SDCS: Simplified Data Communications in Parallel/Distributed Applications

Yong Mao, Yunhong Gu, Jia Chen, and Robert L. Grossman
Laboratory for Advanced Computing, University of Illinois at Chicago
{ymao1, ygu3, jchen5, grossman}@uic.edu

Abstract

This paper presents SDCS (Simple Data Communication and Sharing), a programming model for data communications in parallel/distributed applications. With SDCS, developers can define data communications in shared memory style and have the model translate the declarations into corresponding message passing code. The translation from data sharing declarations to message passing code is based on simple mapping rules to lower runtime overhead and increase understandability of the model. Some frequently seen data communication modes are well supported to enhance its usability. SDCS can effectively reduce the difficulty in programming process communications.[1]

1. Introduction

Message passing and shared memory are two primary data communication programming models for developing parallel/distributed applications. Message passing has high runtime efficiency but it is considered less programmable since for each message passing operation, users need to specify the source or destination process involved and this brings undesired close process coupling.

Shared memory was first applied in multiple-processor computers to enable direct data sharing among processes. DSM (Distributed Shared Memory) [11] extends this idea to processes across more than one machine. Despite its easier programming model, the impact of DSM on non-research users and applications has been small [4]. In our opinion, there are several weaknesses with DSM systems. First, most distributed shared memory implementations tend to introduce new programming languages or extensions to existing programming languages to provide support for shared-memory style operations. This increases the difficulty of incorporating communication modules with other parts of the parallel/distributed applications. Second, many DSM implementations use complicated synchronization protocols and concurrent control mechanisms, which are not cost-effective for those parallel/distributed applications in which chances of data race conditions are small. The final problem with DSM systems is their relatively poor performance compared to message passing models.

Noticing the pros and cons of message passing and DSM, we designed and implemented SDCS, an easy and performance-effective programming model. In this model, developers declare data exchanges using the write/read operations on shared data. These declarations are parsed to produce code that realizes the data communication semantics with a certain message passing mechanism. The conversion from data sharing declarations to messaging passing implementation code is designed to be easy so that we get clear and unambiguous runtime data communication behavior.

The primary contribution of this work is a simple data communication programming model. It retains the ease of writing the communication code in DSM style and also enjoys runtime performance due to its light-weight design and implementation compared to most DSM systems.

The rest of this paper is organized as follows. Section 2 describes the model. Section 3 evaluates its usability and performance. Section 4 discusses the related work and section 5 is the conclusion and future work.

2. Model Description

With SDCS, users express data communications required by their parallel/distributed applications through defining read/write operations on data. The model will translate the declaration to code for users to call from their applications to accomplish the desired operations with the help of the model's runtime. The communications are done by employing a specific message passing approach, which is specified as an option during the code generation process.

[1] This research is supported, in part, by funding from The Searle Funds at The Chicago Community Trust.

```
shared example
{
   int value;
   process p1.write(value);
   process p2.read(value);
}
```

Code List 1:SDCS Declaration Example

Code list 1 gives a concrete example of the data sharing declaration in SDCS. In this example, two processes are trying to exchange an integer value so one defines a write and the other defines a read operation on that piece of data. If we are using C/C++ as the target language, two functions will be generated with function signatures as follows:

```
int SDCS_example_p1_w(int value);
int SDCS_example_p2_r(int * value);
```

The first function is called by the process that wants to write to the value and the second one is called by the process that wants to read that integral value. The only parameters involved in calling these two functions are the data. ***p1*** and ***p2*** in the declaration are actually the names of two process roles involved in this particular data communication. Any physical process can play ***p1*** role or ***p2*** role at runtime as long as it calls the corresponding SDCS function.

At runtime, a match-making process provided by SDCS should be running to match write/read operations. Each SDCS-generated write/read function will contact the match-making process to post write/read requests. If any request match is detected, the match-making process will inform the involved processes so that they can start exchanging data using the chosen message passing approach.

As an initial implementation effort, we chose C/C++ as our target host language and three different underlying message passing mechanisms: socket on TCP, MPI (Message Passing Interface) [12] and a high-speed network transport library UDT [8].

2.1. Data types and structures

One important feature of SDCS is that it supports exchanging data of complex structures directly. Users can declare preliminary types (integers, characters, and floating-point numbers), array types, and composite types. The syntax of declaring array types and composite types are similar to declaring arrays and struct types in C/C++. Users can declare composite types up to unlimited nesting levels. The parameters of generated write/read functions will correctly reflect the data structure appearing in the declaration and the generated code takes care of packing/unpacking issues for data transmissions automatically.

2.2. Communication modes

The Code List 1 example only illustrates a simple data communication mode: one piece of data is moved from one single process to another single process. To achieve good usability, SDSC was designed to directly support several commonly seen modes in parallel/distributed applications. In all the following modes, users make write/read operations on data and the corresponding code is generated by SDCS for users to call from their applications.

2.2.1. Single write with single read. This mode was illustrated in our Code List 1 example.

2.2.2. Broadcast. In this mode, one process writes the data that are to be read by multiple processes.

2.2.3. Scatter. In this mode, one process has a data array split into portions and passes each portion to a reading process. Scatter and Gather (discussed later) are two frequently seen communication patterns in master-slave model-based parallel applications. Scatter mode can split data into smaller pieces and assign them to multiple slave processes for parallel computation and Gather mode can be used to collect partial work results from those worker processes.

2.2.4. Gather. In this mode, one process assembles an array of data by getting array elements from several other processes.

2.2.5. Field write with whole read. In this mode, data of composite type are involved. Several processes write into different fields of a composite type and a single process reads the whole composite type data. This mode is useful when one process collects data of heterogeneous types from more than one source.

3. Model Evaluation

3.1. Usability

If we compare SDCS to MPI, we can find two significant improvements in usability. In MPI, a call to a send/receive operation always has a parameter that specifies the destination process. With SDCS, users only concentrate on how to read/write data. The destination process to be involved in the communication is figured out at runtime by the model automatically. This is a natural consequence of following the shared memory communication paradigm. Second, when handling data with complex structure, MPI requires the users to write explicit code to construct so-called ***datatype*** objects to make the structure known to MPI's runtime system. This effort is spared for SDCS users since the SDCS parser handles that automatically. MPI is considered quite representative of message passing models, so the above arguments apply to most other message passing models. SDCS also provides an easier programming interface than most DSM systems. First, we did not introduce any extension or new constructs into the host language. Data communications are provided in the form of functions with a simple parameter list. The data communication code can be seamlessly combined with other parts of the application. Second, this model provides higher level support for several important data communication modes: scatter, gather and field write. These modes are quite common in parallel/distributed applications but they are not easy to accomplish with the low-level write/read operations provided by most DSM implementations.

As a real case, we applied SDCS in our project for developing a tandem mass spectrometry search engine [13]. This program searches in a given protein database to find the best match for the experimental tandem mass spectrometry data. Due to the large size of the protein database and high computational complexity, we made a parallel version of the program using a master-slave control model. Each run of the search requires 8 data communications between the master process and those slave processes. We expressed all these data communications with 8 SDCS data sharing declaration blocks and used the produced functions to make the data communications. The declaration file we created for this application is less than 100 lines and it was done in one hour. Before we applied this new model, we used manually written code involving sockets to achieve the data communications. The source code added up to about 600 lines and the time we spent on writing and debugging it was about 16 human hours.

3.2. Communication Performance

We will present the experimental performance data from two tests done on an Apple cluster. The hardware configurations are as follows: each node in the cluster is Dual Power G5 at 2 Giga Hz with 4G RAM and the network connection between any two nodes is 1 Gbps.

The first test is to implement a Ping-Pong program with our model and plain socket programming, respectively, and compare their runtime efficiency. Process one sends an integer array to process two, which sends the array back to process one and the round-trip time is measured. We compare a plain TCP implementation to an implementation on a TCP-based SDCS model. We did the tests with various lengths of data arrays. Figure 1 displays the time ratios under these two implementations with different array lengths.

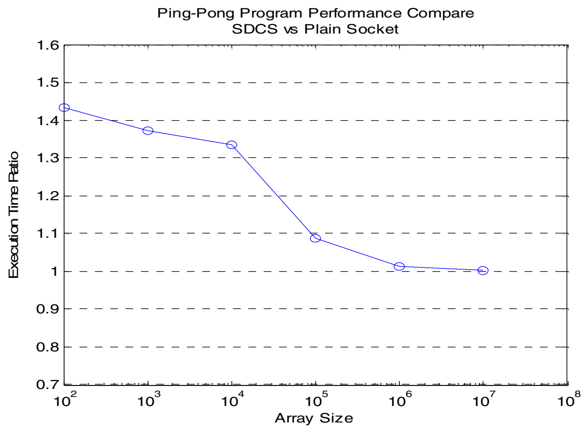

Fig.1. Ping-Pong program time performance comparison. Plots the time ratios of SDCS implementation to plain socket implementation with various array sizes.

The second test is a parallel integral matrix multiplication. In this test, we multiply two n*n square matrices to produce the third result n*n matrix. We used 5 nodes (10 processors in total) in the cluster to carry out the computation. We compare MPI-based SDCS implementation to plain MPI implementation, both using Open MPI [7] with version 1.0rc4. Three sharing modes are involved: broadcast, scatter and gather. This test can give us a good idea of SDCS's efficiency in handling these communication patterns. In the meantime, we also measured the time when the multiplication was done on one node with a serial program. We tested with different matrix sizes. Figure 2 shows the speed-ups over the serial program under two different implementations.

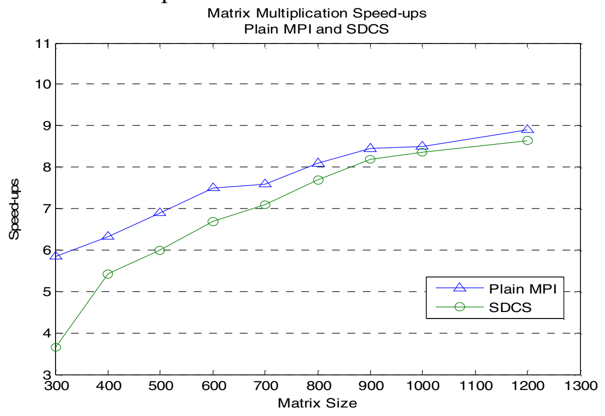

Fig.2. Matrix multiplication Speed-ups. Plots the speed-ups of SDCS implementations and plain MPI implementations over serial multiplication program with different matrix sizes. In the test, 5 cluster nodes (10 CPUs) were used.

When compared to the corresponding plain message passing approach, the overhead introduced into SDCS mainly results from the exchange of data read/write requests and match messages. A request message tells the match-making process about a newly posted write/read operation and a match message contains information used by a process to contact another process for data transmission. Unlike some DSM's data duplication implementation strategy, in SDCS, no user data are duplicated. The only movement of user data is toward the processes that are known to need the data. The number of those request and match messages is independent of the size of the transferred data. So when the size of each data exchange grows, the overhead ratio introduced by SDCS will drop quickly.

4. Related Work

Message passing has always been the most popular programming model for process communications due to its high runtime performance and wide platform support. Low levels of message passing models such as sockets, despite their high runtime efficiency, usually lack good support for higher level application needs. MPI (Message passing interface) [12] and PVM (Parallel Virtual Machine) [7] are two efforts to enhance the usability of those low level message passing programming interfaces by introducing uniform interfaces on different platforms. It is also easier to handle scatter and gather communication patterns with both MPI and PVM.

In the past 20 years, many DSM have been proposed and implemented [2, 3, 5, and 9]. Some systems such as Mirage [5] provide memory sharing at page level and some at variable or user object level such as Munin [3] and Midway [2]. Our model also provides the sharing at user data level. Most DSM systems provide programming interfaces to applications either through the introduction of a new programming language (e.g., Orca [1]) or adding language extensions to a host programming language (e.g., mpC [10]). Our SDCS however, provides data communication services through produced interface functions with a simple parameter list. Many DSM implementations introduce complex constructs for data consistency and access synchronization control. For example in Midway [2], at the level of the programming language, all shared data must be declared and explicitly associated with at least one synchronization object. SDCS provides data sharing modes with simple runtime behavior and thus reduces the programming interface complexity.

5. Conclusion and Future work

SDCS significantly reduces the programming burden on developers by introducing a shared-memory programming style and keeps high runtime efficiency by doing message passing code translation.

We are now working on a parallel programming framework and runtime environment based on SDCS's communication model.

6. References

[1] Bal. H.E., Kaashoek, M.F., and Tanenbaum A.S.: *Orca: A Language for Parallel Programming of Distributed Systems*, IEEE Transactions on Software Engineering, vol. 18, No. 3, March 1992, pp. 190-205.

[2] B. N. Bershad and M. J. Zekauskas: *Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors*. Technical Report CMU-CS-91-170, Pittsburgh, PA (USA), 1991.

[3] John B. Carter, John K. Bennett, Willy Zwaenepoel: *Implementation and Performance of Munin.* SOSP 1991: 152-164.

[4] J. Carter, D. Khandekar, and L. Kamb, *Distributed shared memory: Where we are and where we should be headed.* Proceedings of the Fifth Workshop on Hot Topics in Operating Systems, pp. 119–122,May 1995.

[5] B. D. Fleisch and G. J. Popek: *Mirage: A Coherent Distributed Shared Memory Design.* Proceedings of the 12th ACM Symposium on Operating Systems Principles, pp. 211-223, 1989.

[6] Edgar Gabriel, *et al*: *Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation.* In Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, pages 97--104, September 2004.

[7] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek,and V. Sunderam: *PVM: Parallel Virtual Machine—A User's Guide and Tutorial for Network Parallel Computing.* MIT Press, Cambridge, Mass., 1994.

[8] Yunhong Gu, Xinwei Hong, and Robert Grossman: *Experiences in Design and Implementation of a High Performance Transport Protocol*, SC 2004, Nov 6 - 12, Pittsburgh, PA, USA.

[9] Peter J. Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel: *TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems.* USENIX Winter 1994: 115-132.

[10] Alexey Lastovetsky: *mpC - a Multi-Paradigm Programming Language for Massively Parallel Computers.* ACM SIGPLAN Notices, 31(2):13-20, February 1996.

[11] Li, K.: *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD dissertation, Department of Computer Science, Yale University, September 1986.

[12] Message Passing Interface Forum: *MPI: A Message-Passing Interface standard.* International Journal of Supercomputer Applications, 8(3/4):165–414, 1994.

[13] http://cbc.lac.uic.edu/cbc/login