Portable parallel programming on emerging platforms

Guy Delamarter†, Sandhya Dwarkadas‡*, Adam Frank‡ and Robert Stets‡*

*Department of Computer Science and †Department of Physics and Astronomy, University of Rochester, Rochester, NY 14627–0226, USA

Clusters of symmetric shared memory multiprocessors (SMPs) are fast becoming a highly available platform for parallel computing. There is a need for a uniform programming paradigm that allows users to transparently extend parallelism across multiple SMP nodes. A shared memory paradigm leverages the available hardware to handle sharing within an SMP, in addition to providing programming ease. Software distributed shared memory systems support the illusion of shared memory across the cluster via a software run-time layer between the application and the hardware. This approach can potentially provide a cost-effective alternative to larger hardware shared memory systems for executing certain classes of workloads. We describe here one such system and discuss its interface, performance and portability through an example real-world application from the scientific domain.

1. Introduction

The paradigm for high performance computing is undergoing a fundamental transition. At one time, computational scientists requiring access to the most advanced platforms were dependent on large centralized supercomputer centers. These centers had one or more large, expensive machines (e.g. a Cray-T90) along with support staff and maintenance infrastructure. In the mid-1990s, advances in parallel computing along with changes in the computing industry opened new opportunities in terms of a shift to more accessible platforms (e.g. an Origin 2000 or an IBM SP2). These machines, however, remain prohibitively expensive for many smaller groups of users. In addition, for portability reasons, users tend to avoid a message passing programming model (such as MPI or PVM), requiring considerable programmer effort in terms of distributing the work and communicating the data to participating processes appropriately.

Recent technological advances have resulted in symmetric multiprocessors (SMPs) and low-latency, high-bandwidth system-area networks (SANs) becoming commodity commercial items. Clusters consisting of SMPs connected by SANs are now widely available. Harnessing their power for parallel computing can be done with no additional hardware cost. Such platforms, however, provide multiple communication paradigms in hardware – shared memory within a node, and message passing across nodes. There is a need for a uniform programming paradigm that allows users to extend parallelism across multiple SMP nodes without requiring reprogramming.

The use of a shared memory paradigm leverages the available hardware to handle sharing within an SMP, in addition to providing programming ease. Software distributed shared memory (SDSM) systems support the illusion of shared memory across the cluster via a software run-time layer between the application and the hardware. This approach can potentially provide a cost-effective alternative to larger hardware shared memory systems for executing certain classes of workloads.

In comparison to the traditional network of (uniprocessor) workstations, a cluster of SMP nodes on a high-performance SAN can see much lower communication overhead. Communication within the same node can occur through hardware shared memory, while cross-SMP communication overhead can be ameliorated by the high performance network. Several groups have developed SDSM protocols that exploit low-latency networks and/or clusters of SMPs.

In this paper, we describe one representative SDSM system, Cashmere, which is a state-of-the-art SDSM with performance competitive with other leading systems that have been developed. All current general-purpose processors include hardware support to provide the illusion of a large independent address space for each application, normally referred to as virtual memory. Cashmere leverages this available hardware support to provide entry points to the run-time system so as to provide the illusion of sharing. The result is a system that minimizes overhead in the absence of sharing. Cashmere requires that applications use run-time-provided primitives to synchronize. In addition, if a process expects to see modifications made by another, it must synchronize with that process. Cashmere takes advantage of this requirement in order to optimize inter-process communication.

We demonstrate the utility of the system through one example application – a hydrodynamics simulation code called Total Variation Diminishing (TVD) from the astrophysics domain. Our goal is to convey a sense of
the benefits of the system, as well as to indicate what the application writer needs to know both about the application as well as about the underlying system characteristics in order to obtain both correct and good performance. We also present performance results for this application on a 32-processor cluster of 4-way AlphaStation 4100 SMPs. Our simple parallelization strategy is able to achieve up to 93% efficiency on 8 processors, and up to 62% efficiency on 32 processors. Most importantly, the application is able to effectively and seamlessly use more processors than are available on a single node.

The rest of the paper is organized as follows. Section 2 describes the interface provided by the SDSM system and illustrates its use. Section 3 provides a brief description of the protocol, and the system characteristics that are important to the performance and correctness of an application. Section 4 describes TVD, our example application, and the parallelization strategy used. Section 5 presents and analyses the application’s performance. Finally, Section 6 presents conclusions and future directions for the work.

2. SDSM interface description

The SDSM application programming interface (API) is a simple but powerful process-based shared memory interface (see Figure 1 for a summary of the salient calls in the API). At present, we support both C and Fortran interfaces. Calls are provided for process creation and destruction, shared memory allocation, and synchronization. Shared memory allocation is done through a special malloc routine, `csm_malloc`, in C, and through specially annotated common blocks in Fortran. All other memory is private to each process. The allocated shared memory is globally visible to all processes.

Synchronization calls allow a programmer to make explicit any ordering constraints on accesses to shared memory by different processes. The synchronization primitives we provide include locks and barriers. Locks provide mutually exclusive access to a region of code or data. A lock `acquire` operation gets permission to access the code or data, while a `release` operation releases the hold of process on the code or data. Synchronization using locks is useful when concurrent access to a particular piece of data is not allowed. Barriers are global synchronization primitives, and ensure that all processes have arrived at the same barrier before any process is allowed to continue. Barriers are conceptually equivalent to each process performing a `release` followed by an `acquire`. The SDSM system guarantees that at an `acquire` synchronization, a process sees a consistent view of all data, which reflects the modifications made by processes with which it synchronizes.

The Cashmere SDSM API requires that a process must synchronize with another in order to see its modifications, using synchronization primitives from the Cashmere API. Similarly, two accesses to the same shared memory location by different processors where at least one is a write must be separated by Cashmere-provided synchronization primitives in order to guarantee ordering among the accesses.

2.1 Illustrative example

We use Jacobi, an iterative method for solving partial differential equations, as a simple example to illustrate the use of the Cashmere API. Figure 2 presents the relevant (Fortran) fragments of the sequential and parallel versions of the Jacobi program. During each iteration, the program updates

```c
/* Initialize Cashmere and start up the requested number of processes */
void csm_init(int argc, char **argv)

/* Allocate shared memory (in C) */
char *csm_malloc(unsigned size)

/* Identify shared memory (in Fortran) - variable can be any name */
common /csm_common_variable/ x

/* Block the calling process until every other process arrives at the barrier. */
void csm_barrier(int id)

/* Block the calling process until it acquires the specified lock. */
void csm_lock_acquire(int id)

/* Release the specified lock. */
void csm_lock_release (int id)

/* Terminate the calling process and exit gracefully */
void csm_exit(int ret)
```

Figure 1. Summary of important calls in the Cashmere API.
the elements of a 2-dimensional matrix $b$ with the average of
its nearest neighbours. A scratch matrix $a$ is used to tempo-
rarily store the update in order to avoid over-writing the old
value before it is used.

Figure 2 presents only the portions of the code that
require change (other than creating and terminating the pro-
cesses with $\text{csm\_init()}$ and $\text{csm\_exit()}$ at the start
and end of the program). Three actions are required –
partitioning the work, identifying the shared data, and
synchronizing when there are dependences on data written
by other processors. Partitioning is done by allocating each
processor roughly equal-sized bands of the matrix (the
calculations of $\text{begin}$ and $\text{end}$ do this using the processor
identifier (returned by $\text{csm\_pid()}$) and the number of
processors used (returned by $\text{csm\_num\_pid()}$)). Matrix $b$ is
identified as shared (since $a$ is only used as a scratch array
by each processor individually, it need not be shared).
Synchronization is performed for this application by
inserting barriers whenever there is a dependence of a read
on a write to the same variable by another processor, and
vice-versa. The first barrier ensures that all processors have
read $b$ before it is written. The second barrier ensures that
all processors will compute on the new values of $b$ in the
next iteration.

3. Protocol description

The fundamental problem in supporting shared memory is
that of coherence – ensuring that modifications to shared
data are propagated to the multiple possible copies of the
data. SMPs provide hardware support for coherence. In
order to keep track of the multiple copies, memory is nor-
mally managed in small units referred to as the coherence
unit. This coherence unit is on the order of tens of bytes
with SMP hardware support, and sharing information is
maintained and updated in hardware for each copy of each
unit of shared data. SDSM systems must maintain and
propagate sharing information in software. SDSM systems
either use program instrumentation or existing hardware
mechanisms in general-purpose processors in order to de-
tect accesses to shared data. Cashmere uses the existing
virtual memory (VM) subsystem to track shared data ac-
tesses. Since the VM subsystem manages memory in much
larger units, the minimum coherence unit for Cashmere is
therefore large (an 8 byte virtual memory page on our Al-
pha cluster). The result is a system that minimizes overhead
in the absence of sharing and reduces software overhead
by minimizing the number of protocol operations necessary
to validate data. The large coherence unit, however, has
performance implications for some applications. Data that is
being accessed by two different processors may reside on
the same page, resulting in extra communication if the appli-
cation does not intend to actively share the data (normally referred to as false shar-
ing). Cashmere reduces this additional overhead through
the use of a multiple-writer protocol that allows concurrent
modifications to the same coherence unit by multiple pro-
cesses. Unnecessary communication is only incurred when
the processes synchronize. However, if applications are
written to avoid fine-grain sharing when possible, perfor-
man ince on the page-based SDSM can be greatly improved.

In Cashmere, coherence is implemented by having each
page of shared memory being managed by its own single,
distinguished home node. There is also an entry in a global
page directory for each shared page. The home node main-
tains a master copy of the page. The directory entry contains sharing set information and home node location. Cashmere currently uses an invalidate-based coherence protocol—in other words, copies of the data are eliminated rather than updated on a modification.

The main protocol entry points are page faults (accesses to pages in the shared address space that have been protected as a result of an invalidation and therefore are subsequently vectored into a user-level fault handler) and synchronization operations. On a page fault, the protocol updates the sharing set information in the directory and obtains an up-to-date copy of the page from the home node. If the fault is due to a write access, the protocol will also create a pristine copy of the page (called a twin) and add the page to the dirty list. As an optimization in the write fault handler, a page that is shared by only one node is moved into exclusive mode. In this case, the twin and dirty list operations are skipped, and the page will incur no protocol overhead until another sharer emerges.

At a release operation, the protocol examines each page in the dirty list and compares the page to its twin in order to identify the modifications. These modifications are collected and sent to the home node in order to update the master copy. The protocol then downgrades permissions on the dirty pages and sends write notices (an intimation that the page has been modified) to all nodes in the sharing set. These write notices are accumulated into a list at the destination and processed at the node’s next acquire operation. All pages named by write notices are invalidated as part of the acquire, resulting in a subsequent page fault on an access.

The memory consistency model specifies when and in what order modifications to different locations are visible to other processors. Cashmere implements what is called ‘moderately’ lazy release consistency. Simply stated, this means that modifications are propagated (as invalidation messages) at release operations, but need not be incorporated until a subsequent acquire operation. In other words, an application must synchronize in order to see modifications made by other processors, as has already been mentioned in Section 2.

Cashmere is an SMP-aware protocol. The protocol allows all data sharing within an SMP to occur through existing hardware support for coherence in the SMP. Pages in shared space are physically shared within a node. Software coherence overhead is incurred only when sharing spans nodes. Cashmere uses several novel techniques to reduce synchronization requirements among processes within the same node due to software operations, as well as to coalesce protocol operations on behalf of a node.

Currently, Cashmere is implemented on Compaq’s Tru64 Unix using a Memory Channel II SAN. However, the system is implemented completely at user level and does not rely on any specialized operating system support. Hence, it may be easily ported to other popular operating systems, such as Linux and Windows 2000, as well as to other platforms with low-latency high-bandwidth communication.

4. Example application

We have implemented and evaluated a large number of applications using Cashmere. Here, we describe the parallelization of an existing hydrodynamics simulation code used for astrophysics research that we have recently ported to Cashmere. This particular code is called TVD, after a property that the main computational engine maintains in its representation of the fluid. This code was originally developed by Ryu et al. using the method described by Harten. It has been used to explore astrophysical problems such as the accretion flow of gas around a mass point and adapted to investigate the nonlinear interaction of winds from stars with different types of surrounding environments. It was written in FORTRAN-77 as a sequential program and has been used in that form on a variety of machines, including a Cray YMP, SGI Origin 2000, SPARC 20 and an Intel Linux box.

The code simulates the flow of fluid in a quarter meridional plane of an axially symmetric region. This region is gridded and represented in memory as a three-dimensional fluid array with the indices in the first dimension representing the fluid property (mass density, components of the momentum density and total energy density) averaged over a finite square patch, the indices in the second dimension representing the radius of the center of the patch, and the indices in the third dimension representing the altitude (Z) of the patch. Abstractly, the primary purpose of the code is to set up some interaction between winds and some initial environment, and provide snapshots of the fluid array periodically in simulated time (e.g. every 30 years). We can take these snapshots as disk files to a graphics program after the simulation has run, and visualize the results as a movie, or closely investigate any individual snapshot.

In the following, we sketch the algorithm, describe the parallelization of this sequential application, and present the resulting performance.

4.1 Algorithm

The fluid array is initialized at the beginning of the program to be consistent with some situation we want to model (e.g. a wind with a particular speed coming from a star with a particular mass into some environment). Once the array is initialized, the program enters a loop that repeatedly updates or evolves the fluid array over a single time-step.

The first part of the loop determines how much simulated time this time-step represents. That value depends crucially on the properties in the fluid. For instance, to keep the method numerically stable, we must make sure that this time-step is short enough that no disturbance in the fluid...
has a chance to cross more than a single grid cell edge. In addition, other physical effects such as radiative cooling and gravity from a central star place additional upper bounds on the time-step due to the strength of the effect in each cell. Once the least upper bound on the time-step has been determined, the loop makes some assumptions about the properties of the fluid just outside the main simulated region. One possible assumption is that it is just like the fluid adjacent to it inside the region (transmissive boundary conditions). The loop then applies operations that actually change the fluid values in the fluid array. In the case that we present in the performance section, the effects of radiative cooling and actual motions of the fluid are applied in sequence, but separately. Periodically, the loop writes a snapshot to disk of the fluid array so we can visualize the computation. The loop terminates the program when some total amount of simulated time has passed.

Other checkpoint and protection operations are performed in the loop and in the program as a whole, but this description should be enough to explore the parallelization of our application.

4.2 Parallelization strategy

The primary strategy for parallelization is to split the computations in a manner similar to that presented in Figure 2, so that each process is responsible for its own unique region of the fluid array. Figure 3 pictorially represents the parallelization strategy. We give each process some interval of altitude (Z) to work upon in the array. Given that this code is written in FORTRAN, which uses column-major ordering, and that the altitude index is the last one for the fluid array, this means that each process works within a contiguous region of memory. Such an allocation minimizes any false sharing due to the large coherence unit among the processes. In the simulation code, this simple splitting works very well to parallelize the code, but there are a few cases where there is dependence between regions. Such cases require synchronization and communication between processes (similar to the barriers in the illustrative example). They include: (i) performing a global search for the least upper bound on the time-step through all grid cells, (ii) accounting for fluid disturbances and material passing from the region of one process to another, and (iii) creating the snapshot files.

To perform global searches, we simply have each process perform a local search within its region, and report the result to a shared array which has a slot for each process. Then each process reads this array and finds the least upper bound among them to determine the global value. All processes must be synchronized using a barrier call to make sure the report array is complete before searching it.

Accounting for disturbances passing vertically through the boundaries of a process region requires examining the values in adjacent regions. The potential problem is that the adjacent process may modify the values before they can be examined. This is handled by first copying the adjacent values from the shared array to a local array (similar to the scratch array in Figure 2). Then each process is free to update its local portion and use the ‘frozen’ shared values to accommodate information from adjacent regions in the array.

There are two possible strategies for dealing with snapshots of shared arrays: (1) have each process write a small file for its portion of the array and reassemble the pieces externally; (2) have a single process grab all of the shared array and write a single file. To make the processing required to visualize the results of the parallel program as much like that required for the sequential version as possible, we chose to have a single process perform the I/O. This does result in the other processes remaining idle while the array is communicated and the file is written, but the snapshot occurs typically about once every 100 iterations of the primary loop. This hit in performance appears to be acceptable for now, although it does limit the scalability of the application.

Relatively few changes need to be made to the sequential code to allow it to run under Cashmere. In order to implement the accommodations mentioned above, the shared variables must be placed in specially named common blocks, the limits of loops along the axial dimension of the arrays need to be modified to be functions of the process performing them, and barrier calls need to be introduced to synchronize when cross-process communication might be required. The simplicity of the changes makes
it possible to use the identical code on a sequential non-
Cashmere system without any performance loss. Only stubs
for the Cashmere-specific calls need to be defined, which
return values consistent with having a single process on
the system. In fact, this is exactly how the initial port of the
code to Cashmere was performed: on a sequential (single-
processor) machine. In addition, as we will show, the same
code can be made to run on hardware shared memory ma-
achines as well.

5. Performance evaluation

We evaluate the performance of the system on a set of eight
AlphaServer 4100 5/600 servers, each with four 600 MHz
21164A processors, 8 MB direct-mapped 64-byte line size
per-processor board-level cache, and 2 Gbytes of memory.
The servers are connected with a Memory Channel II user-
level remote-write system area network, a PCI-based net-
work with a peak point-to-point bandwidth of 75 Mbytes/s
and a one-way, cache-to-cache latency for a 64-bit remote-
write operation of 3.3 µs. Previous work has examined the
performance of the system on a variety of standard bench-
marks, as well as a widely used genetic linkage analysis pro-
gram. In this paper, we demonstrate the utility of
SDSM using our example application, TVD.

Figure 4 shows the execution time of the application as
the number of processors is varied from 1 to 32. The test
case uses a 256 × 256 grid and allocates approximately
7.5 Mbytes of shared data. The speedup (when compared to
the execution on a single processor without linking with the
Cashmere library or incurring any additional overhead) at 8
processors is 6.66, while the speedup at 32 processors is
14.6.

Examining performance with up to 4 processors, we see
that the application achieves 95% efficiency, with a
speedup of 3.78 at 4 processors. We also ran the application
using hardware shared memory, in other words, without
linking with Cashmere. No changes were required to the
application in order to accomplish this. We merely linked
with a different library that used system-provided mecha-
nisms for allocating shared memory and hardware primitives
to synchronize. The execution time with Cashmere is equivalent to that using hardware shared mem-
ory alone. This indicates that the Cashmere run-time is able
to achieve its goal of utilizing hardware shared memory
within a node, and avoiding any additional software over-
heads.

At 8 processors (2 nodes), the application continues to
achieve good efficiency – 83%. Cashmere enables the appli-
cation to transparently take advantage of more pro-
cessors than are available on a single SMP.

At 16 processors, the application achieves only 64% effi-
ciency, while at 32 processors, the application achieves
only 45% efficiency. Speedup in this application is partially
inherently limited due to the serialized disk I/O. In addition,

at 32 processors, the application synchronizes using global
barriers approximately every 4 ms, and communicates an
average of 71 kbytes between synchronization intervals.
Inherent load imbalances in the application, coupled with
those caused by protocol perturbation and serialized I/O,
combine to increase synchronization wait-time to 42% of the
total execution time on an average at 32 processors. Elimi-
nating the I/O brings the efficiency at 32 processors up to
62% (speedup of 20 – with a corresponding reduction in the
synchronization wait-time down to 22%), and that at 8 pro-
cessors up to 93%.

6. Conclusions

In this paper, we have provided a brief overview of SDSM
systems, in particular, the Cashmere virtual memory-
based SDSM system. We demonstrated the utility and ef-
ectiveness of SDSM through an example hydrodynamics
simulation code, TVD, from the astrophysics
domain. SDSM systems on clusters of SMPs connected by
SANs can provide a cost-effective alternative for high-
performance computing, and enables an application to
transparently take advantage of more processors than are
available on a single SMP.

Just as in TVD, many of the target computationally inten-
sive applications would benefit from the ability to interact
with the application execution, for example, in order to steer
the computation. Future work will address this need by ex-
tending the sharing capability to more distributed environ-
ments while exploiting application requirements in order to
avoid compromising efficiency or ease-of-use. This will al-
low easy addition of a distributed interface that allows interaction. Additionally, we are examining ways of integrating compiler support for efficient communication and load balancing with the run-time system, as well as providing performance debugging support.


ACKNOWLEDGEMENTS. This work was supported in part by NSF grants CDA-9401142, EIA-9972881, CCR-9702466, AST-0978765, and CCR-9705594; University of Rochester's Laboratory for Laser Energetics through a Frank J. Horton Fellowship; and an external research grant from Digital/Compaq.