1-1-1993

WOULD YOU RUN IT HERE ... OR THERE? (AHS: AUTOMATIC HETEROGENEOUS SUPERCOMPUTING)

H. G. Dietz
Purdue University School of Electrical Engineering

W. E. Cohen
Purdue University School of Electrical Engineering

B. K. Grant
Purdue University School of Electrical Engineering

WOULD YOU RUN IT HERE... OR THERE? (AHS: AUTOMATIC HETEROGENEOUS SUPERCOMPUTING)

H. G. DIETZ
W. E. COHEN
B. K. GRANT

TR-EE 93-5
JANUARY 1993

SCHOOL OF ELECTRICAL ENGINEERING
PURDUE UNIVERSITY
WEST LAFAYETTE, INDIANA 47907-1285

Table of Contents

Would You Run It Here... Or There?

(AHS: Automatic Heterogeneous Supercomputing)

H. G. Dietz, W. E. Cohen, and B. K. Grant

Parallel Processing Laboratory
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907-1285
hankd@ecn.purdue.edu

Abstract

Parallel programs often exhibit strong preferences for different system structures, and machines with the ideal structures may all be available within a single heterogeneous network. There is also the complication that, although a particular application might execute fastest when running by itself on one system, the best turnaround time might result from running the program on a different system that is less heavily loaded at the time the job is submitted.

In this paper, we suggest that application programmers should be able to write their code using whatever programming model, control or data parallel, they feel is most appropriate — without worrying about the heterogeneous nature of the execution environment. The system software should automatically determine where that code will execute most efficiently and cause the code to be executed there. In addition to presenting the basic concepts for building such system software, this paper describes a prototype system, called AHS, that allows a MIMD programming model to be used with heterogeneous supercomputer networks including various types of UNIX systems (including UNIX-based MIMD machines like the Sun 4/600 or the KSR-1) and even massively-parallel SIMD machines (e.g., the MasPar MP-1).

Keywords;: Heterogeneous Supercomputing, MIMD, SIMD, Compilers, Performance Prediction, Scheduling.

† This work was supported in part by the Office of Naval Research (ONR) under grant number N00014-91-J-4013, by the National Science Foundation (NSF) under award number 9015696-CDA, and by the United States Air Force (USAF) Rome Laboratories under award number F30602-92-C-0150.

1. Introduction

In any discussion of programming parallel machines, it is very important to distinguish between the *programming model* and the *execution model.* The programming model is simply the model of parallelism seen by the programmer and expressed in the high-level language code. In contrast, the execution model embodies machine-specific details, and is generally expressed either in native assembly language or in a high-level language augmented by calls to parallelism-related functions. Our claim is simply that as execution models get more complex in the name of efficiency, :programmingmodels must remain relatively stable; without this stability, parallel programming iis the purview of researchers.

Perhaps the most severe complication of parallel execution models has been caused by the combination of advances in networking and proliferation of inexpensive, high-performance, computers: the execution model is no longer one complex target machine, but a heterogeneous network of complex target machines [BeS91]. For application programmers, reflecting this complexity back into the programming model quickly makes programming intractable; selection of the execution model must be automated.

We are not the first, nor will we be the last, to propose that a single programming model should be used for a wide range of target machines. For example, there has always been a strong contingent wanting to simply write programs in Fortran and to have them automatically parallelized by the compiler. Various parallelism-related extensions to Fortran also have been proposed, ranging from the MIMD-oriented directives of The Force [JoB89] to the SIMD-oriented data layout constructs of Fortran-D [HiK92]. However, our focus here is not on what that programming model should be; rather, we are concerned with the mechanics of automating mapping *any programming model* into the *best execution model available at the time the program is run.*

This paper discusses both the general concerns and how a prototype automatic heterogeneous supercomputing system, AHS, operates. Section 2 describes how parallel languages can be designed to facilitate program development, portability, and automatic selection of the best execution model. Several of the most important execution models, and the software environments needed to support them, are described in section 3. Given these languages and execution models, section 4 describes how programs can be automatically distributed, compiled, and run using the execution model and machine choices that result in the fastest expected execution time. Finally, section 5 summarizes the contributions of this work and suggests directions for future research.

2. The MIMDC Language

Although any language could be used to write programs for an automatic heterogeneous supercomputing system, certain language properties can improve the performance of the system and can widen the range of execution models supported.

The key aspect of language design that can help performance is that the language be able to be accurately analyzed by the compiler. This is because the most efficient code structure is highly dependent on the target machine (i.e., execution model), and a compiler can only transform code to match that structure if analysis can determine that the transformation will not

change the meaning of the program. For example, Linda's tuple space [AhC91] allows messages to be routed based on runtime pattern matches that are totally obscured from the compiler's analysis; thus, communications using Linda's tuple space must be *simulated* rather than translated directly into the communication primitives provided by most execution models.

The language semantics also can severely restrict which execution models are feasible; for example, some machines have special hardware to perform "Fetch-and-Add" [Sto84], however, making this instruction part of the language would cause programs using the construct to execute very slowly on any execution model that did not directly support this operation (since: many target machine would have to execute this parallel operation using serial code). Perhaps the best overview of what kinds of semantics allow the widest range of execution models is given in [Phi89].

Although [Phi89] suggests that a single language would suffice for both control and data parallelism, it is not necessary that such a language be used. In this paper, we focus on MIMDC, a simple C-based language embodying a control-parallel programming model. However, there is no reason a data-parallel language could not be similarly supported [QuH91]; we are currently extending AHS to support SIMDC, a data-parallel dialect of C.

2.1. Syntax

Most of the compilers for MIMDC have been constructed using PCCTS, the Purdue Compiler-Construction Tool Set [PaD92]. The PCCTS grammar specifying the syntax of MIMDC is given in figure 1. The notation used by PCCTS is an Extended Backus-Naur Form (EBNF) in which quoted items and names in uppercase are terminal symbols and lowercase names are nonterminals. Meta-symbols are used to simplify the grammar:

{a}	An optional a
(a)*	Zero or more occurrences of a
(a)+	One or more occurrences of a
a|b	Either a or b
"@"	The end of file marker

23. Data Declarations

MIMDC allows both integer and floating point data, and allows data items to be declared either as local to each process or as shared across all processes.

Declaring a variable to have the attribute `poly` indicates that each processor has its own local value for that variable. Thus, modifying a `poly` variable's value in one process has no effect on the value of the variable in any other process. Function arguments and return values are always `poly`; the default storage class for all variables is `poly`.

```
prog : ( type IDENT ( "\[" INT—NUM "\I" | )
         ( ( "," IDENT ( "\[" INT_NUM "\]" | ) )* ";" | def )
       )* "@" ;

type : { POLY | MONO } ( INT | FLOAT ) ;

def : "\(" { type IDENT ( "," type IDENT )* } "\)" stat ;

stat : "\{" ( type IDENT ( "\[" INT_NUM "\]" | ) ";" )*
         ( stat )* "\}"
     | lval ASSIGN expr ";"
     | IF expr stat ( ELSE stat | )
     | WHILE expr stat
     | RETURN expr ";"
     | WAIT ";"
     | HALT ";"
     | ";"
     ,

lval : IDENT { "\[" expr "\]" } { "\[\|\|" expr "\]" } ;

expr : e6 ( OR e6 )* ;
e6 : e7 ( AND e7 )* ;
e7 : e8 ( EQ e8 (NE e8 )* ;
e8 : e10 ( GT e10 | ">" e10 | ">=" e10 | GE e10 )* ;
e10 : ell ( SHR ell | SHL ell )* ;
ell : e12 ( PLUS e12 | NEG e12 )* ;
e12 : e13 ( MULT e13 | DIV e13 | MOD e13 )* ;
e13 : INT—NUM
    | FNUM
    | IDENT ( "\(" { expr ( "," expr )* } "\)"
    | { "\[" expr "\]" } { "\[\|\|" expr "\]" } )
    | NEG e13
    | NOT e13
    | "\(" expr "\)"
```

Figure 1: PCCTS Grammar For MIMDC.

When a variable is declared to have the mono attribute, it means that all processes see the same value for that variable. When one processor modifies a mono variable, all other processors that access that variable afterwards get the new value. If multiple processes store into a mono variable simultaneously, the race is resolved by picking a winner and storing that valur:. Variables defined with the mono attribute are never allocated on the stack; they have the same apparent address in all processes.

23. Communication And Synchronization

There are two basic ways that MIMDC processes can communicate with each other:

1. By storing into, and reading from, mono variables.
2. By storing into, or reading from, poly variables *that reside* in *another process* — by "parallel subscripting."

Thus, MIMDC allows two different models of shared memory to be used. Using mono variables is very efficient for information where a few values will be accessed by many processes; shared memory accesses in which processes simultaneously access many different values are more efficiently implemented by parallel subscripting of poly variables.

The parallel subscript operation allows one process to read or write a global poly variable owned by another process. The variable must be a global variable because local poly variables can be stack allocated, hence, there is no way for another process to locate such a variable within another process (e.g., that process might not even have allocated the variable). An example of the parallel subscript operation is in figure 2. The this variable holds the AHS process number. Thus, in figure 2, process 0 stores the value of 5 in process 1's local copy of a.

```
poly int a;
int main()
{
    if(this == 0) a[|1] = 5;
}
```

Figure 2: Example Of Parallel Subscripting.

Then: are also two basic ways that synchronization can be accomplished:

1. Shared variable accesses can be used to implement semaphores, locks, or any other type of synchronization primitive.
2. All processes can be synchronized using a barrier mechanism invoked by the wait statement.

Effort has been made to make the barrier synchronization relatively efficient. All processes must reach a wait statement before any is allowed to continue executing past the wait. Note that all processes need not be waiting at the same wait statement.

2.4. The Compilers

Proof-of-concept compilers have been written to translate MIMDC into a variety of target languages.. As mentioned earlier, most of these compilers were written using a set of locally-developed software tools called PCCTS [PaD92]. From a single specification, PCCTS automatically constructs the lexical analyzer, parser, and even code to create and manipulate abstract syntax trees (ASTs).

2.4.1. Code Generation

The code generation for expressions is not handled by embedded code generation actions, but by traversals of the automatically-genera child/sibling ASTs. This allows the code generation to make multiple passes over each AST, transforming and optimizing it before final code generation. Nearly all the proof-of-concept compilers use this technique to perform at least constant folding and algebraic simplification. Although the MIMDC compilers for a few targets implement far more complex optimizations, these are not in the purview of this paper. Type coercion is also applied on the ASTs; not only to conversions between integer and floating-point data types, but also to operations mixing `mono` and `poly` values (e.g., all operations except assignment to a `mono` require `poly` values).

For the execution models discussed in this paper, the final code is always some variation on stack code generated by walking the optimized ASTs. However, different kinds of stack code are generated for the different execution models.

2.4.2. Stack Code For The MasPar MP-1

The MasPar MP-1 [Bla90] is a SIMD supercomputer. For it's execution model, described in section 3.1.4, the stack code is actually the MIMD assembly language that can be assembled and executed directly by our MIMD interpreter. This stack code is designed specifically to be efficient for the MIMD interpreter. For example, the concept of a "frame pointer" was omitted to make the interpreter execute more efficiently... even though this makes it much harder for the compiler to track stack offsets for local variables.

Because the MasPar's interpreted MIMD instruction set was designed specifically to treat the MasPar as a shared-memory MIMD with barrier synchronization, the stack code generated never needs to call a runtime support routine. Literally every operation in MIMDC is directly implemented by no more than a few MIMD assembly language instructions. For example, the code for:

```
poly p; mono m; ... p[|m] = 5;
```

Looks like:

```
Push address_of_m    ;address of m in mono memory
LdS                  ;get poly copy of m's mono value
Push address_of_fg   ;address of p relative to local base
Push 5               ;value to store into p on PE m
St.D                 ;store 5 into p[|m] & clean stack
```

2.43. Stack Code For The Other Targets

Although all the other target execution models also use stack instruction sets, the instruction sets are quite different from that of the MasPar. For example, all these models implement frame pointers. Further, the stack instructions are actually implemented by C macros. An integer add for the UNIX file-based execution environment (described in section 3.2.2) is defined as:

```
#define ADD { NOS = (NOS + TOS); POP; }
```

Some of the more complex functions, shared memory accesses and barrier synchronizations, are actually implemented by library support functions. For example, the MasPar's MIMD assembly language LdS instruction roughly corresponds to the following C macro for the file-based execution model:

```
#define LDS { TOS = m_lds(TOS); }
```

but this macro uses `m_lds()`, a non-trivial library support routine. The interfaces to all the library support routines are hidden in this way.

The key concern in design of these "instruction sets" is not efficiency, but portability. The output of the compilers is designed to be easily compiled by whatever C compiler supports the target machine. We simply try to make the C code as obvious as possible — in the hope that the native C compiler will perform appropriate local optimizations.

3. Execution Models

In the previous section, we discussed how the MIMDC language and its compilers work. Here we discuss how the code generated by the compilers actually implements the language features in the various execution models. Since the primary goal of AHS is to be able to automatically run parallel programs in the most expedient place — be it within a single machine or across multiple networked machines — it is vital that the system be completely supported on all targets.

3.1. MIMD On A MasPar MP-1

Since the MasPar MP-1 is a massively-parallel SIMD machine [Bla90], one might assume that it would be disallowed as a MIMDC target. However, even if a SIMD machine achieves only a small fraction of its native performance while simulating MIMD execution, it may still be the fastest place to execute a MIMDC program. Since 1991, techniques have been developed that allow SUMD machines to obtain reasonable efficiency while simulating MIMD execution

[DiC92]. In this section we briefly discuss the MIMDC execution model for the MasPar MP-1.

3.1.1. Simulation Of MIMD On SIMD

It is a relatively simple matter to construct an interpreter that will implement a MIMD execution model on a SIMD machine. Machine registers such as the "instruction register," "program counter," etc. can be stored in data parallel structures so that each PE (processing element) in the SIMD machine can emulate a MIMD PE by:

Basic MIMD Interpreter Algorithm

1. Each PE fetches an "instruction" into its "instruction register" (IR) and updates its "program counter" (PC).
2. Each PE decodes the "instruction" from its IR.
3. Repeat steps 3.1-3.3 for each "instruction" type:
   - 3.1 Disable all PEs where the IR holds an "instruction" of a different type.
   - 3.2 Simulate execution of the "instruction" on the enabled PEs.
   - 3.3 Enable all PEs.
4. Go to step 1.

Several researchers [LiM90] [NiT90] [WiH91] have discussed MIMD interpreters based on this structure. However, it is possible to use a more aggressive approach to achieve higher performance. In the following section we examine why performance of a naive system is poor. Section 3.1.3 discusses how we achieve good performance; in fact, on the MasPar MP-1, MIMD performance is typically between 1/40 and $1/5^{th}$ of peak SIMD performance [DiC92].

3.1.2. Limits On Performance

Two major problems involve characteristics of the SIMD hardware:

- PE indirect addressing. In steps 1 and 3.2 of the MIMD interpretation algorithm, each PE may have to access a different location in its local memory. Some SIMD machines do not support this [Thi90], in which case the accesses must be made sequentially.
- PE masking overhead. Step 3 requires the PEs to enable/disable themselves depending on the values in their "instruction registers." Some SIMD machines either have no enable hardware [Thi90] or only allow the control unit to supply enable masks [SiN90], in which case expensive arithmetic/bitwise operations must be used to nullify the effect of operations on PEs that should have been disabled.

Fortunately, the MasPar MP-1 has hardware support for both indirect addressing and masking; a relatively efficient MIMD emulation should be possible. Two key problems remain, both involving how the simulator is constructed:

- **Interpreter overhead.** Step 2, and the entire interpreter cycle, require code to implement what would be built-into the hardware for the native instruction set. In particular, decoding simulated instructions can take much longer than executing them.
- **SIMD serialization. In** step 3, SIMD hardware can only simulate execution of one instruction type at a time. This serializes the execution of different instruction types, forcing most SIMD PEs to be idle while waiting for their instruction to be interpreted.

Fortunately, these last two problems are largely solved by the approach discussed in [DiC92]. The next section describes how we minimized these last two performance bottlenecks.

3.13. Simulator optimization

The. MIMD execution model for the MasPar has been very aggressively optimized. However, most of the optimizations were implemented by specialized software tools, and could be applied to make simulators to run on other SIMD machines. Here, we focus on what is done and how it improves performance.

3.1.3.1. PE Register Use

Each PE on the MasPar MP-1 has local memory, but each group of 16 PEs shares a single 8-bit mennory interface. To help avoid slow memory references, each PE also has 48 32-bit registers directly accessible. We use these registers to reduce interpreter overhead in two ways.

Stale variables for the simulated MIMD PEs are generally kept in PE registers. This speeds access to the program counters, instruction decode registers, stack pointers, etc.

The second use of registers is a bit less obvious. We would like the simulated MIMD instruction set to be able to use registers for user data, but the MasPar MP-1 architecture restricts accesses to be for the same register in all enabled PEs. This makes it impossible to efficiently implement a MIMD register file using PE registers.

For maximum speed, the simulated MIMD must be limited to a single register for user data. This register will be the source or destination operand for almost all instructions. Rather than use a single accumulator model, we treat this register as a top-of-stack cache. This use averts at least one operand fetch (and one store to memory) on all unary and binary operations.

3.1.3.2. Common Subexpression Induction

Since MIMD instructions are interpreted by a SIMD program, each MIMD instruction actually corresponds to a series of SIMD instructions. Although two different MIMD instructions cannot be interpreted simultaneously, any SIMD instructions that are common to both MIMD instructions can be shared by both. If this is done carefully, most of the execution time for most instructions can be spent in shared SIMD code sequences, thus reducing the SIMD serialization discussed above.

This recognition of common SIMD code sequences can be done by hand for very simple MIMD instruction sets. However, a usably large instruction set makes hand factoring infeasible.

Our simulator uses a new compiler optimization, called "Common Subexpression Induction" (CSI) [Die92], to automatically induce common SIMD subsequences. The CSI algorithm is too complex to detail in this paper. Briefly, operations from various threads are classified based on how they could be merged into single instructions executed by multiple threads, followed by a heavily pruned *search* to find the minimum execution time code schedule using these merges.

The CSI algorithm allowed automated identification of useful subsequences that factored out common instructions from original, unoptimized, emulator instruction sequences. However. the CSI tool produces unstructured masking and control flow which MPL (MasPar's data-parallel C dialect [Mas91]) does not allow, so the most useful CSI output subsequences had to be hand coded for the MIMD interpreter written in MPL.

These sequences included the basic instruction fetch and program counter increment, fetching the value for the next-on-stack (NOS), fetching the value for an 8-bit immediate, and looking-up a 32-bit value in the constant pool. Without this factoring, the interpreter would be several times slower.

3.1.3.3. Subinterpreters

Another way to reduce the SIMD serialization, and also to reduce the interpreter instruction decode overhead, is to make it appear that there are fewer instructions in the instruction set. This can be done by allowing only a portion of the instruction set to be decoded and executed in any given interpreter cycle. Thus, a set of subinterpreters is created, each of which understands only part of the MIMD instruction set, and the interpreter decides, for each interpreter cycle, which subinterpreter to invoke.

A C program automatically generates optimized subinterpreters. In the MasPar MP-1, the control unit can quickly "or" values from all the PEs. By carefully encoding the MIMD instruction set, we can "or" together the MIMD opcodes from all PEs to determine which MIMD instructions PEs want to execute in this interpreter cycle. This result is then used by the control unit to select the cheapest of the 32 subinterpreters that understands all those instructions. .

The MasPar MIMD interpreter also uses a technique called "frequency biasing" to artificially reduce the number of different types of instructions executing in a given interpreter cycle. Frequency biasing simply ignores some instructions for *n* out of every *m* interpreter cycles. This has two effects. First, it reduces the maximum average cycle time of the interpreter by allowing quick instructions to be executed in every interpreter cycle, while only slightly delaying expensive instructions. Second, it groups together (temporally aligns) expensive instructions that are only an interpreter cycle or two misaligned. Alignment improves performance because SIMD execution time is not proportional to the number of PEs active for the operation; e.g., two PEs executing a multiply takes much less time than two multiply operations executed sequentially.

3.1.4. Details Of The Execution Model

The MIMDC language embodies a number of decisions about how the target MIMD machine should function. In this section, we briefly outline how each of these functions is mapped into the interpreted MIMD instruction set.

The MIMDC environment for the MasPar MP-1 is organized as follows. First, the code is compiled to generate a special MIMD assembly language code. This code is then assembled by an assembler called `mimda`, producing an Intel-format absolute object file. Finally, an executable shell script is produced that will invoke the MasPar's mimd interpreter (mimd) and feed it the object file. Thus, if the user does not look too closely, the system appears to be a native compiler for MIMDC.

The MIMD assembly language is fairly straightforward, with operations modeled after those of MIMDC. Local references to `poly` variables are encoded as `Ld` and `St` instructions. Notice that no distinction is made between `int` and `float` variables, since both are 32-bit words.

Although the MasPar MP-1 does not have a shared memory, additional instructions implementing shared memory accesses are used to support parallel subscripting and operations on mono variables. The `LdD` and `StD` instructions are used to access `poly` values with parallel subscripts. The MasPar's message-oriented, SIMD controlled, global router is used to implement transfers; under AHS, each message always holds one 32-bit word of data.

Surprisingly, mono variables are not stored in the MasPar's control unit. They are actually treated as `poly` variables that are accessed by mono instructions. Thus, a `LdS` is actually identical to `Ld`. `StS` picks a winner for each mono being stored into and then broadcasts that value to every PE's copy of that mono.

Shared variables can be used for synchronization, but MIMDC supports a barrier synchronization mechanism that can be implemented much more efficiently as a single interpreted instruction. The `Wait` instruction disables a PE until all active PEs are similarly disabled. When there are no active PEs, the interpreter simply re-enables all PEs that were waiting at barriers.

3.2. MIMD And Timeshared Machines

Perhaps the most generic target machine is a system running multiple processes within a single UNIX platform. This UNIX system may have one or more processors; for example, the same execution model is used for both a single-processor UNIX-based workstation (e.g., Sun 3/50, IBM RS6000/530, Sun 4/490, and DEC 5000/200) and a multiprocessor system (e.g., Dual Gould NP-1, 4 CPU Ardent Titan P3).

In order to make the code generated by the compiler run on nearly any UNIX platform, the compiler is designed to generate a simple stack code that is implemented using C macros. Only the most fundamental methods of process control and interprocess communication can be used, because various different versions of UNIX implement different "fancy" process communication and management. To further enhance portability, there are two implementations of this execution

model: one based on UNIX pipes and another based on accessing a shared file. It is possible to create many more efficient execution models for specific UNIX boxes by using threads, shared memory primitives, etc.; the two implementations presented here are merely the most portable.

3.2.1. Pipe-Based Execution Model

When asked to create *n* processes, this implementation actually creates n+1 processes: *n* PE processes and one control process. The control process is responsible for managing shared memory access, synchronization, and housekeeping functions; only the PE processes execute user code.

When execution begins, the control process creates the PE processes and UNIX pipes to communicate with them. *All* PEs send information to the manager process through a single shared pipe, but the manager has a separate pipe to respond to each PE. This is done so that the server does mt need to use polling or interrupts, which are both less efficient and less portable. Unfortunately, it does require that the control process be able to have n+1 pipes open simultaneously.

Each communication through a pipe is a "packet" sent using a single `write` call (to ensure atomicity). Shared memory (mono) variables can be operated on by a PE sending a packet requesting either that a value be loaded from shared memory (i.e., sent to the requesting PE) or that a supplied value be stored into shared memory. Because pipes are treated as memory buffers in UNIX, the communication overhead is generally dominated by the UNIX task switch time, and communication *can* be quite fast.

Access to local `poly` values is accomplished using ordinary memory references within the PE processes. Unfortunately, the use of parallel subscripting to access the `poly` values of other PEs is very inefficient because the requests are filtered through the control process and are not processed until the PE that owns the value communicates with the control process for some other request. Thus, programs making use of parallel subscripting probably should not be run using this execution model.

A PE *can* also indicate that it is waiting for a bamer synchronization by sending the control process a packet containing the appropriate message. After sending the wait packet, the PE simply attempts to `read` from its input pipe; because UNIX `read` blocks if no input is available, this efficiently puts the PE process to sleep until the control process has sent a dummy packet back to cause the `read` to complete. One might have expected the control process to use a UNIX `signal` to inform the PE processes that a bamer has been reached, however, in some UNIX systems, signals can be lost under certain circumstances.

For the same reason, the PEs are also responsible for sending a message that indicates when they have completed execution. This "death" message is used by the control process to determine when all processes have reached normal termination. Hence, under normal operation, PE processes are only killed by the control process.

3.2.2. File-Based Execution Model

The file-based implementation is quite different from the pipe-based version, and is nearly always more efficient — if it works. In the pipe-based model, all PEs write into the same pipe (to the control process), but each PE reads from a different pipe; in the file-based model, all PE processes, read and write the same file. Although this should function correctly on any UNIX system, and even across UNIX systems, we have found it to be erratic on some multiprocessors (e.g., Dual Vax 11/780 and the Sun 4/600) and on computers using files mounted via NFS (the Network File System).

When execution begins, the first step is the creation of a file to hold the combined state of all PEs and shared (mono) memory. Once this file has been initialized, n-1 additional processes are created for PEs 1 through n-1; the original process becomes PE 0. There is no control process mediating between the processes during execution, but only the contents of the shared file from which each PE can determine the state of all other PEs. Thus, this file acts as a shared memory image for the entire parallel machine.

Access to a mono variable is accomplished simply by an `lseek` followed by a `read` or `write` operation. Because UNIX attempts to buffer file blocks in memory, these accesses are usually as fast as single pipe operations. StS operations are slightly faster than using pipes, but LdS operations are much faster: just one `lseek` and `read` as opposed to two `reads`, two `writes`, and two process context switches (between the requesting PE and the control process).

As with the pipe-based model, local `poly` variables are accessed directly from memory. However, each PE's `poly` variables also have "shadow copies" in the shared file, and these can be accessed by other PEs using parallel subscripting. These shadow copies are not continually updated, hence, parallel subscripting is again somewhat inefficient.

Barrier synchronization is also accomplished using the shared file. A section of the file is used to maintain a counter, for each PE, of how many barriers have been reached since execution began. When a PE has reached a barrier, it updates its counter and then reads the block of counters for all PEs. If every PE's barrier counter is greater than or equal to this PE's barrier count, then the PE resumes execution[1]. This barrier implementation is also quite efficient.

When a PE completes execution, it essentially flags itself at "the final barrier" and then terminates. Only PE 0 waits for this final barrier, after which it deletes the shared file and terminates normally.

3.3. Distributed Computing On A Network

Because networked UNIX workstations are so plentiful, there has been a great deal of interest in using their idle cycles as an economical alternative to parallel supercomputers. The execution model discussed in this section can distribute the PE processes of a program over a

[1] A PE's barrier counter can be greater because that PE may have recognized that all PEs had reached the barrier, and executed up until the next barrier, before this PE was able to determine that all PEs had reached the barrier. However, the barrier counters can never differ by more than one.

group of networked UNIX machines. Each of these UNIX machines may be a uniprocessor or a multiprocessor system. Further, each machine may execute any number of PE processes. For example, a 10 PE process program might be run with 7 PE processes within a 4 CPU multiprocessor machine, 2 more on a uniprocessor workstation, and the final PE process on a heavily loaded timeshared UNIX mainframe.

Unlike the other execution models discussed, AHS's distributed execution model is not responsible for allocating *n* PE processes. Rather, the allocation is managed by the techniques described in section 4.2. The execution model is responsible for creating PE programs that can act together even if some PE processes share the same node and others are on different nodes. One of the most efficient UNIX mechanisms for such interaction is UDP packets sent via a BSD Socket (henceforth referred to as the "UDP Socket" model). Unfortunately, this mechanism is not available on some versions of UNIX.

Whereas PVM [GrS92] creates persistent "daemon processes," and then uses them to mediate between PE processes, AHS uses no daemons. PE processes are created by `rsh`, and communicate directly using a UDP Socket. Because there are no daemons, the PE code must manage asynchronous communication through the UDP Socket, and this requires fairly complex signal-driven event handling code in the PE program. However, the MIMDC compiler generates this code for the user. This code can address PE data structures, hence, communication is much more direct and considerably faster than sending a daemon data for the daemon to send, as one might in PVM.

Superficially, the UDP Socket is treated much like the pipes in the execution model described in section 3.2.1, but there are many differences. There is no control process and only one open file descriptor (one UDP Socket) is needed no matter how many processes are communicating. Further, because the UDP Socket is handled by signals, parallel subscripting of `poly` variables is reasonably efficient. Thus, each mono variable is assigned to a particular PE and is accessed using the same mechanism that implements parallel subscripting.

Messages sent through pipes arrive in order, UDP Socket communication is in the form of messages, but UDP does not ensure that messages will be received in the order in which they were sent. PVM's solution is to impose an ordering by allowing only one message to be pending; a second message is not sent until the acknowledgement for the first has been received. In contrast, we take advantage of two key observations:

1. In most cases. it does not matter if the order of messages is slightly perturbed. Further, the MIMDC compiler can tell exactly when message order does matter. All sends, are generated as unordered UDP messages. Where an order must be imposed. the compiler simply generates code that forces all pending messages to be received before any new ones can be initiated.
2. It so happens that there really is no such thing as a UDP message acknowledgement; literally, the acknowledgement is just another UDP message. Some protocol is needed to ensure that no message is lost, but we need not actually send an acknowledgement for each message. For example, we could send a single

acknowledgement for receipt of several messages. In other words, the acknowledgement can carry information just like any other message. AHS's implementation of barrier synchronization is based on this fact.

Barrier synchronization is implemented using a variation on the usual n^2 algorithm. The algorithm is more complex, but is fairly effective in minimizing the delay in recognizing that a barrier has been reached by all PEs. When a PE reaches a barrier, it sends UDP messages containing bit masks that summarize which PEs it knows have arrived at the barrier. Acknowledgements carry the same information from the receiving PE so that knowledge of the set of waiting PEs spreads much more quickly than in the usual n^2 method. The savings comes for the fact that if PE a tells b that it has arrived, then b tells c that it has arrived, the single message from b informs c that both a and b have arrived.

4. Target Selection And Program Execution

Most research in parallel processing focuses on achieving a large fraction of the target machine's rated peak performance; however, the primary concern in heterogeneous supercomputing really should be achieving the fastest possible execution of the user's code. In other words, we want the system to minimize the expected execution time. For example, if all we have is a Sun Sparc workstation and a 16,384-PE MasPar MP-1, most MIMDC programs with parallelism width 128 should probably be run on the MasPar because it will be faster — althoughi the utilization of the MasPar will be less than 1%. However, if the MasPar has a multitude of jobs waiting and the Sun is idle, running this code on the Sun may result in the smallest expected execution time, so the system should select to run the program on the Sun in this case.

There are a multitude of problems that need to be addressed in order to minimize execution time using heterogeneous supercomputing. The goal of AHS is to take all the key factors into account, yet be as unobtrusive as possible. In other words, we want the user to be able to be blissfully ignorant of how the system works; this approach may sacrifice a little performance, but it should make users more productive.

The following section discusses how the AHS prototype records the vital information about each combination of execution model and machine. Section 4.2 explains how programs are analyzed and, using the execution model and machine information, how the target is selected. Finally, section 4.3 discusses how the program is actually made to execute on the automatically selected target.

4.1. Execution Model And Machine Database

In the AHS prototype, there is a file that contains all vital information about each combination of execution model and machine known to the system. At the time the system is configured, an entry is made for each combination. Much like PVM's machine database file [GrS92], each entry contains information about how each machine can be used:

- The name of the machine. This is typically the internet address of the machine.

- **The width of the machine.** This number is the maximum number of PEs that can execute on the machine. For a parallel computer with a fixed set of PEs and no support for virtual PEs, the width is recorded as the number of actual PEs. For a uniprocessor or multiprocessor UNIX system, we use a width of 0 to indicate that an essentially unlimited number of processes can be executed. Further, because the distributed execution model requires UNIX support, only machines with a width of 0 are able to host PEs for distributed execution.
- **The compile and run script.** This information describes how to initiate a user program using this host and execution model.

However, in AHS, each entry also contains detailed information that can be used to accurately predict relative performance of any given program. This performance-prediction information is composed of two types of timing information: stable properties and load-dependent properties.

4.1.1. Stable Timing Properties

The stable timing properties of a combination of execution model and machine are represented as a list giving the **execution time for each basic operation.** Although the various targets might use different instruction sets, etc., AHS defines a basic set of operations that are used for predicting execution time. The approximate execution time, in seconds, is recorded for each of these basic operations. Normally, the times are determined experimentally, and are entered at the time the system is configured.

A support program (called `timer`) has been created to use UNIX process timing facilities to measure the execution time for each basic operation. UNIX timing is only accurate to $1/60^{th}$ second, so accurate estimates are obtained by timing a set of long-running sections of code and then solving the resulting set of equations to determine the time for each operation. Accuracy is limited by the averaging effect of the long runs and by UNIX scheduling anomalies (e.g., being charged for time spent processing another processes' interrupt). We attempt to minimize these imperfections by using 5-point median filtering on the computed times, but even this gives a typical accuracy of only about +/-10%. The good news is that even a 50% error in one of these estimates is unlikely to have a significant adverse affect on the performance of AHS.

It is also useful to note that some operations might not be supported by a particular target. For example, parallel subscripting is impractical under Some combinations of execution model and machine. In such a case, the unsupported operations are simply not listed. If an unlisted operation is needed for a particular program, the system treats that operation as having an infinite execution time, which forces a different target to be selected.

Table 1 samples the basic operation times for a variety of machines available within Purdue University's School of Electrical Engineering. The first four machines are UNIX-based uniprocessors. The second four are UNIX-based multiprocessors with two or four processors each. The next is a massively-parallel supercomputer with 16,384 processing elements. Finally, the last line is for a typical network of UNIX-based systems (e.g., Sun 4) connected by a single Ethernet. In each case,,the times quoted are single-process times for unoptimized operations.

Machine	Execution Model	ADD Time	LDS Time
Sun 3/50	Pipes Shared File	5×10^{-6}s 5×10^{-6}s	1×10^{-3}s 7×10^{-4}s
IBM RS6000/530	Pipes Shared File	2×10^{-6}s 2×10^{-6}s	5×10^{-4}s 3×10^{-4}s
Sun 4/490	Pipes Shared File	1×10^{-6}s 1×10^{-6}s	3×10^{-4}s 1×10^{-4}s
DEC 5000/200	Pipes Shared File	8×10^{-7}s 8×10^{-7}s	2×10^{-4}s 1×10^{-4}s
Dual Vax 11/780	Pipes	1×10^{-5}s	5×10^{-3}s
Dual Gould NP-1	Pipes Shared File	2×10^{-6}s 3×10^{-6}s	1×10^{-3}s 1×10^{-3}s
Sun 4/600	Pipes	1×10^{-6}s	8×10^{-4}s
4 CPU Titan P3	Pipes Shared File	5×10^{-7}s 2×10^{-7}s	1×10^{-3}s 8×10^{-4}s
16K MasPar MP-1	MIMD Interpreter	5×10^{-5}s	5×10^{-5}s
UNIX network using Ethernet	UDP Socket	*Same time as Pipes*	3×10^{-3}s

Table 1: Sample Operation Times For Some Targets

Several interesting observations can be made from this table. Perhaps the most obvious is that, with the exception of the MasPar, communication time (LDS Time) is always much greater than compute time (ADD Time). It is somewhat surprising that the UDP Socket model can perform communication over an Ethernet nearly as fast most UNIX machines could communicate between processes within a single machine. This is still more surprising in that if we had used PVM [GrS92] to construct the AHS distributed model, the LDS Time would have been about 1.6×10^{-1}s (as we measured it on the same systems used for the UDP Socket timing). However, AHS's UDP Socket model avoids most of PVM's system overhead, and that is actually the dominant portion of the PVM communication time; this is demonstrated by the fact that using PVM for an LDS of a variable that ***resides on the requesting machine*** also yields a time of about 1.6×10^{-1}s.

4.1.2. Lorid-Dependent Timing Properties

While the above numbers appmximate the ***best*** execution time for each basic operation (i.e., they correspond to UNIX "user time" plus "system time"), they do not provide an accurate prediction of what performance will be when a program is run. The reason is simply that other

programs may be running on these machines, hence, execution may be slower by a factor proportional to the number of processes currently sharing each machine. This information is recorded for each combination of execution model and machine as:

- **The last known load average..** This number is a multiplicative factor that indicates how much slower the machine was last known to be executing due to other programs being run on the system. Because not all programs are compute bound, the load average is rarely an integer. If no load average is listed, this is interpreted as meaning that the machine is currently inaccessible.
- **The load average increment.** Whenever this system assigns a job to a machine, the load average for that machiie may change to reflect that another process must be scheduled. This value is the increment by which the load average changes for each additional process scheduled on the machine. Under UNIX on a uniprocessor, it is assumed to be 1.0; for an n-processor UNIX-based multiprocessor, it is assumed to be $1.0/n$. These numbers are based on the premise that any program run under this system is compute bound. If the machine does not run UNIX (i.e., if the width is not 0). the load average increment is usually 0.0 — indicating that using processors that would otherwise be idle does not slow the other processors that are in use.

Notice that only the load average is likely to change after the system has been configured.

Ideally, one might like to automatically update the load average information just before deciding where to run each user program. However, this is usually impractical because one needs to obtain the load average for every machine, and there may be many machines. For example, on Purdue University's Engineering Computer Network, there are over 500 machines that could be available to the system. Thus, AHS allows the user to explicitly issue a command to update the load average database information.

4.2. Minimizing Expected Program Execution Time

As discussed above, the machine load and execution time of each pseudocode operation is available for each of the potential targets. When a MIMDC program is compiled, a cost formula for that program is also computed. The cost formula is simply a weighted sum; there is a constant factor for each instruction type. The weighting is determined by a version of the compiler that does not generate code, but simply records expected execution counts for each type of operation. Currently, the rules by which execution counts are computed for MIMDC are very simple:

- The expected execution count at the beginning of `main()` (or any other function) is assumed to be 1.0.
- In an `if` construct, the then clause frequency is assumed to be 51% (and 49% for the `else` clause, if one is present). Thus, the code within a then clause has an execution count that is 0.51 times the execution count outside the `if` statement.

  Given no other information, 50% is the best estimate for the probability of each possible value of a binary condition. The choice of 51% for the true branch probability

approximates the 50% estimate, but slightly favors the then clause so that the system will be consistent in selecting a target machine when one machine is faster for the then code and another is faster for the else code.

- The body of any looping construct is assumed to be executed 100.0 times, thus, within the loop the execution count is 100.0 times that of code outside the loop. Note that the conditional test in some loops is executed one more time than the body code; in such a case. the condition test code's execution count is multiplied by 101.0 instead of 100.0.

  The value 100.0 may seem rather arbitrary, but it actually reflects a very simple observation. If we assume that a loop body is executed too many times, we don't favor the straight-line code as much as we should, and the program will execute slower... but not much slower. On the other hand, if we assume that the loop is executed too few times, the code in the loop runs slower than it should. and the error is multiplied by the number of times the loop actually executes. Thus, it makes sense to err on the high side; 100.0 is a number that is high enough to strongly favor code in loops, but not entirely ignore code outside loops.

Thus, a program is reduced into a table giving estimated expected execution counts for each of the different operations.

Wren the program is to be run, the user specifies the number of PEs desired. Given that, the expected execution counts, and the execution model and machine database, the fastest target (or set of distributed targets) is selected by:

Target Selection Algorithm

1. **Pick the best single machine target.** For each target that has a specified maximum execution width ≥ the number of PEs requested or that uses the pipe or shared file execution model:
   1.1 Temporarily adjust the load average by adding the product of the number of PEs requested and the load increment.
   1.2 Compute the sum, over all operations, of the product of the operation time (from the database) and expected execution count (for the given program). Multiply this result by the adjusted load average.
   1.3 If the resulting time estimate is less than the best so far, make this target the new best target.
2. **Pick the best set of distributed targets** For i=1 to the number of PEs requested:
   2.1 Set the time of bestPE to infinity.
   2.2 For each target that has a width of 0 and uses the UDP Socket execution model:
      2.2.1 Temporarily adjust the load average by adding the load increment, since we are about to add one PE process to this target.

2.2.2 Compute the sum, over all operations, of the product of the operation time (from the database) and expected execution count (for the given program). Multiply this result by the adjusted load average.

2.2.3 If the resulting time estimate is less than bestPE, make this target the new bestPE.

2.3 Record that this target was selected as the i^{th} bestPE and make the adjustment to the load average from step 2.2.1 permanent.

3. Since the time for the program is the maximum time for any PE, if the time for last value of bestPE > the value of best, then best is the target and we are done.

4. The sequence of distributed targets selected for bestPE are selected. However, rather than representing this as a list that says where each PE's target is, we convert this into a list of targets that specifies which PEs are assigned to each target.

43. Controlling Program Execution

Having selected a target, or set of distributed targets, the final step is to actually cause the program to execute there. This is done by:

1. When the user "compiles" a MIMDC program, it is not actually compiled, but is analyzed and packaged into a "master shell script." This shell script contains the expected execution counts, as well as the full source of the MIMDC program.

2. The user views the master shell script as an ordinary executable object file, and hence runs it by simply typing its filename with at least one command line argument — the desired number of PEs.

3. In execution, the first thing done by this master shell script is to apply the above algorithm to select the fastest target(s). Once target(s) are selected, the program will run to completion on those target(s); running processes are never migrated.

4. Finally, for each of the selected target(s), the master shell script uses rsh to send and execute a second shell script that contains both the MIMDC program and the sequence of commands needed to compile and execute it for all the PEs assigned to that target. Hence, there is no need to keep track of paths to user object files, or to use NFS mounting across machines. The method used in AHS has the overhead of sending and recompiling the source program every time the program is run, but the MIMDC compilers run very fast, and compile time is nearly always small compared to the runtime of typical supercomputing applications.

Notice that, unlike PVM, there are no daemon processes running on the target(s) when the system is not running user code — in fact, there are no daemons running even when user programs are using the target(s). The process overhead is just the initial shell script that the user executed and the shared memory manager in some execution models (e.g., the UNIX pipes model).

5. Conclusions

This paper has taken a very practical approach to the problem of automatically making efficient use of heterogeneous supercomputing. Rather than making a heroic effort to achieve near peak speeds on a particular machine, our system attempts to invisibly seek out and use whatever hardware will make the user's program execute fastest.

Toward this goal, it is necessary that the programming model, and language, be defined to facilitate automatic porting of programs to a wide range of targets. For each potential type of target, there must be an execution model that can support the programming model. Finally, there must be a procedure by which the performance of a program on each potential target can be predicted, and a method by which the system can automatically select the fastest target and cause the program to be executed there. Each of these aspects is covered both as an abstract problem and as a summary of how these issues have been managed by a prototype system.

The prototype system, AHS, accepts a control-parallel dialect of C called MIMDC. It can automatically select the execution model and machine(s) with the fastest expected execution time for any given program, and invisibly causes the program to execute using that target. Execution models support a massively-parallel supercomputer, individual uniprocessor or multiprocessor UNIX machines, and even groups of networked UNIX systems running as a distributed computer. Little benchmarking of the system has been done, but the execution models for the SIMD MasPar MP-1 (as a MIMD) and for groups of UNIX systems are surprisingly efficient.

As presented, AHS is functional, but not complete. It is intended to be the basis for a more sophisticated system. That system will analyze and schedule individual functions within a program and will support additional programming models and languages. When AHS is mature enough, we intend to distribute it as a full public domain software release.

References

[AhC91] S. Ahmed, N. Carriero, and D. Gelemter, "The Linda Program Builder," Advances in Languages and Compilers for Parallel Processing, edited by A. Nicolau, D. Gelemter, T. Gross, and D. Padua, The MIT Press, 1991, pp. 71-87.

[Bla90] T. Blank, "The MasPar MP-1 Architecture," 35th IEEE Computer Society International Conference (COMPCON), February 1990, pp. 20-24.

[BeS91] T.B. Berg and H.J. Siegel, "Instruction Execution Trade-offs for SIMD vs. MIMD vs. Mixed Mode Parallelism," 5th International Parallel Processing Symposium, April 1991, pp. 301-308.

[DiC92] H.G. Dietz and W.E. Cohen, "A Control-Parallel Programming Model Implemented On SIMD Hardware," in Proceedings of the *Fifth Workshop on Programming Languages and Compilers for Parallel Computing,* August 1992.

[Die92] H.G. Dietz, "Common Subexpression Induction," in Proceedings of the *1992 International Conference on Parallel Processing,* Saint Charles; Illinois, August 1992, vol. 2, pp. 174-182.

[GrS92] B.K. Grant and A. Skjellum, "The PVM Systems: An In-Depth Analysis And Documenting Study," in the 1992 MPCI Yearly Report: Harnessing the Killer Micros, Lawrence Livermore National Laboratory, August 1992, pp. 247-266.

[HiK92] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C-W Tseng, "An Overview Of The Fortran-D Programming System," Languages and Compilers for Parallel Computing, edited by U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Springer-Verlag, 1992, pp. 18-34.

[JoB89] H.F. Jordon, M.S. Benten, G. Alaghband, and R. Jakob, "The Force: A Highly Portable Parallel Programming Language," International Conference on Parallel Processing, August 1989, vol. 2, pp. 112-117.

[LiM90] M. S. Littman and C. D. Metcalf, An Exploration of Asynchronous Data-Purallelism, Technical Report, Yale University, July 1990.

[Mas91] MasPar Computer Corporation, *MasPar* Programming Language (ANSI C compatible MPL) Reference Manual, *Software* Version 2.2, Document Number 9302-0001, Sunnyvale, California, November 1991.

[NiT90] M. Nilsson and H. Tanaka, "MIMD Execution by SIMD Computers," Journal of Information Processing, Information Processing Society of Japan, vol. 13, no. 1, 1990, pp. 58-61.

[PaD92] T.J. Parr, H.G. Dietz, and W.E. Cohen, "PCCTS Reference Manual (version 1.00)," *ACM SIGPLAN* Notices, Feb. 1992, pp. 88-165.

[Phi89] M.J. Phillip, "Unification of Synchmnous and Asynchronous Models for Parallel Programming Languages" Master's Thesis, School of Electrical Engineering, Purdue University, West Lafayette, Indiana, June 1989.

[QuH91] M. Quinn, P. Hatcher, and B. Seevers, "Implementing a Data Parallel Language on a Tightly Coupled Multiprocessor." Advances in Languages and Compilers for Parallel Processing, edited by A. Nicolau, D. Gelemter, T. Gmss, and D. Padua, The MIT Press, Cambridge, Massachusetts, 1991, pp. 385-401.

[SiN90] H.J. Siegel, W.G. Nation, and M.D. Allemang, "The Organization of the PASM Reconfigurable Parallel Processing System," Ohio State Parallel Computing Workshop, Computer and Information Science Department, Ohio State University, Ohio, March 1990. pp. 1-12.

[Sto84] H.S. Stone, "Database Applications of the Fetch-And-Add Instruction," IEEE Transactions on Computers, July 1984, pp. 604-612.

[Thi90] Thinking Machines Corporation, Connection Machine Model CM-2 Technical Summary, version 6.0, Cambridge, Massachusetts, November 1990.

[WiH91] P.A. Wilsey, D.A. Hensgen, C.E. Slusher, N.B. Abu-Ghazaleh, and D.Y. Hollinden, "Exploiting SIMD Computers for Mutant Program Execution," Technical Report No. TR 133-11-91, Department of Electrical and Computer Engineering, University of Cincinnati, Cincinnati, Ohio, November 1991.