Parallel Computing

Ekkapot Charoenwanit

Software Systems Engineering
TGGS
KMUTNB

Lecture 1:

Parallel Performance Analysis

Performance Analysis: Speedup

We develop a ***parallel*** program in the hope that it will run ***faster*** than its ***sequential*** counterpart.

Speedup $\psi(n,p)$ is a measure that captures the relative benefit of the parallel program that runs on p processors solving the computational problem of size n as the ***ratio*** between ***sequential execution time*** $T(n,1)$ and ***parallel execution time*** $T(n,p)$.

$$\psi(n,p) = \frac{T(n,1)}{T(n,p)}$$

Performance Analysis: Parallel Execution Time

Parallel execution time $T(n,p)$ is the time that elapses from the moment a parallel computation starts until the moment the last processor finishes execution.

Parallel execution time $T(n,p)$ can be broken down into ***three*** components:

- Computations that must be performed sequentially
- Computations that can be performed in parallel
- Parallelization overhead, i.e., communication operations and redundant computations

Performance Analysis: Parallel Execution Time

A simple model of ***parallel execution time***:

Let

- $\sigma(n,p)$ denote the inherently sequential fraction of the computation,
- $\varphi(n,p)$ denote the fraction of the computation that can be executed in parallel
- $\kappa(n,p)$ denote the time required due to parallel overhead

Performance Analysis: Parallel Execution Time

A ***sequential program*** P_{seq}, executing on a ***single processor***, requires $\sigma(n,p) + \varphi(n,p)$ units of time to execute.

- It requires ***no interprocessor communications*** so its expression for sequential execution time does not contain the parallel overhead term $\kappa(n,p)$.

$$T(n,1) = \sigma(n,p) + \varphi(n,p)$$

Performance Analysis: Parallel Execution Time

Suppose that

- the sequential program P_{seq} contains a fraction that is ***not parallelizable*** and contributes $\sigma(n, p)$ time units to the total execution time of P_{seq}, no matter how many processors are available

- the sequential program P_{seq} contains a fraction that is ***parallelizable*** and contributes $\varphi(n, p)$ time units to the total execution time of P_{seq} and this parallelizable fraction ***divides up perfectly*** among the p processors

  - Thus, the time needed to perform these computations is $\frac{\varphi(n,p)}{p}$.

Therefore, the expression for ***parallel execution time*** can be formulated as

$$T(n, p) = \sigma(n, p) + \frac{\varphi(n,p)}{p} + \kappa(n, p)$$

Performance Analysis: U-Bound on Speedup

Speedup is the ratio between ***sequential*** execution time and ***parallel*** execution time.

$$\psi(n,p) = \frac{T(n,1)}{T(n,p)}$$

$$\leq \frac{\sigma(n,p) + \varphi(n,p)}{\sigma(n,p) + \varphi(n,p)/p + \kappa(n,p)}$$

The ***inequality*** arises from the optimistic assumption that the parallel fraction can be ***divided perfectly*** among the p processors.

- Otherwise, the parallel execution time will be ***larger***, and the speedup will be ***smaller***.

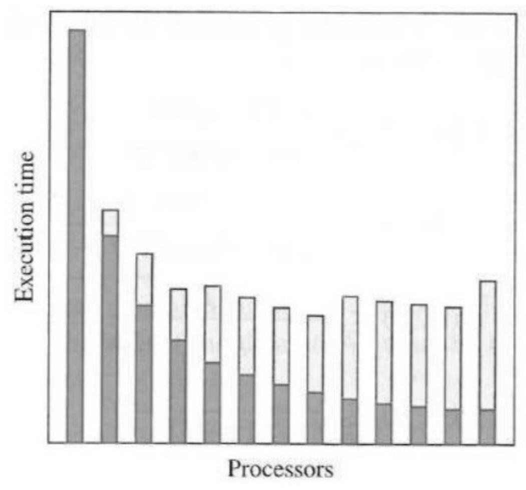
Performance Analysis: Parallel Overhead

Adding more processors ***reduces*** the ***computation time*** by dividing the work among more processors, but ***increases*** the ***communication time***.

At some point, the increase in communication time is larger than the decrease in computation time.

- At this point, the parallel execution time begins to increase.

The figure on the right demonstrates that a non-trivial parallel algorithm has a ***computation component (black bars)*** that is a decreasing function of the number of processors and a ***communication component (gray bars)*** that is an increasing function of the number of processors.



Performance Analysis: Efficiency

The ***efficiency*** $\varepsilon(n,p)$ of a parallel program is a ***measure of processor utilization***.

- In other words, $\varepsilon(n,p)$ measures the fraction of time each processor participates in ***meaningful*** computation.

We define ***efficiency*** $\varepsilon(n,p)$as the ratio between speedup and the number of processors.

$$\varepsilon(n,p) = \frac{\psi(n,p)}{p}$$

Therefore,

$$\varepsilon(n,p) \leq \frac{\sigma(n,p)+\varphi(n,p)}{p(\sigma(n,p)+\varphi(n,p)/p+\kappa(n,p))}$$

$$\varepsilon(n,p) \leq \frac{\sigma(n,p)+\varphi(n,p)}{p\sigma(n,p)+\varphi(n,p)+p\kappa(n,p)}$$

Note that $0 \leq \varepsilon(n,p) \leq 1$.

Note that we can treat p in $\varepsilon(n,p) = \frac{\psi(n,p)}{p}$ as the ***ideal speedup***.

Performance Analysis: Amdahl's Law

By ignoring the ***parallel overhead*** term $\kappa(n, p)$ in

$$\psi(n, p) \leq \frac{\sigma(n,p) + \varphi(n,p)}{\sigma(n,p) + \frac{\varphi(n,p)}{p} + \kappa(n,p)}$$

since $\kappa(n, p) > 0$, we can ***simplify*** the inequality above to obtain a ***looser upper bound*** on $\psi(n, p)$ as follows

$$\psi(n, p) \leq \frac{\sigma(n,p) + \varphi(n,p)}{\sigma(n,p) + \frac{\varphi(n,p)}{p}}$$

This inequality provides the foundation for the derivation of ***Amdahl's Law***.

Performance Analysis: Amdahl's Law

Let f denote the ***inherently sequential fraction*** of the sequential program P_{seq}.

$$f = \frac{\sigma(n,p)}{\sigma(n,p)+\varphi(n,p)}$$

Then,

$$\psi(n,p) \leq \frac{\sigma(n,p) + \varphi(n,p)}{\sigma(n,p) + \frac{\varphi(n,p)}{p}}$$

$$\psi(n,p) \leq \frac{\frac{\sigma(n,p)}{f}}{\sigma(n,p) + \frac{\sigma(n,p)\left(\frac{1}{f}-1\right)}{p}}$$

$$\psi(n,p) \leq \frac{1}{f + \frac{1-f}{p}}$$

Note that $0 \leq f \leq 1$.

Performance Analysis: Amdahl's Law

Amdahl's Law

Let f denote the fraction of operations in a computation that must be performed sequentially, where $0 \leq f \leq 1$.

The maximum speedup $\psi(n, p)$ achievable by a parallel computer with p processors performing the computation is

$$\psi(n, p) \leq \frac{1}{f + \frac{1-f}{p}}$$

Performance Analysis: Amdahl's Law

Amdahl's Law is based on the assumption that we are trying to solve a problem of ***fixed size*** n as quickly as possible.

Assuming there are an ***infinite number of processors*** available,
the maximum speedup achievable is

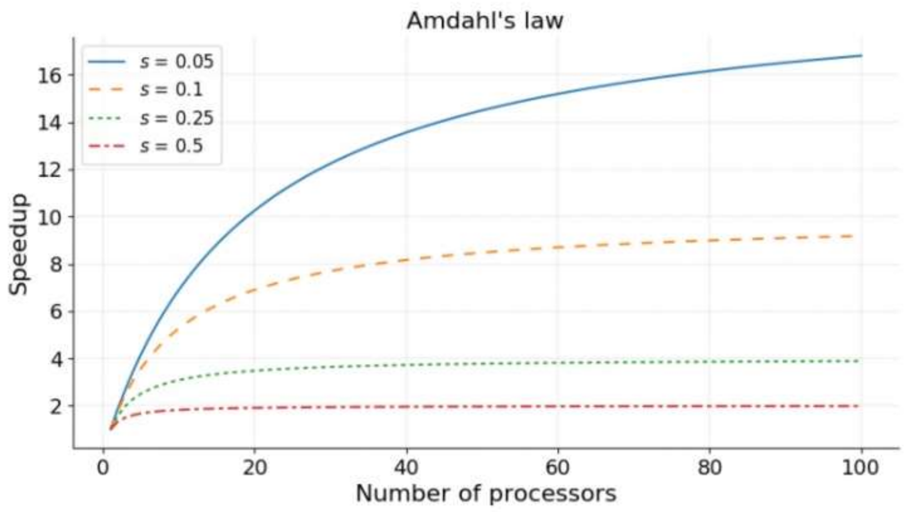
$$\lim_{p \to \infty} \frac{1}{f + \frac{1-f}{p}} = \frac{1}{f}$$

- In other words, this illustrates the fact that the speedup of a parallel program is ultimately dependent on the ***sequential fraction*** of the sequential program.

Performance Analysis: Amdahl's Law

The figure on the right illustrates the ***maximum speedup*** of a parallel program with different values of f.

The ***ideal speedup*** is represented by a ***straight line*** with a slope of 1 that passes through the origin.



Performance Analysis: Amdahl's Law

Example 1

Suppose we are trying to determine whether it is worthwhile to develop a parallel version of a program solving a particular problem. Benchmarking reveals that 80 percent of the execution time is spent inside functions that we believe we can execute in parallel. The remaining 20 percent of the execution time is spent in functions that must be executed on a single processor. What is the maximum speedup that we could expect from a parallel version of the program executing on 16 processors? What is the maximum speedup?

Performance Analysis: Amdahl's Law

Solution to Example 1

By Amdahl's Law,

$$\psi(n,p) \le \frac{1}{f + \frac{1-f}{p}}$$

Plugging $f = 0.20$ and $p = 16$ into the formula above,

$$\psi(n,p) \le \frac{1}{0.20+(1-0.20)/16} = 4.00 \quad \blacksquare$$

Therefore, we should expect a speedup of 4.00 or less.

The maximum achievable speedup is $\lim_{p\to\infty} \frac{1}{f+\frac{1-f}{p}} = \frac{1}{f} = \frac{1}{0.20} = 5.00 \ \blacksquare$

Performance Analysis: Amdahl Effect

Limitations of Amdahl's Law

Amdahl's Law neglects overhead associated with the introduction of parallelism.

- Typically, the parallel overhead $\kappa(n, p)$ has ***lower complexity*** than the parallelizable component $\frac{\varphi(n,p)}{p}$.
- Increasing the problem size increases the computation time ***faster*** than it increases the communication time.

Hence, for a ***fixed number of processors*** p, speedup is usually an increasing function of the problem size.

- This is referred to as the ***Amdahl effect***.

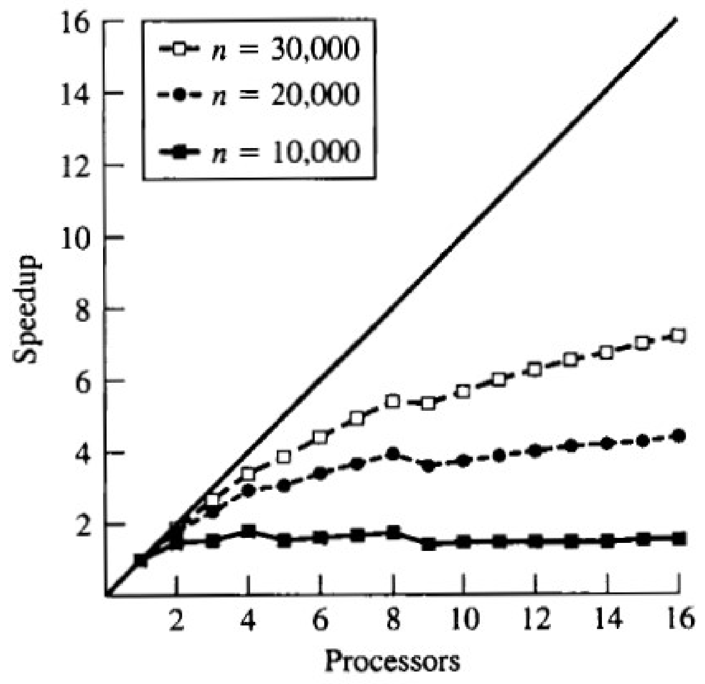
Performance Analysis: Amdahl Effect

Limitations of Amdahl's Law

For any ***fixed*** number of processors, speedup is usually an increasing function of the problem size.

This is called the ***Amdahl effect***.

Amdahl's Law assumes a fixed problem size n so it can ***underestimate*** speedup for large problem sizes.



Performance Analysis: Gustafson’s Law

Amdahl's Law assumes that ***minimizing execution time*** is the focus of parallel computing.

- It treats the ***problem size*** n as a ***constant*** and demonstrates how increasing the number of processors p can reduce execution time.

In reality, however, the goal of parallel computing is to increase the accuracy (larger number of grid points, for example) of the solution that can be computed in a ***fixed amount of time***.

- In other words, we want to be able to solve the problem of ***larger sizes*** with ***more processors*** in ***the same time duration.***

Performance Analysis: Gustafson’s Law

Gustafson’s law, also known as ***Gustafson-Barsis’s law***, addresses these shortcomings of ***Amdahl's Law***.

What happens if we treat ***execution time*** as a ***constant*** and let the problem size n proportionally increase with the number of processors p?

- The inherently ***sequential fraction*** of a computation typically ***decreases*** as problem size ***increases*** due to ***Amdahl effect***.

Performance Analysis: Gustafson’s Law

Consider the expression for speedup

$$\psi(n,p) \le \frac{\sigma(n,p)+\varphi(n,p)}{\sigma(n,p)+\frac{\varphi(n,p)}{p}}$$

Let s denote the fraction of time spent in the ***parallel execution*** that performs the inherently ***sequential operations***.

$$s = \frac{\sigma(n,p)}{\sigma(n,p)+\frac{\varphi(n,p)}{p}}$$

Hence, the fraction of time spent in the ***parallel execution*** that performs ***parallel operations*** ins what remains, or $1-s$.

$$1-s = \frac{\frac{\varphi(n,p)}{p}}{\sigma(n,p)+\frac{\varphi(n,p)}{p}}$$

Performance Analysis: Gustafson's Law

Hence,

$$\sigma(n,p) = \left(\sigma(n,p) + \frac{\varphi(n,p)}{p}\right) s$$

$$\varphi(n,p) = \left(\sigma(n,p) + \frac{\varphi(n,p)}{p}\right)(1-s)p$$

Substituting these two terms into $\psi(n,p) \leq \frac{\sigma(n,p)+\varphi\ (n,p)}{\sigma\ (n,p)\ + \frac{\varphi(n,p)}{p}}$,

$$\psi(n,p) \leq \frac{(\sigma(n,p)+\frac{\varphi(n,p)}{p})(s+(1-s)p)}{\sigma\ (n,p)+\frac{\varphi}{p}}$$

$$\psi(n,p)\ \leq s + (1-s)p$$

$$\psi(n,p)\ \leq p + (1-p)s$$

Performance Analysis: Gustafson’s Law

Gustafson’s Law

Given a parallel program solving a problem of size n using p processors,

let s denote the fraction of time spent in the parallel execution that performs inherently sequential operations.

The maximum speedup $\psi(n, p)$ achievable by this parallel program is

$$\psi(n, p) \leq p + (1 - p)s$$

Performance Analysis: Gustafson’s Law

Amdahl's Law vs Gustafson’s Law

Amdahl's Law begins with a sequential program and attempts to predict how fast that computation could execute on multiple processors in a corresponding parallel program.

Gustafson's Law does the ***opposite:*** It begins with a parallel computation and estimates how much faster the parallel computation is than the same computation on a single processor.

- We refer to the speedup predicted by ***Gustafson’s law*** as ***scaled speedup***.

Performance Analysis: Gustafson’s Law

One ***implicit*** assumption of ***Gustafson’s Law*** is that the sequential time we compare against is the time that would be required to solve the same problem on a single processor, ***if it had sufficient memory***.

In many cases, assuming a single processor is only p times ***slower than*** p processors may be ***overly optimistic***. For example, imagine solving a problem on a parallel computer with 16 processors, each with 1 GB of ***local memory***. Suppose the dataset occupies 15 GB. If we tried to solve the same problem on a single processor, the entire dataset would not fit in the local memory. If the working set of the executing program exceeded 1 GB, it would begin to ***trash***, taking much more than 16 times as long to execute the parallel fraction of the program as the group of 16 processors.

Performance Analysis: Gustafson's Law

Example 2

Suppose the total execution time for a parallel application is $1{,}040$ seconds on 32 cores, but 14 seconds of that time is for sequential execution on one of these 32 cores. What is the ***scaled speedup*** of this application over the same dataset being run on a single thread (if it were possible)?

Performance Analysis: Gustafson’s Law

Solution to Example 2

The serial fraction s is $\frac{14\ seconds}{1040\ seconds} \approx 0.013$.

Plugging $s = 0.013$ and $p = 32$ into $\psi \leq p + (1-p)s$,

$$\psi(n,p) \leq\ 32 + (1-32)(0.013) \approx 31.6 \quad \blacksquare$$

Performance Analysis: Gustafson’s Law

Could we had used ***Amdahl's Law*** to compute this speedup estimate?

If we take 0.013 as the sequential fraction and plug this into the inequality of ***Amdahl's Law***, we will get

$$\psi \leq \frac{1}{f + \frac{1-f}{p}} = \frac{1}{0.013 + \frac{1 - 0.013}{32}} \approx 22.8$$

However, this is a ***wrong*** calculation since the sequential fraction is relative to the parallel execution time of the 32-core execution.

Performance Analysis: Gustafson’s Law

How can we use ***Amdahl's Law*** correctly to get the correct speedup in this situation?

The parallel execution time is $1040 - 14 = \ 1{,}026$ seconds.

That is, the total amount of work done by the 32-core execution is $1{,}026 \cdot 32 + 14 = 32{,}846$ seconds, which is the amount of time the sequential program would take to run the problem of the same size.

- Therefore, the sequential fraction is $\frac{14}{32{,}846} \ \approx 0.000426$ or 0.0426%.

Now we can take 0.000426 as the sequential fraction f.

- Plugging $f\ = 0.000426$ into the inequality of ***Amdahl's Law***,

$$\psi(n,p) \leq \frac{1}{f\ + \frac{1-f}{p}} = \frac{1}{0.00042 \quad \frac{1-0.000426}{32}} \approx 31.6\ ■$$

Performance Analysis: Karp-Flatt Metric

The Karp-Flatt Metric

Because Amdahl’s law and Gustafson's law ignore the parallel overhead term $\kappa(n, p)$, they can overestimate speedup or scaled speedup.

Karp and Flatt proposed another metric, called the ***experimentally determined serial fraction***, which can provide valuable performance insights.

We define the ***experimentally determined serial fraction*** e of a parallel computation as

$$e = \frac{(p-1)\sigma(n,p)+pk(n,p)}{(p-1)T(n,1)}$$

Performance Analysis: Karp-Flatt Metric

$$e = \frac{(p-1)\sigma(n,p)+p\ \ (n,p)}{(p-1)T(n,1)}$$

Hence,

$$e = p\,\frac{T(n,p)-T(n,1)}{(p-1)T(n,1)}$$

We may now rewrite the ***parallel execution time*** as

$$T(n,p) = \mathrm{T}(n,1)e + T(n,1)(1-e)/p$$

Since $\psi = T(n,1)/T(n,p)$, we have

$$T(n,1) = \mathrm{T}(n,p)\psi.$$

Then,

$$T(n,p) = \mathrm{T}(n,p)\psi e + T(n,p)\psi(1-e)/p$$

Performance Analysis: Karp-Flatt Metric

Rearranging $T(n,p) = \mathrm{T}(n,p)\psi e + T(n,p)\psi(1-e)/p$,

$$e = \frac{\frac{1}{\psi} - \frac{1}{p}}{1 - \frac{1}{p}}$$

Performance Analysis: Karp-Flatt Metric

The Karp-Flatt Metric

Given a parallel computation exhibiting speedup ψ on p processors, where $p > 1$, the experimentally determined serial fraction e is defined as

$$e = \frac{\frac{1}{\psi} - \frac{1}{p}}{1 - \frac{1}{p}}$$

Performance Analysis: Karp-Flatt Metric

The experimentally determined serial fraction is a useful metric for ***two*** reasons.

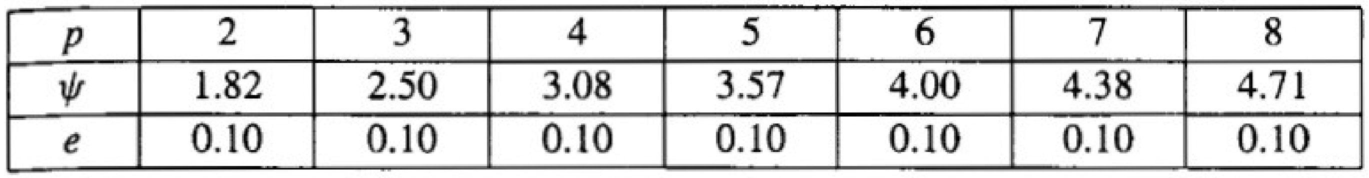
- It takes into account parallel overhead that Amdahl’s law and Gustafson's law ignore
- It can help us detect other sources of overhead or inefficiency that are ignored in our simple model of parallel execution time.

For a problem of fixed size, the efficiency of a parallel computation typically decreases as the number of processors increases.

By using the ***experimentally determined serial fraction***, we can determine whether this inefficiency decrease is due to

1. Limited opportunities for parallelism
2. Increases in algorithmic or architecture overhead

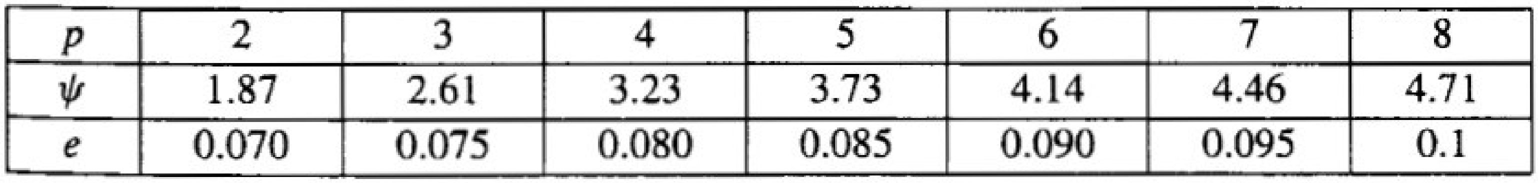
Performance Analysis: Karp-Flatt Metric



p	2	3	4	5	6	7	8
ψ	1.82	2.50	3.08	3.57	4.00	4.38	4.71
e	0.10	0.10	0.10	0.10	0.10	0.10	0.10

Since ***experimentally determined serial fraction*** is not increasing with the number of processors, the primary reason for the poor speedup is the limited opportunity for parallelism, that is, the large fraction of the computation that is inherently sequential.

Performance Analysis: Karp-Flatt Metric



p	2	3	4	5	6	7	8
ψ	1.87	2.61	3.23	3.73	4.14	4.46	4.71
e	0.070	0.075	0.080	0.085	0.090	0.095	0.1

Since ***experimentally determined serial fraction*** is steadily increasing with the number of processors, the primary reason for the poor speedup is parallel overhead.

Performance Analysis: Isoefficiency

The Isoefficiency Metric

We shall refer to a ***parallel program*** executing on a ***parallel computer*** as a ***parallel system***.

- The ***scalability*** of a parallel system is a measure of its ability to increase performance as the number of processors p increases.

Typically, ***speedup*** ψ and hence ***efficiency*** ε are typically an increasing function of the problem size n, because the communication complexity is usually lower than the computational complexity, i.e., ***Amdahl effect***.

- In order to maintain the same level of efficiency ε when the number of processors p is increased, we can increase the problem size n

Performance Analysis: Isoefficiency

$$\psi = \frac{\sigma + \varphi}{\sigma + \frac{\varphi}{p} + \kappa} = \frac{p(\sigma + \varphi)}{p\sigma + \varphi + p\kappa} = \frac{p(\sigma + \varphi)}{\sigma + \varphi + (p-1)\sigma + p\kappa}$$

Let us denote the total amount of time spent by all p processors doing work ***not done*** by the sequential program by $T_o(n, p)$.

- One component of this time is the time $p - 1$ processors spend waiting idly while the single processor executes the ***inherently sequential fraction***: $(p - 1)\sigma(n, p)$

- The other component of this time is the time all p processors spend performing interprocessor communications and redundant computations as part of ***parallelization overhead***: $p\kappa(n, p)$

$$T_o(n, p) = (p - 1)\,\sigma(n, p) + p\kappa(n, p)$$

Performance Analysis: Isoefficiency

Substituting $T_o(n,p)$ into the equation $\psi = \frac{p(\sigma + \varphi)}{\sigma + \varphi + (p-1)\sigma + p\kappa}$,

$$\psi = \frac{p(\sigma + \varphi)}{\sigma + \varphi + T_o}$$

Since $\varepsilon = \psi/p$,

$$\varepsilon = \frac{\sigma + \varphi}{\sigma + \varphi + T_o} = \frac{1}{1 + \frac{T_o}{\sigma + \varphi}}$$

Recalling that $T(n,1)$ represents ***sequential time execution***,

$$\varepsilon = \frac{1}{1 + \frac{T_o}{\sigma + \varphi}} = \frac{1}{1 + \frac{T_o}{T(n,1)}}$$

Performance Analysis: Isoefficiency

Rearranging the equation $\varepsilon(n,p) = \frac{1}{1+\frac{T_O(n,p)}{T(n,1)}}$,

$$T(n,1) = \frac{\varepsilon(n,p)}{1-\varepsilon(n,p)} T_O(n,p)$$

Observation: If we want to maintain a ***constant level of efficiency*** $\varepsilon(n,p)$, the term $\frac{\varepsilon(n,p)}{1-\varepsilon(n,p)}$ must be a ***constant***.

Therefore, the formula simplifies to

$$T(n,1) = CT_O(n,p)$$

Performance Analysis: Isoefficiency

Isoefficiency Relation

Suppose a parallel system exhibits efficiency $\varepsilon(n,p)$.

Define $C = \frac{\varepsilon(n,p)}{(1-\varepsilon(n,p))}$ and $T_o(n,p) = (p-1)\sigma(n,p) + p\kappa(n,p)$.

In order to maintain the same level of efficiency as the number of processors p increases, the problem size n must be increased so that the following equation holds:

$$T(n,1) = CT_o(n,p)$$

Performance Analysis: Isoefficiency

We can use a parallel system's ***isoefficiency relation*** to determine the range of the number of processors p for which ***a particular level of efficiency*** can be maintained.

- Since parallel overhead $\kappa(n, p)$ increases when the number of processors p increases, the way to maintain efficiency $\varepsilon(n, p)$ when increasing p is to increase the problem size n.

- The maximum problem size a parallel system can solve is limited by the ***amount of main memory*** that is available.
    - Therefore, we treat the amount of main memory as the limiting factor

Performance Analysis: Isoefficiency

Suppose a parallel system has ***isoefficiency relation*** of the form:

$$n = f\,(p)$$

Let $M(n)$ denote the amount of memory used to store a problem instance of size n.

Therefore,

$$M(n) = M(f\,(p))$$

$\frac{M(n)}{p}$ denotes the amount of memory per processor used to maintain a constant level of efficiency.

- $\frac{M(n)}{p}$ can be expressed as $\frac{M(f\,(p))}{p}$ in terms of the number of processors p.

Performance Analysis: Isoefficiency

Observe that the total amount of main memory is a linear function of the number processors p used.

The function $\frac{M(f(p))}{p}$ indicates how the amount of main memory used ***per processor*** must increase as a function of p in order to maintain the same level of efficiency.

- $\frac{M(f(p))}{p}$ is referred to as ***the scalability function***.
- The **lower** the complexity of ***the scalability function***, the ***more scalable*** a parallel system
- If $\frac{M(f(p))}{p} = \Theta(1)$, memory requirements per processor are ***constant***, and the parallel system is ***perfectly scalable***.

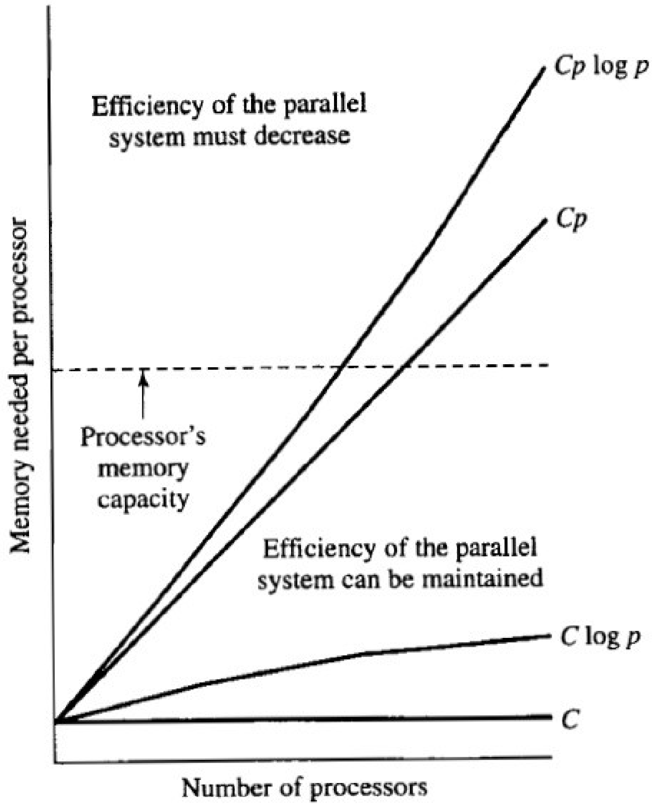
Performance Analysis: Isoefficiency

The way to maintain efficiency ε when increasing the number of processors p is to increase the size n of the problem being solved.

The ***maximum problem size*** n is limited by the amount of memory that is available, which is a linear function of the number of processors p.

Starting with the ***isoefficiency relation***, and taking into account memory requirements as a function of n, we can determine how the amount of memory used ***per processor*** must increase as a function of p to maintain the desired efficiency.

The **lower** the complexity of this function, the ***more scalable*** the parallel system.



Performance Analysis: Isoefficiency

Example 3

Consider a parallel system that performs a reduction operation on n values.

The complexity of the sequential reduction algorithm takes $\Theta(n)$.

The reduction step has time complexity $\Theta(\log p)$ in which ***every*** processor participates.

Therefore, $T_o(n, p) = p \cdot \Theta(\log p) = \Theta(p \cdot \log p)$.

Note that the ***Big-Oh*** notation ***ignores constants***, but we assume they are ***folded into*** the ***efficiency constant*** C.

Performance Analysis: Isoefficiency

Example 3 (continued)

Then, the isoefficiency relation for this parallel reduction algorithm is

$$n = C \cdot p \cdot \log p$$

The ***sequential*** algorithm reduces n values so the ***spatial*** complexity $M(n) = n$.

Therefore,

$$M(f(p)) = M(C \cdot p \cdot \log p) = C \cdot p \cdot \log p$$

Then,

$$\frac{M\big(f(p)\big)}{p} = \frac{C \cdot p \cdot \log p}{p} = C \cdot \log p$$

The problem size per processor must grow as $\Theta(\log p)$.

Performance Analysis: Isoefficiency

Example 4

Let’s determine the isoefficiency relation for a parallel implementation of Floyd’s algorithm.

The sequential algorithm has time complexity of $\Theta(n^3)$.

Each of the p processors executing the parallel algorithm spends $\Theta(n^2 \log p)$ time performing communications.

Hence, the isoefficiency relation is

$$n^3 = C \cdot p \cdot n^2 \cdot \log p$$

Thus,

$$n = C \cdot p \cdot \log p$$

Performance Analysis: Isoefficiency

Example 4 (continued)

The isoefficiency relation is

$$n = C \cdot p \cdot \log p$$

This looks like the same relation we had in ***Example 3***.

However, we have to be careful to consider the memory requirements associated with the problem size n.

In the case of Floyd's algorithm, the amount of memory needed to store a problem size n is n^2, that is, $M(n) = n^2$.

The scalability function of this parallel system is

$$\frac{M\big(f(p)\big)}{p} = \frac{M(C \cdot p \log \cdot p\ )}{p} = \frac{C^2 \cdot p^2 \cdot (\log\ p)^2}{p} = C^2 \cdot p \cdot (\log\ p)^2$$

Thus, this parallel system has poor scalability compared to parallel reduction in ***Example 3***.

Performance Analysis: Isoefficiency

Example 5

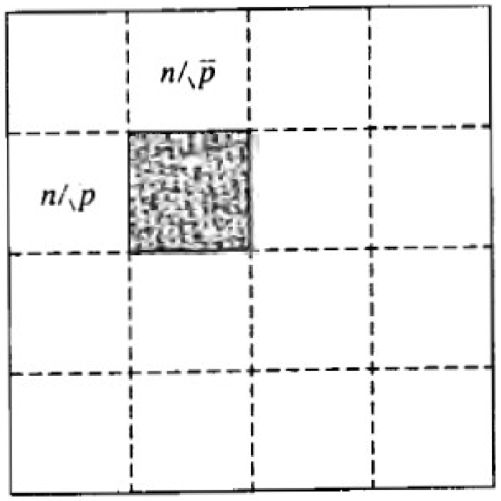
Consider a parallel system implementing a finite difference method to solve a partial differential equation.

The problem is represented by an $n \times n$ grid, where each processor is responsible for a subgrid of size $(\frac{n}{\sqrt{p}}) \times (\frac{n}{\sqrt{p}})$.

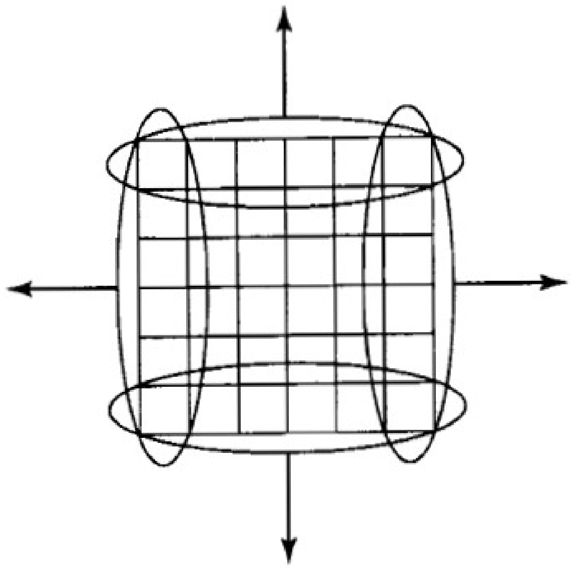
During each iteration of the algorithm, every processor sends boundary values to its neighbors; the time needed to perform these communications is $\Theta(n/\sqrt{p})$ per iteration.

The time complexity of the sequential algorithm solving this problem is $\Theta(n^2)$ per iteration.

Performance Analysis: Isoefficiency



(a)



(b)

Performance Analysis: Isoefficiency

Example 5 (continued)

The isoefficiency relation for this parallel system is

$$n^2 = C \cdot p \cdot \frac{n}{\sqrt{p}}$$

Thus,

$$n = C \cdot \sqrt{p}$$

A problem instance has size n so the grid contains n^2 elements.
Therefore, $M(n) = n^2$.

$$\frac{M\big(f(p)\big)}{p} = \frac{M(C\cdot\sqrt{p})}{p} = \frac{C^2\cdot p}{p} = C^2$$

The scalability function is $\Theta(1)$, meaning that the parallel system is ***perfectly scalable***.

Reference

Michael J. Quinn. 2003. Parallel Programming in C with MPI and OpenMP. McGraw-Hill Education Group.