Integrating a Parallel Computer and a Heterogeneous Workstation Cluster into a Metacomputer System

A Report Submitted as a Master of Science Research Project
by Tony Neff

December 15, 1995

Abstract

Two types of parallel computers commonly used for solving large scientific problems are clusters of workstations and distributed-memory multicomputers. Each system has strengths and weaknesses for this task. Workstation clusters have a high performance to cost ratio and the advantage of the latest processors. Workstations are commonly under-utilized and can provide an inexpensive source of CPU cycles. However, clusters of workstations cannot compete with the performance of a dedicated supercomputer.

This research proposes that creating a metacomputer combining different types of parallel computers can provide some of the advantages of each separate system. Specifically, I have integrated a distributed-memory parallel computer (the MEIKO CS-2) with a heterogeneous cluster of workstations. The integrated system uses the CHARM parallel-programming environment to provide for machine-independence and ease of programming in this heterogeneous environment.

The availability of processing capacity limits the size and complexity of the types of problems that can be efficiently solved. By creating a metacomputer the amount of processing capacity can be increased at relatively low costs. The low cost of the system and the fact that it is easily reconfigurable make it a good choice for solving large-scale Grand Challenge type scientific problems.

Contents

Chapter 1

INTRODUCTION

1.1 Objectives

This research had three main objectives.

1. The porting of the CHARM parallel-programming environment to the MEIKO CS-2 parallel computer
2. Combining CHARM systems for the MEIKO and heterogeneous workstation clusters to create metacomputer CHARM
3. Developing applications to test the performance of the new systems

The CHARM programming environment provides a high-level architecture-independent environment for general-purpose parallel-programming. CHARM has been implemented on a variety of parallel platforms including shared-memory and distributed-memory systems. The MEIKO CS-2 is a distributed-memory parallel computer using standard Sparc workstations connected over a specialized communications network. The first phase of this research adapted an existing workstation version of CHARM for operations on the MEIKO CS-2.

Viewing the MEIKO CS-2 as a collection of workstations provided a clear next step for combining the two versions of CHARM. Integrating the two systems required some significant modifications to support a system with multiple types of communication and different startup requirements.

Testing the new CHARM systems required the development of new applications. The new applications were designed specifically to test differing amounts of communication and to optimize performance on the heterogeneous metacomputer. As the applications were developed, additional modifications for the system were explored and implemented.

1.2 Related Works

In researching this paper, I found another system that combines the operation of a supercomputer with workstations, the Mentat metasystem developed at the University of Virginia. Not specifically metacomputers, several other systems provide for the use of networks of heterogeneous computers. The Message Passing Interface provides a standard for global communications and

portable parallel applications. There are also a number of different parallel computing systems for clusters of heterogeneous workstations, such as Paralex and Dome.

1.2.1 Mentat

Mentat is an object-oriented parallel-programming system developed at the University of Virginia [1]. It provides high performance across a wide variety of programming platforms. The portability of the system is ensured by using a classic layered virtual machine model. The use of the virtual machine model simplifies the construction of a metasystem by always providing the user with the same interface and by hiding details of object location. Mentat has two main components, the Mentat run-time system (RTS) and the Mentat Programming Language (MPL). The RTS controls all communication, knows the types of machines being used, knows the class of data being transported and is responsible for scheduling. In MPL, parallelism is exploited through the use of special C++ object classes, called mentat classes. The object classes parallelized are selected by the user from the object classes with sufficient complexity to exploit parallelism. Data and control dependencies are detected by the construction of dataflow graphs at run-time.

1.2.2 Message Passing Interface(MPI)

The message passing interface (MPI) is a standard for writing portable message-passing parallel programs [2]. In addition to simple point-to-point communications, MPI provides machine-independent abstractions for various global communication operations, such as broadcast, scatter and gather. The system uses multiple logical layers of software to provide for portability and machine-independence. MPI constructs can be used directly or incorporated into parallel-programming systems.

The heterogeneous workstation implementation of MPI uses the User-level Reliable Transport Protocol (URTP) to ensure message arrival over Ethernet or other unreliable broadcast transport media. URTP is implemented using a sliding window protocol and uses a pessimistic immediate request protocol for message retries. Because of the need for efficiency in handling multicast messages for the global communications operations, much of the special processing needed is done in the URTP software layer.

1.2.3 Heterogeneous Workstation Clusters

A number of computing systems provide for a system of heterogeneous workstations in addition to CHARM. While restricted to workstations there are many innovative systems with applications to a metasystem. The Dome parallel-programming environment was developed at Carnegie Mellon University [3]. Dome is primarily a run-time system built on top of PVM. The main issues addressed by Dome are load balancing, fault tolerance and ease of programming. Paralex is a parallel-programming environment developed at Cormell University and the University of Bologna [4]. Like Mentat, Paralex uses a data-flow model, but in Paralex programs are developed using a graphical editor to specify the dependencies.

1.3 Overview

The next chapter will discuss the definitions and requirements for creating a metasystem, a virtual parallel computer made up of different types of computers that are networked together. Chapter three deals with the CHARM parallel-programming environment and how it was adapted to the MEIKO CS-2 and the heterogeneous metasystem. Chapter four discusses the various applications that were used to test the new CHARM systems and specific changes that were made to the applications and the system to improve performance. Chapter five discusses the techniques used to evaluate system performance. The discussion includes the results of the application testing, information about the standard latency applications, and the use of the load balancing window (a graphical performance measuring tool). Chapter six summarizes the results obtained from the applications, discusses the significance of the results, and areas for future research.

Chapter 2

METACOMPUTERS and METASYSTEMS

2.1 What is a Metasystem?

A metasystem is a distributed computing system composed of a heterogeneous group of autonomous computers linked together by a network [1]. Computers in different physical locations can be connected over a network to create a metasystem with great potential for large-scale scientific computing. Moreover, the metasystem is more than just a number of computers connected together. The metasystem is a transparent machine independent programming environment that insulates the programmer from the complexities of machine type, processor type and data representation details. The entire system, however configured, appears to the user as a single parallel multicomputer. Independent of the current configuration, the system provides a standardized interface to data processing and storage services.

There are many difficulties in creating software for a metacomputer. The metasystem involves the creation of a unified software system to deal with the complexities of scheduling, communication, synchronization, decomposition, data conversion, data distribution and differing hardware resources.

2.2 Metacomputer Charm

The CHARM programming environment was selected for implementation of the metasystem for a number of different reasons. First, the system was explicitly designed for machine independence without the loss of efficiency. Originally this was done to provide a single common programming environment usable on many different platforms, but using the same abstractions and management tools the system can be extended. This extension provides additional processing resources for the new virtual multicomputer. Second, the CHARM system can reduce the complexities of parallel-programming by automatically decomposing computation into smaller parallel computation. The CHARM system also decides how and when to schedule specific actions. This removes the need for the programmer to decide how to assign data and computation to specific locations. The language provides for a large variety of application domains, and by the use of a rich set of primitives, is applicable to a large number of target machine architectures. CHARM is discussed in more detail in chapter 3.

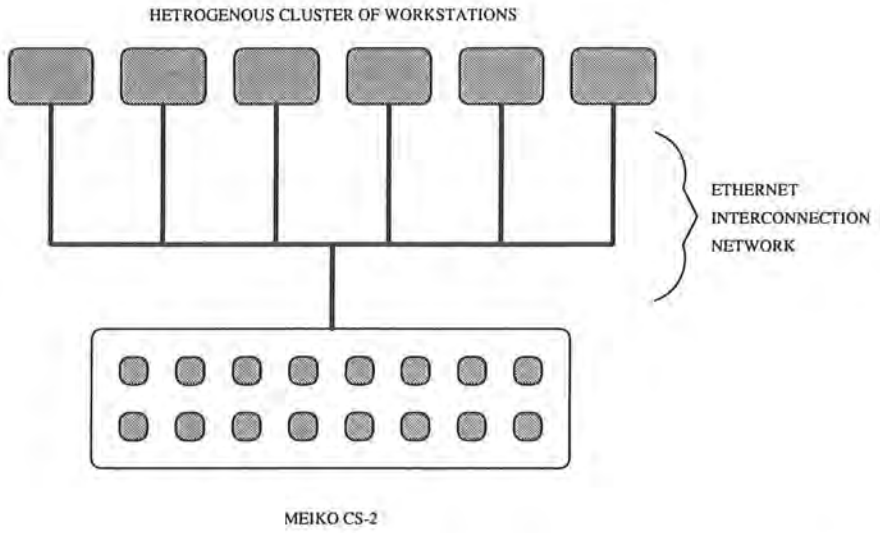


Figure 2.1: The Metacomputer Used in this Research

2.3 The MetaComputer

The Metacomputer developed in this research consists of a cluster of Sun and HP workstations combined with the MEIKO CS-2 distributed-memory parallel computer (see 3.3.1). The computers are connected via UDP over Ethernet. Because each MEIKO node has its own Ethernet interface all communications can be done directly processor to processor. The system provides a uniform interface independent of the machines being used. The user needs only to specify the specific workstations and the number of MEIKO nodes to be used for each configuration. Due to the requirements of the MEIKO resource manager, if any MEIKO nodes are to be used in the configuration then the launching host node must also be a MEIKO node. Typically this is a node on the login partition or the scalar `shark16` node is used.

Chapter 3

CHARM

3.1 Parallel Programming with CHARM

The CHARM system was developed at the University of Illinois at Urbana-Champaign [5]. The system is designed as runtime support for an explicitly-parallel language. Intrinsic to the design was that CHARM should be machine-independent, object-based, and message-driven. The system permits users to develop portable parallel applications without the loss of efficiency. The clear separation of responsibilities between programmer (problem decomposition and process creation) and programming environment (process location and load balancing) helps to control program complexity. The language supported is based on the message-driven paradigm. This means that from the users viewpoint messages are explicitly sent but not explicitly received. The system automatically receives the message and starts program execution based on information in the message. CHARM provides high-level mechanisms to facilitate the task of developing even highly-complex parallel applications.

The basic unit of work in CHARM is a coarse grained process called the *chare*[1]. A chare definition consists of the name of the chare, its local variables and a sequence of message entry points. A block of code within each entry point defines the coarse grained computation. All CHARM programs must have a *main* chare that executes on a single processor, and is a starting point for program execution. All other chares are created by programs statements.

```
CreateChare(processor,processor@Init,start_message);
```

This statement is used to create an instance of the chare `processor`. The call returns instantly and the create request is queued. Sometime later the request is processed and the chare is created. A variety of load balancing schemes can be used to determine on which processor to execute the chare. Execution of this chare begins at the `Init` entry point with the message referenced by `start_message`.

```
SendMsg(processor@receiveLMSG,lmsg, &Dest);
```

This statement is used to send the message `lmsg` to the entry `receivedLMSG` on the chare referenced by `Dest`. The sending chare doesn't need to know the destination processor as that information is contained in the `ChareIDType` variable `Dest`. The call is non-blocking and the

[1] from Old English, meaning chore or task

calling chare can continue to process data while the communication is taking place, permitting the overlap of communication and computation. There is no corresponding receive message call in the language. The message is received by the system at the destination and queued. Execution of the entry begins when the message is processed off the queue.

There are many addition CHARM constructs to aid in programming. The *Branch Office Chare* is a special type of chare that has one representative chare per processor. Several type of distributed data types are also supported: *readonly*, *accumulator*, *monotonic* and *write-once*.

3.2 Charm on a Cluster of Workstations

A cluster of workstations can be regarded as a collection of processors that do not share an address space, much like a distributed-memory parallel computer. The differences between a cluster of workstations and a distributed-memory parallel computer result from the use of UDP socket connections over Ethernet. Because of the need for reliable and efficient message passing, this version of CHARM implements message fragmentation for large messages and a sliding window protocol to reduce the number of acknowledge messages. Load balancing is a key issue on a heterogeneous cluster, and the CHARM system takes processor speed and network load into account when allocating chares to processors.

The CHARM environment reads a `nodes` file to determine which workstations to use. This file contains the machine name, relative processing speed and other information needed to create the UDP socket connection. Changing the configuration of the cluster is a trivial task.

3.3 Charm on the MEIKO CS-2 Parallel Computer

3.3.1 The MEIKO CS-2 Distributed-Memory Parallel Computer

The MEIKO CS-2 parallel supercomputer is a scalable distributed-memory parallel vector computer. The system consists of two main parts, processing elements and the CS-2 interconnection network.

Each processing element consists of a scalar processor, a vector processing element, and a communications processor. The processing element is a SuperSparc chip with a peak processing capacity of 40 MFLOPS. The vector unit consists of another scalar SPARC processor and two Fujitsu vector processors that share a 3-ported memory system with the other processors in the element. The vector processing element has a peak processing capacity of 200 MFLOPS. The communication processor provides a fast interface to the interconnection network.

The CS-2 communication network is a multi-stage packet-switched 4-ary fat-tree in which the bandwidth between stages remains constant. Messages are routed by full crossbar switches at each stage of the fat-tree. The MEIKO processors typically communicate among themselves using the Elan Widget library calls. Messages using Elan Widgets have very small latencies. The calls have been standardized in the Intel message passing library (MPL) `mpsclib`. Message passing latencies can be as low as 24 microseconds between processing elements [8].

3.3.2 Porting CHARM to the MEIKO CS-2

A CHARM program is source translated and compiled to generate a host program and a node program. The host program sends a request to the MEIKO resource manager, which grants the host exclusive access to the requested number of processors. Once the processing elements are available the node program is loaded on each processor and program execution begins.

The configuration used by a CHARM application is determined by a user specified `nodes` file containing the number of nodes to use and the pathname to the appropriate directory containing the application. Once the initialization is complete, the *main* chare begins execution. At the same time each processor enters a loop of picking a task or chare, created with calls to `CreateChare()` or `SendMsg()`, from its own queue and executing it. If communication takes place while executing a chare, the `SendMsg()` call is executed and the message is sent asynchronously. Because the call is non-blocking, it is possible for communication and computation to overlap increasing the efficiency of the system.

The original implementation of the MEIKO version of CHARM was created using the Intel MPL library `mpsclib`. For example, the standard CHARM `SendMsg()` is implemented as an MPL `isend()`, which in turn is implemented using the Elan Widget DMA port.

3.3.3 The Channel Version of CHARM

In an attempt to reduce message passing latency on the MEIKO an additional version of the system was created. This version uses Elan Widget channels as the primary form of communication. It was hoped that by using channels the message passing latency would be reduced as the latency for the Intel NX library has been reported at 78 microseconds verses 24 microseconds for channel communications [8].

Using channels for CHARM communication proved difficult. CHARM uses communication that is inherently asynchronous. Channels, while not strictly synchronous, are better suited to synchronous communication. To achieve asynchronous communication each channel is used for mono-directional communication only. The channel is initially prepared to receive a message with the `ew_chanRxStart()` call. Each channel also has an outstanding transmit for which the message has completed but the blocking `ew_chanTxWait()` has not been issued. Transmitting a message involves issuing the wait for the previous message then starting the current one. Receiving a message involves testing for receipt using `ew_chanRxDone()`. If done, a wait is issued, the message is processed and finally the system resets by issuing a start for the next message. This system produces asynchronous communication but will also block if a second message is sent on the same channel before the first has been completely received, this can potentially lead to deadlock. The implementation uses multiple channels between each pair of processors to reduce the chance of blocking. Another problem with this system is that it is possible to overwrite previously received data. To prevent this, the receiving processor uses an acknowledge message to indicate that the channels can be safely reused. If all channels are in use the system uses NX calls instead of channels. One other problem with the system is that there is no way to know the size of the incoming message until after it has been received. Ports have this ability and the workstation version uses message fragmentation to force all packets to the same size. The system uses a user set value to determine the maximum size of a channel message and allocates a buffer for all messages based on that size. Any message larger than this maximum value is sent via NX.

3.3.4 Vectorization

The MEIKO processing elements each incorporate two Fujitsu μVP vector processors. In order to make better use of the vector processors a number of modifications were made to the CHARM system. The compiler used for producing the vectorized code is the Portland Group C Compiler (pgcc). This compiler became the standard for the MEIKO CS-2 based CHARM system when I found that optimized non-vectorized code ran faster with pgcc than with the original compiler.

The automatic vectorization feature to the pgcc compiler failed to vectorize CHARM programs because of the data structure used to store variables local to a chare. Possible data dependencies prevent the vectorization. Loops to be vectorized need to be forced with the addition of a

```
#pragma nodepchk
```

statement immediately preceding the loop. Changes were made to the charm translator to incorporate pragma statements into the language.

The biggest changes that were made for vectorization were that changes made to the applications. The need for larger grains, removal of code designed to reduce cache misses, and reordering of nested loops is discussed in 4.2.4.

3.4 Implementation of MetaComp Charm

The major obstruction in the creation of MetaComp CHARM was the fact that multiple communications networks are used to connect the processors. It is fairly simple to determine which of the communication networks to use, but the redundancy greatly complicates the system. It was also necessary to make significant changes to accommodate the addition of MEIKO nodes to the heterogeneous network. The main problems dealt with the new operating system (*Solaris* was not previously supported by CHARM) and differences with the file structure on the MEIKO.

3.4.1 Initialization

The `nodes` file to be used is supplied by the user and contains the names of the workstations to be used. This was modified to include the MEIKO. The exact processors used by the metacomputer are assigned by the MEIKO's resource manager. Internal to the system the individual processors are recorded when the needed information is available. This permits UDP communication to individual MEIKO nodes.

3.4.2 Communications

The system uses UDP sockets to connect all processors in the metacomputer. The faster CS-2 network is used when possible, in communications between MEIKO nodes. Originally the system used the MPL communications library on the CS-2. MPL on the MEIKO is implemented using DMA ports. In an attempt to reduce message passing latency, these calls where later augmented with DMA channels. This greatly complicates certain system operations especially detecting and receiving messages. The system has to check the UDP connection and the port connection and all of the channel connections before being sure there are no outstanding messages.

3.4.3 Other Additions

While developing efficient implementations of algorithms for the metacomputer it became clear that a new way was needed to balance the load on the various computers. The standard method previously used, created many chares which were distributed to processors according to the speed and load of the processor. For many algorithms, it is necessary for all of the chares to communicate and the large number of messages required greatly slowed execution. The solution used was to reduce the number of chares and make the amount of work that each chare processes variable. This complicates the program but can greatly reduce the total number of messages required. A new load balancing function

```
CkDataPartition(size,max,base,fraction);
```

is called by the user and information on processor speed and load are used to determine the amount of data for each processor.

A few other constructs where also added to the translator to standardize the use of some non-charm functions, including `CkMemCopy()` replacing the use of `memcpy()` and `McTimer()` replacing calls to `RealTime()` a timer implemented with time-of-day system calls. Corrections were also made to the initialization routines for the load balancing window.

Chapter 4

APPLICATIONS

4.1 All-Pair Shortest Paths

The all-pairs shortest paths problem is to find the shortest path between all pairs of vertices $v_i, v_j \in V$ such that $i \neq j$ in a weighted graph $G(V, E, w)$.

4.1.1 Dijkstra's Algorithm

The parallel formulation of Dijkstra's all-pairs shortest paths distributes the V vertices of the graph among the processors. Each processor P_i then finds the shortest paths from vertex v_i to all other vertices employing Dijkstra's sequential single-source shortest paths algorithm.

In CHARM the adjacency matrix is replicated on all processors using the *readonly* construct. A *chare* is then created for each vertex $v_i \in V$. Note that this is the only applications discussed here that has no floating point operations.

4.2 Matrix Multiplication

The matrix multiplication problem discussed here, takes two $n \times n$ dense, square matrices A and B and yields the product matrix $C = A \times B$.

4.2.1 Conventional Algorithm

Each value of the resulting C matrix is computed using the standard formula

$$C_{i,j} = \sum_{k=0}^{k=n-1} A_{i,k} \times B_{k,j}.$$

The CHARM implementation uses a *readonly* variable to replicate the B matrix on all processors. A number of chares are created on a user specified grain-size input variable. This variable can be set to a fraction of a row per chare (fine grained) or several rows per chare (course grained). Each of these chares contains part of the A matrix. On execution each chare computes the rows of C corresponding to that chare's rows of A.

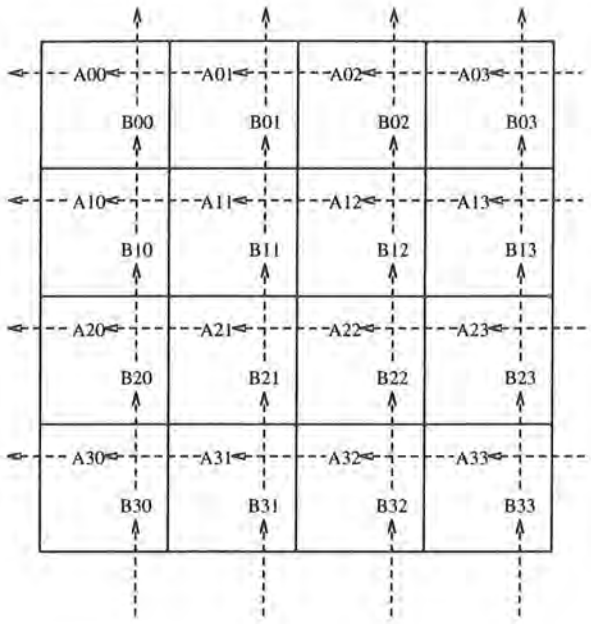


Figure 4.1: Cannon's Data Partitioning and Communication Patterns

4.2.2 Cannon's Algorithm

Cannon's algorithm partitions the A, B and C matrices into a number of blocks equal to p, the number of processors used, in a checkerboard fashion. Blocks of the A and B matrices are passed from processor to processor where each processor accumulates a partial dot product of the current A and B blocks. The number of iterations required is $\sqrt{p}$. The algorithm has the same time complexity as the conventional algorithm but is more memory-efficient requiring only $\frac{1}{p}$ times the memory on each processor ($\frac{2}{p}$ if overlapped communication is used).

The CHARM implementation creates a number of blocks based on user input rather than the number of processors used. This provides a mechanism for the user to control the grain of the program. In CHARM terminology each block is one chare. The experiment used 64 chares in a 8×8 grid. In order to reduce communication costs, communications and computation are overlapped. The two required blocks from A and B are copied to local memory, and the messages are sent on to the destination chares before computation begins. While overlapped communication works well for the low latency MEIKO CS-2 communications network, Ethernet is too slow to allow for a full overlap on the grain size used in this implementation. A balanced load is achieved by varying the number of chares on each processor.

4.2.3 Fox's Algorithm

Fox's algorithm partitions the A matrix by columns onto the processors. The B matrix is partitioned by rows into messages and each message is broadcast to the processors for computation of the dot products.

The CHARM implementation partitions the A matrix based on each node's processing capacity. The result being that a faster (or less loaded) processor would be responsible for a greater number

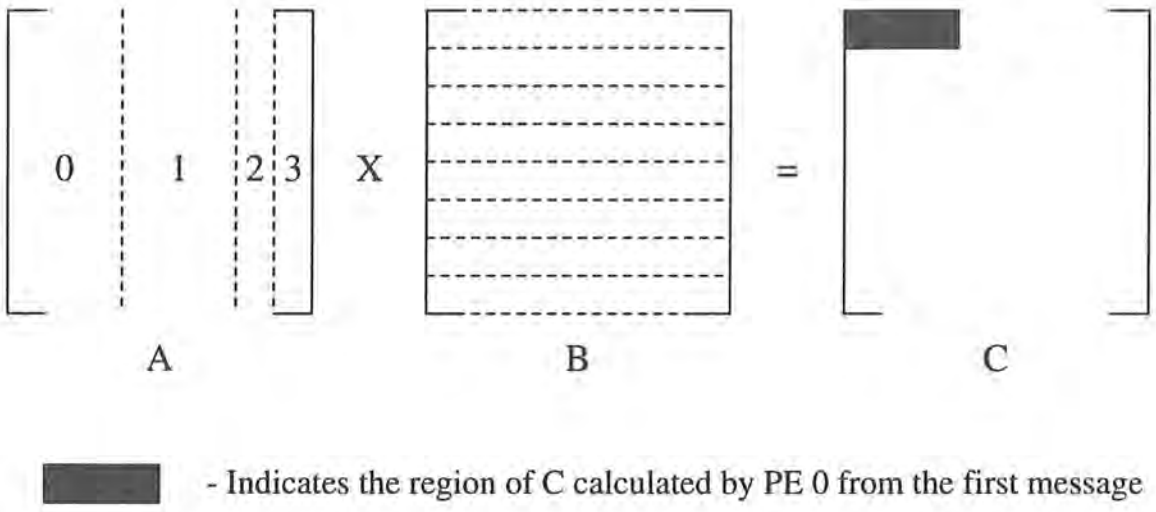


Figure 4.2: Fox's Data Partitioning

of columns. The B matrix is then partitioned by row. The number of rows sent per message can be set by the user and determines the grain size of the application. On completing computation for the message one of the processors then requests that another message be broadcast.

4.2.4 Vectorized Matrix Multiplication

Slight changes where made to implementations for Cannon's and the conventional algorithm to achieve better performance using the MEIKO CS-2 vector processors. Code designed to reduce cache misses in the non-vector application was removed. The code changed the order in which elements of the A and B matrix were accessed and would have unnecessarily complicated the vector code. The grain-size of the problem was increased. The non-vector version of Cannon's algorithm typically used a block size of 128×128 floats (64k bytes), where the vectorized version produces better results with larger blocks (256k bytes). In the non-vector version the size of a block was selected based on the time taken to calculate the partial dot-product for a block. The block size also effected the number of cache misses made during the calculation. Both of these factors had less of an effect in the vector version and grain size could be increased to the point needed for a balanced load. In order to reduce data dependencies the nested loops were reordered. Normally this would have been done by the vectorizing compiler, but because of CHARM'S data structures the compiler could not. Changing the order of the loops in the applications produced significant differences in runtime.

4.3 Systems of Linear Equations

This problem is to solve a system of n linear equations of the form

$$\begin{array}{ccccccccc} a_{0,0}x_0 & + & a_{0,0}x_1 & + & \cdots & + & a_{0,n-1}x_{n-1} & = & b_0, \\ a_{1,0}x_0 & + & a_{1,0}x_1 & + & \cdots & + & a_{1,n-1}x_{n-1} & = & b_1, \\ \vdots & & \vdots & & & & \vdots & & \vdots \\ a_{n-1,0}x_0 & + & a_{n-1,0}x_1 & + & \cdots & + & a_{n-1,n-1}x_{n-1} & = & b_{n-1}. \end{array}$$

usually written in matrix notation as $Ax = b$, where x is the solution vector.

4.3.1 Gaussian Elimination with Back Substitution

This application solves the equations in two steps. Gaussian Elimination with partial pivoting is used to reduce the A matrix to an upper triangular system. Once in upper-triangular form, back substitution is used to solve for x, the solution vector.

The CHARM implementation uses a modified branch office chare to create one chare on each processor. Due to the heterogeneous nature of the processors a variable number of rows of the matrix are assigned to each chare using the `CkDataPartition()` function (see 3.4.3). Rows are eliminated in place rather than swapping in order to maintain balance between the processors. After gaussian elimination is finished, the resulting matrix has to be sorted by row to become upper-triangular. Back substitution then proceeds normally, but some parallelism is lost as some processors finish the substitution before the other processors and broadcast results. As the sorting and backsubstitution are a relatively small fraction of the total computation the backsubstitution has a negligible effect on the total time.

4.3.2 Jacobi Iterative Method

Jacobi is an iterative method for estimating the solution to a system of linear equations. The solution of the i^{th} linear equation can be written as

$$x[i] = \frac{1}{A[i,i]}\left(b[i] - \sum_{i \neq j} A[i,j]x[j]\right)$$

using the solution vector $x[j]$ from the previous iteration to estimate the solution for the current iteration of a specific linear equation $x[i]$. All equations are solved in parallel and compared with the previous solution for that equation to decide if another iteration is needed. The solutions are then collected and made available to all processors for the next iteration.

The CHARM implementation work uses data partitioning very similar to that used with Gaussian elimination. Iterations continue until successive solutions change by no more than a user supplied ϵ for all equations.

4.4 Modifications Specific to Metacomputing

It became clear during the testing of typical CHARM programs, that some additional changes would have to be made in order to run efficiently on the metacomp system. A CHARM program

designed for a homogeneous parallel computer typically uses the creation of many chares to parallelize the program and set the grain size. Because of the different latencies associated with off processor communication and the large amounts of communication required for some of the applications, it is desirable to reduce the number of chares and increase the grain size. However, decrease the number of chares too much and load balancing becomes difficult. Consider the extream case of one chare per processor, if each chare has the same amount of work, differences in processor speed and load will greatly effect the amount of time to do that work. Several different schemes were used to overcome these problems.

The applications presented in this paper use several different kinds of data partitioning. The typical applications Matrix Multiply and Dijkstra's All-Pairs Shortest Paths use a rows based chare with replicated data on all processors. While efficient in terms of communication, they use a great deal of memory on each processor. Because of the large number of chares, This type of partitioning is only practical when little or no communication takes place between chares.

Gaussian elimination, Jacobi and Fox's algorithms all take the opposite approach reducing the number of chares to the minimum of one per processor. This helps in reducing the amount of communication at the expense of a slightly more complex program. Load balancing is achieved by varying the amount of data each chare has to process. This means that dynamic load balancing strategies cannot be used, but static load balancing schemes performed well during tests.

The block partitioned Cannon's algorithm makes use of chares in a different way. The number of chares to use is specified by the user not by the problem size or the number of processors used. The user has some control over the grain-size of the program (the number of blocks must be a square). The application is designed to overlap communications with computation by copying and sending data for the next iteration before beginning computation of the current iteration. This is especially evident when Cannon's algorithm is run on the MEIKO without workstation nodes (see 5.2 and 5.3). In this homogeneous environment the size of the blocks can be optimized in terms of hiding communication cost by overlapping. The block size also can be altered to reduce the effect of cache misses. The program also makes use of a two step chare creation scheme, creating a single chare on each processor then having that chare create a number of the actual data chares based a call to `CkDataPartition()`. This mechanism provides the load balancing that is needed for any heterogeneous computing system, while reducing the amount of off processor communication.

Chapter 5

APPLICATION EVALUATION

5.1 Measuring Performance

The performance results for the metacomputer were obtained by running the application programs under standardized conditions. The load on the workstations was kept low by running the applications after hours and careful load monitoring. The MEIKO resource manager ensures that the user has exclusive access to the MEIKO nodes. The times reported are real elapsed time, calculated by calls to `gettimeofday()`. All load balancing was done statically, taking into account the initial work load of each machine and that machine's relative processing speed. Relative processing speed for each application was determined by executing the application program sequentially on each type of processor used. Relative speed varies slightly from application to application depending on the amount of communication and floating point computation done, with major differences for the all integer Dijkstra's All-Pairs Shortest Paths and vectorized Matrix Multiplication.

All applications were run on a problems size of 1024×1024. All applications but APSP used randomly generated single-precision floating-point numbers. System initialization time is not included in the reported runtimes.

5.2 Meiko Performance

The entirely homogeneous MEIKO provides a simple means for measuring system performance (in contrast to the heterogeneous MetaComp). The only complication was in running vectorized code on 16 processors. The combination of a scalar host (shark 16) as host and vectorized nodes produced suspect results.

Table 5.1: Dijkstra's All-Pairs Shortest Path

Meiko Nodes	Run Time (seconds)	Speedup
1	398	1.00
2	200	2.00
4	100	3.99
8	50	7.96

Table 5.2: Cannon's Algorithm

	Scalar		Vector	
Meiko Nodes	Run Time (seconds)	Speedup	Run Time (seconds)	MFLOPS
1	130	1.00	26.6	81
2	65	1.98	14.8	145
4	33	3.95	8.4	256
8	17	7.74	5.7	375
16	8.5	15.20		

also see the load balancing window 5.5

Table 5.3: Conventional Algorithm for Matrix Multiplication

	Scalar		Vector	
Meiko Nodes	Run Time (seconds)	Speedup	Run Time (seconds)	MFLOPS
1	168	1.00	23.5	91
2	85	1.99	14.3	150
4	43	3.94	11.6	186
8	22	7.75	13.8	154
16	12	14.32		

Table 5.4: Fox's Algorithm for Matrix Multiplication

Meiko Nodes	Run Time (seconds)	Speedup
1	181	1.00
2	123	1.47
4	61	2.94
8	32	5.62

Table 5.5: Gaussian Elimination

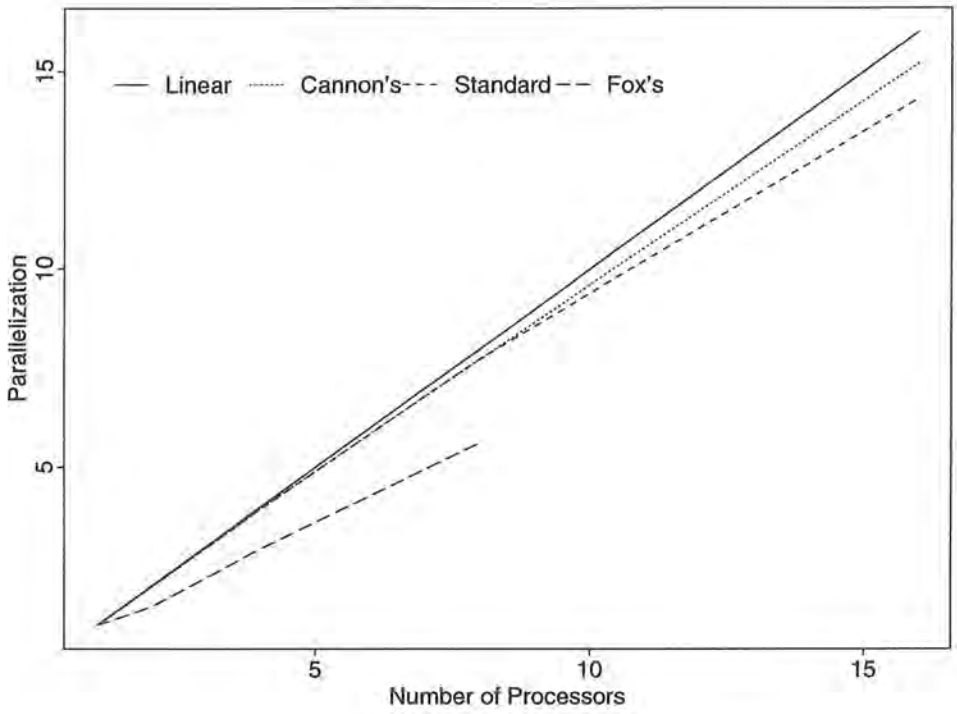
Meiko Nodes	Run Time (seconds)	Speedup
1	97	1.00
2	54	1.80
4	31	3.08
8	22	4.36
16	22	4.48

also see the load balancing window 5.7

Table 5.6: Jacobi Method of Solving Linear Equations

Meiko Nodes	Run Time (seconds)	Speedup
1	92	1.00
2	47	1.95
4	25	3.65
8	15	5.98
16	11	8.67

Matrix Multiply Applications



Gaussian Elimination, Jacobi, and APSP

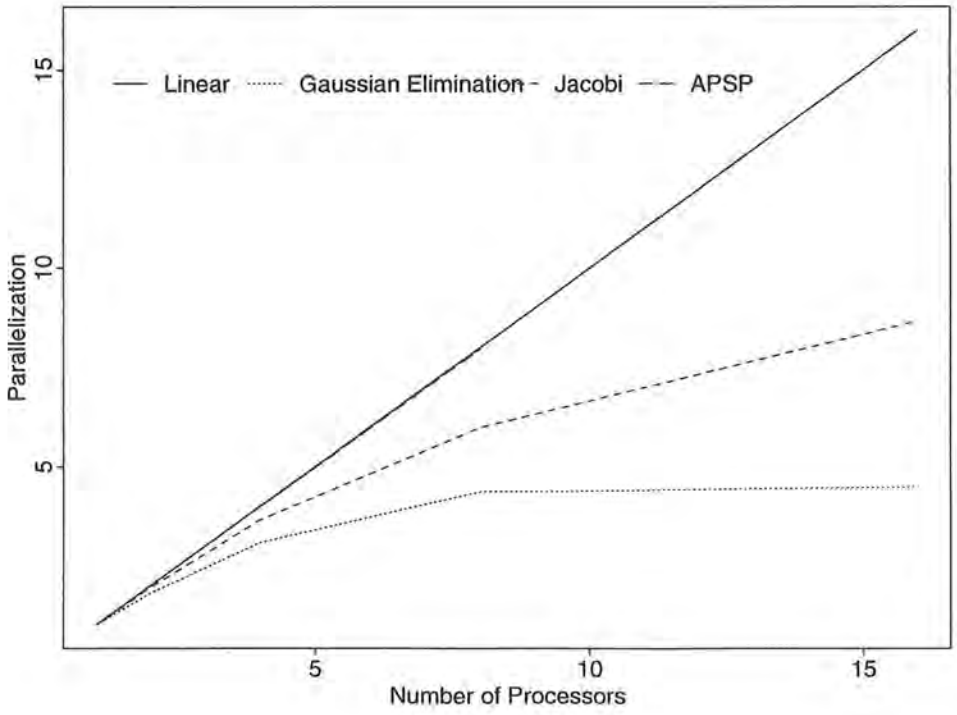


Figure 5.1: Parallelization Plot of Applications on the Meiko

5.3 Metasystem Performance

The measure of system performance on the metasystem is difficult due to the heterogeneous nature of the system. There are too many variable and no single standard sequential runtime for a comparison. The sequential runtimes for the applications were used to calculate the relative number of sequential MEIKO processors it would take to achieve the same runtime. This value for equivalent number of MEIKO processors is is summed for all the processors used in a particular configuration. The actual configurations of processors used in the reported data are listed below. The value of equivalent Meiko's can be misleading in applications where there is significant communication, for example one of the configurations has an equivalent number of 5.45 Meiko nodes but actually uses a total of 13 processors. The communication requirement for this application scales with the actual number of processors.

Table 5.7: Processor Configurations Used

Configuration	Meikos	Sparc20	Sparc10	Sparc5	HP730	HP715	Total
Meta 0	1						1
Meta 1	1					2	3
Meta 2	1	1					2
Meta 3	1	1				2	4
Meta 4	1		1	1	1	2	6
Meta 5	1	2	2	2	1	5	13
Meta 6	1	4	3	3	1	3	15
Meta 7	2	1		1		1	5

Table 5.8: Dijkstra's All-Pairs Shortest Path

Configuration	Equ. Meiko Nodes	Run Time (seconds)
Meta 0	1.00	398
Meta 1	2.10	199
Meta 2	2.10	192
Meta 3	3.20	142
Meta 4	5.15	84

also see the load balancing window 5.2

Table 5.9: Cannon's Algorithm

Configuration	Equ. Meiko Nodes	Run Time (seconds)
Meta 0	1.00	130
Meta 1	1.50	135
Meta 2	1.75	134
Meta 3	2.25	124
Meta 4	4.15	108
Meta 7	6.05	99
Meta 5	6.05	91
Meta 6	8.10	83

also see the load balancing window 5.3

Table 5.10: Conventional Algorithm for Matrix Multiplication

Configuration	Equ. Meiko Nodes	Run Time (seconds)
Meta 0	1.00	168
Meta 1	1.75	113
Meta 2	1.80	148
Meta 3	2.60	106
Meta 4	4.00	74
Meta 7	4.00	57
Meta 5	8.20	52
Meta 6	10.50	56

Table 5.11: Fox's Algorithm for Matrix Multiplication

Configuration	Equ. Meiko Nodes	Run Time (seconds)
Meta 0	1.00	181
Meta 1	1.35	180
Meta 2	1.55	128
Meta 3	1.90	124
Meta 4	3.00	139
Meta 7	3.15	99
Meta 5	5.45	162

also see the load balancing window 5.4

Table 5.12: Gaussian Elimination

Configuration	Equ. Meiko Nodes	Run Time (seconds)
Meta 0	1.00	97
Meta 1	2.10	95
Meta 2	2.30	85
Meta 3	3.40	79
Meta 7	4.00	80
Meta 4	5.25	95

also see the load balancing window 5.6

Table 5.13: Jacobi Method of Solving Linear Equations

Configuration	Equ. Meiko Nodes	Run Time (seconds)
Meta 0	1.00	92
Meta 1	1.90	65
Meta 2	2.00	54
Meta 3	2.90	67

5.4 Measuring Communication Performance

As it was developed, the metasystem used a variety of different communications. It became necessary to measure the performance of the various different types of communications. Four measures of communication speed were used: latency, scatter, gather, and an all to all broadcast. The load balancing window also provided some performance information and graphically demonstrates the result of using the various communication types.

5.4.1 Latency

The latency measured and reported in this report includes all overhead associated with the CHARM system. The times reported are in milliseconds and were calculated through calls to the charm function `CkUTimer()` implemented as `gettimeofday()` with a resolution of microseconds. The size of a message does not include the charm message overhead (80 bytes in this version). Latency is measured by repeatedly sending and receiving a message between two processors. The resulting average round trip time is then divided by two to get the average time for one send-receive pair. The measurement was made using both types of implemented Meiko to Meiko communications. NX refers to the use of the Intel NX library, and CH refers to the use of elan widget channels. The Sun used was a SparcStation 20, and the HP was a HP715 workstation. The reported one processor overhead was measured by running the application on a single Meiko node.

Table 5.14: Message Passing Latency

Message Size	Meiko to Meiko(NX)	Meiko to Meiko(CH)	Meiko to Sun	Meiko to HP
1	0.4	0.4	3.0	4.5
4	0.4	0.4	3.0	4.5
16	0.4	0.4	3.0	4.6
64	0.5	0.4	3.2	4.6
256	0.5	0.5	3.7	5.0
1024	0.5	0.6	5.9	9.0
4k	0.7	0.7	9.8	13.1
16k	1.5	6.5	38.7	36.5
64k	5.1	51.5	116.7	146.3

one processor overhead = 170 μs all other times are in milliseconds
message size is in bytes

5.4.2 Global Communication Operations

Three operations are used as a broader measure of communication performance, being more like what would occur in an actual program. These operations are scatter, gather, and an all to all broadcast. Scatter involves an operation in which a single processor sends a specific message to each processor and each processor then replies with a message indicating receipt. Gather

is similar, a single processor broadcasts a request message, and each processor then sends its message to the requesting processor. The all to all broadcast process has one processor sending a request and receiving data from each processor. This data is collected and then broadcast to all processors. The operation ends when the original processor receives an receipt message from each processor. The operations were tested using a variety of message sizes and various mixes of processors. The *Two PE* measurements used a Meiko node and a Sun SparcStation 20. The *Three PE* measurements used a Meiko node and two HP 715 workstations. The *Four PE* measurements used a Meiko node, a Sun SparcStation 20 and two HP 715 workstations. The reported one processor overhead was measured by running the application on a single Meiko node.

Table 5.15: Average time for a Scatter in milliseconds

Message Size (bytes)	Two PE	Three PE	Four PE	Four Meikos
1	6.7	11.4	21.1	1.8
4	6.6	11.7	17.9	1.8
16	6.4	11.5	18.2	1.8
64	6.6	13.4	18.9	1.8
256	7.0	13.7	24.0	2.0
1024	9.0	15.5	26.7	1.9
4k	13.2	28.7	42.8	2.5
16k	34.7	107.6	117.3	4.4
64k	124.2	370.2	900.7	14.4

one processor overhead 495 microseconds

Table 5.16: Average time for a Gather in milliseconds

Message Size (bytes)	Two PE	Three PE	Four PE	Four Meikos
1	6.8	11.2	17.1	1.8
4	6.6	11.5	16.3	1.9
16	6.6	11.7	18.2	1.9
64	6.4	12.6	19.4	1.9
256	6.9	18.8	24.6	2.0
1024	9.1	71.0	85.5	2.0
4k	155.3	58.0	38.0	2.6
16k	35.0	62.8	108.5	5.1
64k	125.2	302.5	886.1	16.4

one processor overhead = 430 microseconds

Table 5.17: Average time for a All to All Broadcast in milliseconds

Message Size (bytes)	Two PE	Three PE	Four PE	Four Meikos
1	13.8	23.7	29.1	3.8
4	14.0	22.9	36.9	3.9
16	12.9	23.0	32.6	3.7
64	13.4	24.1	35.7	3.8
256	15.9	33.1	37.9	4.3
1024	22.1	111.1	143.3	4.9
4k	219.5	127.2	124.7	9.4
16k	146.5	353.8	479.4	28.4
64k	592.5	1504.0	2327.1	129.4

one processor overhead = 830 microseconds

5.4.3 The Load Balancing Window

The existing load balancing window was an important tool for finding and eliminating sources of idleness in the metasystem. The window displays relative amounts of processor idleness while the application runs. It also gives direct feedback about the balance achieved by the load balancer. Several additions were made to the window including displaying smaller amounts of idle/computation, making it easier to change scales, and the ability for the user to specify the exact end of the initialization phase of the program through a call to `CkInitOver()`. This provides better agreement between the displayed runtime and actual runtime.

The load balancing window displays data on the program as it executes. Three bars report data on the progress of each processor. The bars are (from left to right) computation, cumulative idle and chares processed. The computation bar is time based and shows the state of the processor as the program executes. States are computation, idle and initialization. The cumulative idle bar uses the same data as the computation bar but only displays idle time. Cumulative idle is useful as a rough determination of how much time the processor spent waiting for messages without other work to do. The chares processed bar shows the number of chares that each processor executes. This is used as a check on how well the load balancer is working, and shows little useful information for application with few (or one) chares per processor.

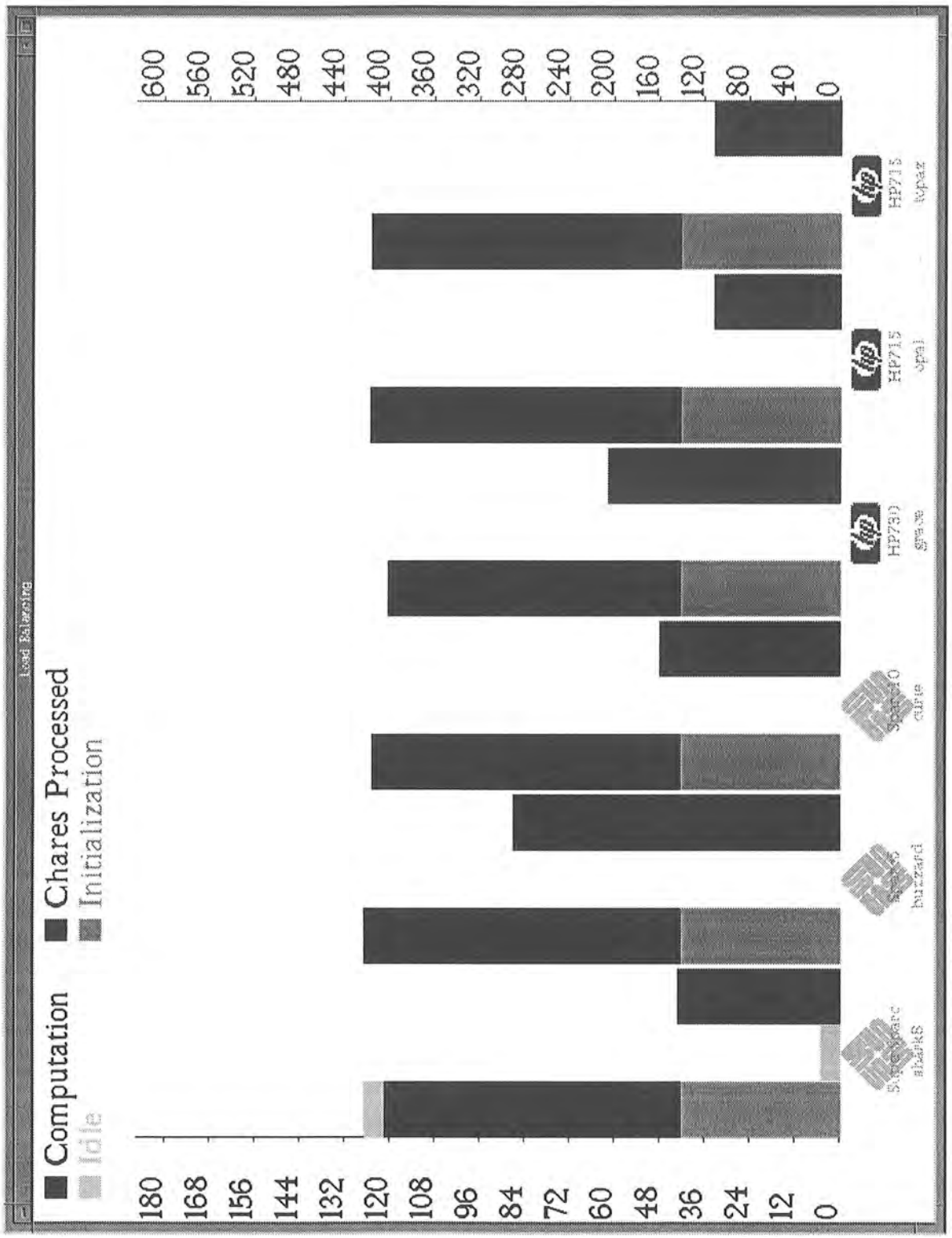


Figure 5.2: LDB Window for Dijkstra's APSP on the metacomputer

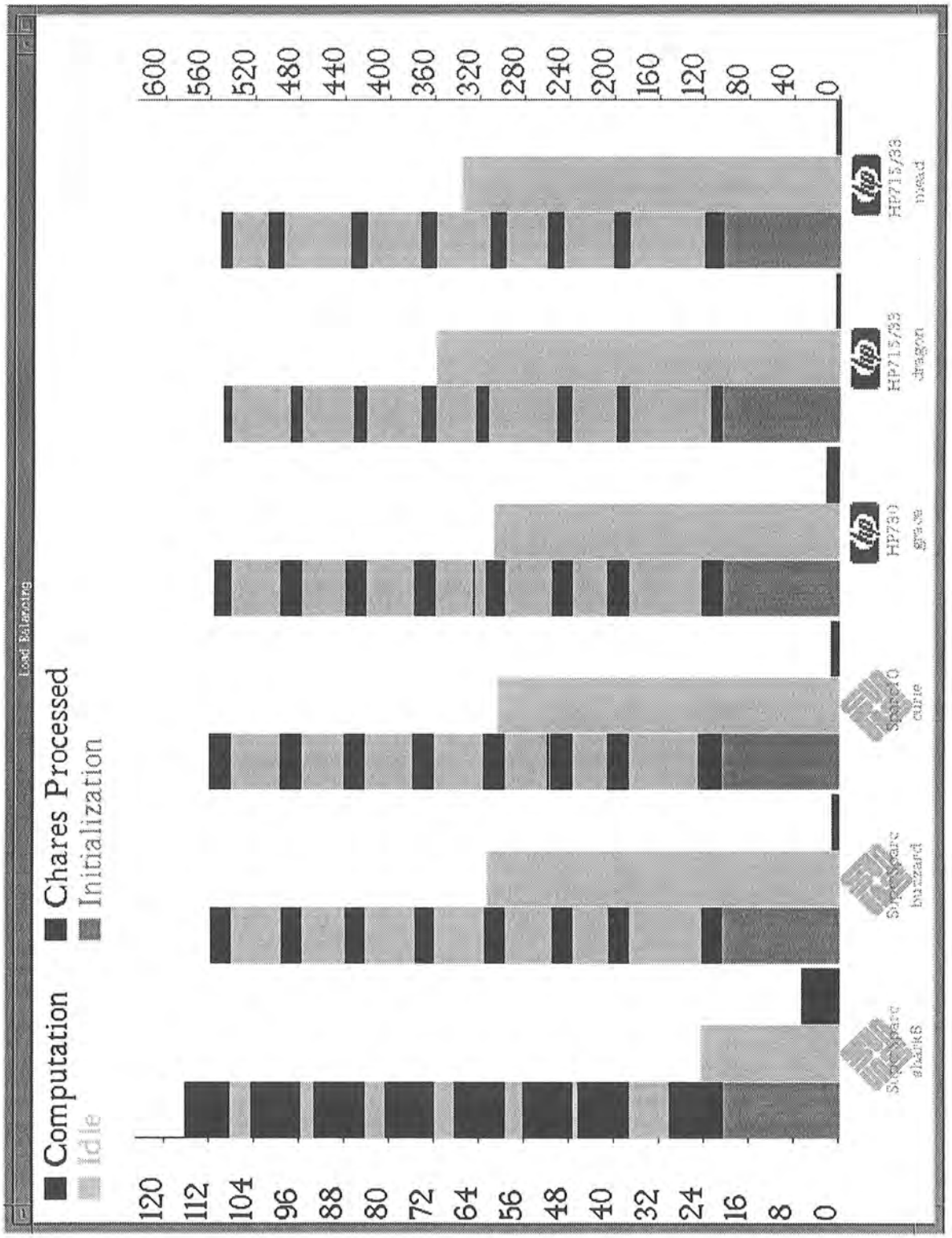


Figure 5.3: LDB Window for Cannon's Matrix Multiply on the metacomputer

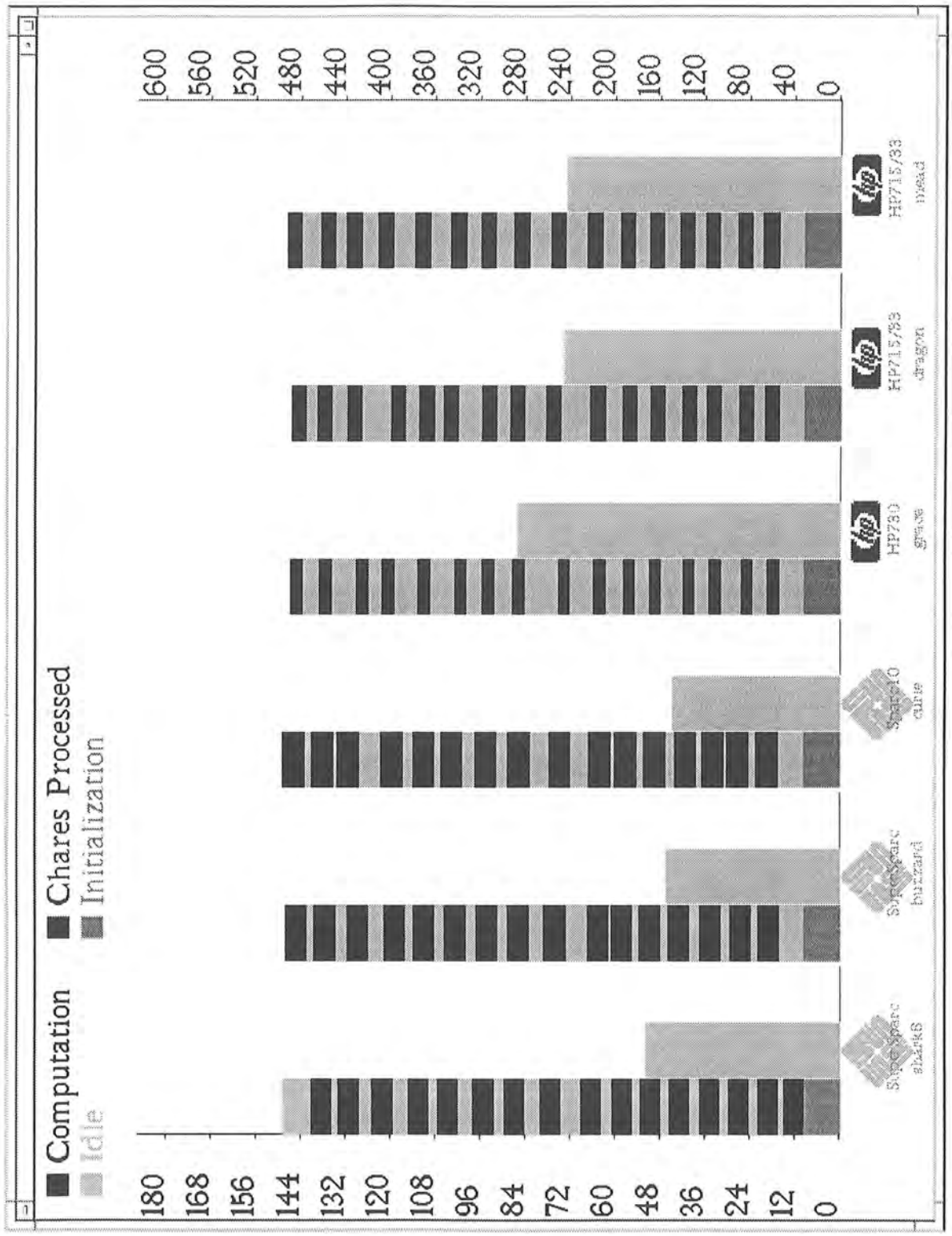


Figure 5.4: LDB Window for Fox's Matrix Multiply on the metacomputer

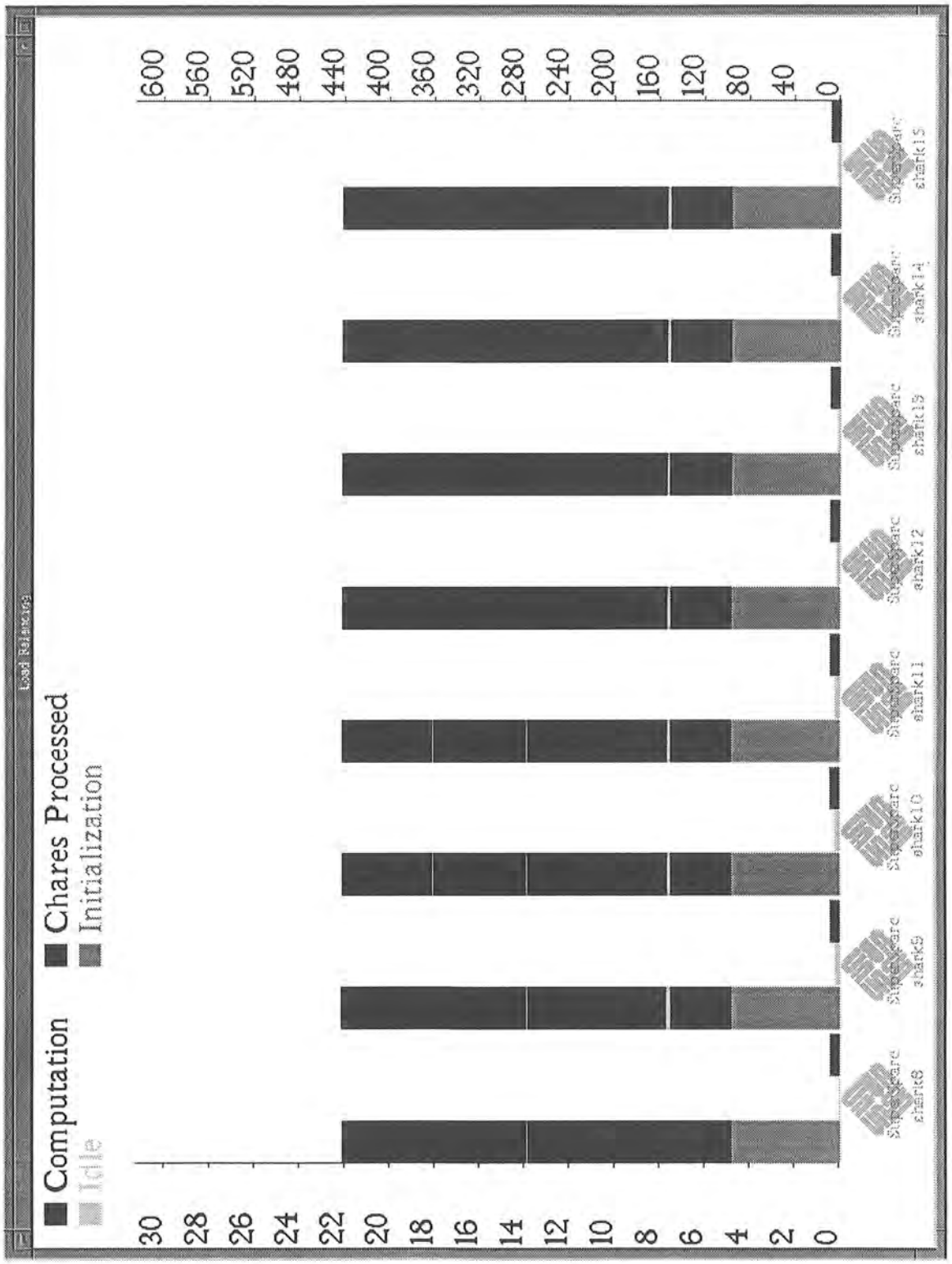


Figure 5.5: LDB Window for Cannon's Matrix Multiply on MEIKO CS-2 only

Figure 5.6: LDB Window for Gaussian Elimination on the metacomputer

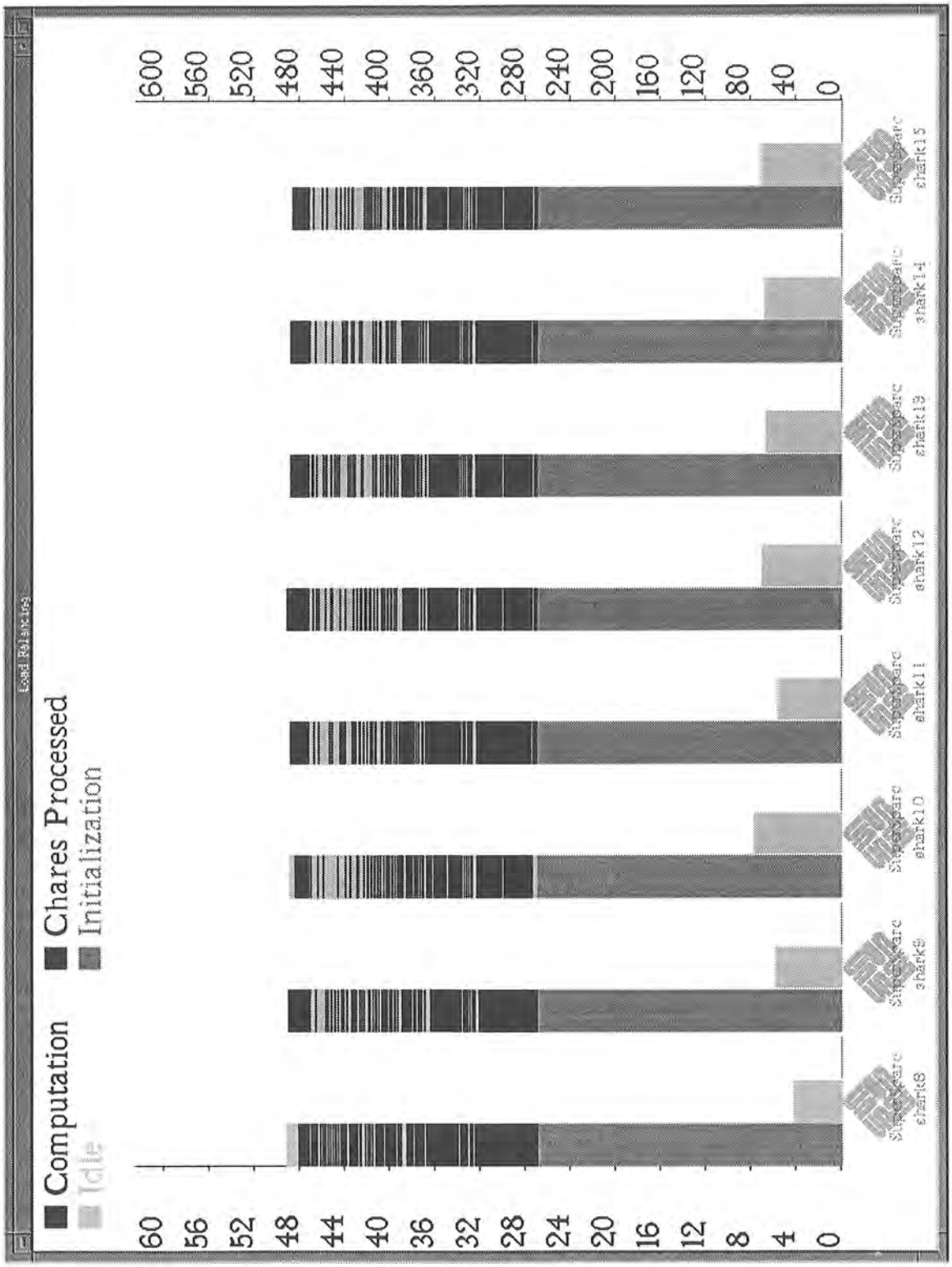


Figure 5.7: LDB Window for Gaussian Elimination on MEIKO CS-2 only

Chapter 6

SUMMARY

6.1 Application Evaluation

6.1.1 Meiko Results

The performance of the applications on the MEIKO system is quite good (compared to the same applications on the metacomputer system). As expected, performance degrades with the amount of communication in the application. Applications with very little communications (APSP) show near linear speedup, while the applications with the most communication (Gaussian and Jacobi) show the least speedup. One interesting result is that the performance of Cannon's algorithm is better than that of the conventional algorithm. Cannon's algorithm has been optimized for execution on the MEIKO by using a grain size that overlaps the computation time with the communication time. As a result the costs of communication are almost totally hidden. Also, the size of the blocks used in Cannon's has a favorable effect on the number of cache misses during computation. Cache miss problems may be the reason that Fox's algorithm performs so poorly especially when the sequential runtime is compared to the sequential runtimes for the other matrix multiply applications.

6.1.2 Metacomputer Results

The results here show only modest improvement over use of the MEIKO component of the various processor mixes. That is to say that the use of a MEIKO node plus several other workstations is only marginally better than the use of a MEIKO node alone. The exact amount of improvement varies with the amount of communication in the application and the number of non-MEIKO processors used. Applications that require frequent processor synchronization (Gaussian, Jacobi) suffer from the long idle times on the metacomputer.

The mix of processors selected for use in the experiment, might have had an additional effect. One of the advantages of using workstations is that new faster processors are typically available earlier for workstations than for multi-computers. This was not the case for workstations available for testing. The MEIKO SuperSparc processor was, for most applications, the fastest workstation used (although the SparcStation 20 was very close). Finding enough fast workstations to significantly increase the metacomputer capacity without using large numbers of workstations should increase the system performance.

6.1.3 Latency Results

Two unexpected results were the poor performance of channel communication on the MEIKO and the large latency overhead added by CHARM.

The performance of the channels using large messages can be attributed to two problems. One, that channels are not well suited to asynchronous communications, and require overhead to prevent blocking and overwriting of messages. Two, the requirement that all messages allocate a number of bytes based on the maximum message size. Node to node communication using channels directly (without CHARM) has latencies of 24 μs, using NX calls the latency increases to 78 μs. There is a discrepancy between these latencies and the measured MEIKO system latency of 400 μ. The one processor overhead can explain some of the difference, in the operations of processing and moving the message, but the other operations should not take hundreds of microseconds.

In general the metacomp latencies were not as surprising. The relatively low values of the latencies when compared to the global communications, shows the source of the slow runtimes produced by high communication metacomp applications. While point to point latencies average to low values the combined worse case aspect of the global measures clearly shows the communication difficultly over Ethernet. These delays can be seen in the load balancing window 5.3 where a synchronization can add seconds of idle to each iteration of the algorithm.

6.2 Conclusions

6.2.1 Metacomputing

The metacomputer system tested in this report was able to improve the performance of all the applications tested. The amount of improvement, however, was not as much as was expected. Based on the data taken, here are some conclusions about the usefulness of the metacomputer system.

The additional resources of workstations are a fast and inexpensive way to increase the capacity of a supercomputer. The exception is when the extra workstation's processing capacity is not great enough to overcome the overhead incurred by the addition to the system. The non-processing resources of the workstation will still aid the system, but speed will not be increased.

The metacomputer speed is highly dependent on the amount of communications in the application. Applications with little communications are a good prospect for use with the metacomputer. Reducing communication is an important way to improve performance, but cannot be done at the expense of load balancing.

6.2.2 Load Balancing

There is an problem with the use of load balancing on the metacomputer. The current system uses various methods to balance the load, but they are all static operations. They do not respond to changes in the system load. The problems is that the current dynamic load balancing techniques used by charm cannot move a chare once it has begun to execute. In practical terms only programs without communication or ones that create new chares as they execute can be dynamically balanced. For execution efficiency the opposite is true. Fewer chares mean less communication between chares. With the relatively small programs being tested this was not a

problem, but large problems running in a multi-user environment will have problems maintaining a balanced load.

6.3 Future Work

6.3.1 New Systems and Networks

The existing metacomputer CHARM environment works only on a few types of computers connected on the Ethernet. A true metacomputer will have to support more machine architectures and different interconnection networks. There are existing systems for a variety of supercomputer systems that could be incorporated into metacomputer CHARM. An existing system uses ATM as an interconnection network for workstations instead of Ethernet, this could also be incorporated into the metacomputer system.

6.3.2 Different Applications

An interesting area for study would be to design or find applications that required more than one type of parallelism. All the applications tested here were SIMD type scientific problems. These types of problems are expected to do well on a parallel computer optimized for data parallel execution. Applications with both MIMD and SIMD components might be a better fit to the metacomputer as the different parts of the problem could be matched with different parallel computers or workstations. This could also help some in the area of dynamic load balancing, as applications that create new chares as the system executes can adapt to changes in system load.

6.3.3 Channel Communication and Global Communication

The poor performance of the channels does not mean that the use of channels should be dropped. With small messages channels work as well as using the NX library calls. A better way to utilize the channels would be to use them for global communications. Rather than using channels for point to point communications, the elan widget `BCHAN` uses channels for a combined barrier and broadcast. The current CHARM system is incompatible with several machine broadcast mechanisms, including the NX library's.

Moving some of the more common global communications into standard CHARM calls would be a step to making the language easier to use. Some already exist, like broadcast when using branch office chares. But a more robust mechanism is needed for implementing operations like: scatter, gather, barrier, and all-to-all broadcast. If built into the language, lower level communications (like the broadcast channel) could be used to optimize these operations.

Bibliography

[1] A.S. Grimshaw, J.B. Weissman, E.A. West and E.C. Loyot, Jr.
Metasystems: An Approach Combining Parallel Processing and Heterogeneous Distributed Computing Systems, Parallel and Distributed Computing 21, pp. 257–270 1994

[2] J. Bruck, D. Dolev, C. Ho, M. Rosu and R. Strong
Efficient Message Passing Interface (MPI) for Parallel Computing on Clusters of Workstations, SPAA '95 Santa Barbara CA, ACM 0-89791-717-0/95/07

[3] J.N.C. Arabe, A Beguelin, B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan
Dome: Parallel programming in a heterogeneous multi-user environment, Supercomputing '95

[4] O. Babaoglu, L. Alvisi, A. Amoroso, R. Davoli, L.A. Giachini
Run-time Support for Dynamic Load Balancing and Debugging in Paralex, TR 91–1251, Department of Computer Science, Cornell University, December 1991.

[5] W. Fenton, B. Ramkumar, V.A. Saletore, A.B. Sinha, and L.V. Kale
Supporting Machine Independent Programming on Diverse Parallel Architectures, Proceedings of the International Conference on Parallel Processing, St. Charles, IL, vol. 2, pp. 193–201, Aug. 1991.

[6] L.V. Kale
The Chare-Kernel Parallel Programming System, Proceedings of the International Conference on Parallel Processing, St. Charles, IL, vol. 2, Jul. 1990.

[7] Vikram A. Saletore, J. Jacob, and M. Padala
Parallel Computations on the Charm Heterogeneous Workstation Cluster, Proceedings of High Performance Distributed Computing (HPDC3), San Francisco, CA, pp. 203–210, Aug. 1994.

[8] Jon Beecroft, Mark Homewood, Moray McLaren
Meiko CS-2 interconnect Elan-Elite design Parallel Computing 20 (1994) pp. 1627–1637

[9] Eric Barton, James Cownie, Moray McLaren
Message passing on the Meiko CS-2, Parallel Computing 20 (1994) pp.497–507

[10] A.L. Cheung and A.P. Reeves
High Performance Computing on a Cluster of Workstations, Proceedings of High Performance Distributed Computing (HPDC1), Syracuse, NY, pp. 152–160, Aug. 1992.

[11] G. Stellner, S. Lamberts and T. Ludwig
NXLIB Users' Guide Institut für Informatik, Technische Univerität München, Oct. 1993.

[12] Meiko Scientific
Elan Widget Library Meiko Limited, Bristol UK, 1993.

[13] Meiko Scientific
Vector Processing Element Overview Meiko Limited, Bristol UK, 1993.