Targeting heterogeneous multi/many core architectures through macro data flow

M. Danelutto

Dept. Computer Science – Univ. of Pisa

Amsterdam, March 2012
Scenario

Data flow → macro data flow

HLL → macro data flow

Experimental results

Ongoing work

Future work & conclusions
Scenario (hw)

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

**Co-processors**
- GPUs → more cores + better control unit management
- larger and better performing FPGAs
- GP coprocessors
  - Intel Many Integrated Core, Tilera
Scenario (sw)

**CPUs**
- 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

M. Danelutto
Targeting heterogeneous multi/many core architectures through macro data flow
Macro Data Flow

**Raise level of abstraction**

- instructions → 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 → coprocessors cores

M. Danelutto
Targeting heterogeneous multi/many core architectures through macro data flow
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

M. Danelutto
Targeting heterogeneous multi/many core architectures through macro data flow
MDF: stream/data parallelism (2)

Data parallelism

- collection → split instruction
- split instruction → collection of fireable instructions
- collection of fired instructions → collect instruction
### MDF: stream/data parallelism (3)

**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)
Coprocessor 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: a more radical and orchestrated approach**
- 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

M. Danelutto
Targeting heterogeneous multi/many core architectures through macro data flow
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
- *de facto* standard
  with calls to num libs

No (suitable) way to compile MDF graphs from arbitrary sequential code
Parallel patterns/skeletons $\rightarrow$ MDF

Parallel design patterns

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

\[
\text{pipeline} \quad \text{map} \quad \text{reduce}
\]
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
  → support performance driven rewriting
Composition: compile time optimizations

M. Danelutto

Targeting heterogeneous multi/many core architectures through macro data flow
Well formed numeric code → MDF

Numeric kernels

- loops
  - calls to numerical libs

FOR \( k = 0..\text{TILES-1} \)
  FOR \( n = 0..k-1 \)
    \[ A[k][k] := \]
    \[ \text{CHERK}(A[k][n], A[k][k]) \]
    \[ A[k][k] := \]
    \[ \text{CPOTF2}(A[k][k]) \]
  FOR \( m = k+1..\text{TILES-1} \)
    FOR \( n = 0..k-1 \)
      \[ A[m][k] := \]
      \[ \text{CGEMM}(A[k][n], A[m][n], A[m][k]) \]
      \[ A[m][k] := \]
      \[ \text{CTRSM}(A[k][k], A[m][k]) \]
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
MDF interpreter: different mechanisms (1)

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)

M. Danelutto
Targeting heterogeneous multi/many core architectures through macro data flow
MDF interpreter: different use cases

- Ideal
- Generic Graph
- Pipeline Graph
- Map Graph

Completion time (secs) vs Parallelism degree

M. Danelutto
Targeting heterogeneous multi/many core architectures through macro data flow
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

M. Danelutto
Targeting heterogeneous multi/many core architectures through macro data flow
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)

Completion time (secs)

Parallelism degree

Ideal
OpenMP
m²df-pipe

Speed up

M. Danelutto
Targeting heterogeneous multi/many core architectures through macro data flow
MDF vs. OpenMP (AMD Magny Cours)

Completion time (AMD 4/2/6 (board/socket/core) Opteron 6174)

- AMD Opteron(tm) Processor 6174: MDF
- AMD Opteron(tm) Processor 6174: OMP

M. Danelutto
Targeting heterogeneous multi/many core architectures through macro data flow
MDF vs. Plasma (Intel Nehalem)
MDF vs. Plasma (AMD Magny Cours)

Completion Time (msec) vs. Parallelism degree for MDF and PLASMA.

- **mdf³**
- **PLASMA static**
- **PLASMA dynamic**

M. Danelutto

Targeting heterogeneous multi/many core architectures through macro data flow
Targeting heterogeneous architectures

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>optimized GPU copies</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg</td>
<td>GPU</td>
</tr>
<tr>
<td></td>
<td>task/core</td>
<td>tasks</td>
</tr>
<tr>
<td>1</td>
<td>53</td>
<td>459</td>
</tr>
<tr>
<td>2</td>
<td>53</td>
<td>406</td>
</tr>
<tr>
<td>4</td>
<td>49</td>
<td>317</td>
</tr>
<tr>
<td>8</td>
<td>40</td>
<td>193</td>
</tr>
<tr>
<td>16</td>
<td>30</td>
<td>42</td>
</tr>
</tbody>
</table>
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 \(\rightarrow\) (max 12 workers)
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