An Implementation of the Vesta Parallel File System API on the Galley Parallel File System

Matthew P. Carter
David Kotz

Technical Report PCS-TR98-329

Department of Computer Science
Dartmouth College
Hanover, NH 03755
matt.carter.98@alum.dartmouth.org, dfk@cs.dartmouth.edu

April 23, 1998

Abstract

To demonstrate the flexibility of the Galley parallel file system and to analyze the efficiency and flexibility of the Vesta parallel file system interface, we implemented Vesta's application-programming interface on top of Galley. We implemented the Vesta interface using Galley's file-access methods, whose design arose from extensive testing and characterization of the I/O requirements of scientific applications for high-performance multiprocessors. We used a parallel CPU, parallel I/O, out-of-core matrix-multiplication application to test the Vesta interface in both its ability to specify data access patterns and in its run-time efficiency. In spite of its powerful ability to specify the distribution of regular, non-overlapping data access patterns across disks, we found that the Vesta interface has some significant limitations. We discuss these limitations in detail in the paper, along with the performance results.

1 Introduction

Many parallel file systems have been developed in recent years. There is as yet no agreement about the application programmer's interface (API). One intriguing interface is that of the Vesta parallel file system [CBF93, CF94, CFP+95, CF96], which seems to support regular, non-overlapping matrix distributions well. The interface for the Galley parallel file system [NK96a, NK96b, KN96, NK97] attempts to be a low-level interface that can be a base for other APIs, so it was natural to consider implementing Vesta's API on Galley. (We have also implemented the Panda API [Tho96] and a linear-file model similar to many older parallel file systems [Nie96].)

We implemented the Vesta file system API on top of the Galley file system running on a cluster of Unix workstations. Our implementation of the Vesta interface was straightforward and efficient, because of Galley's flexible design. Although we were able to implement most of the Vesta functions using Galley, some features of Vesta were impossible to implement with Galley. We discuss them further below.

We tested our Vesta interface with several simple Vesta applications supplied by Vesta's original authors, and with our own parallel, out-of-core matrix-multiplication program. We discovered that

Available at URL ftp://ftp.cs.dartmouth.edu/TR/TR98-329.ps.Z

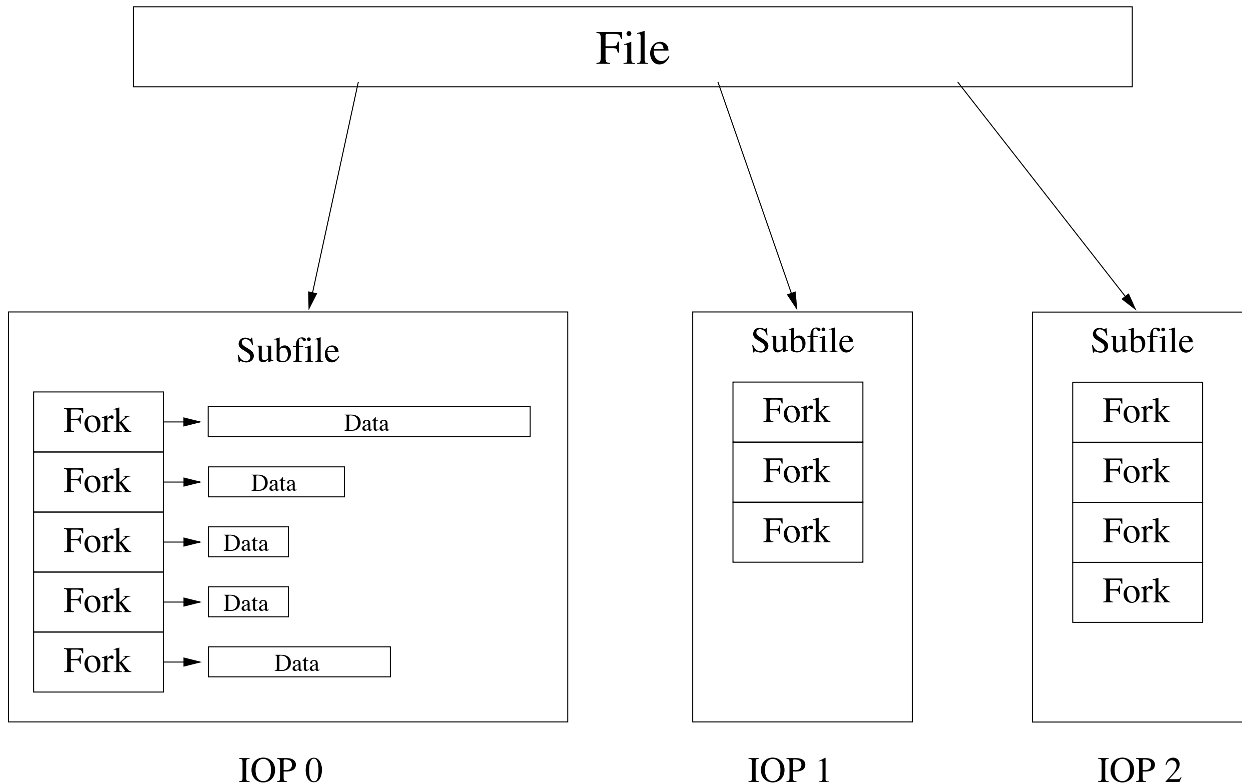


Figure 1: Galley File Layout.

Vesta's file partitioning methods were so regular and inflexible that they were difficult to put to use in solving practical problems (at least, in this case). Even the standard matrix-multiplication algorithm, whose data accesses are very regular, was not cleanly implementable with Vesta.

In the next section, we provide some background on Galley and Vesta. Then in Section 3, we describe our implementation of Vesta on Galley. Sections 4 and 5 describe our experience writing the matrix-multiply program. We conclude in Section 6.

2 Background

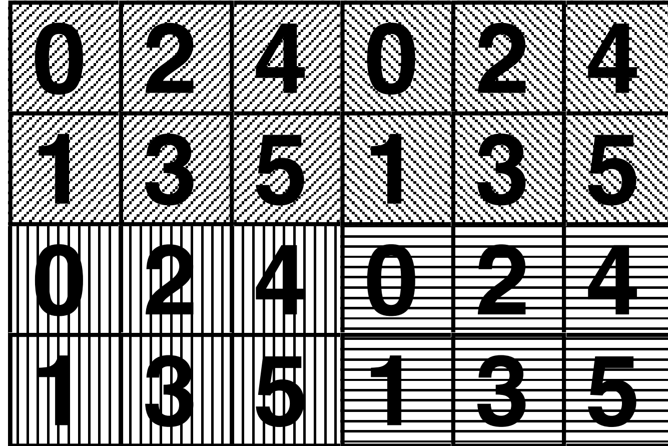
2.1 Galley

The Galley file system organizes files into *subfiles*, each of which resides on its own disk. The number of subfiles is specified by the user at file creation time, and is usually equal to the number of disks in the system. Galley assumes one disk per I/O processor. Each subfile contains one or more *forks*. Forks can be created or deleted after a file has been created (Figure 1). Galley's greatest power comes in its ability to perform a wide variety of complex file-access patterns, such as strided reads and writes and nested-strided reads and writes.[1]

2.2 Vesta

The Vesta file system organizes files into *cells*, each of which resides on a single disk. The number of cells is specified at file-creation time. As such, cells are much like Galley subfiles, although in

[1] See `http://www.cs.dartmouth.edu/~nils/galley.html` for more information on Galley.



Vertical Block Size (Vbs) = 2.
Horizontal Block Size (Hbs) = 3.
Vertical Number (Vn) = 2.
Horizontal Number (Hn) = 2.

Each shading pattern represents a Vesta block.
The numbers identify BSUs within each block.

Figure 2: Sample Vesta Block Group.

Vesta there may be many cells on a given disk. Each disk in the file system has an associated I/O processor that controls the accesses to that disk. Also specified at file-creation time is the *basic striping unit*, or BSU. All Vesta data transfers are specified in terms of BSUs. When a Vesta file is opened for reading or writing, it is logically partitioned into non-overlapping *subfiles* (not the same as Galley subfiles), which may or may not span multiple disks. Each subfile provides a logical address space for file accesses. A column-major rectangular arrangement of Vbs by Hbs BSUs forms a *subfile block* (Vbs is the vertical block size; Hbs is the horizontal block size). A row-major rectangular arrangement of Vn by Hn subfile blocks (one per subfile) forms a *subfile block group* (Figure 2). These are then distributed in row-major order across Vesta cells, though some of them may not fit evenly onto the cells (Figure 3). If a portion of the logical address space of a subfile does not correspond to any cell, reads from that region return zeros, and writes to that region are ignored.

Although the subfile partitioning scheme is fairly complex, it has the advantage that once the file access pattern is specified at file-open time, complex data accesses can subsequently be made with simpler code. Indeed, the Vesta interface provides only simple contiguous reads and writes on subfiles; it is the judicious definition of subfiles that maps those reads and writes onto non-contiguous data in the cells. Additionally, because different compute-processors can specify different (non-overlapping) subfiles to access, multiple processors can access separate sections of a file without danger of interfering with each other's reads or writes.[2]

3 Vesta on Galley

We implemented the Vesta file system interface on Galley with a one-to-one mapping between Vesta files and Galley files. Additionally, Vesta cells have a one-to-one mapping onto Galley forks. If the user requests more cells than there are disks (Galley subfiles) in the system, the cells are allocated across Galley subfiles in round-robin fashion. Thus, multiple Vesta cells can reside on a single disk,

[2]See `http://www.research.ibm.com/people/c/corbett/vesta.html` for more information on Vesta.

Cells:	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	0	2	4	0	2	4	6	8	10	6	8	10	12	14
	1	3	5	1	3	5	7	9	11	7	9	11	13	15
	0	2	4	0	2	4	6	8	10	6	8	10	12	14
	1	3	5	1	3	5	7	9	11	7	9	11	13	15
	18	20	22	18	20	22	24	26	28	24	26	28	30	32
	19	21	23	19	21	23	25	27	29	25	27	29	31	33
	18	20	22	18	20	22	24	26	28	24	26	28	30	32
	19	21	23	19	21	23	25	27	29	25	27	29	31	33
	36	38	40	36	38	40	42	44	46	42	44	46	48	50
	37	39	41	37	39	41	43	45	47	43	45	47	49	51

Subfile 0 Subfile 1 Subfile 2 Subfile 3

Vbs = 2, Hbs = 3, Vn = 2, Hn = 2

Figure 3: Sample Vesta File Layout.

but a single cell, in keeping with the Vesta specification, can never span multiple disks. A metadata fork, stored in the first Galley subfile of every file, stores the number of Vesta cells in the file and the basic striping unit size.

Vesta read and write requests were translated into Galley's strided-read and -write procedures with little difficulty. Since Galley supports asynchronous I/O, we decided to implement Vesta reads and writes with parallel Galley reads and writes for efficiency.

3.0.1 Specifics

A Vesta read takes the following parameters:

- offset (relative offset at which to begin reading)
- count (number of basic striping units to read)
- buffer (address into which to read data)

with the following additional information specified at file-open time:

- vbs (vertical number of basic striping units in a block)
- vn (vertical number of blocks in a block group)
- hbs (horizontal number of basic striping units in a block)
- hn (horizontal number of blocks in a block group)
- subfile (subfile number)

and the following additional information specified at file-create time:

- cells (number of cells)
- bsu (size of basic striping unit)

A Galley non-blocking strided read takes the following parameters:

- Gfork (fork number)
- Gbuffer (address into which to read data)
- Goffset (relative offset at which to begin reading)
- Gsize (amount of data to read in each stride)
- Gfstride (distance to stride through file)
- Gmstride (distance to stride through memory)
- Gquant (number of strides to make)

Vesta parameters are mapped to the Galley parameters as follows:

- Let *Gusno* represent the used Galley subfile number (i.e., the Galley subfile number among the Galley subfiles that are being used in the current read).

- Let *groups* be the horizontal number of Vesta block groups, including those which do not fit evenly within the Vesta cells (like the right-most group in Figure 3).
- Let *Vgroups* be the horizontal number of whole Vesta block groups. (Thus, $groups \equiv Vgroups + 1$ when *cells* is not a multiple of $hbs \times hn$.))
- Let *bsuoff* be Goffset in BSUs.
- Let the pmod function return the positive modulus of its arguments (i.e., $pmod(a, b) \equiv ((a \bmod b) + b) \bmod b$).
- The variables i and j cycle through the Vesta block groups and the BSUs within each block, respectively.

$$\begin{aligned}
Gfstride &= bsu \times vbs \times vn \\
Gmstride &= \left\lceil \frac{cells}{hbs \times hn} \right\rceil \times bsu \times vbs \times hbs \\
Gsize &= vbs \times bsu \\
\\
Vgroups &= \left\lfloor \frac{cells}{hbs \times hn} \right\rfloor \\
\\
groups &= \left\lceil \frac{cells}{hbs \times hn} \right\rceil \\
Gusno &= 0
\end{aligned}$$

for i = 0 ... Vgroups - 1

for j = 0 ... hbs - 1

$$Gfork = i \times hbs \times hn + hbs \times (subfile \bmod hn) + j$$

$$Gquant = \left\lfloor \frac{count}{groups \times hbs \times vbs} \right\rfloor + \left((\lfloor \frac{count}{vbs} \rfloor \bmod (groups \times hbs)) > \text{pmod}(Gusno - \frac{offset}{vbs}, groups \times hbs) \right) ?1:0$$

$$bsuoff = \left\lfloor \frac{subfile}{hn} \right\rfloor \times vbs + \left\lfloor \frac{\lfloor \frac{offset}{vbs} \rfloor}{hbs \times groups} \right\rfloor \times vn \times vbs + \left(\left(\left\lfloor \frac{offset}{vbs} \right\rfloor \bmod (hbs \times groups) > Gusno \right) ?1:0 \right) \times vn \times vbs$$

$$Goffset = bsu \times bsuoff$$

$$Gbuffer = buffer + bsu \times \text{pmod}\left(Gusno \times vbs - offset \bmod (groups \times hbs \times vbs), groups \times hbs \times vbs\right)$$

Issue non-blocking Galley read/write with computed parameters

$$Gusno = Gusno + 1$$

Wait on all non-blocking Galley reads/writes

This pseudocode omits certain details for simplicity. For instance, it assumes that the caller has specified a flag to read/write BSUs rather than bytes, and that the Vesta offset and count parameters are multiples of Vbs. It also ignores Vesta block groups that do not lie entirely on the Vesta cells. Our implementation handles these more complex cases.

We could not implement the following features of Vesta. Each of these features requires specific support in Galley that is not available.

- Permission bits
- File checkpointing

- “Cautious” file-access mode (in which the system insures that two simultaneous accesses to overlapping regions of the same sections of a file will not intermingle to produce non-serializable results)
- Asynchronous pre-fetching of file data to internal buffers
- Internal transfer of data from file to file (bypassing a transfer from an I/O node to a compute node and back)
- File touching (Galley does not store a last-modification time)
- File renaming
- Vesta Xrefs (directories)
- Importing and exporting files from/to external file systems
- System administration (user accounts, quotas, backups)

Some of these could be supported if Galley had implemented typical features like permission bits and modification times; others like the “cautious” mode require unique, fundamental features not available in many file systems.

We did not implement all of the features of Vesta that could be implemented. The following are features that we could have implemented using Galley but did not:

- Row-major distribution of Vesta BSUs across cells
- Column-major distribution of Vesta BSUs within cells
- Fixed-size (preallocated) files

4 Testing Vesta

To test Vesta's performance and capabilities, we wrote a parallel, out-of-core, square-matrix-multiplication program using the standard $\theta(n^3)$ algorithm, using the Vesta file system interface to read the factor matrices and write the product matrix. The program used all available compute processors and I/O processors, and was capable of multiplying square matrices of arbitrary magnitude, provided that at least two rows (or columns) of a matrix could fit into one compute-processor's memory at a time, and that the factor and product matrices were small enough to fit on the file system together.

We wanted a program that:

- used all of the compute processors
- used all of the I/O processors
- had no holes[3] in the files
- had no redundant data in the files
- had identical formats for storing both of the factor matrices and the product matrix

[3] A *hole* is an empty region of a file that occupies no disk space but reads as all zeros.

- logically partitioned the files into Vesta subfiles such that a given compute processor would have access only to the data that it needs

Due to inevitable Unix limits on the number of open files, we assumed that columns of matrices could not each be represented as a cell (since there would be too many). Instead, we chose the number of cells to be equal to the number of I/O processors.

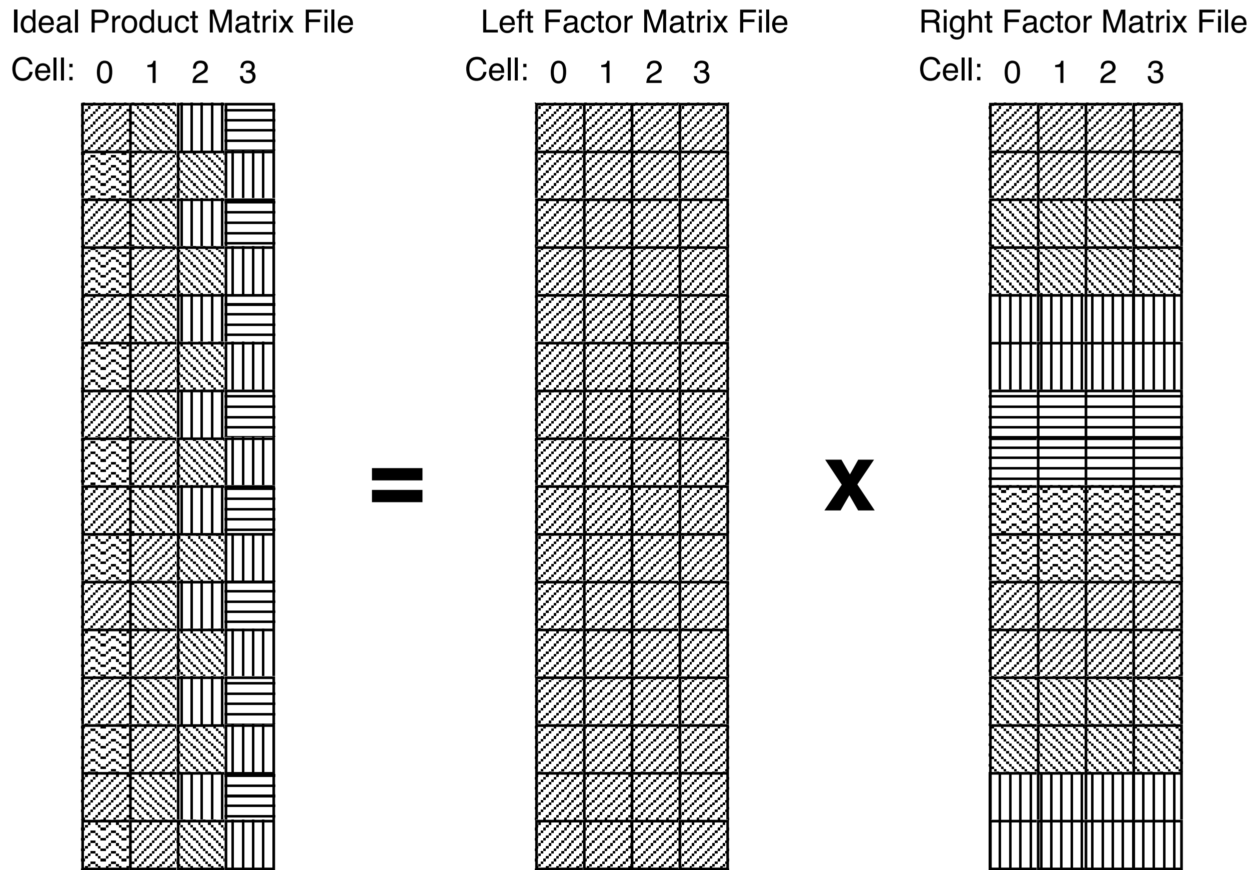
The standard matrix-multiplication algorithm requires that matrix rows be read from the left matrix, and matrix columns be read from the right matrix. Since groups of Vesta blocks are always arranged in row-major order (Figure 3), it is impossible (given the constraint that each compute processor should have access to only the necessary factor elements) to arrange a file partition such that a single read request would stride down an entire Vesta cell, as would be necessary to read columns from a matrix stored in row-major order. Therefore, to make all the read accesses stride across Vesta cells rather than down through one (to concur with the ordering of groups of Vesta blocks), we stored the right factor matrix in column-major order and the left factor matrix in row-major order.

With both the left and right factor matrices stored in the order in which we wanted to access them, each row (of the left factor) or column (of the right factor) occupied N logically contiguous BSUs. Since Vesta blocks have to be in a grid layout (Figure 3), however, if N is not a multiple of the number of cells there would be holes in the factor files. We chose to add the constraint that N must be a multiple of the number of IOPs to avoid these holes in the factor files.

By the definition of matrix multiplication, element $[i, j]$ of the product matrix is the dot product of row i of the left-factor matrix and column j of the right-factor matrix. Our multiplication program took advantage of the fact that these products are all independent of one another, and computed dot products in parallel. If P is the number of compute processors, each row of the product matrix was calculated with N/P sets of P parallel dot products. Processor $j \bmod P$ read row i from the left matrix and column j from the right matrix, calculated the dot product, and wrote the result into $[i, j]$ of the product matrix. Alternatively, processor $i \bmod P$ could have read row i from the left matrix and column j from the right matrix, calculated the dot product, and written the result into $[i, j]$ of the product matrix. In either case, every element of one factor matrix is accessed by every processor, while the rows or columns of the other matrix are accessed cyclically (with order P) by all processors. Thus, we used a single Vesta subfile to access the left-factor matrix, while we used P Vesta subfiles to access the right-factor matrix. Note, however, that the *elements* of the result are accessed cyclically (order P) by all processors. Since there is no guarantee that P elements will fit evenly across the cells of a Vesta file (that is, P is not necessarily a factor of the number of I/O processors), and since subfile block groups must be arranged in a grid (see Vesta file diagram) it was generally impossible to partition the result into multiple subfiles (one per compute processor) so that each processor had access only to the necessary elements (Figure 4). For scalability reasons, we did not want to add the constraint that the number of compute processors be a factor of the number of I/O processors. Therefore, we reluctantly made the entire product matrix accessible to all compute processors as a single subfile, and forced the compute processors to calculate the appropriate offsets (Figure 5).

Because of the Vesta design, we were not able to satisfy all of the "desired features" of the matrix-multiplication program. We could not:

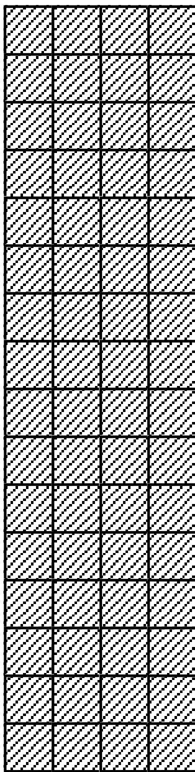
- have identical formats for storing both of the factor matrices and the product matrix, and
- partition the product file into subfiles such that a given compute processor would have access only to the data that it needs.



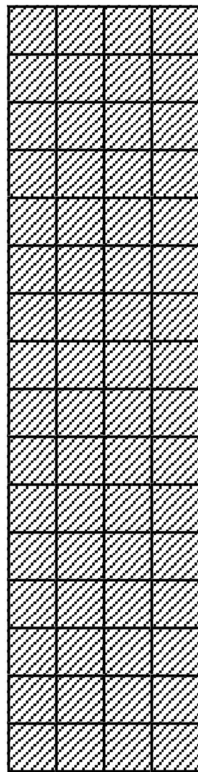
Multiplication of 2 8x8 Matrices with 4 I/O Processors and 5 Compute Processors.

Hypothetical Vesta subfiles, one per compute processor, are each shaded uniquely. Note, however, that the arrangement of subfile blocks in the product matrix cannot be accommodated by any Vesta accessing scheme.

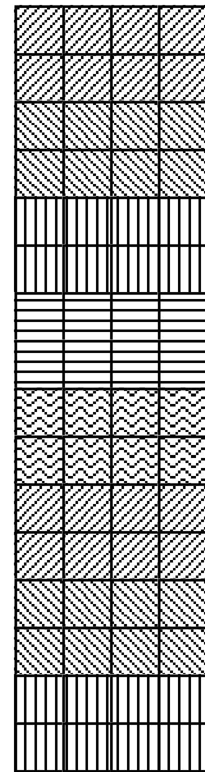
Figure 4: Ideal Matrix Multiplication File Layouts.



=



x



Multiplication of 2 8x8 Matrices with 4 I/O Processors and 5 Compute Processors.

Vesta subfiles, one per compute processor, are each shaded uniquely.

Figure 5: Matrix Multiplication File Layouts.

As a result, to access the product matrix, the programmer needed to compute file offsets as in a simple flat-file representation.

In this case, the drawback of complexity associated with the Vesta file partitioning method did not seem to be offset by usefulness. We could not easily represent a file partitioning that would suit arbitrary-sized square-matrix multiplication with the Vesta interface. (As it was, we restricted N to a multiple of the number of IOPs.)

5 Experiments

Our underlying hardware was an FDDI network of eight IBM RISC System/6000 model 250 workstations running AIX version 4.1, each with a 66 MHz PowerPC 601 CPU, 64 MB of RAM, and two external Seagate 31200N 1GB SCSI2 disks.[4]

By profiling the out-of-core matrix multiplication program described above, we were able to determine how much CPU time was added to the program by the layer of Vesta code on top of Galley. Multiplying two 1000 by 1000 matrices with 4 compute-processors and 4 IOPs took 99.25 minutes of wall time.[5] In such a run, the average CPU time (across all four compute processors) of the Vesta functions alone (not including the underlying Galley functions) was 1.46 minutes.[6]

The time that Vesta spends computing parameters for Galley calls can be thought of as a worst-case overhead of Vesta. In many cases, the user program would have to make similar computations of Galley parameters. Therefore, placing the computations in a filesystem library rather than in the user program serves more to provide abstraction than to add unnecessary overhead.

Figure 6 shows the relationship between problem size and run time for our parallel out-of-core matrix multiplication program on Vesta.

6 Conclusions

In implementing Vesta on top of Galley, we demonstrated the flexibility of the Galley interface. Galley was a good foundation for a basic implementation of Vesta, though it lacked the functionality necessary to implement all of the features of Vesta. Some features would be expected to be found in any complete file system (Galley is a prototype), such as permission bits and modification times, while others are specific to Vesta's features, like checkpointing. We found the run-time overhead of the Vesta layer to be small relative to a program's overall run-time. However, the problems that we encountered when writing a standard out-of-core matrix multiplication program using Vesta convinced us that Vesta's ability to represent only certain highly-regular data accesses significantly decreases its number of practical uses.

7 Acknowledgments

We thank Peter F. Corbett and Dror G. Feitelson of IBM for providing us with sample Vesta programs. We are very grateful to Nils Nieuwejaar for his help both with using Galley and with using the hardware and software in our FLEET lab. We also thank Sivan Toledo for his help with techniques for software performance evaluation in AIX.

[4]See `http://www.cs.dartmouth.edu/~fleetlab/` for more information on our lab.

[5]Clearly, 1000x1000 is a small problem that would rarely need to run out-of-core, but here with our explicit use of remote disks the run time is (not surprisingly) quite slow.

[6]Ideally, we would measure wall time here, but a profiler can only report CPU time.

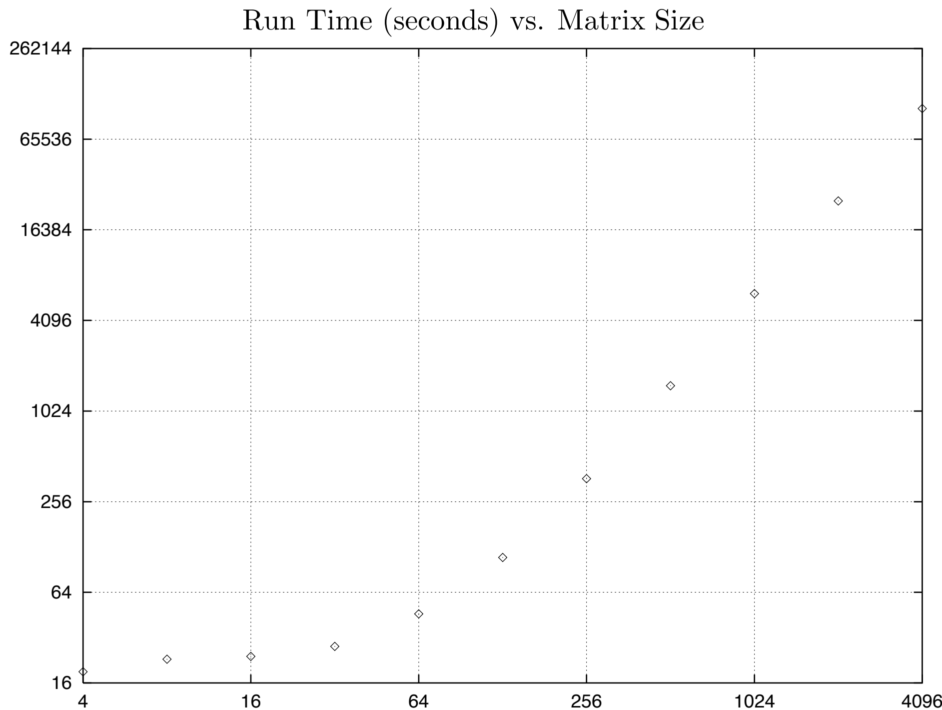


Figure 6: Matrix Multiplication Run Times.

8 Availability

Our library source code may be downloaded at "ftp://ftp.cs.dartmouth.edu/pub/pario/vesta-galley.tar.Z".

References

[CBF93] Peter F. Corbett, Sandra Johnson Baylor, and Dror G. Feitelson. Overview of the Vesta parallel file system. In *Proceedings of the IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 1–16, Newport Beach, CA, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 7–14.

[CF94] Peter F. Corbett and Dror G. Feitelson. Vesta file system programmer's reference. Technical Report Research Report RC 19898 (88058), IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, October 1994. Version 1.01.

[CF96] Peter F. Corbett and Dror G. Feitelson. The Vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3):225–264, August 1996.

[CFP+95] Peter F. Corbett, Dror G. Feitelson, Jean-Pierre Prost, George S. Almasi, Sandra Johnson Baylor, Anthony S. Bolmarcich, Yarsun Hsu, Julian Satran, Marc Snir, Robert Colao, Brian Herr, Joseph Kavaky, Thomas R. Morgan, and Anthony Zlotek. Parallel file systems for the IBM SP computers. *IBM Systems Journal*, 34(2):222–248, January 1995.

[KN96] David Kotz and Nils Nieuwejaar. Flexibility and performance of parallel file systems. In *Proceedings of the Third International Conference of the Austrian Center for Parallel*

Computation (ACPC), volume 1127 of *Lecture Notes in Computer Science*, pages 1–11. Springer-Verlag, September 1996.

[Nie96] Nils A. Nieuwejaar. *Galley: A New Parallel File System for Parallel Applications*. PhD thesis, Dept. of Computer Science, Dartmouth College, November 1996. Available as Dartmouth Technical Report PCS-TR96-300.

[NK96a] Nils Nieuwejaar and David Kotz. The Galley parallel file system. In *Proceedings of the 10th ACM International Conference on Supercomputing*, pages 374–381, Philadelphia, PA, May 1996. ACM Press.

[NK96b] Nils Nieuwejaar and David Kotz. Performance of the Galley parallel file system. In *Proceedings of the Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 83–94, Philadelphia, May 1996. ACM Press.

[NK97] Nils Nieuwejaar and David Kotz. The Galley parallel file system. *Parallel Computing*, 23(4):447–476, June 1997.

[Tho96] Joel T. Thomas. The Panda array I/O library on the Galley parallel file system. Technical Report PCS-TR96-288, Dept. of Computer Science, Dartmouth College, June 1996. Senior Honors Thesis.