Chapter 3 Parallel Algorithm Design

> *"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."* - Brian Kernighan [1]

3.1 Task/Channel Model

The methodology used in this course is the same as that used in the Quinn textbook [2], which is the ***task/channel methodology*** described by Foster [3]. In this model, a parallel program is viewed as a collection of tasks that communicate by sending messages to each other through channels. Figure 3.1 shows a task/channel representation of a hypothetical parallel program.

A ***task*** consists of an executable unit (think of it as a program), together with its ***local memory*** and a collection of ***I/O ports***. The local memory contains program code and private data, i.e., the data to which it has exclusive access. An access to this memory is called a ***local data access***. The only way that a task can send copies of its local data to other tasks is through its output ports, and conversely, it can only receive data from other tasks through its input ports. An I/O port is an abstraction; it corresponds to some memory location that the task will use for sending or receiving data. Data sent or received through a channel is called a ***non-local data access***.

A ***channel*** is a message queue that connects one task's output port to another task's input port. A channel is reliable in the following sense:

- Data values sent to the input port appear on the output port in the same order.
- No data values are lost and none are duplicated.

The I/O model is asymmetric:

- A task that tries to receive data from an empty queue is ***blocked***[1] until a message appears in the queue, but
- a task that sends data into a queue is never blocked.

This is very much like the default semantics of I/O in C/C++: reads are, by default, ***synchronous***, meaning that the process is blocked (i.e., made to wait) until the data is available for reading, but writes are ***asynchronous***, meaning that the process continues as soon as the data is copied into a buffer, and subsequently written to the specified target device by the operating system. The fact that a task never blocks when sending data implies that the I/O model treats the queue as having unbounded size.

There is an inherent, intentional, asymmetry between local and non-local data accesses. Access to local memory is considered to be much faster than access to data through a channel. You should think of a local data access, whether it is a read or a write, as an access to a locally attached RAM, typically through a dedicated processor/memory bus. It might even be an access to a (much faster) cache. In contrast, a non-local access requires data to be transferred across a much slower link.

[1] For those who have not had an operating systems course, ***blocking a process*** (i.e., a task in this context) means changing its state so that it uses fewer resources on the machine. It almost always means that it is not allowed to execute on the processor. It could also mean that it can be swapped out of memory, if the operating system needs more memory for other processes. Blocking is an abstraction that has more than one realizable interpretation, depending on the particular operating system and the cause for which it is blocked, but you should think of a blocked process as one that cannot execute.

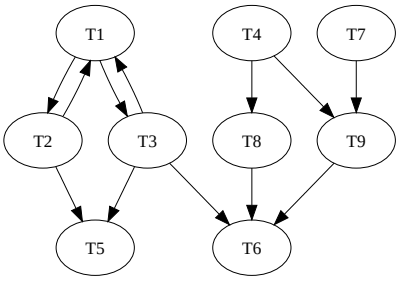


Figure 3.1: A parallel computation viewed as a task/channel graph. The nodes represent tasks; the edges, channels.

The ***lifetime*** of a parallel program or algorithm in the task/channel model is defined as the time during which any task is ***active***. A task is active from the point at which it starts execution until the point that it terminates. Therefore, a program is active from the moment the first tasks starts until the moment that the last task terminates.

3.2 Foster's Design Methodology

It is not easy to design a parallel program from scratch without some logical methodology. It is far better to use a proven methodology that is general enough and that can be followed easily. Otherwise you will not learn from the mistakes of others. In 1995, Ian Foster proposed such a methodology [3], which has come to be called ***Foster's design methodology***. It is a four-stage design process whose input is the problem statement, as shown in Figure 3.2. The four stages, with brief descriptions, are

Partitioning The process of dividing the computation and the data into pieces.

Communication The process of determining how tasks will communicate with each other, distinguishing between local communication and global communication.

Agglomeration The process of grouping tasks into larger tasks to improve performance or simplify programming.

Mapping The process of assigning tasks to physical processors.

In the remainder of this chapter, we describe these steps conceptually and then work through some examples to illustrate how to apply them.

3.2.1 Partitioning

The purpose of ***partitioning*** is to discover as much parallelism as possible. This first stage, partitioning, is the only chance to do this; the remaining stages typically reduce the amount of parallelism, so the goal in this stage is to find all of it. There are two potential sources of parallelism: data and computation, leading to two complementary methods of extracting parallelism.

3.2.1.1 Domain Decomposition

Domain decomposition is a paradigm whose goal is to decompose the data into many small pieces to which parallel computations may be applied. These parallel computations will be called ***primitive tasks***. In domain decomposition, the most parallelism is achieved by identifying the largest and most frequently accessed data object, decomposing it into as many small, identical pieces as possible, and assigning a primitive task to each piece. For example, if we have a 3D model of an object, represented as a collection of 3D points

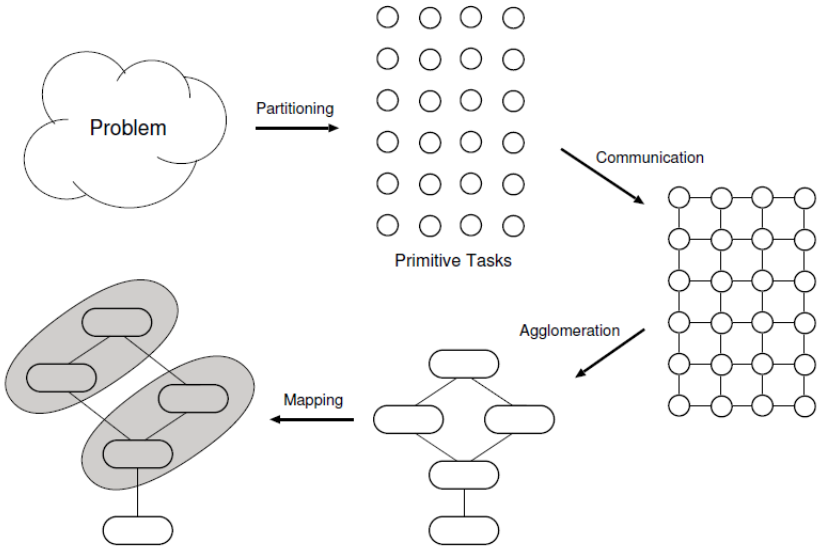


Figure 3.2: Foster's parallel algorithm design methodology.

on the surface of the object, and we want to rotate that object in space, then in theory we can apply the same operation to each point in parallel, and a domain decomposition would assign a primitive task to each point.

Quinn [2] illustrates the concept quite nicely. Imagine that the data object to be decomposed is a three-dimensional matrix of size $m \times n \times k$, where m is the number of rows, n is the number of columns, and k is the number of $m \times n$ planes. The goal of partitioning is to decompose this matrix. A very coarse decomposition would divide it into k planes and assign a task to each plane. A finer decomposition would decompose it into $n \times k$ rows and assign a task to each row. The finest decomposition would assign a task to each individual matrix element, resulting in $m \cdot n \cdot k$ tasks. This last one provides the most parallelism, so if the best choice. Figure 3.3 illustrates the differences; in the figure the shaded nodes are assigned to a single task.

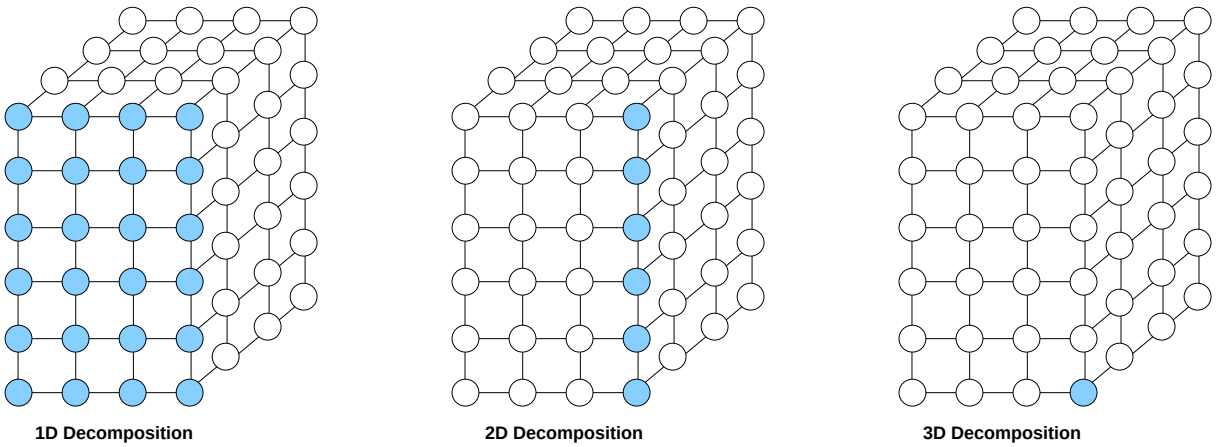


Figure 3.3: Three different domain decompositions of a 3D grid. The coarsest is the 1D decomposition and the finest is the 3D decomposition.

3.2.1.2 Functional Decomposition

Sometimes the problem to be solved does not have a large degree of inherent data parallelism (to be defined precisely later), but exhibits ***functional parallelism***. Functional parallelism exists when there are separate operations that can be applied simultaneously, typically to different parts of the data set. ***Functional decomposition*** is the paradigm in which this functional parallelism is identified, primitive tasks are assigned to these separate functions, and then the data to which these functions can be applied is identified. In this approach, the focus is on the computation that is to be performed rather than on the data manipulated by the computation. The problem is decomposed according to the work that must be done. Each task then performs a portion of the overall work. This is shown conceptually in Figure 3.4. Sometimes this results in a pipelined approach, as in an audio signal processing application, illustrated in Figure 3.5. In audio signal processing, the signal is passed through four different filters. Each filter is a separate process. As the data is passed through a filter it is sent to the next one in the line. This is an example of a ***computational pipeline***.

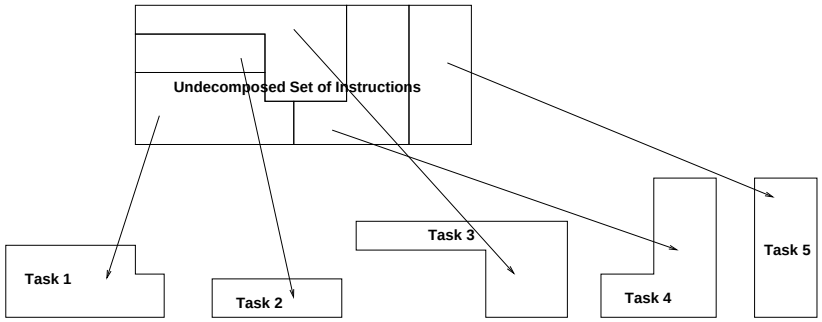


Figure 3.4: Functional decomposition, conceptually.

Sometimes functional decomposition can result in a collection of independent tasks that behave more like the components of a large system, such as an operating system. Figure 3.6, taken from [3], illustrates such a decomposition. A large project often is tackled by first applying functional decomposition, and then, within each large component, using domain decomposition to obtain more parallelism.

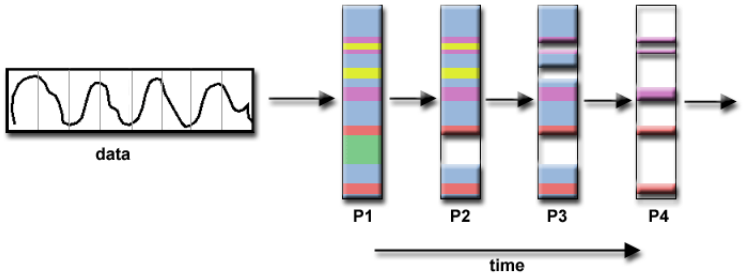


Figure 3.5: Functional decomposition resulting in a pipeline. Audio data processing passes the data through successive filters, each of which is a different function. It may also be possible to use domain decomposition within any one of these filters. (From *https://computing.llnl.gov/tutorials/parallel_comp*)

3.2.1.3 Foster's Checklist for Partitioning

Foster provides a checklist for evaluating the partitioning stage. Your partitioning should satisfy the following criteria as much as possible:

1. The partition defines 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.

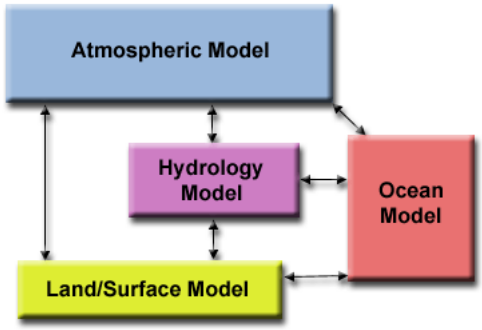


Figure 3.6: Functional decomposition resulting in a set of independent tasks that communicate with each other in a non-pipelined way. Each model component can be thought of as a separate task, which can then be parallelized by domain decomposition. Arrows represent exchanges of data between components during computation (From [3]).

2. Redundant computations and data storage are avoided as much as possible. If not, the resulting algorithm may not be able to deal with large problems.
3. Primitive tasks are roughly the same size. If not, it may be hard to allocate to each processor equal amounts of work, and this will cause an overall decrease in performance.
4. The number of tasks is an increasing function of the 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. You have identified several alternative partitions. You can maximize flexibility in subsequent design stages by considering alternatives now. Remember to investigate both domain and functional decompositions.

3.2.2 Communication

When the entire computation is one sequential program, all of the data is available to all parts of the program. When that computation is divided up into independent tasks that may execute in separate processors, some of the data needed by a task may reside in its local memory, but some of it may reside in that of other tasks. As a result, these tasks may need to exchange data with one another. This information flow is specified in the communication stage of the design. This inter-task communication does not exist in a sequential program; it is an artifact of the parallelization of the computation and is therefore considered to be entirely ***overhead***. Overhead is just a term that means the price we pay to produce something but that is not directly a part of what we produce, like the rent on a retail store. We want to minimize this overhead in our design; therefore it is important to identify it. We also want to design the program so that delays caused by this communication are minimal.

There are two types of communication among the tasks, local and global. ***Local communication*** is when a task needs values from a small number of other tasks. ***Global communication*** is when a great many tasks must contribute data to perform a computation. We will not represent global communication by channels between tasks, because it does not help us when analyzing the design. Local communications will be represented by directed edges between the task nodes; a transfer from a task T_1 to a task T_2 will be drawn as an edge from T_1 to T_2.

We will see that sometimes the communication pattern is very structured and regular, forming something like a binary tree or a two-dimensional grid, especially when it results from a domain decomposition. Figure 3.7 shows a structured communication pattern. Sometimes it will be far less structured, as would be the case for the problem shown in Figure 3.8. Processing a 3D model for display on a graphics device involves many steps, some of which are applied to each vertex in the original model. If the partitioning assigns a

unique task to each vertex, because vertices have an irregular pattern and varying numbers of neighboring vertices, the communication pattern will be unstructured.

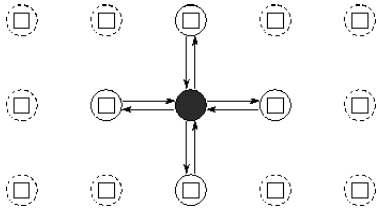


Figure 3.7: Example of a structured communication pattern. From [3]

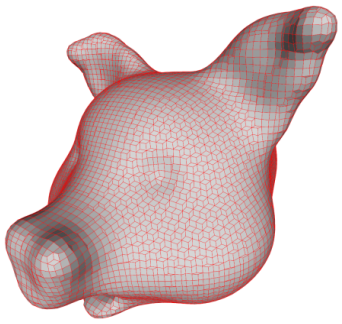


Figure 3.8: Example of a problem requiring an unstructured communication pattern. Vertices have varying numbers of neighbors.

3.2.2.1 Foster's Checklist for Communication

You should evaluate your solution using these criteria:

1. All tasks perform about the same number of communication operations. Unbalanced communication requirements suggest a design that will not scale to a larger size instance of the problem.
2. Each task should communicate only with a small number of neighbors. If each task must communicate with many other tasks, it will add too much overhead.
3. The communication operations should be able to proceed concurrently. If not, your algorithm is likely to be inefficient and non-scalable to a larger problem instance.
4. Tasks can perform their computations concurrently. If not, your algorithm is likely to be inefficient and non-scalable to a larger problem instance.

3.2.3 Agglomeration

In the first two stages of the design process, the computation is partitioned to maximize parallelism, and communication between tasks is introduced so that tasks have the data they need. The resulting algorithm is still an abstraction, because it is not designed to execute on any particular parallel computer. It may also be very inefficient, especially if there are many more tasks than processors on the target computer. This is because:

- Creating a task (process) typically uses overhead, and scheduling tasks on the processor uses overhead[2].
- The communication between tasks on the same processor adds artificial overhead, because if these tasks were combined into a single task, that communication would not exist.

Even if there are as many processors as tasks, there may be inefficiencies in the design due to the inter-task communication.

In the third and fourth stages, the abstract algorithm is made tangible and efficient by considering the class of parallel computers on which it is intended to run. In the third stage, ***agglomeration***, decisions made in the partitioning and communication phases are revisited. Agglomeration means combining groups of two or more tasks into larger tasks, in order to reduce the number of tasks, and make them larger. Its purpose is to improve performance and simplify programming. Agglomeration is an optimization problem; very often the goals are conflicting, and trade-offs are made.

3.2.3.1 Granularity

One way that performance is improved is by lowering communication overhead. When two tasks that exchange data with each other are combined into a single task, the data that was exchanged through a channel is part of a single task and that channel and the overhead are removed. This is called ***increasing locality***. Figure 3.9 shows how two tasks can be combined into one to reduce communication and increase locality.

Figure 3.9: Agglomerating to increase locality.

A second way to reduce communication overhead is by combining groups of tasks that all send and groups of tasks that all receive data from each other. In other words, suppose task S_1 sends to task R_1 and S_2 sends to R_2. If we combine S_1 and S_2 into a single task S and R_1 and R_2 into a single task R, then communication overhead will be reduced. This is because, before we agglomerated the tasks, there were two messages sent by the senders, and afterward, one longer message. The cost of sending a message has two components, the initial start-up time, called the ***latency***, which is independent of how large the message is, and the ***transmission time***, which is a function of the number of bytes sent. The transmission time is not reduced, but we cut the total latency in half. Remember that when a task is waiting for data, it cannot compute, so time spent during latency is time wasted. Figure 3.10 illustrates this type of agglomeration.

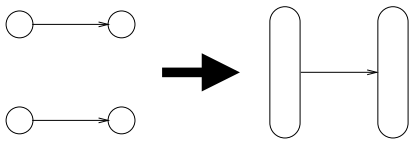


Figure 3.10: Agglomerating to reduce message transmissions. By combining the two tasks that send, and the two that receive, into a single task each, the number of transmissions is reduced, decreasing message latency.

A third way to agglomerate is to combine adjacent tasks without reducing the dimensionality of the solution but to increase granularity. Figure 3.11 shows how a 6×4 matrix that has been partitioned so that each task has one matrix element can be agglomerated by assigning four elements to each task. If the target architecture had only six processors, this might be a good solution.

[2]For those who have not had an operating systems course, the first two points need some explanation. Every task, or process, is represented by a fairly large data structure within the operating system. To create a new task, this structure must be allocated, initialized, and placed on a queue of runnable tasks. When multiple tasks share the processor, they are time-sliced onto it, and the switching from one task to another consumes processor cycles because the operating system must do some saving and restoring of machine state.

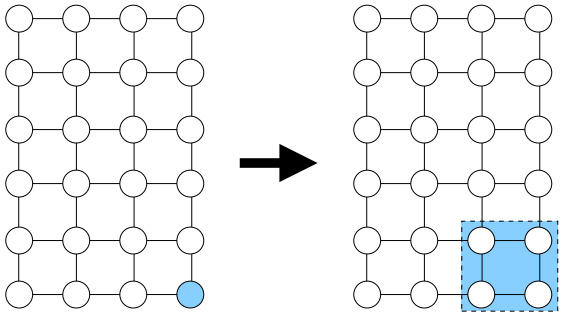


Figure 3.11: Agglomerating a two dimensional matrix by combining adjacent tasks to increase granularity.

Sometimes reducing the communication costs increases redundant computation as well as local storage. This is a factor weighing against too much agglomeration.

Another reason to agglomerate tasks is to reduce overall data storage. If each task has to have a copy of certain data items, then this data is replicated for each. If storage is an issue, then it is better to reduce the number of tasks so that storage is reduced.

3.2.3.2 Flexibility and Scalability

Too much agglomeration is not a good thing. If you reduce the number of tasks too much, then the algorithm may not scale well. For example, if we have partitioned an $N \times M$ matrix problem so that a task is assigned to each matrix element, and now we decide to agglomerate entire rows of tasks, reducing the number of tasks by a factor of M, the width of the matrix, because we have exactly N processors, this may be a shortsighted decision, because we will not be able to port the program so easily to a processor with a much larger number of processors, nor will be able to scale up to larger instances of the problem with more processors without changing the design. *Good parallel algorithms are designed to be resilient to changes in processor count.*

In short, it is always better to have more tasks than processors.

3.2.3.3 Software Engineering Costs

The relative development costs associated with different partitioning strategies is a factor when agglomerating tasks. Code reuse and avoiding code changes are factors to take into consideration. For example, if you already have code that works on two dimensional arrays to perform some task, but you have a three-dimensional grid that you decide to agglomerate into a single dimension, then you lose the choice of using that existing code. This is a factor that cannot be ignored.

3.2.3.4 Foster's Agglomeration Checklist

You should evaluate how well you have agglomerating the design by considering each of the following criteria.

1. Agglomeration should reduce communication costs by increasing locality.
2. If agglomeration has replicated computation, the benefits of this replication should outweigh its costs, for a range of problem sizes and processor counts.
3. If agglomeration replicates data, it should not compromise the scalability of the algorithm by restricting the range of problem sizes or processor counts that it can address.
4. Agglomeration should produce tasks with similar computation and communication costs.
5. The number of tasks can still scale with problem size.
6. There is sufficient concurrency for current and future target computers.

7. The number of tasks cannot be made any smaller without introducing load imbalances, increasing software engineering costs, or reducing scalability.
8. The trade-off between the chosen agglomeration and the cost of modification to existing sequential code is reasonable.

3.2.4 Mapping

Mapping, the final stage of Foster's methodology, is the procedure of assigning each task to a processor. Of course, this mapping problem does not arise on uniprocessors or on shared-memory computers whose operating systems provide automatic task scheduling. Therefore, we assume here that the target architecture is a distributed-memory parallel computer, i.e., a private memory multicomputer.

From the parallel programmer's point of view, the goal in developing mapping algorithms is to minimize total execution time, plain and simple. This is usually achieved by minimizing interprocessor communication and maximizing processor utilization. ***Processor utilization*** is the average percentage of time during which the computer's processors are actively executing code necessary to solve the problem. It is maximized when the computation is balanced evenly, because if there are processors that are idle while others are busy, then it would suggest that a different design might be able to off load the work being done by the busier ones onto the idle ones, making the total computation time smaller. This is not necessarily the case, but it is the general idea.

Two different strategies to minimize execution time are

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

Unfortunately, these two strategies will sometimes be in conflict, in which case the design will involve trade-offs. In addition, resource limitations may restrict the number of tasks that can be placed on a single processor. The general problem of mapping tasks to processors, subject to the resource limitations, in order to minimize execution time is ***NP-hard*** [4], meaning that no computationally tractable (polynomial-time) algorithm can exist for evaluating these trade-offs in the general case. The problem of mapping tasks to processors to maximize processor utilization, subject to resource limitations, is also NP-hard; in fact it is NP-complete [5].

These negative theoretical statements should not dampen our spirits though, because there are many heuristic approaches that give good results, and because in many cases, there are very good solutions that are easy to achieve. For example, usually after a domain decomposition, there is a fixed number of tasks with similar size and similar computing steps. A natural strategy is to divide the tasks among the processors in such a way that each processor has the same number of tasks, and such that adjacent tasks are assigned to the same processor. Figure 3.12 illustrates this idea.

Sometimes, some of the tasks might have a different load than the others, such as those perhaps in the corners of sub-grids. This could create a load imbalance, and a mapping such as the one illustrated in Figure 3.13 might be optimal.

Sometimes a ***cyclic***, or ***interleaved***, assignment of tasks to processors is required to alleviate load imbalances, often by paying the price of higher communication costs. In this mapping, tasks are assigned to processors using some type of modular arithmetic. In the one-dimensional case, with p processors, the assignment would be that task k is assigned to processor $k\%p$. In the two-dimensional case, there are different approaches. Figure 3.14 shows how a cyclic mapping can be applied to the same grid as was used in Figure 3.12.

3.2.4.1 A Decision Strategy for Mapping

In the preceding examples, the communication pattern was regular, so the solutions were regular. When the communication pattern is not regular, but is known in advance, a ***static load-balancing algorithm*** can be used. This is an algorithm that can be applied at compile-time to determine a mapping strategy.

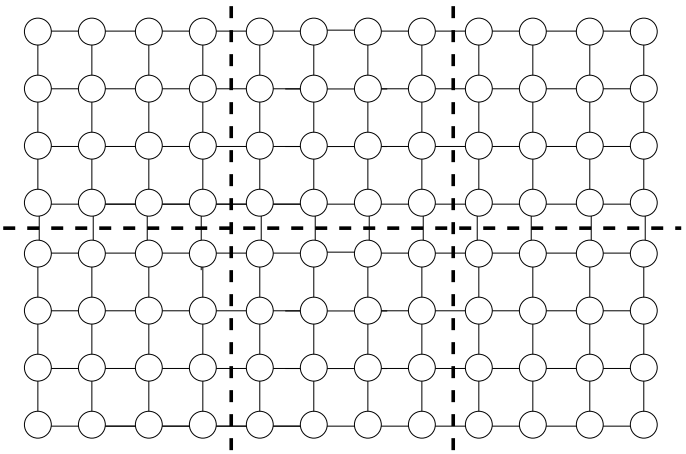


Figure 3.12: Mapping in a 2D grid problem in which each primitive task performs the same amount of computation and communicates only with its four neighbors. The heavy dashed lines delineate processor boundaries. In this example, there are 96 primitive tasks, each assigned a single grid element, agglomerated into six tasks, to be executed on a computer with six processors. The agglomeration and mapping assigns four by four arrays of primitive tasks to each processor.

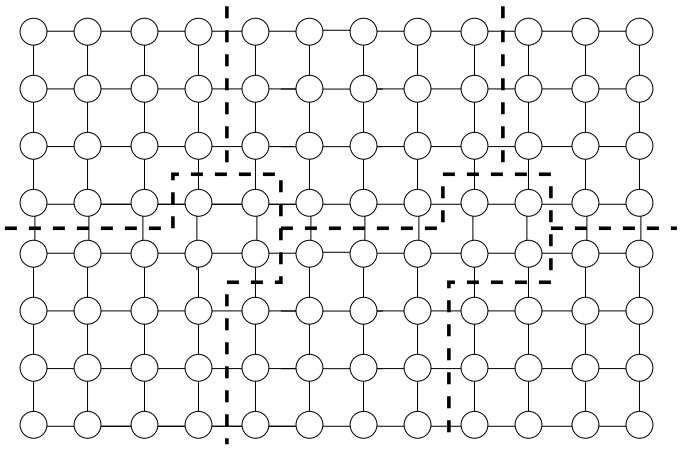


Figure 3.13: Load balancing in an imbalanced grid problem. Variable numbers of tasks are placed on each processor so as to compensate for load imbalances.

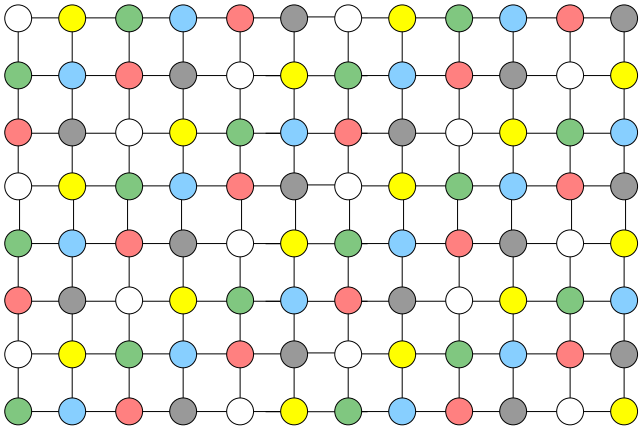


Figure 3.14: Load balancing using a cyclic mapping in a 2D grid problem. The 96 tasks are mapped to six processors, as was done in Figure 3.12, but in this case, they are arranged so that no two neighboring tasks belong to the same processor, and each processor has the same load.

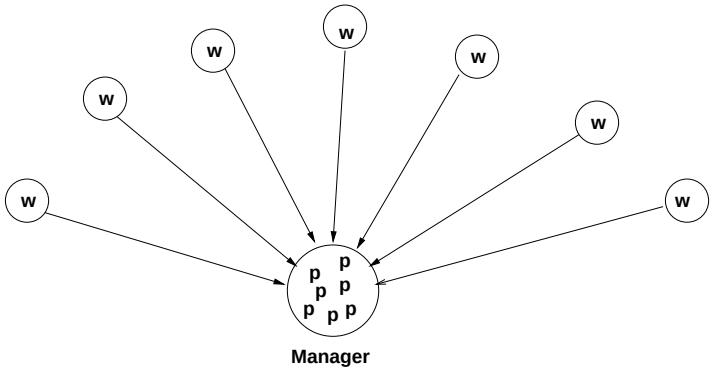


Figure 3.15: Centralized scheduling algorithm with manager and worker pool. The manager has a collection of problems and the worker processors issue requests for more work when they finish.

Sometimes the number of tasks is not known in advance, or if it is, the communication requirements are not. In either of these cases, a ***dynamic load-balancing algorithm*** must be used. A dynamic load-balancing algorithm analyzes the set of running tasks and generates a new mapping of tasks to processors.

Sometimes the tasks are short-lived; they are created on the fly to solve small problems and then they are destroyed, and they do not communicate with each other. In this case, a ***task scheduling algorithm*** runs while the parallel program is running and manages the mapping of tasks to processors. Such an algorithm can be centralized or distributed. In general, task scheduling algorithms can be used when a functional decomposition yields many tasks, each with weak locality requirements.

In a centralized task scheduling algorithm, one processor becomes a manager and the remaining processors are used to run worker tasks. A pool of "problems" is maintained, into which new problems are placed and from which problems are taken for allocation to the worker processors. Each worker processor runs to solve a problem, and when it is finished, it returns its results to the manager and requests a new problem. Figure 3.15 illustrates the idea.

One problem with the centralized scheduler is that the manager becomes a bottleneck. In a distributed scheduling algorithm, there is no manager. Instead, a separate task pool is maintained on each processor, and idle workers request problems from other processors. The task pool is, in effect, a distributed data structure that is accessed by the different tasks asynchronously. This is a more scalable solution than a centralized one.

Quinn presents a decision tree that he uses throughout the textbook for deciding on how to map tasks to

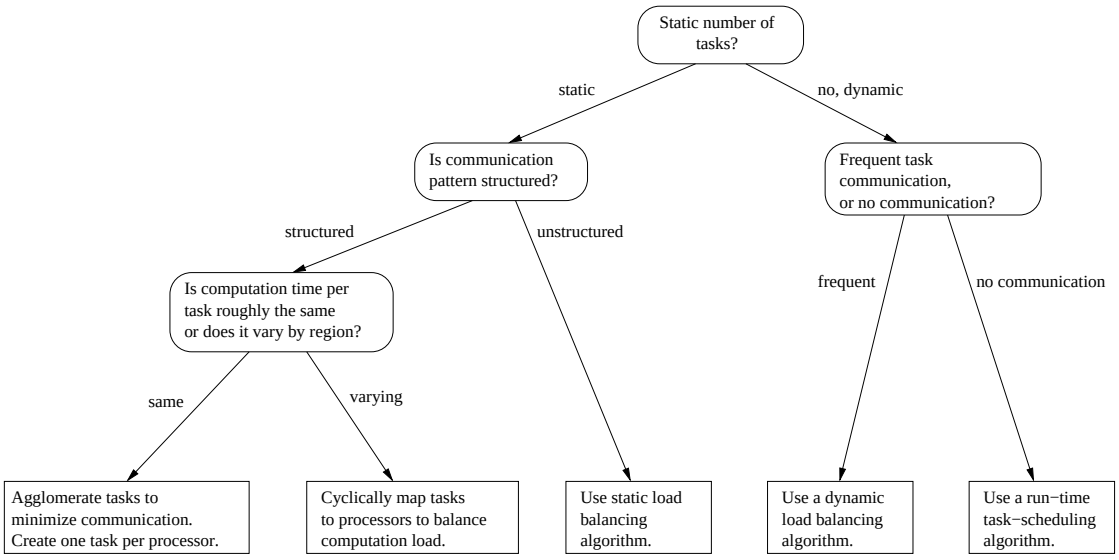


Figure 3.16: Decision tree for deciding a task mapping strategy. (from [2]).

processors. It is replicated in Figure 3.16. We will use this decision tree throughout the remainder of this course.

3.2.4.2 Foster's Mapping Design Checklist

The following checklist can serve as a basis for an informal evaluation of the mapping design.

1. Designs based on one task per processor and multiple tasks per processor have been considered.
2. Both static and dynamic allocation of tasks to processors have been considered.
3. If a centralized load-balancing scheme is chosen, you have made sure that the manager will not become a bottleneck.
4. If a dynamic load-balancing scheme is chosen, you have evaluated the relative costs of different strategies, and should account for the implementation costs in the analysis.

3.3 An Application: Boundary Value Problem

3.3.1 The Problem

Boundary value problems occur in a wide variety of scientific areas. Intuitively, a boundary value problem is one in which conditions at the boundaries of some physical object are known, as well as the physical principles that define how those conditions propagate to the interior over time, and we want to know the conditions at arbitrary points in the interior over time. A precise definition of a ***boundary value problem*** is that it is a differential equation together with a set of additional constraints, called the ***boundary conditions***. A solution to a boundary value problem is a solution to its differential equation which also satisfies the boundary conditions. A ***differential equation*** is an equation that relates a function and that of one or more of its derivatives to each other, such as $f'(x) = f(x)$, which states that the first derivative of $f()$ is $f()$ itself, for all values of x. In this equation, the unknown is the function itself. A solution to a differential equation is a function that satisfies the equation. If you recall your calculus, the derivative of e^x is e^x, so one solution to this is $f(x) = e^x$.

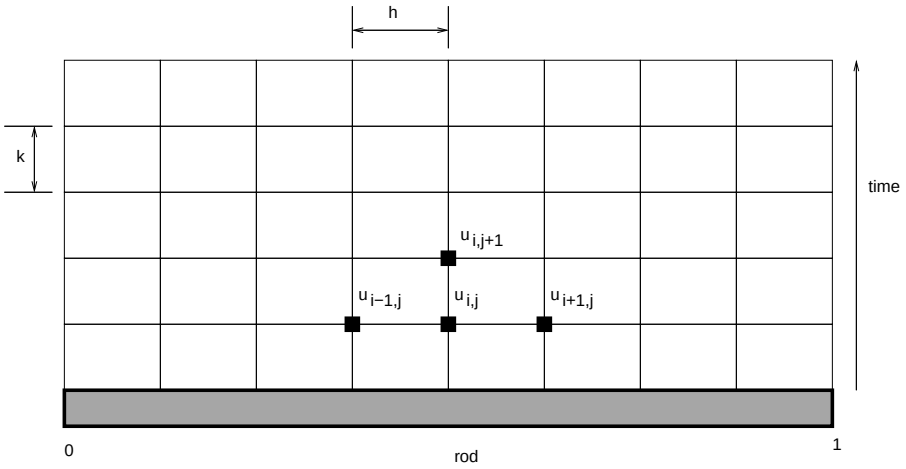


Figure 3.17: Two dimensional grid representing points at which solution is approximated.

The boundary value problem we consider is a one-dimensional instance of the heat transfer problem. Imagine a thin rod of unit length, made of a uniform material, whose ends are connected to ice baths at temperature 0° C (in other words, freezing cold.) The rod is surrounded by an insulating material so that the only temperature changes possible on it are a result of the heat transfer at the ends of the rod and through the rod itself as a result of conduction. The ice baths keep the ends at 0° C at all times. Initially, suppose that the heat distribution across the rod is given by the function

$$U(x) = 100 \cdot \sin(\pi x) \tag{3.1}$$

where x ranges over the interval $[0, 1]$. This is a sin() curve with a maximum value of 100 at $x = 0.5$. (Remember that $\sin(\theta)$ is zero when $\theta = \pi$ and $\theta = 0$ and is maximal at $\theta = \pi/2$.) As no heat is being applied, but the ends of the rod are being cooled, the rod gets colder over time. We would like to model how the temperature changes over time everywhere along the rod.

It is known that the temperature of any point x on the rod at a given time t is governed by a very specific ***partial differential equation***. A partial differential equation is a differential equation with partial derivatives in it. Sometimes we can find the solution to a differential equation analytically, but often we cannot. Here, we will find the solution numerically by using a ***finite difference equation***. A finite difference equation is like the discrete analog of a differential equation. (Differential equations and their relationship to finite difference equations are explained further in the Appendix.) "Numerically" means that we will use an iterative procedure that approximates the solution closer and closer with each iteration, until we have the desired accuracy.

We let $u(x, t)$ denote the temperature at point x on the rod at time t. To approximate the temperature at x, we divide the rod into n segments of uniform length h. Obviously, if the rod is of length 1, then $h = 1/n$, but it is more convenient to let h denote the length of the segment. The larger n is, the smaller h is, and the more accurate the results will be. If we want to estimate the temperature at any point during the time interval from the start time to some arbitrary time T, then we will also divide the time interval from 0 to T into m uniform length intervals of length k. This implies, of course, that $k = T/m$. We only need to evaluate the temperature at the $n - 1$ points h, $2h$, $3h$, ..., $(n - 1)h$ on the rod, because the temperature at the endpoints 0 and $1 = nh$ remain at 0 degrees. We label these points on the rod 0, 1, 2, ..., n such that i is the point at distance $i \cdot h$ from the 0-end of the rod. Time intervals will be labeled j, such that the time elapsed from 0 to j is $j \cdot k$. Therefore, our approximation to $u(x, t)$ is a collection of points on a two-dimensional grid denoted $u_{i,j}$ representing the temperature at point i and time j.

The finite difference equation that predicts the temperature at time $j + 1$ at a given point i, given the temperatures at time j at points i, $i - 1$, and $i + 1$, is

$$u_{i,j+1} = u_{i,j} + \frac{k \cdot u_{i-1,j} - 2k \cdot u_{i,j} + k \cdot u_{i+1,j}}{h^2} \tag{3.2}$$

where h is the distance between adjacent points and k is the difference between times j and $j+1$. This shows that we can model the set of temperatures along the rod at various times using the temperatures at the grid points $u_{i,j}$. Notice that row j has the temperature of the rod at time j for all points i, $1 \leq i \leq n-1$, and column i has the temperature at point i for all times j, $0 \leq j \leq m$. Notice that this equation states that the temperature at point i at time $j+1$ only depends on the temperatures at time j at the three points i, $i-1$, and $i+1$. The problem is how to find the values of the grid points in parallel; we will apply the four-step task/channel method to devise a solution.

3.3.2 Partitioning

Each grid point represents a value to be computed and can be associated with a single task whose mission is to calculate that value. Consequently, there are $m(n-1)$ tasks. This is a two-dimensional decomposition of the domain. While it might be obvious in this case that creating this many tasks is unnecessary, it is a good exercise to start with that many tasks, to understand how the method works.

3.3.3 Communication

There is an asymmetric communication pattern, as shown in Figure 3.18. The task that computes $u_{i,j+1}$needs the values of $u_{i-1,j}$, $u_{i,j}$, and $u_{i+1,j}$, each of which must have a channel to that task. Similarly, it will need to send values to tasks $u_{i-1,j+2}$, $u_{i,j+2}$, and $u_{i+1,j+2}$. Thus, each task, except those on the edges, has three incoming channels and three outgoing channels.

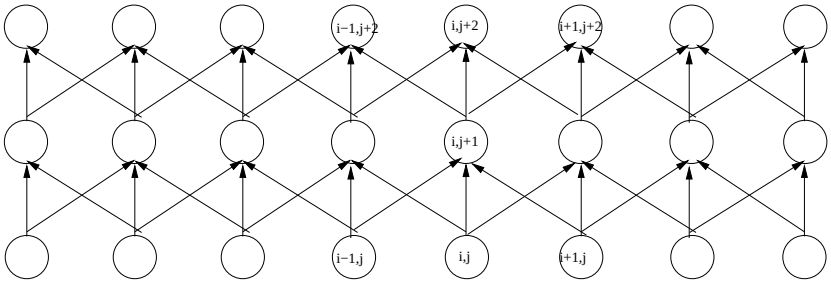


Figure 3.18: Communication among tasks

3.3.4 Agglomeration

If we actually created a task for each grid point, there would be many, many tasks. This by itself is not a problem. The real problem is that the tasks in row j can do no work until the tasks in row $j-1$ do their work, and they cannot do work until the tasks in the preceding row finish theirs, and so on. In short, we would be creating a great many tasks, the majority of which would be idle for most of their lifetimes. The task/channel graph should make it fairly obvious that there is an inherently sequential aspect to this computation, and that creating separate tasks to do what are sequential activities is inefficient. Therefore, it makes sense to agglomerate the tasks in each column of the grid, associating one task with each point on the rod instead, and making that task in charge of computing the temperature at that point for all times t. The resulting task/channel graph is shown in Figure 3.19. This graph has the property that each non-edge task communicates with just two of its neighbors.

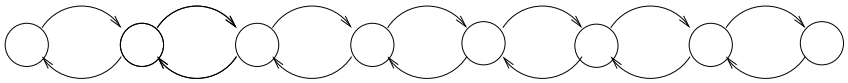


Figure 3.19: After agglomerating the tasks so that each task has a unique grid point.

3.3.5 Mapping

In many problems of this nature, the problem size is related to the desired degree of approximation; as the accuracy is increased, the number of grid points is increased, increasing problem size and, based on how we have partitioned the problem, the number of tasks. It is likely that the number of tasks will exceed the number of processors. Now we use the decision tree from Figure 3.16 to decide which mapping strategy to use. Starting at the top, for any fixed problem, we have a static number of tasks, communicating in a very structured way, so we travel down two left edges in the tree. Each task, except the edge tasks, does the exact same amount of work, so we are led to the conclusion that we should agglomerate tasks to minimize communication, creating one task per processor.

The best way to reduce intertask communication is to map adjacent tasks to the same processor. If there are $n-1$ tasks and p processors, then we assign $\lceil (n-1)/p \rceil$ tasks to each processor, as depicted in Figure 3.20. (One processor will have fewer in case p is not a factor of $n-1$.)

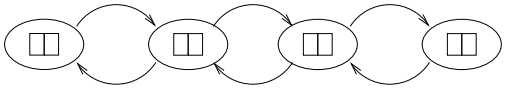


Figure 3.20: After mapping so that there is one task per processor.

3.3.6 Analysis

Whenever we develop a parallel algorithm, we will want to analyze its performance. In particular, we will want to know its running time as a function of the problem size and the number of processors. This is our first practice run.

The rod has n segments of size h. Following Quinn's notation, we will let χ denote the amount of time it takes a single task to compute formula 3.2, to obtain $u_{i,j+1}$ from the values of $u_{i-1,j}$, $u_{i,j}$, and $u_{i+1,j}$. If we ran this algorithm on a single processor, it would apply this formula $n-1$ times in each of m time steps, for a total time of $m(n-1)\chi$.

With p processors, each iteration requires $\lceil (n-1)/p \rceil \chi$ computation time because each processor has $\lceil (n-1)/p \rceil$ points on the rod, and it computes each one after the other. The parallel algorithm has communication costs as well, unlike the sequential algorithm. We address this issue now.

In the task/channel model, a task can only send one message at a time, but it can receive a message at the same time that it sends one. Sending a message requires that the task stop whatever computing it is doing and execute instructions to send the message. Let us denote the amount of time it takes to send a message by λ. We will also assume that λ is the amount of time it takes to receive a message as well. As each task sends two messages to its neighbors on either side, simultaneously with receiving two, each task requires 2λ time units to communicate in each iteration. Therefore, each iteration of the parallel algorithm requires $\lceil (n-1)/p \rceil \chi + 2\lambda$ total execution time, for a total time of $m\,(\lceil (n-1)/p \rceil \chi + 2\lambda)$.

3.4 Reduction: Finding a Maximum Value

3.4.1 Introduction

The purpose of this section is to introduce two important concepts: reduction and binomial trees. The preceding problem was a bit artificial because that particular boundary problem actually has a precise

analytical solution; there was no need to find an approximation. But suppose that we implemented the algorithm and then wanted to know how close it was to the actual answer. It is possible to compute the error between the approximation and the exact answer at any of the $n-1$ points along the rod, in any iteration. If the approximated solution at a given point is denoted b, and the actual answer is denoted by y , then the error is defined by $|(b-y)/y|$. We could augment the algorithm to compute the error at each point and find the maximum error. This would tell us how accurate the method is for that problem. The problem we now solve is how to efficiently find the maximum of the set of error values being held by each of the p tasks.

The maximum of two numbers is a binary operator (as is the minimum.) We usually do not write it as in infix operator, but suppose we did. We will let $a \diamond b$ denote the maximum of a and b. This operator is clearly associative: $(a \diamond b) \diamond c = a \diamond (b \diamond c)$. Given a set of n values $a_0, a_1, a_2, ..., a_{n-1}$, and any associative binary operator $\oplus$, ***reduction*** is the process of computing $a_0 \oplus a_1 \oplus a_2 \oplus \cdots \oplus a_{n-1}$. There are many associative binary operators: addition, multiplication, maximum, minimum, logical-or, logical-and, just to name the obvious ones. Any parallel algorithm that we devise that can perform a parallel reduction can be applied to any of these operators.

Although our original problem was to find the maximum, we will instead show how to perform a parallel reduction using the summation operator, just because it is more familiar. We will devise an algorithm that finds the sum of n numbers, each of which is held by a unique task. Our goal is to use Foster's methodology to develop a parallel algorithm to compute their sum.

3.4.2 Partitioning

This stage is trivial: as there are n distinct numbers, the finest partition would be the one that assigns a task to each number. Hence we start with n tasks.

3.4.3 Communication

Remember that there is no shared memory in our computational model; tasks must exchange data through messages. To compute the sum of two numbers held by tasks T_1 and T_2, one must send its number to the other, which will then sum them. When the algorithm is finished, the sum must be in a single task. This task will be known as the root task, consistent with Quinn's terminology.

A naive solution would be for each task to send its value to the root task, which would add them all up. Letting λ denote the time for a task to send or receive a value from another task, and χ denote the time to add two numbers, this naive algorithm would require $(n-1)$ additions in the root task, taking a total of $(n-1)\chi$ time, and $(n-1)$ receive operations by the root task, totaling $(n-1)\lambda$ time for the communication, for a total of $(n-1)(\chi+\lambda)$ time. The sequential algorithm would only take $(n-1)\chi$ time, so this parallel algorithm is worse than the sequential one! It is no wonder, because in this algorithm, most of the tasks did nothing for most of the time.

We now devise a smarter communication pattern, based on a binary divide-and-conquer approach. We will lead up to it in stages. Imagine first that we replace the single root task by two co-root tasks, and assume that n is even for simplicity. Each co-root task will be sent $(n/2-1)$ values from $(n/2-1)$ of the $(n-2)$ remaining tasks. After each co-root task receives all numbers, it will add up their values and then one will send its partial sum to the other, which will sum these. The first phase would take $(n/2-1)(\chi+\lambda)$ time in each co-root task, and the last communication and sum would take an additional $(\chi+\lambda)$ time. The total time would be $(n/2-1)(\chi+\lambda)+(\chi+\lambda) = (n/2)(\chi+\lambda)$. This halved the running time.

We can extend this idea one step in the same direction. Assume there are four co-root tasks and n is divisible by 4 for simplicity. Each co-root will receive $(n/4-1)$ values from $(n/4-1)$ of the remaining $(n-4)$ tasks and add them up. Next, two of these co-root tasks will send their partial sums to one of the other two. The ones that receive them will add them up and then one would send its partial sum to the other, which will add these, forming the grand total. The time for the first parallel summation would be $(n/4-1)(\chi+\lambda)$. The second communication and summation by the two co-roots would be $(\chi+\lambda)$, and the last communication and summation would be $(\chi+\lambda)$, making the total time $(n/4-1)(\chi+\lambda)+(\chi+\lambda)+(\chi+\lambda) = (n/4+1)(\chi+\lambda)$.

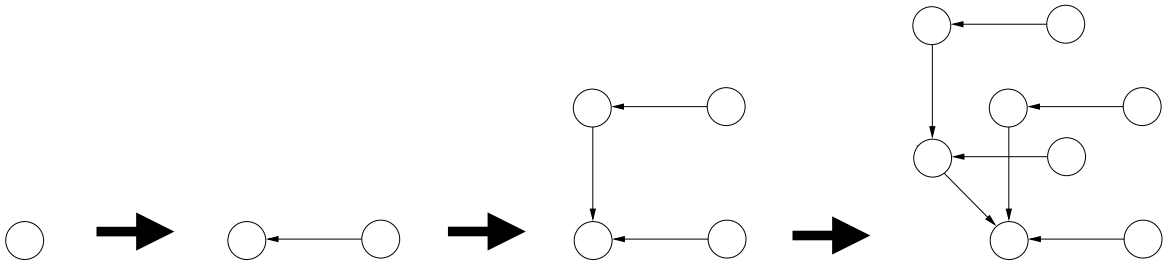
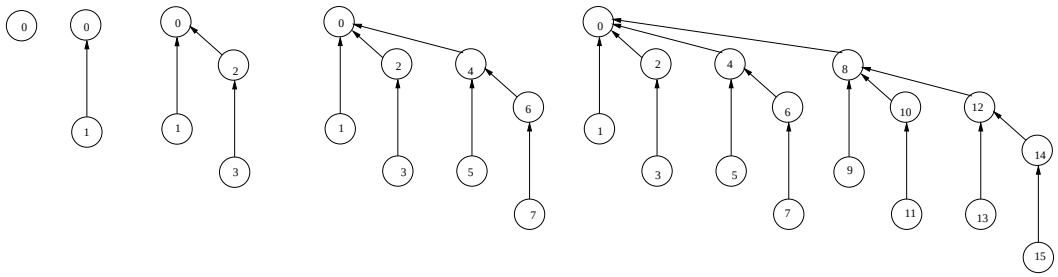
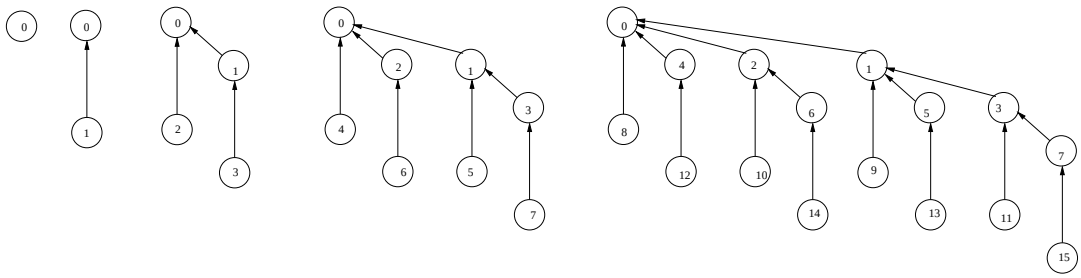


Figure 3.21: Binomial trees with $n = 1, 2, 4, 8$ nodes.

We now take this strategy to the limit. Suppose for simplicity that $n = 2^k$ for some k. Let us denote the tasks by T_0, T_1, T_2, ..., T_{n-1}. The algorithm starts with tasks $T_{n/2}$, $T_{n/2+1}$, $T_{n/2+2}$, ..., T_{n-1} each sending its number to tasks T_0, T_1, T_2, ..., $T_{n/2-1}$respectively, which each perform the sum in parallel. We now have the exact same problem as before, except that n is half of what it was. We repeat this same logic: The upper half of the set of tasks T_0, T_1, T_2, ..., $T_{n/2-1}$ sends its numbers to the lower half, and each task in the lower half adds its pair of numbers. This sequence of actions is repeated until $n = 1$, at which point T_0 has the total. The communication pattern is illustrated in Figure 3.21 for $n = 1$, 2, 4, 8. Such graphs are called ***binomial trees***.



(a) Another representation of binomial trees, showing the recursive nature of their definition. The numbers in the nodes are the labels we use to name the nodes. Notice that the tree of order $k+1$ is formed by cloning the tree of order k and labeling each node in the clone by adding 2^k to its old label. This labeling does not correspond to the reduction algorithm suggested in the text.



(b) This is the same representation of the binomial tree but the nodes have been relabeled to correspond to the reduction algorithm described in the text. The node labels do not correspond in a natural way to how the trees can grow recursively, but they do correspond to how processes can communicate in a parallel reduction.

Figure 3.22: An alternate depiction of binomial trees with two different node labelings. The labeling in Subfigure 3.22a does not correspond to the algorithm described here, whereas the labeling in Subfigure 3.22b does.

Definition 1. A binomial tree B_n of order $n \geq 0$ is a rooted tree such that, if $n = 0$, B_0 is a single node called the ***root***, and if $n > 0$, B_n is obtained by taking two disjoint copies of B_{n-1} and joining their roots by an edge, then taking the root of the first copy to be the root of B_n.

A binomial tree of order n has $N = 2^n$ nodes and $2^n - 1$ edges. (Each node except the root has exactly one

outgoing edge.) The maximum distance from any node to the root of the tree is n, or $log_2 N$. This means that a parallel reduction can always be performed with at most $log_2 N$ communication steps. Notice too that the number of leaf nodes in a binomial tree of order $n > 0$ is 2^{n-1}.

Binomial trees are commonly drawn as depicted in Figure 3.22a, but the labeling used there is different than the one we described in the preceding paragraph. They arise as the communication pattern in many parallel algorithms, which is why they are important. They also arise in financial modeling. You might notice the strong resemblance between a binomial tree and a hypercube, especially the way in which they are drawn in Figure 3.21. In fact, one can obtain a binomial tree of order n from a hypercube of order n (i.e., one with 2^n nodes) by removing an appropriate set of $(n-2)2^{n-1}+1$ edges and converting the remaining edges to directed edges.

Figure 3.23 shows how sixteen numbers distributed across sixteen tasks can be summed using this parallel reduction algorithm in four steps.

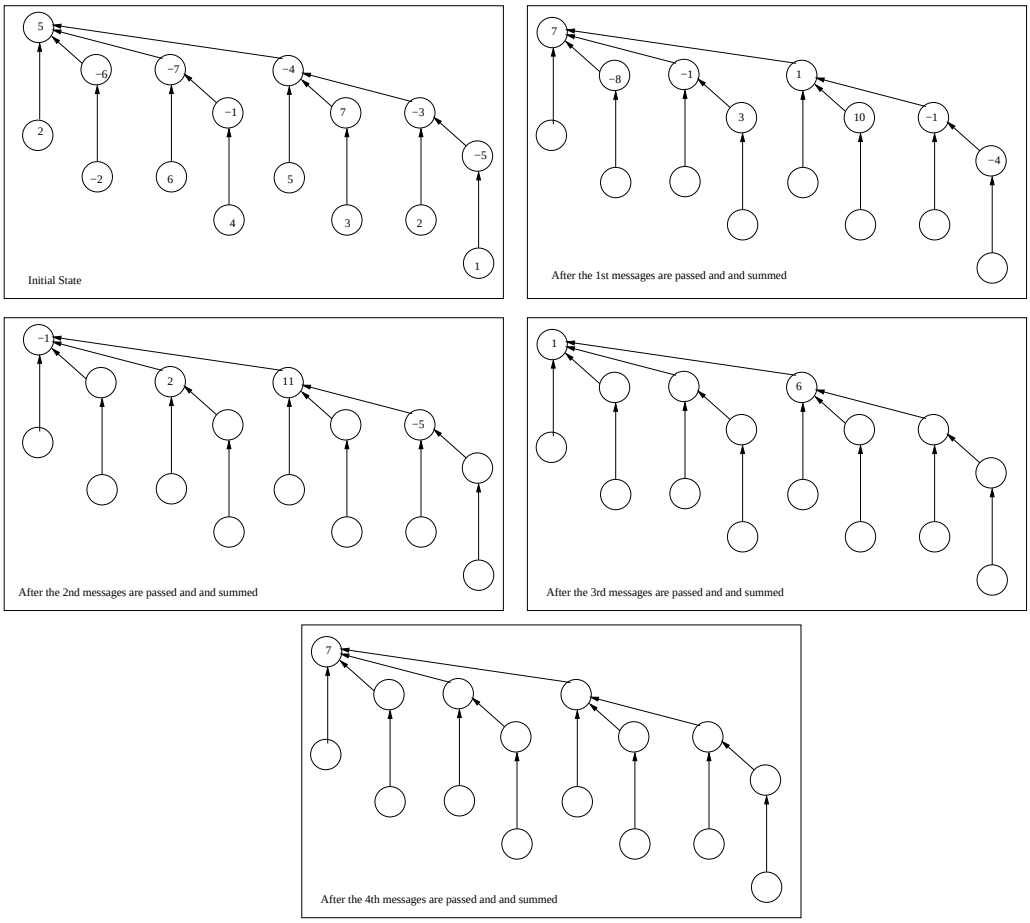


Figure 3.23: Parallel reduction of 16 numbers in 4 communication steps. The five boxes show how the values are passed in each step. The sum is in the root task after the 4th step.

We simplified this algorithm by making n a power of 2. If n is not a power of 2, then there is one extra step. To see this, let $n = 2^k + r$, for $0 < r < 2^k$. What will happen is that in the very first step, r of the tasks will send their messages to r other tasks. At that point, the problem is one in which n is a power of 2 again. The easiest way to think about the problem is to picture the binomial tree of order $k+1$, except with $2^k - r$ imaginary nodes in the tree. We can paint the imaginary nodes red. The imaginary nodes cannot be arbitrary nodes; they must be leaf nodes in the tree. If you look at Figure 3.23 and imagine any seven of the eight numbers in the leaf nodes missing, you will realize that a parallel reduction of $2^k + r$ numbers takes the same amount of time as a parallel reduction of 2^{k+1}numbers. Therefore, if n is a power of 2, the

reduction takes $\log n$ steps and if it is not, it takes $\lfloor \log n \rfloor + 1$ steps. This is equivalent to $\lceil \log n \rceil$ steps.

3.4.4 Agglomeration and Mapping

It is likely that the number of processors p will be much smaller than the number of tasks n in any realistic problem. Assume for now that both n and p are powers of 2. Using the decision tree from Figure 3.16, starting at the top, for any fixed problem, we have a static number of tasks, communicating in a very structured way. Although each task does not do the exact same amount of work, they all follow the same algorithm, with some being active longer than others. The pattern is very regular. We are led to the conclusion that we should agglomerate tasks to minimize communication. While we could assign n/p tasks to each processor, we gain nothing by it, so instead we create one task per processor. Each task will begin by adding n/p numbers. The question is how to do the mapping.

The mapping must be done so that the agglomerated tasks remain a binomial tree. Assume $p = 2^m$ and $n = 2^k$, $m \leq k$. We form a partition of the binomial tree in which 2^{k-m} "leaf" tasks are in each cell as follows. Using the node labeling from Figure 3.22a, and assuming node labels are k bits long, all nodes whose label's upper m bits are the same will be agglomerated into a single task. For example, if $p = 2^2$ and $n = 2^4$, then the nodes whose upper bits are 00 are in one task, 01, in a second, 10 in a third, and 11 in the fourth. If $p = 2^1$, then all nodes whose upper bit is 0 are in one task and those whose upper bit is 1 in the other. Figure 3.22a illustrates two mappings.

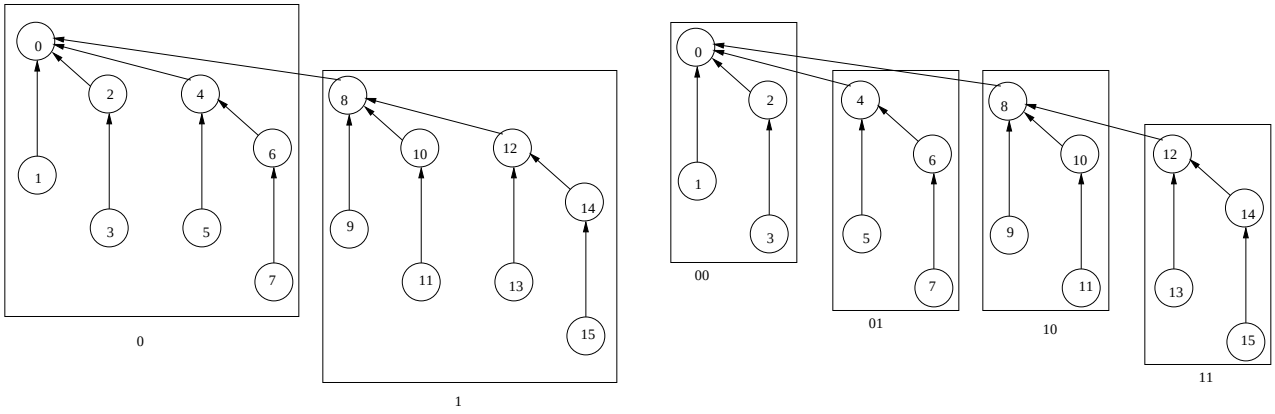


Figure 3.24: Two agglomerations and mappings of a 16-node binomial tree into a 2-node and 4-node tree respectively. The binary numbers below the rectangles are the most significant bits of the node labels of the nodes in that rectangle.

3.4.5 Analysis

The running time of the reduction algorithm depends upon the two parameters we defined earlier:

χ the amount of time to perform the binary operation (whether it is addition, multiplication, maximum, etc.), and

λ the time to transfer a number from one task to another through a channel.

Performing the sequential sum of n numbers takes $n-1$ operations. Since each task has $\lceil n/p \rceil$ numbers, and they all begin by concurrently performing the sum, the initial time to compute these sums is $\chi(\lceil n/p \rceil - 1)$. Once these partial sums are stored in each task, the parallel reduction begins. The parallel reduction requires $\lceil \log p \rceil$ steps. Each process must receive a value and then add it to its partial sum. That means that each step takes $(\lambda + \chi)$ time. Therefore the reduction takes $(\lambda + \chi)\lceil \log p \rceil$ time and the total time for the algorithm is

$$\chi(\lceil n/p \rceil - 1) + (\lambda + \chi)\lceil \log p \rceil$$

3.5 The Gathering Operation: The n-Body Problem

3.5.1 Introduction

There is a large class of problems in which a set of computations must be performed on all pairs of objects in a set of n objects. The most well-known example of this arises in physics and is known as an n-body problem. An n-body problem is one in which n particles or bodies of some kind have all possible pairwise interactions. In Newtonian physics, two particles exert gravitational forces on each other, no matter how far apart they are. The straightforward sequential algorithms that compute the forces, velocities, and positions of the particles have time complexity $\Theta(n^2)$ for a single iteration, as there are a total of $n(n-1)/2$ sets of unordered pairs. Although there are better sequential algorithms, parallelizing this sequential algorithm is a good way to introduce another important concept: the ***gather*** operation.

Let us assume we want to solve the classical n-body problem: given an initial state of a collection of n particles, with their respective masses, positions, and velocity vectors, compute their new positions and velocities as time progresses. It does not matter in our formulation of the problem whether it is a one-, two-, or three-dimensional space in which these particles exist. We do not even care what the formulas are.

3.5.2 Partitioning

We will assign one task to each particle. This is the finest partitioning of the problem. Each task is responsible for updating the state of the particle, i.e., its position and velocity in the next time step. This requires information about the velocities and positions of all other particles in the system.

3.5.3 Communication

Each task must obtain data from all other tasks in each iteration. A ***gather operation*** is a global communication that takes a dataset distributed among a collection of tasks and gathers it into a single task. This is different than a reduction. A reduction performs the composition of a binary reduction operation on all of the data, resulting in a single value in one task. A gather copies the data items from each task into an array of these items in a single task. An ***all-gather*** operation collects the data from all tasks and makes a copy of the entire dataset in each task.

The n-body problem requires an all-gather operation, because all tasks need the data from all other tasks at the start of each iteration. One obvious way to implement the all-gather is to put a pair of channels between each pair of tasks. Two channels per pair of tasks is needed because the channels are one-way only, so we would need a total of $n(n-1)$ channels for n tasks (i.e., $2 \cdot n(n-1)/2$ many channels.) This is not the best way. A better way is to base the implementation of the all-gather operation on a hypercube topology, introduced in Chapter 2. A modified version of the 4-dimensional hypercube is displayed in Figure 3.25 for convenience: each single edge with arrows on each end in the figure represents a pair of input and output channels between the incident nodes. We will assume for simplicity that n is a power of 2, i.e. $n = 2^k$, and k is the dimension of the hypercube.

The easiest way to understand the all-gather based on the hypercube is through the node labels. A k-dimensional hypercube has k bits in its node label. The idea is that there will be k iterations, one for each bit position. In the first iteration, all nodes whose labels are identical except for the most significant bit position will exchange data values. There are 2^{k-1} pairs of such nodes in every iteration. For example, in the first iteration, the following exchanges take place:

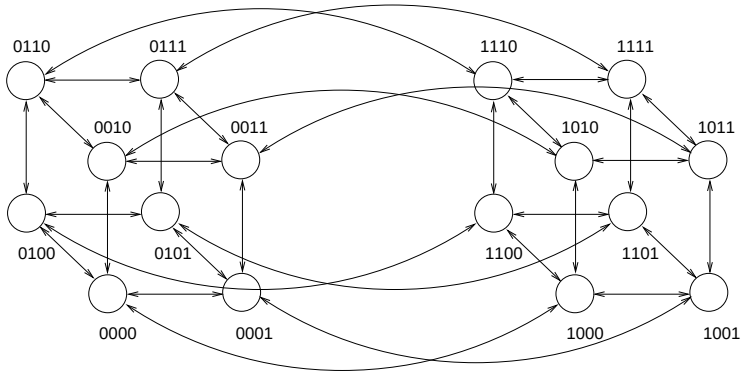


Figure 3.25: Four-dimensional hypercube, with node labels.

$$0000 \Leftrightarrow 1000$$
$$0001 \Leftrightarrow 1001$$
$$0010 \Leftrightarrow 1010$$
$$0011 \Leftrightarrow 1011$$
$$0100 \Leftrightarrow 1100$$
$$0101 \Leftrightarrow 1101$$
$$0110 \Leftrightarrow 1110$$
$$0111 \Leftrightarrow 1111$$

In the second iteration, all pairs of nodes whose labels are the same except for the second most significant bit position will exchange values. This takes place until all bit positions have been used.

Now we need to convince ourselves that when this procedure has finished, all nodes have copies of all of the data items. In fact, the data propagates from each node to all other nodes using the reverse edges of a binomial tree whose root is that node. That tree is illustrated for node 0 in Figure 3.26. For each of the 2^k nodes in the hypercube, there is a binomial tree rooted at that node. Each of these trees represents a sequence of data transfers that propagates that node's data item to the other nodes in the tree. After all iterations, then, each node's data is propagated through its tree to all other nodes. If you now wonder why there are no delays caused by a node needing to receive data from more than one node at a time, this is because, in any given iteration, a node is being sent data by only the node whose label differs from its label in a specific bit position. Obviously, bits have only two values, so there cannot be two different nodes whose labels are exactly the same as a third node's label except for a given bit position.

3.5.4 Agglomeration and Mapping

There will be many more particles n than processors p. The decision tree tells us that, since we have a static number of tasks, with a regular, structured communication pattern, with each task doing the same work, we should associate one task per processor and minimize communication. We therefore distribute n/p particles to each processor. The next step is to analyze the performance of this approach.

3.5.5 Analysis: Latency and Bandwidth

The all-gather communication in each iteration will take $\log p$ steps, but unlike the preceding problems, the size of the message will change in each step. In the first step, the messages have size n/p, in the second, size $2n/p$, in the third, $4n/p$, doubling each time. In short, in the k^{th} step the messages have size $2^{k-1}n/p$.

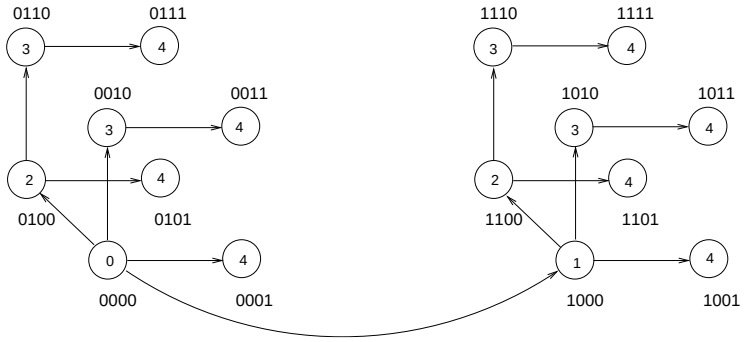


Figure 3.26: Propagation of node 0's data item along a reverse binomial tree rooted at 0. The numbers inside the nodes indicate in which iteration that node first gets node 0's data item. These numbers correspond to the bit position that was used in that step, with 0 being the leftmost bit.

Since there are $\log p$ steps, in the last step they have size $2^{(\log p)-1}n/p = (p/2)n/p = n/2$. In the previous analyses, we did not have to consider the message size in the performance analysis because all messages were the same size. In general, the amount of time that it takes a task to send a message has two components, the ***latency***, which is the time to initiate the transmission, and the ***transfer time***, which is the time spent sending the message through the channel. The longer the message, the longer the transfer time.

The latency will always be represented by λ. The transfer time is a function of the channel's ***bandwidth***, which we will denote by β. In general, the bandwidth of a channel is the number of data items that can be sent through it per unit time. It is often expressed in bytes per second, but it can also be expressed in messages per second or transactions per second. For our purposes, it will be thought of as the number of data items per unit time, whatever that unit may be. It follows that it takes $\lambda + d/\beta$ time to send a message containing d data items through a channel. In the k^{th}step of an all-gather operation, the communication time is $\lambda + (2^{k-1}n)/\beta p)$. Therefore, the total communication time of a single iteration of the algorithm is

$$\sum_{k=1}^{\log p}\left(\lambda + \frac{2^{k-1}n}{\beta p}\right) = \lambda \log p + \frac{n}{\beta p}\sum_{k=0}^{(\log p)-1} 2^k = \lambda \log p + \frac{n}{\beta p}\left(2^{\log p} - 1\right) = \lambda \log p + \frac{n\,(p-1)}{\beta p}$$

To compute the new state of a particle, it is necessary to perform a sequence of operations using that particle's state as well as the state of every other particle. Let us assume that it takes a single task χ time to compute the contribution to the new state of a single particle by a second particle. Then it takes a task $\chi(n/p)(n-1)$ time to update the states of its list of n/p particles. Combining the computation and communication times, we conclude that the total time for a single iteration of the algorithm is

$$\lambda \log p + n\,(p-1)\,/\beta p + \chi(n/p)(n-1)$$

3.6 Input Data

3.6.1 Introduction

We cannot ignore the practical matter of how the input data is initially distributed to all of the tasks that participate in a parallel algorithm. The n-body problem is a good problem to use for illustrating how this can be done. Although there may be some parallel computers that have parallel I/O systems capable of distributing data to each processor, we will take the conservative approach and assume that the I/O system is a traditional one, attached to a single processor. This is the more likely scenario you will encounter in practice.

With this in mind, the task/channel model needs to be modified. We will assume that a single task is responsible for reading in all of the data from a file server, and without loss of generality, we will assume it

is task 0. This task will read the set of initial states for all n particles in the system. We will assume now that an initial state consists of the position vector and velocity vector for the point. In two dimensions, this means that the initial state of a particle consists of 4 values, two for the position vector and two for the velocity vector: (p_x, p_y, v_x, v_y).

Let us now call task 0 the I/O task and let us associate two channels with it, an input channel and an output channel. Let us also denote the latency and bandwidth of I/O operations using these channels as λ_{io} and β_{io} respectively. In the preceding section, Section3.5.5, we concluded that the time to transfer n data elements across a channel is $\lambda + n/\beta$, where λ is the latency and β is the bandwidth of the channel. Then the time it will take the I/O task to read in the data set is $\lambda_{io} + 4n/\beta_{io}$.

3.6.2 Communication

The problem is how the I/O task that stores a data collection can distribute the individual data items from the collection to all of the other tasks. This is the reverse of a gather operation. A gather collects individual data items from all tasks into a data collection stored on a single task. The operation we need now is called a ***scatter*** operation; it scatters the data from one task to all other tasks. Remember that each of the p tasks needs to get n/p elements. One easy but poor method of distributing the data would be for task 0 to iteratively send n/p elements (meaning $(4n/p)$ values) to each task in turn. As there are $p-1$ other tasks, the total time would be

$$(p-1)(\lambda + 4n/(p\beta)) = \lambda(p-1) + 4n(p-1)/(p\beta)) \tag{3.3}$$

By now you might realize that, just as there was a divide-and-conquer solution to the gather operation, there is a divide-and-conquer solution to the scatter operation, and once again, it is based on a binomial tree, in fact, a ***reverse binomial tree***. Look at Figure 3.26 and imagine that instead of node 0 sending a single data value out along the branches, it splits the data set in half, keeps half for itself and sends the other half to node 1. Each of nodes 0 and 1 are the roots of reverse binomial trees, so in the next iteration, using this same idea, nodes 0 and 1 repeat this step: they split the part of the data set that they have in half, keep half of it for themselves and send the other half along the tree edge to the next node. Now, after these two steps, the data set has been quartered and 4 nodes each have a quarter. This procedure is repeated in each node. Clearly, after each iteration, the data set is halved and the number of nodes containing a piece of it has doubled. As the goal is for each node to have n/p data elements, this is repeated until all nodes in the binomial tree have that many items. How long will this take? We again assume for simplicity that $p = 2^m$ for some positive integer m. The data set is of size $4n$ initially. The total time for this algorithm, under this assumption, is

$$\left(\lambda + \frac{1}{2}(4n)/\beta\right) + \left(\lambda + \frac{1}{4}(4n)/\beta\right) + \left(\lambda + \frac{1}{8}(4n)/\beta\right) + \cdots + \left(\lambda + \frac{1}{p}(4n)/\beta\right)$$

Observing that $(1/p) = 1/(2^{\log p})$, the total time is

$$\begin{aligned}
\sum_{j=1}^{\log p} \left(\lambda + 4n/(2^j\beta)\right) &= \sum_{j=1}^{\log p} \lambda + \sum_{j=1}^{\log p} (4n/(2^j\beta) \\
&= \lambda \log p + (4n/\beta) \sum_{j=1}^{\log p} 1/2^j \\
&= \lambda \log p + (4n/\beta) \sum_{j=1}^{m} \left(\frac{1}{2}\right)^j
\end{aligned} \tag{3.4}$$

To simplify the summation term in 3.4, since $p = 2^m$, we multiply each term by $2^m/p$, getting

$$\begin{aligned}
\sum_{j=1}^{\log p} \left(\lambda + 4n/(2^j\beta)\right) &= \lambda \log p + (4n/\beta)\sum_{j=1}^{m} \frac{2^m}{p}\left(\frac{1}{2}\right)^j \\
&= \lambda \log p + (4n/p\beta)\sum_{j=1}^{m} 2^m \left(\frac{1}{2}\right)^j \\
&= \lambda \log p + (4n/p\beta)\sum_{j=1}^{m} 2^{m-j} \\
&= \lambda \log p + (4n/p\beta)\sum_{j=0}^{m-1} 2^{j} \qquad (3.5) \\
&= \lambda \log p + (4n/p\beta)(2^m - 1) \\
&= \lambda \log p + 4n(p-1)/p\beta) \qquad (3.6)
\end{aligned}$$

Quinn [2] makes an interesting and important observation at this point, one worth remembering. The naive, somewhat brute force method of scattering in which the I/O task scatters to all tasks (Eq. 3.3) has a running time of

$$\lambda(p-1) + 4n(p-1)/(p\beta))$$

whereas this method runs in time

$$\lambda \log p + 4n(p-1)/p\beta).$$

The only difference is the latency component. The total time spent transmitting data is the same. In the first method, the I/O task sends the data directly to the task that is supposed to have it, but in the second one, each chunk of data passes through multiple channels along the binomial tree path. This would suggest that the total transmission time would be greater, since each chunk is transmitted multiple times, but it isn't. Why? The answer lies in the fact that, in the task/channel model that we are using, the channels allow concurrent transmission of data; two channels can be transferring data at the same time.

Is this model reasonable? On commercial parallel computers, it is, because they are designed for this purpose. On commodity clusters in which computers are connected directly to network switches, it is also reasonable if the switches have sufficient internal bandwidth. However, if the target system is a collection of workstations connected by hubs or a different shared communication medium, it is not a valid assumption. (A hub shares the total bandwidth among all devices, while a switch provides a dedicated line at full bandwidth between every two devices transmitting to each other.) In fact, if the transmissions were sent in sequence, one after the other, without any parallelism, the number of them would be equal to the number of edges in the binomial tree. Since each node has a single incoming edge except the root, there are $p-1$ such edges and the total latency component of the scatter implemented using the reverse binomial tree would be the same as that of the sequential solution.

3.6.3 Analysis

The total expected time for the parallel n-body algorithm is the sum of the time to perform the input and scatter the data, plus the time to perform the computation some number of iterations, and gather the results and output them. We will assume the algorithm performs M iterations. Collecting the various components of the running time from the preceding sections, we have the following.

The time that the I/O task takes to read in the data is the same as the time to output it, so the total I/O time is

$$2(\lambda_{io} + 4n/\beta_{io})$$

The scattering operation at the beginning, and gathering at the end, take the same amount of time. The total time for them is

$$2\left(\lambda \log\, p + 4n(p-1)/p\beta)\right).$$

In each iteration of the algorithm, there is an all-gather of the particles' positions followed by an update of their state. If we account for the fact that positions are two values each, then the all-gather time is

$$\lambda \log p + 2n\left(p-1\right)/\beta p$$

and the computation time per iteration is

$$\chi(n/p)(n-1).$$

Under the assumption that it runs for M iterations, the total execution time of the parallel algorithm, assuming the number of processors is a power of 2 and n is divisible by p, is

$$\begin{aligned} 2(\lambda_{io} + 4n/\beta_{io}) \quad &+ \quad 2\left(\lambda \log\, p + 4n(p-1)/p\beta)\right) \\ &+ \quad M\left(\lambda \log p + 2n\left(p-1\right)/\beta p + \chi(n/p)(n-1)\right) \end{aligned}$$

3.7 Summary

This chapter has presented a few very important concepts that will be used in the remainder of these lecture notes. The task/channel model is a way to represent a parallel computation as a collection of concurrent processes that interact by exchanging messages through communication channels. Foster's design methodology gives us a systematic approach towards the design of parallel algorithms using the task/channel model. It also provides guiding principles for making those algorithms efficient. The performance goals of parallel algorithm design are often in conflict with one another and the process ultimately is one of optimization.

We got our first glimpse at two types of global, distributed communication operations, a gathering operation and a scattering one. In subsequent chapters we will see how to program these using *MPI*.

References

[1] Brian W. Kernighan and P. J. Plauger. *The Elements of Programming Style*. McGraw-Hill, Inc., New York, NY, USA, 2nd edition, 1982.

[2] M.J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Higher Education. McGraw-Hill Higher Education, 2004.

[3] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Parallel programming / scientific computing. ADDISON WESLEY Publishing Company Incorporated, 1995.

[4] A. Burns. Scheduling hard real-time systems: a review. *Software Engineering Journal*, 6(3):116–128, 1991.

[5] N. Fisher, J.H. Anderson, and S. Baruah. Task partitioning upon memory-constrained multiprocessors. In *Embedded and Real-Time Computing Systems and Applications, 2005. Proceedings. 11th IEEE International Conference on*, pages 416–421, 2005.

Subject Index