Performance of Message-Passing MATLAB Toolboxes

Javier Fernández*, Antonio Cañas*, Antonio F. Díaz*, Jesús González, Julio Ortega*, and Alberto Prieto*

Dept. of Computer Technology and Architecture
ETSII, University of Granada, 18071-Granada, Spain
{javier,acanas,afdiaz,jesus,julio,aprieto}@atc.ugr.es
Tele: +34 958 248994
Fax: +34 958 248993

Abstract. In this work we compare some of the freely available parallel Toolboxes for MATLAB, which differ in purpose and implementation details: while DP-Toolbox and MultiMATLAB offer a higher-level parallel environment, the goals of PVMTB and MPITB, developed by us [7], are to closely adhere to the PVM system and MPI standard, respectively. DP-Toolbox is also based on PVM, and MultiMATLAB on MPI. These Toolboxes allow the user to build a parallel application under the rapid-prototyping MATLAB environment. The differences between them are illustrated by means of a performance test and a simple case study frequently found in the literature. Thus, depending on the preferred message-passing software and the performance requirements of the application, the user can either choose a higher-level Toolbox and benefit from easier coding, or directly interface the message-passing routines and benefit from greater control and performance.

Topics: Problem Solving Environments, Parallel and Distributed Computing, Cluster and Grid Computing.

1 Introduction

MATLAB has established itself as a de-facto standard for engineering and scientific computing. The underlying high-quality state-of-the-art numerical routines, the interactive environment, matrix-oriented language and graphical and visualization tools are some of the most frequently cited reasons for its widespread use [18]. Over 400 MATLAB-based books covering virtually any topic [16] and a user community of more than 500.000 [17] account for two more reasons attracting new users to this scientific computing environment.

In recent years a number of parallel Toolboxes for MATLAB have been made available by researchers and institutions (see [7, Table 1], and [3] for a survey). The timeliness of such Toolboxes is evident, taking into account the rapid popularization of clusters of computers as a valid and cost-effective Parallel Computing platform, and the popularity of MATLAB as a software prototyping tool.

* Supported in part by the Spanish MCYT TIC2000-1348 project.

J.M.L.M. Palma et al. (Eds.): VECPAR 2002, LNCS 2565, pp. 228–242, 2003.
© Springer-Verlag Berlin Heidelberg 2003

Following the cluster computing trend, many standard Linux distributions (RedHat, Suse) now include the PVM [11] and LAM/MPI [15, 14] systems as regular packages, making it yet easier for cluster administrators to provide message-passing environments to their users. Together with some of the Toolboxes considered here, the MATLAB environment could then be used as a fast learning tool for the corresponding message-passing library, due to its interactive nature. In this sense, these Toolboxes make it easier for the extended MATLAB user base to turn to Parallel Computing.

On the other hand, previous knowledge of the message-passing library is of course reusable. Even the highest-level Toolbox commands can be conceptually related to the underlying library calls. For instance, the `Shift` command in MultiMATLAB can be related to the `MPI_Cart_Shift()` and `MPI_Sendrecv()` routines. An experienced C/MPI developer might consider prototyping the desired application under MATLAB, taking advantage of its sophisticated numeric and graphic routines, and of its interactive nature as an additional debugging facility, in order to translate (perhaps compile) the final debugged prototype back into C language.

Finally, these Toolboxes can be used merely to reduce the execution time of sequential code in the interpreted MATLAB environment by explicit (message-passing) parallelization. The medium-coarse granularity of many MATLAB prototypes makes them suitable for the so-called *outer loop* [19] parallelization, in agreement with the manufacturer's opinion on the possible uses of parallelism under MATLAB. Computer and interconnect performances, as well as per-node memory size, have all increased since [19] was written, but it remains true that SMP parallelism is better managed by the OS (and possibly a machine-tuned BLAS, see [20]) than by MATLAB, and that the (explicit) parallelization effort should address the coarsest granularity available in the prototype, which is usually well suited to the message-passing paradigm in clusters of computers.

In Section 2 the four Toolboxes considered here are briefly described. Section 3 shows the results of a ping-pong test programmed and run under each Toolbox, and Section 4 evaluates their performance on a case study. Conclusions are summarized in Section 5.

2 Message-Passing Matlab Toolboxes

Table 1 lists the Toolboxes studied in this work. All of them use the MATLAB ability of calling externally compiled C or FORTRAN programs (MEX-files) in order to gain access to the PVM or MPI library routines, offer a (growing with date) number of commands and interfaced library calls, and provide transparent support for most MATLAB data-types. The user is expected to issue a `spawn` command (`Start` in MultiMATLAB) to start remote MATLAB processes which cooperate in a parallel application by explicit message-passing.

Additional information on these Toolboxes is related below:

– The Distributed Parallel Toolbox [21] is a multi-layered project whose low-level layer (`dplow`, 56 commands) interfaces around 47 PVM routines. It

Table 1. Message-Passing MATLAB Toolboxes

Toolbox	library	cmds	date	URL
DP-TB	PVM	56	Mar–99	http://www-at.e-technik.uni-rostock.de/dp
MultiMATLAB	MPI	27	Nov–97	http://www.cs.cornell.edu/Info/People/lnt/ /multimatlab.html
PVMTB	PVM	93	Jun–99	http://atc.ugr.es/javier-bin/pvmtb_eng
MPITB	MPI	153	Feb–00	http://atc.ugr.es/javier-bin/mpitb_eng

was developed at Rostock University. It also includes an intermediate layer (`dphigh`) and a high level environment (DP-MM), consisting of 15 commands, which means the user does not need to learn PVM.
The group library has not been interfaced, so collective functionality (including `pvm_barrier` and `pvm_reduce`) is missing, but could be implemented in terms of the basic DP-TB routines if required. Coverage of the PVM library is greater than 50%. DP-TB works under both Windows and Linux.

– MultiMATLAB [23, 2, 18] is a high level Toolbox which frees the user from learning MPICH. Around 20 of its 27 commands correspond to MPI bindings. It was developed at Cornell University, with cooperation from The MathWorks. CTC is now working on CMTM [4], which runs under Windows and MPI/Pro, thus covering MPI-based MATLAB demand on this platform.
– PVMTB [7, 9] is a complete interface to PVM, including library calls such as `pvm_spawn` and `pvm_tickle`. It also interfaces the `TEV_MASK` macros and allows MATLAB programmed hooks to be registered as hoster, tasker, resource manager or receive functions.
– MPITB [7, 8] is an almost complete interface to the MPI 1.2 standard, excluding the `MPI_Type_*`, `MPI_Op_*` and `MPI_Pcontrol` bindings. It also includes several MPI 2.0 bindings required to spawn instances and communicate with them —`MPI_Info_*`, `MPI_Comm_spawn_*`, `MPI_Comm_get_parent`, `MPI_Comm_accept`, `MPI_Comm_[dis]connect`, `MPI_*_port` and `MPI_*_name`.

Figure 1 shows the role of the MEX interface in these Toolboxes. Both DP-TB and MultiMATLAB utilize the *directory* technique (called *indirection table* in [18]), in which a centralized MEX-file is linked to the message-passing library and provides entry points (table offsets) which are either library call wrappers (e.g. `pvm_config`, `Nproc`) or more highly elaborated routines (`pvme_upkarray` and `Recv` do support MATLAB data-types, for instance). Strictly speaking, only the directory MEX-file in DP-TB and MultiMATLAB sits on top of the message-passing libraries in Fig. 1.

Data is comfortably received in the left-hand side of the MATLAB call, as in DP-TB's `[data,info]=pvm_upkdouble(nitem,stride)` or MultiMATLAB's `[data,src,tag]=Recv(rank)`, allowing for compact MATLAB expressions such as `Sum=Sum+Recv(i)`.

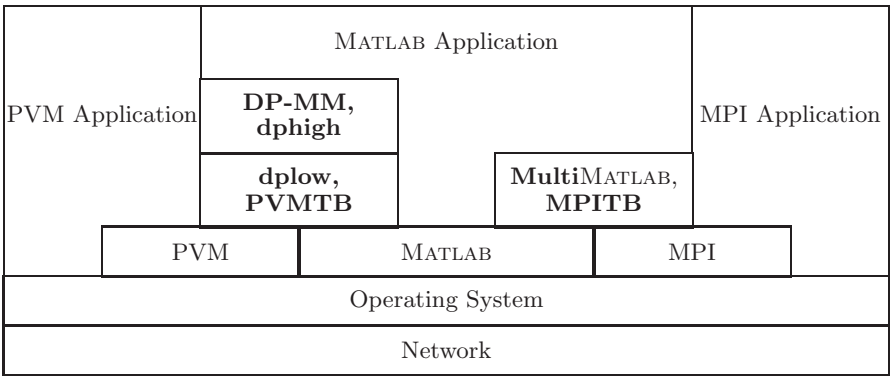


Fig. 1. Overview diagram of Toolbox role. The Toolbox calls PVM/MPI to provide message-passing functionality, like any other PVM/MPI application. It also calls the MATLAB API to pass arguments from/to the MATLAB process. Several MATLAB processes run on distinct cluster computers, each one being an instance of this diagram

Clearly, commands such as MultiMATLAB's `Eval` (or DP-TB's `dpmmeval`) imply coordination with slave processes, which must in some way be enforced at slave process startup. DP-TB uses a MEX *starter code* (`dpmatlab.mex`) for this purpose, equivalent to the MultiMATLAB's "separate top-level MEX routine" [18].

In PVMTB and MPITB designs [7] each message-passing routine has its corresponding MEX wrapper. The libraries have been dynamically linked to avoid disk waste. Source code for the MEX wrappers is extensively reutilized.

Commitment to the original routine parameter list means that PVMTB has commands such as `info=pvm_upkdouble(var,nitem,stride)`, and MPITB has `[info,stat]=MPI_Recv(var,src,tag,comm)`. This requires a more elaborate user code, since `var` is not returned on the left-hand side. Compact MATLAB expressions such as the accumulation shown above must be broken down into two steps (receive-add). However, using this syntax no new storage is allocated for `var`, avoiding additional memory operations in applications where the same storage can be reutilized many times.

With our Toolboxes, MATLAB users can rely on the provided startup M-file for slave MATLAB processes, or design their own M-file *starter codes* in the interpreted MATLAB language (not MEX-files), suited to their own applications. PVM hoster, tasker, resource manager, message handlers and receive functions, as well as MPI error handlers and attribute copy/delete functions, can also be programmed in MATLAB language and registered using the corresponding PVMTB or MPITB commands.

2.1 Spawning Matlab Instances with PVMTB and MPITB

Each PVM-spawned child process has a PVMEPID environment variable (explained in man pvm_reg_tasker(3PVM)) whose existence can be checked in the MATLAB's startup.m file:

```
addpath ( <path to PVMTB> );
if~isempty(getenv('PVMEPID')), startup_Slave;
else,                          disp('Help on PVM: help pvm')
end
```

The same role is played by the LAMPARENT environment variable in MPITB. Spawned MATLAB instances (Fig. 2(c)) can thus be discriminated from the parent instance launched from the user shell (Fig. 2(a)), and forced to perform an additional startup script —startup_Slave in the example above. This script can implement any desired protocol, and is written in MATLAB language.

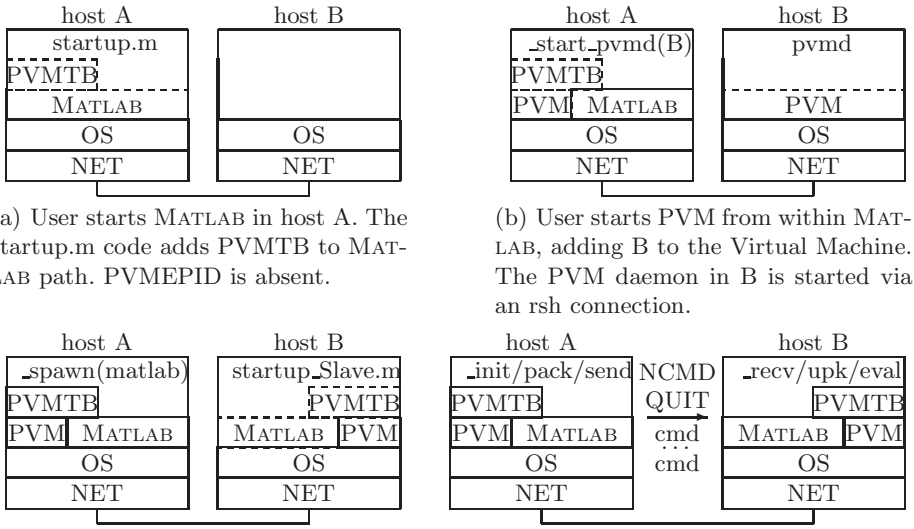
Using the PVMTB commands pvm_start_pvmd (Fig. 2(b)) and pvm_spawn (Fig. 2(c)), the MATLAB user can start the Parallel Virtual Machine and children MATLAB processes from within MATLAB. MPITB users should invoke lamboot from the shell prior to issuing MPI_Init and MPI_Comm_spawn from within MATLAB—since the LAM daemons are not part of the MPI standard, there are no MPI commands to start them.

Most applications only need to specify one or more commands to be executed in the slave instance (Fig. 2(d)). The commands would be sent as strings that could be run on slave computers by using the MATLAB eval built-in. Thus, startup_Slave could be written as:

```
global TAG DEFAULT RAW PLACE,        TAG=7; DEFAULT=0; RAW=1; PLACE=2;
parent=pvm_parent; hostname          % show hostname if "xterm -e matlab"
pvm_recv(parent,TAG); pvm_unpack;    % parent should send NUMCMDS & QUIT
for PVMLoopIdx=1:NUMCMDS
  pvm_recv(parent,TAG); pvm_unpack;  % and NUMCMDS strings called "cmd"
  eval(cmd);                         % containing desired MATLAB commands
end
if QUIT, quit, end
```

A message tag number is reserved for PVMTB use. The parent MATLAB should send two variables to each spawned MATLAB instance: NUMCMDS, indicating the desired number of commands to execute, and QUIT, to finish MATLAB children after the commands are executed. This flag can be cleared while developing or debugging and set when the prototype is finished. It is also customary to spawn xterm -e matlab during the debugging phase, in order to use MATLAB's built-in debugger on children instances.

The pvm_unpack Toolbox command, when called without arguments, will unpack all variables packed in the current receive buffer using their original



(a) User starts MATLAB in host A. The startup.m code adds PVMTB to MATLAB path. PVMEPID is absent.

(b) User starts PVM from within MATLAB, adding B to the Virtual Machine. The PVM daemon in B is started via an rsh connection.

(c) The spawn command causes the pvmd in B to run MATLAB, which in turn sources startup.m. The PVMEPID environment var. is detected and startup_Slave.m is sourced.

(d) MATLAB at B receives NUMCMDS and QUIT, waits for cmd strings, evaluates them as they are received and finally quits if instructed to do so by the QUIT flag.

Fig. 2. Sample control protocol for spawned MATLAB instances

names. So the master is expected to send `NUMCMDS` times a string called `cmd` which will be evaluated at the child instance.

The default `startup_Slave` provided with PVMTB is more elaborate, using a 5s-timed `pvm_trecv` to avoid blocking interactively spawned children (useful for the tutorial), and a conditional branch for `NUMCMDS==0`, so that the loop only ends when `cmd=='break'|'quit'|'exit'` (useful for applications where the total number of commands is not known beforehand).

The `startup_Slave` for MPITB is even more elaborate, building a new intercommunicator which includes the parent and spawned MATLAB processes. The 5s-timed receive is implemented using `MPI_Wtime` and `MPI_Iprobe`.

PVMTB or MPITB users might implement any other conceivable protocol in their startup files, since any desired PVM or MPI call has been interfaced and is available. DP-TB users can reproduce the protocols described here, since all the required PVM calls are interfaced in `dplow`.

A different protocol is used in the following sections, since the child(ren) will run just one (and the same) command. The startup code can be simplified, exporting the command string from the parent as an environment variable under PVMTB/DP-TB, or broadcasting it under MPITB. MultiMATLAB's startup

protocol (included in the top-level C-MEX routine) cannot be avoided or modified using Toolbox commands.

3 Performance Test

The Toolboxes were tested in a cluster of 8 Pentium II 333MHz PCs interconnected with a 100Mbps Fast Ethernet switch. A ping-pong test was programmed under each Toolbox, consisting of two M-files: a master `Ping.m` to be run in the *source* computer, and the slave `Pong.m` script for the *echo* MATLAB process.

In DP-TB and PVMTB, the `Ping.m` script starts the PVM daemon using `pvm[e]_start_pvmd`, the child MATLAB process is run using `pvm_spawn`, and the command (`Pong`) is passed in an environment variable with `pvm_export('CMD')`. This variable is detected and evaluated by the child MATLAB in its `startup.m` script, i.e., the DP-MM starter is not used. Neither are the PVMTB `pvm_psend` and `pvm_precv` commands used; the objective is to produce almost identical code for both Toolboxes. The direct TCP route is prescribed using `pvm_setopt`. The `pvm_barrier` command in the PVMTB `Ping` script fragment shown below has been replaced by `pvm_recv/send` in DP-TB.

```
for SZ=0:ND-1,   array=zeros(SZ,1);  % for each size allocate SZx1 vector
   pvm_barrier('ppong',2); T=clock;  % synch before chrono starts
   for i=1:NTIMES                    % msg-pass loop
      pvm_initsend(PLACE); pvm_pkdouble(array); pvm_send(tid,TAG);
      pvm_recv(tid,TAG);  pvm_upkdouble(array);
   end
   T=etime(clock,T);      Data(SZ+1).time=T/2/NTIMES;
   S=whos('array');       Data(SZ+1).size=S.bytes;
   clear array
end                                  % for SZ
```

The `pvm_initsend` command specifies in-place data encoding. The `Pong` script swaps the order of the two innermost lines (`send/recv`). The `for` loop is repeated `NTIMES=10` times to reduce loop initialization and clock resolution influence on the results. The round-trip time `T/NTIMES` is halved to compute the one-way (transmit, ping) time.

According to [13] (and common practice), the test should have been repeated a number of times in order to obtain reproducible figures (minimum of averages). However, the peaks shown in Fig. 3 can be easily identified and suggest the typical distribution and magnitude of the perturbations that can be expected.

The MPITB code is very similar, using `MPI_Spawn` to start the MATLAB child and `MPI_Intercomm_merge` to create a common intracommunicator. The command is explicitly passed as a string using `MPI_Bcast` in the new communicator. The child's `startup.m` script cooperates in the collective merge and broadcast operations, and then the command string (`Pong`) is evaluated. The `Ping` loop shown below is a direct translation of the former PVM fragment to MPI:

```
MPI_Barrier(NEWORLD); T=clock;     % synch before chrono starts
for i=1:NTIMES                     % msg-pass loop
   MPI_Send(array,1,TAG,NEWORLD);
   MPI_Recv(array,1,TAG,NEWORLD);
end
```

LAM *client-to-client* mode and homogeneous operation is prescribed adjusting the LAM `TROLLIUSRTF` environment variable before `MPI_Init` invocation.

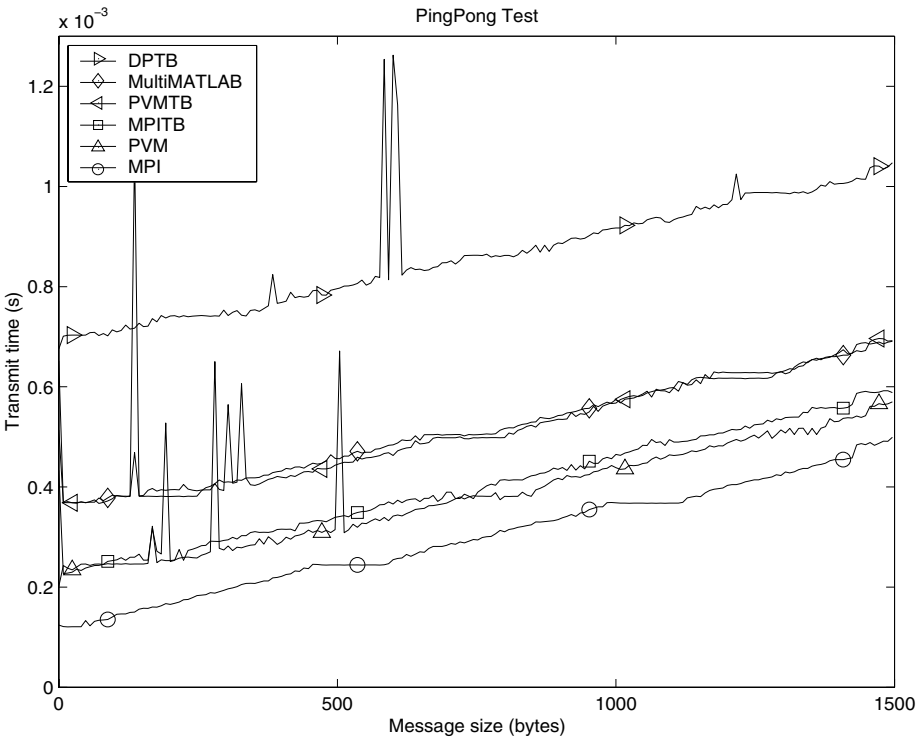


Fig. 3. Results of the ping-pong test

Table 2. Toolbox overhead in the 0–1500 bytes range, segregated by library

Toolbox/library	0B	overhead	1500B	overhead
DP-TB	676 μs	3.350	1047 μs	1.837
MultiMatlab	369 μs	3.037	691 μs	1.384
PVMTB	369 μs	1.827	692 μs	1.214
MPITB	227 μs	1.871	588 μs	1.179
PVM	202 μs	1.000	570 μs	1.000
MPI	121 μs	1.000	499 μs	1.000

The MultiMatlab code is more compact, requiring only a `Start` command to launch the *echo* Matlab process, which becomes the target for the `Eval` command without any additional `startup.m` programming.

```
Eval(1,['Pong(' int2str(ND) ',' int2str(NTIMES) ')']);
for SZ=0:ND-1, array=zeros(SZ,1);    % for each size allocate SZx1 vector
   Barrier;    T=clock;              % synch before chrono starts
   for i=1:NTIMES                    % msg-pass loop
      Send(1,array);
      array=Recv(1);
   end
   T=etime(clock,T);      Data(SZ+1).time=T/2/NTIMES;
   S=whos('array');       Data(SZ+1).size=S.bytes;
   clear array
end                                  % for SZ
```

The tests were run up to 1500 bytes (`ND=188`), and the results are shown in Fig. 3, which also includes C language tests for both PVM and LAM/MPI.

In our setup, LAM has the shortest transmit time, followed by PVM and MPITB, by PVMTB and MultiMatlab, and by DP-TB. PVMTB and MPITB overheads are minimum in the sense that they only add a MEX-file call and input/output argument processing to the respective library call.

Segregating by library, MultiMatlab doubles the MPITB overhead on MPI (18%–38%), and DP-TB overhead is 3–4 times that of PVMTB (21%–84%), due to the additional directory look-up and memory operations required to return the received data on the left-hand side. Table 2 summarizes these conclusions.

4 Application Example

Simple tests like the one above can be misleading. Gropp and Lusk explain in [13] how, as communication software becomes more sophisticated, simple tests become less indicative of the performance of a full application. Some perils of performance measurement are described in a "list of mistakes often made".

In this section we compute π by numeric integration. This example, taken from [22] and cited in [5], is proposed as an exercise in many textbooks. Even this simple, embarrassingly parallel application brings into discussion some of the *common pitfalls* described in [13], such as:

- Ignore overlap of computation and communication.
- Confuse total bandwidth with point-to-point bandwidth.
- Ignore cache effects.
- Use a communication pattern different from the application, measure with just two processors, or measure with a single communication pattern.

Another advantage of this example is that we can show all the relevant Matlab code without getting involved in complex code explanations. Interested readers can find more complex applications in [6] (Control Problems), [7] (Wavelet

Analysis), [10] (Chemical Manufacturing) and [12] (NanoElectronics). In particular, the latter paper includes a detailed scattering model with efficiencies ranging from 98% (2 CPUs, 1.96x speedup) to 84% (120 CPUs, 101x speedup). The Parallel NANOMOS simulator used to implement this model was built on top of PVMTB and is available from the CELAB CVS repository [12].

Returning to the π example, the C sequential code proposed in [5] applies the mid-point integration rule to $\arctan'$ in $[0, 1]$ (there are faster, more accurate methods for approximating π [1]):

```
width = 1.0 / N;       sum = 0;          % subdivision width
for (i=0; i<N; ++i) {                    % subdivision index
  register double x = (i + 0.5) * width; % abscissa mid-points
  sum += 4.0 / (1.0 + x * x);            % area = width * height
}
sum *= width;                 % interchanged width* and 4* for accuracy
```

In this example we chose `N=408800` subdivisions of the $[0, 1]$ interval, accounting for 13 decimal digits of precision and around 0.5s computation time under MATLAB. The loop can be trivially staggered among `C` computers, making each of them compute a subset `for (i=inum-1; i<N; i+=C)`, which only requires an additional *rank* or *instance* number `inum=1...C`. In order to produce intuitive scalability results, time was measured from transmission of arguments to accumulation of results, not including master process setup and spawning of slaves. In the master PVMTB code shown below, only the *inum* argument needs to be transmitted separately (and included in the measurement).

```
cmd=['Work(' int2str(C) ',' int2str(N) ')'];
putenv(['CMD=' cmd]); pvm_export('CMD');
[numt tids]=pvm_spawn('matlab',{'-nosplash'},33,'.',C);
...                                      % Direct TCP route
tic
for numt=1:C                             % inum a-la DP-TB
pvm_initsend(PLACE); pvm_pkdouble(numt); pvm_send(tids(numt),TAG);
end

Psum=0; Sum=0;
for numt=1:C                             % reduce a-la DP-TB
  pvm_recv(-1,TAG); pvm_upkdouble(Sum); Psum=Psum+Sum;
end
Psum = Psum/N;

T=toc;
```

DP-TB allows a more compact `Psum=Psum+pvm_upkdouble(1,1)` expression. As in the ping-pong test, group functionality (`pvm_joingroup` / `pvm_reduce` in this case) was not used under PVMTB to avoid an "apples to oranges" comparison with DP-TB, resulting in very similar code for both Toolboxes.

In this way, not only were the tests run under the same OS and the same library (in each case, the same PVM or LAM/MPI version) compiled with the

same compiler and optimization switches, but also the test codes were kept as similar as possible to each other, avoiding the use of library functionality not available under all Toolboxes.

More complex applications could conceivably benefit from more sophisticated library calls, such as PVM message handlers or mailboxes, elaborate collective MPI operations — or even an `MPI_Sendrecv_replace` call, as in [7], functionality that is not present in all the Toolboxes here considered. For this simple example we can restrict ourselves to the common functionality provided by all Toolboxes without being concerned about hypothetical performance improvements due to the use of different library calls.

The same criterion was followed with MPITB, giving the sentence:

```
for numt=1:C
  MPI_Recv(Sum,MPI_ANY_SOURCE,TAG,NEWORLD); Psum=Psum+Sum;
end
```

The most compact master code is MultiMatlab `Mast.m`:

```
tic
             Psum=0;
Eval([1:C],['Work(' int2str(C) ',' int2str(N) ')'])
for i=1:C ,  Psum=Psum+Recv; end
             Psum=Psum/N;
T=toc;
```

The slave computation loop can be vectorized to avoid its interpretation cost under Matlab. Again, the most compact version is MultiMatlab `Work.m`:

```
rank=ID;
  width=1/N; lsum=0;
  i=rank-1:C:N-1;                % index subset for this rank
  x=(i+0.5)*width;               % vector of abscissa mid-points
  lsum=sum(4./(1+x.^2));         % vectorized - no for loops
Send(0,lsum)
```

MPITB uses `[info rank]=MPI_Comm_rank(NEWORLD)` instead of `ID`, and the more verbose `MPI_Send(lsum,0,TAG,NEWORLD)`, and requires the already mentioned `startup.m` support for merging the communicators, while PVMTB and DP-TB `Work.m` scripts explicitly receive the `inum` index:

```
parent=pvm_parent;
...                                          % Direct TCP route
pvm_recv(parent,TAG); pvm_upkdouble(inum);
  width=1/N; lsum=0;
  ...
pvm_initsend(PLACE); pvm_pkdouble(lsum); pvm_send(parent,TAG);
```

Figure 4 shows the measured time, accounting for slave computation, argument transmit and result collect times. Absolute times are almost indistinguishable, since computation time is in the order of 500ms (expected 62.5ms for `C==8`)

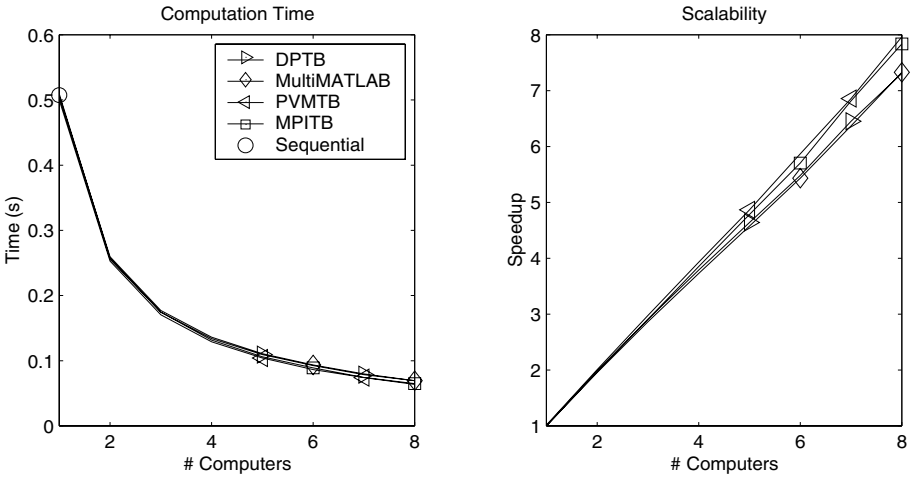


Fig. 4. Results of the scalability test

Table 3. Algorithm Time and Speedup using C=2, 4, 8 computers

Toolbox	Time (ms)			Speedup		
	C=2	C=4	C=8	C=2	C=4	C=8
DP-TB	255.807	133.928	69.440	1.984	3.790	7.309
MultiMatlab	259.408	136.134	69.264	1.957	3.728	7.328
PVMTB	254.102	129.118	63.730	1.997	3.931	7.964
MPITB	257.290	131.676	64.770	1.973	3.854	7.836

while expected transmit time ranges from 0.227ms (MPITB) to 0.676ms (DP-TB, see Table 2), giving computation-to-communication ratios in the order of 10^1–10^3, depending on the number of computers `C` and the Toolbox used. The differences can be highlighted by moving these absolute times to the denominator, as in the associated speedup graph in Fig. 4.

In this application, up to `C==8` messages come back to the master Matlab process, although not in perfect synchronization, due to the initial sequential data distribution. The performance advantage of MPITB over PVMTB in Table 2 is lost in the final accumulation `for` loop in the master process. The greater advantage of MultiMatlab over DP-TB is also lost.

In PVMTB and MPITB, the receive buffer `Sum` is reutilized, while temporal storage is allocated and discarded for each iteration in DP-TB and MultiMatlab code. Table 3 quantifies this small performance loss.

5 Conclusions

Message-passing is the natural programming paradigm for clusters of computers, and several free Toolboxes are available for MATLAB users willing to turn to Parallel Computing in this kind of platform. This is in agreement with the manufacturer's point of view about the possible uses of parallelism under MATLAB: SMP parallelism is better managed by the OS (and possibly a machine-tuned BLAS) than by MATLAB. Also, the parallelization effort should address the coarsest granularity available in the prototype, which is usually well suited to the message-passing paradigm in clusters of computers.

The user can choose the message-passing software preferred, either PVM or an MPI implementation. The small performance differences between the two systems shown in the ping-pong test in Section 3 disappear as soon as MATLAB computation takes place between successive messages, as in the receive-add loop studied in Section 4.

Depending on the granularity of the application, the user can either choose a higher-level Toolbox and benefit from easier, more compact coding, or directly interface the message-passing routines and benefit from greater control and performance. For coarse-grained, embarrassingly parallel applications with high computation-to-communication ratios, the performance loss is very small. Other parallel applications with smaller granularity, complex communication patterns or repeated synchronization or *rendez-vous* steps, might require the higher performance attainable at the cost of more verbose programming.

References

[1] J. Arndt, C. Haenel: "π – Unleashed", Springer-Verlag, 2001, ISBN: 3540665722, **http://www.jjj.de/pibook/pibook.html** 237

[2] C. Chang, G. Czajkowski, X. Liu, V. Menon, C. Myers, A. E. Trefethen, L. N. Trefethen: "The Cornell MultiMATLAB Project", Proceedings of the POOMA'96 Conference, Sanfa Fe, New-Mexico, **http://www.acl.lanl.gov/Pooma96/abstracts/anne/POOMA96.html** 230

[3] R. Choi: Parallel MATLAB survey, **http://supertech.lcs.mit.edu/~cly/survey.html** 228

[4] Cornell Theory Center: "New Parallel Programming Tools for MATLAB", **http://www.tc.cornell.edu/news/releases/2000/cmtm.asp** 230

[5] H. Dietz: "Parallel Processing HOWTO", January 1998, Linux Documentation Project, **http://aggregate.org/PPLINUX/** 236, 237

[6] S. Dormido-Canto, A. P. Madrid, S. Dormido: "Programming on Clusters for Solving Control Problems", Proceedings of the 4th Asian Control Conference ASCC2002, Suntec Singapore, Singapore, September 2002, Session WA9-11 PaperID 1343 in 236
http://www.ece.nus.edu.sg/ascc2002/advprog/Prog_Summary.pdf
http://atc.ugr.es/~javier/investigacion/papers/Seb-ASCC.pdf

[7] J. Fernández-Baldomero: "Message Passing under MATLAB", Proceedings of the HPC 2001 (Adrian Tentner, Ed.), pp.73–82. ASTC 2001, Seattle, Washington, 228, 230, 231, 236, 238
http://www.scs.org/confernc/astc/astc01/prelim-program/html/hpc-pp.html
http://atc.ugr.es/~javier/investigacion/papers/H010FernandezBaldomero002.pdf

[8] J. Fernández-Baldomero: MPITB home page,
http://atc.ugr.es/javier-bin/mpitb_eng 230
[9] J. Fernández-Baldomero: PVMTB home page,
http://atc.ugr.es/javier-bin/pvmtb_eng 230
[10] V. García-Osorio, B. E. Ydstie: "Parallel, Distributed Modeling of Simulation of Chemical Systems", American Institute of Chemical Engineers, Annual Meeting, Reno NV, November 2001. **http://www.aiche.org/conferences/techprogram/ paperdetail.asp?PaperID=2009&DSN=annual01** 237
[11] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, V. Sunderam: "PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing", The MIT Press, Cambridge, Massachusetts, 1994, **http://www.netlib.org/pvm3/book/pvm-book.html** 229
[12] S. Goasguen, A. Butt, K. D. Colby, M. S. Lundstrom: "Parallelization of the Nanoscale Device Simulator nanoMOS 2.0 using a 100 Nodes Linux Cluster", Proceedings of the 2nd IEEE Conference on Nanotechnology IEEE-NANO'02, Arlington VA, USA, August 2002, Session WA7#0840 in
http://www.ewh.ieee.org/tc/nanotech/nano2002/techprog.html
http://atc.ugr.es/~javier/investigacion/papers/SebNano02.pdf
http://atc.ugr.es/~javier/investigacion/papers/IEEE-nano.pdf
http://falcon.ecn.purdue.edu:8080/cgi-bin/cvsweb.cgi/ 237
[13] W. Gropp, E. Lusk: "Reproducible Measurements of MPI Performance Characteristics", Proceedings of the EuroPVM/MPI'99 Conference, Barcelona, Spain, September 1999,
http://www-unix.mcs.anl.gov/~gropp/bib/papers/ 234, 236
[14] W. Gropp, E. Lusk, A. Skjelum: "Using MPI: Portable Parallel Programming with the Message-Passing Interface", The MIT Press, 1994,
http://www.mcs.anl.gov/mpi/usingmpi/index.html 229
[15] LAM Home Page, **http://www.lam-mpi.org** 229
[16] The MathWorks, Inc.: "MATLAB based books" web page,
http://www.mathworks.com/support/books 228
[17] The MathWorks, Inc.: "User stories" web page,
http://www.mathworks.com/products/user_story
http://www.mathworks.com/products/matlab/description/overview.shtml 228
[18] V. S. Menon, A. E. Trefethen: "MultiMATLAB: Integrating MATLAB with High-Performance Parallel Computing", Proceedings of Supercomputing'97, ACM SIG ARCH and IEEE Computer Society, 1997,
http://www.supercomp.org/sc97/proceedings/TECH/MENON/INDEX.HTM
228, 230, 231
[19] C. Moler: "Why there isn't a parallel MATLAB", MATLAB news & notes, Cleve's Corner Spring 1995,
http://www.mathworks.com/company/newsletter/pdf/spr95cleve.pdf 229
[20] C. Moler: "MATLAB incorporates LAPACK", MATLAB news & notes, Cleve's Corner Winter 2000, **http://www.mathworks.com/company/newsletter/clevescorner/ winter2000.cleve.shtml** 229
[21] S. Pawletta, W. Drewelow, P. Duenow, T. Pawletta, M. Suesse: "A MATLAB Toolbox for Distributed and Parallel Processing", Proceedings of the MATLAB'95 Conference (C. Moler, S. Little, Eds.), MathWorks Inc., Cambridge, MA, October 1995, **http://citeseer.nj.nec.com/pawletta95matlab.html** 229
[22] M. J. Quinn: *"Parallel Computing Theory and Practice, 2nd Edition"*, McGraw Hill, New York, 1994. 236

[23] A. E. Trefethen, V. S. Menon, C. C. Chang, G. J. Czajkowski, C. Myers, L. N. Trefethen: "MultiMatlab: Matlab on Multiple Processors", Tech. Report 96-239, Cornell Theory Center, 1996, **ftp://ftp.tc.cornell.edu/pub/tech.reports/tr239.ps**
230