Journal of Mathematical Chemistry 11(1992)1–11

A MASSIVELY PARALLEL APPROACH TO THE QUASICLASSICAL REACTIVE SCATTERING

R. BARAGLIA, R. FERRINI, D. LAFORENZA, R. PEREGO
CNUCE, Italian National Research Council, Via S. Maria 36, 56126 Pisa, Italy

A. LAGANÀ
Dip. di Chimica, Università di Perugia, Via Elce di Sotto 8, 06100 Perugia, Italy

and

O. GERVASI
Centro di Calcolo, Università di Perugia, Piazza dell'Università 2, 06100 Perugia, Italy

Abstract

The suitability of massively parallel architectures for carrying out efficient calculations of quasiclassical rate constants for atom–diatom reactive processes has been investigated. Problems related to the parallel structuring of the computational procedure, fixed and scaled speedups, efficiency factors and their dependence upon the size of the problem, and the number of processors are discussed.

1. Introduction

Numerical solutions of scientific problems quite often rely on the use of intensive computing procedures. This means that to obtain these solutions in a realistic time, use of modern parallel features of advanced computers has to be made. To this purpose, both the numerical approach and the computing strategy need to be properly designed to *take advantage of the parallelism.*

The scientific problem considered in this paper is the calculation of rate constants for a family of elementary gas phase reactions relevant to the modeling of complex non-equilibrium systems starting from first principles. In particular, we have investigated bi-molecular atom–diatom reactions. These systems are interesting per se because important for several modern technological applications [1] and as a model for some larger molecule reactions.

The detailed state to state rate constant $k_{vj,v'j'}(T)$ at a given temperature T can be evaluated by integrating over the collision energy (E_{tr}), the related reactive cross section $S_R^{vj,v'j'}(E_{tr})$, where vj ($v'j'$) are the initial (final) vibrotational quantum numbers. In a quasiclassical mechanics approach, $S_R^{vj,v'j'}(E_{tr})$ can be formulated in terms of $P_{vj,v'j'}(E_{tr})$, the atom–diatom reactive probability. Following a Monte Carlo approach [2], $P_{vj,v'j'}(E_{tr})$ can be estimated from a limited number of trajectories N, by calculating the integral

$$P_{vj,v'j'}(E_{tr}) = N^{-1} \int_0^1 d\xi_1 \int_0^1 d\xi_2 \int_0^1 d\xi_3 \int_0^1 d\xi_4 \int_0^1 d\xi_5 f_{vj,v'j'}(\xi_1,\xi_2,\xi_3,\xi_4,\xi_5), \quad (1.1)$$

where the variables ξ are related to initial position and momenta of the collision partners. The integrand $f_{vj,v'j'}$ is a Boolean function that is unity when the integer value of $j'_c + 1/2$ and $v'_c + 1/2$ obtained from the trajectory calculation are, respectively, equal to j' and v' (j'_c and v'_c are the classical equivalent of the rotational and vibrational quantum numbers evaluated by integrating the classical equations of motion).

Each trajectory being a fully independent calculation, the integration of a batch of trajectories can be carried out in parallel by a pool of asynchronous processes cooperating according to a task farm model. A task farm model is based on a master process that manages the computation and dispatches the work to a set of slave processes. In a hypercube architecture, this cooperation model can be implemented by mapping the master process on the *host node* and the slave processes on the *distributed nodes*.

The paper is organized as follows: in section 2, the used hardware (NCUBE machines) and system software are given as well as the structuring of the program to be run on the parallel architecture are discussed. In section 3, speedups and performances of the restructured code are analysed.

2. Restructuring the code for a hypercube

A typical trajectory program consists of a preliminary section in which the random sequence initiator as well as physical constants are read in to generate some quantities of common use. The largest section of the program is embodied into a loop running over the trajectory index. Inside the loop, the necessary set of pseudo-random numbers are generated to obtain the starting conditions for the considered trajectory. After the validation of initial conditions, the recursive process of integrating in time positions and momenta starts. At the trajectory ending point, a determination of its final properties is performed to derive the products' properties and update the statistical analysis.

As already mentioned, to structure the program for the parallel environment [3–6] we have adopted a task farm model of cooperation [7] consisting of a master process running on the host and a set of identical worker processes loaded on the nodes. The master process (see scheme 1) takes care of all I/O operations, dispatches the trajectory calculations to the nodes, and collects final results. To optimize the load balancing, a self-scheduling method assigning only one trajectory at a time to each worker node has been adopted. An attempt has also been made to determine the number of trajectories assigned at a time to a worker process (granularity) which maximizes the load balancing while minimizing the communication overhead.

```
Host program

Read input data from file and do initializations
Allocate and open a hypercube of N nodes
Load program on nodes
Broadcast initial data to all nodes

Ntraject = 1
seed = initiator

For Nnode = 0 to N - 1 do
      random(seed, new_seed, rand)
      send (Ntraject, seed) to Nnode
      Ntraject = Ntraject + 1
      seed = new_seed
next Nnode

While Ntraject <= Max_traject do
      random(seed, new_seed, rand)
      Receive msg of type msgdone from node M
      send(Ntraject, seed + 1) to node M
      Ntraject = Ntraject + 1
      seed = new_seed
end_while

When a node ends its last trajectory send it a msg of type msgend
Wait final result from node 0
Close hypercube
Write results on file
```

Scheme 1. Structure of the host program.

A parallel structuring of a computer code based on this Monte Carlo approach may not be trivial. Usually, the pseudo-random sequence is generated by an algorithm which transforms a given integer number (seed) into a real number in the interval 0–1 and a new seed. In a scalar code, the integration of different trajectories is sequential and therefore the sequence of calls for the generation of the needed set of pseudo-random numbers is strictly ordered. Instead, in a parallel environment, the generation of the pseudo-random sequence is affected by non-determinism. As a result, the sequence (and therefore the set of initial conditions) generated in one run may not coincide with that of other runs. In a parallel run, the generated sequence may also depend on the value of other working conditions. As an example, a variation of the integration stepsize by changing the duration of the trajectories may cause a different correspondence between the number of the sequence and initial trajectory conditions.

For this reason, to enforce partial and global reproducibility of the calculation, the random number calls of one trajectory were chained to those of the previous one, by generating sequentially at host level only one integer number per trajectory.

Such a number acts as a seed for the generation of the needed subset of numbers inside each worker node. This allows a unique correspondence between the trajectory sequential position and the generated set of pseudo-random numbers. In practice, this was obtained by prescribing that when the host processor starts a new cycle of the trajectory loop, a new random number and seed are generated starting from the seed of the previous trajectory. Then the host waits for the next available worker processor, to which it sends the seed necessary for starting the integration of a new trajectory. Starting from the received seed (see scheme 2), each node locally generates

---

Node program

Receive initial data from host and do initializations
End = false
while End = false do
  Receive msg of any type from host
  if msgtype = msgend then End = true
  else
    Use random seed received to generate other random quantities
    Integrate the trajectory and update statistical indicators
    Send msg of type msgdone to host
  end_while

Cube collapsing algorithm

Scheme 2. Structure of the node program.

all pseudo-random numbers needed to define the initial conditions for the assigned trajectory. By adopting this method, the behaviour of the sampled events is totally deterministic.

Another feature of the parallel implementation is that associated to the collection of results needed to perform the final statistical analysis. During the calculation, trajectory results are stored locally by instructing each node to update the statistical indicators after integrating every trajectory. Once its last trajectory is integrated, each node begins a "*dimensional collapsing*" algorithm [8] to collect the distributed results on the host. According to this algorithm, performed once for each dimension of the n-dimensional hypercube, results are sent by the higher numbered to the next lower numbered neighbour node in that dimension. At iteration m of the algorithm (the iteration index goes from n to 0), the m-dimensional cube is divided into two $(m-1)$-dimensional sub-cubes. All the nodes of the sub-cube for which the bit of position m in the node address is equal to 1 send in parallel their partial results to the nodes of the other sub-cube connected in the m dimension (for which the same bit in the node address is equal to 0). The receiving nodes combine (add, in this case) the received results to local ones and then repeat the same algorithm for the next cube dimension. This continues until global results are contained in node 0

(node 0 forms a 0-dimensional hypercube) and from there are sent to the host. The flow of the results forms a binary tree pattern with logarithmic depth. The same algorithm, in the opposite direction, is used by the operating system to load nodes and to broadcast messages. When the number of nodes is large, such an algorithm with logarithmic-cost turns out to be very efficient.

The parallel trajectory program has been run on three different NCUBE machines [3–6]: NCUBE/4, NCUBE/10 and NCUBE 2 6401.

The NCUBE/4 with 16 processing nodes was used during the development of the program. The NCUBE/10 with 512 processing nodes was used for production runs.

The NCUBE 2 6401 was used for advanced speedup measurements. It belongs to the second generation of NCUBE machines released in June 1989. Many improvements, both in the hardware and in the system software, have been introduced in these machines. The microprocessor (500000 transistors) includes a 64-bit CPU and floating-point unit, 14 bi-directional communication channels, 4 instruction pipelines, a data cache of 8 operands, and an instruction cache of 128 bytes. The message routing functions are hardware implemented. The peak transfer rate is 2.22 Mbytes/s per channel in each direction. The routing unit allows a direct pass-through of messages without interrupting intermediate nodes. Running at a clock rate of 20 MHz, the processor is rated at 7.5 MIPS and 3.3 MFLOPS single precision or 2.4 MFLOPS double precision. Per-node memory ranges from 1 to 64 Mbytes.

3. Performance analysis and results

The main elements which affect the performance of an algorithm on a highly parallel architecture without shared memory are the *load unbalancing* and the *communication overhead*. The different play of these elements leads to a significant variation of the measured speedup.

3.1. SPEEDUPS

The speedup (S) is defined as the ratio between the time required to execute a given program on a single processor and on a set of concurrent processors. If T_s is the execution time of the sequential program on a single processor and T_p is the elapsed time of the parallel program on P processors, the speedup is given by

$$S = \frac{T_s}{T_p}. \tag{3.1}$$

For a real problem like the one reported here, it may be difficult to measure the speedup when using the above formulation. The first difficulty concerns the impossibility of running the sequential code on a single node due to memory requirements. In fact, data for a significant run may not fit into the small local memory of one node. Moreover, also without memory constraints, a time consuming

application like the integration of a batch of four thousand trajectories (as is usually needed for a sufficiently accurate rate constant evaluation) would take more than four days to run on a single NCUBE node, resulting in an impractical and senseless effort. For this reason, the speedup definition given in (3.1) has been used only for reduced size problems (e.g. 256 trajectories).

Table 1

Elapsed time (s), fixed speedup and efficiency for runs of 256 trajectories on an NCUBE/4

	$P=1$	$P=2$	$P=4$	$P=8$	$P=16$
T_p	28997	14563	7320	3718	1902
S	1	1.99	3.96	7.80	15.24
E	1	0.995	0.990	0.974	0.952

Table 1 shows the variation of the parallel execution time T_p, the fixed speedup S, and the efficiency factor E as a function of the number of used nodes P for NCUBE/4. The efficiency E is defined as the ratio between the speedup S and the number of nodes P. The number of computed trajectories (256) and the initial conditions are the same for all runs. It can easily be seen from the values reported in the table that also with a reduced size problem, our reactive scattering program exploits efficiently the parallelism. An efficiency factor of 0.952 when using sixteen nodes is certainly a promising result.

For larger problems and highly parallel environments, it is more convenient to measure the scaled speedup as proposed in refs. [8,9]. The scaled speedup (SS) is defined as

$$\mathrm{SS} = \frac{s + pP}{s + p}, \tag{3.2}$$

where s is the amount of time spent in the scalar part of the program (program loading, I/O operations, data initializations, serial bottlenecks), and p is the amount of time spent in parallel on P processors. In eq. (3.2), the factor $s + pP$ gives an estimate of the time that the parallel program would take on a single processor. Scaled speedup measures, therefore, how much the execution time of a problem of size kn on kP processors approaches that of a problem of size n on P processors.

Table 2 shows the execution times T_p, the scaled speedup SS, and the efficiency factor E as a function of the hypercube dimension for the NCUBE/10. The average number of trajectories computed by every node (16) and the initial conditions are the same for all runs. From the values reported in table 2, it can be seen that when the size of the problem is large with respect to the number of nodes, the efficiency approaches its physical limit. It has to be emphasized here that, for this kind of applications, the degree of parallelism that can be usefully exploited is bound only

Table 2

Elapsed time (s), scaled speedup and efficiency running an average of 16 trajectories per node on an NCUBE/10.

	$P=1$	$P=2$	$P=4$	$P=8$	$P=16$	$P=32$	$P=64$	$P=128$	$P=256$	$P=512$
T_p	1577	1479	1590	1574	1572	1591	1578	1580	1580	1657
SS	1	1.998	3.994	7.986	15.95	31.96	63.88	127.6	255.1	508.4
E	1	0.999	0.998	0.998	0.996	0.998	0.998	0.998	0.996	0.993

to the size of the problem. If the number of trajectories to evaluate is doubled, the number of processors can also be doubled with little loss in efficiency. This means that when increasing the size of the problem, the parallel part p of the program scales much more than the scalar part s. The only bottleneck that could downgrade the scalability of the reactive scattering parallel program is the trajectory dispatching carried out by the single master process. The effect of this centralization is already evident from the value referred to the evaluation of 8192 trajectories on 512 nodes, shown in table 2. To optimize this program for a computer with higher parallelism (thousands of processors), the bottleneck could easily be removed by distributing the dispatching on more processors running in parallel, each one controlling a subset of the slave nodes.

The integration of a batch of four thousand trajectories has been performed also on an NCUBE 2 6401 with sixteen processors. The execution time was about 7.5 times smaller than on the NCUBE/10 using the same number of nodes.

3.2. LOAD UNBALANCING AND COMMUNICATION OVERHEAD

Figure 1 shows a schematic diagram of the program execution times for both the host and the worker nodes. As defined above, the parallel execution time T_p includes a sequential part s ($s' + s''$, where s' is the time needed to read the input data from file, load the program on the nodes and broadcast these data, while s'' is the time needed to perform the statistical analysis and to write results on file) and the parallel time p. The time p is the time elapsed between the sending of the first P trajectories to the P nodes and the last receiving of the results. In the figure, $t_1, t_2, \ldots, t_j, \ldots, t_P$ are the running times relative to $1, 2, \ldots, j, \ldots, P$ individual worker nodes (including communication time), and $t_{w1}, t_{w2}, \ldots, t_{wP}$ are the corresponding node waiting times.

T_p is affected by communication overhead and load unbalancing. In our application, messages are exchanged between the slave processors only during the results collection and therefore the time spent for communication is small. On the contrary, a careful analysis of the execution times shows that running times of different nodes may be significantly unequal because of the difference in number

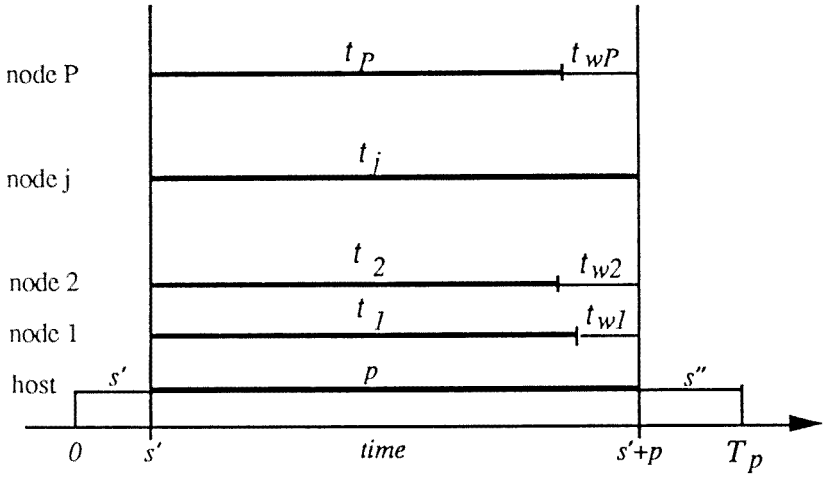

Fig. 1. Schematic diagram of execution times for host and node processors.

and length of the trajectories dispatched on a given node. As an example on an NCUBE/10 node, the average elapsed time per trajectory is about 95 s, while the actual time needed for the integration of a given trajectory may be significantly larger or shorter. Therefore, it may happen that one node begins its execution on the last trajectory in the sequence when all other nodes are close to ending their work. This results in a significant unbalance of the workload.

To quantify the efficiency of the use of parallelism, the quantities T_I and T_W and their relationship with T_p have to be worked out (T_I and T_W are the sum of node individual working and waiting times t_i and t_{wi}, respectively).

If t_j is the time spent by the lowest node (say node j) to evaluate the last ending trajectory, we have

$$p = t_j. \tag{3.3}$$

All other nodes i, with $i \neq j$, accomplish their work before time $s' + p$ and then wait until the execution on node j comes to an end. Then for all nodes we have (see fig. 1)

$$p = t_i + t_{wi}. \tag{3.4}$$

Therefore, the time T_p of a parallel execution can be written as

$$T_p = s + p = s + \frac{1}{P}\sum_{i=1}^{P}(t_i + t_{wi}) = s + \frac{1}{P}\sum_{i=1}^{P} t_i + \frac{1}{P}\sum_{i=1}^{P} t_{wi} = s + T_I/P + T_W/P, \tag{3.5}$$

where the ratio T_I/P represents the time usefully spent by the nodes for processing and communicating, while the ratio T_W/P indicates the time wasted because of load unbalancing. The calculation of both quantities T_I and T_W needs only the evaluation of t_i if use is made of eq. (3.2).

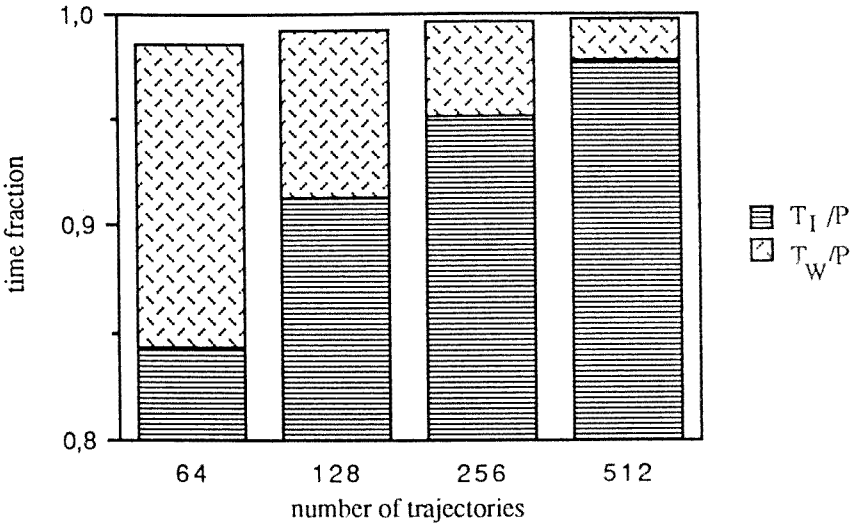

Fig. 2. Fraction of T_p spent as T_I/P and T_W/P for runs of different batches of trajectories on 16 nodes.

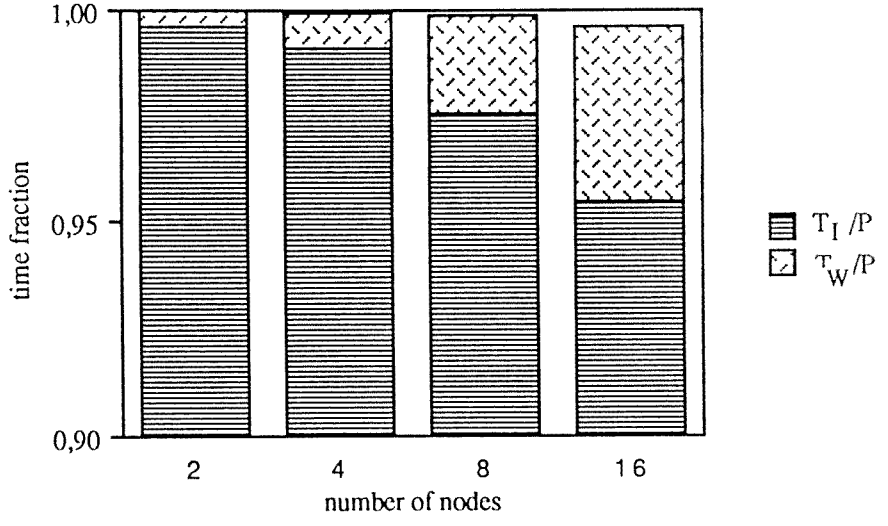

Fig. 3. Fraction of T_p spent as T_I/P and T_W/P for runs of a batch of 256 trajectories on different numbers of nodes.

Figure 2 shows the fraction of time T_p spent as T_I/P and T_W/P for a fixed number of processors (16) when varying the size of the problem. It can easily be seen from the figure that when the number of trajectories is not much larger than the number of processors, T_W/P is a considerable part of T_p. This means that in this case the load unbalance is large.

Figure 3 shows the fraction of time p spent as T_I/P and T_W/P for a fixed size of the problem (256 trajectories) when varying the number of nodes. As can easily

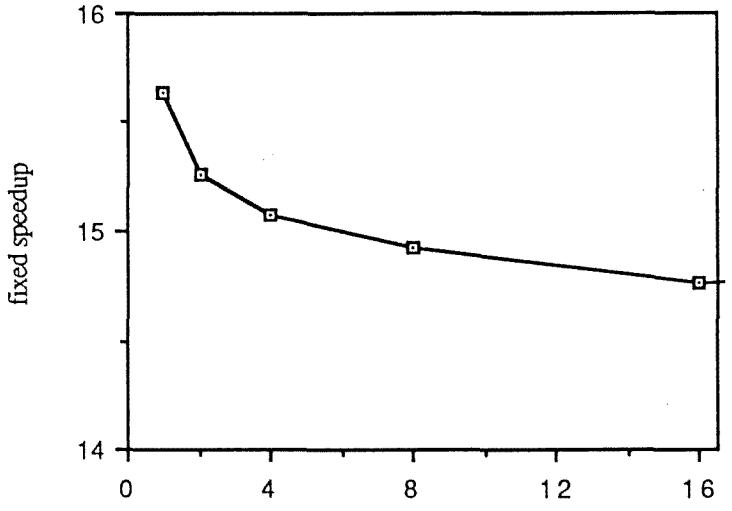

Fig. 4. Value of the fixed speedup for a batch of 512 trajectories evaluated on 16 processors as a function of the scheduling granularity.

be seen from fig. 3, an increase of the number of nodes increases the load unbalance. As a result (and as apparent from tables 1 and 2), this worsens the efficiency of E.

Figure 4 shows the dependence of the speedup from the number of trajectories that the master process sends at a time to each node. Values of the figure are referred to the evaluation of 512 trajectories on 16 nodes. As can easily be seen, the speedup is a sensitive function of the scheduling granularity. In fact, the speedup decreases about 5.5% when the number of trajectories sent at a time increases from 1 to 16. The effect of increasing the number of trajectories dispatched at a time on each node is a decrease of the number of messages exchanged between the host and the nodes; this lower communication overhead, however, does not adequately account for the increase of the load unbalance.

The problem of balancing the workload among the nodes while keeping the communication overhead low is central to achieve high efficiency. As often happens, the best strategy is a mixed one: for large problems, the largest speedup has been obtained by initially sending to each processor more (say 32) trajectories at a time and then, when the largest part of trajectories has been computed, by reducing the scheduling granularity to one. In this way, the number of exchanged messages is significantly reduced without an increase of the load unbalance.

6. Conclusions

We have developed a quasiclassical reactive scattering program for a massively parallel machine. Because each trajectory is a completely independent CPU bound

process, this type of calculation has been found to fit perfectly to a massively parallel architecture. The program has been properly designed for taking advantage of a hypercube computing architecture. Input data have been reorganized to optimize their distribution. The pseudo-random number sequence generation has been structured to guarantee its uniqueness while enhancing the parallelism. The statistical analysis has been broken into partial ones performed on each node during the parallel computation of the trajectories to reduce communications. Communication overhead has also been minimized by collecting final results using a dimensional collapsing algorithm.

As a result, very large fixed and scaled speedups have been measured. Trends of speedups with the variation of the problem size, number of processors and scheduling granularity have been given. A quantification of the impact of waiting times on the efficiency of the program has been attempted.We have found that the algorithm used is scalable and seems to be adaptable, with minor modifications (e.g. by structuring in parallel the dispatching algorithm), to even larger parallel architectures. A comparison with execution times obtained on some popular vector/parallel supercomputers shows that this type of problems can be dealt with more efficiently on highly parallel hypercube machines. From runs performed, further significant execution time reductions are expected to be obtained when using the NCUBEs of the new generation.

In conclusion, our work demonstrates that massive parallelism can be considered not only an exciting field for academic research, but also a practical tool for working out numerical solutions to applied research problems.

Acknowledgements

This work has been supported by the "Progetto Finalizzato: Sistemi Informativi e Calcolo Parallelo" of the Italian National Research Council (CNR). Support and suggestions from the staff of DELPHI and NCUBE Corporation are acknowledged. In particular, we thank Sean Dolan of the NCUBE Corporation for his help in optimizing the code.

References

[1] M. Capitelli and J.N. Barsdley (eds.), *Non-Equilibrium Processes in Partially Ionized Gases* (Plenum Press, New York, 1990).
[2] D.L. Bunker, Meth. Comput. Phys. 10(1971)287.
[3] J.P. Hayes, T.N. Mudge and Q.F. Stout, in: *Proc. 1986 Int. Conf. on Parallel Processing* (IEEE Computer Soc. Press, Washington, 1986), pp. 653–660.
[4] D. Jurasek, W. Richardson and D. Wilde, VLSI System Design (June 1986) 26.
[5] *Users Handbook*, Version P2.1, NCUBE Corporation (October 1987).
[6] *NCUBE 2 6400 Series Supercomputer: Technical Overview*, NCUBE Corporation (1989).
[7] O.A. MacBryan, Parallel Comput. 7(1988)477.
[8] J.L. Gustafson, G.R. Montry and R.E. Benner, SIAM J. Sci. Stat. Comput. 9(1988)609.
[9] J.L. Gustafson, CACM 31(1988)532.
[10] C.P. Kruskal, in: *Control Flow and Data Flow: Concepts of Distributed Programming*, ed. M. Broy, NATO ASI Series (1985), pp. 331–344.
[11] G.M. Amdhal, in: *AFIPS Conf. Proc.*, Vol. 30 (AFIPS Press, Reston, VA, 1967), pp. 483–485.