MDICE – a MATLAB Toolbox for Efficient Cluster Computing

R. Pfarrhofer[a], P. Bachhiesl[a], M. Kelz[a], H. Stögner[a], and A. Uhl[a,b]

[a]Carinthia Tech Institute, School of Telematics & Network Engineering
Primoschgasse 8, A-9020 Klagenfurt, Austria

[b]Salzburg University, Department of Scientific Computing
Jakob-Haringerstr.2, A-5020 Salzburg, Austria

A MATLAB-based toolbox for efficient computing on homogenous and heterogenous Windows PC networks is introduced. The approach does not require a MATLAB client installed at the participating machines and allows other users to employ the involved machines as desktop. Experiments involving a Monte Carlo simulation demonstrate the efficiency and real-world usability of the approach.

1. Introduction

MATLAB has established itself as the numerical computing environment of choice on uniprocessors for a large number of engineers and scientists. For many scientific applications, the desired levels of performance are only obtainable on parallel or distributed computing platforms. With the emerge of cluster computing and the potential availability of HPC systems in many universities and companies, the demand for a solution to employ MATLAB on such systems is obvious. A comprehensive and up-to-date overview on high performance MATLAB systems is given by the "Parallel MATLAB Survey" at `http://supertech.lcs.mit.edu/~cly/survey.html`. Several systems may be downloaded from `ftp://ftp.mathworks.com/pub/contrib/v5/tools/` and also from `http://www.mathtools.net/MATLAB/Parallel/`. There are basically three distinct ways to use MATLAB on HPC architectures:

1. Developing a high performance interpreter
   (a) Message passing: communication routines usually based on MPI or PVM are provided. These systems normally require users to add parallel instructions to MATLAB code [1,6,7].
   (b) "Embarrassingly parallel": routines to split up work among multiple MATLAB sessions are provided in order to support coarse grained parallelization. Note that the PARMATLAB and TCPIP toolboxes our own development is based upon fall under this category.
2. Calling high performance numerical libraries: parallelizing libraries like e.g. SCALAPACK are called by the MATLAB code [9]. Note that parallelism is restricted within the library and higher level parallelism present at algorithm level cannot be exploited with this approach.
3. Compiling MATLAB to another language (e.g. C, HPF) which executes on HPC systems: the idea is to compile MATLAB scripts to native parallel code [2,3,8]. This approach often suffers from complex type/shape analysis issues.

Note that using a high performance interpreter usually requires multiple MATLAB clients whereas the use of numerical libraries only requires one MATLAB client. The compiling approach often does not require even a single MATLAB client. On the other hand, the use of numerical libraries and compiling to native parallel code is often restricted to dedicated parallel architectures like multicomputers or multiprocessors, whereas high performance interpreters can be easily used in any kind of HPC environment. This situation also motivates the development of our custom high performance MATLAB environment: since our target HPC systems are (heterogenous) PC clusters running a

Windows system based on the NT architecture, we are restricted to the high performance interpreter approach. However, running a MATLAB client on each PC is expensive in terms of licensing fees and computational resources. Consequently, our aim in this work is to develop a high performance interpreter which requires one MATLAB client for distributed execution only.

In section 2, we present the fundamentals of our development MDICE. Section 3 describes the basics of an application from the area of numerical mathematics (Monte Carlo simulation) and discusses the respective experimental results applying MDICE. Section 4 concludes the paper.

2. "MDICE" – a Toolbox for Efficient MATLAB Cluster Computing

MATLAB-based **DI**stributed **C**omputing **E**nvironment (MDICE) is based on the PARMATLAB and TCPIP toolboxes. The PARMATLAB toolbox supports coarse grained parallelization and distributes processes among MATLAB clients over the intranet/internet. Note that each of these clients must be running a MATLAB daemon to be accessed. The communication within PARMATLAB is performed via the TCPIP toolbox. Both toolboxes may be accessed at the Mathworks ftp-server (referenced in the last section) in the directories `parmatlab` and `tcpip`, respectively.

However, in order to meet the goal to get along with a single MATLAB client the PARMATLAB toolbox needs to be significantly revised. The main idea is to change the client in a way that it can be compiled to a standalone application. At the server, jobs are created and the solve routine is compiled to a program library (*.dll). The program library and the datasets for the job are sent to the client. The client is running as background service on a computer with low priority. For this reason the involved client machines may be used as desktop machines by other users during the computation (however, this causes the need for a dynamic load balancing approach of course). This client calls over a predefined standard routine the program library with the variables sent by the server and sends the output of the routine back to the server. After the receipt of all solutions the server defragments them to an overall solution. The client-server approach is visualized in Fig. 1.

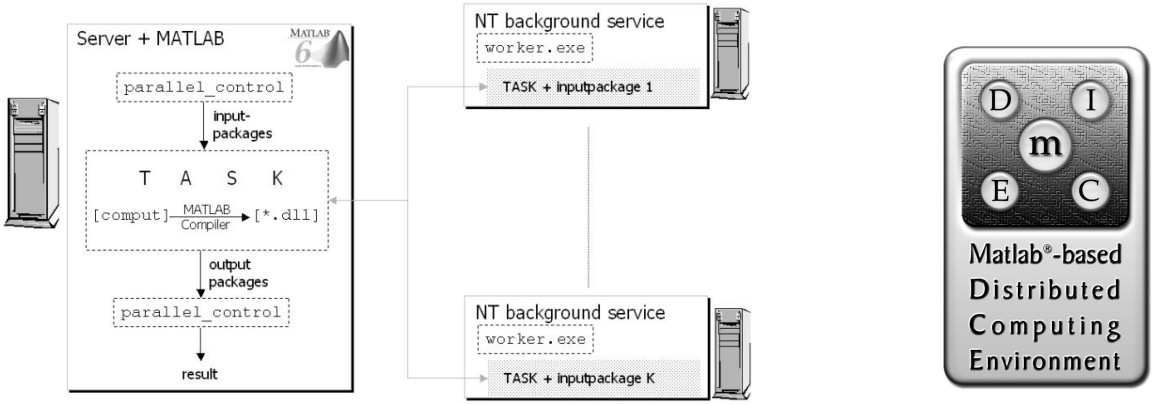


Figure 1. Client-server concept of MDICE.

In order to implement this concept, compilation limitations and restrictions need to be considered as follows:

- Built-in MATLAB functions cannot be compiled. However, most of these functions are available since they are contained in the MATLAB Math C/C++ Library. This library can be transferred to an arbitrary number of clients without the payment of additional licence fees which enables the execution of corresponding code without running MATLAB on the clients. About 70 functions (e.g. `diary`, `help`, `whos`, etc.) are not supported which need to be replaced by custom code if required by the application.

- The arguments of `load`, `save`, and `exist` (usually file names) need to be known at compile-time.

- The arguments of `eval`, `input`, and `feval` (usually data variables) need to be known at compile-time.

For example, `feval` is used in the PARMATLAB toolbox to evaluate the function residing at the client using the arguments sent by the server. It is avoided by sending the *.dll library to the client which has a fixed interface (`fun_task`) for each configuration of input and output variables.

The following example illustrates the replacement of the functions `eval` and `diary`. The code is taken from the file `worker.m` where the result is being sent back to the server after the computation has been done.

```
Original PARMATLAB Code:
========================
%%%% SEND ARGUMENTS
disp('Sending output arguments')
sendvar(ip_fid,hostname)
for i=1:func.argout
 eval(['sendvar(ip_fid,' ...
          'argo' int2str(i) ')'])
end
```

```
MDICE Code:
===========
%%%% SEND ARGUMENTS
displog('Sending output arguments');
sendvar(ip_fid,hostname);
for i=1:func.argout,
 sendvar(ip_fid,var_argout(i).data);
end;
```

In order to facilitate compilation, the functions `disp` and `eval` need to be replaced since `diary` (which uses `disp` output) can not be compiled and `eval` requires arguments known at compile-time. Instead of `disp` we introduce the custom function `displog` which may additionally write data to a log-file besides displaying it. This is especially important if the client is operated in background mode. In the original code, the outgoing arguments of the function are stored in several variables (argo1, argo2, etc.) and the `sendvar` command is constructed dynamically using `eval`. Since this can not be compiled in this form, the results are stored in a variable of type "struct array".

The communication functionalities of the PARMATLAB toolbox have been extended as well. For example, in case of fault-prone file transmission (e.g. no space left on the clients' hard disk) the server is immediately notified about the failure. The underlying TCPIP toolbox requires all data subject to transmission to be converted into strings. For large amounts of data this is fairly inefficient in terms of memory demand and computational effort. In this case, we store the data as MAT-file, compress it (since these files are organized rather inefficiently), and finally convert it into strings. After the computation is finished and the result has been sent, a new job may be processed by the client. Note that the *.dll library and constant variables do not have to be resent since the client informs the server about its status.

MDICE does not support any means of automatic parallelization or automatic data distribution. The user has to specifiy how the computations and the associated data have to be distributed among the clients. The same is true of course for the underlying PARMATLAB toolbox.

3. Applications and Experiments

The computational tasks of the applications subject to distributed processing are split into a certain number of equally sized jobs N to be distributed by the server among the M clients (usually $N \geq M$). Whenever a client has sent back its result to the server after the initial distribution of M jobs to M clients, the server assigns another job to this idle client until all N jobs are computed. This approach is denoted "asynchronous single task pool method" [5] and facilitates dynamic load balancing in case of $N \gg M$. The computing infrastructure consists of the server machine (1.99 GHz Intel Pentium 4, 504 MB RAM, Windows XP Prof.) and two types of client machines (996 MHz Intel Pentium 3, 128 MB RAM, and 730 MHz Intel Pentium 3, 128 MB RAM, both types under Windows XP Prof.). The Network is 100 MBit/s Ethernet. In order to demonstrate the flexibility of our approach, we present results in "homogenous" and "heterogenous" environments. In the case of the homogenous environment, we use client machines of the faster type only, the results of the heterogenous environment correspond to six 996 MHz and four 730 MHz clients, respectively. Note that the sequential reference execution times used to compute speedup have been achieved on a 996 MHz client machine with a compiled (not interpreted) version of the application to allow a fair comparison since the client code is compiled as well in the distributed application. We use MATLAB 6.5.0 with the MATLAB compiler 3.0 and the LCC C compiler 2.4.

3.1. Numerical Mathematics: Monte Carlo Simulation

This problem is known as part of the ArgeSim comparison of parallel simulation techniques[1]. A damped second order mass-spring system is described by the equation

$$m\ddot{x}(t) + kx(t) + d\dot{x}(t) = 0$$

with $x'(0) = 0$, $x(0) = 0.1$, $k = 9000$, and $m = 450$. The damping factor d should be chosen as a random quantity uniformly distributed in the interval $[800, 1200]$. The task is to perform a couple of simulation runs and to calculate and store the average responses over the time interval $[0, 2]$ for the motion $x(t)$ with step size 0.0005. This can be trivially distributed by generating different random quantities d on different clients.

In order to solve the differential equation for a single value of d (which is done on each client), we use the Runge-Kutta-Nyström algorithm [4, p. 960]. Using this approach, a second order ordinary differential equation (ODE) does not need to be decomposed into a system of ODEs of order one. For a given initial value problem of second order

$$\ddot{y} = f(t, y, \dot{y}); \quad y(t = t_0) = y_0; \quad \dot{y}(t = t_0) = \dot{y}_0$$

we compute in each iteration step of the fourth order Runge-Kutta-Nyström algorithm the following experessions:

$$\begin{aligned} y_{i+1} &= y_i + \dot{y}_i h + \frac{h}{3}(k_1 + k_2 + k_3), \\ \dot{y}_{i+1} &= \dot{y}_i + \frac{1}{3}(k_1 + 2k_2 + 2k_3 + k_4), \\ t_{i+1} &= t_i + h \end{aligned}$$

where $k_1 = \frac{h}{2} f(t_i, y_i, \dot{y}_i)$, $k_2 = \frac{h}{2} f(t_i + \frac{h}{2}, y_i + \frac{h}{2}\dot{y}_i + \frac{h}{4}k_1, \dot{y}_i + k_1)$, $k_3 = \frac{h}{2} f(t_i + \frac{h}{2}, y_i + \frac{h}{2}\dot{y}_i + \frac{h}{4}k_1, \dot{y}_i + k_2)$, and $k_4 = \frac{h}{2} f(t_i + h, y_i + h\dot{y}_i + hk_3, \dot{y}_i + 2k_3)$. For each d, we iterate this procedure 4001 times.

3.2. Experimental Results

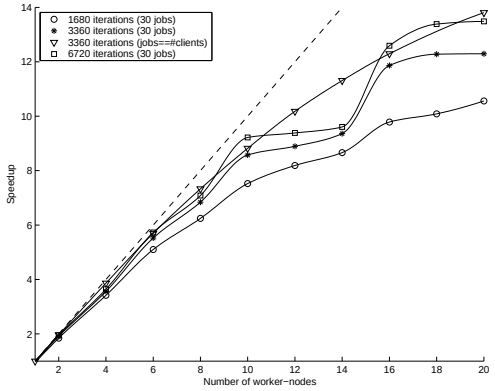
Monte Carlo algorithms are well known to allow for straightforward parallelism. In our case, the partial solution for $x(t)$ within each job is computed on the clients, consequently the server only needs to average N (number of jobs) partial results after the clients have finished their work.

We first discuss the homogeneous environment. Fig. 2.a shows the speedup of this application when varying the problemsize (# of iterations which denotes the number of random quantities d used in the entire computation) and keeping the number of jobs distributed among the clients fixed (30). Sequential execution time is 452, 902, and 1803 seconds for 1680, 3360, and 6720 iterations, respectively. As it is expected, speedup increases with increasing problemsize due to the improved computation/communication ratio.

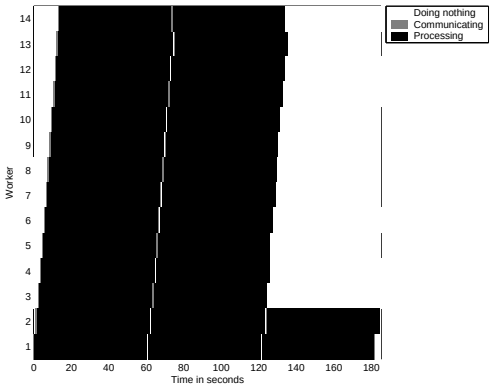
However, also the plateaus resulting from load distribution problems are more pronounced for larger problem size (e.g., 30 jobs may not be efficiently distributed among 14 clients whereas this is obviously possible for 10 clients). Fig. 2.b shows a visualization of this phenonemon during the distributed execution, where black areas represent computation time-intervals, gray areas communication events, and white areas idle times.

The plateaus in speedup can be avoided in two ways. First, by setting $N = M$ as depicted in Fig. 2.a. However, this comes at the cost of entirely losing any load balancing possibilities which would be required in case of interfering other applications on the clients (see below). Second, by increasing the number of jobs. Clearly, a high number of jobs leads to an excellent load distribution, but on the other hand the communication effort is increased thereby reducing the overall efficiency. The tradeoff between communication and load distribution is inherent in the single pool of task approach which means that the optimal configuration needs to be found for each target environment. These facts are shown below in the context of the heterogeneous environment.

[1] http://argesim.tuwien.ac.at/comparisons/cp1/cp1.html

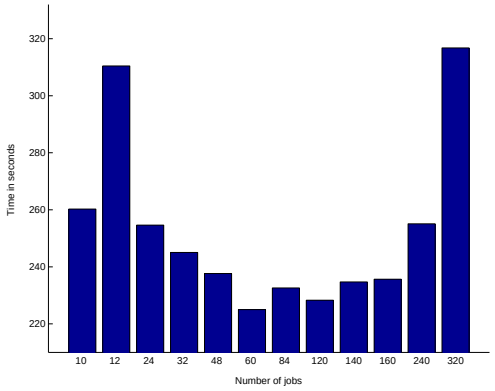


(a) Speedup, varying problem-size

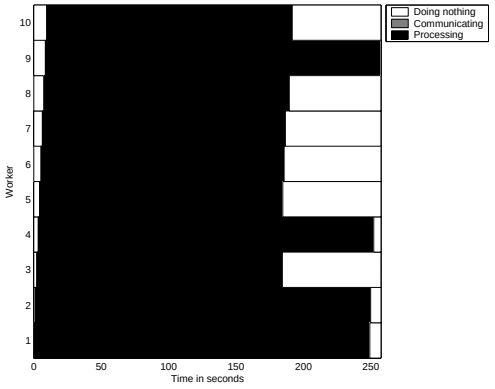


(b) Visualization of execution with 14 clients and 30 jobs (6720 iterations)

Figure 2. Results of Monte Carlo Simulation in homogeneous environment.



(a) Fixed problem-size (6720 iterations), varying job number



(b) Vizualization of execution with 10 jobs on 10 clients

Figure 3. Results of Monte Carlo simulation in the heterogeneous environment.

Lower speedup and less pronounced plateaus are exhibited in case of smaller problem size (Fig. 2.a). Here, the expensive initial communication phase (where the server has to send the compiled code and input data to each of the clients) covers a significant percentage of the overall execution time. This leads to a significantly staggered start of the computation phases at different clients which dominates the other load inbalance problems.

Next we discuss the heterogeneous environment. Fig. 3.a shows the time demand for computing the solution using a fixed problem size (again 6720 iterations) while changing the number of jobs used to distribute the amount of work among the 10 clients (as defined in the last subsection). For our test configuration, the optimal number of jobs is identified to be around 60. A further increase leads to an increase of execution time as well as a lower number causes worse results. Fig. 3.b shows an execution visualization where $N = M = 10$. Of course, the execution is not balanced due to the slower clients involved and these machines (numbers 1,2,4,9) are immediately identified in the figure.

Whereas for $N = 60$ we have relatively little communication effort but obviously sufficiently balanced load, the higher amount of communication required to achieve better balanced load for $N \geq 84$ degrades the overall performance of these configurations.

Finally, we investigate the impact of interfering applications running as a desktop application on MDICE client machines. For that purpose, we execute our Monte Carlo simulation code in sequential (1680 iterations, 452 seconds sequential execution time) on two out of four clients running the Monte Carlo code under MDICE (6720 iterations).

Table 1
Effects of interfering application running on MDICE client machines.

Execution time (seconds)	4 jobs	12 jobs	30 jobs
MDICE MC (6720 iter., 4 dedicated clients)	462	463	495
MDICE MC (6720 iter., seq. MC on 2 clients)	915	761	706
Sequential MC (1680 iterations, 452 sec.)	458	458	458

Table 1 shows at first that increasing the number of jobs in a homogeneous system (4 identical client machines) in fact decreases execution performance. However, similar to the heterogeneous case, increasing the job number in case of load inbalance (here caused by the sequential code running on two machines) clearly improves the results (from 915 seconds using 4 jobs to 706 seconds using 30 jobs). The third line of the table shows that no matter how many jobs are employed within MDICE, the impact of MDICE on running desktop applications remains of minor importance (below 2% of the overall execution time).

4. Conclusion

The custom MATLAB-based toolbox MDICE may take advantage of the large number of Windows NT based machines available in companies and universities. The most important property of MDICE is that no MATLAB client is required on the participating machines. Based on the results obtained from a Monte Carlo simulation, it proves to be very flexible and efficient in terms of low licencing fees and execution behaviour.

REFERENCES

[1] J.F. Baldomero. PVMTB: Parallel Virtual Machine Toolbox. In S. Dormido, editor, *Proceedings of II Congreso de Usarios MATLAB'99*, pages 523–532. UNED, Madrid, Spain, 1999.

[2] L. DeRose and D. Padua. A MATLAB to Fortran 90 translator and its effectiveness. In *Proceedings of 10th ACM International Conference on Supercomputing.* ACM SIGARCH and IEEE Computer Society, 1996.

[3] P. Drakenberg, P. Jakobson, and B. Kagström. A CONLAB compiler for a distributed memory multicomputer. In *Proceedings of the 6th SIAM Conference on Parallel Processing for Scientific Computing*, volume 2, pages 814–821, 1993.

[4] E. Kreyszig. *Advanced Engineering Mathematics, 8th edition.* Wiley Publishers, 1999.

[5] A.R. Krommer and C.W. Überhuber. Dynamic load balancing – an overview. Technical Report ACPC/TR92-18, Austrian Center for Parallel Computation, 1992.

[6] V.S. Menon and A.E. Trefethen. MultiMATLAB: integrating MATLAB with high performance parallel computing. In *Proceedings of 11th ACM International Confernce on Supercomputing.* ACM SIGARCH and IEEE Computer Society, 1997.

[7] S. Pawletta, T. Pawletta, and W. Drewelow. Comparison of parallel simulation techniques - MATLAB/PSI. *Simulation News Europe*, 13:38–39, 1995.

[8] M. Quinn, A. Malishevsky, N. Seelam, and Y. Zhao. Preliminary results from a parallel MATLAB compiler. In *Proceedings of the International Parallel Processing Symposium (IPPS)*, pages 81–87. IEEE Computer Society Press, 1998.

[9] S. Ramaswamy, E.W. Hodges, and P. Banerjee. Compiling MATLAB programs to SCALAPACK: Exploiting task and data parallelism. In *Proceedings of the International Parallel Processing Symposium (IPPS)*, pages 814–821. IEEE Computer Society Press, 1996.