Parallel Numerical Algorithms

Chapter 4 – Parallel Performance

Prof. Michael T. Heath

Department of Computer Science
University of Illinois at Urbana-Champaign

CS 554 / CSE 512

Outline

Parallel Efficiency

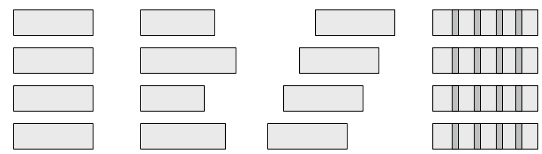
Efficiency: effectiveness of parallel algorithm relative to its serial counterpart (more precise definition later)

Factors determining efficiency of parallel algorithm

- *Load balance*: distribution of work among processors
- *Concurrency*: processors working simultaneously
- *Overhead*: additional work not present in corresponding serial computation

Efficiency is maximized when load imbalance is minimized, concurrency is maximized, and overhead is minimized

Parallel Efficiency



(a) (b) (c) (d)

(a) perfect load balance and concurrency
(b) good initial concurrency but poor load balance
(c) good load balance but poor concurrency
(d) good load balance and concurrency but additional overhead

Basic Definitions

- Memory (M) — amount of storage required (e.g., words) for given problem
- Work (W) — number of operations (e.g., flops) required for given problem, including loads and stores
- Velocity (V) — number of operations per unit time (e.g., flops/sec) performed by one processor
- Time (T) — elapsed wall-clock time (e.g., secs) from beginning to end of computation
- Cost (C) — product of number of processors and execution time (e.g., processor-seconds)

Basic Definitions

- Subscript indicates number of processors used (e.g., T_1 is serial execution time, W_p is work using p processors, etc.)
- We will assume $M_p \geq M_1$, and with no replication of data it is reasonable to assume $M_p = M_1$ for $p \geq 1$, in which case we drop subscript and write just M
- If serial algorithm is optimal and we disregard chance effects, then $W_p \geq W_1$, and in general $W_p > W_1$ for $p > 1$
- *Parallel overhead*: $O_p \equiv W_p - W_1$

Basic Definitions

- Amount of data often determines amount of computation, in which case we may write $W(M)$ to indicate dependence of computational complexity on storage complexity
- For example, when multiplying two full matrices of order n, $M = \Theta(n^2)$ and $W = \Theta(n^3)$, so $W(M) = \Theta(M^{3/2})$
- Since every data item is likely to be used in at least one operation, it is reasonable to assume that work W grows at least linearly with memory M

Processor Speed

- Due to memory hierarchy, effective processor speed depends on amount of memory used
- Assuming processors are identical, we will have $V_p(M) = V_1(M) = V(M)$ for any given M, but in general we may have $V(M) \neq V(N)$ if $M \neq N$
- In particular, for evenly distributed data across p processors, we may have $V(M/p) > V(M)$, since M/p may fit in faster memory for sufficiently large p
- Aggregate speed of p processors is $p\,V(M/p)$

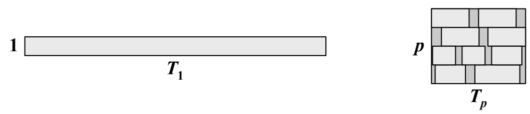
Execution Time and Cost

Execution time $=$ (total work)/(aggregate speed)

- Serial execution time: $T_1 = W_1/V(M)$
- Parallel execution time: $T_p = W_p/(p\,V(M/p))$

Cost $=$ (number of processors) $\times$ (execution time)

- Serial cost: $C_1 \equiv T_1 = W_1/V(M)$
- Parallel cost: $C_p \equiv p\,T_p = W_p/V(M/p)$



Efficiency and Speedup

- *Efficiency*:

$$E_p \equiv \frac{\text{serial cost}}{\text{parallel cost}} = \frac{C_1}{C_p} = \frac{T_1}{p\,T_p} = \frac{W_1}{W_p}\,\frac{V(M/p)}{V(M)}$$

- *Speedup*:

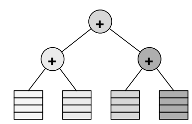
$$S_p \equiv \frac{\text{serial time}}{\text{parallel time}} = \frac{T_1}{T_p} = p\,E_p$$

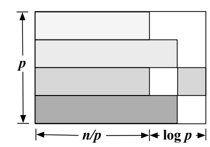
Superlinear Speedup

- If $V(M/p) = V(M)$, then $E_p = W_1/W_p$, and hence $E_p \le 1$ and $S_p \le p$, since we assume $W_p \ge W_1$
- But if $V(M/p) > V(M)$, then we may have $E_p > 1$ and $S_p > p$, depending on whether gain in processor speed offsets increase in work for parallel algorithm
- This often happens when M/p fits in cache but M does not, or when M exceeds main memory of uniprocessor and secondary storage must be used for serial solution
- However, for simplicity we will assume V is constant ($= 1$) when analyzing algorithms, and that $n \ge p$

Example: Summation

- Problem: compute sum of n numbers
- Using p processors, each processor first sums n/p numbers
- Subtotals are then summed in tree-like fashion to obtain grand total





Example: Summation

Serial

- $M_1 = n$
- $W_1 = n - 1 \approx n$
- $T_1 \approx n$
- $C_1 \approx n$

Parallel

- $M_p = n$
- $W_p = p\,(n/p - 1 + \log p) \approx n + p\,\log p$
- $T_p \approx n/p + \log p$
- $C_p \approx n + p\,\log p$

$$E_p = \frac{C_1}{C_p} \approx \frac{n}{n + p\,\log p} = \frac{1}{1 + (p\,\log p)/n}$$

$$S_p = \frac{T_1}{T_p} \approx \frac{n}{n/p + \log p} = \frac{p}{1 + (p\,\log p)/n}$$

Parallel Scalability

- *Scalability*: relative effectiveness with which parallel algorithm can utilize additional processors
- Minimal criterion: algorithm is *scalable* if its efficiency is bounded away from zero as number of processors grows without bound, or equivalently, $E_p = \Theta(1)$ as $p \to \infty$
- Algorithm scalable in this sense may still be impractical if growth rate of problem size causes total execution time to grow unacceptably

Parallel Scalability

Why use more processors?

- solve given problem in less time
- solve larger problem in same time
- obtain sufficient memory to solve given (or larger) problem
- solve ever larger problems regardless of execution time

Larger problems require more memory M and work W_1, e.g.,

- finer resolution or larger domain in atmospheric simulation
- more particles in molecular or galactic simulations
- additional physical effects or greater detail in modeling

Problem Scaling

Problem size characterized by growth rate required to keep some quantity constant as number of processors increases

Some possible invariants for scaling problem size

- serial work: $W_1 = \Theta(1)$
- execution time: $T_p = \Theta(1)$
- serial work per processor: $W_1 = \Theta(p)$
- memory per processor: $M = \Theta(p)$
- accuracy
- efficiency: $E_p = \Theta(1)$

Fixed Serial Work

- Sometimes called *strong scaling*
- Using more processors to solve fixed problem may reduce execution time initially, but gain diminishes as p grows
- Execution time may even increase due to increased parallel overhead
- If $p > M$, then some processors have no data, and if $p > W_1$, then some processors have no work!
- In practice, potential parallelism for fixed problem is exhausted long before these extremes are reached
- In any case, $E_p \to 0$ as $p \to \infty$, so *no* algorithm is scalable for fixed problem

Example: Summation

- For summation example

$$E_p = \frac{1}{1 + (p \log p)/n}$$

so efficiency is high when $n \gg p$, but for fixed n, $E_p \to 0$ as $p \to \infty$

- Once $p > n$, additional processors have no data or work
- Algorithm is *not* scalable for fixed problem

Amdahl's Law

- Assume fraction s of work for given problem is serial, with $0 \le s \le 1$, while remaining portion, $1 - s$, is p-fold parallel
- Then

$$\begin{aligned} T_p &= s\,T_1 + (1-s)\,\frac{T_1}{p} \\ E_p &= \frac{T_1}{p\,T_p} = \frac{1}{s\,p + (1-s)} \\ S_p &= \frac{T_1}{T_p} = \frac{p}{s\,p + (1-s)} \end{aligned}$$

and hence $E_p \to 0$ and $S_p \to 1/s$ as $p \to \infty$

- For example, if serial fraction exceeds 1 percent, then speedup can never exceed 100 for any p

Fixed Execution Time

- Maintaining fixed execution time is applicable when computation must be completed within strict time limit (e.g., real-time constraints) or when user wishes to maintain given turn-around time
- If processor speed V is constant, then $W_p = p\,T_p V$, so if T_p is constant, then W_p grows linearly with p
- If W_p grows faster than linearly with memory M, then M must grow sublinearly with p to maintain constant T_p
- Thus, with fixed execution time, algorithm cannot be scalable unless both memory M and work W_p grow at most linearly with p

Example: Summation

- For summation example, $T_p \approx n/p + \log p$
- To maintain constant $T_p = T$, must have $n = p\,(T - \log p)$, which is impossible for $p \ge e^T$
- Even for $p < e^T$, we have $E_p = 1 - (\log p)/T$, which decreases with increasing p
- Algorithm is *not* scalable with fixed execution time

Fixed Serial Work per Processor

- Sometimes called *weak scaling*
- Not particularly natural in applications, but useful as scalability measure
- If $W_p = \Theta(p)$, then $T_p = W_p/p$ would be constant, so any increase in T_p indicates superlinear growth in parallel overhead O_p
- $E_p = W_1/W_p \to 0$ unless $W_p = \Theta(p)$, so algorithm is not scalable unless parallel overhead grows at most linearly with p

Example: Summation

- Summing $p\,n$ numbers using p processors, we have

$$E_p = \frac{W_1}{W_p} \approx \frac{p\,n}{p\,n + p\,\log p} = \frac{1}{1 + (\log p)/n} \to 0$$

so algorithm is *not* scalable with fixed serial work per processor

Fixed Memory per Processor

- Applicable for problems that saturate all available memory, which in distributed-memory multicomputer grows linearly with number of processors
- If W_1 grows linearly with M, then this invariant is same as fixed serial work per processor
- But if W_1 grows faster than linearly with M, then execution time T_p grows superlinearly with p, so algorithm may be impractical even if efficiency is reasonable

Fixed Accuracy

- For some problems, desired accuracy of solution determines amount of memory and work required
- It is pointless to increase problem size beyond that necessary to achieve desired accuracy
- Choice of resolution can affect serial work W_1 in subtle and complex ways
  - conditioning of problem
  - convergence rate for iterative method
  - length of time step for time-dependent problem

Fixed Efficiency

- Previous scaling invariants determined rate of growth in problem size, and then we analyzed resulting efficiency to determine scalability
- More direct approach is to use efficiency itself as scaling invariant, i.e., we determine minimum growth rate in problem size required to maintain *constant* efficiency
- If this is possible, then algorithm is scalable, but it may still be impractical if required growth rate in problem size is excessive, leading to unacceptably large execution time
- Thus, resulting growth rate in problem size determines *degree* to which algorithm is scalable

Isoefficiency Function

- W_1 typically expressed as function $W_1(n)$ of some parameter n characterizing problem (e.g., $W_1 = \Theta(n^3)$ for multiplying two matrices of order n)
- W_p depends on both n and p, so we write $W_p(n, p)$
- Relationship $W_1(n) - E\,W_p(n, p) = 0$ for constant E implicitly defines n as function of p, which we write as $n(p)$
- *Isoefficiency function*: $W_1(n(p))$
- Isoefficiency function for given p is minimum problem size required to maintain given constant efficiency E
- Specific value depends on E, but in practice only its order of magnitude is of interest

Example: Isoefficiency

- In summation example, for

$$\begin{aligned} W_1 &= E\,W_p \\ n &\approx E\,(n + p\,\log p) \end{aligned}$$

to hold for constant E, must have $n = \Theta(p\,\log p)$, so isoefficiency function is

$$W_1(n(p)) = \Theta(p\,\log p)$$

- So if number of numbers to be summed grows like $p\,\log p$, then efficiency is constant and algorithm is scalable
- Note, however, that execution time T_p grows like $\log p$

Isoefficiency and Scalability

- $T_p = W_1/(p\,E)$ is constant if isoefficiency function is $\Theta(p)$, but otherwise T_p grows with p
- Growth rate of T_p may or may not be acceptable
- Isoefficiency function of $\Theta(p)$ is desirable, but for many problems is not attainable
- More achievable isoefficiency function is $\Theta(p\,\log p)$ or $\Theta(p\sqrt{p})$, for which T_p grows relatively slowly, like $\log p$ or $\sqrt{p}$, respectively, which may be acceptable
- Algorithm with isoefficiency function $\Theta(p^2)$ or higher has poor scalability, since T_p grows at least linearly with p

Modeling Parallel Work

In message-passing model of computation, parallel work can be subdivided into computation, communication, and idle:

$$W_p = W_{\text{comp}} + W_{\text{comm}} + W_{\text{idle}}$$

- W_{comp}: serial work W_1 plus any additional computational work due to parallel execution, such as replicated or speculative work
- W_{comm}: time spent sending and receiving messages
- W_{idle}: time spent idling due to lack of computational work or lack of necessary data

Reducing Idle Time

- Idle time due to lack of work can be reduced by improving load balance, if possible
- Idle time due to lack of data can be reduced by overlapping computation and communication
- Assigning more than one process per processor allows possibility of executing another process whenever given process is blocked awaiting data (*multithreading*)

Example: 3-D Grid Computation

Consider 3-D, $n \times n \times z$ finite difference grid, where n is number of grid points in each of two horizontal dimensions, and z is number of grid points in vertical dimension (typically $z \ll n$)

- *Partition*: assign one grid point per fine-grain task
- *Communicate*: 9-point horizontal stencil
- *Agglomerate*: First, consider 1-D agglomeration along one horizontal dimension of 3-D grid, with subgrid of size $n \times (n/p) \times z$ assigned to each coarse-grain task

Example: 3-D Grid, 1-D Agglomeration

Work:

- With no replicated computation,

$$W_{\text{comp}} = t_c n^2 z$$

  where t_c is computation time per grid point

- Each task exchanges $2nz$ grid points with each of its two neighbors, so

$$W_{\text{comm}} = p\,(2t_s + 4t_w nz)$$

- Assuming p divides n and no idle time waiting for messages,

$$W_{\text{idle}} = 0$$

Example: 3-D Grid, 1-D Agglomeration

Execution time:

$$T_p = \frac{W_p}{p} = \frac{W_{\text{comp}} + W_{\text{comm}} + W_{\text{idle}}}{p} = t_c\frac{n^2 z}{p} + 2t_s + 4t_w nz$$

- T_p decreases with increasing p, but is bounded below by cost of exchanging two array slices
- T_p increases with increasing n, z, t_c, t_s, and t_w

Example: 3-D Grid, 1-D Agglomeration

Efficiency:

$$E_p = \frac{W_1}{W_p} = \frac{T_1}{p\,T_p} = \frac{t_c n^2 z}{t_c n^2 z + 2t_s p + 4t_w nzp}$$

- E_p decreases with increasing p, t_s, and t_w
- E_p increases with increasing n, z, and t_c

Example: 3-D Grid, 1-D Agglomeration

Isoefficiency function:

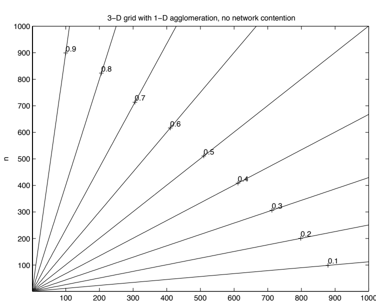
- To maintain constant efficiency, must have

$$t_c n^2 z \approx E\,(t_c n^2 z + 2t_s p + 4t_w nzp)$$

  which holds for sufficiently large p if $n = \Theta(p)$

- Since $W_1 = \Theta(n^2)$, isoefficiency function is $\Theta(p^2)$
- Isoefficiency contours as function of n and p for particular choice of parameters shown next

Example: 3-D Grid, 1-D Agglomeration



Isoefficiency contours using parameter values $t_c = 20$, $t_s = 100$, $t_w = 5$, $z = 10$

Example: 3-D Grid, 2-D Agglomeration

- Next consider 2-D agglomeration along both horizontal dimensions of 3-D grid, with subgrid of size $(n/\sqrt{p}) \times (n/\sqrt{p}) \times z$ assigned to each coarse-grain task
- W_{comp} remains same as before, and assuming $\sqrt{p}$ divides n, load balance is uniform and $W_{\text{idle}} = 0$
- Each task exchanges $2(n/\sqrt{p})z$ points with each of its four neighbors, so

$$W_{\text{comm}} = p\,(4t_s + 8t_w nz/\sqrt{p}\,) = 4t_s p + 8t_w nz\sqrt{p}$$

Example: 3-D Grid, 2-D Agglomeration

Execution time:

$$T_p = \frac{W_p}{p} = \frac{W_{\text{comp}} + W_{\text{comm}} + W_{\text{idle}}}{p} = t_c\frac{n^2 z}{p} + 4t_s + 8t_w nz/\sqrt{p}$$

Efficiency:

$$E_p = \frac{W_1}{W_p} = \frac{T_1}{p\,T_p} = \frac{t_c n^2 z}{t_c n^2 z + 4t_s p + 8t_w nz\sqrt{p}}$$

Example: 3-D Grid, 2-D Agglomeration

Isoefficiency function:

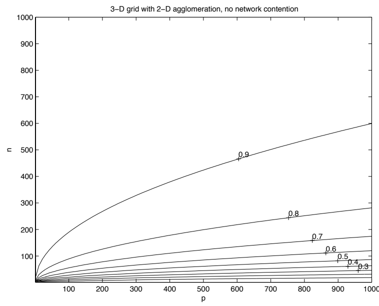
- To maintain constant efficiency, must have

$$t_c n^2 z \approx E\,(t_c n^2 z + 4t_s p + 8t_w nz\sqrt{p}\,)$$

  which holds if $n = \Theta(\sqrt{p}\,)$

- Since $W_1 = \Theta(n^2)$, isoefficiency function is $\Theta(p)$, so 2-D agglomeration is much more scalable than 1-D agglomeration
- Isoefficiency contours as function of n and p for particular choice of parameters shown next
- For given p, far smaller problem is required to achieve given efficiency than with 1-D agglomeration

Example: 3-D Grid, 2-D Agglomeration



Isoefficiency contours using parameter values $t_c = 20$, $t_s = 100$, $t_w = 5$, $z = 10$

Example: 3-D Grid with Network Contention

- Previous analysis assumed no contention for communication bandwidth, which is valid for many networks but not for bus, for which only one processor can send at a time
- If half of processors are sending and other half receiving simultaneously, then contention factor is $S = p/2$
- For 3-D grid with 1-D agglomeration on bus network,

$$W_{\text{comm}} = p\,(2t_s + t_w S(4nz)) = 2t_s p + 2t_w nzp^2$$

Example: 3-D Grid with Network Contention

Efficiency:

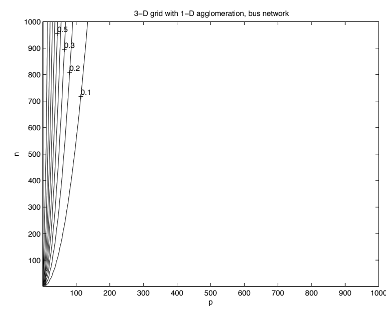
$$E_p = \frac{W_1}{W_p} = \frac{T_1}{p\,T_p} = \frac{t_c n^2 z}{t_c n^2 z + 2t_s p + 2t_w nzp^2}$$

- To maintain constant efficiency, must have

$$t_c n^2 z \approx E\,(t_c n^2 z + 2t_s p + 2t_w nzp^2)$$

  which holds if $n = \Theta(p^2)$
- Since $W_1 = \Theta(n^2)$, isoefficiency function is $\Theta(p^4)$, which indicates extremely poor scalability of algorithm on this network, as can also be seen from isoeffiency contours shown next

Example: 3-D Grid with Network Contention



Isoefficiency contours using parameter values $t_c = 20$, $t_s = 100$, $t_w = 5$, $z = 10$

Example: 3-D Grid with Network Contention

For 3-D grid with 2-D agglomeration on bus network,

$$W_{\text{comm}} = 4t_s p + t_w S(8nz\sqrt{p}) = 4t_s p + 4t_w nzp^{3/2}$$

$$E_p = \frac{W_1}{W_p} = \frac{T_1}{p\,T_p} = \frac{t_c n^2 z}{t_c n^2 z + 4t_s p + 4t_w nzp^{3/2}}$$

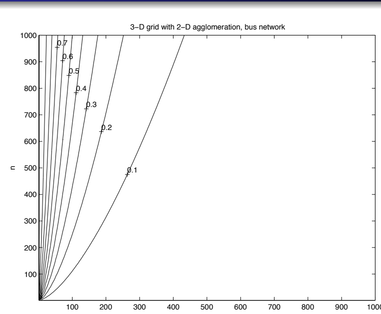
Example: 3-D Grid with Network Contention

- To maintain constant efficiency, must have

$$t_c n^2 z \approx E\,(t_c n^2 z + 4t_s p + 4t_w nzp^{3/2})$$

  which holds if $n = \Theta(p^{3/2})$
- Since $W_1 = \Theta(n^2)$, isoefficiency function is $\Theta(p^3)$, which is slightly better scalability than with 1-D agglomeration, but still quite poor
- Corresponding isoefficiency contours shown next

Example: 3-D Grid with Network Contention



Isoefficiency contours using parameter values $t_c = 20$, $t_s = 100$, $t_w = 5$, $z = 10$

References

- D. L. Eager, J. Zahorjan, and E. D. Lazowska, Speedup versus efficiency in parallel systems, *IEEE Trans. Comput.* 38:408-423, 1989
- A. Grama, A. Gupta, and V. Kumar, Isoefficiency: measuring the scalability of parallel algorithms and architectures, *IEEE Parallel Distrib. Tech.* 1(3):12-21, August 1993
- J. L. Gustafson, Reevaluating Amdahl's law, *Comm. ACM* 31:532-533, 1988
- V. Kumar and A. Gupta, Analyzing scalability of parallel algorithms and architectures, *J. Parallel Distrib. Comput.* 22:379-391, 1994

References

- D. M. Nicol and F. H. Willard, Problem size, parallel architecture, and optimal speedup, *J. Parallel Distrib. Comput.* 5:404-420, 1988
- J. P. Singh, J. L. Hennessy, and A. Gupta, Scaling parallel programs for multiprocessors: methodology and examples, *IEEE Computer*, 26(7):42-50, 1993
- X. H. Sun and L. M. Ni, Scalable problems and memory-bound speedup, *J. Parallel Distrib. Comput.*, 19:27-37, 1993
- P. H. Worley, The effect of time constraints on scaled speedup, *SIAM J. Sci. Stat. Comput.*, 11:838-858, 1990