Execution of Compute-Intensive Applications into Parallel Machines

Catherine Houstis[*] *Sarantos Kapidakis*[†] *Evangelos P. Markatos*[‡]
Erol Gelenbe[§]

September 17, 1996

Abstract

Scheduling and load balancing of applications on distributed or shared memory machine architectures can be executed by optimizing algorithms in various levels of the architecture. We are viewing four different levels, namely, the application layer, the compiler layer, the run time layer, and the operating system layer. The approach to scheduling and load balancing ranges from very specialized and directly dependent on the application, in the application layer, to a more general approach taken by the operating system layer. In the application layer, the application's computation is decomposed and evenly assigned to the processors, while communication and synchronization are minimized. In addition, specific knowledge about the application is taken into account to select the approach to problem solution. In the compiler layer, the application code is automatically decomposed by the compiler, most of the work being concentrated in the parallelization of language constructs. In the run time layer, the results of the application and the compiler layer are implemented. Finally, in the operating system layer, a fair allocation of the processors of the parallel machine is allocated to competing applications.

[*]Institute of Computer Science, FORTH, Heraklion, Crete, Greece.
[†]Institute of Computer Science, FORTH, Heraklion, Crete, Greece.
[‡]Institute of Computer Science, FORTH, Heraklion, Crete, Greece. C. Houstis, S. Kapidakis, and E.P. Markatos are also affiliated with the University of Crete.
[§]Computer Science at EHEI, University of Paris V, France.

1 Introduction

The problem of balancing the load of a parallel system has been investigated from many different points of view. It is a problem that is still of great interest mainly because of significant technology advances every few years.

The benefit of load balancing can be illustrated by the following simple example. Suppose that a particular step of a distributed computation is to be executed on n distinct processors, so that the i-th processor has a total computational load of L_i, expressed in total run time. Assume that the computation has to be synchronized at the end of that step. Let $L_A = [\sum_{i=1}^{n} L_i]/n$ be the average execution time of the processors, and let L_M be the largest of the execution times. Then, processor i will remain idle for a time $L_M - L_i$ at the end of this particular step, and as a whole, the processors will remain idle for a total of:

$$\sum_{i=1}^{n} [L_M - L_i] = n[L_M - L_A] \tag{1}$$

time units. Since nL_A is the total amount of computational work involved and is a constant for that particular step, the computation will be accelerated by making L_M as close to L_A as possible, which summarizes to minimizing imbalance.

One's first reaction would be to suggest policies which use full knowledge of the L_i to make an "optimal" load balancing decision. However, the information available will in general only provide an estimate of these quantities for several reasons:

- In many cases, the granularity of the computation does not allow a detailed evaluation of computation times.
- Also, computation times are often data dependent, and cannot be completely evaluated in advance.
- Even when full information allowing evaluation of computation times is indeed available, the evaluation of these times from the information may itself be particularly time consuming (as in the molecular dynamics simulations, where the exact position of each particle determines computation times).
- Finally, in many cases, the loads at each node can only be broadcast relatively infrequently in order not to generate excessive overhead, so that load information may not be fully up to date.

Thus, adaptive on-line policies for load balancing are of particular interest. Among such policies, those which are decentralized are the simplest since they require less coordination among nodes. Decentralized policies present, however, the obvious danger of being counterproductive, since one may unknowingly overload a node which initially seemed lightly loaded. Thus care must be taken to guarantee that this does not happen.

Non adaptive decentralized policies are also used for compute-intensive applications when the computation is mostly static and data dependencies do not change during execution.

In this study, we begin by reviewing the state of the art in load balancing of parallel systems, with the ambition of covering most facets of it, and provide an integrated view of the problem. We have classified the various aspects of it into different levels, namely, the application layer, the compiler layer, the run time layer and the operating system layer. This reflects the investigations on load balancing that have been undertaken by researchers with various backgrounds and expertise.

In the application layer, we review problems that arise in science/engineering. In this layer applications are decomposed so that the computation is evenly assigned to the system processors, while communication and synchronization among them is minimized. This is usually stated in a mathematical model and optimization techniques are employed for its solution. Most of these optimization problems are NP complete and various heuristics are used for their solution. The heuristic techniques are summarized as clustering, deterministic and stochastic optimization heuristics.

In the compiler layer, usually existing code of the application is automatically decomposed by the compiler with most of the work concentrated in the parallelism of the programming language constructs. Optimization techniques are used in scheduling, such as loop scheduling, thread scheduling and synchronization.

In the run time layer, either the results of the application layer scheduling or the compiler layer scheduling are implemented. Both in the application and run time layers the computation of an application is usually represented by an acyclic graph. Clustering techniques are used for the scheduling of such graphs. Even when the run time environment does not have elaborate graph information it can still take advantage of communication and synchronization information that may be provided by the compiler or gathered dynamically at run time.

Finally, the operating system is responsible for the efficient and fair allocation of the multiprocessor to the competing applications. Usually the operating system has the least amount of information about the parallel applications running, but based on user input or run time monitoring, it can share the multiprocessor among applications efficiently.

In order to run applications as fast as possible, the largest and fastest multiprocessors were used, that have been the distributed memory ones. Thus, most approaches on the applications layer have considered underlying distributed memory multiprocessors. On the contrary, the compiler, run-time, and operating system level research has considered shared-memory multiprocessors. This has been the case because, traditional distributed memory multiprocessors lack the necessary mechanisms for efficient process/thread migration, and thus make scheduling and load balancing both expensive and difficult to implement.

The deep understanding of load imbalance issues at all four levels (applications, compile/run time and operating system) of the system is necessary in order to achieve high processor utilization. In the rest of the paper we present a survey of the load balancing mechanisms and policies in these levels.

2 Application Layer

Compute intensive applications are usually related to scientific computing applications. The computational models of such applications in general, simulate physical situations or artifacts and they depend on the available computing resources. The recent advances in high performance computing technologies have provided the opportunity to speed up computational models and increase dramaticaly their numerical resolution and complexity. Parallel scheduling and load balancing are mainly responsible for exploiting the available speedup capability of the machine. Parallel scheduling methodologies for compute intensive applications have been developed mostly for distributed memory machines or clusters of workstations. At the application level the interest is focused on policies that make direct use of the knowledge of the applications. We present parallel scheduling and load balancing policies for two different applications, Partial Differential Equations and Molecular Biology.

2.1 Partial Differential Equations

PDEs are considered the fundamental tool for describing the physical behavior of many applications in science and engineering. We are focusing on parallel scheduling and load balancing issues for computational tasks of problems related to PDE computing on distributed memory parallel machine architectures. We start with problems using a grid or a mesh to approximate the geometric domain of computation and formulate a problem solution on it. The methodologies we shall be reviewing have their source in solution methods applied to partial differential equations. The same methods can be easily extended to computations associated with numerical simulation and to more complicated computational models [139], [113], [47].

Scheduling the computational tasks of a PDE problem on a target parallel machine has been viewed as a two phase approach, as follows: First, the partitioning of the computations into load balanced tasks is achieved, requiring minimum synchronization and communication among them and second their allocation onto the parallel processors. The methodology to achieve a problem partition in such problems draws from the problem characteristic. The computation is usually based on a large static data structure (domain) and the amount of computation is about the same for each data element. To keep the load balanced among processors one should decompose the data structure as evenly as possible, since the computation is decomposed in the same way. For the data decomposition the Domain Decomposition (DD) approach is used. We review the discrete domain decomposition, which is based on a grid or mesh decomposition of the underlying computation. Continuous DD is also possible and the approach is similar. The discrete DD has been extensively studied for PDE elliptic solvers and is regarded as the most suitable for such problems [25], [99]. The basic idea is to decompose the grid or mesh of the PDE domain into subdomains. This results into splitting the discrete equations corresponding to the node or grid points of the subdomain and their interfaces (boundary). The programming model for such computations is single program-multiple data where parallelism is achieved by partitioning the underlying geometric data structures (grid/mesh) of the PDE problem and

allocating the disjoint subproblems or subcomputations to the parallel processors. During each iteration the processors perform (i) an exchange of local data (interface unknowns) with the processors that handle geometrically adjacent subdomains in order to enforce continuity requirements of the PDE solution (*local synchronization/communication*); (ii) an execution of matrix-vector operations (*local computation*); and (iii) an evaluation of stopping criteria and acceleration of the convergence (*global synchronization*). The high performance of these computations on distributed memory MIMD machines, depends on the minimization of the local and global communication time and on the synchronization delays. Global communication/synchronization depends on the efficient hardware/software implementation of broadcast operations of parallel machines [73], [144].

The main issue in most studies has been the minimization of local communication time per iteration. The local communication time depends both on data partitioning characteristics such as, interface length, degree of connectivity of the subdomains, and machine characteristics. The data partition problem is NP-Complete [54] and many heuristic methods have been proposed for finding good sub-optimal partitions of the data. These heuristics are divided into three classes, namely, data clustering, deterministic optimization and stochastic optimization.

2.2 Scheduling and Load Balancing for Grid/Mesh based problems

The geometric data structures mesh/grid used in discretization procedures of PDE problems, are either an orthogonal grid, for a finite difference method, or a mesh (non orthogonal grid), for a finite element method. In either case, the partitioning and allocation heuristics applied have been the same, thus we shall not distinguish between the two structures.

The objective function for the mapping of a mesh/grid M onto a distributed memory MIMD machine so that the workload of the processors is balanced and the required communication and synchronization among processors is minimum, is formulated in [24] and is as follows:

$$\min_{m} \max_{1\leq i\leq \mathbf{P}} \{W(m(D_i)) + \sum_{D_j \in C_{D_i}} C(m(D_i), m(D_j))\}, \tag{2}$$

where D_i is the set of mesh points (subdomain) that are assigned to the same processor, C_{D_i} is the set of subdomains that are adjacent to the subdomain D_i, m: $\{D_i\}_{i=1}^{\mathbf{P}} \longrightarrow \{P_i\}_{i=1}^{\mathbf{P}}$ is an assignment function that maps the subdomains to processors, $W(m(D_i))$ is the computational workload of the processor $m(D_i)$ per iteration, which is related to the number of mesh points on D_i, and $C(m(D_i), m(D_j))$ is the communication workload required (per iteration) between the processors $m(D_i)$ and $m(D_j)$. $\mathbf{P}$ is the number of available processors of the target parallel machine. The synchronization of the processors is a nonlinear correlation of computational and communication workload. In general, it can be included in the $W(m(D_i))$.

One approach to solve the optimization problem is to approximate its objective function (2) by another function which is smoother, more robust and suitable for the existing optimization methods [53] [51], [157], [98]. A second approach is to split the optimization problem into

two distinct phases corresponding to the *partitioning* and *allocation* of the mesh [24], [23], [136]. In the *partitioning phase* the mesh (or grid) is decomposed in a pre-specified number (usually equal to the number of processors) of subdomains such that the following criteria are approximately satisfied:

(i) the maximum difference in the number of mesh (or grid) points of the subdomains is minimum,

(ii) the ratio of the number of interface points to the number of interior points for each subdomain is minimum,

(iii) the number of subdomains that are adjacent to a given subdomain is minimum,

(iv) each subdomain is a connected domain.

In the *allocation phase* these subdomains are assigned to processors such that the following objective is satisfied.

(v) the communication requirements of the underlying computation between the processors of a given architecture are minimum.

2.3 Geometry Splitting Based Partitioning Methodologies

Most of the heuristic methods proposed for the partitioning phase can be classified into three large classes. The first class involves *clustering* techniques where the main idea is to sort the geometric mesh data in some direction and then partition the resulting sequence of elements in **P** ways. The second class consists of the *deterministic optimization* techniques. The idea is to find "semi"-optimal feasible solutions in linear time. These methods tend to terminate in some local minimum without always being able to move to a global minimum. A more computationally expensive alternative is to use a *stochastic optimization* technique, which locates a global minimum "most" of the time. In [69], simulated annealing and neural network techniques have been implemented for the mesh partitioning problem. Next, we give a brief review of these methodologies.

Clustering Techniques Farhat [46] proposed a method for ordering the topological data of a mesh. The underlying idea of his scheme is equivalent to the well known Cuthill-McKee method of ordering, [91], [57], [58], [59], as applied to order finite element meshes so that the corresponding linear algebraic system of equations has minimum bandwidth and profile. According to this technique one finds all the unlabeled neighbors of element (node) i and labels them in order of increasing connectivity (communication with neighbors). This method is referred as *CM-cluster*. Another naive way for splitting meshes is to sort some geometric data of the mesh (i.e., coordinates of vertices, coordinates of sector origin of the elements,

coordinates of the centroid of the elements) and subdivide the sorted lists in sublists of length $N/\mathbf{P}$, where N is the number of the geometric data of the mesh. The same idea is also applied to the data (topological data) of the mesh. These sorting algorithms are referred as $1 \times \mathbf{P}$ *geometric/topological partitioning* and $\mathbf{P} \times Q$ *geometric/topological partitioning* algorithms.

Deterministic Optimization Techniques The problem of partitioning geometric meshes into load balanced subdomains is formulated on the topological graph of the mesh. This is equivalent to disconnecting a graph into nearly equally sized subgraphs by cutting the minimum number of edges. This problem has been formulated as an optimization problem by defining an objective function whose minimum value corresponds to the optimal partition of the mesh. In order to satisfy the load balancing constraint among the subdomains, the objective function has two components; one which is minimized when there are no edges in the partition (minimum communication), and the other is minimized when an equal number of nodes is assigned to each subdomain. These two sub-objectives tend to *compete* with one another in that the first goal is satisfied when the nodes are uniformly distributed across the specified number of sub-graphs, while the second goal is met (trivially) by collapsing all the nodes into a single partition.

The simplest graph partitioning problem is the 2-way one, in which the graph nodes are divided between two partitions (subdomains), D_1 and D_2. The mathematical foundations of the optimization problem are stated in [23]. A domain decomposer tool [22] has a library of various **P**-way algorithms for minimizing the above cost functions. It includes the well-known Kernighan-Lin heuristic [74], [24] technique for minimizing the cut-cost of the mesh graph, assuming the solution satisfies the constraints of the balanced partitioning. The idea of this approach is to identify an improved feasible solution by interchanging elements among the subdomains that optimize a profit function. In [23] a **P**-way partitioning algorithm has been developed with a modified profit function based on Kernighan-Lin's idea of selecting improved feasible solutions [24]. This method is referred as the GGP (Geometric Graph Partitioning) algorithm. A recursive variation of this algorithm based on a modified 2-way Kernighan-Lin algorithm has been also developed [24] and named GGP-*rec*; this heuristic is also called *orthogonal recursive bisection* [53], [157]. These algorithms require an appropriate initial feasible solution, which is selected by the user out of the set of predetermined initializations.

Stochastic Optimization Techniques A significant advance in optimization was made in 1983 with the invention of simulated annealing (SA) [76]. SA has been used in [51], [53] and [157]. Hopfield neural networks [67] constitute another avenue for solving discrete combinatorial problems. These networks involve many simple computing units (or artificial neurons) whose objective is to minimize an energy function associated with the optimization problem. In [69] various artificial neural networks (ANN) have been considered for solving the partitioning phase of the mapping problem.

In [23] a list of partitioning algorithms have been developed for the *parallel ELLPACK* system [70]. In *parallel ELLPACK* all the existing public domain software for PDE solvers is included that support well defined mathematical models of multiple applications [130], [61],

Table 1: Mesh Partitioning Algorithms

Name	Description
$1 \times \mathbf{P}$	1-D strips
$\mathbf{P} \times Q$	2-D strips
ORB-E	Eigenvalue Ortho. Rec. Bisection [19]
ORB-M	Mass Center ORB [157]
ORB-I	Inertia Axis ORB[Chri91]
ORB-KL	Kernighan-Lin ORB [74]
ORB-GGP	Modified K-L ORB [24]
ORB-ANN	Neural Net [68]
ORB-SA	Simulated Annealing
GGP	P-way Geometric Graph Part [24]
SA	P-way SA [157]
CM-Clustering	Cuthill-Mckee [46]

[129], [137], [97], [108], [109]. The analysis and performance evaluation of these partitioning algorithms is reported in [24], [23], [22]. Four basic types of heuristics have been implemented for partitioning meshes. They include $1 \times \mathbf{P}$ strips, $\mathbf{P} \times Q$ lattices, and 2-way recursive bisection and $\mathbf{P}$-way partitionings. The $1 \times \mathbf{P}$-way partitions are obtained by sorting the x (or y or z) coordinates of the centroid of the elements and subdividing the sorted list in groups of $N/\mathbf{P}$ elements. In this case, the assignment of the subdomains to an array of processors is the identity. This is referred to as algorithm $1 \times \mathbf{P}$. A $\mathbf{P} \times Q$-way partitioning is obtained by sorting the coordinates of the element centroid in x and y directions. We refer to these techniques as $\mathbf{P} \times Q$. Variations of it have been proposed in [119], and the Simulog's system. Another important class of heuristics included in this library are the so-called Orthogonal Recursive Bisection (ORB) techniques based on different 2-way partitioning heuristics. In [22] a number of geometry based, mesh/grid heuristic algorithms have been implemented for the *parallel ELLPACK* system. Comparative preformance analysis of these algorithms is also reported. They are listed in Table 1.

2.4 Molecular Dynamics

Another important application area which will be discussed in this paper pertains to load balancing for *Molecular Dynamics* which is one of the major grand challenge parallel processing applications [1, 107]. It has very broad implications in the understanding of the detailed structure of materials and of large biomolecules, and is a fundamental tool for the computer aided design of new chemical and biochemical compounds. There are many forces acting between the atoms in a biomolecular system, but the simple Coulomb force is the most challenging to

compute, as its long range requires in principle all pairwise combinations of atoms to be considered. The resulting electrostatic N-body problem characterizes the Coulomb interactions between a set of charged bodies. The same equations also characterize interactions between gravitating bodies for astrophysical applications. Molecular dynamics can thus be improved by developing and implementing efficient algorithms for solving the N-body problem for particle systems on parallel and distributed computing platforms [18, 79, 17].

Workload and data (file) imbalance continues to be an impediment to massively parallel implementations of such codes [13, 95]. Now that numerous programmer-years of effort have been invested in hand-optimizing these codes for particular architectures and particular types of particle distributions, we need to research "automatic" balancing of such codes. If an adequate automatic load balancing and data contention reducing scheme can be implemented, future versions will be far more efficient and will require much less "hand optimization". It is therefore important to determine the general principles which can be used to provide optimum on-line load balancing for Molecular Dynamics. The discussion will concentrate on optimized N-body codes which use the linear-time Fast Multipole Algorithm (FMA) [21, 17] and related methods. The same N-body codes are currently used in production molecular dynamics work [17]. Specifically, we will discuss adaptive load balancing based on the gradient descent paradigm to compute optimal on-line decisions on a process-by-process basis, in line with our recent work [55, 56]. The algorithms we consider make all decisions locally, and are activated upon the creation of a task.

2.5 Parallel Scheduling for Molecular Dynamics

Molecular dynamics (MD) has been long known as a major computational task which can revolutionize the way science is done [1, 107, 146]. It is both a major scientific problem area for computational physics and chemistry, and one which has very broad applications in the study and design of biomolecules and of new chemical compounds. This is one of the major grand challenge parallel processing applications [13, 95]. Molecular dynamics studies the Newtonian interactions of a system of interacting particles (atoms), so as to determine their movement, equilibrium state, and other static and dynamical properties. In biomolecular simulations, a wide variety of forces act on each atom, but only the electrostatic force is long-ranged, so that the evaluation of the Coulomb interaction requires the great majority of CPU time ($> 99\%$ for large particle systems). In its electrostatic version, the N-body problem characterizes the Coulomb interactions between a set of charged bodies, and its consequences on the motion of the particles in 3-D space. Molecular dynamics can thus be improved by developing and implementing efficient algorithms for solving the N-body problem for particle systems on parallel and distributed computing platform[18, 79, 17]

The evaluation of electrostatic interactions calls upon a "direct Coulomb solver" algorithm [1, 13], which generally uses two nested loops to consider all pairwise combinations of particles yielding runtime complexity $\mathcal{O}(N^2)$ for N particles. Though it can be a prohibitively time-consuming computation – even in parallel – this "direct" approach can be easily implemented in

fully parallel code, even when the particle distributions in $3-D$ space are highly nonuniform. However, load balancing is still a major issue because each iteration of the algorithm will compute new positions of the particles leading to imbalance between the processors [13]. In this discussion we concentrate on a computationally more efficient method, the Fast Multipole Algorithm [60, 17], which runs in $O(N)$ time complexity.

The Fast Multipole Algorithm (FMA) [60], and other related multipole approximation methods [5, 12, 156], solve this N-body problem in $\mathcal{O}(N \log N)$ or $\mathcal{O}(N)$ time. They hierarchically divide the simulation region (a cubic box for simplicity) into successively finer boxes. In $3-D$, each box is subdivided into eight child boxes; this subdivision is continued until a small (and machine dependent) number of particles remains in each "terminal box" – which we will refer to as a "box" in the sequel.

When particle distributions in space are reasonably uniform, a uniform oct-tree of boxes will result. On the other hand when particle distributions are highly nonuniform, a selectively refined (irregular) tree of boxes can yield a more efficient algorithm [21]. However, this can significantly complicate efforts to parallelize and balance the algorithm [80].

One iteration of any Molecular Dynamics solution algorithm carries out the following four major computational steps:

- **Step 1** It first determines which particles need to interact, and in which way. For instance, in solids certain approximations [95] assume that only particles which are closer than some distance r_{max} need to interact. In our case, we need to consider all particle interactions: the multipole expansion approximates "far" interactions, while simple particle-to-particle Coulomb forces describe "near" interactions.
- **Step 2** Next, it calculates forces (as indicated above).
- **Step 3** Thirdly it carries out time integration.
- **Step 4** Finally it updates the positions of all particles.

Step 2 is the computationally most intensive portion, while the fourth and last step is where the amount of work at each processor is being changed.

There are two principal sources of load imbalance when parallelizing the FMA which limit scalability:

- The first is structural - depending on the boundary conditions used in a given simulation, when creating the hierarchical oct-tree of cubes in space, some boxes are at corners or are on edges or faces and thus have fewer neighbors than boxes on the interior. Interior boxes run slower than exterior boxes, with corner boxes running fastest of all, because of their small numbers of "close" neighbors.

- The second imbalance is data-dependent; even a mildly non-uniform spatial distribution of particles results in different runtimes, as much as 10:1 differences for Step 2 processes. The local direct step also suffers structural imbalance, since particles in corner boxes interact with fewer other particles than those in interior boxes.

Data contention also limits efficiency. It results from the fact that two or more processes may try to interact with the same box at the same time. If we do not address this problem, the resulting implementation may be balanced but still inefficient. Since force and electrostatic potential computations are symmetric (force on A due to B is equivalent force on B due to A with appropriate sign change), it is tempting to update both boxes in one interaction to avoid duplicating the computation. This, however, requires both boxes to be locked in write-mode, while other processes must wait to use the same boxes. If the remote box is kept read-only, then the multipole computation must be repeated when the boxes exchange roles in the procedure. The same problem affects particle interactions in the local direct step.

The multipole parts of the computation are currently partitioned statically among the available processes as described above by distributing level 3 boxes. The structural load imbalance discussed above remains a problem for certain numbers of processors (*i.e.* 8 processors each have one corner, but with 16 processors some boxes have corners and some do not). The local direct step is treated differently: individual boxes or small groups of nearby boxes from the finest level of the tree (as opposed to level 3) are dispatched from a central work queue to the next available processor. By dividing at the finest level, we create many parcels of work for this step so that a processor which happens to be assigned a sparsely populated box (or a corner box with few neighbors) can rapidly complete it and begin on another box while another processor continues to work on a more densely populated box.

Since numerous programmer-years of effort have been invested in hand-optimizing these codes for particular architectures and particular types of particle distributions, we now must find new "automatic" methods for balancing of such codes. If an adequate automatic load balancing and contention reducing scheme can be implemented, future versions of molecular dynamics simulations will require much less effort. Since hand-optimized codes are available to compare with, we can assess the penalty of turning the optimization over to a balancing procedure for a variety of particle systems.

A class of policies which are applicable to Molecular Dynamics – will move work from one node to another in a "grouped" manner, in advance of the actual execution time of tasks. We shall call such policies Load Imbalance Based. In this case we will have to take precautions *not to overlaod* a particular host, which can be caused by poor decentralized decision making. This will be achieved by restricting ourselves to *receiver initiated* policies.

Consider a computation which will be executed on n nodes, and suppose that before the computation is initiated, the system balances the load. Initially, all nodes broadcast the value of their computational load: let L_i be the load at node i. Clearly, before and after load balancing, the application's average computational load L_A remains the same, though there will be additional work and communication related to moving the work.

After each node has broadcast its load value, and this information is received by all nodes, every node knows L_A and can determine two subsets of nodes: S the potential senders of work – and R the potential receivers of work. Clearly, for $s \in S$, $L_s > L_A$, and for $r \in R$, $L_r < L_A$. The receiver initiated policies considered will then proceed as follows:

- Each potential *receiver* r will compute a decision variable π_{sr} for each possible *sender* s – This variable will be interpreted as follows: if $\pi_{sr} = 0$, then r will **not** request work from s, while if $\pi_{sr} = 1$ it will request as much work "as possible" from s.
- The receiver r will then select some senders for which $\pi_{sr} > \theta_r$, from which it wishes to receive work, where θ_r is a threshold value. Then each r will send a message requesting an amount of work W_{sr} to be sent to it, from each *selected sender* s. In order to avoid overload at the receiver, r will never request more work than what is required to move it's load up to the average workload value: $\sum_{r=1}^{n} W_{sr} \leq L_A - L_r$.
- Finally, each sender s which is solicited by a receiver, will respond by sending *at most* as much work as is requested.

Since the objective is to reduce imbalance, we are interested in minimizing some function of the absolute value of the difference between each node's load and the average load. However we should include communication overhead at each node in its load. Let the time taken up in transfering the work W_{sr}, be $f(W_{sr})$ at the receiving node, and $g(W_{sr})$ at the sending node. The functions f and g which relate the amount of work transfered to the time it takes the receiving and sending PU can generally be estimated.

The cost function C to be minimized by the load balancing algorithm can now be expressed as:

$$C = \frac{1}{M} \sum_{r \in R} \{ L_r + \sum_{s \in S} \pi_{sr} f(W_{sr}) - L_A \} + \frac{1}{M} \sum_{s \in S} \{ L_s + \sum_{r \in R} \pi_{sr} g(W_{sr}) - L_A \}^M \quad (3)$$

where the exponent M will be an even number (typically $M = 2$), so that C can be positive.

3 Compiler and Run-time Environment Layer

The parallel execution of scientific/engineering and parallel database related applications use similar representation methods and techniques for task assignment. The problem formulation is usually different since it draws mainly on knowledge about the problem characteristics, such as, the problem can be decomposed if the data can be decomposed. Thus, domain decomposition is used to decompose the data which results into the computation decomposition in grid/mesh based applications. The grain of parallelism is usually medium to coarse and the resulting scheduling is executed at run-time.

The compiler and run-time approach to parallel application execution, is a more general approach and it addresses all applications. It takes a different point of view of an application. Parallel execution is achieved by working directly on the code of an application, without any other knowledge of its characteristics. Optimizing compilers address programming language optimization by techniques like executing as much of the code as possible at compile-time, performing code transformations, creating many threads of execution, parallelizing loops, scheduling threads in efficient ways, investigating synchronization, conditional execution of code, and creating calls to the run-time support system. The techniques used in each area continue to evolve, as not a single framework exists, just isolated efforts to handle special cases and to define a more general framework. These frameworks do not usually achieve the same degree of parallelism as when there is a clear knowledge of the characteristics of the problem.

The programmer may not have enough knowledge of the characteristics of the problem to restructure it in a way that reveals a lot of its parallelism, but may have enough knowledge to indicate some of the possible existing parallelism in his program. In this case he can use directives to indicate optimizations to the compiler, which would otherwise be very hard for the compiler to find. Thus, the knowledge that the parallelizing compiler has about the possible parallelism comes from both the code analysis and user hints. Tools and utilities have been developed that help users to express their knowledge of the problem to compiler directives that will improve the parallelization of the problem. A usual approach is the creation of a consistent set of experimental program variants and the interpretation of compilation and performance result [106], [43]. Bacon et al, in [27] presents an extensive survey of high-level program restructuring techniques for imperative languages.

With the use of a smart compiler, reliable sequential programs can, without being rewritten, be compiled into object codes that are suitable for parallel execution, and also can be fine-tuned to fit various modern computer architectures. A lot of work in the compilers area has been done on revealing parallelism without the programmers hints. In the next sections we will examine these ways of revealing parallelism.

3.1 Loop Transformations

Loop transformations are becoming critical to exploiting parallelism and data locality in parallelizing and optimizing compilers. Loops are the single largest source of parallelism in most applications. Executing the many iterations of a loop on different processors enables applications to take advantage of parallel processors, and thereby reduce their running time. Both static and dynamic loop scheduling methods have been used to assign the iterations of a loop to processors. Static methods assign iterations to processors statically, minimizing run-time synchronization overhead. Traditionally, optimizing compilers attempt to improve the performance of programs by applying source to source transformations, such as loop interchange, loop skewing and loop distribution.

Static scheduling algorithms, such as block scheduling, cyclic scheduling, and block cyclic scheduling, assign a fixed number of loop iterations to each processor.

Block scheduling assigns to each processor an equal number of adjacent loop iterations. If the amount of computation performed by each iteration differs, then block scheduling can perform poorly because of load imbalance.

Cyclic scheduling assigns loop iterations to processors in a cyclic order. It obtains better load balance than block scheduling for iteration spaces where the amount of computation increases/decreases linearly with the iterations.

Block cyclic scheduling is a combination between block scheduling and cyclic scheduling, and assigns blocks of a fixed size to processors in a round robin fashion. The previous block scheduling algorithms can be described as a special case of block cyclic scheduling.

The simple *static scheduling* algorithm divides the number of loop iterations among the available processors as evenly as possible, in the hope that each processor receives about the same amount of work. This algorithm minimizes run-time synchronization overhead, but does not balance the load dynamically. If all iterations do not take the same amount of time, or if processors begin executing loop iterations at different points in time, then load imbalance may arise, which will cause some processors to be idle while other processors continue to execute loop iterations.

Dynamic methods defer the assignment of iterations to processors until run-time, and therefore can achieve better load balancing in the presence of unpredictable transient loads and variable iteration execution times. The major difficulty in dynamic loop scheduling is to keep the run-time synchronization overhead small, while balancing the load.

The simplest dynamic algorithm for scheduling loop iterations is called *self-scheduling* [138, 147]. In this algorithm, each processor repeatedly executes one iteration of the loop until all iterations are executed. The algorithm relies on a central work queue of iterations, where each idle processor gets one iteration, executes it, and repeats the same cycle until there are no more iterations to execute. Self-scheduling achieves almost perfect load balancing, since all processors finish within one iteration of each other. Unfortunately, this algorithm incurs significant synchronization overhead; each iteration requires atomic access to the central work queue. This synchronization overhead can quickly become a bottleneck in large-scale systems, or even in small-scale systems if the time to execute one iteration is small.

Uniform-sized chunking [77] reduces synchronization overhead by having each processor take K iterations, instead of one. This algorithm amortizes the cost of each synchronization operation over the execution time of K iterations, resulting in less synchronization overhead. Uniform-sized chunking has a greater potential for imbalance than self-scheduling however, as processors finish within K iterations of each other in the worst case. In addition, choosing an appropriate value for K is a difficult problem, which has been solved for limited cases only. *Guided self-scheduling* [118] is a dynamic algorithm that changes the size of chunks at run-time, allocating large chunks of iterations at the beginning of a loop so as to reduce synchronization overhead, while allocating small chunks towards the end of the loop to balance the workload. Under guided self-scheduling, each processor is allocated $1/P_{th}$ of the remaining loop iterations, where P is the number of processors. Assuming all loop iterations take the same amount of

time to complete, guided self-scheduling ensures that all processors finish within one iteration of each other and use the minimal number of synchronization operations.

Since processors take only a small number of iterations from the work queue at the end of each loop, guided self-scheduling can suffer from excessive contention for the work queue. If each iteration takes a short time to complete, then processors spend most of their time competing to take iterations from the work-queue, rather than executing iterations. *Adaptive guided self-scheduling* [39] addresses this problem by using a back-off method to reduce the number of processors competing for iterations during periods of contention. This algorithm also avoids assigning all the time-consuming iterations to one processor by assigning consecutive iterations to different processors, which reduces the risk of load imbalance that arises when the execution times of consecutive iterations vary widely but in a correlated fashion (e.g. if the execution time of iterations decreases linearly). As a result of these modifications, adaptive guided self-scheduling performs better than guided self-scheduling in many cases. The Chorus operating system [40] is using guided self-scheduling, with an adaptive mechanism that automatically adjusts granularity appropriately.

In some cases guided self-scheduling might assign too much work to the first few processors, so that the remaining iterations are not sufficiently time-consuming to balance the workload. This situation arises when the initial iterations of a loop are much more time-consuming than later iterations. The *factoring* algorithm [71] addresses this problem. Under factoring, the allocation of loop iterations to processors proceeds in successive phases. During each phase, only a subset of the remaining loop iterations (usually one half) is divided equally among the available processors. Because factoring allocates a subset of the remaining iterations in each phase, it balances load better than guided self-scheduling when the computation times of loop iterations vary substantially. In addition, the synchronization overhead of factoring is not significantly greater than that of guided self-scheduling .

Like the factoring algorithm, the *tapering* algorithm [96] is designed for loops where the execution time of iterations varies in such a way as to cause load imbalance under guided self-scheduling. Tapering is used for irregular loops, where the execution time of iterations varies widely and unpredictably. The tapering algorithm uses execution profile information to estimate the average iteration time and the variance in iteration times. These estimates are used to select a chunk size that, with high probability, limits the amount of load imbalance that can occur to be within a given bound. Although guided self-scheduling minimizes the number of synchronization operations needed to achieve perfect load balancing, the overhead of synchronization can become significant in large-scale systems with very expensive synchronization primitives. *Trapezoid self-scheduling* [154] tries to reduce the need for synchronization, while still maintaining a reasonable balance in load. This algorithm allocates large chunks of iterations to the first few processors, and successively smaller chunks to the last few processors. The first chunk is of size $\frac{N}{2P}$, and consecutive chunks differ in size $\frac{N}{8P^2}$ iterations. The difference in the size of successive chunks is always a constant in trapezoid self-scheduling, whereas it is a decreasing function both in guided self-scheduling and in factoring.

All of these loop scheduling algorithms attempt to balance the workload among the proces-

sors without incuring substantial synchronization overhead. Each of the algorithms assumes that an individual iteration takes the same amount of time to execute on every processor. This assumption is not valid, however, on many shared-memory multiprocessors. The existence of memory that is not equidistant from all processors (such as local memory or a processor cache) implies that some processors are closer to the data required by an iteration than others. Loop iterations frequently have an *affinity* [141] for a particular processor — the one whose local memory or cache contains the required data. By exploiting processor affinity, we can reduce the amount of communication required to execute a parallel loop, and thereby improve performance. Affinity Loop Scheduling is one loop scheduling policy that exploits the affinity an iteration may have for a particular processor [103, 101]. The idea behind Affinity Loop Scheduling is that iterations are assigned statically to processors (to reduce run-time overhead), but can be *rescheduled* when load imbalance happens. Affinity Loop Scheduling has been implemented in several shared-memory multiprocessors, and has shown performance improvements of up to a factor of three. Li *at. al.* have applied a similar idea in the Hector NUMA multiprocessor: each iteration is assigned for execution on the processor where the data it uses have been allocated - if load imbalance happens the iteration may be rescheduled [85].

The static algorithms ignore the fact that the amount of computation performed per iteration may differ and it cannot always be determined a priori and even the speed of each processor may also differ because of multitasking interference. Therefore static scheduling often suffers from load imbalance.

Algorithms that combine static and dynamic loop scheduling have also been created [115], [123], where an initially static assignment of iterations to processor may be dynamically rescheduled, when imbalance is detected. This approach combines the load balancing quality of the dynamic approaches and the low overhead of the static approaches.

When there are dependencies among the loop iterations, the degree of parallelism is more limited, and harder to find. However, there are a lot of techniques for this case too, although they usually consist of revealing the dependency structure and are not implemented in the optimizing compilers, as the profit will be very small.

Overview of loop optimization issues as well as scheduling techniques and comparison results on fine-grained and coarse-grained parallel architectures can be found in [87]. A survey of several experimental studies on the effectiveness of parallelizing compilers, together with an overview of parallelizing techniques, covering dependence analysis techniques of loop transformations too, are found in [11].

Other special cases of loop optimization have also been studied, such as instruction level parallelization applied at the statement level [16], and cases that it is safe to reuse copies of off-processor data and decrease run-time overhead at compile-time, computing global flow information about the communication characteristics of the loops [65]. They even include compiler algorithms to automatically derive efficient message-passing programs based on data decompositions, to minimize load imbalance and communication costs for both loosely synchronous and pipelined loops [66].

3.2 Task Mapping and Scheduling

One way to reduce the need for communication is to use scheduling polices that exploit knowledge of the location of data when assigning processes to processors, improving locality of reference by co-locating a process with the data it will require.

Significant work has been done on compile-time thread scheduling with emphasis on communication minimization. Partitioning and scheduling techniques are necessary to implement parallel languages on multiprocessors. Multiprocessor performance is maximized when parallelism between tasks is optimally traded off with communication and synchronization overhead [127], [94], and the parallel time is minimized when overlapping communication with computation [160]. Task ordering algorithms try to minimize parallel time and overlap communication with computation [50]. Feldmann et al, in [50], have implemented a compiler tool system named PYRROS that integrates their algorithms for scheduling and code generation. The input of this system is a C program with annotated dependence information and the output is optimized parallel C code for nCUBE-I, nCUBE-II and iPSC/860 machines. The code generation problem for message passing architectures is similar: to optimize code by reducing communication overhead, eliminating redundant communication and improving memory utilization [50] [160].

Data partitioning with emphasis on communication minimization has also been studied. In many cases it is possible to prefetch required off-processor data before a loop begins execution, and several loops access the same off-processor memory locations. As long as it is known that the values assigned to off-processor memory locations remain unmodified, it is possible to reuse stored off-processor data. A mixture of compile-time and run-time analysis can be used to generate efficient code for irregular problems, and a number of optimizations can be used to support the efficient execution of irregular problems on distributed memory parallel machines [29]. These primitives coordinate inter-processor data movement, manage the storage of, and access to, copies of off-processor data and minimize interprocessor communication requirements, reducing communication latency and volume. In most approaches the compiler either tries to figure out a good distribution from the control and data flow graphs of a program or explores a small number of canonical distributions for suitable ones [78].

There are still more approaches to data scheduling:

Array statements as included in Fortran 90 or High Performance Fortran (HPF) are a well-accepted way to specify data parallelism in programs. When generating code for such a data parallel program for a private memory parallel system, the compiler must determine when array elements must be moved from one processor to another [143]. The irregular access patterns of the use of indirection arrays for indexing data arrays make it difficult for a compiler to generate efficient parallel code. There are methods for transforming programs using multiple levels of indirection into simpler programs, with at most one level of indirection, thereby broadening the range of applications that a compiler can parallelize efficiently [28]. In another approach, data fields can be defined as functions with explicit restrictions and the static analysis can give information about the extent of a recursively defined data field. This can be used to preallocate

the data fields and map them efficiently to distributed memory [88].

There are many approaches that try to do static data and thread placement at the same time, and automatically find computation and data decompositions that optimize both parallelism and locality. Anderson and Lam, in [2], designed an algorithm for use with both distributed and shared address space machines, that can exploit parallelism in both fully parallelizable loops as well as loops that require explicit synchronization and will trade off extra degrees of parallelism to eliminate communication.

The minimization of different costs will suggest different data and computation partitions. Compiler strategies for mapping FORTRAN programs onto distributed memory computers, to minimize overhead have been developed [114]. One approach to obtain good mappings, is to isolate and examine specific characteristics of executing programs that determine the performance for different mappings on a parallel machine. The process consists of two steps: First, an instrumented input program is executed a fixed number of times with different mappings, to build an execution model of the program. Next, the model is analyzed to obtain a good final mapping of the program onto the processors of the parallel machine. The idea is to find the best use of available memory and communication resources to minimize the global inter-task and intra-task communication overhead [145].

Sometimes better results need user hints. A user specified mapping procedure via a set of compiler directives, allows the user to use program arrays to describe graph connectivity, spatial location of array elements and computational load. Irregular computations can be handled by the HPF compiler effectively [120].

Static thread scheduling based on probability functions, emulations, or even on different thread models have also been a research issue. Ha and Lee, in [64], propose scheduling techniques for static thread assignments, using data-flow graphs representing data-dependent iteration, assuming a known probability mass function for the number of cycles in the data-dependent iteration and showing how a compile-time decision about assignment and/or ordering as well as timing can be made.

Powerful non-strict parallel languages require fast dynamic scheduling. Tolerance to communication latency and inexpensive synchronization are critical for general-purpose computing on large multiprocessors. Schauser et al, in [128], explores how the need for multithreaded execution can be addressed as a compilation problem, to achieve switching rates approaching what hardware mechanisms might provide. Compiler-controlled multithreading is examined through compilation of a lenient parallel language, for a threaded abstract machine, (*TAM*). A key feature of TAM is that synchronization is explicit and occurs only at the start of a thread, so that a simple cost model can be applied. A scheduling hierarchy allows the compiler to schedule logically related threads closely together in time and to use registers across threads. Compiler-controlled multithreading is examined through compilation of a lenient parallel language, Id90, for the threaded abstract machine, TAM.

When the compiler is not sophisticated enough to extract all parallelism, the run-time system schedules the tasks as threads independently. Most work on thread scheduling within

an application has focused on the goal of load balancing alone. For example, in the process control scheme [153], Uniform System [151], Brown Threads [30], and Presto [14], all threads of the same application are placed in a FIFO central work queue. Processors take threads from this queue and run them to completion. The load is evenly balanced in that no processor remains idle as long as there is work to be done.

Anderson *et. al.* [4] argued for the use of per-processor ready queues within a thread package to improve scalability (reducing contention for the single ready queue) and to preserve processor affinity. Under this scheme, a newly created thread is placed on the ready queue of the processor on which it was created. Idle processors scan their own ready queues first, looking for threads to execute. If there are no threads in the local ready queue, a processor looks in the the ready queues of other processors.

All of these user-level policies execute a thread on the processor on which the thread was created, or on whichever processor happens to be idle when the thread reaches the front of the ready queue. Once a thread begins execution, it typically runs to completion on that processor, thus preserving cache affinity. Only in rare cases does a thread establish state on a processor and then migrate to another. However, even in cases where a thread executes on only one processor, it may spend a substantial percentage of its lifetime bringing the data it needs into the local cache.

To avoid the cache-misses associated with naive thread scheduling decisions, recent thread libraries take communication costs into account. For example, U-threads [102], and Mercury [52] assigns threads for scheduling in processors close to their data.

Data locality in load balancing is also addressed by Hamidzadeh and Lilja, in [10], together with scheduling and synchronization costs. They propose a self adjusting scheduling algorithm that dedicates one processor to search for partial schedules concurrently with the execution of tasks previously assigned to the remaining processors.

There are so many different cases that the compiler or the run-time system can load balance the execution of the code, that not all of them can be mentioned here, or not all of them have been studied. The more parallelism revealed in a program, the better the load balancing can be. The operating system is another point of control of the load balancing on the processors of a machine. One big difference on the techniques and methods used for load balancing by the compilers and the ones used by the operating systems, is that the operating systems have no knowledge of the behavior of the executed programs.

4 Operating System Layer

The operating system has traditionally been responsible for the *fair* and *efficient* allocation of resources, in general, and computing power, in particular, to various users (applications). In this section we survey mechanisms and policies for operating system-level scheduling and load balancing in parallel and distributed systems. We focus on policies that have been implemented, or have at least been studied in conjunction with an existing system.

The job of the operating system (as far as scheduling and load balancing are concerned) is to schedule processes to processors so that each application has low response time, the system sustains high job throughput, and the allocation of computing power to applications is done fairly. Because it is difficult to tradeoff these conflicting goals, most operating systems resort to reduce the various sources of overhead experienced by parallel and distributed applications. The idea is that if all sources of overhead are reduced, the throughput of the system increases, which frequently decreases the average response time seen by each application. The major sources of scheduling-related overhead the operating system needs to consider and carefully balance are:

Load Imbalance: When some processors are idle, while some other processors have been assigned several processes to execute, the system is underutilized and load imbalance happens. This imbalance may be the result of a poor *initial placement* of processes to processors, or of uneven assignment of computation to various processes.

Process Migration: To alleviate load imbalance, a scheduler may move a process from one processor to another. Unfortunately, this move, includes the copying of the memory and various other resources the migrated process needs. This overhead (called the *process migration overhead*) and ranges from milliseconds (in closely coupled multiprocessors) to several seconds (in loosely-coupled multicomputers), depending on the architecture, the application and the operating system.

Communication - Synchronization overhead: Although processes belonging to different applications are rather independent, processes belonging to the same parallel/distributed application communicate and synchronize towards solving the same problem. Scheduling processes naively, may result in significant communication/synchronization overhead. For example, if two processes communicate frequently, they should be scheduled on *nearby* processors, or even on the same processor (if there is enough parallelism) to avoid making expensive network transactions each time one process wants to communicate information to the other.

Balancing the above sources of overhead is a difficult task, because (i) the goals of scheduling are usually in conflict with each other (e.g. spreading processes to processors may balance the load, but may create communication overhead) and (ii) most operating system schedulers have incomplete information about user applications.

4.1 Early Work

The first scheduling and load balancing policies in shared-memory multiprocessors were based on their uniprocessor predecessors. Small-scale bus-based shared-memory multiprocessors (like the Sequent Symmetry and Balance [132, 133], the Encore [44], the Firefly workstation [148], and the Silicon Graphics multiprocessor workstations) use a central ready queue. All ready

processes created by all parallel and sequential applications are put in the same central ready queue. Idle processors take the first process from the queue and start executing it. To avoid starvation, processes are periodically interrupted by the hardware clock, suspended and put back in the workqueue so that all processes have a chance to make forward progress. Although conceptually simple, the single ready queue approach has several limitations, including unfair allocation of resources to applications, contention, synchronization overhead, large number of cache misses, etc.

To provide a fair policy based on a central ready queue, Leutenegger suggested that each job should get a quantum inversely proportional to the number of processes it creates [84, 83]. Thus, each job, no matter how many processes it creates, gets its fair share of the system. Leutenegger verified the applicability of his policy using simulations.[1]

A system that uses a central ready queue for scheduling suffers from contention when several processors attempt to get a process from the ready queue to run it. To alleviate this contention problem, distributed ready queues have been proposed. For example, in the Renaissance parallel operating system that has been implemented on top of the Encore shared-memory multiprocessor, the *process-container* abstraction is proposed [125, 124]. Each process belongs to one process-container. Each processor can take ready processes to run from one process-container only. By controlling the mapping of processes and processors to process-containers, various scheduling policies can be implemented. For example, if there is a single process-container for each processor, we have a single ready queue per processor. If there is only one process-container in the system, we have a central-workqueue scheduling policy.

4.2 Synchronization Overhead

Even the above scheduling policies that do not suffer from contention, have significant limitations, because they ignore the fact that processes that belong to the same application (job) cooperate and interact with each other. For example, processes of the same application communicate via critical sections, and synchronize using barriers. It has been shown that synchronization primitives implemented by the run-time library interact with the scheduling policy used by the operating system, and sometimes this interaction results in significant overhead [161, 92, 93]. For example, if a process is preemted while holding a lock, all other processes that want to get the lock will be forced either to spin-wait for it, or suspend, but in any case, they will not be able to make forward progress. As another example, consider the case the parallel application consist of *phases* separated by synchronization barriers. If some of the processes that compute towards the barrier are preempted by the scheduler, then the other processes will probably reach the barrier, but they will not be able to proceed beyond the barrier, unless the preemted processes start running and reach the barrier [100]. Although synchronization primitives for multiprogrammed systems have been proposed [100, 158, 159], some operating systems incorporate scheduling mechanisms to address the above synchronization problems. Psyche, for example, [104, 131] incorporates a mechanism called the *two-minute*

[1] Leutenegger's simulations run on top of the Condor [90] load balancing system!

warning. When a process is about to be descheduled the kernel sends an upcall called the two-minute warning to the user application. When the application receives this upcall, it knows that it is about to get preempted, so it should refrain from getting into a critical region, or do whatever cleanup is necessary. Experimental results suggest that the two-minute warning may improve performance by a factor of three is some cases [104]. In the UltraComputer system [42], each process share a *do-not-preempt-me* bit with the kernel. When a process is about to enter a critical section, it sets the do-not-preempt-me bit, and resets it when it exits the critical section. When the scheduler is about to preempt a process, it checks the bit first. If the bit is set, the scheduler does not preempt the process. To prevent malicious users from monopolizing the CPU, the scheduler abides to the do-not-preempt-me bit, only a limited amount of times. Scheduler activations [3] is another way to address the overhead associated with preemption inside a critical section. Instead of avoiding the preemption problems, as Psyche and Ultracomputer do, Scheduler Activations take an optimistic approach and correct the problems once a process gets preempted inside a critical section and another process wants to enter the same critical section. In this case, the latter process starts executing the rest of the preempted process's code, until it exits the critical section. Then, the running process starts executing its own code and enters the critical section. Scheduler activations are based on the close cooperation of the compiler, the thread library and the operating system.

The Medusa Operating System provides a novel scheduling method called *Coscheduling* [117, 116], which seems to be a general solution to all mentioned synchronization problems. In coscheduling, either all processes of an application run at the same time, or none of them runs. Thus, if a process wants to interact with the other processes of the same application, all processes are running and the interaction can happen. When a process of an application is preempted, all processes of the same application are preemted as well. So, even if the preemted process holds a lock, no other process will ask for the lock. Coscheduling has been implemented in the Medusa distributed operating system [117], and in the Psyche multiprocessor operating system [26]. Although coscheduling addresses the synchronization problems mentioned above, it may lead to processor underutilization: For example, suppose a 10-processor system, and two applications: the one has 10 processes while the other has 5 processes. When the second application runs, only five processors are used effectively. The other five may either be idle, which leads to severe load imbalance, or they may execute five processes of the first application, in which case the first application will suffer from the synchronization problems that coscheduling was designed to address! Simulations done within the Medusa project, suggest that in an average system about 80% of the processors execute coscheduled applications, while the rest 20% execute some (but not all) processes of parallel applications. Coscheduling is also difficult to implement in large scale systems. The major requirement of coscheduling is that all processors context switch at the same time. In a small scale system where all processors can be connected to the same interrupt line, this requirement can be easily satisfied. In a large scale system, such a physical line does not exist, and other scalable software methods must be used [48, 49].

4.3 Communication Overhead

Recent architecture trends have changed drastically the way we should approach multiprocessor scheduling and load balancing. Processors have become significantly faster, increasing disparity between processor and memory speeds. To hide the increasingly higher memory latencies, large caches are being used. This implies that when a process starts running on a processor it will experience several cold misses, but when it builds its working set in the cache, misses will be significantly reduced. If the process is suspended (e.g. due to time-sharing) and later resumed on a different processor, it will experience the same cold misses again, resulting in high overhead. Instead, if the process is started on the processor it was previously run, most of its working set it will still be there[2]. Preserving affinity, improves cache hit rates, but may lead to load imbalance: if processes are statically assigned to processors and *never* migrated, they will preserve their affinity, but they will introduce load imbalance, because some processors will be idle, while other processors will be overloaded with affinity-preserving jobs. To remedy the imbalance, Squillante and Lazowska suggested a hierarchical structure of ready queues of processes. Each processor has its own ready queue; there is also a single system ready queue where newly created processes are placed. Each processor takes processes to execute from its own ready queue. When these processes complete and the queue empties, the processor takes processes from the single system ready queue and places them in its own ready queue. In the (rather rare) case where the single system queue is empty as well, the processor snoops on the queues of other processors and takes processes to execute from there. Squillante simulated his proposed policies and showed that affinity-preserving policies outperform the others [140, 141].

Although Squillante's work preserves affinity in time-sharing multiprocessor scheduling policies, it was soon realized that most scheduling overheads are related to the fact that in several of the proposed scheduling policies, different applications time-share the same processor. Thus, most scheduling overheads will be eliminated if space-sharing policies are considered: processors are divided among the applications in the system, so that no two applications share a processor. The operating system is responsible for giving each application a set of processors, while the application/compiler/run-time-system is responsible for scheduling the useful computation on top of these processors. The division of processors among applications may be static, semi-dynamic (reevaluated at application arrival and departure), or fully dynamic (reevaluated as application's parallelism changes).

Static policies are the easiest to implement. Their main advantage is that they provide applications with a *dependable* environment that is guaranteed not to change for the entire duration of the application. Thus, given a constant number of processors, the applications can optimize their grain of parallelism, their communication pattern and their synchronization primitives, to match the set of processors they are given. Unfortunately, static policies tend to underutilize the multiprocessor. If, for example, an application completes its execution, its processors will not be assigned to any running application, but will be assigned to the next new application that arrives at the system (whenever that happens). To improve system utilization,

[2]The fact that a process has its working set on the cache of some processor is called *cache affinity*.

semi-static space-sharing scheduling policies have been proposed, implemented and simulated.

Crovella *et. al.* implemented a semi-dynamic space-sharing policy on top of a BBN Butterfly plus parallel processor, and compare it to coscheduling and naive time-sharing [26]. Their result suggest that space-sharing outperforms all time-sharing policies. They suggest that the major advantage of space-sharing stems from the fact that parallel applications have sublinear speedup. Thus, an application can use a small number of dedicated processors much better than using a larger number of processors, but on a time-shared basis. For example, suppose that we have an 100-processor system and two applications. It is better to give each application 50 processors, than giving each application half of the time in all 100 processors, because in the latter case the communication/synchronization overhead of the application will be much larger. Tucker and Gupta suggested that run-time systems should cooperate with the operating system to make sure that no application employs more processes than the number of processors it is given. This cooperation will result in a semi-static space-sharing policy. They implemented their approach on top of an Encore shared-memory multiprocessor and showed that it outperformed other time-sharing approaches [153, 152, 63]. Subsequent work by the same authors [62] verifies via simulation that space-sharing outperforms coscheduling and naive time-sharing.

Black has reached similar conclusion when he implemented space-sharing on top of the Mach operating system [15]. In his policy, each application is guaranteed a number of processors for a long period of time (several minutes). After that interval, the application should be prepared to give the processors back, or else it will be scheduled in a processor pool along with several other applications.

Arpaci *et. al.* suggested that a combination of semi-static space-sharing and coscheduling policies should be used for scheduling parallel and sequential applications on a network of workstations [6]. They suggest that parallel applications should be separated from sequential applications, so that the former applications get lots of processor cycles, while the latter applications get interactive response. Parallel applications should be scheduled in their partition using coscheduling. It seems that the results of [6] contradict the rest of the research in multiprocessors that suggests that space-sharing is better than coscheduling [26, 62]. The reason for the seemingly contradicting results is the programming model used in [6] and in the other approaches [26, 62]. The experimental testbed of [26, 62] consists of shared-memory applications running on top of single-address-space multiprocessors, where it is relatively inexpensive to change the number of processes of an application in order to take advantage of a semi-static space-sharing policy. In [6] instead, the experimental testbed consists of message-passing applications that run on top of a loosely-coupled workstation cluster, where changing the number of processes and migrating them is difficult to implement and expensive to use. Thus, coscheduling parallel applications in a workstation cluster seems a reasonable choice that shares the cluster among message-passing parallel applications in a fair way.

Although better than static scheduling, semi-static scheduling may still underutilize the multiprocessor, especially when applications with varying levels of parallelism are considered. Zahorjian and McCann pointed out that several parallel applications have a varying amount

of parallelism. In such cases, allocating a fixed number of processors to each application would result in significant load imbalance, because applications with decreasing parallelism will underutilize their processors, while applications with increasing parallelism could use some extra processors [162]. In their subsequent work, McCann *et. al.* implemented the proposed dynamic space-sharing policies, and showed that they dominate time-sharing policies [105].

4.4 Initial Processor Allocation

Given that each application receives a number of processors for a (rather) long period of time, the scheduler should decide *how many* and *which* processors to give each application. To make the right decision, information is needed by the scheduler. The simplest form of information is how many processes an application has, while the most comprehensive information includes the completion time of the processes, their precedence, and their interactions. Other types of information include average, minimum, maximum parallelism, speedup, completion time, etc.

Using only elementary application performance characteristics, Sevcik proposed that when the load in the system is high, each application should be given a number of processors close to the number it can keep 100% busy (usually this is 1-2 processors). When the load is light, each application should be given a number close to the maximum number it can use [135]. Sevcick in his subsequent work proves that the optimal processor partitioning allocates processors in proportion to the square root of the amount of work each application executes [134]. If the speedup of each application is also known, the scheduler should give each application the number of processors that minimizes the ratio of completion time over efficiency [41], as Eager *et. al.* suggest. If, however, either of them is not known, Eager *et. al.* suggest to give each application a number of processors equal to its average parallelism. Because this policy achieves at least half the speedup, while attaining at least 50% of the efficiency, it provides the guarantee that for each application *either* its speedup, *or* its efficiency will be high, but no matter what the load circumstances are, both speedup and efficiency can not be low.

In some cases, however, it is possible for the scheduler to have lots of information about the parallel application, which is gathered by the compiler and communicated to the run-time system or the operating system. This information usually has the form of a graph. The nodes of the graph are the processes. The edges of the graph represent precedence constraints and/or communication requirements. Processes mapped on the same processor communicate for free, while processes mapped on different processors pay a communication penalty that depends on the distance of the processors, the size of the messages transferred, etc. The task of the scheduler is to map the graph onto the available processors in such a way as to minimize total completion time. There is a non-trivial tradeoff that has to be solved by the scheduler: Placing different processes on the same processor decreases communication cost but increases load imbalance - placing different processes on different processors may reduce imbalance but increases communication cost. Although the tradeoffs of the problem sound simple, the problem has been shown to be NP-complete. Thus, most research has focused on finding suboptimal solutions [126, 72, 81] on placement of tasks on processors. Although

interesting from a theoretical point of view, such solutions are not robust in a real-world environment where not all the information needed is known, and where unpredictable factors (like the existence of other applications) may upset the carefully calculated schedule.

Zhou and Brecht use application performance characteristics to decide not only how many processors an application is going to get, but *which* of the available processors the application should get. They proposed a pool-based scheduling policy for large-scale NUMA multiprocessors in which all processors of the system are partitioned into processor pools [165, 20]. An application is given a set of processors that usually reside in the same pool, unless there are significant performance advantages for an application to span multiple pools. Their work is particularly important given the fact that most recent multiprocessors are based on hierarchies of smaller systems [36, 82, 155]. The problem becomes more complicated if the applications demand processors that are not only close to each other, but are also in a specific topology. For example, in hypercube systems, applications are written as if they run on a dedicated hypercube. If an application is given a set of processors that are not connected in a sub-cube, the application will experience severe performance degradation. To remedy this situation, a significant body of the literature deals with subcube allocation [37], [45], [75] and mesh allocation [86] in multiprocessor systems.

4.5 Dynamic Reassignment of Processes

To alleviate the problems related to a poor initial allocation of processors to applications, some systems employ dynamic reassignment of processors to applications.

4.5.1 Checkpoint-Restart Systems

The simplest mechanism of reclaiming processors from applications involves process checkpoint and restarting: all processes are periodically checkpointed; that is, their memory image is dumped on the disk. When a process must be migrated, itis killed and restarted from its previous checkpoint on the new processor [89]. Checkpoint-restart systems suffer from additional overhead as they have to save the state of a running processes at regular intervals. This overhead can be reduced by increasing the time interval between checkpoints. However, in the case a process needs to be migrated, it will be restarted from its last checkpoint which could have been taken a long time ago; all the work the process has done since its last checkpoint will be wasted. Condor is an example of a checkpoint/restart system that has been implemented on a network of workstations running Berkeley UNIX, and was developed at the University of Wisconsin-Madison [90, 112, 111, 110]. Its main objective is to take advantage of idle workstations by scheduling new processes to run on them. Each workstation is assumed to have an *owner*. Processes are allowed to run on workstations when their owners are not using them. When the owners start using their workstations, processes running on them are stopped and restarted on other workstations. To achieve process suspension and restart, Condor checkpoints processes periodically; when a process must be migrated, it is restarted from the last

checkpoint. The system is most effective for coarse-grain CPU-intensive parallel applications like simulations. Condor is currently being used by several universities and organizations.

4.5.2 Process Migration

To alleviate the overhead associated with checkpoint/restart methods, several systems provide a full-fledged process migration mechanism that saves and restores the state of a process only if the process is migrated. Sprite [35, 31, 33, 34, 32], Charlotte [9, 8, 7], LOCUS [121], the V system [150, 149], Accent [164, 163], DEMOS/MP [122], and AMOEBA [142] are among the operating systems that implement full-fledged process migration. There are several difficulties in implementing process migration: these include virtual memory copy, migration of open files, and migration of communication channels.

The most expensive part in process migration is usually the transfer of the virtual memory of the migrating process [35]. Charlotte and LOCUS transfer the process's entire memory image to the destination machine at migration time. Although simple, this method introduces unnecessary overhead if the process will not need its entire image in the rest of its execution. Moreover, since the transfer of an entire image (several Mbytes) may take several seconds (over a slow Ethernet channel), the process will be idle for that long period of time. To avoid the process being idle for that long, the V-system allows a process to continue execution on the workstations it was executing, while its image is being transferred to the new workstation. After the whole image is transferred, the process is suspended, the pages that have been modified since the image transfer started are transferred (again) as well, and the process is restarted at the new workstation. To avoid the overhead of copying the *entire* memory image of the migrated process, Accent uses a copy-on-reference approach. When a process is migrated, its pages are not copied; only the necessary information to create a new process control block on the destination workstation is transferred. When the process starts running on the new workstation and accessing its memory, it will page fault, and copy only the pages it needs. Sprite uses a similar method: when a process is migrated, its dirty pages are flushed to the disk where its swap space is stored. When the process is restarted on the new workstation it will cause a page-fault and bring from the swap file all the pages it needs.

The limited benefits of process migration: Even when implemented carefully, process migration is an expensive mechanism for load balancing. Eager *et. al.* [38] present the results of several simulation experiments that quantify the benefits of using process migration as a load sharing mechanism. They show "*no* conditions where migration could yield *major* performance improvements". The main reason is that load balancing can be achieved much easier by making good *initial placement* of new processes. When process migration is used only as a mechanism for process *eviction*, then its cost is low, and its benefits are high. For example, remote execution combined with process migration in the Sprite distributed operating system, has resulted in performance improvements of up to an order of magnitude for long running applications [35].

5 Summary

In this paper we present the problem of load balancing in parallel systems and its interactions with communication, synchronization and locality, and precedence constraints. We review load balancing methods at the application, compile/run time system and operating system levels.

Recent architecture trends suggest that computation gets faster at a higher pace than either synchronization or communication. Thus, load imbalance due to synchronization and communication constrains increases with time, which in turns magnifies the importance of load balancing in parallel and distributed systems, more than ever before.

We conclude that a careful balance between synchronization, communication and computation has to be done at all four levels of the system in order to improve the performance of parallel applications.

References

[1] M.P. Allen and D.J. Tildesley. Computer Simulations of Liquids. *Clarendon Press, Oxford*, 1987.

[2] J. M. Anderson and M. S. Lam. "Global Optimizations for Parallelism and Locality on Scalable Parallel Machines". In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*. The Stanford SUIF Compiler Group, June 1993.

[3] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism". *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[4] T. E. Anderson, E. D. Lazowska, and H. M. Levy. "The Performance Implications of Thread Management Alternatives for Shared Memory Multiprocessors". *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.

[5] A. W. Appel. An Efficient Program for Many-Body Simulation. *SIAM J. Sci. Stat. Comput.*, 6:85–103, 1985.

[6] Renzi H. Arpaci, Andrea C. Dusseau, Amin M. Vahdat, Lok T. Liu, Thomas E. Anderson, and David A. Patterson. The Interaction of Parallel and Sequential Workloads on a Network of Workstations. *Proceedings of the SIGMETRICS 1995*, May 1995.

[7] Y. Artsy, H.-Y. Chang, and R. Finkel. "Processes Migrate in Charlotte". Technical report, Univ. of Wisconsin, August 1986.

[8] Y. Artsy and R. Finkel. "Simplicity, Efficiency, and Functionality in Designing a Process Migration Facility". *Proceedings of the Second Israel conference on Computer Systems and Software Engineering*, May 1987.

[9] Y. Artsy and R. Finkel. "Designing a Process Migration Facility: The Charlotte Experience". *IEEE Computer*, 22(9):47–56, September 1989.

[10] Hamidzadeh B. and D. J. Lilja. Self-Adjusting Scheduling: An On-Line Optimization Technique for Locality Management and Load Balancing. *Proceedings of the 1994 International Conference on Parallel Processing*, pages II:39–46, 1994.

[11] U. Banerjee, R. Eigenmann, A. Nicolau, and D.A. Padua. "Automatic Program Parallelization". Technical Report TR-1250, Center for Supercomputing Research and Development (CSRD), 1993.

[12] J. E. Barnes and P. Hut. *Nature*, 324:446, 1986.

[13] D.M. Beazley and P.S. Lomdahl. Message passing multi-cell molecular dynamics on the Connection Machine 5. *Parallel Computing*, 1993.

[14] B. N. Bershad, E. D. Lazowska, H. M. Levy, and D. B. Wagner. "An Open Environment for Building Parallel Programming Systems". In *Proceedings of the ACM/SIGPLAN PPEALS 1988 Symposium on Parallel Programming: Experience with Applications, Languages, and Systems*, pages 1–9, July 1988.

[15] D. L. Black. "Scheduling Support for Concurrency and Parallelism in the Mach Operating System". *IEEE Computer*, 23(5):35–43, May 1990.

[16] M. A. Blumrich, C. J. Brownhill, K. Li, and A. Nicolau. "An Empirical Comparison of Loop Scheduling Algorithms on a Shared Memory Multiprocessor". Technical Report 360-92, Princeton Univ., January 1992.

[17] J. A. Board, Jr., J. W. Causey, J. F. Leathrum, Jr., A. Windemuth, and K. Schulten. Accelerated Molecular Dynamics Simulation with the Fast Multipole Algorithm. *Chem. Phys. Lett.*, 198:89, 1992.

[18] J. A. Board, Jr. and J. F. Leathrum,Jr. The Fast Multipole Algorithm on Transputer Networks. In Alan S. Wagner, editor, *Third North American Transputer Users Group Meeting*. IOS Press: Washington, DC, Apr 1990.

[19] R. B. Boppana. "Eigenvalues and graph bisection: An average case analysis". In *28th Foundation of Computer Science (ACM)*, pages 280–285, 1987.

[20] T. Brecht. *Multiprogrammed Parallel Application Scheduling in NUMA Multiprocessors.* PhD thesis, University of Toronto, Department of Computer Science, 1994.

[21] J. Carrier, L. Greengard, and V. Rokhlin. A Fast Adaptive Multipole Algorithm for Particle Simulation. *SIAM J. Sci. Stat. Comput.*, 9:669–686, 1988.

[22] N. P. Chrisochoides. *"On the mapping of PDE computations to distributed memory machines".* PhD thesis, Purdue Univ., 1991.

[23] N. P. Chrisochoides, C. E. Houstis, and E. N. Houstis. "Geometry based mapping strategies for PDE computation". Technical Report CER-90-16, Computer Science Department, Purdue Univ., West Lafayette, IN 47907, 1990.

[24] N. P. Chrisochoides, C. E. Houstis, E. N. Houstis, S. K. Kortesis, and J. R. Rice. "Automatic load balanced partitioning strategies for PDE computations". In E. N. Houstis and D. Gannon, editors, *Proceedings of International Conference on Supercomputing*, pages 99–107, New York, 1989. ACM Publications.

[25] N. P. Chrisochoides, E. N. Houstis, and J. R. Rice. "Mapping algorithms and software environment for Data Parallel PDE iterative Solvers". *Journal of Parallel and Distributed Computing*, 21:75–95, 1994.

[26] M. Crovella, P. Das, C. Dubnicki, T. LeBlanc, and E. Markatos. "Multiprogramming on Multiprocessors". *Proceedings of the Third Sysmposium on Parallel and Distributed Processing*, pages 590–597, December 1991. Also published as TR 385, URCSD February 1991 (revised May 1991).

[27] S. Graham D. Bacon and O. Sharp. Compiler transformations for high performance computing. *ACM Computer Surveys*, 26(4):345–420, December 1994.

[28] R. Das, J. Saltz, and R. V. Hanxleden. "Slicing Analysis and Indirect Access to Distributed Arrays". In *Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing*, pages 152–168, 1993.

[29] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang. "Communication Optimizations for Irregular Scientific Computations on Distributed Memory Architechtures". Technical Report UMIACS-TR-93-109, University of Maryland, Department of Computer Science, 1993.

[30] T. W. Doeppner Jr. "Threads: A System for the Support of Concurrent Programming". Technical Report CS-87-11, Department of Computer Science, Brown Univ., 1987.

[31] F. Douglis. "Experience with Process Migration in Sprite". *Proceedings of the First USENIX Workshop on Experiences Building Distributed and Multiprocessor Systems*, pages 59–72, October 1989.

[32] F. Douglis. "Transparent Process Migration in the Sprite Operating System". Technical Report 24723, Univ. of California - Berkeley, September 1990.

[33] F. Douglis and J. Ousterhout. "Process Migration in the Sprite Operating System". *Proc. 7-th Int. Conf. on Distr. Comp. Syst.*, pages 18–27, September 1987.

[34] F. Douglis and J. Ousterhout. "Process Migration in Sprite: A Status Report". *IEEE Comp. Soc. Techn. Comm. on Oper. Syst. and Appl. Env. Newsletter*, 3(1):8–10, winter 1989.

[35] F. Douglis and J. Ousterhout. "Transparent Process Migration: Design Alternatives and the Sprite Implementation". *Software Practice and Experience*, November 1989.

[36] T. H. Dunigan. Kendall Square Multiprocessor: Early Experiences and Performance. Technical Report ORNL/TM-12065, Oak Ridge National Laboratory, May 1992.

[37] S. Dutt and J. P. Hayes. "On Allocating Subcubes in a Hypercube Multiprocessor". In *Proc. 3-rd Hypercube Conf.*, pages 801–810, 1988.

[38] D. L. Eager, E. D. Lazowska, and J. Zahorjan. "The Limited Performance Benefits of Migrating Active Processes for Load Sharing". *Proceedings of SIGMETRICS 1988*, pages 63–72, May 1988.

[39] D. L. Eager and J. Zahorjan. "Adaptive Guided Self-Scheduling". Technical Report TR 92-01-01, University of Washington, Computer Science Department, January 1992.

[40] D. L. Eager and J. Zahorjan. Chores: Enhanced Run-Time Support for Shared-Memory Parallel Computing. *ACM Transactions on Computer Systems*, 11(1):1–32, February 1993.

[41] D. L. Eager, J. Zahorjan, and E. D. Lazowska. Speedup Versus Efficiency in Parallel Systems. *IEEE Transactions on Computers*, 38(3):408–423, March 1989.

[42] J. Edler, J. Lipkis, and E. Schonberg. "Process Management for Highly Parallel UNIX Systems". Technical Report 136, Ultracomputer Research Laboratory, New York Univ., April 1988.

[43] R. Eigenmann and P. McClaughry. "Practical Tools for Optimizing Parallel Programs". Technical Report TR-1276, Center for Supercomputing Research and Development (CSRD), 1994.

[44] Encore Computer Corporation. *"Multimax Technical Summary"*, 1987.

[45] F. Ercal, J. Ramanujam, and P. Sadayappan. "Task Allocation onto a Hypercube by Recursive Mincut Bipatitioning" . *Journal of Parallel and Distributed Computing*, 10:35–44, 1990.

[46] C. Farhat. "A simple and efficient automatic FEM domain decomposer". *Computers and Structures*, 28:579–602, 1988.

[47] C. Farhat, L. Fezoui, and S. Lanteri. "Computational fluid dynamics with irregular grids on the connection machine". Technical Report 1411, INRIA, Le Chesnay Cedex, France, 1991.

[48] D. G. Feitelson and L. Rudolph. "Distributed Hierarchical Control for Parallel Processing". *IEEE Computer*, 23(5):65–77, May 1990.

[49] D. G. Feitelson and L. Rudolph. "Mapping and Scheduling in a Shared Parallel Environment Using Distributed Hierarchical Control". *Proceedings of the 1990 International Conference on Parallel Processing*, pages I:1–8, August 1990.

[50] A. Feldmann, M. Y. Kao, J. Sgall, and S. H. Teng. "Optimal Online Scheduling of Parallel Jobs with Dependencies". Technical report, Carnegie-Mellon Univ., September 1992.

[51] J. Flower, S. Otto, and M. Salana. "Optimal mapping of irregular finite element domains to parallel processors". *Parallel Computers and Their Impact on Mechanics*, 86:239–250, 1988.

[52] R. J. Fowler and L. I. Kontothanassis. "Mercury: Object-Affinity Scheduling and Continuation Passing on Multiprocessors". In *Parallel Languages and Architecture Europe (PARLE 94). Athens.*, July 1994.

[53] G. C. Fox. "A review of automatic load balancing and decomposition methods for the hypercube". In M. H. Schultz, editor, *Proceedings of the IMA Institute*, pages 63–76. Springer–Verlag, 1986.

[54] M. R. Gary and D. S. Johnson. *"Computers and Intractability, A Guide to the Theory of NP-Completeness"*. 1979.

[55] E. Gelenbe and R. Kushwaha. Incremental Adaptive Load Balancing in Distributed Systems. 1993. submitted for publication.

[56] E. Gelenbe and R. Kushwaha. Dynamic Load Balancing in Distributed Systems. In *MASCOTS*. IEEE Computer Society, Jan 1994.

[57] A. George. "Nested dissection of a regular finite element mesh". *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973.

[58] A. George and J. W. Liu. "An automatic nested dissection algorithm for irregular finite element problems". *SIAM Journal on Numerical Analysis*, 15(5):1053–1069, 1978.

[59] A. George and D. R. McIntyre. "On the application of the minimum degree algorithm to finite element systems". *SIAM Journal on Numerical Analysis*, 15(1):1053–1069, 1978.

[60] L. Greengard. *The Rapid Evaluation of Potential Fields in Particle Systems*. PhD thesis, MIT Press: Cambridge, MA, 1988.

[61] L. Gross, C. Roll, and W. Schoenauer. "VECFEM for Mixed Finite Elements". Technical Report 50/93, Rechenzentrum der Universitat Karlsruhe, December 1993.

[62] A. Gupta, A. Tucker, and S. Urushibara. "The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications". In *Proceedings of the SIGMETRICS 1991*, pages 120–132, May 1991.

[63] Anoop Gupta, Andrew Tucker, and Luis Stevens. Making Effective Use of Shared-Memory Multiprocessors: The Process Control Approach. Technical Report CSL-TR-91-475A, Computer Systems Laboratory, Stanford UNIV, July 1991.

[64] S. Ha and E. A. Lee. "Compile-time scheduling and assignment of data-flow program graphs with data-dependent iteration". *IEEE Transactions on Computers*, 40(11):1225–1238, November 1991.

[65] R. V. Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. "Compiler Analysis for Irregular Problems in Fortran-D". In *"Proceedings of 5th Workshop on Languages and Compilers for Parallel Computing*, 1992.

[66] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines. *Communications of the ACM*, 35(8):67–80, August 1992.

[67] J. Hopfield. "Neural networks and physical systems with emergent collective computational abilities". In *Proc. Natl. Acad. Sci., USA*, volume 79, pages 2554–2558, 1982.

[68] C. E. Houstis. "Module Allocation of Real-Time Applications to Distributed Systems". *IEEE Transactions on Software Engineering*, 5(7):699–709, July 1990.

[69] E. N. Houstis, S. K. Kortesis, and H. Byun. "A workload partitioning strategy for PDEs by a generalized neural network". Technical Report CSD-TR-934, Purdue Univ., W. Lafayette, IN, 1990.

[70] E. N. Houstis, J. R. Rice, N. P. Chrisochoides, H. C. Karathonases, P. N. Papachiou, M. K. Samartzis, E. A. Vavalis, K. Y. Wang, and S. Weerawarana. "ELLPACK: A numerical simulation programming environment for parallel MIMD machines". *Supercomputing*, pages 3–23, 1990.

[71] S. F. Hummel, E. Schonberg, and L. E. Flynn. "Factoring: A Method for Scheduling Parallel Loops". *CACM*, 35(8):90–101, August 1992.

[72] J. J. Hwang, F. D. Anger, Y. C. Chow, and C. Y. Lee. "Scheduling Precedence Graphs in Systems with Interprocessor Communication Times". *SIAM Journ. Comp.*, 18(2):244–257, April 1989.

[73] L. Johnston and C.-T. Ho. "Optimum broadcasting and personalized communication in hypercubes". *IEEE Trans. Comput.*, 38:1248–1268, 1989.

[74] B. W. Kernighan and S. Lin. "An efficient heuristic procedure for partitioning graphs". *The Bell System Technical Journal*, pages 291–307, 1970.

[75] J. Kim, C. R. Das, and W. Lin. "A Processor Allocation Scheme for Hypercube Computers". In *Proc. Int. Conf. on Parallel Processing*, volume 2, pages 231–238, 1989.

[76] S. Kirkpatrick, C. Gelatt, and M. Vecchi. "Optimization by simulated annealing". *Science*, 220:671–680, 1983.

[77] C. P. Kruskal and A. Weiss. "Allocating Independent Subtasks on Parallel Processors". *IEEE Transactions on Software Engineering*, 11(10):1001–1016, 1985.

[78] K. Kunchithapadam and B. P. Miller. "Optimizing Array Distributions in Data-Parallel Programs". In *7th Annual Workshop on Languages and Compilers for Parallel Computing*, Ithaca, NY, August 1994.

[79] J. F. Leathrum, Jr. and J. A. Board, Jr. Parallelization of the Fast Multipole Algorithm using the B012 Transputer Network. In *Transputing*. IOS Press: Washington, DC, 1991.

[80] James F. Leathrum, Jr. and John A. Board, Jr. Mapping the adaptive fast multipole algorithm onto MIMD systems. *Unstructured Scientific Computation on Scalable Multiprocessors*, pages 161–178, 1992.

[81] C. Y. Lee, J. J. Hwang, Y. C. Chow, and F. D. Anger. "Multiprocessor Scheduling with Interprocessor Communication Delays". *Operations Research Letters*, 7(3):141–147, June 1988.

[82] D. Lenoski, J. Laudon, L. Stevens, T. Joe, D. Nakahira, A. Gupta, and J. Hennessy. The DASH Prototype: Implementation and Performance. *Proceedings of the Nineteenth International Symposium on Computer Architecture*, pages 92–103, May 1992.

[83] S. Leutenegger. "Issues in Multiprogrammed Multiprocessor Scheduling". Technical Report 954, University of Wisconsin-Madison, August 1990. Ph.D. dissertation.

[84] S. C. Leutenegger and M. K. Vernon. "The Performance of Multiprogrammed Multiprocessor Scheduling Policies". *Performance Evaluation Review*, 18(1):226–236, May 1990.

[85] H. Li, S. Tandri, M. Stumm, and K. C. Sevcik. "Locality and Loop Scheduling on NUMA Multiprocessors". *Proc. Int. Conf. on Parallel Processing*, August 1993.

[86] K. Li and K. H. Cheng. "Static Job Scheduling in Partitionable Mesh Connected Systems". *Journal of Parallel and Distributed Computing*, 10(2):152–159, October 1990.

[87] D. J. Lilja. "Exploiting the Parallelism Available in Loops". *IEEE Computer*, 27(2):13–26, February 1994.

[88] B. Lisper and J. F. Collard. "Extent Analysis of Data Fields". Technical Report 03, Royal Inst Tech (Sweden), Teleinformatics, 1994.

[89] M. Litzkow. "Remote UNIX: Turning Idle Workstations Into Cycle Servers". In *Proceedings of the USENIX 1987 Summer Conference*, pages 381–384, june 1987.

[90] M. J. Litzkow, M. Livny, and M. W. Mutka. "Condor - A Hunter of Idle Workstations". In *Proc. 8-th Int. Conf. on Distr. Comp. Syst.*, 1988.

[91] W. H. Liu and A. H. Sherman. "Comparative analysis for the Cuthill-McKee and the reverse Cuthill-McKee ordering algorithms for sparse matrices". *SIAM Journal on Numerical Analysis*, 13(2):199–213, 1976.

[92] S.-P. Lo and V. D. Gligor. "A Comparative Analysis of Multiprocessor Scheduling Algorithms". *Proc. 7-th Int. Conf. on Distr. Comp. Syst.*, pages 356–363, September 1987.

[93] S.-P. Lo and V. D. Gligor. "Properties of Multiprocessor Scheduling Algorithms". *Proceedings of the 1987 International Conference on Parallel Processing*, August 1987.

[94] V. M. Lo. "Process Migration for Communication Performance". *IEEETCOS*, 3(1):28–30, Winter 1989.

[95] P.S. Lomdahl, P. Tamayo, N. Grönbech-Jensen, and D.M. Beazley. 50 GFlops molecular dynamics on the Connection Machine 5. Technical Report LA-UR-93-3078, Los Alamos National Laboratory, Los Alamos, N.M., 1993.

[96] S. Lucco. "A Dynamic Scheduling Method for Irregular Parallel Programs". *ACM SIGPLAN 92 CONF on Programming Language Design and Implementation*, pages 200–211, June 1992.

[97] N. K. Madsen and R. F. Sincovec. "ALGORITHM 540: PDECOL General Allocation Software for Partial Differential Equations". *ACM Transactions on Mathematical Software*, 5(3):326–351, September 1979.

[98] N. Mansour. *"Physical optimization algorithms for mapping data to distributed-memory multiprocessors"*. PhD thesis, Syracuse Univ., 1992.

[99] L. D. Marini and A. Quarteroni. "An iterative procedure for Domain Decomposition Methods : A finite element approach". *First International Symposium on Domain Decomposition Methods for Partial Differential Equations*, 1988.

[100] E. Markatos, M. Crovella, P. Das, C. Dubnicki, and T. LeBlanc. "The Effects of Multiprogramming on Barrier Synchronization". In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 662–669, December 1991.

[101] E. Markatos and T. J. LeBlanc. "Locality-Based Scheduling in Shared-Memory Multiprocessors". In Albert Zomaya, editor, *"Parallel Computations: Paradigms and Applications"*. International Thomson Computer Press, 1996. Also appeared as Techinal Report 94, Institute of Computer Science, FORTH, Greece, August 1993.

[102] E. P. Markatos and T. J. LeBlanc. "Load Balancing Versus Locality Management in Shared-Memory Multiprocessors". In *Proc. Int. Conf. on Parallel Processing*, pages I:258–267, St. Charles, IL, August 1992.

[103] E. P. Markatos and T. J. LeBlanc. "Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors". *IEEE Transactions on Parallel and Distributed Systems*, 5(4):379–400, April 1994.

[104] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. "First-Class User-Level Threads". In *Proceedings 13th Symposium on Operating Systems Principles*, pages 110–121, October 1991.

[105] C. McCann, R. Vaswani, and J. Zahorjan. "A Dynamic Processor Allocation Policy for Multiprogrammed Shared Memory Multiprocessors". *ACM Transactions on Computer Systems*, 11(2):146–178, May 1993.

[106] McClaughry and P. Earl. "PTOPP - A Practical Toolset for the Optimization of Parallel Programs". Technical Report TR-1225, Center for Supercomputing Research and Development (CSRD), 1993.

[107] A.I. Mel'cuk, R.C. Giles, and H. Gould. Computers in Physics. page 311, 1991.

[108] D. K. Melgaard and R. F. Sincovec. "General software for two-dimensional nonlinear partial differential equations". *ACM Transactions on Mathematical Software*, 7(1):106–125, March 1981.

[109] W. F. Mitchell. "Adaptive refinement for arbitrary finite element spaces with hierarchical bases". *Journal on Computational and Applied Math.*, 36:65–78, 1991.

[110] M. W. Mutka. "Estimating Capacity For Sharing in a Privately Owned Workstation Environment". *IEEE Transactions on Software Engineering*, 18(4):319–328, April 1992.

[111] M. W. Mutka and M. Livny. "Profiling Workstations' Available Capacity for Remote Execution". In *Performance*, 1987.

[112] M. W. Mutka and M. Livny. "Scheduling Remote Processing Capacity in a Workstation-Processor Bank Network". In *Proc. 7-th Int. Conf. on Distr. Comp. Syst.*, pages 2–9, 1987.

[113] S. T. Nguyen, B. J. Zook, and X. Zhang. "Distributed computation of electromagnetic scattering problems using finite-difference time-domain decompositions". In *IEEE International Symposium on High-Performance Distributed Computing, IEEE CS Press*, August 1994.

[114] M. O'Boyle. "A Data Partioning Algorithm for Distributed Memory Compilation". Technical Report UMCS-93-7-1, Department of Computer Science, University of Manchester, July 1993.

[115] S. Orlando and R. Perego. Exploiting Partial Replication in Unbalanced Parallel Loop Scheduling on Multicomputers. *Microprocessing and Microprogramming, Elsevier Science Publisher, special issue on Parallel System Engineering*, 1996.

[116] J. K. Ousterhout. "Scheduling Techniques for Concurrent Systems". In *Proc. 2-nd Int. Conf. on Distr. Comp. Syst.*, pages 22–30, 1982.

[117] J. K. Ousterhout, D. A. Scelza, and P. S. Sindu. "Medusa - An Experiment in Distributed Operating System Structure". *Communications of the ACM*, 23:92–105, February 1980.

[118] C. D. Polychronopoulos and D. J. Kuck. "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers". *IEEE Transactions on Computers*, C-36(12), December 1987.

[119] C. Pommerell, M. Annaratone, and W. Fichtner. "A set of new mapping and coloring heuristics for distributed-memory parallel processors". In *Proceedings of Copper Mountain Conference on Iterative Methods*, volume 4, pages 1–27, 1990.

[120] R. Ponnusamy, J. Saltz, and A. Choudhary. "Runtime-Compilation Techniques for Data Partitioning and Communication Schedule Reuse". In *Proceedings Supercomputing '93*, pages 361–370, 1993.

[121] G. J. Popek and B. J. Walker. *"The LOCUS Distributed System"*. The MIT Press, 1985.

[122] M. L. Powell and B. P. Miller. "Process Migration in DEMOS/MP". In *Proc. 6-th ACM Symp. on Operating System Principles*, pages 110–119, November 1983.

[123] O. Plata F. F. Rivera. Combining Static and Dynamic Scheduling on Distributed-Memory Multiprocessors. *1994 ICS*, pages 186–195, July 1994.

[124] V. F. Russo. *"An Object-Oriented Operating System"*. PhD thesis, Univ. of Illinois at Urbana-Champaign, 1991.

[125] V. F. Russo. "Process Scheduling and Synchronization in the Renaissance Object-Oriented Multiprocessor Operating System". In *Proceedings of the Second Symp. on Experiences with Distributed and Multiprocessor Systems*, pages 117–132, Atlanta, GA, March 1991.

[126] V. Sarkar. "Partitioning and Scheduling for Execution on Multiprocessors". Technical report, Stanford Univ., April 1987.

[127] V. Sarkar and J. Hennessy. "Compile-time Partitioning and Scheduling of Parallel Programs". In *SIGPLAN '86 Symposium on Compiler Construction*, pages 17–26, Palo Alto, CA, June 1986. acm, SIGPLAN.

[128] K. E. Schauser, D. E. Culler, and T. von Eicken. "Compiler-Controlled Multithreading for Lenient Parallel Languages". In *Proceedings of the '5th ACM Conference on Functional Programming Languages and Computer Architecture, Cambridge, MA*, pages 50–72, New York, 1991. Springer-Verlag.

[129] M. Schmauder, R. Weiss, and W. Schoenauer. "The Cadsol Program Package". Technical Report 46/92, Rechenzentrum der Universitat Karlsruhe, April 1992.

[130] W. Schoenauer, E. Schnepf, and H. Mueller. "The Fidisol Program Package". Technical Report 27/85, Rechenzentrum der Universitat Karlsruhe, December 1985.

[131] M. L. Scott, T. J. LeBlanc, B. D. Marsh, T. G. Becker, C. Dubnicki, E. P. Markatos, and N. G. Smithline. "Implementation Issues for the Psyche Multiprocessor Operating System". *Computing Systems*, 3,(1):101–137, winter 1990.

[132] Sequent Computer Systems Inc. *"Balance 8000 System"*, 1985.

[133] Sequent Computer Systems Inc. *"Symmetry Multiprocessor Architecture Overview"*, 1991.

[134] K. C. Sevcik. Application Scheduling and Processor Allocation in Multiprogrammed Multiprocessors. *Performance Evaluation*, 9(2-3):107–140, 1994.

[135] K. C. Sevcik. Characterizations of Parallelism in Applications and their Use in Scheduling. *Performance Evaluation Review*, 17(1):171–180, May 1989.

[136] D. H. Simon. "Partitioning of unstructured problems for parallel processing". Technical Report RNR-91-008, NASA Ames Research Center, Moffet Field, CA 94035, 1990.

[137] R. F. Sincovec and N. K. Madsen. "Software for nonlinear partial differential equations". *ACM Transactions on Mathematical Software*, 1(3):232–260, September 1975.

[138] B. Smith. "Architecture and Applications of the HEP Computer System". In *Proceedings of the SPIE, Real-Time Signal Processing IV*, 1981.

[139] D. G. Socha. "Supporting fine-grain computation on distributed memory parallel computers". Technical Report 91-07-01, Univ. of Washington, 1991.

[140] M. S. Squillante. "Issues in Shared-Memory Multiprocessor Scheduling: A Performance Evaluation". Technical Report 90-10-04, University of Washington Computer Science Department, October 1990. Ph.D. dissertation.

[141] M. S. Squillante and E. D. Lazowska. "Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling". *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131–143, February 1993.

[142] C. Steketee, W. P. Zhu, and P. Moseley. "Implementation of Process Migration in Amoeba". *Proc. 14-th Int. Conf. on Distr. Comp. Syst.*, June 1994.

[143] J. Stichnoth, D. O'Hallaron, and T. Gross. "Generating communication for array statements: Design, implementation, and evaluation". *Journal of Parallel and Distributed Computing*, 21(1):150–159, 1994.

[144] Q. Stout and B. Wagar. "Intensive hypercube communication". *Parallel Distribut. Comput.*, 10:167–181, 1990.

[145] J. Subhlok, D. O'Hallaron, T. Gross, P. Dinda, and J. Webb. "Communication and memory requirements as the basis for mapping task and data parallel programs". In *Proc. Supercomputing '94*, 1994.

[146] P. Tamayo, J.P. Mesirov, and B.M. Boghosian. In *Supercomputing*. IEEE Computer Society Press, 1991.

[147] P. Tang and P.-C. Yew. "Processor Self-Scheduling for Multiple Nested Parallel Loops". In *Proceedings 1986 International Conference on Parallel Processing*, pages 528–535, August 1986.

[148] C. P. Thacker and L. C. Stewart. Firefly: A Multiprocessor Workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

[149] M. Theimer. *"Preemtable Remote Execution Facilities for Loosely-Coupled Distributed Systems"*. PhD thesis, Stanford Univ., 1986.

[150] M. Theimer, K. Lantz, and D. Cheriton. "Preemptable Remote Execution Facilities for the V-System". In *"Proceedings of the Eleventh ACM Symposium on Operating System Principles"*, pages 13–22, November 1987.

[151] R. H. Thomas and W. Crowther. "The Uniform System: An Approach to Runtime Support for Large Scale Shared Memory Parallel Processors". In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 245–254, August 1988.

[152] J. Torrellas, A. Tucker, and A. Gupta. "Evaluating the Benefits of Cache-Affinity Scheduling in Shared-Memory Multiprocessors". Technical report, Computer Systems Laboratory, Stanford Univ., August 1992.

[153] A. Tucker and A. Gupta. "Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors". *Proceedings of the Symposium on Operating Systems Principles*, pages 159–166, December 1989.

[154] T. H. Tzen and L. M. Ni. "Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Computers". *IEEE Transactions on Parallel and Distributed Systems*, 4(1):87–98, January 1993.

[155] Z. Vranesic, S. Brown, M. Stumm, S. Caranci, A. Grbic, R. Grindley, M. Gusat, O. Krieger, G. Lemieux, K. Loveless, N. Manjikian Z. Zilic, T. Abdelrahman, B. Gamsa, P. Pereira, K. Sevcik, A. Elkateeb, and S. Srbljic. The NUMAchine Multiprocessor. Technical report, University of Toronto, 1995.

[156] Michael S. Warren and John K. Salmon. A Parallel Hashed Oct-Tree N-Body Algorithm. In *Supercomputing*, pages 12–21. IEEE Computer Society, Washington, 1993.

[157] R. D. Williams. "Performance of Dynamic Load Balancing Algorithms for Unstructured Mesh Calculations". Technical Report C3P913, Concurrent Supercomputing Facility, California Institute of Technology, Pasadena, CA, June 1990.

[158] R. W. Wisniewski, L. I. Kontothanassis, and M. L. Scott. High Performance Synchronization Algorithms for Multiprogrammed Multiprocessors. *Proceedings of the Fifth PPOPP*, July 1995.

[159] Robert W. Wisniewski, Leonidas Kontothanassis, and Michael L. Scott. Scalable Spin Locks for Multiprogrammed Systems. *Proceedings of the Eighth International Parallel Processing Symposium*, pages 583–589, April 1994.

[160] T. Yang. "Scheduling and Code Generation for Parallel Architectures (Ph.D. Dissertation)". Technical Report DCS-TR-299, Rutgers Univ., Laboratory for Computer Science Research, New Brunswick, NJ, 1993.

[161] J. Zahorjan, E. D. Lazowska, and D. L. Eager. "The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Systems". *IEEE Transactions on Parallel and Distributed Systems*, 2(2):180–198, April 1991.

[162] J Zahorjan and C. McCann. "Processor Scheduling in Shared Memory Multiprocessors". *Performance Evaluation Review*, 18(1):214–225, May 1990.

[163] E. Zayas. *"The Use of Copy-on-Reference in a Process Migration System"*. PhD thesis, Carnegie Mellon Univ., Pittsburgh, PA, April 1987.

[164] E. R. Zayas. "Attacking the Process Migration Bottleneck". In *Proc. 11-th ACM Symp. on Operating System Principles*, pages 13–24, 1987.

[165] S. Zhou and T. Brecht. Processor-pool-based Scheduling for Large-Scale NUMA Multiprocessors. In *Proceedings of SIGMETRICS '91*, San Diego, CA, May 1991.