A green perspective on structured parallel programming

Marco Danelutto\textsuperscript{a} P. Kilpatrick\textsuperscript{b}, M. Torquati\textsuperscript{a}

\textsuperscript{a}Univ. of Pisa
\textsuperscript{b}Queen’s Univ. of Belfast

EuroMicro PDP 2015 – Special session on Energy aware computing
Contents

Introduction
  Energy
  Parallel computing

Structured parallel programming
  Skeletons
  patterns
  Structured
  Energy

Experimental results
  Framework
  Results

Conclusions
Energy aware computing

HPC
- operation cost (MW)
- quality of service

Mobile computing
- battery life
- service on/off

power  performance
Energy savings

Smaller number of instructions
- More efficient algorithms,
- Different implementation models

Switching off unused resources
- Power on/off resources
- Using “idle” modes

Offloading to co-processors
- Only for particular computations
- GP-GPUs, FPGAs, DSPs

Frequency throttling (DVFS)
- Limited ranges,
- Per socket option
The art of power management

Ad hoc techniques

- applied with deep knowledge of the power saving techniques
- of the application business logic
- by the application programmers
The art of power management

Ad hoc techniques

- applied with deep knowledge of the power saving techniques
- of the application business logic
- by the application programmers

Based on

- design-implement-test cycle
- approximated power consumption measures
  - from counters (need special permissions)
  - from external measuring devices (ad hoc)
Parallel computing

Parallelism is everywhere:

- mobile phones (CPU + GPU)
- single workstation/server (CPU + GPU + Accelerator)
- data center (multiple nodes, internally parallel)

Strong technical motivations

- Moore law: single component (CPU) → multiple components

Software gap increasingly powerful devices with one century old programming paradigms
Algorithmic skeletons

Since ’90
- introduced by M. Cole
- parametric, reusable, efficient, parallelism
- exploitation mechanisms, provided as normal programming abstractions (library calls, objects, high order functions)

Frameworks
- OO: Sketo (Japan), Muesli (Germany), SKEPU (Sweeden), FastFlow (Italy)
- Functional: OSL (France), skel (UK), OcamlP3L (France/Italy)

Usage
- mostly academic
- several EU funded project (FP6: CoreGRID, GRIDcomp, FP7: ParaPhrase, REPARA, H2020 RePhrase)
Sample skeleton code (FastFlow)

Stage1

Stage2

Stage3  Stage3
Sample skeleton code (FastFlow)

```cpp
#include <ff/farm.hpp>
#include <ff/pipeline.hpp>

fftask_t* Stage1(fftask_t *t, ff_node*const node) {
    for(...) 
        ff_send_out(new ff_task_t(...)); 
    return(EOS); 
}

fftask_t* Stage2(fftask_t *t, ff_node*const node) { ... }

fftask_t* Stage3(fftask_t *t, ff_node*const node) { ... }

int main() {
    std::vector<ff_node*> W = {new Stage3, new Stage3};
    ff_pipe<fftask_t> pipe(new Stage1, Stage2, new ff_farm<>(W));
    if (pipe.run_and_wait_end()<0) error("running pipe");
    return 0;
}
```

http://calvados.di.unipi.it/fastflow
Parallel design patterns

Since ’00s
- from SW engineering community
- key concept → recipes to program common parallel patterns
- structured (layered): concurrency finding, algorithm structure, support structure, mechanism design spaces

Usage
- Mattson et al. “Patterns for parallel programming” book (the “parallel” version of the gang of four book)
- (industrial) Frameworks “inspired by”: Intel TBB, Microsoft TPL
- Advocated to overcome the “software gap” in the Berkeley report

Danelutto, Kilpatrick, Torquati
A green perspective on structured parallel programming
7 Divide and Conquer

Problem

Parallelize a divide and conquer algorithm.

Context

Divide and conquer is widely used in serial algorithms. Common examples are quicksort and mergesort.

Forces

- Problem can be transformed into subproblems that can be solved independently.
- Splitting problem or merging solutions is relatively cheap compared to cost of solving the subproblems.

Solution

There are several ways to implement divide and conquer in Intel® Threading Building Blocks (Intel® TBB). The best choice depends upon circumstances.

- If division always yields the same number of subproblems, use recursion and `tbb::parallel_invoke`.
- If the number of subproblems varies, use recursion and `tbb::task_group`.
- If ultimate efficiency and scalability is important, use `tbb::task` and continuation passing style.

Example

Quicksort is a classic divide-and-conquer algorithm. It divides a sorting problem into two subsorts. A simple serial version looks like:

```c
void SerialQuicksort( T* begin, T* end ) {
    if( end-begin>1 ) {
        using namespace std;
        T* mid = partition( begin+1, end, bind2nd(less<T>(),*begin) );
        swap( *begin, mid[-1] );
        SerialQuicksort( begin, mid-1 );
        SerialQuicksort( mid, end );
    }
}
```

The number of subsorts is fixed at two, so `tbb::parallel_invoke` provides a simple way to parallelize it. The parallel code is shown below:

```c
void ParallelQuicksort( T* begin, T* end ) {
    if( end-begin>1 ) {
        using namespace std;
        T* mid = partition( begin+1, end, bind2nd(less<T>(),*begin) );
        swap( *begin, mid[-1] );
        tbb::parallel_invoke( []{ParallelQuicksort( begin, mid-1 );},
                             []{ParallelQuicksort( mid, end );} );
    }
}
```

Eventually the subsorts become small enough that serial execution is more efficient. The following variation, with changed parts in blue, does sorts of less than 500 elements using the earlier serial code.

```c
void ParallelQuicksort( T* begin, T* end ) {
    if( end-begin>= 500 ) {
        using namespace std;
        T* mid = partition( begin+1, end, bind2nd(less<T>(),*begin) );
        swap( *begin, mid[-1] );
        tbb::parallel_invoke( []{ParallelQuicksort( begin, mid-1 );},
                             []{ParallelQuicksort( mid, end );} );
    } else {
        SerialQuicksort( begin, end );
    }
}
```

The change is an instance of the Agglomeration pattern.

The next example considers a problem where there are a variable number of subproblems. The problem involves a tree-like description of a mechanical assembly. There are two kinds of nodes:

- Leaf nodes represent individual parts.
- Internal nodes represent groups of parts.

The problem is to find all nodes that collide with a target node. The following code shows a serial solution that walks the tree. It records in `Hits` any nodes that collide with `Target`.

---

1 Production quality quicksort implementations typically use more sophisticated pivot selection, explicit stacks instead of recursion, and some other sorting algorithm for small subsorts. The simple algorithm is used here to focus on exposition of the parallel pattern.
A combined view

Parallel design patterns

- insight on parallelization possibilities
- parallelism aware restructuring of (sequential) business code
- alternative parallelisations

Algorithmic skeletons

- implementation of patterns
- different skeletons implement the same target
  - on different target hw
  - best suited to different application features

Approach taken in FP7 STREP projects

Danelutto, Kilpatrick, Torquati


A green perspective on structured parallel programming
Exposing parallel structure

- complete parallel structure exposed through skeleton nesting
- parameters of the skeletons are known
  - parallelism degree
  - e.g. scheduling policy
- rewriting rules known
  - affect parallel semantics
  - not the “business logic” of the code (functional semantics)

\[
\begin{align*}
R1 & \quad \text{pipe}(S_1, S_2) \equiv \text{seq comp}(S_1, S_2) \\
R2 & \quad \text{farm}(S) \equiv S \\
R3 & \quad \text{pipe}(\text{map}(S_f), \text{map}(S_g)) \equiv (\text{map}(S_f), \text{map}(S_g))
\end{align*}
\]
Exposing parallel structure

- complete parallel structure exposed through skeleton nesting
- parameters of the skeletons are known
  - parallelism degree
  - e.g. scheduling policy
- rewriting rules known
  - affect parallel semantics
  - not the “business logic” of the code (functional semantics)

R1: \[ \text{pipe}(S_1, S_2) \equiv \text{seq comp}(S_1, S_2) \]
R2: \[ \text{farm}(S) \equiv S \]
R3: \[ \text{pipe}(\text{map}(S_f), \text{map}(S_g)) \equiv (\text{map}(S_f), \text{map}(S_g)) \]
R3: \[ \alpha(f) \circ \alpha(g) \equiv \alpha(f \circ g) \]
Refactoring

\[ T_S = 10t \]

Simple performance model

- \( L \rightarrow \) latency of a sequential node
- \( T_S \rightarrow \) service time of a node
- service time of a pipeline is the maximum of its stage service time
- service time of a farm with identical workers is the service time of the worker divided by the number of the workers
Structured parallel programming

Refactoring

\[ T_S = t \]

- parallelize the bottleneck stage
- use a farm with as many workers as needed to keep the service time the same of the first stage

\[ L = t \]

\[ L = 10t \]
Refactoring

\[ T_S = \frac{t}{2} \]

- parallelizing first stage (also)
- to lower the overall service time → further parallelize second stage
- (this can be made provided there are no other “system” bottlenecks impairing the effect of new workers in the farm, of course)
Refactoring

\[ T_S = t \quad \text{ResNo} = 20 \]

- invert pipeline and farm: farm out the pipeline
- worker service time is 10\(t\)
- 10 workers move down the outer farm service time to \(t\)
- \textit{we need} \(10 \times (1 + 1)\) \textit{resources} (ideally)

\[ L = t \quad L = 10t \]
Refactoring

\[ T_S = \frac{11t}{10}, \quad \text{ResNo} = 10 \]

\[ S_1, \quad S_2 \]

\[ L = t, \quad L = 10t \]

- simply removing inner pipeline parallelism
- service times stays almost the same
- resource count drops down to half of the original

Danelutto, Kilpatrick, Torquati


A green perspective on structured parallel programming
Exploring space of solutions

\[
\begin{align*}
P(S,S) & \quad F(P(S,S)) \quad C(S,S) \\
P(S,F(S)) & \quad P(F(S),F(S)) \quad P(S,S) \quad F(C(S,S)) \quad C(S,F(S)) \\
P(F(S),F(S)) & \quad C(F(S),F(S)) \quad C(F(S),F(S)) \quad C(F(S),S)
\end{align*}
\]

\(P=\text{pipeline} \quad F=\text{farm} \quad S=\text{sequential}\)

Danelutto, Kilpatrick, Torquati


A green perspective on structured parallel programming
Cost model driven exploration

- model $\equiv$ cost function (some kind of)
- non monotonic process
- may require clever heuristics (or advanced techniques)
- not guaranteed existence of a "feasible" cost function
  - simple (analytic) or more complex (queue theory) models for performance (service time)
  - power consumption ??
Consider that ...

In an algorithmic skeleton framework

- moving from one tree to the next one requires to change a few lines of code
- changing parallelism degree it's a matter of assigning an int
- varying frequency *may be* a single external library call
- mapping is largely an automatic effort (e.g. FastFlow)
Building on previous experience

**Behavioural skeletons**

- rule base autonomic manager associated to skeleton
  - QoS driven heuristics coded
- sensor and actuator interface
  - provides manager with the possibility to read skeleton status and apply changes
Energy management with structured parallel programming

**Synchronization**
- Synchronization → useless (w.r.t functional result) instructions executed → “wasted” energy
- Opportunities:
  - Scheduling
  - Execution model

**DVFS**
- Tradeoff between
  - expected behaviour (performance), and
  - energy consumption
- Same performance with
  - low pardegree, high freq
  - high pardegree, low freq

Danelutto, Kilpatrick, Torquati
A green perspective on structured parallel programming
Green perspective: experimental results

- Multithreading effects
- FastFlow vs OpenMP
- Scheduling effects
- DVFS tradeoff
Experimental framework

Hardware

- Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz
  2x12 cores (2-way hyperthreading → 48 contexts)

Software

- Red Hat 4.4.7-4, Linux kernel 2.6.32
- FastFlow v2
- OpenMP (as of g++ 4.8.1)
- experiments on “dedicated machine” (no other users on the machine)
Threading and hyperthreading (unoptimized)

$\frac{1}{(\text{Completion time} \times \text{Energy consumed})}$

Parallelism degree

CPU frequency (GHz)

Danelutto, Kilpatrick, Torquati


A green perspective on structured parallel programming
Threading and hyperthreading (optimized)

\[
\frac{1}{\text{Completion time} \times \text{Energy consumed}}
\]

Parallelism degree vs. CPU frequency (GHz)

Danelutto, Kilpatrick, Torquati


A green perspective on structured parallel programming
Threading and hyperthreading (optimized, split)
Threading and hyperthreading (optimized)

When multithreading → pick up (high) wave parameters
Pardegree vs. frequency

![Graph showing Pardegree vs. frequency with configurations (1.2GHz pardegree - 2.4 GHz pardegree). The x-axis represents different configurations, and the y-axis represents completion time (msec) and energy consumed (Watts). The graph compares 2.4 GHz Tc, 1.2Ghz Tc, 2.4Ghz energy, and 1.2Ghz energy.](http://www.di.unipi.it, http://www.qub.ac.uk).

Danelutto, Kilpatrick, Torquati

A green perspective on structured parallel programming
A green perspective on structured parallel programming

Danelutto, Kilpatrick, Torquati


Pardegree vs. frequency

Use the larger number of resources at low freq ensuring user performance contract
Scheduling (Auto vs. Round robin)

Danelutto, Kilpatrick, Torquati

A green perspective on structured parallel programming

Scheduling (Auto vs. Round robin)

“Asynchronous” scheduling works better
Introduction

Structured parallel programming

Experimental results

Conclusions

Results

DVFS (1)

Danelutto, Kilpatrick, Torquati


A green perspective on structured parallel programming
Results

DVFS (2)

% difference vs Parallelism degree

Performance and Energy vs Parallelism degree

Danelutto, Kilpatrick, Torquati


A green perspective on structured parallel programming
DVFS

More benefits from DVFS with higher parallelism degrees
FastFlow vs. OpenMP
FastFlow vs. OpenMP

When enough resources are available, FastFlow is better than OpenMP both in terms of performance and energy.
Conclusions

Pillars

- structured parallel programming exposes significant parallel execution parameters
- structured approach to power tuning mechanisms management

Effects

- more cores at lower frequency better than less cores, faster
- synchronizations cost in both performance and energy
- offloading when suitable

Danelutto, Kilpatrick, Torquati


A green perspective on structured parallel programming
Any questions?

FastFlow
http://calvados.di.unipi.it/fastflow

REPARA
http://parrot.arcos.inf.uc3m.es/wordpress/

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

marco.danelutto@unipi.it
torquati@di.unipi.it

Danelutto, Kilpatrick, Torquati