The MHETA Execution Model for Heterogeneous Clusters[*]

Mario Nakazawa
Dept. of Mathematics and Computer Science
Berea College
Berea, Kentucky 40404
mario_nakazawa@berea.edu

David K. Lowenthal and Wendou Zhou
Dept. of Computer Science
University of Georgia
Athens, GA 30602-7404
{dkl,zhou}@cs.uga.edu

ABSTRACT

The availability of inexpensive "off the shelf" machines increases the likelihood that parallel programs run on heterogeneous clusters of machines. These programs are increasingly likely to be *out of core*, meaning that portions of their datasets must be stored on disk during program execution. This results in significant, per-iteration, I/O cost.

This paper describes an execution model, called MHETA, which is the key component to finding an effective data distribution on heterogeneous clusters. MHETA takes into account computation, communication, and I/O costs of iterative scientific applications. MHETA uses automatically extracted information from a single iteration to predict the execution time of the remaining iterations.

Results show that MHETA predicts with on average 98% accuracy the execution time of several scientific benchmarks (with and without prefetching) and one full-scale scientific program that utilize pipelined and other communication. MHETA is thus an effective tool when searching for the most effective distribution on a heterogeneous cluster.

Keywords

Data Distribution, I/O, Modeling Parallel Execution

1. INTRODUCTION

Parallel computing can achieve dramatic improvements in the performance of applications. Due to the increasing availability of a variety of "off-the-shelf" machines, an important platform is a cluster of workstations with potentially different processor speeds, memory capacities and I/O latencies. Designing a program to run efficiently in this environment has many problems that can be overwhelming to computational scientists. These difficulties include distributing data, performing I/O, and handling communication that itself may involve I/O.

[*]This research was funded in part by NSF grants CCF-0429643 and CCF-0234285.

Many researchers have noted that a key obstacle to efficient parallel programs is finding an efficient data distribution [14]. The problem is challenging on a heterogeneous cluster for two primary reasons. First, different processor speeds make balancing the load difficult. Second, I/O will more likely need to be considered due to differing memory capacities between nodes. While the first problem has previously been researched, the second has not been widely studied. This is despite I/O becoming an important factor—scientific programmers commonly design large-scale simulations to explain complex phenomena, requiring larger (and growing) data sets that may not fit into main memory.

Our goal is to develop a runtime system that will automatically determine the best data distribution for iterative scientific applications on this architecture. The key component of this system is a computational model we developed called MHETA (**M**odel of **HET**erogeneous **A**rchitectures). This model takes as input a data distribution and predicts the *execution time* for an application running with that distribution. A separate component of the runtime system uses MHETA to evaluate all candidate distributions as part of a search algorithm. We call this algorithm GBS, for Generalized Binary Search; a separate paper describes the design, implementation, and performance of GBS [26].

This paper details how we implemented MHETA as a general purpose model that consists of a system of parameterized equations. The structure of the application determines how these equations are put together, and instrumented measurements of costs incurred during one iteration of the program are used to set the parameter values. This strategy provides the flexibility to model any application running on a given heterogeneous cluster and to accurately predict the execution of the remaining iterations given possible alternative distributions. Instrumenting programs with manually inserted timers and other functions is tedious and error-prone. We automated a portion of this process and made it transparent to the programmer through a combination of micro-benchmarks and a data collection tool called MPI-Jack [1].

We tested MHETA on a range of benchmarks: Conjugate Gradient from the NAS benchmark suite [3], Jacobi iteration, and a pipelining benchmark based on RNA pseudoknots [5]. We also used one full-scale application, the Lanzcos iterative method for solving linear systems of equations. Our results show that this model accurately predicts execution times for both benchmarks with a simple structure like Jacobi, complex pipelined structure in RNA, as well as a full-scale application like Lanzcos. MHETA is on average $96\% - 98\%$ accurate in predicting execution times for all applications without prefetching, and 98% on Jacobi iteration with prefetching. MHETA is thus an effective evaluation function when searching for an efficient distribution. This is important, as the difference between execution times (in our experiments) given the best and worst distributions was as much as a factor of 4, and which distribution results in poor performance cannot in general be known statically. Because MHETA runs efficiently (about 5.4*ms* per distribution), it can be used on the fly.

The rest of this paper is organized as follows. Sections 2 and 3 cover related work and our framework, respectively. They are followed by Section 4, which provides details of MHETA. Section 5 discusses performance and Section 6 summarizes and describes future work.

2. RELATED WORK

There is a significant amount of related work to this paper, which we broadly divide into four categories: modeling, data distribution, out-of-core parallel programming, and heterogeneous computing. This section discusses each of these in turn and compares them to our work.

2.1 Modeling

Several have developed models of parallel programs. The most popular models are the Bulk Synchronous Parallel model (BSP) [38] and LogP [8]. The BSP model uses *supersteps*, allowing limited communication in each step. LogP models parallel programs using only latency, overhead, gap, and number of processors. The two-level memory model [40] used in TPIE models data moving from multiple disks to memory.

Unlike previous computational models, MHETA incorporates I/O because our target applications are out of core. Our model also measures actual computation times, whereas the LogP model assumes there is a constant amount of work, and we also support finer-grained parallelism than BSP. The main focus of TPIE is to design efficient parallel, high-level coordination of data movement between disk and memory; we use MHETA to estimate execution times with the ultimate goal to find efficient data distributions.

2.2 Data Distribution

One way to distribute data is to provide language annotations and allow the programmer to choose the distribution using application-specific knowledge. This is the approach taken by HPF [17], which was motivated by many others' work (e.g., [15]). Compiler techniques to distribute (and possibly redistribute) data have also been studied extensively (e.g., [2, 14, 32, 20, 11, 30]). The basic idea behind compiler-based systems is to analyze the source code to determine the communication pattern and then choose a `BLOCK`- or `CYCLIC`-based distribution that balances the load. Another key idea is tiling, an optimization to rewrite loops to keep data in cache as long as possible [12, 16]. Approaches employing a run-time system, such as CHAOS [18], AppLeS [35], SUIF-Adapt [23], and CRAUL [33] can use run-time information to find an efficient data distribution. This is especially effective in cases where workload and communication characteristics of a program change at run time.

The work described above on the data distribution problem addresses the data distribution problem, not the heterogeneous data distribution problem; the latter problem allows processor speed, memory size, and I/O speed to all differ. This requires a trade-off between balancing load, minimizing communication, *and* minimizing I/O. The tiling optimizations cited above are compiler transformations that seek a particular tile shape to minimize execution time; our work is to find a data distribution problem at runtime. Further, systems like AppLeS always avoid using a processor if its memory is relatively small, even if that processor could perform useful work.

2.3 Out-of-core Parallel Programming

Many researchers have studied out-of-core parallel applications, where data structures must primarily reside on disk. Generally, these projects fall into the following categories: using virtual memory, high-level interfaces, compiler and run-time analysis to generate efficient out-of-core programs, prefetching support, and collective I/O. Out-of-core parallel programs can be written just as in-core ones, using the virtual memory system for all I/O [6]. TPIE is a high-level libraries used to coordinate data movement between disk and memory efficiently, and it is also used to design efficient algorithms to manipulate data on disk. Compiling out-of-core codes is described in, for example, [19, 4, 29, 36], one basic idea is to extend the data parallel model to out-of-core programs.

One technique to avoid disk latency is prefetching, which has been studied extensively. Mowry et al. [24] use a virtual memory model augmented with prefetching inserted via the compiler. Others monitor access patterns in the operating system and use the results to prefetch data [22, 13]. Finally, in [31], applications advise the OS of their future access patterns. One run-time approach, called SMARTS, schedules work dynamically to help reduce disk accesses [37]. Several have worked on file systems that support parallel access patterns through *collective I/O*, including Panda [34], PASSION [7], and Disk-directed I/O [21]. The idea behind collective I/O is to avoid several small disk requests due to a different layout of in-memory data than on-disk data.

Our work is orthogonal to techniques to improve out-of-core parallel programming. We start from an efficient user program, which can be developed by hand or by the above models.

2.4 Heterogeneous Computing

Several researchers have studied various parallel computing problems on heterogeneous clusters. These problems include gang scheduling, compiling, and message passing libraries. Gang scheduling is a scheme whereby multiple parallel jobs execute concurrently, but the *entire* machine is given to one job at a time (e.g., [9]). Nikolopoulos et al. [28] also manage the trade-off between paging and gang scheduling on a multiprogrammed cluster. There has also been work done in compiling for heterogeneous machines [41]. Finally, Grid MPI is an extension of MPI that enables an MPI program to use grid services [10].

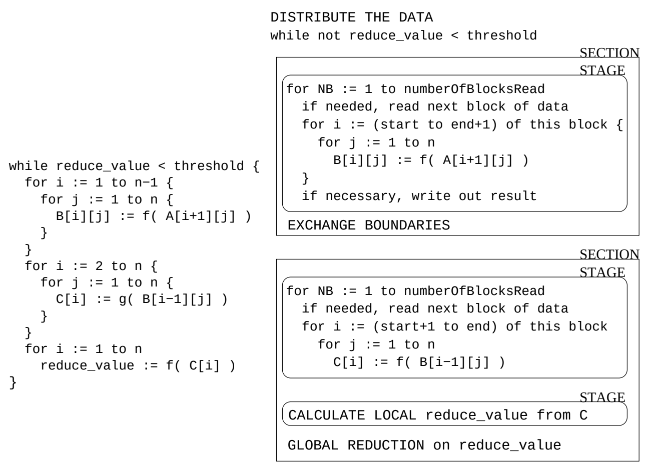
Gang scheduling is expensive to implement and is most often implemented on supercomputers. Also, [28] involves modifying the OS. Our work is focused on the runtime system.

3. FRAMEWORK

This section first discusses our computational model, followed by the assumed execution environment.

3.1 Computational Model

We use a computational model that is similar to BSP [38]. The programs we support are iterative scientific applications. We define a *parallel section* as code in between either a nearest neighbor or



```
while reduce_value < threshold {
  for i := 1 to n-1 {
    for j := 1 to n {
      B[i][j] := f( A[i+1][j] )
    }
  }
  for i := 2 to n {
    for j := 1 to n {
      C[i] := g( B[i-1][j] )
    }
  }
  for i := 1 to n
    reduce_value := f( C[i] )
}
```

```
DISTRIBUTE THE DATA
while not reduce_value < threshold
                                                SECTION
                                                STAGE
for NB := 1 to numberOfBlocksRead
  if needed, read next block of data
  for i := (start to end+1) of this block {
    for j := 1 to n
      B[i][j] := f( A[i+1][j] )
  }
  if necessary, write out result
EXCHANGE BOUNDARIES
                                                SECTION
                                                STAGE
for NB := 1 to numberOfBlocksRead
  if needed, read next block of data
  for i := (start+1 to end) of this block
    for j := 1 to n
      C[i] := f( B[i-1][j] )
                                                STAGE
CALCULATE LOCAL reduce_value from C
GLOBAL REDUCTION on reduce_value
```

Figure 1: A sample conversion of a sequential program into a parallel one with sections and stages. Note the dependencies of array B from one loop to another necessitate the exchange of boundary information; array A is read only, so the necessary rows can be replicated.

reduction communication pattern, at which point a node can send at most one message to another node. A parallel section consists of a set of one or more *tiles*; pipelined applications have multiple tiles per parallel section. A tile is further divided into one or more *stages*, which are bounded explicitly by an outermost loop over a multidimensional array or implicitly by the end of a tile. MHETA can support the case where iterations take a nonuniform amount of time; however, in this paper we discuss only those whose time is uniform—which covers many, if not most, applications[1]. We assume the applications make explicit calls to read and write from disk and are constructed so that data passes through memory as few times as possible. However, note that our model uses stages to handle the case where there need be multiple disk reads and possibly writes in a parallel section. An example of parallel sections and stages are illustrated in Figure 1. The stages measure the duration to perform the computation and I/O in a program and a parallel section sums these before including the communication cost.

We adopt the terminology for out-of-core applications from [4]. An application is considered to be *in core* when all disk accesses for primary data sets are only compulsory, and the data sets remain in its local memory for the duration of the program.We call the subset of the input and working data that is stored locally on a machine its Local Array (LA). An application is necessarily *out of core* if the dataset is larger than total aggregate memory, if a particular data distribution results in at least one node whose LA cannot fit into its memory. If either is the case, the LA is called the Out-of-Core Local Array (OCLA), and the subset that fits into memory is called the In-Core Local Array (ICLA). A node whose local array is too large needs to read and process the entire OCLA in ICLA-sized pieces. An in-core application incurs a single disk read for each of its local arrays, whereas an out-of-core application incurs multiple reads (and possibly writes) for each local array. Note that the problem of determining which arrays are out of core is orthogonal to our research—we currently use a simple heuristic and are primarily interested in creating a model that finds an effective partitioning of data and computation to each node.

[1]Primarily, this simplifies the implementation, as per iteration data needs to be collected for nonuniform iterations.

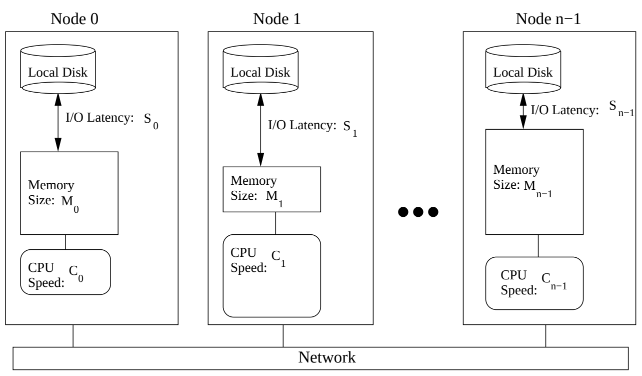


Figure 2: The composition of the heterogeneous architecture emulated for our experiments. Note that the memory capacities, I/O speeds, and processor speeds between any two machines can differ. The sizes of the boxes in the figure for memory and CPUs represent relative storage capacity and CPU power.

We assume that a one-dimensional data distribution is used, and the data is divided into variable-sized blocks (called `GEN_BLOCK` in HPF [17]); each node receives its block and stores it on its local disk. We use the owner computes [15] and the Local Placement [4] rules, in which nodes update data elements that reside in their local disk, but can reference other elements.

3.2 Execution Environment

Figure 2 displays the heterogeneous cluster of workstations we emulated to test our model. The cluster consists of n nodes, numbered 0 to $n-1$, each of which has a local disk. Processor speeds can differ between nodes, so we define relative CPU power, for each node, based on processor speed. The memory capacity is how much physical memory is available for the application, and the I/O latency is the duration required to read and write data from disk. Note we assume a commodity cluster, as opposed to a RAID system or global disk used by all the processors—but our basic model could be extended to support either. At present, we assume a dedicated computing environment—this is a problem we will consider in the future.

4. MODELING PROGRAM EXECUTION

We implemented MHETA as a series of parameterized equations for each parallel section and stage. Some parameter values, such as communication start-up cost, are derived from microbenchmarks. We additionally take measurements during an instrumented iteration (one out of many) of the program. These measurements capture the costs to perform (1) computation, (2) I/O and (3) communication that are specific to the behavior of the application and the characteristics of the architecture. MHETA calculates the total execution time of a single iteration of the application from these instrumented values along with an *arbitrary* data distribution. We first describe how some of this instrumentation is done in Section 4.1, followed by details of MHETA's construction in Section 4.2.

4.1 Extracting MHETA Components

MHETA requires knowledge of a program's structure and runtime-specific information—such as the costs to perform I/O, communication and computation—to calculate its execution time. Some information is extracted via static analysis and uses microbenchmarks, while other information requires an instrumented run of a

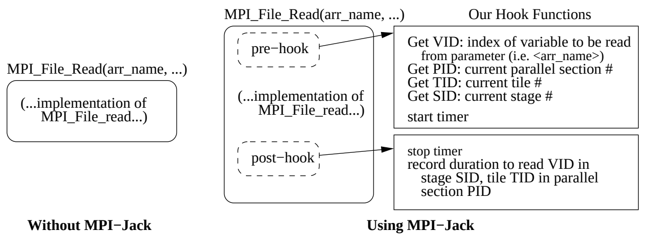

MPI_File_Read(arr_name, ...)
(...implementation of MPI_File_read...)
Without MPI−Jack
MPI_File_Read(arr_name, ...)
pre−hook
(...implementation of MPI_File_read...)
post−hook
Using MPI−Jack
Our Hook Functions
Get VID: index of variable to be read from parameter (i.e. <arr_name>)
Get PID: current parallel section #
Get TID: current tile #
Get SID: current stage #
start timer
stop timer
record duration to read VID in stage SID, tile TID in parallel section PID

Figure 3: A graphical example of how MPI-Jack is used to extract timing information for an MPI_File_read call. Invoking a MPI-Jack function essentially calls the pre and post hook functions before calling the actual MPI function. On the left, these hook functions are undefined and no timing information is recorded. The right implementation has code defined to extract the variable ID involved in the I/O and timers to record the latency to read the array.

single iteration of the application. We currently analyze the application source code manually to determine the number and relationship between the parallel sections, tiles, and stages in the program as well as which variables they use. We store this information in a file read by MHETA. We use microbenchmarks to measure some basic communication costs, such as send and receive overheads and send latency per byte between nodes. We assume these values are relatively constant in our dedicated environment and store and reference them when needed.

The instrumented iteration outputs computation and I/O costs as well as the participants in communication at the beginning or end of each parallel section. Section 4.1.1 details how we automated some of the measurement of the costs of computation and I/O. Some discussion of extracting the identities of nodes (*nIDs*) during a communication pattern is detailed in Section 4.1.2.

4.1.1 Measuring Runtime Costs

This section starts with details of how we measure synchronous and then asynchronous I/O costs. The difficulty in measuring computation durations is discussed next, as this measurement requires knowledge of when a stage begins and ends. This is hard to detect in general. MHETA targets applications that use explicit I/O, so these programs have explicit function calls to read and write data. For program that use MPI function calls to perform I/O, we use our MPI-Jack tool [1], an interface that exploits PMPI, the profiling layer of MPI. MPI-Jack enables a user transparently to intercept any MPI call and execute arbitrary code before and/or after an intercepted call. These are called *pre* and *post* hooks. An example of an MPI read operation is shown in Figure 3. The seek overheads for reading and writing (O_r and O_w) to local disk are the same regardless of the variable involved, so they are measured and output as node-specific data. The corresponding latencies ($T_r(m)$ and $T_w(m)$), in contrast, are specific to the variable m. The instrumenting code extracts the variable ID from the function's parameters to associate the latencies with the corresponding variable. The runtime system computes the latencies ($r(m)$ and $w(m)$) for a single element of m and stores them and the overhead costs into an internal MHETA file.

Note that given a data distribution for the instrumented execution, the node specific portion of m may in fact be in core, and no I/O occurs. The instrumenting code thus would not have any I/O cost information for m. However, if a different and more efficient distribution is discovered, it may assign more of m to the node that it

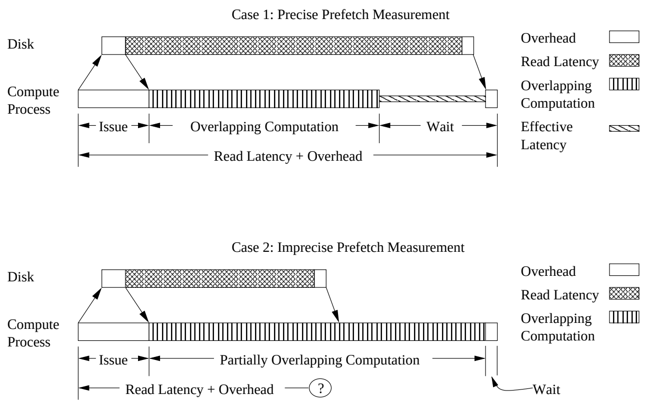


Figure 4: Problems when trying to instrument prefetching code. Case 1 shows how instrumenting prefetching can be precise and accessible from our wrapper functions. We can measure T_A from when the prefetch issue returns right before the wait function is invoked, and O_r + $T_r(m)$ from the beginning of the prefetch issue to the end of the wait function. If, however, $T_A > T_r(m)$ as shown in case 2, we cannot measure $T_r(m)$ + O_r correctly via timers in the wrapper functions.

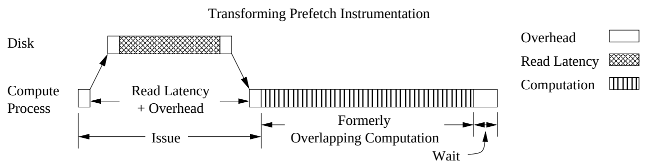


Figure 5: Instrumenting prefetching by forcing issues to be blocking reads, and transforming "wait" functions into no-ops. The read latency and overhead can now be precisely measured, as well as the computation that was formerly overlapping the latency.

can store in memory, thereby causing I/O to occur. MHETA must therefore have I/O costs for m to address this situation; *all* nodes are forced to perform I/O during the instrumented execution for any distributed variables.

Unlike synchronous I/O, asynchronous I/O allows overlap of computation and I/O costs, so predicting the cost of prefetching requires a combination of $T_r(m)$ and the duration of the overlapping computation (T_A). Figure 4 shows that T_A is the time from the prefetch issue until the entry to the wait function, and this duration can be measured using timers. The difficulty arises when we try to measure the duration of $T_r(m)$. If $T_A \leq T_r(m)$, $T_r(m)$ is measured from the start of the prefetch issue to when the wait function returns. Otherwise, there are no timers the runtime system can use to determine when the I/O operations finish.

Our approach to solve this problem cleanly divides the operations by forcing (1) all prefetch issues to be the same as a *synchronous read* and (2) wait functions to be no-ops, as shown in Figure 5. This technique is simpler to implement than polling or inserting timers in the operating system, and yet it accurately measures the durations of T_A and $T_r(m)$. We assume only one iteration is instrumented (out of many), so the higher latencies experienced using this technique are amortized over the remaining iterations that do standard prefetching.

Measuring the computation time for a stage (a separate duration from T_A) depends on knowledge of stage boundaries. The user or preprocessor can insert functions in the source code to indicate when stages begin and end. Only computation and I/O occur within a stage, so by measuring its duration, the computation time can be calculated as the total time for the stage minus time to perform I/O.

4.1.2 *Extracting Communication Participants*

Communication events involve sending and receiving messages to and from various nodes. MHETA requires information about the sender and recipient *nIDs*. The node IDs are obtainable from the parameters of the MPI send and receive calls. Therefore, via the pre hooks in MPI-Jack, we can get this information, making this transparent to the programmer.

4.2 MHETA Implementation

We present here a brief description of the construction of MHETA. We start developing MHETA's structure by considering the execution of a single parallel section on one node, first with single, then multiple stages. We detail calculating both synchronous and asynchronous I/O in a particular stage. Next, we consider the factors involved in communication of pipelined and nearest neighbor message passing when the program is distributed between two nodes. Space limitations prevent us from detailing reduction, or extensions of the equations for communication between more than two nodes. For more details, we direct the reader to [25].

4.2.1 *Single node and parallel section*

First, we describe how MHETA will generate the predicted time to perform only computation and I/O in a single stage in a single parallel section. Let T_Φ be the time spent in computation given W, the amount of work assigned to it by a distribution δ during the instrumented run of the program. For a new distribution δ' that assigns work W', calculating $T_\Phi{}'$ is the original time, multiplied by the ratio of W' and W: $(T_\Phi)' = T_\Phi * (W'/W)$.

Accurately calculating I/O costs involves 1) determining which variable m is out-of-core and 2) computing its ICLA size, $M_\iota(m)$, as well as the number of times the disk is accessed to completely read and write m, $N_L(m)$. MHETA currently uses a simple heuristic to determine if m is out of core for a given δ'. MHETA calculates its $M_\iota(m)$ based on the memory capacity of the node and its OCLA size ($M_\beta(m)$) assigned to the node by δ'. $N_L(m)$ can then be calculated by taking the ceiling of dividing its OCLA size by its ICLA size: $N_L(m) = \lceil M_\beta(m)/M_\iota(m) \rceil$. Any time the node reads data from disk, there is a corresponding write to disk if the results of the computation are stored, such as in our Jacobi application. For the Conjugate Gradient and Lanzcos applications, the array is read-only, and no writes are performed.

The total time spent performing *synchronous* I/O for an out-of-core array m in this stage, $T_\psi(m)$, is the time to perform the reads and writes of its ICLAs (including both overheads and latencies) multiplied by $N_L(m)$:

$$T_\psi(m) = N_L(m) \cdot [O_r + T_r(m) + O_w + T_w(m)] \qquad (1)$$

where $T_r(m) = r(m) \cdot M_\iota(m)$ and $T_w(m) = w(m) \cdot M_\iota(m)$. If m is in core, then $T_\psi(m) = 0$, and if it is read-only, $O_w = T_w(m) = 0$.

Programmers and/or compilers can insert synchronous read calls with prefetching to hide some of $T_r(m)$. A typical technique is to slightly unroll a loop (see Figure 6) over ICLAs that involves a

Original Loop
STAGE

```
for i := 1 to numberOfICLAs {
  Read ICLA(i)
  Process ICLA(i)
  if necessary, write ICLA(i)
}
```

Unrolled Loop
STAGE

```
Read ICLA(1)
for i := 2 to last {
  Prefetch ICLA(i)
  Process ICLA(i-1)
  Wait for ICLA(i)
  if necessary, write ICLA(i-1)
}
Process ICLA(last)
if necessary, write ICLA(last)
```

Figure 6: A sample transformation of a loop over ICLAs to enable prefetching. The stage boundary of course gets extended as necessary.

read, followed by computation on that ICLA, and finally a possible write-back of the results to disk. The program can now prefetch the ICLA for the $i+1^{th}$ iteration while performing computation on the ICLA for the i^{th} iteration. The first ICLA read incurs the full latency, whereas the remaining $N_L(m) - 1$ read latencies can be mitigated using prefetching. The *effective* read latency, $T_s(m)$, is calculated as $T_r(m)$ minus T_A, the time the node spends computing between the prefetch issue and when the data is needed. If the overlap computation is greater than or equal to the read latency, $T_A \geq T_r(m)$, it has been effectively masked: $T_s(m) = 0$. Equation 1 adjusted to take into account prefetching is:

$$T_\psi(m) = \begin{array}{l} [N_L(m) \cdot (O_r + O_A + O_w + T_w(m))] \\ +T_r(m) + [(N_L(m) - 1) \cdot T_s(m)] \end{array} \qquad (2)$$

Prefetching can be more expensive than regular synchronous reads, as the extra overhead (O_A) is incurred regardless of whether the attempt was successful. Note that with no prefetching, Equation 2 reduces to Equation 1 because $T_s(m) = T_r(m)$ and $O_A = 0$.

The total amount of time spent in computation and I/O for this stage (T_γ) is therefore T_Φ (the time for computation) plus I/O costs: $T_\gamma = T_\Phi + \left[\sum_{m \in \beta} T_\psi(m)\right]$. In a single parallel section with N_γ stages, the time a node spends in the parallel section (T_τ) performing computation and I/O is: $T_\tau = \sum_{k=1}^{N_\gamma} T_\gamma(k)$.

4.2.2 *Extension to nodes P_i and P_j*

When considering multiple nodes, MHETA needs to include communication costs. Given a message q on node P_i, the communication cost for a parallel section, $T_\Upsilon(i)$, is calculated as a combination of three components:

- $O_\rightarrow(q)$ - the overhead incurred when sending,
- $T_w(i, q)$ - the time P_i spends when potentially waiting for a message, and
- $O_\leftarrow(q)$ - the overhead when a node processes the incoming message.

An example of the relationship between these parts is shown in Figure 7. The value $O_\rightarrow(q)$ is a combination of two costs. There is always the fixed overhead to prepare and actually copy the message into a system buffer. If the message is generated from an out-of-core array, the node needs to read it from disk; the same modeling is done here as described for I/O above. $O_\rightarrow(q)$ is thus the sum of the fixed overhead plus the time to read the message from disk, if necessary.

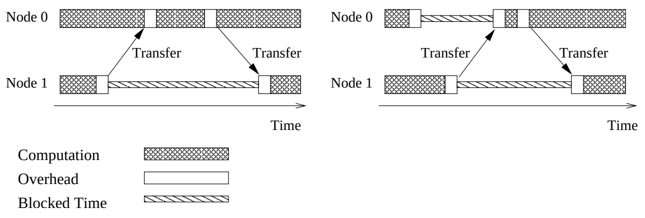


Figure 7: A graphical representation of possible blocked scenarios. The left figure shows when node 1 is blocked waiting for node 0 to send its message, and the right shows the situation when node 0 waits for node 1's message, and node 1 waits for node 0's response.

We now detail how MHETA computes the value of $T_w(i, q)$, first for nearest neighbor and then for pipelined communication. Nodes participating in non-pipelined applications generally start to block *after* performing its stages. Thus, the time P_1 spends waiting for message q to arrive from P_0 is non-zero if it finishes its stages before q arrives:

$$T_w(1, q) = \max \left\{ \begin{array}{l} 0, \\ T_\tau(0) + O_\rightarrow(q) + T_{0\rightarrow 1}(q) \\ \quad -(T_\tau(1) + O_\rightarrow(q)) \end{array} \right\} \quad (3)$$

Note that Equation 3 is symmetric between P_0 and P_1 in this case. We now distinguish T_τ between nodes because a node's blocked time depends on the execution of the sending node.

Pipelined applications have the property that there are potentially many tiles in a parallel section (i.e. the number of tiles, $N_T^i \geq 1$), and nodes start to wait *before* they execute the stages of each tile. We now expand $T_w(0, q, j)$ to distinguish between tiles. Assuming that the pipeline starts from P_0, P_0 does not block at all ($T_w(0, q, j) = 0$ for all tiles). However, P_1 blocks at the start of its j^{th} tile if q arrives any time after it had finished all $j - 1$ previous tiles. The duration of a tile for P_0 consists of the time to process the stages plus the send overhead ($T_\tau(0, j) + O_\rightarrow(q)$), so q for the j^{th} tile is on route after P_0 completes all j tiles. P_1 first blocks, incurs receive overhead and then spends $T_\tau(1, j)$ time in computation for each tile. Thus the wait time for P_1 at the beginning of its j^{th} tile is the total time before q is on route, plus transfer time over the network, minus the total time for P_1 to have finished $j-1$ tiles:

$$\begin{aligned} T_w(1, q, j) = \quad & \left[\textstyle\sum_{k=0}^{j} T_\tau(0, k) + O^q_\rightarrow 0\right] + T_{0\rightarrow 1}(q) - \\ & \left[\textstyle\sum_{k=0}^{j-1} T_w(1, q, k) + O^q_\leftarrow 0 + T_\tau(1, k)\right] \quad (4) \end{aligned}$$

Note that if $T_w(1, q, j) < 0$, then $T_w(1, q, j) = 0$ because P_1 does not block.

We next use the specific communication pattern to calculate the duration the nodes spend in communication (denoted T_Υ^i). Assuming that both nodes perform their sends before blocking in nearest neighbor communication, the value of T_Υ^i is the sum of the send and receive overheads and the time spent waiting for the message:

$$T_\Upsilon(i) = O_\rightarrow(q) + T_w(i, q) + O_\leftarrow(q) \quad (5)$$

In pipeline communication, P_0 has no receives and P_1 has no sends. $T_\Upsilon(i)$ uses the cost $T_w(i, q, j)$ from Equation 4 and $T_w(0, q, j) = 0$,

$$T_\Upsilon(i) = \sum_{j=0}^{N_T^i} \left[T_w(i, q, j) + \left\{ \begin{array}{ll} O_\leftarrow(q) & \textbf{if } i = 1 \\ O_\rightarrow(q) & \textbf{else} \end{array} \right. \right]$$

The total execution time for node P_i in a parallel section, $T_\sigma(i)$, is from when the parallel section starts until it unblocks from the communication at the end: $T_\sigma(i) = T_\tau(i) + T_\Upsilon(i)$. For details on how reduction is modeled and how $T_\sigma(i)$ is calculated when considering more than two nodes, we direct the reader to [25].

4.2.3 *Overall MHETA model*

The total execution time for a single iteration on node P_i is the sum of all its $N_\sigma(i)$ parallel sections:

$$T_I(i) = \sum_{j=1}^{N_\sigma(i)} (T_\sigma(j))$$

5. PERFORMANCE

We tested MHETA on an emulated heterogeneous architecture with three scientific benchmarks: Jacobi iteration (Jacobi), a pipelining application (RNA) similar to RNA pseudoknots, and Conjugate Gradient (CG) from the NAS benchmark suite. In addition, we experimented with one full-scale application, a Lanzcos iterative method for solving a linear system $Ax = b$, where A is a symmetric, positive definite, $N \times N$ dense matrix, and x and b are column vectors. We chose these applications because of their iterative nature. Finally, we also tested MHETA on Jacobi with prefetching. Our experiments were designed to evaluate the effectiveness of MHETA in two respects. First, we determined for each application how accurately MHETA predicted the actual execution time over a set of data distribution candidates. MHETA is on average more than 97% accurate in predicting execution times for all four programs and at least 98% accurate when taking into account prefetching in Jacobi. Second, we focused on the usefulness of MHETA when used in conjunction with a data distribution selecting algorithm.

Our measurements show that evaluating a single distribution in MHETA takes about 5.4 ms. This efficiency is important because we intend to eventually use it within a new MPI-based runtime system that will choose a distribution during runtime.

Section 5.1 describes the setup of the experiments, followed by Section 5.2, which measures the accuracy of MHETA. Finally, Section 5.3 discusses the importance of choosing an effective data distribution on a heterogeneous cluster and how we intend, in the future, to use MHETA for this purpose.

5.1 Experimental Setup

The underlying architecture we used is a cluster of eight Dell Quad servers running Solaris 2.8. We only make use of one out of four processors per node. We emulated a configurable heterogeneous architecture on top of this cluster so that the environment is as described in Figure 2—each node potentially has a different memory size, relative CPU power and I/O latency. We emulate (1) a slower CPU by forcing the process to do extra work, (2) a smaller main memory by placing a limit on the size of memory that applications can use to store their ICLAs, and (3) differing I/O speeds by artificially increasing or decreasing the ICLA sizes read or written to local disk.

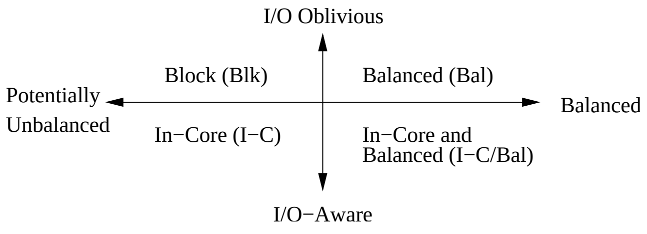


Figure 8: The spectrum of data distributions that we tested.

All applications were compiled with `-O2`, and for all message passing we used LAM-MPI [27]. We currently assume that a program's structural information (the number of parallel sections and stages as well as variable usage information) is available and input to MHETA. When MHETA is integrated with our MPI-based runtime system in the future, structural information will be automatically obtained by static analysis.

Figure 8 shows the distributions we tested, which ranged in two dimensions: (1) how well the load is balanced and (2) to what degree I/O costs are considered. The simplest distribution, "Block" (`Blk`), allocates data evenly across nodes without regard for I/O cost or load balance. A "Balanced" distribution (`Bal`) focuses on balancing the load on the nodes and ignores I/O costs. The "In-Core" (`I-C`) distribution in contrast ignores load and focuses on only minimizing I/O costs. The "In-Core and Balanced" (`I-C/Bal`) distribution first maximizes the number of nodes that have exclusively in-core datasets and then balances the load as much as possible. We start testing the performance of MHETA with `Blk` and progressively generate distributions that move through `I-C`, `I-C/Bal`, `Bal`, and back to `Blk`. For architectures when the relative CPU power of the nodes is identical, a `Blk` distribution already balances the load, so we only vary the distribution between `Blk` and `I-C`. A similar simplification is done when the computing environment has no nodes with memory restrictions (so I/O is not a concern), where we vary the distribution only from `Blk` to `Bal`. The MHETA model extends to two-dimensional data distributions, but such distributions are problematic for run-time data distribution systems because the search space increases greatly. Hence, we focus in this paper on only one-dimensional distributions.

We performed the instrumented iteration of each application using `Blk`. Once these costs were placed into MHETA, we tested MHETA's accuracy by running both the application and MHETA on each of our emulated architectures with identical distributions for 100, 10, 5 and 10 iterations for Jacobi, CG, Lanzcos, and RNA respectively. The number of iterations was chosen to obtain comparable execution times.

We tested MHETA on seventeen and twelve emulated architecture configurations for non-prefetching and prefetching applications, respectively. We give a general summary of its accuracy in Section 5.2. We also focus on four of these configurations as described in Table 1, where we vary the architecture between those where only the relative CPU power varies (*DC*, for "different CPUs"), those where only emulated disk speeds vary (*IO*, for "I/O-induced"), and those where both vary (HY, for "hybrid configurations"). We select two hybrid configurations (*HY1* and *HY2*) to determine the trade-off between balancing load and bringing the data in core. An algorithm searching for a data distribution between `I-C` and `I-C/Bal` can use MHETA to determine which point results in the lowest execution time. A discussions of general analysis and limitations will follow in Section 5.3 and Section 5.4, respectively.

Name	Description
DC	Two nodes have a lower relative CPU power, and two other nodes have higher relative CPU power. The rest are unchanged.
IO	Half of the nodes have high I/O latency and small memories, but all nodes have equal relative CPU power.
HY1	Four nodes have varying relative CPU powers and the other four have low I/O latencies and small memories.
HY2	Four nodes have varying relative CPU power and two nodes have high I/O latencies. The other two have large memories.

Table 1: Four sample configurations of the emulated architectures that are explained in detail.

5.2 Comparison of Predicted and Actual Execution Times

Below, we first present the overall numbers averaged over all the different architectures we emulated. Then, we discuss the four specific architectures described in Table 1.

5.2.1 Overall results

The results of comparing MHETA predicted execution times and the actual run times are shown in Figure 9. These graphs show the maximum, average, and minimum percentage difference between MHETA, where the percentage is computed as the absolute difference divided by the minimum of each application's predicted and actual execution times. The top left graph shows that in all seventeen emulated architectures, MHETA is on average 98% accurate for all four programs without prefetching. The top right graph shows the accuracy of our model for Jacobi with prefetching enabled in a subset of twelve architectures, also at on average 98%. The bottom graphs show the average best-case (RNA) and worst-case (CG) examples. Most errors in MHETA are due to three inherent limitations of our modeling approach, discussed below in Section 5.4. In addition, there can be slight inaccuracies due to perturbations introduced when running the instrumented iteration. (For example, perturbations cause MHETA to have up to a 1% error predicting the running time using a block distribution, even though the instrumented iteration is done using a block distribution.) However, even with unexpected results given these limitations, MHETA still is in general very accurate. These results show MHETA's potential for use by algorithms that find efficient data distributions.

5.2.2 Specific Architectures

Figures 10 and 11 show results using configurations *DC*, *IO*, *HY1* and *HY2*. The graphs show both predicted and actual execution times in seconds. We see in Figure 10 that MHETA performed very well predicting execution times of all four applications in configuration *DC*, and when predicting Jacobi, Lanzcos, and RNA execution times in *IO*. MHETA did slightly overestimate the execution time for these applications right before the `I-C` distribution in *IO* due to better than expected I/O performance of the remaining iterations. This effect diminishes as the distribution approaches `I-C` because a majority of the application execution times are spent performing computation, which are more accurately modeled. MHETA did not fair as well in configuration *IO* in predicting CG's execution time as shown by the two dashed circled points in

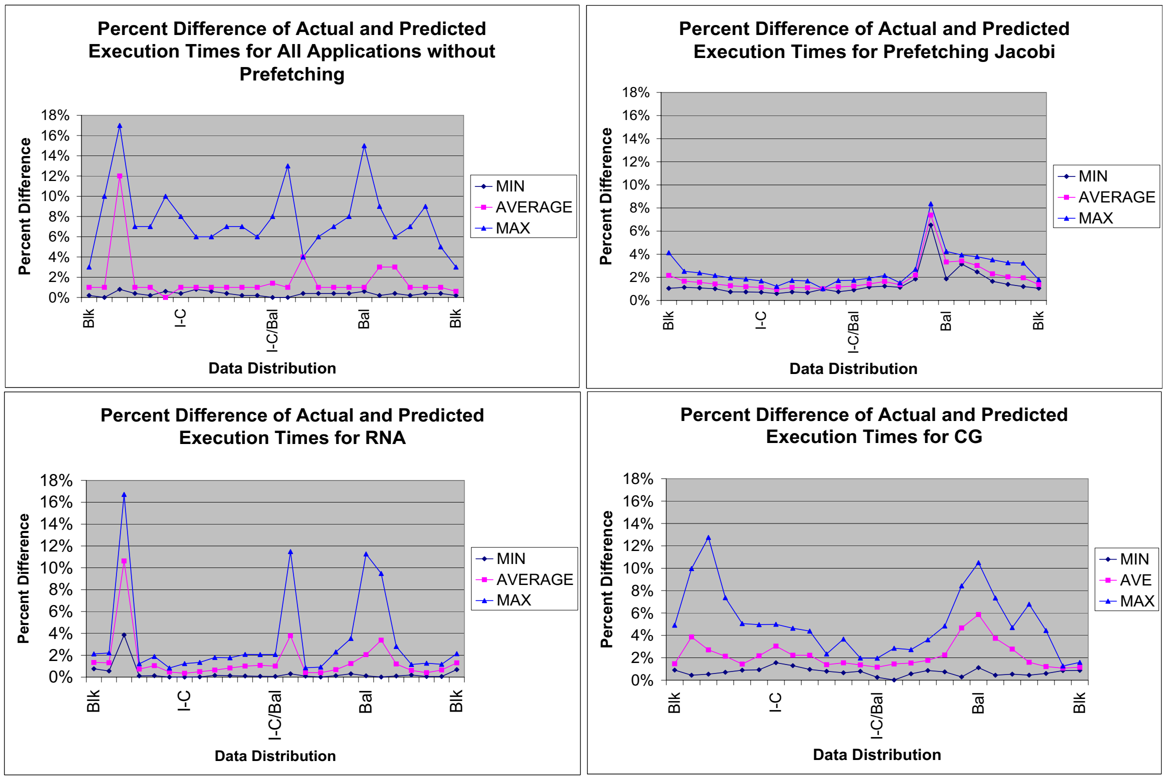


Figure 9: **The minimum, average and maximum percentage difference between the predicted and actual execution times for all applications, and then specifically Jacobi with prefetching, RNA, and then CG. We added lines between the discrete data points for ease of readability.**

the bottom left graph; however, note their difference is only 10%. This inaccuracy results because our algorithm to determine if a variable is out-of-core is too simplistic, and MHETA has difficulties with sparse arrays for CG, as explained in Section 5.4. Finally, MHETA predicted all test applications accurately on *HY1* and *HY2* in Figure 11.

5.3 Analysis

This section discusses first the importance of choosing a data distribution on a heterogeneous cluster, the role of MHETA in the selection process, and finally the limitations of MHETA.

Figures 10 and 11 show that, given the worst data distributions, the execution times for RNA on *DC* and Lanzcos on *HY1* are almost 4 and 3 times as slow, respectively, as when given the best distribution. It may be possible to determine which distribution is best statically on a simpler architecture such as *DC* (if one knew the relative powers of each processor), but in general this determination is nontrivial. For example, due to the low I/O costs and relatively high load imbalances in configuration *HY1*, one would expect the best data distribution to be `Bal`. While this is the case for Lanzcos, the best distribution for Jacobi is instead between `I-C/Bal` and `Bal`. Importantly, this distribution is *significantly* better (28%) than `Bal`. Furthermore, for different architectures that have varied I/O costs and load imbalances on the nodes, it becomes harder to predict which distribution will be best—thus a "guess" may end up far from the best choice, which as stated above can result in a doubling or tripling of execution time. MHETA is able to accurately predict execution times on average 98% of the time, so it can be an effective tool to use when searching for effective distributions. A companion paper describes different data distribution search strategies; MHETA is used as part of four different algorithms (genetic, simulated annealing, generalized binary search, and random) to determine an effective distribution [26].

5.4 Limitations

When MHETA performs poorly, it is in general due to three inherent limitations of our modeling approach. The first limitation is that although the instrumented iteration does in fact capture cache effects, MHETA does not explicitly model the behavior of the memory-cache hierarchy. MHETA therefore occasionally cannot accurately predict when interactions with the cache and main memory given a *different* distribution cause the computation time on a node to grow or shrink. Because out-of-core parallel programs have large datasets that can easily swamp the cache, the likelihood of this error occurring is small.

The second limitation in MHETA is that its algorithm to determine which variables are out of core is not sophisticated, occasionally placing what should be an out-of-core variable in the in-core variable set. MHETA calculates the time spent performing I/O as zero when this error occurs, hence under-predicting execution time. As the data distribution shifts such that more nodes become in core, the effect of this limitation decreases.

Finally, MHETA (along with most data distribution systems) lacks the ability to deal with sparse datasets. Because there is not a simple correlation between number of rows and number of elements per row, resulting in slight load imbalances in CG that our model did not predict. We have found that despite these problems,

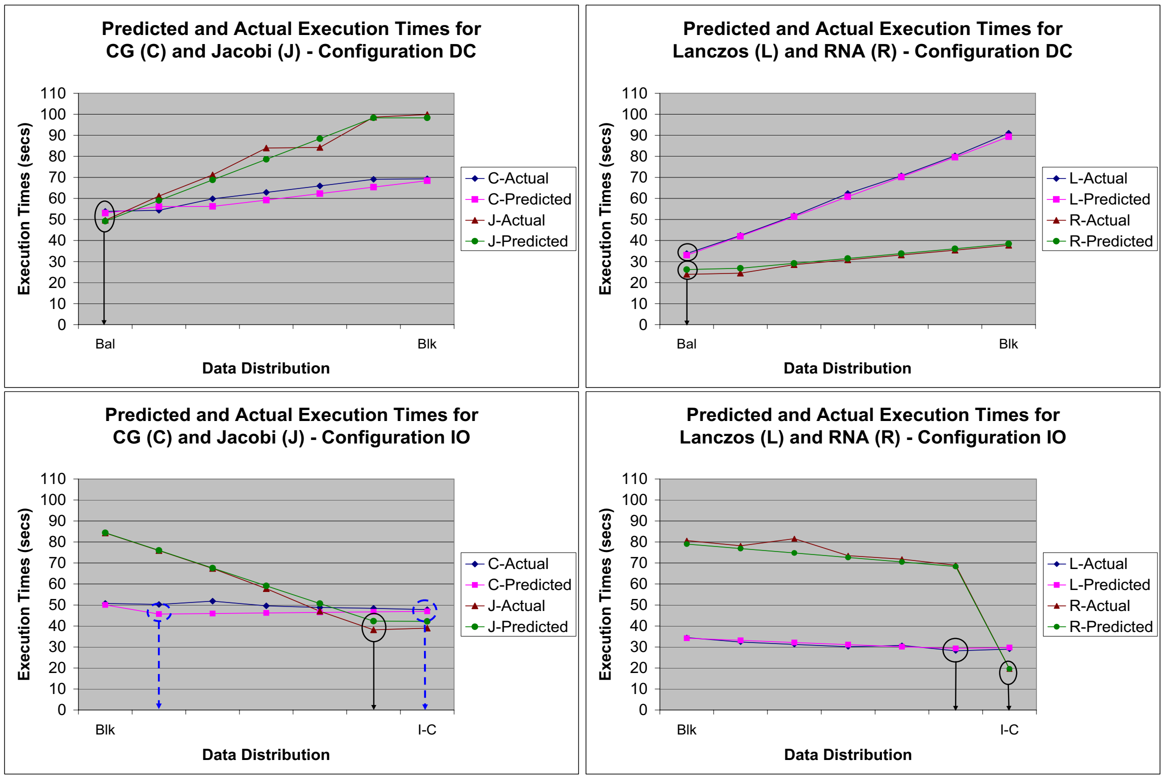


Figure 10: Actual vs. predicted execution times for configurations *DC* (top) and *IO* (bottom) for all four applications. The left graphs show times for Jacobi and CG, while the right show times for Lanczos and RNA. We show the predicted and actual execution times, indicated for example as "L-Predicted" and "L-Actual" respectively for Lanczos. The best distributions in the set investigated for each are circled; where they match, there is only one circle, and where they do not, the predicted time has a dashed circle.

MHETA is very effective in practice.

6. SUMMARY AND FUTURE WORK

This paper has developed a model called MHETA for predicting execution time for out-of-core applications on a heterogeneous cluster. The model takes as input a data distribution and considers the effects of computation, communication and I/O in its prediction. Experiments show that MHETA accurately predicts total execution time; the average difference from the actual execution time for all applications are on average range from 2%, to 5%.

We are currently implementing more applications (including Multigrid) to further increase the types of applications to test MHETA with a wider range of relative communication, computation, and I/O costs. Finally, we are starting development of our new MPI system that will determine the MHETA inputs, use a search algorithm based on MHETA to select a distribution (quickly), and then effect that distribution on the fly. In this way we believe that we can provide an infrastructure for efficient support of out-of-core parallel programs on heterogeneous clusters.

7. REFERENCES

[1] Impie: Interposed message passing interface executor. North Carolina State University website: http://fortknox.csc.ncsu.edu/proj/impie/.

[2] J. Anderson and M. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the SIGPLAN '93 Conference on Program Language Design and Implementation*, pages 112–125, June 1993.

[3] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. RNR-91-002, NASA Ames Research Center, Aug. 1991.

[4] R. Bordawekar, A. Choudhary, K. Kennedy, C. Koebel, and M. Paleczny. A model and compilation strategy for out-of-core data parallel programs. In *Principles and Practice of Parallel Programming*, pages 1–10, July 1995.

[5] L. Cai, R. L. Malmberg, and Y. Wu. Stochastic modeling of rna pseudoknotted structures: A grammatical approach. In *Proceedings of ISMB'03 and Bioinformatics 19(s1)*, pages i66–i73, 2003.

[6] E. Caron, O. Cozette, D. Lazure, and G. Utard. Virtual memory management in data parallel applications. In *HPCN Europe*, pages 1107–1116, 1999.

[7] A. Choudhary, R. Thakur, R. Bordawekar, S. More, and S. Kutipidi. PASSION: Optimized I/O for parallel applications. *IEEE Computer*, 29(6):70–78, June 1996.

[8] D. E. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Principles and Practice of Parallel Programming*, pages 1–12, May 1993.

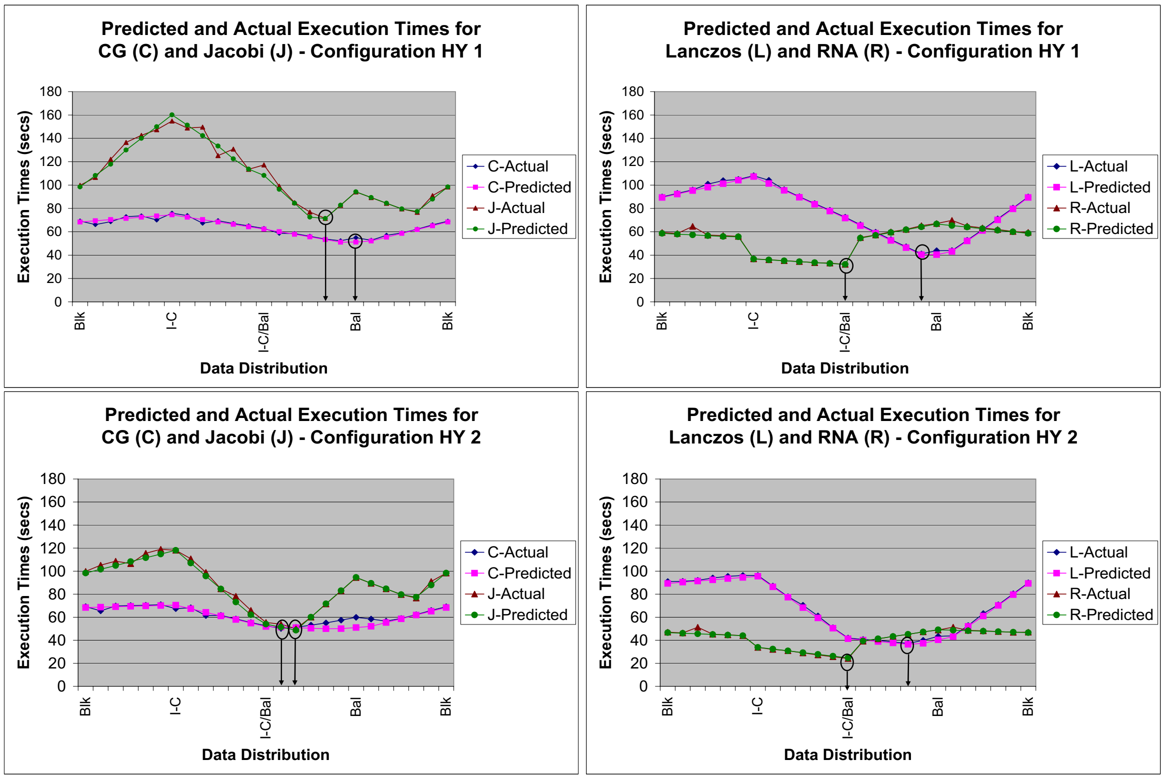


Figure 11: **Actual vs. predicted execution times for configurations *HY1* (top) and *HY2* (bottom) running Jacobi and CG (left) and Lanzcos and RNA (right). The best data distribution is circled.**

[9] D. Feitelson. A survey of scheduling in multiprogrammed parallel systems. Technical Report RC 19790 (87657), IBM T. J. Watson Research Center, October 1994.

[10] I. Foster and N. Karonis. A grid-enabled MPI: Message passing in heterogeneous distributed computing systems. In *Proceedings of SC'98*. ACM Press, 1998.

[11] J. Garcia, E. Ayguade, and J. Labarta. Dynamic data distribution with control flow analysis. In *Supercomputing '96*, Nov. 1996.

[12] G. Goumas, N. Drosinos, M. Athanasaki, and N. Koziris. Automatic parallel code generation for tiled nested loops. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 1412–1419. ACM Press, 2004.

[13] J. Griffioen and R. Appleton. Reducing file system latency using a predictive approach. In *Proceedings of the USENIX Summer 1994 Technical Conference*, pages 197–208, June 1994.

[14] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techiniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, Mar. 1992.

[15] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communication of the ACM*, 35(8):66–80, Aug. 1992.

[16] K. Hogstedt, L. Carter, and J. Ferrante. Selecting tile shape for minimal execution time. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 201–211, 1999.

[17] *High Performance Fortran Language Specification*, Nov. 1994.

[18] Y. Hwang, B. Moon, S. D. Sharma, R. Ponnusamy, R. Das, and J. H. Saltz. Runtime and language support for compiling adaptive irregular programs on distributed-memory machines. *Software—Practice and Experience*, 25(6):597–621, June 1995.

[19] M. Kandemir, J. Ramanujam, and A. Choudhary. Improving the performance of out-of-core computations. In *International Conference on Parallel Processing*, Aug. 1997.

[20] K. Kennedy and U. Kremer. Automatic data layout for distributed-memory machines. *ACM Transactions on Programming Languages and Systems*, 20(4):869–916, 1998.

[21] D. Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74. USENIX Association, 1994.

[22] D. Kotz and C. S. Ellis. Practical prefetching techniques for multiprocessor file systems. *Journal of Distributed and Parallel Databases*, 1(1):33–51, Jan. 1993.

[23] D. G. Morris and D. K. Lowenthal. Accurately computing redistribution cost in distributed shared memory systems. In *Principles and Practice of Parallel Programming*, pages 62–71, June 2001.

[24] T. C. Mowry, A. K. Demke, and O. Krieger. A compiler-inserted I/O prefetching for out-of-core

applications. In *Operating Systems Design and Implementation*, pages 1–17, Oct. 1996.

[25] M. Nakazawa. *I/O Considerations in Heterogeneous Data Distributions*. PHD dissertation, Department of Computer Science, University of Georgia, May 2005.

[26] M. Nakazawa, D. K. Lowenthal, and F. Lowenthal. Efficient data distributions for heterogeneous clusters. Technical report, University of Georgia, available at http://www.cs.uga.edu/ñakazawa/index.html#reports, Feb. 2004.

[27] N. J. Nevin. The performance of LAM 6.0 and MPICH 1.0.12 on a workstation cluster. Technical Report OSC-TR-1996-4, Ohio Supercomputing Center, Columbus, Ohio, 1996.

[28] D. S. Nikolopoulos and C. D. Polychronopoulos. Adaptive scheduling under memory pressure on multiprogrammed clusters. In *Proceedings of CCGrid '02*, Apr. 2002.

[29] M. Paleczny, K. Kennedy, and C. Koebel. Compiler support for out-of-core arrays on data parallel machines. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 110–118, Feb. 1995.

[30] D. J. Palermo and P. Banerjee. Automatic selection of dynamic data partitioning schemes for distributed-memory multicomputers. In *Proceedings of the 8th Workshop on Languages and Compilers for Parallel Computing*, Aug. 1995.

[31] R. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *SOSP*, pages 79–95, Dec. 1995.

[32] J. Ramanujam and A. Narayan. Automatic data mapping and program transformations. In *Workshop on Automatic Data Layout and Performance Prediction*, June 1995.

[33] U. Rencuzogullari and S. Dwarkadas. Dynamic adaptation to available resources for parallel computing in an autonomous network of workstations. In *Principles and Practice of Parallel Programming*, pages 72–81, June 2001.

[34] K. E. Seamons. *Panda: Fast Access to Persistent Arrays Using High Level Interfaces and Server Directed Input/Output*. PhD thesis, University of Illinois at Urbana-Champaign, May 1996.

[35] G. Shao, R. Wolski, and F. Berman. Modeling the cost of redistribution in scheduling. In *Eighth SIAM Conference on Parallel Processing for Scientific Computation*, Mar. 1997.

[36] R. Thakur, A. Choudhary, and G. Fox. Runtime array redistribution in HPF programs. In *Proceedings of Scalable High Performance Computing Conference 94*, pages 309–316, May 1994.

[37] S. Vajracharya, S. Karmesin, P. Beckman, J. Crotinger, A. Malony, S. Shende, R. Oldehoeft, and S. Smith. SMARTS: Exploiting temporal locality and parallelism through vertical execution. In *International Conference on Supercomputing*, pages 302–310, June 1999.

[38] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[39] D. E. Vengroff and J. S. Vitter. I/O efficient scientific computation using TPIE. In *Proc. Goddard Conference on Mass Storage Systems and Technologies*, pages 553–570, 1996.

[40] J. S. Vitter and E. A. M. Shriver. Algorithms for Parallel Memory I: Two-level memories. *Algorithmica*, 12(2/3):110–147, 1994.

[41] G. Weaver, K. McKinley, and C. Weems. Score: A compiler representation for hetergeneous systems. In *The Hetergeneous Computing Workshop*, pages 10–23, Apr. 1996.