PerlDSM: A Distributed Shared Memory System for Perl

Norman Matloff
University of California, Davis
matloff@cs.ucdavis.edu

Abstract

A suite of Perl utilities is presented for enabling Perl language programming in a distributed shared memory (DSM) setting. The suite allows clean, simple parallelization of Perl scripts running on multiple network nodes. A programming interface like that of the page-based approach to DSM is achieved via a novel use of Perl's tie() function.

Keywords and phrases: Perl; parallel processing; distributed shared memory.

1 Introduction

The Perl scripting language has achieved tremendous popularity, allowing rapid development of powerful software tools. It has evolved in reflection of general trends in the software world, one of these being concurrency, with recent versions of Perl featuring multithreading capabilities. Our interest here will be in enabling Perl to perform more general parallel processing functions, with the concurrency now being across multiple machines connected via a network. A typical setting is that of a subdividable, I/O-intensive task being performed by a network of workstations (NOWs).

For example, we may have a Web server written in Perl[1] which we would like to parallelize into a *server pool*. Here a "manager" server would operate a task farm of incoming requests, and "worker servers" at other network nodes would obtain work from the manager.

Or, we may have a Web search engine written in Perl, which we wish to parallelize over various network nodes, with each node searching a different portion of Web space.

Perl applications of this type have been written, but with the parallelization operations being handled on an *ad hoc basis* in each case, i.e. without drawing upon a general package of parallelization utilities.

One such package has been developed, PVM-Perl [6]. This enables the Perl programmer to do message-passing on a network of machines. However, many in the parallel processing community believe that the shared-memory paradigm allows for greater programming clarity [7]. Accordingly, a number of software packages have been developed which provide *software distributed shared memory* (DSM) [5], which provides the programmer with the illusion of shared memory even though the platform consists of entirely separate machines (and thus separate memories) connected via a network.

Due to the clarity and ease of programming of the shared-memory model, our focus here is on DSM systems. These have been designed mainly for C/C++, and before now, DSM had not been developed for Perl. This paper presents such a system, PerlDSM. This set of Perl utilities enables the Perl programmer to do shared-memory paradigm programming on a collection of networked machines.

DSM itself is divided into two main approaches, the *page-based* and *object-based* varieties. The page-based approach is generally considered clearer and easier to program in. However, its implementation for C/C++, e.g. in Treadmarks [1], requires direct communication with the operating system. This kind of solution does not apply to interpreted languages like Perl. At first glance, therefore, it would appear that implementation of a page-based DSM in Perl would necessitate modification of Perl's internal mechanisms.[2]

Yet we have been able to design PerlDSM in a paged-based manner (i.e. giving the programmer an interface like those of paged-based systems) without resorting to modification of Perl internals. This is due to a novel use of Perl's tie() construct.

The details of the PerlDSM system and its implementation are presented in Sections 3 and 4, followed

[1] Or possibly with the server consisting of a Perl driver which runs C/C++ modules.

[2] Perl actually is designed to facilitate working at the internals level, but still this would not be a simple project.

by an example in Section 5. In Section 6, we discuss other issues, such as planned extensions.

2 Page- vs. Object-Based DSM Systems

In a page-based DSM system, C code which operates on shared variables, say x and y, might look something like this:

```
shared int x,y;
...
x = 21;
y = x + 5;
```

First x and y must be set up as shared variables.[3] Then x and y are accessed in the normal C fashion; all the operations for sharing with other network nodes — e.g. fetching the latest copy of x from other nodes — are done behind the scenes, transparent to the programmer.

By contrast, in object-based systems, the code

```
x = 21;
y = x + 5;
```

might look like this:

```
x = 21;
put_shared("x",x);
x = get_shared("x");
y = x + 5;
put_shared("y",y);
```

Here the programmer must bear the burden of explicitly invoking the sharing operations via calls to the DSM system.

There are advantages and disadvantages to each of the two approaches to DSM [1]. The object-based DSM approach has certain performance advantages, but the page-based approach is generally considered clearer and more natural, as can be seen in the comparison above. Thus our goal was to develop some kind of page-based DSM for Perl.[4]

[3]This is often accomplished by calls to a shared version of malloc().

[4]An alternate world view which shares some characteristics with the shared-memory paradigm is that of *tuple spaces*. This was originally proposed in the Linda language [3], and currently in use in JavaSpaces [4] and T-Spaces [8]. Perl versions of this have been proposed. However, these are very similar in terms of programmer interface to object-oriented DSM; the programmer must insert function calls to get tuples from, and put tuples into, the tuple space. We thus did not pursue the tuples approach.

3 Implementation of a Paged-Based Programming Interface in PerlDSM

The page-based approach relies on UNIX system calls which allow modification (actually replacement) of the page-fault handler in the machine's virtual memory system.[5] Only the pages currently owned by the given network node are marked as resident, and when the program reaches a page it does not currently own, a page fault occurs. The DSM has set up the new handler to get the new page from whichever node currently owns the page, rather than from disk as usual.

This means that direct implementation of page-based DSM in Perl, however, would not be possible, as Perl is an interpreted language. True, the data for the Perl program is also data for the Perl interpreter, but reconciliation of all the correspondences would be very difficult.

At first glance, then, it would seem that implementation of a page-based DSM for Perl would require working at the level of Perl internals. Actually, Perl does expose quite a bit of this to the application programmer, but still such an implementation would be rather complex.

However, it turns out that an interesting feature of Perl, the tie() function, can be used to form an extremely elegant solution to this problem. This function associates a scalar variable (or array or hash) with an object.[6] We say that the scalar is *tied* to the object.

The object is required to have three methods, named TIESCALAR (TIEARRAY or TIEHASH in the case of array or hash variables), FETCH and STORE. In Perl, each time a scalar variable appears on the right-hand side of an assignment statement (or in some other context in which its value needs to be fetched), the FETCH method is called and its return value used in place of the scalar variable. Similarly, if the scalar appears on the left-hand side of an assignment, the value on the right-hand side is passed to STORE.

To see how PerlDSM uses tie(), consider our example with x and y above,

```
shared int x,y;
...
```

[5]Hence the term "page" in "page-based DSM."

[6]Here the word "object" is used in the sense of "object-oriented programming," not in the sense of "object-based DSM."

```
x = 21;
y = x + 5;
```

In PerlDSM, this is written as

```
tie $x,'DSMScalar','$x',$SvrSkt
tie $y,'DSMScalar','$y',$SvrSkt
...
$x = 21;
$y = $x + 5;
```

(In Perl, all scalar variable names begin with a dollar sign.)

The two **tie** statements do the association of scalars to objects as mentioned above. DSMScalar is the name of the PerlDSM built-in class for shared scalars, and $SvrSkt is a socket to the PerlDSM variable server.

Then, just as the actual executable code in the C/C++ version,

```
x = 21;
y = x + 5;
```

is written with no distracting and burdensome calls for network fetch and store operations, the same is true for PerlDSM. In other words, the interface afforded the programmer by PerlDSM is similar to those of page-based DSMs for C/C++.[7]

One of the network nodes runs the PerlDSM shared variable server, DSMSvr.pl.[8] The values of the shared variables are maintained by the server.

The call to tie() for, say $x, triggers a call to TIESCALAR. PerlDSM has TIESCALAR check in with the server, registering this shared variable. The server then adds an entry "$x" to a hash, @SharedVariables. In Perl, a hash is an associative array, indexed by character strings; @SharedVariables{"$x"} will contain the value of $x.

The statement

```
$x = 21;
```

triggers a call to STORE, with argument 21. STORE then sends a message

```
write $x 21
```

to the server, which places 21 into @SharedVariables{"$x"}.

Similarly, the appearance of $x on the right-hand side of

```
$y = $x + 5;
```

will trigger a call to FETCH, which will get the current value of $x from the server.[9] Finally, the appearance of $y on the left-hand side of the assignment triggers a call to STORE, etc.

Both FETCH and STORE in DSMScalar.pm perform the necessary socket communications with the server. But again, it must be emphasized that all of this is transparent to the programmer. The programmer does not see the calls to FETCH and STORE, and thus can concentrate on writing clear, natural code.

4 Other PerlDSM Concurrency Constructs

PerlDSM also includes calls to lock and unlock a lock variable, and a two-phase barrier call. At present there is only one lock variable, $LOCK, and only one barrier variable, $BARR, but the server could easily be altered to allow multiple lock and barrier variables.

Both locks and barriers are implemented internally as reads and writes. A PerlDSM application performs a lock operation as a read of the lock variable $LOCK, and the unlock is done by a write, e.g.

```
$Lock = $LOCK;
... (critical section here)
$LOCK = 0;
```

Barriers are invoked in the same manner, e.g.

```
$Barr = $BARR;
```

Both $LOCK and $BARR must be tied by the application program. Note also that the lock and barrier operations are of course blocking, even though they appear syntactically as reads.

[7] Even though there are no "pages" in PerlDSM.

[8] That same node could also be running the PerlDSM application program.

[9] PerlDSM is also extensible, as discussed later, so that some optimization could be done here to prevent an extra trip to the server.

5 Example

Following is a sample complete PerlDSM application, which finds the average load average among all nodes, by running the UNIX **w** command at each node and then averaging over all nodes. It is very simple, for the sake of brevity, but illustrates all the PerlDSM constructs.

```
#!/usr/bin/perl

# example PerlDSM application; finds the
# average load average among all nodes

# "use" is like C's "#include"
use DSMScalar;
use DSMUtils;

package main;

# globals:
$SvrSkt;  # socket to server
$NumNodes;  # total number of nodes
$MyNode;  # number of this node
$SumAvgs;  # sum of all the load averages

# check in with server
($SvrSkt,$NumNodes,$MyNode) =
   DSMUtils::DSMCheckIn();
print "total of ", $NumNodes,
   " nodes, of which I am number ",
   $MyNode, "\n";

# tie shared variables
tie $BARR,'DSMScalar','$BARR',$SvrSkt;
tie $SumAvgs,'DSMScalar','$SumAvgs',$SvrSkt;
tie $LOCK,'DSMScalar','$LOCK',$SvrSkt;

# initialize sum to 0 if I am node 1
if ($MyNode == 1)  {
   $SumAvgs = 0.0;
}

# barrier
$Barr = $BARR;

# get load average at this node, by running
# UNIX w command, and parsing the output
system 'w > tmpout';
open TMP,"tmpout";
$Line = <TMP>;
close TMP;
@Tokens = split(" ",$Line);
$MyAvg = $Tokens[9];

# add to total of all load averages
$Lock = $LOCK;  # lock
$SumAvgs = $SumAvgs + $MyAvg;
$LOCK = 0;  # unlock

# wait for everyone to finish,
# then print answer if I am node 1
$Barr = $BARR;
if ($MyNode == 1)  {
   print $SumAvgs/$NumNodes, "\n";
}

DSMUtils::DSMCloseSocket($SvrSkt);

exit;
```

6 Summary and Future Work

In developing PerlDSM, we have attained our goal of enabling Perl programming with a parallel DSM world view, with the simplicity and clarity of page-based C/C++ DSM systems. We have made PerlDSM available at `http://heather.cs.ucdavis.edu/~matloff/perldsm.html`.

One question which arises is that of efficiency. It is somewhat less of an issue in the context of an interpreted scripting language like Perl than for C/C++. Parallelizable applications of Perl tend to be I/O-intensive rather than compute-bound, so computational efficiency is less of a concern. However, network efficiency can be important.

Consider for example an atomic increment operation, say on a variable $n:

```
tie $n,'DSMScalar','$n',$SvrSkt;
...
$Lock = $LOCK;
$m = $n;
$n = $m + 1;
$LOCK = 0;
```

This requires sending a total of four requests to the server, i.e. eight network communications. Yet it would be easy to implement such an operation within the PerlDSM infrastructure, entailing only one request to the server and thus only two network communications.

A more complex performance enhancement would

be to allow multiple servers. A number of other enhancements are being considered.

A key element of the implementation of PerlDSM was use of Perl's tie() function. This raises the question of whether our approach here could be extended to other languages.

In C++, for example, one could try overloading the operators for assignment, addition, and so on. That would work to some extent, but not in the clean, elegant fashion that Perl's tie() has afforded us here. We could for example easily implement

```
y = x;
```

but could not directly implement

```
y = x + 5;
```

or

```
printf("%d\n",y);
```

In this sense, it is remarkable that Perl's tie() function allowed us to implement DSM in such an simple, elegant manner, enabling us to avoid working at the level of Perl internals. Indeed, Rice University's team, in developing a DSM package for another interpreted language, Java/DSM [9], needed to resort to modification of the Java Virtual Machine interpreter.

References

[1] C. Amza *et al.* Treadmarks: Shared Memory Computing on Networks of Workstations, *IEEE Computer*, January 1995.

[2] M. Bouchard. MetaRuby. `http://www.ruby-lang.org/en/raa-list.rhtml?name=MetaRuby`.

[3] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs: a First Course*, MIT Press, 1990.

[4] Eric Freeman *et al. JavaSpaces(TM) Principles, Patterns and Practice*, Addison-Wesley, 1999.

[5] Peter Kelleher. Distributed Shared Memory Home Pages, `http://www.csm.ornl.gov/pvm/perl-pvm.html`.

[6] Edward Walker. PVM-Perl (software package), `http://www.csm.ornl.gov/pvm/perl-pvm.html`.

[7] Barry Wilkinson and Michael Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*, Prentice Hall, 1999.

[8] P. Wyckoff. T Spaces. *IBM Systems Journal*, November 3, 1998.

[9] Weimin Yu and Alan Cox. Java/DSM: a Platform for Heterogeneous Computing, *Proceedings of the ACM 1997 Workshop on Java for Science and Engineering Computation*, June 1997.