Kaapi / Charm++ preliminary comparison

Xavier Besseron\(^1\)   Thierry Gautier\(^1\)

Gengbin Zheng\(^2\)   Laxmikant Kalé\(^2\)

\(^1\) MOAIS Project – LIG and INRIA
\(^2\) University of Illinois at Urbana-Champaign

22-24 June 2010, Bordeaux
Outline

1. Qualitative comparison
2. NUMA execution
3. Distributed execution
4. Fault-tolerance
5. Preliminary Conclusion
Objective: efficiency

- 90% of efficiency on 10000 cores 1000 cores not used
- strong scalability (fixed problem size)

Comparison of middlewares for exascale architecture

- Overhead of parallel programming API
  - overhead with respect to the execution time of sequential program
- Scheduling
  - quality of scheduling algorithm
- Fault tolerance
  - cost of fault tolerance protocol (coordinated checkpoint/rollback)

Cost of developing parallel application

- Parallel programming complexity
- Add fault tolerance to application
- Other criteria (smallest timestep, memory) for future comparisons
Overview

Charm++
- Library/Compiler (C, C++, Fortran, ...)
- Low level interface: Object based (Chares), message driven
- High level interface: Structured dagger, Charisma, ...
- Production tool

Kaapi
- C++ library, template based, Fortran interface
- High level interface: Data flow graph (Athapascan)
  - At runtime based on active messages + threads + scheduling algorithms ...
- Research software
Which comparison?

Key points

- **Programming model**
  - Preliminary work: skew due to Charm++/Low level - Kaapi/High level
  - Future work: Charisma versus Kaapi

- **Performance of execution**
  - Overhead with respect to execution time of a sequential program
  - Multi-core
  - Cluster and grid

- **Fault Tolerance capability**
  - Checkpoint / Restart protocols ⇒ experimental costs

Methodology

- "Same benchmark" using both frameworks + sequential version
- Experimentation and Analysis
Programming models comparison

Charm++ (Low level interface: object based, message driven)

- Simple: Model and syntax close to the classic object model
  ++ Good documentation, many examples, stable
  ⇒ Easy to understand, easy to use
  ++ Several "production quality" applications
  ⇒ large scale experiments, including the FT protocol
  ++ Automatic dynamic load balancing
  ⇒ Required for irregular computation

+/- Non deterministic execution
  ⇒ Results can depend on the reception order
  - Explicit communications, required some synchronizations
    ⇒ Source of bugs (race conditions)
  - Non-transparent fault-tolerance
    ⇒ Requires source code modification
Programming models comparison

Kaapi (High level interface: macro data flow)

- Complex/unusual programming model: macro data flow

++ Sequential semantic / Implicit communications and synchronizations
  ⇒ Once the (sequential) program is written and run on 1 (or 2) processors, very few bugs (in the application source code)

++ Automatic dynamic load balancing

++ Transparent fault-tolerance
  ⇒ No modification in the source code

+/- Deterministic execution
  ⇒ Always the same results

- Simple documentation, research software
  ⇒ Harder to understand and to start with ?

- Not a language: C++ Template based
  ⇒ Error messages at compilation are very ugly
Scheduling of Charm++ program

- A Charm++ program $\Rightarrow$ a set of Chares

- Scheduler of message queue (priority, ...)

- Dynamic Load balancing algorithm
  - GreedyLB, MetisLB, RecBisectBfLB, NeighborLB, ...
  - Based on automatic instrumentation of the application by the runtime
Scheduling of Kaapi program

• A Kaapi program ⇒ a data flow graph
The benchmark: Poisson3D

Problem

- Solve Poisson equation on a 3D domain: \( A \times X = B \)
- Parallelized by domain decomposition
- Jacobi iterative method

\[
x_i^{k+1} = \frac{1}{a_{ii}} (b_i - \sum_{j=1, j \neq i}^{N} a_{ij} x_j^k) \text{ for } i \text{ from } 1 \text{ to } N
\]

Benchmark data

- \( X, X' \): computation domain & temporary computation domain
- \( A \): Jacobi stencil (sparse matrix)
- \( B \): right-hand side
- \( S \): solution (here the solution is known to check the result)
### Source code

- Common code for "sequential" kernel: 671 lines
- Same data structure and allocation

<table>
<thead>
<tr>
<th>Lines of code (raw values)</th>
<th>Charm++</th>
<th>Kaapi</th>
</tr>
</thead>
<tbody>
<tr>
<td>(De)serialization</td>
<td>105</td>
<td>125</td>
</tr>
<tr>
<td>Task encapsulation</td>
<td>-</td>
<td>160</td>
</tr>
<tr>
<td>Interface file (.ci)</td>
<td>50</td>
<td>-</td>
</tr>
<tr>
<td>Computation Kernel</td>
<td>140</td>
<td>50</td>
</tr>
<tr>
<td>FT/LB</td>
<td>50</td>
<td>-</td>
</tr>
<tr>
<td>Total</td>
<td>483</td>
<td>481</td>
</tr>
</tbody>
</table>

- Sequential program without decomposition: 50 lines
- Sequential program with decomposition: 112 lines
Experiments

- NUMA machine "idkoiff"
  - 8 Opteron 875 dual core, 2.2Ghz, 30GBytes. 8 memory banks
  - gcc/g++ 4.5. All programs compiled with -O4.
- Charm++ 6.2.0 (multicore-linux64.tar.gz)
  - +p 1: one process
  - +ppn <n>: number of PEs per process
  - +setcpuaffinity: useCPU affinity
  - +excludecore <coreid>: to exclude some core
- Kaapi 2.4 (http://kaapi.gforge.inria.fr)
  - Same mapping of threads onto core for Charm++/Kaapi
  - using util.thread.cpuset
- XKaapi 0.1beta
  - New C kernel for Kaapi. Currently only for multi-core machine.

Timings

- Domain size: $4 \times 10^6$ doubles ($100\times200\times200$) split in $i \times j \times k$ subdomains
  - 1 run = average time of one iteration over 50 iterations.
  - Get the best average over 10 runs
Execution time: 1 core (among 16 cores)

- Charm++ SMP
- Kaapi
- X-Kaapi

Decomposition
- Charm++ SMP
- Kaapi
- X-Kaapi
- Seq with decomposition
- Sequential

One-iteration time (s)
Execution time: 8 cores (among 16 cores)

- Few parallelism ⇒ slowdown due to activity of the threads
Execution time: 16 cores (among 16 cores)

- Same slowdown
Distributed execution: experiments

Execution platform

- Two experiments on Grid’5000 clusters:
  - Cluster Griffon at Nancy: 64 nodes with 8 cores, ie 512 cores
  - Cluster GDX at Orsay: 283 nodes with 2 cores, ie 566 cores
- Gigabit ethernet network

Same Poisson 3D benchmark

- Weak scalability: domain size = \(4 \times 10^6\) doubles per core
- Two cases for Kaapi and Charm++:
  - 1 process per node (named SMP)
  - 1 process per core
- Charm++ 6.1.3 net-linux-x86_64 [-smp]
- Kaapi (git version 03/01/2010, branch besseron/master)
Distributed execution: cluster Griffon

Weak scalability: $4 \times 10^6$ doubles per core, 4 subdomains per core
Up to 512 cores (64 nodes of cluster Griffon of Nancy)
Distributed execution: cluster GDX

Weak scalability: $4 \times 10^6$ doubles per core, 4 subdomains per core
Up to 566 cores (283 nodes of cluster GDX of Orsay)
Checkpoint protocols in Charm++ and Kaapi

Charm++

Double blocking coordinated checkpoint on disk
- double disk $\equiv$ 2 copies: local and on a remote buddy processor
- $\Rightarrow$ tolerate only 1 failure in worst case
- global consistency requires an explicit barrier in the code

Kaapi

Blocking coordinated checkpoint on disk
- 1 copy per process on a Kaapi checkpoint server
- $\Rightarrow$ assume that checkpoint servers are stable,
- global consistency is ensured by the checkpoint protocol
Experimental conditions

3D Poisson Benchmark

- Cluster Griffon at Nancy: 80 nodes with 8 cores, i.e., 640 cores
- Domain size = $10^6$ doubles per core
- Charm++ net-linux-x86_64-syncft (git version on 04/12/2010) with three different load-balancers: NullLB, GreedyLB, OrbLB
- Kaapi with FT support (git version on 04/12/2010, branch besseron/master)

Remote buddy processors / Kaapi checkpoint servers

- Located on the next node (according to the node list)

<table>
<thead>
<tr>
<th>Charm++ PEs</th>
<th>Remote buddy processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kaapi processors</td>
<td></td>
</tr>
<tr>
<td>griffon-01</td>
<td>→</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>griffon-79</td>
<td>→</td>
</tr>
<tr>
<td>griffon-80</td>
<td>→</td>
</tr>
</tbody>
</table>
**Checkpoint time**

80 nodes with 8 cores, ie 640 cores
Domain size = $10^6$ doubles per core

### Checkpoint to disk
- **Total = 27.1 s**

### Checkpoint to memory
- **Total = 16.0 s**
- **Total = 5.4 s**
- **Total = 3.9 s**

**Send data**
- **Kaapi**: 5.0 s
- **Charm++ NullLB**: 5.0 s
- **Charm++ GreedyLB**: 3.9 s

**Wait for acknowledgment**
- **Kaapi**: 202 MB/node
- **Charm++ NullLB**: 455 MB/node
- **Charm++ GreedyLB**: 455 MB/node

---

X. Besseron, T. Gautier, G. Zheng, L. Kalé

Kaapi / Charm++ preliminary comparison
Restart protocols in Charm++ and Kaapi

Scenario

- Domain size = $10^6$ doubles per core
- 1 failure $\Rightarrow$ restart on 79 nodes = 632 cores $\Rightarrow$ load balancing

Restart protocols

- **Charm++**: Global restart + load balancing
  - Failed processes restart from their remote checkpoint
  - Alive processes restart from their local copy
- **Kaapi**: Global restart + rescheduling
  - All processes restart from their remote checkpoint
  - Rescheduling: static partitionning + data redistribution
Restart and load balancing time

1 failure ⇒ restart on 79 nodes with 8 cores, i.e. 632 cores

<table>
<thead>
<tr>
<th>Time (s)</th>
<th>Global restart</th>
<th>Load balancing</th>
<th>Static partitioning</th>
<th>Data redistribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kaapi</td>
<td>3.3 s</td>
<td>1.7 s</td>
<td>2.1 s</td>
<td></td>
</tr>
<tr>
<td>Charm++ OrLB</td>
<td>4.6 s</td>
<td>2.2 s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Charm++ GreedyLB</td>
<td>6.2 s</td>
<td>2.1 s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Charm++ NullLB</td>
<td>1.0 s</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total = 7.0 s
Total = 6.8 s
Total = 8.4 s
Total = 1.0 s
Conclusion of this preliminary work (1/2)

Two different levels

- Charm++: low level API
  - Charisma will be the next candidate
- Kaapi: high level API
  - Also recursive applications + work-stealing
  - Heterogeneous architecture (CPUs + GPUs)

Execution performance

- NUMA: new XKaapi implementation performs well
  - comparison with NUMA features [Pousa, Méhaut (INRIA), Gioachin, Mei, Kalé (UIUC)] for Charm++
- Distributed execution: SMP Charm++ execution is better
Fault tolerance

- Checkpoint is the most costly part of the protocols
  - Charm++ tolerates only 1 failure in worst case without stable servers
  - Kaapi tolerates the failure of one cluster but required "stable servers"

⇒ Stable memory issues
  - diskless checkpointing: k-duplication, erasure encoding, locality

- Restart
  - Global restart is simple, but lost work can be large
  ⇒ Partial restart in Kaapi (based on the data flow graph)