USING DCT/DCE IN PARALLEL ROTOR COMPUTATIONS

O. Jakl, T. Musil

Institute of Geonics, Czech Academy of Sci. & VŠB – Technical University Ostrava

Abstract

Four years ago, The MathWorks, the producer of Matlab, fulfilled the wishes of many users and released the first version of the DCT/DCE tools enhancing Matlab's capabilities towards parallel processing, to take advantage of the modern multiprocessor/multicore computer architectures. We followed DCT/DCE's development, investigated this computing environment from the point of view of the "classical" parallel programming and took advantage of it in a set of Matlab codes for modelling of nonlinear dynamics of rotors. This paper[1] summarizes our (throughout positive) experience.

1 Introduction

Matlab [1] since its introduction in the seventies has developed to a mighty general-purpose mathematical solver with many add-on capabilities and toolboxes, which effectively finds a continuously growing number of applications in all fields of science and technology. The users, stretching nowadays far beyond the mathematical community, appreciate that it is rich of features, efficient, user friendly, extensible, multiplatform, continuously developing, etc., etc. Moreover, Matlab gains new clientele also among the users of traditional languages such as C or Fortran, who change to it to reduce the application development cycle and get acceptable performance of the resulting code in most cases.

However, those users that wanted to take advantage of Matlab for the implementation of really demanding numerical algorithms, have been missing one important feature in it: the support for parallel processing. For demanding computations, parallel processing is namely imperative nowadays. It was as late as in 1995, when Cleve Moler, the author of the first version of Matlab and the chief scientist at The MathWorks, argued against parallel processing in Matlab in [4]. But its users had different opinion — there were many attempts to enrich Matlab with some form of parallel processing without support of The MathWorks. [6] mentions 27 projects of this kind in 2003.

The development in IT promoting parallelism of many forms made The Mathworks to change its negative standpoint (see Cleve Moler's explanation in [5]) and to accelerate the development of "parallel Matlab" of its own. It appeared in 2004 and represents a border stone in Matlab's history.

We believe that in deserves much more attention also in the *Technical Computing Prague* conference series, where, to our best knowledge, so far only one contribution [9] appeared in the past.

2 On the Development of Parallel Matlab

The MathWorks' parallel Matlab appeared in 2004 in form of two products: *Distributed Computing Toolbox* (DCT) [2] and *Distributed Computing Engine* (DCE) [3]. DCT is a toolbox that adds new constructs to the Matlab's C-like scripting programming language to provide parallel functionality. DCE, a parallel run-time environment, provides the backend (behind-the-scenes) support for the DCT on larger parallel systems and is quite transparent to users.

[1]We acknowledge support of the projects FRVŠ 1720/2008 and AV ČR S3086102.

Version 1 of DCT/DCE let you develop just coarse-grained parallel algorithms, which give rise to independent tasks without mutual interaction. This (in Matlab's somewhat misleading terms) "distributed" computing model follows in fact the *manager-worker task-scheduling* scheme and is suitable for *embarrassingly/perfectly parallel* problems only, the concurrent workers of which are not allowed to change data during the computation: They can only return their partial results to the master.

Since this initial release, The Mathworks is strongly pushing forward the development of DCT/DCE, resulting in a sequence of new version with substantial improvements released every year.[2]

The most important novelty came with DCT/DCE version 2 in 2005: The support for explicit communication among interdependent tasks, taking advantage of the industry-standard *Message Passing Interface* (MPI) [7]. Functions for send, receive, broadcast, barrier, and probe operations are available in the toolbox. This makes DCT/DCE to a *message passing system* and allows to develop true *parallel* applications based on the general *message passing model* without substantial restrictions. Although those users, who are familiar with MPI, might miss some advanced features of its rich suite of message-passing routines, DCT provides a reasonable abstract of complex details, which makes them easy to use, just in the spirit of Matlab programming, clearly characterized by Cleve Moller's "Think matrices, not messages". There are other benefits of this approach, e.g. the deadlock detection capability helps timely identification of mismatched send-receive calls, potentially saving time that would be spent on these hard-to-debug problems.

Version 3 of 2006 presented additional interesting parallel constructs, *distributed arrays* and the `parfor` loop, where one can recognize strong motivation from the so called *data parallel* approach typical for e.g. *High Performance Fortran* (HPF).[3]

In fact, The MathWorks recommends using these higher-level constructs for easy-to-understand, maintainable code, whereas the low-level message passing functions are meant for finer control over the parallelization scheme. Note that while using these functions, the programmer bears the responsibility of managing the synchronization between concurrent sections of his code.

From the user's budget point of view the good news is, that since version 3.1 (2007) one can make use of up to four processors/cores of a symmetric multiprocessor machine without purchasing the rather expensive DCE licence.

After completing the portfolio of the main parallelization constructs in DCT, The MathWorks returned to the improve the API in recent versions (parallel profiler, administration GUI, configurations manager, etc.). Moreover, as a tax for rapid development, some corrections in the language had to be made, e.g. the `parfor` loop was modified in syntax and semantics, which may enforce modifications of older codes. And finally, since version 3.3, the parallel Matlab changed its somewhat misleading name: DCT to *Parallel Computing Toolbox* and DCE to *Distributed Computing Server*. From pragmatic reasons, we shall continue to use the original names in this article.

3 Parallel Matlab in Rotor Computations

Our concrete experience with DCT/DCE arises primarily from an application in computations of nonlinear dynamics of rotors. See [8] for more informations on the physical and mathematical background.

[2] This can be demonstrated on the size of the DCT manual: Version 2.0 (2005) – 224 pages, version 3.0 (2006) – 374 pages, Version 3.2 (2007) – 487 pages.

[3] Interestingly, HPF was not a very successful programming language.

3.1 Modelling of the Dynamics of Rotors

The study of dynamics of rotors with journal bearings is very important because the majority of high performance industrial rotating machinery is supported by this kind of bearings, thanks to their high capacity and damping and low friction dissipation. On the other hand, poorly designed journal bearings tend to be unstable — they can collapse without any previous sign of failure and cause serious damage of the whole machine.

From the point of view of mechanics, journal bearings introduce a strong local non-linearity into the computational model of the rotor system as these bearings are usually incorporated by means of non-linear force couplings. It is necessary to know the pressure distribution in the bearings to determine the components of the coupling vector. The pressure function is described by an elliptical partial differential equation together with appropriate boundary conditions. In the literature, this equation is reffered to as the *Reynolds equation* and its analytical solution is known for some special cases only.

The computational model of the rotor itself, see Fig. 1, is typically discretized into a few dozens of finite elements, which incorporate the influence of the inertia and gyroscopic effects. The shaft is represented by a beam-like body and external and viscous damping can be also included. The wheels are commonly supposed to be thin axisymmetric rigid bodies.

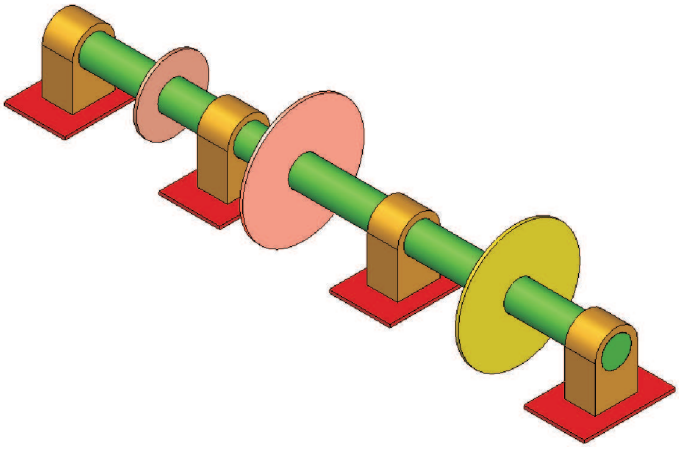

Figure 1: A model of a rotor with four bearings

For testing purposes, how developed algorithms reacts on increasing computational load, the model rotor is subsequently supported by two, three and four identical journal bearings.

The numerical solution of the Reynolds equation and of the subsequent numerical integration of the pressure function is a computationally demanding procedure. Profiling of the (sequential) algorithm revealed that the total execution time of the matrix operations is insignificant compared to the time needed for the repeated solution of the Reynolds equation. This is the reason for the long execution times — not the "large" system of algebraic equations arising from the discretization of the rotor assembly. In this situation, parallel processing might be the proper way how to shorten the computation.

The basis analyses made on nonlinear rotor systems include (1) calculation of the equilibrium position and evaluation of its stability, (2) determination of the response of the system on periodic or harmonic excitation and (3) stability assessment of periodic steady-state response. We have developed and implemented parallel algrithms for all those computations, but for the demonstration of DCT let us describe just the last one in more detail.

3.2 Calculating the Steady-state Response on Periodic Excitation

If the steady-state response of rotors on periodic excitation needs to calculated, it is appropriate to use some of the approximate methods. Among these, the *trigonometric collocation method*,

which the steady-state response approximates by a finite number of terms of Fourier series, is very efficient. The determination of the response is transformed to the solution of a set of non-linear algebraic equations, of order that is proportional to the number of the bearings' parameters and the terms in the Fourier series, and the *Newton-Raphson's method* was applied.

Since the pressure distribution in a bearing depends only on displacements and velocities of the shaft at the bearing, the components of the coupling vector can be evaluated concurrently, without any mutual dependency, in an embarrassingly parallel computation. The calculation of hydraulical bearing forces in one bearing (or in a group of bearings) can be chosen as one unit of parallel processing. This fact can be easily made use of e.g. when Jacobi matrix is to be assembled (expensive operation).

3.3 Look and Feel of Matlab Parallel Programming

Matlab's parallel job is based on a single task (code) several instances of which run simultaneously as *labs* (in Matlab's notation) on the computational nodes (processors) of a parallel machine, processing different data. The parallel job is typically initialized by a sequence similar to that in Fig. 2. This code contains the whole life cycle of the parallel job, and it is the only part of code that runs on a client session. It includes finding parallel computing resources (`findResource`), provided by DCE, creating the parallel job (`createParallelJob`) and setting its properties, specifying the task (`createTask`) and its code (`par_kolokace`), submitting the job (`submit`), waiting for its completion (`waitForState`) and collecting the results (`getAllOutputArguments`).

```
JobManager=findResource('scheduler','type','jobmanager');
ParJob=createParallelJob(JobManager,'Name','kolokace2b');
set(ParJob,'MaximumNumberOfWorkers',nlabs)
set(ParJob,'MinimumNumberOfWorkers',nlabs)
p={'/home/parallel_kolokace','\\pck\home\parallel_kolokace'};
set(ParJob,'PathDependencies',p)
ParTask=createTask(ParJob,@par_kolokace,3,{M,B,G,K,KC,fst});
submit(ParJob)
waitForState(ParJob,'finished')
results=getAllOutputArguments(ParJob);
destroy(ParJob)
```

Figure 2: Life cycle of a parallel job

The code of the (single) task has to contain the functionality of all labs. The differentiation as achieved via the `labindex` function (returns the unique number of the lab), as shown in Fig. 3. Here, if the number of labs is greater than two, then lab 1 acts as the *master* process and the other labs are *slave* processes.

```
function [x0,xc,xs]=par_kolokace(M,B,G,K,KC,fst)
if numlabs==1                % only 1 lab available - sequential code
    [x0,xc,xs]=kolokace(M,B,G,K,KC,fst);
else                         % more than 1 lab available - parallel code
    if labindex==1           % master process
        [x0,xc,xs]=parallel_kolokace(M,B,G,K,KC,fst);
        labSend(1,2:numlabs) % message "Job is finished"
    else                     % slave processes
        parallel_slozky_sily();
                             % all labs have to send results
        x0='x0'; xc='xc'; xs='xs';
    end
end
```

Figure 3: Handling lab numbers

In Fig. 4 the code executed by the slave process is shown. The slaves communicate with the master by means of *message passing* functions – *labSend labReceive*, with the lab numbers

of the recipient or sender as arguments. In a loop, the slaves get a portion of data from the master, process it and return the results back. Unfortunately, the corresponding code of the master process is much more complicated and from space reasons it cannot be included in this paper.

```
function []=parallel_slozky_sily()
data_out=cell(1,1);          % initialisation of output
data_in=labReceive(1);       % receive message from master
while size(data_in,2)>1      % until "Job is finished" arrives do
                             % calculate bearing forces
    [Ploz]=tlakova_fce_sym(data_in{1,1},data_in{1,2});
    [data_out{1,1}]=slozky_sily(Ploz,data_in{1,4});
    labSend(data_out,1)      % send results to master
    data_in=labReceive(1);   % receive message from master
end
```

Figure 4: Fragment of the slave code

3.4 Good Performance

We run the steady-state response code in the Matlab DCT/DCE 3 parallel environment on a Beowulf cluster (8 identical computing nodes with AMD Athlon 1.4 GHz CPU, 1.2 GB RAM each, Fast Ethernet interconnect) and on a symmetric multiprocessor (2 dual-core AMD Opteron 2.4 GHz CPU's, 4 GB shared RAM). Fig. 5 shows nearly ideal (linear) speed-up[4] which we could achieve on both platforms, when employing increasing number of processing units. The performance was nearly independent of the number of bearings involved in the mathematical model of the rotor.

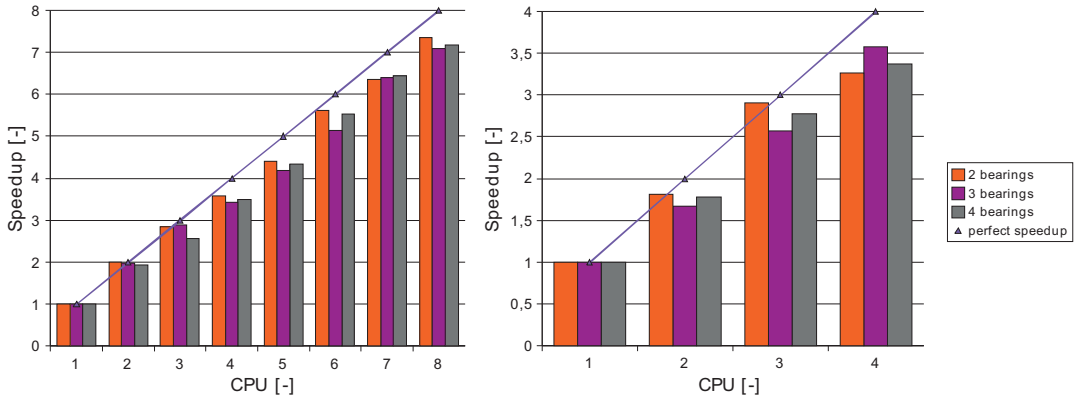

Figure 5: Speed-ups achieved on a cluster (*left*) and symmetric multiprocessor (*right*)

Without going into details, let us note that it is even easier to write a Matlab's "distributed" variant of this embarrassingly parallel algorithm, without explicit message passing (cf. Section 2). However, in our case this code suffered form the overhead of the repeated creating and destroying distributed jobs and was slower, probably because DCE has to test the status of all computational nodes of a parallel computer.

4 Conclusions

Although The Mathworks somewhat "hesitated" with introducing parallelism into Matlab, they finally completed a very good work. The Matlab engineers did their best to design the parallelization API in a Matlab-like, i.e. very user-friendly manner (including peculiar terminology).

[4]Efficiency about 90 %

As far as it is in parallel processing possible, black box solutions are provided for inexperienced users, whereas experts can e.g. make use of message passing almost in the same way as in the native message passing systems (MPI). Thus, Parallel Matlab (DCT/DCE) is an ideal environment for a smooth transition to the world of parallel scientific computing, especially for current Matlab users. In our application, thanks it its parallelization in Matlab we were able to reduce its execution time to a fraction of the sequential version.

References

[1] Matlab product page. http://www.mathworks.com/products/matlab

[2] Distributed/parallel Computing Toolbox product page.
http://www.mathworks.com/products/parallel-computing

[3] Distributed Computing Environment/Server product page.
http://www.mathworks.com/products/distriben

[4] Moler, C.: *Why there isn't a parallel MATLAB.* The MathWorks News & Notes, Spring 1995.
http://www.mathworks.com/company/newsletters/news_notes/pdf/spr95cleve.pdf

[5] Moler, C.: *Parallel MATLAB: Multiple Processors and Multiple Cores.* The MathWorks News & Notes, June 2007.
http://www.mathworks.com/company/newsletters/news_notes/june07/clevescorner.html

[6] Choy, R., Edelman, A.: *Parallel MATLAB: Doing it Right.* Massachusetts Institute of Technology, Cambridge, MA 02139. November 15, 2003.

[7] MPI Formum WWW page. http://www.mpi-forum.org

[8] Musil, T., Jakl, O.: *Parallel algorithm of trigonometric collocation method in nonlinear dynamics of rotors.* Applied and Computational Mechanics, Volume 1, Number 2, ZČU Plzeň, 2007, pp. 555-564.

[9] Krupa, J., Pavelka, A., O. Vyšata, O., Procházka, A.: *Distributed Signal Processing.* In: Proceedings of the conference Technical Computing Prague 2007, Humusoft, Praha, 2007, p. 81.

Ondřej Jakl
Institute of Geonics, Academy of Sciences of the CR
Studentská 1768, 708 00 Ostrava – Poruba
E-mail: jakl@ugn.cas.cz

Tomáš Musil
Department of Mechanics, VŠB–TU
17. listopadu 15, 708 33 Ostrava – Poruba
E-mail: tomas.musil.fs@vsb.cz