Chapter 2

Background

Mirror, mirror on the wall, who's the fairest of them all?

This chapter first presents two concepts, load balancing and scans, that are central to understanding Scan Directed Load Balancing, and then examines the class of computations for which Scan Directed Load Balancing is most effective and the class of parallel computers on which these computations run.

2.1 Load Balancing

The *Computational Load* of a program in a parallel computer is a measure of the amount of work to be done in executing the program or, equivalently, of the cost of executing the program. A parallel program consists of a set of *tasks* each of which must be executed on a single processor. Each task has its own computational load, and *load balancing* is the process of distributing tasks to processors in such a way that each processor has the same load. A successful load balance results in each processor running for the same length of time before completing its assigned tasks; this gives optimal processor utilization and better performance than would be obtained if the load were unbalanced.

As an example of a computation that can be load balanced, consider a diffusion problem on a two dimensional array of values. A task in this computation consists of setting one value in the array to the average of its neighbors. Since all tasks have approximately the same load, the load on each processor is proportional to the number of tasks assigned to it. Assuming that there are more tasks than processors, one way of load balancing this computation is to assign the same number of tasks to each processor.

2.1.1 Grain Size

Load balancing is not applicable to sequential computations; it is only on parallel computers that the problem of keeping processors busy arises. Moreover, this dissertation only concerns itself with *distributed memory* computers, computers in which each processor has its own local memory and communicates with other processors by passing messages over a *network*.

The *grain size* of a parallel computation is the amount of resources required by a typical task; likewise, the grain size of a parallel computer is the amount of resources available to a typical processor. In large grain parallel computations each task is referred to as a *process*. Individual processing elements keep queues of processes, and each process includes both code and data; except for interprocess communication and synchronization, each process executes entirely in its own address space, independently of other processes. In a small grain parallel computation each task is referred to as a *data point*. Individual processing elements keep arrays of data points, and the code for each data point is managed separately from the data; each data point executes within a common address space.

The diffusion computation described earlier is a small grain computation, since the amount of work to be done for each data point is small and it would be wasteful to dedicate an entire process to each data point.

If two computations do the same amount of work, the computations with the smaller grain size will have a larger number of tasks. Because each task does less work and because of the larger number of tasks, small grain computations are more frequently communication bound than large grain computations. If the small grain computation is executed on a small grain parallel computer, the high frequency of communication is exacerbated by the large number of processors which prevents processors from being fully connected; the restricted connectivity also imposes routing overhead on communication, and communication performance may suffer from collisions among messages sent by different processors. Minimizing communication costs is therefore a major concern in small grain parallel computations [80].

Because small grain computations do not have a separate address space for each task, thinking of a task as the computation that has to be performed on a unit of data is more natural than thinking of a task as an independent unit of code its own data. This *data parallel* [41] formulation of parallel computation has been used increasingly in recent years, in part because of the increasing availability of small grain parallel computers such as the Connection Machine [40]. The algorithms presented in this dissertation have so far only been applied to small grain computations (though there is every reason to believe they would be even more effective in load balancing large grain computations), so in what follows the term *data point* refers to both a unit of data with its attending storage, and the task that processes the data point.

2.1.2 Static and Dynamic Load Balancing

Sometimes the load of each data point can be determined in advance before the computation begins. If the load is known exactly, it is theoretically possible to distribute the load among processors optimally so that each processor does the same amount of work. Distributing load optimally is known to be NP-complete [36], since it is equivalent to the bin packing problem. The problem of assigning data points to processors is further constrained by the need to minimize communication distance; these constraints may however reduce the complexity of finding an optimal solution, and geometrical approaches can often be used to give an acceptable distribution at low cost.

In the diffusion example, each point has the same load, so assigning the same number of points to each processor balances the load, but unless the data is distributed hierarchically, accessing the value of neighbors for each data point can become so expensive in time that the benefits of load balancing are lost. In practice, specifying that the data should be distributed hierarchically simplifies the problem of finding an optimal distribution because it restricts the field of possible solutions.

Parallel computations in which the load of each data point is known exactly in advance are less common than those in which the load of each data point is not known in advance because it changes over time. As a simple approximation, data points can be classified at any point in time as either *active* or *inactive*. Active points are those that require processing, and inactive points are those that don't. If the diffusion example were modified so that on every iteration only points that change their value by more than a threshold are diffused on the next iterations, initially all points would be made active, and many would become inactive after the first iteration; as the computation continued, active points could become inactive or inactive points could be re-activated if a neighbor changed its value by a sufficiently large amount. Picking only the active data points and distributing the same number of active points to each processor gives a well balanced load if the load balancing is performed frequently compared to the average length of time that a data point is active.

If the load balancing algorithm picks a data distribution only once, before the computation begins, the algorithm is a *static* load balancing algorithm. If the load is balanced repeatedly, the algorithm is a *dynamic* load balancing algorithm. Static load balancing algorithms often much require information about a computation to wisely distribute the load, but can afford to take their time doing so, since they are executed only once. Another strategy for static load balancing is to use statistical properties of the computation in selecting an algorithm for distributing the load; this is true for cut and stack [85], for instance.

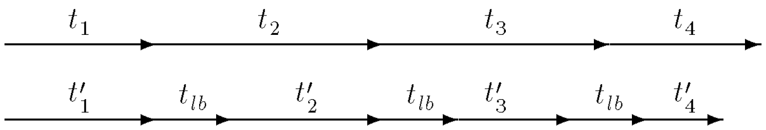
In contrast to static load balancing, dynamic load balancing algorithms are called repeatedly at run time and must be efficient; if the information on which balancing decisions are based was inaccurate, the error can be corrected the next time the load is balanced, so the information need not be highly accurate. The advantage of high speed is illustrated in Figure 2.1.

Since most static load balancing algorithms use detailed or statistical information about the computation, they are usually specific to the computation itself, whereas dynamic load balancing algorithms can usually be applied with few or no changes to a variety of computations. In general, dynamic load balancing is effective for any computation for which static load balancing is effective, since the information available initially can be used to balance the load once at the beginning of the computation. The converse does not usually hold: static load balancing algorithms are often ineffective on computations for which dynamic load balancing is effective, since the load of each data point may not be known until run time. Because most computations cannot be balanced statically and dynamic load balancing algorithms are more generally applicable than static load balancing algorithms, dynamic algorithms such as Scan Directed Load Balancing, the subject of this dissertation, are more interesting and useful than static load balancing algorithms.

Given a parallel computation divided into T successive phases, where phase$_i$ takes time t_i, and a dynamic load balancing algorithm that is run after every phase, takes time t_{lb}, and changes the running time of phase i to t'_i (where normally $t'_i < t_i$ because the load is more balanced after load balancing), the load balanced computation completes sooner than the unbalanced computation if

$$t_{lb}(T-1) + \sum_{i=1}^{T} t'_i < \sum_{i=1}^{T} t_i$$

A comparison with $T = 4$ of a run with and without load balancing might look like the following:



In the top timeline, each phase after the first takes longer than in the lower timeline because the load is unbalanced and some processors have more work to do before reaching the end of the phase. With load balancing, time must be spent balancing the load, but the net result can be a performance improvement.

Figure 2.1: Costs and Benefits of Dynamic Load Balancing

2.1.3 Load Sharing

A widely-researched variant of dynamic load balancing for large grain systems is *Load Sharing* [29, 90, 21]. Load sharing tries to make sure that each processor has at least one active process at all times; load balancing, in contrast, tries to make sure that each processor has the same number of active processes. In a system that uses load sharing, if the single active process on a processor becomes inactive, the processor must be able to quickly locate and begin executing another active process, otherwise its idle time will significantly slow down the computation.

A well known algorithm for dynamic load balancing is the Gradient Model Load Balancing Method developed by Lin and Keller [59]. In this algorithm processes diffuse along a mesh of processors according to the gradient of the load curve. Each processor determines the local value of the gradient by looking at the load of its neighbors. The algorithm is a variant of Load Sharing, since if the minimum load is sufficiently high, no balancing takes place.

Load Sharing can be effective on large grain computers with large grain processes because the load is balanced frequently compared to the length of time that a process remains active, and the cost of executing a process is high compared to the cost of acquiring a process from another processor. In small grain computations, active data points typically execute for about the same short length of time before they must synchronize, and if some processor has many active data points, any processor that only has a single active data point will be mostly idle until it is assigned more active points.

2.1.4 Constrained Load Balancing

In computers with few large grain fully connected processors, processors are often interchangeable in the sense that there is no advantage, other than equalizing the load, to executing the computation for a data point on one or on another processor, since each processor is connected directly to every other processor. More interesting, and more common with small grain parallel computers, is the case in which processors are structured into a network, so that each processor can directly communicate with at most a small subset of the processors in the system. Most load balancing algorithms have provisions for favoring particular processors when redistributing load. In particular, both the Gradient Model [59] and Contracting Within a Neighborhood [53, 78] are designed to keep communication costs low by reassigning data points to other processors that are "nearby."

Insuring that pairs of data points that need to exchange much information are assigned to a single processor or to pairs of processors that are directly connected minimizes the distance that information has to travel, and with it the time delay and the chance of collision. A data distribution for a given communication pattern is *local* in the extreme case, when all exchange of information is among pairs of data points that are in pairs of directly connected processors. If processors are arranged in a k dimensional *rectangular mesh* (hereafter simply referred to as "mesh"), and data points are arranged in a k dimensional *rectangular grid* ("grid") for some k, one way of achieving locality of communication is to divide the data *hierarchically*, using one coarse grid that has one cell for every processor in the mesh, and assigning all the data points from each cell of the coarse grid to the corresponding processor.

One major difference between a mesh of processors and a grid of data is that there is a processor at every intersection of a mesh, whereas there need not be a data point at every intersection of a grid. Computations that always have a data point at every grid intersection are called *dense*, those that don't are *sparse*. In sparse computations, e.g. in fluid flow or molecular dynamics simulations, data points may even move from one grid intersection to another.

If processors and data both have dimension k, partitioning the data hierarchically to insure locality of communication is a matter of placing boundaries (the "coarse grid") in the data in each of the k dimensions. For example if $k = 2$, boundaries can be vertical and horizontal lines in the grid of data points; a boundary separates the data that belongs in one processor row from the data in another processor row, and likewise for the columns. If $k = 3$, each boundary is a plane, and each plane of processors has a boundary plane separating its data from the data of other processors. This class of boundaries, generalized to k dimensions, is the basis for our approach to load balancing and will be discussed in detail in Chapter 3.

In k dimensions, each boundary is a $k-1$ dimensional hyperplane that partitions the data assigned to one hyperplane of processors from that assigned to another hyperplane of processors. Load balancing is achieved by shifting the hyperplanes within the data grid to reduce the maximum load on any processor.

Many existing algorithms for load balancing grids of data on meshes or hypercubes of processors achieve optimal load balance at the expense of giving up the locality of communication. Baden [5], and Berger and Bokhari [8] first introduced the Orthogonal Recursive Bisections (ORB) strategy for load balancing; more recent work on ORB has been done by, among others, Baden [6], Dragon and Gustafson [28], and Walker [89]. Orthogonal Recursive Bisection splits both the data grid and the processor mesh into two halves; the partition for the processor mesh is in the middle, so each half has the same number of processors, and the partition for the data grid is selected such that the subgrid on each side of the partition has exactly one half of the total load. This subdivision is repeated recursively in each half of the mesh. In two or more dimensions, each partition is taken orthogonal to the preceding one. Walker [89] has studied variations on this that split first those dimensions that are less likely to be unbalanced.

The result of Orthogonal Recursive Bisection is that each processor has its own rectangular subdomain of data, each with almost exactly the same load. The drawback is that, even if the communication is strictly between data points that are adjacent in the data grid, the data points may be assigned to pairs of processors that are an arbitrary distance apart. This is not crucial if the processors are at the vertices of a hypercube, since a hypercube of dimension k has a small ($O(k)$) diameter and offers high performance for non-local communication as long as the *average* distance is small. A processor mesh of dimension k with p^k processors has a large diameter ($O(p)$), and communication over long distances can slow down a computation substantially even if the average distance is small. A variant of Orthogonal Recursive Bisection for two dimensions developed by Saltz and others [79] assigns each row of data to a single row of processors and optimizes the decomposition of data within each row. Such "jagged" decompositions have greater locality than Orthogonal Recursive Bisection, but some nonlocal communication is still required.

An alternative approach called Orthogonal, Monotonic, and Surjective distribution (OMS), studied in this dissertation, allows the load balance to be suboptimal but insures that all communication remains strictly local, so each data point's neighbors are stored at most one link away. With this approach, as with load sharing, the goal is explicitly an *improvement* in the load balance, as opposed to the almost exactly balanced load achieved by Orthogonal Recursive Bisection. Because communication is strictly local, the communication overhead is less and the computation as a whole may be faster than with perfect load balance and non-local communication, even though the load is imperfectly balanced and the processing of instructions may therefore take longer to complete. The OMS approach to load balancing is especially good for machines that lack a hardware router, but is useful and effective for any computation that spends a significant amount of time doing communication. Much of the work of this thesis is based on the OMS approach.

A variant of OMS called *Rectilinear Partitioning* has been developed independently by Nicol [67], and used in simulated load balancing of one, two, and three dimensional systems. For balancing a load associated with n data points on a one-dimensional array of p processors, the load balancing algorithm finds the *optimal* distribution of load in time $O(n + (p \log n)^2)$; in two dimensions, a *locally* optimal load balance is achieved by repeatedly computing the optimal balance in each dimension; the work required to achieve this balance is reportedly $O(p(n + p \log n))$, though this result has not been proven. The load balance is locally optimal because it cannot be improved by changing either the row or the column decomposition, though changing both dimensions at once could improve the balance. Optimal load balancing in three dimensions is shown to be NP complete by showing that it can be used to compute solutions to the monotonic 3SAT problem [36]. The algorithm is unfortunately not used on any real-world problems; simulations for machine sizes of up to 64×64 processors on particular grids used in fluid dynamics computations show that Rectilinear Partitioning gives better processor utilization than Orthogonal Recursive Bisection if the cost of global communication is high.

Scan Directed Load Balancing, by comparison, does not try as hard to achieve optimal load balance; the possibly suboptimal load balance is computed quickly enough that the algorithm is useful for real problem. The asymptotic performance of Scan Directed Load Balancing is analyzed in Section 3.5.4.

2.1.5 Computational Load for Subsets of Data

For dynamic load balancing the cost of measuring the load directly affects run-time and therefore simple formulas are best; most algorithms for dynamic load balancing use the number of active data points (or processes) as the load of each processor, ignoring the storage and other resources required for the computation. The assumption is then that each active data point has a fixed load (arbitrarily set to 1), and each inactive data point has a load of 0, since it does not require computational resources.

A more elaborate scheme assigns a given cost to each active data point and a lesser cost to each inactive data point. In this way each data point has a load that reflects not only the length of time required to process its instructions, but also the amount of storage required. The latter is an important concern for small grain parallel computers, in which the memory in each processor is substantially less than would be required to store all the inactive data points from the entire machine. If the load of a data point includes a value for the storage required by the data point, improving the load balance not only decreases the computation time, but does so without assigning too many data points to any one processor; this scheme is described in detail in Section 3.4.

OMS decompositions preserve orthogonal relationships between data points, so for example if one point is directly to the North of another in a two dimensional data grid, it is always located either in the same processor or in a processor that is directly to the North. Scan Directed Load Balancing changes the data distribution by shifting planes of data as units, so a data point is only shifted if all the other data points that lie in all directions orthogonal to the shift are shifted with it; if the original distribution is OMS, the final distribution is also OMS.

In one dimension, there are no directions orthogonal to the shift, and data points can be shifted individually. In two dimensions, whenever any one data point is shifted to the right or left, its entire column is shifted with it. Whenever any one data point is shifted up or down, its entire row is shifted with it. In three dimensions, planes of data are shifted as units, and in k dimensions, hyperplanes of dimension $k-1$ are shifted as units. The following paragraph discusses the case $k = 3$, in which planes of data are shifted.

To determine whether a plane of data should be shifted, Scan Directed Load Balancing computes a single load for each plane, row, or column of processors. One possible measure of the load of a plane of processors is the count of active points in it. This count is, however, a poor measure of the plane's load, since a plane with many active data points, but only one active point in each processor intersected by the plane, has a lower load than a plane with just a few active points, all of which are in a single processor. A more

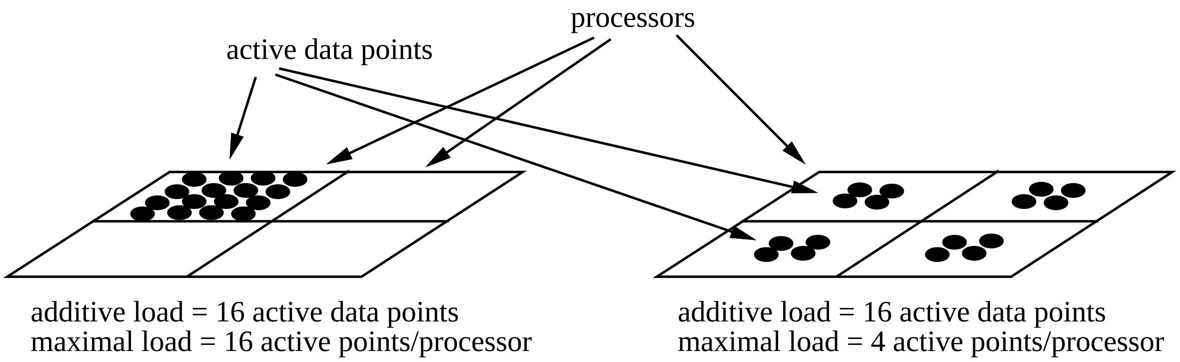


Figure 2.2: Additive and Maximal Loads for Planes of Data

appropriate measure of the load of a data plane is the maximum number of active points belonging to the plane within any one processor. This is illustrated in Figure 2.2, in which the two planes shown (each of which is a slice from a cube of processors and data) have a load of 16 active points each if the first method is used; if the second method is used, the left plane still has a load of 16 active points but the right plane has a load of only 4 active points. This strategy for computing load gives a better indication of the time that will be required to process the different data points.

2.2 Scans

A scan is an operation which computes all the applications of an operator to all prefixes of a vector. For instance, a scan with the operator *plus*, which has identity 0, applied to the vector `[3 1 -2 9]` computes as its result the vector `[0 0+3 0+3+1 0+3+1-2]` =`[0 3 4 2]`. A scan with *plus* as the operator is a *plus scan*.

Another name for a scan is *prefix computation*, and the parallel implementation of a scan is a *parallel prefix computation*. Ladner and Fischer [57, 58] show that a parallel prefix computation on an associative operator can be performed by boolean circuits of depth[1] $\lceil \log_2 n \rceil$ and size $O(n)$. These results

[1]The *depth* of a circuit is the maximum number of gates between any input and any output, and is proportional to the time required to compute the output; the *size* of a

are analyzed in greater detail by Meijer and Akl [65] for parallel prefix sums. Historically, the word *scan* was first used in APL to denote what became known as *prefix computation* in Computer Science Theory.

An APL scan computes a function that is slightly different from the one defined above: the same scan as above, applied to the same vector, would compute the vector `[3 3+1 3+1-2 3+1-2+9]` =`[3 4 2 11]`. Both forms of the scan are useful for different computations, but note that the result of an APL scan can be obtained by adding the original vector to the result of the scan introduced above, which can in turn be obtained by shifting the result of the APL scan one place to the right and using the identity as the leftmost element[2]. Since, in a parallel implementation of a scan, shifting involves communication among processors and is likely to be slower than addition of local values, which doesn't, the scan described above has been used throughout this thesis in place of the APL scan.

In his dissertation [12, 13], Blelloch asserts that APL-type scans form a useful class of primitive parallel operations, in part because of the linear size and the logarithmic speed of the circuits needed to compute them. Blelloch also gives an algorithm for load balancing that has three steps:

1. enumerate the m active points, which are a subset of the elements of a vector of size $n \geq m$. This assigns a distinct consecutive integer, the *destination position*, to each of the active points.

2. place each active point into the result vector (of length m) at the destination position.

3. assign m/p active points to each of the p processors using a permutation.

Enumerating the elements is done by a plus-scan of a vector F, where if A is the vector to be balanced, $|F| = |A|$ and if an element of A is active, the corresponding element of F has the value 1, and the value 0 otherwise. The result of the scan is an assignment of consecutive integers to the active

circuit is the number of gates it contains, and is proportional to the circuit's cost.

[2]For scans with invertible operations such as *plus*, the result of the original scan could also be obtained by subtracting the argument from the APL result; if the operation is not invertible, a shift is required

elements of A; these integers are then used in permuting the elements to assign them to the various processors.

Although this algorithm and Scan Directed Load Balancing are closely related, the above load balancing algorithm does not work with multi dimensional vectors, requires a permutation operation which is potentially slow, makes no mention of the inactive elements of the vector, and does not move inactive data points. This dissertation addresses each of these issues in depth; in addition, this dissertation offers two case studies where Scan Directed Load Balancing significantly improves performance.

The use of scans on multi-dimensional arrays of data is perhaps the most distinctive feature of Scan Directed Load Balancing. In the Scan Directed Load Balancing algorithm, scans are used as a primitive that both computes the state of the system as a whole and communicates the relevant state to each processors. Other algorithms for load balancing use explicit message passing or (less commonly) shared memory to determine the load distribution. On many systems, especially SIMD computers, these mechanisms are unavailable, and simulation of one mechanism by another can be expensive. Scans, on the other hand, are mathematically defined operations that can be implemented efficiently on a variety of systems, and provide an efficient, machine independent mechanism for communication. A computer system that provides hardware supported scan primitives is particularly appropriate for Scan Directed Load Balancing, but even if a system fails to provide hardware support for scans, scans can be implemented efficiently enough that Scan Directed Load Balancing can effectively be used to balance many different computations.

Portability is only one reason for basing any algorithm, and in particular Scan Directed Load Balancing, on scans. The clear semantics and the synchronous nature of scans simplifies analysis of any algorithm based on them. The analysis of Scan Directed Load Balancing relies on both properties to show that particular assertions hold about the data distribution after the load balancing has completed. The scans in Scan Directed Load Balancing are used for computation and communication simultaneously, which is efficient in terms of both concepts and execution time.

2.3 Continuum Computations

2.3.1 Definition and Examples

There is a large class of computations that compute local functions on grids of data. To quote Scott Baden [6],

> An important class of scientific calculations operate on a mesh. Furthermore, they are spatially localized; they spend most of their time computing influences among nearby mesh boxes, and employ relatively inexpensive calculations to couple the distant boxes.

The computations described by Baden include finite difference calculations that use a fixed rectangular mesh or adaptive mesh refinement, finite element methods, and ray tracing for computer graphics. Specific applications include image processing [11], molecular modeling, molecular and fluid dynamics, and genome matching. The common characteristic of these computations is that they arrange data spatially and that, during most of the computation, new data values are being calculated as a function of neighboring data values. Numerical Weather Prediction programs, for instance, have three dimensional grids of data, updated on each time step of the computation [86], with each cell only communicating physical quantities to neighboring cells. Another example of computations on fixed meshes are fluid dynamics computations, which have two or three dimensional grids of data.

The term "continuum computation" is taken from Scherson and Corbett [80].

Definition 1 *A* Continuum Computation *is a computation on a data grid in which most of the effort goes to updating data points from values that are near in the data grid.*

Most simulations of physical systems are continuum computations, with the physical system mapped onto an orthogonal k dimensional grid. An efficient algorithm developed by Campbell, Carmona, and Walker for plasma simulations by the particle-in-cell method [89, 19] decomposes both the particle and the field calculations hierarchically to minimize communication across processor boundaries, but still finds that communication is a major component of the cost of the computation.

Many programs that simulate fluid flow are also continuum computations. Baden [6] simulates an incompressible inviscid fluid by a vortex method, and uses it as an example for his GenMP virtual machine. Boghosian [15] has surveyed lattice gas automata, a class of algorithms for continuum computations, for solving equations modeled by partial differential equations. Boyle and Harmer [17] implement an algorithm developed by Garbey and Levine [33] which uses a cellular automaton to compute the characteristics of the solution to a hyperbolic PDE that solves a fluid dynamics problem. To quote Boyle and Harmer,

> A cellular automaton consists of many identical cells, each having simple, locally determined behavior.
>
> ...
>
> The update rule is said to be "local" because it depends only on the state information at the cell being updated and that of some set of its neighboring cells.

Since the computation involves nothing but updating the state of each cell from local and neighboring data, the computation is a continuum computation. This holds for all cellular automata and systolic algorithms; many problems that can be solved by parallel algorithms can therefore be cast as continuum computations.

A Continuum Computation executed on a parallel computer may have substantial nonlocal communication, if the data is partitioned in such a way that neighboring cells are assigned to non-neighboring processors. The OMS data decomposition developed in this dissertation insures that all communication among neighboring grid points is local, which is especially beneficial if data is *dense* and each grid point has an associated data point.

Interesting continuum computations are Variable Conductance Diffusion and the reduction of FFP. Variable Conductance Diffusion is a fairly typical example of a diffusion algorithm, and any load balancing algorithm that is successful for it should be applicable to most other versions of diffusion and to many other computations. The reduction of FFP is a highly dynamic computation, where data points are often active for very short times; in the absence of load balancing, very substantial load imbalances can occur.

2.3.2 Variable Conductance Diffusion

Diffusion algorithms can be seen as cellular automata in which each cell sets its value to a weighted average of the values of its neighbors; diffusion algorithms differ in the size of neighborhood taken for averaging and in the weighting assigned to each cell. Computations, such as diffusion, that place a weighted sum of a fixed pattern (a *stencil*) of neighbors of each element of a two dimensional array into another two dimensional array are are sufficiently common that an entire compiler [18] has been developed to optimize them. In a diffusion algorithm, the distance over which data must be fetched is fixed and small, so diffusion algorithms are good examples of continuum computations.

One common application of diffusion algorithms is in image processing, where diffusion is used for blurring the image to eliminate speckle noise. Successive steps assimilate each pixel of the image to the average of its neighbors, iteratively removing the high spatial frequencies from the image. The result is that large features remain visible and small features, and in particular individual pixels that are corrupted by noise, are smoothed out. The drawback of diffusion is that sharp edges are also blurred, and the image loses sharpness. Variable Conductance Diffusion [72] avoids blurring sharp edges by weighting the update of each pixel by the inverse of the gradient of the image at the pixel. The areas of the image near a sharp edge have large gradients, so pixels in those areas do not change their values by much and the edges remain sharp; pixels in other areas are near small gradients, so they adjust their values to match their neighbors' and any noise is eliminated.

Cromartie and Pizer [24] have used Variable Conductance Diffusion to make medical images clearer and easier to interpret. A parallel implementation of Variable Conductance Diffusion using OMS decomposition partitions the image into as many rectangular tiles as there are processors. On each diffusion step, processors exchange the values of border pixels so the subsequent averaging operation and gradient computation can be done locally without communication. Since communication distances are small because the decomposition is OMS, most of the time is spent computing the gradient and the weighted average for each pixel; if communication were slow because of a large average distance between processors that hold neighboring tiles, most of the time would be spent communicating border values.

In Variable Conductance Diffusion, values of pixels near a sharp edge change little, as do values of pixels that already match the average of their neighbors. There are classes of images for which only few pixels modify their value; for these images, Scan Directed Load Balancing can be very effective in speeding up the computation of Variable Conductance Diffusion.

2.3.3 Reduction of FFP

A functional programming language is a language with no assignments and no side effects. Each program in a functional language applies one function to the program's input; each function computes a value, possibly structured, which is its only and entire result; the whole program therefore returns a single structured value.

The constituent parts of a program in a functional language are themselves programs in the same language, combined into a single program by functional forms. Functional programming languages are generally considered to be higher level than conventional imperative languages, and therefore better for producing maintainable, and reusable software [49, 76]. One major advantage of programming in a functional language is that, because of the lack of side effects, each function can execute as soon as its arguments are available, and more than one function can execute in parallel without changing the value of the result [22]. It is easy to tell for each function exactly which data it depends on, so scheduling parallel invocation of different functions is much simpler and safer than if the processes were to communicate after they were created. Functional languages have no primitives for communication; communication is achieved by transferring data within individual processes, then starting new processes on the moved data. Even Sabot's Paralation System [77], which can be integrated into languages that are not functional, prevents parallel tasks from communicating directly with one another. This contributes to both program clarity and efficiency, and makes deadlock impossible, without unduly restricting the range of legal communication operations that a program is allowed to use.

The FFP functional programming language was first described by John Backus in his Turing Award Lecture [4]. It is a language with a very simple syntax, reminiscent of LISP [64], and in which each function only takes a single, possibly structured, argument. The language provides powerful operators and *functional forms*, ways of joining operators to build more complex functional expressions. The *composition* functional form, for instance, composes functions. A unique feature of FFP is *metacomposition*, a construct that allows a program to easily manipulate its own code. Metacomposition allows a program to build its own control structures to achieve the effects of arbitrary recursion or iteration.

functional form	usage	argument	result
apply to all	<aa f>	$< v_1, \ldots, v_n >$	$< f(v_1), \ldots, f(v_n) >$
construct	<cr $f_1 \ldots f_n$>	v	$< f_1(v), \ldots, f_n(v) >$
insert	<ins $\oplus$>	$< v_1, \ldots, v_n >$	$v_1 \oplus \ldots \oplus v_n$

Table 2.1: Functional Forms for vector processing in FFP

FFP is a vector-oriented language, and has several functional forms, similar to the corresponding operators in APL [52], for operating on entire vectors at once. The *apply to all* functional form applies its expression to each element of its argument vector, and returns the vector of the results. The *construct* functional form applies each of its expressions to its argument, and returns a vector of results. The *insert* functional form repeatedly applies an expression to pairs formed by elements of a vector or intermediate results, until the final result is computed. These functional forms are summarized in Table 2.1.

The process of applying a functional program to its argument and computing the result is known as *reduction*. One strategy for reducing a function to its result is *graph* reduction [73], whereby a function keeps pointers to each of its subexpressions. Once a subexpression reduces to its value, the subexpression's location in memory is overwritten with the computed value, so further invocations do not need to recompute the value. Though this implementation uses side effects for efficiency, the mechanism for communication between processes is simple because only one process ever writes to any one memory location. Many machines supporting parallel graph reduction have been designed [38, 74, 91] and built in experimental settings.

Graph reduction is efficient because it evaluates common subexpressions only once. Unfortunately, accessing an arbitrary memory location through a pointer on a parallel machine, as required for graph reduction, can be an expensive operation [80], requiring either expensive hardware or a long time to complete. Hudak and Mohr's description [48] of their implementation of graph reduction on a parallel computer noted that

> Communication time accounts for most of the overhead in our current implementation ...
>
> Although faster architectures can help, we feel that preserving locality of reference is the crucial line for future research.

As an alternative to sharing of subgraphs in graph reduction, the subgraphs could be kept local to the space of each task that requires it and copied whenever a duplicate is needed by a newly created task. As Hudak and Mohr point out,

> [Duplicating shared subtrees] may be problematic, since programs with a high degree of concurrency tend to exhibit a lot of sharing, thus leading to large program graphs.

Copying is therefore not without its perils, first among them the possibility of running out of memory space. However, memory size can increase linearly with the number of processors, for any given technology; on the other hand, as machines grow and the number of processors increases, the cost of accessing arbitrarily distant memory locations is bound to increase faster than linearly, since the cost of networks for constant time access is at least quadratic in the number of processors, and other networks give slower access as the number of processors increases. In other words, scalability considerations favor copying over sharing for parallel computations.

Gyula Magó has designed [61, 62] a parallel architecture for reducing FFP, called the FFP Machine. The FFP Machine has no address space and favors copying over sharing.

The FFP Machine has a linear array of processors connected by a communication network capable of several different combining operations, including scans. The network is partitionable: any contiguous sequence of processing elements may be made into a virtual processor with its own linear array of processing elements and its own virtual network. All virtual processors may operate in parallel without interference, and communication only happens within a virtual processor. If more than one process needs to share a data item, as is true for instance for all the subexpressions of a `construct` functional form (see Table 2.1), the item is copied next to the operator that will be applied to it, then the processing elements and the network are partitioned so each `(operator, data)` pair can be reduced in parallel.

```
(atom <a b c>)                        F
(plus <2 3>)                          5
(<aa plus> <<1 2> <3 4> <5 6>>)       <3 7 11>
(<cr atom null> x)                    <T F>
(<ins times> <1 2 3 4>)               24
(primes 20)                           <2 3 5 7 11 13 17 19>
```

Figure 2.3: Reducible FFP expressions and their reduced values

A juxtaposition of an operator and an operand within parentheses in the FFP machine is a *reducible expression* (*redex*) if it is innermost; as soon as an expression becomes reducible, a virtual processor is allocated for it and reduction begins. Some reducible expressions are shown in Figure 2.3. Each redex can operate independently of all others because it has all the data it needs to compute a result. The reduction from the redex on the left to the result shown on the right may involve many steps and the formation of many intermediate virtual machines. The last example, for instance, computes a vector of prime numbers. No communication with virtual machines outside the redex is ever required, since all communication takes place within redexes.

The symbols that make up an FFP program are stored in linear order along the linear array of processing elements that make up the FFP Machine, one symbol to a cell, where syntactic markers such as parentheses and brackets are also symbols.

The only operation supported by the communication network is plus-scans, and the network is designed to be partitionable so independent scans can be computed simultaneously for different redexes. Allocating a virtual machine to a new reducible expression is designed to be fast operation with low overhead because all cells in a redexes are contiguous; the operation that is expensive is maintaining the linear order of symbols within the processors while copying and deleting symbols. Unless the linear order is maintained, creating virtual processors becomes an expensive operation; copying and deleting symbols with the linear order constraint, on the other hand, requires some mechanism for computing how far to shift data, and for shifting data between processors. The shifting itself can be fast, however, since it is only among neighboring processors and involves no routing.

The FFP interpreter described in Chapter 5 employs the FFP Machine model of reduction and is equivalent to a simulator for a particular version of the FFP Machine. The interpreter's communication network supports plus-scans, which are global operations, but maintaining the total ordering of symbols by shifting them among processors is the expensive operation, so this interpreter is a continuum computation.

2.3.4 Data Distribution

Many continuum computations have a uniform grid of cells with a data point at every intersection of the grid; load imbalance is caused by the different loads of different points. This is the case for Variable Conductance Diffusion, where the grid is laid along the scan lines and columns of the image and there is a pixel for each intersection of a row and a column. After the first few iterations, most pixels reach a value that is close (differing by less than some value ϵ) to the average of their neighbors and become quiescent, that is, they no longer need processing. The remaining active points are often not evenly distributed among processors, and load balancing becomes necessary. However, balancing the load by moving only the active points is inappropriate, since adjacent points that are not active are still used in the computation for active points; inactive points should therefore be shifted along with active points.

In the above case storage is *dense*, because there is a data point at every point of a uniform grid; when there are many more grid positions than data points, storage is sparse. A prototypical example of a sparse computation is John Conway's *game of life* [34, 35]. In the game of life, it is enough to store which points are "alive", and all other grid points need not store any information. In a large grid, at any time most grid positions correspond to no live point and have nothing to store. Simulations with discrete particles resemble the game of life in having grid points that correspond to no data, though they may differ in the speed with which data points may move and the distance over which they may interact.

All active data points require storage to record the values associated with the particle. A sparse data distribution is one in which only active data points are represented in the storage of the computer, a dense data distribution is one in which all grid positions are represented in the storage of the computer.

The important difference between dense and sparse data distributions is that with a dense distribution, assigning all inactive points to a single processing element can be infeasible because of limitations on per-processor memory. If load balancing of a dense distribution only considers active points in deciding which portions of a data grid are assigned to each processor, there is always the possibility that too many inactive points will be assigned to a single processor. With a sparse data distribution, inactive data points are not represented in storage and play no role in the computation or its cost.

The reduction of FFP, described above, is a computation that has sparse data distribution, since the "empty cells" of an FFP Machine require no storage space and can, for all practical purposes, be ignored; all other cells of an FFP Machine require computation, so all storage is associated with active points. Balancing the load for this computation effectively balances the storage required of any processing element.

2.4 Mesh Connected Computers

There is a staggering variety of parallel computers and parallel computer designs. Most sequential computers, and even computers with a small degree of parallelism, are designed to execute the same von Neumann programming languages and have comparable performance for comparable operations. On the other hand massively parallel computers, with thousands of processing elements, often vary dramatically in the time required to execute the simplest parallel operation. The speeds and capabilities of individual processors are usually within a factor of 10 of each other for similar technologies, but available interconnection networks span orders of magnitude in cost and performance, and vary dramatically in the time required for completing individual operations such as permutations and scans.

There are several parallel programming languages that ignore the particular interconnection network of the machine [40, 13, 83, 20]. Though the main goal of such an approach is portability, these approaches are often based on the PRAM model [47] in which all communication is through a common shared memory that can be accessed at unit cost. Recent work on a parallelizable version of Fortran [66, 30] allow the expression of local operations, but also provide arbitrary permutations which are only fast on machines that support the PRAM model. Implementing the PRAM model on a network with limited interconnectivity often leads to slow programs, and the resulting inefficiency negates much of the potential of parallel machines [77, 48].

A programming model that is often used as an alternative to PRAM is message passing. In *message passing* [46, 51, 81, 2], also known as the *distributed memory* model, accesses by each processor to local memory are inexpensive, but communication with other processors remains uniformly expensive. The overall efficiency of the program is greater than with the global memory model, because the programmer need not express local memory accesses with the same expensive mechanism used for communication operations, but there is no distinction between processors that are "near" each other and processors that aren't, so an arbitrary permutation is considered to be as expensive as an exchange between neighboring processors. While message passing offers portability because it doesn't force the programmer to be aware of the network structure, in practice efficiency considerations[3] dictate that the programs be customized to the particular interconnection network.

Both the PRAM model and Message Passing have been implemented on binary hypercube connected computers such as the Connection Machine [40], the Cosmic Cube [31], and the Intel iPSC [71]. Binary hypercubes are attractive because they provide a fast nonlocal interconnect at a reasonable cost, and are more scalable than complete p to p crossbar switches among p processors. Unfortunately binary hypercube interconnection networks are not arbitrarily scalable because each node has $\lceil \log p \rceil$ connections and because some of the connections are necessarily long and therefore slow, power-hungry, or both. The cube connected cycles topology [75] solves the problem of the variable number of connections per processor, but not the problem of long wires.

Many parallel computers, beginning with the Illiac IV [7], have processors connected in a one, two, or three dimensional mesh. Many recent parallel computers have $p = 2^k$ processors connected in a binary hypercube of dimension k. A binary hypercube of a given dimension k is identical to a mesh of dimension k with $p = 2^k$ processors; meshes, however, may have more than $p = 2^k$ processors, whereas in a binary hypercube the number of processors

[3] After all, speed is the main reason for using parallel computers.

is always determined by the dimension k. In a hypercube, each processor has $\log p$ connections to other processors, whereas in a mesh the number of connections on each processor is k and independent of the number of processors. The additional connections of a hypercube network allow it to have diameter $\lceil \log p \rceil$, whereas an arbitrary mesh has diameter $\sqrt[k]{p}$, but the long wires that reduce the diameter of a hypercube network are expensive, and as Dally [26] shows, two and three dimensional meshes are often more cost effective for generalized communication.

The time required for communication on mesh connected computers has been analyzed by Scherson and Corbett [80] in terms of the average distance of communication in a particular algorithm: locality of communication in algorithms that have no sequential component gives linear speedup on both meshes and hypercubes. They conclude that:

> An interesting result of this paper is that structures with lower connectivity than that of the binary hypercubes may have performance as measured by speedup comparable to that of the binary hypercube. This is a significant result since it shows that high performance multiprocessor systems can be constructed at a cost lower than that required by binary hypercubes.

If the dimension k' of the data grid differs from the dimension k of the processor mesh, known techniques can be used to map the data grid to the processor mesh. Memory virtualization techniques [32] can map higher dimensional data grids to lower dimensional meshes, and grid embedding strategies [9, 45] can be used to place lower dimensional grids in higher dimensional meshes and still preserve locality. Tao and Ma [84] present and analyze embeddings of meshes of size $p^{k'}$ in meshes of dimensions $k \neq k'$ and with the same number of nodes. The mesh of dimension k can simulate the mesh of dimension k' "optimally", that is, in constant time if $k' < k$ and k' divides k, and in time $p^{(k'-k)/k}$ if $k' > k$. Give these embeddings, data of dimension $k' < k$ can be allocated to the processors directly, and data of dimension $k' > k$ can be structured into a mesh of dimension k' which is then simulated by the mesh of dimension k with a predictable slowdown in communication time.

Some of these results are recent, but the use of meshes as interconnection networks for both special purpose and general purpose parallel computers is not. General purpose mesh connected computers that have actually been built include the Illiac IV [7] at the University of Illinois, the Distributed Array Processor (DAP) [68, 69] at ICL, the Massively Parallel Processor (MPP) [37], the iWARP system [16], the Maspar MP-1 [63], and to some extent Thinking Machine's CM-2 [60]. Special purpose mesh connected computers abound; they include one-dimensional parallel processors for string matching [92], two-dimensional mesh connected parallel processors that are especially well-suited for image processing [3], and other systolic arrays [54, 55].

Of these mesh-connected machines, of particular importance for this dissertation is the Maspar MP-1, on which Scan Directed Load Balancing has been implemented. To name just some of its the features, the MP-1 provides both a general purpose router with small diameter and relatively low bandwidth, and a two dimensional toroidal mesh connecting each processor with each of its four neighbors in the plane[4]. The MP-1 is a Single-Instruction, Multiple-Data stream (SIMD) computer, which means that a single master processor controls the flow of instructions; each of the slave processing elements (no fewer than 1024 and no more than 16384) executes the same instruction in parallel, unless it is de-selected. De-selected processing elements sit idle until they are re-selected by a special instruction. The MP-1's router was not used for either Variable Conductance Diffusion or the reduction of FFP, the two experiments described in this dissertation, since the mesh has a much higher bandwidth than the router: both of the test programs for Scan Directed Load Balancing only require local communication and scans, and the scans are implemented using the mesh.

This chapter has introduced load balancing, has summarily presented some existing load balancing algorithms, and has introduced concepts, such as scans, that will be used in presenting Scan Directed Load Balancing. The latter half of the chapter has described the class of computations and of computers to which Scan Directed Load Balancing is best suited. Chapter 3 uses all these concepts to describe and analyze in detail the algorithm of Scan Directed Load Balancing. The following two chapters do the same for two computations on which Scan Directed Load Balancing has been used.

[4]The MP-1 also provides diagonal connections among processors: these are useful in practice, but are do not significantly affect the material presented in this dissertation