High-level lock-less programming for multi-core

Fabio Tordini°, Marco Aldinucci°, and Massimo Torquati†
° University of Torino, Italy † University of Pisa, Italy

The Problems

Peak performance is hard to achieve on cache-coherent multi-core architectures and requires substantial programming and tuning efforts. Performance portability is even harder.

Performance is (often) not about Flops, it is about data movement.

Coarse Grain Concurrency is nearly exhausted. Programming systems should be designed to support fast data movement and enforce locality. They should be efficient at fine grain.

Non-blocking algorithms coupled with concurrent data structures can be fast but are complex to be exploited. They can be hardly composed and should be abstracted out.

High-level approaches should be used to provide effective abstractions. A computer language is not a computing model. A library is not a computing model. A litmus paper: system programmers use the techniques they advocate?

The Approach

- Dijkstra's structured programming approach (*"go-to statement considered harmful"*)
  - Are **send/receive**, **lock/unlock** and **CAS** *harmless* than **go-to**?
- Manage communications and synchronisations by way of high-level constructs
  - Skeletons & Patterns [Col89]. Exploited in several frameworks, inter alia: Intel TBB, Fastflow [ADKT12], Google's MapReduce.
  - Patterns typically used to discipline true dependencies and process-to-CPUs mapping.
- Message-passing/shared-memory is not a dichotomy. They can be coupled in a richer **programming model**, e.g. messages for synchronizations and shared-memory for data exchanges.

FastFlow: The Big Picture – http://mc-fastflow.sourceforge.net

Applications *are build using Fastflow patterns or extending them in a OO style. Synchronizations are hidden within the library.*

Applications on multicore and manycore
Efficient and portable - designed with high-level patterns

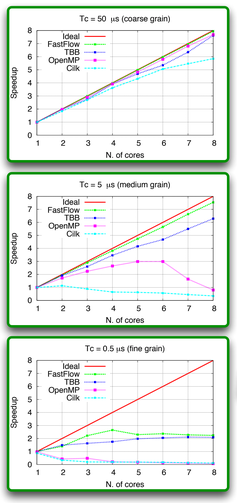
FastFlow

Streaming network patterns
Skeletons: pipeline, map farm, reduce, D&C, ...

Arbitrary streaming networks
Lock-free SPSC/MPMC queues + FF nodes

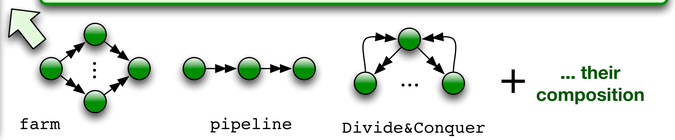
Simple streaming networks
Lock-free SPSC queues + threading model

Multicore and manycore
cc-UMA & cc-NUMA - TSO and WO consistency



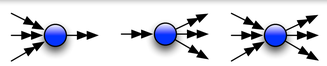
Streaming network patterns *layer provides (streaming) parallel programming patterns:* ***farm***, ***farm-with-feedback*** *(i.e. Divide&Conquer),* ***pipeline***, *and their arbitrary nesting and composition. Patterns discipline true dependencies whereas data is moved via shared memory.*

```
#include <iostream>
#include <string>
#include <ff/pipeline.hpp>
using namespace std;

class Stage1: public ff::ff_node {
private: int c;
public:
  Stage1(): c(0) {};
  void * svc(void * task) {
    string *s = NULL;
    if (c++<3) s = new string("Hello");
    return s;
  }
};

class Stage2: public ff::ff_node {
public:
  void * svc(void * task) {
    string *s = (string *) task;
    cout << s->append(" world!\n");
    delete (s);
    return GO_ON;
  }
};

int main(int argc, char * argv[]) {
  ff::ff_pipeline pipe;
  pipe.add_stage(new Stage1());
  pipe.add_stage(new Stage2());
  pipe.run_and_wait_end();
}
```



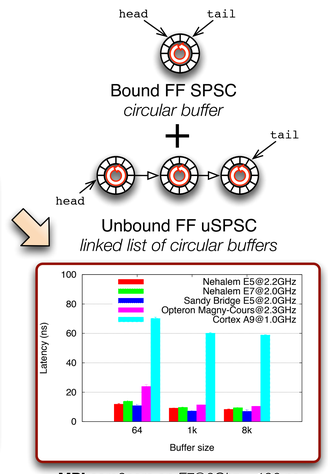
Arbitrary streaming networks *layer implements the* `ff_node`*, i.e. the building block of networks and serves as a container for business code and "mediators". Cyclic networks use uSPSC to avoid deadlocks.*

```
class ff_node {
protected:
  virtual bool push(void* data) {
    return qout->push(data);
  }
  virtual bool pop(void** data) {
    return qin->pop(data);
  }
public:
  virtual void* svc(void * task) = 0;
  virtual int   svc_init () {
    return 0;
  };
  virtual void  svc_end() {}
  ...
private:
  SPSC* qin;
  SPSC* qout;
};
```



Simple streaming networks *layer implements efficient SPSC bound and unbound wait-free queues. They requires no CAS and no fences (TSO, a WMB for WO consistency). SPSC: 2x faster than Lamport queue, uSPSC 20x faster than Michael-Scott linked-list [ADKM12].*

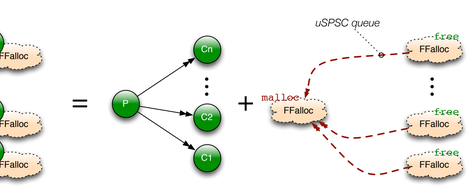
```
bool push(void* data) {
  if (buf[pwrite]==NULL) {
    WMB(); // write-memory-barrier
    buf[pwrite] = data;
    pwrite+=(pwrite+1>=size)?(1-size):1;
    return true;
  }
  return false;
}
bool pop(void** data) {
  if (buf[pread]==NULL)
    return false;
  *data = buf[pread];
  buf[pread]=NULL;
  pread+=(pread+1>=size)?(1-size):1;
  return true;
}
```

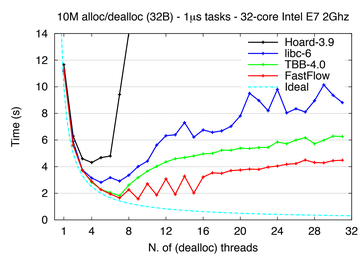


MPI *core2core on E7@2Ghz: ~190 ns*

Example: a Stream-oriented Mem Allocator Built with Fastflow

```
Producer P:
for(i=0;i<10M;i++){
  pi = malloc(rnd(size));
  *pi=...;
  dispatch_RR pi;
}

Consumer Ci:
while (pi=get())
  do_work(1µs,pi);
  free(pi);
}
```





References

ADKM12 M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati. An efficient unbounded lock-free queue for multi-core systems. In Proc. of Euro-Par, LNCS, Rhodes Island, Greece, Aug. 2012. Springer.

ADKT12 M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. Fastflow: high-level and efficient streaming on multi-core. In Programming Multi-core and Many-core Computing Systems, Parallel and Distributed Computing, chap. 13. Wiley, 2012.

Col89 M. Cole: Algorithmic Skeletons: Structured Management of Parallel Computations. Research Monographs in Par. and Distrib. Computing. Pitman, 1989.

Future Directions

A significant speed edge over state-of-the-art parallel allocators can be achieved by specializing a (relatively simple, built on top of the SLAB allocator) memory allocator with high-level patterns. The allocation technique get advantages from the **low-overhead of the run-time** (based on lock-free uSPSC) and the **knowledge of high-level semantics** (producer-consumer). We believe the approach can be improved on both directions, i.e.

Memory Affinity concerns mapping and allocation of data structures in memory. Data structures can be coupled with parallel patterns (also thanks to specialized allocation strategies). Experimentation is feasible thanks to the lock-free allocator already implemented in Fastflow.

Lock-free run-time support can be extended with more (location-aware) Multiple-Producer-Multiple-Consumer data structures and transactional primitives.