Parallel Programming using FastFlow

Massimo Torquati  <torquati@di.unipi.it>

Computer Science Department, University of Pisa - Italy

Karlsruhe, September 2nd, 2014
Outline

- Structured Parallel Programming
  - Algorithmic Skeletons & Parallel Design Patterns

- FastFlow: A data-flow framework for heterogeneous many-core platforms
  - In this talk we consider mainly multi-core systems

- Applications developed using FastFlow & Structured Parallel Programming
Structured Parallel Programming

- Structured parallel programming aims to provide *standard* (and *effective*) rules for composing parallel computations in a machine-independent way

  - **Goal**: reducing the complexity of parallelization problems by introducing *constraints*
    
    i.e. restricting the computation structure

  - **Modularity**, **portability**, and **programmability** are the keywords

- **Parallel paradigms** are the base components of parallel applications

- Using structured parallel programming force to think parallel
Structured Parallel Programming

- The parallel programmer is relieved from all concerns related to the implementation of the parallel paradigms on the target platform.
- The parallel programmer has to concentrate “only” on computational aspects.

Separation of concerns principles
Skeletons & Patterns

Algorithmic Skeletons

- From HPC community
- From early '90
- Pre-defined parallel high-order functions provided as constructs or lib calls

Parallel Design Patterns

- From SW engineering community
- From early '00
- “Recipes“ to handle parallelism (name, problem, algorithms, solutions, ...)

- The same concept at different abstraction levels
- We use the two terms patterns and skeletons, interchangeably.
  - We want to emphasise the similarities of these two concepts
Structured Parallel Programming using Patterns

- Map, Reduce, Stencil, ...
- Pipeline, Task-Farm
- Divide & Conquer, MDF, ...

Which pattern?

App developer

High-level code

Problem

System developer

Patterns
- Data Parallel
- Stream Parallel
- Task Parallel

run-time support

select
compose
extend
instantiate

iterate
Structured Parallel Programming: example

- **Problem**: apply the function F to all N elements of the input array A

  - Select the Map pattern
  - Instantiate the Map pattern with *static partitioning*

  **App developer**

  **run-time support**

  - Analyse the result: the workload is unbalanced
  - Instantiate the Map pattern with *dynamic partitioning*

  **App developer**

  **run-time support**

  - Select the “best” Map implementation on the target platform with *static partitioning* of data
  - Execute the code and provide the result

  - Select the “best” Map implementation on the target platform with *dynamic partitioning* of data
  - Execute the code and provide the result
Assessment (algorithmic skeletons)

Separation of concerns
- Application programmer: what is computed
- System programmer: how the result is computed

Inversion of control
- Program structure suggested by the programmer
- The run-time selects the optimization for the target platform

Performance
- Close to hand tuned code (sometimes better)
- Reduced development time. Lower total cost to solution.

“Structured Parallel Programming” by Marco Danelutto
Available on-line as SPM course material at M. Danelutto web page
http://www.di.unipi.it/~marcod
The FastFlow framework

- C++ class library
- **Promotes structured parallel programming**
- It aims to be flexible and efficient enough to target **multi-core, many-core** and **distributed** systems.
- Layered design:
  - **Building blocks** minimal set of mechanisms: *channels*, *code wrappers*, *combinators*.
  - **Core patterns** streaming patterns (*pipeline* and *task-farm*) plus the *feedback* pattern modifier
  - **High-level patterns** aim to provide flexible reusable parametric patterns for solving specific parallel problems

http://mc-fastflow.sourceforge.net
http://calvados.di.unipi.it/fastflow
FastFlow Building Blocks

- nodes are concurrent activities
  - POSIX threads
- arrows are shared-memory channels
  - implemented as **SPSC lock-free queues**

```c
struct myNode: ff_node {
    void *svc(void *task) { … return task; } 
};
```

**Minimal definition of a node :**
Stream Parallel Patterns (“core” patterns)

A stream is a sequence of data items having the same type

pipeline

ff_pipe<myTask> pipe(S1,S2,...,Sn);
pipe.run_and_wait_end();

Emitter: schedules input data items

Collector: gathers results

task-farm

std::vector<ff_node*> WorkerArray;
ff_farm<> farm(WorkerArray, &E, &C);
farm.run_and_wait_end();
FastFlow “core” patterns

**task-farm**

**pipeline**

Specializations

Patterns
Core Patterns Composition

pipeline + task-farm + feedback
Example: filtering images

Time = 288s

4-stage pipeline

Time = 112s

farm with pipeline workers

Time = 75s

parallelizing I/O too

Time = 33s

2 Xeon E5-2695 @ 2.4GHz, 1 disk storage
Data Parallel Patterns (1)

ParallelFor/ParallelForReduce

- \(M: \text{map function}\)
- \(R: \text{reduce function}\)

\[\text{ParallelFor pf;}\]
\[\text{pf.parallel_for}(0,N,[&](\text{const long } i) \{\]
\[\text{A[i]} = \text{F}(i);\]
\[\});\]

\[\text{ParallelForReduce<double> pfr;}\]
\[\text{pfr.parallel_reduce} (\text{sum}, 0.0,\]
\[0,N,[&](\text{const long } i) \{\]
\[\text{sum }+= \text{F}(i);\]
\[\});\]

ParallelForPipeReduce<task_t> pfr;
auto MapF = [&](task_t *task) { 
...;
\[\text{n.put}(\text{task});\]
};
auto ReducdF = [&](double &sum, const double v) { 
\[\text{sum }+= \text{v};\]
};

\[\text{pfr.parallel_reduce_idx}(0,N,\text{step},\text{chunk},\]
\[\text{MapF, ReduceF});\]

- Static and dynamic scheduling of tasks
- With or without scheduler thread (Sched)
Data Parallel Patterns (2)

- May be used inside stream parallel patterns (*pipeline* and *task-farm*)
- On multi-core systems all of them are implemented on top of ParallelFor* patterns
  - Algorithms + ParallelFor* wrappers
- For example, the **Map** pattern is a FastFlow node with inside an “optimized instantiation“ of the ParallelForReduce pattern

```c
struct myMap: ff_map<> {
    void *svc(void *task) { ....;
        ff_map<>::parallel_for(......);
    };
}
```

- **Targeting GPGPUs**: support for both CUDA and/or OpenCL available for some data-parallel patterns
  - Work still in progress
Example: Mandelbrot set (1)

- Very simple data-parallel computation
  - Each pixel can be computed independently
  - Simple ParallelFor implementation
- Black-pixel requires much more computation
- A naïve partitioning of the images quickly leads to load unbalanced computation and poor performance
  - Let's consider the minimum computation unit a single image line (image size 2048x2048, max $10^3$ iterations per point)
    - Static partitioning of lines (48 workers) MaxSpeedup 14
    - Dynamic partitioning of lines (48 workers) MaxSpeedup 37

Data-partitioning may have a big impact on the performance
Example: Mandelbrot set (2)

- Suppose now we want to compute a number of Mandelbrot images (for example varying the computing threshold per point)

- We have basically two options:
  1. One single parallel-for inside a sequential for iterating over all different threshold points
  2. A task-farm with map workers implementing two different scheduling strategies

- Which one is better having limited resources?
  - Depends on many factors, too difficult to say in advance

Moving quickly between the two solutions is the key point
Task-parallel patterns

Macro-Data Flow (MDF)

- The MDF executes data-dependency graph (DAG)
- Is a general approach to parallelism
- The user has to specify data dependency using a sequential function “taskGen” and to generate tasks (by using the AddTask method)
- The run-time automatically takes care of dependencies and then schedules ready task to Ws

```c
void taskGen( mdf ....) {
    Param 1 = {&A, INPUT};
    Param 2 = {&A, OUTPUT};
    mdf->AddTask(Params, F, A);
}
ff_mdf<Params> mdf(taskGen);
mdf.run_and_wait_end();
```

Divide&Conquer

- Currently is a task-farm with feedback channel with specialized Emitter and Collector.
Example: Strassen's algorithm

- Matrix multiplication using Strassen's algorithm:

\[
\begin{array}{cccc}
\text{A11} & \text{A12} & \times & \text{B11} & \text{B12} \\
\text{A21} & \text{A22} & \quad & \text{B21} & \text{B22} \\
\end{array}
\]

\[
= \begin{array}{c}
\text{C11} \\
\text{C12} \\
\text{C21} \\
\text{C22} \\
\end{array}
\]

- The sequential function \textit{taskGen} is responsible for generating instructions S1, S2, P1, S3, P2 ....... in any order specifying INPUT and OUTPUT dependencies.

- Each macro instruction can be computed in parallel using a ParallelFor pattern or optimized linear algebra matrix operations (BLAS, PLASMA, Lapack...)

\[
\begin{align*}
S1 &= A11 + A22 \\
S2 &= B11 + B22 \\
P1 &= S1 \times S2 \\
S3 &= A21 + A22 \\
P2 &= S3 \times B11 \\
S4 &= B12 - B22 \\
P3 &= A11 \times S4 \\
S5 &= B21 - B11 \\
P4 &= A22 \times S5 \\
S6 &= A11 + A12 \\
P5 &= S6 \times B22 \\
S7 &= A21 - A11 \\
S8 &= B11 + B12 \\
P6 &= S7 \times S8 \\
S9 &= A12 - A22 \\
S10 &= B21 + B22 \\
P7 &= S9 \times S10 \\
C11 &= P1 + P4 - P5 + P7 \\
C12 &= P3 + P5 \\
C21 &= P2 + P4 \\
C22 &= P1 - P2 + P3 + P6
\end{align*}
\]
Real applications (some)

Stream Parallel

- Bowtie (BT) and BWA Sequence Alignment Tools
- Peafowl, an open-source parallel DPI framework

Task Parallel

- YaDT-FF: fast C4.5 classifier (*)
- Block-based LU & Cholesky factorizations

Data Parallel

- Two Stage Image and Video Denoiser

Stream: Bowtie (BT) and DWA Sequence Alignment Tools

- Very widely used tools for DNA alignment
- Hand-tuned C/C++/SSE2 code
- Spin-locks + POSIX Threads
- Reads are streamed from memory-mapped files to worker threads

- Task-farm+feedback implementation in FastFlow
- Thread pinning + memory affinity + affinity scheduling
- Quite substantial improvement in performance

Stream: 10Gbit Deep Packet Inspection (DPI) on multi-core using Peafowl

- Peafowl is an open-source high-performance DPI framework with FastFlow-based run-time
  - Task-farm + customized Emitter and Collector
- We developed an HTTP virus pattern matching application for 10 Gibit networks
- It is able to sustain the full network bandwidth using commodity HW

Task-Parallel: LU & Cholesky factorizations using the MDF pattern

- Dense matrix, block-based algorithms
- Macro-Data-Flow (MDF) pattern encoding dependency graph (DAG)
  - The DAG is generated dynamically during computation
- Configurable scheduling of tasks, affinity scheduling
- Comparable performance w.r.t. specialized multi-core dense linear algebra framework (PLASMA)


DAG represents, 5 tiles, left-looking version of Cholesky algorithm
Data-Parallel: Two stage image restoration

- Detect: adaptive median filter, produces a noise map
- Denoise: variational Restoration (iterative optimization algorithm)
  - 9-point stencil computation
- High-quality edge preserving filtering
- Higher computational costs w.r.t. other edge preserving filters
  - without parallelization, no practical use of this technique because too costly
- *The 2 phases can be pipelined for video streaming*

Salt & Pepper image restoration

Original Baboon standard test image 1024x1024

10% impulsive noise

50% impulsive noise

90% impulsive noise

Restored

PNSR 43.29dB MAE 0.35

PNSR 32.75dB MAE 2.67

PNSR 23.4 MAE 11.21
Stream+Data-Parallel: Video de-noising, possible parallelization options using patterns

**Video de-noising: different deployments**

- Best option is to use all available GPGPUs

<table>
<thead>
<tr>
<th>Experiment name</th>
<th>Deployment</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPUs C++</td>
<td>①+②</td>
<td>Detect C++/1CPU, Denoise C++/14CPUs</td>
</tr>
<tr>
<td>CPUs OCL</td>
<td>①+②</td>
<td>Detect C++/1CPU, Denoise OpenCL/14CPUs</td>
</tr>
<tr>
<td>OCL CPUs+GPGPU(RR)</td>
<td>①+②</td>
<td>Detect C++/1CPU, Denoise OpenCL 14CPUs+1GPGPU, Round-Robin scheduling</td>
</tr>
<tr>
<td>OCL CPUs+GPGPU(OD)</td>
<td>①+②</td>
<td>Detect C++/1CPU, Denoise OpenCL 14CPUs+1GPGPU, On-Demand scheduling</td>
</tr>
<tr>
<td>1 GPGPU</td>
<td>①+②</td>
<td>Detect C++/1CPU, Denoise OpenCL/1GPGPU</td>
</tr>
<tr>
<td>2 GPGPUs</td>
<td>①+②</td>
<td>Detect C++/1CPU, Denoise OpenCL/2GPGPUs</td>
</tr>
</tbody>
</table>
Video de-noising demo

Thanks to Marco Aldinucci for the demo video
Conclusions

- Structured Parallel Programming models have been here from many years
  - *It is proved they work*
  - clear semantics, enforce separation of concerns, allow rapid prototyping and portability of code, almost same performance as hand-tuned parallel code.

- FastFlow: a C++ class library framework
  - *Research project* framework
  - Promotes structured parallel programming
  - Can be used at different levels, with each level providing a small number of building-blocks/patterns with clear parallel semantics
Thanks for your attention!

Thanks to:

- Marco Danelutto (Pisa)
- Marco Aldinucci (Turin)
- Peter Kilpatrick (Belfast)

FastFlow project site:

http://mc-fastflow.sourceforge.net
http://calvados.di.unipi.it/fastflow

EU-FP7 projects using FastFlow:

REPARA