IEEE 14th International Conference on High Performance Computing And Communication
June 2012

The Multiscale Hybrid Programming Model
A Flexible Parallelization Methodology

Nader KHAMMASSI
Jean-Christophe Le Lann
Jean-Philippe Diguet
Alexandre Skrzyniarz
» Structured Parallel Programming
Structured Parallel Programming With Skeletons

- High-Level approach based on a collection of recurrent deterministic execution patterns.
- Execution Patterns ↔ Algorithmic Skeletons ↔ Parallel Constructs
- Skeletons abstracts program description and hide low-levels multithreading details and many complexities inherent in parallel programming.
- Skeletons automates many parallel programming paradigm-related routines such communication, synchronization and data partitionning...
- Skeletons are reusable.
- Skeletons provides enough hardware abstraction to let the programmer focus on algorithm instead of architecture.
Task-Based Parallelization Methodology
The First Step: Program Decomposition Into Tasks

- The sequential program can be decomposed into coarse-grain tasks.
- These tasks can be decomposed in turn into finer grain tasks... etc
- Programmer can control task granularity to extract the maximum amount of available parallelism.

Parallelization Methodology
Parallelism Specification

- **The ideal case**: All tasks can be executed simultaneously!
Parallelism Specification

- **The ideal case**: All tasks can be executed simultaneously!
- Unfortunately, programs are “more-or-less” parallelizable → contains a varying amount of parallelism depending on algorithmic constraints and task-data dependencies.

“Ideal Case: Perfect Parallel Execution”
Parallelism Specification: Traditional Fork/Join Pattern

- Depending on the available parallelism, tasks can be executed either 
  **simultaneously or sequentially**
- This **Hybrid Execution** can be specified using the traditional Fork/Join execution pattern.
Parallelism Specification: Fork/Join Limitations

- However, **parallelism can be available at several levels of granularity**!
- Simple Fork/Join execution pattern specify parallelism at a single level of granularity.
- However, parallelism can be available at several different levels of granularity.

“2 Levels Task Parallelism Specification”
Parallelism Specification: Fork/Join Limitations

- However, **parallelism can be available at several levels of granularity**!
- Simple Fork/Join execution pattern specify parallelism at a single level of granularity.
- However, **parallelism can be available at several different levels of granularity**.

"2 Levels Task Parallelism Specification"

"3 Levels Task Parallelism Specification"
Parallelism Specification: Fork/Join Limitations

- However, parallelism can be available at several levels of granularity!
- Simple Fork/Join execution pattern specify parallelism at a single level of granularity.
- However, parallelism can be available at several different levels of granularity.

"2 Levels Task Parallelism Specification"

"3 Levels Task Parallelism Specification"
Parallelism Specification: Fork/Join Limitations

- However, parallelism can be available at several levels of granularity!
- Simple Fork/Join execution pattern specify parallelism at a single level of granularity.
- However, parallelism can be available at several different levels of granularity.

“2 Levels Task Parallelism Specification”

“3 Levels Task Parallelism Specification”

“4 Levels Task Parallelism Specification”
Parallelism Specification: The Hierarchical Task Graph

- The Hierarchical Task Graph is composed from hierarchical Fork/Join constructs.
- The HTG can specify task parallelism at many levels of granularity.
Parallelism Specification: The Hierarchical Task Graph

- The Hierarchical Task Graph is composed from hierarchical Fork/Join constructs.
- The HTG can specify task parallelism at many levels of granularity.
Parallelism Specification: The Hierarchical Task Graph

- The **Hierarchical Task Graph** is composed from hierarchical Fork/Join constructs.
- The HTG can specify task parallelism at many levels of granularity.
- However, a program may exhibit different types of parallelism and not only fork/join-based task parallelism.
Parallelism Specification: Parallelism Types

- A program may exhibit different types of parallelism and not only fork/join-based task parallelism.

**Sequential Execution**

**Task Parallelism**

**Hybrid Execution: Parallel/Sequential**

**Temporal Parallelism** (Pipelining)

**Data Parallelism**

Massive vectorization TLP/ILP (SIMD) on GPU/SSE/FPGA ...
» HTGG: The Hierarchical Task Group Graph
Parallelism Specification: The HTGG

- The **Hierarchical Task Group Graph (HTGG)** uses Task Group instead of simple task to accommodate execution patterns heterogeneity.
- The HTGG is a rich program intermediate representation which specify many different types of parallelism at different levels of granularity.
Parallelism Specification : The Task Group Interface

- The “task_group” is a common abstract interface which can be implemented by an extendable collection of execution patterns specifying different types of parallelism.
- “task_group” interface accommodate the HTGG heterogeneity and allow specification of different execution configurations into a single homogeneous construct.
Example : Parallelization of a simplified Multimedia Application
Example

- Parallelization of a simplified multimedia application

- An example of a sequential multimedia application
  - Read an input stream
  - Process and encode audio bitstream
  - Process and encode video bitstream
  - Write common output stream

- Firstly, the application is decomposed into a set of sequential tasks

- How to parallelize it?
Example

- Coarse-Grain Task Parallelism Specification
Example

- Finer Grain Data Parallelism Expression

```
+-----------------+
| load_input_stream |
+-----------------+
    |                |
    v                v
+-----------------+    +-----------------+
| extract_audio_samples | extract_video_frames |
+-----------------+    +-----------------+
    |                |                  |
    v                v                  v
+-----------------+    +-----------------+    +-----------------+    +-----------------+
| process_audio_samples | process_video_frames | encode_audio_samples | encode_video_frames | write_audio_to_stream | write_video_to_stream |
+-----------------+    +-----------------+    +-----------------+    +-----------------+    +-----------------+
    |                |                  |                  |                  |                |          |
    v                v                  v                  v                  v    v
+-----------------+    +-----------------+    +-----------------+    +-----------------+    +-----------------+    +-----------------+
| flush_output_stream |                |                  |                  |                  |                |          |
+-----------------+    +-----------------+    +-----------------+    +-----------------+    +-----------------+    +-----------------+
```

« Data Parallelism »

« Main Task Parallelism »
Example

- Temporal Parallelism Specification

- Data Parallelism

Pipeline

Example

Temporal Parallelism Specification

- Data Parallelism

Example

Temporal Parallelism Specification

- Data Parallelism
Data Parallelism

- More Parallelism...

Temporal Parallelism

Parallel for

- Pipeline

Example
Example

- Data Parallelism Specification At Finer Grain

```
Example

- Data Parallelism
- Temporal Parallelism

Pipeline

- noise_fltr
- band_pass_fltr
- gain_fltr

Parallel_for

- 0..i
- i..j
- ... m..n
- encode
- encode
- encode

load_input_stream

- extract_audio_samples
- process_audio_samples
- encode_audio_samples
- write_audio_to_stream

- extract_video_frames
- process_video_frames
- encode_video_frames
- write_video_to_stream

flush_output_stream

- sharpen
- blur
- contrast
- multiply

« Data Parallelism »

« Main Task Parallelism »

« Temporal Parallelism »
```
Example

- More Data Parallelism At Finer Grain

```
0..i  i..j  ...  m..n
```

:: gain :: gain :: gain

```
noise_fltr
\downarrow
band_pass_fltr
\downarrow
gain_fltr
```

```
load_input_stream
```

```
extract_audio_samples
\downarrow
process_audio_samples
\downarrow
encode_audio_samples
\downarrow
write_audio_to_stream
```

```
extract_video_frames
\downarrow
process_video_frames
\downarrow
encode_video_frames
\downarrow
write_video_to_stream
```

```
flush_output_stream
```

```
0..i  i..j  ...  m..n
```

:: encode :: encode :: encode

```
Parallel_for
```

```
More Data Parallelism At Finer Grain
```

```
Main Task Parallelism
```

```
Temporal Parallelism
```

```
Temporal Parallelism
```

```
Data Parallelism
```

```
Data Parallelism
```

```
Massive SIMD Operations (Parallel Vectors)
GPU / SSE ...
```

```
Lab-STICC
```

```
ENSTA Bretagne
```

```
Thales
```

Ce document est la propriété de Nader KHAMMASSI et il ne peut être reproduit ou communiqué sans autorisation écrite de l'auteur.
Parallelism Expression From Graphical to Code
The XPU Framework
Parallelism Expression

Parallelism Expression : Graphical → Code

- The HTGG has an intuitive graphical representation.
- But, How to translate such “complex” construct into simple and compact code?
Parallelism Expression : The XPU Framework

- XPU : X Processing Unit (Target Heterogeneous Multicore Architectures)
- XPU Framework is a pure C++ implementation of our programming model.
  - Easy to learn (Traditional C++ without any extensions).
  - Easily portable to many systems.
- Exploit C++ meta-programming capabilities to provide a friendly programming interface allowing easy expression of parallelism and producing efficient code.
- XPU is designed for high productivity and programmability.
- XPU automates many parallel programming paradigms-related routines such as tasks synchronization and shared memory management.
- XPU has an Intelligent Run-Time System (IRS) which optimize dynamically program execution on the underlying architecture : Cache-aware Task Scheduling, Scalable Data Parotionning.
The XPU Programming Interface: Task Definition

- Task definition in XPU is designed to promote the reuse of sequential code through a flexible programming interface.
- XPU Task support an extendable list of piece of code, including functions, class methods or lambda expression.
- Task implementations can be extended to support remote functions call on distributed systems.
The XPU Programming Interface: Task Definition

```
1 int load_input_stream(char * in_stream);
2 int extract_audio_samples(char * in_stream,
3     char * audio_samples);
4 int main()
5 {
6   task load_stream_t(load_input_stream, in_stream);
7   task extract_audio_t(extract_audio_samples,
8       __read_only(in_stream),
9       audio_samples);
10  ...
11   load_stream_t(); // or load_stream_t.run() to
12 }                 // to execute task
```

« Function as Task »
The XPU Programming Interface: Task Definition

```cpp
1 class image
2 {
3    public:
4        int sharpen(int val);
5        int blur(...);
6    ...
7  }
8
9 int main(
10 {
11    image img("img.jpg");
12    task sharpen_t(&img, &image::sharpen, 11);
13    }
```

« Class Method as Task »
The XPU Programming Interface: Task Definition

```c++
int main()
{
float * f;
xpu::task low_pass([&](float * samples, int freq) {...code... }, audio_samples, 7000);
}
```

« Lambda Expression as Task (C++0x or greater) »
The XPU Programming Interface: Task Parallelism

- Task parallelism can be specified at all levels of granularity using only two keywords: "parallel" and "sequential".
- XPU is able to extract transparently task-data dependencies.
- XPU detects automatically shared data accesses by concurrent tasks and protect it against potential race condition.
- This same information can be used by the run-time system to perform cache-aware scheduling and reduce communication overhead.

```c
1 void main() {
2 task t1(function, data_1), // task definition
3       t2(&o, cls::method, data_2), ...;
4 task_group * program;
5 program = parallel(sequential(t1, parallel(t3,t4)),
6                     sequential(t2, t5));
7 init();
8 program->run(); // 'data_4' protected automatically
9 clean();
10 }
```
The XPU Programming Interface: Expressiveness & Programmability

- **Expressiveness**: XPU vs Threading Building Block
  - We consider a sequential application composed from 7 functions.
  - We count required extra-lines of code to express parallelism (as specified in the previous figure) and the number of reused lines of sequential code.

→ XPU requires much lesser extra-code to express parallelism and reuse most of the sequential code with negligible modifications.
The XPU Programming Interface : Temporal Parallelism

- An example of a 4-stages pipeline.
- Tasks are used as processing pipeline stages.

```
1 void sharpen(int i, vector<image> * imgs) // i = frame index
2 { imgs[i]->sharpen(); }
3
4 void multiply(int i, vector<image> * imgs, image * mask)
5 { imgs[i]->multiply(mask); }
6
7 int main()
8 {
9   vector<image> frames(size);
10  ...
11 task sharpen_t(sharpen, 0, &frames),
12     blur_t(blur, 0, &frames),
13     multiply_t(multiply, 0, &frames, &mask);
14 task_group * process_image = pipeline(size, sharpen_t,
15                                        blur_t,
16                                        contrast_t,
17                                        multiply_t);
18 process_image->run(); // frame index “i” will be updated
```
The XPU Programming Interface: Temporal Parallelism

- Internal Pipeline Design: Transparent communication between processing stages through events.
The XPU Programming Interface : Data Parallelism

- Parallel For Loop
  - The XPU Intelligent Run-time system ensure dynamically scalable data partitioning on the underlying architecture.

```c
int process(int from, int to, int step, image* images) {
    for (int i=from; i<to; i+=step) ...
}

void main()
{
    image * images = ... ;
    task process_t(process, 0,0,0, images);
    task_group * pf;
    pf = new parallel_for(0, image_count, 1, &process_t);
    pf->run();
}
```
The XPU Programming Interface : Data Parallelism

- **Parallel Vector**
  - Express massive data parallelism on heterogeneous multicore architecture.
  - Generate transparently OpenCL kernel at run-time and manage transparently memory transfers and workload scheduling.
  - Not yet implemented as “task_group” ...

```c
#define size 1000000
int main()
{
    xpu::vector<float> A(size), B(size), C(size);
    A = B + C; // Transparent addition on GPU/CPU/FPGA...
}
```
Recapitulation: The MHPM Architecture
MHPM Architecture Overview

ISO C++ Programming Interface

Execution Pattern Collection

Sequential Program

Local Task Group Implementations

Data Parallelism: loops, vectors

Temporal Parallelism: pipeline

Intermediate Program Representation

Hierarchical Task Group Graph

Task 1

Task 2

Task 3

Task 4

Task 5

Task 6

Task 7

Task Parallelism

Data Set

data_1

data_2

data_3

data_4 (shared)

Intelligent Runtime System

Cache-Aware Scheduling

Shared Memory Management

Scalable Data Partitioning

Hardware Abstraction Layer

Optimized Task-Processor Mapping

Improved Data Locality

Hardware: Heterogeneous Parallel Architecture

General-Purpose Processing Cores (Multicore CPU / SMP)

Specialized Processing Cores: GPU, SPU...

Cache Topology

Memory Hierarchy
Conclusion and Future Works
Conclusion Future Works

- The HTGG is rich program intermediate representation which encapsulate parallelism specification and task ordering and task-data dependencies.
- The hardware abstraction layer give us dynamically a description of the underlying architecture: available processing units, their execution capabilities and processor cache topology.

**Cache-Aware Task Scheduling**
We are trying to exploit this valuable information about both hardware architecture and task-data dependencies to design an efficient cache-aware scheduling which will dynamically improve spatial and temporal data locality.

**Automatic Parallelization**
Task-data dependencies can be also exploited to generate automatically the HTGG from a sequence of tasks or task groups.

- Instead of building task execution graph manually (current version):
  \[
  \text{parallel}(\text{sequential}(\text{task1, task2}), \text{sequential}(\text{task3,...}) \ldots, \text{taskN})
  \]
- we generate transparently the HTGG
  \[
  \text{parallelize}(\text{task1, task2, task3, ...}, \text{taskN})
  \]
Thank You
Questions ?