FastFlow
a pattern-based programming framework for multicores

Marco Aldinucci
Computer Science Dept. - University of Torino (Turin) - Italy

a joint work with
Massimo Torquati and Marco Danelutto
Computer Science Dept. - University of Pisa - Italy

Massimiliano Meneghin
IBM Technology Campus, Dublin Software Lab, Ireland

Peter Kilpatrick
Queen’s University Belfast, U.K.
Outline

From grid to multicore

Low-level parallel programming
- Mutual exclusion & Producer Consumer
- Producer Consumer, the FastFlow way

High-level programming & FastFlow
- Architecture and implementation
- Some experimental results & Applications

Composition, features and knows for auto-tuning
- ; and {} and self-offloading
- example: allocator
- example: composition and SIMD extension
- example: composition and other libraries (threading libraries, events, callbacks, ...)

http://mc-fastflow.sourceforge.net
Our frustrations, as GCM designers

---

Pretty flexible and composable but quite verbose
- lot of XML & java code factories to fill the object-to-component abstraction gap
- designed for grid (and for shared memory as byproduct)

---

Slow Monitoring, slow reconfigurations (seconds)
- suitable for very coarse grain parallelism
- tuning often complex due to observation-reaction drift

---

Not always suitable for multicore
- requires an unnaturally coarse grain, especially for shared memory programming
- ultimately failing to project variation points at high-level (i.e. patterns and their parameters) into real optimisation knobs at the run-time level
What to keep, what to rethink?

Pattern-based/skeletal approach

- high-level semantics, attacking **idiom recognition** problem ✔
- going (even too much) mainstream ✔
- human productivity ✔

Variation points mapping

- fully dynamic optimisations (rethink code factories) ✗
- streaming supported but not efficient ✗
- commodity multicore support ✗
[2009] QuickPath
(MESI-F Directory Coherence)
two features - two problems

Memory/Cache Coherence

Deal with multiple replicas of the same location in different caches

Memory Consistency

Deal with the ordering in which writes and reads at different locations take effect in memory (issued by either the same or different processors/cores)

- x86 (TSO), PPC (WO), alpha (RC), ...
Concurrent programming
basic mechanisms and paradigms
Basic low-level interaction models

- Low-level synchronisation in the shared memory model
  - Mutual Exclusion (mutex)
    - Typically used as basic building block of synchronisations
  - Producer Consumer

- They are not equally demanding
  - Mutual Exclusion is inherently more complex since requires deadlock-freedom
    - Require interlocked ops (CAS, ...), that induces memory fences, thus cache invalidation
    - Dekker and Bakery requires Sequential Consistency (++)
  - Producer Consumer is a cooperative (non cyclic) process
Lamport & FastFlow FIFO queues

Proved to be correct under SC
doesn’t work under weaker models
Pushing lot of pressure on coherence subsystem because both producer and consumer need to share both head and tail index of the queue

Lamport FIFO
1983

```c
push_nonblocking(data) {
    if (NEXT(head) == tail) {
        return EWOULDBLOCK;
    }
    buffer[head] = data;
    head = NEXT(head);
    return 0;
}

pop_nonblocking(data) {
    if (head == tail) {
        return EWOULDBLOCK;
    }
    data = buffer[tail];
    tail = NEXT(tail);
    return 0;
}
```
Lamport & FastFlow FIFO queues

push_nonbocking(data) {
    if (NEXT(head) == tail) {
        return EWOULDBLOCK;
    }
    buffer[head] = data;
    head = NEXT(head);
    return 0;
}

pop_nonblocking(data) {
    if (head == tail) {
        return EWOULDBLOCK;
    }
    data = buffer[tail];
    tail = NEXT(tail);
    return 0;
}

push_nonbocking(data) {
    if (NULL != buffer[head]) {
        return EWOULDBLOCK;
    }
    buffer[head] = data;
    head = NEXT(head);
    return 0;
}

pop_nonblocking(data) {
    data = buffer[tail];
    if (NULL == data) {
        return EWOULDBLOCK;
    }
    buffer[tail] = NULL;
    tail = NEXT(tail);
    return 0;
}

Lamport FIFO
1983

FastFlow FIFO
(based on FastForward PPoPP 2008)
Lock-free and CAS-free (fence-free)

---

**Single-Producer-Single-Consumer FIFO queues**

- Lamport et al. 1983 Trans. PLS (Sequential consistency only - passive)
- Giacomoni et al. 2008 PPoPP (Relaxed consistencies (e.g. TSO) - passive)

---

**Multiple-Producers-Multiple-Consumers FIFO queues**

- with CAS (at least one) - passive ... a plethora
- without CAS - passive ➔ Cannot be done
- without CAS - active ➔ FastFlow

---

Augmenting the picture with locks will destroy the picture
Unbounded queue

Recent development
- unpublished

Works as a buffer of buffers + iterator
- same properties of bounded queue, coherence-friendly
- single spinlock for (rare) reallocs

fastest than STL deque
- which is already pretty fast, and it is not concurrent
High-level paradigms
skeletons/patterns
Skeletons in Pisa (and Torino)

MPP

P3L
1991

SKE
1997

OCamlP3L
1998

Eskimo
2003

ASSIST
2001

ASSISTant
2008

GCM
2008

Lithium
2002

Muskel
2006

FastFlow
2009

Beowulf

Shared memory

Grid

Macro Data Flow

Autonomic
Skelettons: rationale

Abstract parallelism exploitation pattern by parametric code
- e.g. higher order function, code factories, C++ templates, ...
- Hopefully, in such a way they can composed and nested as programming language constructs

Provide user with mechanisms to specify the parameters
- functional (seq code) and extra-functional (QoS) parameters

Provide state-of-the-art implementation of each parallelism exploitation pattern
Data Parallel

is a method for parallelizing a single task by processing independent data elements of this task in parallel. The flexibility of the technique relies upon stateless processing routines implying that the data elements must be fully independent. Data Parallelism also supports Loop-level Parallelism where successive iterations of a loop working on independent or read-only data are parallelized in different flows-of-control and concurrently executed.

Task Parallel

is explicit in the algorithm and consists of running the same or different code on different executors (cores, processors, machines, etc.). Different flows-of-control (threads, processes, etc.) may communicate with one another as they work. Communication usually takes place to pass data from one thread to the next as part of the same data-flow graph.

Stream Parallel

can be used when there exists a partial or total order in a computation. By processing data elements in order, local state may be maintained in each filter. The set of skeletons provided by FastFlow could be further extended by building new C++ templates on top of the Fastflow low-level programming layer.
FastFlow
Design objectives

High-level
- expressivity
- possibility to compose high-level artefact compiled in isolation

Efficient support for multicore and streaming
- efficient and synchronisations, predictable performance
- lot of tuning knobs (possibility of auto-tuning)
E.g. farm (a.k.a. master-worker)

Common paradigm

- Model foreach and Divide & Conquer
- Can be used to build data-flow engine
- Exploit it as a high-order language construct
  - A C++ template factory exploiting highly optimised implementation
E.g. farm with POSIX lock/unlock

![Graph showing speedup vs number of cores]

- Ideal
- 50 μS
- 5 μS
- 0.5 μS

average execution time per task

Speedup vs Number of Cores

Wednesday, May 12, 2010
Very fine grain (0.5 μS)

Number of Cores

Speedup

Ideal
POSIX lock
CAS
Embarrassing parallelism?

Theoretically simple? Yes ...

- in our experience it is enough to model many streaming problems
- can be coupled with shared memory and concurrent data structures

... embarrassingly simple?

- Might be, or might be not.
- The work is independent, but the stream comes and goes to a single channel, thus the overall work is not independent at all
- More, the problem transfer from CPU to I/O and memory that is already a huge problem in multicore cache-coherent platforms
FastFlow: template

- Efficient applications for multicore and manycore
  - Smith-Waterman, N-queens, QT, C4.5, FP-Growth, ...
- Autonomic Behav.Skeletons
- Simulation Montecarlo
- Accelerator self-offloading
- Streaming networks patterns
  - Skeletons: Pipeline, farm, D&C, ...
- Arbitrary streaming networks (building blocks)
  - Lock-free SPSC, SPMC, MPSC, MPMC queues
- Simple streaming networks (building blocks)
  - Lock-free SPSC queues and general threading model
- Multi-core and many-core
  - cc-UMA or cc-NUMA featuring sequential or weak consistency

High-level programming
- Lock-free/fence-free non-blocking synchronisations
- C++ STL-like implementation

Wednesday, May 12, 2010
SPMC and MCSP via SPSC + control

**SPMC(x)** fence-free queue with x consumers
- One SPSC “input” queue and x SPSC “output” queues
- One flow of control (thread) dispatch items from input to outputs

**MPSC(y)** fence-free queue with y producers
- One SPSC “output” queue and y SPSC “input” queues
- One flow of control (thread) gather items from inputs to output

x and y can be dynamically changed

**MPMC** = **MCSP** + **SPMC**
- Just juxtapose the two parametric networks
Coarse grain (50 μS workload)
Medium grain (5 μS workload)
Fine grain (0.5 μS workload)
No free lunches

Threads do busy waiting (nonblocking synchronisations)

We need to waste some additional thread to solve MPMC case

those thread do busy waiting as well, thus consumes real resources
Pattern composition

- **C++ STL-like implementation**
  - used to generatively compile skeletons into streaming networks
  - fully memory barrier free implementation

- **High-level pattern compose with `;` and `{ }**
  - their implementation as parametric streaming networks (graphs)
  - performance can be optimised as in streaming graphs (network of queues)
Patterns, and they comp. implementation

farm

pipe

farm{ pipe }  

farm ; farm

D&C = farm + wrap

any variation of them requiring additional synch ...
Example: ff-allocator

faster than posix, hoard, TBB (unpublished, but available)
Accelerator & self-offloading

- Target the parallelisation of legacy code
  - No need to redesign the application
  - Local intervention in the code
- Transform loops and D&C in streaming then offload them into dynamically created (skeletal) software accelerators using spare cores
- Experimented with complex codes
  - Parallel C4.5 developed in 3 days
  - Stochkit in few days, here in Edinburgh (even if ...)
Self-offloading example

// Original code
#define N 1024
long A[N][N],B[N][N],C[N][N];
int main() {
    // < init A,B,C>
    for(int i=0;i<N;++i) {
        for(int j=0;j<N;++j) {
            int _C=0;
            for(int k=0;k<N;++k)
                _C += A[i][k]*B[k][j];
            C[i][j]=_C;
        }
    }
    // Here join
}

#define N1 0 2 4
long A[N][N],B[N][N],C[N][N];
int main() {
    // < init A,B,C>
    for(int i=0;i<N;++i) {
        for(int j=0;j<N;++j) {
            int _C=0;
            for(int k=0;k<N;++k)
                _C += A[i][k]*B[k][j];
            C[i][j]=_C;
        }
    }
    // Here join
}

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

class Worker: public ff::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;
    }
};

// FastFlow accelerated code
#define N 1024
long A[N][N],B[N][N],C[N][N];
int main() {
    // < init A,B,C>
    std :: vector<ff::ff_node *> w;
    for(int i=0;i<PAR_DEGREE;vit
        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.offload(task);
        }
    }
    farm.offload((void *)ff::FF_EOS);
    farm.wait(); // Here join
}

Self-offloading example
Nokia-QT integration
SIMD composition

see Smith Waterman app, code available online
Auto-tuning
Knobs & features (high-level)

- Static variables, static methods ...
- Dynamic allocation, copy constructors, parameter passing
  - Very often they have a little impact on sequential code and huge impact on multithread code
  - Example StochKit: Changing from value to value-return a single method call bring improved 30% the performance of sequential and 10x the performance of parallel
Knobs & features (run-time support)

Fastflow can:

- work in fully nonblocking fashion
  - either CAS-free o with spin-locks
- work combined blocking and non-blocking fashion
  - thread can move from busy waiting to blocked (and vice-versa)
Knobs & features (low-level)

- Graph-level rewriting
  - Emitter and collector can be collapsed
  - Emitter can behave as a worker
  - Collector can be removed
  - Worker over-provisioning
  - Worker number can be dynamically changed (ongoing)
  - Pipeline stages can be collapsed
- Queues
  - bound(n), unbounded
Knobs & features (allocator)

- Type of allocator (perf)
- Number of allocators (perf, mem)
- Allocator network queue size (perf, mem)
- Number of chunks for bucket, bucket size (perf, mem)

Chunks locality
- with respect to threads
- with respect to successive allocations
- with respect to physical memory partition (NUMA)
- with respect to usage pattern and skeleton

- Padding and false sharing
Applications
Applications

Micro-benchmarks & Testing Units
- N-Queens, Simple Mandelbrot, Nokia's QT Mandelbrot, Cholesky decomposition, parsec (on going right now)

Smith-Waterman

C4.5 (data mining)

FP-growth (data mining)

pbzip2 (parallel version of bzip2 compressor)

StochKit (toolkit for montecarlo simulations)
Smith-Waterman testbed

Each query sequence (protein) is aligned against the whole protein DB

- E.g. Compare unknown sequence against a DB of known sequences

SWPS3 implementation exploits POSIX processes and pipes

- Faster than POSIX threads + locks

Results

UniProtKB
Swiss-Prot
471472 sequences
167326533 amino-acids

Query sequence (e.g. GAATTC...)

Shared memory (read-only)
Smith Waterman (5-2k gap penalty)
C4.5 (YaDT-FF) on dual quad-core

Divide&Conquer, heavily unbalanced, wide range of computation grain

Unpublished (submitted, TR available) - Porting time from sequential: 3 days
StochKit-FF on eight dual-core

Complete framework for Stochastic Sims

MapReduce on streams with “selective memory” data aligment

Unpublished (submitted, TR available) - Porting time from sequential: 5 days

Wednesday, May 12, 2010
THANK YOU! QUESTIONS?

http://sourceforge.net/projects/mc-fastflow/

started in Nov ’09, at today 1200 software downloads