Parallelism – Some Theory

Computational Science II (CAAM 520)

Christopher Thiele

Rice University, Spring 2020

Overview

- Why parallelize?
- Shared and distributed memory parallelism
- Processes and threads
- OpenMP and MPI
- Terminology
- Strong and weak scalability
- Amdahl’s law

Why parallelize?

There are (at least) two possible motivations:

- Utilization of more computational resources to reduce time-to-solution.
- Utilization of more computational resources to solve larger problems.

Furthermore, contemporary hardware, e.g., multi-core CPUs, makes parallelization somewhat mandatory.

Shared memory parallelism

Shared memory parallelism targets platforms on which all compute units have access to the same memory.

Examples:

- Multiple CPU cores on a single chip accessing main memory
- Multiple CPUs who can access each other’s memory
- GPGPUs

Distributed memory parallelism

Distributed memory parallelism targets platforms on which each compute unit has its own, private memory.

Example:

- A computer cluster in which compute nodes are connected via network.

Shared and distributed memory parallelism

The boundaries are fluid due to technologies such as ***remote direct memory access*** (RDMA).

Besides the hardware perspective, shared and distributed memory parallelism can also be viewed as software concepts.

→ Just because compute units can share memory, they do not have to.

Processes and threads

Modern operating systems (OS) for desktop and laptop computers, servers, etc. support multi-tasking, i.e., tasks or applications can run concurrently.

Multi-tasking can be implemented in different ways:

- Time-sharing: All tasks run on the same CPU (core), and the OS switches between them.
- True multi-tasking: Tasks run on separate CPUs or CPU cores, i.e., truly concurrently.

Processes and threads

Processes are independent tasks that can be scheduled by the OS using either multi-tasking approach.

Threads are tasks that are created by a process. They typically and share a process ID, memory, etc. with the parent process, but they can be scheduled independently.

→ Specific differences between processes and threads depend on the operating system.

Processes and threads

Threads are suitable for shared memory environments.

- We will use OpenMP for shared memory parallelism.
- OpenMP applications spawn multiple threads which access shared memory.

Distributed memory environments require processes.

- We will use the Message Passing Interface (MPI) to develop distributed memory parallel applications.
- MPI applications spawn multiple processes which exchange messages.

Terminology

Some conventions:

- ***Thread***: the software concept discussed before, i.e., not a hardware feature as in hyperthreading
- ***CPU*** or ***processor***: an entire physical CPU, may consist of multiple CPU cores
- We always distinguish a ***process*** (software) from a ***processor*** (hardware).
- ***(Compute) node***: an entire computer, i.e., the whole box

Terminology

Consider a parallel application which does the (normalized) amount of work

$$s + p = 1,$$

where

- p is the fraction of work that can be performed in ***parallel***, and
- s is the fraction of work that must be performed ***sequentially***.

Terminology

Let T_f^1 be the time required to do the ***fixed*** amount of work $s + p$ ***sequentially***.

Let $T_f^N = s$ be the time required to do the same amount of work $s + p$ in ***parallel*** using N ***workers***.

(***Note:*** Our definitions differ from those used by Hager & Wellein.)

Terminology

For $N \geq 1$, let us define ***performance*** as work over time, i.e.,

$$P_f^N = \frac{s + p}{T_f^N}.$$

Terminology

We define the ***speedup*** as

$$S^N = \frac{P_f^N}{P_f^1} = \frac{T_f^1}{T_f^N}.$$

$\longrightarrow$ The speedup tells us how much performance increases, and how much time-to-solution decreases when using N workers instead of one.

Terminology

We define ***parallel efficiency*** as

$$E_f^N = \frac{P_f^N}{NP_f^1} = \frac{T_f^1}{NT_f^N} = \frac{S^N}{N}.$$

When using N workers instead of one, we should expect a speedup of at most N.

$\longrightarrow$ Parallel efficiency tells us what fraction of the ideal speedup can is achieved.

Strong scalability

So far we kept the amount of work fixed, and we increased the number of workers.

The speedup, as defined for this setting, measures ***strong scalability.***

Weak scalability

Let us now increase the number of workers, keeping the amount of work per worker fixed.

In other words, consider the time T_v^N it takes to do the ***variable*** amount of work

$$N = N \underbrace{(s + p)}_{=1}$$

using N workers.

(Note that $T_v^1 = T_f^1$ by definition.)

Weak scalability

Let us define performance and efficiency as

$$P_v^N = \frac{N}{T_v^N}$$

and

$$E_v^N = \frac{P_v^N}{P_v^1} = \frac{T_v^1}{T_v^N}.$$

The efficiency, as defined for a variable amount of work, measures ***weak scalability***.

$\longrightarrow$ Compare it to the definition of $E_f^N = \frac{T_f^1}{NT_f^N}$ in the strong scalability setting!

Amdahl's law

Let us separate T_f^N into

$$T_f^N = T_{f,s} + T_{f,p}^N,$$

where

- $T_{f,s}$ is the time required to do the fraction s of the total amount of work $s + p = 1$ which cannot be parallelized, and
- $T_{f,p}^N$ is the time required to do the remaining fraction p of the work in parallel.

Amdahl’s law

Assume perfect[1] strong scalability, i.e.,

$$T^N_{f,p} = \frac{T^1_{f,p}}{N}.$$

Then the speedup is

$$S^N = \frac{T^1_f}{T^N_f} = \frac{T^1_f}{T_{f,s} + T^N_{f,p}} = \frac{T^1_f}{T_{f,s} + \frac{T^1_{f,p}}{N}} = \frac{T^1_f}{T_{f,s} + \frac{T^1_f - T_{f,s}}{N}}.$$

[1] In some circumstances, ***superlinear*** scaling can be observed, i.e., $T^N_{f,p} < T^1_{f,p}/N$.

Amdahl's law

After simplifying, we obtain

$$S^N = \frac{T_f^1}{T_{f,s} + \frac{T_f^1 - T_{f,s}}{N}} = \frac{T_f^1}{\left(1 - \frac{1}{N}\right) T_{f,s} + \frac{T_f^1}{N}} = \frac{1}{\left(1 - \frac{1}{N}\right) s + \frac{1}{N}},$$

i.e.,

$$S^N = \frac{1}{s + \frac{1-s}{N}}$$

(Amdahl's law).

Amdahl's law

Amdahl's law tells us the speedup assuming perfect strong scalability.

In particular, it implies that the speedup is limited, because

$$S^N = \frac{1}{s + \frac{1-s}{N}} \xrightarrow{N\to\infty} \frac{1}{s}.$$

Amdahl's law

Example: Suppose that 90% of an application's work can be done in parallel, while 10% ($s = 0.1$) must be done sequentially.

Assuming perfect strong scalability, the speedup achieved when using $N = 10$ workers instead of one is

$$S^N = \frac{1}{s + \frac{1-s}{N}} = \frac{1}{0.1 + \frac{0.9}{10}} \approx 5.26.$$

Furthermore, the speedup can never exceed

$$\frac{1}{s} = \frac{1}{0.1} = 10.$$

Amdahl's law

More generally, Amdahl's law says that

$$S = \frac{1}{s + \frac{1-s}{S_p}},$$

where S_p is the speedup experienced by the fraction p of the total amount of work.

So far we assumed $S_p = N$.

Amdahl’s law

In its general form, Amdahl’s law is useful in many contexts.

Example: By code optimization, 43% of the total work in a sequential code can be accelerated by a factor of 1.6.

Then the speedup for the overall application is

$$S = \frac{1}{0.57 + \frac{0.43}{1.6}} \approx 1.19.$$

Amdahl’s law

In its general form, Amdahl’s law is useful in many contexts.

Example: An application spends 17% of its time reading from and writing to a hard drive.

The hard drive is replaced with a newer drive that is faster by a factor of 3.2.

Then the speedup for the overall application is

$$S = \frac{1}{0.83 + \frac{0.17}{3.2}} \approx 1.13.$$