SCALABILITY ANALYSIS
PERFORMANCE AND SCALABILITY OF PARALLEL SYSTEMS

• **Evaluation**
  - **Sequential:** runtime (execution time)
    \[ T_s = T (\text{InputSize}) \]
  - **Parallel:** runtime (start-->last PE ends)
    \[ T_p = T (\text{InputSize}, p, \text{architecture}) \]
  Note: Cannot be Evaluated in Isolation from the Parallel architecture

• **Parallel System:** Parallel Algorithm \( \propto \) Parallel Architecture

• **Metrics** - Evaluate the Performance of Parallel System

**SCALABILITY:** Ability of Parallel Algorithm to Achieve Performance Gains proportional to number of PE
PERFORMANCE METRICS

1. **Run-Time**: The serial run time \((T_s)\) of a program is the time elapsed between the beginning and the end of its execution on a sequential computer. The parallel run time \((T_p)\) is the time elapsed from the moment that a parallel computation starts to the moment that the last processor finishes execution.

2. **Speedup**:  
   - How much Performance is gained by running the application on "p" (Identical) processors.  
   - Speedup is a measure of relative benefit of solving a problem in parallel.

   \[
   \text{Speedup} \quad (S = \frac{T_s}{T_p})
   \]
PERFORMANCE METRICS

where,

\( T_s \): Fastest Sequential Algorithm for solving the same problem

**IF** – not known yet (only lower bound known)

or

– known, with large constants at runtime that make it impossible to implement

**Then:** Take the fastest known sequential algorithm that can be practically implemented

Formally, Speedup \( S \) is the ratio of the serial run time of the best sequential algorithm for solving a problem to the time taken by the parallel algorithm to solve the same problem on \( P \) processors.

⇒ Speedup – relative metric
(a) Initial data distribution and the first communication step

(b) Second communication step

(c) Third communication step

(d) Fourth communication step

(e) Accumulation of the sum at processing element 0 after the final communication
PERFORMANCE METRICS

- $S \leq p \quad S > p$ (Super linear)

Algorithm of adding “n” numbers on “n” processors (Hypercube)

- Initially, each processor is assigned one of the numbers to be added and, at the end of the computation, one of the processors stores the sum of all the numbers.

- Assuming $n = 16$, Processors as well as numbers are labeled from 0 to 15. The sum of numbers with consecutive labels from $i$ to $j$ is denoted by $\sum_{i}^{j}$. 

- $T_{S} = \Theta(n)$  \quad  $S = \Theta\left(\frac{n}{\log n}\right)$  \quad  (n = p = $2^{k}$)

- $T_{p} = \Theta(\log n)$
Algorithm of adding “n” numbers on “n” processors (Hypercube)

- **Efficiency (E):** measure of how effective the problem is solved on P processors
  \[ E = \frac{S}{P} \]
  \[ E \in (0,1) \]

- Measures the fraction of time for which a processor is usefully employed

- Ideal parallel system can deliver a speedup equal to P processors. Ideal behavior is not achieved because processors cannot devote 100 percent of their time to the computation of the algorithm.
  
  If \( p = n \)
  \[ E = \Theta\left(\frac{1}{\log n}\right) \]
Algorithm of adding “n” numbers on “n” processors (Hypercube)

**Cost:** Cost of solving a problem on a parallel system is the product of parallel run time and the number of processors used.

\[
C_{seq,fast} = T_s \\
C_{par} = p T_p \\
C_{seq,fast} \sim C_{par} \quad \rightarrow \quad \text{Cost-Optimal}
\]

\[
C_{seq,fast} = \Theta(n) \\
C_{par} = \Theta(n \log n) \quad \rightarrow \quad \text{Not Cost-Optimal}
\]

- P = n \rightarrow \text{Fine granularity}
- E = \Theta\left(\frac{1}{\log n}\right)
- P < n \rightarrow \text{Coarse granularity}

Scaling down
**Effects of Granularity on Cost-Optimality**

- Assume: $n$ virtual PEs;
- If $p$ – physical PEs, then each PE simulates $\frac{n}{p}$ – virtual PEs → the computation at each PE increases by a factor: $\frac{n}{p}$

- **Note:** Even if $p<n$, this doesn't necessarily provide Cost–Optimal algorithm
Figure 5.5. Four processing elements simulating 16 processing elements to compute the sum of 16 numbers (first two steps). \( \Sigma^i_j \) denotes the sum of numbers with consecutive labels from \( i \) to \( j \). Four processing elements simulating 16 processing elements to compute the sum of 16 numbers (last three steps).

(a) Four processors simulating the first communication step of 16 processors
(b) Four processors simulating the second communication step of 16 processors
(c) Simulation of the third step in two substeps

(d) Simulation of the fourth step

(e) Final result
Algorithm of adding “n” number on “n” Processors (HYPERCUBE) (p<n)

\[ n = 2^k \quad \text{Eg: n=16, p=4} \]
\[ p = 2^m \]

- Computation + Communication (First 8 Steps)
  \[ \Theta\left( \frac{n}{p} \log p \right) \]

- Computation (last 4 Steps)
  \[ \Theta\left( \frac{n}{p} \right) \]

Parallel Execution Time \( T_{par} \) = \[ \Theta\left( \frac{n}{p} \log p \right) \]

\[ C_{par} = p \Theta\left( \frac{n}{p} \log p \right) = \Theta(n \log p) \]
\[ C_{seq, fast} = \Theta(n) \]

→ P ↑ asymptotic – Not Cost Optimal
Figure 5.6. A cost-optimal way of computing the sum of 16 numbers using four processing elements.
A Cost Optimal Algorithm:

- Compute $\Rightarrow \Theta(\frac{n}{p})$
- Compute + Communic $\Rightarrow \Theta(n)$

$\Theta(\frac{n}{p} + \log p)$

$n > p \log p$

$T_{par} = \Theta\left(\frac{n}{p}\right)$

$C_{par} = \Theta(n) = p \cdot T_{par}$

$C_{seq.f} = T_{seq.fast} = \Theta(n)$

$\Rightarrow$ Cost Optimal
a) If algorithm is cost optimal:

- P – Physical PEs
- Each PE stimulates $\frac{n}{p}$ virtual PEs

Then,

- If the overall communication does not grow more than $\frac{n}{p}$ (Proper Mapping)

- Total parallel run-time grows at most:

$$\frac{n}{p} T_c + \frac{n}{p} T_{comm} = \frac{n}{p} T_{total} = \frac{n}{p} T_p = T_{\frac{n}{p}}$$

$$C_{par}^{p=n} = p T_{P_{(n=p)}}$$

$$C_{par}^{p<n} = p \left( \frac{n}{p} T_p \right) = \frac{n}{p} T_p = n T_p = C_{par}^{p=n}$$

- New algorithm using $\frac{n}{p}$ processors is cost-optimal (p<n)
b) If algorithm is not COST-OPTIMAL for p = n:

- If we increase the granularity
  ⇒ The new algorithm using $\frac{n}{p}$ (p < n)

may still not be cost optimal

Example: Adding “n” numbers on “p” processors

**HYPERCUBE**

n = $2^k$  
Eg: n = 16,

p = $2^m$  
p = 4

- Each virtual PE (i) is simulated by physical PE (i mod p)
- First log p (2 steps) of the log n (4 steps) in the original algorithm are simulated in $\frac{n}{p} \times \log p \times \frac{16}{4} * 2 = 8$ Steps on p = 4 processors

- The remaining steps do not require communication (the PE that continue to communicate in the original algorithm are simulated by the same PE here)
THE ROLE OF MAPPING COMPUTATIONS ONTO PROCESSORS IN PARALLEL ALGORITHM DESIGN

- For a cost-optimal parallel algorithm
  \[ E = O(1) \]
- If a parallel algorithm on \( p = n \) processors is not cost-optimal or cost-non-optimal then \( \Rightarrow \) if \( p < n \) you can find a cost optimal algorithm.
- Even if you find a cost-optimal algorithm for \( p < n \) then \( \Rightarrow \) you found an algorithm with best parallel run-time.
- Performance (Parallel run-time) depends on
  - Number of processors
  - Data-Mapping (Assignment)
• Parallel run-time of the same problem (problem size) depends upon the mapping of the virtual PEs onto Physical PEs.
• Performance critically depends on the data mapping onto a coarse grained parallel computer.
• Example: Matrix multiply nxn by a vector on p processor hypercube \([p \text{ square blocks vs } p \text{ slices of } \frac{n}{p} \text{ rows}]\)

Parallel FFT on a hypercube with Cut-Through Routing

• \(W\) – Computation Steps \(\Rightarrow P_{max} = W\)
• For \(P_{max}\) – each PE executes one step of the algorithm
• For \(p < W\), each PE executes a larger number of steps
• The choice of the best algorithm to perform local computations depends upon \#PEs
  (how much fragmentation is available)
Optimal algorithm for solving a problem on an arbitrary #PEs cannot be obtained from the most fine-grained parallel algorithm.

The analysis on fine-grained parallel algorithm may not reveal important facts such as:

**Analysis of coarse grain Parallel algorithm:**

Notes:
1) If message is short (one word only) => transfer time between 2 PE is the same for store-and-forward and cut-through-routing
2) if message is long => cut-through-routing is faster than store-and-forward
3) Performance on Hypercube and Mesh is identical with cut-through routing
4) Performance on a mesh with store-and-forward is worse.

**Design:**
1) Devise the parallel algorithm for the finest-grain
2) mapping data onto PEs
3) description of algorithm implementation on an arbitrary # PEs

**Variables:**
- Problem size,
- #PEs
SCALABILITY

S ≤ P

S(p)  E(p)

Example: Adding n numbers on a p processors Hypercube

Assume: 1 unit time (For adding 2 numbers or to communicate with connected PE)

1) adding locally \( \frac{n}{p} \) numbers

Takes: \( \frac{n}{p} - 1 \)

2) p partial sums added in logp steps

( for each sum: 1 addition + 1 communication) => 2logp

\[ T_p = \frac{n}{p} - 1 + 2\log p \]

\[ T_p = \frac{n}{p} + 2\log p \]  \( (n \uparrow, p \uparrow) \)

\[ T_s = n - 1 = n \]

\[ S = \frac{n}{\frac{p}{n+2\log p}} = \frac{np}{n+2p\log p} \]  \( \Rightarrow S(n,p) \)

\[ E = \frac{S}{P} = \frac{n}{n+2p\log p} \]  \( \Rightarrow E(n,p) \)

Can be computed for any pair of n and p
As $p \uparrow$ to increase $S$ => need to increase $n$ (Otherwise saturation)

$\Rightarrow E \downarrow$

Larger Problem sizes. $S \uparrow$, $E \uparrow$ but they drop with $p \uparrow$.

$E = Ct$

**Scalability**: of a parallel system is a measure of its capacity to increase speedup in proportion to the number of processors.
Efficiency of adding “n” numbers on a “p” processor hypercube

For cost optimal algorithm:

\[
S = \frac{np}{n+2p\log p} \quad E = \frac{n}{n+2p\log p}
\]

\[
E = E(n,p)
\]

Table 5.1. Efficiency as a function of \( n \) and \( p \) for adding \( n \) numbers on \( p \) processing elements.

<table>
<thead>
<tr>
<th>( n )</th>
<th>( p = 1 )</th>
<th>( p = 4 )</th>
<th>( p = 8 )</th>
<th>( p = 16 )</th>
<th>( p = 32 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>1.0</td>
<td>0.80</td>
<td>0.57</td>
<td>0.33</td>
<td>0.17</td>
</tr>
<tr>
<td>192</td>
<td>1.0</td>
<td>0.92</td>
<td>0.80</td>
<td>0.60</td>
<td>0.38</td>
</tr>
<tr>
<td>320</td>
<td>1.0</td>
<td>0.95</td>
<td>0.87</td>
<td>0.71</td>
<td>0.50</td>
</tr>
<tr>
<td>512</td>
<td>1.0</td>
<td>0.97</td>
<td>0.91</td>
<td>0.80</td>
<td>0.62</td>
</tr>
</tbody>
</table>
\[ n = \Omega(p \log p) \]
\[ E = 0.80 \text{ constant} \]

For \( n = 64 \),
\[ p = 4, \ n = 8p \log p \]

For \( n = 192 \),
\[ p = 8, \ n = 8p \log p \]

For \( n = 512 \),
\[ p = 16, \ n = 8p \log p \]

**Conclusions:**
- For adding \( n \) numbers on \( p \) processors hypercube with cost optimal algorithm.
  - The algorithm is cost-optimal if \( n = \Omega(p \log p) \)
  - The algorithm is scalable if \( n \) increases proportional with \( \Theta(p \log p) \) as \( p \) is increased.
PROBLEM SIZE

For Matrix multiply:

Input size \( n \Rightarrow O(n^3) \)

\[ n' = 2n \Rightarrow O(n'^3) \equiv O(8n^3) \]

Matrix addition

Input size \( n \Rightarrow O(n^2) \)

\[ n' = 2n \Rightarrow O(n'^2) \equiv O(4n^2) \]

Doubling the size of the problem means performing twice the amount of computation.

Computation Step: Assume takes 1 time unit

Message start-up time, can be normalized with per-word transfer time, respect to unit per-hop time

\[ W = T_s \] (for the fastest sequential algorithm on a sequential computer)
Overhead Function

\[ E = 1 \quad \text{(Ideal)} \quad \quad \quad E < 1 \quad \text{(In reality)} \]
\[ S = p \quad \text{overhead (interprocessor communic ..... etc)} \]

=> **Overhead function**

The time collectively spent by all processors in addition to that required by the fastest sequential algorithm to solve the same problem on a single PE.

\[ T_o = T_o(W,p) \]
\[ T_o = p(T_p - W) \]

○ For cost-optimal algorithm of adding n numbers on p processors hypercube

\[ T_s = W = n \]
\[ T_p = \frac{n}{p} + 2\log p \]
\[ T_o = p\left(\frac{n}{p} + 2\log p\right) - n = 2p\log p \]
\[ T_o = 2p\log p \]
ISOEFFICIENCY FUNCTION

Parallel execution time = (function of problem size, overhead function, no. of processors)

\[ T_p = T(W, T_o, p) \quad \text{and} \quad T_p = \frac{W + T_o(W, p)}{p} \]

\[ T_o = pT_p - W \]

Speedup \( S \) = \( \frac{T_s}{T_p} = \frac{W}{T_p} = \frac{Wp}{W + T_o(W, p)} \)

Efficiency \( E \) = \( \frac{S}{P} = \frac{W}{W + T_o(W, p)} \quad E = \frac{1}{1 + \frac{T_o(W, p)}{W}} \)

If \( W = \text{constant} \), \( P = \uparrow \) then \( E \downarrow \)
If \( p = \text{constant} \), \( W \uparrow \) then \( E \uparrow \) for parallel scalable systems.

- we need \( E = \text{constant} \) - for scalable effective systems.
Example 1

\[ P, W \text{ exponentially with } p \]
\[ \Rightarrow \text{then problem is poorly scalable} \]
\[ \text{since we need to increase the problem size very much to obtain good speedups} \]

Example 2

\[ P, W \text{ linearly with } p \]
\[ \Rightarrow \text{then problem is highly scalable} \]
\[ \text{since the speedup is now proportional to the number of processors.} \]

\[ E = \frac{1}{1 + \frac{T_o(W,p)}{W}} \Rightarrow W = \frac{E}{1 - E} T_o(W, p) \]

\[ E = ct \Rightarrow \frac{E}{1 - E} = ct \]

Given \( E \) \( \Rightarrow \frac{E}{1 - E} = k \)

- \( W = KT_o(W,p) \) Function dictates growth rate of \( W \) required to keep the \( E \) Constant as \( P \) increases.

Isoefficiency \( \notin \) in unscalable parallel systems because \( E \) cannot be kept constant as \( p \) increases, no matter how much or how fast \( W \) increases.
Overhead Function  (adding n numbers on p processors Hypercube)

\[ T_s = n \]
\[ T_p = \frac{n}{p} + 2\log p \]
\[ T_o = pT_p - T_s = p\left(\frac{n}{p} + 2\log p\right) - n = 2p\log p \]

Isoefficiency Function

\[ W = kT_o \quad (W,p) \]
\[ T_o = 2p\log p \quad (\text{Note}: \quad T_o = T_o \quad (p)) \]
\[ W = 2kp\log p \]

=> asymptotic isoefficiency function is \( \Theta(p\log p) \)

Meaning:

1) \( \#\text{PE} \uparrow p' > p \Rightarrow \text{problem size has to be increased by } \left(\frac{p'\log p'}{p\log p}\right) \) to have the same efficiency as on p processors.
2) #PE↑ \( p' > p \) by a factor \( \frac{p'}{p} \) requires problem size to grow by a factor \( \frac{p' \log p'}{p \log p} \) to increase the speedup by \( \frac{p'}{p} \)

Here **communication overhead** is an exclusive function of \( p \) : \( T_o = T_o(p) \)

In general

\[
T_o = T_o(W, p)
\]

\( W = k T_o(W, p) \) (may involve many terms)

Sometimes hard to solve in terms of \( p \)

**E= constant** need **ratio** \( \frac{T_o}{W} \) **fixed**

As \( p \uparrow, W \uparrow \)

\( E' \geq E \implies T_o \) should not grow faster than \( W \)

None of \( T_o \) terms should grow faster than \( W \).
If $T_o$ has **multiple terms**, we balance $W$ against each term of $T_o$ and compute the respective isoefficiency functions for corresponding individual terms.

The component of $T_o$ that requires the problem size to grow at the **highest rate** with respect to $p$, determines the overall asymptotic isoefficiency of the parallel system.

**Example 1:**

$$T_o = p^{3/2} + p^{3/4} W^{3/4}$$

$W = k p^{3/2} \Rightarrow \Theta(p^{3/2})$

$$W = k p^{3/4} W^{3/4}$$

$W^{1/4} = k p^{3/4}$

$W = k^4 p^3$

$\Rightarrow \Theta(P^3)$

→ take the highest of the two rates

Isoefficiency function of this example is $\Theta(P^3)$
Example 2:

- $T_o = p^{1/2} W^{1/2} + p^{3/5} + p^{3/4} W^{3/4} + p^{3/2} + p$
- $W = K T_o$
  \[= K \left(p^{1/2} W^{1/2} + p^{3/5} + p^{3/4} W^{3/4} + p^{3/2} + p\right)\]
- $W = K p^{1/2} W^{1/2}$
- $W = K p^{3/5}$
- $W = K p^{3/4} W^{3/4} = \Theta(P^3)$
- $W = K p^{3/2}$
- $W = K p$

Therefore \( W = \Theta(P^3) \) (\( T_o \) has this isoefficiency function which is the highest of all)

→ \( T_o \) ensure E doesn't decrease, the problem size needs to grow as \( \Theta(P^3) \) (Overall asymptotic isoefficiency)
Isoefficiency Functions

- Captures characteristics of the parallel algorithm and architecture
- Predicts the impact on performance as \#PE↑
- Characterizes the amount of parallelism in a parallel algorithm.
- Study of algorithm (parallel system) behavior due to hardware changes (speed, PE, communication channels)

Cost-Optimality and Isoefficiency Function

Cost-Optimality = \( \frac{T_s}{pT_p} = ct \)

\( P T_p = \Theta(W) \)
\( W + T_o (W,p) = \Theta(W) \) \hspace{1cm} (\( T_o=pT_p - W \))
\( T_o (W,p) = O(W) \)
\( W=\Omega(T_o (W,p)) \)

A parallel system is cost optimal iff (of 2 only if) its overhead function does not grow asymptotically more than the problem size
Relationship between Cost-optimality and Isoefficiency Function

Eg: Add “n” numbers on “p” processors hypercube

a) Non-optimal cost

\[ W = O(n) \]
\[ T_p = O(\frac{n}{p} \log p) \]
\[ T_o = pT_p - W = \Theta(n \log p) \]
\[ W = k \Theta(n \log p) \] not true for all K and E

Algorithm is not cost-optimal, and \( \not\exists \) isoefficiency function

not scalable

b) Cost-Optimal

\[ W = O(n) ; \ W \approx n \]
\[ T_p = O(\frac{n}{p} + \log p) \]
\[ T_o = \Theta(n + p \log p) - O(n) \]
\[ W = k \Theta(p \log p) \]

\[ W = \Omega(p \log p) \quad n >> p \quad \text{for cost optimality} \]

Problem size should grow at least as plogp such that parallel system is scalable.
ISOEFFICIENCY FUNCTION

- Determines the ease with which a parallel system can maintain a constant efficiency and thus, achieve speedups increasing in proportion to the number of processors.
- A small isoefficiency function means that small increments in the problem size are sufficient for the efficient utilization of an increasing number of processors. => indicates the parallel system is highly scalable.
- A large isoefficiency function indicates a poorly scalable parallel system.
- The isoefficiency function does not exist for unscalable parallel systems, because in such systems the efficiency cannot be kept at any constant value as $p \uparrow$, no matter how fast the problem size is increased.
Lower Bound on the Isoefficiency

- **Small isoefficiency function => higher scalability**
  - For a problem with $W$, $P_{max} \leq W$ for cost-optimal system (if $P_{max} > W$, some PE are idle)
  - If $W < \Theta(p)$ i.e problem size grows slower than $p$, as $p \uparrow \uparrow$ then at one point #PE > $W$ => $E \downarrow \downarrow$
  - => asymptotically $W= \Theta(p)$
  - Problem size must increase proportional as $\Theta(p)$ to maintain fixed efficiency

  $W = \Omega(p)$  \hspace{1cm} ($W$ should grow at least as fast as $p$)
  $\Omega(p)$ is the asymptotic lower bound on the isoefficiency function
  But $P_{max} = \Theta(W)$  \hspace{1cm} ($p$ should grow at most as fast as $W$)

  => The isoefficiency function for an ideal parallel system is: $W = \Theta(p)$
Example (which one overhead function is most scalable)

- $T_{o1} = ap^5 W^{4/5} + bp^3$
- $T_{o2} = cp^5 + dp^3 + ep^2$
- $T_{o3} = fp^5 + gp^5 W^{4/5} + hp^{3/4} W^{3/4}$

First calculate isoefficiency function of all three overhead function and then find the lowest of these three.

Isoefficiency function of

- $T_{o1} = \Theta(P^{25})$
- $T_{o2} = \Theta(P^5)$
- $T_{o3} = \Theta(P^{25})$

Therefore $T_{o2} = \Theta(P^5)$ has the lowest isoefficiency function which is most scalable among three.
Degree of Concurrency & Isoefficiency Function

- Maximum number of tasks that can be executed simultaneously at any time
- Independent of parallel architecture

\[ C(W) \] – no more than \( C(W) \) processors can be employed effectively

Effect of Concurrency on Isoefficiency function

Example: Gaussian Elimination:

\[ W = \Theta(n^3) \]
\[ P = \Theta(n^2) \]
\[ C(W) = \Theta(W^{2/3}) \]

\( \Rightarrow \) at most \( \Theta(W^{2/3}) \) processors can be used efficiently

Given \( p \)

\[ W = \Omega(p^{3/2}) \]

\( \Rightarrow \) problem size should be at least \( \Omega(p^{3/2}) \) to use them all

\( \Rightarrow \) The Isoefficiency due to concurrency is \( \Theta(p^{3/2}) \)

The Isoefficiency function due to concurrency is optimal, that is, \( \Theta(p) \) only is the degree of concurrency of the parallel algorithm is \( \Theta(W) \)
SOURCES OF OVERHEAD

- **Interprocessor Communication**
  - Each PE Spends $t_{\text{comm}}$
  - Overall interprocessor communication: $pt_{\text{comm}}$
  (Architecture impact)

- **Load imbalance**
  - Idle vs busy PEs (Contributes to overhead)
  Example: In sequential part
  1PE : $W_s \leftarrow$ Useful
  p-1 PEs: $(p-1) W_s$ contribution to overhead function

- **Extra-Computation**
  1) Redundant Computation (eg: Fast fourier transform)
  2) $W$ – for best sequential algorithm
  $W'$ – for a sequential algorithm easily parallelizable
  $W' - W \rightarrow$ contributes to overhead.
  $W = W_s + W_p \Rightarrow W_s$ executed by 1PE only
  $\Rightarrow(p-1) W_s$ contributes to overhead
  - Overhead of scheduling
If the degree of concurrency of an algorithm is less than $\Theta(W)$, then the **Isoefficiency function** due to concurrency is worse, i.e. greater than $\Theta(p)$

Overall Isoefficiency function of a parallel system:
$$Isoeff_{system} = \max(Isoeff_{concurr}, Isoeff_{commun}, Isoeff_{overhead})$$

**SOURCES OF PARALLEL OVERHEAD**

- The overhead function characterizes a parallel system
- Given the overhead function $T_o = T_o(W,p)$
  
  We can express:
  $$T_p, S, E, pT_p(cost) \text{ as } f_i(W,p)$$
- The overhead function encapsulates all causes of inefficiencies of a parallel system, due to:
  - Algorithm
  - Architecture
  - Algorithm –architecture interaction
MINIMUM EXECUTION TIME
(Adding “n” Number on a Hypercube)

(Assume p is not a constraint)

As we increase the number of processors for a given problem size, either the parallel run time continues to decrease and asymptotically approaches a minimum value, or it starts rising after attaining a minimum value.

\[ T_p = T_p(W,p) \]

For a given \( W \), \( T_p^{\text{min}} = 2 \)

\[ \frac{d}{dp} T_p = 0 \Rightarrow P_0 \text{ for which } T_p = T_p^{\text{min}} \]

Example:
The parallel runtime for the problem of adding \( n \) numbers on a \( p \)-processor hypercube be approx. by
$$T_p = \frac{n}{p} + 2\log p$$

we have,
$$\frac{dT_p}{dp} = 0$$

$$-\frac{n}{p^2} + \frac{2}{p} = 0$$

$$-n + 2p = 0$$

$$\Rightarrow P_o = \frac{n}{2}$$

$$T_p^{min} = 2\log n$$

The processor-time product for $p = p_o$ is $\Theta(n\log n)$

Cost-Sequential : $\Theta(n)$ not cost optimal

Cost-Parallel : $\Theta(n\log n)$ since $P_o \cdot T_p^{min} = \frac{n}{2} \times 2 \log n$

Not Cost optimal

$\Rightarrow$ running this algorithm for $T_p^{min}$ is not cost-optimal But this algorithm is COST-OPTIMAL
Derive:

**Lower bound for** $T_p$ **such that parallel cost is optimal:**

$T_p^{\text{cost-opt}}$ – Parallel run time such that cost is optimal.

- $W$ fixed.

- If Isoefficiency function is $\Theta(f(p))$
  
  Then problem of size $W$ can be executed Cost-optimally only iff: $W = \Omega(f(p))$

  $$P = O(f^{-1}(W)) \{ \text{Required for a cost optimal solution} \}$$

  parallel runtime $T_p$ for cost cost-optimal solution is $= \Theta\left(\frac{w}{p}\right)$

Since

$$pT_p = \Theta(W)$$
$$T_p = \Theta\left(\frac{w}{p}\right)$$
$$P = \Theta(f^{-1}(W))$$

The lower bound on the parallel runtime for solving a problem of size $W$ is cost optimal iff:

$$T_p^{\text{cost-opt}} = \Omega\left(\frac{w}{f^{-1}(W)}\right)$$
MINIMUM COST-OPTIMAL TIME FOR ADDING N NUMBERS ON A HYPERCUBE

A) isoefficiency function:

\[ T_0 = p T_p - W \]
\[ T_p = \frac{n}{p} + 2 \log p \]

\[ \Rightarrow T_0 = p \left( \frac{n}{p} + 2 \log p \right) - n = 2p \log p \]

\[ W = k T_0 = k \times 2p \log p \]

\[ W = \Theta(p \log p) \text{  \{isoefficiency function\}} \]

- If \( W = n = f(p) = p \log p \)
  \[ \Rightarrow \log n = \log p + \log \log p \]
  \[ \log n \approx \log p \text{ (ignoring double logarithmic term)} \]

- If \( n = f(p) = p \log p \)
  \[ P = f^{-1}(n) \]
  \[ n = p \log p \Rightarrow p = \frac{n}{\log p} \approx \frac{n}{\log n} \]
  \[ f^{-1}(n) = \frac{n}{\log n} \]
  \[ f^{-1}(W) = \frac{n}{\log n} \]
  \[ f^1(W) = \Theta \left( \frac{n}{\log n} \right) \]
B) The cost-optimal solution

\[ p = O(f^{-1}(W)) \]

=> for a cost optimal solution

\[ P = \Theta(n \log n) \{ \text{the max for cost-optimal solution} \} \]

For \( P = \frac{n}{\log n} \),

\[ T_p = T_p^{\text{cost-opt}} \]

\[ T_p = \frac{n}{p} + 2 \log p \]

=> \( T_p^{\text{cost-opt}} = \log n + 2 \log \left( \frac{n}{\log n} \right) \)

= \( 3 \log n - 2 \log \log n \)

\[ T_p^{\text{cost-opt}} = \Theta(\log n) \]

Note:

\[ T_p^{\min} = \Theta(\log n) \]

\[ T_p^{\text{cost-opt}} = \Theta(\log n) \]

\{ cost optimal solution is the best asymptotic solution in terms of execution time \}

\( T_p^{\min} \implies P_o = \frac{n}{2} > P_o = \frac{n}{\log n} \iff (T_p^{\text{cost-opt}}) \)

=> \( T_p^{\text{cost-opt}} = \Theta(T_p^{\min}) \)

Both \( T_p^{\min} \) and \( T_p^{\text{cost-opt}} \) for adding \( n \) numbers on hypercube are \( \Theta(\log n) \), thus for above problem, a cost-optimal solution is also the asymptotically fastest solution.
Parallel System where $T_p^{cost-optimal} > T_p^{min}$

$T_o = p^{3/2} + p^{3/4}W^{3/4}$

$T_p = \frac{W+T_o}{p}$

$\frac{d}{dp} T_p = 0$

$\frac{d}{dp} T_p = -\frac{W}{p^2} + \frac{1}{2p^{1/2}} - \frac{W^{3/4}}{4p^{5/4}} = 0$

$-W + \frac{1}{2} p^{3/2} - \frac{1}{4} W^{3/4} p^{3/4} = 0$

$p^{3/4} = \frac{1}{4} W^{3/4} \pm \left(\frac{1}{16} W^{3/4} + 2W\right)^{1/2}$

$= \Theta(W^{3/4})$

$P_o = \Theta(W)$ (where $P_o = P$)

$T_p^{min} = \Theta(W^{1/2})$ ..............(equation i)
Isoefficiency Function:
\[ W = k \ T_o = k^4 p^3 = \Theta(p^3) \]
\[ \Rightarrow P_{max} = \Theta(W^{1/3}) \] \hspace{1cm} (\text{equation ii}) \{\text{Max #PE for which algorithm is cost-optimal}\}

\[ T_p = \frac{w}{p} + p^{1/2} + \frac{w^{3/4}}{p^{1/4}} \]
\[ p = \Theta(W) \quad \Rightarrow T_p^{\text{cost-optimal}} = \Theta(W^{2/3}) \]
\[ \Rightarrow T_p^{\text{cost-optimal}} > T_p^{\text{min}} \text{ asymptotically} \]

\text{equation i and ii shows that} \ T_p^{\text{cost-optimal}} \text{ is asymptotically greater than} \ T_p^{\text{min}}.

Deriving \( T_p^{\text{min}} \) it is important to be aware of \( P_{max} \) that can be utilized is bounded by the degree of concurrency \( C(W) \) of the parallel algorithm.

It is possible that \( P_o > C(W) \) for parallel system. i.e Value of \( P_o \) is meaningless and \( T_p^{\text{min}} \) is given by

\[ T_p^{\text{min}} = \frac{w+T_o(W,C(W))}{C(W)} \]
Example (showing whether parallel algorithm is cost-optimal with respect to sequential algorithm or not)

Given

\[ T_{s1} = an^2 + bn + c \quad a, b, c \in R \]
\[ T_{s2} = a'n^2 \log n + b'n^2 + c'n + d \quad a', b', c', d \in R \]

\[ T_p = \frac{n^2}{p} + 64 \log p \]

Cost optimality if \( pT_p \sim T_s \)

\[ \frac{T_s}{pT_p} = \text{constant} \]

we have,

\[ pT_p = n^2 + 64 \log p \Rightarrow pT_p = n^2 + \Theta (p \log p) \]
\( T_{s1} = \Theta (n^2) \)

- The parallel algorithm is cost-optimal w.r.t. to \( T_{s1} \) iff \( n^2 \) grows at least by \( \Theta (p \log p) \) ⇒ \( n^2 = \Omega (p \log p) \)

\( T_{s2} = \Theta (n^2 \log n) \)

\( pT_p = \Theta (n^2) \)

\( \frac{T_{s2}}{pT_p} = \Theta (\log n) \neq O(1) \)

- The parallel algorithm is not cost-optimal w.r.t. to \( T_{s2} \) because \( \frac{T_{s2}}{pT_p} = \Theta (\log n) \neq O(1) \)
Example (condition when parallel runtime is minimum)

\[ \frac{d}{dp} T_p = 0 \]

\[ \Rightarrow \frac{-n^2}{p^2} + \frac{64}{p} = 0 \]

\[ P = \frac{n^2}{64} \]

\[ T_p = 64 + 128(\log n - \log 8) \]

\[ pT_p = \frac{n^2}{64} \cdot (64 + 128 (\log n - \log 8)) \]

\[ = \Theta (n^2 \log n) \]

\[ T_p = O(\log n) \]
\( T_{s1} : \)

\[
\frac{T_{s1}}{pT_p} = \frac{\Theta(n^2)}{O(n^2 \log n)} = O\left(\frac{1}{\log n}\right) \neq O(1)
\]

Therefore it is not optimal w.r.t \( T_{s1} \).

\( T_{s2} : \)

\[
\frac{T_{s2}}{pT_p} = \frac{\Theta(n^2 \log n)}{O(n^2 \log n)} = O(1)
\]

This is cost-optimal w.r.t \( T_{s2} \).
Asymptotic Analysis of Parallel Programs

Ignores constants and concern with the asymptotic behavior of quantities. In many cases, this can yield a clearer picture of relative merits and demerits of various parallel programs.

Table 5.2. Comparison of four different algorithms for sorting a given list of numbers. The table shows number of processing elements, parallel runtime, speedup, efficiency and the $pT_P$ product.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
<th>A4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$</td>
<td>$n^2$</td>
<td>$\log n$</td>
<td>$n$</td>
<td>$\sqrt{n}$</td>
</tr>
<tr>
<td>$T_P$</td>
<td>1</td>
<td>$n$</td>
<td>$\sqrt{n}$</td>
<td>$\sqrt{n} \log n$</td>
</tr>
<tr>
<td>$S$</td>
<td>$n \log n$</td>
<td>$\log n$</td>
<td>$\sqrt{n} \log n$</td>
<td>$\sqrt{n}$</td>
</tr>
<tr>
<td>$E$</td>
<td>$\frac{\log n}{n}$</td>
<td>1</td>
<td>$\frac{\log n}{\sqrt{n}}$</td>
<td>1</td>
</tr>
<tr>
<td>$pT_P$</td>
<td>$n^2$</td>
<td>$n \log n$</td>
<td>$n^{1.5}$</td>
<td>$n \log n$</td>
</tr>
</tbody>
</table>
Consider the problem of sorting a list of $n$ numbers. The fastest serial programs for this problem run in time $O(n \log n)$. The objective is to determine which of these four algorithms is the best. Perhaps the simplest metric is one of speed; the algorithm with the lowest $T_p$ is the best. By this metric, algorithm A1 is the best, followed by A3, A4, and A2. This is also reflected in the fact that the speedups of the set of algorithms are also in this order.

In practice, we will rarely have $n^2$ processing elements as are required by algorithm A1. Resource utilization is an important aspect of practical program design. Algorithms A2 and A4 are the best, followed by A3 and A1. The costs of algorithms A1 and A3 are higher than the serial runtime of $n \log n$ and therefore neither of these algorithms is cost optimal. However, algorithms A2 and A4 are cost optimal.