Parallel Computing

Jingyuan Sun
ChatGPT Claude

Dec, 2024

1 Sources of Overhead in Parallel Programs

1.1 Parameters

In parallel programming, the runtime of a program depends on several parameters that directly influence performance:

- **Wall Clock Time**:
  - **Definition**: The time from the start of the first processor to the stopping time of the last processor in a parallel ensemble.
  - **Explanation**: It is the most intuitive metric to calculate the real execution time of a program, including communication delays and processor waiting times.
- **Parallel Runtime**:
  - **Definition**: The parallel runtime depends on the input size, number of processors, and communication parameters of the machine.
  - **Explanation**: Although parallel runtime is generally shorter than serial runtime, it is affected by various factors such as synchronization delays among processors.
- **Speedup and Efficiency**:
  - **Definitions**:
  $$\text{Speedup} = \frac{\text{Serial Runtime}}{\text{Parallel Runtime}}, \quad \text{Efficiency} = \frac{\text{Speedup}}{\text{Processors}}$$
  - **Explanation**: Speedup measures the acceleration achieved by parallel algorithms, and Efficiency quantifies how effectively the computational resources are utilized.
- **Raw FLOP Count**:
  - **Definition**: FLOP (Floating Point Operations Per Second) measures the computational capacity of a program but is insufficient to evaluate performance in practical problems.
  - **Explanation**: In solving real-world problems, factors like communication delay and data synchronization must also be considered.

1.2 Sources of Overhead in Parallel Programs

The performance of parallel programs is often limited by the following factors:

- **Idling (Processor Idle Time)**:
  - **Definition**: Processors enter idle states while waiting for other processors to complete their tasks.
  - **Explanation**: Similar to workers waiting for others to finish before continuing their own work.
- **Communication Overhead**:
  - **Definition**: Overhead arises due to data sharing or synchronization among processors.
  - **Explanation**: Like workers who need to confirm their progress with each other, causing delays.
- **Synchronization Overhead**:
  - **Definition**: To ensure task order is correct, processors must synchronize, leading to waiting times.
  - **Explanation**: Similar to workers ensuring tasks are completed in the correct sequence, which may involve waiting.
- **Excessive Computations**:
  - **Definition**: Some processors perform excessive calculations, causing delays compared to others.
  - **Explanation**: Similar to one worker being assigned too much work while others remain idle.

2 Performance Metrics for Parallel Algorithms

2.1 Serial Runtime T_s

The serial runtime of a program is the time elapsed between the beginning and the end of its execution on a sequential computer. It is denoted by T_s.

2.2 Parallel Runtime T_p

The parallel runtime is the time that elapses from the moment the first processor starts to the moment the last processor finishes execution. It is denoted by T_p.

2.3 Total Parallel Overhead

The total time collectively spent by all the processing elements is given by:

$$T_{all} = p \cdot T_p,$$

where p is the number of processors.

The total overhead is defined as:

$$T_o = T_{all} - T_s = p \cdot T_p - T_s.$$

2.4 Speedup S

Speedup measures the improvement gained by parallel execution compared to serial execution. It is defined as:

$$S = \frac{T_s}{T_p}.$$

In theory, the speedup is bounded by $0 < S \leq p$.

2.5 Efficiency E

Efficiency measures the fraction of time for which processing elements are usefully employed. It is given by:

$$E = \frac{S}{p},$$

where $0 < E \leq 1$.

2.6 Example: Adding Numbers

To sum n numbers using n processors arranged in a logical binary tree:

- Total steps required: $\log n$.
- Parallel runtime:

$$T_p = \Theta(\log n).$$

- Speedup:

$$S = \frac{\Theta(n)}{\Theta(\log n)} = \Theta\left(\frac{n}{\log n}\right).$$

- Efficiency:

$$E = \frac{S}{p} = \Theta\left(\frac{1}{\log n}\right).$$

2.7 Improved Example: Distributed Addition

Each processing element adds $\frac{n}{p}$ numbers locally, followed by aggregation:

- Local computation time:

$$\Theta\left(\frac{n}{p}\right).$$

- Aggregation time:
$$\Theta(\log p).$$
- Total parallel runtime:
$$T_p = \Theta\left(\frac{n}{p} + \log p\right).$$
- Speedup:
$$S = \frac{\Theta(n)}{\Theta\left(\frac{n}{p} + \log p\right)}.$$
- Efficiency:
$$E = \frac{S}{p}.$$

2.8 Scaling Characteristics of Parallel Algorithms

For the problem of adding n numbers on p processing elements:

$$T_p = \frac{n}{p} + 2\log p, \quad S = \frac{n}{\frac{n}{p} + 2\log p}, \quad E = \frac{1}{1 + \frac{2p\log p}{n}}.$$

2.8.1 Key Formula Explanations

1. **Parallel Runtime $T_p = \frac{n}{p} + 2\log p$:**

- $\frac{n}{p}$: Time for each processor to compute its local portion of the input.
- $2\log p$: Time for aggregating results in a binary tree structure (logarithmic depth with 2 communication rounds per level).

2. **Speedup $S = \frac{n}{\frac{n}{p}+2\log p}$:**

- Numerator (n): Represents the serial time.
- Denominator $(\frac{n}{p} + 2\log p)$: Represents the total parallel time.
- Overall: Measures the improvement in runtime due to parallelization.

3. **Efficiency $E = \frac{1}{1+\frac{2p\log p}{n}}$:**

- 1: Ideal efficiency when there is no overhead.
- $\frac{2p\log p}{n}$: Captures the effect of communication overhead relative to problem size n.
- Overall: Highlights that efficiency decreases as overhead increases or problem size decreases.

2.8.2 Graphical Insight

- Speedup increases with n if T_o grows sublinearly with T_s. - Efficiency remains constant by simultaneously increasing n and p.

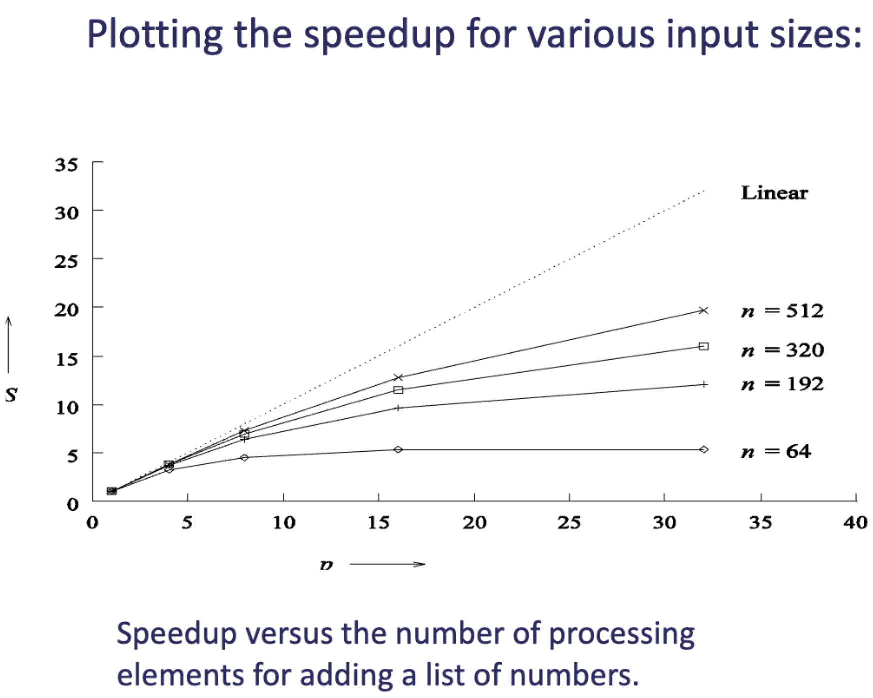

Figure 1: Speedup-Processors

3 Isoefficiency Metric of Scalability

3.1 Definition and Key Concepts

The **isoefficiency metric** is used to evaluate the scalability of a parallel system. It defines the rate at which the problem size W must grow with respect to the number of processing elements p to maintain a constant efficiency E. This metric helps determine how effectively a parallel system can scale.

3.1.1 Efficiency Behavior

- **Graph (a):** When the problem size W is fixed, efficiency E decreases as the number of processors p increases.
- **Graph (b):** When the number of processors p is fixed, efficiency E increases as the problem size W grows.

3.2 Key Formulas

3.2.1 Parallel Runtime

$$T_P = \frac{W + T_o(W, p)}{p} \tag{1}$$

- W: Total computational work (problem size).
- $T_o(W, p)$: Total overhead as a function of W and p.
- T_P: Total parallel runtime.

3.2.2 Speedup

$$S = \frac{W}{T_P} = \frac{Wp}{W + T_o(W,p)} \tag{2}$$

3.2.3 Efficiency

$$E = \frac{S}{p} = \frac{W}{W + T_o(W,p)} = \frac{1}{1 + \frac{T_o(W,p)}{W}} \tag{3}$$

- Efficiency E depends on the ratio $\frac{T_o(W,p)}{W}$.

3.2.4 Isoefficiency Function

To maintain constant efficiency E, the overhead ratio $\frac{T_o(W,p)}{W}$ must be constant:

$$W = \frac{E}{1-E} T_o(W,p) \tag{4}$$

- $K = \frac{E}{1-E}$: A constant dependent on the desired efficiency.

3.2.5 Asymptotic Isoefficiency

Substituting $T_o(W,p) \approx 2p \log p$, we get:

$$W = K2p \log p \tag{5}$$

- The asymptotic isoefficiency function for this parallel system is $\Theta(p \log p)$.

3.3 Scalability Interpretation

A scalable parallel program maintains constant efficiency by increasing the problem size proportionally to the isoefficiency function. For instance, if the number of processors increases from p to p', the problem size W must increase by a factor of $\frac{p' \log p'}{p \log p}$ to maintain efficiency.

3.4 Amdahl's Law and Isoefficiency

For fixed problem sizes, scalability is limited by the sequential portion of the workload:

$$S_p = \frac{W}{\alpha W + \frac{(1-\alpha)W}{p}} \tag{6}$$

- α: Fraction of the workload that is sequential.

As $p \to \infty$, the speedup is limited by:

$$\text{Speedup is limited by } \frac{1}{\alpha}. \tag{7}$$

4 Communication-Avoiding Algorithms and Scalability

4.1 Naïve Matrix Multiplication

The **naïve matrix multiplication** computes the product $C = A \times B$, where A, B, and C are $n \times n$ matrices. The element $C(i, j)$ is computed as:

$$C(i, j) = \sum_{k=1}^{n} A(i, k) \cdot B(k, j)$$

4.1.1 Complexity Analysis

- **Arithmetic Operations:** $O(n^3)$ scalar multiplications and additions.
- **Memory Access:**
  - n^2 reads for rows of A.
  - n^2 reads for columns of B.
  - n^2 writes for C.
  - Total memory reads/writes: $n^3 + 3n^2$.

4.1.2 Pesudo Code

The naive algorithm computes C by iterating through rows of A and columns of B:

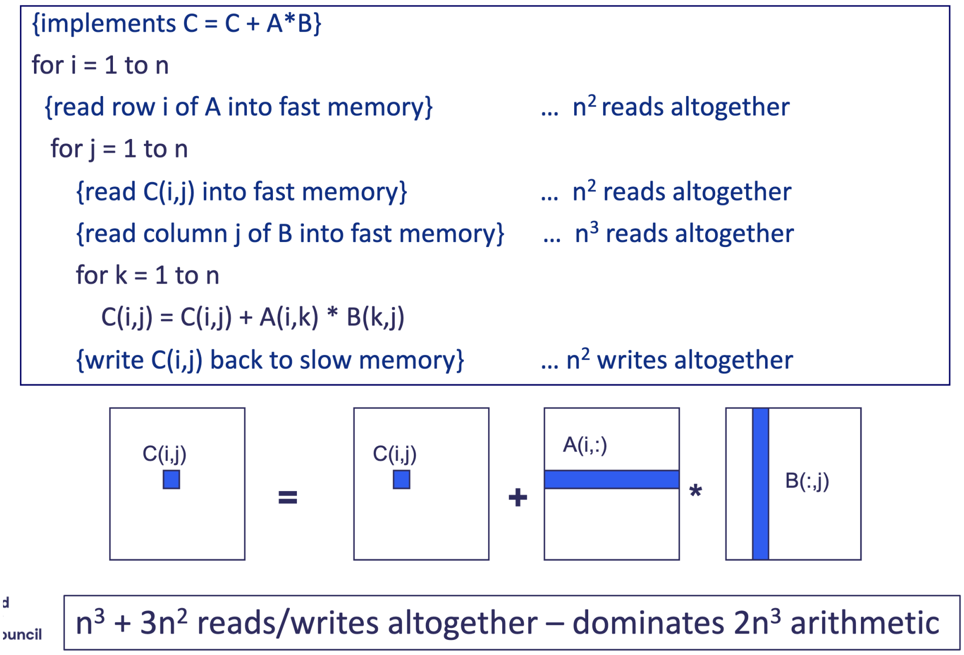

Figure 2: Caption

4.2 Blocked (Tiled) Matrix Multiply

To reduce memory traffic, blocked matrix multiplication divides A, B, and C into $b \times b$ blocks.

4.2.1 Algorithm

Each block of C is computed using corresponding blocks of A and B:

$$C[i,j] += A[i,k] \cdot B[k,j], \quad \text{for all blocks } i,j,k.$$

4.2.2 Complexity Analysis

- **Arithmetic Operations:** $O(n^3)$.
- **Memory Access:**
$$O\left(\frac{2n^3}{b} + 2n^2\right)$$
due to reduced redundant memory reads and writes.

4.2.3 Pesudo Code

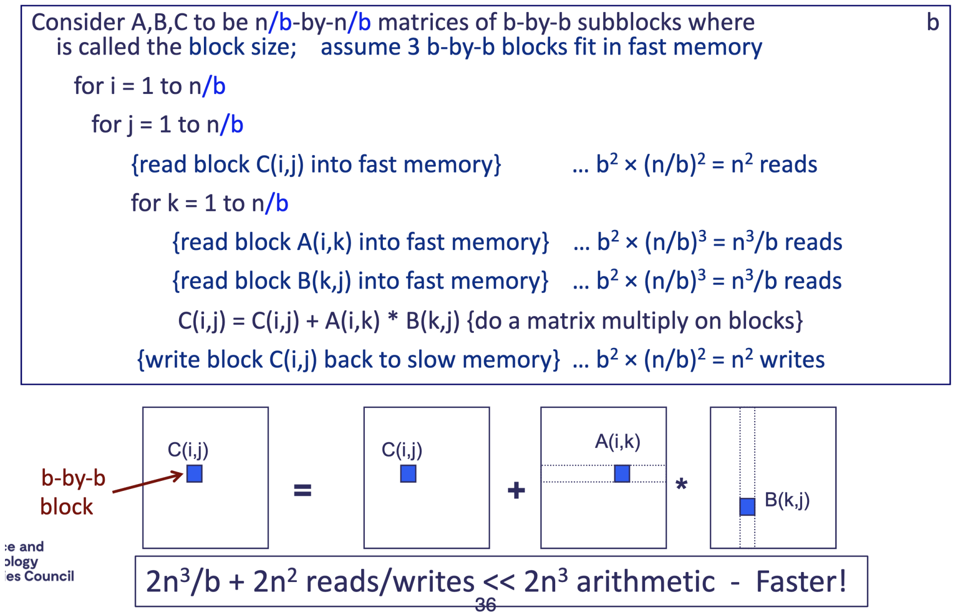

Figure 3: Caption

4.3 SUMMA Algorithm

The **Scalable Universal Matrix Multiply Algorithm (SUMMA)** optimizes matrix multiplication for distributed memory systems. It uses a $P^{1/2} \times P^{1/2}$ processor grid.

4.3.1 Algorithm

- Divide A, B, and C into blocks.
- Broadcast submatrices of A and B along processor rows and columns.
- Compute partial results for each block in parallel.

4.3.2 Complexity Analysis

- **Arithmetic Operations:** $O(n^3)$.
- **Communication Cost:**
  - Words moved: $O(n^2 \log P)$.
  - Messages sent: $O(\log P)$.

4.3.3 Python Implementation

```
def summa_matrix_multiply(A, B, block_size, processor_grid):
    n = len(A)
    C = np.zeros((n, n))
    p = len(processor_grid)  # Number of processors

    # Iterate over blocks
    for k in range(0, n, block_size):
        for i in range(0, n, block_size):  # Processor rows
            for j in range(0, n, block_size):  # Processor columns
                # Broadcast blocks
                A_block = A[i:i+block_size, k:k+block_size]
                B_block = B[k:k+block_size, j:j+block_size]

                # Compute local block
                C[i:i+block_size, j:j+block_size] += A_block @ B_block
    return C
```

4.4 Comparison of Methods

Method	Arithmetic Complexity	Memory Access	Communication Cost
Naïve Multiply	$O(n^3)$	$O(n^3 + 3n^2)$	$O(n^3)$ (sequential)
Blocked Multiply	$O(n^3)$	$O(2n^3/b + 2n^2)$	$O(n^2)$
SUMMA	$O(n^3)$	$O(n^2 \log P)$	$O(\log P)$

Table 1: Comparison of Matrix Multiplication Algorithms