Structured approaches for multi/many core targeting

Marco Danelutto
M. Torquati, M. Aldinucci, M. Meneghin,
P. Kilpatrick, D. Buono, S. Lametti

Dept. Computer Science, Univ. of Pisa
CoreGRID Programming model Institute

Sept. 18, 2012, IRILL – Paris
Contents

Introduction
   Hardware
   Programming models

FastFlow
   FastFlow intro
   Core FastFlow details

FastFlow improvements
   Distributed FastFlow
   New skeletons
   Co-processor targeting

Perspectives
   ParaPhrase
   Ocaml perspectives

Danelutto & co.
Structured approaches for multi/many core targeting

http://www.di.unipi.it
Hardware scenario

General purpose computing devices

- multi-core
- 4 → 8/16 per socket
- decreased core complexity
- multiple thread contexts per core
- NUMA
Hardware scenario

Co-processors

- GPUs
  - more “cores” & SMPs
  - direct access to memory hierarchy
  - RDMA & cooperation within clusters

- Many core
  - 50 to 100 cores
  - simpler interconnection networks
  - much simpler cores
Programming model shift

Good sequential code parallelized to tune performance
Programming model shift

Good sequential code parallelized to tune performance

Good parallel code with seq portions optimized to tune performance subject to parallelisation constrain
Programming model scenario

Distributed memory
- MPI

Shared memory
- OpenMP
  - data parallel
  - task parallel
  → data flow (DAG)

Co-processors
- CUDA
- (OpenMP)
Programming model scenario

Distributed memory
- MPI

Shared memory
- OpenMP
  - data parallel
  - task parallel
  → data flow (DAG)

Co-processors
- CUDA
- (OpenMP)

- relatively low level
- sync&comm entirely in charge of the application programmer
- require quite deep target hw knowledge to tune performance
- efficient co-processor exploitation
  → nightmare
Skeletons and design patterns

Algorithmic skeletons

- since ’90s, from HPC community
- languages (P3L), libraries (eSkel, Skipper, Calcium/Skandium, SkeTo, Muskel, ...)
- key concept → efficient, parametric, language constructs/library entries modelling common parallel patterns available to the application programmers
Skeletons and design patterns

Algorithmic skeletons

- since ’90s, from HPC community
- languages (P3L), libraries (eSkel, Skipper, Calcium/Skandium, SkeTo, Muskel, ...)
- key concept → efficient, parametric, language constructs/library entries modelling common parallel patterns available to the application programmers

Parallel design patterns

- since ’00s, from SW engineering community
- key concept → recipes to program common parallel patterns
- structured (layered): concurrency finding, algorithm structure, support structure, mechanism design spaces
Patterns (the skeleton way ...)

Berkeley report

- feasible way to attack parallel programming
- and to form new generations of programmers
Patterns (the skeleton way ...)

Berkeley report
- feasible way to attack parallel programming
- *and to form new generations of programmers*

Algorithmic skeleton community
- promoting since decades the concept
- with notable frameworks (SkeTo, OcamlP3L)
Patterns (the skeleton way ...)

Berkeley report

▶ feasible way to attack parallel programming
▶ and to form new generations of programmers

Algorithmic skeleton community

▶ promoting since decades the concept
▶ with notable frameworks (SkeTo, OcamIP3L)

User community (application programmers)

▶ asking more and more abstraction to “hide” (encapsulate) parallelism exploitation
Patterns (major commercial actors)

- Intel TBB
  - include simple patterns (e.g. pipeline)
  - elaborate on the design pattern concept
Patterns (major commercial actors)

- Intel TBB
  → include simple patterns (e.g. pipeline)
  → elaborate on the design pattern concept
- Microsoft TPL
  → include simple patterns (data and stream parallel)
  → evolving to macro data flow
Patterns (major commercial actors)

- Intel TBB
  - include simple patterns (e.g. pipeline)
  - elaborate on the design pattern concept
- Microsoft TPL
  - include simple patterns (data and stream parallel)
  - evolving to macro data flow
- Google
  - MapReduce
  - further elaborating on pattern composition (Flume)
FastFlow (What/Why)
Parallel programming framework

▷ algorithmic skeleton approach
▷ C++ based
▷ streaming “process” networks
▷ targeting shared memory architectures
FastFlow (What/Why)

Parallel programming framework

- algorithmic skeleton approach
- C++ based
- streaming “process” networks
- targeting shared memory architectures
- extremely efficient in inter-concurrent activity communication (in the range of 10nsecs)
- better or comparable efficiency/performance w.r.t. “standard” programming environments (OMP, Cilk, TBB)
FastFlow (What/Why)

Parallel programming framework

- algorithmic skeleton approach
- C++ based
- streaming “process” networks
- targeting shared memory architectures
- extremely efficient in inter-concurrent activity communication (in the range of 10nsecs)
- better or comparable efficiency/performance w.r.t. “standard” programming environments (OMP, Cilk, TBB)
- raising the level of abstraction exposed to the user
  → impacting programmer efficiency and time-to-deploy
FastFlow kernel

- layered design
- incremental implementation of efficient inter-thread communication mechanisms (efficient, lock-free, wait-free bounded and un-bounded SPSC queue $\rightarrow$ SPMC and MPSC queues)
- simple streaming patterns provided as customizable skeletons (pipeline and task farm)
- software accelerator abstraction

Efficient applications for multicore and manycore

- Streaming network patterns
  - Skeletons: pipeline, farm, divide&conquer, ...
- Arbitrary streaming networks
  - Lock-free SPSC, SPMC, MPSC, MPMC queues
- Simple streaming networks
  - Lock-free SPSC queues and general threading model (e.g. Pthread)
- Multicore and Manicore
  - cc-UMA and cc-NUMA featuring sequential or weak consistency
FastFlow queue(s)

Built on previous work:
- only hosts pointers
- push and pop operations
- NULL is a special (non queue-able) value
- active wait on pop from empty and push on full (bounded) queues
- unbounded version (supporting buffer re-use)
**FastFlow ff_node**

Main concept:

- concurrent activity with associated input and output queue
- wrapper of seq code or skeleton
- arbitrary compositions: sequential ff_node $\rightarrow$ farm or pipeline
- three methods:
  1. `void * svc(void *)`
     computes a result out of a task
  2. `int svc_init(void)`
     called once before starting the application code
  3. `void svc_end(void)`
     called before application shutdown
Stream parallel skeletons

- pipeline
  arbitrary number of stages
- farm
  Emitter-StringOfWorkers-Collector
  or Master/Worker
- feedback channel
  only in outer skeleton, feeds back (partial) results from last stage
  output to first stage input
- free composition → quite complex process graphs
Stream parallel skeletons

- pipeline
  arbitrary number of stages
- farm
  Emitter-StringOfWorkers-Collector or Master/Worker
- feedback channel
  only in outer skeleton, feeds back (partial) results from last stage output to first stage input
- free composition → quite complex process graphs
Accelerator interface

Standard streaming programs:
- first stage generates the input stream
- middle stage computes results
- last stage outputs results

Accelerator mode:
- create skeleton program
- offload tasks to be computed when needed
- get results asynchronously
- uses “spare cores”

---

```
// FastFlow accelerated code
#define N 1024
long A[N][N], B[N][N], C[N][N];
int main() {
    // < init A, B, C>
    ff::farm<> farm(true /* accel */);
    std::vector<ff::node *> w;
    for(int i=0; i<PAR DEGREE;++i)
        w.push_back(new Worker);
    farm.add_workers(w);
    farm.run_then_freeze();

    for (int i=0; i<N; i++) {
        for (int j=0; j<N; ++j) {
            task_t * task = new task_t(i,j);
            farm.ofload(task);
        }
    }

    farm.ofload((void *)ff::FF_EOS);
    farm.wait(); // Here join
}

// Includes
struct task_t {
    task_t(int i, int j):i(i), j(j) {}
    int i, int j;
};

class Worker: public ff::node {
public: // Offload target service
    void * svc(void *task) {
        task_t * t = (task_t *)task;
        int _C=0;
        for(int k=0; k<N; ++k)
            _C += A[t->i][k]*B[k][t->j];
        C[t->i][t->j] = _C;
        delete t;
        return GO.ON;
    }
};
```
Overall (FastFlow)

Skeletons:
- template process networks (skeletons?)
- customizable semantics

Shared memory:
- “capabilities” (pointers) passed between threads
- consistent usage of the capabilities in charge to the programmer
- extremely efficient comms (fwd: see Tilera implementation)

C++:
- suitable abstraction mechanisms to provide skeletons/templates
Distributed FastFlow

TCP/IP FastFlow queues:
- on top of ZMQ, same signature as “internal” queues
- unicast, broadcast, scatter, on-demand, fromany, gatherall protocols

Distributed skeletons:
- Either input or output channel → “distributed” queue
- SPMD-like model to launch sender and receiver nodes
- supports coordinated execution of FastFlow programs on different NOW PEs
Distributed FastFlow

Impact of network bandwidth (MM benchmark)

Danelutto & co.
http://www.di.unipi.it
Structured approaches for multi/many core targeting
Distributed FastFlow

Impact of network bandwidth (MM benchmark)
Distributed FastFlow

Serialization/deserialization

- Homogenous machine assumption
- In charge to the application programmer (!)
  - Prepare method when sending iovec of \(\langle address, length\rangle\) pairs
  - Prepare and unmarshalling when receiving prepare buffers
    rebuild structure from memory segments
- Extremely versatile but also low level ...
Data parallel skeletons

Original FastFlow:

- data parallel implemented with farm
- emitter → decompose data to tasks
- collector → rebuild result from partial results

ff_map skeleton:

- “leaf” node (two tier model)
- splitter policy in charge to the programmer
- efficient implementation using “farm like” template
- supports map, stencil and reduce like data parallel computations
FastFlow and co-processors

Two different kind of co-processors considered:

1. GP-GPUs
   - suitable for data parallel computations only
   - available almost everywhere, possibly in multiple instances
   - extremely cheap and power efficient

2. many core, general purpose co-processors
   - POSIX thread compliant
   - ranging in the 10 to 100 of (simple) cores
   - not so cheap nor largerly available (e.g. Tilera, Intel MIC)
FastFlow for Tilera Pro64

Different memory subsystem
- re-engineering the SPSC queue
- barrier needed (weak store consistency)

#ifdef __x86_64__ // x86 32/64-bit: no memory fence is needed.
define WMB() __asm__ __volatile__ ("": : "memory")
#endif
#endif
#ifndef __tile__ // Tilera: using a compiler intrinsic for memory fence.
define WMB() __insn_mf();
#endif
FastFlow for Tilera Pro64

Cache coherence protocol
- may be disabled
- may be “customized”

Experiments:
- Hash home node (HHN)
- No coherency (NHN)
- Fixed home node (FHN) (Protocol directed allocation)

Matrix multiplication (stream of 64x64 matrixes)
FastFlow for Tilera Pro64

![Speedup/Scalability Graph]

- Ideal
- Speedup
- Scalability

Danelutto & co.

http://www.di.unipi.it

Structured approaches for multi/many core targeting
FastFlow for Tilera Pro64

Co-processor targeting

Host (BC)
```
cost string h2t="/dev/tilerapi/h2t/0";
cost string t2h="/dev/tilerapi/t2h/0";
ff_tileAccelerator TA(h2t,t2h);
TA.init(num_buffers_h2t, num_buffers_t2h);

TA.offload(task,size);
......
TA.get_result(task);

TA.close();
```

TilePro64
```
cost string h2t="/dev/tilerapi/h2t/0";
cost string t2h="/dev/tilerapi/t2h/0";
ff_tileAccelerator TA(h2t,t2h);
TA.init();

farm

 TA.get(task);

TA.put(task);

TA.close();
TA.init(num_buffers_h2t, num_buffers_t2h);
```

structured approaches for multi/many core targeting
FastFlow & GPUs

Two-tier model:
- Stream parallel skeletons $\rightarrow$ upper part of the skeleton tree
- Data parallel skeletons $\rightarrow$ lower part of the skeleton tree
- Sequential wrappers $\rightarrow$ skeleton tree leaves
FastFlow & GPUs

Two-tier model:

- Stream parallel skeletons → upper part of the skeleton tree
- Data parallel skeletons → lower part of the skeleton tree
- Sequential wrappers → skeleton tree leaves
- GPU code embedded into `ff_nodes`
FastFlow & GPUs

Two-tier model:

- Stream parallel skeletons → upper part of the skeleton tree
- Data parallel skeletons → lower part of the skeleton tree
- Sequential wrappers → skeleton tree leaves

- GPU code embedded into `ff_node`
  - any kind of code to manage GPUs:
    - Cuda, OpenCL but also SkePu, ...
  - single/multiple GPU management + kernel scheduling
    - under user responsibility

Danelutto & co.
http://www.di.unipi.it

Structured approaches for multi/many core targeting
FastFlow & GPUs

#define SKEPU_CUDA

UNARY_FUNC(iters, float, a, {a++; return(a);})

class MapStage:public ff_node {
  void *svc(void * task) {
    skepu::Vector<float> * v =
      (skepu::Vector<float> *) task;
    skepu::Vector<float> * r =
      new skepu::Vector<float>(NN);
    skepu::Map<iters> skepumap(new iters);
    skepumap(*v, *r);
    free(task); return((void *) r);
  }
};
FastFlow & GPUs

Alternatively:

- user “worker” function provided as fixed signature C++ function
- wrappers for GPUs and CPUs
e.g.
  - OpenMP for CPUs
  - CUDA/OpenCL for GPUs
Sample CPU wrapping

template<typename IN, typename OUT>
class ff_map_1D_cpu : public ff_node {
private:
    OUT (*f)(IN input);
public:
    ff_map_1D_cpu(OUT (*func)(IN)) { f = func; }

    void * svc(void * task) {
        task_1D<IN>* task1 = (task_1D<IN>*) task;
        IN* input_array = task1->get_array();
        size_t size = task1->get_size();
        OUT* result = new OUT[size];

        #pragma omp parallel for schedule(dynamic)
        for(size_t i = 0; i < size; i++) {
            result[i] = f(input_array[i]);
        }
        return (void*) new task_1D<OUT>(result, size);
    }
};
Sample GPU wrapping

Classic C function (with known types):

```c
out_type f(in_type arg) { ... }```

Wrapped through macro:

```c
#define MapKernel(func_name, out_type, in_type, arg_name) \
  __device__ out_type func_name(in_type arg_name); \
__global__ void MapKernel(in_type *input, out_type *output) \
{ \n  ... data -> GPU 
  ... kernel computation 
  ... results -> CPU 
}
```
Problems

Reliable and efficient GPU targeting within the `ffi_node`:

- CUDA? OpenCL? OpenAAC?
- kernel parameters?

GPU access:

- One `ffi_node` controller per GPU?
- (CUDA 5.0 → up to 32 concurrent controllers per (Kepler 110) GPU)

Multi GPU management:

- special `ffi_node` controller

Host/GPU memory traffic:

- CUDA 4.0: direct access to Host (pinned) memory
- CUDA 5.0: RDMA through PCIe (?)
ParaPhrase perspective

- EU FP7 STREP project
- StAndrews, RGU, QUB, Erlang Solutions (UK), UNIPI, UNITO (IT), SCCH (A), Stuttgart (D), Mellanox (Israel)
- Parallelism managed through patterns, refactored and optimized, eventually compiled to skeleton code
- Independent but similar toolchains for Erlang and C++/FastFlow
ParaPhrase perspective

FastFlow

- candidate framework to validate project methodology
- FastFlow skeletons introduce through user assisted refactoring of seq code
- FastFlow skeleton optimization through refactoring
- CPU/GPU targeting managed automatically (refactoring + mapping)
- run time adaptation of refactoring/mapping choices
ParaPhrase perspective

FastFlow

- candidate framework to validate project methodology
- FastFlow skeletons introduce through user assisted refactoring of seq code
- FastFlow skeleton optimization through refactoring
- CPU/GPU targeting managed automatically (refactoring + mapping)
- run time adaptation of refactoring/mapping choices

Erlang

- pure functional + CSP style processes with send/receive
- FastFlow-like skeleton set experimented on real applications
Ocaml perspectives

Ocaml for multicore

Experience with parmap

- possibility to exploit pretty efficient medium grain parallelism
- relying on processes rather than threads (*by necessity*)
- quite an amount of different “parallel” libraries for Ocaml
  - proof-of-concept → need for some “language” parallelism
  - from different applicative domains → common features

Danelutto & co.
http://www.di.unipi.it
Structured approaches for multi/many core targeting
Ocaml for multicore

Experience with parmap

- possibility to exploit pretty efficient medium grain parallelism
- relying on processes rather than threads \((\text{by necessity})\)
- quite an amount of different “parallel” libraries for Ocaml
  - proof-of-concept \(\rightarrow\) need for some “language” parallelism
  - from different applicative domains \(\rightarrow\) common features

Wish list (high level, tentative):

- efficient “native” mechanisms
- supporting customizable parallel patterns
- minimal disruption w.r.t. current Ocaml program practices
Ocaml for multicore

Efficient "native" mechanisms

▶ efficient threading mechanisms (PThreads?)
▶ efficient comm/synch mechanisms (FastFlow?)

Customizable patterns:

▶ customizable patterns à la FastFlow + abstraction à la Ocaml

Minimal disruption w.r.t. current Ocaml program practices:

▶ application programmer exposed with high level, domain specific patterns
  ▶ completely embedding parallelism management
  ▶ composable + sequential embedding
Ocaml: inheriting from FastFlow

Accelerator:

- FastFlow accelerator (farm)
- each worker gets a (shared memory) closure and computes it
  - needs access to Ocaml (full) interpreter
  - within an independent (pinned) thread

Communication mechanisms:

- completely re-usable in threaded Ocaml environment
- grants fine grain comms and synchrons

Efficient allocator:

- built on top of base FastFlow layers
- greatly improves performance of programs with large numbers of “small” mallocs
Conclusions

FastFlow

- mature technology
- targeting different *state-of-the-art* architectures including
  - Intel (Nehalem, Sandy Bridge), AMD (Magny Cours),
  - Tilera (Tile64pro), Nvidia (GeForce, Tesla) (through SkePu)
- comparable or better performance w.r.t. standard environments
- completely open source (sourceforge/mc-fastflow)
- suitable to support different experiments with other (sequential) programming environments
Any questions?

(remember: this is team work with Torquati, Aldinucci, Meneghin, Kilpatrick, Buono, Lametti, ...)

http://calvados.di.unipi.it/fastflow