Metasystems: An Approach Combining Parallel Processing and Heterogeneous Distributed Computing Systems

Andrew S. Grimshaw
Jon B. Weissman
Emily A. West
Ed C. Loyot Jr.

Abstract

A metasystem is a single computing resource composed of a heterogeneous group of autonomous computers linked together by a network. The interconnection network needed to construct large metasystems will soon be in place. To fully exploit these new systems, software that is easy to use, supports large degrees of parallelism, and hides the complexity of the underlying physical architecture must be developed. In this paper we describe our metasystem vision, our approach to constructing a metasystem testbed, and early experimental results. Our approach combines features from earlier work on both parallel processing systems and heterogeneous distributed computing systems. Using the testbed we have found that data coercion costs are not a serious obstacle to high performance, but that load imbalance induced by differing processor capabilities can limit performance. We then present a mechanism to overcome load imbalance that utilizes user-provided callbacks.[1]

Send correspondence to:
Andrew Grimshaw
Department of Computer Science
Olsson Hall
University of Virginia
Charlottesville, VA 22903
804-982-2204
Fax: 804-982-2214
grimshaw@virginia.edu

1. This work partially funded by NSF grants ASC-9201822 and CDA-8922545-01, NASA grant NGT-50970, and NLM grant LM04969.

1.0 Introduction

A metasystem is a distributed computing system composed of a heterogeneous group of autonomous computers linked together by a hierarchical network. For example, consider a metasystem consisting of computers at three different sites. The first site has an Intel Paragon, a SUN workstation cluster, and an SGI workstation cluster connected by a LAN. The second site has an HP workstation cluster. The third site has two Cray supercomputers connected by a high speed network. All of the sites are connected via the Internet. Such a metasystem would have tremendous potential for high-performance computing.

Unfortunately, writing software for such a system is difficult. First, the programmer must deal with all the complexities present in any parallel programming system: managing communication and synchronization, scheduling a potentially large number of independent tasks, and decomposition and distribution of data. In a heterogeneous system these tasks become even more difficult, and there is the additional problem of data conversion. What is needed is a software system that *transparently* manages communication and synchronization, performs data decomposition, distribution and coercion, and schedules all hardware resources across the different platforms. No such software currently exists.

Our approach to solving this metasystem problem is to combine ideas from both heterogeneous distributed computing and parallel processing. We are using Mentat [14][15], an object-oriented parallel processing system, as a testbed for our metasystem research. Mentat significantly simplifies parallel programming because all communication and synchronization is managed by the compiler and run-time system. We have constructed a metasystems testbed by extending Mentat from a homogeneous system to a heterogeneous system, incorporating ideas from heterogeneous distributed computing into the programming language and run-time system.

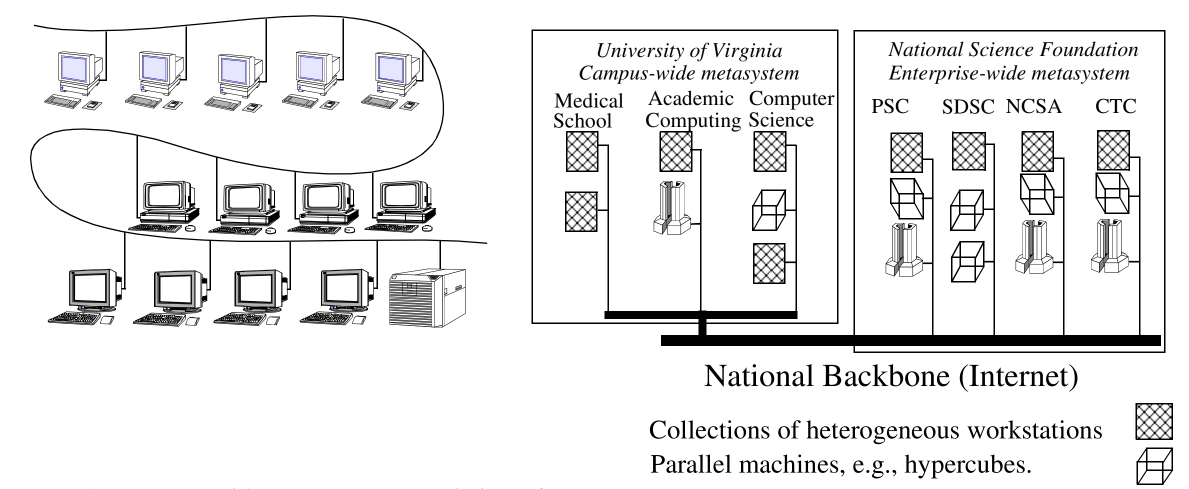
We have found that in a heterogeneous environment, scheduling, which includes both problem decomposition and task placement, has a dramatic effect on a program's performance. An inappropriate problem decomposition or task placement decision can result in poor performance due to load imbalance. Effective scheduling in a heterogeneous environment is a difficult problem. We will show that this problem can be overcome using relatively simple heuristics if appropriate callbacks are provided by the application programmer. Callbacks are user-defined functions that can be called by the run-time system while the program is executing. These callbacks permit the scheduler to determine the computation and communication complexity of data parallel computations. This information is then used to decompose the problem and schedule the work units in a manner that provides good load balance, and thus good performance. An interesting aspect of the callback mechanism is the flexibility it provides; it can be used to partition not only regular, two-dimensional array problems, but any problem for which a suitable callback can be constructed.

We begin our presentation by providing a more precise definition of a metasystem and by describing our vision of an ideal metasystem. We discuss our design goals and constraints, many of which are driven by practical considerations, not by fundamental principles. We then describe our approach to solving the metasystem problem, beginning with an introduction to Mentat, and a discussion of our heterogeneous testbed. Next, we present the performance results that motivated our investigation into automatic methods for problem partitioning and decomposition. The results show two things. First, data coercion costs do not significantly impact performance. This is important because it means that heterogeneous computing can achieve better performance than the alternative of not using heterogeneous resources. Second, if the work in a data parallel computation is distributed evenly between workers, a load imbalance will develop because the fast processors will finish their computations before the slow processors. This leads to reduced utilization and, hence, worse performance than a distribution that reflects processor speeds.

Having motivated the scheduling problem, we present a restricted system model, and a set of callbacks and scheduling heuristics that can be used to dynamically distribute computations across heterogeneous processors. The results from re-running the early experiments using these heuristics are clear: significantly better load-balance is achieved.

2.0 Metasystems

A metasystem is a distributed system composed of heterogeneous hosts (both parallel processors and conventional architectures), possibly controlled by separate organizational entities, and connected by an irregular interconnection network. The interconnection network consists of several different layers of networks, each with potentially different bandwidths and latencies. Some of the subnetworks may be regular, i.e., they can be easily analyzed, while others may be irregular. For example, some subnetworks may be hypercubes or meshes that will be dedicated to the use of the metasystem, while others will carry non-metasystem traffic, e.g., the internet. Figure 1 shows four different possible types of successively larger



(a) A department-wide metasystem consisting of heterogeneous workstations and a single MIMD parallel machine.

(b) A possible nationwide metasystem composed of the University of Virginia campus-wide metasystem and a possible NSF enterprise-wide metasystem.

Figure 1. Different types of metasystems.

and more complex metasystems, department-wide, campus-wide, enterprise-wide, and nationwide. Each is constructed from one or more instances of the smaller systems, tied together by an interconnection network. Note that even in the smallest, department-wide, metasystem of Figure 1 (a) that the network is heterogeneous and multilevel; typical MIMD parallel machines have their own internal network.

A metasystem, though, is more than a collection of connected computers. A metasystem is a seamless computational environment. The user's view of an ideal metasystem is one of a very powerful monolithic virtual machine that provides computational and data storage services. The user need not be concerned with the details of machine and processor type, data representation and physical location of processors and data, or the existence of other competing users. The user does not see system boundaries, only objects, both application objects, i.e. executables, and data objects, i.e. files. Object location and representation are hidden.

Naturally, to achieve high performance, applications in an ideal metasystem are parallel, and often composed of both task and data parallel components. The components have both resource requirements (memory, bandwidth, etc.) and architectural affinities, i.e., certain types of architectures on which they perform better. The parallel nature of the application is hidden.

In our vision of a metasystem user session, the user sits down at a terminal[1] and invokes an application on a data set. It is then the responsibility of the metasystem to *transparently* schedule application components on processors, manage data transfer and coercion, and manage communication and synchronization in such a manner as to minimize[2] execution time via parallel execution of the application components. The user-visible resources are applications, data, and specific I/O devices (the particular terminal). For each visible resource, we mask out, or make transparent, aspects of the underlying system. We must provide architecture/processor type transparency, network/communication transparency, parallelism transparency, location transparency, replication transparency, and fault transparency.

That the construction of a metasystem presents challenges is clear. Many of the difficulties are the result of the component heterogeneity. There are two broad categories of heterogeneity, hardware and software. Hardware heterogeneity has many different facets. The obvious forms of heterogeneity are different processor architectures, instruction sets, data representation formats, and data alignments. These can be accommodated using techniques developed for heterogeneous distributed computing [4][5][6][13][19][28]. The primary question with respect to these techniques is cost: how much overhead do they introduce, and will the overhead cancel out the gains from parallel computation?

Less obvious forms of heterogeneity are related to the configuration of the hosts. Even when two hosts share the same processor architecture, they can differ in ways that can significantly affect both performance and whether a computation can even execute on the hosts. These configuration differences include the clock frequency, the amount of physical memory, the amount of swap and /tmp space, whether the processor has a floating point unit, and so on. All of these factors can influence the quality of a scheduling decision.

A final form of host hardware heterogeneity is the model of computation supported by the hardware, e.g., sequential, SIMD, vector, shared memory MIMD (both UMA and NUMA), and distributed memory MIMD. Some codes will execute much more efficiently on one class of machines. On the other hand, some

1. We use terminal in its most liberal sense. Terminals may include any human I/O device, glass tty, X interfaces, or virtual reality interfaces such as head-mounted displays and data gloves.
2. In general this is NP-hard. We really mean “do a good job” using a heuristic.

codes cannot exploit the capabilities of a certain type of host, e.g., non-vectorizable codes on a vector processor. In any event, to fully utilize a metasystem comprised of heterogeneous model architectures requires knowledge of both the architectures' capabilities and of the software modules' affinities.

The management of software heterogeneity presents a challenge in the metasystem environment as well. Even in a homogeneous hardware environment there may be software heterogeneity. Software heterogeneity includes differences in the host operating system (the services and interfaces provided), the process support, interprocess communication, the compilers available (the languages and versions of languages available, as well as the quality of the generated code), the file system, and the database systems available. These differences must be masked.

2.1 Approach

We consider a metasystem to be an combination of two different types of systems, parallel processing systems (PPS) and heterogeneous distributed computing systems (HDCS) [1][2][4][5][6][11][13][22][23][25][26][30][31][21]. Borrowing from the object-oriented lexicon, any solution to the metasystems problem will inherit attributes and behaviors from both areas (Figure 2). While it is important that we inherit

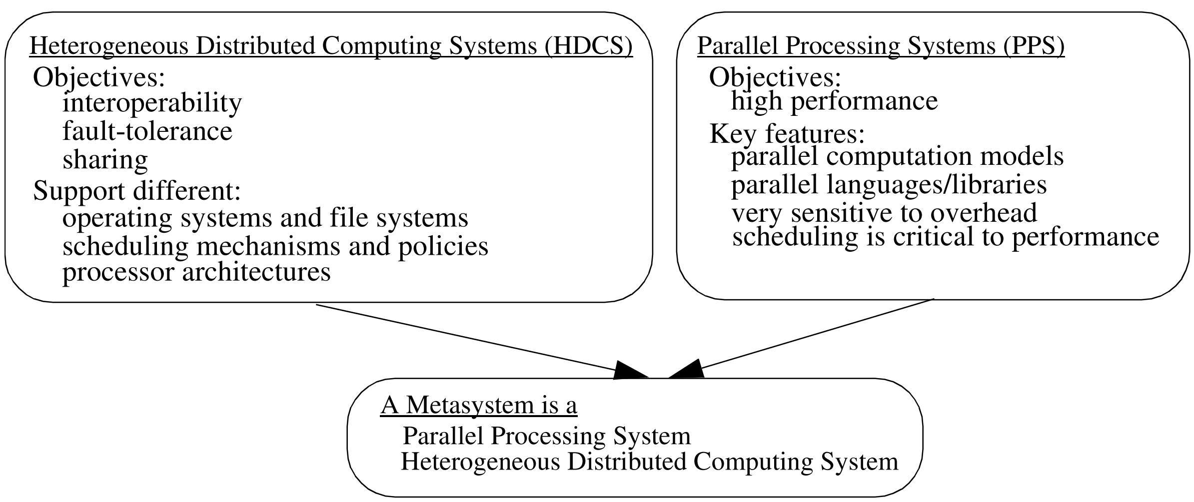


Figure 2. Metasystems inherit objectives, features, problems, technology and techniques from both heterogeneous distributed computing systems and parallel processing systems.

many features from both PPS and HDCS, some features of these systems are at odds with one another. For example, a goal of many HDCS's is interoperability, allowing services on different platforms to transparently operate with one another even if different data formats and communications protocols are used by the services. This is accomplished by multiple layers of translations to a common data format and protocol, often even if the services use the same formats and protocols. Multiple translations take time and increase overhead. This negatively impacts performance. A trade-off must be made between interoperability and performance. In general, decisions about which features to inherit, and how to trade off conflicting goals, are needed.

The ultimate objective of our work is to understand and develop metasystems. Any long journey begins with the first step, and ours is to attack the computational side of the problem, i.e., how to mask the heterogeneity of computation and communication resources from the user. We have not at this stage addressed fault tolerance, file systems, naming, or many of the remaining difficult problems.

Our approach is evolutionary rather than revolutionary. To realize our vision of a seamless metasystem we have chosen to begin by first constructing a metasystem testbed by extending Mentat, an existing object-oriented parallel processing system. The testbed provides us with an ideal platform to rapidly prototype and try out ideas, forcing the details and hidden assumptions to be carefully examined, and exposing flaws in the ideas or in the system components. We chose to extend an existing system rather than redesign and re-implement from scratch primarily because doing so does not require that tremendous amounts of code be written before the first application can be executed. Further, we can incrementally add new capabilities as each problem is tackled and its solution is incorporated. While using an existing system as a starting point will constrain design choices and somewhat limit innovation, it does provide a usable prototype with which to increase our understanding of the problems involved in developing a metasystem.

2.2 The Mentat Metasystem Testbed

We have constructed a metasystem testbed using Mentat. Using the testbed we can measure the performance effects of system and language features on real applications already implemented using Mentat. This is then be fed back into our design effort. We have chosen Gaussian elimination with partial pivoting as a test application for our early work, although other applications are not precluded from executing on the testbed. Gaussian elimination was chosen because it requires frequent communication and synchronization. We next examine the design objectives and constraints of the testbed effort, followed by a brief introduction to Mentat and a description of the test application. For more information on Mentat see [14][15][17].

2.2.1 Testbed design goals and constraints

We have distilled three design goals and two constraints for our metasystem testbed effort. Several of the goals are very closely related.

1) High performance via parallelism. Performance is the *raison d'etre* for metasystems. The system must support easy-to-use parallel processing with large degrees of parallelism. The system must support both task and data parallelism. While it is true that many application component kernels are primarily data parallel, many applications consist of multiple communicating components. Further, because of the nature of the interconnection network, the system must be latency tolerant, and be capable of managing hundreds to thousands of processors. This implies that the underlying computation model and programming paradigms must be scalable.

2) Easy to use. The system must be easy to use or it won't be used except by the most dedicated users. The system must mask the complexity of the hardware environment (heterogeneity, interconnection, etc.), and the complexity of communication and synchronization of parallel processing. Further, the system should provide the user and programmer with a uniform interface to services. Otherwise the user may be

overwhelmed with the different choices and mechanisms needed to accomplish applications tasks. These requirements argue for a high-level language interface. The features and capabilities of the system must be accessible via a high-level language, rather than via a set of library routines available to the user. We feel that to ask programmers to write applications in such a complicated environment using low-level constructs such as send and receive is asking for trouble. It will be far too complex for all but the best of programmers. Even for those, there will be many opportunities that a human will forgo in order to maintain simplicity. Therefore, as much as possible, compilers, acting in concert with run-time facilities, must manage the environment for the user.

3) Exploit heterogeneity for high-performance. The system must be able to exploit the hardware and data resources available in a metasystem, executing sub-tasks of large applications on different heterogeneous processors, and using heterogeneous data sources. Some architectures are better than others at executing particular kinds of code, e.g., vectorizable codes. These affinities, and the costs of exploiting them, must be factored into scheduling decisions and policies.

In addition to the above design goals, our design is restricted by two constraints.

1) We cannot replace host operating systems. This restriction is required for two reasons. First, organizations will not permit their machines to be used if their operating systems must be replaced. Operating system replacement would require them to rewrite many of their applications, retrain many of their users, and possibly make them incompatible with other systems in their organization. Our experience with Mentat indicates that it is sufficient to layer a system on top of an existing host operating system.

2) Performance cannot be sacrificed. If performance is sacrificed, then one of the prime motivations for metasystems is lost.

The reader may note one design constraint that is not in our list: support for existing applications and native operating system services. This is a design goal of many heterogeneous distributed computing projects that we have chosen not to incorporate. We believe that a requirement of supporting existing applications and services is fundamentally at odds with our high-performance objectives. Therefore, we have decided to sacrifice upward compatibility.

2.2.2 Mentat

Mentat is a portable, object-oriented parallel processing system that provides easy-to-use parallelism, high performance via parallel execution, and applications portability across a wide range of platforms. The premise underlying Mentat is that writing programs for parallel machines does not have to be hard. Instead, it is the lack of appropriate abstractions that has kept parallel architectures difficult to program, and hence inaccessible to mainstream, production system programmers.

There are two primary components of Mentat: the Mentat Programming Language (MPL) and the Mentat run-time system (RTS). The MPL is an object-oriented programming language based on C++ [27] that masks the complexity of the parallel environment from the programmer. The programmer uses application domain knowledge to specify those object classes that are of sufficient computational complexity to warrant parallel execution. These classes are called Mentat classes. The granule of computation is the Mentat class member function. Data and control dependencies between Mentat class instances are automatically detected and managed by the compiler and run-time system without programmer intervention.

The Mentat run-time system supports portable parallelism using a classic layered virtual machine model. The virtual machine provides support routines that perform run-time data dependence detection, program graph construction, program graph execution, scheduling, communication, and synchronization. The compiler generates code that communicates with the run-time system to correctly manage program execution [17]. The virtual machine model permits the rapid transfer of Mentat to new architectural platforms. Only the machine-specific components need to be modified. Because the compiler uses a virtual machine model, porting applications to a new architecture does not require any user source-level changes. Once the virtual machine has been ported, user applications are re-compiled and can execute immediately.

The virtual machine design also simplifies the construction of metasystems. There are two advantages the model provides over manual systems. First, a user of an object is not aware of where an object is located (location transparency). All communication is carried out under the control of the run-time system. The run-time system knows the types of machines involved and the class (type) of the data being transported. It is in the perfect position to selectively and transparently coerce data between representations as required. Second, the run-time system, not the user, is responsible for scheduling [16]. The run-time system selects the best host on which to instantiate an object at run-time depending on current system load. It selects the best instance of a particular host type, e.g., hypercube one or hypercube two.

By using Mentat as our starting point we can satisfy two of our three design objectives immediately, high-performance and ease of use. We have a compiler for a high-level parallel language (MPL) that can be modified to support features needed in a heterogeneous environment. Further, we have an extensible run-time system that is known to be portable, and is well suited to accommodate additional machinery to support heterogeneity.

2.2.3 Gaussian elimination with partial pivoting

The problem is to solve for x in $Ax=b$, given A, an NxN matrix, and b, a vector of size N. Our solution consists of a master object and k workers. We partition A and b into equal size pieces by row, giving each worker the same number of rows. The solver has two phases, reducing the system to upper triangular form, and back substitution. The bulk of the work is in reducing the system to upper triangular form. There are N outer iterations in the reduction. In each iteration each worker is given the pivot row. They reduce their portion of the matrix by the pivot row and return a candidate pivot row for the next iteration. The master selects from all candidate pivot rows the row with the largest absolute value in the current column position. This row becomes the pivot row for the next iteration.

Efficient execution of Gaussian elimination in a metasystem environment poses a number of challenges that are unique to the heterogeneous computing environment. They are the topic of the next section.

3.0 Obstacles to High Performance

Gaussian elimination highlights two important obstacles to achieving high-performance in a metasystem, coercion and scheduling. One property of this application is that it requires frequent communication and synchronization. Communication between workers on nodes of different architecture type may require frequent data coercions. The overhead due to data coercion must not dominate the performance gains achieved by using heterogeneous parallel processing.

Another problem experienced in Gaussian elimination is the need for effective scheduling. In particular, data decomposition (or partitioning) is critical to the performance of this application. Recall that the problem is decomposed into a set of workers. In many existing parallel processing systems, the programmer must specify the number of workers to apply to the problem, i.e. choose the granularity. However, different granularities are appropriate for different architectures. Additionally, the data decomposition needed to achieve the best overall load balance depends on the processing power of the nodes used. An equal partition of work will lead to good load balance in a homogeneous environment, but will likely lead to a load imbalance in a heterogeneous environment.

For metasystems to be successful, both of these problems must be overcome. In the next section we discuss the problems of coercion and scheduling in more detail, and present solutions for a class of data parallel problems. Our experiments indicate that coercion costs are quite manageable, and while scheduling in metasystems is an open research problem, effective heuristics can be developed for a class of data parallel problems.

3.1 Overcoming the Obstacles

3.1.1 Data Coercion

The need for data coercion is driven by hardware diversity. Different architectures may have different data representations. Fortunately, there has been extensive experience with this problem in the HDCS community, and there are several well-known solutions [19][25][26].

In the testbed, data coercion is handled by class-specific coercion functions that transform data from one representation to another. Here the structure of the Mentat run-time system simplifies the problem. A set of low-level routines move data in and out of Mentat objects. The MPL compiler generates code to call these functions. The MPL compiler can determine the class of all data structures that will be communicated, and can set up pointers to the appropriate coercion functions. We have extended the data movement functions to include pointers to the coercion functions. Local copies of the RTS store the type of processor on which they are executing. Incoming data is tagged with a source-architecture tag field. If the local architecture and tag field are different, the RTS invokes the appropriate coercion function. Otherwise, it passes the data through unchanged. Another possibility is to coerce the data before it is sent using the same mechanism. In either case, this process can be visualized as a set of filter objects placed along the edges of the program graph (Figure 3) that coerce data if needed, and otherwise pass it through unchanged.

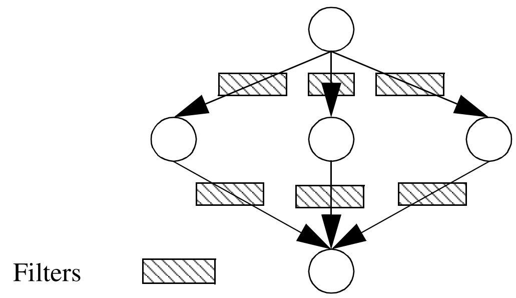


Figure 3. Data coercion filters. The circles are computations. The edges represent data dependencies in the program. Data flows in messages along the edges. The filters coerce the data as necessary.

Currently, the coercion functions are generated by hand. Ultimately, they will be generated by the compiler. We chose to hand-generate the coercion functions so that we could experiment with different coercion function costs and measure the effect on performance before expending the effort to modify the compiler.

We constructed coercion functions with which we could easily simulate a range of coercion costs. We then simulated a network consisting of one, two, and four different processor types using four Sun IPC's in the testbed. Each IPC was labeled with a pseudo-architecture type. If a message arrived from a processor of a different type, the message was coerced. As the number of architecture types is increased, the probability that any given message will require coercion approaches one. For example, if the number of pseudo-architectures is four, then every message from another host will be coerced. We also varied the coercion cost function: coercion function 1 performs a bit-wise XOR for each byte of each word of the data portion of the message, coercion function 2 does the coercion function 1 operation twice, and coercion function 3 performs an endian conversion using byte swapping. We believe that the synthetic coercion functions represent an upper-bound on the most common types of coercion that will be needed.

We measured the wall clock times of Gaussian elimination for a 512x512 problem with four workers, and compared the performance using the two synthetic coercion functions and the endian conversion. The results are shown in Table 1. Speed-up is relative to an equivalent C program[3]. The results show that,

Table 1: Effect of Coercion Costs

Number of pseudo-architectures	Time with coercion function 1 (msec)	Time with coercion function 2 (msec)	Time with coercion function 3 (msec)	Speed-up (function 1, function 2, function 3)
1	27562	27562	27562	3.04, 3.04, 3.04
2	27949	28131	27881	2.99, 2.98, 3.01
4	28059	28273	27922	2.99, 2.96, 3.00

while there is a negative performance impact, it is not significant. One might question the impact of coercion overhead as the system is scaled. If the number of processors is increased, the percentage of communication overhead due to coercion may decrease. As more processors are added for a fixed problem size, the size of individual messages will either decrease or remain the same, and the per-message coercion cost is a function only of the message size. The number of messages will likely increase as the number of processors is increased, but message coercion will often occur in parallel (Figure 3), thus introducing no additional overhead. Furthermore, as the number of processors is increased, communication overhead will likely increase due to increased contention, and coercion will become an even smaller part of the overall communication overhead.

3.1.2 Scheduling

Effective scheduling of program components is critical to the high-performance objective of the metasystem. The general scheduling problem consists of two parts, problem partitioning and placement. Problem partitioning is driven by two criteria: granularity considerations and load balance. The granularity of each scheduled program component must be sufficient for the processor upon which it will run. In general, it is better to constrain parallelism and choose larger grains than to schedule program components that are too fine-grained. Furthermore, the load across all processors should be as balanced as possible. Achieving both of these objectives will lead to reduced completion time.

The second part of the scheduling problem is placement, matching the program component to the "best" available processor. This decision may include granularity, i.e., that the granularity of the component best matches the capabilities of a given processor. It may also include other affinities. For example, the best place to schedule a vectorizable component is on a vector processor, assuming such a processor is available. A data parallel nearest-neighbor two-dimensional grid component might best be placed on a mesh.

The Mentat scheduler [16] implements a dynamic load-sharing policy that is targeted toward homogeneous systems. It places Mentat objects on processors when it receives instantiation requests. The scheduler handles task parallelism very well, although it does not consider object-processor affinity during placement[4]. In the case of data parallel objects, the programmer is responsible for problem partitioning and data decomposition. The scheduler places the objects, but does not assist in deciding how many objects to create or how to decompose the data to the objects. The result is that most data parallel objects in Mentat take the number of workers to create as a parameter, and then evenly distribute the workload. This leads to inferior partitions and load imbalance in a metasystem. Testbed results confirm this problem.

To measure the effect of the load imbalance, we executed Gaussian elimination on a metasystem consisting of two Sun 4/40 IPC's, and two Silicon Graphics Iris (SGI) workstations. The SGI and Sun IPC use the same data format, so no explicit coercion was required. The SGI's are almost twice as fast (1.84 times,

3. The equivalent C program uses the same algorithm as the Mentat version. It is not a Mentat program running on one processor. Both the C program and the Mentat program have the same inner loop, and have had the same level of compiler optimization applied.
4. We will not discuss scheduling tasks because the scheduler manages it well. A complete discussion is beyond the scope of this paper.

floating point) as the Sun IPC's. We conducted two experiments. In the first experiment, we partitioned the work equally among four workers, with one worker on each of the four processors. In the second experiment, we created six workers each with equal work, and scheduled two workers on each of the two SGI's to balance the load. The results are shown in Table 2 for a problem of size 1024x1024. The effect of the

Table 2: Effect of unbalanced load, P=1024.

Number of Workers	Parallel execution time (seconds)	Speedup relative to IPC time	Speedup relative to SGI time	SGI sequential time (seconds)	IPC sequential time (seconds)
4	173	3.87	2.1	368	672
6	131	5.1	2.8	368	672

four-worker schedule is that the two SGI's are not fully utilized.

In the six-worker case we have better utilization, and hence, better performance. The problem with this approach, breaking the problem into more Mentat objects and placing more objects on faster processors, is that overhead is linear with the number of objects. This is a serious problem. In order to get acceptable load balance with an unknown number of variable-power processors, the programmer would need to partition the problem into many relatively fine-grained pieces and let the scheduler place them on processors. Unfortunately, this will lead to increased overhead and poor performance.

What is needed is a mechanism for partitioning the problem into appropriate-sized pieces and distributing the pieces to workers on different processors. If all problems were based on regular two dimensional arrays in which computational effort was linearly dependent on the number of array elements received, a single "hard-wired" mechanism would be adequate. However, not all problems have a regular two dimensional array structure. The mechanism must be extensible in order to support problem-specific decompositions and must automatically partition the problem according to the decomposition. Once the problem is partitioned, the workers must be assigned to processors. Providing such a mechanism is the topic of our next section. We return to the Gaussian elimination example, and show how our approach can be used to get significantly better load balance at lower cost.

3.1.3 Partitioning Data Parallel Components

We have identified three stages in partitioning data parallel components: load selection, data decomposition, and data distribution. The stages are shown in Figure 4. In load selection, we compute how many workers will be applied to the problem, their placement, and the percentage of work that will be allocated to each worker. In our model, each worker is assigned to a separate processor. The load selection phase is driven by granularity and load balance considerations, given information about the computation and communication structure of the data parallel component.

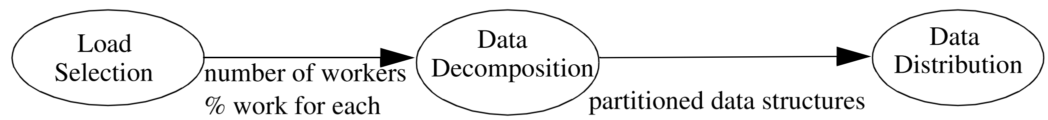


Figure 4. Three-stage data parallel partitioning.

Once a set of workers and a partition of the problem have been determined by the load selection phase, the data domain is then decomposed into work units of the specified size and distributed to the workers. This process is complicated by the fact that not all problems decompose the same way.

To solve each stage of the partitioning problem we have extended the Mentat philosophy of exploiting the strengths of both programmers and computers. The problem is that it is difficult for the compiler to determine the computation granularity for arbitrary problems. It is also difficult for the compiler to determine how to decompose and distribute arbitrary data structures. The programmer, on the other hand, knows the application and can fairly easily provide information about the application such as the amount of computation required for a given problem size[5]. Further, the programmer can most easily specify the way in which the data domain is to be decomposed and distributed. What the programmer cannot do is make good granularity and placement decisions without a great deal of system state information. The run-time system, however, can quickly and accurately determine the number and type of processors to use, and how much work to assign to each, if granularity information can be acquired.

Our solution involves the use of programmer-written functions called *callbacks* that the run-time system invokes to determine computation and communication requirements, decompose problems into variable size partitions, and distribute the partitions to the workers. In the following two sections these callbacks are motivated and described. In order to reduce the number of parameters that must be managed we have restricted the metasystem structure and the application domain. These restrictions are for a proof-of-concept. We plan to relax them later, and extend our approach to encompass both a more complex metasystem, and applications with varying degrees and types of parallelism.

3.2 Load Selection via Callbacks

Load selection is the first and most complex stage in data parallel program partitioning. Recall that the problem is to determine the number of workers, and what percentage of the work will be allocated to each worker. Our approach to load selection is dynamic, and managed by the run-time system (RTS). This is a major improvement over many existing systems in which both of these are done by the programmer. To solve the partitioning problem in the testbed we have made a number of simplifying assumptions.

3.2.1 The Restricted Model

First, while the machines on the heterogeneous network are shared, and in general their load may change unpredictably, we make the assumption that load fluctuation due to other users during program execution is minimal. This assumption allows us to make an initial placement decision based on machine load with-

5. For some regular problems, compilers may be able to automatically determine this information.

out rebalancing during the computation. Second, we assume the simplest heterogeneous network, a single-layer network of single-CPU machines of differing computation power (Figure 5). This assumption allows us to ignore application communication topology when making placement decisions. Third, we assume that the cost of moving data between any of these machines is uniform and depends only upon the interconnection network (IN). We also assume that a set of message cost equations that describe the communication cost of the underlying message delivery system on this IN are known to the RTS. These cost equations are of the form: T (b bytes, k processors) = T_s (k) + bT (1, k). The dependence of T on the num-

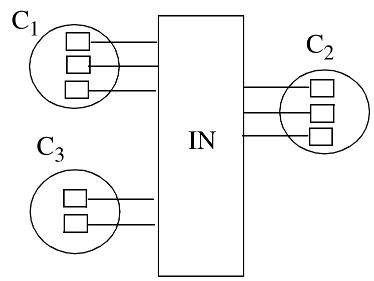


Figure 5. Simple one-layer heterogeneous network.

ber of communicating processors k is one way to model the effect of contention. The term T_s is the start-up time or latency also in usec. Fourth, the number of distinct architecture classes C_1, C_2, ... C_m is known to the RTS *a priori*, as are the following machine parameters:

W_i, a weighting factor that characterizes the relative processor speed of C_i

S_i, a cost function that defines the cost of instruction execution on processors of class C_i in usec

The communication cost functions T, W_i and S_i are determined by benchmarking.

Fifth, since we are looking at data parallel components, we assume the application programmer can determine how to decompose the underlying data structures into regions of the appropriate size. This is best illustrated by regular data parallel problems with uniform computation. For example, a 5-point stencil problem based on a two-dimensional grid can be decomposed by row, column, or block. Consider a row decomposition. The grid is decomposed into some number of adjacent rows given to each worker. We refer to the smallest unit of decomposition as a *minimal work unit*. In the 5-point stencil problem, the *minimal work unit* is a row. The data domain is decomposed into multiples of the minimal work unit. We refer to this partition as a *work region*. The size of a *work region* is the number of *minimal work unit*s that it contains.

3.2.2 Granularity Callbacks

In order to determine the number of workers and a partition of the data domain, the application programmer provides two callback functions that describe the computation and communication structure of the problem:

`f(r)` returns the number of instructions that are executed for a work region of size r between successive communications

`g(r)` returns a tuple (number of bytes communicated/region r, number of communications/r)

For example, consider an NxN 5-point stencil problem with row distribution and the workers communicating in a 1D topology to exchange boundary rows. To keep things simple, suppose that the computation consists primarily of floating point operations, and the stencil grid contains single-precision (4-byte) floating-point values. The callback functions are:

`f(r)` = r*5*N

`g(r)` = (4*N, 2)

For `f(r)`, there are five floating point operations or instructions per row element with 5*N operations total per row, which yields r*5*N for a work region of size r (i.e., r rows). For `g(r)`, each communication consists of a single border row which contains 4*N bytes with two communications required per worker in a 1D (north-south) topology.

These functions are used in a heuristic decision process. The output of the decision process is a *partition vector*. Each element of the partition vector contains the name of a worker, and the number of minimal work units assigned to that worker. The name of the worker includes the processor id. The number of workers selected determines the length of the vector. The decision process is described below.

3.2.3 The Decision Heuristic

Given our simplified model, we now describe the run-time partitioning heuristic for data parallel problems. Since placement is not an issue with respect to communication costs, we can ignore it. Since workers are mapped one-to-one to processors, choosing the number of processors is equivalent to choosing the number of workers. The algorithm we describe here determines the number and type of processors to apply to the problem with a single worker placed on each processor selected. Some notation:

p =	smallest unit of decomposition for the problem, a minimal work unit
P =	total number of minimal work units that must be assigned to processors, e.g. for a 2D row decomposition, P = # of rows
h =	the degree of f(r). For the stencil problem h =1, if f(r) is $O(r^2)$ then h=2
m =	number of architecture classes
M_i =	number of processors of class C_i that will be applied to the problem
n_i =	number of available processors of class C_i
A_i =	size of the work region in multiples of p assigned to each processor in C_i

There are two parts to the load selection problem: 1) determining the M_i for each architecture class C_i, and 2) determining the A_i, the sizes of the work regions that will be assigned to the workers on each of the M_i processors within each class C_i. For a balanced load decomposition, we will assign the same size work region to each processor (i.e., worker) within a class C_i. Thus, a single A_i is computed per class C_i.

We are experimenting with several heuristic algorithms for load selection to compute the partition vector. Because load selection is a run-time process, efficient algorithms are of particular interest. The general form of the load selection algorithms is described shortly. The algorithms begin by computing the M_i for each architecture class, i.e., the number of workers. Once M_i is determined, computing the A_i associated with each architecture class follows naturally from our load balance objective. A_i is a function of the processor power, W_i, the number of processors used, M_i, and the computation complexity, `f()`. Since we have assumed uniform communication cost, A_i does not depend on `g()` directly. The expression for A_i is given below:

$$A_i = P \cdot \frac{(W_i^{1/h})}{\sum_{j=1}^{m} (W_j)^{1/h}}, where \sum_{j=1}^{m} A_j \cdot M_j = P$$

This heuristic for A_i has the following two properties: it yields a load balanced partition of the data domain which can be easily verified, and it covers the entire data domain (i.e., the sum over all A_i equals P). A larger W_i (i.e., a more powerful processor) induces a larger A_i. How much larger depends on the amount of computation performed on a minimal work unit, p. For example, suppose we have C_1 and C_2 with $W_1 = 1$ and $W_2 = 2$. For linear problems where `f(p)` is O(p), A_2 will be twice A_1, since h=1. If `f(p)` is $O(p^2)$, then A_2 will be sqrt(2) times A_1, since h=2.

In order to compute M_i, we assume that the RTS knows the number of "free" processors n_i for each architecture class, where "free" is determined by a threshold policy parameterized by architecture class. For example, all processors below some load threshold are considered to be available. Clearly, M_i must lie between 0 and n_i. The RTS will acquire information about the state of processors on the network as needed, and has this information available at the time a scheduling request comes in. If for some reason this information is stale, the RTS must recollect this information.

The load selection algorithm is guided by a simple heuristic that estimates completion time based on `f()` and `g()`:

$$T_c = T(g((A_i), M_i)) + S_i \cdot f(A_i)$$

The completion time estimate T_c is defined as the sum of communication (the first term) and computation cost (the second term). Since `g()` actually returns a tuple, we assume that the message cost function T has been parameterized appropriately and will compute the total cost of moving all of the messages during the computation for A_i. A heuristic that computes the number and type of processors that minimizes T_c has been developed. We describe the simplest form of the algorithm for a single IN in which communication costs are uniform and processors are idle or lightly loaded.

The algorithm begins by ordering the architecture classes C_1... C_m by decreasing W_i, i.e., the fastest processors will be considered first. The algorithm is greedy in the sense that it first considers processors belonging to the most powerful architecture class. Under our assumptions it can be shown that such an algorithm will lead to the best completion time when the data parallel component is considered in isolation. The algorithm iterates over the architecture classes in decreasing power order, deciding how many processors of each class can be effectively applied to the problem.

For each C_i considered in turn, the algorithm uses a form of binary search to locate M_i. For each tentative value of M_i, A_i is computed by the expression presented earlier. Once A_i is known, T_C can be computed and the value of M_i that induces the smallest T_C is determined. If it is determined that all processors n_i can be effectively used (i.e., n_i induces a minimum T_C), then the granularity is too coarse, and the next strongest processor class is considered, and so on. At this point, it is guaranteed that at least the n_i processors within C_i, in addition to the n_j processors within C_j ($j < i$), will be part of the final allocation. If at any point $M_i < n_i$, then the algorithm terminates with the M_j ($j <= i$), the corresponding A_j, and the associated partition vector.

The completion time estimate T_C assumes that computation and communication are not overlapped and the resulting decisions are therefore conservative[6]. It is not difficult to account for such overlap. If computation and communication were fully overlapped, then in the expression for T_C, "+" would become "max".

If there are m architecture classes and N available processors across all classes, then the worst case running time of the algorithm is $0(m\log_2(N))$. The $\log_2(N)$ term accounts for the binary search to locate M_i within C_i. Since m is typically small, and N is likely to be less than 100 (in a workstation network environment), this algorithm incurs very little run-time overhead. On a Sparc2 with m=4, N=100, and P=100000 (a large problem guarantees that the algorithm will iterate over all classes), the total elapsed time to compute the partition vector for the stencil problem was less than 10 msec using an algorithm that had no optimization whatsoever. Given that Gaussian elimination on a small problem size (512x512) ran in the tens of seconds, a few msecs is quite small. In general, we believe that a problem requiring tens to hundreds of workstations is likely to be long-running and the overhead of run-time partitioning almost negligible. The most important point to make about this algorithm is that it scales well ($O(\log_2)$ as the number of processors increases), and has very small constant factors.

3.3 Data Decomposition and Distribution via Callbacks

Once the partition vector has been determined, the program's data must be partitioned (decomposed) and distributed accordingly. We will use the *master/worker* model as a framework for presenting our solution to the decomposition and distribution problem. In general, the characteristics of the application's communication topology must be reflected in the data decomposition. We intend to address these considerations in future work. For now, we are developing a suite of common data structures whose communication requirements are well known (e.g., 2D stencil problems, 2D arrays, and trees). These classes will be provided for experimentation and development.

3.3.1 Master/Worker Model

In the master/worker model, a computation is divided among several worker objects that are coordinated by a master object. Each worker object is responsible for performing its computation on some region of the

6. Overlapped computation and communication is supported in Mentat.

data set. The worker objects map one-to-one to the set of processors selected to execute the computation. The master object hides the distributed nature of the computation from the programmer by providing an interface that allows the worker collection to be treated as a unit.

In Figure 6, we show the master/worker model as it relates to our testbed. The master object interacts with the RTS via `the load_select()` function which is discussed below. The `load_select()` function computes M_i and A_i for all classes C_i, creates a worker object on each selected processor, and passes back to the master the names of the worker objects and the size of their data regions in the partition vector, `w_p[]`. For each processor *j* selected, `w_p[j].object` is the name of the worker located on that processor, and `w_p[j].partition` is the associated A_i in minimal work units.

Once the RTS has created the worker objects via the `load_select()` function, the master passes the appropriate data region to each worker via the `initialize()` function. The details of the `initialize()` function are discussed below. The nature of the data region depends on the problem at hand. For instance, it may be a physical data region, or simply indices into the files that contain the data.

Once the workers have been initialized, the actual computation may begin. The master engages the workers to perform their specified operations, and then collects any return results. At this point, the user may request the results from the master.

3.3.2 Partitioning Gaussian Elimination

Our Gaussian elimination implementation uses the master/worker model outlined above. The class definitions are shown in Figure 7. The master class is called `gauss_master` and the worker class is called `gauss_worker`. The details of the implementation are discussed below. We use this example as a framework to illustrate how data partitioning and distribution is achieved using callbacks. The ideas presented in this example are applicable to many parallel processing applications.

There are two steps involved in the decomposition and distribution of data. The data must first be partitioned to match the granularity specification for each worker, and then each data partition must be communicated to its associated worker. To achieve this, the programmer must define several member functions. A *distribute* function is defined in the master class, an *initialize* function is defined in the worker class, and a *carve* function is defined for each data structure that must be distributed. The actual names of these func-

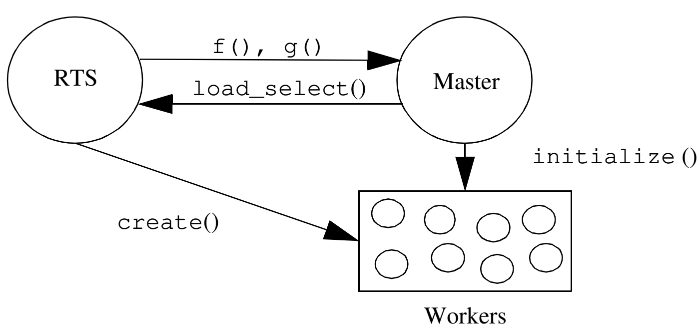


Figure 6. Master/Worker Model - The master invokes `load_select()` which in turn calls `f()` and `g()`, then creates the workers and returns resultant `w_p` partition_vector back to the master.

tions are not important since they are used only within code written by the programmer.

The Gaussian elimination computation begins when the `solve()` function is invoked on the master object. This function acquires the partition vector from the RTS, partitions and distributes the data, then performs the computation. The first step is to obtain the partition vector. This is done by calling the RTS function `load_select()`. `Load_select()` returns the partition vector in the variable named `w_p` (Figure 7, line 31). The partition vector contains the names of the workers, and the number of minimal work units allocated to each.

Next, the data is partitioned and distributed by invoking the `distribute()` function (Figure 7, line 32). We use two distributed data structures in our example, the matrix containing the equation coefficients, and a vector that will contain the problem solution. Both of these data structures are implemented using the `DD_floatarray` class which represents an in-core two-dimensional array of floating-point numbers. The `DD_floatarray` class has a member function, `carve_by_rows()`, that takes the worker number and the partition vector as parameters. It returns a `DD_floatarray` containing the appropriate rows.

The `distribute()` function takes one argument, the partition_vector `w_p[]`. It "carves" the data

```
1:  class DD_floatarray: public DD_array {
2:        // Other member functions
3:        DD_floatarray* carve_by_rows(int piece,partitition_vector *w_p);
4:  };
5:
6:  class gauss_master {
7:        // Other member functions
8:        int f(int r); // Returns computation complexity.
9:        tuple g(int r); // Returns communication complexity
10:       void distribute(partition_vector &w_p);
11:       DD_floatarray *gauss_master::solve(DD_floatarray*, DD_floatarray*)
12: };
13:
14: class gaussworker {
15:       void initialize(DD_float_array *mat,DD_float_array *vec);
16: };
17:
18: void gauss_master::distribute(partition_vector &w_p) {
19:       int j;
20:       DD_float_array *arg1, arg2;
21:       for (j = 0;j < w_p.count(); j++) {
22:             arg1 = A->carve_by_row(j,w_p);
23:             arg2 = b->carve_by_row(j,w_p);
24:             (w_p[j].object).initialize(arg1, arg2);
25:       }
26: }
27:
28: DD_floatarray *gauss_master::solve(DD_floatarray *A, DD_floatarray *b) {
29:       partition_vector *w_p;
30:       ...// set up local data structures
31:       w_p = load_select(this);
32:       distribute(*w_p);
33:       ...// actually do the work solving the system.
34: }
```

Figure 7. Gaussian Elimination Class Definitions

structures into the work regions specified by the partition_vector, then initializes each worker with its corresponding region. To do this, it iterates over the workers specified in the partition_vector (Figure 7, lines 21-25). The function `w_p.count()` returns the number of workers to be used. For each worker *j*, the data regions given by `w_p[j].partition` are extracted using the `carve_by_rows()` functions (Figure 7, lines 22, 23). A work region may contain portions of multiple data structures. This is accommodated by providing multiple carve functions, one for each partitionable data structure. Each carve function specifies how to extract one minimal work unit of a data structure, and allows multiples of this minimal unit to be extracted. The net effect is that each worker receives the data that defines its work region. The actual data can be marshalled into the arguments of the initialize function, or alternatively, the arguments can consist of descriptive information that directs the worker to its data (e.g. a location on disk). This last method is preferable when large amounts of data are required for each worker.

Once the data regions have been extracted, they are passed to the `initialize()` function which is invoked on worker *j* whose name is contained in `w_p[j].object` (Figure 7, line 24). At this point, the data has been distributed and control returns to the `solve()` function. We are now ready to perform the actual computations.

In summary, the programmer's involvement in the data decomposition and distribution process consists of the specification of a *distribute* function, a *carve* function for each distributed data structure, an *initialize* function for the workers, and the proper calling of the `load_select`() and distribute functions. The system takes care of determining which processors will be used and how much work will be assigned to each.

3.3.3 Discussion

The mechanism presented in this section solves some of the more difficult programming problems in a heterogeneous system. First, the system determines the number of workers to be applied to the problem based on the current system state. In existing systems, this decision is made during implementation or by application code at run time, and represents at best a "good guess" by the programmer. Moving this responsibility into the system domain removes from the programmer the onus of knowing how many processors will be available, the relative computation power of the available processors, and the current load on the available processors.

Second, the system determines which processors to use based on the processor's capabilities. This decision is made in concert with a determination of the proper workload for each processor in the processor set. Typically, this granularity decision can be quite difficult to make in a heterogeneous environment. Our solution relieves the programmer of this burden.

Lastly, the actual decomposition and distribution of the data is greatly simplified. The programmer provides a function that is used to extract pieces of the data set, and a fairly simple distribute function that handles the actual extraction and distribution of the data to the workers. While these functions may appear complicated, library classes can be constructed for the standard data structures. These classes can be easily extended via inheritance.

Our decomposition strategy is limited to static (though not necessarily regular) decompositions where the amount of computation performed on a minimal work unit is known and is not data dependent. If the amount of computation is data dependent, this strategy will not be not always perform well. A second limitation is that the external load caused by other users may vary, the load then becomes unbalanced. In these cases, some form of dynamic rebalancing, as is provided in DataParallel C [24], may be needed. We are investigating rebalancing mechanisms as well.

3.4 Effect of Partitioning

To test the efficacy of our approach, we applied the problem partitioning technique to Gaussian elimination. The resulting workers with their computed problem sizes were scheduled and executed using Mentat. We used a 4-node heterogeneous network consisting of two SGI's (C_1), $M_1 = 2$, and two Sun-4 IPC's (C_2), $M_2 = 2$. The SGI's are almost twice as fast as the IPC's, thus $W_1 = 2$ and $W_2 = 1$. The testbed uses a portable communication library, MMPS[18] based on sockets. The message cost equation for MMPS is $T(n) = 2.65 + 0.0015n$. The problem size is P = 1024 for a 1024x1024 matrix.

The results are shown in Table 3. For comparison, the results of the experiment described in Section

Table 3: Effect of balanced load, P=1024.

Work partitions (SGI, IPC)	Parallel execution time (seconds)	Speedup relative to IPC time	Speedup relative to SGI time	SGI sequential time (seconds)	IPC sequential time (seconds)
(256, 256)	173	3.87	2.1	368	672
(341, 171)	121	5.53	3.03	368	672

3.1.2 are shown in the first row (the 256,256 partition). Since this is a row decomposition, the work partitions refer to the number of rows in each worker's work region. Since the amount of work between successive communications of the candidate pivot is linear in the number of rows, i.e., O(r), we would expect load balance to be achieved when the SGI's are given about twice as much work (ratio of W_1/W_2) as the IPC's. The results indicate that such a distribution of work, where each SGI gets 341 rows and each IPC gets 171, results in reduced completion time and a speedup very close to optimal. The speedup is superlinear with respect to the IPC sequential time, and around 3 with respect to the SGI sequential time. This result is intuitive since two IPC's are approximately equal in compute power to one SGI.

When compared to the performance of six equal size pieces (Table 2) the improvement of the (341,171) partition is less dramatic, 121 versus 131 seconds. However, an approach based on breaking the problem into many equal size pieces and then placing the pieces based on processor power is not scalable. Many more workers than processors would be required to achieve good load balance on a large system[7]. Since communication in Mentat is usually linear in the number of workers, such an approach would greatly increase overhead. Therefore, breaking the problem up into many relatively fine-grain pieces is not a viable alternative.

4.0 Related work

Heterogeneous distributed computing systems have been an active area of research for some time [2][4][5][6][13][22][23][25][26][30][31]. Our work differs from the work in the HDCS community in our objectives and the trade-offs we are willing to make. The primary objectives in much of the HDCS work are interoperability, sharing, and availability. Unlike our work, high performance is not the objective.

Applications portability across parallel architectures is an objective of many projects. Examples include PVM [29], Linda [7], the Argonne P4 macros [1], and Fortran-D [12]. Our effort shares with these and other projects the basic idea of providing a portable virtual machine to the programmer. The primary difference is the level of the abstraction. Low-level abstractions such as in [1][8][29] require the programmer to operate at the assembly language level of parallelism. This makes writing parallel programs more difficult. Others [3][12] share our philosophy of providing a higher level language interface in order to simplify applications development. Our work differs from Fortran D [12] in several ways. First, Fortran D is portable but is not targeted to a heterogeneous environment. Second, Fortran D supports data parallelism only, not both task and data parallelism. Third, Fortran D "hard-wires" support for low-dimensional arrays into the language, while our approach is to allow the user to provide arbitrary distribution functions for arbitrary data structures.

Our work differs from HeNCE [3] in several ways. First, HeNCE is explicitly parallel. The programmer specifies program graph nodes and the data dependencies between them. In Mentat, the program graphs are transparently constructed for the programmer. Second, in HeNCE graph nodes are pure functions. HeNCE essentially implements coarse-grain data flow. Mentat supports both pure, side-effect free functions and persistent objects. We have found persistent objects to be very useful. Not only do they model many problems well, but the use of explicit state can significantly reduce the overhead associated with data flow, improving performance.

Recently there have been efforts towards combining parallel processing and heterogeneity [3][8][9][10][20][24][29]. Once again, the level of service distinguishes our work from systems such as PVM [29]; they enable, rather that support, heterogeneous parallel computing. Our work differs from that of Paragon [9] and Dataparallel C [24] in several ways. First, as with Fortran D, Paragon and Dataparallel C are data parallel only, and support only regular multi-dimensional arrays. Second, they support processor heterogeneity only in the processor power sense. They cannot accommodate different data representations. Our method of determining the size of the partitions is similar in some respects to Paragon. However, we dynamically partition the work based on resource availability and problem size, while it appears that their work is targeted toward compile time analysis.

7. Having many more workers than processors is similar to the concept of virtual processors. Load balance is achieved by scheduling virtual processors to physical processors such that each processor has a number of virtual processors proportional to its power and external load. This approach is used in [24], and is best for non-communicating virtual processors, or when the communication pattern is very regular, i.e. stencils.

5.0 Summary and Future Work

In this paper we have introduced our approach to constructing high-performance, easy-to-use metasystems comprised of heterogeneous components. Our approach combines our previous work in portable parallel processing with ideas from heterogeneous distributed computing, inheriting features from each to achieve our goals. Our approach is evolutionary. Rather than constructing a system from the ground up, we have built on Mentat, an object-oriented parallel processing system, and extended it to support heterogeneity. The resulting system is an excellent testbed for new ideas. We have used the testbed to conduct experiments on the efficacy of our approach.

We have found that data coercion costs are not a serious impediment to high-performance. We have also found that load imbalance in data parallel components, caused by differences in processor capability, can lead to reduced performance if problems are decomposed into equal size partitions. To address this problem, we have developed a technique using simple heuristics and application callbacks that enables the run-time system to make the decisions of both how many workers to use, thus specifying granularity, and how much work to allocate to each processor, based on the processors' capability. The RTS then returns a partition to the application, which in turn distributes the work, usually in the form of data structures, to the workers. This technique significantly improves performance over equal size partitions. Further, this technique is very flexible. It can be used for wide range of data structures with arbitrary computational complexities, not just regular problems with low-dimension arrays.

Further work in this project falls into four areas, expanding to larger more diverse systems, developing scheduling models and heuristics for multi-level networks, examining techniques for performing dynamic load rebalancing, and developing language mechanisms to more clearly express the distribution of work and the topological relationship between the workers. We have, for example, recently acquired access to the new Sandia Heterogeneous Environment and Testbed (HEAT), a collection of fifty high performance workstations connected by fiber. The testbed contains ten workstations from each of five vendors. We intend to port our metasystem testbed to the Sandia HEAT to further test our approach.

6.0 References

[1] J. Boyle et al., *Portable Programs for Parallel Processors*, Holt, Rinehart and Winston, New York, 1987.

[2] E. Balkovich, S. Lerman, and R.P. Parmelee, "Computing in Higher Education: The Athena Experience," *Communications of the ACM* vol. 28, no. 11, pp. 1214-1224, November, 1985.

[3] A. Beguelin et al., "HeNCE: Graphical Development Tools for Network-Based Concurrent Computing," *Proceedings SHPCC-92*, pp. 129-136, Williamsburg, VA, May, 1992.

[4] G. Bernard et al., "Primitives for Distributed Computing in a Heterogeneous Local Area Network Environment", *IEEE Trans on Soft. Eng*. vol. 15, no. 12, December 89.

[5] B. N. Bershad, and H. M. Levy, "Remote Computation in a Heterogeneous Environment." Tech. Rep. 87-06-04, Dept. of Computer Science, University of Washington, Seattle, June, 1987.

[6] B. N. Bershad, et al., "A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems," *IEEE Trans. Software. Eng. SE*, vol. 13, no. 8, pp. 880-894, August, 1987.

[7] N. Carriero and D. Gelernter, "Linda in Context," *Communications of the ACM*, vol. 32, no. 4, pp. 444-458, April, 1989.

[8] N. Carriero, D. Gelernter, and T.G. Mattson, “Linda in Heterogeneous Computing Environments,” *Proceedings of WHP 92 Workshop on Heterogeneous Processing*, IEEE Press, pp. 43-46, Beverly Hills, CA, March, 1992.

[9] A.L.Cheung, and A.P. Reeves, “High Performance Computing on a Cluster of Workstations,” *Proceedings of the First Symposium on High-Performance Distributed Computing*, pp. 152-160, Syracuse, NY, Sept., 1992.

[10] R. F. Freund and D. S. Cornwell, “Superconcurrency: A form of distributed heterogeneous supercomputing,” Supercomputing Review, Vol. 3, Oct. 1990, pp. 47-50.

[11] G. Fox et al.,*Solving Problems on Concurrent Processors Volume I*, Prentice Hall, Englewood Cliffs, NJ, 1988.

[12] G. C. Fox, et al., “Fortran D Language Specifications,” Technical Report SCCS 42c, NPAC, Syracuse University, Syracuse, NY.

[13] P. B. Gibbond, “A Stub Generator for Multi-Language RPC in Heterogeneous Environments,” *IEEE Trans. Software. Eng. SE*, vol. 13, no. 1, pp. 77-87, January, 1987.

[14] A. S. Grimshaw, “Easy to Use Object-Oriented Parallel Programming with Mentat,” *IEEE Computer*, pp. 39-51, May, 1993.

[15] A. S. Grimshaw, W. T. Strayer, and P. Narayan, “Dynamic Object-Oriented Parallel Processing,” *IEEE Parallel & Distributed Technology: Systems & Applications*, pp. 33-47, May, 1993.

[16] A. S. Grimshaw and V. E. Vivas, “FALCON: A Distributed Scheduler for MIMD Architectures”, *Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems*, pp. 149-163, Atlanta, GA, March, 1991.

[17] A. S. Grimshaw, J. B. Weissman, and W. T. Strayer, “Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing,” Computer Science Technical Report, CS-93-40, University of Virginia, July, 1993.

[18] A. S. Grimshaw, D. Mack, and T. Strayer, “MMPS: Portable Message Passing Support for Parallel Computing,” *Proceedings of the Fifth Distributed Memory Computing Conference*, pp. 784-789, Charleston, SC., April 9-12, 1990.

[19] M. Jones, R. F. Rashid, and M. R. Thompson, “An Interface Specification Language for Distributed Processing.” *Proceedings of the 12th ACM Symposium on Principles of Programming Languag*es, pp. 225-235, 1985.

[20] Ashfaq Khokhar, et. al., “Heterogeneous Supercomputing: Problems and Issues,” *Proceedings of WHP 92 Workshop on Heterogeneous Processing*, IEEE Press, pp. 3-12, Beverly Hills, CA, March, 1992.

[21] T. G. Lewis and H. El-Rewini, *Introduction to Parallel Computing*, Prentice Hall, Englewood Cliffs, NJ, 1992.

[22] J.H. Morris, et al., “Andrew: A distributed personal computing environment”, *Communications of the ACM*, vol. 29, no. 3, March 1986.

[23] R. Mirchandaney, D. Towsley, and J. Stankovic, ‘Adaptive Load Sharing in Heterogeneous Distributed Systems,’ *Journal of Parallel and Distributed Computing*, Academic Press, no. 9, pp. 331-346, 1990.

[24] N. Nedeljkovic, and M.J. Quinn, “Data-Parallel Programming on a Network of Heterogeneous Workstations,” *Proceedings of the First Symposium on High-Performance Distributed Computing*, pp. 28-36, Syracuse, NY, Sept., 1992.

[25] D. Notkin., et al.,“Heterogeneous Computing Environments: Report on the ACM SIGOPS Workshop on Accommodating Heterogeneity,” *Communications of the ACM*, vol. 30, no. 2, pp. 132-140, February, 1987.

[26] D. Notkin, et al., “Interconnecting Heterogeneous Computer Systems,” *Communications of the ACM*, vol. 31, no. 3, pp. 258-273, March, 1988.

[27] B. Stroustrup, *The C++ Programming Language*, 2nd ed. Addison-Wesley, Reading, Mass., 1991.

[28] Sun Microsystems. *External Data Representation Reference Manual.* Sun Microsystems, Jan. 1985.

[29] V.S. Sunderam, "PVM: A framework for parallel distributed computing," *Concurrency: Practice and Experience*, vol. 2(4), pp. 315-339, December, 1990.

[30] M.M. Theimer, and B. Hayes, "Heterogeneous Process Migration by Recompilation," *Proc. 11th Intl. Conference on Distributed Computing Systems*, Arlington, TX, May, 1991, pp. 18-25.

[31] S. A. Yemini, et al., "Concert: A High-Level-Language Approach to Heterogeneous Distributed Systems", *Proc. 9th IEEE International Conference on Distributed Computing Systems*, Newport, CA., pp. 162-171, June, 1989