# UsulDSM: A Page-based Recoverable Distributed Shared Memory Project Report

Buğra Gedik
Georgia Institute of Technology
College of Computing
Atlanta, GA 30332, U.S.A.
bgedik@cc.gatech.edu

November 16, 2002

**Abstract**

UsulDSM is a page-based recoverable software distributed shared memory system designed for network of computers that don't have access to a physically shared memory. In this report we describe architecture of the UsulDSM and discuss its design and implementation. We also evaluate its performance through a simple parallel application that uses UsulDSM. We also analyze UsulDSM's scalability and the overhead induced by its recoverability property.

## 1 Introduction

Software distributed shared memory (DSM) is a well known mechanism that provides a globally shared memory abstraction through an easy to use shared memory interface for programming parallel applications on both network of work stations (NOWs) and NORMA style multi-computers. For a large class of applications DSM provides a more convenient programming model than message passing which is the available communication mechanism in no shared memory systems.

Although DSM provides a powerful abstraction for programming parallel applications, for long running computations like scientific applications or complex simulations, the failure of a node might force the application that uses DSM to restart from the beginning, which might be costly in terms of the consumed time. As a result a recoverable distributed shared memory [9, 5] is a useful tool for these kinds of applications.

## 2 UsulDSM Overview

UsulDSM is an efficient DSM implementation that also gives limited support for recoverability. In this section we provide a general overview of the UsulDSM system.

### 2.1 Target Platforms

UsulDSM is targeted for network of computers with separate address spaces. The current implementation works only on Solaris systems running on Sun SPARC platforms, however it can easily be extended to UNIX based OSs running on various platforms. In order to operate correctly the nodes of the system should have the same page size.

### 2.2 Offered Functionality

UsulDSM provides a globally shared memory view through globally shared memory segments. Nodes participating in the system can create or join globally shared memory segments. UsulDSM provides a sequential memory consistency model [8] on the globally shared memory. It supports one process on each node participating in the DSM system, however each process can be multi-threaded and multiple shared segments can be created given that they will not be accessed concurrently by multiple threads within the same process. UsulDSM also provides traditional lock and barrier synchronization. Locks are mutual exclusion locks that are used to build critical sections. Barriers are simply used to synchronize between different stages of a parallel application.

UsulDSM also provides some level of recoverability. Shared memory is checkpointed (coordinated checkpointing [2] is used) and in case of a failure last globally consistent memory state, that can be extracted from the individual checkpoints at each node, is constructed. However UsulDSM does not checkpoint process states. As a result the recovery procedure is not transparent to the application developer. After a failure, the application needs to read the most recent globally consistent shared memory segments (that are computed automatically and made available) and resume its processing after analyzing the state of the shared memory segments.

### 2.3 Programming Interface

UsulDSM is presented as a library to the application developer, which is linked with the application's executable on each node that will participate in the system. A slightly simplified subset of UsulDSM API is given in Figure 1.

The application should initialize the DSM subsystem by using the `initDSM` function and call `freeDSM` when it is done with the distributed shared memory in order to release the allocated resources. `openSegment` and `closeSegment` functions are used to open and close globally shared memory segments. Each segment is specified by a segment number. `openSegment` function returns a memory pointer, which can be used by the application in order to access the shared memory segment just like ordinary memory. Functions related with synchronization have straightforward interpretation and are listed in Figure 1.

```
/* contains the most recent error code */
int dsm_error;

/* initializing and freeing the DSM system */
int    initDSM(...);
void   freeDSM();

/* opening and closing DSM segments */
void * openSegment(int segNum, int segLen,
            bool initOwned, int numOfUsers);
bool   closeSegment(const int segNum);

/* barrier synchronization functions */
bool   createBarrier(int barNum, int numOfUsers);
bool   waitAtBarrier(int barNum);
bool   freeBarrier(int barNum);

/* lock synchronization functions */
bool   createLock(int locNum);
bool   acquireLock(int locNum);
bool   releaseLock(int locNum);
bool   freeLock(int locNum);
```

Figure 1: A slightly simplified subset of UsulDSM API

## 3 UsulDSM Design

There are several design issues in developing a software DSM system. In this section we first discuss the design decisions that has been taken in UsulDSM and then describe the UsulDSM architecture in more detail.

### 3.1 Design Decisions

An important design decision is the consistency model that will be supported by the DSM system. Although there exists several weaker consistency models (like release consistency [3], lazy release consistency [4], or causal consistency [1]) UsulDSM supports sequential consistency. This means that applications can be written as if the underlying system is a conventional shared memory machine. Implementation of the sequential memory is simpler than other possibly more complex consistency models which trade generality with performance. In UsulDSM there exists at most one writer for a page in shared memory at any time. If there exists a writer for a page then there exists no read copies of the page in the system. However multiple read copies can exist when a writer do not exist for a page. An invalidation based protocol is used to ensure sequential consistency, similar to invalidation based cache consistency protocols in cache coherent shared memory multi-processors [7].

At this point another design decision is to decide where the data and functionality related with tracking page ownerships is placed in the system. There are three choices here, namely (i) assigning the job of page ownership tracking to one of the nodes in the system which also participates in the parallel application equally with other nodes, (ii) designating a special coordinator node that tracks page ownerships, (iii) distributing the job of page ownership tracking to all nodes of the system by using a deterministic algorithm to match pages to nodes of the system. UsulDSM currently supports (ii), however (i) and (iii) can be supported easily with small modifications. In UsulDSM pages are always interchanged between nodes of the system in a peer-to-peer fashion, it is only the page ownership tracking and related functionality that is not fully decentralized at this point. Actually this choice is made explicitly, since the recovery is based on a global coordinator in UsulDSM as it will be described shortly. As a result there exists one node in the system that tracks page ownerships, implements locks and barriers, and coordinates the checkpointing, however the actual pages are interchanged between regular nodes of the system.

The obvious design decision regarding recovery is whether to use a coordinated checkpointing or independent checkpointing [2] approach for recording the state of the shared memory. Independent checkpointing is straightforward but has two small complications. Firstly, extracting the most recent global state of the shared memory requires some non-trivial processing. Secondly, there is a probability that domino effect might take place which will force a rollback to the initial state of the shared memory. And thirdly, the collection of saved checkpoints will also require some extra processing. As a result, we abandon this choice. UsulDSM uses a periodic globally coordinated checkpointing approach where a consistent state is recorded during each checkpoint. Although a decentralized checkpointing algorithm based on distributed snapshots algorithm is also a viable solution, globally coordinated checkpointing is much easier to implement.

There are also a few implementation related design decisions, first of them being the choice between a kernel level implementation and a user level implementation. A kernel level implementation has the advantage of having a lower latency and a better performance. However a user-level implementation is more flexible and portable. As a result UsulDSM is implemented in user level as a run-time library.

Another implementation level design decision is related with the underlying communication mechanism that is to be utilized by the DSM system. The two common choices here are UNIX sockets and RPC. The advantages of using RPC over sockets in building distributed applications is numerous and well recognized. However there are subtle considerations in using RPC when building a DSM system. For instance the RPC timeouts should be carefully chosen in order to properly implement blocking synchronization calls. Another issue is the underlying communication protocol to be used with RPC. For instance if the RPC middleware supports "at least once" procedure call semantics when working on top of UDP protocol, it is important to ensure that the remote procedures are implemented as idempotent operations.

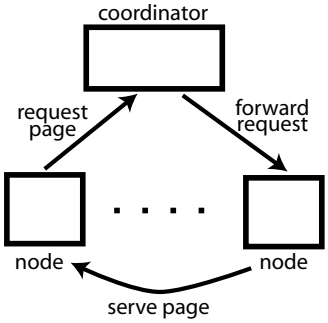


Figure 2: UsulDSM Architecture

### 3.2 UsulDSM Architecture

As described in Section 3, there is a special coordinator process in UsulDSM that is responsible for tracking page ownerships, forwarding page requests, handling synchronization and coordinating checkpointing. Figure 2 depicts the general architecture of the system. In the following subsections we discuss the details of how the distributed memory system, synchronization constructs and checkpointing works. The basic DSM architecture of our design is influenced by the design presented in [6].

#### 3.2.1 Distributed Memory

Consider a shared memory segment consisting of several pages. The coordinator process keeps a set of state variables associated with each page in a shared memory segment, namely *status*, *writer*, *read-set* and *lock*. *status* of the page represents the mode of the page, which could either be *shared* or *exclusive*. *writer* is the process, if exists, that can write the page. *read-set* contains the set of processes that can read the page. *lock* is used to serialize concurrent page requests as we will describe shortly. When a page is in shared status it is being read by one or more processes. Any write operation on the page and any read operation from processes that do not have the page (that are not in read-set) will need to contact the coordinator. When a page is in exclusive status there exists only one process, the writer, that can either read or write the page. Any read or write operation from a different process than the writer will need to contact the coordinator.

Each process that has access to a shared segment initially maps part of its address space to page aligned memory using `mmap`, depending on the size of the shared segment. Each process except the initial segment owner also sets the memory protection of the mapped memory to inaccessible using `mprotect`. Only the process who initially owns the segment will set the memory protection to readable and writable. Each process also registers a signal handler function which will handle the access faults when an access that conflicts with the protection of the shared memory is performed. When a fault occurs, the signal handler function contacts the coordinator through RPC, which might generate further RPC requests from the coordinator to other processes and from one another process to the one where the signal handler executes. Note that processes are multi threaded, meaning that when waiting the completion of the RPC to the coordinator, a process can receive an RPC message from another process which will probably install the required page.

In the following discussion we will describe how read and write faults are handled in UsulDSM. The pseudo code for the distributed algorithms to be described is presented in Figure 3. When an access fault occurs on the shared memory the faulted instruction is first examined to see whether we have a read fault or read fault, and the faulted memory location is examined to determine the page no. First lets assume we have a read fault. Then the faulted process, say process $A$, makes an RPC call to the coordinator requesting read service. The coordinator firsts acquires the lock associated with the page. Then it checks whether the page is in shared or exclusive mode. If it is in shared mode, it selects a random process from the read-set of the page, say process $B$, and makes an RPC call to $B$ requesting

```
node.fault(faulted_mem_address, faulted_instruction){
  pageNo = use faulted_mem_address to find out the faulted page no
  analyze the faulted_instruction
  if we have a read fault {
    coordinator.serve_read(pageNo, node);
    set protection of segment[pageNo] to READ
  }
  else{
    coordinator.serve_write(pageNo, node);
    set protection of segment[pageNo] to READ_WRITE
  }
}

node.install_page(page, pageNo){
  set protection of segment[pageNo] to WRITE
  segment[pageNo] = page;
}

node.invalidate_page(pageNo){
  set protection of segment[pageNo] to NONE
}

node.serve_page(pageNo, dnode, reduce){
  if(reduce)
    set protection of segment[pageNo] to READ
  dnode.install_page(segment[pageNo], pageNo);
}
```

```
coordinator.serve_read(pageNo, dnode){
  acquire(locks[pageNo]);
  if(status == SHARED){
    snode = randomly select an entry from read_set
    snode.serve_page(pageNo, dnode, FALSE);
    read_set.add(dnode);
  }
  else{ /* status = EXCLUSIVE */
    writer.serve_page(pageNo, dnode, TRUE);
    read_set.add(writer);
    read_set.add(dnode);
    status = SHARED;
  }
  release(locks[pageNo]);
}

coordinator.serve_write(pageNo, dnode)
  acquire(locks[pageNo]);
  if(status == SHARED){
    for each node snode in read_set
      if(snode != dnode)
        snode.invalidate_page(pageNo);
    read_set.clear();
    status = EXCLUSIVE;
  }
  else /* status = EXCLUSIVE */
    writer.invalidate_page();
  writer = dnode;
  release(locks[pageNo]);
}
```

Figure 3: Pseudo-code for the UsulDSM distributed shared memory

it to serve the page to $A$. Upon receiving the RPC call, $B$ reads the page which is in its memory and makes an RPC call to $A$ passing the page as a parameter. Then $A$ installs the page into its memory. When the last two RPC calls unwind back to the coordinator, $A$ is added into the page's read-set and the page's lock is released. Finally the initial RPC call returns from the coordinator to $A$, where the protection of the page is set to read. If the page was in exclusive mode at the first place, then the coordinator makes an RPC call to the writer of the page, say process $B$, requesting it to serve the page to $A$. Upon receiving the RPC call, $B$ reads the page which is in its memory, reduces the protection of the page to read and makes an RPC call to $A$ sending the page as a parameter. Then $A$ installs the page in its memory. When the last two RPC calls unwind back to the coordinator, page's read-set is set to $\{A, B\}$, the status of the page is set to shared and the page's lock is released. Finally the initial RPC call returns from the coordinator to $A$ where the protection of the page is set to read.

Now lets assume we have a write fault. Then the faulted process, say process $A$, makes an RPC call to the coordinator requesting write service. The coordinator firsts acquires the lock associated with the page. Then it checks whether the page is in shared or exclusive mode. If it is in shared mode, it makes an RPC call to each process in the page's read-set, requesting them to invalidate the page in their memory. When a process receives the invalidation request through the RPC call it sets the protection of the page to none. When all the invalidations are performed, the coordinator clears the page's read-set, sets the page's writer to $A$, the status of the page to exclusive and releases the page's lock. Finally the initial RPC call returns from the coordinator to $A$, where the protection of the page is set to write. If the page was in exclusive mode at the first place, then the coordinator makes an RPC call to the writer of the page requesting it to invalidate the page in its memory. Then the coordinator sets the page's writer to $A$ and releases the page's lock. When the initial RPC call returns to $A$ the protection of the page is set to write.

### 3.2.2 Synchronization

Synchronization primitives in UsulDSM consist of mutual exclusion locks and barriers. In order to acquire a lock a process needs to contact the coordinator through an RPC call. After receiving the call, the coordinator will block the request if there is another process holding the lock. Otherwise it will record the lock as acquired and return from the call. When a process releases the lock, it again makes an RPC call to the coordinator. The coordinator will signal one of the blocked requests on the lock if one exists and will return from the call.

Barriers also work in a straightforward manner, where each process makes an RPC call to the coordinator requesting synchronization on the barrier. When the coordinator receives the call, it first increments a variable representing the number of synchronization requests on the barrier. It blocks the request if the number of synchronization requests on the barrier is not equal to the number of participating processes, else it signals all other waiting requests, sets the number of synchronization requests on the barrier to zero, and returns from the call.

### 3.2.3 Checkpointing

UsulDSM uses a globally coordinated checkpointing that is coordinated by the coordinating process which also deals with synchronization and page ownerships. The coordinator initiates the checkpoints periodically. Each process checkpoints its memory during the checkpointing. In order to record a globally consistent view of the shared memory, a flush protocol should be executed so that during the checkpointing no pages are exchanged between processes. This is achieved by a three phase protocol in UsulDSM.

**The flush protocol**
The coordinator first makes an RPC call to each process, notifying them of the start of the checkpointing. Since RPC is a blocking call, each process receiving the start request wakes up a *checkpointing thread* that will deal with the rest of the checkpointing protocol, and immediately returns to the coordinator. A recovery thread first sets a *checkpoint flag* indicating that the checkpoint has started. An access fault handler will block if it sees this flag on. An access fault handler also sets and clears an *access fault flag* at appropriate points indicating whether it is currently processing an access fault or not. After setting the checkpoint flag the checkpointing thread waits until the access fault flag is clear. At this point the checkpointing thread is sure that there is no access fault processing in progress. However the shared memory is still in danger of being accessed, since the process may need to serve possible access faults of other nodes. As a result, the checkpointing thread makes an RPC call to the coordinator, which is equivalent to a barrier call. After returning from the barrier the recovery thread records in stable storage the portions of the memory that are readable and writable to it. Then the recovery thread synchronizes on a second barrier, again by making an RPC call to the coordinator. After returning from the barrier, it clears the checkpointing flag and blocks until the next checkpointing event that will be signalled by the coordinator in the future.

The advantages of the globally coordinated checkpointing is that there is no problem of garbage collection of old checkpoints, there is no domino effect, there is no complexity in constructing the most recent globally consistent shared memory. One downside of the UsulDSM implementation is that currently read-shared pages are redundantly recorded into the stable storage by each process. Supporting non-redundant recording of the pages requires a very small modification to the current implementation. However, it is not implemented because the ultimate approach to recording the shared memory is to perform an incremental checkpointing [10] where only the modified pages are recorded. This is left as future work.

## 4 Implementation Details

UsulDSM is implemented in C using Sun RPC as the underlying communication mechanism. It consists of $\sim$ 3000 lines of code, of which $\sim$ 800 are RPC generated. UsulDSM is presented as a user-level library to the application developers.

The implementation uses POSIX threads and is quite portable with a small exception. The code that determines whether an access fault to a memory location is a read fault or a write fault is inherently architecture dependent. The SIGSEGV signal provides access to the faulted instruction and the memory address that was accessed without proper rights. In order to resolve the type of the fault, one might

need to examine the faulted instruction or both the instruction and the memory address. The former is true in case we are running on a RISC architecture, which has simple instruction set. This enables looking only to the faulted instruction in order to determine the type of access fault. Since the current implementation of the UsulDSM runs on Sun SPARC machines, this approach is taken. However, porting to a CISC architecture might require a complicated examination of both the instruction and the memory address in order to resolve the access fault type at the user-level, if possible at all.

# 5 Experimental Results

In order to verify the correctness of the implementation and to measure its performance and scalability, we have experimented with several small parallel applications that use UsulDSM. In this section we present results obtained from a matrix multiplication application. The code for the application is given in 7. In the following experiment we present results for a 2048x2048 integer matrix multiplication.

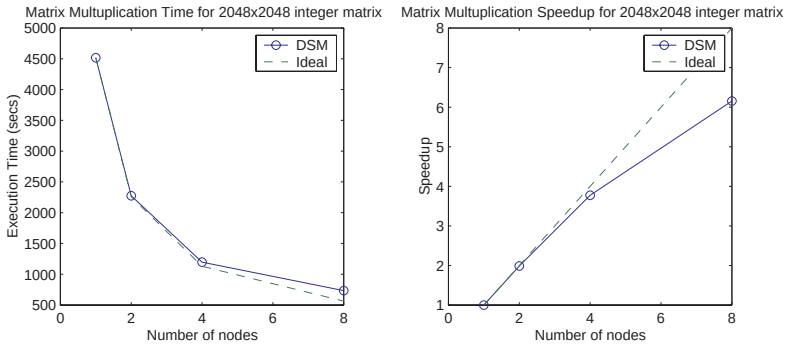


Figure 4: Matrix multiplication speedup

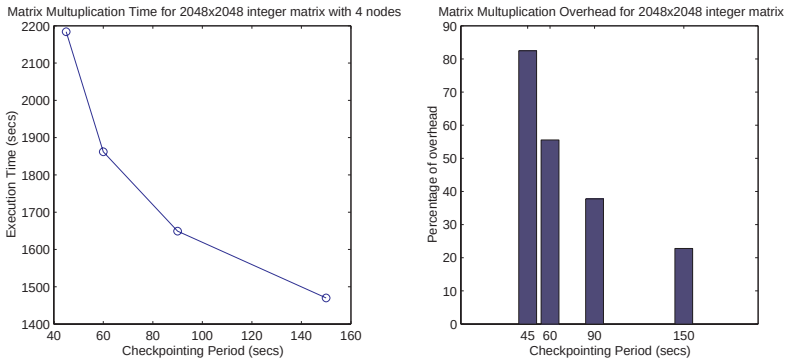


Figure 5: Checkpointing overhead

Each matrix takes up 16MB of space and the total shared memory segment size is 48MB, corresponding to two input matrices and one output matrix. The job of multiplying two matrices is distributed between processes in a straightforward manner. Figure 4 shows both the execution time and the speed up gained as a function of number of nodes (processes) participating in the computation. For this experiment checkpointing is disabled.

As seen from the figure, up to four number of processes the speed-up achieved is close to optimal. A degradation is seen when the number of processes increase to eight. This is mostly due to our deployment environment. The four of the machines used were less powerful than the other four. In general the DSM implementation shows a reasonable scalability. However there exists an extreme amount of parallelism in the matrix multiplication problem. As a result it would be interesting to measure UsulDSM's scalability through more complex applications, for instance TSP (traveling salesman problem) which requires a

master/slave style of parallelization.

In our second experiment we measure the overhead introduced by the checkpointing functionality. Again we consider a 2048x2048 matrix multiplication but this time with checkpointing enabled. We instrument with different values of the checkpointing period. Figure 5 compares the execution times of the matrix multiplication for the four process case as a function of checkpointing period. Figure 5 also shows percentage of overhead introduced by the checkpointing as a function of checkpointing period. It is clear from the Figure 5 that the cost of checkpointing is quite high when the checkpointing period is small. For instance when we have a checkpointing period of 45 secs, the overhead is $\sim 80\%$. However the overhead drops to $\sim 20\%$ when the recovery interval is increased to 150 secs. This is a considerable overhead, as the recoverability comes with a price. However the overhead could be decreased by using more sophisticated techniques such as incremental checkpointing as described in Section 3.2.3.

# 6 Conclusion

We have presented design and implementation of UsulDSM, a recoverable page-based software distributed shared memory system. UsulDSM is very robust and enables rapid development of parallel applications. The design of UsulDSM led us to consider several design choices and gave insight to various problems and their solutions in distributed systems. UsulDSM implementation provided us with significant experience in programming distributed systems using remote procedure calls.

# References


[1] M. Ahamad, G. Neiger, P. Kohfi, J. Burns, and P. Hutto. Causal memory: Definitions, implementation, and programming. Technical Report 93/55, Georgia Institute of Technology, College of Computing, 1993.

[2] M. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message passing systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, Oct. 1996.

[3] K. Gharachorloo, D. Lenoski, J. Laudon, P. B. Gibbons, A. Gupta, and J. L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *25 Years ISCA: Retrospectives and Reprints*, pages 376–387, 1998.

[4] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, pages 13–21, 1992.

[5] A.-M. Kermarrec, G. Cabillic, A. Gefflaut, C. Morin, and I. Puaut. A recoverable distributed shared memory integrating coherence and recoverability. In *Proc. of the 25th Annual Int'l Symp. on Fault-Tolerant Computing (FTCS-25)*, pages 289–298, 1995.

[6] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. In *Proceedings of the fifth annual ACM symposium on Principles of distributed computing*, pages 229–239. ACM Press, 1986.

[7] D. J. Lilja. Cache coherence in large-scale shared-memory multiprocessors: issues and comparisons. *ACM Computing Surveys (CSUR)*, 25(3):303–338, 1993.

[8] M. Mizuno, M. Raynal, and J. Z. Zhou. Sequential consistency in distributed systems: Theory and implementation. Technical Report RR-2437, 1995.

[9] C. Morin and I. Puaut. A survey of recoverable distributed shared memory systems. *IEEE Trans. on Parallel and Distributed Systems*, 8(9):959–969, 1997.

[10] J. S. Plank, Y. Chen, K. Li, M. Beck, and G. Kingsley. Memory exclusion: optimizing the performance of checkpointing systems. *Software Practice and Experience*, 29(2):125–142, 1999.

# 7 APPENDIX

The following is the code for the parallel program that performs a matrix calculation using the distributed shared memory. One of the processes designated as the master creates two random matrices and writes them to the shared memory segment, and also to a file for later verification. Then each process calculates its portion of the output matrix, that is the product of the previously created two matrices. After all processes are done with their calculation, the master reads the output matrix from the shared memory segment and writes it to a file.

```
#include ...
#include "DSM.h"

#define SEGNUM 444
#define BARNUM 555
#define LOCK   666

/* size of the portion of the matrix that will be
 * assigned to each process, in terms of pages
 */
#define K 512
#define FILENAME "mop.txt"

int main(int argc, char *argv[]){
  int master = 0;
  int i, res;
  int n, t, ps, bc = 0;
  int dim, size, nrows, r, c;
  int *base, *A, *B, *C;
  char * seg;
  FILE * file;

  if( argc != 5 && argc != 4){
    printf("usage %s <server> <i> <n> ['m']\n", argv[0]);
    exit(1);
  }

  if( argc == 5 && argv[4][0] == 'm' )
    master = 1;

  n = atoi(argv[3]); /* number of processes */
  t = atoi(argv[2]); /* no of this process */
  ps = (int) sysconf(_SC_PAGESIZE); /* page size */

  dim = calcMatrixDimension(n, ps); /* dimension of the matrix */
  if(dim == -1)
    abort();
  nrows = dim/n; /* number of rows per process */
  size = 3*dim*dim*sizeof(int); /* total size of all matrices, two input, one output */


  res = initDSM(DSM_CLIENT, argv[1], 1);
  if( 0 != res){
    fprintf(stderr, "initDSM DSM Error %d\n", dsmerror);
    exit(1);
  }

  if(master){
    file = fopen(FILENAME, "w");
    if(file == NULL){
      fprintf(stderr, "unable to open file for reading");
      exit(1);
    }
  }

  /* init the shared segment */
  if(master){
    base = (int *) initSegment(SEGNUM, size, master, n);
    if (base == NULL){
      fprintf(stderr, "master init segment DSM Error %d\n", dsmerror);
      exit(1);
    }
    bc = createBarrier(BARNUM, n);
    if (bc != 0 && dsmerror != DSM_BARRIER_UNAVAILABLE){
      fprintf(stderr, "master create barrier DSM Error %d\n", dsmerror);
      exit(1);
    }
    waitAtBarrier(BARNUM);
  }
  else{
    bc = createBarrier(BARNUM, n);
    if (bc != 0 && dsmerror != DSM_BARRIER_UNAVAILABLE){
      fprintf(stderr, "create barrier DSM Error %d\n", dsmerror);
```

```
      exit(1);
    }
    waitAtBarrier(BARNUM);
    base = (int *) initSegment(SEGNUM, size, master, n);
    if (base == NULL){
      fprintf(stderr, "init segment DSM Error %d\n", dsmerror);
      exit(1);
    }
  }

  if(master){
    createMatrix(base, dim);
    writeMatrix(base, dim, file);
    writeMatrix(base + dim*dim, dim, file);
  }
  waitAtBarrier(BARNUM);
  /* matrix multiplication */
  A = base + t*nrows*dim;
  B = base + dim*dim;
  C = B + dim*dim + t*nrows*dim;
  for(r=0; r<nrows; r++){
    for(c=0; c<dim; c++){
      *C = 0;
      for(i=0; i<dim; i++)
        *C += A[i] * (B+i*dim)[c];
      C++;
    }
    A += dim;
  }
  waitAtBarrier(BARNUM);

  if(master){
    writeMatrix(base+2*dim*dim, dim, file);
    fflush(file);
    fclose(file);
  }
  waitAtBarrier(BARNUM);

  if(bc)
    freeBarrier(BARNUM);
  res = closeSegment(SEGNUM);
  if(res != 0){
    fprintf(stderr, "DSM Error %d\n", dsmerror);
    exit(1);
  }
  freeDSM();
  if(master)
    fclose(file);
  return 0;
}

/* creates a random integer matrix */
int createMatrix(int *base, int dim){
  int r, c, k, i;
  for(i=0; i<2; i++){
    for(r=0; r<dim; r++){
      for(c=0; c<dim; c++){
        k = random()%16;
        *(base++) = k;
      }
    }
  }
}
/* writes a matrix to the specified file */
void writeMatrix(int *base, int dim, FILE *file){
  int r, c;
  for(r=0; r<dim; r++){
    for(c=0; c<dim; c++)
      fprintf(file, "%d ", *(base++));
    fprintf(file, "\n");
  }
  fprintf(file, "\n");
}

/* calculates matrix dimension
 * returns -1 if the matrix cannot be partititoned
 * into equal number of pages per process
 */
int calcMatrixDimension(int n, int ps){
  int dim = (int)sqrt((K*ps*n)/sizeof(int));
  if(dim*dim*sizeof(int) != K*ps*n || dim%n !=0)
    return -1;
  printf("matrix dim: %dx%d\n", dim, dim);
  printf("rows per process (1/%d): %d\n", n, dim/n);
  return dim;
}
```