Charm++ is an elegant, general-purpose parallel programming model backed by an adaptive runtime system. This combination yields portable performance and a spectrum of real-world productivity benefits that have been demonstrated in production applications. A Charm++-based benchmark suite was submitted to HPC Challenge competition (which is aimed at comparing productivity); in 2011 we were co-winners, in 2012, we were finalists.

### Salient Features

**Object-based**

Parallel programs in Charm++ are implemented in an object-based paradigm. Computations are expressed in terms of work and data units that are natural to the algorithm being implemented and not in terms of physical cores or processes executing in a parallel context. This immediately has productivity benefits as application programmers can now think in terms that are native to their domain.

The work and data units in a program are C++ objects, and hence, the program design can exploit all the benefits of object-oriented software architecture. Classes that participate in the expression of parallel control flow (chares) inherit from base classes supplied by the programming framework.

Chares are typically organized into indexed collections, known as chare arrays. Chares in an array share a type, and hence present a common interface of entry methods.

**Message-Driven**

Messaging in Charm++ is sender-driven and asynchronous. Parallel control flow in Charm++ is expressed in the form of method invocations on remote objects. Each chare simply uses its entry methods to describe its reactions when dependencies (remote events or receipt of remote data) are fulfilled. Once this happens, it can perform appropriate computations and also trigger other events whose dependencies are now fulfilled.

The parallel program then becomes a collection of objects that trigger each other via remote (or local) invocations by sending messages.

**Runtime-assisted**

Once an application has been expressed as a set of overridable composed message-driven objects, these can be mapped onto the available compute resources and their executions managed by a runtime system. The programming model permits an execution model where the run-time system can:

- maintain a queue of incoming messages, and deliver them to entry methods on local chares.
- overlap data movement required by a chare with entry method executions for other chares.
- observe computation / communication patterns, and move chares to balance load and optimize communication.
- allow run-time composition (interleaving) of work from different parallel modules.

**Runnable**

In our implementation, blocks are first-class entities that in a collection that expands and contracts as the mesh is refined or coarsened, without requiring synchronization. Refinement decisions are local to each block and propagated as far as algorithmically required. We use scalable termination detection built into our runtime to globally determine when all refinement decisions have been finalized, reducing many overheads.

**Sparse Triangular Solver**

The matrix is divided into blocks of columns then analyzed to find its independent rows for computation. Dense regions below the diagonal section are divided into new blocks. Each diagonal block starts the computation with its independent parts and waits for required messages from the left. The column blocks are mapped round-robin, which is essential for this solver and managing the blocks manually is burdensome for the programmer.

**Random Access**

Our implementation provides dynamic, memory-constrained lookahead so that panel factorizations are overlapped as much as memory usage limits will allow. The placement of matrix blocks on processors is independent of the main factorization routines; it is encapsulated in a sequential function. We use asynchronous collectives for pivot identification reductions so they can be overlapped with updating the rest of the sub-panel.

**Molecular Dynamics**

LeanMD simulates the behavior of atoms based on the Lennard-Jones potential, which mimics the short-range non-bonded force calculation in NAMD and resembles miniMD in the Mantevo suite. Charm++ fully automated load balancing enables exceptional strong scaling. To enable automatic load balancing decisions, the user simply specifies a flag, *MetaLB* and the run-time system automatically identifies a load balancing period.

**AMR**

In our implementation, blocks are first-class entities that in a collection that expands and contracts as the mesh is refined or coarsened, without requiring synchronization. Refinement decisions are local to each block and propagated as far as algorithmically required. We use scalable termination detection built into our runtime to globally determine when all refinement decisions have been finalized, reducing many overheads.

**Dense LU**

The global table is partitioned across the nodes in the run. Each element of the group allocates its part of the global table, generates random update keys, and sends the updates to the appropriate destination. The Charm++ Mesh Streamer library automates aggregation and routing, based on network topology information provided by Charm++ Topology Manager.

**1D FFT**

The global table is partitioned across the nodes in the run. Each element of the group allocates its part of the global table, generates random update keys, and sends the updates to the appropriate destination. The Charm++ Mesh Streamer library automates aggregation and routing, based on network topology information provided by Charm++ Topology Manager.

### Performance Highlights

#### 1D FFT

<table>
<thead>
<tr>
<th>Code</th>
<th>C++ CI Benchmark Subtotal</th>
<th>Driver Total</th>
<th>Machine</th>
<th>Max Cores</th>
<th>Performance Highlight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1D FFT</td>
<td>54 29 83</td>
<td>102 185</td>
<td>BG/P BG/Q</td>
<td>64K 16K</td>
<td>2.71 TFlop/s 2.31 TFlop/s</td>
</tr>
<tr>
<td>Random Access</td>
<td>76 15 91</td>
<td>47 138</td>
<td>BG/P BG/Q</td>
<td>128K 16K</td>
<td>43.10 GUPS 15.00 GUPS</td>
</tr>
<tr>
<td>Dense LU</td>
<td>1001 316 1317</td>
<td>453 1770</td>
<td>XT5</td>
<td>8K</td>
<td>55.1 TFlop/s (65.7% peak)</td>
</tr>
</tbody>
</table>

#### Additional Benchmarks

| Molecular Dynamics | 571 122 693 | n/a 693 | BG/P BG/Q | 128K 16K | 24 ms/step (2.8M atoms) 44 ms/step (2.8M atoms) |
| AMR | 1126 118 1244 | n/a 1244 | BG/Q | 32k | 22 steps/sec, 2d mesh, max 15 levels refinement |
| Triangular Solver | 642 50 692 | 56 748 | BG/P | 512 | 48x speedup on 64 cores with helm2x403 matrix |

---

**Charm++ Migratable Objects + Active Messages + Adaptive Runtime = Productivity + Performance**

Laxmikant V. Kale, Anshu Arya, Nikhil Jain, Akhil Langer, Jonathan Liflander, Harshitha Menon, Xiang Ni, Yanhua Sun, Ehsan Totoni, Ramprasad Venkataraman, Lukasz Wesolowski