

# Parallel Numerical Algorithms

## Chapter 2 – Parallel Thinking Section 2.3 – Parallel Performance

Michael T. Heath and Edgar Solomonik

Department of Computer Science  
University of Illinois at Urbana-Champaign

CS 554 / CSE 512

# Outline

- 1 Efficiency
  - Parallel Efficiency
  - Basic Definitions
  - Execution Time and Cost
  - Efficiency and Speedup
- 2 Scalability
  - Definition
  - Problem Scaling
  - Isoefficiency
- 3 Example
  - Atmospheric Flow Model
  - 1-D Agglomeration
  - 2-D Agglomeration

# Parallel Efficiency

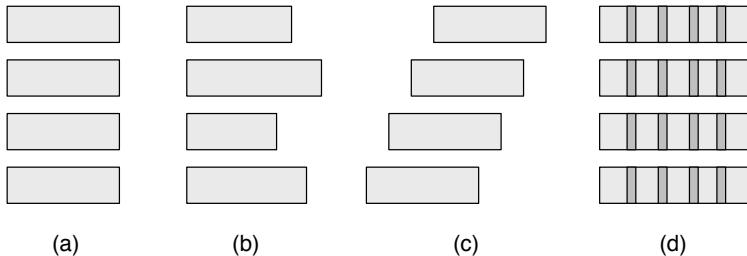
*Efficiency*: effectiveness of parallel algorithm relative to its serial counterpart (more precise definition later)

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) 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

# Algorithm Attributes

- **Memory** ( $M$ ) — overall memory footprint of the algorithm in words
- **Work** ( $Q$ ) — total number of operations (e.g., flops) computed by algorithm, including loads and stores
- **Depth** ( $D$ ) — longest sequence (chain) of dependent work operations
- **Time** ( $T$ ) — elapsed wall-clock time (e.g., secs) from beginning to end of computation, expressed using
  - $\alpha$  — time to transfer a 0-byte message
  - $\beta$  — bandwidth cost (per-word)
  - $\gamma$  — time to perform one local operation (unit work)

Note that effective  $\gamma$  is generally between the time to compute a floating point operation and the time to load/store a word, depending on local computation performed

# Scaling of Algorithm Attributes

- Subscript indicates number of processors used (e.g.,  $T_1$  is serial execution time,  $Q_p$  is work using  $p$  processors, etc.)
- We assume the *input size*, an attribute of the **problem** rather than the **algorithm**, is  $M_1$
- Most algorithms we study will be *memory efficient*, meaning  $M_p = M_1$  in which case we drop subscript and write just  $M$
- If serial algorithm is optimal then  $Q_p \geq Q_1$
- *Parallel work overhead*:  $O_p := Q_p - Q_1$

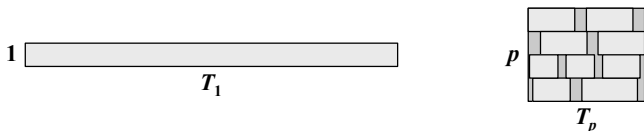
# Basic Definitions

- Amount of data often determines amount of computation, in which case we may write  $Q(M)$  to indicate dependence of computational complexity on the input size
- For example, when multiplying two full matrices of order  $n$ ,  $M = \Theta(n^2)$  and  $Q = \Theta(n^3)$ , so  $Q(M) = \Theta(M^{3/2})$
- In numerical algorithms, every data item is typically used in at least one operation, so we generally assume that work  $Q$  grows at least linearly with the input size  $M$

# Execution Time and Cost

Execution time  $\geq$  (total work)/(overall processor speed)

- Serial execution time:  $T_1 = \gamma Q_1$
- Parallel execution time:  $T_p \geq \gamma Q_p/p$



We can quantify  $T_p$  in terms of the critical path cost (sum of costs of longest chain of dependent subtasks)

**Cost** :=  $(L, W, F)$  := (#messages, #words, #flops)

$$\max(\alpha L, \beta W, \gamma F) \leq T_p \leq \alpha L + \beta W + \gamma F$$



# Efficiency and Speedup

- *Speedup*:

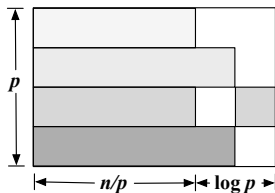
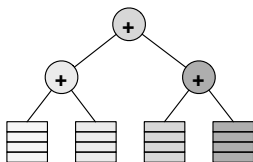
$$S_p := \frac{\text{serial time}}{\text{parallel time}} = \frac{T_1}{T_p}$$

- *Efficiency*:

$$E_p := \frac{\text{speedup}}{\text{number of processors}} = \frac{S_p}{p}$$

# 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

Generally,  $\alpha \gg \beta \gg \gamma$ , which we use to simplify analysis

## *Serial*

- $M_1 = n$
- $Q_1 \approx n$
- $T_1 \approx \gamma n$

## *Parallel*

- $M_p = n$
- $Q_p \approx n$
- $T_p \approx \alpha \log(p) + \gamma n/p$

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

$$E_p = \frac{S_p}{p} \approx \frac{1}{1 + (\alpha/\gamma)(p/n) \log p}$$

To achieve a good speed-up want  $\alpha/\gamma$  to be small and  $n \gg p$

# Parallel Scalability

- *Scalability*: relative effectiveness with which parallel algorithm can utilize additional processors
- A 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  $p \rightarrow \infty$
- Algorithm scalability in this sense is impractical unless we permit the input size to grow or bound the number of processors used

# 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  $M_1$  and work  $Q_1$ , 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

The relative parallel scaling of different algorithms for a problem can be studied by fixing

- input size: constant  $M_1$
- input size per processor: constant  $M_1/p$

The relative parallel scaling of different parallelizations of an algorithm can be studied by fixing

- amount of work per processor: constant  $Q_1/p$
- efficiency: constant  $E_p$
- time: constant  $T_p$

In all cases, we seek to quantify the relationship between parameters of the problem/algorithm with respect to the performance (time/efficiency)

# Strong Scaling

*Strong scaling* – solving the same problem with a growing number of processors (constant input size)

- Ideal strong scaling to  $p$  processors requires  $T_p = T_1/p$
- When problem is not embarrassingly parallel, the best we can hope for is  $T_p \approx T_1/p$  (i.e.,  $E_p \approx 1$ ) up to some  $p$
- We say an algorithm is *strongly scalable* to  $p_s$  processors if

$$E_{p_s} = \Theta(1)$$

i.e., we seek to asymptotically characterize the function  $p_s(Q_1)$  such that  $E_{p_s(Q_1)}(Q_1) = \text{const}$  for any  $Q_1$

## Example: Summation

- For summation example,

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

- The binary tree summation algorithm is therefore strongly scalable to  $p_s = \Theta((\gamma/\alpha)n / \log((\gamma/\alpha)n))$  processors
- The term  $\alpha/\gamma$  is constant for a given architecture, but can range from  $10^3$  to  $10^6$  on various machines
- Ignoring the dependence on this constant, the algorithm is strongly scalable to  $p_s = \Theta(n / \log(n))$  processors



# Basic Bounds on Strong Scaling

- Since all processors have work to do only if  $Q_p/p \geq 1$  for any  $p$  the speed-up is bounded by

$$S_p \leq \frac{Q_1}{Q_p/p} \leq Q_1$$

- It is possible but rare to achieve  $S_p > M_1$  by using additional memory  $M_p > M_1$ , as otherwise some processors have no data to work on

# Amdahl's Law

- *Amdahl's law*: if a fraction  $1/s$  of the computation is done sequentially, the achievable speed-up is at most  $s$
- Refers to most expensive unparallelized section of code
- Recall that the depth ( $D$ ) of an algorithm is the longest chain of dependent operations, i.e., this chain of operations is *inherently sequential*
- Amdahl's law implies that

$$S_p = \frac{T_1}{T_p} \leq \frac{Q_1 \gamma}{D \gamma} = \frac{Q_1}{D}$$

in words, **speedup  $\leq$  work / depth**

- The law provides a basic strong scaling limit  $p_s = O(Q/D)$ , although communication cost often gives a tighter bound

# Weak Scaling

We refer to *weak scaling* as solving a problem with a fixed input size per processor ( $M_1/p = \text{const}$ )

- In literature, weak scaling often refers to fixed work per processor  $Q_1/p$ , which is the same only if  $Q_1(M_1) = \Theta(M_1)$
- This scaling mode ( $M_1/p = \text{const}$ ) is natural when parallelism is being used to solve larger problems
- An algorithm is *weakly scalable* to  $p_w$  processors if

$$E_{p_w}(p_w M_0) = \Theta(1) \quad \Rightarrow \quad \frac{T_{p_w}(p_w M_0)}{T_1(M_0)} = \Theta\left(\frac{Q_1(p_w M_0)}{p_w Q_1(M_0)}\right)$$

meaning when increasing  $p$  with constant  $M_1/p = M_0$ , the time grows roughly as the work per processor until  $p > p_w$

- If  $Q_1(M)$  is linear with  $M$  then the right side is  $\Theta(1)$

## Example: Summation

If considering the binary tree summation where  $M_1 = n$  and  $Q_1(M_1) = M_1$ , weak scalability to  $p_w$  processors requires

$$\frac{T_{p_w}(p_w n)}{T_1(n)} = \Theta(1)$$

$$\frac{T_{p_w}(p_w n)}{T_1(n)} \approx \frac{\alpha \log(p_w) + \gamma n}{\gamma n} = 1 + (\alpha/\gamma) \log(p_w)/n$$

Therefore, the algorithm is weakly scalable up to  $p_w = \Theta(2^{n\gamma/\alpha})$ . We can conclude the following about the scalability of the binary tree algorithm with respect to  $n$ :

- it is strongly scalable to  $p_s = \Theta(n/\log(n))$  processors
- it is weakly scalable to  $p_w = \Theta(2^n)$  processors

## 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
- Since  $T_p \geq Q_1/p$ ,  $Q_1/p$  must be constant or decreasing
- If  $Q_1$  grows faster than linearly with input size  $M_1$ , then  $M_1$  must grow sublinearly with  $p$  to maintain constant  $T_p$
- To achieve perfect execution time scalability, all cost components ( $L, W, F$ ) of the algorithm must stay constant when  $Q_p$  and  $p$  grow by the same factor
- Easier to achieve than strong scaling, but harder than weak scaling, where  $Q_p$  can increase as  $p$  and  $M_1$  grow

# Fixed Accuracy

- For some problems, desired accuracy of solution determines amount of memory and work required
- It is pointless to increase input size beyond that necessary to achieve desired accuracy
- Choice of resolution can affect serial work  $Q_1$  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
- An alternative approach is to use efficiency itself as scaling invariant, i.e., we determine minimum growth rate in work required to maintain *constant* efficiency
- If this is possible, then algorithm is scalable, but it may still be impractical if required growth rate in work is excessive, leading to unacceptably large execution time
- Thus, resulting growth rate in work determines *degree* to which algorithm is scalable

# Isoefficiency Function

*Isoefficiency function*  $\tilde{Q}(p)$  is the amount of work required to maintain given constant efficiency  $E_p$

- The scaling with input size associated with the isoefficiency function,  $\tilde{M}(p)$  is defined by solving for  $M_1$  in  $Q_1(M_1) = \tilde{Q}(p)$ , i.e.,  $\tilde{M}(p) = Q_1^{-1}(\tilde{Q}(p))$
- So more precisely, we want to find  $\tilde{Q}(p) = Q_1(\tilde{M}(p))$  so

$$E_p(\tilde{M}(p)) = \text{const.} \quad \text{for increasing } p$$

- In practice we are only concerned with the asymptotic scaling of  $\tilde{Q}(p)$



## Example: Isoefficiency

To get the isoefficiency function for the binary tree sum:

- 1 Find  $\tilde{M}(p)$  so  $E_p(\tilde{M}(p)) = \Theta(1)$ , which for the binary tree is

$$E_p(\tilde{M}(p)) \approx \frac{1}{1 + (\alpha/\gamma)(p/\tilde{M}(p)) \log p} = \Theta(1)$$

$$(\alpha/\gamma)(p/\tilde{M}(p)) \log p = \Theta(1)$$

$$\tilde{M}(p) = \Theta((\alpha/\gamma)p \log(p))$$

- 2 Determine  $\tilde{Q}(p) = Q_1(\tilde{M}(p))$ , which for the binary tree is just  $\tilde{Q}(p) = \tilde{M}(p)$

So, for the binary tree, constant efficiency is maintained so long as the work scales as  $Q_1 = n = \Theta(p \log(p))$ .

However, in this scaling mode, the time  $T_p$  and memory footprint per processor  $\tilde{M}(p)/p$  grow with  $\log p$

# Isoefficiency and Scalability

- If we scale with constant efficiency,  $T_p = \Theta(\tilde{Q}(p)/p)$  stays constant if isoefficiency function is  $\tilde{Q}(p) = \Theta(p)$ , but otherwise  $T_p$  grows with  $p$
- Growth rate of  $T_p$  or  $\tilde{M}(p)/p$  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  $p$

## Example: Atmospheric Flow Model

Lets now analyze a simplified version of the previously mentioned iterative method for the atmospheric flow model

- 3-D  $n_x \times n_y \times n_z$  grid with  $n_z \ll n_x, n_y$
- 5-point stencil on  $x, y$  (horizontal) planes
- implicit solves along  $z$  (vertical) fibers

Assuming we can solve for each  $z$ -fiber with  $\Theta(n_z)$  work,

- sequential work is  $Q_1 = \Theta(n_x n_y n_z)$  per iteration
- depth  $D = \Theta(n_z)$  per iteration assuming each implicit solve is nonparallelizable

# 1-D Agglomeration Strategy

- *Partition*: assign one grid point per fine-grain task
- *Communicate*: near-neighbor communication for 5-point horizontal stencil, all-to-all vertical communication for vertical solve
- *Agglomerate*: First, consider 1-D agglomeration along one horizontal dimension of 3-D grid, with subgrid of size  $n_x \times (n_y/p) \times n_z$  assigned to each coarse-grain task

## Cost Analysis: 3-D Grid, 1-D Agglomeration

We would like to find the costs  $(L, W, F)$  that will model the execution time as  $T \approx \alpha L + \beta W + \gamma F$

- Since the parallel algorithm subdivides the mesh in a *load balanced* way and works in a *fully concurrent* manner,

$$F = Q_p/p = Q_1/p = \Theta(n_x n_y n_z / p)$$

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

$$W = 2n_x n_z \quad \text{and} \quad L = 2$$

- Thus

$$T_p = \alpha 2 + \beta 2n_x n_z + \Theta(\gamma n_x n_y n_z / p) = \alpha 2 + \beta 2n_x n_z + T_1/p$$

# Efficiency Analysis: 3-D Grid, 1-D Agglomeration

*Efficiency:*

$$\begin{aligned} E_p &= \frac{S_p}{p} = \frac{T_1}{pT_p} = \frac{T_1}{p(\alpha 2 + \beta 2n_x n_z + T_1/p)} \\ &= \frac{1}{1 + \alpha 2p/T_1 + \beta 2n_x n_z p/T_1} = \frac{1}{1 + \frac{\alpha}{\gamma} \frac{2p}{n_x n_y n_z} + \frac{\beta}{\gamma} \frac{2p}{n_y}} \end{aligned}$$

*Strong Scaling:*

- 1-D agglomeration is strongly scalable ( $E_{p_s} = \Theta(1)$ ) to

$$p_s = \Theta(\min[(\gamma/\alpha)n_x n_y n_z, (\gamma/\beta)n_y])$$

processors, for a given machine configuration  $p_s = \Theta(n_y)$

- Amdahl's law gives us a lower bound,

$S_{p_s} \leq Q_1/D = \Theta(n_x n_y n_z / n_z) = \Theta(n_x n_y)$ , so we observe that 1-D agglomeration may not be optimal

## Weak Scalability: 3-D Grid, 1-D Agglomeration

We have  $E_p(n_x, n_y, n_z) = 1 / (1 + \frac{\alpha}{\gamma} \frac{2p}{n_x n_y n_z} + \frac{\beta}{\gamma} \frac{2p}{n_y})$

*Weak Scaling:*

- To reason about weak scaling, we need a notion of *increasing* input size for this problem
  - can increase  $n_x, n_y, n_z$  proportionally
  - can increase  $n_x, n_y$  while keeping  $n_z$  constant
- Assuming the latter, the weak scalability is characterized by constant

$$E_{p_w}(p_w^{1/2} n_x, p_w^{1/2} n_y, n_z) = 1 / \left( 1 + \frac{\alpha}{\gamma} \frac{2}{n_x n_y n_z} + \frac{\beta}{\gamma} \frac{2\sqrt{p_w}}{n_y} \right)$$

- As  $p_w$  grows, the last term in the denominator grows, so 1-D agglomeration is weakly scalable to

$$p_w = \Theta(((\gamma/\beta)n_y)^2) \quad \text{processors}$$

## Isoefficiency: 3-D Grid, 1-D Agglomeration

Isoefficiency gives a relative growth rate  $\tilde{n}(p) = n_x(p) = n_y(p)$  needed to maintain constant efficiency, i.e.,

$$E_p(\tilde{n}(p), \tilde{n}(p), n_z) = 1 / \left( 1 + \frac{\alpha}{\gamma} \frac{2p}{\tilde{n}(p)^2 n_z} + \frac{\beta}{\gamma} \frac{2p}{\tilde{n}(p)} \right) = \Theta(1)$$

The last term in the denominator implies we need  $\tilde{n}(p) = \Theta(p)$

- The isoefficiency function is then  $\tilde{Q}(p) = \Theta(p^2)$
- Memory footprint grows in the same fashion  $\tilde{M}(p) = \Theta(p^2)$
- Further, we would have  $T_p = \Theta(pT_1)$
- Both the memory footprint per processor and the execution time must grow linearly with the number of processors to maintain constant efficiency



## Cost Analysis: 3-D Grid, 2-D Agglomeration

- Next consider 2-D agglomeration along both horizontal dimensions of 3-D grid, with subgrid of size  $(n_x/\sqrt{p}) \times (n_y/\sqrt{p}) \times n_z$  assigned to each coarse-grain task
- For simplicity, we assume  $n_x = n_y = n$ , which is consistent with the scaling of input size of interest
- Each task exchanges a total of  $2n_x n_z/\sqrt{p} + 2n_y n_z/\sqrt{p} = 4nn_z/\sqrt{p}$  points with its four neighbors, so

$$T_p = \alpha 4 + \beta 4nn_z/\sqrt{p} + \gamma n^2 n_z/p$$

## Efficiency Analysis: 3-D Grid, 2-D Agglomeration

2-D agglomeration gives  $E_p(n, n_z) = 1 / \left( 1 + \frac{\alpha}{\gamma} \frac{4p}{n^2 n_z} + \frac{\beta}{\gamma} \frac{4\sqrt{p}}{n} \right)$

- Setting  $E_{p_s}(n, n_z) = \Theta(1)$  shows strong scalability to

$$p_s = \Theta(\min[(\gamma/\alpha)n^2 n_z, (\gamma/\beta)^2 n^2]) \quad \text{processors}$$

- Meaning 2-D agglomeration will strong scale until each processor owns a constant-sized subgrid of vertical fibers (where the constant depends on relative values of  $\alpha, \beta, \gamma$ )
- Observing  $E_p(n\sqrt{p}, n_z) = \Theta(1)$  for any  $p$ , shows the algorithm is weakly scalable to an arbitrary number of processors!
- Since efficiency is maintained unconditionally when work increases at the same rate as the number of processors, the isoefficiency function is  $\tilde{Q}(p) = \Theta(p)$

## Network Topology Mapping: 3-D Grid

We consider mapping 1-D and 2-D agglomeration onto ideal choices of mesh networks

- 1-D mesh, 1-D agglomeration
  - For 1-D agglomeration, we can map blocks of agglomerated tasks onto each processor
  - Only neighboring processors communicate, so there is no network contention
  - Any network that can embed a 1-D mesh is as good
- 2-D mesh, 2-D agglomeration
  - For 2-D agglomeration, we can map 2-D blocks of agglomerated tasks onto each processor
  - Again only neighboring processors communicate, so there is no network contention
  - Any network that can embed a 2-D mesh is as good

# Network Topology Mapping with Contention

The effect of network contention is evident when trying to map 2-D agglomeration onto a 1-D mesh

- 1 Map block-columns of agglomerated tasks to each processor, effectively yielding 1-D agglomeration, and avoiding network contention
- 2 Map a 2-D block of agglomerated tasks to each processor
  - One dimension can be mapped continuously, preserving near-neighbor communication
  - The other dimension would correspond to communication between processors  $\sqrt{p}$  hops away from each other, yielding  $\Theta(\sqrt{p})$  slow-down due to network contention
  - Our execution time then becomes

$$T_p \approx \alpha 2\sqrt{p} + \beta 2nn_z + \gamma n^2 n_z / p$$

same bandwidth cost as 1-D agglomeration, but more msgs

## 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