

HPC – Unit 4: Performance Metrics & Matrix Computation

May–June 2023 (Paper [6004]-493)

Q3 a) Parallel Matrix-Matrix Multiplication Algorithm with Example [7 Marks]

Matrix-matrix multiplication $C = A \times B$ where A is an $m \times k$ matrix and B is a $k \times n$ matrix. The sequential algorithm requires $O(m \cdot k \cdot n)$ operations. Parallel algorithms distribute either rows, columns, or sub-blocks across processors.

Method 1: Row-wise Distribution

Each of the p processors is assigned m/p rows of matrix A . Every processor needs all of B to compute its rows of C .

- Phase 1 (Broadcast B): One-to-all broadcast B to all processors — cost: $O(\log p \times k \cdot n)$.
- Phase 2 (Local multiply): Each processor computes (m/p) rows of C independently.
- Parallel time: $T_p \approx (m \cdot k \cdot n)/p + \log p \times k \cdot n$.

Example: Multiply two 4×4 matrices on 2 processors.

$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 0 & 1 & 2 \\ 3 & 4 & 5 & 6 \end{bmatrix}$	$B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
--	--

P_0 gets rows 0,1 of $A \rightarrow$ computes rows 0,1 of $C =$ rows 0,1 of A (since $B=I$)

P_1 gets rows 2,3 of $A \rightarrow$ computes rows 2,3 of C

Method 2: Block (Cannon's) Algorithm

For square $n \times n$ matrices on p processors arranged in a $\sqrt{p} \times \sqrt{p}$ mesh, Cannon's algorithm avoids replicating the full matrix by using circular shifts.

- Sub-matrices are distributed: processor (i,j) gets block $A[i,j]$ and $B[i,j]$.
- Initial alignment: shift rows of A left by i positions and columns of B up by j positions.
- Each step: processors multiply their local sub-blocks and accumulate into C . Then A shifts left by 1 and B shifts up by 1.
- After \sqrt{p} steps, each processor has its complete $C[i,j]$ block.

Parallel time (Cannon's): $T_p = O(n^3/p) + O(\sqrt{p} \times n^2/p \times t_{\text{comm}})$

For large n , communication is dominated by computation — good scalability.

Note: For SPPU exams, explaining the row-distribution approach with a small numerical example (2×2 or 4×4 with 2 or 4 processors) is usually sufficient for full marks.

Q3 b) Performance Metrics for Parallel Systems

[6 Marks]

Several metrics are used to evaluate how well a parallel program utilizes available processors:

1. Speedup (S)

$$S = T_s / T_p$$

where T_s = best sequential execution time, T_p = parallel execution time on p processors.

- Ideal speedup = p (linear). In practice, $S < p$ due to overhead.
- $S > p$ (superlinear speedup) can occur due to cache effects — each processor's working set fits in cache.

2. Efficiency (E)

$$E = S / p = T_s / (p \times T_p)$$

Ranges from 0 to 1; ideal = 1 (100% efficient).

- Efficiency drops as p increases because overhead (communication, synchronisation, idle time) grows.

3. Cost (C)

$$C = p \times T_p$$

Total processor-time product. A cost-optimal algorithm has $C = O(T_s)$.

- Cost optimality means the parallel algorithm uses total work proportional to the sequential algorithm — no wasted processor cycles.

4. Speedup Factor (Amdahl's Perspective)

$$S \leq 1 / (f + (1-f)/p)$$

where f = fraction of program that is inherently sequential (cannot be parallelised).

5. Utilisation (U)

$$U = (\text{Total useful work}) / (p \times T_p) = E$$

- Utilisation equals efficiency — measures what fraction of processor time is spent on useful computation (vs. overhead).

6. Execution Rate

- Measured in FLOPS (floating-point operations per second) or MIPS.
- Parallel FLOPS = $p \times (\text{sequential FLOPS per processor}) \times \text{efficiency}$.

7. Redundancy

$$R = \text{Total operations in parallel} / \text{Operations in best sequential algorithm}$$

- $R \geq 1$ always; $R = 1$ means no redundant work. Some parallel algorithms intentionally do extra work to reduce communication.

Q3 c) Minimum Execution Time and Minimum Cost-Optimal Execution Time

[4 Marks]

Minimum Execution Time

The minimum execution time T_{\min} is the fastest possible wall-clock time achievable using any number of processors, regardless of cost. It is obtained when we use as many processors as the problem's parallelism allows.

$$T_{\min} = \lim_{p \rightarrow \infty} T_p$$

For a problem with a sequential fraction f (Amdahl's Law):

$$T_{\min} = f \times T_s \quad (\text{the inherently serial portion cannot be parallelised})$$

- Using more processors beyond a point gives diminishing returns because the sequential bottleneck dominates.

Minimum Cost-Optimal Execution Time

A parallel algorithm is cost-optimal if the total cost $C = p \times T_p$ equals the sequential time T_s (asymptotically). The minimum cost-optimal execution time T_{\min_cost} is the fastest time achievable while keeping cost optimal:

T_{\min_cost} is achieved when p is chosen such that:
 $p \times T_p = \Theta(T_s)$ (cost matches sequential work)

For example, sorting n elements sequentially: $T_s = O(n \log n)$

Parallel bitonic sort on n processors: $T_p = O(\log^2 n)$

Cost = $n \times \log^2 n > n \log n \rightarrow$ not cost-optimal.

Using $p = n/\log n$ processors gives $T_p = O(\log n \times \log(n/\log n)) \approx O(\log^2 n)$ with cost $O(n \log n) \rightarrow$ cost-optimal.

- Cost-optimal algorithms are preferred in practice because they are economically efficient — you get maximum speedup per unit of hardware cost.

Note: Minimum execution time is about speed; minimum cost-optimal execution time balances speed and resource cost. Both are key concepts in algorithmic complexity for parallel systems.

Q4 a) Granularity and Its Effects on Performance

[7 Marks]

Granularity refers to the ratio of computation to communication in a parallel program. It determines how work is partitioned among processors.

Types of Granularity

- Fine-grained: Very small tasks per processor. Many small computations, frequent communication. High communication overhead relative to computation.
- Coarse-grained: Large tasks per processor. Few, infrequent communications. Reduced parallelism (fewer concurrent pieces).
- Medium-grained: A balance that gives good speedup and manageable communication overhead — often the sweet spot.

Granularity = (Computation time per processor) / (Communication time)

Fine: ratio $\ll 1$ (communication dominates — bad)

Coarse: ratio $\gg 1$ (underutilised parallelism — also potentially bad)

Optimal: ratio $\approx O(1 \text{ to } 10)$ depending on interconnect speed

Effect of Granularity on Performance — Adding n Numbers on p Processors

Consider summing n numbers distributed across p processors:

Phase 1 (local sum): Each processor sums its n/p numbers.

$$T_{\text{local}} = n/p \times t_{\text{add}}$$

Phase 2 (tree reduction): $\log_2(p)$ rounds of communicate-and-add

$$T_{\text{comm}} = \log_2(p) \times (t_s + t_w)$$

Total: $T_p = n/p \times t_{\text{add}} + \log_2(p) \times t_s$

Speedup: $S = (n \times t_{\text{add}}) / (n/p \times t_{\text{add}} + \log_2(p) \times t_s)$

- As n increases (coarser grain for fixed p): speedup approaches p — good efficiency.
- As p increases for fixed n (finer grain): local work shrinks, communication dominates, efficiency drops.
- As $n/p \rightarrow 0$: $T_p \approx \log_2(p) \times t_s$ — all communication, no useful work.

Note: This example directly shows why granularity matters: a fine-grained decomposition (small n , large p) can actually run slower than a coarser partition. Always tune granularity based on the ratio of computation to communication bandwidth.

Q4 b) Sources of Overhead in Parallel Systems

[6 Marks]

Overhead T_o is the extra time spent by a parallel program over and above the best sequential execution. Formally, $T_o = p \times T_p - T_s$. It comes from several sources:

1. Inter-Process Communication

Moving data between processors is the dominant overhead in most parallel programs. Each message incurs a startup cost (latency) and a per-byte transfer cost. As p increases, more messages are needed, driving up this overhead.

2. Load Imbalance

If the work is not evenly distributed, some processors sit idle while others are still computing. The parallel time is determined by the slowest (busiest) processor. Load imbalance overhead = $(T_{\max} - T_{\text{avg}}) \times p$.

3. Synchronisation Overhead

Barriers, locks, and other synchronisation primitives cause processors to wait for each other. Even a short delay at a barrier ripples across all p processors.

4. Extra Computation

Some parallel algorithms do more total work than the best sequential algorithm (e.g., sorting networks). This redundant computation is overhead. Quantified by the redundancy metric R .

5. Task Scheduling and Management

In dynamic parallel frameworks (OpenMP task pools, work-stealing schedulers), there is overhead in creating, scheduling, and terminating tasks. For very fine-grained tasks this can dominate.

6. Memory Access and Cache Effects

In shared-memory systems, false sharing (two processors share a cache line but access different variables) causes unnecessary cache invalidations, stalling processors.

Summary: $T_o = T_{\text{comm}} + T_{\text{idle}} + T_{\text{sync}} + T_{\text{redundant}} + T_{\text{sched}}$

Total parallel time: $T_p = (T_s + T_o) / p$

Efficiency: $E = T_s / (T_s + T_o) \rightarrow \text{lower overhead} \Rightarrow \text{higher efficiency}$

Q4 c) Scaling Down (Downsizing) a Parallel System

[4 Marks]

Scaling down refers to taking a parallel algorithm designed for p processors and running it on $p' < p$ processors (fewer processors than originally designed for). This is common when the full machine is not available or when running smaller problem sizes.

Technique: Virtual Processors (Folding)

Each physical processor simulates multiple virtual processors. If the original algorithm used $p = 64$ processors and only $p' = 8$ are available, each physical processor handles $p/p' = 8$ virtual nodes.

Example: 1D prefix-sum designed for $p=8$ nodes, now on $p'=2$ nodes.

Each physical node simulates 4 virtual nodes.

Physical node 0 handles virtual nodes {0,1,2,3}
 Physical node 1 handles virtual nodes {4,5,6,7}

Intra-node communication becomes local computation (no network needed).

Inter-node communication only when virtual nodes on different physical nodes must talk.

Effect on performance:

- Computation time increases by factor p/p' (each processor does more work).
- Communication time may decrease (some messages become local).
- Net effect: $T_{p'} \leq (p/p') \times T_p$ — often better than expected because of eliminated communication.
- Efficiency can actually improve when scaling down, since fewer boundary communications are needed.

Note: Scaling down is an important practical technique for running HPC codes on smaller clusters without rewriting the algorithm. It demonstrates the ISO-efficiency relationship: maintaining efficiency as both p and problem size change.

May–June 2024 (Paper [6263]-94)

Q3 a) Sources of Overhead in Parallel Systems

[7 Marks]

[REPEATED] – See Q4 b) in May–June 2023 above for the complete answer.

Q3 b) Effect of Granularity on Performance – Adding n Numbers on p Processors

[6 Marks]

[REPEATED] – See Q4 a) in May–June 2023 above for the complete answer (includes the n -number addition example).

Q3 c) Amdahl's Law and Gustafson's Law

[4 Marks]

Amdahl's Law

Amdahl's Law gives an upper bound on the speedup achievable by parallelising a program, given that a fraction f of the program is inherently sequential:

$$S(p) \leq 1 / (f + (1-f)/p)$$

$$\text{As } p \rightarrow \infty: S_{\max} = 1/f$$

Example: If $f = 0.1$ (10% sequential), $S_{\max} = 10x$ — even with infinite processors.

- Implication: even a small sequential section sharply limits speedup.
- Pessimistic view — assumes problem size is fixed as p grows.

Example table for $f = 0.1$:

Processors (p)	Speedup S
1	1.0

2	1.82
4	3.08
8	4.71
16	6.40
∞	10.0

Gustafson's Law (Scaled Speedup)

Gustafson argued that as we add more processors, we typically also increase the problem size (scaled workload). The sequential fraction then becomes a smaller portion of the total work:

$$\text{Scaled speedup } S_{\text{scaled}} = p - f(p - 1)$$

where f = fraction of parallel runtime that is sequential.

Equivalently: $S_{\text{scaled}} = p - \alpha(p-1)$ where α = sequential fraction of parallel runtime.

- Gustafson's view is optimistic: with larger problems, the parallel portion dominates and speedup can scale linearly with p .
- More realistic for HPC workloads where scientists always want to solve bigger problems when given more processors.

Comparison:

Aspect	Amdahl's Law	Gustafson's Law
Problem size	Fixed	Scales with p
Focus	Strong scaling	Weak scaling
Speedup bound	$1/f$ (hard ceiling)	$p - f(p-1)$ (grows with p)
Outlook	Pessimistic	Optimistic
When accurate	Real-time constraints	Scientific computing

Note: In SPPU exams, always write both formulas, give a numerical example for each, and compare them. Examiners expect both laws in a single question.

Q4 a) Performance Metrics for Parallel Systems

[7 Marks]

[REPEATED] – See Q3 b) in May–June 2023 above for the complete answer. All metrics (Speedup, Efficiency, Cost, Utilisation, Execution Rate, Redundancy) are covered there.

Q4 b) Parallel Matrix-Matrix Multiplication Algorithm with Example

[6 Marks]

[REPEATED] – See Q3 a) in May–June 2023 above for the complete answer.

Q4 c) Scalability of Parallel Systems

[4 Marks]

Scalability measures how effectively a parallel system (algorithm + architecture) maintains its efficiency as both the problem size n and the number of processors p grow simultaneously.

ISO-Efficiency Function

A scalable parallel system keeps efficiency $E = E_0$ (constant) as p increases, provided the problem size W is scaled up at the same rate. The ISO-efficiency function $W(p)$ gives the minimum problem size needed to maintain efficiency E_0 on p processors:

$$E = T_s / (T_s + T_o) = E_0$$

$$\Rightarrow T_o = ((1 - E_0)/E_0) \times W$$

$$\text{So: } W \geq (E_0 / (1 - E_0)) \times T_o(p)$$

Example: If $T_o = O(p \log p)$ (overhead for a broadcast-heavy algorithm), then W must grow as $O(p \log p)$ to keep efficiency constant.

This means the ISO-efficiency function is $W(p) = O(p \log p)$ — scalable.

Types of Scalability

- Strongly scalable: Efficiency stays constant even for fixed problem size. Requires T_o to be small relative to T_s .
- Weakly scalable: Efficiency stays constant only if problem size grows with p . Most practical parallel algorithms fall here.
- Not scalable: T_o grows faster than W for any reasonable scaling of W . Overhead overwhelms useful work.

Factors Affecting Scalability

- Communication overhead — grows with p , degrades scalability.
- Load imbalance — unequal work distribution worsens with heterogeneous tasks.
- Synchronisation points — global barriers become bottlenecks as p grows.
- Memory per processor — if each processor needs access to global data, memory pressure can limit p .

Note: A good rule of thumb: an algorithm with $T_o = O(p)$ has a linear ISO-efficiency function $W(p) = O(p)$ — it scales well. An algorithm with $T_o = O(p^2)$ is poor; W must grow quadratically to maintain efficiency.

Additional Concepts & Quick Reference

Key Formulas Summary

Metric	Formula	Ideal Value
Speedup	$S = T_s / T_p$	p (linear)
Efficiency	$E = S/p = T_s/(p \cdot T_p)$	1 (100%)
Cost	$C = p \times T_p$	$O(T_s)$ (cost-optimal)
Overhead	$T_o = p \cdot T_p - T_s$	0

Amdahl bound	$S \leq 1/(f+(1-f)/p)$	$1/f$ as $p \rightarrow \infty$
Gustafson	$S = p - f(p-1)$	Grows with p

Matrix Multiplication – Complexity Summary

- Sequential: $O(n^3)$ for $n \times n$ matrices (naive); $O(n^{2.37})$ with Strassen-like algorithms.
- Parallel row-distribution on p processors: $T_p = O(n^3/p) + O(n^2 \log p)$ — good for large n .
- Cannon's algorithm on $\sqrt{p} \times \sqrt{p}$ mesh: $T_p = O(n^3/p) + O(n^2/\sqrt{p})$ — near-linear speedup.
- Cost-optimal when $p \leq n^2 / \log n$ (so that communication doesn't dominate).