Introducing Parallelism

Ettore Speziale

Politecnico di Milano

Contents

Contents

Introducing Parallelism

Ettore Speziale

Introduction

Implicit Parallelism

Explicit Parallelism

Conclusions

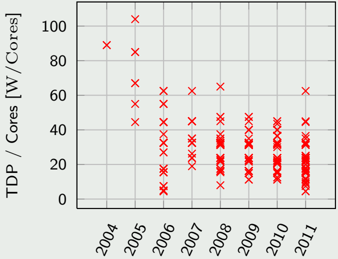
Bibliography

Why Bother About Parallelism

Current trend in computer architectures:

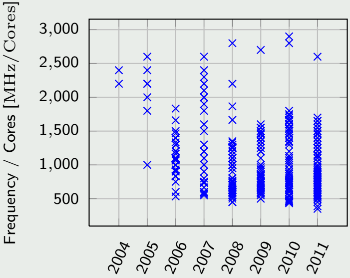
- increasing cores count

TDP [a] vs Cores Count



[a] Thermal Design Power

Frequency vs Cores Count



Caused by:

- power & memory walls

Walls

Limits in performance scaling identified by walls:

frequency performance can scale with frequency, at the cost of more power-hungry processors – not sustainable

memory improvement of processor technology is faster than the one of memory elements – bottleneck becomes feeding processors with data

Continuing on this road leads to:

- fast and power-hungry processors
- wasting cycles waiting for data from memory

Power Wall

Traditional way to improve micro-processors performance was:

- increase clock speed

However, it directly influences absorbed power [1]:

$$P_{dynamic} \backsim \frac{1}{2} \cdot C \cdot v^2 \cdot f$$

Lowering voltage requirements allows limiting $P_{dynamic}$:

- partially masks frequency contribution
- allows continue exploiting frequency increase
- positive effect also on static power:

$$P_{static} \backsim i_{static} \cdot v$$

Nowadays frequency must be faced

[1] CMOS technology

Computer Designer Answer

Containing power is trivial:

- remove power-hungry components [2] from designs
- limit core frequency

Guarantee performance is little bit difficult:

- increase number of available processing elements
- increase number of independent channels for accessing memory

Strategy is clear:

- we cannot improve execution latency
- split computation into chunks
- focus on increasing throughput

[2] e.g. branch predictor

Programmers Observation

The programming model is different:

- old abstraction of a single flow of control automatically optimized by compiler/hardware does not hold
- parallelism must be explicitly expressed inside the language
- ...

Computer designers have exposed a more complicated model to guarantee performance:

- more effort required to programmers in order to write efficient code

The 13 Dwarves

A key rule when dealing with parallel application is that there is not a preferred language/technique/design:

- for each problem, you should select the best language/technique/architecture combination

Relevant problems has been analyzed in [1]:

- 13 problems used as a reference to drive parallel architectures/programming research
- can be used to identify the best parallelization strategy for a kind of problem

Contents

Instruction Level Parallelism

Instruction Level Parallelism overlap the execution of different instructions:

- aims at maximizing instruction completion throughput
- dependences among instructions limit its applicability

In order to fully exploit Instruction Level Parallelism:

- instructions are analyzed while executing them, in order to detect dependencies
- instructions are scheduled considering only the dependencies detected at run-time
- independent instructions are used to fill execution slots not usable due to some dependency among other instructions

Consider the following example:

Sequential Addition

```
for(i = 0; i < 10; ++i) {
  c[i].x = a[i].x + b[i].x;
  c[i].y = a[i].y + b[i].y;
}
```

- branch predictor used to detect whether multiple iterations can be overlapped

- instructions in loop body analyzed to detect whether they can be overlapped/re-ordered
- caches used to exploit the regular access pattern to the array – 0, 1, . . . , i, i + 1, . . . , 9

Data Parallelism

Some applications access data in a predictable way:

- e.g. visit arrays using a regular access pattern

These application exposes parallelism at the data level:

- there is a large data-set
- operations on each element of the data set is quite independent from the other

There is a natural source of parallelism:

- independent operations on data

Vectors

Basic example of independent operation on data comes from high-performance computing:

- data is a set of bi/tri/quad dimensional points representing some kind of space – e.g. the speed of the airflow surrounding the wing of an aircraft
- you have to add/sum/multiply these points – e.g. simulate the evolution of the airflow speed around the wing, varying the wing angle of attack

Parallelism gathered from operations between the components of the vector:

- you have to use a special data structure, a vector

Hardware performs operations between vector efficiently:

- no need to check dependencies

Multi-media Instruction Set Extensions

Vectors introduced by vector processors:

- allows using vectors with an arbitrary number of components

Vectors also used in multimedia applications [3]:

- modern architectures exposes specialized instruction set for performing operations between vectors of a fixed size
- slightly different from vector processors

Fixed size is an hardware constraint:

- Intel MMX uses vectors of 64 bits
- Intel SSE uses vectors of 128 bits
- ...

[3] MPEG4 decoding

A vector can contains a different number of components:

- an SSE-enabled [4] processors handles parallel operations between vectors of 2 int64_t or 4 int32_t or ...

Speedup comes from:

- member-wise operations performed in parallel
- memory operations focus on throughput rather than on latency – specialized load/store units

[4] Vector size is 128 bit

Hardware vectors exposed to programmer through a compiler- and architecture-specific interface:

Defining Vectors with GCC-compatible Compilers

```
typedef
__attribute__((vector_size(16)))
int64_t storage_t;
```

Vector types are identified by attaching the vector_size attribute to a native type:

- the native type is vector element type
- the attribute takes as parameter the vector size
- defined vector size must match architecture vector size [5]

[5] With SSE $128bit = 2 \times 64bit = 16bytes$

Vector types can be used as native types:

- a + b, a − b, ...
- a == b, a != b, ...

Vector elements can be accessed using brackets:

a[0], a[1], a[i], ...

Advanced operations performed through builtins:

- function calls replaced by the compiler with optimized hardware instructions

Vector Shuffle

```
a = __builtin_shufflevector(a, a,
                            3, 2, 1, 0);
```

- shuffle extract elements from vector operands – a and a
- indices following vector operands identify which elements to extract
- vector operands must have the same type

Assembly

```
pshufd $27, %xmm0, %xmm0
```

Vector Shuffle

```
a = __builtin_shufflevector(a, a,
                            3, 2, 1, 0);
```

Let Sz be the lenght of vector operands:

- index $i \in [0, Sz - 1]$ refers to the i-th element of the first vector operand
- index $i \in [Sz, 2 * Sz - 1]$ refers to element $i - Sz$ of the second vector

Vectors allows to perform more complex operations in parallel:

- you have to use architecture-specific builtins

Square Root Source

```
#include <xmmintrin.h>

...

y = _mm_sqrt_ps(x);
```

Square Root Assembly

```
sqrtps  %xmm1,
        %xmm0
```

SAXPY

Single precision A times X Plus Y

Showing code on slides is both boring and error-prone, so I will use as much as possible vi and the shell. All sources are available on the course site. They are heavily commented. On slides there are only some tips.

"Talk is cheap, show me the code" [4]

Given two vectors y and x and a scalar a, SAXPY computes:

$$y_i = a \cdot x_i + y_i$$

In `saxpy.cpp` the kernel is implemented using different techniques. In order to see the effectiveness of each technique, compile `saxpy.cpp` without optimizations.

Contents

Processes and Threads

Explicitly parallelizing an application requires finding chunks of code that can run in parallel:

- each chunk of code is executed by a process/thread
- sometimes, synchronization is needed

Suitable for coarse-grain parallelization:

- e.g. serving multiple HTTP connections in parallel

Requires application-specific optimizations:

- e.g. use a process/tread pool for serving connections

Tasks as Elementary Units of Work

Working with threads/processes is difficult because:

- you have to manually split work between them
- synchronization is needed

Splitting work is actually a multi-phase process:

- identify an unit of work that can be run in parallel – a task
- group task in order to equally distribute them between the available processes/threads
- decide how many processes/threads create

Modern parallel framework focus on defining tasks:

- assigning tasks to executors – e.g. processes or thread – is automatically managed

Tasks and Data Parallelism

With data-parallelism a task is the work performed on a section of accessed data:

Scalar SAXPY

```
for(unsigned i = 0, e = n; i != n; ++i)
  y[i] += a * x[i];
```

The update of the element y[i] is independent from the others:

- it is a task

You have n different tasks, one for each iteration:

- the loop can be fully parallelized
- the unit of work – the tasks – in this context is the single iteration of the loop
- we call it parallel iteration

OpenMP

OpenMP is an extension of C, C++, Fortran focused on handling data-parallel computations through parallel loops:

Parallel Sum

```
#pragma omp parallel for
for(i = 0; i < 10; ++i) {
  c[i].x = a[i].x + b[i].x;
  c[i].y = a[i].y + b[i].y;
}
```

The **#pragma** tells compiler that all iterations are independent, thus they can be executed in parallel

OpenMP
Programming Model

An application is executed by multiple worker threads:

- one of them, executes the main function – master thread

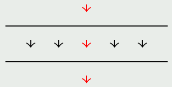
When the master thread encounter a parallel section:

- it awakes all the other worker threads
- the parallel section is executed by all workers, in parallel

Parallel Section

```
#pragma omp parallel
{ ... }
```

Fork-Join Execution



At the end of the parallel section:

- each thread wait for the others – barrier synchronization
- after waiting, only the master continues execution

During the execution of a parallel section, threads encounter work-sharing constructs:

- all threads collaborates to execute the construct

Explicit Parallel For

```
#pragma omp parallel
{
#pragma omp for
 for(i = 0; i < 10; ++i) {
   c[i].x = a[i].x + b[i].x;
   c[i].y = a[i].y + b[i].y;
 }
}
```

#pragma omp **for** defines 10 iterations:

- automatically partitioned between threads

A **#pragma** omp parallel containing only a **#pragma** omp **for** can be written in a more compact form:

Parallel For

```
#pragma omp parallel for
for(i = 0; i < 10; ++i) {
  c[i].x = a[i].x + b[i].x;
  c[i].y = a[i].y + b[i].y;
}
```

A parallel loop

for(init ; comp; update)

must respect the following constraints:

- init must initialize an integer variable iv
- comp must compare iv with a run-time constant using one operator from $\{ <, <=, >=, > \}$
- iv must be incremented/decremented by a run-time constant – e.g. iv $+=$ 4, iv $-=$ stride

These constraints allow the compiler parallelizing the execution of loop iterations

Synchronization **#pragma** predicates over a code-block:

Synchronization

```
#pragma omp parallel for
for(i = 0; i < 10; ++i) {
  #pragma omp critical
  foo();

  #pragma omp master
  bar();

  #pragma omp barrier
  baz();
}
```

omp critical

- critical section

omp master

- code-block executed only by the master thread. No synchronization at block enter/exit

The **#pragma** omp barrier directive performs a barrier synchronization:

- all threads must meet at the barrier
- a thread is allowed to leave the barrier only after all other threads reach the barrier

An implicit barrier is executed after each **#pragma** omp **for**

Contents

Parallel Programming

Writing parallel code is difficult:

- many different aspects to be considered at the same time
- difficult to think considering more than one execution flow

The goal of this course is to teach you that:

- there is not the best language
- for each problem, you should choose the most suited language

The same holds for parallel programming:

- for each problem, you should chose the most suited language/technique

Contents

Bibliography I

Krste Asanovic, Ras Bodik, Bryan C. Catanzaro, Joseph J. Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William L. Plishker Lester, John Shalf, Samulel W. Williams, and Katherine A. Yelick.
The Landscape of Parallel Computing Research: A View from Berkeley.
Technical report, EECS Department, University of California, Berkeley, 2006.

Bruce Eckel.
Thinking in C++ – Volume One: Introduction to Standard C++.
http://mindview.net/Books/TICPP/ThinkingInCPP2e.html.

Bibliography II

Bruce Eckel and Chuck Allison.
Thinking in C++ – Volume Two: Practical Programming.
http://mindview.net/Books/TICPP/ThinkingInCPP2e.html.

Linus Torvalds.
Re: SCO: "thread creation is about a thousand times faster than onnative.
https://lkml.org/lkml/2000/8/25/132.