High-level parallel programming: (few) ideas for challenges in formal methods

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

- Multi-core and many-core with efficiency in mind
  - Memory is the problem. Focus on it.
- Low-level parallel programming. Locks, CAS, send/receive, ...
  - Mutual exclusion & Producer Consumer
  - Producer Consumer, the FastFlow way
- High-level programming & FastFlow
  - Architecture and implementation
  - Techniques and performance
- In the light of formal verification (as far I’ll succeed ...)
  - allocator
  - self-offloading and software acceleration technique
Performances: motivations
(all of them from real industrial applications)

Throughput

- High-frequency trading (e.g. ION trading)
  - 1 ms of advantage on sell/buy data stream may generate 1 MEuro
- Deep packet inspection on fast networks 1-10 GB/s (e.g. IBM, CISCO)
  - analysis 10 packet per ms
- High-throughput processes (e.g. devstudio.it, adobe, ...)
  - rendering and printing 50 MPixels/inch, encoding/decoding, real-time encryption, ...
- Multimedia streams, surveillance camera, games, ...

Latency

- Industrial processes control (e.g. Siemens)
E.g. task farm with POSIX lock/unlock
Lock vs CAS at fine grain (0.5 μS)
Programming at the low-level
Scatter, computer, then gather
Scatter, computer, then gather
Scatter, computer, then gather
Scatter, computer, then gather
Scatter, computer, then gather
```c
#include <stdio.h> #include "mpi.h" 
define MAXPROC 8 /* Max number of processes */ define NAMELEN 80 /* Max length of machine name */ define LENGTH 24 /* Length of send buffer is divisible by 2, 4, 6 and 8 */

main(int argc, char* argv[]) {
  int i, j, np, me;
  const int nametag  = 42;    /* Tag value for sending name */
  const int datatag  = 43;    /* Tag value for sending data */
  const int root = 0;         /* Root process in scatter */
  MPI_Status status;          /* Status object for receive */
  char myname[NAMELEN];             /* Local host name string */
  char hostname[MAXPROC][NAMELEN];  /* Received host names */
  int x[LENGTH];        /* Send buffer */
  int y[LENGTH];        /* Receive buffer */

  MPI_Init(&argc, &argv);                /* Initialize MPI */
  MPI_Comm_size(MPI_COMM_WORLD, &np);    /* Get nr of processes */
  MPI_Comm_rank(MPI_COMM_WORLD, &me);    /* Get own identifier */
  gethostname(&myname, NAMELEN);        /* Get host name */

  if (me == 0) {    /* Process 0 does this */
    /* Initialize the array x with values 0 .. LENGTH-1 */
    for (i=0; i<LENGTH; i++) {
      x[i] = i;
    }
    /* Check that we have an even number of processes and at most MAXPROC */
    if (np>MAXPROC || np%2 != 0) {
      printf("You have to use an even number of processes (at most %d)\n",

MAXPROC);
      MPI_Finalize();
      exit(0);
    }
    printf("Process %d on host %s is distributing array x to all %d processes\n", 
      me, myname, np);
    /* Scatter the array x to all processes, place it in y */
    MPI_Scatter(&x, LENGTH/np, MPI_INT, &y, LENGTH/np, MPI_INT, root, 
      MPI_COMM_WORLD);
  }
  else { /* all other processes do this */
    /* Receive the scattered array from process 0, place it in array y */
    MPI_Scatter(&x, LENGTH/np, MPI_INT, &y, LENGTH/np, MPI_INT, root, 
      MPI_COMM_WORLD);
    /* Send own name back to process 0 */
    MPI_Send (&myname, NAMELEN, MPI_CHAR, 0, nametag, MPI_COMM_WORLD);
    /* Send the received array back to process 0 */
    MPI_Send (&y, LENGTH/np, MPI_INT, 0, datatag, MPI_COMM_WORLD);
  }
  MPI_Finalize();
  exit(0);
}

Monday, July 4, 2011
Additional multicore-specific issues
[< 2004] Shared Font-Side Bus (Centralized Snooping)
[2005] Dual Independent Buses (Centralized Snoopin)
[2007] Dedicated High-Speed Interconnects (Centralized Snooping)
[2007] Dedicated High-Speed Interconnects (Centralized Snooping)
[2009] QuickPath
(MESI-F Directory Coherence)
This and next generation multi-cores

- Exploit cache coherence
  - and it is likely to happen also in the next future

- Memory fences are expensive
  - Increasing core count will make it worse
  - Atomic operations do not solve the problem (still fences)

- Fine-grained parallelism appear hard to achieve
  - I/O bound problems, High-throughput, Streaming, Irregular DP problems
  - Automatic and assisted parallelization
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), ...

**Example Diagram**

- Thread 1: write(A,3)
- Thread 2: write(A,1) read(A,?)

Monday, July 4, 2011
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
Bakery (Lamport 1976)

```java
class Bakery implements Lock {
    boolean[] flag;
    Label[] label;
    public Bakery (int n) {
        flag = new boolean[n];
        label = new Label[n];
        for (int i = 0; i < n; i++) {
            flag[i] = false; label[i] = 0;
        }
    }
    public void lock() {
        int i = ThreadID.get();
        flag[i] = true;
        label[i] = max(label[0], ..., label[n-1]) + 1;
        while ((3k != i) | (flag[k] & (label[k], k) <= (label[i], i))) {};
    }
    public void unlock() {
        flag[ThreadID.get()] = false;
    }
}
```

Figure 2.9 The Bakery lock algorithm.

- Works for n threads, require SC (or PRAM Consistency)
- Deadlock-free, fair (first come first served), O(n)
Are classic mutex working on a x86?

No!

try them, they are going to fail half of the times

So, what can we do?

“transactional” operations (CAS, LL/SC)

extend the “register” model with “transactional” operations (CAS)

- Compare-And-Swap, Test-And-Set, Load-Linked-Store-Conditional

what do they do?

- execute a read AND a write as an atomic operation
- acts a memory fences, all in-flight operations are committed before proceeding
Lock with CAS? Easy job.

```c
volatile int lock = 0;
void Critical() {
    while (TestAndSet(&lock) == 1); // acquire lock
    critical section //only one thread can be in this section at a time
    lock = 0 // release lock
}
```

Atomic operations are memory fences
- each atomic operation requires the reconciliation of caches
- significant effect on performance
Can we avoid locks?

Yes, in many ways using CAS (under relaxed memory models)

- actually building concurrent data structures accessed via CAS
- they perform better than locks-based, but still they fence the memory

and what about lock-free, CAS-free?

- Mutex cannot, Producer Consumer can be done
  - also under some relaxed memory model, not all of them, however
  - notice that Producer Consumer is inherently weaker with respect to Mutex because it does requires the cooperation of partners whereas Mutex is required to be deadlock-free
Lamport & FastFlow FIFO queues

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 FIFO
1983

---

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
Finally, FastFlow SPSC queues

Lamport FIFO

```c
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[head];
    tail = NEXT(tail);
    return 0;
}
```

FastFlow FIFO

```c
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;
}
```

(WMB)
Finally, FastFlow SPSC 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;
}
```

Lamport FIFO

```
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;
}
```

FastFlow FIFO

```
WMB enforce store ordering on successive cells/indexes on non-TSO.
Also it enforces transitivity in pointer traversal.
```
Lock-free and CAS-free (fence-free)

Single-Producer-Single-Consumer FIFO queues
- Lamport et al. 1983 Trans. PLS (Sequential consistency only - passive)
- Higham and Kavalsh. P1C1 (Rel. Cons. (e.g. TSO)+proof - passive)
- Giacomoni et al. 2008 PPoPP (TSO + cache slipping - passive)

Multiple-Producers-Multiple-Consumers FIFO queues
- with CAS (two of them) - Michael and Scott (PODC96)
  - Also implemented in FastFlow, require deferred reclamation (expensive) to avoid ABA problem
- without CAS - passive ➞ Cannot be done
- without CAS - active ➞ FastFlow

We now know that augmenting the picture with locks will be “useless” ....
High-level patterns & FastFlow
Pattern-based approach: 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
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
FastFlow: architecture

Efficient applications for multicore and manycore
Smith-Waterman, N-queens, QT, C4.5, FP-Growth, ...

Autonomic
Behav.Skeletons

Simulation
Montecarlo

Accelerator
self-offloading

Streamlining networks patterns
Skeletons: Pipeline, farm, D&C, ...

Arbitrary streamlining networks (building blocks)
Lock-free SPSC, SPMC, MPSC, MPMC queues

Simple streamlining 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

Monday, July 4, 2011
E.g. farm (a.k.a. master-worker)

---

Model \texttt{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
Medium grain (5 µS workload)
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

\[
\text{farm} \\
\text{pipe} \\
\text{farm\{ pipe \}} \\
\text{farm ; farm} \\
\text{D&C = farm + wrap} \\
\text{any variation of them requiring additional synch ...}
\]
Many open problems
Many open problems

1) Mechanisms e concurrency theory
   - new queues and data containers, new allocation techniques, ...
   - cc-NUMA: mapping tools; smart-network support (RDMA)

2) Formal Quantitative
   - performance analysis, optimisation, ...

3) Formal Qualitative
   - correctness, protocol proofs, ...

4) Design and tools
   - language evolution, compiler evolution, new features, meta-programming technique evolution, staged compilation, adaptive support
Example: FF-allocator
Example: FF-allocator
Example: FF-allocator

The graph is now cyclic (with bound queues)

- faster than posix, hoard, TBB
- unpublished, but available on sourceforge

FF allocator
Possible solutions

- Use unbound queue to “break” cyclic dependencies
  - unbound queue is slower than bound queue

- Currently
  - generate streaming network by growing a graph (via C++ class/templates)
  - turn bound queue into unbound in case cycles may appear

- However
  - patterns can be extended by the programmer (using standard OO)
  - correctness is not guaranteed (unless using all unbound queues)
  - The exploitation of unbound queue is suboptimal
    - break the graph into DAGs connected by an unbound queue
Is this complexity worth?

From performance viewpoint, yes

- Core-to-core synchronisation latency
  - less than 20 clock cycles
  - real speedup achieved even synchronising every 10 ns on a standard core2 @ 2.5Ghz
  - a single CAS (atomic op) o cache miss is an order of magnitude more expensive

- Throughput
  - the synchronisation itself does not introduce additional cache misses
  - depend on access patterns, but anyway close to the theoretical limit

- Faster than TBB, OpenMP, Cilk on all applications we tested

From design viewpoint

- we achieved the parallelisation of third party complex legacy codes in few days
- C4.5, k-means,
Accelerator & self-offloading

- Target the parallelisation of legacy code
  - No need to redesign the application
  - Local intervention in the code

Variable streamization (i.e. dynamic privatization onto a stream)

- Transform loops and D&C in streaming then offload them into dynamically created (pattern-based) software accelerators using spare cores
- More powerful than expansion. Also do-across cycles can be managed

```plaintext
Original

```...for (i=0; i<N;++i) {
    temp=A[i]+2;
    B[i]=2*temp;
}...

Privatization

```...for (i=0;i<N;++i) {
    private temp=A[i]+2;
    B[i]=2*temp;
}...

Expansion

```...for (i=0;i<N;++i) {
    temp[i]=A[i]+2;
    B[i]=2*temp[i];
}...
```
Self-offloading example

// FastFlow accelerated 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)
        _C += A[i][k] * B[k][j];
    C[i][j] = _C;

// 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;
Is correctness guaranteed?

- **Lock-free and fence-free mechanism correctness**
  - Is your machine TSO? Do you need enforce WriteBarriers on pointer traversal?
  - Is the dynamic memory allocation suffering from ABA problem?
  - Proving correctness require to model write and read

- **Offloading, interesting correctness issues**
  - Pointers should be managed as values (with possible read-only aliasing)
  - Data-hazards analysis \((w \rightarrow w, r \rightarrow w, w \rightarrow r)\)

- **Huge demand for static and dynamic analysis tool**
  - But not just theoretical tools ...
No conclusions! We just started. Thank you.

FastFlow: an open source project

- Many contributes from the open source community worldwide
- Over 25K website visits, 6K downloads form 120 different countries in 1 year and half

Currently supported by

- HPC advisory board academic award 2011 (announced at Supercomputing 2011)
- ParaPhrase STREP (FWP7 - 3.5 MEuro, starting Oct 2011, 3 years)
- BioBITS (Italian Project, Regione Piemonte, 2009-2011)

Many existing benchmarks and applications

- C4.5, k-means, pbzip-ff, smith-waterman, Stochkit-ff, Parallel MonteCarlo, N-queens ...
- Many on my laptop, just ask if you interested
Ideas and...