Parallelizing MOPAC on distributed computing systems within AVS framework

Tseng-Hui Lin Tomasz Haupt Geoffrey C. Fox
Northeast Parallel Architecture Center at Syracuse University
Syracuse, NY 13244-4100

Email: *thlin/haupt/gcf@npac.syr.edu*

Abstract

MOPAC [1] is a general-purpose semi-empirical molecular orbital package for the study of chemical structures and reactions. Semiempirical Hamiltonians are used in the electronic part of the calculation to obtain molecular orbital, the heat of formation and its derivative with respect to molecular geometry. The computation time increases rapidly as the numbers of molecules increased. Parallelizing MOPAC on distributed computing systems will improve the computation time in low costs.

Molecular geometries was described by atoms coordinates in text. 3-D look AVS graphic molecular geometries will give the users of MOPAC a better image of molecular structures.

Index Terms : MOPAC, molecular, Hamiltonians, heat, derivative, parallelizing, distributed system, 3-D look, AVS.

1 Introduction

1.1 What is MOPAC?

MOPAC is a general-purpose semi-empirical molecular orbital package for the study of chemical structures and reactions. Semiempirical Hamiltonians are used in the electronic part of the calculation to obtain molecular orbitals, the heat of formation and its derivative with respect to molecular geometry.

Using these results MOPAC calculates the vibrational spectra, thermodynamic quantities, isotopic substitution effects and force constrains for molecules, radicals, ions, and polimers. For studying chemical reactions, a transition-state location routine and two transition state optimization routines are available, too.

1.2 Problem

Although MOPAC has been very successfully used in many research projects, its application is limited to small molecules, typically consisting of no more than 60 atoms. The limitation comes from the CPU demand that increases with the number of atoms n roughly as n-square. A possible solution is to port the code to a modern, parallel or distributed computer architecture.

1.3 Project description

Within this project, the computational structure of the program is to be examined in order to identify opportunities for parallelization and to select an appropriate hardware configuration for implementation. A model implementation of selected fragments of code on architecture of choice will be provided as well.

2 Program Analyzing

2.1 Executive Summary

Analyze of the MOPAC code is not an easy task. The source consist of almost 30,000 lines written by many authors. The idea behind creating MOPAC was to integrate several independent codes into a single packet. As a result, MOPAC's control structure is complicated, typically known only at runtime. Even though the code has been created according to rules that makes software management easier, we found it necessary to develop dedicated tools to trace and profile the execution for the benchmark data sets. These tools are described in section 2.2.1.

The structure of the MOPAC is described in section 2.3. It can be summarized as follows. By selecting input parameters, the user may choose one of *nine* basic paths of execution. Our benchmark data sets select FLEPO followed by POLAR modules, and we concentrate our analysis on them. FLEPO optimizes a geometry by minimizing the energy, and POLAR calculates the

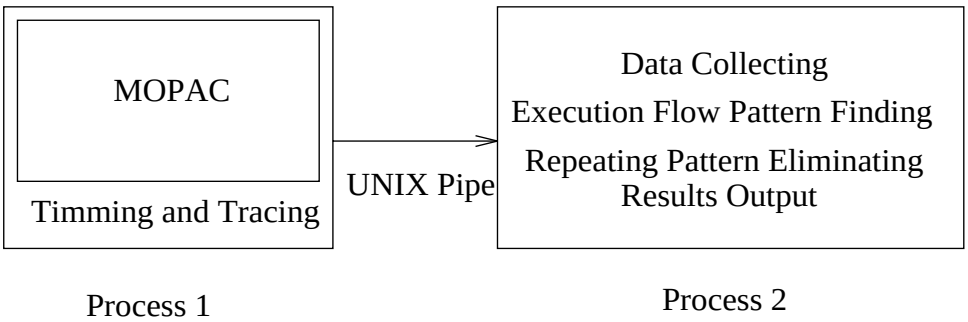

Figure 1: The tracing process structure

polarizability volumes for a molecule, using SFC calculations. We also looked at other modules, and we found essentially similar computational pattern.

The main computational burden is carried by the subroutine COMPFG that calculates the total heat of formation of the supplied geometry, and the derivatives, if requested. COMPFG consumes typically about 99% of total CPU time and it is called many times at various phases of computations. The actual computation path inside COMPFG is different for different invocations of COMPFG. However, the main computational load is always taken either by subroutine ITER or ITER and DERIV (typically, together they constitute about 97% of the total CPU time usage).

Subroutine ITER constructs one- and two-electron matrices, and then calculates the fock and density matrices, and the electronic energy. Most of the computation is concentrated in subroutines DENSIT (Coulson electron density matrix), HQRII (rapid diagonalization routine) and DIAG (pseudo-diagonalization). Preliminary analysis indicates that DENSIT can be efficiently parallelized, while diagonalization routines DIAG and HQRII should be replaced by new ones that will employ algorithms better suited for parallel execution.

Subroutine DERIV calculates the derivatives of the energy w.r.t. the geometric variables. Its computational structure promises easy parallelization.

To conclude, majority of computation is performed by 4 critical subroutines, and we will provide parallel implementation of them. To support such a mixed sequential and parallel programming paradigm we propose IBM RS6000 workstations (sequential part) and SP-2 (parallel part written in Fortran77 + PVM) integrated using AVS system. The details of the architecture are given in section 4.1.

2.2 Execution flow tracing

The most important data we need is execution flow which shows how much time each subroutine uses and how many times they are called. We'd like to know how much time each subroutine spends and how many times they are called when executing. A tracing utility program is developed and some code is added into MOPAC source program to collect the execution and timing information of MOPAC subroutines. We choose 3 test data and run them on different platforms to see the real execution flow. The 3 test data may not be general enough for us to

find out all execution flow patterns. However, they show us some general idea.

2.2.1 The tracing utility

The structure of the tracing utility is shown in figure 1. All monitoring programs introduce some timing error. To minimize the timing error introduced by the tracing utility, we divide our tracing utility into two parts and run as two processes. When the tracing program starts, it forks a subprocess. The main process executes MOPAC program and sends tracing information to the subprocess via a pipe. The subprocess handles all other jobs, data collection, execution flow pattern finding, repeating execution flow pattern elimination, and results output. The timing utility calculates only CPU time used by MOPAC subroutines and the timing utility itself. Time used by subprocess will not be included. Thus minimize the timing error.

2.2.2 Test data

The test data sets are chosen from the Bryan's test sets. They are:

- apsbtest.in
- porphin.in
- tetrabenz.in

2.2.3 Timing Errors

We run the test data sets with and without our tracing utility on several machines to see if the error introduced by our tracing utility is small enough to reflect the real execution timings. The machines we used are :

- nova.npac.syr.edu : Sun Sparc-10 model 31
- oldnova.npac.syr.edu : Sun4-40MHz
- merlin.npac.syr.edu : IBM RISC6000 model 370
- mel.npac.syr.edu : IBM RISC6000 model 550

The total execution time of our test data sets with and without our tracing utility is shown in table 1 The timing error introduced by our tracing utility for all 3 test data sets on all machines are all less than 2%. The results of the tracing utility should be accurate enough to reflect the real execution timings.

apsbtest	with trace	without trace	error
nova	11038.96	10937.87	0.92%
oldnova	17157.69	16953.11	1.21%
merlin	4817.16	4760.03	1.20%
mel	7331.18	7257.36	1.02%
porphin	with trace	without trace	error
nova	1671.85	1653.97	1.08%
oldnova	2625.37	2580.25	1.55%
merlin	734.74	720.35	2.00%
mel	1131.10	1109.45	1.95%
tetrabenzi	with trace	without trace	error
nova	2923.46	2907.54	0.55%
oldnova	2778.68	2757.24	0.78%
merlin	2905.28	2869.90	1.23%
mel	3168.95	3128.21	1.30%

Table 1: Timing error introduced by tracing program

2.3 MOPAC flow diagram

In this section, we are going to show the program structure of time consuming subroutines and their timing. We will use the percentages of whole program execution time to show the timing of subroutines. The results of our test data are similar on different machines. We are going to describe our analysis by using the results generated on mel.npac.syr.edu.

2.3.1 Computation in main program

Program structure of main program The following is the flow diagram of MOPAC main program. The execution flow is that it reads in a set of keywords and a set of atom geometry, checks keyword to decide which subroutine to call, and goes back to begin of program if there are more sets of atom geometry to work on. The main loop in main program is controlled by the variable "ISOK". ISOK is set to TRUE in the beginning of main program and will be set to FALSE in deriv.f and readmo.f. However, from the tracing results of all test data sets, ISOK is never set to FALSE thus execution flow never normally ends. Instead, it always reaches a "STOP" statement in subroutine GETTXT called by READMO.

Subroutine GETTXT reach STOP statement when all input atom geometry have been read. Thus, the main loop in main program loops for only one or several times. In all our test sets, the main loop loops for only once. It's not the main computation loop we are looking for.

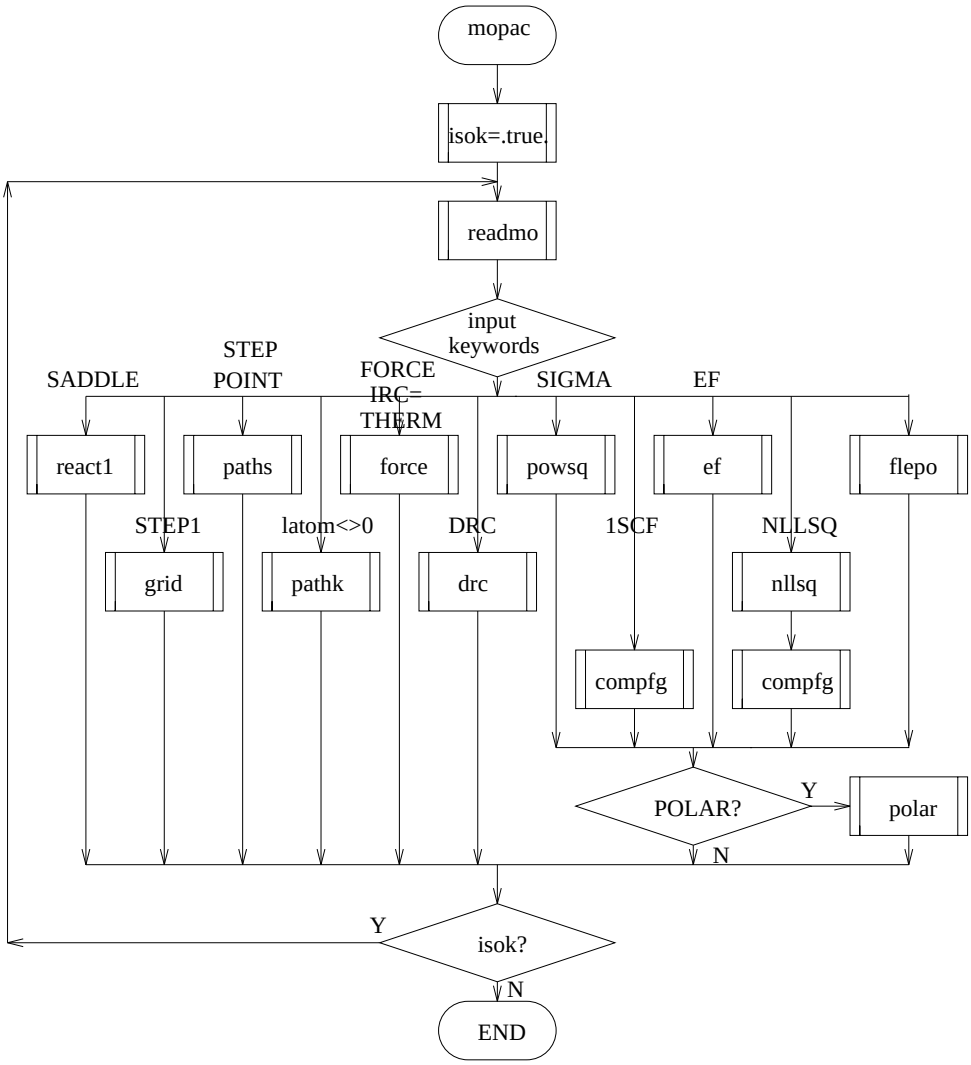

Figure 2: The flow diagram of main program

	apsbtest			porphin			tetrabenz		
	Total	Caller	Times	Total	Caller	Times	Total	Caller	Times
FLEPO	50.44	50.44	1	99.97	99.97	1	99.99	99.99	1
POLAR	49.56	49.56	1	0.00	0.00	0	0.00	0.00	0
Sum	100.00	100.00		99.97	99.97		99.99	99.99	

Table 2: Main program timing

	apsbtest			porphin			tetrabenz		
	Total	Caller	Times	Total	Caller	Times	Total	Caller	Times
COMPFG	10.01	19	57	35.04	35	49	22.24	22	36
LINMIN/COMPFG	39.91	$79 \times 99 = 78$	65	63.90	$63 \times 99 = 62$	50	77.09	$77 \times 99 = 76$	38
Sum	49.92	97		98.94	97		99.33	98	

Table 3: Timing of subroutine COMPFG called from FLEPO and LIMIN

Computation timing of main program All our 3 test data go into the last subroutine, FLEPO, in main program. Since all 3 test data sets contains only one atom geometry set, the main loop in main program loops for only once. The execution flow goes to READMO to read in a set of atom geometry then FLOPE and POLAR then STOP in READMO/GETTXT when it finds the input file reaches EOF.

The timings of the test sets are shown in table 2. Almost all time is taken by FLEPO and POLAR. Since they are called for only once, the computation should be inside the two subroutines. As we have mentioned above, the computation kernel of these subroutines is COMPFG. We should trace into these two subroutines to see how COMPFG works inside these two subroutines.

Computation in FLEPO and POLAR

Subroutine FLEPO FLEPO has a main loop which calls COMPFG and LINMIN. The main loop loops for 53, 40, and 32 times in apsbtest.in, porphin.in, and tetrabenz.in respectively. The most time is consumed by COMPFG and LINMIN. An interesting timing is observed that most of time used in LINMIN is to call COMPFG. That means most of time used in FLEPO is to call COMPFG directly from FLEPO and indirectly from LINMIN. The timing is shown in table 3

COMPFG takes 97%, 97%, and 98% of computation time out of FLEPO in apsbtest.in, porphin.in, and tetrabenz.in respectively. Time taken by codes other than COMPFG is almost

	apsbtest			porphin			tetrabenz		
	Total	Caller	Times	Total	Caller	Times	Total	Caller	Times
COMPFG	4.06	8	1	0.00	0	0	0.00	0	0
FFHPOL/COMPFG	45.49	91×99	36	0.00	0	0	0.00	0	0
		$= 90$					0.00	0	
Sum	49.55	98		0.00	0		0.00	0	

Table 4: Timing of subroutine COMPFG called from POLAR

	apsbtest		porphin		tetrabenz	
	Total	Times	Total	Times	Total	Times
FLEPO/COMPFG	10.01	57	35.04	49	22.24	36
FLEPO/LINMIN/COMPFG	39.91	65	63.90	50	77.09	38
POLAR/COMPFG	4.06	1	0.00	0	0.00	0
POLAR/FFHPOL/COMPFG	45.49	36	0.00	0	0.00	0
Sum	99.47	159	98.94	99	99.33	74

Table 5: Timing of subroutine COMPFG called from FLEPO and POLAR

nothing.

Subroutine POLAR POLAR is called only in apsbtest.in. There is no loop in POLAR. It calls COMPFG and FFHPOL once each. The most time is consumed by COMPFG and FFHPOL. Just like FLEPO, most of time used in FFHPOL is to call COMPFG. That means most of time used in POLAR is to call COMPFG directly from POLAR and indirectly from FFHPOL. The timing is shown in table 4.

COMPFG takes 98% of computation time out of POLAR in apsbtest.in Time taken by codes other than COMPFG is almost nothing.

Conclusion of computation time The subroutine called in main loop are mutual-exclusive from each other. Only one of them will be called at a time depending on the input keywords. Every subroutine called in main loop has similar execution structure.

They call COMPFG directly or indirectly. The codes of these subroutines are mostly preparing data for calling COMPFG and organizing results generated by COMPFG. Subroutine COMPFG is the computation kernel of those subroutines.

Lets combine the above timing tables. The most time is taken by directly and indirectly calls to COMPFG.

We can conclude from table 5 that COMPFG is really the computation kernel of whole

	apsbtest		porphin		tetrabenz	
	Total	Times	Total	Times	Total	Times
FLEPO/COMPFG/DERIV	7.13	56	12.34	48	8.79	36
FLEPO/COMPFG/ITER	2.83	4	22.35	9	13.31	4
FLEPO/LINMIN/COMPFG/ITER	39.10	65	61.96	50	75.76	38
POLAR/COMPFG/ITER	4.05	1	0.00	0	0.00	0
POLAR/FFHPOL/COMPFG/ITER	45.05	36	0.00	0	0.00	0
Sum	98.16		96.65		97.86	

Table 6: Timing of subroutine COMPFG

program. Also, we find COMPFG is not called any other place. COMPFG is called for only 159, 99, and 74 times in our test sets. To parallel the loops which call COMPFG is not big deal since the loop is not big.

From the above results, we have the following conclusions:

- Time taken by codes other than COMPFG is almost nothing.
- To speed-up MOPAC means to speed-up COMPFG.
- Parallelization should be checked inside COMPFG instead of outside COMPFG.

2.3.2 Computation in COMPFG

Program structure of COMPFG Figure 3 shows the COMPFG flow diagram.

The "Basic Statements" in COMPFG flow diagram means a set of statements which don't call any other subroutine and have no loop.

There is no major loop in COMPFG. There are only two small DO loops. The first DO loop just do some basic statements for NUMAT times, which is small (less than 100) in our test sets. The second DO loop calls DIHED while DIHED never been called in our example. The time used by the two loops is little and can be ignored in computation flow tracing.

Since no significant time is consumed by the loops in COMPFG, the real time consuming parts is not inside COMPFG but inside the subroutines called by COMPFG. COMPFG calls SYMTRY, GMETRY, HCORE, ITER, DIHED, DERIV, and MECIP. All subroutines except GMETRY may or may not execute depending on the conditions given by the input parameters of COMPFG.

Computation timing of COMPFG From the previous session, COMPFG is called from 4 different callers. According to the tracing results, almost all computation time is taken by DERIV and ITER.

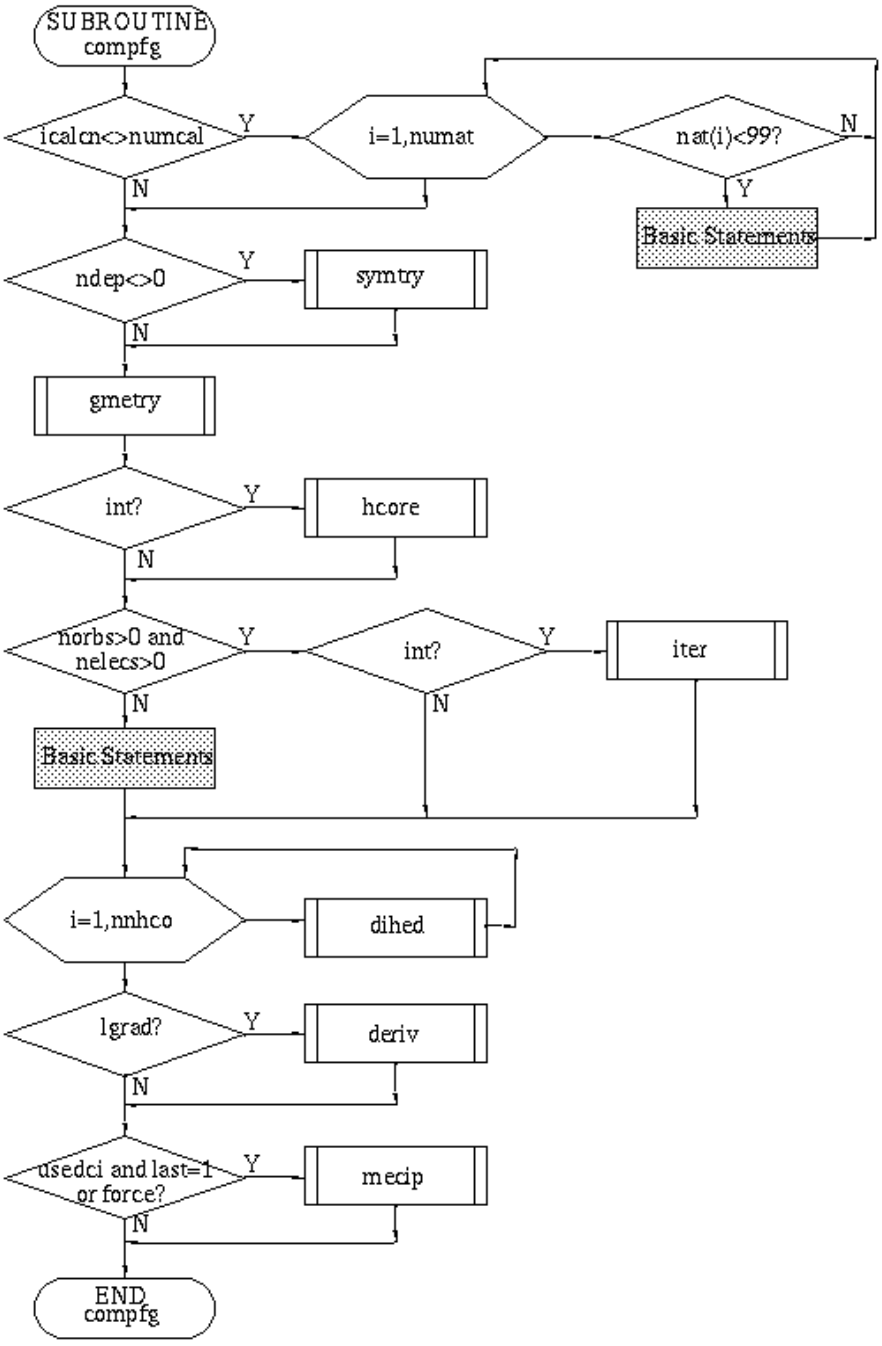

Figure 3: The flow diagram of subroutine COMPFG

	apsbtest		porphin		tetrabenz	
	Total	Times	Total	Times	Total	Times
FLEPO/COMPFG/DERIV	7.13	56	12.34	48	8.79	36
FLEPO/COMPFG/DERIV/DCART	6.51	56	11.56	48	8.22	36
FLEPO/COMPFG/DERIV/DCART/DHC	6.25	656208	11.09	134976	7.87	272304

Table 7: Timing of subroutine deriv

Table 6 shows the timing of subroutine COMPFG. DERIV is called by FLEPO/COMPFG only while ITER is called by all four cases. Again, the sum of time spent by DERIV and ITER is almost everything. We should look inside of DERIV and ITER.

2.3.3 Computation in DERIV and ITER

Subroutine DERIV The timing of DERIV and its subroutines are shown in table 7. DERIV is called in only one location, FLEPO/COMPFG. There is no major computation loop in DERIV. The most important subroutine called by DERIV is DCART. DCART is called in only one place in DERIV and takes 91%, 93%, and 93% of computation time from DERIV in apsbtest.in, porphin.in, and tetrabenz.in respectively. DERIV is almost nothing but DCART.

Go further into DCART, there is a set of loops calling DHC. Our tracing results show that DHC takes 96%, 95%, and 95% of computation time from DCART in apsbtest.in, porphin.in, and tetrabenz.in respectively. The most time of DCART is spent on DHC.

Moreover, DHC is called for 656208, 134976, and 272304 times and takes 6.25%, 11.09%, and 7.87% of total CPU time in apsbtest.in, porphin.in, and tetrabenz.in respectively. Those are big numbers of calls. Although DHC takes 6.25%, 11.09%, and 7.87%, which are 458.33, 125.44, and 249.27 seconds, each call to DHC takes only 0.7, 0.9, and 0.9 ms. It should be better to parallelize the loops in DCART than parallelize DHC itself.

Subroutine ITER ITER is called all the time when COMPFG is called in all cases. It takes much more CPU time than DERIV does. The most time of ITER is used by DENSIT, DIAG and HQRII. Table 8 shows the timing of DENSIT, DIAG and HQRII called from ITER.

Subroutines DENSIT and HQRII do not call any other subroutine. DIAG calls EPSETA, which does not call any subroutine. They are the lowest level of subroutines while take most of CPU time. To speed-up MOPAC need to speed-up these subroutines first.

Subroutine DENSIT DENSIT is used to construct Coulson electron density matrix. It doesn't call any other subroutines. A two level major computation loop eats the most time. The loop should be parallelized.

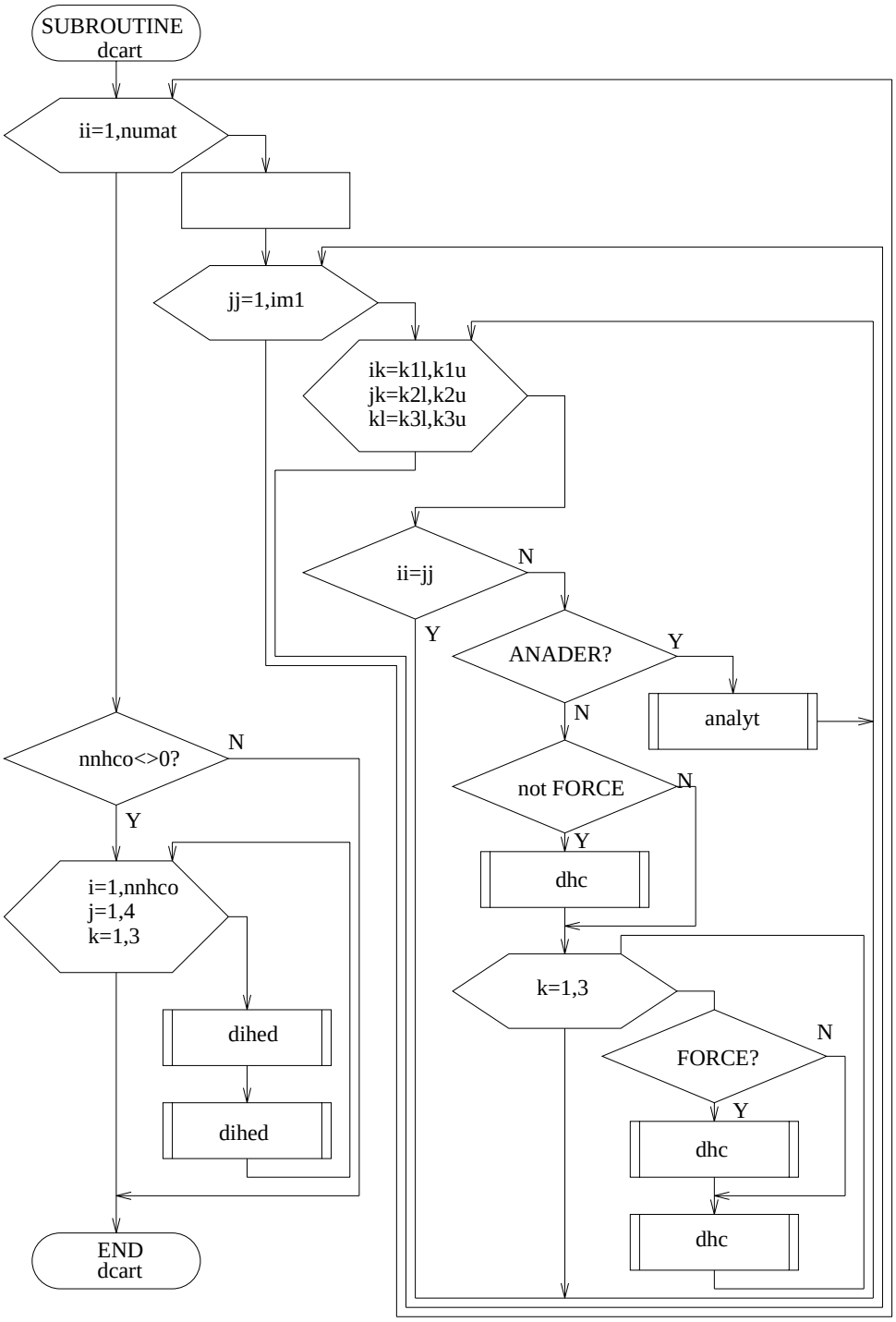

Figure 4: The flow diagram of subroutine DCART

	apsbtest		porphin		tetrabenz	
	Total	Times	Total	Times	Total	Times
FLEPO/COMPFG/ITER/DENSIT	0.48	47	4.00	123	2.39	45
FLEPO/LINMIN/COMPFG/ITER/DENSIT	7.03	688	10.14	314	13.44	253
POLAR/COMPFG/ITER/DENSIT	0.57	56	0.00	0	0.00	0
POLAR/FFHPOL/COMPFG/ITER/DENSIT	8.10	799	0.00	0	0.00	0
Sum	16.18	1590	14.14	437	15.83	298

(a) DENSIT

	apsbtest		porphin		tetrabenz	
	Total	Times	Total	Times	Total	Times
FLEPO/COMPFG/ITER/DIAG	1.74	42	15.49	118	8.96	42
FLEPO/LINMIN/COMPFG/ITER/DIAG	30.51	688	48.49	314	59.73	253
POLAR/COMPFG/ITER/DIAG	0.90	23	0.00	0	0.00	0
POLAR/FFHPOL/COMPFG/ITER/DIAG	30.02	763	0.00	0	0.00	0
Sum	63.17	1516	63.98	432	68.69	295

(b) DIAG

	apsbtest		porphin		tetrabenz	
	Total	Times	Total	Times	Total	Times
FLEPO/COMPFG/ITER/HQRII	0.51	7	1.72	7	1.52	4
FLEPO/LINMIN/COMPFG/ITER/HQRII	0.00	0	0.00	0	0.00	0
POLAR/COMPFG/ITER/HQRII	2.47	34	0.00	0	0.00	0
POLAR/FFHPOL/COMPFG/ITER/HQRII	5.24	72	0.00	0	0.00	0
Sum	8.22	113	1.72	7	1.52	4

(c) HQRII

Table 8: Timing of subroutines called by iter

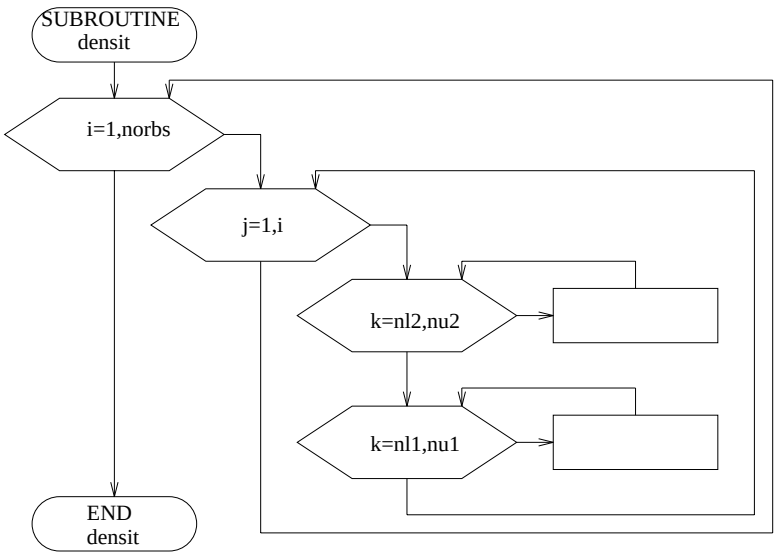

Figure 5: The flow diagram of subroutine DENSIT

	apsbtest		porphin		tetrabenz	
	Total	Times	Total	Times	Total	Times
DCART	6.51	56	11.56	48	8.22	36
DENSIT	16.18	1590	14.14	437	15.83	298
DIAG	63.17	1516	63.98	432	68.69	295
HQRII	8.22	113	1.72	7	1.52	4
Sum	94.08		91.40		94.26	

Table 9: Timing of the four most time-consuming subroutines

Subroutine DIAG DIAG is the pseudo-diagonalization program. It contains two major loops. The first loop can be divided into two smaller loops. No special statements or subroutine calls in the two loops. The two loops should be able to be parallelized.

Subroutine HQRII HQRII is a rapid diagonalization program. It was written in complex FORTRAN 4 style. It is un-structure and complex. To re-write it may be better than parallelize it. There are several diagonalization algorithms can be used. Just like DIAG, we can find a good parallel algorithms to rewrite it into parallel version.

2.3.4 Time of DCART, DENSIT, DIAG, and HQRII

The total time spent by the four subroutines is shown in table 9. The four subroutines take more than 90% of CPU time in all three test data. The four subroutines are the targets of parallelization.

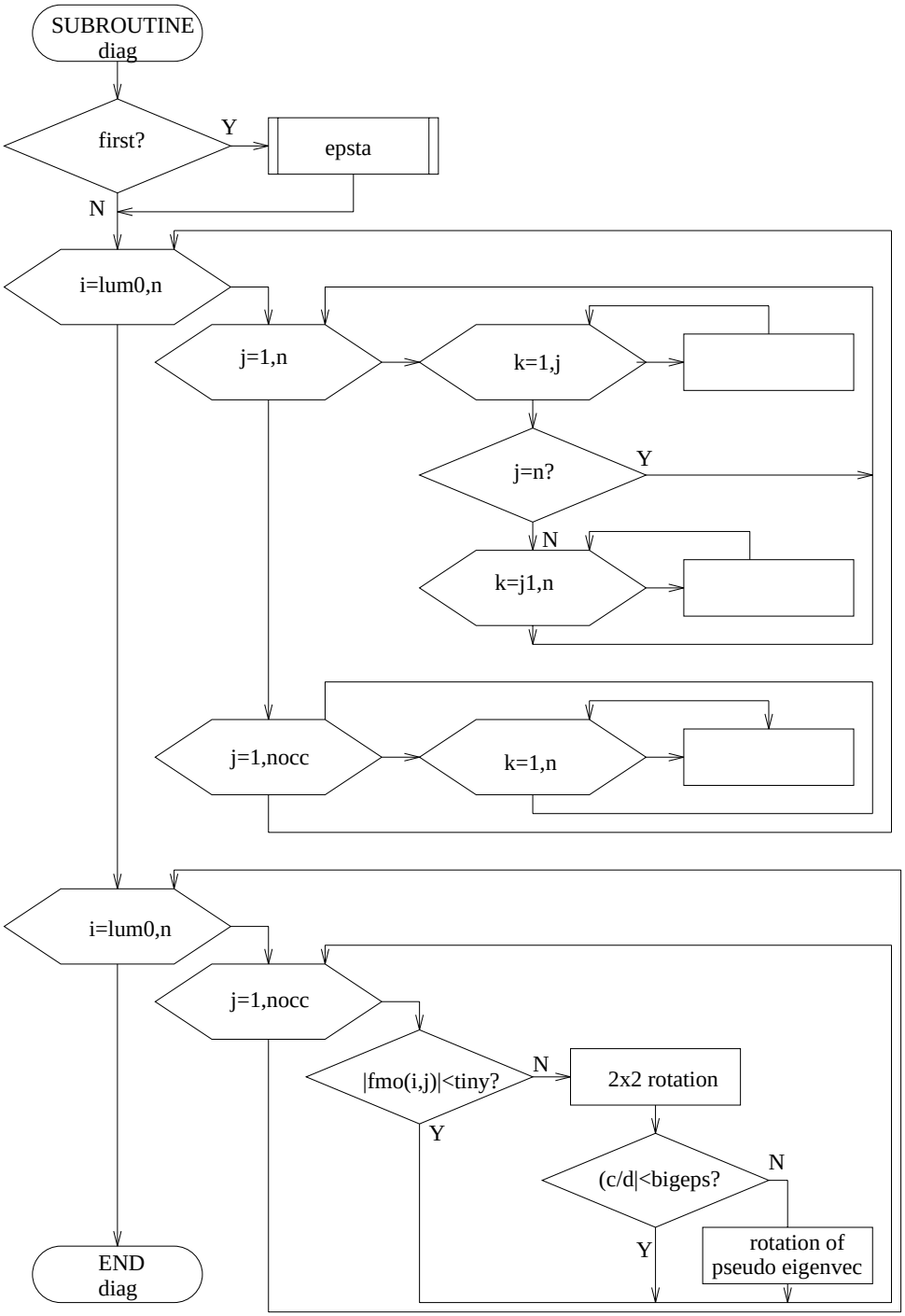

Figure 6: The flow diagram of subroutine DIAG

2.3.5 Conclusion

From the above disscussion, we can get the following concludsion:

- The execution of MOPAC depends on the input keywords. The keywords cause the execution flow goes through one of several mutual-exclusive execution flow.
- The CPU time is consumed by several kernel subroutines. Parallelize these subroutines will improve the execution a lot.
- The kernel subroutines we are going to work on are:
  - DCART : parallelize the main loop
  - DENSIT : parallelize the main loop
  - DIAG : Parallelize the two main loops
  - HQRII : Re-write by using parallel algorithms

3 Parallelization of MOPAC

3.1 Before parallelization

What's not good for parallelization Parallelization is not possible for essentially sequential codes. A segment of sequential code is a set of program statements which have strong data dependencies among some of these statements that cause the segment of code need to be execute one by one. No statement can be executed until its immediate previous statement has been executed. Statements in a segment of sequential code must be executed one after another and can not be parallelized.

To divide work load, distribute data, and collect results, we need to introduce some extra code. These codes are usually small and takes little CPU time. However, some highly optimization load balancing algorithms do be very complex and take much CPU time. Sometimes they take so much time that become useless for real applications. If the parallelable code won't take long, use a sub-optimized algorithm or even give up parallelization will be better.

Besides CPU time, parallelization introduces communications. When a segment of code is divided and distributed, every computation node owns only parts of data and results thus it may need to exchange data with other nodes. Communications are needed to distribute data and collect results. Communications are expensive since they are slow comparing with CPU speed and usually take long. Workstation clusters usually use Ethernet for communication. The bendwith of Ethernet is only 10Mbits/sec. Even for parallel computers which equip with special high speed communication channels, the communications are still far slower than computing. Parallel code with much communications may run slower than sequential code.

Communications also imply synchronization. Synchronization means idle and wait. If one node finishes its computation early, then it needs to wait for the data it needs from other slower nodes to continue its computing. All faster nodes need to wait for slower nodes for some time

```
S1  : loop I=1 to 1000
S2  :       loop J=1 to I

S3  :             Computation

S4  :       endloop J
S5  : endloop I
```

Figure 7: Example program of range of inner loop depends on the index of outer loop.

especially in global operations. Unfortunately, parallel programs usually need to synchronize all nodes to make sure everybody has reached some check points or has received all data it needs before they can start a new stage of computation. Only tightly coupled computation nodes with highly load balanced code can minimize the synchronization idle. [2, 3] [REF: J.C.Wang's communication scheduling papers] Unless the communication and synchronization time is very small, fine grain codes may not worthy for parallelization. In loosely coupled parallel machines with slow inter-processor communications, parallel codes may even run slower than their sequential counterparts for fine grain codes. Only big chunks of computations with few communications are adequate for parallelization.

According to the reasons above, we may give up parallelizing a segment of code even if it is easy to parallelize.

Load balance We define a segment of pure parallel code as a segment of code which has no data dependency among any processors and no inter-processor data exchange required when executing the segment of code. When we parallelize a segment of pure parallelable code, the code for dividing the loops and distributing data is added before the pure parallel code and the code for collecting results is added after the pure parallel code. The sizes of the pure parallel segments should be as big as possible to keep the communication as few as possible and load balancing should be considered to minimize the synchronization time. For example, if a two level loop is parallelable, we will divide the outer loop instead of the inner loop. Also, if the outer loop is not big enough comparing with the number of computation nodes, say, 20 times of loop divided for 32 nodes, the inner loops are also considered to make the load balance.

In some cases, the computational space is not regular. For example, the range of inner loop depends on the index of outer loop. Suppose we have a program shown in figure 7.

The computational space is a triangle. If we divide only outer loop, the load won't be balance due to the one computes bigger $I's$ has more heavy load than the one computes smaller $I's$. If we divide the inner loop, the size of pure parallel segment will be reduced and more communication

```
S1  : I=1
S2  : J=1
S3  : loop K=1 to 1000*(1000+1)/2

S4  :       Computation

S5  :       J=J+1
S6  :       if J¿I then
S7  :             J=1
S8  :             I=I+1
S9  :       endif
S10 : endloop K
```

Figure 8: Algorithm of moving inner index out

and synchronization will be introduced. In this case, we need to consider both loops. Let's introduce another variable K and rewrite the program as figure 8.

We add one increment statement and an if structure in the new program and make it one level loop. The program looks longer than the original one. In fact, computers deal a loop as increment the index and check if the index exceeds its range internally. What we did is just write it explicitly. We lose nothing and make the parallelization much easier.

Overlapping communication with computation Usually, we divide program into several stages of computations and communications. Results are exchanged after each stage of computation. Communications are not parts of computations and they are expensive. The communication channel is idle during computing. Why don't we send partial results when they are ready and the communication channel is idle. We may overlap communication with computation to minimize communication time.

The best way to overlap communication with computation is to use asynchronous communications. However, only few communication library supports asynchronous communications. Most communication libraries do support non-blocking send. It is also good for overlapping communication with computation. The only problem is that the receiving node will need to buffer a lot of incoming messages. We need to check for incoming messages and consume them periodically to release the message buffers.

Another problem is the performance issue. Sending several small messages will be slower than packing them into one big message. The smaller the data size, the better the overlapping. On the other hand, the larger the message, the less the communication time. The best data

size depends on the communication and computation speeds. We will need to some tests to fine tune the chunk size of messages.

3.2 Parallelizing subroutine DIAG

As we described before, subroutine DIAG takes the most CPU time. It takes 63.17%, 63.98%,and 68.69% of total CPU time in our test sets. It is the most wanted one for parallelization.

3.2.1 Program analysis

We check if it is possible to extend the size of the parallelable segment to the caller of DIAG. DIAG is called in ITER only. However, there are two calls to DIAG in ITER and they both have complex if structure surrounding instead of loop. We can not extend the parallel segment to ITER unless we want to parallelize other sequential parts of subroutine ITER also.

We need to check if the subroutines which are called by DIAG are parallelable. DIAG calls only one subroutine, EPSETA. EPSETA calculates ETA, the smallest representable number, and EPS, the smallest number for which $1 + EPS \neq 1$. Neither input parameter nor common block is used in subroutine EPSETA. The only thing which affects the values of EPS and ETA is the floating point precision of the target machine. As long as the target machine remain the same, the results of subroutine EPSETA keep the same. That means, we need compute EPS and ETA once and save them for later calls to EPSETA. There is no computation in EPSETA except the first call. Subroutine EPSETA needs to be re-written instead of parallelizing.

DIAG is a pseudo-diagonalization procedure. It contains two main loops. We need to examine the data dependency for parallelization.

3.2.2 Data dependency

The only modified non-local data is 2-D array $VECTOR$. Common 1-D array FMO is modified also. However, it is is common block $SCRACH$ which is a working space and the values of all variables in common block $SCRACH$ won't be used in any other subroutines. $VECTOR$ is the only output array.

The result array $VECTOR$ depends on input parameters N, $NOCC$ and EIG, scratch array FMO and local variable $TINY$. The input parameters N, $NOCC$ and EIG are not changed anywhere in DIAG. FMO and $TINY$ are the results of the first loop. Array $VECTOR$ also depends on itself. Figure 9 is the simplified program structure:

There are three loops in this part.. There is a conditional structure in S4. Statement S4 will skip some computation. If we distribute over either loop S1 or S2, load balance won't be guaranteed.

Statements S7 and S8 may reference the results of statements S9 and S10. Also, if we distribute over either loop S1 or S2, there will be communications required for either S7 and S9 or S8 and S10. The data dependence prevents us from parallelizing loop S1 and S2.

We check the innermost loop S6. There is no data dependency if we distribute over S6. Elements of array $VECTOR$ do not dependent on themselves. No communication is required

```
S1  :       DO 90 I=LUMO,N
S2  :           DO 80 J=1,NOCC
S3  :C          Some statements here
S4  :           IF(ABS(C/D).LT.BIGEPS) GOTO 80
S5  :C          Some statements here
S6  :           DO 70 M=1,N
S7  :               A=VECTOR(M,J)
S8  :               B=VECTOR(M,I)
S9  :               VECTOR(M,J)=ALPHA*A+BETA*B
S10 :               VECTOR(M,I)=ALPHA*B-BETA*A
S11 : 70            CONTINUE
S12 : 80        CONTINUE
S13 : 90    CONTINUE
```

Figure 9: Simplified DIAG program structure

in loop S1 or S2 either. We can distribute over loop S6 in front of loop S1 and collect the results after S13. The loop is distributed easily, the work load is balance, and the communication is minimized.

The first loop of subroutine DIAG generates array FMO and variable $TINY$. There is no data dependence for array FMO if we distribute the outermost loop directly. There is no inner loop ranges depend on the index of the outermost loop either. We can distribute the outermost loop directly.

Statement S20 generates variable $TINY$. $TINY$ will be the largest value of $ABS(SUM)$ in the end of loop S1$\sim$S22. We will have local maximum of variable $TINY$ if we distribute over loop S1. We need to make a global maximum in the end of loop S1$\sim$S22.

3.3 Implementation

As described above, we make the following change:

1. We modify subroutine EPSETA such that the value of EPS and ETA are calculated only when EPSETA is first time called. The value of EPS and ETA then be saved. The saved value are recalled if EPSETA is called again.

2. The first loop is distributed over the outermost loop and the communication for data collection of array FMO is added in the end of the first loop.

3. The communication code for finding global maximum of variable $TINY$ is added also.

```
S1  :       DO 60 I=LUMO,N
S2  :            KK=0
S3  :            DO 30 J=1,N
S4  :                 SUM=0.D0
S5  :                 DO 10 K=1,J
S6  :                      KK=KK+1
S7  : 10              SUM=SUM+FAO(KK)*VECTOR(K,I)
S8  :                 IF(J.EQ.N) GOTO 30
S9  :                 J1=J+1
S10 :                 K2=KK
S11 :                 DO 20 K=J1,N
S12 :                      K2=K2+K-1
S13 : 20              SUM=SUM+FAO(K2)*VECTOR(K,I)
S14 : 30            WS(J)=SUM
S15 :            DO 50 J=1,NOCC
S16 :                 IJ=IJ+1
S17 :                 SUM=0.D0
S18 :                 DO 40 K=1,N
S19 : 40              SUM=SUM+WS(K)*VECTOR(K,J)
S20 :                 IF(TINY.LT.ABS(SUM)) TINY=ABS(SUM)
S21 : 50            FMO(IJ)=SUM
S22 : 60    CONTINUE
```

Figure 10: Parallel algorithm of DIAG

```
S1  :       L=0
S2  :           DO 40 I=1,NORBS
S3  :           DO 30 J=1,I
S4  :               L=L+1
S5  :C          Some statements here
S6  : 30        P(L)=(SUM2+SUM1*FRAC)*SIGN
S7  : 40    P(L)=CONST+P(L)
```

Figure 11: DENSIT looping structure

4. The second loop is distributed over the innermost loop. However the data collection code is not added immediate after the innermost loop but the end of the outermost loop because there is no data dependency for array $VECTOR$ if we do so.

5. We overlap communication with computation in first main loop. The computation of second main loops can not be overlaped because of the data dependence described above.

3.4 Parallelizing subroutine DENIST

Subroutine DENSIT computes the density matrix given the eigenvector matrix, and information about the M.O. occupancy. It takes about 15% of total CPU time in our test sets.

Program analysis First we check if the size of the parallelable segment can be extended to the caller of DENSIT. DENSIT is called in subroutines ITER and MULLIK. There are 3 calls to DENSIT in ITER and one in MULLIK. Like DIAG, all calls to DENSIT are surrounded by if structures instead of loops. We can not extend the size of the parallelable segment to the caller of DENSIT.

DENSIT does not call any other subroutine. It contains an if structure and only a set of loops. The if structure just sets some boundary variables according to the mode parameter. The computation is in the 3-level loop. Figure 11 is the simplified program of the loop structure of DENSIT.

The two innermost loop are just accumulating variable SUM1 and SUM2. We do not consider them at this time. The range of loop S3 depend on the index of loop S2. The outer two loops S2 and S3 forms a triangular computational space. The variable L is the index of the triangular computational space. Variable L is assigned 0 in the beginning but increase one immediate enter the loop. The index L actually starts as 1. As described in previous section, we can rewrite the code into the algorithm shown as figure 12

```
S1  :       DO 50 L=1,NORBS*(NORBS-1)/2
S2  :C          Some statements here
S3  :           P(L)=(SUM2+SUM1*FRAC)*SIGN
S4  :           IF(J.LT.I)THEN
S5  :               J=J+1
S6  :           ELSE
S7  :               P(L)=CONST+P(L)
S8  :               J=1
S9  :               I=I+1
S10 :           ENDIF
S11 : 50    CONTINUE
```

Figure 12: Parallel algorithm of DENSIT

Data dependency The only modified non-local data is the 1-D array P which contains the packed density matrix. There is no data dependency in the rewritten DENSIT loop. It is clear and easy to distribute over loop S1 because it is only one level and array P uses loop index L as its data index purely.

implementation It is easy to distribute the re-written 1-D loop. However, we need to keep variable I and J for statement S2. We need a translation from index L to indices I and J.

Let's see the relation of I, J, and L. L is total count of loops. Since the computational space is triangular, the formula of L should be:

$$L = (I-1) \times I/2 + J \quad (1)$$

The range of inner loop is from 1 to I. The index of inner loop should be less or equal to I.

$$0 < J \leq I \quad (2)$$

From (1) and $J > 0$, we have:

$$(I-1) \times I < 2 \times L$$
$$(I-1/2)^2 < 2 \times L + 1/4$$
$$I < SQRT(2 \times L + 1/4) + 1/2$$

Since I is an integer, the final formula for finding I from L is:

$$I = INT(SQRT(2 \times L + 1/4) + 1/2) \quad (3)$$

and the formula for finding J from L and I is:

$$J = L - (I-1) \times I/2 \quad (4)$$

By the above formulas (3) and (4). We can distribute the work load over main loop by

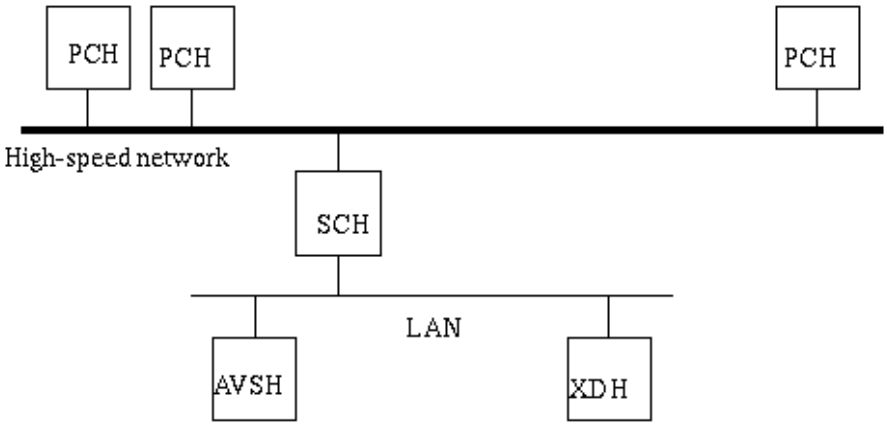

Figure 13: The MOPAC executiong environment

dividing the range 1 to $NORBS \times (NORBS-1)/2$ to all computation nodes and calculate the start indices of I and J from distributed loop index L.

Since there is no data dependence for array P under index L, we can overlap communications with computations.

4 Running MOPAC

4.1 Environment

The environment for running MOPAC can be divided into 4 parts.

- PCH (Parallel Computation Host) : a parallel computer which handles the computation of time comsuming modules. For portablity reason, we use PVM communication library.
- SCH (Sequential Computation Host) : a sequential computer which handles file I/O, data distribution, and other sequential modules.
- AVSH (AVS server) : an AVS server which handles graphic output and controls the execution of modules.
- XDH (X-windows Display Host) : a color workstation or X-terminal which displays the graphic output.

The structure of the 4 components are shown in figure 13.

4.2 Preparing MOPAC

MOPAC directory tree looks like figure 14

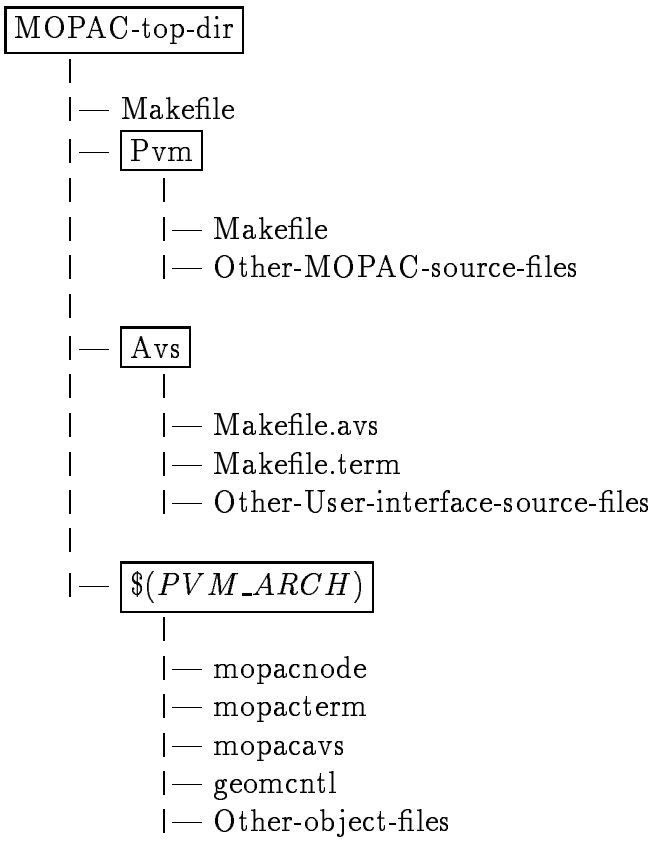

Figure 14: MOPAC directory structure

The top level MOPAC directory tree contains a top-level Makefile, subdirectory Pvm, Avs, and a working directory. Subdirectory Pvm contains the MOPAC parallelized code. Subdirectory Avs contains the user interfaces of MOPAC. We offer two user interfaces. One is for AVS display and the other is for non-graphic terminals. The working directory, which is named after the environment variable $\$(PVM_ARCH)$, is used for compiling the above source files. For example, if you are using Suns, then the working directory will be SUN4. This naming scheme allows MOPAC to be installed on different platforms without conflict.

We support two versions of MOPAC user interface. One requires AVS to display graphic atom structures, the other does not require AVS and can run MOPAC on a regular terminal.

Four executables will be generated in the working directory.

- mopacnode: The parallelized kernel MOPAC program.
- mopacterm: The MOPAC user interface for terminal users.
- mopacavs: The MOPAC user interface for AVS users.
- geomcntl: The atom control module for AVS MOPAC user interface.

4.3 Running MOPAC

We have two different user interfaces for users with and without AVS. AVS is a visual system which supports nice graphic display. The AVS MOPAC user interface will show the atom geometric when MOPAC is running. For users who do not have AVS, the terminal version of MOPAC save the atom structures in a disk file in text format.

Both user interfaces require PVM since the distributed tasks use PVM to communicate with other tasks. PVM supports heterogeneous configurations. Our MOPAC also allow heterogeneous configurations. Currently, we supports Sun4 running SunOs, RS6000 and SP2 running AIX, Alpha running OSF/1, and [345]86 IBM PC running Linux or BSDs. You need to configure the participating hosts and start PVM before you run MOPAC.

4.3.1 Running MOPAC through AVS user interface

Our user interface is divided into two modules, mopacavs and geomcntl. Module mopacavs is the real user interface. It serves as the controller and server of the parallelized MOPAC tasks. It is also the bridge between MOPAC and AVS. It receives the internal atom geometric information and send them to module geomcntl. Module geomcntl transforms the atom geometric information into AVS geometry objects. The AVS build-in module geometry view takes the objects and displays them on then screen.

To make the modules work, you need to connect them into an AVS network. We have save the necessary connections into a disk file. You can click on the network tool menu to get it activated or you can do it manually. The network shown in figure 15 will appear on the AVS network editor panel.

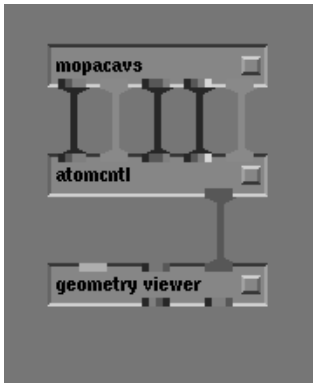

Figure 15: AVS module network

Figure 16: MOPAC execution control pannel

After the network has connected, a blank geometry viewer window and a MOPAC execution control panel will pop up. The MOPAC execution control panel shown in figure 16 asks for the input file name and number of tasks you want to distribute. The default number of tasks is 0 which means the MOPAC program will detect the number of participating machines and distribute one task for each participating machines.

When MOPAC reads in a new atom geometry or changes an existing one, the atom geometry will be displayed in the geometry viewer window. User can use the MOPAC geometry control panel to scale, move, or rotate the atom graphic object.

For data which generates multi atom geometries, module geomcntl keeps all atom geometries. Users can view any one by clicking on the MOPAC geometry control panel. User can also click on the "slide show" button on the MOPAC geometry control panel to view all atom geometries one after another.

4.3.2 Running MOPAC through terminal user interface

Although AVS display is attractive, it is not possible for users who has no color X display and AVS. For those who can not run MOPAC through AVS user interface, we have a terminal

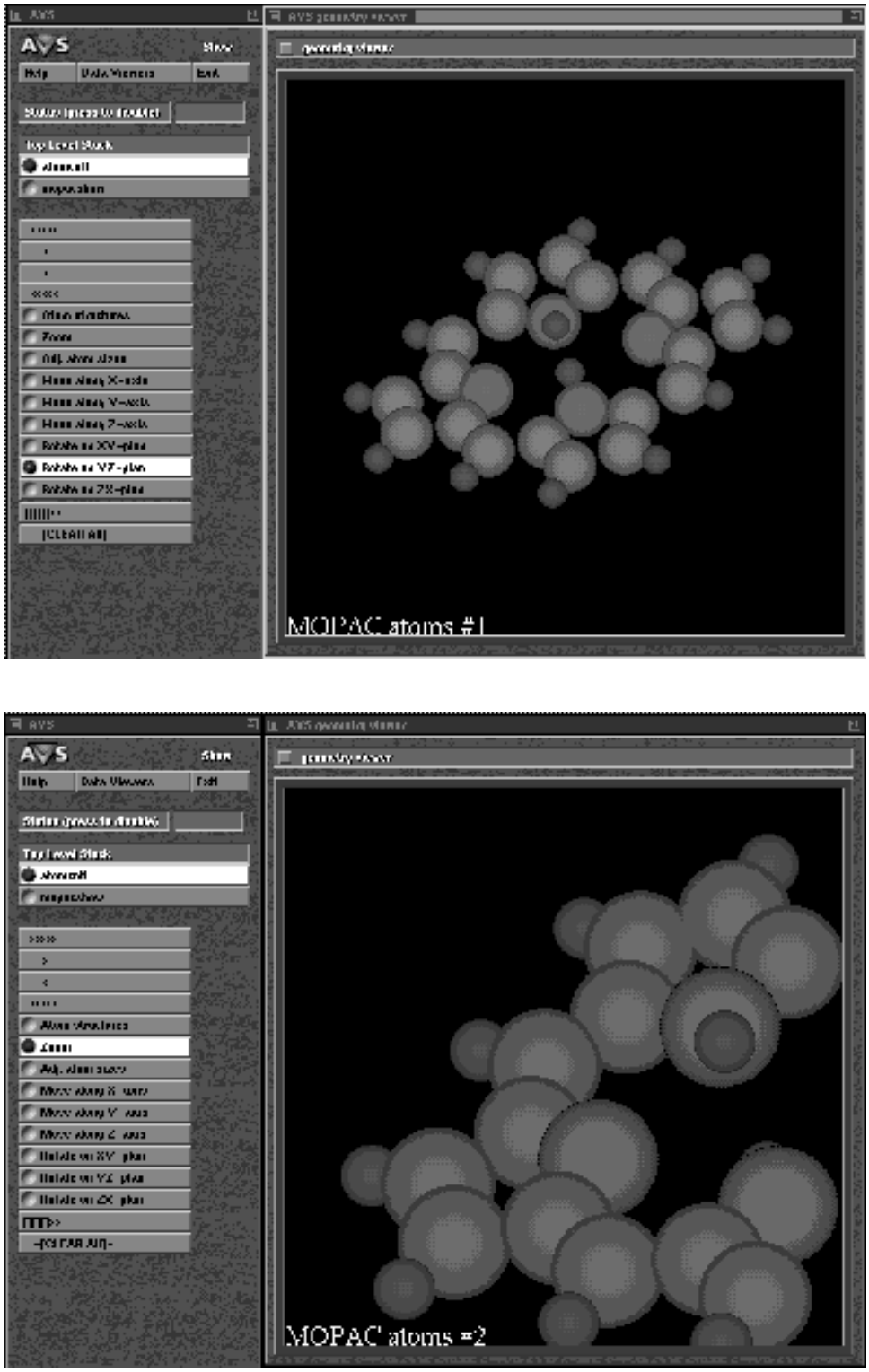

Figure 17: MOPAC atom structure examples

	apsbtest	porphin	tetrabenz
Tp	85.86%	89.68%	92.74%
$P = 2$	1.75	1.81	1.86
$P = 4$	2.81	3.05	3.28

Table 10: The maximum speed-up of the test set

user interface. All important MOPAC results are written into disk files. Users still can examine results by checking those files. The only difference between the two user interfaces is the graphic display. The atom geometry information is saved into a file. A tool program is developed to read the file and feeds the data into AVS to display graphic atom structures.

4.3.3 Some results

It is difficult to measure the real performance of parallel programs on a workstation cluster. You need exclusive access to those workstations and the network. We'd like to know how much speed-up can be achieved by our parallelization.

Parallelization usually need to add some code to handle distributing work and data. It will kill some speed-up. Communication is the most terrible speed-up killer. We gain more speed-up by using more computation nodes but the communication time raises at the same time. It limits the speed-up we can achieve significantly.

Theoretical speed-up The maximum speed-up we can achieve depends on the portion of parallelized code. The more the portion of parallelized code, the higher the speed-up. Suppose the time needed for parallelized code before parallelization is Tp, the time needed for sequential code before parallelization is $(1 - Tp)$, the number of computation nodes is P. The maximum speed-up is:

$$Speed - up = \frac{1}{(1 - Tp) + \frac{Tp}{P}}$$

Single node PVM can spawn processes on one computer to simulate the parallel environment. The only difference between the multi-processes-on-one-machine and one-process-per-machine is that the former uses UNIX inter-process communication instead of real network. The computation time is the same and the communication time should be faster due to the UNIX inter-process communication is faster than the real network. The major difference should be the synchronization time. CPU won't stay idle to wait for data. It switches to the one who lagged behind instead. Timer won't count during the time CPU switch to another processes. Thus no synchronization time is counted. The results will look better than they should. We will run full benchmarker as soon as all four kernel subroutines have been parallelized.

	apsbtest			porphin			tetrabenz		
	Seq	Para	Speed-up	Seq	Para	Speed-up	Seq	Para	Speed-up
merlin(SP-2)	2382	930	2.56	363	139	2.61	1421	487	2.91
mel(550)	6086	2507	2.42	980	372	2.63	3685	1292	2.85

Table 11: Timing of running MOPAC on RS6000s in simulation mode

	apsbtest			porphin			tetrabenz		
	Seq	Para	Speed-up	Seq	Para	Speed-up	Seq	Para	Speed-up
$P = 1$	31672	31672	1.00	4869	4869	1.00	19007	19007	1.00
$P = 2$	31672	20125	1.57	4896	3123	1.56	19007	10804	1.76
$P = 4$	31672	17204	1.84	4896	2533	1.93	19007	8588	2.21

Table 12: Timing of running MOPAC on RS6000s

Table 11 shows the results of running MOPAC on two kinds of IBM Rs6000s. We spawn 4 processes, that means we simulate a 4 node system. The following table shows the time (in seconds) of running sequential version, the time (in seconds) of running parallel version and the speed-up.

From the table in previous section, our results are very close to the expected speed-up.

Real network We also run the parallelized code on 4 Sun4 SLCs connected by Ethernet. Table 12 shows the results. Although we run our experiences after mid-night, the workstations still have some light load and the network is still used by some other machines. Thus the speed-up will be lower than above.

5 Future work

	apsbtest	porphin	tetrabenz
Tp	96.08%	91.40%	94.26%
$P = 2$	1.92	1.84	1.89
$P = 4$	3.58	3.18	3.41
$P = 8$	6.28	4.99	5.71

Table 13: The maximum speed-up of the test set with HQRII

1. Re-write subroutine HQRII into a parallel diagonalisation routine. After parallelizing HQRII, the theoretical speed-up should be raised as shown in table 13

2. We use only 3 test data to choose the most time consuming subroutines to parallelize. We should check if other subroutines are also good for parallelization.

3. The Sun4 experience speed-up gets smaller than expected. The problem maybe on network. Run some experiences on machines with better network, e.g. SP2, to see if the speed-up won't reduce so fast.

4. Run experiences on a bigger machine to see how many computation nodes is the best configuration for workstation cluster parallel machine.

5. Display more visual data.

References

[1] Frank J. Seiler Research Laboratory, United States Air Force Academy, CO 80840. *MOPAC Manual*, dec-3100 edition edition, December 1990.

[2] Sanjay Ranka, Jhy-Chun Wang, and Geoffrey C. Fox. Static and Runtime Algorithms for All-to-Many Personalized Communications on Permutation Networks. In *Proceedings of the 1992 International Conference on Parallel and Distributed Systems*, pages pp. 211–218, HsinChu, Taiwan, December 1992.

[3] Jhy-Chun Wang, Tseng-Hui Lin, and Sanjay Ranka. Distributed Scheduling of Unstructured Collective Communication on the CM5. In *Proceedings of the 27th Hawaii International Conference on System Sciences*, Hawaii, January 1994. To appear.