Metrics to measure the performance of programs

Degree of Parallelism, average parallelism

Effective work

Speedup

Parallel efficiency
Degree of Parallelism

Degree of Parallelism (DOP)

- The number of processors engaged in execution at the same time
- Two forms of functions: continuous form and discreet form
Degrees of Parallelism

Factors that effect the DOP include:

- **Application properties**
  - Data dependency,
  - sequential parts of the application,
  - Communication overhead

- **Resource limitations**
  - number of processors,
  - memory, I/O

- **Algorithms**
  - how does the algorithm divide up work?
Effective Work

- This is the total amount of computation executed within a given time interval.
- Effective work Relates to DOP
Effective work

Calculating Effective work

- $n$ homogeneous processors
- Processing capacity of a single processor (execution rate) = $\Delta$
- $DOP(t)$ = Number of busy PEs at time $t$ in $[t_1, t_2]$
- Total effective work in continuous form:

\[
W = \Delta \int_{t_1}^{t_2} DOP(t) \, dt
\]

- In discrete form: (m is the maximum DOP between $t_1$ and $t_2$)

\[
W = \Delta \sum_{i=1}^{m} i \cdot t_i
\]

where $t_i$ is the total time that $DOP = i$ and $\sum_{i=1}^{m} t_i = t_2 - t_1$
Average Parallelism

Average parallelism:

- **Continuous form:**

  \[ A = \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} \text{DOP}(t) \, dt \]

- **Discrete form:**

  \[ A = \frac{\sum_{i=1}^{m} i \cdot t_i}{\sum_{t=1}^{m} t_i} \]
Speedup

We desire to know the improvement (or not) brought about by parallelising an application code.

The improvement can be measured by *speedup*

- In the simplest form, speedup is the ratio of execution time of a serial implementation to the execution time of a parallel implementation.

If *n* processors are used, then:

\[ S(n) = \frac{t_1}{t_n} \]

- \( t_1 \) is the worst case execution time of the optimal serial algorithm.
- \( t_n \) is the worst case execution time of the parallel algorithm using \( n \) processors.
Speedup

What is “good” speedup?

- Linear speedup is regarded as optimal
- Maximum speedup for a parallel algorithm with $n$ processors is $n$.

To illustrate this:

- Consider the execution time of an application is $t_1$
- The application is split into $n$ processes
- Assume no overheads, communications, synchronisation etc.
- The least execution time is $t_1 / n$
- So the maximum speedup is $S(n) = t_1 / (t_1 / n) = n$

- Not always true (we may achieve superlinearity in some special circumstances)
Speedup

After reaching a maximum speedup. Adding further processors is of no benefit and will harm performance.
Speedup

Some tasks can exceed linear speedup.

- This is superlinear speedup ($S(n) > n$)

Reasons

- Evidence of sub-optimal sequential algorithm.
- Cache effects
  - Not stable, problem size / data may alters execution in unpredictable ways.
Speedup

How to obtain good $S(n)$ for large $n$?

- The applications can exhibit good speedup, when:
  - The application has a good percentage of inherent (data) parallelism.
  - Minimal sequential component.
  - Minimal communication overhead.
  - Load balancing techniques are employed.

- Maintaining a high ratio of computation to communication is helpful
  - Increase the size of the work run by each processor.
  - Low frequency of communications.
Parallel efficiency

Parallel efficiency:

\[ E(n) = \frac{S(n)}{n} \]

Parallel programs are not usually 100% efficient, i.e. \( S(n) \ll n \)

Main issues that affect parallel efficiency are:

- **Ratio of computation to communication**
  - Divide up the work into a proper granularity size

- **Communication bandwidth & latency**
  - High bandwidth and low latency reduce the communication overhead
Iso-efficiency

Constant Efficiency:

- how the amount of computation performed (N) must scale with processor number P to keep parallel efficiency E constant
- The function of N over P is called an algorithm’s iso-efficiency function
- An algorithm with an iso-efficiency function of $O(P)$ is highly scalable
- An algorithm with a quadratic or exponential iso-efficiency function is less scalable

$$E_2 = \frac{5N^2}{5N^2 + 2P^2 + 4NP}$$

- In this example, N=P
Achieving High Parallel Performance

Communication has crucial impact on the performance of parallel programming

How to reduce the impact of communication:

- Minimize the amount of communication (e.g. by maintaining a good data locality)
- Overlap communications with computation where possible.
- Reduce latency and overhead by sending a few large messages, rather than a lot of small messages.
- At the hardware level, can reduce latency by using fast (but expensive) communications.
Four approaches to modelling application performance

- Speedup
- Amdahl’s law
- Asymptotic analysis
- Modelling execution time
Speedup approach

Using speedup approach, we can say something like “this algorithm achieved a speedup of $S$ on $p$ processors with problem size $N$”

This approach can give us some ideas about the algorithm quality, but we cannot judge the quality of an algorithm by this single data

Elaborate this point in the following example
Consider a sequential algorithm and its optimal execution time $T = N + N^2$, where $N$ is the problem size.

- **Parallel Algorithm 1:** $T = N + (N^2 / p)$
  - Partitions the computationally expensive $O(N^2)$
  - No other costs.

- **Parallel Algorithm 2:** $T = ((N + N^2) / p) + 100$
  - Partitions the whole computation
  - Introduces fixed overhead cost of 100.

- **Parallel Algorithm 3:** $T = ((N + N^2) / p) + 0.6p^2$
  - Partitions the whole computation
  - Introduces variable overhead cost of $0.6p^2$
These algorithms all achieve a speedup of about 10.8 when $p = 12$ and $N = 100$, but differentiates with each other when $p$ becomes large.
Amdahl’s Law

Applications may contain elements that are not amenable to parallelisation.

Let this serial fraction be $f$:

- If we make the remaining part $n$ times faster by running it on $n$ processors, then the time $T_n$ is:

$$T_n = \frac{(1-f)T_1}{n} + fT_1$$

- Hence, speedup is:

$$S(n) = \frac{n}{(1-f) + nf} \leq \frac{1}{f}$$

- For example, an application does a final (non-parallelisable) collective operation at the end of each iteration which accounts for 8% of the computation time - the maximum achievable speedup is 12.5.

- This is Amdahl’s Law.
Application of Amdahl’s law

Part A takes 75% and part B takes 25% of the whole computation time.

If we make part B 5 times faster, then

$$ speedup = \frac{1}{\frac{0.25}{5} + 0.75} = 1.25 $$

If we make part A 2 times faster, then

$$ speedup = \frac{1}{0.25 + \frac{0.75}{2}} = 1.6 $$

Therefore, making A twice faster is better than making B five times faster.
Amdahl’s Law

- Amdahl’s law shows us the limitation of parallelising codes

- Disadvantages
  - Can only tell the upper bound of the speedup for a particular algorithm
  - Cannot tell if other algorithms with greater parallelism exist for the problem.
Asymptotic analysis

In this modelling approach, we can say something like “the algorithm takes the time of $O(n \log n)$ on $n$ processors”

Disadvantage:

- Ignore the lower-order term:
  - e.g. $O(n \log n)$ could be $10n + n \log n$, when $n$ is small, $10n$ domains

- Only tell the order of the execution time of a program, not its actual execution time:
  - e.g. $1000n \log n$ is better than $10n^2$ when $n$ exceeds a certain value, but $10n^2$ is less than $1000n \log n$ when $n$ is less than 996