Theor Chim Acta (1991) 79: 337–347

Theoretica Chimica Acta

A parallel version of ARGOS: A distributed memory model for shared memory UNIX computers*

Robert J. Harrison and Rick A. Kendall**
Theoretical Chemistry Group, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439, USA

Received July 20, 1990/Accepted September 6, 1990

Summary. A distributed memory programming model was used in a fully parallel implementation of the *ab initio* integral evaluation program ARGOS (R. Pitzer (1973) *J. Chem. Phys.* 58:3111), on shared memory UNIX computers. The method used is applicable to many similar problems, including derivative integral evaluation. Only a few lines of the existing sequential FORTRAN source required modification. Initial timings on several multi-processor computers are presented. A simplified version of the programming tool used is also presented, and general consideration is given to the parallel implementation of quantum chemistry algorithms.

Key words: Parallel algorithms – Shared memory computers – *Ab initio* algorithms

Introduction

The general application of *ab initio* chemistry methods is still limited by the vast amount of computer resources required to perform electronic structure calculations of any quality on even small molecular systems. For this reason computational chemists have been tracking developments in computer technology, and developing algorithms and programming models appropriately. Notable examples would be the development of matrix based algorithms for vector computers (e.g. the CRAY-1 [1]), the use of local attached array processors (e.g. the FPS-164 [2]), the the use of large memory algorithms [3]. More recently there has been active interest in the exploitation of parallel computers [4–7]. It is noted that nearly all current super- and mini-super-computers use multiple processors

* Work performed at Argonne National Laboratory under the auspices of the Division of Chemical Sciences, Office of Basic Energy Sciences, U.S. Department of Energy under contract W-31-109-Eng-38. Pacific Northwest Laboratory is operated for the U.S. Department of Energy by Battelle Memorial Institute under contract DE-AC06-76RLO 1830

** *Current address:* Mail Stop K2-18, Molecular Science Research Center, Battelle Pacific Northwest Laboratories, Richland WA 99352

to achieve their peak performance. Increased parallelism seems to be the only path to a new generation of cost-effective, high-performance computers, the debate really focusing around machine architecture and software development.

Shared memory parallel machines[1] are generally assumed to be easier to use and program than distributed memory machines[2]. This, in part, is because the environment is familiar, most programs can run unmodified (albeit inefficiently) and compilers seemingly automate much of the work required to parallelize a subroutine. In addition, optimized scientific libraries can provide an easy route to high performance, and there is usually no explicit involvement of the programmer in inter-process communication. However, in typical quantum chemistry applications, the task granularity within single subroutines is not sufficient to permit efficient parallelization, and one must retreat further up the subroutine calling tree to find the required level of parallelism. Compilers are not yet capable of doing meaningful analysis at this level of program structure and the programmer is left having to do significant amounts of re-writing.

Some algorithms do admit a straightforward and efficient implementation on distributed memory machines. Examples would be many Monte-Carlo models [4, 8], classical trajectories [4], and *ab initio* integral evaluation [6, 7]. These algorithms have in common that the work of each process can be driven by local data, the required infrequent inter-process communication not being fundamental to the work performed by each process. Programmers and vendors of shared memory machines often seem to have ignored this experience.

ARGOS (ARGonne and Ohio State [9]) is a general program to evaluate the one and two electron integrals required in *ab initio* electronic structure calculations, allowing for symmetry adaptation of generally contracted basis sets. Whilst it may not be the most efficient such program, it is certainly one of the most general and widely used. An efficient, portable, parallel implementation would be very useful. We emphasize the need for portability as this program is used by many groups on a wide variety of multi-processor computers, the most notable of which would be Cray. Below we describe a simple, efficient parallelization of ARGOS for all shared memory UNIX based computers, using a distributed memory programming model. We present simplified versions of the tools used; the actual versions used are available from the authors upon request.

Parallel implementation

The structure of integral evaluation programs pre-disposes them to a coarse-grain, data-driven parallel implementation. Typically, after some setup, a nest of four loops (over the shells of basis functions, or some equivalent) is entered with the actual integrals being evaluated within a complex subroutine calling tree. ARGOS, in common with nearly all such programs, passes arguments to and between these low level routines through both common blocks and formal parameters. The low level routines do not have sufficient granularity to parallelize well. Any attempt to seek a higher level of parallelism in a shared memory

---

ared memory parallel machines have a memory which is directly addressable by all the essors

stributed memory parallel machines do not have any globally addressable memory, and ssors must share data by message passing, explicit or otherwise

model would require manual determination of data dependencies created by the common block usage, and substantial re-coding. Great reliance would also be put upon the ability of optimizing compilers to generate correct re-entrant code, which seems to be a problem on some machines. On a distributed memory machine it suffices to allow different processors to handle successive iterations of the nest of four loops; all data dependencies being eliminated since no data is shared. There have been several implementations of two electron integral evaluation programs on distributed memory machines (for example [6, 7]), all using this scheme, and all remarking on how straightforward it is. Here we develop the *minimum* tools necessary to provide a *portable* distributed memory implementation on shared memory computers.

There are many 'portable' memory programming environments that are being or have been developed, supporting hardware platforms from hypercubes and networks of workstations, to shared memory multi-processors. An incomplete list might include PARMACS [10], Cosmic Environment [11], LINDA [12], SR [13], and STRAND [14]. Of these only STRAND and LINDA are available as commercially supported products, and then only on a relatively small number of platforms (mostly workstations). Since the requirements of tools for computer science research into parallel computing and for physical science production computing are quite different (we require robustness, high performance, portability, full support for FORTRAN I/O), most of these tools are not yet suitable for our use. We also have neither the expertise nor the desire to port large amounts of O/S specific code, so for the simple application under discussion here we shall develop our own small set of tools with a minimum of functionality. However, for more complex applications, packages such as STRAND may have a role to play by providing a level of abstraction from the hardware that is not possible with vendor provided tools. Using PARMACS [10] we already have model Hartree–Fock programs which are portable between workstation networks and shared memory machines. One could anticipate sophisticated configuration interaction programs portable between hypercubes and Crays, if such programs were deemed valuable.

All that is needed for a distributed memory implementation of ARGOS (or any similar application) is some means of: (1) creating a small number of identical processes, (2) initializing the data for each process, (3) dividing (deterministically or dynamically) the work between the processes, (4) assemble the final list of integrals, and (5) killing the superfluous processes. Fortunately UNIX makes this straightforward, and by using the C UNIX system call interface we achieve a degree of portability that would not be so easy in FORTRAN. This functionality is implemented in the tools presented in program listing (1), whose use is exemplified by the example in listing (2). To handle steps (1) and (2) we use the UNIX system call **fork()**, which makes a copy of the calling process that is identical to its 'parent' except for process identification and some file and system parameters. To the parent **fork()** returns the process number of the child, while to the child **fork()** returns zero. Step (5) may be implemented by having the parent explicitly wait for its children to finish their allotted work. Step (3) is achieved by assigning each process a logical process number (0–nproc-1, nproc being the number of processes). Iterations of subsequent loops may be assigned deterministically on the basis of this value, possibly by the mechanism described in reference [7] and which is also used in program listing 2. We discuss dynamic load balancing in the next section. Assuming that each process writes its own distinct file of integrals, the final list of integrals is

most easily assembled (step (4)) by simple file concatenation. It would also be possible for the processes to communicate with another process whose job is just to write the file, or programs reading the integrals could simply read the multiple files. The first solution places heavy demand on inter-process communication, which is not otherwise needed, and the second solution requires modification of many independent programs.

The forking of processes is at the heart of UNIX and on many virtual memory machines is very efficient, as new physical copies of pages in the parent's memory space are not made until they are written to. Real memory machines, such as Cray, have to do much more explicit copying of memory and so are slower at starting up processes. However, at the very worst the time to fork a sizable process is measured in tenths of a second, and is negligible compared to the work each process is doing here.

A call to the subroutine **parallel** creates the requested number of essentially identical processes returning to each its logical process value. A call to **serial** causes the parent to wait for its children to die, and causes a child to exit normally. The net effect is that all code between calls to **parallel** and **serial** is executed by multiple processes with independent data. Care must be taken with files; the best course is for each process to open and close files with unique names. The structure of the integral code is illustrated in the test example (listing 2), which merely writes a list of numbers to a file, and then reads and sums them. Multiple processes write multiple files, which are then concatenated before the addition. The FORTRAN program first reads in the requested number of processes. This step corresponds to the data initialization of the integral program, which is much more complex and involves extensive use of COMMON blocks, etc. The program then calls **parallel** to create the new processes, which open unique files and execute distinct iterations of the DO loop. Note that nests of DO loops are readily parallelized by the modulo mechanism used [7]. The call to **serial** explicitly ends the parallel section and subsequently the master process concatenates the files and computes the sum.

Only the two-electron integral evaluation has been parallelized within ARGOS, as the one-electron integrals take just a few seconds. The only code modifications required were in the two-electron driver routine: (1) outside the nest of loops **parallel** is called and unique integral files are opened, (2) within the body of the loops work is shared in the same manner as in the example, and (3) just after the loops, **serial** is called and the integral files are concatenated into one. About a dozen lines of the original driver routine were modified or added and new routines were written to handle the concatenation of the integral and log files. Including extra statements for timing purposes and comments, only 500 lines of source were modified or written in total. We note that our versions of **parallel** and **serial** perform much more error checking than those in listing 1, and that the distribution of work has actually been hidden inside a subroutine call (which just returns the next value of a counter given its previous value, the total number of processes and a logical process number). This increased modularity cleans up the code and makes load-balancing schemes straightforward to implement.

Results and discussion

This program has been run on all multi-processor Cray models, an Alliant FX/8 and a four processor Ardent Titan. Table 1 presents timings for water in a 41

Table 1. Parallel ARGOS timings for C_{2v} water with a 41 function segmented basis set. All times reported are in seconds. The parent wall time is the total wall clock execution time for the job. The parent cpu time is the total cpu time used by the parent process. The two-electron times report the minimum, maximum and average cpu times spent evaluating two-electron integrals by the processes. Values in parentheses are ratios to times for a single process

Machine	No. of	Parent process		Two-electron cpu		
	Processes	cpu	wall	min.	max.	ave.
Ardent Titan	1	91.8 (1.0)	94.0 (1.0)	90.1	90.1	90.1
	4	27.2 (3.4)	37.0 (2.5)	24.8	25.8	25.5
Alliant FX/8	1	87.6 (1.0)	90.0 (1.0)	85.7	85.7	85.7
	4	24.17 ()	30.0 (3.0)	21.4	24.2	22.7
	8	14.0 ()	21.0 (4.3)	10.7	13.0	12.2
CRAY-XMP	1	6.4 (1.0)	6.6 (1.0)	6.26	6.26	6.26
	4	1.6 (4.0)	3.1 (2.1)	1.47	1.68	1.57
CRAY-YMP	1	5.36 (1.0)	5.38 (1.0)	5.25	5.25	5.25
	4	1.34 (4.0)	2.25 (2.4)	1.24	1.41	1.32
	8	0.67 (8.0)	1.58 (3.4)	0.56	0.71	0.66

function segmented TZP basis set [16], using C_{2v} symmetry. Table 2 contains timings for D_{2h} ethylene in a 62 function 6-311G** basis [17]. Table 3 contains timings for $CaFSi_4H_9$ in a 101 function generally contracted DZ basis set [18], with no symmetry. All times are in seconds. Calculations on the Alliant and

Table 2. Parallel ARGOS timings for D_{2h} ethylene with a 62 function 6-311G** basis. All times reported are in seconds. The parent wall time is the total wall clock execution time for the job. The parent cpu time is the total cpu time used by the parent process. The two-electron times report the minimum, maximum and average cpu times spent evaluating two-electron integrals by the processes. Values in parentheses are ratios to times for a single process

Machine	No. of	Parent process		Two-electron cpu		
	Processes	cpu	wall	min.	max.	ave.
Ardent Titan	1	392.6 (1.0)	396.0 (1.0)	389.3	389.3	389.3
	4	108.4 (3.6)	138.0 (2.9)	91.3	111.9	101.6
Alliant FX/8	1	332.4 (1.0)	339.0 (1.0)	327.8	327.8	327.8
	4	91.2 (3.6)	104.0 (3.3)	80.9	91.1	84.7
	8	43.0 (7.7)	80.0 (4.2)	31.1	65.2	45.5
CRAY-XMP	1	26.4 (1.0)	27.5 (1.0)	26.1	26.1	26.1
	4	6.8 (3.9)	21.4 (1.3)	6.1	7.0	6.52
CRAY-YMP	1	22.6 (1.0)	22.8 (1.0)	22.3	22.3	22.3
	4	5.9 (3.8)	6.6 (3.5)	5.2	6.0	5.6
	8	2.8 (8.0)	4.4 (5.2)	2.0	3.4	2.8

Table 3. Parallel ARGOS timings for $CaFSi_4H_9$ with a 101 function generally contracted DZ basis set. All times reported are in seconds. The parent wall time is the total wall clock execution time for the job. The parent cpu time is the total cpu time used by the parent process. The two-electron times report the minimum, maximum and average cpu times spent evaluating two-electron integrals by the processes. Values in parentheses are ratios to times for a single process

Machine	No. of	Parent process		Two-electron cpu		
	Processes	cpu	wall	min.	max.	ave.
Ardent Titan	1	21851.4 (1.0)	22616 (1.0)	21817.2	21817.2	21817.2
	4	5813.6 (3.8)	6919.0 (3.3)	5445.2	5839.9	5635.8
Alliant FX/8	1	18594.8 (1.0)	18671.0 (1.0)	18560.0	18560.0	1856.0
	4	4969.2 (3.7)	5194.0 (3.6)	4683.5	4999.2	4825.7
	8	2715.6 (6.9)	2927.0 (6.4)	2488.1	2692.3	2561.9
CRAY-XMP	1	–	–	–	–	–
	4	358.2	–	344.5	364.0	351.8
CRAY-YMP	1	1209.6 (1.0)	1950.5 (1.0)	1203.5	1203.5	1203.5
	4	306.8 (3.9)	859.2 (2.3)	295.6	839.5	301.6
	8	155.9 (7.8)	1253.7 (1.6)	148.0	159.1	151.1

Ardent were run on dedicated machines. The Crays were not dedicated, thus elapsed times on the Crays are not as meaningful due to variable load from other users. The parent wall and cpu times are the total times for the parent process, including all setup, computation of the one-electron integrals, its share of the two-electron integrals and concatenation of the integral files. The ratio of the elapsed time to the time for a single process is an accurate measure of the speed-up from parallelization. For more detailed analysis of the parallelization of the two-electron integrals we report the minimum, maximum and average cpu times spent by processes in that section of the code. If the work were perfectly shared between processes these numbers would be equal.

First consider the non-load balanced timings. The single process cpu times reflect that ARGOS is dominated by scalar floating point operations. Only the atomic to symmetry orbital transformation has significant vector content. The Ardent Titan consistently shows poorer wall time speed-up than the Aliant. This is due to the relatively slow single drive SCSI file system that was used (approximately 0.6 Mbytes/s transfer rate). A two way striped SCSI file system or SMD disks would alleviate this. For all the problems the CRAY-YMP comes in with a parent process cpu time speed-up of nearly eight on eight processors, while the Alliant is in the range of six to seven. Since the work distribution is deterministic in these tests one would naively expect the same cpu speed-up on all machines. We attribute the variation to system timing differences, variable load on the Crays (cpu timings can vary by at least 10%), the cache on the Alliant, and extra system cpu overhead for I/O on the Alliant. As the number of processes increase so does the amount of I/O that must be done to concatenate the integral files. This I/O cannot be overlapped with computation and contributes to the poor scaling of the elapsed time. However, for the largest problem

we achieve a respectable 6.4 speed-up for the total job elapsed time on eight processors of an Alliant FX/8. Faster disks and load balancing will improve this. Deficiencies in the distribution of work show up in the times spent actually evaluating integrals. In these examples ARGOS computes integrals over symmetry adapted combinations of atomic basis functions. The nested loops in ARGOS effectively run over symmetry unique basis functions. Thus a large number of symmetry equivalent centers would increase the time taken to compute integrals involving these centers. This has the effect of degrading load balancing with the simple deterministic algorithm for apportioning the work. As expected, the small water case and the highly symmetric ethylene molecule show poor load balancing. On the Alliant we have implemented a dynamic load balancing scheme using a counter shared between all the processes (this is implemented using a shared memory region). Instead of each process evaluating every nproc'th set of integrals, each process merely computes the next set which needs doing. Use of the shared counter introduces negligible overhead. Times on the Alliant using load balancing are given in Table 4 for the water and ethylene examples. The two-electron integral cpu times show a marked improvement.

Conclusions

It is possible to generate efficient portable parallel programs, but one cannot rely upon vendor provided tools. Shared memory models are not necessarily the way to use shared memory machines. The integral evaluation program illustrates this very well. The mechanism used to parallelize ARGOS would be even more efficient when applied to evaluation of SCF energy gradients, as no I/O is involved. A down side to distributed memory models is their profligate use of memory, but all parallel algorithms suffer from this to some extent. The simple model used here serves to ilustrate the value of distributed memory programming models. To significantly improve on current performance we need either to modify the manner in which the integral files are joined or to buy a faster disk sub-system. The best solution seems to require a server process writing a single file, communicating with the other processes through shared memory buffers.

Table 4. Parallel ARGOS timings with load balancing for the water and ethylene examples on the Alliant FX/8. All times reported are in seconds. The parent wall time is the total wall clock execution time for the job. The parent cpu time is the total cpu time used by the parent process. The two-electron times report the minimum, maximum and average cpu times spent evaluating two-electron integrals by the processes. Values in parentheses are ratios to times for a single process

Example	No. of	Parent process		Two-electron cpu		
	Processes	cpu	wall	min.	max.	ave.
water	1	87.6 (1.0)	89.0 (1.0)	85.8	85.8	85.8
	4	25.8 (3.4)	29.0 (3.1)	22.2	23.0	22.5
	8	15.9 (5.5)	22.0 (4.0)	12.0	13.0	12.4
ethylene	1	332.1 (1.0)	339.0 (1.0)	327.8	327.8	327.8
	4	91.5 (3.6)	100.0 (3.4)	84.3	85.2	84.8
	8	53.0 (6.3)	64.0 (5.3)	43.4	50.2	46.1

Thus the best features of distributed memory models (modularity and data independence) and shared memory models (fast synchronization and communication through shared data) will be exploited simultaneously. We would encourage computer vendors to support several portable parallel programming languages/environments (e.g. LINDA and STRAND). This helps the programmer as well as the vendor by providing a wider software base.

The simple tools used here implement a crude form of message passing using shared files. We have since developed a much more sophisticated toolset, using shared memory and TCP sockets to achieve portability and performance. These tools are very similar to the PARMACS [10] but substantially more robust.

Increased parallelism is not a substitute for an inefficient algorithm. On a single processor of a CRAY-XMP the U.K. version of GAMESS [19] takes just 6.8 cpu seconds to evaluate the integrals (symmetry unique list) for the ethylene example, rather than the 26.4 seconds used by ARGOS (symmetry adapted list). Indeed GAMESS performs the entire integrals and SCF calculation in 12.4 seconds. However, GAMESS is not capable of generating symmetry adapted integrals, or handling large generally contracted basis sets. This and other functionality means that ARGOS will stay in wide use for a while to come.

Acknowledgements. This work was supported by the U.S. Department of Energy, Office of Basic Energy Sciences, Division of Chemical Sciences, under contract W-31-109-ENG-38.

Program 1. C source for primitive version of PARALLEL and SERIAL

```
/*  Primitive versions of PARALLEL and SERIAL.

    Compile with the machine type defined on the cc command line
    e.g. cc -c -DSUN parallel.c

    Tested on: CRAY (X, Y, 2), ARDENT, SUN, ALLIANT */
# include <stdio.h>
# include <signal.h>

/*  Define routine names for compatibility with FORTRAN linking conventions. */
# define SERIALP SERIAL
# define PARALLELP PARALLEL
# if defined(SUN) || defined(ALLIANT)
#  undef SERIALP
#  undef PARALLELP
#  define SERIALP serial_
#  define PARALLELP parallel
# endif

/*  max_proc sets a sensible maximum number of forked processes */
# define max_proc 7
static int pid_list[max_proc];

/*  n_proc_made is the number of processes that are actually forked.
    This will be correct only for the parent process */
static int n_proc_made = 0;
```

```
/*  Cleanup blindly kills all child processes, and does exit(1). */
static void cleanup()
{
  while (n_proc_made--)
    (void) kill(pid_list[n_proc_made],SIGKILL);

  exit(1);
}

/*  Called by the parent serial waits for all child processes to complete. If called by
a   child it does exit(0). n_proc_made is re-set to 0 for the next call to parallel. */

void SERIALP (nproc, iproc)
    int *nproc, *iproc;
{
int pid;

(void) fflush(stdout);
(void) fflush(stderr);

if (*iproc != 0)
    exit(0);

while (n_proc_made) {
    --n_proc_made;
    pid = wait(int*) NULL);
    (void) printf("SERIAL: Child finished, pid = %d.\n", pid);
  }
}

\* Parallel forks nproc-1 copies and returns iproc with a unique value 0- > nproc-1,
the parent process having value 0. Upon any error it tries to kill the child processes
and then aborts. */

void PARALLELP (nproc, iproc)
    int *nproc, *iproc;
{
  int i, pid;

  if ((*nproc > (max_proc + 1)) || (*nproc < =0)) {
        (void) fprintf(stderr, "PARALLEL: nproc = %d, min = 1, max = %d.\n",
                        *nproc, max_proc + 1);
        exit(1);
}

(void) fflush(stdout);
(void) fflush(stderr);
*iproc = 0;

for (i = 1; i < *nproc; i++) {
    if ((pid = fork()) = = -1) {
        (void) fprintf(stderr, "PARALLEL: Error forking process %d.\n", i);
        cleanup();
    }
    else if (pid = = 0) {
          *iproc = i; return;
```

```
    }
    else {
        (void) printf("PARALLEL: Forked process %d, pid = %d.\n", i, pid);
        (void) fflush(stdout);
        pid_list[n_proc_made++] = pid;
    }
  }
}
```

Program 2. FORTRAN source illustrating use of the parallel and serial subroutine calls

```
      program main
      character*40 name
      data limit/10/
c
c     Demo program for parallel and serial calls.
c

 11   write(6, *) 'Input number of processes, 0 to quit'
      read (5, *) nproc
      if (nproc.eq.0) call exit(0)
c
c     Go parallel. All processes then execute code up to call to serial.
c
      call parallel(nproc,iproc)
c
c     Each process opens its own data file, testxxxx
c
      write(name, '("/tmp/test",i4.4)') iproc
      open(1,form = 'unformatted',status = 'unknown',file = name)
      rewind 1
c
c     Share out the work using icount. The mod(icount,nproc)
c     mechanism allows a nest of loops to be parallelized.
c
      icount = iproc
      do 10 i = 1,limit
          icount = icount + 1
          if (mod(icount,nproc).ne.0) goto 10
          write(1)i
 10   continue
c
c     Go serial. Explicitly close the children's data file first.
c
      if (iproc.ne.0) close(1,status = 'keep')
      call serial(nproc,iproc)
c
c     Append other files to the end of the main process's file
c
      do 20 iproc = 1,nproc-1
          write(name,'("/tmp/test",i4.4)')iproc
          write(6, *) 'Appending file',name
          nval = 0
          open(2,form = 'unformatted',status = 'unknown',file = name)
```

```
 30    read(2,end = 35) j
             nval = nval + 1
             write(1) j
             goto 30
 30    close(2,status = 'delete')
             write(6, *) 'Number of values on the file were',nval
 20    continue
c
c      Check the results.
c
       write(6, *)'Checking results'
       rewind 1
       jsum = 0
       isum = 0
       do 40i = 1,limit
             read(1) j
             write(6, *) j
             isum = isum + j
             jsum = jsum + j
 40    continue
c
       write(6, *) 'isum = ',isum,'jsum = ',jsum
       close(1,status = 'delete')
       goto 11
c
       end
```

References

1. Saunders VR, Guest MF (1982) Comput Phys Commun 26:389
2. Bair RA, Dunning Jr TH (1984) J Comp Chem 5:44
3. Bauschlicher Jr CW (1987) Theor Chim Acta 71:105
4. a) Clementi E (1985) J Phys Chem 89:4426; b) Corongiu G, Detrich JH (1985) IBM J Res Dev 29(4):422
5. Bauschlicher Jr CW (1989) Theor Chim Acta 76:187
6. Guest MF, Harrison RJ, Lenthe JH van, Corler LCH van (1987) Theor Chim Acta 71:117
7. a) Clementi EC, Corongiu G, Detrich JH, Khanmohammadbaigi H, Chin S, Laarksonen A, Nguyen HL, IBM Technical Report, KGN-2, May 20, 1984; b) Watts JE, Dupuis M, Villar, HO, IBM Technical Report, KGN-78, August 29, 1989; c) Dupuis M, Watts JD (1987) Theor Chim Acta 71:91
8. Fox GC, Otto SW (1984) Physics Today 37.5:50
9. Pitzer R (1973) J Chem Phys 58:3111
10. Boyle J, Butler R, Disz T, Glickfeld B, Lusk E, Overbeek R, Patterson J, Stevens R (1987) Portable Programs for Parallel Processors, Hold, Rinehart and Winston Inc
11. Seitz CL (1985) Caltech report; Communications of the ACM 28 (1985) 22
12. Carriero N, Gelertner D (1985) ACM Transactions on Computer Systems 3:77
13. Andrews G, Olsson R (1988) TOPLAS (ACM Transactions on Programming Languages and Systems) 10.1:51
14. Foster I, Taylor S (1990) Strand, New Concepts in Parallel Programming, Prentice Hall, New Jersey
15. Watts JD, Dupuis M (1988) J Comp Chem 9:158
16. Handy NC (1980) Chem Phys Lett 74:280
17. Krishnan R, Binkley S, Seeger R, Pople J (1980) J Chem Phys 71:650
18. Dunning TH, private communication of unpublished work
19. Guest MF (1989) GAMESS, Users's guide and reference manual. Science and Engineering Research Council, Daresbury Laboratory