Memory Access in Shared Virtual Memory*

Rudolf Berrendorf †
Zentralinstitut für Angewandte Mathematik
Forschungszentrum Jülich (KFA)
Postfach 1913
D-5170 Jülich
Federal Republic of Germany
R.Berrendorf@kfa-juelich.de

ANL/CP--76324

DE92 019594

Abstract

Shared virtual memory (SVM) is a virtual memory layer with a single address space on top of a distributed real memory on parallel computers. We examine the behavior and performance of SVM running a parallel program with medium-grained, loop-level parallelism on top of it. A simulator for the underlying parallel architecture can be used to examine the behavior of SVM more deeply. The influence of several parameters, such as the number of processors, page size, cold or warm start, and restricted page replication, is studied.

1 Introduction

Several basic models, or paradigms, exist for programming parallel machines, most of which are related to real machine models (e.g., shared-memory model, message-passing model, data-flow model, or graph reduction). A relatively simple model for parallel programs is the shared-memory model, where all processors operate on one (flat) shared memory. In this model all processors have the same view of memory: immediately after a write executed by one processor, all the other processors can access this memory location with the new value (*strong coherence*). Mapping this model onto shared-memory hardware would be easy if the hardware ensured proper coherence efficiently by itself. But the problem with shared-memory hardware is its scalability. Bus-based systems (usually with *snoopy* cache protocols to enforce coherence in multiple caches), which solve the problem for a limited number of processors, are restricted by bus bandwidth. Approaches to tackle this problem are NUMA-systems (Non-Uniform Access Architectures, e.g., NYU Ultracomputer [Got86], IBM RP3 [PBG+85], and BBN TC2000 [Inc90]), where a performance penalty up to an order of magnitude exists for remote accesses, as well as multicache systems [LLG+90], where data is replicated to local caches and consistency is ensured by special cache protocols (e.g., directory-based protocols [CF78], [ASHH88]). Because of complexity restrictions with hardware implementations, these protocols have to be kept simple.

One way to achieve hardware scalability is to use distributed-memory computers (e.g., Intel's Paragon XP/S, nCUBE's NCUBE 2, or Thinking Machine's CM-5). Such computers, however, do not hide distribution of data from the programmer. (An exception is the array data type in CM-FORTRAN.) Every remote data access has to be programmed explicitly, and automatic compiler generation of correct and efficient communication statements to access nonlocal data [ZBGH86][CK88] is difficult. Other problems with this model are process migration and the passing of pointers or complex data structures between distinct address spaces.

Hardware scalability and ease of programming may be reconciled by the use of a virtual memory layer with a single address space on top of a distributed real memory (see Figure 1). Such a configuration gives the user and compiler the appearance of shared memory with a single address space, analogous to the way virtual memory hides restrictions of real memory from the programmer. This software layer, which is

*This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

†This work was performed when the author worked as a visiting scientist at the Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill.

DISCLAIMER

called *shared virtual memory* (or SVM) [Li86], replicates memory pages for performance reasons and ensures proper coherence between copies through special protocols.

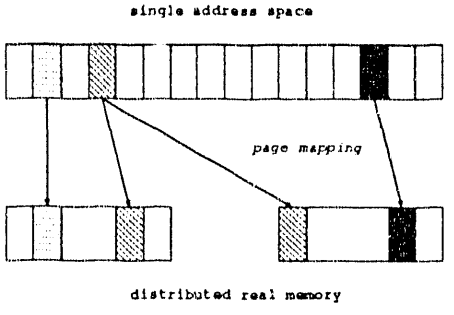


Figure 1: Page mapping in SVM

While fine-grained parallelism on the statement or basic block level is handled more efficiently with superscalar or pipelining CPUs, coarse-grained parallelism (e.g., multitasking on the outer subroutine level) is usually difficult to manage by programmers or by parallelizing compilers. In our study we are concerned mainly with medium-grained parallelism as, for example, expressed in the outer loops of computationally intensive kernels.

In this paper we examine the behavior of software-controlled memory coherence mechanisms with medium-grained parallelism. We consider the influence of several parameters, including the number of processors, page size, cold or warm start, and restricted page replication. To overcome the restrictions of a concrete hardware, we simulated a simple abstract parallel machine which accepts as input a parallel program trace and emulates the memory behavior of an SVM.

The paper is structured as follows. After giving a brief overview of the memory coherence problem, we describe our abstract machine model and the simulation process. In Section 4 we present our simulation results, in Section 5 we discuss related work, and in Section 6 we offer some concluding remarks about future extensions to our work.

2 Memory Coherence

The basic problem in memory systems with possible data replication (e.g., multicache systems, shared virtual memory) is keeping memory coherent. A memory is called *strongly coherent* [CF78] if the value returned by a read operation is always the same as the value written by the most recent write operation to the same address. A relaxation of this is a *weak coherent* scheme [DSB86], where memory consistency has to be ensured only at synchronization points.

The problem of memory coherence first came up with multicache systems. In the directory scheme proposed by Censier and Feautrier [CF78], a bit vector held in a centralized directory represents which caches hold copies of cache lines. In order to reduce memory, a restriction of this scheme was proposed by Agarwal et al. [ASHH88], where only up to i simultaneous data copies are allowed and directory pointers refer to actual copy holders. If these pointers are exhausted, copies must be invalidated. Agarwal et al. gave two variations of this idea: the Dir_iB- and Dir_iNB-schemes (*Dir* for Directory), the former invalidating all i copies through a broadcast message (*B* for broadcast), the latter sending individual invalidation messages to copy holders (*NB* for no broadcast). Usually, hardware approaches in such multicache systems are limited by complexity restrictions enforcing easy protocols.

Another approach to keep caches coherent is to use software [CV90] [CKM88]. For example, Cheong and Veidenbaum proposed a compiler-directed cache management where appropriate cache invalidation commands are generated by compilers. But such approaches are restricted to multiprocessors with shared memory and private caches and thus offer no real solution in avoiding shared memory in hardware. Also, in the absence of exact information, software-controlled cache invalidation has to be conservative.

3 Simulation of SVM

Examining the behavior of shared virtual memory on real machines is restricted to system parameters such as hardware page size or compiler and linker assistance in generating parallel programs. To study the behavior of SVM in detail, we have implemented a simulator that models the behavior of SVM on an abstract parallel machine. We have abstracted from the concrete hardware because we are interested in memory behavior and memory performance on the level of reference counts and page faults rather than cycle times on a specific hardware implementation. However, the results of our work provide input for work in this field, too. The machine model we introduce is a good compromise between simplicity and accuracy. The simulation process is divided into two parts. In the first step

(see Figure 2) an appropriate parallel program trace is generated that is used as input for the simulator. In the second step the simulator takes this trace and several parameters and simulates the behavior of shared virtual memory on an abstract parallel machine.

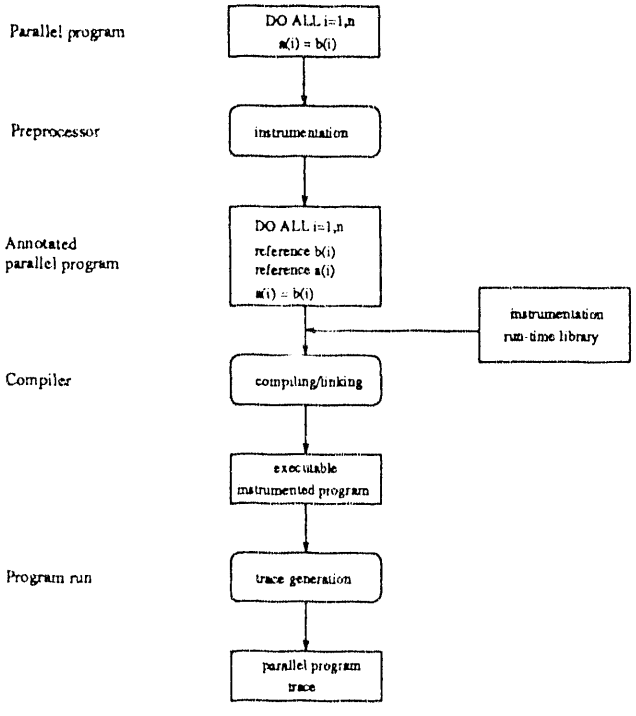


Figure 2: Trace generation process

The first step in the simulation process is to produce an appropriate memory and parallelism trace of a program. Generating accurate parallel program traces is difficult, as the behavior of these programs often critically depends on the ordering and timing of events such that every delay (e.g., for performance information gathering) can influence the overall program behavior. The traces we gather have no time stamps; rather, the ordering of events as memory accesses and parallel events is important and is preserved. In the absence of exact timed references, these traces are more suited for regular problems without racing conditions.

To generate a trace, one neeeds a program where parallelism is controlled through parallel loops, parallel regions, critical sections, and barrier synchronization, similar, for example, to PCF-Fortran [Par88] or Fortran extensions of several vendors of shared-memory parallel computers. The parallel program is run through a preprocessor [Ber88] which generates an annotated version of the program. This program is compiled and linked with appropriate run-time support libraries and generates at execution time a trace of the memory reference behavior interspersed with parallelization information. The parallel program trace gives memory access information, such as referenced memory address, access type (read or write access, access to private or shared location), size of referenced location, and a back-reference to the accessed variable. In this study we have instrumented only subscripted variables, since most scalar variables usually can be held in registers or private memory locations.

The abstract machine model of our simulated parallel computer has N units, each with a processor, (unlimited) private memory, an MMU (Memory Management Unit), and a communication processor; we call each of this units a *node*. One processor is the *master processor* executing all serial code. If all processors executed the serial code, unnecessary access conflicts would be generated. Each processor has an input queue with memory references to interpret; processor steps are performed along the queues in a round-robin fashion. If the simulator encounters in the memory trace the beginning of a parallel loop or parallel section, the distinct iterations or sections are split to the input queues of the processors, which are determined with a parameterized scheduling algorithm. After each parallel loop or region, a barrier synchronizes all processors that have participated in the loop. If a processor reaches a critical section, all other processors wishing to enter this section are blocked.

We have restricted the input language to loops where the number of iterations is known on entry of the loop; jumps out of the loop are prohibited. This approach as well as the specific model of how loop iteration are spread over nodes helps us explain the cost model given below, without assumptions about synchronization hardware.

Parallel loop iterations and parallel region cases are scheduled statically. The set of all processors is subdivided into intervals of the form $I = [p_1, p_n]$ (initially one interval $[1, N]$) to which iterations or cases will be spread. The *loop master processor*, the first processor in an interval (initially the *master processor*), initiates parallel execution of other processors in the interval by sending them the interval (e.g., numbers of first and last processors) and the number of iterations to handle. With this information, all processors in the interval are able to decide which subinterval they belong to and which iterations they should work on, or whether they have no work at all (if there are more processors than iterations). For l iterations and

N processors,

$$n_i = max(1, \lfloor \frac{(i+1) \times N}{l} \rfloor - \lfloor \frac{i \times N}{l} \rfloor - 1), \quad (1)$$

consecutive processors form subinterval i. With this model in mind, the costs of initiating a parallel loop are the costs of sending the above information from the *loop master node* to the subinterval. This task can be done in time $O(log_2(n))$ (without broadcast) until each processor has the information, where n is the number of participating processors in the subinterval. Technically, we distribute these costs as if all participating processors immediately get this information and then synchronize for the given time. Similarly, if a barrier synchronization has to be done at the end of a loop or region, the costs on each processor are $O(2 \times log_2(n))$ after all processors have reached the synchronization point (which correspond to posting and acknowledging of synchronization in a tree-like fashion). Actually, this cost model is conservative, as not all processors reach the synchronization point at the same time. Thus, posting could be started earlier.

In our implementation of SVM, we used an algorithm with a *write-invalidate*-based protocol and a strong coherence scheme similar to Li's dynamic distributed scheme [Li86], which itself is based on the Berkeley Ownership scheme [KEW+85]. In this concept, every page has an owner who maintains a copy set with all nodes that currently have a read-only copy of the page. On a page fault, the faulting page is requested from the current owner, who can be reached over a chain of probable owners. For write page faults, the ownership changes to the requester, who invalidates all copies before writing to the page.

In our model, page faults and invalidation messages block the faulting node for a specific time (simulation parameters t_{wait} and t_{inval}; see below) while serving nodes (e.g., page owners sending the page to the faulting node) are not blocked at all; requests, for instance, are handled by the communication processor.

As a starting partition of the virtual memory, all memory pages are spread uniformly over all available nodes such that page p is located on node p *mod* N.

3.1 Simulation Input Program

As the input program for our simulations we used a parallel matrix multiply as given in Figure 3 for a square matrix of size $n \times n$ where $n = 64$; the two outer loops were parallelized. It was not our purpose to find a very efficient version of the matrix multiply for SVM; rather, we were interested in an application program with reasonable memory references as an input for our simulator. We chose matrix multiply, because it is memory intensive, with significant data reuse as well as potential parallelism.

There exist six major versions of matrix multiply dependent on the loop order [GvL89]. We have chosen the *jik*-version (named after the loop order) such that write references to the result array are on contiguous memory locations. We have specified that loop iterations be spread in blocks over all available processors (as opposed, e.g., to interleaved spreading).

```
DO ALL j=1,n
  DO ALL i=1,n

    tmp = 0.0
    DO k=1,n
      tmp = tmp + b(i,k) * c(k,j)
    END DO
    a(i,j) = tmp

  END DOALL
END DOALL
```

Figure 3: Parallel matrix multiply

The memory space for each array a, b, c is 4096 words; each iteration of the inner parallel loop has a total of 129 memory references to subscripted variables (128 read accesses, 1 write access); the whole program has 528,384 memory accesses to subscripted variables. We will use in our simulations between 4 and 1024 processors such that the granularity for a parallel task in the inner parallel loop will be between $64 \times 129 = 8256$ and $4 \times 129 = 516$ memory references.

3.2 Simulation Parameters

In the following sections, we express all execution times in multiples of *memory ticks*, where one memory tick should be seen as the cost to access one word in local memory on a node. Although we will not specify a concrete value, the relation of all following values to this reference value is sound.

We have chosen a page fault wait time t_{wait} (in memory ticks) as given in Equation 2, where t_{fault} are costs for page fault handling, $n_{copy}(p)$ is the number of nodes that hold a current copy of page p (the faulting page), r is the number of page requests along a chain

$$t_{wait} = \begin{cases} t_{fault} + (n_{copy}(p) + 1)t_{inval} & \text{if node is page owner} \\ t_{fault} + (r+1)t_{startup} + t_{send} \times s_{page} & \text{not owner, read fault} \\ t_{fault} + (n_{copy}(p) + 1)t_{inval} + (r+1)t_{startup} + t_{send}(s_{page} + n_{copy}(p)/2) & \text{not owner, write fault} \end{cases} \quad (2)$$

of probable page owners, $t_{startup}$ are startup costs for communication, t_{send} are costs for sending one word between two nodes, and s_{page} is the size of a page in words. If the page is found on the faulting node (e.g., write-fault on a node that is owner and has a read-only copy of that page), the costs are t_{fault}. Any invalidation message sent to another node and waiting to be acknowledged has additional costs of $2 \times t_{inval}$; multiple invalidations can be pipelined. If the faulting node is not the page owner, a page request is sent along the chain of probable owners; the owner node sends the page and on write faults in addition the copy set. We assume that two node identifiers can be packed into one word. During a page fault, the processor waits and is not able to, say, synchronize.

Loop startup costs are $log_2(n) \times t_{sync}$; barrier synchronization costs after a loop finishes are $2 \times log_2(n) \times t_{sync}$, where n is the number of participating nodes. We distinguish between an unlimited number of read page copies and a restriction to a fixed limit after which page copies have to be invalidated before a further page copy can be send. We ran the simulations for N = 4, 16, 64, 256, and 1024 processors. Table 1 shows the actual parameter values we have chosen in our simulations.

Table 1: Basic parameter values.

t_{fault}	50 ticks	fault startup time
$t_{startup}$	50 ticks	startup time for send
t_{send}	2 ticks	time to send one word
t_{inval}	60 ticks	page invalidation time
t_{sync}	60 ticks	basic synchr. time
s_{page}	4-1024	page size in words
N	4,16,64,256,1024	number of nodes

4 Simulation Results

We distinguish between two types of initial page and data distributions. The first type, which we call *cold start*, resembles the behavior of applications without any predistribution of data or programs with very different types of data access pattern between distinct program phases. In this type of simulation each page exists on program startup on one node, which has write access to the page and is also owner of it. Initially all pages are distributed uniformly over all nodes in an interleaved scheme as already described.

The second type of initial configuration, which we call *warm start*, is similar to an application where a kernel is called inside an application program and former parts of the program have similar data access patterns. To get a realistic page distribution, we run the simulation twice. The final page distribution of the first run gives the start distribution for the second run, which gives the overall result for the *warm-start*.

Further, we distinguish between two simulation types with arrays of different dimension. In the first type, 64×64 simulations are done with properly dimensioned arrays. In the second type, 65×65 simulations are done with arrays that are dimensioned 65×65 but for which only the upper 64×64 submatrices are used.

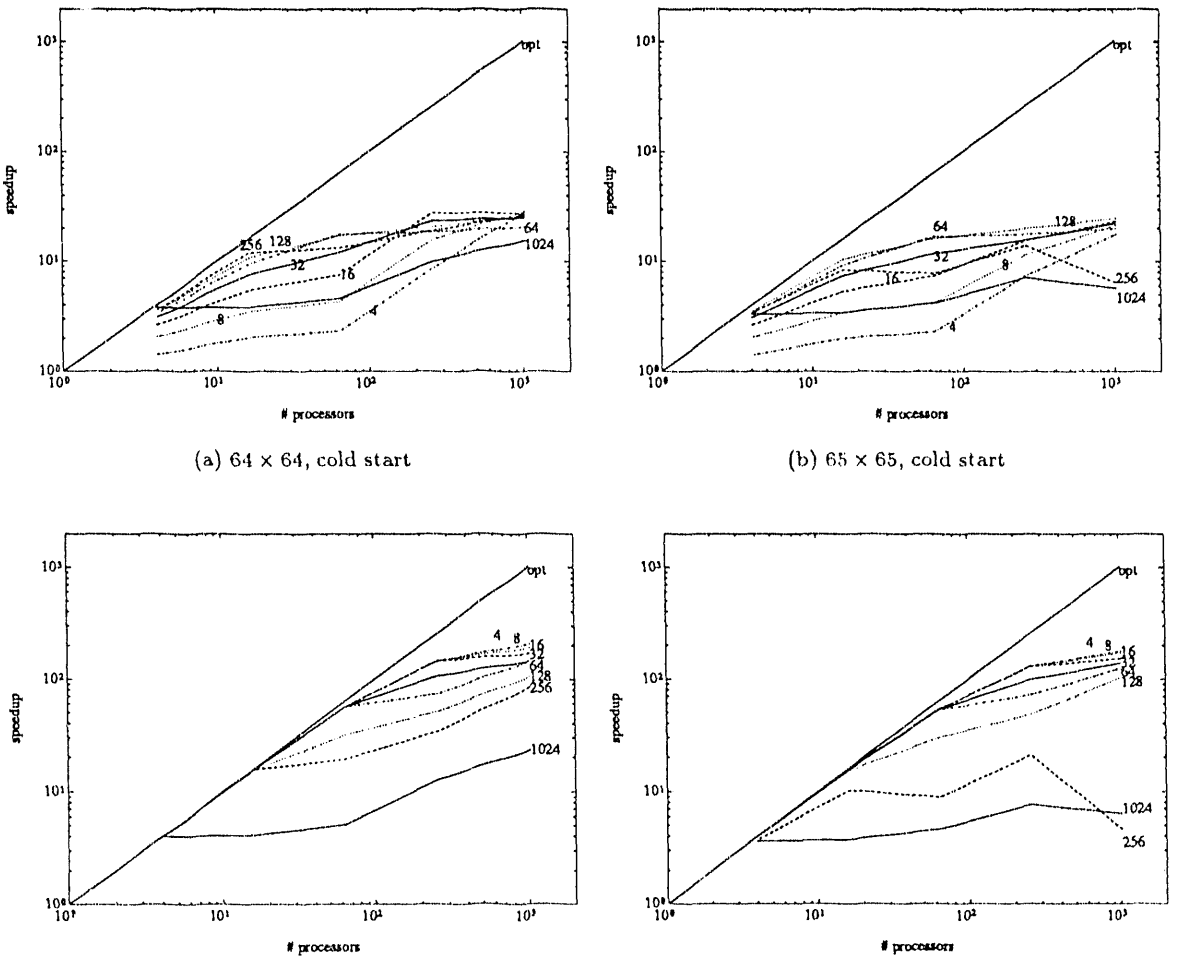
The results, as shown in the next sections, are given in a log-log scale. Each line represents one data set for a specific page size (in words). We refer to results given in the next section as the *base case*.

4.1 Variation in Page Size

Page size is a critical parameter in the design of a memory system; if this size is too large, contention effects and false sharing prohibit good performance. On the other hand, page fault latencies and communication startup costs mostly dominate page-fault handling in software-controlled memory systems. Thus a fair compromise has to be chosen between reducing contention and avoiding unnecessary page faults. While page fault latency and overhead are system parameters and usually independent of an application, contention and false sharing problems critically depend on given access patterns. Figure 4 shows speedups[1] for several page sizes and number of processors for given parameter values and input program.

Most of the execution time with small page sizes in the cold-start model is spent in getting pages the first time

[1] Speedup values are total execution times (in memory ticks) of the parallel version related to memory ticks of the sequential version, which is the number of memory references to subscripted variables.



(a) 64 × 64, cold start

(b) 65 × 65, cold start

(c) 64 × 64, warm start

(d) 65 × 65, warm start

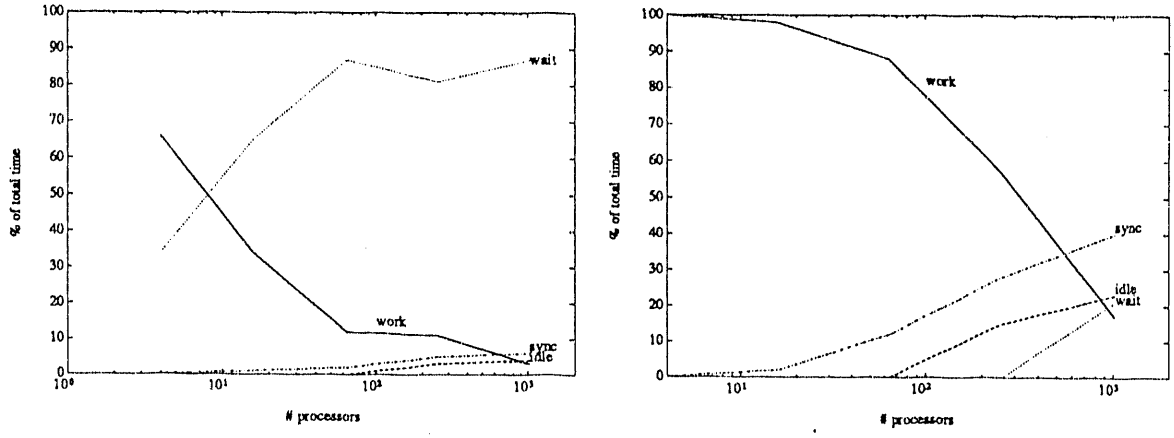
Figure 4: Base case

(high number of startup overheads), because the initial data distribution, uniformly spread over all nodes, is different from the initial demand of data on nodes. Since the working set for matrix multiply is relatively small because of reuse of data, cold-start effects would get even worse for larger working sets and an increasing number of processors. While small page sizes have this cold-start effect, large page sizes (1024 words) have contention problems as the number of processors increases.

In the warm-start model, speedup results are up to 23 times higher than cold-start results. The reason is that, since all nodes already have read-copies of the b- and c-arrays (final page distribution of the previous), these pages do not need to be fetched. Problems preventing higher speedups are contention and false-sharing for write-array a, as well as synchronization overhead on start and exit of parallel loops. While synchronization overhead is due to the abstract machine model we have chosen, the other problems are affected by large page sizes.

For the 65×65-dimensioned matrix, unnecessary false-sharing occurs, since matrix columns are not aligned on page boundaries. For pages with 256 words and larger, write array a shares a page with read array b such that on every write to this page, copies are invalidated on nodes that access affected parts of array b.

Figure 5 shows the relative amount of time (accumulated per-processor times) spent in different simulation



(a) 64 × 64, cold start

(b) 65 × 65, warm start

Figure 5: Relative amount of time for different states with $s_{page} = 16$ (base case)

states for a page size of 16 words; time types are explained in Table 2. In the cold-start model, page-fault wait times consume a significant portion of the total time; in the warm-start model, however, the work time, dominating for a small number of nodes, is (relatively) reduced by synchronization and idle times. Idle times arise if more than 64 processors are available at program start, while the outer loop gives work for only 64 processors. Processors that are not involved in running the outer loop are idle until the outer loop is spread over 64 processors and work starts for the inner parallel loop.

Table 2: Time types for different states

type	description
work	handling memory references.
wait	page fault wait times.
idle	idle times.
sync	loop startup costs and barrier synchronization.

4.2 No Postloop Synchronization

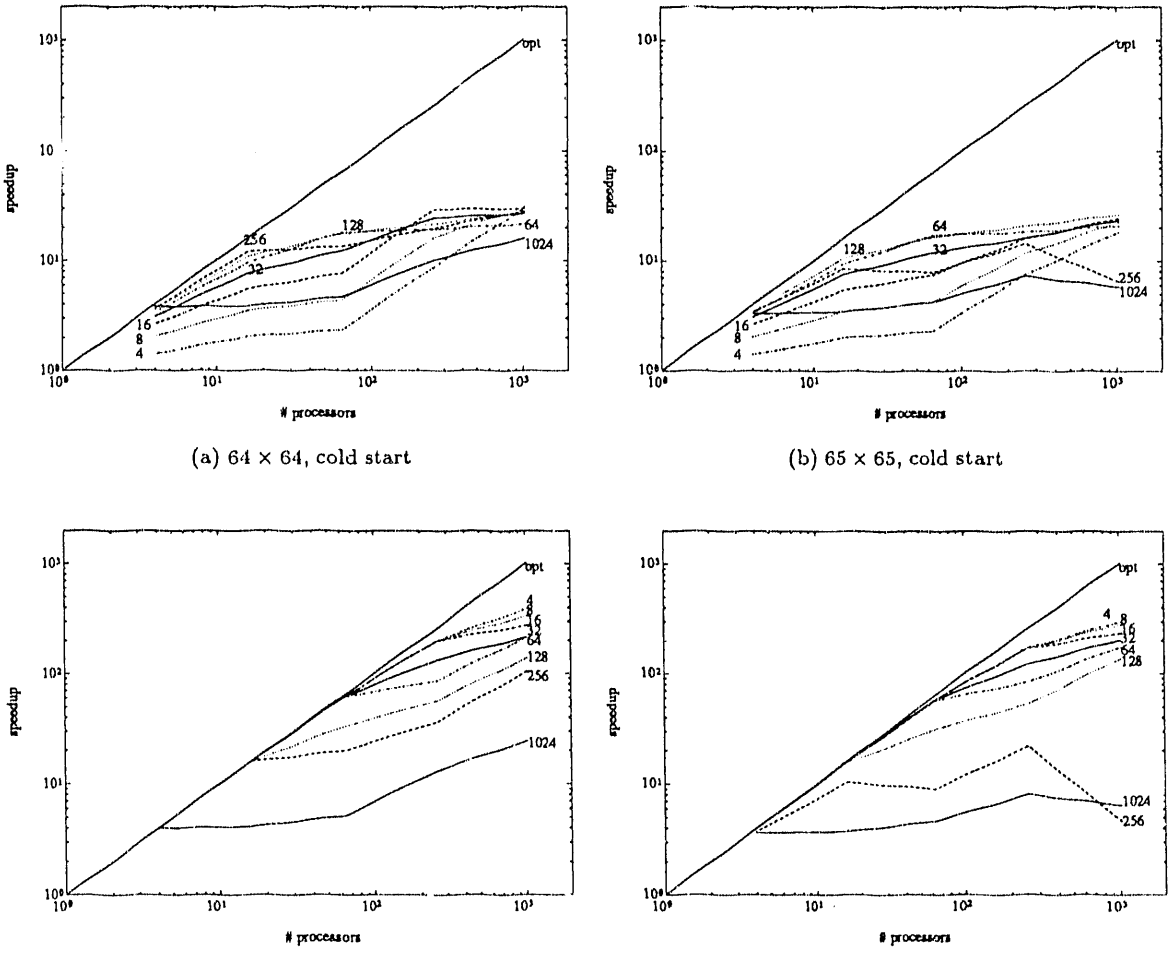
As can be seen in Figure 5, one slowdown in the program is barrier synchronization after a parallel loop ends. This problem can be avoided with our parallel matrix-multiply algorithm. As a consequence, we have specified in a different algorithm version that no barrier synchronization has to be executed on exit of a parallel loop Figure 6 show speedups for execution without postloop synchronization.

For the cold-start model and for a moderate number of processors, the effects are small, since synchronization time (startup costs for loop initiation, barrier synchronization at loop exit) is small compared to page fault times and total execution time. But for a large number of processors and small page sizes, speedup values nearly doubled as the relative amount of synchronization time increased in the base case.

The limitation for even higher speedups is startup costs for loop initiation. The total execution time on each of the 1024 processors with a page size of 4 words (the best speedup reached) in the warm-start model is 1,356 memory ticks (compared with 528,384 memory ticks in the serial case), of which 38% are working time while the rest is spent in loop startups. With our parallel execution model (parallel loop initiation time of $O(\log n)$ with n participating nodes) and chosen parameter values, these costs cannot be further reduced.

4.3 Different Access Order

In the base case (loop order *jik*), write accesses to array *a* are done in subsequent order in memory under Fortran memory mapping of arrays. As loop iterations are spread in blocks over available processors, blocks of memory are written by the same processor such that write-sharing of pages is reduced. For an *ijk* loop order, written memory locations are accessed in smaller blocks (dependent on the number of nodes) in an interleaved fashion, and thus write-sharing of pages



(a) 64 × 64, cold start

(b) 65 × 65, cold start

(c) 64 × 64, warm start

(d) 65 × 65, warm start

Figure 6: Without barrier synchronization

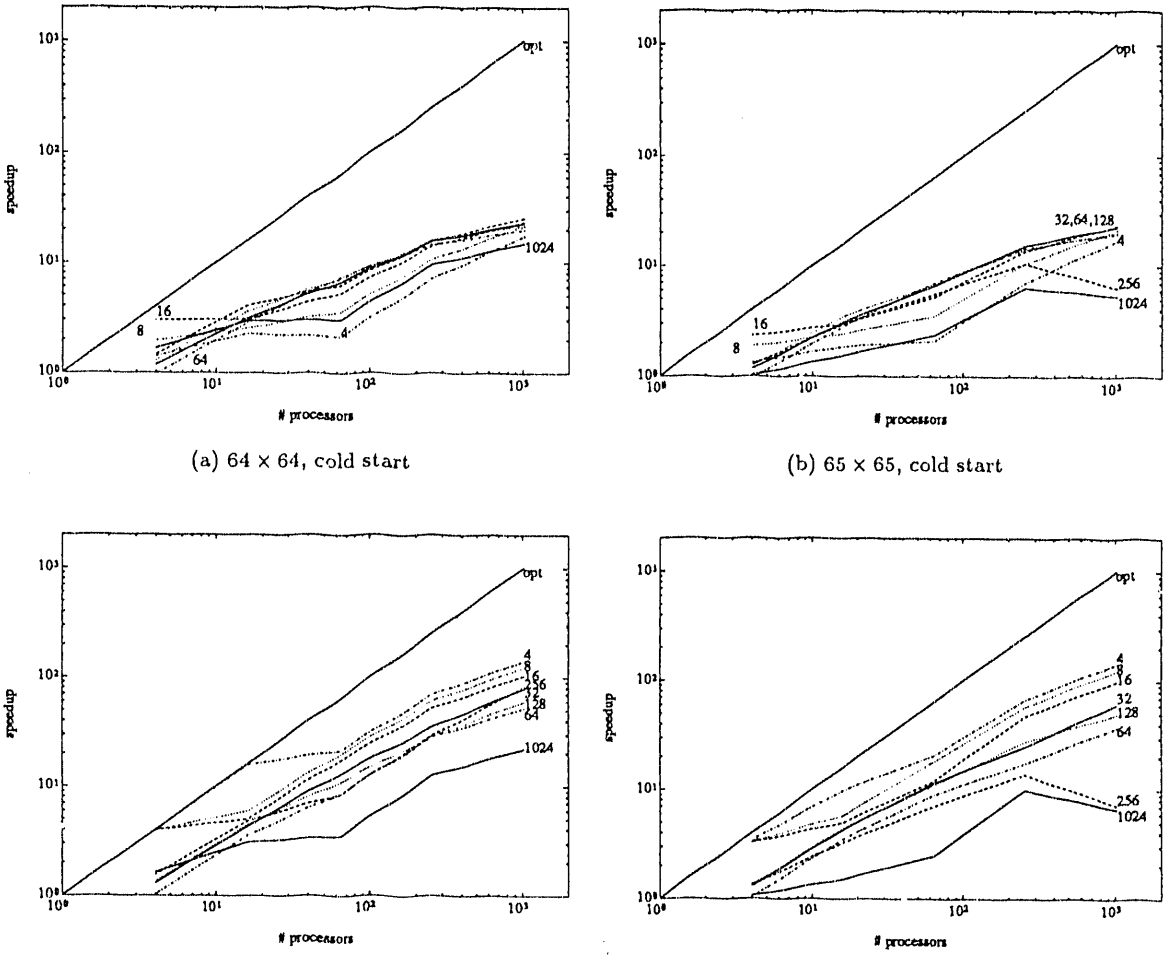
occurs more often. Figure 7 shows speedups for loop order *ijk*.

In cold-start simulation, only medium to large page sizes with a small number of nodes show a significant performance reduction compared with the base case. Otherwise, speedup numbers are greater than 60% of the base case. Although more often write accesses in loop order *ijk* are page misses as write page-sharing is enhanced, costs for write faults do not dominate total costs; total costs are mainly caused by initial page distribution, as shown already for the base case.

For warm-start simulations, 64-node speedups show a significant performance degradation for medium to small page sizes (large page sizes do not perform well in the base case, too). The reason is that now all iterations of the outer loop are spread over distinct nodes and subsequent memory locations are written by different processors.

Although, in matrix multiply, nodes share write pages but no single memory location, nearly every write access with loop order *ijk* and a large number of nodes results in a write page fault. One possible solution to this problem is a strategy shown in Myrias machines [Cor90]: distinct processors write to private page copies, which are later merged to one final page.[2]

[2]Crucial to this idea is that page merging can be done efficiently even with a large number of pages.



(a) 64 × 64, cold start

(b) 65 × 65, cold start

(c) 64 × 64, warm start

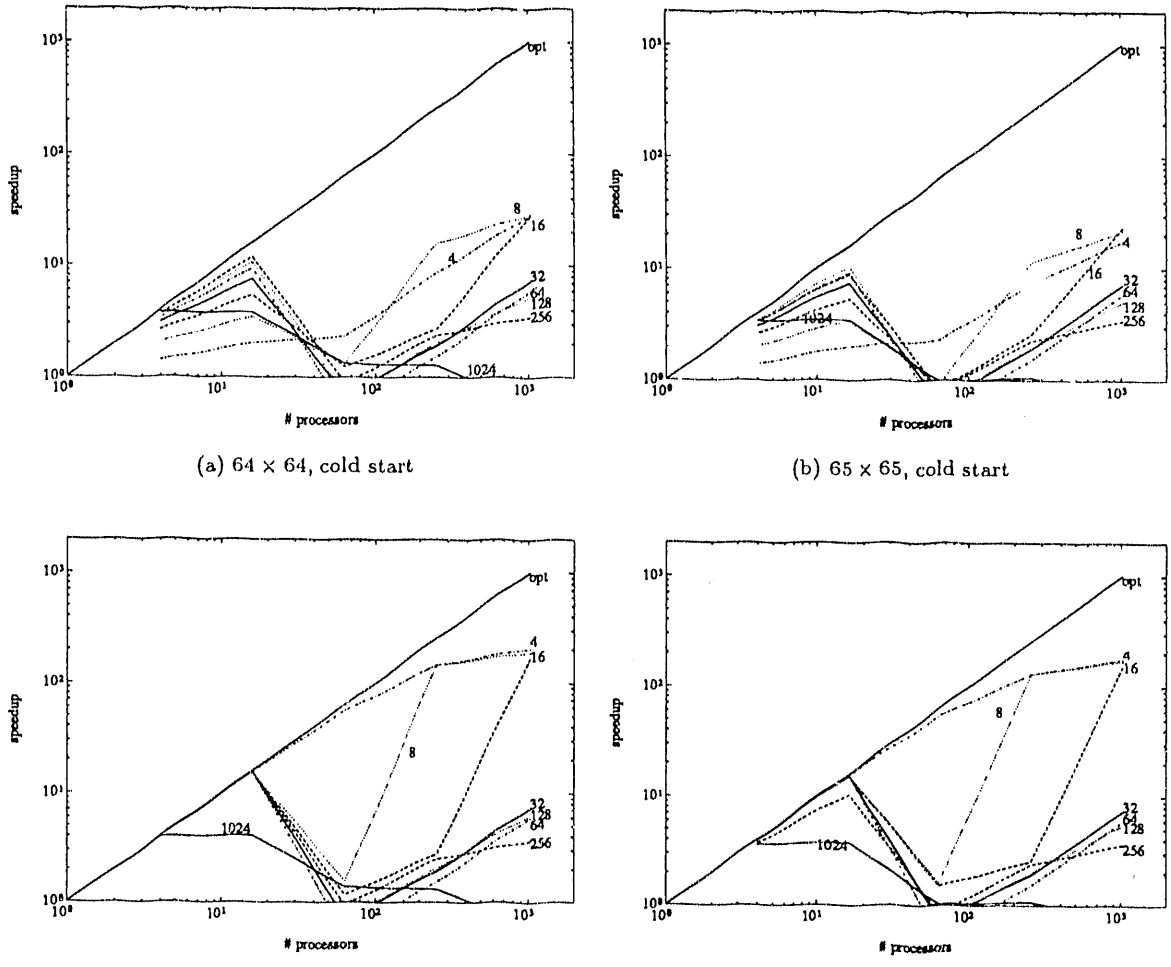
(d) 65 × 65, warm start

Figure 7: Different access order

4.4 Restriction of Page Copies

So far, we have kept track of all page copies. The space complexity for full information, as it would be implemented in a straightforward bit-vector approach with a bit-vector part of each page descriptor, is $O(MN)$ bits on each node, where M is the number of pages and N is the number of processors. For a whole system this means a quadratic increase with the number of processors. As already noted, Agarwal et al. [ASHH88] proposed with their Dir_iNB- and Dir_iB-schemes a limitation of copies, thus restricting the information to be kept on each node; after this limit is reached, copies have to be invalidated. Figure 8 shows speedups for a restriction to 32 copies; after this limit is reached, a copy is chosen randomly for invalidation. Simulations with different limitation values show similar effects.

The results show that, for our implementation of matrix multiply and with a straightforward implementation of the Dir_iNB-scheme, the limitation of read copies means a severe restriction, as all nodes need partial copies of read arrays b and c. Three sections can be seen in the figures. In the first section the number of nodes (and the number of requests for page copies) is smaller than the copy limit. Thus there is no difference from the base case. As soon as the number of nodes is larger than the copy limit, however, the system is blocked with invalidating pages, as pages are shared heavily between nodes. In the third section, with a large number of nodes, page sharing is



(a) 64 × 64, cold start

(b) 65 × 65, cold start

(c) 64 × 64, warm start

(d) 65 × 65, warm start

Figure 8: Restriction of page copies

reduced, since only smaller portions of the whole data are needed on every node. In general, restriction to a limited number of copies penalizes access patterns where many processors share a read copy, accessing data several times.

As a copy set is needed only on page owner nodes, most of the bit vectors in the straightforward implementation of page tables are not used. While dynamic allocation of memory is difficult with hardware controllers, programs are able to allocate space for bi vectors on demand. In our implementation, a bit vector is allocated if a node gets page ownership and is released on losing ownership. With this solution, the total space complexity is reduced from $\mathcal{O}(MN^2)$ to $O(MN)$.

Several other approaches allow any number of read copies without invalidations and without the space complexity of a bit vector in each page descriptor; these ideas were originally proposed for scalable multicache systems. Primarily, the differences lie in the data structures and operations used. In the *Scalable Coherent Interface* [JLGS90] double-linked lists represent copy holders of cache blocks, while the *Stanford Distributed-Directory Protocol* [TD90] is based on single-linked lists of distributed directories. Maa, Pradhan, and Thiebaut [MPT91] proposed a *tree directory* and a *hierarchical full-map directory* to keep track of copy holders.

5 Related Work

Li [Li86] [LH89] discussed the basic concepts for centralized and distributed algorithms with a strong coherence scheme and showed the practical applications of his ideas in prototype implementations on a ring of workstations and on a hypercube parallel computer. He implemented a variation of the Berkeley Ownership Protocol [KEW+85].

Bennett, Carter, and Zwaenepoel [BCZ90] give a classification for objects in distributed memory in order to handle them efficiently. In addition to the usual classes of type *private* and *general read-write*, they distinguished between frequencies (*write once, mostly read*, etc.) and special concurrency types (*synchronization*). For each class they had a special coherence treatment.

Myrias [Cor90], a computer manufacturer now out of business, implemented on their parallel machines a software layer simulating a single address space. At the beginning of a parallel loop or section, child processes are generated with their own private memory, initially a copy of the memory of the father process. At the end of the parallel loop, child memories are merged to one memory in which the father process continues execution.

Eggers and Katz [EK88] [EK89] examined effects of sharing in multiprocessor systems with invalidation-based multicaches. For their (coarse-grained) programs they gave a low percentage of shared write accesses (2% of total references) and low contention. They found that with the invalidation-based protocol, large block sizes are favorable for programs with per-processor locality, as the number of invalidations is not high.

6 Conclusions

As shown for our implementation of matrix multiply, in *cold starts* most of the execution time is spent in getting the correct initial data distribution; especially with small page sizes, these distribution costs dominate the overall performance. This startup overhead will even be worse for problems with fewer operations per data item. Matrix multiply favors *warm starts*, since after an initial run, read-only data is already distributed according to the access pattern. Data layout specifications in programs (e.g. as in Fortran-D [FHK+91] or Vienna FORTRAN [CMZ91]) can help to overcome initial distribution costs. In our model, synchronization costs limit even better performance on a large number of nodes.

For chosen system parameter values, there is no overall preference for a particular page size. For a small to medium number of nodes, medium page sizes show good performance; on the other hand, small page sizes are favorable with a large number of nodes. The reasons are that contention effects are reduced and that, with smaller granularities of parallel tasks, smaller portions of memory are accessed by each task.

A severe restriction for the parallel matrix-multiply program is a limitation for read copies as proposed in Dir_iNB- and Dir_iB-schemes. The reason is that with a heavy reuse of data and simultaneous accesses of parallel tasks to the same memory regions the demand of page copies exceeds the number of possible copies which, in turn, results in trashing.

Two possible extensions to our research are first to examine a broader range of applications with different and irregular access behavior, and second to incorporate weak coherence schemes, especially for applications with different access patterns.

Acknowledgments

I am grateful to Christian Bischof of the Mathematics and Computer Science Division, Argonne National Laboratory, for his useful comments on this paper.

References

[ASHH88] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An evaluation of directory schemes for cache coherence. In *15th Intl. Symposium on Computer Architecture*, pages 280–289. IEEE, 1988.

[BCZ90] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. *ACM Sigplan Notices*, 25(3):168–176, 1990.

[Ber88] Rudolf Berrendorf. Der FORTRAN-Parser PAFF als wiederverwendbares Werkzeug für Programmier-Tools. Technical report, Kernforschungsanlage Jülich, 1988.

[CF78] M. Censier and P. Feautrier. A new solution to the coherence problems in multicache systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[CK88] David Callahan and Ken Kennedy. Compiling programs for distributed-memory multiprocessors. *The Journal of Supercomputing*, (2):151–169, 1988.

[CKM88] Ron Cytron, Steve Karlovsky, and Kevin P. McAuliffe. Automatic management of programmable caches (extended abstract). In *IEEE International Conference on Parallel Processing*, pages 229–238. IEEE, 1988.

[CMZ91] Barbara M. Chapman, Piyush Mehrotra, and Hans P. Zima. Vienna FORTRAN - a FORTRAN language extension for distributed memory multiprocessors. Technical report, University Vienna, Department of Statistics and Computer Science, 1991.

[Cor90] Myrias Computer Corporation. *Parallel Programmer's Guide*, 2.4 edition, March 1990.

[CV90] Hoichi Cheong and Alexander V. Veidenbaum. Compiler-directed cache management in multiprocessors. *IEEE Computer*, pages 39–47, June 1990.

[DSB86] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory access buffering in multiprocessors. In *13. Annual Intl. Symposium on Computer Architecture*, pages 434–442. IEEE, 1986.

[EK88] Susan J. Eggers and Randy H. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *15th Annual Intl. Symposium on Computer Architecture*, pages 373–382. ACM, 1988.

[EK89] Susan J. Eggers and Randy H. Katz. The effect of sharing on the cache and bus performance of parallel programs. In *ASPLOS-III*, pages 257–270, 1989.

[FHK+91] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical report Rice COMP TR90079, Rice University, Department of Computer Science, March 1991.

[Got86] Allan Gottlieb. An overview of the NYU ultracomputer project. Technical Report 86, NYU, July 1986.

[GvL89] Gene H. Golub and Charles F. van Loan. *Matrix Computations*. Johns Hopkins Series in the Mathematical Sciences. The Johns Hopkins University Press, Baltimore, second edition, 1989.

[Inc90] BBN Advanced Computers Inc. *Inside the TC2000 Computer*. BBN Advanced Computers Inc., 10 Fawcett St., Cambridge Mass., 02138, rev. edition, 1990.

[JLGS90] David V. James, Anthony T. Laundrie, Stein Gjessing, and Gurindar S. Sohi. Scalable coherent interface. *IEEE Computer*, pages 74–77, June 1990.

[KEW+85] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon. Implementing a cache consistency protocol. In *12. Intl. Symposium on Computer Architecture*, pages 276–283. IEEE, June 1985.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[Li86] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. Ph.D. thesis, Yale, September 1986.

[LLG+90] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *17th Intl. Symposium on Computer Architecture*, pages 148–159. IEEE, 1990.

[MPT91] Yeong-Chang Maa, Dhiraj K. Pradhan, and Dominique Thiebaut. Two economical directory schemes for large-scale cache coherent multiprocessors. *ACM SIGARCH Computer Architecture News*, 19(5):10–18, September 1991.

[Par88] The Parallel Computing Forum. *PCF Fortran: Language Definition*, August 1988.

[PBG+85] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, A. Norton, and J. Weiss. The IBM research parallel processor prototype (RP3): Introduction and architecture. In *IEEE International Conference on Parallel Processing*, pages 764–771, August 1985.

[TD90] Manu Thapar and Bruce Delagi. Scalable cache coherence for large shared memory multiprocessors. In *CONPAR90/VAPP IV, LNCS 457*, pages 592–603, 1990.

[ZBGH86] Hans P. Zima, Heinz-J. Bast, Michael Gerndt, and Peter J. Hoppen. Semi-automatic parallelization of Fortran programs. In *CONPAR 86*, pages 287–294, 1986.

END

DATE
FILMED
9/29/92