Advanced Algorithms And Parallel Programming

Laboratory

Christian Rossi

Academic Year 2024-2025

Abstract

This course begins with an exploration of randomized algorithms, specifically Las Vegas and Monte Carlo algorithms, and the methods used to analyze them. We will tackle the hiring problem and the generation of random permutations to build a strong foundation. The course will then cover randomized quicksort, examining both worst-case and average-case analyses to provide a comprehensive understanding. Karger-s Min-Cut Algorithm will be studied, along with its faster version developed by Karger and Stein. We will delve into randomized data structures, focusing on skip lists and treaps, to understand their construction and application. Dynamic programming will be a key area, where we will learn about memoization and examine examples such as string matching and Binary Decision Diagrams (BDDs). The course will also introduce amortized analysis, covering dynamic tables, the aggregate method, the accounting method, and the potential method to equip students with robust analytical tools. Additionally, we will touch on approximate programming, providing an overview of this important concept. Finally, the competitive analysis will be explored through self-organizing lists and the move-to-front heuristic.

The second part of the course shifts to the design of parallel algorithms and parallel programming. We will study various parallel patterns, including Map, Reduce, Scan, MapReduce, and Kernel Fusion, to understand their implementation and application. Tools and languages essential for parallel programming, such as Posix Threads, OpenMP, and Message Passing Interface, will be covered, alongside a comparison of these parallel programming technologies. The course will also focus on optimizing and analyzing parallel performance, providing students with the skills needed to enhance and evaluate parallel computing systems. Practical examples of parallel algorithms will be reviewed to solidify understanding and demonstrate real-world applications.

Contents

CHAPTER 1

Implicit SPMD Program Compiler

1.1 Introduction

Single Instruction, Multiple Data (SIMD) architectures are designed to apply the same instruction to multiple data points simultaneously. In contrast, languages like C and C++ are primarily oriented toward single-threaded execution, supporting parallelism through well-known solutions such as OpenMP and `std::thread`. However, leveraging SIMD hardware poses unique challenges, as it typically requires either manual low-level programming or reliance on compilers to automatically vectorize code.

ISPC (Intel SPMD Program Compiler) is a C-based programming language specifically tailored for parallel computation. It employs a Single Program, Multiple Data (SPMD) model to facilitate parallelism across SIMD lanes, while also providing tasks to enable parallelism across multiple cores.

The ISPC compiler, built on the LLVM framework, is freely available and well-documented, making it accessible for developers looking to optimize their applications.

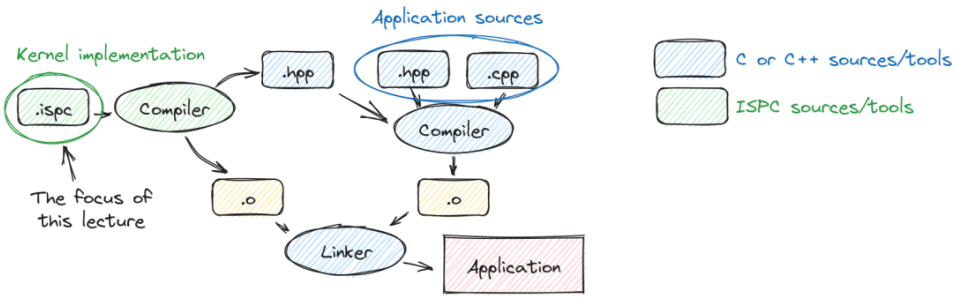

Figure 1.1: ISPC compiler

The ISPC computation model allows application code to be executed by a single processor as usual, while functions implemented in ISPC are executed by a gang of program instances, enabling efficient use of SIMD capabilities.

1.2 Syntax

The ISPC language is a C-based programming language designed for parallel computation. In ISPC, each local variable is associated with one of two qualifiers:

- *Uniform*: this qualifier indicates that the variable is shared across all instances within a gang.
- *Varying*: this qualifier denotes that the variable is private to each instance (this is the default behavior).

1.2.1 Loops

ISPC introduces specific statements to facilitate parallelization. For example, loops can be specified over potentially multi-dimensional domains of integer ranges using the following syntax:

```
foreach(identifier = start; identifer = end; identifier++) {
    /* body */
}
```

1.2.2 Communication

The ISPC language provides two sets of functions for inter-instance communication:

- *Cross-program instance operations*: these are low-level communication facilities that include operations such as `broadcast`, `rotate`, and `shuffle`.
- *Reductions*: these are high-level communication facilities that enable operations like `any`, `reduce_add`, and `reduce_max`.

The ISPC compiler implicitly defines two essential variables:

- `programIndex`: identifies each instance within a gang.
- `programCount`: represents the size of the gang.

To exchange values between instances, ISPC relies on the shuffle function, defined as follows:

```
int32 shuffle(int32 value, int permutation)
```

To retrieve a value from a specific ISPC process, the extract function is used:

```
uniform int32 extract(int32 value, uniform int index)
```

Additionally, the ISPC compiler can utilize the PRAM model to enhance performance. It is recommended to use reductions whenever available; if not, the low-level communication interface should be employed.

1.3 Scaling

In ISPC, a task is defined as a program that executes asynchronously, with each task being executed by a gang of program instances. The language provides two key keywords to manage the lifecycle of tasks:

- `launch[<n>]`: this keyword is used to spawn the execution of n tasks.
- `sync`: this keyword is employed to wait for the completion of all spawned tasks.

To define a task, you use the `task` prefix in a function declaration, ensuring that the function returns `void`. The syntax is as follows:

```
task void my_task_func( /* parameters */ ) {
    /* body */
}
```

Within the function body, two built-in variables can be utilized:

- `taskIndex`: represents the index of the current task.
- `taskCount`: denotes the total number of tasks.

It is important to note that the mapping between system threads and tasks is determined by the user, allowing for flexibility in implementation.

To effectively scale across different cores, tasks can be employed. While the mapping between system threads and tasks is adaptable, it is also advisable to consider external frameworks for managing cross-core parallelism. Solutions such as OpenMP, Intel Threading Building Blocks (TBB), or pthreads can complement ISPC's capabilities in handling parallel execution efficiently.