Parallel patterns + Macro Data Flow for multi-core programming

M. Aldinucci\textsuperscript{o}, L. Anardu\textsuperscript{t}, M. Danelutto\textsuperscript{t}, P. Kilpatrick\textsuperscript{*}, M. Torquati\textsuperscript{t}

\textit{Univ. of Torino}\textsuperscript{o} – \textit{Univ. of Pisa}\textsuperscript{t} – \textit{Queen’s Univ. of Belfast}\textsuperscript{*}

20\textsuperscript{th} Euromicro PDP – Garching (D) – Feb. 15–17, 2012
Talk contents

- Scenario
- Data flow → macro data flow
- HLL → macro data flow
- Experimental results
- Ongoing work
- Future work & conclusions
Scenario (hw)

Multicores

- cores per socket $\uparrow$ $(O(10) \rightarrow O(100))$
- core complexity $\downarrow$ (no OoO, Branch prediction, ...)
- memory hierarchy mechanisms complexity $\uparrow$
  (snoop optimizations, transactional mechanisms, ...)

Co-processors

- GPUs $\rightarrow$ more cores $+$ better control unit management
- larger and better performing FPGAs
- GP coprocessors
  - Intel Many Integrated Core, Tilera
Scenario (sw)

CPU
- OpenMP
- Pthreads + Parallel Design Patterns

GPU, Accelerators
- Cuda → OpenCL
- Vector compiler technology → OpenAAC

COW/NOW
- MPI
Data flow

Well known technology

- flow of instructions
  - determined by data availability
  - rather than by Program Counter
- used in
  - compiler (register allocation, code optimization, VLIW, ...)
  - processor (instruction scheduling, resource allocation, ILP, ...)

- fine grained
  - enhance execution of sequential programs on Von Neumann architectures
  - by removing unnecessary dependencies
Well known technology

- flow of instructions
  - determined by data availability
  - rather than by Program Counter
- used in
  - compiler (register allocation, code optimization, VLIW, ...)
  - processor (instruction scheduling, resource allocation, ILP, ...)

- fine grained
  - enhance execution of sequential programs on Von Neumann architectures
  - by removing unnecessary dependencies
Macro Data Flow

Raise level of abstraction

- instructions $\rightarrow$ block of code
  - e.g. functions, method calls, legacy code wrappers, ...

Consequences

- improve management overhead masking
- define intermediate formalism suitable to target multi/many
data parallel sub-graphs or complex instructions $\rightarrow$
coprocessors cores
Macro Data Flow (MDF)

Concept

- Programs → MDF instruction graphs
- MDF instruction → data flow instruction with coarse grain “function” (sequential code portion)
- MDF interpreter → scheduling → fireable instructions to available interpreters (executors)
MDF: stream/data parallelism (1)

Streaming (stream parallelism)

- input stream
- data items $\rightarrow$ instances of MDF graphs
- fireable MDF instruction: from the same graph instance or from different instances
MDF: stream/data parallelism (2)

Data parallelism

- collection ➔ split instruction
- split instruction ➔ collection of fireable instructions
- collection of fired instructions ➔ collect instruction
Mechanisms

- same in the two cases

Fireable instruction scheduling

- optimization possible (affinity scheduling)

Coprocessor targeting

- identify data parallel sub-graphs (keep info from compiler)
- identify stream parallel computation (keep info from compiler)
Coproessor targeting (data parallelism)
Coprocessor targeting (data parallelism)
MDF vs. “tasks”

Task

- OpenMP, TPL (Microsoft), TBB (Intel), Task SuperScalar (BSC), ...
- user identified portion of code suitable for concurrent/parallel computation
- A.K.A. MDF instructions (fully embedded in “traditional” code)

MDF

- compiler $\rightarrow$ MDF graphs
- MDF instructions $\rightarrow$ “tasks”
MDF: implementation

application programmer concerns

- HLL source
- input data

system programmer concerns

- compiler
- MDF graph
- input manager
- task pool

Parallel MDF interpreter

- mdf th 1
- mdf th 2
- mdf th n

- core1
- core2
- core n
MDF: obtaining graphs

From structured high level code

- parallel design patterns
- algorithmic skeletons

Numeric applications

- *de facto* standard
  with calls to num libs
MDF: obtaining graphs

From structured high level code

- parallel design patterns
- algorithmic skeletons

Numeric applications

- \textit{de facto} standard
  with calls to num libs

\textbf{No (suitable) way to compile MDF graphs from arbitrary sequential code}
Parallel design patterns

- stream parallel: pipeline, farm, ...
- data parallel: map, reduce, stencil, ...
- typically translate into a small number of MDF sub-graphs:

pipeline

\[
\begin{align*}
& f \rightarrow g \rightarrow h \\
& x \rightarrow y \\
\end{align*}
\]

map

\[
\begin{align*}
& \text{decomp} \rightarrow f \rightarrow \text{recomp} \\
& x \rightarrow y_1 \rightarrow \text{x} \rightarrow y \\
\end{align*}
\]

reduce

\[
\begin{align*}
& \text{partition} \rightarrow (+) \rightarrow \text{x} \rightarrow \text{y} \\
& x \rightarrow y_1 \rightarrow y_2 \\
\end{align*}
\]
Parallel pattern composition

- skeleton → graph (single input and output token (arc))
- skeleton composition → graph composition (I/O arcs merged)
Composition: “accommodate diversity”

Cole’s skeleton manifesto

- allow users to express parallelism not available with current pattern set
- “We must be careful to draw a balance between our desire for abstract simplicity and the pragmatic need for flexibility.”

Provide API:

- to develop and name MDF graphs
- with single in/out token
- and to use them as “patterns”
Composition: “accommodate diversity”
Composition: compile time optimizations

Pattern rewriting rules

- well known
- organized in libraries
- possibly associated with cost models
  \[\rightarrow\] support performance driven rewriting
Composition: compile time optimizations

rewriting rule library

\[ \text{farm}(A) = A \]
\[ \text{pipe}(A,B) = \text{comp}(A, B) \]
\[ \text{comp}(	ext{map}(A), \text{map}(B)) = \text{map}(	ext{comp}(A,B)) \]

pipe

\[ f \]

map

\[ g \]
\[ g \]

merge

\[ h \]
\[ h \]

merge

pipe

\[ f \]

map

\[ g \]
\[ g \]

merge

\[ h \]
\[ h \]
Well formed numeric code $\Rightarrow$ MDF

**Numeric kernels**

- loops
  + calls to numerical libs

```plaintext
FOR k = 0..TILES-1
  FOR n = 0..k-1
    A[k][k] :=
    CHERK(A[k][n],A[k][k])
    A[k][k] :=
    CPOTRF2(A[k][k])
  FOR m = k+1..TILES-1
    FOR n = 0..k-1
      A[m][k] :=
      CGEMM(A[k][n],A[m][n],A[m][k])
      A[m][k] :=
      CTRSM(A[k][k],A[m][k])
```
Experimental validation

“show the pay-back” (Cole’s manifesto)

▶ MDF interpreter (different versions)
▶ Stream/Data parallelism
▶ Comparison with state-of-the-art-tools
▶ Distributed interpreters
▶ Heterogeneous architecture targeting
Implementation with pipes

- single request pipe
  worker threads send thread id to ask fireable instruction
- per thread task pipe
  task pool thread send fireable instruction pointers

Implementation "Pthreads"

- request and task objects protected with mutexes
MDF interpreter: different mechanisms (2)

![Graph showing completion time (secs) vs. speed up for different parallelism degrees.]

- Ideal
- m^2 df-pipe
- m^2 df-pthread

Completion time (secs) vs. Parallelism degree.
MDF interpreter: different use cases

![Graph showing completion time vs. parallelism degree for different graph types: Ideal, Generic Graph, Pipeline Graph, and Map Graph. The graph indicates speed up as a function of parallelism degree.](attachment:image.png)
Streaming vs. MDF

When processing streams of tasks:

- optimal → farm with sequential workers computing the whole single task (normal form of stream parallel skeletons [PDCS’99])

Compared with MDF:

- additional parallelism in the computation of the single input stream task
  → overhead expected, but ...
Streaming vs. MDF

![Graph showing speedup vs. parallelism degree for different stream lengths and MDF configurations.](image)
Comparison with *state-of-the-art*

**MDF vs. OpenMP**
- matrix multiplication code
- kind of “ideal case” for OpenMP
  (but some option tuning is necessary (chunk, scheduling))

**MDF vs. Plasma**
- Univ. of Tennessee
- specifically designed and optimized to target shared cache multi-core platforms
- Cholesky factorization
  (one of the use cases demonstrating Plasma features)
MDF vs. OpenMP (Intel Nehalem)

![Graph showing performance comparison]

- **Completion time (secs)**
- **Speed up**
- **Parallelism degree**
- **Ideal**
- **OpenMP**
- **m²df-pipe**
MDF vs. Plasma (Intel Nehalem)
MDF vs. Plasma (AMD Magny Cours)

![Graph showing completion time vs. parallelism degree for MDF and Plasma]

- MDF
- PLASMA static
- PLASMA dynamic

Completion Time (msec)
Parallelism degree
mdf
PLASMA static
PLASMA dynamic
Distributed task pool (ongoing)

Distributed task pool

- (fireable) MDF instructions at the TP nodes in a tree
- fireable MDF instruction stealing
- preliminary results demonstrate feasibility & load balancing (24 core NUMA architecture)

- speedup on dual AMD (2x12 core) machine → (max 12 workers)
Modified interpreter (HPLGPU’12)

- one thread feeding data parallel tasks to
- balanced usage of CPU and GPU cores achieved

<table>
<thead>
<tr>
<th>#cores</th>
<th>unoptimized GPU copies</th>
<th></th>
<th></th>
<th>optimized GPU copies</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg task/core</td>
<td>GPU tasks</td>
<td>GPU %</td>
<td>Avg task/core</td>
<td>GPU tasks</td>
<td>GPU %</td>
</tr>
<tr>
<td>1</td>
<td>53</td>
<td>459</td>
<td>89%</td>
<td>43</td>
<td>469</td>
<td>91%</td>
</tr>
<tr>
<td>2</td>
<td>53</td>
<td>406</td>
<td>79%</td>
<td>37</td>
<td>438</td>
<td>85%</td>
</tr>
<tr>
<td>4</td>
<td>49</td>
<td>317</td>
<td>61%</td>
<td>34</td>
<td>377</td>
<td>73%</td>
</tr>
<tr>
<td>8</td>
<td>40</td>
<td>193</td>
<td>37%</td>
<td>30</td>
<td>272</td>
<td>53%</td>
</tr>
<tr>
<td>16</td>
<td>30</td>
<td>42</td>
<td>8%</td>
<td>23</td>
<td>141</td>
<td>27%</td>
</tr>
</tbody>
</table>
Conclusions

- feasible intermediate programming model for multi/many cores
- feasible compiler technology from HLL and Parallel Design Patterns (Algorithmic Skeletons)
- seamless integration of stream & data parallelism
- results partially derived from Muskel experience (pure Java algorithmic skeleton framework, MDF)
- experimental results assess feasibility
  - linear speedup on medium grain computations
  - on different state-of-the-art architectures (Intel (Nehalem, Sandy Bridge, Westmere), AMD (Magny Cours), nVidia (GeForce, Fermi), (TileraPro ongoing))
Paraphrase perspective

ParaPhrase

- FP7 EU Strep “Parallel Patterns for Adaptive Heterogeneous Multicore Systems”
- 9 partners involved, including QUB, UNITO and UNIPI
- started on Oct. 2011, 3 year project

Perspective

- possible usage of MDF technology to support pattern implementation (skeletons)
- alternative to template based implementation(s)
THANK YOU, any questions?

marcod@di.unipi.it

http://www.di.unipi.it/~marcod