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

Rajeev Barua*
MIT Laboratory for Computer Science
Cambridge, Massachusetts 02139

1 Introduction

Loop and data partitioning for shared distributed memory multiprocessors has been studied by many researchers[1]. *Loop partitioning* distributes iterations in nested loops accessing data arrays among processors to get maximum cache data reuse, keeping good load balance. For NUMA machines, *data partitioning* tries to place data where it is likely to be accessed locally. This tiles the data space among memory modules of the processing nodes. Data partitioning complicates addressing as tiles become discontiguous in physical memory.

The two current methods for finding physical addresses are *hardware virtual memory*(section 3), and *software address computation*(section 2). While common, both have drawbacks. This paper introduces a new method, *software virtual memory*, (section 4) that combines hardware virtual memory's efficiency with software address computation's flexibility. We have implemented the new scheme for MIT Alewife, a globally cache-coherent distributed-memory multiprocessor, using the partitioning scheme in [1].

2 Software Address Computation

Addressing an element is finding its physical address, specified by a processor number and offset in that processor's memory. In a shared memory machine this information is contained in one global address. Here we discuss software address computation, a current addressing method. This usually involves linearizing the data tile points, 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 addressing. Processor numbers are assigned in row or column major order, as are the offset numbers within a block. 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 on the above giving the processor number.
3. n subtractions and one row-major to find the block offset.
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.

*E-mail: {barua}@lcs.mit.edu. The research reported on herein was supervised and co-researched by Professor Anant Agarwal and Dr. David Kranz. It was funded in part by NSF grant # MIP-9012773, in part by ARPA contract # N00014-91-J-1698, and in part by a NSF Presidential Young Investigator Award.

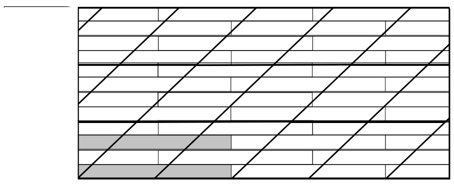

Figure 1: Loss of locality due to large hardware pages.

Footprint analysis may allow loop-invariant code hoisting of steps 1 and 2. Strength reduction may be possible for step 3. For hyperparallelepiped tiles, often needed for good locality with affine array index functions, the steps are much more expensive, but we will not go into details. The code for software virtual memory, shown in section 4 will be seen to be significantly simpler.

3 Hardware Virtual Memory

Another addressing scheme is hardware virtual memory(HVM). In a shared memory machine, HVM allows arrays to be distributed pagewise. Pages are allocated to different processors, each on the processor with maximum overlap with it. In this paper we are concerned only with address translation provided by virtual memory and not with backing store or protection issues.

HVM suffers in that most machines have page sizes too big to allow linear pages to approximate multidimensional data tiles unless the tiles are very wide. HVM deals with data movement through paging and a large page size is needed to amortize large I/O costs. Figure 1 shows how HVM pages can be used to cover the tiled array from the previous example. The shaded pages are allocated to a single processing node. We see that large pages poorly approximate data tiles, resulting in poor locality.

4 Software Virtual Memory

In Software virtual memory(SVM), given a loop and data partitioning, the compiler *stripes* the data array with small pages as indicated in Figure 2. 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 on the processor corresponding

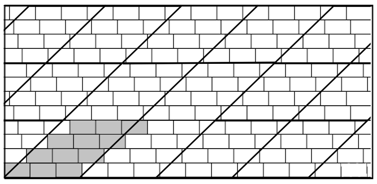

Figure 2: Approximation of a parallelogram data tile by small software pages.

to the data tile having maximal overlap with the page. For efficiency of translation, page sizes are required to be a power of two. The pagemap is used at load time to construct a page table in memory. 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, and are overhead beyond the normal index computation on a uniprocessor.

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

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

For Alewife, the above code is an overhead of only 5 cycles per memory access, assuming cache hits for all instructions and the page table lookup. We expect the cache hit rate to degrade little even for small page sizes, as the page table entries are small compared to a page. For example, the software page tables for 128-byte pages is less than 5% of the data size. For rectangular data tiles, a simple transformation similar to loop invariant code hoisting can be done on this code by subdividing the inner loop to iterate across a page, eliminating almost all SVM overhead.

This code sequence is always better than that for n-dimensional blocking, where an n-dimensional calculation is needed to obtain the block number and offset. Thus, SVM can better approximate arbitrary tiles than HVM and is more efficient than blocking.

5 Experimental Data

We present results versus HVM only, not software address computation. For reasons given earlier, SVM's data locality will always be the same as the latter's, and the addressing overhead smaller.

We ran a Jacobi iteration for rectangular tiles(figure 3), and a synthetic application (pgram) for hyperparallelepiped tiles(figure 4) on the simulator for the Alewife machine, for 128x128 arrays. Because the overhead and locality of memory references in large programs is likely to be similar, these results should be representative. 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 for SVM out of the inner loop was not performed.

Given the same page size for both, the straight overhead of SVM over HVM is the difference between the two curves. For Jacobi, at

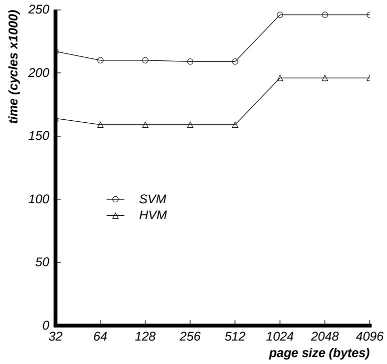

Figure 3: Running times for Jacobi with different page sizes.

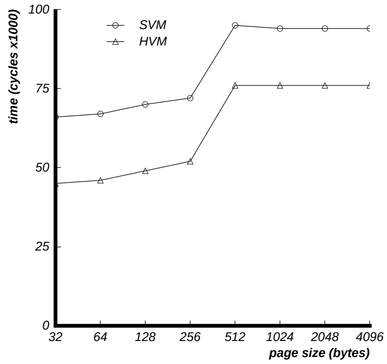

Figure 4: Running times for pgram with different page sizes.

the 128 byte page size the compiler chose, the SVM time is 32% greater. However the relevant comparison here is between large (real 4k byte) HVM pages vs small SVM pages, when the overhead drops to 7%. We note that the sharp dropoff in performance in both graphs, beyond 512 bytes for Jacobi, and 256 bytes for pgram, beyond which pages are too big to closely approximate the data tiles, 512 bytes wide here. The dropoff is earlier in pgram as smaller tiles are required to accurately cover a parallelogram.

For pgram, SVM (32 bytes selected) outperformed HVM(4K bytes), because of HVM's poorer locality on parallelogram tiles.

6 Conclusions

- SVM has modest overhead over HVM, but better locality. Improved locality results in comparable performance as HVM for small machines. We expect performance to surpass HVM's for machines with higher remote latencies, and also larger versions of Alewife(which have higher latencies).
- SVM is faster than blocking, as argued in section 4.
- SVM addresses complex tile shapes with no extra overhead.
- SVM could possibly allow dynamic global data allocation.

References

[1] A. Agarwal, D. Kranz, and V. Natarajan. Automatic Partitioning of Parallel Loops for Cache-Coherent Multiprocessors. In *22nd Intl Conf on Parallel Processing*, August 1993.