Theor Chem Acc (1997) 97:211–226

Theoretical Chemistry Accounts
© Springer-Verlag 1997

Network computing: the Hartree-Fock calculation as a model

Hans Peter Lüthi

Swiss Center for Scientific Computing, ETH Zentrum, CH-8092 Zürich, Switzerland

Received: 2 February 1997 / Accepted: 24 March 1997

Abstract. Distributed computation over local and wide-area networks is gaining importance and may soon become the primary means in high performance computing. The development in all areas of scientific computing is closely coupled with the development of efficient application software that fully utilizes the power of the modern computer resources. In numerical quantum chemistry, the adaptation of method and program development to parallel and distributed-parallel computing has shown remarkable results. The contributions of Jan Almlöf in this area of research are briefly reviewed in this paper. The integral-direct Hartree-Fock calculation (zeroth, first and second derivatives) will be used as a model to investigate network-computing techniques and paradigms.

Key words: Parallel algorithms – Gradients – Force constants – Direct methods – Network computing

1 Introduction

Solving chemical problems by computation has been the motive of a whole generation of quantum chemists, among them Jan E. Almlöf. In the early 1970s, at the beginning of Almlöf's career, it was clear that most problems where a theoretical-computational approach could have an impact would require a *quantitative* description of molecules consisting of hundreds of atoms, a goal which at the time seemed to be remote. Still, it was not unrealistic to believe that one day this goal could be reached. The advances in computer hardware as well as in method and algorithm research promised rapid growth of the computational capabilities.

Years later, in his lecture notes from the tutorial on "Methods of modern Hartree-Fock theory" given at the European Summer School in Quantum Chemistry [1], we find the following quote:

However, the rapid progress in computer hardware of the last decades, combined with the even faster refinement of computational methods during the same period, has made [*the quantitative description of molecules in the range of* 10^2 *to* 10^4 *atoms*] *a very realistic objective which could well be reached before the end of this century.*

Between the early days and the moment the above statement was made, enormous progress had been made in computer technology, numerical algorithm research, and, most of all, in the development of quantum chemistry methods. There is, however, a close interaction among these three disciplines; the adaptation of software development to the realities created by computer hardware is an interesting process. In his paper "Direct methods in electronic structure theory" published in the monograph "Modern electronic structure theory" [2], Almlöf reminds us that "[*the*] *restructuring of application software to fully utilize the awesome powers of current computer hardware is one of the most significant challenges facing contemporary computational chemistry. Considering the wide variety of available computer architectures, and the ephemeral nature of the cutting edge technology upon which they are based, this is not a one-time task, but rather an ongoing development project.*"

In computational quantum chemistry, one of the most important examples of this adaptation process is the development of the *direct methods* initiated by Almlöf [2–4]. For reasons of computational expense, the Hartree-Fock calculation had traditionally been implemented as a two-step process, with the generation and the processing of the integrals treated as separate events. The rapid advances in central processor unit (CPU) technology, with a doubling of the floating-point processing performance every 18–24 months, and the much slower progress in the development of fast input/output capabilities, changed this perspective. This trend had already been foreseeable in the late 1970s, and the break with the established approach by treating data generation and data processing as one single process was the obvious consequence.

Accepted with some scepticism initially, the direct methods have turned out to be a powerful computational strategy, and are now also being used for correlated methods (see e.g. [5–7]). With the computer architectures available today, it appears that only input/output-free (i.e. direct) methods will grant scalability. Although parallel input/output technology will see dramatic improvements since it is driven by the gigantic demand from business applications, it is unlikely that these developments will breath life back into the classical schemes.

Parallel computing has almost unanimously been praised as the approach to the solution of the "Grand Challenge" class problems. But after about a decade of practical experience, there are still many ways of performing a computation in parallel, and the discipline of parallel computing is not yet very well defined. Still, the adaptation of program development to parallel computers has already inspired major changes in the way methods and algorithms are designed. The focus has shifted away from algorithms allowing high processing speeds, usually measured in floating-point operations per second (flops), to methods capable of delivering performance increases proportional to the processing power increases of massively parallel computers, i.e. methods that show the property of *scalability*.

Despite the fact that massively parallel computing entered the Teraflops era [10], the world of computing has become less monolithic, and the "supercomputers" no longer constitute the main high-performance computing hardware platforms. Instead, most researchers have access to a variety of hardware over the network. These usually represent a diversity of properties (CPU power, memory, input/output capacity; architecture, programming model). It is evident that network technology has changed the way we perform computations, and the phrase "The network is the computer" appears to be more than just a (trademarked) slogan of a leading workstation manufacturer [11]. As a matter of fact, network computing is gaining importance and may soon become the primary means in high-performance computing.

In this paper we will focus on network computing as an approach for the evaluation of molecular properties at high efficiency. We will try to illustrate the benefit of distributing computations over networks of compute servers from the point of view of parallel computing, method development, and software engineering. Again, we will see examples of beneficial interactions of method development with hardware realities. Network computing is an area of interest shared with Jan Almlöf, and this paper, dedicated to his memory, is also a review of one of our most exciting projects. The concepts presented here are implemented in the Disco and/or Supermolecule codes [8, 9].

In the following section schemes for the parallelization of the Hartree-Fock calculation (energy, gradients, and force constants) are reviewed. The focus will be on schemes which can be generalized and which will be useful in network computing. In chapter 3 the tools needed to distribute a computation over a network of computers are presented, and in chapter 4 the results obtained are reviewed and some future developments addressed.

2 Parallel Hartree-Fock algorithms

There are a number of ways to parallelize the Hartree-Fock calculation, and the optimum choice critically depends on the compute resource targeted. In the present work, the focus will be on network-computing systems of the client-server type. The servers may be heterogeneous, and range from large-scale massively parallel processors (MPP) via shared-memory parallel computers (SMP) to single processor workstations.

2.1 Why Hartree-Fock theory?

It may seem that the progress in method development will make the Hartree-Fock method obsolete in the not too distant future. Such a development certainly looks very plausible for *applications* using traditional implementations of the method (i.e. explicit calculation of the Coulomb and exchange repulsion). For the computation of derivatives of the energy such as force constants or hyperpolarizabilities, the Hartree-Fock method, at least in the short term, will maintain its position. The method is well understood, and the wealth of experience gathered allows for the definition of empirical correction procedures such as the "scaled quantum mechanical (SQM)" force field technique [12].

From a *method development* point of view, however, the Hartree-Fock wave function and energy are the basis for virtually all electron correlation methods based on atomic orbital schemes. The fact that method development is shifting towards correlated methods with linear scaling properties will not change this. Implementations of correlated methods based on localized orbitals, for example, require a Hartree-Fock first-order density matrix for the formulation of the space of occupied and correlating orbitals (see, for example, the local MP2 method by Pulay and Sæbo [13]).

Finally, from a *parallel computing* perspective, the Hartree-Fock calculation is representative of more sophisticated theories due to its use of irregular data access patterns. It also is sufficiently complex to serve as a model for (heterogeneous) network computing.

2.2 Energy calculation

2.2.1 The replicated Fock matrix algorithm

The main task in a Hartree-Fock energy calculation is the generation of the integrals $(\mu\lambda \parallel \nu\sigma)$ and their processing ("contraction") with the density matrix $\mathbf{P}$

$$\mathbf{G}(\mathbf{P})_{\mu\nu} = \sum_{\lambda\sigma} \mathbf{P}_{\lambda\sigma}(\mu\lambda \parallel \nu\sigma) \qquad (1)$$

to the two-electron part of the Fock matrix.[1] The remaining steps, namely the computation of the two-index quantities $\mathbf{S}$ (overlap matrix) and $\mathbf{h}$ (one-electron part of the Fock matrix), the formulation of the Fock matrix $\mathbf{F}$

[1] McWeeny notation [14]

$$\mathbf{F} = \mathbf{h} + \mathbf{G}(\mathbf{P}) \ , \tag{2}$$

the solution of the SCF equations

$$\mathbf{FC} = \mathbf{ESC} \ , \tag{3}$$

the generation of a new density matrix from the eigenvectors of the Fock matrix

$$\mathbf{P}_{\lambda\sigma} = 2\sum_{v}^{occ} C_{\lambda v} C^{*}_{\sigma v} \ , \tag{4}$$

as well as the computation of the Hartree-Fock energy

$$\epsilon = \langle \mathbf{HP} \rangle + \tfrac{1}{2}\langle \mathbf{PG}(\mathbf{P}) \rangle + V_{nuc} \ , \tag{5}$$

are comparatively minor tasks. Therefore, with the exception of the solution of the Roothaan-Hall Eq. (4), these processes will not be addressed in the present discussion.

The pseudocode for the contraction of a general integral $(ij\,|\,kl)$ with the corresponding elements of the density matrix to build the two-electron part of the Fock matrix according to Eq. (1) is displayed in Fig. 1 [(11–22) notation; using permutational symmetry of integral indices].

In the most straightforward implementation, the parallel region consists of (the code for) the evaluation of (a batch of) integrals, labelled *ijkl*, and their processing to elements of the Fock matrix. Each processor updates its private, partial Fock matrix. At the end of the loop these Fock matrices are accumulated to the full (symmetrized) Fock matrix.

This *replicated Fock-matrix algorithm* has led to excellent results on systems as diverse as a Cray Y-MP [15], clusters of workstations [17, 18], or an Intel Touchstone Delta MPP [19].

On the Cray-YMP, in shared-memory parallel mode, performances well beyond 1 GFLOPS on eight-processor machines of that type were observed using the DISCO codes [15, 20, 21]. For an SCF cycle on the unsymmetric 38-atom organic carbonate in Fig. 2, a performance of 1.53 GFLOP per wall-clock second on a Cray Y-MP8/128 was measured in dedicated mode. Using a 6-311G basis set of atomic orbitals (314 contracted basis functions), the computation was split into 157,461 tasks. In 945 wall-clock seconds, 7,515 CPU seconds were performed, corresponding to a CPU to wall-clock ratio of 7.91.[2] This value, not to be confused with the actual speedup,[3] indicates very good load balancing. Feyereisen and Kendall (FK; [19]), using a similar version of the DISCO codes, reported speedups on the 512 processor Touchstone Delta MPP which were linear to about 200 nodes, and which increased to a value of about 300 for 512 nodes.

The problem with the replicated Fock-matrix approach, however, is scalability. At the MPP end, the limits are set by the size of the memory per node. In their benchmarks, with each node of the Touchstone Delta having 16 MBytes of physical memory, FK were restricted to examples smaller than 400 basis functions. For (uniform access) shared-memory parallel architectures such as the Cray Y-MP, scalability is restricted by the small number of processors. Truly scalable algorithms, however, are available for the SCF (see e.g. [22]) and even for the second-order Møller-Plesset computation [23]. These algorithms distribute the data (Fock and density matrix in the SCF case) over all processors of an MPP, and use (high-speed) asynchronous interprocessor communication to ensure uninterrupted computation. Exploiting the large aggregate memory of an MPP, even superlinear speedups can be achieved (see e.g. [23]).

```
loop $i \le n$
    loop $j \le i$
        loop $k \le i$
            loop $l \le k$
                $F_{ij} = F_{ij} + 4\,P_{kl} \ast (ij|kl)$
                $F_{kl} = F_{kl} + 4\,P_{ij} \ast (ij|kl)$
                $F_{ik} = F_{ik} - P_{jl} \ast (ij|kl)$
                $F_{il} = F_{il} - P_{jk} \ast (ij|kl)$
                $F_{jk} = F_{jk} - P_{il} \ast (ij|kl)$
                $F_{jl} = F_{jl} - P_{ik} \ast (ij|kl)$
endloop
```

Fig. 1. Generation of the Fock matrix (closed shell case; *n* basis functions)

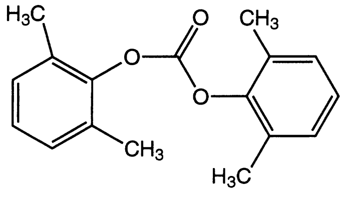

Fig. 2. The molecule used in the first benchmark series: *bis*-(2,6-dimethylphenyl)-carbonate ($C_{17}O_3H_{18}$)

2.2.2 A blocked Fock-matrix algorithm

The data access pattern in the construction of the Fock matrix can be exploited to resolve the memory requirements and to reestablish scalability. An alternative to the replicated Fock-matrix algorithm can be obtained by simply moving the innermost loop index into the parallel region (loop over *l* in Fig. 1). The data access pattern shows locality which can be exploited in several ways.

Foster et al. [24, 25] extensively studied this algorithm and variants thereof. If tasks are generated by keeping the triples of indices (i, j, k) fixed, we find the data access pattern shown in Fig. 3 (canonical loop ordering assumed).

The communication pattern in this scheme which generates $O(N^3)$ tasks shows interesting properties (N is the number of integral batches). If a worker gets to perform a task with a new index k, but with the same outer indices (i, j), only the elements D(k,*) and F(k,*)

[2] When comparing this with later benchmarks using that same example, it was found that due to an input error in the specification of the supershells the outermost s-function of the basis set was treated as a p-function. The $(11s5p/4s3p)$ basis thus turned into a $(10s6p/3s4p)$ type basis

[3] The CPU time recorded includes the parallelization overhead. The estimated "true" speedup for this example is 7.65. See [15] for details

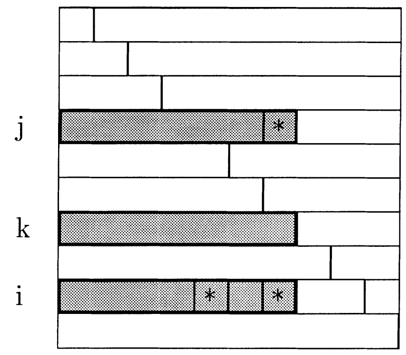

Fig. 3. Data access pattern in Fock and density matrices when keeping the innermost index *l* inside the parallel region (using canonical loop ordering, i.e. $i \geq j$; $i \geq k \geq l$). The elements marked with an *asterisk* are referenced (multiple times) also as scalars. For cases where $k < j$ scalars occur outside the vector elements (*shaded*), but are still part of the same row of the matrix

have to be communicated. The elements D(i,*), D(j,*) as well as F(i,*), F(j,*) can be reused. Therefore, if the load-balancing situation permits, the master may want to schedule as many tasks to the same worker without modification of indices (i, j) to reduce the communication overhead.

It should be noted that in a network-computing scheme, the fact that the workers flush their partial results regularly contributes to the fault resistance of the scheme. It is easier to recover from worker failure than in the replicated matrix scheme.

Foster et al. also show that for an MPP-type environment the performance of this algorithm can be enhanced by switching from a canonical loop ordering to a triple sort method (i.e. loop order $i \geq j \geq k \geq l$ rather than $i \geq j$; $i \geq k \geq l$). This scheme, which requires the computation of three integrals in the innermost loop (i.e. $(ij|kl)$, $(ik|jl)$, and $(il|jk)$), shows a slight advantage with respect to communication, and, more importantly, the symmetrization of the Fock matrix can be avoided. In a network-computing environment, however, we assume that there will be at least one server with the appropriate size memory. Therefore, the option to bypass the $O(n^2)$ memory requirement for the symmetrization of the Fock matrix in the present context is of little interest.

2.2.3 The small-memory parallel strip algorithm

When processing an integral $(ij|kl)$ according to the scheme presented in Fig. 1, for the exchange part, only elements in rows *i* and *j* of the Fock and density matrices are referenced. If we were to invert the loop order from *ijkl* to *ikjl*, we would have the same situation for the Coulomb contributions, in that only elements in rows *i* and *k* of the two matrices would be referenced. By splitting the loop structure into a Coulomb and an exchange pass, and by keeping the two innermost loop indices inside the parallel region (i.e. tasks indexed *ij* in the exchange pass, and *ik* in the Coulomb pass), only rows of the Fock and density matrices have to be communicated for the evaluation of a task.

This algorithm, named *small-memory parallel strip algorithm* (*SMPSA*) by its authors (Almlöf, Sargent and Feyereisen (ASF); [26]), requires two passes over the two-electron integrals. Load balancing may be somewhat more difficult, and there are $O(n^2)$ communication events of size $O(n)$ each.

Unfortunately, ASF do not provide performance data for large applications. The examples they present, only 254 basis functions in size, still show a ratio of 1.84 between the SMPSA and the ordinary (one-pass) replicated Fock matrix algorithm. Also, the degree of parallelism they report for this application apparently shows traces of the communication overhead (see Table 1.2 in [26]).

2.2.4 Splitting Coulomb and exchange contributions

What may look like an "uphill battle" at first sight turns out to be an algorithm with great potential. Having divided the evaluation of the Coulomb and exchange contributions to the Fock matrix, we can now take advantage of type-specific integral pre-screening using the thresholds

$$\kappa_{ij}\kappa_{kl}\max(4|D_{ij}|,\ 4|D_{kl}|) \geq \tau_C \tag{6}$$

and

$$\kappa_{ij}\kappa_{kl}\max(|D_{ik}|,\ |D_{il}|,\ |D_{jk}|,\ |D_{jl}|) \geq \tau_{Ex} \tag{7}$$

rather than the combined criteria for the two types of interaction

$$\kappa_{ij}\kappa_{kl}\max(4|D_{ij}|,\ 4|D_{kl}|,\ |D_{ik}|,\ |D_{il}|,\ |D_{jk}|,\ |D_{jl}|) \geq \tau \tag{8}$$

The estimators κ_{ij} are obtained from

$$\kappa_{ij} = \sqrt{|(ij|ij)|} \tag{9}$$

and D_{ij} represent elements of the density matrix (see also [27] for a discussion of integral pre-screening using the Schwarz inequality).

Due to the local character of the exchange contribution – in a non-metallic system (insulator), the size of an exchange integral decays exponentially with the distance of the centers involved – the exchange cycle is expected to essentially converge to an $O(n)$ step (see e.g. [28]).

However, division of the Coulomb and exchange part also offers the opportunity to use specialized methods or algorithms for both contributions. The Coulomb part can be evaluated using fast multipole methods (FMM; [34]). In a recent paper, Challacombe et al. present a code for the fast assembly of the Coulomb matrix [29]. Their hierarchical multipole method, named "quantum chemical tree code", is shown to scale better than $O(n^2)$ in "systems of chemical relevance". More recently, Schwegler and Challacombe have also presented a near-$O(n)$ algorithm for the computation of the exchange matrix [30]. In the meantime, the development of FMM-based methods has turned into an exciting area of research. Concepts such as the "J-engine" of White and Head-Gordon (Coulomb term; [31]), or the near-field/far-field methods of Scuseria and coworkers (Coulomb and exchange; [32, 33]) promise robust near-linear scaling methods for Hartree-Fock, density functional theory (DFT), or hybrid methods in the not too distant future.

2.3 Gradient calculation

Differentiation of the Hartree-Fock energy expression (Eq. 5) with respect to a parameter x yields

$$\epsilon^x = \langle \mathbf{h}^x\mathbf{P} + \mathbf{h}\mathbf{P}^x \rangle + \tfrac{1}{2}\langle \mathbf{P}\mathbf{G}^x(\mathbf{P})\rangle + \langle \mathbf{P}^x\mathbf{G}(\mathbf{P})\rangle + V^x_{nuc} \ , \tag{10}$$

where the term involving the differential of the density matrix $\mathbf{P}^x$ can be eliminated by using the two relationships

$$\langle \mathbf{P}^x\mathbf{F}\rangle = -\langle \mathbf{S}^x\mathbf{PFP}\rangle = -\langle \mathbf{S}^x\mathbf{W}\rangle$$

and

$$\langle \mathbf{P}^x\mathbf{G}(\mathbf{P})\rangle = \langle \mathbf{P}\mathbf{G}(\mathbf{P}^x)\rangle$$

to obtain a simple expression which can be evaluated from the computation of the zeroth and first-derivative integrals solely:

$$\epsilon^x = \langle \mathbf{h}^x\mathbf{P}\rangle + \langle \mathbf{P}\mathbf{G}^x(\mathbf{P})\rangle - \langle \mathbf{S}^x\mathbf{W}\rangle + V^x_{nuc} \tag{11}$$

This expression can be evaluated using the same methods presented in the Hartree-Fock energy calculation. If the tasks are defined by quadruples of loop indices $ijkl$, we will have to evaluate the derivative integrals $(ij\,|\,kl)^x$ and compress them with the appropriate products of density matrix elements ($4D_{ij}D_{kl}$ for Coulomb part; $D_{ik}D_{jl}$, $D_{il}D_{jk}$ for exchange part).

Differentiation of the electron-repulsion integrals with respect to the nuclear coordinates x, y, and z generates a sextuplet of integrals. The total number of tasks will remain the same as in the energy calculation, but now a task will consist of the evaluation and processing of six integrals. Because of the contraction of the derivative integrals into a linear array of size N_A, the dimension of the energy gradient, we have both better data reduction and a better communication to computation ratio than in the Fock-matrix evaluation. Parallel gradient computations, at least from a speedup perspective, in general perform better than the corresponding SCF calculation. An example that illustrates this difference in a quite dramatic way is discussed in Sect. 4.1.4.

Should the memory requirement for the storage of the density matrix be excessive, alternative schemes analogous to the ones presented in the energy calculation can be applied.

2.4 Force constant calculation

Frisch, Head-Gordon, and Pople (FHGP) derived expressions for the analytic calculation of SCF second derivatives [35] for an *integral-direct* scheme. In their scheme, implemented in the programs Gaussian [36] and Disco [37], all expressions, including the coupled perturbed Hartree-Fock (CPHF) equations, are derived in terms of AO integral computations and contractions.

The differentiation of the expression for the gradient (Eq. 11) finally yields

$$\epsilon^{xy} = \langle \mathbf{h}^{xy}\mathbf{P}\rangle + \tfrac{1}{2}\langle \mathbf{P}\mathbf{G}^{xy}(\mathbf{P})\rangle + \langle \mathbf{P}^y\mathbf{F}^{(x)}\rangle - \langle \mathbf{S}^{xy}\mathbf{W}\rangle - \langle \mathbf{S}^x\mathbf{W}^y\rangle + V^{xy}_{nuc} \tag{12}$$

An expression for $\mathbf{P}^y$ in Eq. (12) can be obtained from the differentiation of the general SCF condition

$$\mathbf{FPS} = \mathbf{SPF} \tag{13}$$

where projecting out the virtual-occupied space finally leads to an expression for the CPHF equations (Eq. 20) in the work of FHGP [35]):

$$\mathbf{F}\mathbf{P}^x_{ov} - \mathbf{P}^x_{ov}\mathbf{F} - (\mathbf{G}(\mathbf{P}^x_{ov}))_{ov} = \mathbf{F}^{(x)}_{ov} - (\mathbf{G}(\mathbf{S}^x_{ov}))_{ov} - \mathbf{F}\mathbf{S}^x_{ov} \tag{14}$$

Note that in Eq. (14) the term $\mathbf{F}^{(x)}$ is not the derivative Fock matrix $\mathbf{F}^x$, but rather the "Fock matrix" built from first derivative integrals

$$\mathbf{F}^x = \mathbf{h}^x + \mathbf{G}^x(\mathbf{P}) + \mathbf{G}(\mathbf{P}^x) = \mathbf{F}^{(x)} + \mathbf{G}(\mathbf{P}^x) \ . \tag{15}$$

Equation 14 is solved iteratively. In each iteration, the LHS has to be constructed from the current solution for the derivative density matrix, whereas the RHS is a constant.

The calculation of the force constant matrix (Eq. 12) can therefore be divided into the computation of the terms representing contractions of (first and second) derivative integrals, and into the computation of the derivative density matrix. The two steps are often referred to as the "integral Hessian" (or simply "Hessian"), and CPHF steps. A list of the terms (four-index quantities only) that have to be evaluated is presented in Table 1. The resource requirements for the two steps are quite different. In both cases we have effective $O(n^3)$ computations with sizable prefactors. The data reduction, however, is into an array of dimension N_A^2 for the "Hessian step", and into N_A arrays of dimension n^2 for the CPHF part.[4] As in a force constant calculation N_A typically adds two orders of magnitude, this storage requirement either calls for a large-memory server, or for a non-replicated matrix algorithm as presented in Sect. 2.2.2.

Table 1. Two-electron integrals and their contraction in a Hartree-Fock energy, gradient, and force constants calculation. LHS and RHS stand for left-hand and right-hand side of the CPHF equations, respectively. "Hessian" here stands for the "Fock matrices" built from derivative integrals as occurring in the second and third term of Eq. (12)

	Energy	Gradient	Force constants		
			Hessian	RHS	LHS
$(ij\,\|\,kl)$	G(P)			G(P)	G(P)
				G(S^x)	G(P^x)
$(ij\,\|\,kl)^x$		G^x(P)	G^x(P)	G^x(P)	
$(ij\,\|\,kl)^{xy}$			G^{xy}(P)		

[4] NA stands for the number of degrees of fredom (or "perturbations")

Table 2. Resource requirements for some of the key services in a Hartree-Fock calculation. See text for further explanations

	Service						
	Fock	Grad.	Hess.	LHS	RHS	CPHF	Diag.
Computation	pn^3	qn^3	rn^3	pn^3	qn^3	$N_A n^2$	sn^3
Data reduct.	n^2	N_A	N_A^2	$N_A n^2$	$N_A n^2$	–	–
Memory (min)	n^2	n^2	n^2	$N_A n^2$	$N_A n^2$	$N_A n^2$	n^2
Ideal server	any	any	any	SMP	SMP	SMP	SMP

The beauty of this *integral-driven* concept for the generation of the Hessian is that again the bulk of the work consists of integral evaluations and their contraction with various matrices. Therefore we can use the same types of approaches used in the parallel Fock-matrix or nuclear-gradient computation: tasks are defined as the generation and the processing of a batch of integrals. One of the benefits of this approach is that it is well explored, and that the various computations can be implemented for a number of programming models and/or hardware architectures (see [24] and references therein).

In contrast to the classical approaches for the solution of the CPHF equations which are MO based (see, for example, the program Gamess for a parallel implementation [38]), the AO integral-driven scheme of FHGP involves a greater operation count,[5] but is free of large external storage requirements. The somewhat higher operation count for the direct scheme may also be overcompensated by the better integral-prescreening opportunities.

The Disco codes offer MPI (IBM SP2) and SMP (Cray, NEC) implementations of the Hessian calculation, and a (partly parallel) SMP implementation of the CHPF step.

2.5 Distributed-parallel Hartree-Fock calculation

A merit of the integral-driven computational schemes presented in Sects. 2.2, 2.3, and 2.4 is that they can be generalized for networks of (parallel) computers with very little modification. This is essentially due to the fact that all computational tasks of the general type

$$\mathbf{G}^{ij}(\mathbf{P}^k)_{\mu\nu} = \sum_{\lambda\sigma} \mathbf{P}^k_{\lambda\sigma}(\mu\lambda \parallel \nu\sigma)^{ij} \tag{16}$$

can, in principle, be executed independently and in random order. Restrictions may be due to load-balancing or memory-requirement considerations.

Once distributed over a network of servers, each server will evaluate defined fractions of Eq. (16). The results will be accumulated by the client, who will also perform all input/output operations.

If the servers are parallel computers using their processors in the native programming model, we have a *distributed-concurrent* [43], or, in more modern jargon, a *globally distributed-locally parallel* concept of computation. The evaluation of the terms in expression (16) will be *services* implemented on the various servers. The requests for services will be issued by the client. The parallel algorithms presented in Sect. 2 can all be used for the implementation of these services. Architecture and communication bandwidth considerations, however, are criteria to the honored.

Tasks consisting of pure matrix operations such as the symmetrization and the diagonalization of the Fock matrix, the evaluation of the energy expression (5), or the solution of the CPHF Eq. (14) are a (partial) exception to this scheme. Finding distributed memory algorithms for the matrix operations is certainly possible. However, part of the bonus of network computing is the potential availability of hardware which allows execution of these tasks and at very high efficiency (see Sect. 4.3).

In Table 2 a representative set of services required in Hartree-Fock calculations is listed and characterized according to criteria such as scalability ("Computation"), degree of contraction ("Data reduct[ion]", i.e. the dimension of $\mathbf{G}^{ij}(\mathbf{P}^k)$ in expression 16), and the (minimal) server memory requirements. On the bottom line the ideal type of server is listed. For the scaling parameters in "Computation", we have $p < q < r$ and $s < p$. Clearly, this table refers to "straightforward" implementations of the respective services, and tries to illustrate the diversity of computations encountered in a Hartree-Fock calculation.

3 Network computing: interprocess communication

3.1 Objectives and concepts

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 in the earliest successful attempt to distribute computations over several machines [44], but rapidly became obsolete once dedicated communication hardware and software became available.

The focus on computer-architecture independence rather than on performance has made the parallel virtual machine (PVM; [52, 53]) system the most widely used communication tool for the development of distributed memory applications. PVM also was one of the most influential contributors in the definition of the message passing interface (MPI; [45]) standard, which rapidly became the de facto standard for explicit message passing. A number of communication tools, including

[5] $O(N_A N^4)$ versus an (OoN^4) and an $O(o^2 v^2 N_A)$ step with o and v representing the dimension of the occupied and the virtual space, respectively

some implementations of PVM, now use MPI for their message passing.

In computational quantum chemistry, PVM, MPI, and the global array tools (GA; [46]) are the most widely used communication tools. Gaussian, Inc., on the other hand, has selected Linda [47] as the parallelization tool. Linda provides a virtual shared memory called "tuple space" through which data are transferred between processes or from which idle processors can fetch processes (tasks) to be executed. The GA tools, developed by R.J. Harrison and coworkers, support global addressing for distributed arrays from within a MIMD parallel subroutine call tree. The GA tools have been designed in light of emerging standards, in particular high-performance Fortran (HPF; [48]). The one-sided, random access to distributed arrays makes parallel programming much easier, and the GA tools have been used in a number of major code-development projects.

However, in 1990, when we were prepared to distribute SCF computations over a network of (multiprocessor) computers, none of the communication tools available at the time satisfied our modest but somewhat biased requirements. We[6] therefore decided to build a communication tool, later called Sciddle, based on the following design objectives:

- Support distributed applications (client-server model)
- Adhere to semantics of subroutine call by using (asynchronous) remote procedure call (RPC) for invocation of services
- Use TCP/IP sockets for networking clients and servers
- Ensure portability.

The use of TCP/IP sockets was motivated in part by the idea of not being restricted to local-area networks, but also having the option of connecting servers which are geographically remote. For coarse-grain parallelism, a client-server scheme is a common choice of programming model. Portability was an issue because the package would have to be installed on a number of (heterogeneous) servers. The most important decision, however, was to *avoid explicit message passing* and to adhere to the semantics of subroutine calls by using a special implementation of the remote procedure call. The concept was to hide all message passing within these subroutine calls, and to have a stub compiler generate the necessary communication routines automatically.

3.2 Sciddle and Sciddle-PVM

The communication software built according to these design objectives, Sciddle, provides parallelism through an implementation of non-blocking (or asynchronous) remote procedure calls. The user therefore moves within

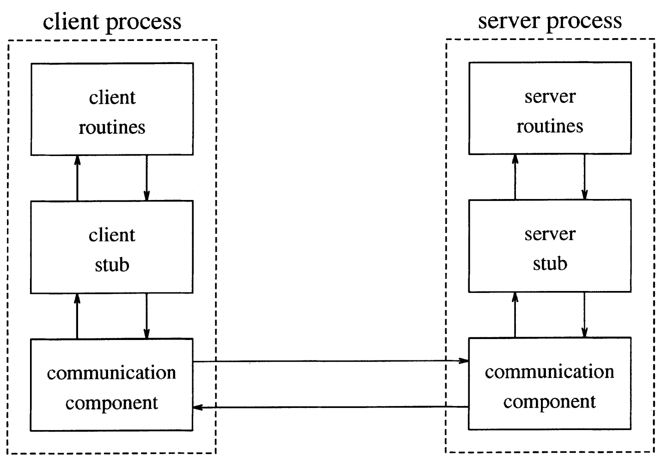

Fig. 4. The data flow between client and server. The stubs are generated automatically by the Sciddle stub compiler, and implement the interface between client and server programs

paradigms well known from sequential programming, and there is no need to send and receive messages explicitly. In general, the server routine is identical to the corresponding part of the sequential version of the program.

Procedure calls to servers are declared by *interface definitions*, which specify the procedures executed by the servers together with the parameters transferred (data type, direction of transfer, etc.) using a simple declarative language. The Sciddle *stub-compiler* translates these interface descriptions to stubs that contain the necessary communication code (see Fig. 4).

The Sciddle *runtime system* (RTS) offers additional subroutines for starting, monitoring, and terminating server processes. Multiple ongoing asynchronous RPCs can be managed by means of call groups [49]. For a complete example see either [50] or the reference manual [51].

Its wide acceptance has led numerous computer vendors to provide high performance PVM implementations. To be able to exploit these optimized PVM implementations, Sciddle was put on top of PVM. The new software also becomes easier to maintain as the Sciddle RTS relies on PVM to support message passing between client and server stubs. These, however, are still generated by the Sciddle stub generator.

Even though another software layer is added on top of PVM, we only observe a minor degradation of performance. This small loss is outweighed by the new Sciddle software safety and ease of use. For a complete description of the new software, see [51, 56].

3.3 Disco-Sciddle

Figure 5 describes the structure of a distributed SCF or nuclear gradient computation as implemented in Disco-Sciddle.

Once the servers and the client are in operation, the client opens the connection to the servers (Server Connect). After completion of some sequential work,

[6] These design objectives were defined on a hike at Lake Lucerne (Switzerland) by Jan Almlöf, John Mertz (Cray Research, Inc.) and the present author on a hot summer day in August 1990. Our ideas were then "refined" and implemented by our colleagues, computer scientists Peter Arbenz and Walter Scott (ETH Zürich)

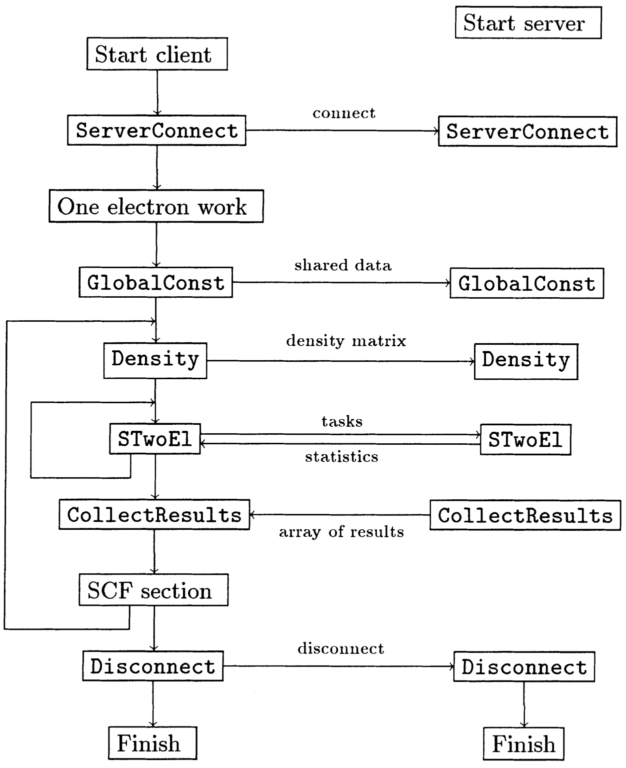

Fig. 5. The structure of the distributed computation. See text for description of the flow of control and data

the client sends out the data needed by the servers to perform the computation (mainly shared variables). This "initialization" of the servers is performed as an RPC by calling the procedures (or "services") GlobalConst and Density, whose only work consists of the transfer of data from the client to the server memory (in the MPI version of Disco both calls are implemented as *broadcasts*). Before entering the inner loop in Fig. 5, the client generates the task tables (*task buffer*), from where packages of tasks are constructed according to the prescriptions of the scheduler, and are sent for processing by a call to the STwoEl service. The argument list of the call to STwoEl only contains the task indices (i.e. the loop indices *i*, *j*, *k* and *l* in Fig. 1), plus – on return – statistical data for the scheduler. The first are arguments declared as IN, and the latter are declared as OUT in the interface definition of the procedure.

STwoEL, in a replicated-matrix scheme, will evaluate and process two-electron integrals and accumulate the results (partial Fock matrix or gradient). At the end of the loop over tasks, the client will call CollectResults to gather the partial results for the final processing. At the end of the cycle the client disconnects the servers using Sciddle library calls.

It should be noted that all calls to server routines are performed by the client. The vertical line connecting the server procedures to indicate the flow of control and data is therefore missing. Common data on the server are available to all routines there. The server does have a main program, but it only performs library calls to have its service(s) registered by the system, and to prepare to process tasks requested by a client.

3.4 Network-computing environments

Even though Sciddle-PVM could be used to parallelize any application which adheres to a tree-shape topology, it is most likely to remain a tool to connect clients to services in a network-computing environment. Interfacing between the RPC and other paradigms is relatively straightforward, and a client can therefore attach any type of server. Ideally, the service called will be a highly optimized implementation for that particular computer architecture, similar to the routines of the basic linear algebra library system (BLAS). In principle, services can be generated that represent modules used in a computation (integral generators, equation solvers, etc.), and will reside on a network of servers accessible to the users registered [60].

With increasing network complexity there is also a need to address the issue of network management at the application level. Most recently, we have observed strong signals coming from the World Wide Web (WWW; the Web) and its user interface and software technology. After changing the way information is communicated, it appears that the Web is also changing the way education is disseminated [58, 59], and, most probably also the way (network) computing will be performed. Many of the concepts developed for network communication can also be applied to network computing.

4 Network computing: experiments and applications

4.1 Early network computing experiments

4.1.1 Overview

A summary of the early networking experiments using Disco-Sciddle is given in Table 3. Among the examples used are the two organic carbonates shown in Figs. 2 and 6.

Strictly speaking, these experiments were not the first network-computing experiments. Clementi and coworkers, with their loosely coupled array of processors (1983 and later), performed the first successful attempt to distribute a computation over several machines [44]. At about the same time, Lüthi and McLean tried to connect several IBM mainframes for direct SCF calculations using electronic mail for message passing between the master and slaves (client and servers in modern jargon). Even though the concept was very promising (an input/output-free algorithm; using general software rather than dedicated hardware for communication), the experiment never reached the production stage [61].

4.1.2 Networks of supercomputers: distributed-concurrent computation

The first major network-computing benchmark using the Disco-Sciddle (Version 1.0) program system was taken with a cluster consisting of five Cray Y-MPs with a total of 20 processors. For the first time we recorded a performance which was significantly greater than the performance of an eight-processor Cray Y-MP, the largest machine of its time. Successful from a functional

Table 3. Early network computing studies Disco-Sciddle

Configuration	Sites[a]	CPU[b]	Comm.	Perf.[c] [Gflops]	Speedup[d] % Parall.	Date	Ref.
SMP (Ref.)							
Cray Y-MP	CCN	8	–	1.52	7.91	9/90	[15]
SMP cluster							
5 Cray Y-MP dedicated	CCN	20	FDDI	3.2	16.2 98.7%	5/91	[64]
Trans-Atlantic							
Cray Y-MP/2	ETH	14	Int'net	1.2	–	9/91	[43]
Cray-2/4	EPF		35 kBs	–			
Cray X-MP/4	MSC						
Cray-2/4 non-dedicated	MSC						
WS cluster							
IBM RS6000 non-dedicated	IBM	27	Eth'net	≥ 0.5	– 99.5%	8/92	[63]
Meta center							
Cray C90	CCN	32	Int'net	16.2	29.5	8/93	[62]
Cray C90 dedicated	PSC		50 kBs		99.7%		

[a] For the sites involved, see the text
[b] Sum of processors of all machines in the cluster
[c] From Amdahl's law without further corrections
[d] The speedups quoted do not always refer to single processor timings, but rather indicate the ratio between wall-clock and CPU time

point of view, the speedups obtained, due to the lack of a flexible task scheduler, remained moderate (16.6 for a complete SCF cycle).

Free from the restriction of having client and/or servers share the same floor space, we extended the concept from local area to wide area networks (from LANs to WANs) by connecting the machines of various supercomputer centers. Probably the most impressive among these experiments was the *Trans-Atlantic Cluster* where the two Cray computers of the Minnesota Supercomputer Center were connected via the Internet with the two Cray computers of the Swiss Federal Institute of Technology (ETH) to form a network of four servers based at three locations (Minneapolis, Zürich and Lausanne) and a total of 14 processors. There were three types of servers involved (Cray X-MP, Y-MP and two Cray-2), all running a UNICOS operating system (see entry "Trans-Atlantic" in Table 3). With this cluster we consistently obtained compute performances well beyond one GFLOPS even during regular production hours.

These early experiments showed that (local and wide area) network computing is a promising approach in large-scale computing. It also became evident that in order to obtain good results, the *load-balancing problem* would have to be addressed. The general observation in the early experiments was that excellent server-internal (or second-level) speedups could be attained by using simple measures such as sorting the batch of tasks according to size, but that the load balancing between servers (first-level load balancing) was the rate-determining step. This was certainly true for servers that would show an unpredictable response. Fault tolerance, on the other hand, turned out to be a very minor issue as we rarely encountered server, network, or client failures.

4.1.3 Networks of workstations

The group of Ahlrichs, using their Turbomole codes [16], was among the first to demonstrate that clusters (or "farms") of workstations are a useful tool for generating high-speed performance at low cost [17, 65]. From a price/performance point of view, the "WS cluster" computations certainly are the most important entry in Table 3.

The focus in the experiments with the 27 IBM RISC workstation cluster was to study task scheduling in systems under regular production conditions. Using a simple scheduler, we demonstrated that the direct SCF energy and gradient calculation can be parallelized to obtain nearly optimal speedups on a distributed memory system. Even under regular service conditions speedups which reflect 99.5% parallelism (Amdahl's law) or an asymptotic speedup of 200 could be obtained.[7] Most importantly, it was possible to generate supercomputer performance at an estimated 20–25% of the cost (see [63]).

4.1.4 Meta-computing

Around 1992/1993, networking computing took on a political dimension when the linkage of the compute resources of the United States National Science Foundation (NSF) supercomputer centers to a "meta center" became an issue.

To study "meta-computing" as a high-performance computing concept, a series of experiments involving three Cray C90 computers located at three different sites were performed. The machines, connected by Internet with a bandwidth of ≥ 50 kBytes/s at the time of the test, were the two 16-processor Cray C90/916 of the Cray Corporate Computing and Networking Facility (CCN) in Eagan (Minnesota), and of the Pittsburgh Supercomputer Center (PSC) – both machines in dedicated mode – along with the Minnesota Supercomputer Center (MSC) 8-processor Cray C90/908.

[7] The maximum speedup ("asymptotic speedup") that can be reached with a 0.5% serial code is 200

For this experiment, the structure of the molecule in Fig. 6 (no symmetry; 6-311G basis, 374 contracted basis functions) was evaluated starting from a minimal basis geometry. Each SCF and gradient computation consumes 1,187 and 8,781 s CPU time, and generates 548 and 564 MFLOPS, respectively (single processor measurements).

The gradient calculation, using the CCN and PSC machines (32 processors in total) was executed in 298 s wall-clock time, corresponding to a CPU to wall-clock time ratio of 29.5, and a processing speed of 16.16 GFLOPS. The corresponding SCF calculation only experienced a moderate speedup upon the clustering of the two servers.

In the gradient calculation, the serial time measured (t_{ser}) was five, the communication time (t_{comm}) nearly 10 s. The amount of data transferred from client to servers was 560 kBytes, whereas the nuclear gradient data returned from the servers was 1.5 kBytes (replicated data algorithm). For the SCF calculation $t_{\mathrm{ser}} = 7$ s, with (almost precisely) twice the communication time of the gradient calculation. The parallelism in that implementation of the SCF calculation is insufficient to generate a significant speedup upon networking the two machines. Adding a third Cray C90 to the network made it possible to increase the performance of the

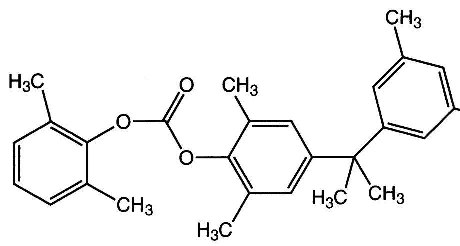

Fig. 6. The example used in the "meta-computing" tests

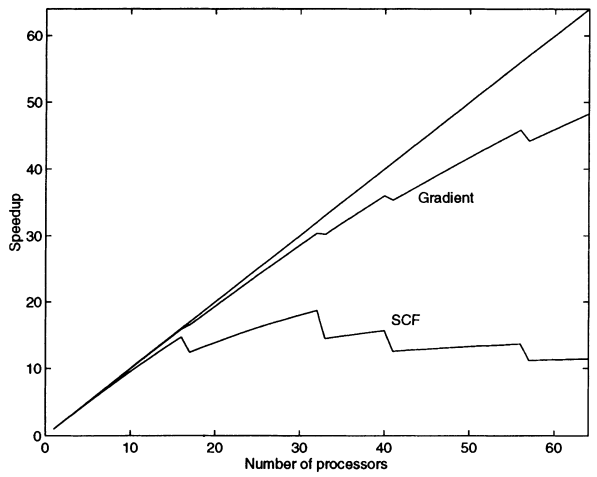

Fig. 7. Projected speedups for the SCF and gradient calculation as a function of the size of the meta-cluster of cray C90 computers (see text) using Eq. (17) with the parameters described in the text. The "sawteeth" show the impact of the communication overhead created each time a new server is added to the network

gradient calculation. The performance of the SCF calculation, however, began to deteriorate.

Using a simple extension of Amdahl's law (Eq. 17) with the performance data collected in these experiments, it is possible to model an extended "meta computing" experiment. The assumptions are that the communication latency for the transfer of the Fock and density matrices between the client and all servers is identical (10 s, corresponding to a bandwidth of 55 kBytes/s, that there is no change in the quality of the first-level load balancing, and that we experience perfect second-level load balancing. These "predictions" therefore represent an upper bound.

$$\mathrm{Speedup} = \frac{t_{\mathrm{CPU}}}{t_{\mathrm{Wall\text{-}clock}}} = \frac{t_{\mathrm{ser}} + t_{\mathrm{par}}}{t_{\mathrm{ser}} + t_{\mathrm{par}}/p + (m-1)t_{\mathrm{comm}}} , \tag{17}$$

In Eq. (17) p stands for the total number of processors in the cluster, and m denotes the number of servers involved in the computation.

The plan to perform a large scale "meta center" experiment by adding the Cray C90 computers of the NASA Ames Research Center in Moffett Field (California), and the San Diego Supercomputer Center (SDSC) to the network was never realized for two reasons: a benchmark involving five different centers under three different concepts of operation (academic, national laboratory, industrial) was difficult to coordinate, and it also seemed that the experiment would not provide insight beyond what could be studied by using simulation. The projected speedups for the SCF and gradient cycles are shown in Fig. 7.

The total calculation took 12 gradient cycles and 80 SCF iterations, which, with a ratio of 5.5 between gradient and SCF cycle time, amounts to 51 h CPU time on a single C90 CPU (approx. 500 MFLOPS.). Using 16 CPUs of one C90, the total time for the computation is reduced to about 200 min. About 45% of this time is spent is SCF cycles. Distributing the computation over the "meta center", this structure optimization terminates in slightly less than 2 h, but now as much as 70% of the time is needed for the SCF computations. The part which distributes poorly (i.e. the SCF part) due to its communication to computation ratio acts like the serial portion in a uniformly parallel application. Unless the example is scaled up drastically, a "meta cluster" of Cray C90s (or equivalent) with the present communication bandwidth is *not* the ideal compute resource.

4.2 Scalability aspects

In network computing, scalability also depends on the performance of the task schedulers (i.e. the load balancing at all levels of parallelization), the inter-node communication overhead, and the completeness of the hardware functionality present (i.e. the type of servers available in the network). An other concern that grows with the complexity of the network is the issue of fault tolerance, or, if addressed in a preventive context, fault resistance.

4.2.1 Task scheduling and load balancing

In a typical production environment, the servers are not dedicated to the user, and therefore any optimal task scheduling has to include the effects of the varying and often unpredictable availability of clock cycles on each server in the network. If the task size, the communication bandwidth, and the server performance (i.e. the number and quality of clock cycles awarded to a task by the server) could be predicted accurately, the task scheduling problem could be solved analytically, and unnecessary server idle time at the synchronization points could be eliminated. In a non-dedicated network none of these parameters can be predicted accurately. They can, however, be measured "on the fly" as the computation proceeds. We therefore decided to design a task scheduler that would generate and distribute tasks based on an algorithm whose parameters would be dynamically updated during the process (see Table 4 below). This scheduler would predict the processing time T_s (and therefore the completion time) of a task issued at *time* $= t$ according to the expression

$$T_s(t) = \frac{W(t)}{SM_sL_s} \tag{18}$$

The tasks have to be issued in "task packages", so that a server will get to process an entire batch of client requests. Obviously, a good strategy is to issue large packages at the beginning of the computation, and then to reduce the package size gradually to a predefined minimal size (usually no task packages smaller than 1% of the total workload were generated). This ensures that the client-server communication events do not scale as the number of tasks (i.e. $O(N^4)$ with N as the number of integral batches), but only as roughly the logarithm of the number of tasks, and to maintain good second-level load balancing.

Within the frame of this model, the optimal amount of work to be assigned to server s is

$$W_s = f\left\{W_{\text{tot}} + \sum_{s' \neq s}(T_{s'} - T)R_{s'}\right\}\frac{R_s}{R_{\text{tot}}} \tag{19}$$

Table 4. Description and policies used for scheduling parameters used in Eqs. (18) and (19)

Item	Description	Update policy	Alternatives
$W(t)$	(Work-)size of task t (estimate: number of primitive integrals)	–	Create log file
M_s	Number of integrals processed by server s per unit time	Measured and updated after completion of cycle	–
L_s	Fraction of server s awarded by server OS to process	Updated after each task	Separate communication
S	Effect of integral screening (global)	Updated after each cycle	Include in M_s for each task
f	Fraction of W_s actually handed out to server s	static	update

where $W_{\text{tot}} = \sum_s W_s$ is the amount of work left; $T_{s'}$ is the point in time at which server s' is scheduled to return (i.e. to complete its current package); T, the current time; $R_s = SM_sL_s$ (i.e. the response of server s); and $R_{\text{tot}} = \sum_s R_s$, the total response of the network.

This scheduling scheme allowed us to obtain excellent results for SCF and gradient calculations on workstation clusters under regular production conditions. With optimal scheduler settings 99.5% parallelism, corresponding to an asymptotic speedup of 200, could be achieved [63].

The measurements showed that because of the stochastic nature of some of the components in the load-balancing problem, it appears very difficult to design an algorithm that outperforms the one presented without introducing a much higher level of complexity. At some point it would be necessary to know the details of the scheduling algorithms used by the *operating system* of the various servers. The time slices awarded to a task of a given size by a server under a given load scenario will differ if the servers are running different operating systems. But even if the network were homogeneous, certain servers may use the same scheduler with different *scheduler settings*. Very small tasks are typically processed with high priority, and thus the scheduler presented here will update the parameter L_s in an undue manner, as the better response is interpreted as a change in the load situation on that server. Trying to avoid falling into vastly different policy regimes of the server schedulers was another reason for introducing a minimal task size concept.

Even more difficult to predict is the network performance. Not much of a concern when using (fast) LANs, the communication delays in WAN clusters may cause major disturbance. Attempts to include the (anticipated) communication bandwidth into the scheduler were unsuccessful. Once centers become connected through communication services which provide a sustained bandwidth, such as ATM networks, this problem will no longer exist.

4.2.2 Matrix operations

Once the parallelization of the quantities

$$\mathbf{G}^{ij}(\mathbf{P}^k)_{\mu\nu} = \sum_{\lambda\sigma} \mathbf{P}^k_{\lambda\sigma}(\mu\lambda \parallel \nu\sigma)^{ij}$$

has been completed, most of the remaining serial part of the computation is in the solution of the matrix equations (Roothaan-Hall; CPHF) to generate a new matrix $\mathbf{P}^k$. In energy calculation with 2,000 or more basis functions, there will be an increasing dependency of the overall speedup on the diagonalization of the Fock matrix, an approximate $O(n^3)$ process, for which no efficient distributed memory algorithms are available yet.

However, with the introduction of update methods to bypass the diagonalization of the Fock matrix as developed by Almlöf and Fischer [40], or with the schemes presented by Shepard [41] and Rendell [42], the need for explicit diagonalization has been greatly reduced. Diagonalization is still required to create an

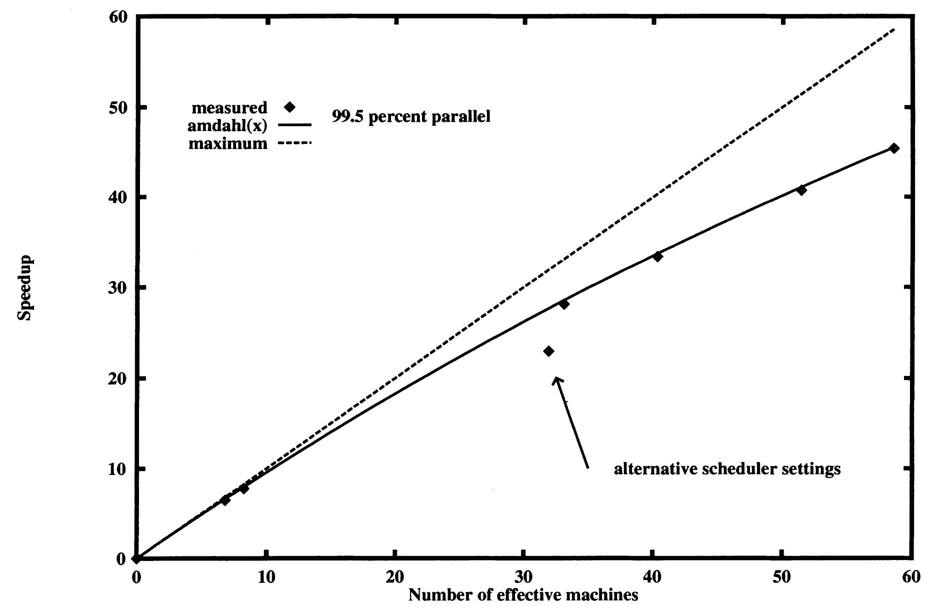

Fig. 8. Task scheduling on a workstation cluster under regular production conditions

initial set of orbitals, *canonical* orbitals at the end of the SCF process, and, in some cases, for the generation of a Fock-transformation matrix (Löwdin orthogonalization).

Algorithms which parallelize for shared memory multiprocessor systems are available. A modern way to address this problem is to use the source code provided by a network library such as Netlib,[8] compile it using the parallelizing compiler of the target shared memory multiprocessor system, and make it available as a *service* in the compute network. The speedups obtained, about 3.3 on an eight-processor machine, only make it possible to "hide" a 40% increase in problem size when using the parallel codes.

4.2.3 Communication

Socket communication between client and servers is a serial process, i.e. the client can exchange messages with only one server at a time. To minimize the impact on scalability in Disco-Sciddle communication is *overlapped* with computation by enabling a server to start working immediately after being initialized by the client. In the "Trans-Atlantic" experiments, for instance, to optimize the overlap of communication and computation, the local servers were initialized *before* the servers across the Atlantic (see Table 3 and Fig. 3 in [43]).

The implementation of a client-server *broadcast* function would require a major redesign of Sciddle and "true" broadcast performance would only be obtained if the network hardware and protocols used were to support such communication.

Generic Pvm – and therefore also Sciddle – relies on UNIX sockets for interprocess communication. For an increasing number of platforms, implementations of Pvm are available which either "sit" on top of MPI or even on the native message-passing system. The Sciddle RPCs, in these cases, are executed using these native protocols, as negotiated by Pvm. RPCs between a client and servers residing on nodes switched through high-speed communication systems such as a HIPPI channel are formally also executed using socket communication, but with protocols performing better than TCP/IP. In these cases, with communication bandwidths up to 100 MBytes/s (see Chap. 4.4), the communication latencies in Hartree-Fock calculations become negligible even in large force-constant calculations.

4.2.4 Fault resistance

Somewhat surprisingly, in network-computing applications we rarely encountered problems such as fatal server or network failure. By far the most common source of problems were server processes which had exhausted their resource limits (mainly the CPU time allocation). In most applications the servers had to be run in batch mode, and it often occurred that a server would be more active than anticipated at submit time. Therefore there was the danger that it would be aborted by the kernel. This example, again, illustrates the need for a more sophisticated network-computing software infrastructure.

Using algorithms other than the replicated matrix schemes, more dynamical reaction on changes in the network performance during processing becomes possible. At times, communication delays between the client and a particular server create idle situations that can be avoided by re-issuing the last task accepted by the server responsible for the delay.

4.3 Heterogeneous networks: Almlöf's eclectic approach

The incorporation of new functionality in a computer network increases the flexibility of the method developer

[8] See: http://www.netlib.org

with respect to algorithm and implementation design. In the January 1993 edition of the SIAM News [67], ASF show that by using two different machine types, with each treating the portions of the algorithm that are appropriate for its architecture, a calculation can be performed considerably faster than on either of the two computers individually. Using a massively parallel computer (a Connection Machine-5 with 512 processors) to build the Fock matrix, and a vector-type computer (a Cray-2) for the remaining part of the SCF cycle, it would take the top-performer machine of that generation (a Cray C90) to beat the performance of – what they called – *the eclectic approach*.

In a more recent paper [26], the same authors again use a vector processor for the matrix algebra part of the SCF calculation in order to "hide" the poor performance of the CM-5 on this segment of the computation, but also to "export" the memory bottleneck away from the MPP. The bypass of the memory bottleneck allows them to perform computations with more than 1,000 basis functions using their small-memory parallel SCF algorithm (see Sect. 2.2.3).

ASF show that the eclectic approach allows one to bypass architectural bottlenecks (memory, input/output) with which a homogeneous compute resource may confront its user. They also show that diverse functionality may be used to enhance the speedup in a distributed computation. If the concept of meta-computing is to become reality, it will be for the fact that it offers resource heterogeneity. In high-performance computing, it is the option to use approaches such as the eclectic approach that will have to generate the "added value" to compensate for the "investment" involved.

4.4 "SuperClusters"

4.4.1 Concept

From the combination of shared-memory multiprocessing and high-speed networking, both well-established technologies, a new line of computer architectures has evolved. From a *hardware* point of view, we have a set of shared-memory multiprocessor modes connected through high-speed networks. From a *programming model* point of view, some of these systems offer a global address space under non-uniform memory access (NUMA) conditions, whereas others have to be programmed using message-passing techniques.

At ETH Zürich, a cluster consisting of four Cray J90 shared memory multiprocessor computers connected through a HIPPI channel was installed. This "SuperCluster" with a total of 40 processors is to give the user a single image view of the resource using system software tools such as the network queuing environment (NQE) and the distributed file system (DFS). For the programmer, this resource is a network-computing platform with very high inter-node communication speed.

4.4.2 Uniform-versus mixed models

If the algorithm permits, the programmer has two options to distribute a computation. One is to reference each node in the supercluster as a multiprocessor server using its own (native) multitasking system. In that case, we have a *mixed model* which combines client-server with shared memory parallelism (i.e. globally distributed – locally parallel).

However, with the latest implementations of MPI, the programmer will have to consider the option of referencing each processor in each SMP server separately. It is expected, at least for systems with eight or more processors, that spawning n processes on an n-processor SMP architecture will be more efficient than using this server as a shared-memory parallel resource [68]. In this case, a client will spawn processes that all will execute in serial mode. Parallelism is controlled and invoked at the client-server level. Apart from taking advantage of potential turnover in processing speed, this bypass of shared memory parallelism when using an SMP server also frees the user from having to maintain several versions of the server codes. Since only one programming model is involved here, we will refer to this approach as the *uniform model*.

4.4.3 Early examples using the uniform model

Using a 470 basis functions Hartree-Fock energy calculation as an example, the prediction that the uniform model will be competitive could be confirmed [69]. The results, graphically displayed in Fig. 9, show that the performance difference between the uniform and the mixed model is very small even for an SMP server with only a few processors (an eight-processor Cray J90).

The communication overhead, $2n$ rather than just two sets of matrices (replicated scheme) have to be transferred, does not seem to critically affect the performance of the uniform model. As long as only one server is involved, differences in load balancing do not show. In both cases the outermost loop is over the number of processors, and the tasks are distributed from a task buffer. We should note that on certain SMP systems it may be more efficient not to generate task buffers (a serial process), but to loop over tasks instead. The memory-time integral for the uniform model, however, will be n-times larger. Also, in applications such as the computation of the LHS of the CPHF equations, the memory requirement set by the algorithm may not allow the use of the uniform model (see Table 2).

5 Experiences, conclusions, and outlook

The first phase of network computing has been concluded. After about 5 years of practical experience, it appears that the most important "evolution product" of the first network computing era is the workstation cluster. Viewed initially as a way to accumulate *sheer computing power*, the focus finally shifted towards *computational efficiency*. Combining the power of low-cost processors, often scattered beyond domain boundaries, makes it possible to run calculations at supercomputer speeds. This shift of focus is illustrated by the fact that many of the popular quantum chemical

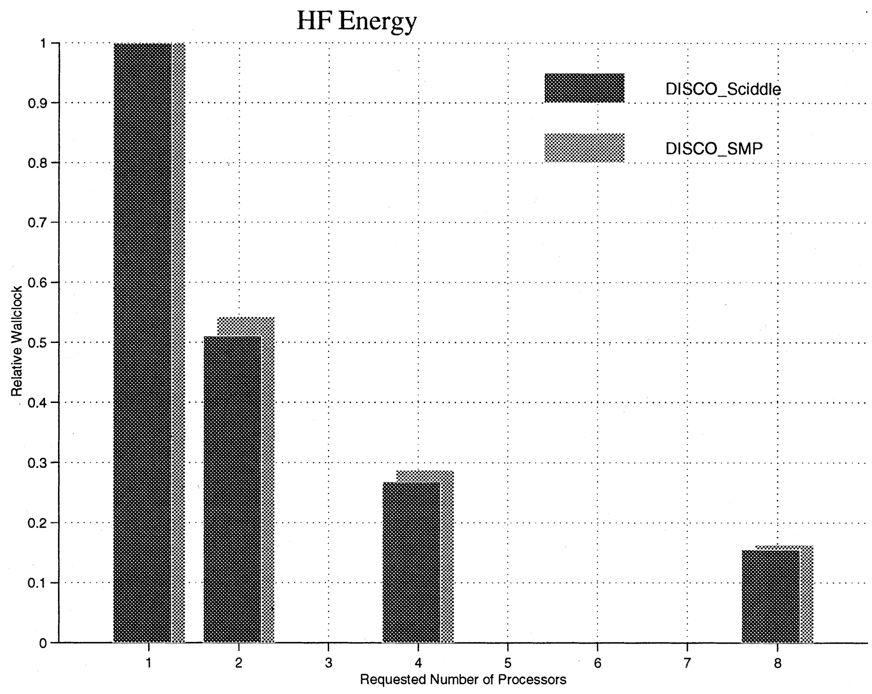

Fig. 9. Tasking versus uniform model

softwares such as Gaussian or Turbomole offer "network versions" of their codes.

Network computing has demonstrated that computations can be performed faster than on any other single computer. It has also been demonstrated that computations can be performed more effectively, if the hardware diversity of a heterogeneous network permits response to resource requests of an algorithm which are difficult or impossible to satisfy by the remaining servers in the network (the eclectic approach). Network computing, however, has not yet been able to demonstrate that computations larger than those possible on a single server can be performed. Methods which scale linearly in problem size, or applications that pose resource requests impossible to honor by one single computer architecture, will change this observation.

Once communication softwares such as PVM or the GA tools became available, the development of networked versions of applications advanced quite rapidly. Fortunately, a standard in message passing is emerging (the message-passing interface, MPI). Convergence in the area of computing infrastructure will remove many obstacles (and uncertainties) from method, algorithm, and program development.

Meta-computing, on the other hand, has not yet lived up to the expectations. Among the reasons are the lack of the corresponding software infrastructure and the fact that the Internet, the main communication network connecting academic compute centres, is burdened by the WWW and other services. One exception is the "MetaCenter for Supercomputing in Norway",[9] which links the compute resources of the four Norwegian universities through a 34 Mbit/s network. Similar "research networks" are planned elsewhere. Also, dedicated inter-center GBits/s network connections have been installed to facilitate the networking of compute resources. The availability of these communication facilities will certainly have a very positive impact on developments in network computing.

The adaptation of method and algorithm design to parallel computer architectures has stimulated a number of interesting developments. The call for linear-scaling methods, for example, has initiated some very promising efforts to compute the Coulomb matrix in Hartree-Fock, DFT and hybrid methods at near-$O(n)$ scaling. All examples, including the "quantum chemical tree code (QCTC)" of Almlöf and coworkers, are based on fast multipole methods.

In their QCTC paper, the authors note that "*the decoupling of the exchange and Coulomb matrices provides the freedom to pursue specialized algorithms for the evaluation of each.*" Initially a recipe to reduce the memory requirements of the replicated Fock-matrix algorithm, an $O(n^2)$ request impossible to satisfy in an MPP, the small-memory parallel strip algorithm has evolved into much more than an elegant bypass of a hardware bottleneck. In the meantime, near-$O(n)$ methods for the computation of the exchange matrix have been presented.

Even if network computing has not stimulated any such important method developments, its impact on the way we perform computations, and on the way we design application programs, has been noticeable. Splitting computations into services, an approach related to object-oriented programming, creates the opportunity to utilize similar concepts to those used to implement libraries such as the basic linear algebra system (BLAS). The availability of algorithms that, at least for a certain part of the computation, provide extremely high performance (superlinear speedups on an MPP, for example), will invite the program developers to reconsider some of the concepts.

[9] See: http://www.metacenter.uio.no/

The accomplishments reported in the application-software area, the more transparent situation with regard to computing models and paradigms, along with the expected increase in network capacities, will greatly facilitate advances in network computing, and bring the slogan "The network is the computer" closer to reality.

Acknowledgements. I gratefully acknowledge the encouragement and support of Jan Almlöf, to whose memory this work is dedicated. I would also like to acknowledge the collaboration of P. Arbenz, T.H. Fischer, M. Feyereisen, J. Hutter, J. Mertz, W. Scott, C. Sprenger, S. Vogel, and U. von Matt, whose contributions were essential for the success of this project. This work was supported by grants from the Swiss Science Foundation, Cray Research, Inc., the Swiss Center for Scientific Computing, and the ETH Zürich Research Funds. Finally, I would like to thank J. Hutter and G. Vacek for helpful comments and constructive criticism of the manuscript.

References

1. Almlöf JE (1994) In: Roos BO (ed) Lecture Notes in Chemistry II, vol 64. Springer, Berlin Heidelberg New York, p 1
2. Almlöf JE (1995) In: Yarkony DR (ed) Modern electronic structure theory. Advanced series in physical chemistry, vol 2. World Scientific, Singapore
3. Almlöf JE, Fægri K Jr, Korsell K (1982) J Comp Chem 3:385
4. Almlöf JE, Taylor PR (1984) In: Dykstra C (ed) Advanced theories and computational approaches to the electronic structure of molecules. NATO ASI Series C, 133. Reidel, Dordrecht, pp 107–125
5. Sæbo S, Almlöf JE (1989) Chem Phys Lett 154:83
6. Klopper W (1995) J Chem Phys 102:6168 Noga J, Klopper W, Kutzelnigg W, In: Bartlett RJ (ed) Modern ideas in coupled-cluster methods. World Scientific, Singapore
7. Koch H, Jørgensen P, Helgaker T (1996) J Chem Phys 104:9528
8. Disco (ETH Version) is a direct SCF program written by Almlöf JE, Fægri K, Fischer TH, Korsell K, Feyereisen MW, Lüthi HP
9. Supermolecule, a program written by Almlöf JE, Feyereisen MW
10. The first teraflops performance on a Linpack benchmark was reported on 17 December 1996. "The configuration that ran the 1.06 teraflops MP Linpack rating consisted of 57 cabinets, 7,264 Pentium Pro processors, and 454 gigabytes of system memory" (a quote from the Intel press release).
11. "The network is the computer" is a trademark of Sun Microsystems, Inc.
12. Pulay P, Fogarasi G, Pongor G, Boggs JE, Vargha A (1983) J Am Chem Soc 105:7037; Pulay P (1995) In: Yarkony DR (ed) Modern electronic structure theory. Advanced series in physical chemistry, vol 1. World Scientific, Singapore
13. Pulay P, Sæbo S (1986) Theor Chim Acta 69:357
14. McWeeny R (1960) Rev Mod Phys 32:335; McWeeny R (1961) Phys Rev 128:1028
15. Lüthi HP, Mertz JE, Feyereisen MW, Almlöf JE (1992) J Comp Chem 13:160
16. Ahlrichs R, Bär M, Häser M, Horn H, Kölmel C (1989) Chem Phys Lett 162:165
17. Brode S, Horn H, Ehrig M, Moldrup D, Rice JE, Ahlrichs R (1993) J Comp Chem 14:1142
18. Petterson LGM, Faxen T (1993) Theor Chim Acta 85:345
19. Feyereisen M, Kendall RA (1993) Theor Chim Acta 84:289
20. 1990 Gigaflop Performance Awards, Cray Research, Inc., Eagan (Minn.), 1990, p 25
21. Computerworld, Vol XXV, No 26 (1 July 1991), p 59
22. Furlani TR, King HF (1995) J Comp Chem 16:91
23. Schütz M, Lindh R (1997) Theor Chim Acta 95:13
24. Foster IT, Tilson JL, Wagner AF, Shepard RL, Harrison RJ, Kendall RA, Littlefield RJ (1996) J Comp Chem 17:109
25. Harrison RJ, Guest MF, Kendall RA, Bernholdt DE, Wong AT, Stave M, Anchell JL, Hess AC, Littlefield RJ, Fann GL, Nieplocha J, Thomas GS, Elwood D, Tilson JL, Shepard RL, Wagner AF, Foster IT, Lusk E, Stevens R (1996) J Comp Chem 17:109
26. Sargent AL, Almlöf JE, Feyereisen MW (in press) In: Astfalk G (ed) Applications on advanced architecture computers. SIAM Philadelphia. See also: University of Minnesota Supercomputer Institute Report, UMSI 95/170, August 1995
27. Häser M, Ahlrichs R (1989) J Comp Chem 10:104
28. Panas I, Almlöf JE, Feyereisen MW (1991) Int J Quantum Chem 40:797
29. Challacombe M, Schwegler E, Almlöf JE (1996) J Chem Phys 104:4685
30. Schwegler E, Challacombe M (1996) J Chem Phys 105:2726
31. White CA, Gordon HH (1996) J Chem Phys 104:2620
32. Strain MC, Scuseria G, Frisch M (1996) Science 271:51
33. Burant JC, Scuseria G, Frisch MJ (1996) J Chem Phys 105:8969
34. Greengard L, Rohklin V (1989) Chem Scr A29:139
35. Frisch M, Head-Gordon M, Pople J (1990) Chem Phys 141:189
36. Frisch MJ, et al (1995) Gaussian94. Gaussian, Inc., Pittsburgh, Pa
37. Fischer TH (unpublished work)
38. Gamess-uk, a program written by Guest MF, et al
39. Horn H, Weiss H, Häser M, Ehrig M, Ahlrichs R (1991) J Comp Chem 12:1058
40. Fischer TH, Almlöf JE (1992) Phys Chem 96:9768
41. Shepard R (1993) Theor Chim Acta 84:343
42. Rendell A (1994) Chem Phys Lett 229:204
43. Lüthi HP, Almlöf JE (1993) Theor Chim Acta 84:443
44. 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
45. MPI Forum (ed) (1994) MPI: a message-passing interface standard. Int J Supercomputer Appl 8:165
46. Nieplocha J, Harrison RJ, Littlefield RJ, J Supercomputing (in press)
47. Carriero N, Gelernter D, Mattson TG, Sherman AH (1994) Parallel Computing 20:633
48. High performance Fortran forum (1993) Technical report version 1.0, Rice University
49. Sprenger C (1993) User's guide to Sciddle version 3.0. Technical report 208, ETH Zürich, Computer Science Department, December 1993
50. Arbenz P, Sprenger C, Lüthi HP, Vogel S (1995) Concurrency: practice and experience 7:121
51. von Matt U (1996) User's guide to Sciddle version 4.0. Swiss Center for Scientific Computing Technical Report, CSCS/SCSC TR 40-96, December 1996
52. Geist A, Beguelin A, Dongarra J, Jiang W, Manchek R, Sunderam V (1994) PVM, parallel virtual machine: a user's guide and tutorial for networked parallel computing. MIT Press, Cambridge, Mass
53. Sunderam V, Geist GA, Dongarra J, Manchek R (1994) The PVM concurrent computing system: evolution, experiences, and trends. Parallel Computing 20:531
54. Casanova H, Dongarra J, Jiang W (1995) Technical report CS-95-301, University of Tennessee, Computer Science Department, Knoxville, Tenn, August 1995
55. Poxon H, Costello L (1995) Network PVM performance. Cray Research Inc., Software Division, Eagan, Minn (unpublished manuscript)
56. Arbenz P, Gander W, Lüthi HP, von Matt U, Liddell H, Colbrook A, Hertzberger B, Sloot P (eds) (1996) Lecture notes in computer science. Springer, Berlin Heidelberg New York
57. Strumpen V (1995) The network machine. Diss. ETH No. 11227, Swiss Federal Institute of Technology (ETH), Zürich
58. Lüthi HP, Vacek G, Hilger A, Klopper W (1997) Computational chemistry: current trends, vol 2. World Scientific, Singapore (1977)

59. Vacek G, Flükiger P, Hilger A, Lüthi HP (1997) *Chimia* 51:100
60. Thiel W (private communication)
61. McLean AD, Lüthi HP (unpublished work)
62. Projects in Supercomputing 1994. Pittsburgh Supercomputer Center, Pittsburgh, Pa
63. Vogel S, Hutter J, Fischer TH, Lüthi HP (1993) Int J Quantum Chem 45:665
64. Arbenz P, Lüthi HP, Mertz JE, Scott W (1992) Int J High Speed Computing 4:87
65. Bode S (1992) Future Generation Computer Syst 8:27
66. Anderson E, Bai Z, Bischof C, Demmel J, Dongarra J, Du Croz J, Greenbaum A, Hammarling S, Mc.Keeney A, Ostrouchov S, Sorensen D (1995) LAPACK user's guide, (2nd edn). SIAM, Philadelphia
67. Sargent AL, Almlöf JE, Feyereisen MW (1993) SIAM News, 14 January
68. Hempel R (MPI Forum) (private communication)
69. Vacek G, Lüthi HP (to be published).