Charm++
Migratable Objects + Asynchronous Methods + Adaptive Runtime
= Performance + Productivity

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

Parallel Programming Laboratory
Department of Computer Science
University of Illinois at Urbana-Champaign

*{kale, ramv}@illinois.edu
Benchmarks

**Required**
- 1D FFT
- Random Access
- Dense LU Factorization

**Optional**
- Molecular Dynamics
- Adaptive Mesh Refinement
- Sparse Triangular Solver
### Metrics: Performance and Productivity

#### Our Implementations in Charm++

<table>
<thead>
<tr>
<th></th>
<th><strong>Productivity</strong></th>
<th><strong>Performance</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Code</strong></td>
<td>C++ CI</td>
<td><strong>Benchmark Subtotal</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Driver</td>
</tr>
<tr>
<td>1D FFT</td>
<td>54 29</td>
<td>83</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Random Access</td>
<td>76 15</td>
<td>91</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dense LU</td>
<td>1001 316</td>
<td>1317</td>
</tr>
<tr>
<td>Molecular Dynamics</td>
<td>571 122</td>
<td>693</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Triangular Solver</td>
<td>642 50</td>
<td>692</td>
</tr>
<tr>
<td>AMR</td>
<td>1126 118</td>
<td>1244</td>
</tr>
</tbody>
</table>

- **C++**: Regular C++ code
- **CI**: Parallel interface descriptions and control flow DAG
Capabilities
Demonstrated Productivity Benefits

- Automatic load balancing
- Automatic checkpoints
- Tolerating process failures
- Asynchronous, non-blocking collective communication
- Interoperating with MPI

For more info
http://charm.cs.illinois.edu/
Capabilities: Automated Dynamic Load Balancing

- Measurement based fine-grained load balancing
  - Principle of persistence - recent past indicates near future.
  - Charm++ provides a suite of load-balancers.
Capabilities: Automated Dynamic Load Balancing

- Measurement based fine-grained load balancing
  - Principle of persistence - recent past indicates near future.
  - Charm++ provides a suite of load-balancers.

- How to use?
  - Periodic calls in application - AtSync().
  - Command line argument - +balancer Strategy.

Kale et al. (PPL, Illinois)
Capabilities: Automated Dynamic Load Balancing

- Measurement based fine-grained load balancing
  - Principle of persistence - recent past indicates near future.
  - Charm++ provides a suite of load-balancers.

- How to use?
  - Periodic calls in application - AtSync().
  - Command line argument - +balancer Strategy.

- MetaBalancer - When and how to load balance?
  - Monitors the application continuously and predicts behavior.
  - Decides when to invoke which load balancer.
  - Command line argument - +MetaLB
Capabilities: Checkpointing Application State

- Checkpointing to disk for split execution
  CkStartCheckpoint(callback)
  - Designed for applications need to run for a long period, but cannot get all the allocation needed at one time.
- Restart applications from checkpoint on any number of processors
Capabilities: Tolerating Process Failures

- Double in-memory checkpointing for online recovery
  $\text{CkStartMemCheckpoint}(\text{callback})$
  - To tolerate the more and more frequent failures in HPC system.
- Injecting failure and automatically detection of failures
  $\text{CkDieNow}()$
Capabilities: Interoperability

Invoke Charm++ from MPI

- Callable like other external MPI libraries
- Use MPI communicators to enable the following modes:
  (a) Time Sharing
  (b) Space Sharing
  (c) Combined

Kale et al. (PPL, Illinois)
Capabilities: Interoperability
Trivial Changes to Existing Codes

- Initialize and destroy Charm++ instances
- Use interface functions to transfer control

```c
//MPI_Init and other basic initialization
{ optional pure MPI code blocks }

//create a communicator for initializing Charm++
MPI_Comm_split(MPI_COMM_WORLD, peid%2, peid, &newComm);
CharmLibInit(newComm, argc, argv);

{ optional pure MPI code blocks }

//Charm++ library invocation
if(myrank%2)
   fft1d(inputData,outputData,data_size);

//more pure MPI code blocks
//more Charm++ library calls

CharmLibExit();
//MPI cleanup and MPI_Finalize
```
Capabilities: Asynchronous, Non-blocking Collective Communication

- Overlap collective communication with other work
- Topological Routing and Aggregation Module (TRAM)
  - Transforms point-to-point communication into collectives
  - Minimal topology-aware software routing
  - Aggregation of fine-grained communication
  - Recombining at intermediate destinations
- Intuitive expression of collectives through overloading constructs for point-to-point sends (e.g. broadcast)
doFFT()

for(phase = 0; phase < 3; ++phase) {
    atomic {
        sendTranspose();
    }
    for(count = 0; count < P; ++count)
        when recvTranspose[phase] (fftMsg *msg) atomic {
            applyTranspose(msg);
        }
    if (phase < 2) atomic {
        fftw_execute(plan);
        if(phase == 0)
            twiddle();
    }
}
FFT: Performance
IBM Blue Gene/P (Intrepid), 25% memory, ESSL /w fftw wrappers

Charm++
Kale et al. (PPL, Illinois)
FFT: Performance
IBM Blue Gene/P (Intrepid), 25% memory, ESSL /w fftw wrappers

Charm++ all-to-all using TRAM
Asynchronous, Non-blocking, Topology-aware, Combining, Streaming

Kale et al. (PPL, Illinois)
Random Access

Productivity
- Use point to point sends and let Charm++ optimize communication
- Automatically detect and adapt to network topology of partition

Performance
- Automatic communication optimization using TRAM
  - Aggregation of fine-grained communication
  - Minimal topology-aware software routing
  - Recombining at intermediate destinations
Random Access: Performance
IBM Blue Gene/P (Intrepid), BlueGene/Q (Vesta)

Kale et al. (PPL, Illinois)
LU: Capabilities

- Composable library
  - Modular program structure
  - Seamless execution structure (interleaved modules)
LU: Capabilities

- Composable library
  - Modular program structure
  - Seamless execution structure (interleaved modules)

- Block-centric
  - Algorithm from a block’s perspective
  - Agnostic of processor-level considerations
LU: Capabilities

- Composable library
  - Modular program structure
  - Seamless execution structure (interleaved modules)

- Block-centric
  - Algorithm from a block’s perspective
  - Agnostic of processor-level considerations

- Separation of concerns
  - Domain specialist codes algorithm
  - Systems specialist codes tuning, resource mgmt etc

<table>
<thead>
<tr>
<th></th>
<th>Lines of Code</th>
<th>Module-specific Commits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CI</td>
<td>C++</td>
</tr>
<tr>
<td>Factorization</td>
<td>517</td>
<td>419</td>
</tr>
<tr>
<td>Mem. Aware Sched.</td>
<td>9</td>
<td>492</td>
</tr>
<tr>
<td>Mapping</td>
<td>10</td>
<td>72</td>
</tr>
</tbody>
</table>
LU: Capabilities

- Flexible data placement
  - Experiment with data layout
- Memory-constrained adaptive lookahead
LU: Performance

Weak Scaling: (N such that matrix fills 75% memory)
LU: Performance

... and strong scaling too! \((N=96,000)\)
Optional Benchmarks

Why MD, AMR and Sparse Triangular Solver

- Relevant scientific computing kernels
- Challenge the parallelization paradigm
  - Load imbalances
  - Dynamic communication structure
- Express non-trivial parallel control flow
LeanMD
SLOC: 693

1. Mimics short-range force calculation in NAMD
2. Resembles *miniMD* of Mantevo project (SLOC ≈ 3000)
3. Advanced features:
   - **Metabalancer**: automated dynamic load balancing
   - **Fault tolerance**: in-memory checkpointing based restart
   - **Split Execution**: checkpoint on $x$ cores, restart on $y$ cores

![Diagram](image-url)
Code for FT and LB

```c
if (stepCount % ldbPeriod == 0) {
    serial { AtSync(); }
    when ResumeFromSync() { }
}

if (stepCount % checkptFreq == 0) {
    serial {
        //coordinate to start checkpointing
        contribute(CkCallback(CkReductionTarget(Cell,startCheckpoint),thisProxy(0,0,0)));}
    if (thisIndex.x == 0 && thisIndex.y == 0 && thisIndex.z == 0) {
        when startCheckpoint() {
            CkCallback cb(CkReductionTarget(Cell,recvCheckPointDone), thisProxy);
            if (checkptStrategy == 0) CkStartCheckpoint(logs.cstr(), cb);
            else CkStartMemCheckpoint(cb);
        }
    }
    when recvCheckPointDone() { }
}

//kill one of processes to demonstrate fault tolerance
if (stepCount == 30 && thisIndex.x == 1 && thisIndex.y == 1 && thisIndex.z == 0) {
    serial {
        if (CkHasCheckpoints()) {
            CkDieNow();
        }
    }
}
```
if (stepCount % ldbPeriod == 0) {
    serial { AtSync(); }
    when ResumeFromSync() { }
}

if (stepCount % checkptFreq == 0) {
    serial {
        //coordinate to start checkpointing
        contribute(CkCallback(CkReductionTarget(Cell,startCheckpoint),thisProxy(0,0,0)));
    }
    if (thisIndex.x == 0 && thisIndex.y == 0 && thisIndex.z == 0) {
        when startCheckpoint() serial {
            CkCallback cb(CkReductionTarget(Cell,recvCheckPointDone), thisProxy);
            if (checkptStrategy == 0) CkStartCheckpoint(logs.c_str(), cb);
            else CkStartMemCheckpoint(cb);
        }
    }
    when recvCheckPointDone() { }
}
Code for FT and LB

if (stepCount % ldbPeriod == 0) {
    serial { AtSync(); }
    when ResumeFromSync() {
}
}

if (stepCount % checkptFreq == 0) {
    serial {
        //coordinate to start checkpointing
        contribute(CkCallback(CkReductionTarget(Cell,startCheckpoint),thisProxy(0,0,0)));
    }
    if (thisIndex.x == 0 && thisIndex.y == 0 && thisIndex.z == 0) {
        when startCheckpoint() serial {
            CkCallback cb(CkReductionTarget(Cell,recvCheckPointDone), thisProxy);
            if (checkptStrategy == 0) CkStartCheckpoint(logs.c_str(), cb);
            else CkStartMemCheckpoint(cb);
        }
        when recvCheckPointDone() {
    }
}

//kill one of processes to demonstrate fault tolerance
if (stepCount == 30 && thisIndex.x == 1 && thisIndex.y == 1 && thisIndex.z == 0) serial {
    if (CkHasCheckpoints()) {
        CkDieNow();
    }
}
MD: Performance

1 million atoms. IBM Blue Gene/P (Intrepid)

Performance on Intrepid (2.8 million atoms)

- No LB
- Hybrid LB

Time per step (ms)

Number of cores

Kale et al. (PPL, Illinois)

Charm++
LeanMD Checkpoint Time on BlueGene/Q

- 2.8 million
- 1.6 million
LeanMD Restart Time on BlueGene/Q

- 2.8 million
- 1.6 million
Meta-Balancer vs Periodic Load Balancing

- Frequent load balancing increases total execution time
- Infrequent load balancing leads to load imbalance and results in no gains
- Meta-Balancer adaptively performs load balancing to obtain best total execution time

### Elapsed time vs LB Period (BlueGene/P)

<table>
<thead>
<tr>
<th>Cores</th>
<th>No LB (s)</th>
<th>Periodic LB (s)</th>
<th>Meta-Balancer (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8k</td>
<td>666</td>
<td>504</td>
<td>413</td>
</tr>
<tr>
<td>16k</td>
<td>336</td>
<td>260</td>
<td>277</td>
</tr>
<tr>
<td>32k</td>
<td>171</td>
<td>131</td>
<td>131</td>
</tr>
<tr>
<td>64k</td>
<td>122</td>
<td>104</td>
<td>100</td>
</tr>
<tr>
<td>128k</td>
<td>73</td>
<td>54</td>
<td>52</td>
</tr>
</tbody>
</table>
Sparse Triangular Solver- Matrix Decomposition

\[
\begin{bmatrix}
l_{11} & l_{21} & l_{22} \\
l_{33} & l_{43} & l_{44} \\
l_{54} & l_{55} & l_{66} \\
l_{76} & l_{77} & l_{88} \\
l_{81} & & & l_{99}
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
x_3 \\
x_4 \\
x_5 \\
x_6 \\
x_7 \\
x_8 \\
x_9
\end{bmatrix} =
\begin{bmatrix}
b_1 \\
b_2 \\
b_3 \\
b_4 \\
b_5 \\
b_6 \\
b_7 \\
b_8 \\
b_9
\end{bmatrix}
\]

Column Decomposition

Multicast

Dense Parts Decomposition
Sparse Triangular Solver - Parallel Algorithm

if (onDiagonalChare) {
    serial { thisProxy[thisIndex].indepCompute(...) }
    overlap {
        while (!finished) {
            when recvData(int len, double data[len], int rows[len])
                serial { if(len>0) diagReceiveData(len, data, rows); }
            when indepCompute(int a) serial { myIndepCompute(); }
        }
    }
} else {
    when getXvals(xValMsg* msg) serial { nondiag_compute(); }
    while (!finished) {
        when recvData(int len, double data[len], int rows[len])
            serial { nondiagReceiveData(len, data, rows); }
    }
}
Sparse Triangular Solver - Performance vs. SuperLU_DIST

Kale et al. (PPL, Illinois)
More complicated (higher performance) algorithm in 692 total SLOCs
  ▶ vs. 897 SLOCs of SuperLU_DIST triangular solver

Overdecomposition (with Round-Robin mapping) is essential
  ▶ Communication computation overlap, load balance

Creation of parallel units dynamically
  ▶ Distributing dense regions

Message-driven nature and priorities
  ▶ No need for something like MPI_Iprobe
Adaptive Mesh Refinement

Sample simulation

Propagation of refinement decision messages

Finite state machine for each block’s decision update

Kale et al. (PPL, Illinois)
Adaptive Mesh Refinement

Sample simulation

Propagation of refinement decision messages

Finite state machine for each block’s decision update
Adaptive Mesh Refinement

Charm++ Implementation

- Blocks as virtual processors instead of each process containing many blocks
  - simplifies implementation
- Blocks addressed with bit vector indices
  - CharmRTS handles physical locations
- Dynamic distributed load balancing

Algorithmic Improvements

- $O(\frac{\#blocks}{P})$ vs $O(\#blocks)$ memory per process
- 2 system quiesence states vs $\#level$ reductions for mesh restructuring
- $O(1)$ vs $O(\log P)$ time neighbor lookup

---

Adaptive Mesh Refinement

Timesteps per second strong scaling on IBM BG/Q with a max depth of 15.

The non-overlapped delay of remeshing in milliseconds on IBM BG/Q. The candlestick graphs the minimum and maximum values, the 5th and 95th percentile, and the median.
Temperature-aware load balancing  Tue @ 2:00 pm
NAMD at 200K+ cores  Thu @ 11:00 am

For more info
http://charm.cs.illinois.edu/
Charm++ Programming Model

Object-based
Express logic via indexed collections of interacting objects (both data \textit{and} tasks)

Over-decomposed
Expose more parallelism than available processors
Charm++
Programming Model

Runtime-Assisted scheduling, observation-based adaptivity, load balancing, composition, etc.

Message-Driven Trigger computation by invoking remote entry methods

Non-blocking, Asynchronous Implicitly overlapped data transfer
Charm++

Program Structure

- Regular C++ code
  - No special compilers
Charm++

Program Structure

- Regular C++ code
  - No special compilers
- Small parallel interface description file
  - Can contain control flow DAG
  - Parsed to generate more C++ code
Charm++

Program Structure

- Regular C++ code
  - No special compilers
- Small parallel interface description file
  - Can contain control flow DAG
  - Parsed to generate more C++ code
- Inherit from framework classes to
  - Communicate with remote objects
  - Serialize objects for transmission
- Exploit modern C++ program design techniques (OO, generics etc)