Parallel patterns, data-centric concurrency, and heterogeneous computing

Marco Aldinucci, University of Torino, Computer Science Department

HPCC keynote talk, Paris, France. Aug 20, 2014
Outline

- Parallel patterns and structured parallel programming
  - On the hierarchy of abstractions for parallel programming
- Data-centric concurrency
  - FastFlow: a data-centric run-time support for heterogeneous platforms
- Applications
  - With FastFlow and other frameworks
- ... and lot of cars
MPI is like a car, you can drive it as you like
MPI, threads, HW-CAS spinlocks, … are like cars, you can drive it as you like.
#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 np (at most %d)
", MAXPROC);
            MPI_Finalize();
            exit(0);
        }

        /* 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);

        /* Print out own portion of the scattered array */
        printf("Process %d on host %s has elements", me, myname);
        for (i=0; i<LENGTH/np; i++) {
            printf(" %d", y[i]);
        }
        printf("\n");

        /* Receive messages with hostname and the scattered data */
        /* from all other processes */
        for (i=1; i<np; i++) {
            MPI_Recv (&hostname[i], NAMELEN, MPI_CHAR, i, nametag, 
                       MPI_COMM_WORLD, &status);
            MPI_Recv (&y, LENGTH/np, MPI_INT, i, datatag, MPI_COMM_WORLD, &status);
            printf("Process %d on host %s has elements", i, hostname[i]);
            for (j=0; j<LENGTH/np; j++) {
                printf(" %d", y[j]);
            }
            printf("\n");
        }
    }

    if (me == 0) {    /* Process 0 does this */

        /* Check Sanity of the user */
        if (np>MAXPROC || np%2 != 0) {
            MPI_Finalize();
            exit(0);
        }

        /* 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);

        MPI_Finalize();
        exit(0);
    }

    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);

        MPI_Finalize();
        exit(0);
    }
}
bool push(void *const data) {
    unsigned long pw, seq;
    element_t * node;
    unsigned long bk = BACKOFF_MIN;
    do {
        pw = pwrite.load(std::memory_order_relaxed);
        node = &buf[pw & mask];
        seq = node->seq.load(std::memory_order_acquire);
        if (pw == seq) { // CAS
            if (pwrite.compare_exchange_weak(pw, pw+1, std::memory_order_relaxed))
                break;
            for(volatile unsigned i=0;i<bk;++i);
            bk <<= 1;
            bk &= BACKOFF_MAX;
        } else
            if (pw > seq) return false; // queue full
    } while(1);
    node->data = data;
    node->seq.store(seq+1,std::memory_order_release);
    return true;
}
Deadlock on GPGPU (unless nvcc -G).

A different execution model. Impossible to make any assumptions about scheduling.

Data-dependency can be managed only via lock-free algorithms.
Message-passing (e.g. MPI)  
Shared-memory (e.g. threads + mutex/CAS)

- (Can be) Efficient
- Available in almost all platforms
  - Often the only access to net, e.g. HRLS Cray XE6 Hermit: 32 x 3552 nodes = 113 664 cores
  - De-facto standards, used for decades
- Parallel primitives fully inter-waived with business code
- How compose “computing phases”, “SW modules” …
  - Parallel behaviour and data layout not explicit in the code. Primitives often do not compose.
- Often to be coupled with shared-memory for intra-node
  - e.g. MPI & Pthreads, MPI & OpenMP, PGAS-MPI & OpenCL, …
I ♥ MPI
I ♥ threads

No whiners
Unstructured programming considered harmful
Val D’Orcia, Tuscany, Italy

Designer utopia
False sharing

Reality

Irregular data

Hot spots
Loop parallelism helps

- OpenMP is great example
  - It simply works well on loops
- Not really useful for all problems
  - Tasks, Graphs, …

Cholesky (Tile 5)
Courtesy: J. Dongarra
Writing programs that scale with increasing numbers of cores should be as easy as writing programs for sequential computers.

BY KRSTE ASANOVIC, RASTISLAV BODIK, JAMES DEMMEL, TONY KEAVENY, KURT KEUTZER, JOHN KUBIATOWICZ, NELSON MORGAN, DAVID PATTERSON, KOUSHIK SEN, JOHN WAWRZYNEK, DAVID WESSEL, AND KATHERINE YELICK

A View of the Parallel Computing Landscape

Over the past 60 years, the IT industry has improved the cost-performance of sequential computing by about 100 billion times overall.

For most of the past 20 years, architects have used the rapidly increasing transistor speed and budget made possible by silicon technology advances to double per-performance every 18 months. The implicit hardware/software contract was that increased transistor count and power dissipation were OK as long as architects maintained the existing sequential programming model. This contract led to innovations that were inefficient in terms of transistors and power (such as multiple instruction issue, deep pipelines, out-of-order execution, speculative execution, and prefetching) but that increased performance while preserving the sequential programming model.

The contract worked fine until we hit the power wall a chip is able to dissipate. Figure 1 reflects this abrupt change, plotting the projected microprocessor clock rates of the International Technology Roadmap for Semiconductors in 2005 and then again just two years later. The 2005 prediction was that clock rates should have exceeded 10GHz in 2008, topping 15GHz in 2010. Note that Intel products are today far below even the conservative 2007 prediction.

After crashing into the power wall, architects were forced to find a new paradigm to sustain ever-increasing performance. The industry decided the only viable option was to replace the single power-inefficient processor with many efficient processors on the same chip. The whole microprocessor industry thus declared that its future was in parallel computing, with increasing numbers of processors, or cores, each technology generation every two years. This style of chip was labeled a multicore microprocessor. Hence, the leap to multicore is not based on a breakthrough in programming or architecture and is actually a retreat from the more difficult task of building power-efficient, high-clock-rate, single-core chips.

Many startups have sold parallel computers over the years, but all failed, as programmers accustomed to continuous improvement in sequential performance saw little need to explore parallelism. Convex, Encore, Floating Point Systems, Inmos, Kendall Square Research, and others have failed at trying to succeed in its recent dramatic shift to parallel computing. Failure could jeopardize both the IT industry and the portions of the economy that depend on rapidly improving information technology. Here, we review the issues and, as an example, describe an integrated approach we're developing at the Parallel Computing Laboratory, or Par Lab, to tackle the parallel challenge.

Over the past 60 years, the IT industry has improved the cost-performance of sequential computing by about 100 billion times overall.
And it is paramount also at the large scale

- Coarse grain concurrency is nearly exhausted
- Often, it is not about FLOPS, it is about data movement
- Programming systems should be designed to support fast data movement and enforce locality
- Variable coherency & inter-socket messaging

The Abstract Machine Model & Execution Model

- A computer language is not a programming model
  - “C++ is not scalable to exascale”
- A communication library is not a programming model
  - “MPI won’t scale to exascale”
- A unique application inventory response...
  - Should we be talking “Execution Model”?
- What is a programming model?

Thought: real issue is mapping science problem to execution model and run-time system
MPI, the current dominant programming model for parallel scientific programming, forces coders to be aware of the exact mapping of computational tasks to processors. This style has been recognised for years to increase the cognitive load on programmers, and has persisted primarily because it is expressive and delivers the best performance. [Snir et al 1998] [Gursoy and Kale 2004]

Because we anticipate a massive increase in exploitable concurrency, we believe that this model will break down in the near future, as programmers have to explicitly deal with decomposing data, mapping tasks, and performing synchronisation over thousands of processing elements. [Asanovic et al 2006]
Abstraction (D. Reed)

“To date, attempts to develop higher level programming abstractions, tools and environments for HPC have largely failed.

There are many reasons for this failure, but I believe many are rooted in our excessive focus on hardware performance measures.

By definition, the raison d’être for high-performance computing is high performance, but FLOPS need not be the only measure.

Human productivity, total cost and time to solution are equally, if not more important.”
Low-level programming models: assessment

- Efficient (can be) and widely supported
- Not likely to scale to mainstream (industrial) development
- Not compositional
- Parallel behaviour not explicit (can be hard to catch)
- At the bottom line, higher-level “mental” overlays are already there
  - In the mind of designers (i.e. data organisation, partition, data patterns, …)
Programming model: my wish list

- Should enforce to think to problems in parallel & and at high-level of abstraction
  - Clear semantics: functional and extra-functional (parallel), describing collective behaviours
  - Trading memory coherency for power, and power for parallelism should be a matter of the implementation

- Should support containment and composition
  - At large scale: clear fault model with containment & recovery

- Should integrate synchronisation/communication with scheduling
  - Weak execution model rather than per device. Multicore, GPGPUs, distributed with an unifying vision

- System programmers should use the techniques they advocate
An alternative mobility approach
Skeletons and Patterns

- From HPC community
- Started in early ’90 (M. Cole’s PhD thesis)
- Pre-defined parallel higher-order functions, exposed to programmers as constructs/lib calls

- From SW engineering community
- Started in early ‘00
- “Recipes” to handle parallelism (name, problem, algorithms, solutions, …)
Evolution of the concept

- **'90**
  - Complex patterns, no composition
  - Targeting clusters
  - Mostly libraries (run-time system)

- **'00**
  - Simple data/stream parallel patterns
  - Compositional patterns, targeting COW/NOW
  - Libraries + first compilers

- **'10**
  - Optimised, compositional building blocks
  - Targeting cluster of heterogeneous multicore + GPGPU
  - Quite complex tool chain (compiler + run-time system)
Evolution of the concept

- **'90**
  - Cole PhD thesis skeletons
  - P3L (Pisa), SkiE (QSW/Pisa)
  - SCL (Imperial college London)

- **'00**
  - Lithium/Muskel (Pisa), Skandium (INRIA)
  - Muesli (Muenster), SkeTo (Tokio)
  - OSL (Orleans), Mallba (La Laguna), Skipper

- **'10**
  - SkePu (Linkoping)
  - FastFlow (Pisa/Torino)
  - Intel/TBB/CnC (?), Microsoft/TPL (?), Google/MapReduce (?), …
Parallel Design Patterns

Decomposition (task, data)
Dependency analysis (group, tasks, order tasks, data sharing)

Organise by task (task parallelism, Divide&Conquer)
Organise by data (geometric, stencil, recursive) or data flows (pipeline, events)

Program structure (SPMD, master-worker, loop parallelism, fork/join)
Data structures (shared data, shared queues, distributed arrays)

Synchronisation, communication, process management

Problem

Parallel Design Patterns

Finding concurrency

Algorithmic space

Supporting structure

Implementation

follow, learn, use

App developer

Low-level code

Programming tools
Algorithmic skeletons

Map, Reduce, Stencil …

Divide&Conquer, Pool, MDF, …

Pipeline, farm, …

Data parallel

Task parallel

Stream parallel

Algorithmic Skeleton library

runtime support

Problem

App developer

High-level code

System developer
Structured parallel programming

Parallel Design Patterns
- methodological to compose and extend

Algorithmic Skeletons
- instantiate
- supporting structure and implementation mechanisms

Problem
- App developer

Programming tools
- Flexible high-level code

System developer
MapReduce
a success story

Innovative implementation
(even if theoretically an instance of Map+Reduce)
Patterns are natural for GPGPUs

Courtesy: J. Owen, UC Davis

Think In Parallel

- The GPU is a data-parallel processor
  - Thousands of parallel threads
  - Thousands of data elements to process
  - All data processed by the same program
  - SPMD computation model
  - Contrast with task parallelism and ILP

- Best results when you “Think Data Parallel”
  - Design your algorithm for data-parallelism
  - Understand parallel algorithmic complexity and efficiency
  - Use data-parallel algorithmic primitives as building blocks

Data-Parallel Algorithms

- Efficient algorithms require efficient building blocks
- This talk: data-parallel building blocks
  - Map
  - Gather & Scatter
  - Reduce
  - Scan
Maybe also on FPGA

Role of FPGA in FastFlow

- FPGA can act as entire FastFlow subgraph

![Diagram showing the role of FPGA in FastFlow]

Courtesy: A. Koch, TU Darmstadt
Assessment

**Separation of concerns**
- Application programmer: **what to do**
- System programmer: **how to make it in parallel**

**Inversion of control**
- Structure suggested by programmer - no idiom recognition
- Clear functional and parallel semantics

**Performance**
- Close to hand tuned code (better for not expert programmers)
- Reduced development time. Graceful tuning curve.
FastFlow (FF) and its data-centric run-time support

Parallel applications
- efficient and portable

High-level patterns
- parallel_for, parallel_forReduce, ...

Core patterns
- pipeline, farm, feedback

Building blocks
- queues, ff_node, ...

Multicore and many-core platforms
- Clusters of multicore + many-core

FastFlow

CUDA
OpenCL
TCP/IP
IB/OFED
Building blocks

non-blocking threads
(can switch to blocking at runtime by way of a native protocol)

nonblocking thread
GPU/accelerators
(OpenCL, CUDA)

process of nonblocking threads
distributed zero-copy
TCP/IP, OFED/IB, MPI (ongoing), HW/SW PGAS (ongoing)
Building blocks
Everything is SPSC (in the shared-memory)

- Enough to support producer-consumer
  - Inherently weaker w.r.t. mutual exclusion
  - Weaker execution model requirements
    - Mutex not really possible on SIMT model (GPU)
    - Mutex requires memory-fences and leverages on (expensive) cache coherency on multicore
- Deadlock is cyclic networks avoided via unbound queue (wait-free)
**Queues: performance**

- **MVAPICH (shm) ~ 190ns**
- **Faster and more scalable than CAS/test-and-set implementation**

- **Comparable with MVAPICH (distributed)**

**Shared-memory**

**Distributed**

<table>
<thead>
<tr>
<th>Message size (bytes)</th>
<th>ib_write_bw (Mb/s)</th>
<th>MPI (Mb/s)</th>
<th>FastFlow /IB (Mb/s)</th>
<th>FastFlow/ZMQ /PoIB (Mb/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>300</td>
<td>192</td>
<td>129</td>
<td>0.7</td>
</tr>
<tr>
<td>100</td>
<td>3,600</td>
<td>1,816</td>
<td>1,200</td>
<td>7.0</td>
</tr>
<tr>
<td>1,024</td>
<td>22,900</td>
<td>13,936</td>
<td>10,591</td>
<td>70.0</td>
</tr>
<tr>
<td>5,000</td>
<td>25,200</td>
<td>23,880</td>
<td>19,761</td>
<td>300.0</td>
</tr>
<tr>
<td>10,000</td>
<td>25,500</td>
<td>25,128</td>
<td>20,479</td>
<td>500.0</td>
</tr>
<tr>
<td>25,000</td>
<td>25,700</td>
<td>12,408</td>
<td>20,051</td>
<td>1,100.0</td>
</tr>
<tr>
<td>50,000</td>
<td>25,800</td>
<td>16,232</td>
<td>21,019</td>
<td>1,950.0</td>
</tr>
<tr>
<td>65,536</td>
<td>22,900</td>
<td>17,472</td>
<td>20,889</td>
<td>1,980.0</td>
</tr>
<tr>
<td>200,000</td>
<td>25,800</td>
<td>21,208</td>
<td>21,211</td>
<td>3,800.0</td>
</tr>
<tr>
<td>400,000</td>
<td>25,800</td>
<td>22,532</td>
<td>21,226</td>
<td>6,200.0</td>
</tr>
</tbody>
</table>

**TABLE 1:** Comparing throughput of different implementations of the *unidirectional bandwidth* test for several message sizes.
Streamlining patterns: fast data movements + computation-to-core pinning
  - Full control of locality and memory affinity

Fully compositional and deadlock-free (by construction)

Minimalistic run-time
  - Two patterns and one pattern-modifier: farm, pipeline and feedback
  - Configurable scheduling and gathering policies and implementations

Full expressivity
  - Enough to build higher-level abstraction: Master-worker, D&C, Map, Reduce, Pool, Stencil, …
Core patterns: farm, pipeline, feedback

+ Nonblocking/blocking, dynamic/static scheduling, E/C policy configuration, core pinning
Core patterns can be arbitrarily composed

pipe(farm(s), farm(s))

farm(pipe(s1, s2))

farm(farm(s))

pipe(farm(s), farm(s), loopback)

pipe(s1, farm(s2))
Master-worker (0.5 μS workload)
Master-worker (5 μS workload)
Master-worker (5 μS workload)
Core patterns: a programming model

✦ A low/medium-level **data-centric programming model**
  ✦ Concurrent computation modelled as a (cyclic) graph
  ✦ Nodes are parallel activities. Edges are true data dependencies
  ✦ Synchronisation are messages, data can be moved as messages or shared memory
    ✦ Can be realised with or without coherency, in shared-memory, distributed, PGAS, …

✦ Not exactly a Kahn’s net, more a CSP-actor hybrid model
  ✦ Processes are named and the data paths between processes are identified
High-level patterns

- Address application needs
  - Loop parallelism (OpenMP fashion)
    - Parallel-For, Parallel-For-Reduce
  - Data Parallelism
    - Stencil, Stencil-Reduce, Map, MapReduce (pipelined)
  - Task & Stream
    - Pool (e.g. genetic alg.), Macro-Data-Flow (e.g. linear algebra, dynamic programming, …)
    - Farm, Pipeline

- Implementation-wise, just OO extensions of composition of core patterns
High-level patterns

- Think in parallel & high-level
- Efficiency, portability, time-to-market

High-level parallel patterns
- Describing collective behaviour
- Can be: expressive, efficient, compositional
- Multicore, GPGPUs, distributed with an unifying vision

Thanks to clear semantics, can be autonomically reconfigured
- not discussed in this talk - see ParaPhrase EU STREP FP7 project

Example: parallel for

- Currently a method call on a C++11 Lambda function (loop body)
  - All other high-level patterns in the same style

- Moving to C++11 generalised attribute syntax (N2761)
  - Within REPARA EU-FP7 STREP project

```cpp
[[ff::target(ff::cpu, ff::gpu), ff::input(A), ff::output(A), ...]] for( ; ; ; ) { ... }
```
Applications

Stream: Sequence Alignment Tools
Tasks: Linear Algebra
Data: Image Filtering
System programmers should use the techniques they advocate: memory allocation
Stream: Bowtie (BT) and BWA sequence alignment tools

- Top tools for parallel DNA alignment
- Hand-tuned C/C++/SSE2 code
- Spin locks + Pthreads
- Sequences (reads) are streamed from MMIO files to workers
- Memory bound
- FastFlow master-worker
- Memory affinity, pinning, affinity scheduling (embedded in the pattern)
- BT: up to 200% speedup
- BWA: up to 10% speedup over originals

Graphs of tasks
LU and Cholesky

- Macro-Data-Flow (MDF) pattern encoding dependence DAG
  - pipeline(TaskGen, farm(TaskExec))
  - Configurable scheduling, affinity, …

- Dynamic generation of the DAG

- Comparable or faster than specialised linear algebra frameworks (e.g. PLASMA)
  - MDF is general, can be used for a wide range of applications, Dynamic Programming, Divide&Conquer, …
Data Parallelism: Two-stage restoring

- statistic detection + variational restoration
  - High quality, edge-preserving filtering
  - Much more computational demanding, not really viable without parallelism
  - Matlab on a single 256x256 image with 50% of noise requires dozen of minutes
  - Stages can be pipelined

- progressive-switching/adaptive median
- neural/bayesian networks, fuzzy, …
- variational
- statistic
Effective noise filtering (variational)
Salt-and-Pepper, Gaussian, …
Detect C++ business code CPUs only

Denoise C++ or OpenCL business code
CPUs only, GPGPUs only
CPUs+GPGPUs

FastFlow node
FastFlow stencil-reduce

FastFlow node
FastFlow farm
(on independent frames)

FastFlow node

FastFlow stencil-reduce
(on partitions of a single frame)

stream of independent < frames, outlier_mask >

Alternative deployments for each worker W

Alternative deployments

① Sequential C++ on CPU
② Parallel C++ on CPUs
③ Parallel OpenCL on CPUs
④ Parallel C++ on CPUs
⑤ Parallel OpenCL on CPUs and GPUs

using namespace ff;

template<typename DenoiserCUDA taskType, typename DenoiserCUDA mapF>
class cudaDenoiserAUTO: public Denoiser, public FFSTENCILREDUCECUDA(DenoiserCUDA taskType, DenoiserCUDA mapF, reduceF) {
public:
    cudaDenoiserAUTO(void *kernel_params_, unsigned int height, unsigned int width, bool fixed_cycles, unsigned int max_cycles, bool trace_time):
        kernel_params(kernel_params_),
        FFSTENCILREDUCECUDA(DenoiserCUDA taskType, DenoiserCUDA mapF, reduceF)(max_cycles),
        Denoiser(height, width, fixed_cycles, max_cycles, trace_time) {
    }

    void *svc(void *t) {
        ((denoise_task *)t)->kernel_params = kernel_params;
        ((denoise_task *)t)->fixed_cycles = fixed_cycles;
        return Denoiser::svc(t);
    }

d: svc_end() {} ;

unsigned int restore(unsigned char *in, unsigned char *out, int *noisymap, unsigned int *noisy, unsigned int n_noisy, void *task) {
    unsigned int height = this->height;
    unsigned int width = this->width;
    memcpy(out, in, height * width * sizeof(unsigned char));
    FFSTENCILREDUCECUDA(DenoiserCUDA taskType, DenoiserCUDA mapF, reduceF)::svc(task);
    return this->getIter();
}

private:
    void *kernel_params;
};
Performance (CPUs + GPGPUs)
Video frames 768x512

Nehalem 32 cores + 1 K20
- 32 cores + K20
- 32 core

SandyBridge 16 cores + 2 Tesla M2090
- 16 cores + 2xM2090
- 16 cores + 1xM2090
- 16 cores

no difference w.r.t. hand-written CUDA code
Parallel memory allocation (lock-free)
Parallel memory allocation (lock-free)
Parallel memory allocation (lock-free)
Parallel memory allocation (lock-free)

- A network that connects data allocations-deallocations paths
- Faster than posix, often faster than hoard and TBB
  - unpublished, code available on sourceforge
- Implements deferred deallocation to avoid ABA problem
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);
Conclusion

- Low-level approach for a better performance/scalability
  - It is already a myth, a medium scale. Does someone still believe assembler is faster than C++?
  - Not proved, but we believe it will be even more evident at the large scale (exa-scale)

- FastFlow: header-only C++11 library
  - Research framework, portable everywhere exists a C++11 compiler, tiny codebase
  - Efficient, scalable

- A data-centric parallel programming model is paramount
  - High-level with a clear parallel semantics, compositional, enhancing locality and fast data movements
Think different.

Stay foolish, play with cars
Stay hungry, keep looking to new toys
Thanks

University of Turin
M. Aldinucci
G. Peretti
A. Secco
F. Tordini
M. Drocco
C. Misale

University of Pisa
M. Danelutto
M. Torquati

Queen's University Belfast
P. Kilpatrick

EU-FP7 - 3.5M€
Unito - 440K€
EU-FP7 - 3.7M€
EU-FP7 Network of Excellence

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