Theor Chim Acta (1993) 84: 443–455

Theoretica Chimica Acta

Network supercomputing: A distributed-concurrent direct SCF scheme

Hans P. Lüthi[1] and J. Almlöf[2]

[1] Interdisciplinary Project Center for Supercomputing, ETH Zürich, CH-8092 Zürich, Switzerland

[2] Department of Chemistry and Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, MN 55415, USA

Received October 1, 1991/Accepted April 15, 1992

Summary. The direct SCF algorithm has been parallelized such that the computation can be split over a number of computers. The server ("slave") machines use all their processors in parallel to execute the tasks distributed by the client ("master"), a mode of operation best described as *distributed-concurrent* parallel computing. Relatively high speedups resulting in performance of several GigaFLOPS have been demonstrated in realistic applications with clusters of Cray computers in dedicated mode. Since the communication and synchronization requirements are modest, machines that are accessible via communication services like Internet can be networked. GigaFLOPS performances during regular production hours have been obtained when combining computers located at different centers, even if these were on both sides of the Atlantic.

Key words: SCF – Distributed-concurrent parallel computing

1 Introduction

For numerically intensive applications, distributed computing is an interesting option from several points of view. Combining a network of powerful workstations is one way to achieve computation at high speed for relatively low cost. At the other end, connecting a number of supercomputers to work on the same job is an obvious way to address the need for extreme performance. Such an approach will allow calculations which are too large to be executed on any single computer, even if all processors of that machine were used.

Recently, the direct SCF method as implemented in the program package DISCO [1] has been parallelized to obtain GFLOPS performances on single, shared memory type computers such as the Cray Y-MP/8 [2, 3]. Parallelism is achieved by splitting the computation of the two-electron integrals and their processing to form a Fock matrix into a number of tasks that can be executed independently and at random order. This coarse-grain implementation of parallelism is not limited to execution on one machine only. Tasks can be distributed across several computers, each one executing a (partial) copy of the program. Once all tasks have been issued, the client (master) machine collects all partial

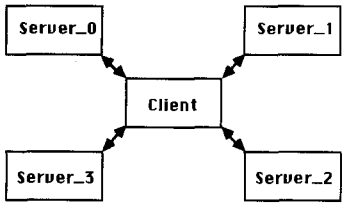

Fig. 1. Connectivity graph for the client-server network. The servers are depicted as simple units here, but may in practice be shared-memory multiprocessor machines

Fock matrices from the servers (slaves), combines them into one and concludes the current SCF iteration.

To communicate data and messages between client and servers, a medium such as a communication network, a network file system (NFS) or a shared memory device has to be available. The latter technique has been used by Clementi and coworkers for the loosely coupled array of processors (LCAP) systems (1983 and later), the earliest successful attempt to distribute a computation over several machines [4, 5]. NFS and local area network based schemes have been implemented more recently (see e.g. [6, 7]).

In the present paper we describe an implementation of the direct SCF method for a cluster of computers connected through a network based on the TCP/IP protocol suite. To enable the communication between client and servers, the Sciddle communication system [8] has been used. All data or messages are broadcast from the client to the server using socket connections. The servers do not perform any Input/Output operations other than network communication, and ideally the server machines are shared memory parallel computers that process the task-packages distributed by the client using all their processors in parallel (concurrently), a mode of operation best described as distributed-concurrent computation. Since the servers do not communicate with each other, we obtain a star-shaped connectivity graph between client and servers (see Fig. 1).

A relatively small amount of data is transferred during the calculation, and thus the network communication speed need not be extreme. For this reason the servers do not have to be geographically close to the client machine, but can be located at different computer centers and linked through a communication service like Internet.

The scheme presented here allows a number of computers of different types to be linked in a heterogeneous network. The system is scaleable, with an additive performance increase whenever a new computer is included in the network. We show that GFLOPS performance for direct SCF calculations can be obtained on a network of computers during regular service hours. In dedicated mode, a peak performance of more than 3 GFLOPS has been measured on a network of Cray Y-MP computers with a total of 20 processors.

2 The communication environment

Central to any distributed application is the interface between client and servers. Here, the Sciddle system [8] has been used for the communication environment. Sciddle provides a small library with the most important communication primitives, and allows the programmer to call procedures which are running on the servers as if these were ordinary subroutine calls. For each server procedure, an interface description which labels the variables of the argument list (data-type,

direction (in/out) and arraysize) has to be provided. The Sciddle compiler or preprocessor translates this interface description to standard C code which can be compiled and linked directly into the application program.

For the communication between client and server(s), Sciddle uses an implementation of *non-blocking* remote procedure calls (RPC's). The advantage of this implementation is that the calls to the servers are asynchronous. As soon as the client has finished writing into the socket, it can immediately start another communication with another server. The client is not bound to wait for the server to complete the procedure as in a regular RPC.

To run an application, a server process is started on a number of remote machines. The servers create a socket with a port number attached to it. The client process which is started last requests connection to the servers. When calling a server-routine, the client writes the procedure reference number as defined by Sciddle into the socket followed by the argument list. The server who is listening at its socket will notice the arrival of information. It will decode the procedure reference number to properly accept and execute the call. When completed, the server returns the response requested in the designated format.

The client on the other hand, after having called the servers, issues a syncronization (wait) command implemented in the Sciddle library, and then goes to sleep. It is woken up by the UNIX kernel as soon as data have arrived in one of the sockets. The client continues to work after having received (i.e. read from the socket(s)) the response by any one server or by all servers. At the end of the computation the client disconnects the servers.

The socket communication itself, however, is sequential: only one client-server or server-client communication can take place at a time. In the present implementation of Sciddle, the client has to wait till a server has finished the transmission of the data, or, in the opposite direction, till the server has read all data (except the last buffer) from its socket. The user determines the sequence the servers are addressed. This assures, at least for the start of the computation, that in networks with vastly different communication bandwidths no "communication backlog" can occur (see also Fig. 3 later in the text).

3 Shared memory parallel direct SCF calculation

The structure of the direct SCF algorithm is illustrated by the following schematic diagram:

```
loop over groups of basis functions on atoms A
     loop over groups of basis functions on atoms B
          loop over groups of basis functions on atoms C
               loop over groups of basis functions on atoms D

                    call TWOEL       !  compute the two-electron
                                     !  integrals   (A B|C D)
                    call FOCK        !  update the Fock matrix with
                                     !  the current batch of integrals
continue
```

This suggests a natural definition of a "task" as the calculation of a batch of integrals (A B|C D) and the corresponding update of the Fock matrix with the contributions from those integrals. The resulting parallel region thus comprises the routines TWOEL and FOCK above. The calculation of the two-electron

integrals in TWOEL can be arranged such that they can be evaluated independently without loss of efficiency. The processing of the integrals in routine FOCK, however, can lead to data dependencies, since an integral can access Fock matrix elements which may be referenced at the same time from another task. This problem can be resolved by having each processor update its own copy of the Fock matrix. At the end of the loops over A, B, C, and D, when all integrals and their contributions to the Fock matrix have been computed, the partial Fock matrices are added into one.

Here, this scheme is implemented in a slightly different form [2]. The loops over the groups of basis functions are unrolled, and the loop indices along with pointers and flags which are set before the call to TWOEL are collected in a look-up table, containing a total of sixteen entries per loop-iteration. In this task-generation step, the call to TWOEL is omitted. TWOEL and FOCK are called from the unrolled loop which now goes over all tasks.

In a typical application, more than 98% of the work done in a direct SCF computation is performed in this section of the codes. For the remaining portions of the calculation, parallel processing can very often be obtained by using the mathematical library routines of the system.

The performance obtained, however, critically depends on how balanced the load of tasks is. Depending on the application, the size of the tasks generated can be quite uneven, often differing by one or two orders of magnitude. This may cause an unbalanced load, in particular in cases where the big tasks are executed last, thus leaving processors idle from the moment where the number of tasks left to be processed is less than the number of processors available.

For this implementation of parallelism very high speedups resulting in performances well beyond 1 GFLOPS have been observed. In an SCF calculation on bis-(2,6-dimethylphenyl)carbonate e.g., a 38 atom organic molecule with no symmetry, using a 6-311G basis set (314 contracted basis functions), more than 1.5 GFLOPS with a speedup of 7.65 on an eight processor Cray Y-MP have been measured in dedicated mode. The wall-clock to CPU time ratio was 7.91, a value which reflects the efficiency of a CPU-bound method like direct SCF when running in parallel mode: 945 sec wall-clock time were needed to complete 7515 sec of CPU time for a direct SCF calculation of this application.

4 Distributed-concurrent direct SCF calculation

4.1 Concept

Since the tasks generated by the program can be executed independently and at random order, they can in principle be processed simultaneously by a network of machines. Embedded in the Sciddle communication environment, the client process initiates the calculation, sends out the tasks to the server processes, and finally accepts the partial results returned from the servers. Several implementations of this scheme are possible. The simplest one would be to give each server a copy of the entire program, a copy of the input, and a list of the tasks to be performed. We did choose a somewhat different concept which is described below.

The basic idea is to have the servers as compact as possible, and to implement them as pure compute-servers. All Input/Output operations are performed by the client (no files opened on the server machines). The input is read and processed

by the client, and all data are broadcast to the servers using socket communication as implemented in Sciddle. The client does not actively participate in the two-electron computation, and only controls the distribution of the tasks. However, a server process may be started on the same machine that hosts the client.

The servers therefore only need a copy of the "parallel region" of the program (routine TWOEL and FOCK) plus the interface routines for the client/server communication.

4.2 Implementation

Figure 2 shows the structure of the distributed computation as implemented in a prototype version of DISCO. The servers are started first and report the number of processors of their host machine to the client once they are connected. The client processes the input, computes the one-electron Hamiltonian, sets up the look-up table, and forms packages of tasks. At this point the start-up phase of the direct SCF calculation has been completed, and the client initializes the

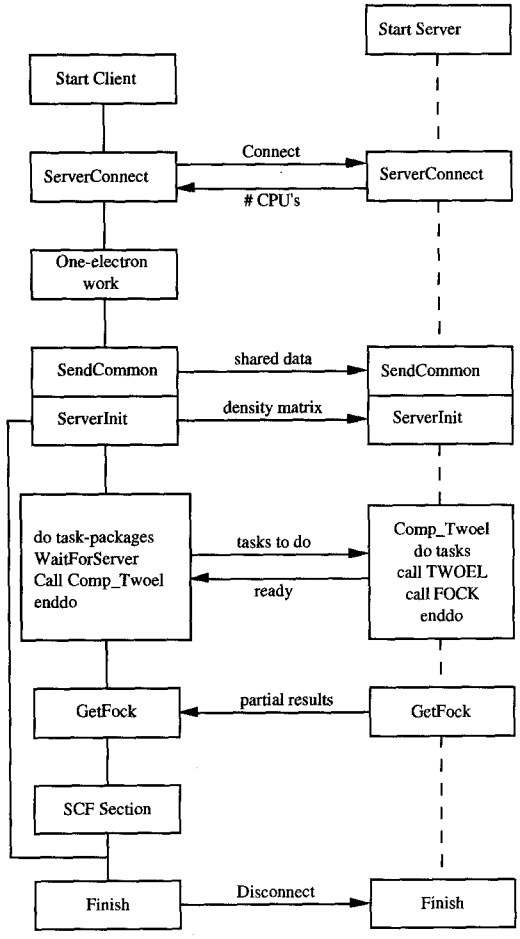

Fig. 2. The structure of the distributed direct SCF computation

servers by broadcasting the contents of the shared variables needed for the execution of their codes. After that, the density matrix, either from an initial guess or from a previous calculation, is broadcast to the servers. When a server acknowledges the receipt of this information, the client issues the first package of tasks for that server.

Once the first server has computed all of its tasks, the client starts to collect the partial Fock matrix from that server. The diagonalization of the Fock matrix is performed by the client. This concludes the present SCF cycle. In the version of the program discussed here, all client program code is sequential (single processor).

4.3 Communication

Like in any other distributed memory system, the message or data passing between client and servers, since it is a serial process, is one of the primary concerns. Since with the present distributed computing system we plan to link computers which are not necessarily connected through high performance communication channels, but also by public networks like Internet, the client-server communication has to be given some attention.

Apart from some handshakes between client and servers, the main portion of communication is in the transfer of the shared variables and the density matrix to the servers, and in the return of the partial Fock matrices to the client. In the examples presented here, both matrices are transferred as full triangular matrices. Once applications with well beyond one thousand basis functions will be performed, the transfer of these data, more than a million words, may turn into a significant process, and solutions will have to be sought where only parts of the matrices are communicated.

Even if broadcast only once, the shared data communication in large applications can be quite substantial. However, a significant fraction of these data can be recomputed by the servers as soon as the unique shared information (atomic coordinates, basis sets etc.) is available. This redundant computation, including the recomputation of the look-up table, takes little time and does not require a significant amount of extra program code. With the present version of the program, less than 10 kwords of shared variable information are transferred in most applications. Only a few words of information have to be transferred with each task-package.

4.4 Load balancing

In this implementation of parallelism where the servers typically are shared memory multiprocessor machines, a two-level load balancing problem is encountered. It is somewhat similar to the one encountered in a three layer tree-structure parallel computer. The distribution of packages of tasks has to be balanced to keep the server idle time at a minimum (first-level load balancing). Similarly, a package should be built such that a multiprocessor server can execute at a maximum internal speedup (minimal server processor idle time; second-level load balancing).

In a high speed network of dedicated multiprocessor computers of the same type (cluster of IBM 3090-600J or Cray Y-MP/8 e.g.), the load balancing problem is not too different from the situation encountered in a single, shared memory

multiprocessor machine. The client will try to give each server an equally big package with a task constellation that minimizes server processor idle times.

Currently, for a network of heterogeneous or non-dedicated machines, the client builds packages of tasks proportional to the number of processors available in the network. Building too many packages may cause communication overhead, and may also result in internal performance degradation for the servers. Too few packages may cause excessive server idle times if the response of the machines in the network varies strongly. The task-packages are issued in an unprejudiced round-robin (or asynchronous queue) fashion: the next available server gets the next task, even if it might be advantageous to skip that server and wait for a more responsive one to become available.

4.5 Task issuance: The "State Machine"

Since the data transfer rates of the communication network as well as the response of the server machines may vary considerably, the client keeps track of the state of each server. Each time a server signals availability, the client knows what state this particular server is in, and what action to take next (transfer density matrix, issue package of tasks, collect Fock matrix). Therefore the fastest server may be executing the first tasks while the slowest server is still busy with the transfer of the shared data.

This asynchronous mode of operation avoids unnecessary synchronization points and keeps the servers as active as possible. It may occur that servers which show poor response will never get to compute a task, since all packages have been processed by the other servers while this server was still accepting data from the client ("Server runaway").

5 Results and discussion

5.1 Network of dedicated machines

On a network of five Cray Y-MP computers (dedicated mode) with a total of twenty processors, a speedup of 16.6 and a performance of 3.3 GFLOPS for application B described in Table 1 were obtained. This cluster consisted of a Y-MP/8, a Y-MP/6, three Y-MP 2E, and was connected via high speed communication channels (FDDI, Hyperchannel). The measurements were taken on machines of the computing facility of Cray Research Inc. at Eagan (Minnesota).

The total communication time to pass data between the client and the servers, approximately 100 kwords per server, was 2 sec. In the unprejudiced round-robin mode, the speedup within the parallel region (generation of the Fock matrix) was 16.2. Adding the 4 sec elapsed time for the diagonalization of the Fock matrix and the conclusion of the SCF cycle, a serial process in this version of the program, the overall speedup is reduced to 15.7. Within the 130 sec elapsed time spent in the parallel region, the two bigger servers idled for 16 and 27 sec, whereas the three two processor servers idled 22, 4 and 0 sec. The resulting CPU idle time integral is 342 sec (17.1 sec per processor).

All task-packages were processed at a very high internal speedup. The worst speedup observed for a single package was 7.64 for the 8 processor server (CPU/elapsed time ratio).

Table 1. Results of the 20 processor (5 servers) network in dedicated mode for example B (Table 2). The data refer to the parallel region (generation of the Fock matrix) only, unless noted. For one complete direct SCF iteration add 4 sec of wall-clock time. (In this version of DISCO the diagonalization of the Fock matrix is a serial process)

Application:	Bis-(2,6-dimethylphenyl)-carbonate; example B
Network:	one Y-MP/8, one Y-MP/6, three Y-MP 2E 20 processors communication: Hyperchannel/FDDI
1 processor:	2,094 sec wall-clock time, Speedup = 1.00 199 MFLOPS
20 processors:	round-robin 130 sec wall-clock time, Speedup = 16.2 3,220 MFLOPS
	Static-packages: 126 sec wall-clock time, Speedup = 16.6 3,300 MFLOPS
	Static packages (complete SCF cycle): 130 sec wall-clock time, Speedup = 16.2 3,220 MFLOPS

For the static distribution mode, where the client gives each server a fixed amount of tasks proportional to the number of CPUs of that server, a slightly better result was obtained. The server idle times, 18 and 5 sec for the two biggest machines, plus 43, 4 and 0 sec for the three smaller servers, indicate that the performance of the cluster is determined by the first-level load balancing also in this mode of task distribution. The CPU idle time integral is 268 sec (13.4 sec per processor), resulting in a reduction of the wall-clock time of 4 sec compared to the computation using round-robin package distribution. The speedup obtained here is 16.6 with a processing speed of 3,300 MFLOPS (16.2 and 3,220 MFLOPS for the complete SCF cycle).

The relatively big server idle times show the need for a client program with a more sophisticated task-scheduling algorithm. Even a relatively simple algorithm will help to improve the performance of the program system quite substantially. In the round-robin calculation, twenty packages containing 1,606 tasks had been formed. The size of the packages was relatively uneven and varied between 85 and 135 CPU sec. Taking statistics about the task size during run-time will allow the client to control the size of the packages formed, and optimize their distribution in the following SCF iterations. The second-level load balancing does not appear to be a problem. DISCO tries to add small tasks ($\langle ss|ss \rangle$ batches of integrals e.g.) at the end of each package to reduce processor idle times, a concept which seems to work relatively well.

5.2 Wide-area networks

The example calculations are defined in Table 2, and the machines involved in the various "wide-area" networks are listed in Table 3. The pyridine example

Table 2. Specification of the examples used in the test calculations. The primitive basis set is from Ref. [9], augmented with diffuse *s* and *p* functions (case B). In case A two direct SCF iterations were performed, including the one-electron part. In case B one complete direct SCF iteration was performed

		A Pyridine (C_5H_5N)
		Basis sets:
C, N		(10*s*6*p*2*d* / 4*s*3*p*2*d*)
H		(5*s*1*p* / 3*s*1*p*)
	180	contracted basis functions
	14706	tasks
	451	seconds wall-clock time
		(Cray Y-MP single processor, dedicated)
		B Bis-(2,6-dimethylphenyl)carbonate ($C_{17}O_3H_{18}$)
		Basis sets:
C, O		(8*s*4*p* / 4*s*3*p*)
H		(4*s* / 3*s*)
	314	contracted basis functions (general contraction)
	32131	tasks
	2098	seconds wall-clock time
		(Cray Y-MP single processor, dedicated)

(example A in Table 2) is the biggest one in a suite of test cases used to improve the functionality of the program system as well as to make the first performance studies. The main objective of these early applications was not to reach impressive MFLOPS rates, but to obtain information about possible communication-bottlenecks or other shortcomings (load balancing, round-robin procedure).

Therefore this example was designed to show a relatively unfavorable communication to computation ratio, and, for this small application, a very big number of tasks. Case B is very similar to the application referenced in Sect. 3,

Table 3. List of the machines which were networked in this experiment, and their location. (MSC = Minnesota Supercomputer Center, NCSA = National Center for Supercomputing at Urbana-Champaign, SDSC = San Diego Supercomputer Center, EPFL and ETHZ = Swiss Federal Institute of Technology in Lausanne and Zürich, respectively

Machine	Label	Site	# Processors
Cray-2/4-512	sc	MSC	4
Cray X-MP/4-64	sf	MSC	4
Cray Y-MP/4-64	uy	NCSA	4
Cray-2/4-128	u2	NCSA	4
Cray Y-MP/8-64	yl	SDSC	8
Cray Y-MP/2-64	cy	ETHZ	2
Cray-2/4-256	c2	EPFL	4

except that it uses a smaller primitive basis. This example performs about 2,100 sec of computation at a communication load of 100 kwords per server and per iteration, whereas example A shows a communication load of 33 kwords within only 10% as many seconds of CPU time.

For example B, using the 6 processor cluster formed by the c2 and the cy, the "Swiss cluster", the best elapsed time observed during normal service hours (weekend) was 666 sec. During the time of this measurement, one foreign job on the cy and three to four jobs on the c2 (all serial jobs) were competing for the CPUs. This elapsed time corresponds to a performance of 640 MFLOPS, and to a performance-equivalent of 3.2 dedicated Y-MP processors. The theoretical peak performance of the cluster for this example would be around 810 MFLOPS, the equivalent of slightly more than 4 Y-MP processors (for this example DISCO is nearly twice as fast on a cy processor than on a c2 processor).

An impressive performance was observed when adding the sc and sf of the University of Minnesota to the "Swiss network" to form a 14 processor cluster. A best elapsed time of 353 sec corresponding to a performance of 1.2 GFLOPS was measured for the bigger of the two applications (example B), during a period with little competition from other users. Performances between 800 MFLOPS and 1 GFLOPS for that cluster and that same example have been recorded routinely.

The servers processed 5 (cy), 3 (c2), 4 (sc) and 4 (sf) task-packages. Unlike in the experiment taken under dedicated conditions, the 16 packages formed here were much more balanced in size (about 133 Y-MP CPU sec each). Figure 3 shows the course of the computation as a function of time ("event log").

A cluster of 6 servers based on two continents with a total of 28 processors was the biggest network set up (see Table 4). In this application, using example A, all machines of Table 3 except the cy were involved. The client process was

Table 4. Performance of several clusters using example run A (pyridine). In parentheses the number of CPUs for each server. The client process is always on sc. The CPU availability is the average CPU to wall-clock ratio measured for the execution of the individual tasks

Cluster		CPUs	Wall-clock time (sec)	Speedup	CPU availability
a	cy (1) (Reference)	1	451	1.00	0.99
b	sc (4), sf (4)	8	122	3.70	not measured
c	sc (4), sf (4) u2 (4), uy (4)	16	92	4.90	2.79, 3.28 2.05, 2.00
d	sc (4), sf (4) u2 (4), uy (4) y1 (8)	24	104	4.34	not measured
e	sc (4), sf (4) u2 (4), uy (4) y1 (8), c2 (4)	28	390	—	0.56, 0.54 1.23, 0.90 0.90, 3.03
f	sc (4), sf (4) u2 (4), uy (4) c2 (4)	20	110	4.10	2.67, 3.22 2.08, 2.24 3.81

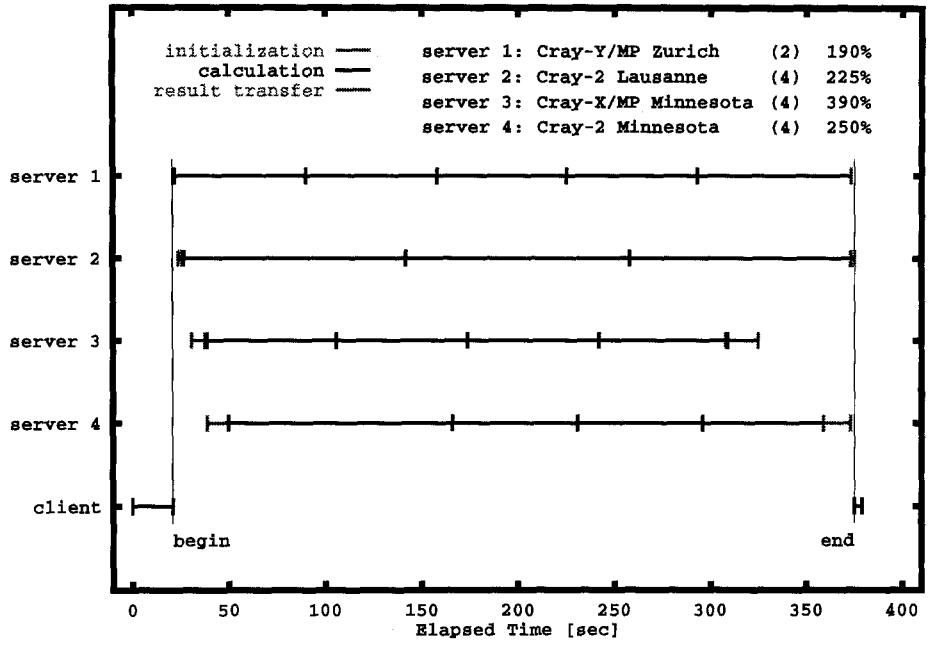

Fig. 3. The event log of the Switzerland-Minnesota cluster for the example-run which yielded 1.2 GFLOPS. In the figure, the client activity also includes the one-electron integral computation. The vertical bars mark the beginning and the end of the distributed computation. The number of processors of each machine (numbers in parentheses) and the response obtained from each server are displayed in the text within the figure (250% indicate a net response of 2.5 processors). The figure shows the nearly simultaneous conclusion of the computation and the transfer of the results by servers 1, 2, and 4 as well as the relative size and the impact of the (sequential) communication overhead

hosted by the sc. For this small example, the best performance (speedup), however, was obtained by combining the University of Minnesota with the NCSA machines (sc, sf with uy, u2), a cluster of 16 processors. The responses of the c2 and the y1 were too limited to compensate for the communication overhead introduced by these two servers.

In this set-up, the response of the c2 was limited by the transfer rate of the socket connection (25 kBytes/sec). The y1, apart from a somewhat slow connection (typically 50 kBytes/sec), showed poor CPU availability due to a very big user-load. The fastest machinery assembled in Table 4, the MSC-NCSA cluster with inter-site transfer rates of 60 to 100 kBytes/sec (same as the "Swiss cluster"), even on an example with a poor computation to communication ratio, showed performances which would be difficult to obtain under equivalent conditions (regular service mode) from an eight processor Cray Y-MP.

6 Extensions and future developments

At this point of development, the key factor that determines the performance of the supercomputer network is the first-level load balancing. The measurements under dedicated conditions show that the internal server performance does not

appear to be an issue with the present implementation. The data also show that measures like asynchronous communication (State-machine) and redundant computation of data on the server side have drastically reduced the communication overhead in DISCO. Before that, performance degradation due to load-balancing effects was hidden behind the communication overhead.

The size of the communication is now determined by the transfer of the density- and Fock matrices. Molecular symmetry can be exploited to block both matrices, such that only the lower triangle of each block has to be transferred. The packages of tasks may also be generated such that only certain sections of both matrices are referenced. This approach, however, might be more difficult to implement as it will collide with load balancing issues.

In future implementations of the program system, the client program will try to reduce sever idle times based on more sophisticated task-scheduling techniques combined with the generation of different or even flexible size packages (large packages in the beginning, smaller packages towards the end of the computation). This scheduler, based on the statistics it takes at run-time, tries to make predictions on the server return times, and distributes the packages accordingly. The statistics taken will also help to generate task-packages which allow to achieve optimal second-level load balancing in the subsequent SCF iterations. These measures will help to increase the speedups observed quite substantially.

Another issue that will have to be addressed is fault tolerance. The client should be able to redistribute work that has been issued to a server that failed to complete (connection problems, server failures etc.). A server that starts to approach its CPU time limit should stop accepting more tasks. Operating a (wide area) network of computers requires substantial infrastructural work from the programmer to obtain a comfortable degree of automation. It is quite unlikely that system software which removes this problem from the user-level, like in local area network (LAN) distributed computing, will be available in the very near future.

Connecting machines of different vendors will be a straightforward matter as long as these support socket connections and as long as their performances are within the same order of magnitude. To resolve incompatible data representations between systems, Sciddle uses the ISO/OSI XDR protocoll.

The present work demonstrates that several computers can be networked with near-additive performance increase in direct SCF calculations. Many computer centers operate several high-performance computers with compatible systems and architectures, and it may therefore be very attractive to network these.

The main difficulty when connecting machines of different centers is not in the communication, but in the fact that these machines may be operated and administrated in different modes which make them difficult to combine. Large production runs will have to be executed in batch mode. This, however, requires that the batch-queue systems are set up in a compatible way, and that the servers get into execution within a relatively small period of time. Combining 5 to 6 machines from 3 or 4 different centers (clusters *e* and *f* in Table 4) is not representative for normal production operations.

The greatest impact of this work will therefore be from combining networks of workstations and high-performance computers operated by the same center. Networking computers based at different sites, but operated in compatible modes, is an attractive option to perform extreme (non-routine) applications.

Acknowledgements. This work was supported by the Swiss Science Foundation (SNF), the National Science Foundation (NSF), and the Army High Performance Computing Research Center (AHPCRC). The distributed calculations have involved the following computer centers: ETH Zurich, EPF Lausanne, MSC Minneapolis, NCSA Urbana-Champaign, and SDSC San Diego. Generous grants of computer time, as well as excellent technical support from personnel at these centers have been instrumental for the successful completion of this project. The authors would also like to thank Cray Research Inc. for providing access to a network of dedicated machines.

References

1. Almlöf J, Faegri Jr K, Feyereisen MW, Fischer T, Korsell K, Lüthi HP: DISCO, a direct SCF and MP2 code. For reference, see e.g. Almlöf J, Faegri K, Korsell K (1982) J Comput Chem 3:385
2. Lüthi HP, Mertz JE, Feyereisen MW, Almlöf J (1992) J Comput Chem 13:160
3. Cray Research Inc. (1990) Gigaflops Awards, page 25. Computerworld, Vol. XXV, No. 26 (July 1, 1991), page 59
4. Clementi E (1990) Modern techniques in computational chemistry. Epson Science Publ, 1990, Chap 1: Folsom D, ibid, Chap 27
5. Clementi E (1988) Phil Trans R Soc Lond A 326:445. For an early application of the LCAP system: Detrich JH, Corongiu G, Clementi E (1984) Chem Phys Lett 112:426
6. Brode S, Ahlrichs R: A distributed implementation of TURBOMOLE(DSCF) (private communication). The TURBOMOLE system is described in: Ahlrichs R, Bär M, Häser M, Horn H, Kölmel C (1989) Chem Phys Lett 162:165
7. Harrisson RJ, the TCGMSG message passing system, private communication
8. Sciddle adheres to the client/server model, and is based on Berkeley 4.3 BSD sockets. For reference see: Arbenz P, Lüthi HP, Mertz JE, Scott W, Intl J High Speed Computing, in press
9. van Duijneveldt FB (1971) IBM Res Reports RJ 945