Parallel Program Design

The content of this document is taken entirely from
http://www.mcs.anl.gov/~itf/dbpp/text/node15.html#SECTION02310000000000000000
http://www.mcs.anl.gov/~itf/dbpp/text/node16.html
http://www.mcs.anl.gov/~itf/dbpp/text/node17.html
http://www.mcs.anl.gov/~itf/dbpp/text/node18.html
http://www.mcs.anl.gov/~itf/dbpp/text/node19.html

I have adjusted section and figure numbers and deleted certain parts that are too detailed or not relevant to our class. I have also reformatted the document.

---

1. Introduction

Most programming problems have several parallel solutions. The best solution may differ from that suggested by existing sequential algorithms. The design methodology that we describe is intended to foster an exploratory approach to design in which machine-independent issues such as concurrency are considered early and machine-specific aspects of design are delayed until late in the design process. This methodology structures the design process as four distinct stages: partitioning, communication, agglomeration, and mapping. (The acronym PCAM may serve as a useful reminder of this structure.) In the first two stages, we focus on concurrency and scalability and seek to discover algorithms with these qualities. In the third and fourth stages, attention shifts to locality and other performance-related issues. The four stages are illustrated in Figure 1 and can be summarized as follows:

1. *Partitioning.* The computation that is to be performed and the data operated on by this computation are decomposed into small tasks. Practical issues such as the number of processors in the target computer are ignored, and attention is focused on recognizing opportunities for parallel execution.

2. *Communication.* The communication required to coordinate task execution is determined, and appropriate communication structures and algorithms are defined.

3. *Agglomeration.* The task and communication structures defined in the first two stages of a design are evaluated with respect to performance requirements and implementation costs. If necessary, tasks are combined into larger tasks to improve performance or to reduce development costs.

4. *Mapping.* Each task is assigned to a processor in a manner that attempts to satisfy the competing goals of maximizing processor utilization and minimizing communication costs. Mapping can be specified statically or determined at runtime by load-balancing algorithms.

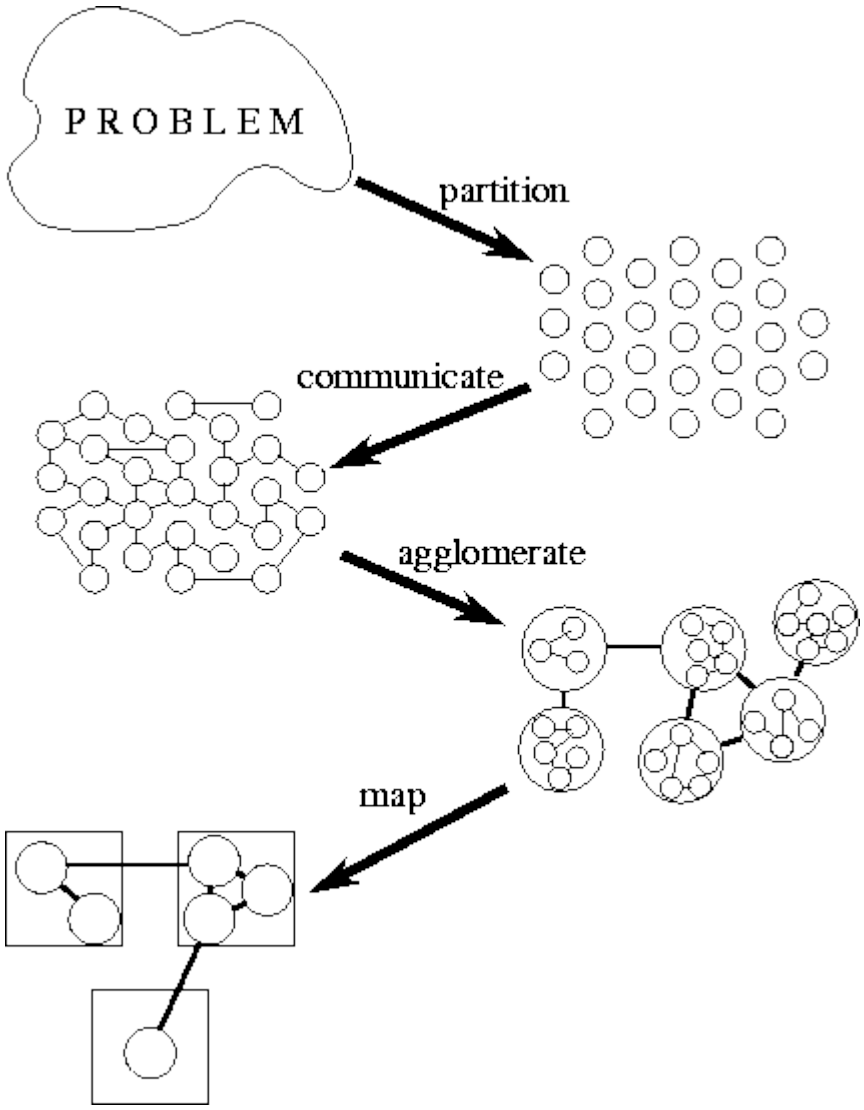

Figure 1: *PCAM: a design methodology for parallel programs. Starting with a problem specification, we develop a partition, determine communication requirements, agglomerate tasks, and finally map tasks to processors.*

The outcome of this design process can be a program that creates and destroys tasks dynamically, using load-balancing techniques to control the mapping of tasks to processors.

Algorithm design is presented here as a sequential activity. In practice, however, it is a highly parallel process, with many concerns being considered simultaneously. Also, although we seek to avoid backtracking, evaluation of a partial or complete design may require changes to design decisions made in previous steps.

2. Partitioning

The partitioning stage of a design is intended to expose opportunities for parallel execution. Hence, the focus is on defining a large number of small tasks in order to yield what is termed

a *fine-grained* decomposition of a problem. Just as fine sand is more easily poured than a pile of bricks, a fine-grained decomposition provides the greatest flexibility in terms of potential parallel algorithms. In later design stages, evaluation of communication requirements, the target architecture, or software engineering issues may lead us to forego opportunities for parallel execution identified at this stage. We then revisit the original partition and agglomerate tasks to increase their size, or granularity. However, in this first stage we wish to avoid prejudging alternative partitioning strategies.

A good partition divides into small pieces both the *computation* associated with a problem and the *data* on which this computation operates. When designing a partition, programmers most commonly first focus on the data associated with a problem, then determine an appropriate partition for the data, and finally work out how to associate computation with data. This partitioning technique is termed *domain decomposition*. The alternative approach---first decomposing the computation to be performed and then dealing with the data---is termed *functional decomposition*. These are complementary techniques which may be applied to different components of a single problem or even applied to the same problem to obtain alternative parallel algorithms.

In this first stage of a design, we seek to avoid replicating computation and data; that is, we seek to define tasks that partition both computation and data into disjoint sets. Like granularity, this is an aspect of the design that we may revisit later. It can be worthwhile replicating either computation or data if doing so allows us to reduce communication requirements.

2.1 Domain Decomposition

In the domain decomposition approach to problem partitioning, we seek first to decompose the data associated with a problem. If possible, we divide these data into small pieces of approximately equal size. Next, we partition the computation that is to be performed, typically by associating each operation with the data on which it operates. This partitioning yields a number of tasks, each comprising some data and a set of operations on that data. An operation may require data from several tasks. In this case, communication is required to move data between tasks. This requirement is addressed in the next phase of the design process.
The data that are decomposed may be the input to the program, the output computed by the program, or intermediate values maintained by the program. Different partitions may be possible, based on different data structures. Good rules of thumb are to focus first on the largest data structure or on the data structure that is accessed most frequently. Different phases of the computation may operate on different data structures or demand different decompositions for the same data structures. In this case, we treat each phase separately and then determine how the decompositions and parallel algorithms developed for each phase fit together.

Figure 2 illustrates domain decomposition in a simple problem involving a three-dimensional grid. (This grid could represent the state of the atmosphere in a weather model, or a three-dimensional space in an image-processing problem.) Computation is performed repeatedly on each grid point. Decompositions in the x , y , and/or z dimensions are possible. In the early stages of a design, we favor the most aggressive decomposition possible, which in this case defines one

task for each grid point. Each task maintains as its state the various values associated with its grid point and is responsible for the computation required to update that state.

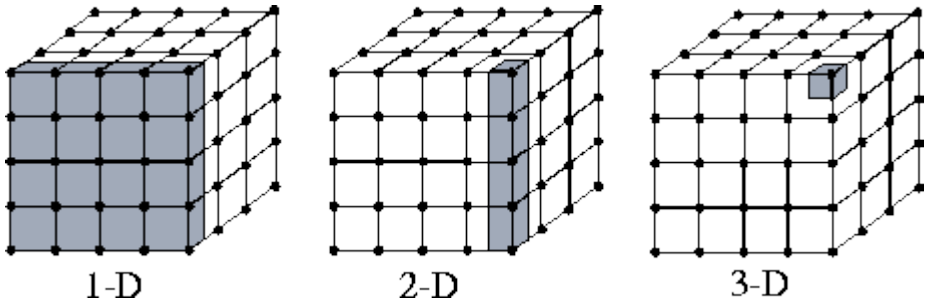

Figure 2: *Domain decompositions for a problem involving a three-dimensional grid. One-, two-, and three-dimensional decompositions are possible; in each case, data associated with a single task are shaded. A three-dimensional decomposition offers the greatest flexibility and is adopted in the early stages of a design.*

2.2 Functional Decomposition

Functional decomposition represents a different and complementary way of thinking about problems. In this approach, the initial focus is on the computation that is to be performed rather than on the data manipulated by the computation. If we are successful in dividing this computation into disjoint tasks, we proceed to examine the data requirements of these tasks. These data requirements may be disjoint, in which case the partition is complete. Alternatively, they may overlap significantly, in which case considerable communication will be required to avoid replication of data. This is often a sign that a domain decomposition approach should be considered instead.

While domain decomposition forms the foundation for most parallel algorithms, functional decomposition is valuable as a different way of thinking about problems. For this reason alone, it should be considered when exploring possible parallel algorithms. A focus on the computations that are to be performed can sometimes reveal structure in a problem, and hence opportunities for optimization, that would not be obvious from a study of data alone.

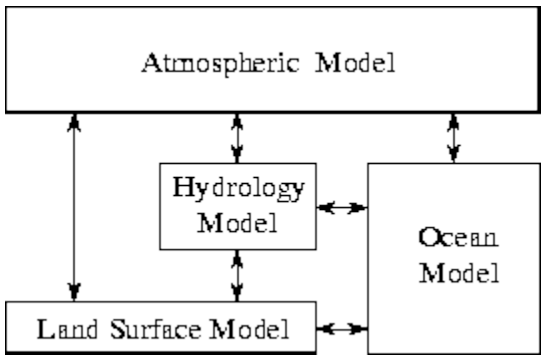

Figure 3: *Functional decomposition in a computer model of climate. Each model component can be thought of as a separate task, to be parallelized by domain decomposition. Arrows represent exchanges of data between components during computation: the atmosphere model generates*

wind velocity data that are used by the ocean model, the ocean model generates sea surface temperature data that are used by the atmosphere model, and so on.

Functional decomposition also has an important role to play as a program structuring technique. A functional decomposition that partitions not only the computation that is to be performed but also the code that performs that computation is likely to reduce the complexity of the overall design. This is often the case in computer models of complex systems, which may be structured as collections of simpler models connected via interfaces. For example, a simulation of the earth's climate may comprise components representing the atmosphere, ocean, hydrology, ice, carbon dioxide sources, and so on. While each component may be most naturally parallelized using domain decomposition techniques, the parallel algorithm as a whole is simpler if the system is first decomposed using functional decomposition techniques, even though this process does not yield a large number of tasks (Figure 3).

2.3 Partitioning Design Checklist

The partitioning phase of a design should produce one or more possible decompositions of a problem. Before proceeding to evaluate communication requirements, we use the following checklist to ensure that the design has no obvious flaws. Generally, all these questions should be answered in the affirmative.

1. Does your partition define at least an order of magnitude more tasks than there are processors in your target computer? If not, you have little flexibility in subsequent design stages.
2. Does your partition avoid redundant computation and storage requirements? If not, the resulting algorithm may not be scalable to deal with large problems.
3. Are tasks of comparable size? If not, it may be hard to allocate each processor equal amounts of work.
4. Does the number of tasks scale with problem size? Ideally, an increase in problem size should increase the number of tasks rather than the size of individual tasks. If this is not the case, your parallel algorithm may not be able to solve larger problems when more processors are available.
5. Have you identified several alternative partitions? You can maximize flexibility in subsequent design stages by considering alternatives now. Remember to investigate both domain and functional decompositions.

Answers to these questions may suggest that, despite careful thought in this and subsequent design stages, we have a “bad” design. In this situation it is risky simply to push ahead with implementation. We may also wish to revisit the problem specification. Particularly in science and engineering applications, where the problem to be solved may involve a simulation of a complex physical process, the approximations and numerical techniques used to develop the simulation can strongly influence the ease of parallel implementation. In some cases, optimal sequential and parallel solutions to the same problem may use quite different solution techniques.

3. Communication

The tasks generated by a partition are intended to execute concurrently but cannot, in general, execute independently. The computation to be performed in one task will typically require data associated with another task. Data must then be transferred between tasks so as to allow computation to proceed. This information flow is specified in the *communication* phase of a design.

Recall that in our programming model, we conceptualize a need for communication between two tasks as a channel linking the tasks, on which one task can send messages and from which the other can receive. Hence, the communication associated with an algorithm can be specified in two phases. First, we define a channel structure that links, either directly or indirectly, tasks that require data (consumers) with tasks that possess those data (producers). Second, we specify the messages that are to be sent and received on these channels. Depending on our eventual implementation technology, we may not actually create these channels when coding the algorithm. Nevertheless, thinking in terms of tasks and channels helps us to think quantitatively about locality issues and communication costs.

The definition of a channel involves an intellectual cost and the sending of a message involves a physical cost. Hence, we avoid introducing unnecessary channels and communication operations. In addition, we seek to optimize performance by distributing communication operations over many tasks and by organizing communication operations in a way that permits concurrent execution.

In domain decomposition problems, communication requirements can be difficult to determine. Recall that this strategy produces tasks by first partitioning data structures into disjoint subsets and then associating with each datum those operations that operate solely on that datum. This part of the design is usually simple. However, some operations that require data from several tasks usually remain. Communication is then required to manage the data transfer necessary for these tasks to proceed. Organizing this communication in an efficient manner can be challenging. Even simple decompositions can have complex communication structures.

In contrast, communication requirements in parallel algorithms obtained by functional decomposition are often straightforward: they correspond to the data flow between tasks. For example, in a climate model broken down by functional decomposition into atmosphere model, ocean model, and so on, the communication requirements will correspond to the interfaces between the component submodels: the atmosphere model will produce values that are used by the ocean model, and so on.

In the following discussion, we use a variety of examples to show how communication requirements are identified and how channel structures and communication operations are introduced to satisfy these requirements. For clarity in exposition, we categorize communication patterns along four loosely orthogonal axes: local/global, structured/unstructured, static/dynamic, and synchronous/asynchronous.

- In *local* communication, each task communicates with a small set of other tasks (its "neighbors'"); in contrast, *global* communication requires each task to communicate with many tasks.
- In *structured* communication, a task and its neighbors form a regular structure, such as a tree or grid; in contrast, *unstructured* communication networks may be arbitrary graphs.
- In *static* communication, the identity of communication partners does not change over time; in contrast, the identity of communication partners in *dynamic* communication structures may be determined by data computed at runtime and may be highly variable.
- In *synchronous* communication, producers and consumers execute in a coordinated fashion, with producer/consumer pairs cooperating in data transfer operations; in contrast, *asynchronous* communication may require that a consumer obtain data without the cooperation of the producer.

3.1 Local Communication

A local communication structure is obtained when an operation requires data from a small number of other tasks. It is then straightforward to define channels that link the task responsible for performing the operation (the consumer) with the tasks holding the required data (the producers) and to introduce appropriate send and receive operations in the producer and consumer tasks, respectively.

3.2 Global Communication

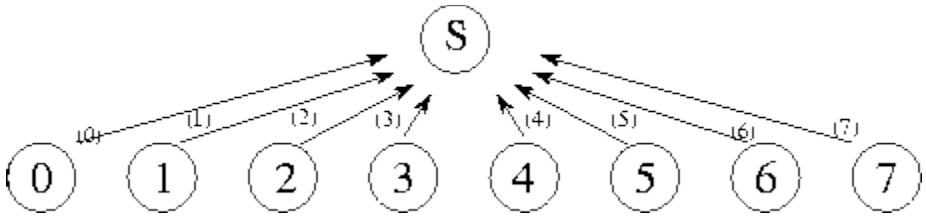

Figure 4: *A centralized summation algorithm that uses a central manager task (S) to sum N numbers distributed among N tasks. Here, N=8 , and each of the 8 channels is labeled with the number of the step in which they are used.*

A *global communication* operation is one in which many tasks must participate. When such operations are implemented, it may not be sufficient simply to identify individual producer/consumer pairs. Such an approach may result in too many communications or may restrict opportunities for concurrent execution. For example, consider the problem of performing a *parallel reduction* operation, that is, an operation that reduces *N* values distributed over *N* tasks using a commutative associative operator such as addition:

$$S = \sum_{i=0}^{N-1} X_i.$$

Let us assume that a single "manager" task requires the result *S* of this operation. Taking a purely local view of communication, we recognize that the manager requires values X_0, X_1, etc., from tasks 0, 1, etc. Hence, we could define a communication structure that allows each task to communicate its value to the manager independently. The manager would then receive the values

and add them into an accumulator (Figure 4). However, because the manager can receive and sum only one number at a time, this approach takes $O(N)$ time to sum *N* numbers---not a very good parallel algorithm!

This example illustrates two general problems that can hinder efficient parallel execution in algorithms based on a purely local view of communication:

1. The algorithm is *centralized* : it does not distribute computation and communication. A single task (in this case, the manager task) must participate in every operation.
2. The algorithm is *sequential* : it does not allow multiple computation and communication operations to proceed concurrently.

We must address both these problems to develop a good parallel algorithm.

3.3 Unstructured and Dynamic Communication

The examples considered previously are all of static, structured communication, in which a task's communication partners form a regular pattern such as a tree or a grid and do not change over time. In other cases, communication patterns may be considerably more complex. For example, in finite element methods used in engineering calculations, the computational grid may be shaped to follow an irregular object or to provide high resolution in critical regions Here, the channel structure representing the communication partners of each grid point is quite irregular and data-dependent and, furthermore, may change over time if the grid is refined as a simulation evolves.

3.4 Asynchronous Communication

The examples considered in the preceding section have all featured synchronous communication, in which both producers and consumers are aware when communication operations are required, and producers explicitly send data to consumers. In *asynchronous* communication, tasks that possess data (producers) are not able to determine when other tasks (consumers) may require data; hence, consumers must explicitly request data from producers.

3.5 Communication Design Checklist

Having devised a partition and a communication structure for our parallel algorithm, we now evaluate our design using the following design checklist. These are guidelines intended to identify nonscalable features, rather than hard and fast rules. However, we should be aware of when a design violates them and why.

1. Do all tasks perform about the same number of communication operations? Unbalanced communication requirements suggest a nonscalable construct. Revisit your design to see whether communication operations can be distributed more equitably. For example, if a frequently accessed data structure is encapsulated in a single task, consider distributing or replicating this data structure.

2. Does each task communicate only with a small number of neighbors? If each task must communicate with many other tasks, evaluate the possibility of formulating this global communication in terms of a local communication structure.
3. Are communication operations able to proceed concurrently? If not, your algorithm is likely to be inefficient and nonscalable. Try to use divide-and-conquer techniques to uncover concurrency.
4. Is the computation associated with different tasks able to proceed concurrently? If not, your algorithm is likely to be inefficient and nonscalable. Consider whether you can reorder communication and computation operations.

4. Agglomeration

In the first two stages of the design process, we partitioned the computation to be performed into a set of tasks and introduced communication to provide data required by these tasks. The resulting algorithm is still abstract in the sense that it is not specialized for efficient execution on any particular parallel computer. In fact, it may be highly inefficient if, for example, it creates many more tasks than there are processors on the target computer and this computer is not designed for efficient execution of small tasks.

In the third stage, *agglomeration*, we move from the abstract toward the concrete. We revisit decisions made in the partitioning and communication phases with a view to obtaining an algorithm that will execute efficiently on some class of parallel computer. In particular, we consider whether it is useful to combine, or *agglomerate*, tasks identified by the partitioning phase, so as to provide a smaller number of tasks, each of greater size (Figure 5). We also determine whether it is worthwhile to *replicate* data and/or computation.

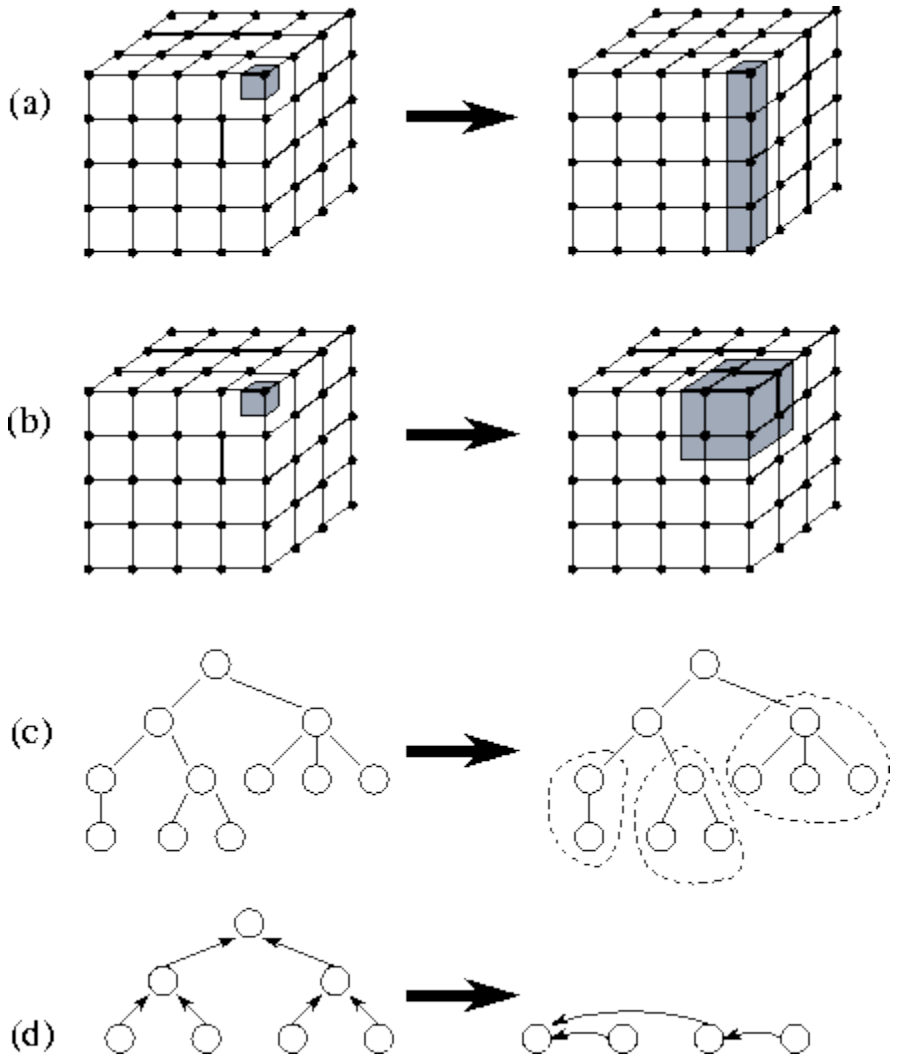

Figure 5: *Examples of agglomeration. In (a), the size of tasks is increased by reducing the dimension of the decomposition from three to two. In (b), adjacent tasks are combined to yield a three-dimensional decomposition of higher granularity. In (c), subtrees in a divide-and-conquer structure are coalesced. In (d), nodes in a tree algorithm are combined.*

The number of tasks yielded by the agglomeration phase, although reduced, may still be greater than the number of processors. In this case, our design remains somewhat abstract, since issues relating to the mapping of tasks to processors remain unresolved. Alternatively, we may choose during the agglomeration phase to reduce the number of tasks to exactly one per processor. Three sometimes-conflicting goals guide decisions concerning agglomeration and replication: reducing communication costs by increasing computation and communication *granularity*, retaining *flexibility* with respect to scalability and mapping decisions, and reducing *software engineering* costs. These goals are discussed in the next three subsections.

4.1 Increasing Granularity

In the partitioning phase of the design process, our efforts are focused on defining as many tasks as possible. This is a useful discipline because it forces us to consider a wide range of

opportunities for parallel execution. We note, however, that defining a large number of fine-grained tasks does not necessarily produce an efficient parallel algorithm.

One critical issue influencing parallel performance is communication costs. On most parallel computers, we have to stop computing in order to send and receive messages. Because we typically would rather be computing, we can improve performance by reducing the amount of time spent communicating. Clearly, this performance improvement can be achieved by sending less data. Perhaps less obviously, it can also be achieved by using fewer messages, even if we send the same amount of data. This is because each communication incurs not only a cost proportional to the amount of data transferred but also a fixed startup cost.

Replicating Computation.

We can sometimes trade off replicated computation for reduced communication requirements and/or execution time. For an example, we consider a variant of the summation problem presented earlier, in which the sum must be replicated in each of the *N* tasks that contribute to the sum.

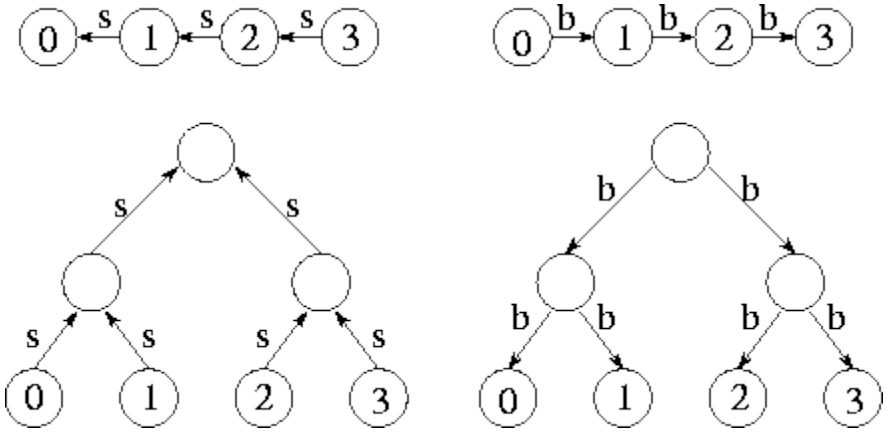

Figure 6: *Using an array (above) and a tree (below) to perform a summation and a broadcast. On the left are the communications performed for the summation (s); on the right, the communications performed for the broadcast (b). After* $2(N-1)$ *or* $2\log N$ *steps, respectively, the sum of the* ***N*** *values is replicated in each of the* ***N*** *tasks.*

A simple approach to distributing the sum is first to use either a ring- or tree-based algorithm to compute the sum in a single task, and then to *broadcast* the sum to each of the *N* tasks. The broadcast can be performed using the same communication structure as the summation; hence, the complete operation can be performed in either *2(N-1)* or $2\log N$ steps, depending on which communication structure is used (Figure 6).

These algorithms are optimal in the sense that they do not perform any unnecessary computation or communication. However, there also exist alternative algorithms that execute in less elapsed time, although at the expense of unnecessary (replicated) computation and communication. The basic idea is to perform multiple summations concurrently, with each concurrent summation producing a value in a different task.

4.2 Preserving Flexibility

It is easy when agglomerating tasks to make design decisions that limit unnecessarily an algorithm's scalability. For example, we might choose to decompose a multidimensional data structure in just a single dimension, reasoning that this provides more than enough concurrency for the number of processors available. However, this strategy is shortsighted if our program must ultimately be ported to larger parallel computers. It may also lead to a less efficient algorithm.

The ability to create a varying number of tasks is critical if a program is to be portable and scalable. Good parallel algorithms are designed to be resilient to changes in processor count. This flexibility can also be useful when tuning a code for a particular computer. If tasks often block waiting for remote data, it can be advantageous to map several tasks to a processor. Then, a blocked task need not result in a processor becoming idle, since another task may be able to execute in its place. In this way, one task's communication is overlapped with another task's computation.

A third benefit of creating more tasks than processors is that doing so provides greater scope for mapping strategies that balance computational load over available processors. As a general rule of thumb, we could require that there be at least an order of magnitude more tasks than processors. This issue is discussed in the next section.

The optimal number of tasks is typically best determined by a combination of analytic modeling and empirical studies. Flexibility does not necessarily require that a design always create a large number of tasks. Granularity can be controlled by a compile-time or runtime parameter. What is important is that a design not incorporate unnecessary limits on the number of tasks that can be created.

4.3 Reducing Software Engineering Costs

So far, we have assumed that our choice of agglomeration strategy is determined solely by a desire to improve the efficiency and flexibility of a parallel algorithm. An additional concern, which can be particularly important when parallelizing existing sequential codes, is the relative development costs associated with different partitioning strategies. From this perspective, the most interesting strategies may be those that avoid extensive code changes.

Frequently, we are concerned with designing a parallel algorithm that must execute as part of a larger system. In this case, another software engineering issue that must be considered is the data distributions utilized by other program components. For example, the best algorithm for some program component may require that an input array data structure be decomposed in three dimensions, while a preceding phase of the computation generates a two-dimensional decomposition. Either one or both algorithms must be changed, or an explicit restructuring phase must be incorporated in the computation. Each approach has different performance characteristics.

4.4 Agglomeration Design Checklist

We have now revised the partitioning and communication decisions developed in the first two design stages by agglomerating tasks and communication operations. We may have agglomerated tasks because analysis of communication requirements shows that the original partition created tasks that cannot execute concurrently. Alternatively, we may have used agglomeration to increase computation and communication granularity and/or to decrease software engineering costs, even though opportunities for concurrent execution are reduced. At this stage, we evaluate our design with respect to the following checklist. Several of these questions emphasize quantitative performance analysis, which becomes more important as we move from the abstract to the concrete.

1. Has agglomeration reduced communication costs by increasing locality? If not, examine your algorithm to determine whether this could be achieved using an alternative agglomeration strategy.
2. If agglomeration has replicated computation, have you verified that the benefits of this replication outweigh its costs, for a range of problem sizes and processor counts?
3. If agglomeration replicates data, have you verified that this does not compromise the scalability of your algorithm by restricting the range of problem sizes or processor counts that it can address?
4. Has agglomeration yielded tasks with similar computation and communication costs? The larger the tasks created by agglomeration, the more important it is that they have similar costs. If we have created just one task per processor, then these tasks should have nearly identical costs.
5. Does the number of tasks still scale with problem size? If not, then your algorithm is no longer able to solve larger problems on larger parallel computers.
6. If agglomeration eliminated opportunities for concurrent execution, have you verified that there is sufficient concurrency for current and future target computers? An algorithm with insufficient concurrency may still be the most efficient, if other algorithms have excessive communication costs; performance models can be used to quantify these tradeoffs.
7. Can the number of tasks be reduced still further, without introducing load imbalances, increasing software engineering costs, or reducing scalability? Other things being equal, algorithms that create fewer larger-grained tasks are often simpler and more efficient than those that create many fine-grained tasks.
8. If you are parallelizing an existing sequential program, have you considered the cost of the modifications required to the sequential code? If these costs are high, consider alternative agglomeration strategies that increase opportunities for code reuse. If the resulting algorithms are less efficient, use performance modeling techniques to estimate cost tradeoffs.

5. Mapping

In the fourth and final stage of the parallel algorithm design process, we specify where each task is to execute. This mapping problem does not arise on uniprocessors or on shared-memory computers that provide automatic task scheduling. In these computers, a set of tasks and associated communication requirements is a sufficient specification for a parallel algorithm; operating system or hardware mechanisms can be relied upon to schedule executable tasks to

available processors. Unfortunately, general-purpose mapping mechanisms have yet to be developed for scalable parallel computers. In general, mapping remains a difficult problem that must be explicitly addressed when designing parallel algorithms.

Our goal in developing mapping algorithms is normally to minimize total execution time. We use two strategies to achieve this goal:

1. We place tasks that are able to execute concurrently on *different* processors, so as to enhance concurrency.
2. We place tasks that communicate frequently on the *same* processor, so as to increase locality.

Clearly, these two strategies will sometimes conflict, in which case our design will involve tradeoffs. In addition, resource limitations may restrict the number of tasks that can be placed on a single processor.

The mapping problem is known to be *NP -complete*, meaning that no computationally tractable (polynomial-time) algorithm can exist for evaluating these tradeoffs in the general case. However, considerable knowledge has been gained on specialized strategies and heuristics and the classes of problem for which they are effective.