The UHPC/Runnemedede Execution Model

S. ZUCKERMAN    J.Suetterlein    G.Gao

University of Delaware
Computer Architecture and Parallel Systems Laboratory (CAPSL)

October 25, 2011
Outline

Introduction
  Why UHPC?
  The UHPC Teams
  What Is an Execution Model?

Our Concurrency Model: The Codelet Model
  The Codelet Execution Model
  Codelet Graphs
  Well-Behaved Codelets

Memory Model
  Hardware-Oriented Consistency Models
  Software-Oriented Consistency Models
  Our Basis for the Codelet Memory Model: Location Consistency

Self-Awareness

Conclusion
Introduction

Why UHPC?
The UHPC Teams
What Is an Execution Model?

Our Concurrency Model: The Codelet Model
The Codelet Execution Model
Codelet Graphs
Well-Behaved Codelets

Memory Model
Hardware-Oriented Consistency Models
Software-Oriented Consistency Models
Our Basis for the Codelet Memory Model: Location Consistency

Self-Awareness

Conclusion
What You Already Know

- Power is the #1 issue for HPC nowadays.
- Processors will increase their core count dramatically in the coming years.
  → ... In fact, they already are doing so.
- The increase in the number of cores does not feature an increase in ease of use/programmability.
- Did I mention power issues?
What You Already Know

- Power is the #1 issue for HPC nowadays.
- Processors will increase their core count dramatically in the coming years
  → ... In fact, they already are doing so
- The increase in the number of cores does not feature an increase in ease of use/programmability.
- Did I mention power issues?
What You Already Know

➤ Power is the #1 issue for HPC nowadays.
➤ Processors will increase their core count dramatically in the coming years
   → ... In fact, they already are doing so
➤ The increase in the number of cores does not feature an increase in ease of use/programmability.
➤ Did I mention power issues?
What DARPA wants: Ubiquitous High-Performance Computing

A “generic” computer, which can fit both in a tank or in supercomputing center (with many other manycore computers linked together). With a few additional constraints, as it must:

- fit into a cabinet
- provide $\approx 1$ PFLOPS
- be power-efficient: $\approx 57 \text{KW}$
- be fault-tolerant
- self-aware (more on that later)
- provide security features
- be *programmable*

1000 cabinets $\rightarrow$ 1 exaflop supercomputer
What DARPA wants: Ubiquitous High-Performance Computing

A “generic” computer, which can fit both in a tank or in supercomputing center (with many other manycore computers linked together). With a few additional constraints, as it must:

▶ fit into a cabinet
▶ provide $\approx 1$ PFLOPS
▶ be power-efficient: $\approx 57$ KW
▶ be fault-tolerant
▶ self-aware (more on that later)
▶ provide security features
▶ be programmable

1000 cabinets $\rightarrow$ 1 exaflop supercomputer
The Four Selected Teams

DARPA has agreed to distribute 100 million dollars among these four teams:

- Intel/Runnemede team
- Nvidia team
- MIT/Angstrom team
- Sandia/X-Caliber team
How the Teams Will Be Evaluated

The teams will have to show how versatile their proposed system is, by showing how they perform on five challenge problems:

- Streaming sensors (SAR)
- Graphs traversal, connected component finding, etc.
- Decision problem: chess (minimax, alpha-beta)
- Molecular Dynamics
- Hydrodynamics (Lagrangian relaxation)
The UHPC/Runnemede Team

Intel has decided to match DARPA’s funding (25M$). The various partners of this team include:

- Intel for the hardware (well, duh!)
- UIUC for architecture research (led by J. Torrelas)
- ETI for the implementation of a runtime system (SWARM, led by R. Kahn)
- Intel for CnC (Concurrent Collections, developed by Kath Knobe)
- UIUC for the implementation of HTA/Chapel (led by D. Padua)
- Reservoir Labs to adapt their R-Stream compiler to the future UHPC runtime system (R. Lethin, N. Vasilache)

... and the University of Delaware is in charge of designing the parallel execution model (PXM)
The UHPC/Runnemedede Team

Intel has decided to match DARPA’s funding (25M$). The various partners of this team include:

- Intel for the hardware (well, duh!)
- UIUC for architecture research (led by J.Torrelas)
- ETI for the implementation of a runtime system (SWARM, led by R.Kahn)
- Intel for CnC (Concurrent Collections, developed by Kath Knobe)
- UIUC for the implementation of HTA/Chapel (led by D.Padua)
- Reservoir Labs to adapt their R-Stream compiler to the future UHPC runtime system (R.Lethin, N.Vasilache)

... and the University of Delaware is in charge of designing the parallel execution model (PXM)
The UHPC/Runnemede Team

Intel has decided to match DARPA’s funding (25M$). The various partners of this team include:

- Intel for the hardware (well, duh!)
- UIUC for architecture research (led by J.Torrelas)
- ETI for the implementation of a runtime system (SWARM, led by R.Kahn)
- Intel for CnC (Concurrent Collections, developed by Kath Knobe)
- UIUC for the implementation of HTA/Chapel (led by D.Padua)
- Reservoir Labs to adapt their R-Stream compiler to the future UHPC runtime system (R.Lethin, N.Vasilache)
- ...and the University of Delaware is in charge of designing the parallel execution model (PXM)
What Is an Execution Model?

A Definition
A program execution model defines the interactions between the components of the whole computer system. It is a “vertical” concept in that it traverses the whole software system stack down to the HW (high-level languages and/or compilers, runtime systems, OSes, hardware, and anything in-between, such as libraries etc.). Traditionally, there are three components to a program execution model:

▶ A threading/concurrency model
▶ A memory model
▶ A synchronization model
The Big Picture
Introduction
Why UHPC?
The UHPC Teams
What Is an Execution Model?

**Our Concurrency Model: The Codelet Model**

The Codelet Execution Model
Codelet Graphs
Well-Behaved Codelets

Memory Model
Hardware-Oriented Consistency Models
Software-Oriented Consistency Models
Our Basis for the Codelet Memory Model: Location Consistency

Self-Awareness

Conclusion
UD’s Proposal: the Codelet Execution Model

- Fine-grain parallelism
- Scalable
- Expose maximal parallelism
- Limits non-determinism (determinate-by-default)
- Handles dynamic events (power, resiliency, resource constraints in general)
The Codelet Abstract Machine
The Concept of Codelets

A codelet is a sequence of machine instructions which act as an atomically-scheduled unit of computation.

Properties

- event driven
  - availability of data and resources
- communicates through inputs and outputs
- non-preemptive
  - may yield but never give up its register window
- requires all data and code to be “local”
A CDG is a directed graph containing:

- **Nodes** — A node represents a codelet
- **Arcs** — An arc represents a data dependency between two codelets
- **Tokens** — a token represents data traveling along a given arc

Codelet graphs are analogous to dataflow graphs [Dennis(1974), Dennis et al.(1974)Dennis, Fosseen, and Linderman].
CDGs: Operational Semantics

- **Codelet Firing Rule**
  - Codelet actors are *enabled* once tokens are on each input arc
  - Codelet actors fire by
    - consuming tokens
    - performing the operations within the codelet
    - producing a token on each of its output arcs

- **States of a Codelet**
  - Dormant: Not all tokens are available
  - Enabled: All *data* tokens are available
  - Ready: All tokens are available
  - Active: The codelet is executing internal operations
Control Structures

Codelets still require glue to permit conditional execution and loops. We provide:

- Conditional split
- Conditional merge
- T-gate and F-gate

(a) Conditional

(b) Loop
Threaded Procedures (TP)

TPs are containers for codelet graphs, with additional meta-data.

Description

- invoked in a control-flow manner
- called by a codelet from another CDG
- feature a frame which contains the context of the CDG
An Example of Computation Using Threaded Procedures
Well-Behaved

**Basic rule**
Upon the presentation of its inputs, an actor consumes all tokens and places a token on all its output arcs.

Well-behaved codelets and codelet graphs ensure *determinate* results.

**How to ensure well-behaved codelet graphs?**

- Build CDGs from the ground up
- Follow the DAG construction rules
- Use well-formed schema rules

It is highly inspired by dataflow schemas [Dennis et al.(1972)Dennis, Fosseen, and Linderman].
Introduction

Why UHPC?
The UHPC Teams
What Is an Execution Model?

Our Concurrency Model: The Codelet Model
The Codelet Execution Model
Codelet Graphs
Well-Behaved Codelets

Memory Model
Hardware-Oriented Consistency Models
Software-Oriented Consistency Models
Our Basis for the Codelet Memory Model: Location Consistency

Self-Awareness

Conclusion
A Motivating Example

Table: Initially, $x = y = 0$. Is it possible to have $r1 = r2 = 0$?

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \leftarrow 1$</td>
<td>$y \leftarrow 1$</td>
</tr>
<tr>
<td>$r1 \leftarrow y$</td>
<td>$r2 \leftarrow x$</td>
</tr>
</tbody>
</table>
A Motivating Example

Thread 0
\[ x \leftarrow 1 \]
\[ r1 \leftarrow y \]

Thread 1
\[ y \leftarrow 1 \]
\[ r2 \leftarrow x \]

Table: Initially, \( x = y = 0 \). Is it possible to have \( r1 = r2 = 0 \)?
A Motivating Example

Table: Initially, $x = y = 0$. Is it possible to have $r1 = r2 = 0$?
What Memory Consistency is All About

Q  What happens when at least two concurrent memory operations arrive at the same memory location \( x \)?

→  What happens when a data-race (i.e. at least one of the two memory operations is a write) occurs at some memory location \( x \)?

▶  Memory Consistency Models try to answer that question.
What Memory Consistency is All About

Q What happens when at least two concurrent memory operations arrive at the same memory location $x$?
   → What happens when a data-race (i.e. at least one of the two memory operations is a write) occurs at some memory location $x$?

▶ Memory Consistency Models try to answer that question.
What Memory Consistency is All About

Q What happens when at least two concurrent memory operations arrive at the same memory location $x$?

→ What happens when a data-race (i.e. at least one of the two memory operations is a write) occurs at some memory location $x$?

▶ Memory Consistency Models try to answer that question.
The Answer of the Message Passing and “Pure” Dataflow Crowd

It can never happen: data is explicitly sent and received. This answer is fine, but . . .

▶ We do not live in a pure message-passing world
▶ Memory is shared on most super-computers, e.g.:
  ▶ Efficient MPI runtime systems make the distinction between intra-node and inter-node communications
  ▶ Inter-node communications work as advertised, but . . .
  ▶ Efficient intra-node communications make the use of shared-memory segments, i.e. shared memory
The Answer of the Message Passing and “Pure” Dataflow Crowd

It can never happen: data is explicitly sent and received. This answer is fine, but . . .

- We do not live in a pure message-passing world
- Memory is shared on most super-computers, e.g.:
  - Efficient MPI runtime systems make the distinction between intra-node and inter-node communications
  - Inter-node communications work as advertised, but . . .
  - *Efficient* intra-node communications make the use of shared-memory segments, i.e. shared memory
Sequential Consistency [Lamport(1978)]

A system is SC if
- All memory operations *appear* to follow some total order
- Memory operations (*appear to*) follow program order
Is it possible to have \( r_1 = r_2 = 0 \)?

**NO.** → There is no total linear order which allows both Thread 0 and Thread 1 to see memory operations happening in the same order such that \( r_1 = r_2 = 0 \).
Back to our Example

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \leftarrow 1$</td>
<td>$y \leftarrow 1$</td>
</tr>
<tr>
<td>$r1 \leftarrow y$</td>
<td>$r2 \leftarrow x$</td>
</tr>
</tbody>
</table>

Table: Initially, $x = y = 0$.

Is it possible to have $r1 = r2 = 0$?

NO. $\rightarrow$ There is no total linear order which allows both Thread 0 and Thread 1 to see memory operations happening in the same order such that $r1 = r2 = 0$
Sequential Consistency and its Popularity

- It behaves pretty much as one would expect in the context of a uniprocessor-multithread execution
  - It is considered very intuitive
- It offers strong guarantees: a modification to memory must be seen by all other threads in a given program
The Drawbacks of Sequential Consistency

It offers strong guarantees: a modification to memory must be seen by all other threads in a given program

→ How complicated is it to implement such a system in hardware?
   → What about caches? Write buffers? etc.
→ How scalable is it?
→ How expensive is it to implement that kind of consistency model?
Weak Consistency
[Dubois et al.(1986) Dubois, Scheurich, and Briggs]
Weak Ordering [Adve and Hill(1990)]

A system is WC/WO if

- all *synchronizing* accesses have performed before any *
  ordinary* access (load or store) is allowed to perform, and
- all *ordinary* accesses (load or store) have performed before any *
  synchronizing* access is allowed to perform
- *synchronizing* accesses are SC
Release Consistency

[Gharachorloo et al. (1990) Gharachorloo, Lenoski, Laudon, Gibbons, Gupta, and Hennessy]

RC refines synchronizing accesses into two types: acquire and release. They are used to label instructions (Gharachorloo speaks about properly labeled programs). A system is RC if:

- all ordinary memory operations have performed before an acquire operation is performed
- release accesses must have performed before any ordinary operation is performed
- Synchronizing accesses (acquire or release) are SC
More Examples (See [Adve et al. (1999) Adve, Pai, and Ranganathan])

Thread 0
Data1 = 64
Data2 = 55
Flag = 1

Thread 1
while (Flag != 1) ;
reg1 = Data1
reg2 = Data2

Table: Ex1: What are the legal values in SC? PC? WC? RC?

Solution
SC, PC reg1 = 64 ; reg2 = 55
WC, RC reg1 = 64 or 0 ; reg2 = 55 or 0
More Examples (See \cite{Adve1999})

\begin{verbatim}
Thread 0
Data1 = 64
Data2 = 55
Flag = 1

Thread 1
while (Flag != 1) ;
reg1 = Data1
reg2 = Data2
\end{verbatim}

Table: Ex1: What are the legal values in SC? PC? WC? RC?

| SC, PC | reg1 = 64 ; reg2 = 55 |
| WC, RC | reg1 = 64 or 0 ; reg2 = 55 or 0 |

Solution
The Java Memory Model

There are two models:

- The first one [Gosling et al.(1996)Gosling, Joy, and Steele], which is broken, and
- the new one
  [Manson et al.(2005)Manson, Pugh, and Adve], which fixes many problems of the first model
  
  coupled with which is also \textit{kinda} broken
  
  [Polyakov and Schuster(2006), Botinčan et al.(2010)Botinčan, Glavan, and Runje] (w.r.t. causality requirements)

However, even with all its problems, it still offers some guarantees:

- accesses to synchronizing variables (declared with the keyword \texttt{volatile}) are SC
- incorrectly synchronized programs should still provide “out-of-thin-air” guarantees: no \textit{self-justifying write} should be allowed, and causality relations should be obeyed (this last part has been proved to be undecidable).
The Java Memory Model

There are two models:

- The first one [Gosling et al. (1996) Gosling, Joy, and Steele], which is broken, and
- the new one [Manson et al. (2005) Manson, Pugh, and Adve], which fixes many problems of the first model
  - ... and which is also kinda broken [Polyakov and Schuster (2006), Botinčan et al. (2010) Botinčan, Glavan, and Runje] (w.r.t. causality requirements)

However, even with all its problems, it still offers some guarantees:

- accesses to synchronizing variables (declared with the keyword `volatile`) are SC
- incorrectly synchronized programs should still provide “out-of-thin-air” guarantees: no `self-justifying write` should be allowed, and causality relations should be obeyed (this last part has been proved to be undecidable).
The Java Memory Model

There are two models:

▶ The first one [Gosling et al.(1996)Gosling, Joy, and Steele], which is broken, and
▶ the new one [Manson et al.(2005)Manson, Pugh, and Adve], which fixes many problems of the first model
  ▶ . . . and which is also kinda broken [Polyakov and Schuster(2006), Botinčan et al.(2010)Botinčan, Glavan, and Runje] (w.r.t. causality requirements)

However, even with all its problems, it still offers some guarantees:

▶ accesses to synchronizing variables (declared with the keyword `volatile`) are SC
▶ incorrectly synchronized programs should still provide “out-of-thin-air” guarantees: no `self-justifying write` should be allowed, and causality relations should be obeyed (this last part has been proved to be undecidable).
The C++ Memory Model

Very easy to understand:

- Synchronizing accesses (through the `atomic` keyword) are SC
- any incorrectly synchronized behavior implies an *undefined behavior*,
  
  - ...which really means by issuing a data-race you can have initiated a new TCP connection in order to order 20 elephants to be delivered by next Saturday
- This is intentional: the C++0x committee wants to flag data-races as bugs.
The C++ Memory Model

Very easy to understand:

- Synchronizing accesses (through the `atomic` keyword) are SC
- any incorrectly synchronized behavior implies an *undefined behavior*,
  - ...which really means by issuing a data-race you can have initiated a new TCP connection in order to order 20 elephants to be delivered by next Saturday
- This is intentional: the C++0x committee wants to flag data-races as bugs.
A Few Comments on the Previous Models

- Memory coherence is assumed by all the “hardware-oriented” models.
- “Software-oriented” models are weaker and do not assume coherence, but they break the causality constraint (i.e. arbitrary values can occur).
- In general, it is assumed that multiple stores to a given location will be serialized in some order (each new store erasing the previous one).

Location Consistency (LC) [Gao and Sarkar(2000)] takes a different path.
A Few Comments on the Previous Models

- Memory coherence is assumed by all the "hardware-oriented" models.
- "Software-oriented" models are weaker and do not assume coherence, but they break the causality constraint (i.e. arbitrary values can occur).
- In general, it is assumed that multiple stores to a given location will be serialized in some order (each new store erasing the previous one).

Location Consistency (LC) [Gao and Sarkar(2000)] takes a different path.
Can the Coherence Assumption Be Safely Removed?

- LC says yes.
  - if the program needs coherence at a given level, it should express this need explicitly.

- Each location which is written to is associated with a partially-ordered multiset (pomset):
  - as long as no chain of synchronizing accesses is performed on a given location $x$, its pomset can only grow, and any subsequent read request can return any of the values contained in the pomset.

- If a given (set of) location is used in combination with acquire-release pairs, then their pomset is reduced to one element.
An Example of Execution for LC

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w_1 : L \leftarrow val_1$</td>
<td>$w_3 : L \leftarrow val_3$</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>$w_2 : L \leftarrow val_2$</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>$r_1 : \ldots \leftarrow L$</td>
<td>$r_2 : \ldots \leftarrow L$</td>
</tr>
<tr>
<td>$sync(t_1, t_2)$</td>
<td>$sync(t_1, t_2)$</td>
</tr>
<tr>
<td>$r_3 : \ldots \leftarrow L$</td>
<td></td>
</tr>
</tbody>
</table>

Values read

$r_1 \{ val_2, val_3 \}$

$r_2 \{ val_1, val_2, val_3 \}$

$r_3 \{ val_2 \}$ OR $\{ val_3 \}$
Two Open Questions

- Should the hardware allow for more than one routing path from one core to a given memory location?
- If the answer is yes, should the hardware allow for multiple operations issued by the same core to arrive out-of-order?

LC’s answer is yes to both.
Introduction

Why UHPC?
The UHPC Teams
What Is an Execution Model?

Our Concurrency Model: The Codelet Model

The Codelet Execution Model
Codelet Graphs
Well-Behaved Codelets

Memory Model

Hardware-Oriented Consistency Models
Software-Oriented Consistency Models
Our Basis for the Codelet Memory Model: Location Consistency

Self-Awareness

Conclusion
Power

As with everything else in our model, self-awareness will have to be handled in various hierarchies:

- at the user level: the user defines a goal for the overall computation when starting a computation
- at the high-level programmer/compiler level: provide locality hints to the underlying runtime system and hardware to avoid useless data and/or code movement (percolation).
- at the runtime level: some events come from hardware probes (“Hot! Too hot!”), and the runtime system needs to change its scheduling policy, clock-gate or power-gate cores on the chip, etc.
Resiliency

- With hundreds or thousands of cores, not all will have the same reliability for all frequencies and all voltages.
- Near-threshold voltage and in general power-scaling or frequency-scaling can make a (set of) core behave strangely (read: compute incorrect results).

Therefore, the runtime system must be able to handle such cases. For example:

- Run a same computation multiple times in parallel to verify the correctness of a particularly important result.
- Give hints about where and when to perform check-pointing.
- Have the hardware perform automatic check-pointing anyway.
Introduction

Why UHPC?
The UHPC Teams
What Is an Execution Model?

Our Concurrency Model: The Codelet Model

The Codelet Execution Model
Codelet Graphs
Well-Behaved Codelets

Memory Model

Hardware-Oriented Consistency Models
Software-Oriented Consistency Models
Our Basis for the Codelet Memory Model: Location Consistency

Self-Awareness

Conclusion

Zuckerman
Quick Recap

- Codelets are small groups of machine instructions scheduled atomically. They are grouped into codelet graphs (CDG), which are contained in threaded procedures. If CDGs are well-behaved, then the computation is determinate.

- The memory consistency model we want to use as a basis for the Codelet PXM is Location Consistency, where everything is local by default. If some memory accesses need to be synchronized, explicit instructions must be added.
Alright, alright, I know. You want the results.

... We don’t have any (for now!). But here is what we are about to do:

▶ Create a “reference” runtime system, which will be usable by all UHPC members (not only Intel, but Sandia, Nvidia, MIT too). We are in the design phase.

▶ Create a “codelet-aware” C compiler, using LLVM. Some work is already under way.

▶ Bridge the gap between the high-level languages (CnC, HTA) and our codelet model / ETI’s runtime system (SWARM) – and hopefully our reference runtime system.
Alright, alright, I know. You want the results.

... We don’t have any (for now!). But here is what we are about to do:

▶ Create a “reference” runtime system, which will be usable by all UHPC members (not only Intel, but Sandia, Nvidia, MIT too). We are in the design phase.
▶ Create a “codelet-aware” C compiler, using LLVM. Some work is already under way.
▶ Bridge the gap between the high-level languages (CnC, HTA) and our codelet model / ETI’s runtime system (SWARM) – and hopefully our reference runtime system.
S. Adve, V. Pai, and P. Ranganathan.
Recent advances in memory consistency models for hardware shared memory systems.
ISSN 0018-9219.
doi: 10.1109/5.747865.

S. V. Adve and M. D. Hill.
Weak ordering—a new definition.

M. Botinčan, P. Glavan, and D. Runje.
Verification of causality requirements in java memory model is undecidable.
ISBN 3-642-14402-0, 978-3-642-14402-8.
URL http://portal.acm.org/citation.cfm?id=1893586.1893595.

J. Dennis, J. Fosseen, and J. Linderman.
Data flow schemas.
J. B. Dennis.
First version of a data-flow procedure language.

J. B. Dennis, J. B. Fosseen, and J. P. Linderman.
Data flow schemas.

M. Dubois, C. Scheurich, and F. Briggs.
Memory access buffering in multiprocessors.

G. R. Gao and V. Sarkar.
Location consistency-a new memory model and cache consistency protocol.
ISSN 0018-9340.
doi: 10.1109/12.868026.
URL http://portal.acm.org/citation.cfm?id=354862.354865.


S. Polyakov and A. Schuster.
Verification of the java causality requirements.
URL http://dx.doi.org/10.1007/11678779_16.
10.1007/11678779_16.