IN3200/IN4200 Summary

May 2022

Course content

Excerpts from the official course description:

In this course, you will learn about the basic concepts of parallel programming and high performance computing, as well as the most basic communication commands in MPI and OpenMP.

Students will gain the knowledge they need to effectively use modern architecture to solve computationally high-tech scientific issues.

Syllabus

- Textbook “Introduction to High Performance Computing for Scientists and Engineers” (see the semester webpage for the required chapters & sections)
- Lecture slides used to teach these book chapters

Chapter 1: Modern processors

- A *high-level overview* of the architecture of modern cache-based microprocessors
- Introduction of important concepts
- Discussion of inherent performance limitations

The “stored-program computer” concept

- Instructions (produced by a *compiler*) and data are stored in memory
- Instructions are read and executed by a control unit
- An arithmetic/logic unit “does the work”, which is coded in the instructions
- The speed of memory determines how fast instructions and data can be fed to the control and arithmetic units—**limitation of performance**
- I/O facilities enable interaction with users

Performance metrics

The performance at which a CPU's floating-point (FP) units generate results for multiply and add operations is measured in **floating-point operations per second (Flops/sec)**.

Feeding the arithmetic units with operands is a complicated task. The most important data paths from the programmer's point of view are those to and from the caches and main memory. The bandwidth (performance) of those paths is quantified in **GBytes/sec**.

Pipelining

Subdividing complex operations into simple components that can be executed using different functional units, it is possible to increase *instruction throughput*—the number of instructions executed per clock cycle.

This is the most elementary example of *instruction-level parallelism* (ILP).

Optimally pipelined execution may lead to a throughput of one instruction per cycle per pipeline.

It is the job of the compiler to arrange instructions in such a way as to make efficient use of all the different pipelines.

Superscalarity

Goal: To produce more than one “result” per cycle.

- Multiple instructions are fetched and decoded concurrently
- Address and other integer calculations are performed in multiple integer (add, mult, shift, mask) units
- Multiple floating-point pipelines run in parallel
- Caches are fast enough to sustain more than one load or store operation per cycle

Superscalarity is a special form of parallel execution, and a variant of ILP.

Hardware out-of-order execution and compiler optimization must work together to fully exploit superscalarity.

SIMD (single-instruction-multiple-data)

Modern cache-based processors have instruction set extensions (**mostly enabled by compiler**) for both integer and floating-point operations. They allow the concurrent execution of arithmetic operations on "wide" registers, each holding multiple numerical values.

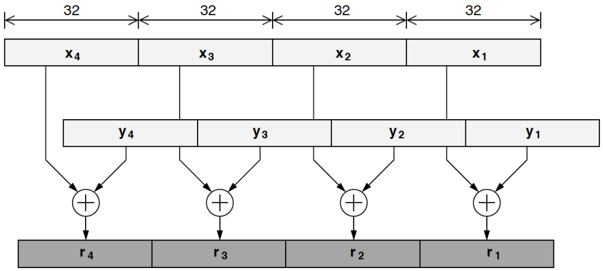

Figure 1.8: Example for SIMD: Single precision FP addition of two SIMD registers (x,y), each having a length of 128 bits. Four SP flops are executed in a single instruction.

Memory hierarchy

Data can be stored in a computer system in many different ways.

CPU has a set of registers, which can be accessed without delay. (**Each operand of an instruction must find its way from memory to a register first.**)

In addition, there are several levels of *cache*, holding copies of recently used data items.

Main memory of a computer is much slower (than the caches).

Cache

Caches are low-capacity, high-speed memories that are commonly integrated on the CPU die.

- L1 (level 1) data cache
- L1 instruction cache
- L2 and L3 (data & instruction) unified caches

The purpose of cache—reducing the impact of main memory's small bandwidth and high latency.

Cache lines

The content of a cache is organized as *cache lines*. (A cache line has space for multiple data items.)

All data transfers between caches and main memory happen on the cache line level.

If a code has good *spatial locality*, that is, the probability of successive accesses to neighboring items is high, the latency problem can be significantly reduced.

Prefetch

Prefetching supplies the cache with data ahead of the actual requirements from an application code.

Typically, a **hardware pre-fetcher** can detect regular access patterns and try to read ahead the needed data.

To completely hide the cache miss latency, the memory subsystem must be able to sustain a certain number of outstanding prefetch operations.

Chapter 2: Basic optimization techniques for serial code

Most important content:

- "Common sense" and simple optimization strategies for serial code

Very simple code changes can sometimes lead to significant performance boost.

The most important "common sense" principle: **avoiding performance pitfalls**!

Some typical “common sense” optimizations

- “Do less work”
- Avoid expensive operations
- Strength reduction
- Shrinking the work set
- Avoid branching

Chapter 3: Data access optimization

- What is the maximumly achievable performance?
  - Balance analysis and “lightspeed” estimates
- Data access optimization techniques

Importance of data access

Applications in science and engineering mostly consist of **loop-based** code that moves large amounts of data in and out of the CPU.

Accessing data in the memory hierarchy (from L1 cache to main memory) is often the most prominent performance limiter.

Modern microprocessors have a very impressive theoretical peak performance (in number of FP operations executable per second), but the memory system is "**too slow**".

Understanding the “limitation”

Bandwidth-based performance modeling—to get a rough idea about the maximum performance for a code.

One can *estimate* the theoretically achievable performance of loop-based code, if it is bound by bandwidth limitations.

The concept of “machine balance”

Machine balance, B_m, of a processor is the ratio between the maximum memory bandwidth and the peak FP performance:

$$B_\mathrm{m} = \frac{\text{memory bandwidth [GWords/sec]}}{\text{peak FP performance [GFlops/sec]}} = \frac{b_\mathsf{max}}{P_\mathsf{max}}$$

Access latency is assumed to be hidden completely (for example thanks to prefetch).

“Word” = one DP value (8 bytes)

“Memory bandwidth” could also be substituted by the bandwidth to caches or even network bandwidth.

The concept of “code balance”

To characterize a loop, we can calculate the **code balance** B_c:

$$B_\mathrm{c} = \frac{\text{data traffic [Words]}}{\text{floating-point operations [Flops]}}$$

That is, you should count the number of FP operations (easy), and also count (or estimate) the amount of data transfered over the performance-limiting data path (can be difficult).

Note: $\frac{1}{B_\mathrm{c}}$ is called **computational intensity**.

What is the expected maximum performance of a loop?

When you know the machine balance B_m of a CPU, and you want to run a loop that has B_c as its code balance.

What will be the maximum achievable performance P (in Flops/sec)?

$$P = \min\left(P_\mathsf{max}, \frac{b_\mathsf{max}}{B_\mathrm{c}}\right)$$

Recall: P_max denotes the maximum FP performance, b_max denotes the maximum bandwidth of the performance-limiting data path.

How realistic is b_{max}?

In reality, even the simplest memory-intensive loops are not able to achieve the theoretical hardware maximum memory bandwidth b_{max}.

The well-known *stream* micro-benchmarks can be used to measure the realistically achievable maximum memory bandwidth b_{S}.

Then, the realistically achievable maximum FP performance is estimated as

$$P = \min\left(P_{\mathsf{max}}, \frac{b_{\mathrm{S}}}{B_{\mathrm{c}}}\right)$$

Data access optimization techniques

- Algorithm class $O(N)/O(N)$: loop fusion
- Algorithm class $O(N^2)/O(N^2)$: loop unroll & jam, loop blocking
- Algorithm class $O(N^3)/O(N^2)$: loop unroll & jam, loop blocking

Chapter 4: Parallel computers

- An introduction to the fundamental variants of parallel computers
  - The *shared-memory* type
  - The *distributed-memory* type
- A glimpse at basic design rules and performance characteristics for communication networks

Shared-memory computers

A *shared-memory parallel computer* has a number of CPUs (cores) that work on a shared physical address space.

Two varieties:

- *Uniform Memory Access* (UMA) systems hae a “flat” memory model: latency and bandwidth are the same for all processors and all memory locations. (Typically, single multicore processor chips are “UMA machines”.)
- *Cache-coherent Nonuniform Memory Access* (ccNUMA) systems have a physically distributed memory that is *logically shared*. The aggregated memory appears as one single address space. Memory access performance depends on the which CPU (core) accesses which parts of memory (“local” vs. ”remote” access).

Caches are not (completely) shared

A shared-memory system, no matter UMA or ccNUMA, has multiple CPU cores.

Although there is a single address space (shared memory), there are private caches, or partially shared caches, for the different CPU cores.

Therefore, copies the same cache line **may** reside in several local caches.

Cache coherence

Problematic situations when copies of the cache line reside in several caches:

- If the cache line in one of the caches is modified, the other caches' contents are *outdated* (thus invalid).
- If different parts of the cache line are modified by different processors in their local caches $\rightarrow$ no one has the correct cache line anymore.

Cache coherence protocols (**supported in hardware**) guarantee *consistency* between cached data and data in the shared memory at all times.

ccNUMA

- A *locality domain* (LD) is a set of processor cores together with locally connected memory. This "local" memory can be accessed by the set of processor cores in the most efficient way, without resorting to a network of any kind.
- Each LD is a UMA building block.
- Multiple LDs are linked via a coherent interconnect, which can mediate direct, cache-coherent memory accesses. (This mechanism is transparent for the programmer.)
- The whole ccNUMA system has a shared address space (memory), runs a single OS instance.

Penalty for non-local transfers

The *locality problem*: Non-local memory transfers (between LDs) are more costly than local transfers (within a LD).

The *contention problem*: If two processors from different LDs access memory in the same LD, fighting for memory bandwidth.

Both problems can be “solved” (alleviated) by carefully observing the data access patterns of an application and restricting data access of each processor (mostly) to its own LD, through **proper programming**.

Typical modern distributed-memory systems

A cluster of shared-memory "*compute nodes*", interconnected via a *communication network*.

Each node comprises at least one network interface that mediates the connection to the communication network.

A serial process runs on each CPU (core). Between the nodes, processes can communicate by means of the network.

The layout and speed of the network has a considerable impact on application performance.

Basic performance characteristics of networks

- Point-to-point communication (from one compute element to another)
- Bisection bandwith (a measure of the “whole” network)

Simple performance of point-to-point communication

Time spent on transferring a message of size N [bytes] from a "sender" process to a "receiver" process:

$$T = T_\ell + \frac{N}{B}$$

This is a simplified model:

- T_ℓ: latency
- B: maximum network point-to-point bandwith [bytes/sec]

T_ℓ and B are considered as constants, but in reality they can both depend on N, as well as on the locations of the two processes.

Bisection bandwidth

How to quantify the “total” communication capacity of a network?

Bisection bandwidth of a network, B_{b}, is the sum of the bandwidths of the minimal number of connections cut when splitting the system into two *equal-sized* parts.

Chapter 5: Basics of parallelization

High-performance computing = efficient serial computing + effective parallel processing → needs parallel programming

But before actually engaging in parallel programming, it is vital to know some fundamental things in parallelization:

- The most common strategies for parallelization
- Simple theoretical insights into the factors that can hamper parallel performance

Why parallelize?

- We want to solve the problems faster, but the speed of a single CPU core has “saturated”.
- We want to solve larger problems, but the main memory available on a single system is not large enough.

So, **we need to identify parallelism** in a given computational problem, so that parallel programming can produce a parallel implementation that can efficiently use many processor cores, on a shared-memory or distributed-memory system.

- Data parallelism (the dominant parallelization concept, with many variants)
- Functional parallelism

Once the parallelism is identified, a parallel algorithm can be devised accordingly.

Parallel scalability

The *ideal* goal: If a problem takes time T to be solved by one worker, we expect the solution time by using N identical workers to be T/N—a perfect **speedup** of N.

However, perfect speedup is often not achievable in reality, why?

Factors that limit parallel execution

Reasons for non-perfect speedup:

- Not all workers might execute their tasks equally fast, because the problem was not (or could not be) partitioned into equal pieces—load imbalance;
- There might be shared resources which can only used by one worker at a time—serialization;
- New tasks may arise due to parallelization, such as communication between workers—overhead.

Scalability metrics

How well can a computational problem be parallelized?

Scalability metrics help to answer the following questions:

- How much faster can a given problem be solved with N workers instead of one?
- How much more work can be done with N workers instead of one?
- What impact do the communication requirements have on performance and scalability?
- What fraction of the resources is actually used productively?

Strong scaling

Single-worker (serial) normalized runtime for a fixed-size problem:

$$T_{\mathrm{f}}^{\mathrm{s}} = s + p$$

where s is the serial, non-parallelizable fraction, p is the perfectly parallelizable fraction.

Solving the same problem using N workers will require a runtime of

$$T_{\mathrm{f}}^{\mathrm{p}} = s + \frac{p}{N}$$

This is called **strong scaling**, because the total amount of work stays constant no matter how many workers are used.

Here, the goal of parallelization is minimization of time-to-solution for a given problem.

Weak scaling

For **weak scaling**, the goal is to solve an increasingly larger problem with more workers N.

More specifically, the total amount of work is scaled with some power of N

$$s + pN^{\alpha}$$

which means that single-worker runtime for the variable-sized problem **would have been** $T_{\mathrm{v}}^{\mathrm{s}} = s + pN^{\alpha}$.

Using N workers, the parallel runtime is

$$T_{\mathrm{v}}^{\mathrm{p}} = s + pN^{\alpha-1}$$

Here, we have also assumed that s doesn't grow with N.

The most typical choice is $\alpha = 1$, then $T_{\mathrm{v}}^{\mathrm{s}} = s + pN$ and $T_{\mathrm{v}}^{\mathrm{p}} = s + p$.

Simple scalability laws

How to calculate speedup?

$$\text{application speedup} = \frac{\text{serial runtime}}{\text{parallel runtime}}$$

"Amdahl's Law" (see Chapter 5) gives the upper limit of speedup in the context of strong scaling (fixed problem size).

Parallel efficiency is defined as

$$\varepsilon = \frac{\text{speedup}}{N}$$

This will be a value between 0 and 100%.

Chapters 6,7,8: OpenMP programming

The applicable hardware context: **shared memory**

- All processors can directly access all data in a shared memory, no need for explicit communication between the processors
- OpenMP: A parallel programming standard for shared-memory parallel computers
    - A set of compiler directives (with additional clauses)
    - A small number of library functions
    - A few environment variables

Threads in OpenMP

- The central execution entities in an OpenMP program are **threads**—lightweight processes.
- The OpenMP threads share a common address space and can mutually access data.
- Spawning a thread is much less costly than forking a new process, because threads share everything except the instruction pointer, stack pointer and register state.
    - If wanted, each thread can have a few “private variables” (by means of the local stack pointer).

OpenMP's parallel execution model

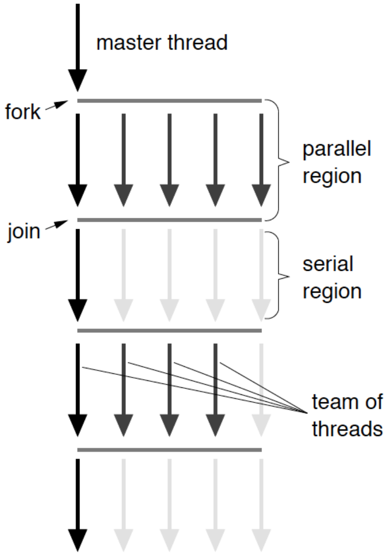

Figure 6.1: Model for OpenMP thread operations: The master thread "forks" team of threads, which work on shared memory in a parallel region. After the parallel region, the threads are "joined," i.e., terminated or put to sleep, until the next parallel region starts. The number of running threads may vary among parallel regions.

Data scoping

Any variables that existed before a parallel region still exist inside the parallel region, and are by default **shared** between all threads.

Often it will be necessary for the threads to have some *private* variables.

- Each thread can either declare new local variables inside the parallel region, these variables are private “by birth”;
- Or, each thread can “privatize” some of the shared variables that already existed before a parallel region (using the `private` clause)
- Each “privatized” variable has one (*uninitialized*) instance per thread;
- The private variables’ scope is until the end of the parallel region.

OpenMP performance pitfalls

- OpenMP is prone to the “standard problems” of parallel programming: **serial fraction** (Amdahl’s law) and **load imbalance**.
- Communication (in terms of data transfer) on shared memory is usually much less costly than on distributed memory, but ccNUMA can potentially cause performance problem (discussed in Chapter 8).
- There are specific performance problems inherent with shared-memory programming (see details in Chapter 7).

Penalty for non-local transfers (see Chapter 8)

The *locality problem*: Non-local memory transfers (between LDs) are more costly than local transfers (within a LD).

The *contention problem*: If two processors from different LDs access memory in the same LD, fighting for memory bandwidth.

Both problems can be "solved" (alleviated) by carefully observing the data access patterns of an application and restricting data access of each processor (mostly) to its own LD, through proper programming.

Data initialization is key.

Chapters 9,10,11: MPI programming

- Explicit "**message passing**" is required on distributed-memory systems
- The same program runs on all processes (Single Program Multiple Data, or **SPMD**)—no difference from OpenMP programming in this regard
- The work of each process is implementation in a sequential language (such as C)
    - Data exchange (sending and receiving messages) is done via calls to an appropriate library
- All variables in a process are **local** to this process (nothing is shared)

An abstract "picture" of parallel execution

Process 0

```
#include <stdio.h>
#include <mpi.h>

int main (int nargs, char** args)
{
  int size, my_rank;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  printf("Hello world, I've rank %d out of %d procs.\n",
         my_rank,size);
  MPI_Finalize ();
  return 0;
}
```

Process 1

```
#include <stdio.h>
#include <mpi.h>

int main (int nargs, char** args)
{
  int size, my_rank;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  printf("Hello world, I've rank %d out of %d procs.\n",
         my_rank,size);
  MPI_Finalize ();
  return 0;
}
```

$\cdots$

Process P-1

```
#include <stdio.h>
#include <mpi.h>

int main (int nargs, char** args)
{
  int size, my_rank;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  printf("Hello world, I've rank %d out of %d procs.\n",
         my_rank,size);
  MPI_Finalize ();
  return 0;
}
```

One scenario for deadlock

Deadlocks may occur if the possible synchronousness of MPI_Send is not taken into account.

```
int rank, size, left, right,in_buf[N], out_buf[N];
MPI_Status;

// .......

MPI_Comm_size (MPI_COMM_WORLD, &size);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);

right = rank==size-1 ? 0 : rank+1;
left = rank==0 ? size-1 : rank-1;

MPI_Send(out_buf,N,MPI_INT,right,0,MPI_COMM_WORLD);
MPI_Recv(in_buf,N,MPI_INT,left,0,MPI_COMM_WORLD,&status);
```

Note: Deadlock may arise in other scenarios!

How costly is a point-to-point message?

We already learned in Chapter 4, but repeated in Chapter 9:

$$T = T_\ell + \frac{M}{B}$$

- M: size of the message [bytes]
- T_ℓ: latency
- B: maximum network point-to-point bandwith [bytes/sec]

Some technical topics (see Chapters 10 & 11)

- Danger of implicit serialization
- Domain decomposition in the context of MPI programming
- Mapping of MPI processes to physical processors
- Message aggregation
- Asynchronous communication
- Hybrid MPI+OpenMP programming