Campos, Luis Miguel; Silva, Fabricio; Scherson, Isaac; Eshaghian, Mary
An Information Power Grid Resource Management Tool
Instituto Politécnico Nacional
Distrito Federal, México

Available in: http://www.redalyc.org/articulo.oa?id=61540303
An Information Power Grid Resource Management Tool

Luis Miguel Campos¹, Fabricio Silva¹, Isaac Scherson¹ and Mary Eshaghian²

¹ Dept. of Information and Computer Science
University of California-Irvine
Irvine, CA 92697, USA.
e-mail: {lcampos, fsilva, isaac}@ics.uci.edu
² Department of Computer Engineering
Rochester Institute of Technology
Rochester, NY 14623-5603
e-mail: mmeec@grace.rit.edu

Article received on February 15, 2000; accepted on August 23, 2000

Abstract
Heterogeneous Computing (HC) is defined as a special form of parallel and distributed computing. Computations are carried out using a single autonomous computer operating in both SIMD and MIMD modes, or using a number of connected autonomous computers (a.k.a. Cluster Computing). Information Power Grid (IPG) is a form of HC in which high performance computers located at geographically distributed sites are connected via a high-speed interconnection network. It is desirable to make IPG accessible to general users as easily and seamlessly as electricity from the electric power grid. The users should be able to submit jobs at any site, the IPG should be able to handle the computations using the resources available, and return the results to the user. One of the many challenges in making IPG work is the issue of Resource Management. We present a Resource Management Tool for IPG called IPG-SaLaD (Static mapper, and Load balancer, and Dynamic scheduler). This tool uses the Cluster-M mapping paradigm for initial allocation of tasks of a given job onto resources. Later as new jobs come in or as the status of the system changes, the tool uses the Rate of Change algorithm for load balancing, and the Concurrent Gang Scheduling for scheduling. Currently a window-NT based implementation of the IPG-SaLaD using RVM is being constructed.

1 Introduction and Background

Since the early 90’s, significant amount of attention has been given to the type of computing in which multiple computers are used concurrently in solving single problems. These computers may be of different type and brand with different operating systems and capabilities. Furthermore, they may be geographically distributed. A number of different terms have been used for introducing this concept such as Heterogeneous Computing, Cluster Computing, Meta Computing, Wide-area Computing, and recently Information Power Grid (IPG) computing.

In an IPG environment a national computing infrastructure allows users to access the information resources of the nation in much the same way as they access electrical power today.

IPG is a topic that has been of interest to the DoD and several other organizations such as DOD, NSF. To this date, a number of sites have taken part in IPG, such as Caltech/JPL, ISI/USC, SDSC, and NCSA. IPG offers many benefits such as enabling collaborative research among geographically dispersed teams in a virtual environment, enabling solutions of problems that could not be done otherwise, and cost and time savings through optimization of scarce computing resources. The design issue

This tool uses the Cluster-M mapping paradigm for initial allocation of tasks of a given job onto resources. Later as new jobs come in or as the status of the system changes, the tool uses the Rate of Change algorithm for load balancing, and the Concurrent Gang Scheduling for scheduling. Currently a window-NT based implementation of the IPG-SaLaD using RVM is being constructed.
ution environments that are portable and scalable. This includes design of efficient resource management techniques, data storage and migration techniques. Tools need to be developed to enable the use of the execution environment such as automated tools for porting legacy code, collaborative problem solving environments, formal, portable programming paradigms, languages and tools that express parallelism and support synthesis and reuse. These execution environments should support applications for the future such as application software that uses 1000 or 10000 processors. Furthermore, the execution environment, user environment and applications all need to be integrated.

The focus of our work is in the area of execution environment design. More specifically we will concentrate on an IPG tool for resource management. The proposed tool called SaLaD (Static allocator, and Load balancer, and Dynamic scheduler), uses a three step mechanism. During the first step, using the Cluster-M static allocator, it will take a program-task (or so called a job) that is entered by the user (or by a compiler) and will break it into subtasks and will map/allocate the subtasks onto available processors and or computers so that overall execution time is minimized. As new jobs arrive and/or finish execution, it maybe required to redistribute individual subtasks using the Rate of Change load balancer. This step guarantees that no processing element ever goes idle. Finally the Concurrent Gang dynamic scheduler is used to schedule the execution sequence of multiple independent jobs residing on the processors. The optimal use of resources, and resource allocation based on the workload contents and site specific capabilities is consistent with the IPG objectives specified by NASA. In this paper, we will present our preliminary results towards the development of this integrated resource management tool. Currently a simple version which is NT-window-based and uses PVM is operational.

The rest of the paper is organized as follows. In the remaining part of this section we briefly present a brief background on the three areas studied in this paper, namely, static allocation, load balancing and dynamic scheduling. In the next section, we present 1.1 Static Allocation of subtasks to a single job

The mapping problem, in its general form, is known to be NP-complete and has been extensively for homogeneous parallel computing. For the past two decades [Ber87, Bok81, ERL90, ES94, LA87, LRG+90, PSD+93], mapping, an application task and a component are usually modeled in terms of a task graph and a system graph. The problem, then, is how to map efficiently the task flow graph to the system graph. A task flow graph is a directed acyclic graph (DAG) that consists of a set of vertices and directed edges. A vertex denotes a task, composed from the given task. Each vertex is associated with a weight that denotes the amount within the corresponding task of a directed edge joining two task modules indicates data communication and dependency between the two task modules. The weight of a vertex represents the amount of data communication. The task flow graph is usually directed, the system graph is usually undirected. A set of vertices in a system graph denote processors and a set of directed edges indicate physical communication links for processor pairs. The weight of a vertex represents the speed (bandwidth) of the corresponding processor (communication link). The graph as nonuniform if and only if the weights of all vertices or the weights of all edges are the same; otherwise it is uniform.

Mapping can be static or dynamic. In mapping, the assignments of the nodes on the system graphs onto the system graphs are determined in advance to the execution and are not changed during the execution. Static mapping can be classified into two general ways. The first one is based on the topology of task and system graphs [CE95]. Based on this, the mapping is classified into four groups: (1) mapping tasks onto specialized systems, (2) mapping tasks onto arbitrary systems, (3) mapping arbitrary tasks onto specialized systems and (4) mapping arbitrary tasks onto arbitrary systems. The second classification can be based on the similarity of the weights of the nodes and edges in the two graphs, as well as the number of vertices of the graphs.
[ST85, Lo88]. In IPG, the task and system graphs can be nonuniform. Therefore, the mapping problem in HC can be viewed as mapping of an arbitrary nonuniform task graph onto an arbitrary nonuniform system graph.

In this paper, we first concentrate on static mapping of arbitrary nonuniform task graphs onto arbitrary nonuniform system graphs. The existing mapping techniques in this group include El-Rewini and Lewis’ mapping heuristic algorithm [ERL90] for directed task graphs and Lo’s max flow min cut mapping heuristic [Lo88] for undirected task graphs. The time complexity of these two heuristics are $O(M^2N^3)$ and $O(M^4N \log M)$, respectively, where $M$ is the number of task modules and $N$ is the number of processors. Another algorithm in this category is called Cluster-M which can map arbitrary, nonuniform, architecturally independent, directed task graphs onto arbitrary, nonuniform, undirected, task-independent system graphs in $O(M^2)$ time, where $N \leq M$. Cluster-M will be explained in detail in 3.1.

1.2 Load Balancing by Redistribution of Subtasks

Dynamic Load Balancing (DLB) is an important system function aimed at distributing workload among available processors to improve throughput and/or execution times of parallel computer programs either uniform or non-uniform (jobs whose workload varies at run-time in unpredictable ways). Non-uniform computation and communication requirements may bog down a parallel computer if no efficient load distribution is effected.

Load balancing strategies fall broadly into either one of two classifications, namely static or dynamic. A multicomputer system with static load balancing distributes tasks across the processing elements (PEs) before execution using a priori known task information and the load distribution remains unchanged at run time. A multicomputer system with DLB uses no a priori task information, and must satisfy changing requirements by making task distribution decisions during run-time. DLB can be further classified as centralized or distributed. In a centralized strategy, load balancing decisions are made by

sponsibility of achieving global load balance. Other factor often used to classify load balancing strategies, implicit or explicit, is strictly speaking a system issue rather than a load balancing one. Implicit load balancing refers to load balancing performed automatically by the system, whereas explicit means it is up to the user to decide which tasks should be migrated and when.

Many Dynamic Load Balancing strategies have been proposed in the literature. Three of the most known are: the gradient model [LK91], the sender initiated diffusion [WLR93], and the central job patcher [LHWS96]. The gradient model employs a gradient map of the proximities of under-loaded processors in the system to guide the migration of jobs between loaded and under-loaded processors. The sender initiated diffusion is a highly distributed local approach which makes use of near-neighbor load information to apportion surplus load to heavily loaded processors to under-loaded neighbors in the system. Global balancing is achieved by assigning tasks from heavily loaded neighborhoods to lightly loaded areas in the system. In the central dispatcher strategy, one of the network processors acts as a central load balancing master. The dispatcher maintains a table containing the number of waiting tasks in each processor. Whenever a job arrives at or departs from a processor, the processor notifies the central dispatcher of its new load status. When a state change message is received or a job transfer decision is made, the central dispatcher updates the table accordingly. The network bases its balancing on this table and notifies the most heavily loaded processor to transfer tasks to a requested processor. The network also notifies the requesting processor of the decision.

1.3 Dynamic Scheduling of Multiple Jobs or Tasks

Parallel job scheduling is an important problem whose solution may lead to better utilization of modern parallel computers. It is defined as: “Given an aggregate of all tasks of multiple jobs in a parallel system, find a spatial and temporal allocation that will execute all tasks efficiently”. Each job in a parallel chip is composed by one or more tasks. For the
ing), while in others, like time slicing, the arriving job receives service immediately through a processor sharing discipline.

Gang scheduling has been widely used as a practical solution to the dynamic parallel job scheduling problem. Parallel tasks of a job are scheduled for simultaneous execution on a partition of a parallel computer. Gang Scheduling has many advantages, such as responsiveness, efficient sharing of resources and ease of programming. However, there are two major problems associated with gang scheduling: scalability and the decision of what to do when a task blocks.

2 Preliminaries

The IPG tool which is presented in this paper, called SalAD consists of three main components. In the next section we will describe how they work together in an integrated fashion. In this section, we present preliminary background on each of these components namely the Static Allocator, Load Balancer, and the Dynamic Scheduler. The static allocation technique used is based on the Cluster-M technique. Load Balancing is achieved using the Rate of Change algorithm, and Dynamic Scheduling is performed using a generalization of gang-scheduling dubbed Concurrent Gang.

2.1 Cluster-M Static Allocation

Cluster-M is a programming tool that facilitates the static allocation and mapping of portable parallel programs [CE95]. Cluster-M has three main components: the specification module, the representation module and the mapping module. In the specification module, machine-independent algorithms are specified and coded using the Program Composition Notation (PCN [FT93]) programming language [ES94]. Cluster-M specifications are represented in the form of a multi-layer clustered task graph called Spec graph. Each clustering layer in the Spec graph represents a set of concurrent computations, called Spec clusters. A Spec graph can also be obtained by applying one of the appropriate Cluster-M clustering algorithms to any given task graph. A Cluster-M representation is a set of multi-layer partitioning (task) graphs. It is a machine-independent (application-independent) clustering, therefore not necessary to be repeated for different machines. For this reason, the time complexities of the clustering algorithms are not included in the complexity of the Cluster-M mapping algorithm. If the mapping module, a given Spec graph is mapped onto a given Rep graph. In an earlier paper [CE95, Esh96] two Cluster-M clustering algorithms and a mapping algorithm were presented, for both uniform and non-uniform graphs.

Below, we present a set of experimental results. The following criteria are used for comparison of our algorithm with other techniques: (1) the total time for executing a mapping algorithm, $T_{e}$; (2) the total execution time of the generated mappings, $T_{m}$; (3) the number of processors used, $N_{m}$; and (4) the total time for both clustering (task and system) and mapping algorithms, $T_{cm}$. From (2) and (3), we can compute the speedup $S_{m} = \frac{T_{e}}{N_{m}}$ and efficiency $\eta = \frac{S_{m}}{N_{m}}$, where $T_{e}$ is the sequential execution time of the task graph.

In Table 1, comparison results are shown for mapping nonuniform random task graphs ranging from 100 to 1000 nodes onto the random system of size 100, where the speed of the process-communication channels is ranged between 10 and 100. In other words, even though there are 100 processors, but they are very slow, and therefore the speedup is expected to be low. What is really important is that both MH and Cluster-M lead to much better speedups than MH both asymptotically and experimentally. The running time of MH grows significantly as the size of the task graph grows. Whereas the running time of Cluster-M remains relatively stable, in large cases, Cluster-M obtains a better speedup. But in all cases Cluster-M has a significant advantage in time complexity.

The experimental results shown in the paper were obtained by running a set of simulations on a SUN UltraSPARC I workstation, and all times $(T_{e}, T_{cm})$ are measured in milliseconds. The non-uniform task graphs are randomly generated.

2.2 Rate of Change Load Balancing
one load unit at all times. The rationale is that if any given PE is busy executing tasks then the load differential with respect other PEs is irrelevant since no performance gain can be obtained by transferring load. Hence the decision to initiate load transfers should not depend on a PE's absolute number of load units, but on how the load changes in time.

Our novel approach uses the Rate of Change (RoC) of the load on each PE to trigger any load balancing activity. It can be described as a dynamic, distributed, on demand, preemptive and implicit load balancing strategy. Dynamic, because it does not assume any prior task (unit of load in this study) information and must satisfy changing requirements by making task distribution decisions at run-time. Distributed, because all load balancing decisions are made locally and asynchronously by each processor. On demand, because only PEs that "need" tasks are allowed to initiate any migration activity. Preemptive, because running tasks may be suspended, moved to another PE and restarted. Implicit, because all load balancing activity is done by the system, without user assistance.

Furthermore, our proposed RoC-Load Balancing strategy achieves the goal of minimizing processor idling times without incurring into unacceptably high load balancing overhead. It does so by striking a balance between the cost of evaluating load information which now is a local activity to each PE, and the cost of transferring tasks across the system.

Experimental results have been obtained and are:

<table>
<thead>
<tr>
<th>Size of Random Task Graph</th>
<th>T_{m}</th>
<th>T_{s}</th>
<th>T_{cm}</th>
<th>T_{e}</th>
<th>T_{m}</th>
<th>T_{s}</th>
<th>T_{cm}</th>
<th>T_{e}</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>286</td>
<td>88.80</td>
<td>3.22</td>
<td>831.4</td>
<td>4.5</td>
<td>95.8</td>
<td>2.99</td>
<td>606.49</td>
</tr>
<tr>
<td>200</td>
<td>630</td>
<td>133.20</td>
<td>4.73</td>
<td>1056.6</td>
<td>4.5</td>
<td>231.8</td>
<td>2.72</td>
<td>24380.1</td>
</tr>
<tr>
<td>300</td>
<td>855</td>
<td>345.55</td>
<td>2.47</td>
<td>1440.6</td>
<td>4.7</td>
<td>240.25</td>
<td>3.56</td>
<td>55305.5</td>
</tr>
<tr>
<td>400</td>
<td>1162</td>
<td>478.40</td>
<td>2.43</td>
<td>1915.9</td>
<td>4.5</td>
<td>496.30</td>
<td>2.34</td>
<td>98888.8</td>
</tr>
<tr>
<td>500</td>
<td>1514</td>
<td>550.80</td>
<td>2.75</td>
<td>2547.9</td>
<td>4.5</td>
<td>458.07</td>
<td>3.31</td>
<td>153099.0</td>
</tr>
<tr>
<td>600</td>
<td>1793</td>
<td>358.20</td>
<td>5.01</td>
<td>3436.1</td>
<td>4.9</td>
<td>599.07</td>
<td>2.99</td>
<td>222381.1</td>
</tr>
<tr>
<td>700</td>
<td>2075</td>
<td>690.85</td>
<td>3.00</td>
<td>4334.0</td>
<td>4.7</td>
<td>685.57</td>
<td>3.03</td>
<td>299372.2</td>
</tr>
<tr>
<td>800</td>
<td>2376</td>
<td>474.00</td>
<td>5.01</td>
<td>5621.3</td>
<td>4.8</td>
<td>958.87</td>
<td>2.48</td>
<td>391142.3</td>
</tr>
<tr>
<td>900</td>
<td>2653</td>
<td>1113.80</td>
<td>2.38</td>
<td>6656.3</td>
<td>4.7</td>
<td>1120.17</td>
<td>2.37</td>
<td>495050.7</td>
</tr>
<tr>
<td>1000</td>
<td>2966</td>
<td>850.15</td>
<td>3.49</td>
<td>8074.3</td>
<td>4.7</td>
<td>1087.08</td>
<td>2.73</td>
<td>613118.2</td>
</tr>
</tbody>
</table>

Table 1: Comparison of Cluster-M and MH on a random system with 100 nodes.

...
to consider gang service are responsiveness [FA97], efficient sharing of resources [Jet97] and ease of programming. In gang service the tasks of a job are supplied with an environment that is very similar to a dedicated machine [Jet97]. It is useful to any model of computation and any programming style. The use of time slicing allows performance to degrade gradually as load increases. Applications with fine-grain interactions benefit of large performance improvements over uncoordinated scheduling [FR92].

One main problem related with gang scheduling is the necessity of multi-context switch across the nodes of the processor, which causes difficulty in scaling [ea98]. In our work, we use a class of scheduling policies, dubbed concurrent gang, that is a generalization of gang-scheduling and allows for the flexible simultaneous scheduling of multiple parallel jobs in a scalable manner.

Concurrent Gang is an strategy that increases utilization and throughput in parallel machines when compared with other implementations of gang service, for the same resource sharing strategy, as simulation studies indicate [SS99b, SS99a]. The utilization in Concurrent Gang is improved because, in the event of an idle slot or a blocked thread, Concurrent Gang always tries to schedule other tasks that are either local tasks or tasks that do not require, at that moment, coordinated scheduling with other tasks of the same job. This is the case, for instance, of I/O intensive tasks and Computation intensive tasks. Increased processor utilization in turn leads to improved application performance.

<table>
<thead>
<tr>
<th></th>
<th>Mesh</th>
<th>Hypercube</th>
<th>Fully Connected</th>
<th>Network of Works</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gradient Model</td>
<td>0.66</td>
<td>0.69</td>
<td>0.73</td>
<td>0.52</td>
</tr>
<tr>
<td>Sender Initiated Diffusion</td>
<td>0.70</td>
<td>0.71</td>
<td>0.74</td>
<td>0.59</td>
</tr>
<tr>
<td>Central Job Dispatcher</td>
<td>0.59</td>
<td>0.62</td>
<td>0.63</td>
<td>0.37</td>
</tr>
<tr>
<td>Rate of Change Model</td>
<td>0.79</td>
<td>0.79</td>
<td>0.87</td>
<td>0.61</td>
</tr>
</tbody>
</table>

Table 2: Stable situation.

<table>
<thead>
<tr>
<th></th>
<th>Mesh</th>
<th>Hypercube</th>
<th>Fully Connected</th>
<th>Network of Works</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gradient Model</td>
<td>0.72</td>
<td>0.74</td>
<td>0.76</td>
<td>0.54</td>
</tr>
<tr>
<td>Sender Initiated Diffusion</td>
<td>0.65</td>
<td>0.65</td>
<td>0.68</td>
<td>0.52</td>
</tr>
<tr>
<td>Central Job Dispatcher</td>
<td>0.65</td>
<td>0.66</td>
<td>0.65</td>
<td>0.45</td>
</tr>
<tr>
<td>Rate of Change Model</td>
<td>0.74</td>
<td>0.75</td>
<td>0.81</td>
<td>0.60</td>
</tr>
</tbody>
</table>

Table 3: Unstable situation

3 IPG-SaLaD Components

IPG-SaLaD is composed of three interdependent components: Static Allocator, Load Balancer, and Dynamic Scheduler. Their interaction can be viewed as a three stage process with a feedback loop, as shown in figure 1.

During the first stage, arriving jobs are passed to the static allocator which is responsible for performing a suboptimal map (SOM) of individual tasks that compose a job to the available compute nodes. The task assignment from the static allocator is fed to the load balancer whose aim is to minimize idle time among the processing nodes that constitute the system. Note, from the presentation of the SOM that the Rate of Change model, which is used as an input to the load balancer, is a time-variant parameter that may change during runtime and affect the scheduling of the multiple jobs present in the system. This may require redistribution of the load. The load balancer also receives input from the scheduler in the form of the current tasks load for each resource. Given both the input from the dynamic scheduler and the static allocator, the load balancer decides based on the Rate of Change model whether or not to further redistribute the load. This interaction between load balancer and the dynamic scheduler continues throughout the execution of the system.
3.1 SaLaD Static Allocation

There are a number of reasons and benefits in clustering task and system graphs in the Cluster-M fashion. Basically Cluster-M clustering causes both task and system graphs to be partitioned so that the complexity of the mapping problem is simplified and efficient mapping results can be obtained. In clustering an undirected graph, completely connected nodes are grouped together forming a set of clusters [CE95, ES94, Esh96]. Clusters are then grouped together again if they are completely connected. This is continued until no more clustering is possible (see figure 2). When an undirected graph is a task graph, then doing this clustering essentially identifies and groups communication-intensive sets of task nodes into a number of clusters called Spec clusters (see figure 3). Similarly for a system graph, doing the clustering identifies well-connected sets of processors into a number of clusters called Rep clusters. In the mapping process, each of the communication intensive sets of task nodes (Spec clusters) is to be mapped onto a communication-efficient subsystem (Rep cluster) of suitable size (see figure 4). Clustering directed graphs (i.e., directed task graphs) produces two types of graph partitioning: horizontal and vertical. Horizontal partitioning is obtained because, as part of clustering, we divide a directed graph into a layered graph such that each layer consists of a number of computation nodes that can be executed in parallel and a number of communication nodes that can be scheduled as needed. Vertical graph partitioning is obtained because part of the clustering the nodes from consecutive layers are merged or embedded. All the nodes in a layer are merged to form a cluster if they have a common parent node in the layer above or a common child node in the layer below. Doing this traces the flow of data. This information will be used later as part of the mapping so that the tasks are placed on the processors in a way that total communication overhead is minimized. For example, to avoid unnecessary communication overhead, the task nodes along a path may be embedded into one another, so that they are assigned to the same processor.

Both horizontal and vertical graph partitioning are accomplished by performing the clustering in a bottom-up fashion. The Cluster-M mapping process can then be performed in a top-down fashion by mapping the Spec clusters one layer at a time onto the Rep clusters.

3.2 SaLaD Load Balancing

Vertical graph partitioning is obtained because part of the clustering the nodes from consecutive layers are merged or embedded. All the nodes in a layer are merged to form a cluster if they have a common parent node in the layer above or a common child node in the layer below. Doing this traces the flow of data. This information will be used later as part of the mapping so that the tasks are placed on the processors in a way that total communication overhead is minimized. For example, to avoid unnecessary communication overhead, the task nodes along a path may be embedded into one another, so that they are assigned to the same processor.

Both horizontal and vertical graph partitioning are accomplished by performing the clustering in a bottom-up fashion. The Cluster-M mapping process can then be performed in a top-down fashion by mapping the Spec clusters one layer at a time onto the Rep clusters.

Vertical graph partitioning is obtained because part of the clustering the nodes from consecutive layers are merged or embedded. All the nodes in a layer are merged to form a cluster if they have a common parent node in the layer above or a common child node in the layer below. Doing this traces the flow of data. This information will be used later as part of the mapping so that the tasks are placed on the processors in a way that total communication overhead is minimized. For example, to avoid unnecessary communication overhead, the task nodes along a path may be embedded into one another, so that they are assigned to the same processor.

Both horizontal and vertical graph partitioning are accomplished by performing the clustering in a bottom-up fashion. The Cluster-M mapping process can then be performed in a top-down fashion by mapping the Spec clusters one layer at a time onto the Rep clusters.
<table>
<thead>
<tr>
<th>Simulation time</th>
<th>Gang</th>
<th>Concurrent Gang</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seconds</td>
<td>Jobs Completed</td>
<td>Total Idle Time (%)</td>
</tr>
<tr>
<td>5000</td>
<td>5</td>
<td>24</td>
</tr>
<tr>
<td>10000</td>
<td>11</td>
<td>17</td>
</tr>
<tr>
<td>20000</td>
<td>23</td>
<td>12</td>
</tr>
<tr>
<td>30000</td>
<td>35</td>
<td>11</td>
</tr>
<tr>
<td>40000</td>
<td>44</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 6: Experimental results - Computation intensive workload

<table>
<thead>
<tr>
<th>Simulation time</th>
<th>Gang</th>
<th>Concurrent Gang</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seconds</td>
<td>Jobs Completed</td>
<td>Total Idle Time (%)</td>
</tr>
<tr>
<td>5000</td>
<td>4</td>
<td>46</td>
</tr>
<tr>
<td>10000</td>
<td>14</td>
<td>43</td>
</tr>
<tr>
<td>20000</td>
<td>24</td>
<td>40</td>
</tr>
<tr>
<td>30000</td>
<td>41</td>
<td>39</td>
</tr>
<tr>
<td>40000</td>
<td>47</td>
<td>39</td>
</tr>
</tbody>
</table>

Table 7: Experimental results - Synchronization intensive workload

is as follows: if any given PE is busy executing tasks then the load differential with respect to any other PE is irrelevant since no performance gain can be obtained by load transfer. Moreover the decision factor to initiate a load transfer request should not be a PE’s absolute load but instead how much its load has changed since the previous time interval. Our goal is then to minimize processor idling time without incurring high load balancing overhead. To do so, an optimal tradeoff between the processing (cost of evaluating load information to determine task migration) and communication (cost of acquiring load information and informing other PEs of load migration decisions) overhead and the degree of knowledge used in the balancing process must be sought.

The load balancing problem can be thought of as a four phase decision process, namely:

- When to initiate task migration
- Where, which PE, to send the task migration request
- How many tasks to migrate
- Which tasks to migrate

In RoC-LB these four phases occur asynchronously at each PE and are purely distributed. This point. First, each PE may calculate this load differentially independently from other PEs in the system, there is no concept of global synchronous clock system. Second, the length of the sample interval may vary with time for a given PE, depending, for instance, on the number of load requests received, or network traffic. As a consequence different PEs may use different sampling intervals at any given time. The sampling interval is therefore an adaptive parameter. The sampling interval is usually expressed as a multiple of the time slice duration. The shorter the sampling the faster we detect the need for load balancing, but the greater overhead we incur.

Each PE uses its own $DL$ as a predictor for how many tasks will finish in subsequent intervals. If a PE assumes that $DL$ will remain the same for the next interval, it will take to reach an idle state (no task is running). If the number of intervals (time since last PE startup) is less than the network delay ($ND$), the PE will initiate a migration request.

From the above, one can conclude that only if $DL$ is negative (reduction in load) will a PE consider sending a migration request to other PEs. Network delay is an adaptive parameter. It is defined as the time it takes between the initiation of a load request and the reception of load. It may vary with time and PE. Each PE may use a different value depending on its load state.
Clustering Nonuniform Undirected Graphs (CNUG) Algorithm

For all nodes $p_i$ do
begin
Make a cluster for $p_i$ at clustering layer 1
Set the parameters of the cluster to be $(1, B_i, 0, 0)$
end
Set cluster layer $L$ to be 1
While there is at least one edge linking two clusters of layer $L$ do
begin
Sort all edges linking any two clusters in descending order
While sorted edge list is not empty, do
begin
Take the first edge $(c_i, c_j)$ from sorted edge list
Delete the edge from the list
Merge $c_i$ and $c_j$ into cluster $c'$ at layer $(L + 1)$
Calculate the parameters of $c'$
Delete clusters $c_i$ and $c_j$ from current layer $L$
For each edge $(c_x, c_y)$ in sorted edge list
begin
If $(c_x$ is a sub-cluster of $c')$ and
$(c_y$ is not a sub-cluster of any cluster) and
$(c_y$ is connected to all other sub-clusters of $c')$ then
begin
Merge $c_y$ into $c'$
Recalculate the parameters of $c'$
end
Delete $(c_x, c_y)$ from edge list
end
Else if $c_x$ and $c_y$ are sub-clusters of two different clusters at layer $(L + 1)$, then
begin
Add the weight of $(c_x, c_y)$ to the edge between the two super-clusters
Delete $(c_x, c_y)$ from edge list
end
end
end
Increment clustering layer $L$ by 1
end

Figure 2: Clustering Nonuniform Undirected Graphs (CNUG) algorithm.

In order to understand why it chooses to do so, let us introduce the three thresholds used by the algorithm. High threshold $(HT)$ and Low threshold $(LT)$ are used to determine the load status of the processor. If a PE’s load is greater or equal to $HT$ it is considered a Source PE. If on the other hand it is less or equal to $LT$ it is considered a Sink PE. If its load lies between these two thresholds then the PE is in a neutral state. If however a PE’s load falls below a Critical threshold $(CT)$ the PE immediately initiates a request for load regardless of the predicted future load based upon the current DL value. We decide to request load even though we do not expect to reach the idle state since even a small change in load at this level will result in immediate task starvation by the PE. The only exception to this rule is that if a PE has already a request pending in the network it will not issue another until either load is received from other PE(s) or the request comes back initiate a load request because at this load value of DL necessary to force the PE to become must be quite high. There is a good chance that such a value is rare and short lived in which case during the next sample interval the newly calculated DL will be such that it does not warrant an initiation of a transfer request. By delaying any action until the load value falls below $HT$ we are immune to spikes in load that may occur over time.

3.2.2 The Where Phase

Each PE keeps two local tables containing system load information. One contains information regarding the location of sink PEs, called Sink table, the other of source PEs, called Source table. Any that initiates a request for load is considered to a sink by the receiving PE(s). This is true regardless of the level of load at the time the request
Clustering Nonuniform Directed Graphs (CNDG) Algorithm

Divide the directed graph into horizontal layers in top-down fashion
For each layer of the task graph do
begin
    If \( L = 1 \) then
        Make each node at layer 1 into a cluster and calculate its parameters
    else
        For all edges \((t_i, t_j)\) where at least either \( t_i \) or \( t_j \) is not embedded or merged yet, do
        begin
            If \( t_i \) is a fork-node then
                begin
                    Embed the child node with the largest edge weight to \( t_i \)
                    If the child nodes of \( t_i \) are not in some common cluster then
                        begin
                            Merge them with \( t_i \) into a cluster
                        end
                        Calculate the parameters of the new cluster
                    end
            end
            If \( t_j \) is a join-node then
                begin
                    Embed the parent node with the largest edge weight to \( t_i \)
                    If the parent nodes of \( t_j \) are not in some common cluster then
                        begin
                            Merge them with \( t_j \) into a cluster
                        end
                        Calculate the parameters of the new cluster
                end
        end
end

Figure 3: Clustering Nonuniform Directed Graphs (CNDG) algorithm.

given job, the entries in the source/sink tables are updated with this new information. The selection process described above is used by all PEs alike, whether they are the request initiator or the recipient of a load transfer message who might need to select a source PE to whom forward the message to.

3.2.3 The How Many Phase

At the beginning of each sample interval each PE calculates its \( DL \). Using \( DL \) it computes how many tasks it would have, assuming \( DL \) would stay constant, after a length of time equal to \( ND \). Let us call this quantity Predicted Load (\( PL \)). If \( PL \) is greater or equal to zero then the PE does not expect to become idle within the next \( ND \) period and therefore does not initiate a request for load. If on the other hand \( PL \) is lesser than zero the PE requests load, according to the mechanism described in 3.2.2, in the amount of \( \text{abs}(PL) \). On the receiver side, a PE will only transfer tasks if its load is above the \( HT \) level, in which case it transfers tasks above this value up to the requested transfer amount.

cost [HBD90] Age in this case refers to the time a task has used thus far and not how long the task was created measured in wall time clock.

Figure 5 shows the internal data structures involved during the load transfer update process.

3.3 SaLaD Dynamic Scheduling

In this section we describe the Concurrent Greedy scheduler algorithm. We first introduce the concepts of cycle, slice, period and slot, which are fundamental to understand the internal workings of our algorithm. Then we describe the task classification that is used by the algorithm; we shall see that this task classification is used by Concurrent Gang to decide which task to schedule if the current task block is

3.3.3 the algorithm itself is detailed, with the description of the components of a Concurrent Greedy Scheduler.

3.3.1 Time Utilization

In parallel job scheduling, as the number of processors increases, the time to complete a job often decreases. This is because parallel processing allows tasks to be executed simultaneously, which reduces the overall execution time.

### Example

Consider a simple parallel job scheduling problem with 4 processors and 8 tasks. Each task requires 2 units of time on a single processor to complete. In this scenario, it would take 8 units of time for all tasks to complete sequentially on a single processor. However, if these tasks were split across 4 processors, each processor would execute 2 tasks, completing all tasks in 4 units of time.

### Algorithm

The algorithm for scheduling parallel jobs can be described using a greedy approach. The algorithm selects tasks that can be executed in parallel, minimizing the overall execution time. This approach is often implemented using scheduling heuristics or algorithms that take into account factors such as task dependencies, processor availability, and task execution times.

### Challenges

One of the main challenges in parallel job scheduling is managing task dependencies. If tasks must be executed in a specific order, parallel execution may not be feasible. Additionally, assigning tasks to processors in a way that balances load across processors is crucial to achieving optimal performance.

### Conclusion

Parallel job scheduling is a critical aspect of high-performance computing. By effectively managing the allocation of resources and minimizing execution time, systems can deliver faster results, enabling more efficient and faster processing of complex tasks.

---

This description is based on a fictional document that simulates an information technology context, focusing on scheduling algorithms and their optimizations.
Mapping Algorithm
For each layer of Spec graph (starting from maximal layer number) do begin
Sort all Spec clusters at current layer in descending order of $\sigma S_i^u$, $\delta S_i^u$, $\Pi S_i^u$, and $\pi S_i^u$.
Sort all Rep clusters at current layer in descending order of $\sigma R_j^v$, $\delta R_j^v$, $\Pi R_j^v$, and $\pi R_j^v$.
Calculate $f(u,v)$, if $f(u,v) > 1$, let $f(u,v) = 1$.
Calculate the required size of the Rep cluster matching $S_i^u$ to be $f(u,v) \times \sigma S_i^u$.
For each Spec cluster at current layer sorted list, do begin
If the cluster has only one sub-cluster, then
  Go to a lower layer where there are multiple or no sub-clusters
  If at least a Rep cluster of required size is found, then
    begin Select the Rep cluster of required size with minimum estimated execution time
      Match the Spec cluster to the Rep cluster
      Delete the Spec and Rep cluster from Spec and Rep list
    end
  end
end
For each unmatched Spec cluster, do begin
If the size of the first Rep cluster $> \text{the required size}$, then
  begin Split the Rep cluster into two parts with one part of the required size
    Match the Spec cluster to this part
    Insert the other part to proper position of the sorted Rep cluster list
  end
else begin
  Merge Rep clusters with largest size until the sum of sizes $\geq \text{the required size}$
  If $=\$, then
    Match the Spec cluster to the merged Rep cluster
  else
    begin Split the merged Rep cluster into two parts with one of required size
      match the Spec cluster to this part
      Insert the other part to the sorted Rep list
    end
  end
end
For each matching pair of Spec cluster and Rep cluster, do begin
If the Rep cluster contains only one processor, then
  Map all the modules in the Spec cluster to the processor
else if Inequality is satisfied, then begin
  Select the sub-cluster of the Spec cluster with the largest size
  Embed the nodes of other sub-clusters to the connected nodes of the selected sub-cluster
  to the connected nodes of the selected sub-cluster
  embed these sub-clusters onto the selected one
  Calculate the parameters for the new cluster
  Insert it into the sorted Spec cluster list
end
else begin
  Delete the Spec cluster from Spec cluster list
  Delete the Rep cluster from Rep cluster list
  Go to the sub-clusters of the Spec and Rep cluster (thus they are pushed to current layer)
  Call the same mapping algorithm for these clusters
end
end
algorithm. One such diagram is illustrated in figure 6. A Workload change occurs at the arrival of a new job, the completion of an existing one, or through the variation of the number of eligible tasks of a job to be scheduled. The time between workload changes is defined as a cycle. Between workload changes, we may define a period that depends on the workload and the spatial allocation. The period is the minimum interval of time where all jobs are scheduled at least once. A cycle/period is composed of slices; a slice corresponds to a time slice in a partition that includes all processors of the machine. Observe that the duration of the slice for Concurrent Gang is defined by the period of the global clock. A slot is the processors’ view of a slice. A Slice is composed of N slots, for a machine with N processors. If a processor has no assigned task during its slot in a slice, then we have an idle slot. The number of idle slots in a period divided by the total number of slots in that period defines the Idling Ratio. Note that workload changes are detected between periods. If, for instance, a job arrives in the middle of a period, corresponding action of allocating the job is only taken by the end of the period.

3.3.2 Task Classification

In Concurrent Gang, each PE classifies each one of its allocated tasks into classes. Examples of such classes are: I/O intensive, Synchronization intensive, and computation intensive. Each one of these classes is similar to a fuzzy set [Zad65]. A fuzzy set associated with a class A is characterized by a membership function $f_A(x)$ with associates each task $T$ to a real number in the interval $[0,1]$, with the value of $f_A(T)$ representing the “grade of membership” of $T$ in A. Thus, the nearer the value of $F_A(T)$ to unity, the higher the grade of membership of $T$ in A. For instance, consider the class of I/O intensive tasks, with its respective characteristic function $f_{I/O}(T)$ the value of $f_{I/O}(T) = 1$ indicates that the task T have I/O statements, while a value of $f_{I/O}(x) = 0$ indicates that the task T have executed no I/O statement at all.

In principle, four major classes are possible: I/O intensive, Computing intensive, Communication intensive, and (point to point) intensive and synchronization intensive. We will see in the next subsection that only subset of them are used in Concurrent Gang.

3.3.3 Description of Concurrent Gang

A Concurrent Gang scheduler is composed by a set of local schedulers, one for each PE, and a mechanism...
for the coordination of the global context switch in the
machine, which can be either a central controller or a global synchronizer.

A local scheduler in Concurrent Gang is composed
two main parts: the Gang scheduler and the local
task scheduler. The Gang Scheduler defines the
next task to be scheduled by the arrival of the global
context switch signal coming from a synchronizer or
a central controller. The local task scheduler is re
sponsible for scheduling sequential tasks and parallel
tasks that do not need global coordination, as de
scribed in the next paragraph, and it is similar to
a UNIX scheduler. The Gang Scheduler has pre
ence over the local task scheduler.

We may consider two types of parallel tasks in
a concurrent gang scheduler: Those that should
be scheduled as a gang with other tasks in other
processors and those that gang scheduling is not
mandatory. Examples of the first class are tasks
that compose a job with fine grain synchronization
interactions [FR92] and communication intensive
jobs[e998]. Second class task examples are local
tasks or tasks that compose an I/O bound parallel
job, for instance. On the other side a traditional UNIX
scheduler does a good job in scheduling I/O bound
tasks since it gives high priority to I/O blocked tasks
when the data became available from disk. As those
tasks typically run for a small amount of time and
then blocks again, giving them high priority means
running the task that will take the least amount
time before blocking, which is coherent to the
theory of uniprocessors scheduling where the best
scheduling strategy possible under total completion
time is Shortest Job First [MPT94]. In the local
task scheduler of Concurrent Gang, such high prior
ity is preserved. Another example of jobs where gang
scheduling is not mandatory are embarrassingly par
allel jobs. As the number of iterations among tasks
belonging to this class of jobs are small, the basic
requirement for scheduling an embarrassingly parallel
job is to give those jobs the larger fraction of CPU
time possible, even in an uncoordinated manner.

Differentiation among tasks that should be gang
scheduled and those that a more flexible scheduler
is better is made using the grade of membership infor
mation computed by the local scheduler (as ex
plained in the last subsection) about each task asso
ciated with the respective processor. The grade of
membership of the task previously scheduled is then
computed, but the grade might considering only the
bership of a task to each one of the major classes
described in the previous subsection. Formally, the
priority of a task T in a PE is defined as:

$$Pr(T) = \max(\alpha \times \lambda_{IO}, \lambda_{COMP})$$

Where $\lambda_{IO}, \lambda_{COMP}$ are the grade for membership
of task T to the classes I/O intensive and Com
putation intensive. The objective of the param
eter $\alpha$ is to give higher priority to I/O bound jobs
($\alpha > 1$). The choices made in equation 1 intend
to give high priority to I/O intensive jobs and com
putation intensive job, since such jobs can benefit
the most from uncoordinated scheduling. The multi
plication factor $\alpha$ for the class I/O intensive gives
higher priority to I/O bound tasks over computa
tion intensive tasks, since those jobs have a higher prob
ably to block when scheduled than computing bound
tasks. By other side, synchronization intensive and
communication intensive jobs have low priority since
they require coordinated scheduling to achieve effi
cient execution and machine utilization[FR92, e998].
A synchronization intensive or communication intensive
phase will reflect negatively over the grade of
memberhip of the class computation intensive, in
ducing the possibility of a task be scheduled by the
local task scheduler. Among a set of tasks with the
same priority, the local task scheduler uses a rou
bin strategy. The local task scheduler also defines
a minimum priority $\beta$. If no parallel task has prior
larger than $\beta$, the local task scheduler considers that
all tasks are either communication or synchroniza
tion intensive, thus requiring coordinated schedul
ing.

In practice the operation of the Concurrent Gang
scheduler at each processor will proceed as follow:
The reception of the global clock signal will generally
an interruption that will make each processing ele
ment schedule tasks as defined in the trace diag
If a task blocks, control will be passed to another
task as a function of the priority assigned to each
one of the tasks until the arrival of the next clock
signal. The task chosen is the one with higher pri
ority.

In the event of a job arrival, a job termination
or a job changing its number of eligible tasks the from
lead Concurrent Gang Scheduler will:

1. Update Eligible task list
Between Workload Changes

- If a task blocks or in the case of an idle slot, the local task scheduler is activated, and it will decide to schedule a new task based on:
  
  - Availability of the task (task ready)
  - Priority of the task defined by the local task scheduler.

All processors change context at same time due to a global clock signal coming from a central synchronizer. The local queue positions represents slots in the trace diagram. The local queue length is the same for all processors and is equal to the number of slices in a period of the schedule. It is worth noting that in the case of a workload change, only the PEs concerned by the modification in the trace diagram are notified.

It is clear that once the first job, if any, in the general queue is allocated, the remaining available resources can be allocated to other eligible tasks by using a predefined partitioning strategy.

In the case of creation of a new task by a parallel task, or parallel task completion, it is up to the local scheduler to inform the front end of the workload change. The front end will then take the appropriate actions depending on the pre-defined space sharing strategy.

4 Current Implementation of SaLaD

Software solutions for IPG are still in their infancy. As examples of some experimental meta-computing systems we have the NOW project and Legion, which enable users to use resources across multiple machines. A number of attempts are being made to address the problem of allowing programmers to write portable code that will run across networks of computers. One such tool is HENCE (Heterogeneous Network Computing Environment), an X-window based software environment designed to assist in developing parallel programs that run on a network of computers. HENCE allows a user to lay out tasks in a graphical format and execute them. HENCE uses a static table of costs that describes the cost of executing various program modules.

In this section, a Windows NT based implementation of the SaLaD tool for distributed execution of arbitrary heterogeneous tasks onto arbitrary systems using PVM is presented. The main components of this tool are the Concurrent Gang Scheduler Module, the Graphical User Interface Module, the System Graph Module, the Distribution Module, and the Graphical Interface Module for PVM based Distribution Module. The user submits the task and system graphs using the tool’s graphical user interface module, where the Node task graph represents an arbitrary executable program written in C, C++ or Fortran, and the nodes of the System Graph represent any system PVM implementation. Once the mapping is done, the Distribution Module uses PVM dispatcher to distribute the tasks on the underlying heterogeneous system and displays the results graphically. Next, the Load Balancer and Concurrent Gang Scheduler will redistribute the jobs and the tasks as necessary.

The current implementation of SaLaD instead of the application programs to be described by graphs, SaLaD graphs are variants of directed acyclic graphs or DAGs. There are two kinds of graphs namely the System and the System Graphs. Nodes of the task graph represent executables and the arcs represent data dependencies between the executables. Similarly Node of the system graph represent computer system nodes, the arcs represent the communication bandwidth available between them. SaLaD uses the Graph algorithms for clustering and mapping of the system graphs and the system graphs in an efficient manner based on the data dependency between the tasks and the communication capacity available between the systems. Once the tasks have been mapped onto the systems, individual tasks are distributed using PVM. As the load of the system changes and/or as new jobs come in the Rate of Change load balancer and concurrent Gang scheduler are invoked and they redistribute the tasks using PVM. SaLaD is composed of integrated graphical tools for creating, displaying, and analyzing parallel programs. During execution, SaLaD displays an event-ordered animation of computation execution. Through SaLaD, a user can schedule and execute applications written in a language over an existing collection of workstation and supercomputers. SaLaD depends on PVM and the task graph modules can reside on any computer.
which nodes represent executables written in C or Fortran. Arcs represent execution ordering constraints. A node may execute when all nodes that precede it in the graph have executed. The restriction placed on the task modules is that they should be using the PVM library to send back results to the SaLaD tool, if it has to be passed to other dependent modules. If the task modules have dependencies in the order of execution but do not have to pass data amongst themselves, then no change needs to be done to the executables. The above restriction is placed due to the fact those unlike tools where users write their programs graphically, SaLaD is more of a Cluster-M scheduling tool for executable modules. Unlike Task modules there is no restriction on the systems. Any system for which a PVM implementation is available can be part of the system graph.

When the SaLaD executable is run the opening screen is divided into four windows as follows: 1. Information Window - Shows the names of the currently loaded task and system graphs. 2. Drawing Window - Used to draw task and system graphs. 3. Map Window - Shows the result of the Cluster-M mapping. 4. Run Window - Show the results of executing the tasks (graph) on the systems (graph) using PVM.

The steps a user must perform to create and run a parallel program under SaLaD are as follows: 1. Draw a task and system graph in the Drawing Window that shows the desired parallel structure. 2. Once the Task and system graphs have been drawn, use the "Map" menu command to Map the task graph on the system graph. Once the mapping is done, the Rep and Spec Clusters and the Mapping results are displayed in the mapping information window. 3. After the tasks have been mapped, they can be executed on the respective systems by using the "Run" Menu command. The Run Results Window displays the status of the execution as it progresses.

5 Conclusion and Future Work

The main goal of our studies has been to concentrate on the development of tools that can be used in the execution environment of IPGs. Mainly, our focus is to complete building the SaLaD tool that Gang Dynamic-scheduler. In section 3, we presented how these three components operate in an integrated fashion, and section 4 we presented a simple implementation of SaLaD using PVM which distributes the tasks to the locations determined by the mapper.

We propose to extend the windows-NT based implementation of SaLaD using PVM with a portable programming interface. Currently a user has to specify a set of threads which can be C or C++ code and their interdependency. Then SaLaD takes this information and produces a mapping using Cluster-M mapping routines, and then the load balancer and dynamic schedulers are triggered as needed. We propose a portable programming interface such that the user just enters the program and then the program is decomposed into a task graph showing the dependencies and extracted parallelisms, and then the information gets supplied to the Cluster-M mapper. One way of pursuing this would be to use a portable parallel language from beginning such as PCN. Our initial investigation shows that clustering constructs can be implemented in PCN so that the process of clustering the graphs is automated. Another approach would be to use a Functional and/or object-oriented programming language such as SISAL or JAVA.

References


Luis Miguel Campos received his Ph.D. degree in Computer Science from the University of California in 1999, where he is currently a faculty member. Dr. Campos’ interests are the area of resource management in distributed and parallel systems and mobile agents. He is the co-author of the computer language Tea and the web application server Tea. He is a former Fulbright scholar.

Fabricio Silva received his Ph.D. degree in Computer Science from Universite Pierre et Marie Curie in 2000. He is currently a postdoctoral researcher at the University of California at Irvine. Dr. Silva is a member of IEEE and ACM and has authored numerous scientific articles.

Isaac D. Scherson is currently a Professor in the Department of Information and Computer Sciences and Electrical and Computer Engineering at the University of California, Irvine. He received BSEE and MS degrees from the National University of Mexico (UNAM) and a Ph.D. in Applied Mathematics (Computational Science) from the Weizmann Institute of Science, Rehovot, Israel. He held faculty positions in the Dept. of Electrical Engineering at Princeton University (1987-1991). Dr. Scherson has been a member of the Technical Program Committee for several professional conferences and served as Workshops Chair for Frontiers’92. He is the editor of the book that resulted from the Frontier’s 92 workshop and a co-editor of the IEEE Tutorial on Interconnection Networks. He was the Guest Editor of the Special Issue of The Computer on Foundations of Ray Tracing (June 1990). Dr. Scherson is a member of the IEEE Computer Society and of the ACM. Since July, 1992, he has contributed to the IEEE as member of the IEEE Computer Society Technical Committee on Computer Architecture. His research interests include concurrent computer systems (parallel and distributed), scalable server architectures, scheduling and load balancing, interconnection networks, performance evaluation, and algorithms and their complexity.