CMST 9(1) 137-145 (2003) DOI:10.12921/cmst.2003.09.01.137-145

COMPUTATIONAL METHODS IN SCIENCE AND TECHNOLOGY 9(1-2), 137-145 (2003)

EFFICIENCY OF MATRIX ELEMENTS COMPUTATIONS ON PARALLEL SYSTEMS*

MIECZYSŁAW TORCHAŁA AND JACEK KOMASA

Quantum Chemistry Group, Faculty of Chemistry, Adam Mickiewicz University, Grunwaldzka 6, 60-780 Poznań, Poland

(Rec. 29 January 2004)

Abstract: Experience with adapting sequential programs to a parallel environment is shared with the reader. Our programs are used in quantum-chemical calculations but certain parts of them are of general application and our results can be adapted to other types of problems. Several PC nodes are connected through a fast network and consolidated to a cluster. Our applications make use of the Message Passing Interface environment. Encouraging results concerning speedup and efficiency have been obtained. Experiments leading to a superlinear speedup using the hyperthreading technology are also reported.

1. INTRODUCTION

Enormous progress in computer technology has stimulated continuous adaptation of software to quickly changing computational environment. Often, an old algorithm has to be implemented on new machines appearing in the market. In particular, parallel computers become more and more accessible which calls for a modification of the existing software to make use of the new opportunities.

In quantum chemical calculations, as well as in other areas of computational science, the opportunity of shortening the time one waits for the final result cannot be overestimated. In this paper we describe our effort of adapting algorithms used in quantum chemical calculations to the situation when a set of several PC machines are connected forming a cluster. In other words, we want to share our experience with parallelizing sequential programs using a Message Passing Interface (MPI) environment.

Our paper is addressed to the reader who has got an access to more than one processor and wants to utilize this to speeding up his computations. We show, in an elementary way, how to set up a cluster, i.e. how to make two or more processors to communicate, using the MPI technology. Then we present results for a speedup and efficiency obtained in a simple case of filling up a matrix with the elements which are assumed to be mutually independent. And finally, we share our experience in dealing with a new technology called hyperthreading, which results in a superlinear speedup.

---

* Dedicated to the memory of Professor Jacek Rychlewski

2. MOTIVATION AND QUANTUM-CHEMISTRY ESSENTIALS

In this section we introduce some quantum-chemical notions to show the origin of our interest in the parallelization of the matrix algebra programs. As in most quantum-chemical methods we start with the Schrödinger equation

$$\hat{H}\Psi = E\Psi. \tag{1}$$

Here, Ψ represents a wave function describing a given quantum-mechanical system, E - energy of this system, and $\hat{H}$ is the Hamiltonian operator specifying components of the energy included in the model. We shall focus on variational solutions to the Schrödinger equation. The variational method is based on the theorem saying that for any square-integrable trial function Φ, the so called Rayleigh quotient

$$\varepsilon = \frac{\int \Phi^* \hat{H} \Phi \, d\tau}{\int \Phi^* \Phi \, d\tau} \tag{2}$$

satisfies the following inequality $\varepsilon \geq E$ which means that the energy computed using an approximate wave function gives an upper bound to the exact energy, E (see e.g. [1]). This general principle is employed as a guide in search for possibly accurate approximations to the exact wave function and energy. To make use of the variational principle we expand the trial wave function Φ in the form of a properly symmetrized linear combination of some known basis functions, ϕ_k:

$$\Phi = \sum_{k=1}^{K} c_k \phi_k. \tag{3}$$

Substitution of (3) into (2), subject to the stationary condition on the linear coefficients, $\partial\varepsilon/\partial c_k = 0$, leads to a matrix form of the Schrödinger equation

$$\mathbf{Hc} = \varepsilon \mathbf{Sc} \tag{4}$$

with matrix elements $h_{kl} = \int \phi_k^* \hat{H} \phi_l \, d\tau$ and $s_{kl} = \int \phi_k^* \phi_l \, d\tau$. Eq. (4) has a well known form of the general symmetric eigenvalue problem (GSEP) and can be solved using standard linear algebra methods [2].

As the Hamiltonian operator is Hermitian, the matrices involved are symmetric - it is sufficient to compute only the lower or upper triangle of the matrix - which is an important and time saving feature.

In our applications, every basis function ϕ_k depends on several nonlinear parameters. The total trial wave function is expanded in thousands of such basis functions, hence it depends on a huge number of linear and nonlinear parameters which have to be optimized with respect to the energy. During such an optimization the eigenvalue ε of Eq. (4), has to be evalu-

ated millions of times. This is a quite time-consuming task and it is definitely worth looking for some savings inside the algorithm evaluating the goal function, ε.

Every single evaluation of the energy consists of two computationally distinct parts. In the first part, the matrix elements have to be updated in accordance with the changes in the nonlinear parameters governed by an optimization algorithm. Effort needed to compute a single matrix element depends primarily on the number of electrons, n, a particular quantum system is built of, and, roughly speaking, scales as $n!$. This $n!$-dependence of matrix elements is a real bottleneck of quantum chemical calculations which prevents the most accurate methods to be applied to large atoms and molecules. This part of the algorithm operates mostly on scalar variables and, because of the mutual independence of the matrix elements, it suits very well any parallelization scheme. In this work we concentrate on this aspect of the energy computation.

The second part - the solution of the GSEP is typical linear algebra task and its parallelization is not trivial [3]. However, if the work needed to compute matrix elements exceeds significantly the diagonalization effort then performing the matrix algebra on a single processor is an acceptable solution. The interplay between these two parts was investigated previously in Ref. [4] where we concluded that for quantum systems with two elections the most time consuming is the matrix algebra part but for larger systems, with four and more electrons building the matrices requires much more CPU time and, hence, the programmer's attention. The work described in this paper is the step towards improving the efficiency of our multielectron optimization programs.

3. SETTING UP A CLUSTER

Our cluster was set up by means of MPICH - a freely available implementation of the Message Passing Interface libraries created by Mathematics and Computer Science Division [5, 6]. The "CH" in MPICH stands for "Chameleon", symbol of adaptability to one's environment and thus of portability. It has been installed on a six-node cluster. Each node was equipped with two Pentium III 800 MHz processors with 256 MB RAM on an SMP (symmetric multiprocessor) motherboard. The nodes were connected by means of a 100 MB/s local network. Our algorithm was coded in Fortran 77 under Linux Mandrake 7.2 operating system. Several simple adjustments of the operating system have to be performed to enable full functionality of the collection of the nodes as a single cluster. They are shortly described below.

One node was selected a server (often called master) and the remaining were configured as workstations (slaves). A common disk space was made available using a NFS - network file system, which allows a client node to perform transparent file access over the network. To grant the slaves an access to its disk space the master node requires the following entry in the /etc/exports file

```
/directory workstation! (rw) workstation2(rw)...
```

where /directory is the path name of the served filesystem. On the workstations the file /etc/fstab was added the following line

```
server/directory    /directory    nfs bg,intr,noac 0 0
```

The machines were set up to communicate through the SSH (Secure Shell) client-server system. In the iptables file, we allowed traffic on ports from 0 to 1023 and on 2049 (input and output), and we blocked incoming traffic from other hosts than these in our cluster.

The Fortran code was compiled and linked by calling a shell script included in the MPICH package

```
./mpif77 -o prog prog.f
```

In order to run the program we type

```
./mpirun -np number_of_processes prog
```

An MPI program written in Fortran has to have in the main module a directive include 'mpif.h'. This file, supplied with the MPICH distribution, contains all the definitions and functions needed to compile an MPI program. In coding, we used only six MPI functions shortly described here. The first three of them: MPIJNIT, MPI_COMM_RANK, and MPI_COMM_SIZE initialize the MPI calculation by setting appropriate variables, the next two functions MPI_SEND, MPI_RECEIVE perform the interprocess communication, and MPI_FINALIZE concludes the MPI run of the program. Syntax and other details of using these functions can be found in the MPI documents [7, 8].

Creating an algorithm to be run in parallel we have to realize some issues connected with physical execution of the job on multiple processors. Execution of the mpirun command places a copy of the executable on a number of processors depending on the value of the option, -np. Additionally, the user can decide on which nodes the processes are allowed to run through the option -machinefile file_name, where the file file_name contains a list of the hostnames accompanied by a number of available processors in the following format:

```
hostname1:2
hostname2:2
    ⋮
```

Depending on the algorithm coded, different processes can follow different routes by executing branching commands referring to the process rank - a non-negative integer. We say that the MPI program works within the Single Program Multiple Data (SPMD) paradigm.

3.1. Load balance and programming issues

Having this in mind one can think of load balance issues i.e. a possibly equal distribution of the work assigned to each process. The parallel program is only as fast as its slowest component and a good load balance is the main feature of any efficient parallel program.

Computing an upper triangle of a $K \times K$ symmetric matrix we used the following mapping of the matrix element position (i,j) onto the process number I out of the set of N processes

$$\begin{aligned} k &= i + (j-1) * K - (j-1) * j/2 \\ I &= \mathrm{mod}\,(k, N). \end{aligned} \tag{5}$$

Such a mapping ensures equal distribution of the matrix elements among the processes. The number of matrix elements assigned to a process differs at most by one from that assigned to any other process. To describe this issue quantitatively let us introduce a measure of the load imbalance [4]

$$\sigma = \max_I \left(\left| \frac{\eta_I - \eta}{\eta} \right| \right), \tag{6}$$

where η and η_I are the ideal and actual number of matrix elements assigned to the I-th processor. For instance, when computing the last column of a 1600 x 1600 matrix using the above mapping and 12 processors we obtain $\sigma = 0.5\%$. The model described above works effectively on a cluster built of identical processors. Another interesting issue is how to balance the workload when a job is to be run in an heterogeneous environment. A detailed discussion on this topic can be found in Refs. [4, 9].

In principle, a parallel algorithm can be based on two different paradigms: symmetric and nonsymmetric (master-slave). After some numerical experiments we have chosen the latter model as slightly more effective. We realize, however, that the preferences observed can revers when the type or even size of the problem changes. A general conclusion from our tests was that with the currently available network throughput and for the given problem size (K= 1600) the communication time was not very meaningful and no additional modification of the algorithm was needed. An example of a simple realization of the ideas described above is presented in the form of a schematic code listed below.

```
call MPIJNIT(error)
call MPI_COMM_RANK(MPI_COMM_WORLD, who_am_I, error)
call MPI_COMM_S[ZE(MPI_COMM_WORLD, N, error)
if (who_am_I .eq. 0) then
  do j = 1, K
    do i = j, K
      {DIVIDING TASKS e.g. according to Eq. (5)}
      if (I .eq. 0) then
        {PROCESS 0 COMPUTES MATRIX ELEMENT}
      endif
  . enddo
  enddo
  do source = 1, N-1
```

```
    {RECEIVING MATRIX ELEMENTS using MPI_RECV}
  enddo
else
  do j = 1, K
    do i = j, K
      {SEEKING FOR ELEMENTS TO COMPUTE, Eq. (5)}
      if (I .eq. who_am_I) then
        {COMPUTING AND STORING MATRIX ELEMENT}
      endif
    enddo
  enddo
 {SENDING COMPUTED MATRIX ELEMENTS using MPI_SEND}
endif
call MPI_FINALIZE(error)
```

4. RESULTS AND DISCUSSION

To estimate the gain obtained from the parallelization we measured the time elapsed between two checkpoints: one placed before calling the MPI initialization functions and the other after the finalizing function. The CPU time T_1 measured from within Fortran code when running on a single processor, without calling MPI functions, was then compared to that

Table 1. The dependence of the speedup, S_N, and efficiency, W_N, on the number of processors employed

N	S_N	W_N
1	1.000	1.000
2	1.998	0.999
3	2.994	0.998
4	3.983	0.996
5	4.973	0.995
6	5.961	0.993
7	6.952	0.993
8	7.936	0.992
9	8.919	0.991
10	9.897	0.990
11	10.882	0.989
12	11.850	0.987

obtained for an N-processor job, T_N, measured on the master processor. A commonly used measures of the parallelization gain are the speedup

$$S_N = \frac{T_1}{T_N} \tag{7}$$

and the efficiency

$$W_N = \frac{S_N}{N}. \tag{8}$$

The final numerical experiments were performed for $K = 1600$. Their results are presented in a tabular and graphical form. Table 1 presents the dependence of the speedup and efficiency on the growing number of processors involved in the MPI calculations. Graphically these results are displayed in Fig. 1. On both graphs, the dashed line marks the theoretical

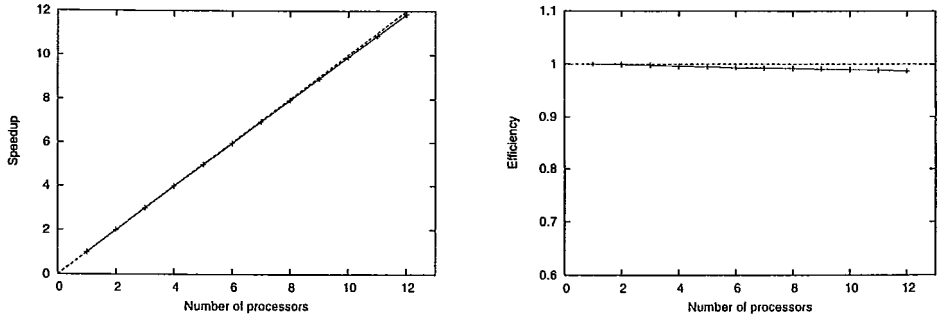

Fig. 1. The dependence of the speedup, S_N, and efficiency W_N, on the number of processors employed

limit: $S_N = N$ and $W_N = 1$. As can be concluded from these pictures and the numerical values given in the table, the computation of the matrix elements can be parallelized very efficiently in a simple way.

5. HYPER-THREADING TECHNOLOGY

In certain circumstances, the speedup can exceed the theoretical limit - we speak then of superlinear speedup. Such a situation can appear when the distribution of the data between the processors makes better use of the cache memory then in the sequential run. This additional gain in speed may cause the efficiency to become higher that 1. With the advent of a new class of processors, which use the so-called Hyper-Threading Technology (HTT) [10], designed to take advantage of such a feature, an additional increase in the speedup became possible. With Hyper-Threading Technology each processor has one set of execution resources (as any regular processor) but its architectural state is duplicated. E.g. one node with two processors on board can be declared to have total of four logical processors: two real and two virtual. Each logical processor is capable of responding independently to system interrupts. Such a configuration results in a more effective use of execution resources.

Obviously, the increase in performance depends highly on the particular application but additional performance gain up to 25% has been reported [10].

Table 2. Scaling of the speedup, S_N, and efficiency, W_N, with the number of physical processors, N, without and with the Hyper-Threading

	Without HTT		With HTT	
N	S_N	W_N	S_N	W_N
1	1.000	1.000	1.000	1.000
4	3.972	0.995	4.402	1.101
6	5.971	0.995	6.603	1.101
8	7.863	0.983	8.790	1.099
10	9.883	0.988	10.944	1.094
12	11.843	0.987	13.050	1.087
14	12.738	0.980	15.261	1.090
16	15.704	0.982	17.386	1.087
18	17.637	0.980	19.482	1.082
20	19.431	0.972	21.630	1.081

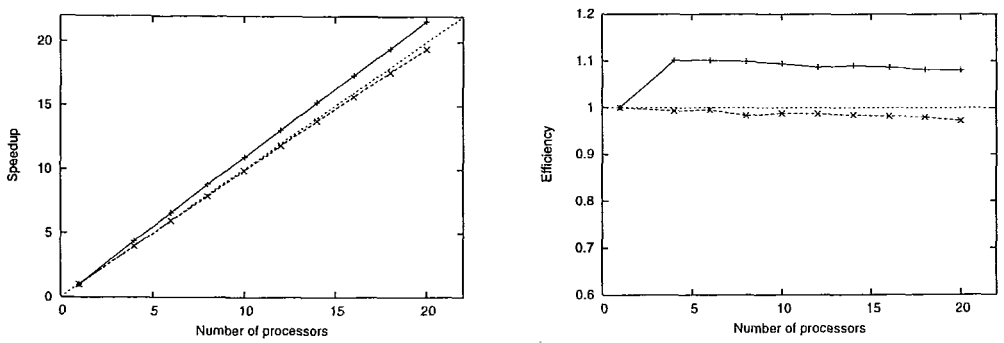

Fig. 2. The scaling of the performance in both no-HTT and HTT modes

Table 2 and Figure 2 present results of our own tests performed on a cluster built of 20 physical processors collected in 10 nodes (Intel Xeon 2.8 GHz x2, 512MB RAM). The MPI program used was the same as previously - no special adaptation to the HTT mode was required. After switching the operating system to the HTT mode the jobs were run as if the number of processes in each node was doubled. E.g. to take advantage of all 20 processors available in the HTT regime we run the program with -np 40 option. The efficiency greater than 1 was observed for all processor configurations as listed in the last column of Table 2. The additional boost obtained in the HTT mode was ca. 11%.

6. CONCLUSIONS

The overall conclusion from our numerical experiments with computing mutually independent matrix elements is fairly optimistic. The modification of the source code connected with the adaptation of the program to the MPI environment is minimal. Scalability observed on the cluster is very encouraging, particularly, when the Hyper-Threading Technology is utilized.

Acknowledgment

This work was supported by the Polish State Committee for Scientific Research grants T09A 17118 and SPB/COST/T-09/DWM572. Support from Poznań Networking and Supercomputing Center is also gratefully acknowledged.

References

[1] I. Mayer, *Simple Theorems, Proofs, and Derivations in Quantum Chemistry,* Kluwer Academic 2003.

[2] W. Cencek, *The Role of Efficient Programming in Theoretical Chemistry and Physics Problems,* in: *Computational Methods in Science and Technology,* (edited by J. Rychlewski, J. Węglarz, K. W. Wojciechowski) Scientific Publishers OWN, Poznań, Vol. 1, 1996, p. 7-18.

[3] J. Komasa, J. Rychlewski, Parallel Computing **26**, 999 (2000).

[4] W. Cencek, J. Komasa, and J. Rychlewski, *High-performance Computing in Molecular Sciences* in: *Handbook on Parallel and Distributed Processing,* eds. J. Błażewicz, K. Ecker, B. Plateau, D. Trystram, Springer 2000, p. 505.

[5] http://www.mcs.anl.gov/.

[6] http://www-unix.mcs.anl.gov/mpi/mpich/.

[7] http://www-unix.mcs.anl.gov/mpi/.

[8] P. S. Pacheco, W. C. Ming, *MPI Users' Guide in Fortran,* http: //www.phyast.pitt.edu/beowulf/Tutorial.html.

[9] W. Cencek, *High-performance Computing on Heterogeneous Systems,* in: *Computational Methods in Science and Technology* (edited by J. Rychlewski, J. Węglarz, K. W. Wojciechowski) Scientific Publishers OWN, Poznań, Vol. 5, 1999, p. 7-19.

[10] http://developer.intel.com/technology/hyperthread/.