SF2568: Parallel Computations for Large-Scale Problems

Lecture 2: Parallel Techniques and Performance Evaluation

January 18, 2024

Acknowledgements

These slides are an extension of slides by Michael Hanke and Niclas Jansson.

A First Example: Rank Sort

Problem

Given n (pairwise distinct) numbers. Sort the numbers in increasing order.

Solution

- Let the **rank** r_i of an element a_i of the set M of numbers be the number of elements being less than that element,

$$r_i = \#\{a_j \in M | a_j < a_i\}$$

- In a fully sorted list, the position of element i is just its rank r_i.

Rank Sort – Algorithm

Fortran Version

```fortran
do i = 1, n
  r(i) = 0
  do j = 1, n
    if (a(j) .lt. a(i)) then
      r(i) = r(i) + 1
    end if
  end do
end do
```

C version

```c
for (int i = 0; i < n; i++) {
  r[i] = 0;
  for (int j = 0; j < n; j++) {
    if (a[j] < a[i]) {
      r[i]++;
    }
  }
}
```

How to parallelize either of these versions?

What do we look for?

Rank Sort – Parallelization

Observation:

- The innermost loops (and the initialization) are completely independent of each other

```
do i = 1, n
  r(i) = 0
  do j = 1, n
    if (a(j) .lt. a(i)) then
      r(i) = r(i) + 1
    end if
  end do
end do
```

Parallelization idea:

- Assume that we have n processors
- Then the inner part can be assigned to processor i
- Once r_i is known, processor i can put element a_i in the correct position

```
r(i) = 0
do j = 1, n
  if (a(j) .lt. a(i)) then
    r(i) = r(i) + 1
  end if
end do
```

Definitions

Definition

A **processor** is a physical hardware device that has the capability of accessing the memory and computing new data values.

Definition

A **process** is a computational activity assigned to a processor.

The process is defined by the program code and the data it works on. Note that, on modern operating systems, every processor executes many processes in parallel (usually more than it has computational nodes)

How Efficient is our Algorithm

Definition

- $\mathbf{T_S^*}$ denotes the execution time of the fastest serial algorithm
- The **parallel runtime $\mathbf{T_P}$** on P processors is the time which elapses from the moment a parallel execution starts to the moment the last processing element finishes execution
- The **parallel speedup $\mathbf{S_P}$** is the quotient of T_p and T_S^*,

$$S_p = \frac{T_S^*}{T_P}$$

How Efficient is our Algorithm

- The best possible speedup is $S_P = P$
- By definition,

$$T_S^* \leq T_1,$$

  the execution time of the parallel algorithm on one processor.
- Equality does not hold in general
- Fortunately, for many algorithms we have $T_S^* \approx T_1$
- Estimating the efficiency of a parallel program includes also an estimation of **memory** and **communication** overhead

Rank Sort – Speedup

- Estimating the complexity of an algorithm is a very complex problem. Often, we can only estimate the leading term for a large number n of data. The big-Oh notation becomes handy:

Definition

It holds $f(n) = \mathcal{O}(g(n))$ if and only if $f(n) \leq c\, g(n)$ for some constant c and all sufficiently large n.

  - We will silently assume that the bound g is the smallest possible.

Rank Sort – Speedup

- Let's analyse the algorithm
  - What is the complexity?
- The execution time on one processor is
$$T_1 = \mathcal{O}(n^2)$$

```
do i = 1, n
  r(i) = 0
  do j = 1, n
    if (a(j) .lt. a(i)) then
      r(i) = r(i) + 1
    end if
  end do
end do
```

- Assume that we have a **shared memory machine** (multiprocessor). In that case, there is no additional memory needed and no communication overhead. The body of the i-loop has an execution time,
$$t_i = (1 + n + r_i)\tau_a = \mathcal{O}(n)$$
where τ_a is the time for one arithmetic operation
- The parallel execution time is, for $n \leq P$,
$$T_p = \max_{0 \leq i < n} t_i = 2n$$

Rank Sort – Speedup

Fact

The best known sequential (comparison based) sorting algorithm has a complexity

$$T_S^* = \mathcal{O}(n \log n)$$

Corollary

Rank sort on a shared memory machine has a parallel speedup of

$$S_P = \frac{T_S^*}{T_P} = \mathcal{O}(\log P)$$

which is obtained for $n = P$

Remember: Optimal speedup is $S_P = P$

Rank Sort on Distributed Memory Machines

Problem

Each process need the complete data set M.
How do the processes obtain access to the data?

Assume that one process $(p = 0)$ holds M. An algorithm could like:

1. Process $p = 0$ sends M to all other processes
2. All Processes $p > 0$ receive M
3. On all processes $p = i$, the rank r_i is computed
4. All processes send r_i to process 0
5. On process 0, the sorted sequence is available

Message Passing Primitives

- A means for starting a number of processes on different PEs (processors)
- Routines for sending data to other PEs
- Routines for receiving data from other PEs
- Synchronization routines

The Message Passing Interface (MPI)

- The most commonly used paradigm is **message passing**. The data items are encapsulated into messages which are exchanged between PEs
- The programming strategy is SIMD

Rank Sort – Efficiency on Distributed Memory Machines

Without going into details in this lecture, we observe:

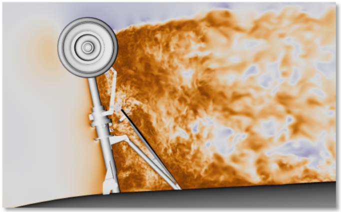
- Memory efficiency is bad because all **data is duplicated** in all PEs
- There is a considerable **communication overhead**

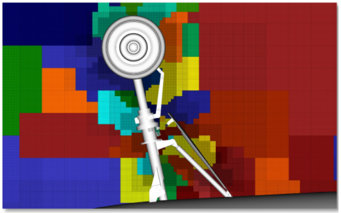
Data Parallelism

- A large number of different data items are subjected to identical or similar processing, all in parallel
- Example: rank sort, vector operations

Data Partitioning

- Special type of data parallelism
- The data space is naturally **partitioned into adjacent regions**
- Each region is operated on in parallel by a different processor
- **Examples:** many numerical algorithms, image processing etc


Computed solution

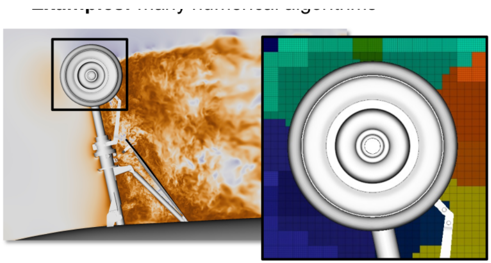
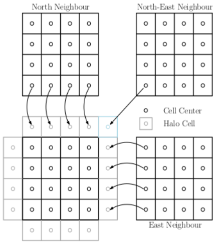

Data Partitioning

Relaxed Algorithm

- Also known as embarrassingly parallel
- Each process computes in a self-sufficient manner with no synchronisation or communication between processes
- **Trivial to implement, ideal performance**
- **Examples:** rank sort, Monte Carlo algorithms, fractals

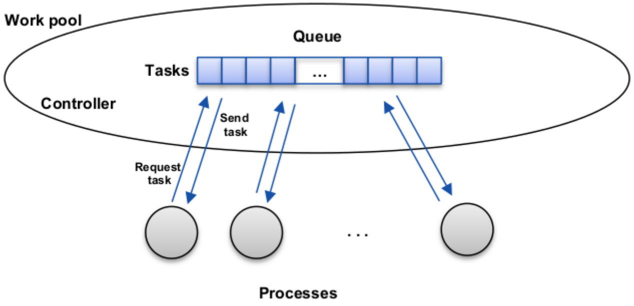
Synchronous Iteration

- Each processor performs the same iterative computation, but on a different portion of data
- However, the processors must be synchronized at the end of each iteration
- **Examples:** many numerical algorithms

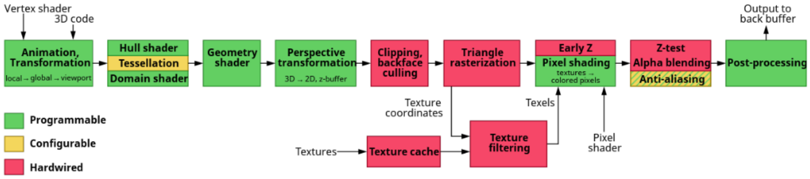
Replicated Workers

- A central pool of similar tasks is maintained
- A number of worker processes retrieve tasks from the pool
- The computations ends when the task pool is empty
- **Examples**: combinatorial problems, data base queries,...



Pipelined Computation

- The processes are arranged in a structure
- Each process performs a certain phase of the computation
- **Examples**: microprocessors, graphics processing units, . . .



Graphics rendering pipeline

Sources of Inefficiency

- Excessive sequential code
- Communication delay
- Synchronization delay
- Communication/memory contention
- Process creation time

Scalability

- Various factors may limit performance as we increase the number of processing elements P

Definition

Scalability of a parallel system is a measure of its capacity to increase speedup in proportion to the number of processing elements.

- We say that a computation is **scalable** if performance increases linearly with P, or maybe even $\mathcal{O}(P/\log(P))$

Limits to Scalability

- Periods, when not all processors can perform useful work and are simply idle
- Extra computations (overhead) in the parallel version not appearing in the sequential version; for example, recomputing constants locally
- “Non-productive” work, e.g., communication

Question

What is the maximum speedup?

Amdahl's Law (1967)

- Assume that, for a given algorithm, there is a certain fraction f of the computation which cannot be divided into concurrent tasks (**serial fraction**)
- The serial part is then $t_s = f\,T_1$
- On P processing elements, we can only eliminate $(1-f)T_1$ of the runtime,

$$T_P = f\,T_1 + \frac{(1-f)T_1}{P}$$

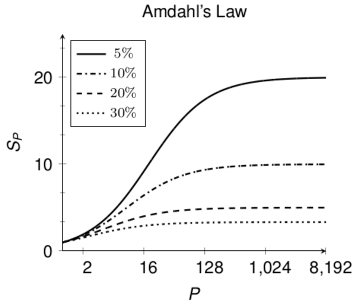
- The speedup becomes

$$S_P = \frac{T_1}{f\,T_1 + (1-f)T_1/P} = \frac{P}{1+(P-1)f}$$

Conclusion:
Even for an infinite number of processors, **the maximum speedup is limited by 1/f**,

$$\lim_{P\to\infty} S_P = \frac{1}{f}$$

Amdahl's Law (1967)



Amdahl's law led many to take a pessimistic outlook onto the benefits of parallelism

Question:

What could be wrong with this reasoning?

Scaled Speedup

Observation

Amdahl's law assumes that the workload W remains fixed

Fact

But parallel computers are used to tackle more ambitious workloads:

- W increases with P
- f often decreases with W

Gustafson's Law (1988)

- Assume that the **parallel** execution time T_P is constant
  - For Amdahl's law, the sequential execution time is fixed
- Let f' be the fraction of serial time spent in the parallel program
- Now it holds $T_1 = f'T_P + (1 - f')P\,T_P$
- Scaled speedup

$$S'_P = f' + (1 - f')P$$

- **The scaled speedup can become arbitrarily large**

Common Metrics in Parallel Computing

Definition

- The parallel efficiency η_P is given by

$$\eta_P = \frac{S_P}{P}$$

- The cost C is given by

$$C = T_P \, P$$

SIMD/SPMD Execution Model

- The common way parallel programming is implemented is via SIMD (also know as Single Program, Multiple Data)
- Programs execute as a set of P processes
  - We specify P when we run the program
  - Each process is usually assigned to a different physical processor
- Each process
  - Is initialized with the same code
  - Has an associated **rank**, a unique integer in the range $0, \ldots, P-1$
  - Executes instructions at its own rate
- Processes communicate via messages or shared memory (we'll assume **message passing**)
- The sequence of instructions each process executes depends on the process' rank and the outcome of the communication

Parallel Execution Time

- With the computation time t_{comp} and the communication time t_{comm}, it holds

$$T_P = t_{comp} + t_{comm}$$

- Between two synchronisation points, we have

$$t_{comp,i} = f_i(n, P)$$

- The overall computation time becomes

$$t_{comp} = t_{comp,1} + t_{comp,2} + t_{comp,3} + \dots$$

- Time to execute one basic operation: t_a (order 1 ns)

Communication Time

- Similar to computation time, the total communication time is the sum of all individual communication steps,

$$t_{comm} = t_{comm,1} + t_{comm,2} + t_{comm,3} + \dots$$

- In a good approximation, it holds

$$t_{comm,i} = t_{\mathsf{startup}} + w\, t_{\mathsf{data}}$$

where

  - t_{startup} is the startup time, or latency
  - t_{data} is the time needed to send one word (bandwidth)
  - w is the number of words to be sent

Communication Time

- Typical figures are:
  - t_{startup} is of the order 1μs
  - t_{data} is of the order 1 ns
  - Computational speed (much) larger than 1 GFlop/s
- This is the Achilles' heel of message passing
- **Send only large blocks as seldom as possible!**

- The simple model for communication time is based on the assumption that any two processors have the *same distance* from each other, i.e., the exchange of identical information takes a time independent of the processors involved.

In practice, this is far from being true!

Mandelbrot set

Definition

Let f be a function of two complex variables. For any $z \in \mathbb{C}$ consider the iteration

$$z_{k+1} = f(z_k, d).$$

A **Mandelbrot set** is a set of points d in the complex plane that are quasi-stable, i.e., the sequence $\{z_k\}$ remains bounded for $z_0 = 0$.

Problem

Visualise a Mandelbrot set in the complex plane for the function

$$f(z) = z^2 + d.$$