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

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

Technical Report No. CS-92-43
December 29, 1992

This work was partially funded by NSF grants ASC-9201822 and CDA-8922545-01, NASA grant NGT-50970, and NLM grant LM04964969.

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

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

Abstract

Large metasystems comprised of a variety of interconnected high-performance architectures are becoming available to researchers. To fully exploit these new systems, software must be provided that is easy to use, supports large degrees of parallelism in applications code, and manages the complexity of the underlying physical architecture for the user. This paper describes our approach to constructing and exploiting metasystems. Our approach inherits features of earlier work on parallel processing systems and heterogeneous distributed computing systems. In particular, we build on Mentat, an object-oriented parallel processing system developed at the University of Virginia that provides large amounts of easy-to-use parallelism for MIMD architectures.[1]

1.0 Introduction

The task of programming parallel MIMD architectures is plagued by three problems. First, writing parallel programs by hand is very difficult. The programmer must manage communication, synchronization, and scheduling of tens to thousands of independent processes. The burden of *correctly* managing the environment often overwhelms programmers, and requires a considerable investment of time and energy. Second, once implemented on a particular MIMD architecture, the resulting codes are usually not usable on other MIMD architectures; the tools, techniques, and library facilities used to parallelize the application are specific to a particular platform. Thus, considerable effort must be re-invested to port the application to a new architecture. Given the plethora of new architectures and the rapid obsolescence of existing architectures, this represents a continuing time investment, and discourages users from parallelizing their code in the first place. Third, a wide variety of systems are often available to researchers on a LAN or WAN. Even if the researchers have been able to port their application to each of the architectures, there is no software support to treat the collection of heterogeneous hardware entities as a metasystem that *transparently* schedules hardware resources, and manages data coercion, communication, and synchronization across the different platforms. Instead, the additional complexity of the heterogeneous environment must be managed by the programmer, further complicating the already difficult task of writing parallel software.

We are currently addressing the ease-of-use and portability issues using Mentat [13-16], a medium-grain

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

object-oriented parallel processing system that has been under development for four years at the University of Virginia. Mentat has clearly demonstrated that portable parallel software can be readily and easily developed if the right tools are available [18,22].

In this paper we address the third of these problems, the construction and exploitation of metasystems. Our approach is to combine ideas from both heterogeneous distributed computing systems and parallel processing systems. Specifically, we have extended Mentat from a portable, homogeneous parallel processing system, to a heterogeneous parallel processing system. This has been accomplished by incorporating ideas from heterogeneous distributed computing into the Mentat programming language and run-time system.

What we have found is that once the traditional problems of heterogeneous computing such as data format and alignment have been overcome, that scheduling, and especially problem decomposition and placement, rapidly come to dominate the performance of data parallel components. We will show that these scheduling problems can be overcome using relatively simple heuristics if appropriate call-backs are provided by the application. These call-backs permit the scheduler to determine the computation and communication complexity of data parallel operations. This information is used to decompose the problem, place the units of work on appropriate processors, and provide good load balance.

The most interesting aspect of the call-back mechanism that we have developed is the flexibility that it provides. The call-back mechanism can be used to partition not only regular, two dimensional array problems, but any problem for which suitable call-backs can be constructed, e.g., trees, three dimensional arrays, and so on. This is important, as many systems work only for regular structures.

We begin our presentation by defining exactly what we mean by a metasystem and by providing a vision of an ideal metasystem. We discuss our design constraints, many of which are driven by practical considerations, not by fundamental principles. We then present our approach, beginning with an introduction to Mentat, a discussion of our heterogeneous testbed, and of 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 with Mentat can achieve better performance than the alternative of not using heterogeneous resources. Second, if the work in a data parallel component is divided evenly between workers, a load imbalance between the processors will develop because the fast processors will finish their work before the slow processors. This leads to reduced utilization of the faster processors, and hence, worse performance than is possible under a distribution that reflects the processor capabilities.

Having motivated the application decomposition problem, we present a restricted system model and a set of call-backs and scheduling heuristics that can be used to dynamically decompose applications across heterogeneous processors. We then present results from re-running the early experiments using the output from the heuristics. The results are clear: significantly better load-balance is achieved.

2.0 Metasystems

A metasystem is a 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 metasystem shown in Figure 1 is an example of the kind of system which can be

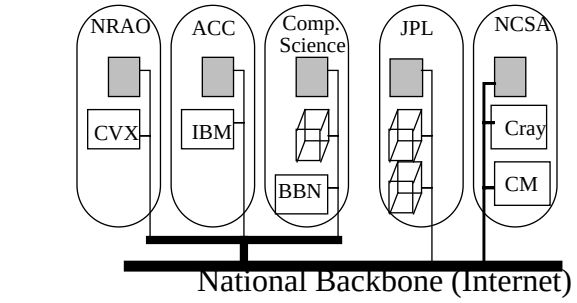

Figure 1: Example of a Metasystem.

utilized if the resources are properly managed. It is composed of systems operated by five distinct organizations, the National Radio Astronomy Observatory (NRAO) in Charlottesville, the Academic Computing Center of the University of Virginia (ACC), the Computer Science Department of the University of Virginia, The Jet Propulsion Laboratory at Caltech, and the National Center for Supercomputer Applications (NCSA). Each organization has a collection of hosts connected together by an interconnection network.

Why are metasystems useful? The most compelling reason is high-performance. There are users whose need for speed is insatiable. The performance of a metasystem is potentially greater than that of any one of its individual components. Also, some applications are composed of different modules that are best suited to different platforms. On a metasystem with a diversity of architectures, each module can be scheduled on an appropriate platform.

Before we continue, we define a metasystem more precisely. We define a metasystem to be composed of a heterogeneous, hierarchical interconnection network, and of heterogeneous hosts. 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. Exploiting the capability of the network and maintaining locality are both critical and difficult in such an environment.

The hosts in a metasystem are heterogeneous. The types of heterogeneity fall into two broad categories, hardware and software. Hardware heterogeneity has many different faces. The obvious forms of heterogeneity are different processor architectures, instruction sets, data representation formats, and data align-

ments. These can be accommodated using techniques developed for heterogeneous distributed computing such as data coercion [4-6,12,19,27]. 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 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 code), the file system, and database systems available. These differences must be masked.

The way users and applications programmers view an ideal metasystem should not be confused with its physical structure. The user's view of a metasystem should be one of a monolithic virtual machine that provides computational and data storage services. The user does not want to 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. In an ideal meta-system, the user/programmer will be presented with a very powerful virtual machine. The user will not see system boundaries, only objects. The objects, both application objects, i.e. executables, and data objects, i.e. files, will be "seen" in much the same way that they are "seen" on contemporary networked workstations: only their names will be seen. The location of the objects and their representation will be irrelevant.

Applications in the ideal metasystem will be parallel, and composed of both functional 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 will perform better. These requirements and affinities will be handled by the metasystem software.

In our vision of a metasystem user session, the user sits down at a terminal[2] and invokes an application

2. 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.

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[3] 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.

To begin to realize our vision of a metasystem, we have chosen to extend an existing object-oriented parallel processing system, Mentat. The advantage of this approach is that the existing system can be used as a testbed to experiment with ideas for solutions to the numerous problems, without requiring that tremendous amounts of code be written before the first application can be executed. Further, one 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.

2.1 Testbed design goals and constraints

The ultimate objective of our work is to understand and develop metasystems as described above. Taken as a whole it is too ambitious. As any long journey begins with the first step, our first step 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 are not, at this stage, going to address fault tolerance, file systems, naming, or many of the remaining difficult problems. 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 functional 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. In particular, care must be taken when accommodating heterogeneity to keep overhead down.

2) Easy to use. The system must be easy to use or it won't be used. 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-

3. In general this is NP-hard. We really mean "do a good job" using a heuristic.

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 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) [1,2,4-6,12,20,21,24,25,29,31]. Borrowing from the object-oriented lexicon, any solution to the metasystems problem will inherit attributes and behaviors from both areas (Figure 2 below). While it is important that we inherit 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.

Our plan for metasystems design and implementation is to begin with a PPS, Mentat, and to experiment

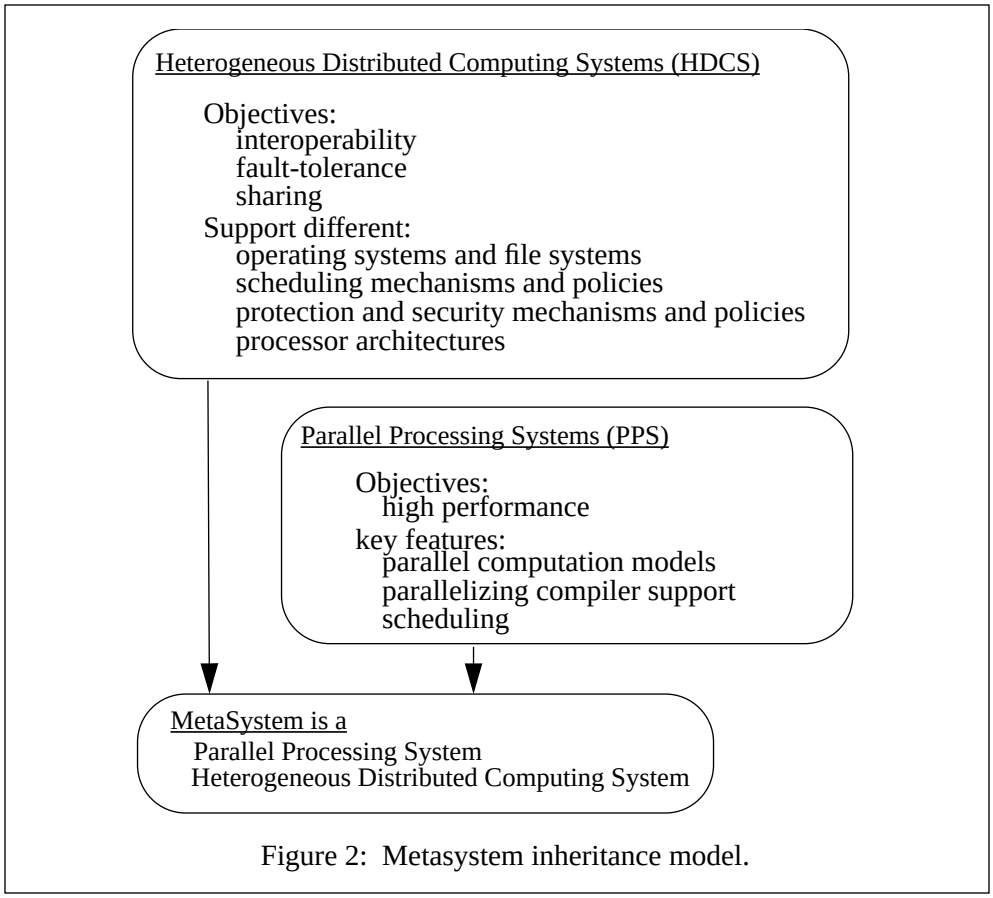

Figure 2: Metasystem inheritance model.

with and construct a metasystem testbed by inheriting those HDCS features and approaches that help us satisfy our goals. We confess that in our approach there is a bias toward PPS goals. This bias follows both from our high-performance goals and from our background. The testbed provides us with an ideal platform for trying out ideas, forcing the details and hidden assumptions to be carefully examined, and exposing flaws in the idea or in the system components. This ability to rapidly prototype ideas and test them is a benefit of the testbed approach.

Below we briefly describe Mentat. Additional information is available in [13-16]. We then describe our plan for testbed construction.

2.3 Mentat

The three primary design objectives of Mentat are to provide: 1) easy-to-use parallelism, 2) high performance via parallel execution, and 3) 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.

The Mentat approach exploits the object-oriented paradigm to provide high-level abstractions that mask the complex aspects of parallel programming, communication, synchronization, and scheduling from the programmer. Instead of worrying about and managing these details, the programmer is free to concentrate on the

details of the application. The programmer uses application domain knowledge to specify those object classes that are of sufficient computational complexity to warrant parallel execution. The complex tasks are handled by Mentat.

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++ [26] that masks the complexity of the parallel environment from the programmer. The granule of computation is the Mentat class member function. Mentat classes consist of contained objects (local and member variables), their procedures, and a thread of control.

Mentat extends object encapsulation from implementation and data hiding to include parallelism encapsulation. Parallelism encapsulation takes two forms that we call *intra-object* encapsulation and *inter-object* encapsulation. Intra-object encapsulation of parallelism means that callers of a Mentat object member function are unaware of whether the implementation of a member function is sequential or parallel. Inter-object encapsulation of parallelism means that programmers of code fragments (e.g., a Mentat object member function) need not concern themselves with the parallel execution opportunities between the different Mentat object member functions they invoke. Thus, the data and control dependencies between Mentat class instances involved in invocation, communication, and synchronization are automatically detected and managed by the compiler and run-time system without further programmer intervention.

The Mentat run-time system supports portable parallelism using a 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.

The virtual machine model (Figure 3) 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.

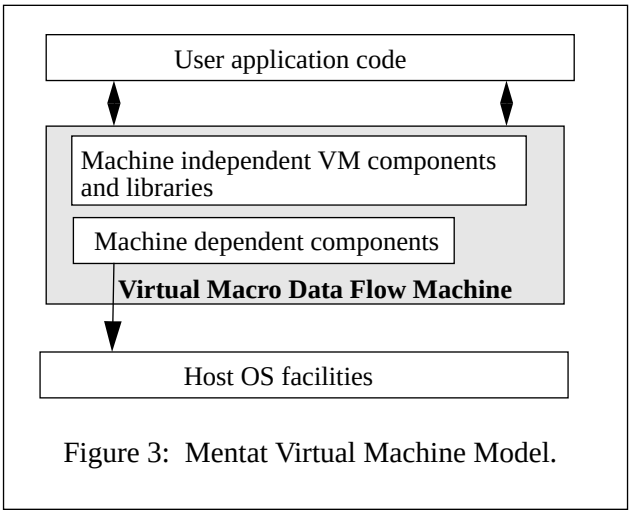

Figure 3: Mentat Virtual Machine Model.

Another benefit of the virtual machine design lies in the exploitation 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 [15]. 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.

2.4 The Mentat Metasystem Testbed

We have constructed a metasystems testbed using Mentat. We can measure the performance effects of system and language features on real programs. This can then be fed back into our design effort. Also, it is a flexible environment for experimentation. 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.

The application: Gaussian elimination:

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 metasystems environment poses a number of challenges that are unique to the heterogeneous computing environment. In the next section, we discuss these problems.

3.0 Obstacles to High Performance

Gaussian elimination highlights two important obstacles to achieving high-performance in a heterogeneous environment, 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, 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 decomposition needed to achieve the best overall load balance depends on the power of the nodes used. In a homogeneous environment, an equal partition of work will lead to good load balance, but will likely lead to a load imbalance in a heterogeneous environment.

3.1 Overcoming the Obstacles

In this section, we discuss the problems of coercion and scheduling in more detail, and offer potential solutions. Our experiments indicate that coercion costs are quite manageable, and while scheduling in heterogeneous systems is an open research problem, effective heuristics can be developed for a useful class of problems.

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 [19,24-25]. In the testbed, data coercion is handled by class-specific coercion functions that transform data from one representation to another. The coercion functions for system internal structures have been generated by hand. They enable communication between system components.

Coercion of application data structures is handled similarly. Here the structure of the Mentat implementation 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 knows at compile time 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 know on what type of processor they are executing. Incoming data is tagged with a source architecture tag field. If the two (local architecture and tag field) are different, the RTS invokes the appropriate coercion function. Otherwise it passes the data through. One way to think of this is as a set of filter objects

placed along the edges of the program graph (Figure 4) that coerce data if needed, and otherwise pass it

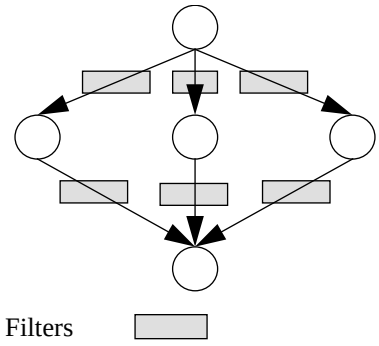

Figure 4: Use of data coercion filters.

through.

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 functions costs and measure the effect on performance before we undertook the effort to modify the compiler. [30]

We constructed coercion functions with which we could easily simulate a range of coercion costs. We then simulated a network consisting of one, two, four, and eight different processor types using eight 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. 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, and coercion function 2 does the coercion function 1 operations twice.

We measured the wall clock times of the solver for a 512x512 problem with four workers. The results are shown in Table-1. Speed-up is relative to an equivalent C program[4]. The results clearly show that,

Table 1: Effect of Coercion Costs

Number of pseudo-architectures	Time with coercion function 1 (msec)	Time with coercion function 2 (msec)	Speed-up 1	Speed-up 2
1	28228	28228	2.97	2.97
2	28679	28792	2.92	2.91
4	29073	29612	2.68	2.63

while there is a negative performance impact, it is not significant.

3.1.2 Scheduling

Functional parallelism results when one can execute different functions on either the same or different data. The parallelism derives from the multiplicity of functions. The classic example of functional parallelism is a pipeline, where each stage of the pipe is performing a different function in parallel with the other stages. In data parallelism, the same operation, either an instruction or a function, is performed in parallel on a set of distinct data items. The parallelism derives from the multiplicity of data items.

Effective scheduling of data parallel and functional program components is critical to the high-performance objective of the metasystem. The general scheduling problem consists of two parts, problem partitioning or decomposition, and placement. Problem partitioning is driven by two criteria: granularity considerations and load balance. The granularity of each scheduled program component, whether data parallel or functional, 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-machine processor, assuming such a processor is available. A data parallel nearest-neighbor 2D grid component might best be placed on a mesh.

It is clear that the general scheduling problem in metasystems is a hierarchical one. At the highest level, we may schedule a program component on a multiprocessor such as the Delta, but within the domain of the multiprocessor, the individual processors must also be scheduled. And so on. The general multi-level scheduling problem is quite difficult.

The Mentat scheduler [15] 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 functional parallelism very well, although it does not consider object-processor affinity during placement[5]. In the case of data parallel objects, the programmer is responsible for problem partitioning and decomposition. The scheduler places the sub-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 IPC's and two Silicon Graphics Iris (SGI) workstations. The SGI and Sun IPC use the

4. 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.

5. We will not discuss scheduling functional parallelism because FALCON manages it well. A complete discussion is beyond the scope of this paper.

same data format, so no explicit coercion was required. The SGI's are almost twice as fast (1.84 times, floating point) as the Sun IPC. We conducted two experiments. In the first we partitioned the work equally among four workers, with one worker on each of the four processors. In the second we created six workers each with equal work, and scheduled two workers on each of the two SGI's. The results are shown in Table 2 for a problem of size 1024x1024. The effect of the four worker schedule is that the two SGI's are not

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

Number of Workers	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

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 "stronger" processors, is that overhead is linear with the number of objects.

This is a serious problem. In order to get a good 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. However, that will lead to increased overhead and poor performance.

What is needed is a mechanism that will partition the problem into appropriate-sized pieces and distribute those pieces to the workers on different processors. If all problems were regular two dimensional array problems 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 interesting problems have a regular two dimensional array format. The mechanism must be extensible in order to support problem specific decompositions and must automatically map the partitions 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.

3.1.3 Partitioning Data Parallel Components

We have identified three stages in partitioning data parallel components, load selection, data/problem decomposition, and data distribution. The stages are shown in Figure 5. In load selection we compute how many workers to use to solve the problem and what percentage of work should be allocated to each worker (processor). The load selection phase is driven by granularity and load balance considerations, given information about the computation and communication complexity of the application component. Load selection can be handled automatically by the run-time system. This is a major improvement over many existing systems (including Mentat) in which both of these are done by hand.

Once a set of processors and a load distribution have been established by the load selection phase, the problem is decomposed into work units of a specified size and distributed to the processors. This process is complicated by the fact that not all problems decompose the same way.

We have solved each of these three problems by extending the Mentat philosophy of exploiting the strengths of both humans and computers. The problem is that it is difficult for the compiler to determine the computation granularity and communication complexity for arbitrary problems. It is also difficult for the compiler to determine how to partition and distribute arbitrary data structures. The programmer, on the other hand, knows the application and can fairly easily determine the computation required for a given problem size. Further, the programmer can most easily specify the data partition and distribution that is appropriate. What the programmer cannot do is make good granularity and placement decisions without a great deal of system state information. The run-time system can, however, quickly and accurately determine the number and type of processors to use if it can acquire granularity information.

Our solution involves the use of programmer-written call-backs 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 call-backs 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.

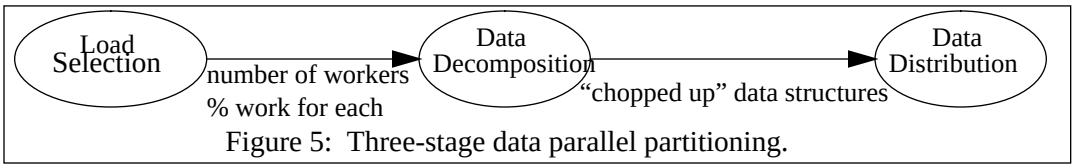

Figure 5: Three-stage data parallel partitioning.

3.2 Load Selection via Call-backs

Load selection is the first 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 testbed run-time system (RTS). 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 multiprogrammed (time-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 without rebalancing during the computation. Second, we assume the simplest heterogeneous network, a single-layer network of single-CPU machines of differing computation power (Figure 6). 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 the message cost equation of the underlying mes-

sage delivery system for the IN, T(n bytes) = T_s + nT(1), is known to the RTS. Fourth, the number of distinct

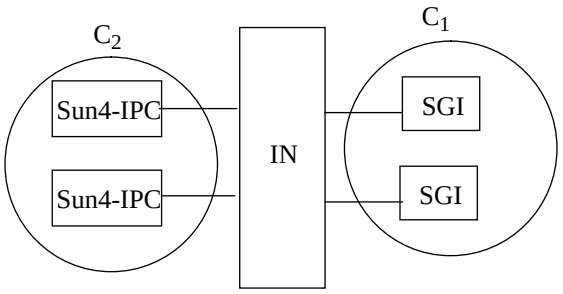

Figure 6: Simple 1-layer heterogeneous network.

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 nodes of class C_i.

We require that the units of T(n) and S_i be the same, e.g. units of time such as msec. The derivation of a suitable granularity decision function for problem partitioning based on both T(n) and S_i is straightforward.

Fifth, since we are looking at data parallel components, we assume the application 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 2D 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*. The data is decomposed into multiples of the minimal work unit. We refer to this data region as a *work region*.

3.2.2 Granularity Call-backs

In order to determine the grain size, and hence the number of workers, the application provides several call-back functions that describe the computation and communication structure of the problem:

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

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

`ok(num_nodes)` -- returns 1 if selected decomposition is logical given the problem

For example, for a 5-point stencil problem with row distribution and N columns/row, these functions are:

`f(r)` = r*10*N

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

These functions are used in a heuristic decision process. The output of the decision process is a `partition_vector`, `w_p[]`, of partition records. The number of workers selected determines the length of the vector. Each partition record contains the name of a worker, `w_p[i].object`, and the number of minimal work units assigned to that worker, `w_p[i].partition`. The decision process is described below.

Those readers not interested in the details may skip to Section 3.2.4, the run-time model.

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. 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 nodes, e.g. for a 2D row decomposition, P = # of rows
- m = number of architecture classes
- M_i = number of nodes of class C_i that will be applied to the problem
- n_i = number of available nodes of class C_i
- A_i = size of the data-region in multiples of p assigned to each node in C_i

There are two parts to the load selection problem: 1) determining the M_i for each architecture class C_i, i.e., into how many pieces (workers) to decompose the problem, and 2) determining the A_i, the sizes of the work regions that will be assigned to each of the workers. For a balanced load decomposition (our goal), we will assign the same size work region to each node 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 [30]. Of particular interest to us is that the algorithms are fast. The general form of the load selection algorithms is described shortly. The algorithms compute the M_i for each architecture class. Once M_i is determined, computing A_i for each architecture class is simple due to our load balance objective. A_i is a function of the processor power, W_i, the number of nodes used, M_i, and the computation complexity, `f()`. The general form for A_i is given below (M and W are the vectors of all M_i and W_i respectively):

$$A_i = P \cdot h(W, f, M), where \sum_{j=1}^{m} A_j \cdot M_j = P$$

The function `h()` has the following two properties: a larger W_i (i.e., a more powerful node) induces a larger work assignment, and the sum over all work assignments must equal P. How much larger the work assignment is will depend on how much work is done in a minimal work unit, p. Recall that this is captured by `f()`. 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. If `f(p)` is $O(p^2)$, then A_2 will be sqrt(2) times A_1.

In order to compute M_i, we assume that the RTS knows the number of "free" nodes n_i for each architecture class, where "free" is determined by a threshold policy parameterized by architecture class. Clearly, M_i must lie between 0 and n_i. The RTS will acquire information about the state of nodes 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 granularity heuristic function G. Below we present a simple granularity decision function based on `f(r)` and `g(r)`:

$$G(A_i) = \frac{S_i \cdot f(A_i)}{T(g(A_i))}$$

If G (A_i) > granularity threshold (usually 1), then we say the work assignment A_i is *feasible* for nodes M_i. In a feasible work assignment, the time spent doing useful computation exceeds the time spent in communication during the computation of A_i, i.e., it is profitable for a node of class C_i to compute on a work region of size A_i. However, a better load balanced distribution may exist if the granularity is too coarse. 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. The granularity function does not include communication overlapped with computation and the resulting decisions are therefore conservative[6]. At worst, the chosen grain size will be somewhat high.

The load selection algorithm is shown in Figure 7. The algorithm is greedy in the sense that it first considers nodes 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 nodes of each class may be applied to the problem. Since M_i implies A_i, a feasible assignment can be determined by choosing M_i such that the resulting A_i satisfies the granularity constraint for architecture class C_i. The number of nodes is determined by a procedure `find_grain`() called for each architecture class. `Find_grain` determines the maximum number of nodes that results in a feasible assignment. We have developed a heuristic for `find_grain`() based on binary-search [30]. If the granularity is too large (determined by `find_grain`()), the algorithm continues to the next strongest class to apply more nodes to the problem, and so on. If not, the algorithm terminates with the nodes previously computed. Only these nodes will be applied to the problem. Once the M_i are known, the A_i are determined.

```
begin
for i = 0 .. m
    M_i = 0;
    g = find_grain (i); // returns best M_i of C_i
    switch (g)
          case GRAIN_TOO_LARGE: M_i = n_i; break;

          // These two cases cause termination
          case GRAIN_TOO_SMALL : M_i = 1; exit_for;
          default : M_i = g; exit_for;
    end switch;
end for;
end;
```

Figure 7: General form of the load selection alogorithm. The architecture classes C_0 ... C_m are ordered such that $W_i >= W_{i+1}$ for i = 0...m-1.

6. Overlapped computation and communication is supported in Mentat.

If there are m architecture classes and N available nodes across all classes, and `f()` and `g()` have constant execution times, 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 the best grain for C_i which is at the heart of *find_grain*. Since m is typically small, and N is likely to be less than 100, this algorithm incurs very little run-time overhead. On a Sparc-2 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 assignment for the stencil problem was less than 1 msec! Given that it takes over 2 msec to send a message on a Sparc-2, we feel this is justifiable overhead for the size problems we are investigating.

3.2.4 Run-time Model

The load selection algorithm determines the number of nodes selected and the associated problem sizes A_i. How is this information used? In a *master-worker* model, the worker objects map one-to-one to the nodes selected (M_i), and are responsible for computing their portion of the problem, determined by A_i. The worker objects are managed by the master object, which hides the underlying problem distribution from the programmer, and provides an interface that allows the worker collection to be treated as a unit.

In Figure 8, we show the run-time model. The master interacts with the RTS via `load_select()`, which computes M_i and A_i for all classes C_i, creates a worker object on each chosen processor, and passes back to the master the names of the worker objects and the size of their problem portion. These are passed in the `w_p[] partition_vector`. For each node j selected, `w_p[j].object` is the name of the worker located on processor j, and `w_p[j].partition` is the associated A_i in minimal work units.

Once the RTS has created the worker objects via `load_select()`, the master passes the workers their problem portion determined from w_p[i] via *init*(). What the nature of the "portion" is depends on the problem at hand. For instance, it may be a physical data-region, or indices into files that contain the data.

Once the master has initialized the workers, the programmer may initiate an action via the master. The master then engages the workers to begin the specified operation and collects return results if there are any. At this point the user may request the results from the master. We are working on the high-level language interface and mechanisms that support this model. Since it is largely orthogonal to the issues of heterogeneity, we do not discuss it further.

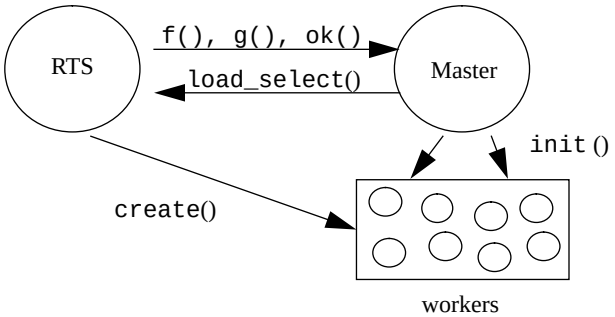

Figure 8: Master-RTS interaction. The master invokes `load_select()`, which in turn calls `f()`, g(), `ok()`, and then creates the workers, passing the resultant `w_p` vector back to that master.

3.3 Data Decomposition and Distribution via Call-backs

The second phase of our solution to the partitioning problem in a heterogeneous environment involves extracting the work regions for the individual workers, and invoking each with their work region so that the computation may begin. The difficult aspect of this phase is to divide the data to match the specification of work regions contained in the `w_p` partition vector, and then to communicate the data to the worker. In general, the characteristics of the application's communication topology must be reflected in the decomposition. We intend to address this problem in future work and to move the responsibilities of worker invocation and data distribution, worker interaction, and result collection into the domain of the system. As an intermediate solution, we are developing a suite of common data structures whose communications requirements are well known (e.g. 2D stencil problems, 2D arrays, trees), and providing these structures as templates to the programmer. For this discussion we will restrict our model to handle the initial invocation and distribution of work for a master/worker model. We begin by presenting the mechanism of our solution to the data decomposition and distribution problem. We then continue the Gaussian elimination example.

Two steps are involved in the initialization of the workers. First, the data must be "carved" to match the granularity specification for each worker, and second, each portion of data must be communicated to the proper worker. In order to achieve this, we require the programmer to define two member functions in the master class, `distribute`() and `carve`(), and an `init()` function for the worker class.

The programmer first invokes the `load_select()` function in the `master_main`() (Figure 9). `Load_select()` returns the `w_p` vector that contains the names of the workers, and the number of minimal work units allocated to each (Figure 9, line 18). Next, the programmer invokes the `distribute`() function with `w_p` as an argument (Figure 9, line 19). For each worker *j*, the work assignments given by `w_p[j].partition` are extracted and marshalled into arguments using the carve functions (Figure 9, lines 6-9). Once the arguments have been extracted, they are passed along with other necessary parameters to the `init`() function invoked on the name contained in `w_p[j].object`. The interior of the distribute function, and the calling sequence for performing the invocation and data distribution are also shown.

A work region may contain portions of multiple data structures. This is accommodated with 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 multiple of these units to be extracted and marshalled. The net effect is that each worker receives the data that defines its work region.

A few notes about this process are needed. First, the actual names of the distribute and carve functions are not important since they will be used by the programmer to specify the particular method of extraction from the data structures, i.e. by row or by column. Second, the actual extraction of the data can be in one of two forms. The actual data can be marshalled into the arguments of the `init`() 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 at each worker.

In summary, the programmer's involvement in the data extraction and communication process consists of the specification of a `distribute`() function, a `carve`() function for each data structure that must be

```
1   distribute (partition_vector *w_p) {
2       int i;
3       void arg1, arg2, ..., argn;
4           for (i = 0; i < w_p.count(); i++) { // for each worker
5           // “carve” the correct portion of each data structure
6           arg1 = carve(w_p[i].k, Data_structure_1);
7           arg2 = carve(w_p[i].k, Data_structure_2);
8           ...
9           argn = carve(w_p[i].k, Data_structure_N);
10          // Call the init() function for each worker, passing “carved”arguments,
11          // filter and i are additional, non-carved arguments.
12          (w_p[i].object).init(arg1, arg2, ..., argn,filter,i);
13      }
14  }
15
16  master_main() {
17      ...
18      w_p = load_select(this);
19      distribute(w_p);
20      ...
21  }
```

Figure 9: General example of the distribute function and calling sequence for data extraction and distribution.

partitioned over the workers, an `init()` function for the workers, and the proper calling of the `load_select`() and `distribute`() functions. The system takes care of the actual determination of how much work will be assigned to each processor and which processors will be used. We next return to the example of Gaussian elimination.

3.3.1 Partitioning Gaussian Elimination

The class definitions are shown in Figure 10. We declare three classes, `DD_floatarray`, `gauss_master`, and `gauss_worker`. The `DD_floatarray` class represents an in-core two-dimensional array of floating point numbers. It has a member function `carve_by_rows`() that takes as parameters the worker number and `w_p`. It returns a `DD_floatarray` with the appropriate rows. The class `gauss_master` is the master, and has two member functions, `solve`() and `distribute`(). The function `solve`() takes two `DD_floatarray` arguments, `A`, an NxN matrix, and `b`, an Nx1 matrix. Only the member function `init`() is shown.

Solve() begins by executing `load_select`(). The returned `w_p` vector contains the names of the workers that have been created by the system. The user is unaware of where the workers are located. The `distribute`() function then uses the information in the `w_p` vector to `carve_by_rows`() from both the matrix and the vector for each worker. The number of rows that will be extracted for each worker is contained in `w_p[j].partition`. Because the sub-unit of work for the matrix data structure is one row, the number of rows extracted for any worker *j* is (1 * `w_p[j].partition`). The same is true for the *b* vector. These objects are then sent as the arguments to the `init`() function of each worker. At this point the data has been distributed and we are ready to solve the system of equations.

The mechanism presented in this section solves three of the more difficult problems presented to the programmer in a heterogeneous system. First, the system decides the actual number of workers which will be

```
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).init(arg1, arg2);
25:       }
26: } /* end distribute */
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: } /* end main */
```

Figure 22: Gauss class definition including distribute, carve, and initialization functions.

applied to the problem. This decision is based on the current system state. In traditional homogeneous parallel processing systems, and in current state-of-the-art heterogeneous message passing systems such as PVM, this decision is made during implementation or at run-time, and represents at best a “good guess” by the programmer. By moving this responsibility into the system, we have removed from the programmer the responsibility for determining in advance which machines to use and knowing the relative powers of all machines in the system. Second, the determination of which processors to use based on their capability is made by the system as well. This decision is made in concert with a determination of the proper workload for each node in the selected set of processors. The programmer making this decision on his own would have to be aware of all the varying capabilities of the machines at his disposal. Typically, this granularity decision can be quite hard to make in a heterogeneous environment because it must be done for many different machines. 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 conveys the method needed to extract one work unit, and the distribute function handles the actual extraction and distribution of the proper number of work units for each worker. While these functions may appear complicated, template classes can be constructed for the standard data structures that can be easily extended via inheritance.

3.3.2 Limitations

Our decomposition strategy is limited to static decompositions (though not necessarily regular) where the amount of computation performed on a minimal work unit is known and is not data-dependent. In the case where the amount of computation is data-dependent, this strategy will not be not always perform well; some form of dynamic rebalancing may need to be done as in DataParallel C [23].

3.4 Effect of Partitioning

We turn again to the Gaussian elimination example. To test the efficacy of our approach, we have applied the problem partitioning metric by hand to the problem and scheduled the resulting workers with their computed problem sizes using Mentat. We then compared this with the even distribution shown earlier. For a row decomposition, the work assignments refer to the number of rows in each worker's work region. The problem size is P =1024 for a 1024x1024 problem. For this experiment, 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, W_1= 2 and W_2 = 1. The testbed uses a portable communication library, MMPS [17] based on sockets. The message cost equation for MMPS is:

$T(n) = 2.65 + 0.0015n$.

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 that load balance is achieved when the SGI's are given about twice as much work (ratio of W_1/W_2). The results shown in Table 3 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

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

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

around 3 with respect to the SGI sequential time. This makes sense because two IPC's are approximately equal in 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 having the scheduler place them based on processor power is not scalable. Communication in Mentat is usually linear in the number of workers. To achieve good load balance on a large system would require many more workers than processors[7], increasing 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 [1,2,4-6,12,20,21,24,25,29,31]. 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[28], Linda[7], the Argonne P4 macros[1], and Fortran-D [11]. 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 [1,7,28] require the programmer to operate at the assembly language level of parallelism. This makes writing parallel programs more difficult. Others [3,11] share our philosophy of providing a higher level language interface in order to simplify applications development. Our work differs from Fortran D [11] 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 functional 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 [3,9,23,28]. Once again, the level of service distinguishes our work from systems such as PVM [28]; they enable, rather that support, heterogeneous parallel computing. Our work differs from that of [9,23] in several ways. First, as with Fortran D, they are data parallel only, and support only regular low-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 [9]. 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 [23], 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 reach 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 call-backs 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 arbitrary data structures with arbitrary computational complexities, not just regular problems with low-dimension arrays.

Further work in this project falls into three areas, extending the platforms to larger, more diverse systems, developing scheduling models and heuristics for multi-level networks, 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 Applications 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," *Commun. 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, 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.

[8] T. L. Casavant, and J. G. Kuhl, "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems," *IEEE Transactions on Software Engineering*, vol. 14, pp. 141-154, February, 1988.

[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] D.L. Eager, E. D. Lazowska, and J. Zahorjan, 'Adaptive Load Sharing in Homogeneous Distributed Systems', *IEEE Transactions on Software Engineering*, vol. 12, pp. 662-675, May, 1986.

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

[12] 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.

[13] A. S. Grimshaw, "Easy to Use Object-Oriented Parallel Programming with Mentat," to appear in *IEEE Computer*, May, 1993.

[14] A. S. Grimshaw, E. Loyot Jr., and J. Weissman, "Mentat Programming Language (MPL) Reference Manual," University of Virginia, Computer Science TR 91-32, 1991.

[15] 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.

[16] A. S. Grimshaw, "The Mentat Run-Time System: Support for Medium Grain Parallel Computation," *Proceedings of the Fifth Distributed Memory Computing Conference*, pp. 1064-1073, Charleston, SC., April 9-12, 1990.

[17] 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.

[18] A.S. Grimshaw, E. A. West, and W. Pearson, "No Pain and Gain! - Experiences with Mentat on Biological Application," *Proceedings of the First Symposium on High-Performance Distributed Computing*, pp. 57-66, Syracuse, NY, Sept., 1992.

[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 Language*s, pp. 225-235, 1985.

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

[21] 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.

[22] P. Narayan, et al., "Portability and Performance: Mentat Applications on Diverse Architectures," TR-92-22, Department of Computer Science, University of Virginia, July, 1992.

[23] 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.

[24] 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.

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

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

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

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

[29] 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.

[30] J. Weissman, E. A. West, and A. S. Grimshaw, "Automatic Problem Decomposition and Scheduling in Heterogeneous Parallel Processing Systems," University of Virginia, Computer Science TR 92-33, in progress, 1992.

[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