that each chare knows the ID of its four neighboring chares. However, this method of creating a network of chares is quite cumbersome. Instead Charm++ supports indexed collections of chares, called chare-arrays. A Charm++ computation may include multiple chare-arrays. Each chare array is a collection of chares of the same type. Each chare is identified by an index that is unique within its collection. Thus, an individual chare belonging to a collection of chares (i.e. a chare-array) is completely identified by the ID of the collection, and its own index within the collection. Common index structures include dense as well as sparse multidimensional arrays, but arbitrary indices such as strings or bit vectors are also possible. Elements in a chare-array may be created all at once, or can be inserted one at a time. Method invocations can be broadcast to an entire chare-array by omitting the index in the call. Reductions over arrays are also supported, where each chare in a chare-array contributes a value, and all submitted values are combined via a commutative-associative operation. Unlike in other programming models, where the reductions are a collective operation that block all callers, reductions in Charm++ are non-blocking, i.e., asynchronous. The `contribute` call simply deposits the value created by the calling chare into the system, and continues on. At some later point after all the values have been combined, the system delivers them to a user-specified callback. The callback could be a broadcast to an entry method of the same chare-array, for example. Charm++ does not allow generic global variables, but it does allow "specifically shared variables". The simplest of these are read-only variables, which are initialized in the main chare's constructor, and are treated as constants for the remainder of the program. The runtime system (RTS) makes sure that a copy of each read-only variable is available on all physical processors.

Benefits of Message-Driven Execution

The message-driven execution model confers several performance and/or productivity benefits.

Automatic and adaptive overlap of computation and communication

Since objects are scheduled based on availability of messages, no single object can hold the processor while it waits for some remote data. Instead, objects that have asynchronous method invocations (messages) waiting for them in the scheduler's queue are allowed to execute. This leads to a natural overlap of communication and computation, without extra work from the programmer. E.g., a chare may send a message to a remote chare, and wait for another message from it before continuing. The ensuing communication time, which would otherwise be an idle period, is naturally and automatically filled in (i.e., overlapped) by the scheduler with useful computaion, i.e., processing of another message from the scheduler's queue for another chare.

Concurrent composition

The ability to compose in parallel two individually parallel modules is refered to as concurrent composition. Consider two modules P and Q that are both ready to execute and have no direct dependencies among them. With other programming models (e.g. MPI) that associate work directly with processors, one typically has two options: Either one divides the set of processors so that a subset is executing one module (P) while the remaining processors execute the other (Q). Alternatively, one sequentializes the modules, executing P first, followed by Q, on all processors. Neither alternative is efficient. Allowing the two modules to interleave the execution on all processors is often beneficial, but is hard to express even with wild-card receives, and it breaks abstraction boundaries between the modules in any case. With message driven execution, such interleaving happens naturally, allowing idle time in one module to be overlapped with computation in the other. Coupled with the ability of the adaptive runtime system to migrate communicating objects closer to each other, this adds up to strong support for concurrent composition, and thereby for increased modularity.

Prefetching data and code

Since the message driven scheduler can peek ahead at its queue, it knows what the next several objects scheduled to execute are and what methods will they be executing. This information can be used to asynchronously prefetch data for those objects, while the system executes the current object. This idea is used by the Charm++ runtime system for increasing efficiency in various contexts, including on accelerators such as the Cell processor, for out-of-core execution and for prefetching data from DRAM to cache.

Capabilities of the adaptive Runtime System Based on Migratability of Chares

Other capabilities of the runtime system arise from the ability to migrate chares across processors, and the ability to place chares on processors of its choice when they are created.

Supporting Task Parallelism with Seed Balancers

When a program calls for the creation of a singleton chare, the RTS simply creates a seed for the new chare, which includes the constructor arguments and class information needed to create a new chare. Typically , these seeds are initially stored on the same processor where they are created, but may be passed form processor to processor under the control of a runtime component called the seed balancer. Many different seed balancer strategies are available to be linked in. For exaample, a strategy may be to monitor queue size on one's own processor and neighboring processors and balance the queue of seeds as it sees fit. Another strategy might request work from a random processor when the local processor goes idle — a work stealing scheme [5, 6] . Charm++ also includes strategies that balance priorities and workloads simultaneously, trying to ensure high priority work gets executed faster over the entire system.

Migration-based load balancers

Elements of chare-arrays can be migrated across processors, either explicitly by the programmer or by the runtime system. The Charm++ RTS leverages this capability to provide a suite of dynamic load balancing strategies. One class of such strategies is based on the *principle of persistence*: in most science and engineering applications expressed in terms of their natural objects, computational loads and communication patterns tend to persist over time, even for dynamically evolving applications. Thus, recent past is a reasonable predictor of near future. Since the runtime system mediates communication and schedules computations, it can automatically instrument its execution so as to measure computational loads and communication patterns accurately. Load balancing strategies can use these measurements, or alternatively, any other mechanisms for predicting such patterns. Multiple load balancing strategies are available to choose from. The choice may depend on the machine context and applications, although one can always use the default strategy provided. Programmers can write their own strategy, either to specialize it to the specific needs of the application or in the hope of doing better than the provided strategies.

Shrinking or Expanding the Sets of Processors

A Charm++ program can be asked to change the set of processors it is using at runtime. This requires no effort by the programmer. It accomplishes this by migrating objects and adjusting its runtime data structures, such as spanning trees used in its collective operations. Thus, a 1024x1024x1024 cube of data partitioned into 16x16x16 array of chares, each holding 64x64x64 data subcube, can be shrunk from 256 cores to 247 cores (to pick an arbitrary number) without significantly losing efficiency. Each core will house 17 objects at most, instead of 16 it did before.

Fault tolerance

Charm++ provides multiple levels of support for fault tolerance, including alternative competing strategies. At a basic level it supports automated application-level checkpointing, by leveraging its ability to migrate objects. With this, it is possible to create a checkpoint of the program without writing extra user code. More interestingly, it is also possible to use a checkpoint created on P processors to restart the computation on a different number of processors than P.

On appropriate machines and with job schedulers that permit this, Charm++ can also automatically detect and recover from faults. It requires that the job scheduler not kill the job if one of its node fails. At the time of this writing, these schemes are available on workstation clusters. The most basic strategy uses the checkpoint created on disk, as described above, to effect recovery. A second strategy avoids using disks for checkpointing, creating multiple copies of the chare objects in the memory of two processors instead. It is suitable for those applications whose memory footprint at the point of checkpointing is relatively small compared with the available memory. Fortunately, many applications such as molecular dynamics and computational astronomy fall into this category. When it does apply, it is very fast, often accomplishing checkpointing in less than a second, and recovery in a few seconds. Both the above strategies send all the processors back to their checkpoints even when just one out of a million processors has failed. This wastes all the computation performed, and is ultimately not sustainable as the number of processors increases and the mean time between failures (MTBF) decreases sufficiently. A third experimental strategy in Charm++ sends only the failed processor(s) to their checkpoints by using a message-logging scheme, and leverages the over-decomposition and migratability of Charm++ objects to parallelize the restart process. I.e., the objects from failed processors are reincarnated on multiple other processors, where they re-execute, in parallel, from their checkpoints using the logged messages. Charm++ also provides a fourth *proactive* strategy to handle situations where a future fault can be predicted, say based on heat sensors, or estimates of increasing (corrected) cache errors. The runtime simply migrates objects away from such a processor, and readjusts its runtime data structures.

Associated Tools

The Charm++ system has several tools associated with it. *LiveViz* allows one to inject messages into a running program and display attributes and images from a running simulation. *Projections* supports performance analysis and visualization, including live visualization, parallel on-line analysis, and log-based post-mortem analysis. *CharmDebug* is a parallel debugger that understands Charm++ constructs, and provides online access to runtime data structures. It also supports a sophisticated record-replay scheme and provisional message delivery for debugging non-deterministic bugs. The communication required by these tools is integrated in the runtime system, leveraging the message-driven scheduler. No separate monitoring processes are necessary.

Code Example

Figures 2 and 3 show fragments from a simple Charm++ example program to give a flavor of the programming model. The program is a simple Lennard-Jones molecular dynamics code: the computation is decomposed into a 1-dimensional array of LJ objects, each holding a subset of atoms. The interface file (Fig. 2) describes the main Charm-level entities, their types and signatures. The program has a read-only integer called `numChares` that holds the size of the LJ chare-array. In this particular program, the main chare is called Main and has only one method, namely its constructor. The LJ class is declared as constituting a 1-dimensional array of chares. It has two entry methods in addition to its constructor. Note the `others[n]` notation used to specify a parameter that is an array of size `n`, where `n` itself is another integer parameter. This allows the system to generate code to serialize the parameters into a message, when necessary. CkReductionMsg is a system defined type, and is used as a target of entry methods used in reductions.

```
mainmodule ljdyn {
  readonly int numChares;
  ...
  mainchare Main {
    entry Main(CkArgMsg *m);
  };

  array [1D] LJ {
    entry LJ(void);
    entry void passOn(int home, int n, Particle others[n]);
    entry void startNextStep(CkReductionMsg *m);
  };
};
```

Figure 2: A simple molecular dynamics program: interface file

Some important fragments from the C++ file that define the program itself are shown in Figure 3. Sequential code not important for understanding the program is omitted. The classes `Main` and `LJ` inherit from classes generated by a translator based on the interface file. The program execution consists of a number of time steps. In each time step, each processor sends its particles on a round-trip to visit all other chares. Whenever a packet of particles visits a chare via the `passOn` method, the chare calculates forces on each of the visiting (`others`) particles due to each of its own particles. Thus, when the particles return home after the round trip, they have accumulated forces due to all the other particles in the system. A sequential call (`integrate`) then adds local forces to the accumulated forces, and calculates new velocities and positions for each owned particle. At this point, to make sure the time step is truly finished for all chares, the program uses an "asynchronous barrier" via the `contribute` call. To emphasize the asynchronous nature of this call, the example makes the call before `integrate`. The contribute call simply deposits the contribution into the system and continues on to `integrate`. The `contribute` call specifies that after all the array elements of LJ have contributed, a callback will be made. In this case, the callback is a broadcast to all the members of the chare-array at the entry-method `startNextStep`. Inherited variable `thisproxy` is a proxy to the entire chare-array. Similarly, `thisIndex` refers to the index of the calling chare in the chare-array to which it belongs.

Language Extensions and Features

The baseline programming model described so far is adequate to express most parallel interaction structures. However, for programming convenience, increased productivity and/or efficiency, Charm++ supports a few additional features. For example, individual entry methods can be marked as "threaded". This results in the creation of a user level, lightweight thread whenever the entry method is invoked. Unlike normal entry methods, which always complete their execution and return control to the scheduler before other entry methods are executed, the threaded entry methods can block their execution; of course they do so without blocking the processor they are running on. In particular, they can wait for a "future", wait until another entry method unblocks them, or make blocking method invocations. An entry method can be tagged as blocking (actually called a

```
/*readonly*/ int numChares;

class Main : public CBase_Main {
  Main(CkArgMsg* m)  {
    // Process command-line arguments
    ...
    numChares = atoi(m->argv[1]);
    ...
    CProxy_LJ arr = CProxy_LJ::ckNew(numChares);
  }
};

class LJ : public CBase_LJ {
  int timeStep, numParticles, next;
  Particle * myParticles;
  ...

  LJ() {
    ...
    myParticles = new Particle[numParticles];
    ...    // initialize particle data
    next = (thisIndex + 1) % numChares;
    timeStep = 0;
    startNextStep((CkReductionMsg *)NULL);
  }

  void startNextStep(CkReductionMsg* m)  {
    if (++timeStep < MAXSTEPS) {
      if (thisIndex == 0) { ckout << "Done\n" << endl;  CkExit(); }
    } else
      thisProxy[next].passOn(thisIndex, numParticles, myParticles);
  }

  void passOn(int homeIndex, int n, Particle* others)  {
    if (thisIndex != homeIndex) {
      interact(n, others); //add forces on "others" due to my particles
      thisProxy[next].passOn(homeIndex,n, others);
    } else { // particles are home, with accumulated forces
      CkCallback cb (CkIndex_LJ::startNextStep(NULL), thisProxy);
      contribute(cb); // asyncronous barrier
      integrate( n, others); // add forces and update positions
    }
  }

  void interact(int n, Particle* others) {
    /* add forces on "others" due to my particles */
  }

  void integrate(int n, Particle* others) {
    /* ... apply forces, update positions ... */
  }
};
```

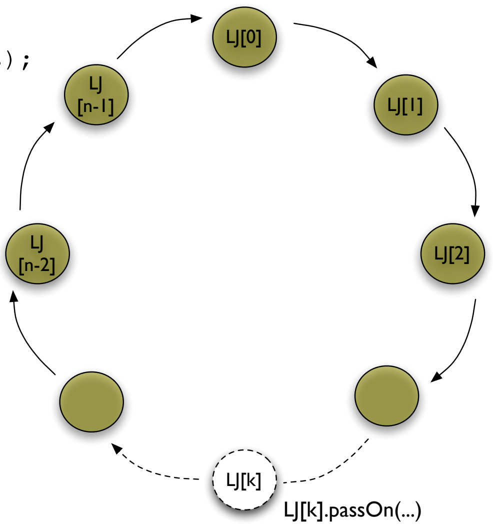

Figure 3: A simple molecular dynamics program: fragments from the C++ file

"sync" method), and such a method is capable of returning values, unlike normal methods that are asynchronous and therefore have a return type of `void`.

Often, threaded entry methods are used to describe the life cycle of a chare. Another notation within Charm++, called "structured dagger", accomplishes the same effect without the need for a separate stack and associated memory for each user level thread, and the, admittedly small, overhead associated with thread context switches. However, it requires that all dependencies on remote data be expressed in this notation within the text of an entry-method. In cotrast, the thread of control may block waiting for remote data within functions called from a threaded entry method.

Charm++ as described so far does not bring in the notion of a "processor" in the programming model. However, some low-level constructs that refer to processors are also provided to programmers and especially library writers. For example, when a chare is created, one can optionally specify which processor to create it on. Similarly, when a chare-array is created, one can specify its initial mapping to processors. One can create specialized chare-arrays, called *groups* that have exactly one member on each processor, which are useful for implementing services such as load balancers.

Languages in the Charm family

AMPI or Adaptive MPI is an implementation of the message passing interface standard on top of the Charm++ runtime system. Each MPI process is implemented as a user level thread that is embedded inside a Charm++ object, as a threaded entry method. These objects can be migrated across processors, as is usual for Charm++ objects, thus bringing benefits of the Charm++ adaptive runtime system, such as dynamic load balancing and fault tolerance, to traditional MPI programs. Since there may be multiple MPI "processes" on each core, commensurate with the overdecomposition strategy of Charm++ applications, the MPI programs need to be modified in a mechanical, systematic fashion to avoid conflict among the global variables. Adaptive MPI provides tools for automating this process to some extent. As a result, a standard Adaptive MPI program is also a legal MPI program, but the converse is true only if the use of global variables has been handled via such modifications. In addition, Adaptive MPI provides primitives such as asynchronous collectives, which are not part of the MPI 2 standard. An asynchronous reduction, for example, carries out the communication associated with a reduction in the background while the main program continues on with its computation. A blocking call is then used to fetch the result of the reduction.

Two recent languages in the Charm++ family are multiphase shared arrays (MSA) and Charisma. These are part of the Charm++ strategy of creating a toolbox consisting of incomplete languages that capture some interaction modes elegantly and frameworks that capture the needs of specific domains or data structures, both backed up by complete languages such as Charm++ and AMPI. The compositionality afforded by message driven execution ensures that modules written using multiple paradigms can be efficiently composed in a larger application.

MSA is designed to support disciplined use of shared address space. The computation consists of collections of threads and multiple user-defined data arrays, each partitioned into user-defined pages. Both entities are implemented as migratable objects (i.e., chares) available to the Charm++ runtime system. The threads can access the data in the arrays, but each array is in only one of a restrictive set of access modes at a time. *Read-only*, *exclusive-write*, and *accumulate* are examples of the access modes supported by MSA. At designated synchronization points, a program may change the access modes of one or more arrays.

Charisma, another language implemented on top of Charm++, is designed to support computations that exhibit a static data flow pattern among a set of Charm++ objects. Such a pattern, where the flow of messages remains the same from iteration to iteration, even though the content and the length of messages may change, is extremely common in science and engineering applications. For such applications, Charisma provides a convenient syntax that captures the flow of values and control across multiple collections of objects clearly. In addition, Charisma provides a clean separation of sequential and parallel code that is convenient for collaborative application development involving parallel programmers and domain specialists.

Frameworks atop Charm++

In addition to its use as a language for implementing applications directly, Charm++ is seen as backend for higher level frameworks and languages described above. Its utility in this context arises because of the interoperability and runtime features it provides, which one can leverage to put together a new domain-specific framework relatively quickly. An example of such a framework is ParFUM, which is aimed at unstructured mesh applications. ParFUM allows developers of sequential codes based on such meshes to retarget them to parallel machines, with relatively few changes. It automates several commonly needed functions including the need to exchange boundary nodes (or boundary layers, in general) with neighboring objects. Once a code is ported to ParFUM, it can automatically benefit from other Charm++ features such as load balancing and fault tolerance.

Applications

Some of the highly scalable applications developed using Charm++ are in extensive use by scientists on national supercomputers. These include NAMD (for biomolecular simulations), OpenAtom)for electronic structure simulations), and ChaNGa (for astrophysical N-body simulations).

Origin and History

The Chare Kernel, a precursor of the Charm++ system, arose from the work on parallel Prolog at the University of Illinois in Urbana-Champaign in the late 1980s. The implementation mechanism required a collection of computational entities, one for each active clause (i.e. its activation record) of the underlying logic program. Each of these entities typically received multiple responses from each one of its children in the proof tree, and they needed to create new nodes in the proof tree which had to fire new "tasks" for each of its active clauses. The implementation led to a message driven scheduler and dynamic creation of the seeds of work. Following the terminology used by an earlier parallel functional-languages project, RediFlow, these entities were called *chares*. The work on Chare Kernel essentially separated this implementation mechanism from its parallel Prolog context, into a C-based parallel programming paradigm of its own. Charm had similarities (especially the message driven execution) with the earlier research on reworking of the Hewitt's Actor framework by Agha and Yonezawa [7, 8]. However, its intellectual progenitors were in parallel logic and functional languages. With the increase in popularity of C++, Charm++ became the version of Charm for C++, which was a natural fit for its object-based abstraction. As the attention in parallel computing shifted to scientific computations, the indexed collections of migratable chares were developed in Charm++ to simplify addressing chares in mid-1990s. In the late 1990s, Adaptive MPI was developed in the context of applications being developed at the Center for Simulation of Advanced Rockets at Illinois. Charm++ continues to be developed and maintained from the University of Illinois, and its applications are in regular use at many supercomputers around the world.

Availability and Usage

Charm++ runs on most parallel machines available at the time this article was written, including multicore desktops, clusters and large-scale proprietary supercomputers, running Linux, Windows and other operating systems. Charm++, its associated software tools and libraries can be downloaded in source code and binary forms from `http://charm.cs.illinois.edu` under a license that allows free use for non-commercial purposes.

RELATED ENTRIES

Actors
NAMD
ChaNGa
Parallel Combinatorial Search

BIBLIOGRAPHIC NOTES AND FURTHER READING

One of the earliest papers on the Chare Kernel, the precursor of Charm++ was published in 1989 [9], followed by a more detailed descriptions of the model and its load balancers [10, 2]. This work arose out of earlier work on parallel Prolog [3]. The message-driven aspects of Charm++ make it an Actor-like model, which was developed in earlier research by Hewitt, Agha, and Yonezawa [7, 8]. The C++ based version was described in an OOPSLA paper in 1993 [1], and was expanded upon in a book on parallel C++ in [11]. Some of the early work on quantifying benefits of the programming model is summarized in a later paper [12].

Some of the early applications using Charm++ were in symbolic computing, and specifically, in parallel combinatorial search [4]. A scalable framework for supporting migratable arrays of chares is described in a paper [13], that is also useful for understanding the programming model as well. An early paper [14] describes support for migrating Chares for dynamic load balancing. Recent papers describe an overview of Charm++ [15] and its applications [16].

BIBLIOGRAPHY

[1] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[2] W.W. Shu and L.V. Kalé. Chare Kernel - a runtime support system for parallel computations. *Journal of Parallel and Distributed Computing*, 11:198–211, 1990.

[3] L. V. Kalé. Parallel execution of logic programs: the REDUCE-OR process model. In *Proceedings of Fourth International Conference on Logic Programming*, pages 616–632, May 1987.

[4] L.V. Kale, B. Ramkumar, V. Saletore, and A. B. Sinha. Prioritization in parallel symbolic computing. In T. Ito and R. Halstead, editors, *Lecture Notes in Computer Science*, volume 748, pages 12–41. Springer-Verlag, 1993.

[5] Yow-Jian Lin and Vipin Kumar. And-parallel execution of logic programs on a shared-memory multiprocessor. *Journal of Logic Programming*, 10(1/2/3&4):155–178, 1991.

[6] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)*, volume 33 of *ACM Sigplan Notices*, pages 212–223, Montreal, Quebec, Canada, June 1998.

[7] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[8] A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. *ACM SIGPLAN Notices, Proceedings OOPSLA '86*, 21(11):258–268, Nov 1986.

[9] L. V. Kalé and W. Shu. The Chare Kernel base language: Preliminary performance results. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages 118–121, St. Charles, IL, August 1989.

[10] L.V. Kale. The Chare Kernel parallel programming language and system. In *Proceedings of the International Conference on Parallel Processing*, volume II, pages 17–25, August 1990.

[11] L. V. Kale and Sanjeev Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[12] Attila Gursoy and L.V. Kalé. Performance and Modularity Benefits of Message-Driven Execution. *Journal of Parallel and Distributed Computing*, 64:461–480, 2004.

[13] Orion Sky Lawlor and L. V. Kalé. Supporting dynamic parallel object arrays. *Concurrency and Computation: Practice and Experience*, 15:371–393, 2003.

[14] Robert K. Brunner and Laxmikant V. Kalé. Handling application-induced load imbalance using parallel objects. In *Parallel and Distributed Computing for Symbolic and Irregular Applications*, pages 167–181. World Scientific Publishing, 2000.

[15] Laxmikant V. Kale and Gengbin Zheng. Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects. In M. Parashar, editor, *Advanced Computational Infrastructures for Parallel and Distributed Applications*, pages 265–282. Wiley-Interscience, 2009.

[16] Laxmikant V. Kale, Eric Bohm, Celso L. Mendes, Terry Wilmarth, and Gengbin Zheng. Programming Petascale Applications with Charm++ and AMPI. In D. Bader, editor, *Petascale Computing: Algorithms and Applications*, pages 421–441. Chapman & Hall / CRC Press, 2008.