CS 554 / CSE 512: Parallel Numerical Algorithms
Lecture Notes
Chapter 2: Parallel Thinking

Michael T. Heath and Edgar Solomonik
Department of Computer Science
University of Illinois at Urbana-Champaign

November 10, 2017

1 Design

A common partition/communicate/agglomerate/map methodology is effective for parallelization of many numerical and data-intensive algorithms [17]. We develop this methodology in general and provide a few typical examples. A model of the cost and execution time of these algorithms is then derived using the communication and execution models from the previous chapter. These models provide a lens for analyzing the quality of the parallelizations in terms of efficiency, a notion we develop rigorously in the latter part of this chapter. Note, however, that the design methodology introduced in this section provides a method of parallelizing algorithms. However, for a given *problem* some computationally efficient algorithms may be **inherently sequential** or simply less parallelizable than alternatives. In this sense, parallelization is only a part of the overall design process of finding the best parallel numerical algorithm for a given problem [3].

1.1 Parallelization Methodology

A natural way to design parallel algorithms from a given sequential computation/algorithm is to subdivide the computation into minimal chunks (**tasks**), then consider dependencies among them. We can then consider assigning each chunk to a unique processor, treating the dependencies as **communication channels**. This way of thinking stems from the PRAM model, which aims to exploit the maximum amount of concurrency available in the algorithm, often resulting in the assignment of each processor to very fine-grained parts of the computation. However, in general, the number of processors may be much smaller than the number of computational operations. The mapping of the operations onto processors and (to a lesser extent) the mapping of communication channels onto the network affects the efficiency and scalability of parallel algorithms.

We first define the decomposition more precisely, using the following notions:

- **Task** – a subset of the overall algorithm, with a set of inputs and outputs,
- **Communication channel** – connection between two tasks over which information is passed (messages are sent and received).

In some cases, tasks will correspond to a set of computations that is executed atomically (once all inputs are received, the task is processed and computes all outputs). However, in regular iterative algorithms, it can be convenient to think of tasks as being persistent (receiving inputs and computing outputs multiple times),

so that the parallelization makes repeated use of the same communication channels. Rather than map fine-grained tasks to processors, we first aggregate them into coarser-grain tasks, so as to consider locality, task dependencies, and communication channels at a more abstract level. Finally, we map these **agglomerated tasks** onto processors, while trying to enable as much **concurrency** of execution as possible among the processors and minimizing the communication between them.

The overall parallel algorithm design methodology has four steps (demonstrated in Figure 1):

- **Partition** – decompose problem into fine-grain tasks, maximizing number of tasks that can execute concurrently,
- **Communicate** – determine communication pattern among fine-grain tasks, yielding **dependency graph** with fine-grain tasks as nodes and communication channels as edges,
- **Agglomerate** – combine groups of fine-grain tasks to form fewer but larger coarse-grain tasks, thereby reducing communication requirements,
- **Map** – assign coarse-grain tasks to processors, subject to tradeoffs between communication costs and concurrency.

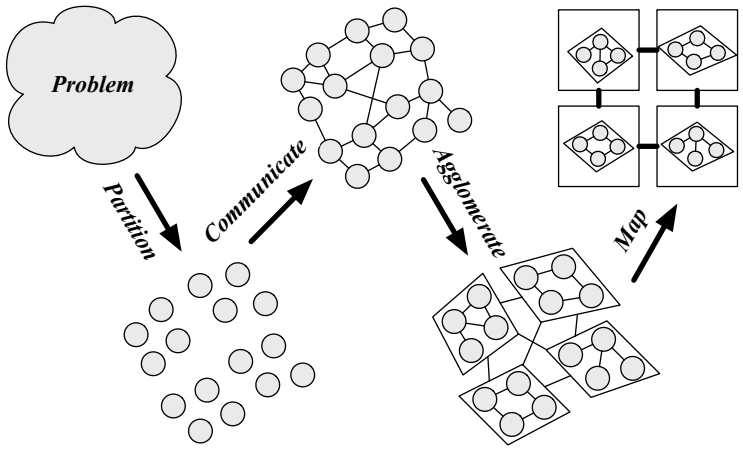


Figure 1: Illustration of the proposed parallel algorithm design methodology.

Our design methodology can be interpreted in terms of graph embeddings:

- the partitioning step defines the nodes of the task graph,
- the communicate step defines the edges of the task graph,
- the agglomeration step coarsens the task graph, embedding it in a smaller virtual network graph,
- the map step embeds the coarser task (virtual network) graph onto the physical network graph.

When ignoring network topology, we can think of the map step as embedding the coarse grain task graph onto a fully-connected graph. Some system-level abstractions (e.g., the Charm++ programming framework [15]) automate the map step, requiring the user only to define the coarse grain tasks and their data interchanges.

1.1.1 Partitioning

The partitioning step most commonly involes a data-driven decomposition of the algorithm, although there are other important sources of concurrent tasks. The following are common partitioning strategies:

- **domain partitioning** – subdivide geometric domain into subdomains [6, 25],
- **functional decomposition** – subdivide algorithm into multiple logical components [7, 8],
- **independent tasks** – subdivide computation into tasks that do not depend on each other, i.e., partition **embarrassingly parallel** parts of the algorithm [20],
- **array parallelism** – subdivide data stored in vectors, matrices, or other arrays [1, 2, 12, 13, 21, 23],
- **divide-and-conquer** – subdivide problem recursively into tree-like hierarchy of subproblems [4],
- **pipelining** – subdivide sequences of tasks performed by the algorithm on each piece of data [11, 27, 30].

With all or a mix of these partitioning strategies, a general goal is to maximize the potential for concurrent execution and to maintain load balance, i.e., ensure the tasks are roughly uniform in size. Generally, it is also desirable that the tasks remain of constant size as we increase the problem size. For instance, if we can partition a cube $n \times n \times n$ domain with three strategies (displayed in Figure 2 with $k = n/2$)

- define tasks corresponding to partitions of size $n \times n \times k$,
- define tasks corresponding to partitions of size $n \times k \times k$,
- define tasks corresponding to partitions of size $k \times k \times k$.

If we consider larger cubes (increase the problem size by increasing n), the tasks defined by the first two partitioning strategies grow in size. However, the last strategy maintains a fixed task size for a fixed k. While the number of such fine-grained tasks may grow with the problem size, we can always agglomerate them as appropriate, and the third strategy gives us the most flexibility in doing so.

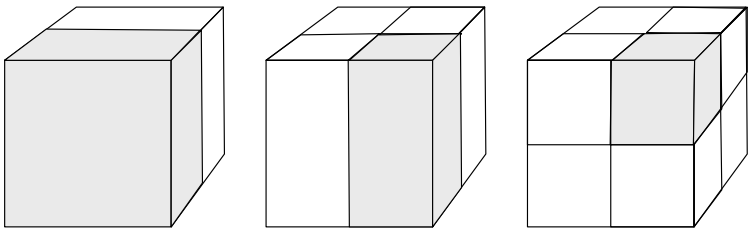


Figure 2: Three partitioning strategies for a computation with a cubic geometric structure.

1.1.2 Agglomeration

Agglomeration is generally done with account for the specifics of the communication pattern defined by the fine-grained task graph. In particular, agglomeration should leverage **locality** in the communication pattern. Generally, the communication is local if, for some geometric arrangement of tasks, only nearby tasks establish communication channels among each other.

Agglomeration is easiest to do statically if the communication pattern is conveniently structured, but in some cases it can be unstructured (although potentially still local) or dynamically changing, rather than persistent. In these cases, the agglomeration may need to be done dynamically rather than statically. However, in many scenarios dynamic agglomeration may require partitioning a very large graph, in which cases static, albeit possibly imperfect agglomeration can be preferrable.

1.1.3 Mapping

As with agglomeration, mapping of coarse-grain tasks to processors should maximize concurrency, minimize communication, and maintain good load balance. Often different mappings involve trade-offs between these goals. Taking into account the interconnect topology makes mapping substantially more challenging. While the connectivity of coarse-grain task graph is inherited from that of fine-grain task graph, whereas connectivity of target interconnection network is independent of problem. Communication channels between tasks may or may not correspond to physical connections in underlying interconnection network between processors.

As a simple demonstration of the trade-offs involved in mapping agglomerated tasks onto processors, consider two persistent tasks, that interchange information and executed chunks of work independently. These two communicating tasks can be assigned to

- one processor, avoiding interprocessor communication but sacrificing concurrency,
- two adjacent processors, so communication between the tasks is directly supported, or
- two nonadjacent processors, so message routing is required.

When taking into account the interconnecition netwrok (distinguishing the latter two choices), making these choices optimally for an arbitrary number of processors in NP-complete [22], as subgraph isomorphism is a special case of the problem [5]. However, much like we saw with agglomeration, for many problems, the task graph has regular structure that can make static mappings possible and efficient. If communication is mainly global, then communication performance may not be sensitive to placement of tasks on processors, which can make random mappings attractive. In particular, if the communication pattern is very irregular and nonuniform, randomization serves to balance the communication traffic experienced by any processor or network link, avoiding **network contention** [26].

A few static mapping strategies are particularly prevalent in parallel numerical algorithms. Consider n tasks and p processors consecutively numbered in some ordering. The following regular mappings types often serve to exploit locality (reduce communication) while maintaining concurrency (see also Figure 3):

- **block mapping** – blocks of n/p consecutive tasks are assigned to successive processors,
- **cyclic mapping** – task i is assigned to processor $i \mod p$,
- **reflection mapping** – like cyclic mapping except tasks are assigned in reverse order on alternate passes,
- **random** – tasks are permuted at random then assigned in a block mapping, so that they are partitioned evenly among processors (n/p each).

By combination of agglomeration and mapping we can derive combinations of these mapping strategies,

- **block-cyclic mapping** – blocks of tasks assigned to processors cyclically.
- **block-reflection mapping** – blocks of tasks assigned in reverse orders in different blocks.

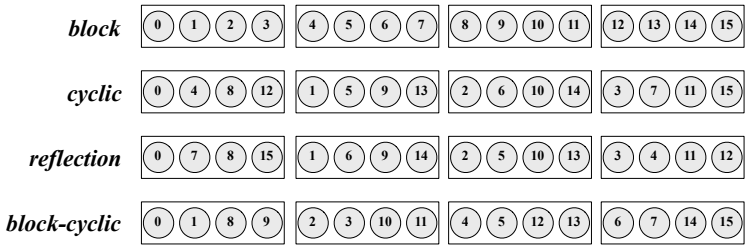
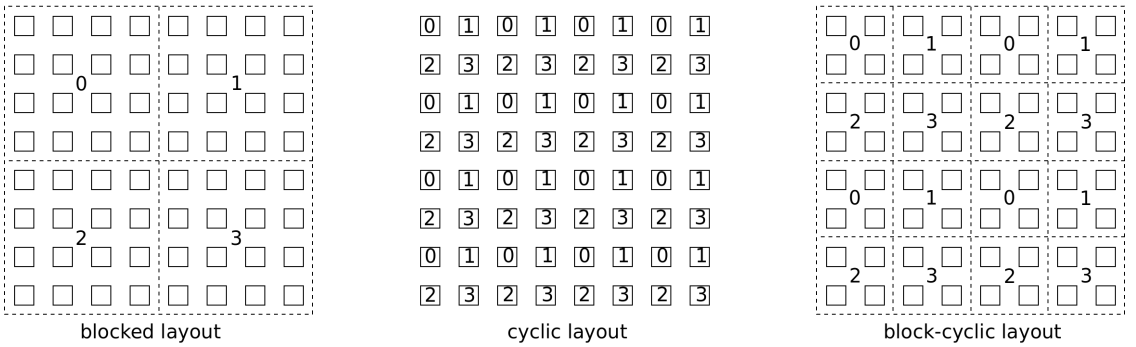


Figure 3: Illustration of a few common static mapping strategies.

Block-cyclic mappings will be of interest as they can present the distribution with a control mechanism (block-size adjustment) between locality and load-balance.

These mappings generalize naturally to multidimensional grids of tasks, when mapping to multidimensional mesh of processors. In particular, Figure 4 demonstrate the cyclic, blocked, and block-cyclic mappings of a matrix onto a 2-by-2 mesh of processors. Cyclicity (cyclic and block-cyclic mappings) will be used in parallel matrix-based algorithms to achieve better load balance and concurrency within algorithms that operate on successive panels of the matrix [2, 24].



(a) Blocked Matrix Mapping (b) Cyclic Matrix Mapping (c) Block-Cyclic Matrix Mapping

Figure 4: Illustration of a few common static mapping strategies in 2-dimensions.

The mapping strategies above are used **statically** to determine data distributions that are typically maintained for a fixed portion or entirety of a computation. In contrast to these, **dynamic** mapping strategies, are typically associated with load balancing the assignment of tasks to processors periodically. Dynamic mapping (load-balancing) is often benficial when the load of each tasks is variable throughout the computation. However, rebalancing tasks is typically associated with a reditribution of data that can outweigh the benefit of the improvement in the subsequent load balance. Dynamic load balancing strategies that exploit locality, such as hierarchical load balancers, can limit the cost of computation [32]. In some cases, measurement-driven dynamic strategies decide between global rebalancing methods and minimal perturbative rebalancing methods [33].

2 Cost Analysis

The **costs** of a parallel algorithm correspond to a quantification of its complexity of usage of different hardware resources. To understand the scalability and model execution time of different parallel algorithms

we need to quantify the degree to how well the parallel algorithms fare in

- **load balance** – evenness of distribution of overall work among processors,
- **concurrency** – ability of processors to perform work simultaneously,
- **overhead** – additional work and communication [16].

Figure 5 provides visual examples of how these effects influence execution time.

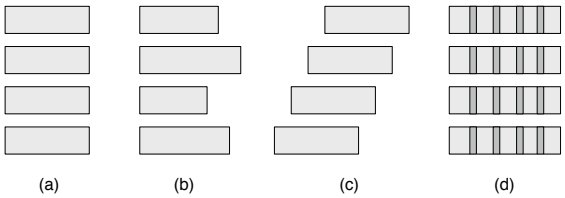


Figure 5: Assignments of tasks to 4 processors demonstrating (a) perfect load balance and concurrency, (b) good initial concurrency but poor load balance, (c) good load balance but poor concurrency, (d) good load balance and concurrency but additional overhead.

For distributed-memory algorithms a critical limiting feature is their memory footprint. We generally associate this quantity with the input/output size for sequential exeucution, while being potentially greater within a parallel algorithm.

- **input/output size** – M_1 – minimal memory footprint of the problem in words (typically maximum of input and output size),
- **memory footprint** – M_p – overall memory footprint of parallel algorithm on p processors in words.

We focus on the following To reason about computational cost, load balance, and concurrency, we quantify the total number of operations and the longest inherently sequential subsection:

- **sequential work** – $Q_1(M_1)$ – number of operations performed by reference sequential algorithm,
- **work** – $Q_p(M_1)$ – overall number of operations done by of parallel algorithm on p processors,
- **depth** – $D(M_1)$ – longest sequence of dependent computational operations in algorithms.

These are expressed as functions of M_1 as we are most interested in their dependence on the input/output size of the algorithm (M_p could also be parameterized as $M_p(M_1)$).

Work and depth measures are insufficient for modelling the execution time of a parallel algorithm. To do so, we extend the machinery introduced in the previous chapter. The **parallel execution time** is the number of seconds it takes a parallel algorithm to complete on our model distributed-memory machine representation. We will obtain lower and upper bounds on these times in terms of the parameters:

- α – time to transfer a 0-byte message,
- β – bandwidth cost (per-word),
- γ – time to perform one local operation (unit work).

To bound the execution time of an algorithm we seek to determine a cost tuple

$$\begin{aligned} \text{Cost} &= \big(S, W, F\big), \quad \text{where} \\ S &= \text{number of messages communicated}, \\ W &= \text{number of words communicated}, \\ F &= \text{amount of work done}, \end{aligned}$$

so that the execution time satisfies

$$T_p = \Theta(\alpha S + \beta W + \gamma F) \,.$$

The sequential execution time is the time it takes to execute the sequential computational operations,

$$T_1(M_1) = \gamma Q_1(M_1).$$

The parallel execution time satisfies

$$T_p(M_1) \quad \geq \quad \gamma Q_p(M_1)/p \quad \geq \quad \gamma Q_1(M_1)/p = T_1(M_1)/p.$$

The **parallel speed-up** is given accordingly by

$$S_p(M_1) = T_1(M_1)/T_p(M_1) \,,$$

in general $S_p \leq p$. An ideal speed-ups are achievable only by full concurrencywith no communication overheads. We can bound this speed-up by considering what part of the algorithm is executed sequentially.

Amdahl's law – if $1/s$ of the computation is done sequentially, the achievable speed-up is at most s.

Amdahl's law [10] bounds the speed-up relative to the size of the most expensive unparallelized section of code. We can derive an optimality criterion using Amdahl's law, by considering the part of the algorithm that is inherently sequential i.e., cannot be parallelized. Recall that the depth (D) of an algorithm is the longest chain of dependent operations in the algorithm, which is exactly the most expensive inherently sequential portion of the algorithm. Amdahl's law implies that [31]

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

or in words,

$$\text{speedup} \quad \leq \quad \text{work} \quad / \quad \text{depth} \,.$$

3 Scaling Efficiency

The **parallel efficiency** of an algorithm is its effectiveness relative to the performance of its serial counterpart. More precisely, it is the fraction of maximal speed-up attained [16],

$$E_p(M_1) \quad = \quad \text{speedup} \quad / \quad \text{number of processors} \quad = \quad S_p(M_1)/p \,.$$

The **scalability** of an algorithm is the relative effectiveness with which parallel algorithm can utilize additional processors. There are many ways to define relative scalability metrics, and generally the most

important metric is application-dependent. We analyze various metrics including the two key modes for application scaling to more processors. The most basic mode target lowering the time to solution.

Strong scaling – execution of a parallel algorithm with an increasing number of processors with constant overall input/output sizes.

Other scaling modes associate an increase of processors with the solution of a larger or more refined problem. These scaling modes assume that there are a family of problems of increasing input/output size or complexity. We distinguish the notion of scaling with constant input/output size per processor as weak scaling, because this mode is fundamental to the scaling of the *problem* rather than the *algorithm*. We note that in literature weak scaling is often defined as parallel scaling with constant work per processor.

Weak scaling – execution of a parallel algorithm with an increasing number of processors with constant input/output size per processor.

Weak scaling to larger problems that require more memory M_1 and work Q_1 is often desirable, e.g., for

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

Accuracy-based scaling modes may make sense in specific applications, but often mirror similar considerations to the weak scaling regime.

Perfect scaling, $E_p(M_1) = 1$ for p up to Q_1 is possible only for **embarassingly parallel** algorithms, where all operations are decoupled from one another. In general, we are interested in the *degree* to which an algorithm is scalable. When considering strong and weak scaling modes, we will quantify this degree as the number of processors to which the algorithm will scale with respect to a base input/output size, before it begins to see a drop-off in efficiency. In other words, we will bound the number of processors to which the algorithm will scale with nearly linear speed-up.

3.1 Strong Scaling Efficiency

When the problem is not embarrassingly parallel, we are most often interested in obtaining the maximal possible acceleration in time to solution, and not doing so wastefully with respect to computational resources. Efficiency provides us with a measure of the extent to which resources are being effectively used by the parallel algorithm.

Strong scalabity to p_s processors – attaining speed-up of $S_{p_s} = \Theta(p_s)$ when using p_s processors for sufficiently large input/output size.

In this sense, strong scalability defines the number of processors to which the algorithm will scale, before seeing a drop-off in efficiency (e.g., exhaust concurrency or become dominated in cost by communication overhead). The above notion is equivalent to saying that an algorithm is

strongly scalable to p_s processors if $E_{p_s} = \Theta(1)$.

In effect we seek to asymptotically characterize the function relating the maximum number of processors we can use efficiently to the amount of work required for the problem, $p_s(Q_1)$, such that $E_{p_s(Q_1)}(Q_1) \geq 1/C$ for any Q_1 and some constant C.

The maximal speed-up achievable for an algorithm is at least proportional to the number of processors to which the algorithm can strong scale, $\max_p S_p = \Omega(p_s)$. This relationship implies the strong scalability limit,

$$p_s = O(\max_p S_p) = O(Q_1/D).$$

On the other hand, S_p can be asymptotically greater than p_s, but if this is the case, than it must occur on a number of processors p, where efficiency is deteriorating asymptotically, so $E_p \ll 1$.

The strong scalability properties are deductible from the execution time model of an algorithm. For example, consider the strong scalability of computing a summation of n numbers using a binary reduction tree. The algorithm consists of two stages, displayed in Figure 6,

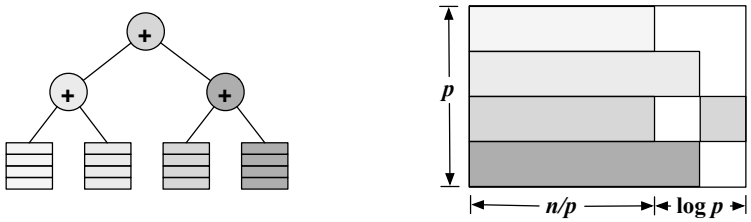


Figure 6: Execution stages in a parallel binary tree summation.

- each of p processors sums n/p elements sequentially, taking time $\gamma n/p$,
- the partial sums are passed up the binary tree, taking time $\Theta((\alpha + \beta + \gamma)\log p) = \Theta(\alpha \log p)$.

The execution time of this binary tree summation algorithm is

$$T_p(n) = \Theta(\alpha \log p + \gamma n/p).$$

Its efficiency is

$$E_p(n) = T_1(n)/(pT_p(n)) = \Theta\Big(1/\Big(1 + (\alpha/\gamma)p\log(p)/n\Big)\Big).$$

To determine the strong scalability limit, we seek $f(n)$ so that so long as $p_s = \Theta(f(n))$, $E_{p_s} = \Theta(1)$. The efficiency function $E_p(n)$ stays over $1/2$ so long as $(\alpha/\gamma)\log(p)/n \leq 1$. Once p is sufficiently high for this critical point to occur, which defines p_s, then for larger p the efficiency deteriorates as $\Theta(1/\log(p/p_s))$. This means that the algorithm is strongly scalable to p_s processors, where

$$p_s \log(p_s) = \Theta((\gamma/\alpha)n).$$

We can then use the fact that

$$x \log x = y \quad \Rightarrow \quad x = y/\log x = y/\log(y/\log(x)) = y/(\log(y) - \log\log(x)) \approx y/\log(y),$$

with $x = p_s$ and $y = (\gamma/\alpha)n$, to conclude that the asymptotic strong scalability limit is

$$p_s = \Theta((\gamma/\alpha)n/\log((\gamma/\alpha)n).$$

The dependence on the problem size for fixed architectural parameters γ and α is $p_s = \Theta(n/\log(n))$. The work-depth ratio of the problem ($Q_1 = \Theta(n)$ to $D = \Theta(\log(n))$) affirms that $Q_1/D = \Theta(n/\log(n))$ is the maximum possible speed-up.

The fraction γ/α corresponds to the ratio for the time it takes to perform a floating point scalar addition and sending a message between two processors. On current architectures the ratio is very small, $\gamma/\alpha \in [10^{-6}, 10^{-3}]$, and is presumed to decrease further, especially if γ is the aggregate floating point rate of a node. When running with p_s processors, each will perform $F_{p_s} = \Theta(n/p)$ work, so the p_s limit tells us each processor must perform at least $\Theta((\alpha/\gamma)\log((\gamma/\alpha)n))$ work, or the latency cost associated with the binary tree of messages dominates in cost.

Having identified p_s, we can express the efficiency function as

$$E_p(n) = \Theta\left(1/\left(1 + \frac{p\log(p)}{p_s\log(p_s)}\right)\right) = \begin{cases} \Theta(1) & : p \leq p_s, \\ O(p_s/p) & : p > p_s. \end{cases}$$

So we can observe that the efficiency quickly deteriorates when p surpasses p_s in strong scaling.

This type of scaling behavior is common and generally desirable. We can define some useful strong scalability algorithm classifications for a fixed architecture (α, β, γ).

Strong log-scalability – attaining strong scalability to $\Omega(Q_1/\log(Q_1))$ processors for an algorithm requiring Q_1 work.

In general, strongly log-scalable algorithms must have **logarithmic depth**, i.e., $D = O(\log(Q_1))$. Algorithms with constant depth $D = O(1)$ are most often nearly embrassingly-parallel, and can potentially strongly scale to $\Theta(Q_1)$ processors.

Unconditional strong scalability – attaining strong scalability to $\Theta(Q_1)$ processors for an algorithm requiring Q_1 work.

3.2 Weak Scaling Efficiency

Strong scalability is always limited by the amount of work and generally the degree of concurrency of the reference problem. Often, parallelism is needed to solve problems of larger scale efficiently. In such cases, we would like to study the parallel efficiency of the algorithm on a growing number of processors for a sequence of growing problem sizes [28]. The definition of problem size and its growth are specific to the particular problem. Throughout literature, different variants of weak scaling modes are considered, which generally increase the number of processors while keeping some algorithmic problem size parameter fixed per processor. Our focus will be on weak scaling as increasing the number of processors with a constant input/output size per processor, i.e., increasing p and M_1 so that $M_1/p = \text{const}$.

Most often, we would like to determine the number of processors to which the application can weak scale while using resources efficiently.

Weak scalabity to p_s processors – attaining speed-up of $S_{p_w}(M_0 p_w) = \Theta(p_w)$ when using p_w processors on input/output size of $M_1 = M_0 p_w$ for sufficiently large input/output size per processor M_0.

Our weak scaling condition on speed-up is the same as saying that good efficiency is maintained, i.e.,

$$E_{p_w}(M_0 p_w) = \Theta(1).$$

A constant efficiency implies that the ratio of execution time to work per processor stays constant

$$\frac{p_w T_{p_w}(p_w M_0)}{Q_1(p_w M_0)} \approx \frac{T_1(M_0)}{Q_1(M_0)}.$$

Overall quantifying the dependence of p_s, the weak scalability limit, on input/output size problem parameters, allows us to compare algorithms in relative terms as well as predict the amount of resources that can be effectively used on a distributed-memory computing platform to solve large problems.

As an example, consider the binary tree summation, for which the execution and efficiency function are

$$T_p(n) = \Theta(\alpha \log p + \gamma n/p) \quad \text{and} \quad E_p(n) = T_1(n)/(pT_p(n)) = \Theta\Big(1/\Big(1 + (\alpha/\gamma)p\log(p)/n\Big).\Big)$$

To determine the weak scalability limit, p_w, we can find the largest p for which $E_p(pn_0) = \Theta(1)$. We have

$$E_{p_w}(p_w n_0) = \Theta\Big(1/\Big(1 + (\alpha/\gamma)\log(p_w)/n_0\Big)\Big).$$

We set $(\alpha/\gamma)\log(p_w)/n_0 = \Theta(1)$, to deduce

$$p_w = \Theta(2^{(\gamma/\alpha)n_0}).$$

Thus, binary tree summation is weakly scalable to a processor count that is exponential in the number of messages that can be amortized within the local computational time $n_0\gamma$. The height of the binary tree is logarithmic in the number of processors as is the number of messages, so the weak scaling limit tells us at what point the latency cost associated with the binary tree will begin to dominate the execution time. For instance, if we have exactly $T_p(n) = 1 + (\alpha/\gamma)p\log_2(p)/n$, and select $p_w = 2^{(\gamma/\alpha)n_0}$, we can observe that

$$E_p(pn_0) = 1/\Big(1 + \log_2(p)/\log_2(p_w)\Big) = 1/\big(1 + \log_{p_w}(p)\big),$$

so the efficiency is above 50% until $p = p_w$, then begins to drop-off at a slow logarithmic rate (logarithm base is p_w). We specifically classify algorithms that achieve weak scalability in this sense.

Weak log-scalability – attaining weak scalability to $p_w = \Omega\big(2^{M_0}\big)$ processors given sufficiently large M_0 (input/output size per processor).

Algorithms that are weakly log-scalable will be a superclass of algorithms that are weakly scalable up to any number of processors.

Unconditional weak scalability – attaining weak scalability to any number of processors $p_w = \infty$.

The binary tree summation example algorithm is weakly log-scalable but not unconditionally weakly scalable. Any unconditionally strongly scalable algorithm is unconditionally weakly scalable. Similarly, any strongly log-scalable algorithm is weakly log-scalable. The converse statements do not hold generally.

3.3 Isoefficiency

Our strong and weak scaling anlyses consider efficiency for $M_1(p) = M_1$ and $M_1(p) = M_0 p$, respectively. The scalability was determined by the point at which the parallel algorithm will deteriorate in efficiency, in terms of $M_1(1)$. An alterntative approach is to use efficiency itself as scaling invariant, i.e., we determine minimum growth rate in work required to maintain *constant efficiency* [9].

Isoefficiency function – $\tilde{Q}(p)$ – gives the minimum amount of work $Q_1 = \tilde{Q}(p)$ so that

$$E_p(\tilde{Q}(p)) = \Theta(1).$$

If the amount of work grows too quickly, good efficiency will be difficult to sustain for the algorithm. We have that $\tilde{Q}(p) = \Omega(p)$ as otherwise efficiency will deteriorate due to insufficient work. Generally, we have a one-to-one relatonship between input/output size and work, i.e., functions $M_1(Q_1)$ and $Q_1(M_1)$. We can therefore, associate an **isoefficiency input/output size function** $\tilde{M}(p)$ with $\tilde{Q}(p)$, which describes the way input/output size must scale with the number of processors to maintain constant efficiency.

For the binary-tree example, we obtain $\tilde{Q}(p) = \Theta((\alpha/\gamma)p\log(p))$, which we again observe to be near-ideal scaling (now, in the isoefficiency sense, where ideal means $\tilde{Q}(p) = \Theta(p)$). Since $Q_1 = \Theta(M_1)$ for the binary tree summation, we have that $\tilde{Q}(p) = \tilde{M}(p)$. When $\tilde{Q}(p)$ is superlinear, we can deduce that the execution time must increase as $\Theta(\tilde{Q}(p)/p)$ with the number of processors p to maintain constant efficiency. When $\tilde{M}(p)$ is superlinear, we can deduce that the memory footprint per processor must increase as $\Theta(\tilde{M}(p)/p)$ to maintain constant efficiency. Having to increase the memory footprint is generally more prohibitive for resource-efficient parallel scalability.

An alternative scaling model that is important for real-time applications is constant execution-time scaling. Like in isoefficiency analysis, the model quantifies the scaling of the input/output size of the problem $\hat{M}_1(p)$, but now aiming to maintain constant execution time rather than efficiency, so $T_p(\hat{M}_1(p)) = \Theta(1)$. This scaling mode gives us a relationship between execution time, work, and processor count, which is useful for determining how to execute a given problem within a particular time-budget.

4 Example

To illustrate the design and analysis techniques introduced in this chapter, we consider a model application, a fictitious fluid dynamics atmospheric model. The evolution of the atmospheric system is described by partial differential equations on a 3-D physical domain. The domain is discretized by a $n_x \times n_y \times n_z$ mesh of points, where the vertical dimension (altitude) z, much smaller than horizontal dimensions (latitude and longitude) x and y, so $n_z \ll n_x, n_y$. Derivatives in PDEs approximated by finite differences in the x and y dimensions, but involve a linear-time implicit solve in the z dimension to model solar radiation. Simulation proceeds through successive discrete steps in time.

4.1 Algorithm

The algorithm performs finite difference approximations at each mesh point using a 5-point **stencil** rule for the Laplace operator,

$$\left(\frac{d^2}{dx^2}+\frac{d^2}{dy^2}\right)f(x,y,z) \approx \frac{1}{h^2}[f(x+h,y,z)+f(x,y+h,z)+f(x-h,y,z)+f(x,y-h,z)-4f(x,y,z)],$$

where h is the mesh spacing. With a 5-point stencil tasks on a regularly spaced mesh $\boldsymbol{x} \otimes \boldsymbol{y} \otimes \boldsymbol{z}$ the ith approximation to the derivative $\left(\frac{d^2}{dx^2}+\frac{d^2}{dy^2}\right)f(x_j,y_k,z_l)$ is computed as an average of ith approximation of the function values at neighboring mesh points,

$$G^{(i)}(x_j,y_k,z_l) = \frac{1}{h^2}\Big[F^{(i)}(x_{j+1},y_k,z_l) + F^{(i)}(x_j,y_{k+1},z_l) + F^{(i)}(x_{j-1},y_k,z_l) + F^{(i)}(x_j,y_{k-1},z_l) - 4F^{(i)}(x_j,u_k,z_l)\Big].$$

Solar radiation computations require communication throughout each vertical column of mesh points. The computed values of $G^{(i)}$ on the mesh-points should be combined with the results of the implicit solve to compute the next iterates values of $F^{(i+1)}$ on all the mesh points. We assume solar radiation is modelled by solving a fully-coupled system of equations with $O(n)$ operations. As the implicit solves require global communication along z-fibers, we do not consider parallelizing them.

4.2 Parallelization

1. *Partition*: Persistent fine-grain tasks can be associated with z-fibers of the $n_x \times n_y \times n_z$ mesh of which there are $n_x n_y$. Each fine-grain task contains mesh points that store associated with data values such as pressure and temperature.

   Our partitioning of the mesh is a domain-decomposition strategy that yields persistent tasks with local communication. Also of interest are strategies that assign separate tasks to all mesh-points or even all mesh-points at every time step [14, 18, 19, 29]. These partitioning strategies would enable more variants of agglomeration and mapping, beyond the scope of what we demonstrate in this example.

2. *Communicate*: The 5-point stencil scheme implies that the elements of a z fiber at iteration $i+1$, given by values $F^{(i+1)}(x_j, y_k, \star)$ for some x_j and y_k, are dependent on the values at the previous iteration of its four neighboring z-fibers: $F^{(i)}(x_{j-1}, y_k, \star)$, $F^{(i)}(x_j, y_{k-1}, \star)$, $F^{(i)}(x_{j+1}, y_k, \star)$, $F^{(i)}(x_j, y_{k+1}, \star)$. Our communication pattern is therefore a near-neighbor exchange on a 2-D grid of tasks.

3. *Agglomerate*:

   - 1-D agglomeration: combine tasks in the y-direction, forming n_x agglomerated tasks, each containing $n_y n_z$ mesh points (can do same in x-direction),
   - 2-D agglomeration: combine tasks into $b \times b$ horizontal blocks, yielding $(n_x/b)(n_y/b)$ agglomerated tasks, each containing $b^2 n_z$ mesh points.

   In both cases, we combine tasks into blocks (rather than selecting a cyclically or randomly distributed subset), to avoid communication in near-neighbor data exchanges.

4. *Map*: Blocked mappings minimize the surface area to volume ratio of the computation done by each processor. For 1-D agglomeration, the blocked mapping strategy assigns each processor n_x/p consecutive planes of $n_y n_z$ mesh points. For 2-D agglomeration:

   - a 1-D block mapping strategy reduces back to 1-D agglomeration,
   - a 2-D block mapping strategy assigns each processor an $((n_x/b)/\sqrt{p})$-by-$((n_y/b)/\sqrt{p})$ subgrid of agglomerated tasks, (a $(n_x/\sqrt{p})$-by-$(n_y/\sqrt{p})$-by-n_z overall submesh).

   Cyclic or random mappings would increase the surface area of the overall subvolume owned by each processor. Block-cyclic mappings could be viewed as cyclic mappings of 2-D block agglomerated tasks of size corresponding to the block dimension. These mappings would warrant consideration if there was a source of load-imbalance, e.g., the implicit solves performed to model solar effects took variable time throughout different areas.

4.3 Cost Analysis

For simplicity, we assume in the cost and efficiency analysis that $n_x = n_y$. To bound the execution time of different parallelizations of the atmospheric flow model problem, we quantify the costs (S_p, W_p, F_p) so that the execution time is $T_p \approx \alpha S_p + \beta W_p + \gamma F_p$. Both of our agglomeration strategies produce coarse-grain tasks that are load balanced and can execute completely concurrently so long as $p \leq n$ in the 1-D case and $p \leq n^2$ in the 2-D case (limits which we assume hold below). Consequently, both agglomeration strategies have a computational cost of

$$F_p = Q_p/p = Q_1/p = \Theta(n^2 n_z/p)$$

operations per time-step of the atmospheric flow simulation. Moreover, both parallel algorithms are memory efficient, $M_p = \Theta(M_1/p)$.

In the 1-D agglomeration scheme, each task exchanges $2nn_z$ grid points with each of its two neighbors, so the bandwidth cost is $W_p^{1D} = 2nn_z$ words and the latency cost if $S_p^{1D} = 2$ messages per time-step of the simulation. Consequently, the *execution time of the 1-D scheme* is

$$T_p^{1D}(n, n_z) = \alpha 2 + \beta 2nn_z + \Theta(\gamma n^2 n_z/p) = \alpha 2 + \beta 2nn_z + T_1(n, n_z)/p.$$

The overhead, $T_p^{1D} - T_1^{1D}/p$ are the communication costs associated with the near-neighbor data exchanges. Efficiency will be decided by whether most of the execution time is spent doing these **halo exchanges** as opposed to working on the local computation.

The 2-D agglomeration strategies, permits each mesh subdomain (of size $n^2 n_z/p$) to have a smaller *surface-area to volume ratio*, needing to exchange a total of only $2nn_z/\sqrt{p} + 2nn_z/\sqrt{p} = 4nn_z/\sqrt{p}$ points with its four neighbors. Consequently, the 2-D agglomeration scheme has a bandwidth cost of $W_p^{2D} = 4nn_z/\sqrt{p}$ words and $S_p^{2D} = 4$ messages per time-step of the simulation. Consequently, the *execution time of the 2-D scheme* is

$$T_p^{2D}(n, n_z) = \alpha 4 + \beta 4nn_z/\sqrt{p} + \Theta(\gamma n^2 n_z/p).$$

The bandwidth cost is reduced by a factor of $\Theta(\sqrt{p})$ by doing a higher-dimensional blocking, however, twice the number of messages is required.

4.4 Efficiency Analysis

Our cost and execution time analysis provides us with a basic performance model and terms for comparative evaluation. Efficiency analysis provides us with secondary information regarding how the overhead cost terms will affect parallel scalability. Understanding scalability quantitatively provides us with the relative and absolute metrics that are usually sought after.

4.4.1 Parallel Efficiency

We can gauge the strong and weak scalability as well as the isoefficiency from the efficiency functions of the parallel algorithms. These are directly related to our execution time models. The *efficiency function of the 1-D agglomeration scheme* is

$$\begin{aligned} E_p^{1D} &= S_p^{1D}/p = T_1/[pT_p^{1D}] \\ &= T_1/\Big[p(\alpha 2 + \beta 2n_x n_z + T_1/p)\Big] \\ &= 1/\Big[1 + \alpha 2p/T_1 + \beta 2nn_z p/T_1\Big] \\ &= 1/\Big[1 + (\alpha/\gamma)2p/(n^2 n_z) + (\beta/\gamma)2p/n\Big]. \end{aligned}$$

Similarly, *the efficiency function of the 2-D agglomeration scheme* is

$$\begin{aligned} E_p^{2D} &= S_p^{2D}/p = T_1/[pT_p^{2D}] \\ &= 1/\Big[1 + \alpha 4p/T_1 + \beta 4nn_z\sqrt{p}/T_1\Big] \\ &= 1/\Big[1 + (\alpha/\gamma)4p/(n^2 n_z) + (\beta/\gamma)4\sqrt{p}/n\Big]. \end{aligned}$$

In both efficiency, we have terms with coefficients α/γ and β/γ. These coefficients tell us the number of computational operations that can be performed, respectively, in the time it takes to send a 0-byte message and in the the time it takes to communicate a word of data. These coefficients correspond to ratios associated

with the processing rates of a given computer architecture. Sometimes, we are interested in the behavior of the efficiency only as a function of input/output size and the number of processors, i.e., in these cases, we consider α, β, and γ to be fixed constants. However, in general $\beta/\gamma \gg 1$ and $\alpha/\gamma \gg \beta/\gamma$, so carrying the dependencies on this terms will allow for different qualitative conclusions. For instance, since $W_p \geq S_p$, ignoring dependence on α and β would imply ignoring dependence on latency cost altogether.

4.4.2 Strong Scalability

Both agglomeration schemes have cost overheads that grow polynomially with the number of processors, implying that neither will be unconditionally strong scalable or strongly log-scalable. The 1-D agglomeration scheme will scale to p_s^{1D} processors, where p_s^{1D} is determined by setting

$$E_{p_s^{\text{1D}}}^{\text{1D}} = 1/\Big[1 + (\alpha/\gamma)2p_s^{\text{1D}}/(n^2 n_z) + (\beta/\gamma)2p_s^{\text{1D}}/n\Big] = \Theta(1).$$

Rearranging the above equation algebraically, we deduce that *1-D agglomeration is strongly scalable to*

$$p_s^{\text{1D}} = \Theta\Big(\min\Big[(\gamma/\alpha)n^2 n_z, (\gamma/\beta)n\Big]\Big) \quad \text{processors.}$$

As mentioned previously, the 1-D agglomeration scheme is only work-efficient so long as $p \leq n$, as it generates at most n tasks. In light of this, we can conclude that the latency cost is likely not prohibitive to strong scaling (unless $\alpha \geq \beta n n_z$). The bandwidth cost tells us that the strong scaling limit is $p_s = O((\gamma/\beta)n)$, which is a factor of γ/β smaller than the limit we can deduce from the work distribution alone. An interpretation of this 1-D strong scaling limit bound is that good efficiency is maintained so long as the number of yz-plains assigned to each processor require longer to perform local computation for than to communicate $\Theta(nn_z)$ words to another processor. The depth of the sequential algorithm is $D = \Theta(n_z)$, which suggests that the maximum speed-up could be as high as $S_\infty = \Theta(Q_1/D) = \Theta(n^2)$, suggesting the 1-D scheme could be suboptimal.

A similar analysis shows that the *2-D agglomeration is strongly scalable to*

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

For fixed (constant) γ/β, 2-D agglomeration can achieve strong scaling to $\Theta(n^2)$ processors, attaining the maximum speed-up possible for the work and depth of the sequential algorithm. However, the analysis tells us that the maximum possible speedup is in fact a factor of $(\gamma/\beta)^2$ less processors than $\Theta(n)$. With respect to the 1-D agglomeration algorithm, 2-D agglomeration generally permits strong scalability to $\Theta((\gamma/\beta)n)$ more processors. Assuming the scalability is bandwidth-limited ($\alpha \geq \beta n n_z$), we have that the 2-D agglomeration strong scalability limit p_s^{2D} is the square of the 1-D agglomeration strong scalability limit p_s^{1D}. A similar interpretation as of the 1-D strong scaling limit can be applied to the 2-D agglomeration algorithm, except now the amount of data sent decreases with the number of processors, so we are now interested in when the surface-area to volume ratio decreases so that the halo-exchange dominates the local computation.

4.4.3 Weak Scalability

To reason about weak scaling, we need to a notion of *increasing input/output size* for the atmospheric flow model problem. The context of the problem yields two particularly sensible options,

- increase n, n_z proportionally,
- increase n while keeping n_z constant.

In the first case, we grow the mesh with the same ratio of dimensions, while in the latter case we grow the two horizontal dimensions, increasing the amount of concurrency with the amount of processors. We assume that the latter scaling mode is of most interest, as it captures larger subsections of the atmosphere to the same level of refinement. The overall input/output size is $n^2 n_z$, so we assume n^2 increases proportionately to p in weak scaling the problem.

Provided the above input/output size scaling characteristics, the weak scalability is described for 1-D agglomeration by scaling from a problem of dimensions $n_0 \times n_0 \times n_z$ to a problem of size $n_0\sqrt{p_w} \times n_0\sqrt{p_w} \times n_z$, where p_w is defined so that

$$E^{\text{1D}}_{p_w}(n_0\sqrt{p_w}, n_z) = 1/\left(1 + \frac{\alpha}{\gamma}\frac{2}{n_0^2 n_z} + \frac{\beta}{\gamma}\frac{2\sqrt{p_w}}{n_0}\right) = \Theta(1).$$

We can conclude that *1-D agglomeration is weakly scalable to*

$$p_w = O((\gamma/\beta)^2 n_0^2) \quad \text{processors},$$

coincidentally the same limit as the one to which the 2-D agglomeration scheme was strongly scalable. Notably, we can observe that the weak scaling limit is independent of latency cost, as the number of messages per time-step stays constant. We can also deduce a work-based weak scaling limit of $p_w = O(n_0^2)$ for the 1-D algorithm, as it partitions work among at most $n = n_0\sqrt{p}$ processors, running out of work to assign processors if $\sqrt{p} > n_0$.

For 2-D agglomeration, we can observe that

$$E^{\text{2D}}_{p}(n_0\sqrt{p}, n_z) = 1/\left(1 + \frac{\alpha}{\gamma}\frac{4}{n_0^2 n_z} + \frac{\beta}{\gamma}\frac{4}{n_0}\right)$$

stays constant as p increases. Therefore, *2-D agglomeration achieves unconditional weak scaling* for the atmospheric flow model problem. This conclusion makes sense in light of the fact that the subdomain assigned to each processor stays the same in the weak scaling regime for 2-D agglomeration, as do the latency, bandwidth, and computational costs.

4.4.4 Isoefficiency

To determine the isoefficiency function $\tilde{Q}(p)$, we first determine a relative growth rate $\tilde{n}(p) = n_x(p) = n_y(p)$ needed to maintain constant efficiency. For the 1-D agglomeration case, this implies that we need,

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

While the equation is the same as what we had in the strong scaling analysis, we are now interested in solving for $\tilde{n}(p)$ rather than p. The last term in the denominator of the efficiency function implies that we need $\tilde{n}(p) = \Theta((\beta/\gamma)p)$. The *isoefficiency function for 1-D agglomeration* is

$$\tilde{Q}(p) = n_z \tilde{n}(p)^2 = \Theta((\beta/\gamma)^2 n_z p^2),$$

or for fixed α, β, γ, $\tilde{Q}(p) = \Theta((\beta/\gamma)^2 n_z p^2)$. This isoefficiency function implies that the work per processor must grow with the number of processors to maintain constant efficiency. Moreover, this implies the same growth in the memory-footprint, which is very prohibitive.

For 2-D agglomeration, we can observe that the efficiency function is

$$E^{\text{2D}}_{p}(\tilde{n}(p), n_z) = 1/\left[1 + (\alpha/\gamma)4p/(\tilde{n}(p)^2 n_z) + (\beta/\gamma)4\sqrt{p}/\tilde{n}(p)\right].$$

Therefore, constant efficiency is maintained so long as for a fixed α, β, γ, we have $\tilde{n}(p) = \Theta(\sqrt{p})$. Consequently, the *isoefficiency function for 2-D agglomeration* is $\tilde{Q}(p) = \Theta(p)$. In fact, the isoefficiency function is linear for any parallel algorithm that is unconditionally weakly scalable.

5 General References

General references on parallel algorithm design

- K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, 1988
- I. T. Foster, *Designing and Building Parallel Programs*, Addison-Wesley, 1995
- A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, 2nd. ed., Addison-Wesley, 2003
- T. G. Mattson and B. A. Sanders and B. L. Massingill, *Patterns for Parallel Programming*, Addison-Wesley, 2005
- M. J. Quinn, *Parallel Computing: Theory and Practice*, McGraw-Hill, 1994

General references on parallel algorithm analysis

- J. JáJá. An introduction to parallel algorithms. Vol. 17. Reading: *Addison-Wesley*, 1992.
- 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
- V. Kumar and A. Gupta, Analyzing scalability of parallel algorithms and architectures, *J. Parallel Distrib. Comput.* 22:379-391, 1994
- 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

References

[1] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzarán, D. Padua, and C. Von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–57. ACM, 2006.

[2] L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[3] G. Blelloch. Parallel thinking, 2009. Presented at Principles and Practices of Parallel Programming (PPoPP), 2009.

[4] G. E. Blelloch. NESL: A nested data-parallel language. Technical report, Pittsburgh, PA, USA, 1992.

[5] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[6] C. Farhat and M. Lesoinne. Automatic partitioning of unstructured meshes for the parallel solution of problems in computational mechanics. *International Journal for Numerical Methods in Engineering*, 36(5):745–764, 1993.

[7] M. Girkar and C. D. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):166–178, Mar 1992.

[8] M. B. Girkar. *Functional Parallelism: Theoretical Foundations and Implementation*. PhD thesis, Champaign, IL, USA, 1992. UMI Order No. GAX92-15814.

[9] A. Grama, A. Gupta, and V. Kumar. Isoefficiency function: A scalability metric for parallel algorithms and architectures. In *IEEE Parallel and Distributed Technology, Special Issue on Parallel and Distributed Systems: From Theory to Practice*. Citeseer, 1993.

[10] J. L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[11] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[12] W. D. Hillis and G. L. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[13] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS operating systems review*, volume 41, pages 59–72. ACM, 2007.

[14] H. Jia-Wei and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proceedings of the thirteenth Annual ACM Symposium on Theory of Computing*, STOC '81, pages 326–333, New York, NY, USA, 1981. ACM.

[15] L. V. Kale and S. Krishnan. CHARM++: a portable concurrent object oriented system based on C++. In *ACM Sigplan Notices*, volume 28, pages 91–108. ACM, 1993.

[16] V. Kumar and A. Gupta. Analyzing scalability of parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, 22(3):379 – 391, 1994.

[17] H. Kung. The structure of parallel algorithms. volume 19 of *Advances in Computers*, pages 65 – 112. Elsevier, 1980.

[18] C. E. Leiserson, S. Rao, and S. Toledo. Efficient out-of-core algorithms for linear relaxation using blocking covers. In *Foundations of Computer Science, 1993. Proceedings., 34th Annual Symposium on*, pages 704–713. IEEE, 1993.

[19] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick. Minimizing communication in sparse matrix solvers. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 36. ACM, 2009.

[20] W. Neiswanger, C. Wang, and E. Xing. Asymptotically exact, embarrassingly parallel MCMC. *arXiv preprint arXiv:1311.4780*, 2013.

[21] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global Arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10:169–189, 1996.

[22] M. G. Norman, S. Pelagatti, and P. Thanisch. On the complexity of scheduling with communication delay and contention. *Parallel Processing Letters*, 5(03):331–341, 1995.

[23] R. W. Numrich and J. Reid. Co-Array Fortran for parallel programming. In *ACM Sigplan Fortran Forum*, volume 17, pages 1–31. ACM, 1998.

[24] J. Poulson, B. Marker, R. A. Van de Geijn, J. R. Hammond, and N. A. Romero. Elemental: A new framework for distributed memory dense matrix computations. *ACM Transactions on Mathematical Software (TOMS)*, 39(2):13, 2013.

[25] H. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2(2):135 – 148, 1991. Parallel Methods on Large-scale Structural Analysis and Physics Applications.

[26] O. Sinnen and L. A. Sousa. Communication contention in task scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 16(6):503–515, 2005.

[27] J. Subhlok and G. Vondran. Optimal use of mixed task and data parallelism for pipelined computations. *Journal of Parallel and Distributed Computing*, 60(3):297 – 319, 2000.

[28] X.-H. Sun and J. L. Gustafson. Toward a better parallel performance metric. *Parallel Computing*, 17(10-11):1093–1109, 1991.

[29] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The Pochoir stencil compiler. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 117–128. ACM, 2011.

[30] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 356–369. IEEE Computer Society, 2007.

[31] U. Vishkin and A. Wigderson. Trade-offs between depth and width in parallel computation. *SIAM Journal on Computing*, 14(2):303–314, 1985.

[32] M. H. Willebeek-LeMair and A. P. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Transactions on parallel and distributed systems*, 4(9):979–993, 1993.

[33] G. Zheng, E. Meneses, A. Bhatele, and L. V. Kale. Hierarchical load balancing for Charm++ applications on large supercomputers. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 436–444. IEEE, 2010.