

# A Cost/Benefit Model for Dynamic Resource Sharing

Dimitrios Katramatos  
Dept. of Computer Science  
University of Virginia  
Charlottesville VA, 22903  
dk3x@cs.virginia.edu

Deepak Saxena, Nehal Mehta, Steve J. Chapin  
Dept. of Electrical Engineering and Computer Science  
Syracuse University  
Syracuse, NY 13244  
{deepakas,nvmehtaaq}@hotmail.com, chapin@ecs.syr.edu

## Abstract

*The use of multicomputer clusters composed of cheap workstations connected by high-speed networks is common in modern high-performance computing. However, operating system research in such environments has lagged. Our research aims at enhancing the functionality of the operating system by providing management functions that allow dynamic resource sharing and performance prediction in a clustered environment supporting distributed shared memory and multithreading. Central to this approach is the development of a parametric cost model that can predict the performance ramifications of policy choices and allow applications and middleware to adapt to the computing environment and achieve better performance.*

## 1 Introduction

The goal of our research is to develop and evaluate a parametrical cost/benefit model to be used as a decision-making tool for managing dynamic system resource sharing in an environment of high-performance heterogeneous clusters. Multicomputer clusters are capable of achieving computational rates equal or higher than those of conventional supercomputers. However, the performance these systems deliver to applications is usually just a fraction of their maximum capacity, even in cases of applications that can theoretically achieve much higher computation rates. Software inefficiency has to be blamed for this phenomenon. We are developing mechanisms that better share system resources and promote parallelization of tasks. These mechanisms allow applications to self-adapt to the varying availability of system resources according to their own varying resource requirements and thus run more efficiently.

The key idea in this research is the notion of cost, in terms of execution time, and the ramifications of certain operating system services (including process/thread creation, process/thread placement, and inter-process communica-

tion). This cost varies with several parameters such as application requirements, application behavior, time, system configuration, and network topology. Realistic prediction of the cost of system services and choices gives an application the ability to make its own decisions regarding the execution environment that best suits its needs.

We are therefore developing a parametric cost model for predicting the cost of certain operating system services while taking into account system characteristics and application information, as well as a set of software extensions to the operating system to support the function of this model and facilitate the dynamic sharing of system resources. The result will be an operating system with flexible resource control, able to deliver a higher percentage of underlying system performance to applications and middleware.

Section 2 of this paper presents the motivation for this work, section 3 the hardware and software environment under consideration, and section 4 the family of applications of interest. Section 5 describes our approach in more detail, while section 6 discusses related work. Section 7 gives a brief summary.

## 2 Motivation

In recent years there has been a new trend in high-performance computing: the use of multicomputers built from common, off-the-shelf components. These systems are capable of sustaining computation rates that rival or surpass the rates sustained by conventional supercomputers, but the demonstrated performance is just a fraction of the maximum theoretical performance. For example, the Centurion cluster of phase I achieved 3.7 sustained gigaflops on 49 CPUs during a run of an ocean modeling application [1]. This code gets just 650 megaflops on a single Cray T90 CPU. While this comparison is encouraging, Centurion of phase I had a maximum capacity of over 60 gigaflops. While it would be naive to expect to sustain a rate of computation approaching the maximum, there is certainly room to improve the performance substantially, for several cat-

egories of applications. We identify software overhead as one of the primary culprits in this reduced performance issue. To overcome this overhead we have each software layer expose to higher layers information describing the behavior of the individual layer and implications that behavior has for performance. Also, the application provides information regarding its own behavior to lower software layers as hints to let them better estimate the impact on performance.

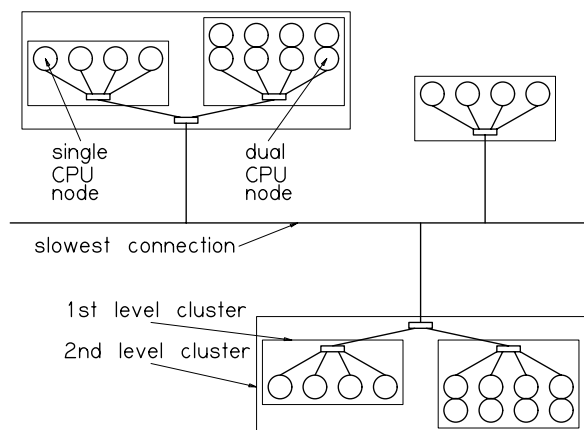
Our approach chooses to work from the bottom up, because all applications, regardless of middleware system, use the operating system. By adding a set of services to the operating system, if possible only as user-level modules for reasons of portability, we aim to allow dynamic resource sharing and also provide a resource consumer (not necessarily only an application) with information about the cost of a specified operation. For example, runtime systems will be able to get information to make service guarantees to applications, and applications will be able to self-adapt to various system configurations and varying resource availability. Load balancing environments will be able to better estimate system efficiency and better understand the effect various task mappings have on performance. In these environments dynamic granularity control will also be possible. Finally, smart compilers will be able to take into account operating system cost information during the compilation phase of an application to optimize the executable code for specific system configurations and loads. To complement this, it will also be possible to have an application loader that will take into account application supplied information and create the most suitable environment within a system's limits for running a specific application.

### 3 The Hardware and Software Environment

The environment in which the problem is considered is a distributed system consisting of a variety of nodes connected with networks of various speeds. We view such a system as having a physical and a logical organization.

The physical organization of the system is based on the principle of hierarchical clustering [13]. Groups of neighboring nodes form local clusters. Local clusters can be grouped together and form second level superclusters, second level supercluster groups can form third level superclusters, etc. (see figure 1). The main criterion for forming the various hierarchy echelons is the cost of communications—these echelons reflect the underlying networks. That is, echelon 0 corresponds to communications between processors of the same node (intra-node)—if nodes have more than one processor—which have hardware shared memory and thus the least cost for sharing information. Echelon 1 corresponds to intra-cluster communication, echelon 2 to cross-cluster communication between clusters of the same 2nd level supercluster, echelon 3 to cross-cluster communi-

cation between clusters of different 2nd level superclusters, etc. Another way to think of this organization is as a system map.

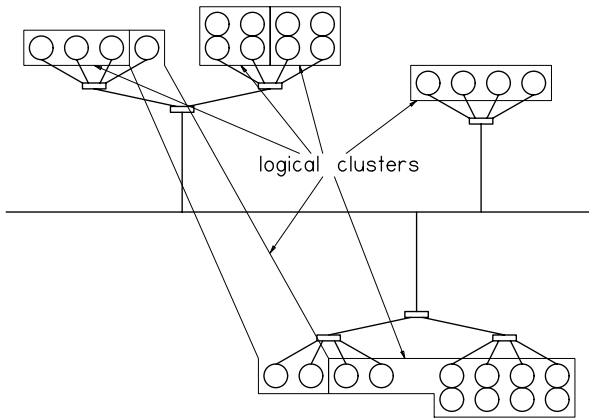


**Figure 1. A clustered multicomputer - physical clustering**

Over the physical clusters is laid an organization of logical clusters. The nodes of a logical cluster share information using distributed shared memory. Note the difference between physical and logical clusters: physical clusters are defined by the physical organization of a system, mainly by the nature of the interconnecting network, and are fixed; logical clusters are groups of processors that share memory using software-based distributed shared memory (DSM) and can change dynamically. Such a cluster can be a portion of a physical cluster, a whole physical cluster, or it can span several physical clusters incorporating parts or the whole of them (see figure 2)).

An important aspect of the environment is the support for threads and for distributed shared memory. The elementary computing entity here is the thread and an application includes a dynamically changing number of threads. The enhanced operating system provides a single address space for the threads of an application. This address space is visible by a number of processors within the logical cluster structure and the threads are mapped on these processors without the knowledge of the application. Thread migration is possible in this environment, as well as migration of thread groups, even whole applications, within (intra-cluster) or between logical clusters.

The choice of software DSM as the means of forming a logical cluster is a trade-off between raw performance and ease of programming. The pros and cons of the message passing and the distributed shared memory paradigm are known and there have been numerous publications in this



**Figure 2. Logical clustering**

area. We believe that there is a lot of room for improving the performance of certain categories of applications on multicomputer clusters. We expect that careful use of DSM in combination with cost/benefit prediction and dynamic adaptation to resource demand/availability will demonstrate definite performance improvements over the current figures without sacrificing usability and programming ease on the altar of performance. In any case, the idea of using a cost model to predict the cost/benefit of certain operating system operations is independent of using one paradigm or the other and can contribute in all cases to the better administration of a system's resources.

## 4 The Applications

The type of application of interest is a parallel application with high resource demands, such as a large scientific simulation. This type of application usually presents a dynamic change of resource requirements, that is, it presents irregularity: "data structures, communication patterns, or computation are not defined by simple, repeating structures" [4].

Our computational model mixes shared-memory programming (done with threads) with distributed-memory programming (done with a message passing environment such as MPI). Our initial work was to support Pthreads, the POSIX-standard threads library, but we are moving to support the emerging standard for multiprocessing, OpenMP, in conjunction with MPI.

As an example of this application type, consider a weather simulation, tracking the progress of a storm front as it moves across a landscape which has been mapped onto a grid. Each grid cell can be mapped onto a multithreaded MPI process. The grid cells containing the storm front

will require the most computation, and as the front moves, the computational hotspots will move across the grid. To achieve good performance, we want to rebalance the workload. Historically this has been done either through data migration or computational migration.

Static thread allocation for an application with dynamic resource requirements leads either to processors sitting idle, when resource usage is overestimated, or to delay from load imbalance when usage is underestimated. Clearly, such a computation can only be performed efficiently with the combination of a load balancing technique with dynamic granularity control [3]. Using our approach, one can add extra computational power within the address space of a hotspot, thus spreading the load to more threads. In the example of storm front simulation, we can add threads to the cells containing the front (these new threads might be put on local processors, remote processors via DSM in an expanded logical cluster, or we might even choose to migrate the entire process to a larger SMP and add threads there). Once the front passes, the extra threads are no longer necessary and can be killed, freeing the occupied processors. Each multithreaded MPI process can be assigned to a logical cluster with a certain number of nodes that share memory and provide the illusion of a single address space to the threads of the process, while each thread runs on a different CPU. The MPI processes communicate with cross-cluster messages. Processors can be dynamically added to/removed from each logical cluster in response to load variations.

## 5 Our Approach

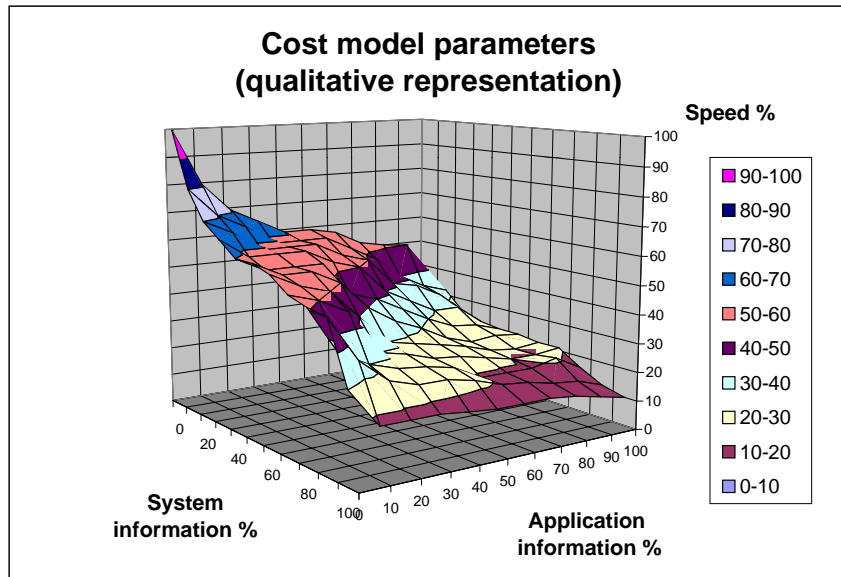
Our work is based on Linux, which is the de facto standard for free software cluster operating systems. This has the obvious advantage of allowing us to add code to the kernel as we deem necessary. We are building two sets of software to support our needed functionality: kernel extensions and user-level libraries. Our work to date has focused on determining the required functionality, and the line between what should go in the kernel and what should go in the user library is not yet set.

### 5.1 Additional Functionality

We are augmenting Linux with heterogeneous distributed shared memory and multithreading functionality. Functionality similar to the Mermaid prototype [8] is desired as well as conformance to the OpenMP standard [17] and the popular Pthreads interface.

The additional functionality that we are currently adding to the system is as follows:

1. Add a thread or a group of threads to an existing shared address space.



**Figure 3. The main parameters of the cost model**

2. Add a processor or remove a processor from a shared address space.
3. Dynamically migrate a thread from one processor to another. The target processor should automatically be added to the shared address space, if not already included, and the source processor should possibly be removed, if no other thread runs on it. Removing a processor from an address space might not always be desirable as it can cause thrashing on processor add/remove.
4. Form, modify, manage, and cancel a logical cluster of nodes. These functions will correspondingly create, modify, manage, and delete the necessary data structures that need to be maintained to support logical clusters.
5. Evaluate cost functions to estimate the cost/benefit of performing certain operations, as for example to create/cancel a cluster, add/remove a processor to/from an address space, start/kill a thread on a specific processor, and migrate a thread, which may result to the inclusion of the target processor in the shared address space and/or the removal of the source processor, as mentioned above.

## 5.2 The Cost/Benefit Model

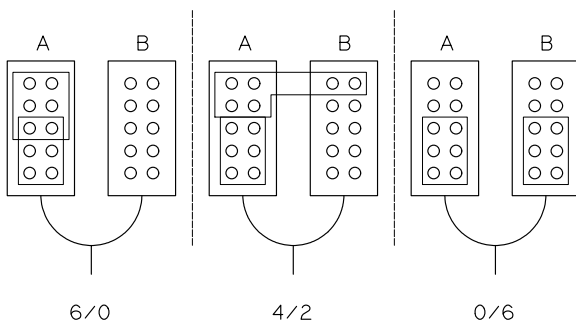
The operating system extensions are mechanisms, not policies. Policy decisions will be made either in middleware layers or by the application itself, e.g. whether or not to add a thread to a running process. To assist the higher software layers in making these decisions, we have developed a cost model so that the operating system can accurately predict the ramifications of policy decisions.

Our cost model moves qualitatively along three axes: the first axis has as parameter the amount of system information taken into account; the second axis, the amount of application information; and the third, the speed of the cost estimation. Figure 3 represents a qualitative picture of the cost model behavior. The accuracy of the model is generally inversely proportional to the speed of the estimation. The speed of the model depends on the type and quantity of application and system information that must be processed. Accuracy increases and speed decreases as the volume of information increases. The cost of a cost estimation has to be low when compared with the benefit that a correct policy decision will offer. On the other hand, the estimated cost needs to be accurate enough to enable correct decisions. Clearly, this is a complex problem and there are several factors that need to be considered. Therefore, the cost model needs to be parametric and provide for various trade-offs of

accuracy for speed.

In its simplest form, the cost model views the cost of operations as set of discrete cost classes. This classification is of the least accuracy and maximum speed and it works when a fast answer is needed and no information is available from the application side. As an example, in a system with two clusters where one cluster is formed by dual-processor x86 boxes and assuming 3 cost classes, the cost of adding a thread to the second processor of a node belongs to class #0. The cost of adding a thread to a new (unused) processor in the local cluster belongs to cost class #1. The cost to migrate a thread to the neighboring cluster is of class #2, the highest. Here the accuracy of the model is the lowest as it relies only on basic system information.

In its richest form, the model can provide a set of cost information about an operation and the impact this operation will have on the performance of the application. For this purpose, it is necessary to combine information from both the system and the application. The system maintains a vector of state and cost information by directly accessing information maintained by the operating system kernels and by periodically running a set of suitable benchmarks to measure other important cost-affecting quantities, like communication latencies or node loads (cf. the Network Weather Service [26]). The application passes in a description of its behavior, e.g. memory access patterns, thread running patterns, acceptable delays, etc. Statistical information, gathered during previous runs of the application, can help identify its behavior when this behavior is not known in advance. The cost model combines the supplied information to produce a cost rating, a quantity that can be used in comparing the cost of different system operations.



**Figure 4. Three possible configurations**

For example, consider two neighboring (physical) clusters, A and B, with 10 processors each, and an application

running on six processors of cluster A, which form a logical cluster, with six threads, one per processor (see figure 4)). Suppose that at a certain phase during execution the application wants to spawn another group of six threads. Would it be better to use six processors of cluster A (4 free ones and two already running a thread each), use the remaining 4 processors of cluster A and two more from cluster B, or use six processors of cluster B instead? Cross-cluster communication is more expensive than intra-cluster communication so the first option seems more attractive than the second and the second more attractive than the third. However, this is not necessarily the best order. The first option suffers from the fact that two processors have to run two threads each. Also, what is of highest benefit depends on how the application behaves. If the new group of threads works in close cooperation with the initial thread group, then the first option is probably better. If the group of these six new threads performs a separate task and communicates infrequently with the initial thread group, that is, data shared between the two groups are infrequently accessed, then the third option is probably better. This is because the infrequent communication between the two thread groups keeps the communication cost low. In this case, if the second option were chosen, the frequent communication between the two threads running on cluster B with the other four on cluster A would impose a heavy increase in the communication cost.

The following is a cost rating estimation for the three thread placement options considered. Since a highly detailed estimation would be too long to present here, certain simplifying assumptions are necessary. The first assumption is that the system costs are identical for all processors. A more detailed estimation would include the cost of performing certain operations on each individual node, or type of node. A second assumption is that the costs for sending messages are estimated using the basic LogP model [16]). Consider now the following costs:

- $s$  cost to start a thread on a processor,
- $e$  cost to include a processor in a shared address space,
- $e_r$  cost to include a processor that belongs to a neighboring cluster in a shared address space,
- $c_i$  cost for sending a message across the intra-cluster network,
- $c_c$  cost for sending a message across the cross-cluster network,  $c_c \geq c_i$ ,

Let the application supply information about itself in the form of two weight factors,  $w_g$ , and  $w_r$ .  $w_g$  expresses the percentage of accesses to variables shared between the new group of threads, and  $w_r$ , expresses the percentage of accesses to variable shared between the old and the new thread

groups.  $w_g + w_r = 100\%$ . The application declares that the threads of the new group communicate frequently with each other by having  $w_g \gg w_r$ , that is, intra-group communication is much more frequent than between groups. We'd like to compare the three basic thread group placement options, in the present case all six threads on cluster A (option 6/0), four threads on cluster A and two on B (option 4/2) or all six on cluster B (option 0/6).

The first thing to estimate is the overhead for spawning the new thread group: for the 6/0 option  $6(s + \frac{4}{6}e)$ , for the 4/2 option  $6(s + \frac{4}{6}e + \frac{2}{6}e_r)$ , and for the 0/6 option  $6(s + e_r)$ . Since  $e_r \geq e$ , the 0/6 option has more overhead than the 4/2 option, and that in turn more than the 6/0 option, which is expected.

This overhead is not enough to make a decision. Here the cost rating has to have at least two parts, a fixed part, the overhead, and a variable part, a rating for the communication and the running cost.

For estimating the communication cost rating, assume that  $m$  is the number of messages associated with the access of shared data by threads of the new group per time unit (message rate). Then the 6/0 option has a rating of  $m(w_g + w_r)c_i = mc_i$ . For the 4/2 option assume a split factor  $k$  to distinguish between the messages send within A and B and those that cross the boundary between them. That is,  $k\%$  of messages are intra-cluster within clusters A and B, and  $(1 - k)\%$  cross-cluster between A and B. Then the cost rating is  $m(kc_i + (1 - k)c_c)$ . Finally, for the 0/6 option the rating is  $m(w_g c_i + w_r c_c)$ . To compare the rating of 0/6 to that of 4/2 we subtract the first from the second. After the calculations we obtain  $m(c_c - c_i)(w_g - k)$ . Because  $c_c \geq c_i$  and  $w_g \gg w_r$ , this quantity is positive when  $w_g > k$ . Having  $w_g < k$  is incompatible with the assumption that the threads of the new group communicate frequently with each other and infrequently with the old group; the value of  $w_g$  is close to 100% and a value for  $k$  approaching  $w_g$  would mean that the communication between the subgroup of the two threads and that of the four threads is infrequent, in direct contrast with the above assumption. Thus, the cost rating for the 4/2 option is higher than that of the 0/6 option. In turn, the cost rating of 0/6 is higher, but only slightly, than 6/0. So far, 6/0 has the lowest overhead and cheapest communications, with 0/6 second and 4/2 last.

To make the final decision, it is necessary to estimate the cost of running for the three options. Assume a time period  $\Delta t$ . During this period the new thread group causes the sending of  $m\Delta t$  messages. Also assume that during  $\Delta t$  the maximum number of thread instructions available for execution are  $I_t$  and that a processor can execute  $x$  instructions per time unit.

For cases 4/2 and 0/6, each thread will take  $I_t/x$  time. For the 6/0 case, two of the threads are running essentially at half speed  $x/2$  since they run on already occupied proces-

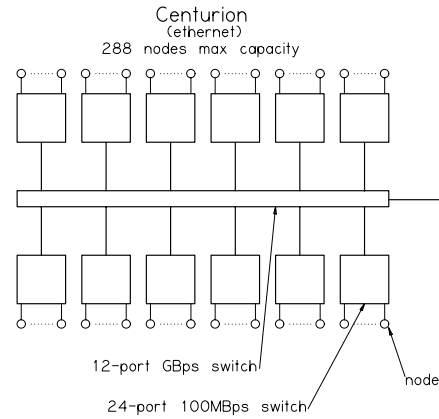


Figure 5. Centurion, ethernet connections

sors. Assuming that in general we have CPU-bound threads, these two threads will take double the time of the other four threads,  $\frac{2I_t}{x}$ , probably delaying their work (the interference pattern of the threads is another factor to take into account for more accurate estimations). It can be concluded now that option 0/6 is probably the best choice, because it has almost the same communication cost rating with 6/0 but half the running cost rating, and in the long run the overhead can be ignored.

There are other factors that can affect the cost of each option. For example, if clusters A and B contain processors of different architectures, cross-cluster communication may be much more expensive than before, due to the necessary data conversions. This cost can be counterbalanced if the processors of cluster B are faster than the ones of cluster A, thus providing for higher execution rates. Finally, one has to take into account the effect of the already existing load of the processors and the network as it changes with time, as there can be other applications or tasks sharing the system resources.

### 5.3 The Experimental Computing Environment

The hardware we are using for experimentation is a metacluster comprising the Centurion machine [2] and the Syracuse Orange Grove cluster. Centurion has 256 nodes, with 128 533MHz DEC Alpha nodes and 128 400MHz dual Pentium II nodes. Each node is connected to a 100Mbps fast ethernet switch and multiple such switches are joined via gigabit ethernet switches (see figure 5). The initial 64 Alpha nodes are also joined in a complex mesh by a 1.2Gbps Myrinet fabric (see figure 6). The Orange Grove cluster has 16 533MHz DEC Alpha nodes and 48 450MHz dual Pentium III nodes, and has a system area network similar to Centurion's. Figure 7 presents a simplified overall view of

the total system. The only real restriction is that each CPU needs to run the modified Linux system.

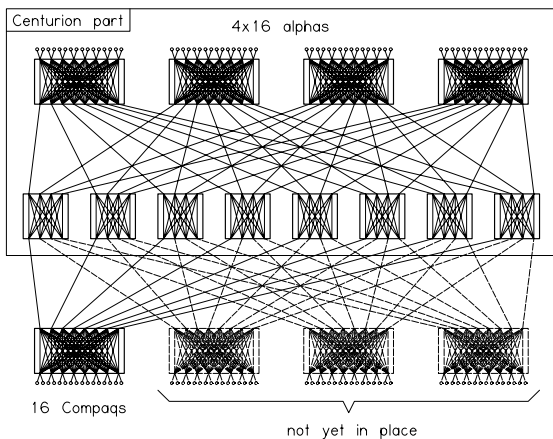


Figure 6. Myrinet connected part of Centurion

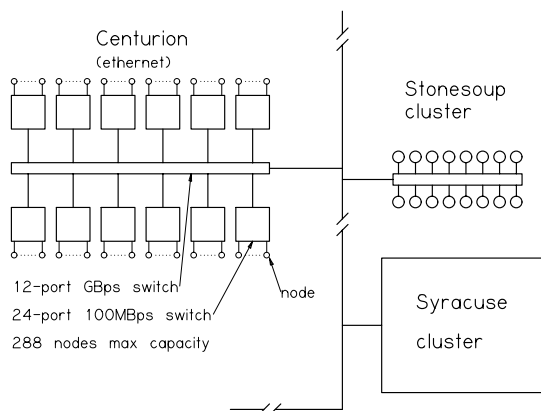


Figure 7. The experimental system

## 6 Related Work

Much of research work has been and is being dedicated to issues related to clusters of commodity workstations. Our approach is novel, to the best of our knowledge, in the way it supports clustering, in providing the mechanisms for dynamic resource sharing in this framework, and most of all for using cost models to estimate the cost/benefit of certain o.s. operations and aid in decision making. The current section presents work related to our research with respect

to operating systems, distributed shared memory, and other relevant areas.

### 6.1 Related Work in Operating Systems

The first work of interest is a research project of the Computer Systems Research Institute of the University of Toronto, Canada, targeted to hierarchically structured NUMA clusters [14]. Part of this project was the design and construction of the Hector hierarchically structured shared memory multiprocessor. Hurricane is the operating system that was specially developed for Hector. The structure of this operating system reflects the structure of the multiprocessor. Hurricane uses tight coupling within a local cluster and loose coupling within clusters. Operating system services are designed to take advantage of high speed connections and locality of data. Cluster size is determined statically. Processes of an application are scheduled on the same cluster, unless there is a benefit for a job to span multiple clusters. Within a cluster load is balanced at a fine granularity and cross-cluster scheduling performs coarse-grain load balancing by assigning and migrating processes to specific clusters. Tests of the system have shown that applications need to be adapted to Hector/Hurricane to perform well.

The computing environment targeted in our research is substantially different from the Hector/Hurricane environment. The Hector/Hurricane approach was an attempt to create a cost-effective, scalable NUMA machine, utilizing specially designed hardware. Experimenting with hardware is not in our intentions. Although the targeted environment maintains the notion of tightly-coupled sub-clusters within a larger hierarchical system, there is no hardware support for shared memory, except for the case of a multiprocessor PC box. Also, architecture homogeneity is not necessary, the cluster hierarchy is not of fixed depth, and the (logical) cluster size is not determined statically. However, there are several lessons from the Hector/Hurricane approach that can be useful to us, e.g. that the factors found to be crucial for application performance depend on application behavior and system behavior. This is exactly what our work investigates. The fact that applications needed to be specifically adapted to run efficiently on Hector under Hurricane underlines the importance of having operating systems that can provide the necessary information and allow applications to self-adapt to the dynamically changing resource availability of a system.

A more recent work is the Berkeley Network of Workstations (NOW) project [18]. This project is targeted at workstation clusters and includes operating-system level work software such as the GLUnix layer [19]. GLUnix is a multi-user, user-level system for a cluster of workstations. It is designed to provide transparent remote execution by maintaining a single-system image across an entire cluster and

supports interactive parallel and sequential jobs and load balancing. The proposed research differs in that it is not dedicated to single-cluster but to hierarchically clustered systems with several cluster levels. Although it is possible to provide a single-system image across clusters and levels of hierarchy, the approach taken here is to provide the users with the necessary data to make informed choices about state sharing issues. In any case, increasing the degree of sharing is usually accompanied by a decrease in performance, and users may be willing to sacrifice transparency for performance. GLUnix supports load balancing through intelligent job placement but doesn't support task migration nor any form of dynamic resource sharing based on cost/benefit prediction.

## 6.2 Related Work in Distributed Shared Memory

There is a vast variety of research works on distributed shared memory (DSM). A possible classification of DSM systems can be based on:

- the level of implementation (user, kernel, hybrid),
- the coherence protocols (SRSW, MRSW, MRMW, etc.),
- the consistency model (sequential, processor, release, etc.).

Three general techniques have been used [20]. The first technique simulates a multiprocessor by using a modified pagefault trap to do paging over the network, e.g. IVY [23]. The second technique focuses on shared variables rather than pages. The programmer is required to annotate the shared variables with their anticipated access patterns. These annotations are then used by the DSM runtime system for selecting the most suitable coherence protocol. This technique is used in Munin [5]. The third technique is object-oriented and requires a high-level programming model, e.g. Linda [21], Orca [22].

Relatively few approaches have dealt with the issue of heterogeneity. Mermaid [8] is an implementation of heterogeneous DSM as an extension to the IVY system. According to Mermaid's creators, heterogeneous DSM is feasible and presents comparable performance to homogeneous DSM. However, many issues need to be addressed due to heterogeneity.

Providing yet another DSM system is not in the scope of our research. Our main interest is in developing a DSM system that can provide cluster-oriented functionality and cost data on typical DSM operations. The preferred way to follow here is to adopt an existing system and subsequently modify and augment it to serve the intended purposes. Several decisions need to be made, but the main direction is

to adopt the simplest approach possible, decide the level of implementation, and add the minimum functionality necessary. In this sense, the IVY and Mermaid systems seem to be attractive. The issue of fully supporting heterogeneity remains open, because forming clusters with machines of heterogeneous architectures or sharing memory among clusters containing homogeneous hardware but of different architecture from cluster to cluster are options of high complexity and questionable performance. The Munin approach is also attractive because of the focus on the different access patterns of shared variables. Because shared variable access patterns constitute an important parameter of the cost model, it would be interesting to see the impact the utilization of such a DSM approach has on performance. However, the use of a Munin-like system requires additional programmer effort, the annotations of shared variables, which is not particularly desirable here if it means many modifications to already existing programs.

Several other works on DSM need to be mentioned here, because they contain useful elements for our work. Iftode et al. [6] study the sharing patterns of applications running on DSM systems and identify several factors affecting performance. Lu et al. [7] add limited compiler support and modifications to TreadMarks [24] to eliminate unnecessary computation and communication during the execution of irregular applications. Yoon and Malek [9] propose the creation of a single address space per running program instead of a global address space shared by all processing nodes. This is similar to the definition of logical clusters with the difference that not only one program can run on a logical cluster, no matter what effect this fact has on performance. Kim and Vaidya [10] propose an adaptive DSM system where statistics about memory accesses, collected over a sampling period, are used for determining the protocol that has the minimum cost for each memory page. Erlichson et al. [11] present a kernel-level implementation of DSM on an 8-node, 8-processors/node cluster and study the cost of DSM primitives and the effects of clustering on performance. Finally, Quarks [12] is a relatively new simple DSM approach. Quarks is a descendant of Munin and is aimed at reducing the communication overhead. Its basic abstraction is that of shared regions, which are page-aligned byte ranges of variable length. For this DSM system there exists a freely available Linux port.

## 6.3 Other Relevant Works

In terms of functionality, the most closely related work to our approach is the Scalable Concurrent Programming Library (SCPLib) [4] of the Scalable Concurrent Programming Laboratory at Syracuse University. SCPLib is the descendant of the concurrent graph library [25] and is aimed at supporting irregular applications on parallel hardware. The

library provides heterogeneous communication and file I/O, load balancing, and dynamic task granularity control. The user needs to supply a set of special support routines for the library to successfully perform migration of tasks and granularity adjustments. The load balancing mechanism is based on the concept of heat diffusion [3]. Communication cost is taken into account when deciding task movements.

Our work is targeted at a more general and complex environment than SCPLib. The goal is to provide functionality for a hierarchy of clusters and help applications and middleware make decisions by estimating as accurately as possible, or to the desired degree of accuracy, the cost of certain operating system services especially when dynamically sharing resources. In view of this, the proposed research can address all issues covered by SCPLib. Although load balancing is not in the scope of our research, the use of the cost model can greatly facilitate a sophisticated load balancing mechanism by providing the cost of various task placement options when deciding how to redistribute the load and not only the cost of communications. Also, SCPLib is based on message-passing whereas we support DSM and mixed mode computations.

Another research work, which has common ground with the proposed research, is that by Kravets et al. [15]. This work proposes a cooperative solution to the dynamic management of communication resources. In this solution, application requirements, expressed in the form of "payoff" functions, and network resource availability, expressed in the form of service availability curves, are taken into account by a configurable communication layer. The goal is to better exploit communication resources by allowing applications and networks to adapt to each other. Our approach has as ultimate goal to allow applications to self-adapt to the changing availability of all system resources and doesn't focus specifically on the communication layer.

## 7 Summary

Modern clusters connected by high-speed networks are capable of outperforming supercomputers, but the performance delivered to scientific applications is only a fraction of this maximum. Identifying software overhead as a key reason for this discrepancy, we have described our approach to solving this problem. We focus on an hardware environment with hierarchically organized clusters of computing nodes. In this environment we form logical groups of nodes by using software DSM and multithreading, and augment the capabilities of the operating system with dynamic resource sharing primitives and a decision-making framework based on a parametrical cost/benefit model. The model works for a variety of situations with data availability ranging from minimal to high. Our goal is to predict the performance ramifications of policy choices and thus to allow

applications and middleware to adapt to their computing environment and achieve better overall performance.

## References

- [1] G. Lindahl, S. Chapin, N. Beekwilder, A. Grimshaw. Experiences with Legion on the Centurion Cluster. Technical Report CS-98-27, Department of CS, University of Virginia, 1998
- [2] Centurion: The Legion project testbed. <http://legion.Virginia.EDU/centurion>
- [3] J. Watts, M. Rieffel, S. Taylor. Dynamic Management of Heterogeneous Resources. In *High Performance Computing: Grand Challenges in Computer Simulation*, pp.151-6, April 1998.
- [4] SCP Laboratory. Scalable Concurrent Programming Library. [http://www.scp.syr.edu/html/scp\\_library.html](http://www.scp.syr.edu/html/scp_library.html)
- [5] J. Bennett, J. Carter, W. Zwaenepoel. Munin: Distributed Shared Memory based on type-specific memory coherence. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp.168-176, March 1990.
- [6] L. Iftode, J. PalSingh, K. Li. Understanding application performance on shared virtual memory systems. In *Proceedings of the 23rd annual international symposium on Computer architecture*, pp.122-133, 1996.
- [7] H. Lu, A. Cox, S. Dwarkadas, R. Rajamony, W. Zwaenepoel. Compiler and software distributed shared memory support for irregular applications. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp.48-56, 1997.
- [8] S. Zhou, M. Stumm, K. Li, D. Wortman. Heterogeneous Distributed Shared Memory. In *IEEE Transactions on Parallel and Distributed Systems*, Vol.3, No.5, pp.540-554, September 1992.
- [9] M. Yoon, M. Malek. Configurable Shared Virtual Memory for Parallel Computing. Technical Report CS-TR-94-21, University of Texas, Austin, July 1994.
- [10] J. Kim, N. Vaidya. A cost-comparison approach for adaptive distributed shared memory. In *Proceedings of the 1996 international conference on Supercomputing*, page 44, 1996.
- [11] A. Erlichson, N. Nuckolls, G. Chesson, J. Hennessy. SoftFLASH: analyzing the performance of clustered

- distributed virtual shared memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.210-220, 1996.
- [12] M. Swanson, L. Stoller, J. Carter. Making Distributed Shared Memory Simple, Yet Efficient. To appear in the *Proceedings of the Third International Workshop on High-Level Parallel Programming Models and Supportive Environments*, March 1998.
- [13] R. Unrau, M. Stumm, O. Krieger. Hierarchical Clustering: A Structure for Scalable Multiprocessor Operating System Design. Technical Report CSRI-268, Computer Systems Research Institute, University of Toronto, March 1992.
- [14] R. Unrau, O. Krieger, B. Gamsa, M. Stumm. Hierarchical Clustering: A Structure for Scalable Multiprocessor Operating System Design. *Journal of Supercomputing*, 1995.
- [15] R. Kravets, K. Calvert, K. Schwan. Payoff-Based Communication Adaptation based on Network Service Availability. In *Proceedings of IEEE Multimedia Systems '98*, 1998.
- [16] D. Culler, R. Karp, D. Patterson, A. Sahay, E. Santos, K. Schauser, R. Subramonian, T. von Eicken. LogP: a Practical Model of Parallel Computation. In *Communications of the ACM*, Vol.30, No.11, pp.79-85, November 1996.
- [17] OpenMP  
OpenMP C and C++ Application Program Interface, Version 1.0, October 1998.  
<http://www.openmp.org>
- [18] The Berkeley NOW project.  
<http://now.cs.berkeley.edu>
- [19] GLUnix: A Global Layer Unix for NOW.  
<http://now.cs.berkeley.edu/Glunix/glunix.html>
- [20] A. Tanenbaum. *Distributed Operating Systems*. Prentice-Hall, Inc., 1995.
- [21] D. Gelernter. Generative Communication in Linda. In *ACM Transactions on Programming Languages and Systems*, vol.7, pp.80-112, January 1985.
- [22] H. Bal, M. Kaashoek, A. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. In *IEEE Transactions on Software Engineering*, vol.18, pp.190-205, March 1992
- [23] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proceedings of the International Conference on Parallel Processing*, pp.94-101, August 1988.
- [24] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, W. Zwanenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. In *IEEE Computer*, Vol.29, No.2, pp.18-28, February 1996.
- [25] S. Taylor, J. Watts, M. Rieffel, M. Palmer. The Concurrent Graph: Basic Technology for Irregular Problems. In *IEEE Parallel and Distributed Technology*, 4(2):15-25, 1996.
- [26] R. Wolski. Dynamically Forecasting Network Performance to Support Dynamic Scheduling Using the Network Weather Service. In *Proceedings of the Sixth International Symposium on High-Performance Distributed Computing (HPDC-6)*, August 1997.

**Steve Chapin** is an Associate Professor of Electrical Engineering and Computer Science at Syracuse University. Prior to joining Syracuse, he served on the faculties of the University of Virginia and Kent State University. He received his Ph.D. in Computer Science from Purdue University in 1993.

**Deepak Saxena** and **Nehal Mehta** are M.S. students in the department of Electrical Engineering and Computer Science at Syracuse University.

**Dimitrios Katramatos** is a Ph.D. candidate at the University of Virginia. He received his M.S. in Computer Science from Kent State University.