Performance and productivity in the multi-core era
challenges in software engineering and formal methods

Marco Aldinucci - ParaPhrase project
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 Research, Ireland

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

✴ Paraphrase
✴ Issues for multicore era
  ✦ Mostly focusing on shared memory hardware
✴ High-level patterns & FastFlow
  ✦ A productive and efficient methodology in the small
  ✦ A new programming model that can scale in the large
✴ The challenge for Formal methods.
  ✦ Questions and thoughts in this context (in open order)
  ✦ How our work might benefit or influence on formal method community
ParaPhrase: Parallel Patterns for Adaptive Heterogeneous Multicore Systems

9 partners from five countries

http://www.paraphrase-ict.eu/

Budget
- Total Cost: € 3.54 million
- Funding: Seventh Framework Programme (FP7) STREP project (contract no: 288570)
- Project start date: 1 October 2011 - Duration: 36 months
## Paraphrase partners

<table>
<thead>
<tr>
<th>Project partners</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>University of St Andrews</td>
<td>UK</td>
</tr>
<tr>
<td>Robert Gordon University</td>
<td>UK</td>
</tr>
<tr>
<td>Mellanox Technologies Ltd.</td>
<td>Israel</td>
</tr>
<tr>
<td>Software Competence Center Hagenberg</td>
<td>Austria</td>
</tr>
<tr>
<td>Erlang Solutions Ltd</td>
<td>UK</td>
</tr>
<tr>
<td>University of Pisa</td>
<td>Italy</td>
</tr>
<tr>
<td>High Performance Computing Center Stuttgart</td>
<td>Germany</td>
</tr>
<tr>
<td>University of Torino</td>
<td>Italy</td>
</tr>
<tr>
<td>Queen’s University Belfast</td>
<td>UK</td>
</tr>
</tbody>
</table>
Paraphrase

Issues for multicore era
- Mostly focusing on shared memory hardware

High-level patterns & FastFlow
- A productive and efficient methodology in the small
- A new programming model that can scale in the large

The challenge for Formal methods.
- Questions and thoughts in this context (in open order)
- How our work might benefit or influence on formal method community
Small scale

- Multi-core now ubiquitous
  - Desktops, laptops, mobile phones, ...
  - Accelerators and GPGPUs

- Shared-memory architectures
  - cache-coherent, weakly memory consistent
  - also GPUs are shared-shared-memory
    - memory hierarchy and caches in charge of the programmers
The large scale challenge

“Ultimately, developers should start thinking about tens, hundreds, and thousands of cores now in their algorithmic development and deployment pipeline.”

Anwar Ghuloum, Principal Engineer, Intel Microprocessor Technology Lab

“The dilemma is that a large percentage of mission-critical enterprise applications will not "automagically" run faster on multi-core servers. In fact, many will actually run slower. We must make it as easy as possible for applications programmers to exploit the latest developments in multi-core/many-core architectures, while still making it easy to target future (and perhaps unanticipated) hardware developments.”

Patrick Leonard, Vice President for Product Development
Rogue Wave Software
Large scale: hints

- Increasing scalability requires to decrease concurrency grain
- Programming systems should be designed to support fast data movement and enforce locality
  - It is not about Flops, it is about data movement
- Novel computing models are needed
  - A computer language is not a computing model. A library is not a computing model.
Programming issues

- We can muddle through on 2-8 cores
  - maybe even 16
  - modified sequential code may work
  - we may be able to use multiple programs to soak up cores
  - BUT larger systems are much more challenging

- Fundamentally, programmers must learn to “think parallel”
  - this requires new high-level programming constructs
  - you cannot program effectively while worrying about deadlocks etc
    - they must be eliminated from the design!
  - you cannot program effectively while fiddling with communication etc
    - this needs to be packaged/abstracted!
Efficiency challenge for multi-core (basic mechanisms)
Micro-benchmarks: farm of tasks

Used to implement: parameter sweeping, master-worker, etc.

```c
void Emitter () { 
    for ( i =0; i <streamLen;++i){
        task = create_task ();
        queue=SELECT_WORKER_QUEUE();
        queue ->PUSH(task);
    }
}

void Worker() {
    while (!end_of_stream){
        myqueue ->POP(&task);
        do_work(task) ;
    }
}

int main () {
    spawn_thread( Emitter ) ;
    for ( i =0; i <nworkers;++i){
        spawn_thread(Worker);
    }
    wait_end () ;
}
```

![Diagram of Emitter and Worker](image.png)
Task farm with POSIX lock/unlock

average execution time per task

Number of Cores

Speedup

Ideal 50 μS 5 μS 0.5 μS
Can we avoid locks?

- Under relaxed memory models, using CAS/atomic ops
  - “lock-free” data structures
  - they perform better than lock-based
  - they fence the memory and pay cache coherency reconciliation overhead
CompareAndSwap queues

![Diagram showing the comparison and swap operation on queues]

The graph illustrates the speedup as a function of the number of cores, with three different latency values: 50 μS, 5 μS, and 0.5 μS. The ideal speedup is shown as a straight line, while the actual speedup for each latency value is represented by different colored markers.

- **Ideal Speedup**
  - Number of Cores: 2, Speedup: 1
  - Number of Cores: 3, Speedup: 2
  - Number of Cores: 4, Speedup: 4
  - Number of Cores: 5, Speedup: 5
  - Number of Cores: 6, Speedup: 6
  - Number of Cores: 7, Speedup: 7
  - Number of Cores: 8, Speedup: 8

- **50 μS**
  - Number of Cores: 2, Speedup: 0.5
  - Number of Cores: 3, Speedup: 1
  - Number of Cores: 4, Speedup: 2
  - Number of Cores: 5, Speedup: 3
  - Number of Cores: 6, Speedup: 4
  - Number of Cores: 7, Speedup: 5
  - Number of Cores: 8, Speedup: 6

- **5 μS**
  - Number of Cores: 2, Speedup: 0.1
  - Number of Cores: 3, Speedup: 0.5
  - Number of Cores: 4, Speedup: 1
  - Number of Cores: 5, Speedup: 1.5
  - Number of Cores: 6, Speedup: 2
  - Number of Cores: 7, Speedup: 2.5
  - Number of Cores: 8, Speedup: 3

- **0.5 μS**
  - Number of Cores: 2, Speedup: 0.01
  - Number of Cores: 3, Speedup: 0.05
  - Number of Cores: 4, Speedup: 0.1
  - Number of Cores: 5, Speedup: 0.15
  - Number of Cores: 6, Speedup: 0.2
  - Number of Cores: 7, Speedup: 0.25
  - Number of Cores: 8, Speedup: 0.3
Lock vs CAS at fine grain (0.5 μS)
Re-starting from the basics

✴ Reducing the problem to the bare bones
✦ Producer-Consumer model (streaming)
✦ Directly control thread blocking using non-blocking synchronizations
✦ Directly design the “data channel”
  • Having clear how data move in the whole memory hierarchy

✴ Restarting from the FIFO queue
Interaction models: theoretical background

✴ 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
FastFlow SPSC queues

Lamport FIFO - 1983

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

FastFlow FIFO - derived from PICI 1997

```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;
}
```
FastFlow SPSC queues

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;
}
```

FastFlow FIFO - derived from PICI 1997

```
push_nonblocking(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;
}
```

- head and tail are mutually invalidated by producer and consumer
- 1 cache miss every push and pop (at least)
- producer read/write head
- consumer read/write tail
- no misses
- excluding “true” deps
- extended domain on void *

Paraphrase
Recall: 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)
Memory Consistency

Processors:
- x86, x86_64: Total Store Order
- PowerPC: Weak Ordering (PowerEN?)
- ARM Cortex: Weak Ordering
- Alpha: Release Consistency

Any Sequential Consistency?
- No! It is not efficient
Mem Consistency: Seq. Consistency

Can both “if” be evaluated to TRUE?
- Ideally NO, under Sequential Consistency NO
- Under more relaxed models? Not guaranteed ...

Java memory model doesn’t expose this complexity
- at the price of performance
FastFlow SPSC queues

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

Lamport FIFO - 1983

```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;
}
```

FastFlow FIFO - derived from PICI 1997

(WMB)

For any model weaker than TSO
FastFlow SPSC queues

Lamport FIFO - 1983

FastFlow FIFO - derived from PICI 1997
Related Work: Lock-free and CAS-free

**Single-Producer-Single-Consumer FIFO queues**
- Lamport et al. 1983 Trans. PLS (Sequential consistency only - passive)
- Higham and Kavalsh. 1997 ISPAN (PICI - 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/hazard pointers to avoid ABA problem
- without CAS - passive ➞ Cannot be done
- without CAS - active ➞ FastFlow

*Extending the taxonomy with locking algorithms is clearly useless ....*
Paraphrase

Issues for multicore era
- Mostly focusing on shared memory hardware

High-level patterns & FastFlow
- A productive **and** efficient methodology in the small
- A new programming model that can scale in the large

The challenge for Formal methods.
- Questions and thoughts in this context (in open order)
- How our work might benefit or influence on formal method community
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

High-level programming
- Lock-free/fence-free non-blocking synchronisations
- C++ STL-like implementation
E.g. farm (a.k.a. master-worker)

- **Common paradigm** (compute something)
  - 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
Common paradigm (compute something)

- 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
The code

```cpp
using namespace ff;

class Worker: public FF_node {
    public:
        void *svc(void *task) {
            int *t = (int *)task;
            std::cout << "Worker " << FF_node::get_my_id() << " received task " << *t << "\n";
            return task;
        }
    }

class Collector: public FF_node {
    public:
        void *svc(void *task) {
            int *t = (int *)task;
            if (*t == -1) return NULL;
            return task;
        }
    }

class Emitter: public FF_node {
    public:
        Emitter(int max_task):ntask(max_task) {
            if (farm.run_and_wait_end()) {
                error("running farm\n");
                return -1;
            }
            std::cerr << "DONE, time= " << farm.ffTime() << " (ms)\n";
            farm.ffStats(std::cerr);
            return 0;
        }
    private:
        int ntask;
    }

int main(int argc, char * argv[]) {
    if (argc<3) {
        std::cerr << "use: "
                   << argv[0]
                   << " nworkers streamlen\n";
        return -1;
    }
    int nworkers=atoi(argv[1]);
    int streamlen=atoi(argv[2]);
    if (!nworkers || !streamlen) {
        std::cerr << "Wrong parameters values\n";
        return -1;
    }
    FF_farm<> farm; // farm object
    Emitter E(streamlen);
    farm.add_emitter(&E);
    std::vector<FF_node *> w;
    for(int i=0;i<nworkers;++i) w.push_back(new Worker);
    farm.add_workers(w); // add all workers to the farm
    Collector C;
    farm.add_collector(&C);
    std::cerr << "DONE, time= " << farm.ffTime() << " (ms)\n";
    farm.ffStats(std::cerr);
    return 0;
}
```
Medium grain (5 μS workload)
Edge-preserving denoiser. Live demo!

Lena* with 90% of noise is restored in 4 seconds
Next best result in literature is about 180 seconds
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 ...
The challenge for formal methods (in this specific regard)

- Paraphrase
- Issues for multicore era
  - Mostly focusing on shared memory hardware
- High-level patterns & FastFlow
  - A productive **and** efficient methodology in the small
  - A new programming model that can scale in the large

The challenge for Formal methods.
- Questions and thoughts in this context (in open order)
- How our work might benefit from or influence formal method community
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
From patterns to metal

★ Graphs can be used as compilation (intermediate) layer

★ Is this good or a bad news?
  ◆ Graphs well understood
    • Thread pinning, thread affinity, addresses locality, concurrent code optimisation, ..., can be modelled as graph
    • Traditional tool of formal method community
  ◆ Everything concerning “graph” is complex by its very nature
    • At least for myself
 SHR Inference rules...in one slide

Parallel
\[ \Gamma \vdash G_1 \xrightarrow{\Lambda} \Phi \vdash G_2, \quad \Gamma' \vdash G_1' \xrightarrow{\Lambda'} \Phi' \vdash G_2' \]
\[ (\Gamma \cup \Phi) \cap (\Gamma' \cup \Phi') = \emptyset \]
\[ \Gamma, \Gamma' \vdash G_1|G_1' \xrightarrow{\Lambda \cup \Lambda'} \Phi, \Phi' \vdash G_2|G_2' \]

Restrict
\[ \Gamma, x \vdash G_1 \xrightarrow{\Lambda} \Gamma, x \vdash G_2, \quad \Lambda(x) = \epsilon \lor \Lambda(x) = \tau \]
\[ \Gamma \vdash \nu x \ G_1 \xrightarrow{\Lambda \setminus \{x\}} \Gamma \vdash \nu x \ G_2 \]

Merge
\[ \Gamma, x, y \vdash G_1 \xrightarrow{\Lambda} \Phi \vdash G_2 \]
\[ \Gamma[x/y] \vdash G_1[x/y] \xrightarrow{\Lambda'(x, \tau)} \Phi[x/y] \vdash \nu U \ G_2[x/y][\rho] \]

The system can do whatever disjoint subsystems do.

The system can do any transition not requiring any synchronisations on restricted node.

x and y can be fused provided that they perform compatible synchronisation actions.

The system can do...

E. Tuosto and myself. Towards a Formal Semantics for Autonomic Components. Sensoria and CoreGRID FP6
Programming model

- Producer-Consumer and mutual exclusion have a different pragmatics: cooperation vs competition
- FastFlow advocates Producer-Consumer
  - Synchronisation via message-passing, data exchange via both message-passing and shared memory
  - Allow mutual exclusion on business code under the full responsibility of the programmer because this is not efficient at fine grain
    - They are additional bi-directional arrows in the graph
- Can be complemented with transactional memory
  - Competition
  - IBM BlueGene/Q implements at hardware
    - LL/SC with versioning - Efficient - directly affect FastFlow model
  - Can be statically checked as begin and end of transaction are well defined
A simple proof system for lock-free concurrency

Luís Caires, Carla Ferreira, and António Ravara

Operational Semantics
A transition relation

Transition system - memory manipulation

\[
E_{s, h} = v \\
(s, h, x = E) \rightarrow ((s[x \leftarrow v], h, \text{skip}))
\]

(assignment)

\[
E_{s, h} = v_1 \land h(v_1) = (v, _) \\
(s, h, x = +E) \rightarrow ((s[x \leftarrow v], h, \text{skip}))
\]

(load)

\[
E_{s, h} = v_2 \land E_{s, h} = v_1 \land h(v_1) = (_, b) \\
(s, h, +E_1 = E_2) \rightarrow (s, h[v_1 \leftarrow (v_2, b == 1 ? 2 : b)], \text{skip})
\]

(store)

Transition system - transaction-like primitives

\[
\text{LoadLink (LL)} (loc, v_{old}, v_{new}) \text{ atomically exchange the value in the memory location } loc \text{ with value } v_{new}, \text{ provided the current value in } loc \text{ is } v_{old}.
\]

LoadLink (LL) / StoreConditional (SC)

The pair of primitives have a transaction-like semantics:

- \( \text{LL}(loc) \) returns the address of the memory location \( loc \)
  - May be interpreted as the transaction start
- \( \text{SC}(loc, v) \) tries to store \( v \) in the memory location \( loc \):
  - if the location has not been written by any concurrent thread since \( \text{LL} \), \( v \) is written atomically and the instruction returns true; otherwise, it returns false, not writing the value
  - May be interpreted as the transaction commit, returning either “success” or “abort”
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,
Example: Allocator
ff-allocator (unpublished)

- Build on top of lock-free queues
  - based on SLAB
  - target streaming patterns (producer allocate, consumer deallocate)

- Streaming-oriented
  - define a graph that coupled with application make it possible to send back memory chunks from freeing thread to allocating thread

- General purpose
  - as before but the graph is dynamically build during usage
    - slightly more expensive

- Fast
  - always faster than libc and hoard
  - often faster than TBB (scalable allocator)
Example: FF-allocator
Example: FF-allocator

Diagram showing the allocation and deallocation processes involving new and delete operations for objects A, B, and C within an OS allocator.
Example: FF-allocator

- Faster than posix, hoard, TBB
- Unpublished, but available on sourceforge

The graph is now cyclic (with bound queues)
Possible solutions

✴ Currently
✦ Generate streaming network by growing a graph (via C++ class/templates)
✦ Use unbound queue to “break” cyclic dependencies
  • unbound queue is slower than bound queue

✴ 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

✴ Possible developments
✦ Modelling the problem with Session or Global types (w Padovani/Dezani)
✦ Modelling the problem with Objects&Artifacts (w Baldoni)
On programming model

- Shared memory or message passing (data) + message passing (synchronisations)
  - Graphs exactly describes the (true) data dependency pattern
  - Additional synchronisations can be added (e.g. locks) in the user code

- Queue can pass pointers or data
  - Passing data means copying it. A proper usage of allocator might significantly enhance locality. Copying non strictly needed data is overhead.
  - The balance depends on the application. Should be studied more.
On programming model

- Shared memory or message passing (data) + message passing (synchronisations)
  - Graphs exactly describes the (true) data dependency pattern
  - Additional synchronisations can be added (e.g. locks) in the user code

- Queue can pass pointers or data
  - Passing data means copying it. A proper usage of allocator might significantly enhance locality. Copying non strictly needed data is overhead.

- The balance depends on the application. Should be studied more.
On programming model

- **Shared memory or message passing (data)**
  + message passing (synchronisations)
  - Graphs exactly describes the (true) data dependency pattern
  - Additional synchronisations can be added (e.g. locks) in the user code

- **Queue can pass pointers or data**
  - Passing data means copying it. A proper usage of allocator might significantly enhance locality. Copying non strictly needed data is overhead.
  - The balance depends on the application. Should be studied more.
Example: 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

---

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

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

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

1 // Original code
2 #define N 1024
3 long A[N][N], B[N][N], C[N][N];
4 int main() {
5    // < init A, B, C>
6    for(int i=0;i<N;++i) {
7        for(int j=0;j<N;++j) {
8            int _C=0;
9            for(int k=0;k<N;++k)
10                _C += A[i][k]*B[k][j];
11            C[i][j]=_C;
12       }
13    }
14 }
15 }

16 // FastFlow accelerated code
17 #define N 1024
18 long A[N][N], B[N][N], C[N][N];
19 int main() {
20    // < init A, B, C>
21    ff::ff_farm<> farm(true /* accel */);
22    std::vector<ff::ff_node *> w;
23    for(int i=0;i<PAR_DEGREE;++i)
24        w.push_back(new Worker);
25    farm.add_workers(w);
26    farm.run_then_freeze();
27
28    for (int i=0;i<N;i++) {
29        for(int j=0;j<N;++j) {
30            task_t * task = new task_t(i,j);
31            farm.offload(task);
32        }
33    }
34    farm.offload((void *)ff::FF_EOS);
35    farm.wait(); // Here join
36 }

37 // Includes
38 struct task_t {
39    task_t(int int i, int j):i(i),j(j) {}
40    int i; int j;};
41
42 class Worker: public ff::ff_node {
43    public: // Offload target service
44    void * svc(void *task) {
45        task_t * t = (task_t *)task;
46    int _C=0;
47    for(int k=0;k<N;++k)
48        _C += A[t->i][k]*B[k][t->j];
49    C[t->i][t->j] = _C;
50    delete t;
51    return GO_ON;
52    }
53    void * svc(void *task) {
54        task_t * t = (task_t *)task;
55        int _C=0;
56        for(int k=0;k<N;++k)
57            _C += A[t->i][k]*B[k][t->j];
58        C[t->i][t->j] = _C;
59        delete t;
60        return GO_ON;
61    }
62};
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 ...
Good news: people use FastFlow to do their own research
A More Efficient and Type-Safe Version of FastFlow *

Zalán Szügyi and Norbert Pataki

Department of Programming Languages and Compilers, Eötvös Loránd University
Pázmány Péter sétány 1/C H-1117 Budapest, Hungary
lupin@ludens.elte.hu, patakino@elte.hu

Abstract. Nowadays, one of the most important challenges in programming is the efficient usage of multicore processors. Many new programming languages and libraries support multicore programming. FastFlow is one of the most promising multicore C++ libraries. Unfortunately, a design problem occurs in the library. One of the most important methods is pure virtual function in a base class. This method supports the communication between different threads. Although, it cannot be template function because of the virtuality, hence, the threads pass and take argument as a void* pointer. The base class is not template neither. This is not typesafe approach. We make the library more efficient and safer with the help of generative technologies.
Automatically Optimising Parallel Skeletons

Alexander James Collins

Master of Science
Computer Science
School of Informatics
University of Edinburgh
2011

Table of Contents

1 Introduction 1
   1.1 Contributions ......................................................... 2
   1.2 Structure of the Dissertation ...................................... 2

2 Background 3
   2.1 Parallel Systems and Abstractions ................................. 3
   2.2 Parallel Skeletons .................................................... 4
   2.3 FastFlow .............................................................. 6
      2.3.1 Lock-free Single-Producer Single-Consumer Queues ....... 7
      2.3.2 Skeletons ......................................................... 9
      2.3.3 Memory Allocation ............................................. 9
      2.3.4 Why Choose FastFlow? ....................................... 10
   2.4 Auto-tuning .......................................................... 10
   2.5 Statistical Techniques ............................................. 10
      2.5.1 Confidence Intervals ....................................... 11
      2.5.2 Hypothesis Testing .......................................... 11
      2.5.3 Outlier Removal ............................................. 12
      2.5.4 Principal Components Analysis ............................ 13
   2.6 Stochastic Search Techniques ..................................... 14
      2.6.1 Hill-Climbing .................................................. 14
      2.6.2 Hill-Climbing with Random Restarts ....................... 14
      2.6.3 Hill-Climbing with Adaptive Restarts ..................... 15
      2.6.4 Simulated Annealing ....................................... 15
      2.6.5 Genetic Algorithms ......................................... 16
In the large?
A computer language is not a computing model. A library is not a computing model.

- Data communication happen via both shared-memory and messages. Synchronisations are realised via message-passing (FIFO queues).
- Synchronisation are local (no barriers) and determined by high-level algorithmic patterns. Data races are identified and solved at design time.

Increasing scalability requires to decrease concurrency grain. Programming systems should be designed to support fast data movement and enforce locality.

- FastFlow: inter-core communication latency ~7-10 ns on core2 2Ghz. Better than other approaches at fine grain.
Issue for the large scale

✶ How to describe concurrency exploitation at large scale?
  ✦ Parametric patterns. QoS/performance as first-class concept. Couple data with flow-of-control beyond OO (how?)

✶ How we promote scalability “by design” and performance portability?
  ✦ Development tools. Mapping/affinity, should be automatically managed. Graphs very expressive. Graph-to-graph mapping encode semantic-preserving transformation (i.e. optimisation).

✶ Functional-style coding?
  ✦ Empirically the only way I found to write SSE/AVX.