Addressing Partitioned Arrays in Distributed Memory Multiprocessors – the Software Virtual Memory Approach

Rajeev Barua
David Kranz
Anant Agarwal

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

October 1993

Abstract

Partitioning distributed arrays to ensure locality of reference is widely recognized as being critical in obtaining good performance on distributed memory multiprocessors. Data partitioning is the process of tiling data arrays and placing the tiles in memory such that a maximum number of data accesses are satisfied from local memory. Unfortunately, data partitioning makes it difficult to physically locate an element of a distributed array. Data tiles with complicated shapes, such as hyperparallelepipeds, exacerbate this addressing problem.

In this paper we propose a simple scheme called software virtual memory that allows flexible addressing of partitioned arrays with low runtime overhead. Software virtual memory implements address translation in software using small, one-dimensional pages, and a compiler-generated software page map. Because page sizes are chosen by the compiler, arbitrarily complex data tiles can be used to maximize locality, and because the pages are one-dimensional, runtime address computations are simple and efficient. One-dimensional pages also ensure that software virtual memory is more efficient than simple blocking for rectangular data tiles.

Software virtual memory provides good locality for complicated compile-time partitions, thus enabling the use of sophisticated partitioning schemes appearing in recent literature. Software virtual memory can also be used in systems that provide hardware support for virtual memory. Although hardware virtual memory, when used exclusively, eliminates runtime overhead for addressing, we demonstrate that it does not preserve locality of reference to the same extent as software virtual memory.

Keywords: multiprocessors, compilers, addressing, data partitioning, loop partitioning, pages, virtual memory, locality.

Authors' e-mail: {barua,kranz,agarwal}@lcs.mit.edu. Authors' phone: (617)253-8438.

1 Introduction

The problem of loop and data partitioning for distributed memory multiprocessors with global address spaces has been studied by many researchers [1, 3, 6, 13]. The goal of *loop partitioning* for applications with nested loops that access data arrays is to divide the iteration space among the processors to get maximum reuse of data in the cache, subject to the constraint of having good load balance. For architectures where non-local memory references are more expensive than local memory references, the goal of *data partitioning* is to place data in the memory where it is most likely to be accessed by the local processor. Data partitioning tiles the data space and places the individual data tiles in the memory modules of the processing nodes. Data partitioning introduces addressing difficulties because the data tiles can become discontiguous in physical memory. This paper focuses on the problem of generating efficient code to access data in systems that perform loop and data partitioning.

Current methods for addressing data in systems that perform data partitioning fall into two general classes. The first class relies on *hardware virtual memory* to resolve addresses. The second class uses *software address computation* to determine the physical location of a data element. In this paper, when we refer to virtual memory, we are concerned only with the virtual to physical address translation component, and not with issues of backing store or protection.

Systems that support global virtual memory in hardware can access data in the same way as on a uniprocessor. Each array occupies a contiguous portion of the virtual address space and the page tables are set up so that the data in a given tile is placed on pages that are allocated in the local memory of the processor accessing the tile. The problem with hardware virtual memory results from the large page sizes (for example, 4K bytes) relative to the dimensions of data tiles. For example, a 1024-word data tile might require as many as 1024 pages to cover it, if it is poorly aligned with the pages. The problem is even more serious with multidimensional data tiles because the individual dimensions of data tiles can be much smaller than the page size, even when the overall size of the data tile is large. Thus, because the page size is fixed by the hardware, ensuring good locality with a large number of processors may require running very large problem sizes.

Software address computation is commonly used on multicomputer architectures that do not provide hardware support for a shared address space. In such systems, the data is divided into blocks that are distributed across the processors. Processors maintain mapping functions that map array elements to their physical locations. Because this mapping is accomplished entirely in software, systems can choose block sizes of any size and shape. Systems also have the flexibility to use different sizes and shapes for different arrays. Unfortunately, as demonstrated in Section 5, runtime address computations are very expensive, even for the simplest shapes and sizes of the tiles. Hyperparallelepiped shapes, or tile dimensions that that are not powers of two, result in even more complicated addressing functions. Therefore, in practice, compilers use rectangular sub-blocks of the array, with individual dimension lengths that are powers of two, resulting in a loss in locality.

This paper, introduces a new method, called *software virtual memory*, that combines the efficiency of hardware virtual memory with the flexibility of software address computation. Software virtual memory is akin to hardware virtual memory in that it covers data tiles with one-dimensional pages, allowing a simple address translation. It differs from hardware virtual memory in that runtime software is used to translate virtual addresses to physical addresses, thus allowing arbitrary page sizes that can be different for different arrays. The resulting flexibility is of tremendous advantage because if pages are made small enough, one can approximate closely any shape in the data space, thus allowing smaller problem sizes on a large machine.

How is software virtual memory different from software address computation? Previous methods of software address computation can be viewed as attempting to cover data arrays with "pages" whose shape and size are identical to those of the data tiles, and using software to accomplish the mapping function. For example, a system that performs software address computation views a three dimensional array tiled using cubical blocks as a three dimensional array covered with relocatable pages that are themselves three dimensional. As demonstrated in Section 4, address computations for multidimensional pages with complicated shapes incur severe runtime overhead. Software virtual memory can be viewed as a software address computation scheme that restricts the relocatable units to be small, one-dimensional pages. Software virtual memory borrows the use of one-dimensional pages from hardware virtual memory to simplify the mapping function.

The above three systems trade off the cost of computing the location of an array element and the ratio of local to remote memory accesses. Hardware virtual memory eliminates the the cost of computing the location of array elements, but suffers from poor locality when the pages are larger than data tile dimensions. Software address computation optimizes locality of reference, with a significant loss in addressing efficiency. Software virtual memory allows a compiler to make the tradeoff between locality and addressing efficiency. In general, smaller pages result in better locality, but result in larger software tables and more cache pollution, while large pages result in poor locality and reduced addressing overhead. By choosing appropriate page sizes, we demonstrate that a compiler can retain near-perfect locality, while incurring only a modest loss in addressing efficiency over the hardware virtual memory scheme. Note that in distributed memory machines without a shared address space, software virtual memory has more to offer because hardware virtual memory is not supported.

We have implemented the software virtual memory scheme in the compiler and runtime system for the Alewife machine [2], a globally cache-coherent distributed-memory multiprocessor. We use the method of loop and data partitioning described in [3]. In this paper we demonstrate that:

- The overhead of software virtual memory is small in general. Furthermore, if rectangular data partitions can be used, simple compiler transformations can eliminate almost all of the overhead.
- Software virtual memory can use page sizes as small as 32 bytes without significant loss in efficiency. This allows precise covering of arbitrary data tile shapes, and near optimal locality.
- For many realistic problem sizes, the large size of hardware virtual memory pages can cause very poor data locality.
- Software virtual memory has significantly lower addressing overhead than software address computation.

The rest of the paper is organized as follows. Section 2 describes issues involving loop and data partitioning. Section 3 gives an overview of the problem of distributed array access and related work. Section 4 describes the software virtual memory scheme and estimates its cost compared to other approaches. Section 5 contains some experimental results on the locality/addressing tradeoff. We conclude in Section 6 .

2 Loop and Data Partitioning

Most existing work in compilers for parallel machines has focused on parallelizing sequential code and executing it on machines where each processor has a separate address space, *e.g.* CM-5 or Intel iPSC. It is usually assumed that the programmer specifies how data is distributed and the compiler tries to optimize communication by grouping references to remote data so the high cost of remote accesses can be amortized [5, 7, 8, 10, 9, 11, 12, 14, 16]. These methods only work well when the granularity of the computation is large and regular.

Some recent work has looked at compilation for machines with a shared address space, physically distributed memory and globally coherent caches [3, 6]. In these machines, each processor controls a local portion of the global memory; references to the local portion have lower latency than references that access remote data over the communication network. On such machines there is more opportunity to compile finer-grain or less regular programs because the hardware supports finer-grain remote data access and prefetching. The formulation of the problem in this context is as follows. The compiler takes an explicitly parallel program as input. This program may have been written by a user or produced from a sequential program by a parallelizing tool, and is assumed to consist of some number of parallel loop nests and arrays that they access. It is the compiler's job to divide the loop iterations and data among the processors so as to maximize data reuse and minimize the number of remote memory accesses.

In this paper we assume that loop and data partitioning has been done and look at the question of how to generate code to access the data. If we look at the portion of the iteration space running on some processor P, we can determine the footprint in the data space, *i.e.* the set of data elements accessed. We would like to allocate those data elements to the local memory of P. Doing this for all of the loops and data will yield a function that maps array element indices to physical memory locations. The code for this mapping must be executed at each array reference and will result in overhead o. The tradeoff we examine is between making o small and having a large number of references be local. This follows from the fact that a simple mapping implies that the data will be mapped to processors in large, regular chunks. These chunks may not match the data mapping that would minimize the number of remote memory accesses.

This tradeoff is captured in the following equation that expresses estimated running time of a loop iteration:

$$T = ctime + M * (o + n_{cachehits} * C + n_{local} * L + n_{remote} * R)$$

where C is the cache hit time, M is the number of memory references in the loop, o is the overhead introduced by software virtual memory, L is the local memory latency and R is the remote memory latency. $ctime$ is the time spent in actual computation.

The loop and data partitions determine the fraction of cache hits ($n_{cachehits}$) and cache misses that go to local memory (n_{local}), while the target architecture determines C and L. R, the remote memory latency, is a more difficult parameter to account for because it depends on n_{remote} as well as the architecture. If n_{remote} is large, contention and bandwidth limitations of the interconnect in the multiprocessor may increase R significantly. In the rest of the paper we look at various mappings, their access costs, and their effects on data locality.

3 The Addressing Problem

In this section we define the addressing problem and study the various alternatives to solving it.

Addressing an element involves finding its physical address – specified by a processor number and offset within that processor's memory. In a shared memory machine this information is usually contained in one global address. As discussed earlier, there are two general approaches to solving the addressing problem: software address computation and hardware virtual memory (HVM). This section describes these two methods and the efficiency of addressing of each, and the next section describes software virtual memory (SVM).

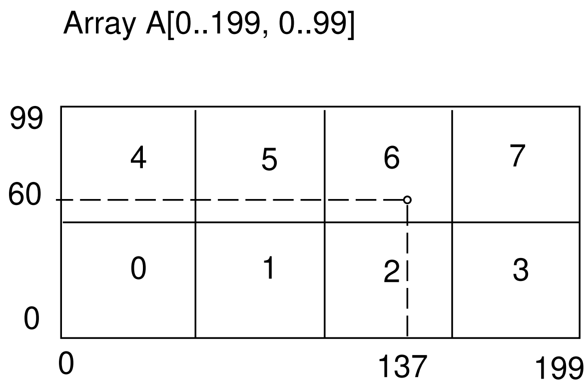
3.1 Software Address Computation

There are many approaches to software address computation. One approach is to calculate the address of an element by linearizing the points in the data tile, and using some geometrically derived formula to find the processor number and offset. The special case for rectangular data tiles is commonly referred to as *blocking*, and is the most widely used form of data allocation in multiprocessors. Figure 1(a) shows a two dimensional array blocked among processors. Processor numbers are assigned in row or column major order, as are the offset numbers within a block. Figure 1(b) shows an example address calculation for this scheme for a 2-D array.

The steps for addressing an element using blocking are:

1. A processor index calculation (division) in each of n processor dimensions in the data space.
2. A row major computation on the above to find the processor number.
3. n subtractions and one row-major computation to find the offset within the block.
4. A load from a vector of distributed block base addresses.
5. An add of the base to the offset to get the desired address.

We note that compiler footprint analysis may be able to perform a loop-invariant code hoisting of steps 1 and 2. Strength reduction optimizations may also be possible for step 3. We shall show the code needed for addressing an element using software virtual memory in Section 4, and see that it is always significantly simpler.



(a) Data Space showing blocks

To find the address of A[137,60]:
Processor index (dim 1) = $\lfloor 137/50 \rfloor = 2$
Processor index (dim 2) = $\lfloor 60/50 \rfloor = 1$
$\Rightarrow$ Processor number = 2 + 1 * 4 = 6 (row major)
Offset = (137 - 100) + (60 - 50) * 50 = 537

(b) An example address calculation

Figure 1: Blocking of a distributed array.

```
Doall (i=0:149, j=0:49)
  A[i,j] = B[i+j,j]+B[i+j+1,j+2]
EndDoall
```

Figure 2: Example of code requiring a parallelogram partition.

Another software approach is to allocate data tiles of complex shapes, for example, parallelograms, to each processor. This is a generalization of blocking. As shown in [3], parallelogram partitions are often required to ensure optimal locality when array accesses contain affine index functions.

Let us illustrate the difficulty of addressing parallelogram data tiles with the following example. Consider the nested **Doall** loop in Figure 2. Suppose the iteration space is partitioned uniformly using a rectangular tile shape as depicted in Figure 3. The shape of the data tile that comprises the data elements of array **B** accessed by the loop tile (also known as the footprint of the loop tile) is shown in Figure 4.

Now, suppose the data array is tiled using the parallelogram from Figure 4 to maximize locality, as illustrated in Figure 5. Finding the address of an element now involves a coordinate transformation to find the processor number, and another to find the offset. Both operations are very expensive at runtime. A common simplification is to allocate the smallest enclosing rectangular window around the data tile, but this still requires the processor number calculation, and wastes memory. Using this simplification, the steps for addressing

Figure 3: Loop tile at the origin of the iteration space. $(50, 0)$ and $(0, 25)$ are the bounding vectors of the loop tile.

Figure 4: Data tile in the data space corresponding to the references to array **B**.

an element are:

1. A basis resolution along the parallelogram basis to find a processor index for each of n dimensions.
2. A floor operation on each of the above.
3. A row major computation on the above to find the processor number.
4. n subtractions and one row-major computation to find the offset within the block.
5. A load from a vector of distributed block base addresses.
6. An add of the base to the offset to get the desired address.

Step 4 may be strength reducible.

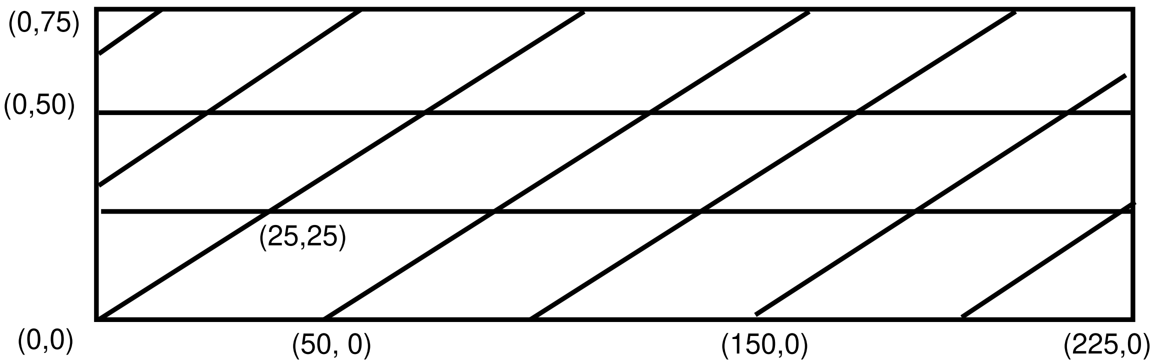


Figure 5: Data tiling of array **B** using parallelograms.

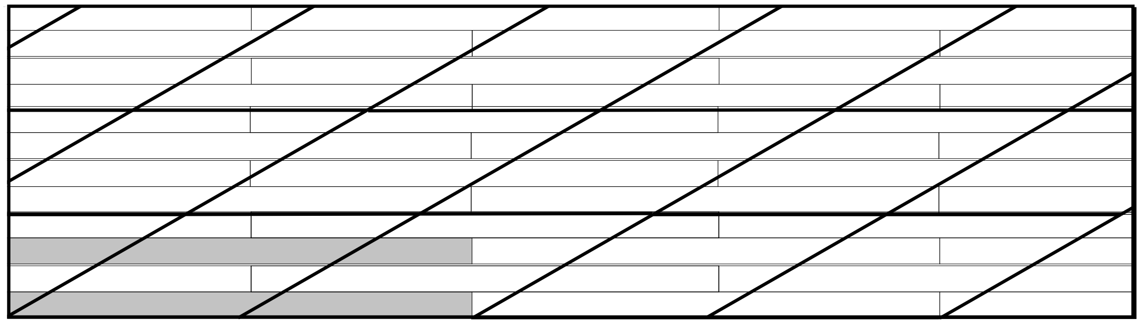


Figure 6: Loss of locality due to large hardware pages.

3.2 Hardware Virtual Memory

An alternative to software address calculation schemes is to use hardware virtual memory. In a shared-address space machine, hardware virtual memory allows arrays to be distributed pagewise, with different pages possibly allocated to different processors. A page is placed on a processor with maximum overlap with it. In this paper we are concerned only with the address translation provided by virtual memory and not with backing store or protection issues.

Pages are one-dimensional (linear) blocks that cover the distributed arrays, which may be multidimensional. The virtual memory approach in hardware or software has an advantage over blocking because the only multidimensional row major computation is to compute the virtual address. Blocking needs to do another to calculate the processor and offset.

The problem with hardware virtual memory, however, is that most real machines have page sizes that are too big to allow linear pages to approximate multidimensional data tiles unless the tiles are very large. Hardware virtual memory deals with actual movement of data through paging and a large page size is needed to amortize large I/O costs. Figure 6 shows how hardware pages can be used to cover the tiled data array from the previous example. The shaded pages are allocated to a single processing node. As we can see, large page sizes result in a poor approximation of the data tile, resulting in poor locality. On the other hand, hardware virtual memory systems that support multiple page sizes might reduce some of the problems with fixed page sizes. Although multiple-page-size systems merit further exploration, they do not appear very promising because only the simplest of these solutions are practical to build [15], and the need to support very small pages further complicates the hardware.

To overcome the problems of the above approaches we propose using the same type of paging structure as hardware virtual memory, but performing the address translation in

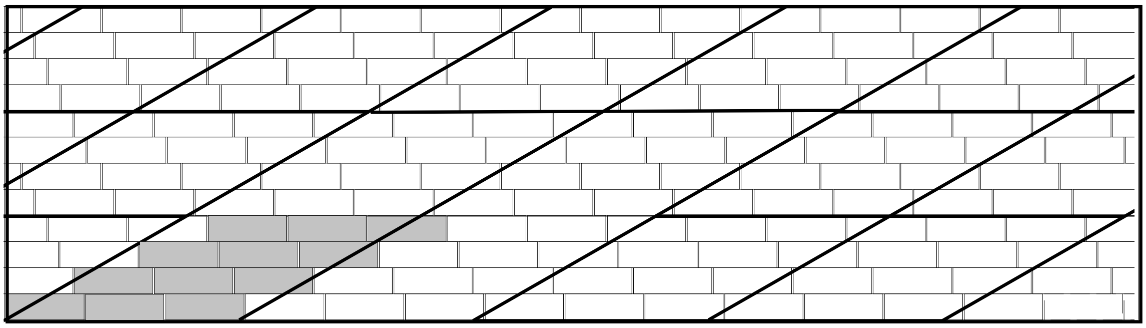


Figure 7: Approximation of a parallelogram data tile by small software pages.

software, so that the compiler can choose page sizes to fit data tile shapes.

4 Software Virtual Memory

The method of software virtual memory (SVM) is the following. Given a loop and data partitioning, the compiler *stripes* the data array with small pages as indicated in Figure 7. The compiler also constructs a pagemap that stores the processor at which each page will be allocated. A *pagesize estimator* finds the largest pagesize which is still small enough to give good locality. Each page is placed according to its relation to a data tile. If a page is contained wholly within a data tile it is allocated the processor with the most accesses to that tile. If a page crosses the boundary between tiles, it is allocated as if it were contained in the data tile that has maximal overlap with the page. For efficiency of translation, page sizes are required to be a power of two. This pagemap will be used at load time to construct a page table in memory.

Figure 7 shows the approximation of a parallelogram data tile pattern by small software-allocated pages. Virtual addresses of elements are the same as in a uniprocessor, in either row-major or column-major order. By using small enough pages we can approximate the shape of any data tile and get good data locality.

At runtime, the following steps are needed for an array access. These steps are overhead beyond the normal index computation for array accesses on a uniprocessor.

```
srl r2,log(pagesize),r3   # get page number (shift right)
sll r3,2,r3               # convert to offset into page table (shift left)
ld [r1+r3],r4             # get physical page base
and r2,pagesize-1,r3      # get offset within page (mask)
lddf [r4+r3],fp0          # do real load (double precision)
```

Figure 8: Code sequence for software virtual memory

1. A fetch from the page table generated by the compiler using the page number obtained from the virtual address.

2. An add of the offset obtained from the virtual address to the physical page base.

Figure 8 shows the SVM code for a doubleword memory reference, assuming the base of the page table is in `r1` and the virtual address is in `r2`. The sequence adds an overhead of only four instructions (the `lddf` would be done anyway).

On the Sparcle processor [4] used in Alewife, these 4 instructions require 5 cycles, assuming all instructions and the page table lookup hit in the cache. We expect that the cache hit rate will not degrade significantly even for small page sizes, because the page table entries are small compared to a page. For example, the software page tables for 128-byte pages occupy less than than five percent of the area occupied by the data. Furthermore, the software page table comprises read-only entries, each of which is accessed multiple times for each page. Thus, the cache is not significantly polluted, and subsequent accesses to a given entry hit in the cache. This issue is discussed further in Section 5.

It is important to note that when data partitions are simple rectangles, a simple compiler transformation similar to loop invariant code hoisting can be performed on this sequence by subdividing the inner loop to iterate across a page. This transformation would eliminate almost all of the SVM overhead.

Hardware virtual memory would give us the functionality of the first four instructions in this sequence for free, assuming TLB hits, but at the possible cost of making the real load remote rather than local.

This code sequence is always better than what would be obtained by n-dimensional blocking because an n-dimensional calculation is required to obtain the block number and offset. Thus, software virtual memory can better approximate arbitrary data tiles *and* is more efficient than the commonly used partitioning methods.

5 Experimental Data

We have described a software memory scheme and explained why it should compare favorably to software blocking and have a small overhead compared to hardware virtual memory. In this section we give some quantitative measure of what these overheads are. Because the overhead of memory references, both from addressing costs and the cost of remote references, depends so strongly on the ratio of computation to communication, we will just present data from two small programs that have a realistic number of memory references.

5.1 Comparison to HVM

To compare software virtual memory to hardware virtual memory we have to consider these questions:

- What is the general overhead of doing the address translation in software?
- Given that the main advantage of SVM is the ability to use small pages, how does address translation overhead depend on the size of the page?
- What is the overhead due to reduced data locality if large page sizes are used?

We examine these questions in the context of the equation introduced in Section 2, repeated here:

$$T = ctime + M * (o + n_{cachehits} * C + n_{local} * L + n_{remote} * R)$$

Since software virtual memory increases o in order to reduce n_{remote}, the right tradeoff may be very different for different target architectures and machine sizes. In large machines it may be worth a higher fixed array access overhead in order to reduce n_{remote} because available bandwidth may not grow linearly with the number of processors. It should also be noted that if a program does a lot of calculation on the data it accesses, or reuses data in the cache most of the time, performance will be largely insensitive to the details of data partitioning, as in matrix multiply.

We ran two small programs on a simulator for the Alewife machine: [description of Alewife in full paper] a Jacobi relaxation that can achieve good locality with rectangular data partitions and a synthetic application that needs a parallelogram data partition for good locality, (pgram). To compare with HVM, we modified the simulator to perform address translation for free, thus modeling a perfect HVM system with no TLB misses. To be conservative, the optimization of lifting the address translation out of the inner loop was not performed in any of these programs. We know that in the cases where this optimization is possible the generated code will be almost exactly the same as if we had hardware virtual memory.

5.1.1 Rectangular Partitions

Each inner-loop iteration of the Jacobi program has five memory references, four additions, and a division. The total grid size was 128x128 double precision elements and the program was run on 16 processors, each one operating on a 32x32 submatrix. We ran this program on page sizes ranging from 32 bytes to 4-Kbytes and with SVM and HVM. The results are shown in Figure 9. The page size chosen by our compiler's heuristic was 128 bytes.

Given the same page size for both, the straight overhead of SVM over HVM is the difference between the two curves. For the 128 byte page size the compiler chose, the SVM time is 32% greater. We note that the results include the cost of cache misses on the page table entries in the software scheme.

Of course, this is for an idealized HVM system that supports 128 byte pages without TLB misses. If we compare the 128-byte page size using SVM to a more realistic 4-Kbyte hardware page size (still, with zero TLB faults), the overhead drops to 7%.

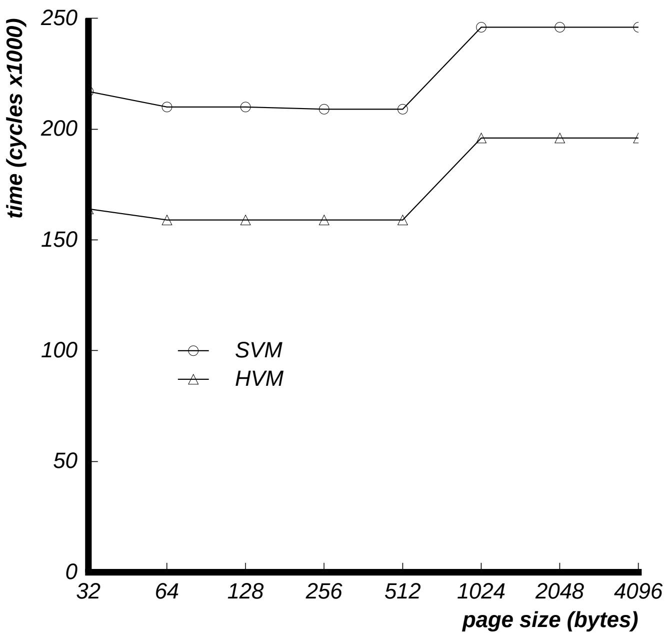


Figure 9: Running times for Jacobi with different page sizes.

In the figure, the decrease in performance when we go from 512 to 1024 byte page sizes is so large because the dimension of each processor's tile is 512 bytes. When we try to cover this with larger pages some processors experience very poor locality. Even though some processors may still have good locality, the execution time is the time of the slowest processor. Alewife actually has a farily low remote latency. Machines with higher remote memory latencies would suffer more when large page size causes poor locality.

These results also show that the overhead of software virtual memory is not very sensitive to the page size. The SVM curve is flat in the region of good locality and the runtime for 32 bytes is only 3% more than the runtime for larger page sizes. Large page tables are not a big concern because even with a 128-byte page size used for all program data, and one word per page table entry, only 3% of the memory will be used by the page table.

5.1.2 Parallelogram Partitions

We also ran a synthetic application (pgram) that requires a parallelogram data partition to get good locality. The data accessed by each processor was about 32x32 double precision as in Jacobi. The count of operations was roughly the same but with no divisions. The results for parallelogram partitions are shown in Figure 10. These results show that, as expected, smaller page sizes are more important for parallelogram partitions. In this case the SVM page size of 32 bytes gave the best performance, and actually had better performance than the HVM with a 4-Kbyte page size. Compared to Jacobi, the steep dropoff in performance happened at 512 instead of 1024 bytes because smaller tiles are required to accurately cover a parallelogram.

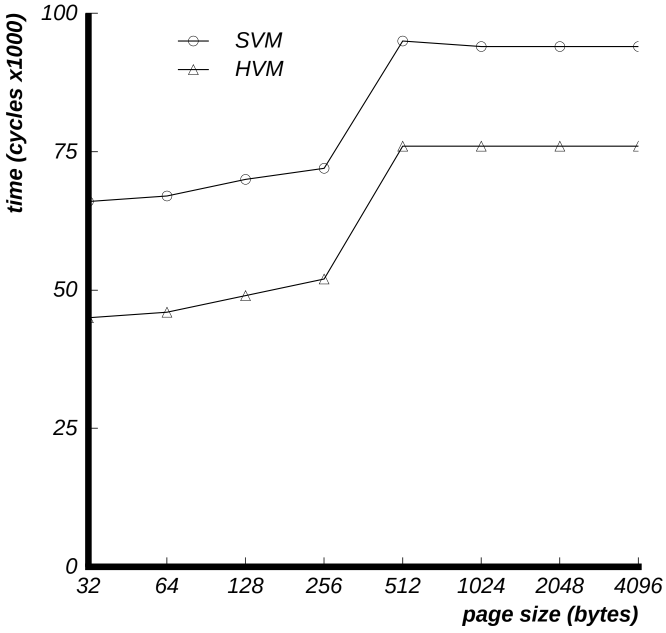


Figure 10: Running times for pgram with different page sizes.

5.2 Comparison to Software Blocking

Because software blocking is normally used on multicomputers with no shared address space, we cannot make a direct comparison of runtimes. We just note that for multicomputers, good performance is only possible if remote references can be aggregated into large messages. This will only be possible it the data tiles are large and rectangular.

We have not implemented an optimized version of software blocking for shared address-space machines but simply observe that the data locality using SVM will always be at least as good as in software blocking and the addressing overhead will always be smaller. How much difference will depend on the application. For example, in the Jacobi case with a 128 byte page size, there was about a 50000 cycle difference between SVM and HVM that is due to the address overhead. Thus each cycle of address computation that software blocking imposes would add an extra 10000 cycles (because the SVM overhead is 5 cycles per reference).

6 Conclusions

The performance of multiprocessors with physically distributed memory depends greatly on the data locality in applications. The goal of this research has been to provide a method to address distributed data automatically, while providing good data locality and low addressing overhead. This method, software virtual memory, is a significant improvement over previous methods of data partitioning. The shape of data tiles can be closely approximated by using small one-dimensional pages resulting in good data locality.

We have implemented software virtual memory in a compiler for shared address-space machines with distributed memory. Simulations of one such machine, Alewife, indicate that the addressing overhead is modest when compared to an idealized hardware virtual memory, and insignificant when compared to hardware virtual memory with realistic page sizes. In the special case of simple rectangular tiles, a straightforward code transformation similar to loop invariant hoisting can reduce the overhead to almost zero.

In summary, software virtual memory can have several advantages over hardware virtual memory or blocking.

- Software virtual memory has a modest array access overhead compared to hardware virtual memory, but it results in better locality. As shown in section 5, the improved locality afforded by software virtual memory results in roughly the same performance as hardware virtual memory for small machines. We expect its performance to surpass that of hardware virtual memory for machines that are larger than the ones we simulated, where remote memory access costs are greater.
- Software virtual memory is more efficient than simple blocking of data, as argued in section 4.
- Software virtual memory provides an illusion of continuity of the data space (over blocking or other direct calculation methods), which allows the user pointer arithmetic on virtual addresses. Blocking fragments the data space.
- Software virtual memory can be used for any complex data tiling pattern with no extra overhead.
- Software virtual memory could possibly be used to dynamically allocate distributed data.

In the future we would like to look more closely at the question of the general importance of data partitioning taking prefetching and architectural issues into account. We would also like to run our experiments for larger data sets.

References

[1] S. G. Abraham and D. E. Hudak. Compile-time partitioning of iterative parallel loops to reduce cache coherency traffic. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):318–328, July 1991.

[2] Anant Agarwal, David Chaiken, Godfrey D'Souza, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, Dan Nussbaum, Mike Parkin, and Donald Yeung. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991. An extended version of this paper has been submitted for publication, and appears as MIT/LCS Memo TM-454, 1991.

[3] Anant Agarwal, David Kranz, and Venkat Natarajan. Automatic Partitioning of Parallel Loops for Cache-Coherent Multiprocessors. In *22nd International Conference on Parallel Processing*, St. Charles, IL, August 1993. IEEE. A version of this paper appears as MIT/LCS TM-481, December 1992.

[4] Anant Agarwal, John Kubiatowicz, David Kranz, Beng-Hong Lim, Donald Yeung, Godfrey D'Souza, and Mike Parkin. Sparcle: An Evolutionary Processor Design for Multiprocessors. *IEEE Micro*, 13(3):48–61, June 1993.

[5] Saman P. Amarasinghe and Monica S. Lam. Communication Optimization and Code Generation for Distributed Memory Machines. In *Proceedings of SIGPLAN '93, Conference on Programming Languages Design and Implementation*, June 1993.

[6] Jennifer M. Anderson and Monica S. Lam. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In *Proceedings of SIGPLAN '93, Conference on Programming Languages Design and Implementation*, June 1993.

[7] J. Ferrante, V. Sarkar, and W. Thrash. *On Estimating and Enhancing Cache Effectiveness*, pages 328–341. Springer-Verlag, August 1991. Lecture Notes in Computer Science: Languages and Compilers for Parallel Computing. Editors U. Banerjee and D. Gelernter and A. Nicolau and D. Padua.

[8] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5:587–616, 1988.

[9] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD Distributed Memory Machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[10] F. Irigoin and R. Triolet. Supernode Partitioning. In *15th Symposium on Principles of Programming Languages (POPL XV)*, pages 319–329, January 1988.

[11] K. Knobe, J. Lukas, and G. Steele Jr. Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, 1990.

[12] C. Koelbel and P. Mehrotra. Compiling Global Name-Space Parallel Loops for Distributed Execution. *IEEE Transactions on Parallel and Distributed Systems*, October 1991.

[13] J. Ramanujam and P. Sadayappan. Compile-Time Techniques for Data Distribution in Distributed Memory Machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):472–482, October 1991.

[14] Anne Rogers and Keshav Pingali. Process Decomposition through Locality of Reference. In *SIGPLAN '89, Conference on Programming Language Design and Implementation*, June 1989.

[15] Madhusudhan Talluri, Shing Kong, Mark D. Hill, and David A. Patterson. Tradeoffs in Supporting Two Page Sizes. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[16] M. Wolfe. More Iteration Space Tiling. In *Proceedings of Supercomputing '89*, pages 655–664, November 1989.