Theor Chem Account (2008) 120:133–153
DOI 10.1007/s00214-007-0306-6

REGULAR ARTICLE

Towards a special-purpose computer for Hartree–Fock computations

What's on the table, and how do we take it?

Tirath Ramdas · Gregory Egan · David Abramson · Kim Baldridge

Received: 21 November 2006 / Accepted: 12 February 2007 / Published online: 28 April 2007

Abstract We propose the development of a special-purpose computer for the Hartree–Fock method, which generally suffers quartic time scaling. We conduct a qualitative assessment of the various computational components, with a focus on electron repulsion integrals (ERI), and consequently map various architectural traits to the various computational components. A quantitative analysis of one component is also presented. We go on to mull over the idea of mixed precision arithmetic. These analyses will aid the practical development of a specialized high performance multi-architecture computer.

Keywords Hartree–Fock · Ab initio · Electron repulsion integrals · Supercomputing · Numerical analysis

Contribution to the Mark S. Gordon 65th Birthday Festschrift Issue.

T. Ramdas (✉) · G. Egan
Center for Telecommunications and Information Engineering,
Monash University, Melbourne, Australia
e-mail: tirath.ramdas@eng.monash.edu.au

G. Egan
e-mail: greg.egan@eng.monash.edu.au

D. Abramson
Centre for Distributed Systems and Software Engineering,
Monash University, Melbourne, Australia
e-mail: david.abramson@infotech.monash.edu.au

K. Baldridge
Organic Chemistry Institute, University of Zurich,
Zurich, Switzerland
e-mail: kimb@oci.unizh.ch

1 Introduction

Over the last few decades there has been a considerable shift away from carefully architected supercomputers towards large-scale distributed memory computational clusters comprised of commodity building blocks. Even large US supercomputing facilities have not escaped this trend. The arguments for generality (particularly savings in cost and development time) are well established, and undermine the feasibility of any proposal that involves an exclusively application specific solution.

However, the growing gap between theoretical peak performance and achieved real-world application performance on general-purpose computer systems provides an impetus to consider new strategies that will better serve the needs of scientific numerical computation. Such a strategy is the concept of a *science-driven system architecture* (SDSA), proposed by IBM and several leading US national computing laboratories [1]. A key aspect of this strategy is for systems to be deliberately designed from the ground up (as opposed to expedient reactionary adaptation of existing systems) based on a careful evaluation of the specific needs of the target application. In this paper, the target application is *computational quantum chemistry* (CQC).

The principle goal of quantum chemistry is to solve *Schrödinger's equation*:

$$\hat{H}\Psi = E\Psi \tag{1}$$

and specifically we target the commonly employed *Hartree–Fock* (HF) method. Frontiers in drug discovery, nanotechnology, and many other areas, demand order-of-magnitude improvements in computational power to extend large-scale quantum chemistry simulation and modeling capabilities into these domains. CQC has long been frustrated by the inability to practically tackle molecules larger than a few hundred

atoms at most. This is due to the $O(N^3) - O(N^4)$ problem size scaling, and much beyond this for post-Hartree-Fock methods.

This scaling applies to the number of *electron repulsion integral* (ERI) function calls, and computation of each ERI is non-trivial. In fact, CQC suffers from a "double-whammy" – (1) the large number of ERIs to compute, and (2) each ERI is relatively computationally demanding.

Our approach hinges on the hypothesis that a special-purpose processor architecture may efficiently exploit far more parallelism inherent in the computation, beyond the capabilities of mainstream general-purpose computers. Our intention is that through sheer parallelism our solution will yield order-of-magnitude performance improvements over mainstream general-purpose parallel systems, in-spite of the implicit clock-speed and logic area penalties associated with low-volume custom systems.

It should come as no surprise that we derive some inspiration for our approach from the very successful *GRAvity PipE* (GRAPE) line of special-purpose processors [2]. The general goal of the GRAPE project was to accelerate astrophysical N-body computations, and is also applicable to molecular dynamics. The *Protein Explorer* project [3] proposes to utilize a large number of GRAPE processors to construct a special-purpose high performance molecular dynamics computer. We envision a similar solution for the CQC problem, although in this paper we focus on a single-node.

In this paper we present a comprehensive analysis of the target problem and consequently map the constituent subtasks to optimal but practical architectures and architectural traits guided by the *design-space approach* espoused by Sima et al. [4]. We illustrate that some recent trends in mainstream computing fail to address the needs of the problem — for instance, there is a growing realization that architectures that depend upon multi-level memory cache structures for performance have reached the limit of manageable complexity and associated power consumption, while at the same time do little to enhance the throughput of the computation. The computer architectures of the last 60 years are well entrenched, however as far as we are aware there has not been a comprehensive architectural exploration of the overall HF method with a particular focus on ERI.

This paper is organized as follows. We begin with a discussion of why a special-purpose approach is warranted. We then go on to a brief commentary on supercomputing paradigms as they relate to this project. We then provide background information on the HF method and ERI computation in Sect. 4. A review of previous architectural designs are reviewed in Sect. 5. In Sect. 6 we lay out our design-space analysis of the overall HF computation, as well as a more in-depth analysis of ERI computation. In Sect. 7 we present a quantitative analysis of one of the stages of the ERI algorithm. A discussion on numerical precision is presented in Sect. 8. Finally we mention future work avenues and primary conclusions.

2 Motivating a special-purpose architecture

General-purpose commodity microprocessors are undergoing a major transition in their internal architectures, embracing parallel processing in the form of multi-core (multiple processors) and multi-threading. Much of the research for this was done over a decade ago but has only just become mainstream as the clock rates and power consumption limits of traditional von Neumann processors have been reached. It is unlikely that the peak performance of these new microprocessors will be reached given their complex internal memory structures in particular.

Commodity processors in the form of application accelerators still provide apparently attractive options if we are to believe the claims for peak performance. Two of the more noteworthy are the Cell Broadband Engine (Cell BE) [5] and the ClearSpeed CSX multithreaded array processor [6]. We will discuss their potential later but most of us are now well aware of the pitfalls of peak performance figures.

We believe that a ground-up application directed approach to processor design is warranted, and likely to be rewarding. Our approach is to *make the processor fit the application*, and in this regard we derive much inspiration from Joel Emer's *relaxing constraints* philosophy [7].

The *make the processor fit the application* mantra leads us to *field-programmable gate arrays* (FPGAs), at least for initial prototyping (we will touch on this more in the next section). Use of FPGAs is made much easier by system integrators, such as with Cray's XD-1 system. It is likely that we will make use of such a system to ease development of the prototype, however we will not embrace any constraints imposed by these existing systems (such as availability of bandwidth and off-chip high-speed RAM) early in the architecture development because doing so will undermine the *make the processor fit the application* mantra. Accepting such constraints early is likely to be counter-productive anyway, as new or updated systems emerge, bringing new interconnection topologies and resource/bandwidth allocation with them. This is true even for FPGAs, where even within product lines there is significant differentiation between individual FPGA series (eg. Xilinx Virtex-4 FX vs. LX vs. SX, each of which is optimised for a different application scope [8]). This leads us to another contribution of our ground-up approach: our analyses can help influence the evolution of systems by proposing desirable architectural shifts.

Perhaps most ambitious of all is the possibility that our system will lead to the spawning of a general-purpose system with wide applicability to a range of supercomputing applications. The most famous recent example of this would

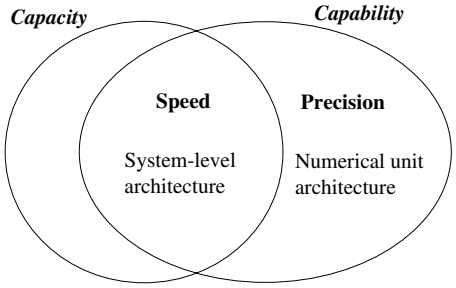

Fig. 1 Supercomputing paradigms, from the perspective of this project

be the development of the IBM BlueGene platform, which was significantly influenced by the QCDOC special-purpose quantum chromodynamics computer [9,10].

We are committed to the construction of the system that we propose. At some point we will have to accept platform and technology constraints in order for the system to materialise, however we chose to delay the acceptance of such constraints, in other words we will initially perform an *optimistic design-space exploration*, which is more likely to reveal opportunities for breakthrough performance. Once we have identified these opportunities, we will consider constraints and how to overcome or avoid them.

3 Capacity computing or capability computing?

At the outset, one may consider which supercomputing paradigm one should pursue. However, we first consider several realities that would dictate the nature of the beast that we construct:

- Algorithmic advances will continue to be a major source of progress in ab initio computation—the solution must be able to incorporate such advances.
- In-house supercomputing is a luxury few can afford—the solution should be something that supercomputing centers can easily embrace and serve out to remote users.
- Fully custom integrated circuit (IC) fabrication is very expensive. Although we have not ruled this out, we have strong reason to consider the use of programmable logic devices instead, specifically FPGAs, which have been applied to many supercomputing problems such as lattice quantum chromodynamics [11] and molecular dynamics [12]. FPGAs have the principal advantage of rapid development cycles of weeks rather than years for custom implementations, and a performance curve which is now growing more rapidly than commodity processors based on custom ICs [13] (Fig. 1).

We do not propose to do away with conventional computational clusters—the intent is to augment them with the proposed accelerator hardware, via high performance interconnects. In this, one can simply retrofit this accelerator into existing installations. It is most likely that the immediate implementation will be based on FPGAs. In addition to an accelerated performance curve [13], an approximate calculation shows that a Xilinx Virtex-5 FPGA is already capable of outperforming a 3.0 GHz Intel Xeon processor between 1.5 and 5 times depending on the mix of floating-point operators [14]. Although projecting performance of applications on FPGAs is not an exact science, it is clear that there is potential for performance gains. Further accentuating our optimism regarding FPGAs is that industry in general is becoming increasingly supportive of the *reconfigurable computing* paradigm (the current-generation manifestation of which are FPGAs). The recent announcement by Hewlett-Packard regarding *field-programmable nanowire interconnect* (FPNI) technology suggests gains of a factor of eight in FPGA density using current fabrication lines [15]. The increase in functionality offered for large FPGA dies is substantial. It is also broadly accepted that the clock rates for FPGAs will continue to increase.

Nevertheless the current work is about architectural options, and we are not committed to implementation technologies, whether full-custom ASIC on one extreme or commodity FPGA on the other. The present work is a study on special-purpose computer architecture. Let us now consider the potential gains extracted from such a study.

3.1 Dimensions of scalability

There are at least two distinct and important facets of *scalability* to which our work can promote large-scale computations. The first facet is *reduction of computation-time*, which is the main thrust underscoring all supercomputing projects, regardless of whether the motivating paradigm is capacity computing or capability computing. Simply put, if one can crunch numbers quicker, one can:

1. do a greater number of simulations in a given period of time (larger capacity).
2. attack larger problems within a feasible period of time (larger capability).

The second facet is *improved precision*. With regards to computer architecture, the problem of precision (and indeed accuracy) is tied to *floating point error accumulation*. This problem is often ignored by practitioners of large-scale numerical computation, which is disturbing because the validity/confidence of conclusions based on values resulting from these computations is not known. Large-scale problems—i.e. high *capability* problems—are particularly at risk, because very wide reduction operations may significantly accumulate floating-point errors. Further compounding this

problem is the reasoning presented in [16] that larger problems (as characterized by larger basis sets and higher proportion of heavy elements) in fact require energy term values of greater precision to allow chemically meaningful conclusions to be formed—in other words, larger problems require better precision, but the computed values are in fact of poorer precision.

Unfortunately, discounting the possibility of major algorithmic breakthroughs, strictly from the perspective of computer architecture, computation-time and floating-point precision are at odds with each other. Improving accuracy—for example by increasing floating-point wordsize (and thus increasing precision)—or incorporating interval/affine arithmetic mechanisms to provide accuracy guarantees would consume time and/or logic resources that could otherwise be devoted to higher parallelism and therefore quicker computation. However, we are mindful of one simple fact: there is limited need for accuracy improvement/validation of computational results without the ability to compute larger problems to begin with, and thus one should initially place a higher degree of importance on improving computation-time—though obviously we shall not wantonly sacrifice precision.

Let us return to our original question—what is the underlying goal: *capacity computing or capability computing*? The issues confronting us are illustrated in Fig. 1. In the context of this particular project, we can reframe this question as follows: *how much importance is placed on floating point precision*? In fact, we actually do not have to answer this question, because of our particular strategy.

3.2 Methodology

The electronics industry now depends for its survival on ease of design and the ready ability to produce high-performance low-power systems quickly within months of conception to product. While their efforts are largely directed at high volume embedded applications, the design tools and the devices they target suit our needs. It is the availability of modern system-level design tools such as SystemC [17] that encourages us to pursue this project where where a decade ago others met with considerable frustration [18].

We will initially partition our design efforts into two basically independent levels: (1) system-level architecture, and (2) numerical functional unit specification. Any contemplation of capability computing will be driven by the latter, and the system-level architecture design of the processor will be unaffected.

We are able to partition in this manner due to rapid strides in digital design methodology and tools, which allow very high abstraction levels while giving us the ability to defer low level design decisions, such as the precision and range of arithmetic units and indeed the numerical representation used, until very late in the design, and to change these decision at will with little or no cost. Changes in technologies are dealt with automatically within the design tools—e.g. we can leverage evolutionary growth in FPGA logic capacity by specifying an increase in the number of functional units, and the tools will carry out the low-level tasks of allocating datapaths and placement of the various units within the FPGA fabric.

Inevitably, the question arises: *are these automated tools capable of high performance implementation*? The industry drivers are now enormous and we believe the answer is yes. We will show as much in future communications. In this paper, however, the focus is on *big-picture* architectural issues.

4 The Hartree–Fock algorithm

A comprehensive introduction to the theory behind the HF method is beyond the scope of this paper, but we will develop sufficient context to justify decisions and design motivations discussed.

This section is organized as follows. First we present an overview of the HF method, and elaborate the computational bottleneck. Then we quickly expand on the notion of basis functions that are fundamental to the method as it is the input data. We go on to deal with the computation of the ERIs. Finally, we briefly touch on post-FH electron correlation computations.

Before addressing the details of the HF method, one additional scope limiting decision to be discussed, concerning the computation vs. storage of the ERI results. HF is an iterative method, however the computation of the ERIs themselves are identical in each iteration. This means one can simply compute the ERIs at the start of the computation, store all the results, and read the ERI results from storage in subsequent iterations. Often time this is considered the default or conventional SCF method (and so it is just referred to as the de facto SCF method, but for the sake of clarity we will call this the *stored-integral* approach). However, as previously mentioned, a very large number of ERIs are typically required – $O(N^4)$. For example, a relatively small molecule Naphthalene ($C_{10}H_8$), computed using the 6-311G basis set already requires 1GB of storage.[1] The factor limiting the maximum problem size quickly becomes storage capacity. This sentiment is shared by the workers of the *Tensor Contraction Engine* project [20], among others.

Since the ultimate desire is to pursue computations for much larger molecular systems, the goal is to establish methods that are highly scalable. One strategy would be to accept potential penalties such as recomputation. Guided by this

[1] Single-point energy RHF/6-311G computation with 298 basis functions, performed using GAMESS [19].

philosophy, we embrace the *direct* approach to such calculations (as proposed in the seminal paper by Almlöf et al. [21]) instead of the stored-integral approach. With the direct approach, ERIs are recomputed each iteration, and because the ERI results may be reduced to a $O(N^2)$ data structure as they are obtained (see Sect. 4.1 for elaboration), one avoids the need to store $O(N^4)$ ERI results. The obvious trade-off is the need to do $N_{\text{iterations}} \times N_{ERI}$ integrals, however $N_{\text{iterations}}$ is typically reasonably small, usually less than 20 [22].

Furthermore, direct-SCF places pressure upon processing power whereas stored-integral-SCF places pressure upon storage components and interconnects. In general progress in the former has significantly outpaced progress in the latter. This has led to the observation that on many conventional systems (including commodity clusters) direct-SCF outperforms stored-integral-SCF.

4.1 Self-Consistent Field Method

Recall Schrödinger's equation (Eq. 1). The individual components have the following significance:

- $\hat{H}$ – the Hamiltonian operator; the transformation function.
- Ψ – the wavefunction; eigenvector.
- E – the energy; eigenvalue.

The Hartree–Fock method is a numerical procedure used to solve the Schrödinger equation for multi-electron atoms or molecules described in the fixed-nuclei approximation (i.e. the Born-Oppenheimier approximation) by the electronic molecular Hamiltonian. Because of the complexity of the differential equations for any but the smallest of molecules, the problem is usually impossible to solve analytically, and thus, the numerical technique of iteration is used. The method, referred to as the *self-consistent field* (SCF) method, is iterated until a set of convergence criteria is met. Figure 2 depicts the procedure.

The ERIs and density matrix are required to form the *two-electron component* of the *Fock matrix* which is subsequently diagonalized to obtain the molecular wavefunction, which encapsulates much valuable information about the molecule. Henceforth we will refer to the two-electron component of the Fock matrix as the *G matrix*. The computation of the G matrix is described programatically in Fig. 3.

To further support our claim that ERI and G matrix construction should be the subject of our attention, consider profiling results for two test cases presented in [23] which we summarize in Table 1. These figures clearly suggest that as problem size increases, the ERI and G matrix construction dominates the computation time of the overall SCF algorithm. Similar findings are presented in Table 1 of [18]. For the Morphine example in Table 1, which is a relatively small molecule, the maximum possible speedup one can achieve is a factor of 50. Furthermore the maximum speedup roughly scales linearly as a function of basis size — we discuss this in Sect. 6.

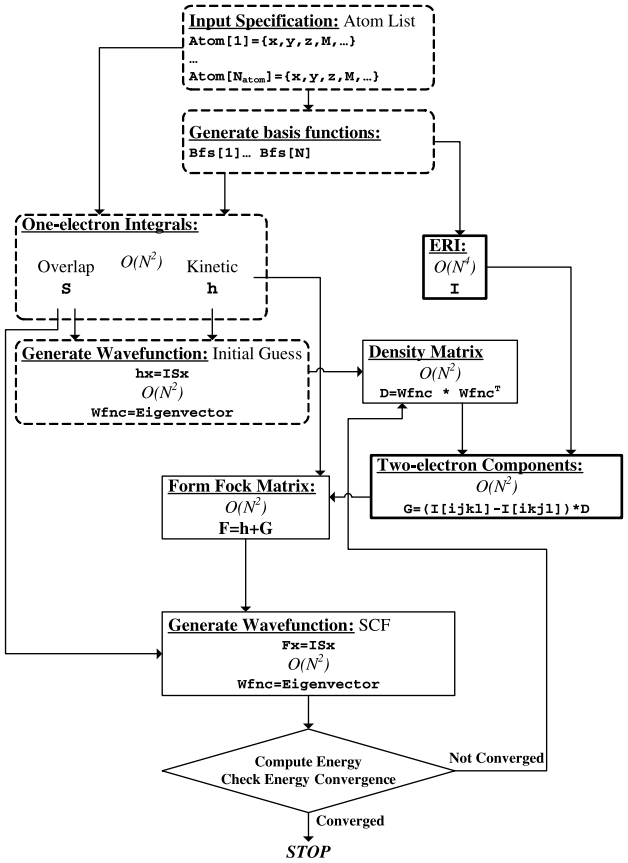

Fig. 2 SCF procedure. *Dashed blocks* indicate computations that need to be done only once, while *solid blocks* indicate computations that are done for all SCF iterations, i.e. until the energy convergence criteria has been satisfied. The *bolded blocks* are the focus of the present work. Order of complexity specified in each *block* indicates the storage requirements of the resulting data structure(s). There are many possible variations of the procedure (particularly the kernel in the two-electron components block); what is depicted here is merely one possible variation, though the general flow indicated is fairly consistent for all variations

```
for (i=0 to N){
  for (j=0 to N){
    G[i,j]=0;
    for (k=0 to N){
      for (l=0 to N){
        temp=2*ERI(i,j,k,l)-
             0.5*ERI(i,k,j,l)-
             0.5*ERI(i,l,k,j);
        G[i,j]+=temp*D[k,l];
```

Fig. 3 Two electron component construction. This code is simplified for clarity, and applies to the *Restricted Hartree–Fock* case. With the direct approach, the ERI subroutine would calculate the ERIs; with the stored-integral approach, the ERI subroutine would access a stored-ERI 4-index array

Table 1 ERI and G matrix construction computation time [23]

Molecule	Basis Functions	ERI and G matrix time
Nitrobenzene	91	87% of total
Morphine	124	98% of total

4.2 Basis functions

The representation of the molecular system in the above numerical procedure is a set of approximate one-electron orbital functions. For atoms, these are typically the orbitals for the hydrogen atom, while for a molecule, the initial approximation is a linear combination of atomic orbitals (LCAO) specially formulated in terms of a Slater determinant to satisfy the anti-symmetric nature of a molecular electron system. The basis set (LCAOs) is the set of mathematical functions, which are finite in number and orthogonal in construction. The basis functions, in simple terms, represent probability clouds describing the distribution of the electrons. Various basis sets are used in practice, most of which involve the use of Gaussian functions, as described below. Whenever discussing the performance scaling of quantum chemistry computations, N almost always refers to the number of basis functions describing the molecule. With this measure, assuming no further approximations, problems involving 1,000 basis functions are typically very large for most considerations, although not infeasible with current computer architectures. For the purpose of our work, we target a problem size of >10k basis functions.

The basis functions have a Gaussian form $\eta = C(x - R_x)^{n_x}(y - R_y)^{n_y}(z - R_z)^{n_z}e^{-\zeta(r-R)^2}$, and this is called a *Gaussian-type orbital* (GTO), also often referred to as a *primitive function* or *Gaussian primitive*. The characterizing elements for each function η are:

- Center coordinate – r:{x,y,z} [Real × 3]
- Exponent – ζ [Real]
- Angular momenta – n_x, n_y, n_z [Int × 3]
- Coefficient – C [Real]

The angular momenta are an important component of the GTO in the sense that the computation time of individual ERI depend on the angular momenta of the four input GTOs. The physical significance of the angular momenta is related to the electron configuration over "spdf..." shells.

Typically, a set of GTOs are combined linearly, or *contracted*, to form a single *contracted*-GTO (CGTO). The level of contraction K specifies the number of primitive functions collected into the single contracted function—higher K generally leads to higher accuracy. The CGTOs χ are the basis functions. Each CGTO consists of a collection of GTOs, however all the GTOs share a common center coordinate and common angular momenta. A CGTO may be thought of as a container with the following elements:

- Center coordinate – $r : \{x, y, z\}$
- Angular momenta – n_x, n_y, n_z
- K *partial* GTOs (pGTO), which only consist of a coefficient and an exponent.

The ERI computation involves a quartet (4-tuple) of CGTO, where each CGTO may have a different level of contraction K. A typical range for K is $1 \rightarrow 6$. This is all summarized diagrammatically in Fig. 4. To get a rough idea of the correlation between molecule size and basis set size, please refer to Table 2.

4.3 Electron repulsion integrals

The input data for the ERI function has a Gaussian form. Integration over Gaussian functions allows the application of several mathematical rules that simplify the computation, such as the Gaussian product rule [22]. Mathematical derivations are outside the scope of this paper; interested readers are referred to the reference lists of the papers cited within this section. Rather, in this section we describe the broad characteristics common to many ERI algorithms, and then elaborate one specific ERI algorithm in detail.

The computation time for each ERI is determined by the level of contraction and the angular momenta associated with each CGTO, with higher values implying higher necessary computational effort. Most ERI algorithms consists of two broad steps:

- Initial "*bootstrap*" step.
- *Recurrence* step.

As far as possible, complex arithmetic operations (such as exponential and square root evaluations) are limited to just the bootstrap step, which often executes in constant time. Conversely, the recurrence step—which may require up to $O(N'^2)$ time, where N' is related to the angular momentum of the input GTOs—is limited to multiplication and addition operations as far as possible. Several salient points regarding the computational requirements of ERIs, common to all ERI algorithms, are:

- All ERI are independent of each other, therefore, all ERIs may be computed in parallel.
- The computation time for individual ERIs may vary substantially.

The selected ERI algorithm is the *Rys quadrature* [24–26], which is an efficient yet generally applicable ERI

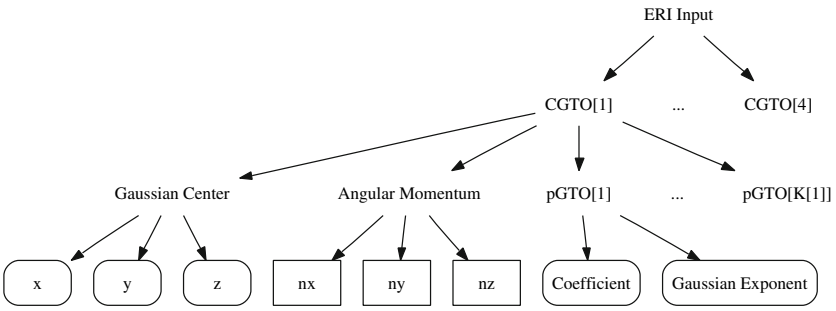

Fig. 4 ERI input. Typical range for K is $1 \rightarrow 6$. All pGTO/CGTO components are real values except the angular momentum $\{nx, ny, nz\}$ triplet, which are unsigned integers

Table 2 Basis set size of various molecules (631** basis)

Molecule	Basis functions
Water, H_2O	25
Nitrous oxide, N_2O	45
Butane, C_4H_{10}	110
Nicotine, $C_{10}H_{14}N_2$	250
Fullerene buckyball, C_{60}	900
Chlorophyll a, $C_{55}H_{72}O_5N_4Mg$	1,339
Azurin II (PDB ID:*1DYZ*)	14,395

algorithm.[2] The implementation of the algorithm is also compact and relatively simple, which are desirable traits for a dedicated hardware implementation—in fact a dedicated hardware implementation of this algorithm was suggested as far back as 1990 by Auspurger et al. [26], who specifically cite the concise/simple nature of the algorithm (and especially their particular implementation) as making it a good candidate for specialised hardware. Like many other ERI algorithms, Rys quadrature consists of a *bootstrap* stage and a *recurrence* stage. In addition, there is also a *roots and weights generation* stage.

Rys quadrature is a form of Gaussian quadrature, involving a N' point numerical quadrature using Rys polynomials—specifically the roots u and weights W of a polynomial of order N', where N' is related to the total angular momentum of the four GTOs. The standard polynomial generation techniques typically employed for Gaussian quadratures may be employed, and the *discretized Stieltjes* [27] method is known to be especially good (in terms of computational efficiency and accuracy) for Rys quadrature. However, because the angular momentum is low in practice (we target N' up to 13, which is sufficient for most practical computations[3]) it is possible to partially tabulate values to generate the required roots and weights with less computational effort. Therefore, we will not elaborate on polynomial generation methods here but will cover this in a future communication.

The *Gaussian primitive* ERI problem is expressed as:

$$[\eta_\mu \eta_\nu | \eta_\lambda \eta_\sigma] \equiv 2(\rho/\pi)^{1/2} \sum_{\alpha=1}^{N'} I_x(u_\alpha) I_y(u_\alpha) I_z(u_\alpha) W_\alpha \quad (2)$$

Beginning with the bootstrapping, one obtains:

$$I_x(0,0,0,0,u) = \frac{\pi}{\sqrt{AB}} \exp\left(-\frac{\zeta_i \zeta_j}{\zeta_i + \zeta_j}(x_i - x_j)^2 - \frac{\zeta_k \zeta_l}{\zeta_k + \zeta_l}(x_k - x_l)^2 \right) \quad (3)$$

Recall that ζ are the exponents associated with the GTOs. The evaluation of A and B are given in Eqs. 20 and 21 of [25]. Next the constants B_{00}, B_{10}, B'_{10}, C_{00} and C'_{00} (see Eqs. 41–43 of [25] or 12–14 of [26] for details) are evaluated, requiring the quadrature root u. However, u is not required in subsequent stages, so the u dependence can be omitted in expressions of the I-factors.

The following set of recurrence relations is then computed:

$$\begin{aligned} I_x(n+1,0,m,0) &= nB_{10} I_x(n-1,0,m,0) \\ &\quad + mB_{00} I_x(n,0,m-1,0) \\ &\quad + C_{00} I_x(n,0,m,0) \quad (4) \\ I_x(n,0,m+1,0) &= mB'_{10} I_x(n,0,m-1,0) \\ &\quad + nB_{00} I_x(n-1,0,m,0) \\ &\quad + C'_{00} I_x(n,0,m,0) \quad (5) \\ I_x(a_x,b_x,m,0) &= I_x(a_x+1,b_x-1,m,0) \\ &\quad + (x_b - x_a) I_x(a_x,b_x-1,m,0) \quad (6) \\ I_x(a_x,b_x,c_x,d_x) &= I_x(a_x,b_x,c_x+1,d_x-1) \\ &\quad + (x_d - x_c) I_x(a_x,b_x,c_x,d_x-1) \quad (7) \end{aligned}$$

[2] Some algorithms are limited to only low angular momentum basis functions, which means the algorithm is only applicable to "sp" shells for example.

[3] If we later find we need to increase N' beyond 13, there would exist a greater degree of parallelism in the computation, which would place greater gains on the table. Unfortunately it would also mean a greater level of *imbalance* to deal with—see Sect. 6.2 for clarification.

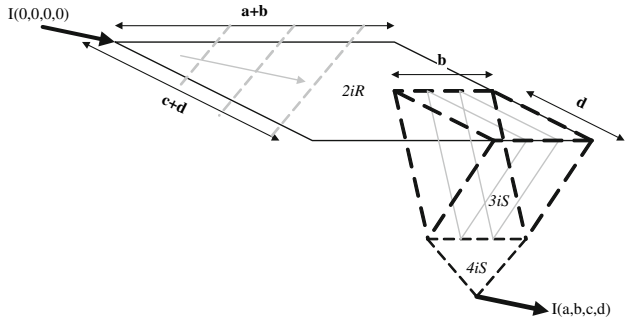

Fig. 5 Rys quadrature recurrence relations. This figure depicts the manner in which the three stages—*2-index recurrence* (2iR), *3-index shift* (3iS) and *4-index shift* (4iS)—are related. Note that a, b, c, and d are related to the angular momentum of the four input GTOs, and a typical maximum value for any of the four non-negative integers is 6

```
for (q=0 to CGTO1.K){
  for (r=0 to CGTO2.K){
    for (s=0 to CGTO3.K){
      for (t=0 to CGTO4.K){
        p_eri(CGTO1.xyz, CGTO1.coeffs[q],
              CGTO1.angmom, CGTO1.exp_1[q],
              CGTO2.xyz, CGTO2.coeffs[r],
              CGTO2.angmom, CGTO2.exp_1[r],
              CGTO3.xyz, CGTO3.coeffs[s],
              CGTO3.angmom, CGTO3.exp_1[s],
              CGTO4.xyz, CGTO4.coeffs[t],
              CGTO4.angmom, CGTO4.exp_1[t]);
```

Fig. 6 ERI with CGTO. The idea is to simply iterate over the pGTO contained in each of the four CGTO. Recall that typically $K \leq 6$

Equations 4 and 5 together populate a discrete two-dimensional plane, the width and breadth of which are a function of total angular momentum (we elect to limit the maximum dimensions to 13×13); this stage is referred to as the *2-index recurrence* (2iR). Next, a set of reduction operations is performed to effectively "shift" the dependence from two indices to all four indices. Equation 6 essentially reduces values along partial strips of the 2iR plane; this stage is referred to as the *3-index shift* (3iS). Finally, Eq. 7 performs a reduction to similarly "shift" the dependence to all 4 indices, and is termed the *4-index shift* (4iS). Figure 5 illustrates the computation of the recurrence relations. Observe the workload dependent nature of the 3iS and 4iS stages. The "footprint" of the 3iS reduction tree set is input-dependent in terms of dimensions and is always aligned to the lower right corner of the 2iR plane, but the dimensions of the 2iR plane itself are also workload-dependent. The width of the 4iS reduction tree depends on the width of the 3iS "footprint". This is repeated for the y and z axes.

Thus far, only the Gaussian primitives (GTOs) have been dealt with. The contracted functions (CGTOs) must also be considered. To do this, one may simply iterate over all the pGTO contained within a CGTO, as depicted in Fig. 6.

4.4 Electron correlation

The usefulness of our system would be limited if electron correlation computations were not also accelerated. We have focussed primarily on Hartree-Fock computations because it is a necessary first step, however it would not be difficult to expand the scope of the machine to also serve post-HF computations. This is because even with post-HF computations, ERI are the hot-spot, and in general there are computational similarities between HF and post-HF methods.

Let us consider the four-index transformation, an important component of electron correlation computations, which scales $O(4N^5)$ with basis size [28]. The four-index transform operation requires four terms with similar form to the following:

$$[iq|rs] = \sum_{p=1}^{N} T_{ip}[pq|rs] \quad i, q, r, s = 1, ..., N \tag{8}$$

Once again the heart of the problem is generating, evaluating, and contracting integrals. The requirements of this stage are therefore similar to the requirements of HF. We will conduct more rigourous analysis of different post-HF methods in a later communication.

5 Architectural reviews

There is considerable literature on parallelization of the HF algorithm for mainstream parallel computers (and readers are referred to [29] for a review on this). With reference to the present work, we consider only those that are immediately relevant.

Here we briefly discuss general-purpose processors (note that this does not limit us to *mainstream* processors) and the application specific ERIC processor which, as far as we are aware, is the only special-purpose processor for ERI reported to date.

5.1 General-purpose processors

The Cell Broadband Engine (Cell BE) [5] supports only single-precision arithmetic directly, which is, as we shall emphasise in Sect. 8, insufficient for the target application. It's performance on double-precision arithmetic is relatively poor.[4] Williams et al. have proposed microarchitectural improvements that would significantly improve double precision floating point arithmetic performance by a factor of

[4] See Table 2 of [5]: for dense matrix multiplication, performance for single precision is 204.7 GFLOPS while performance for double precision is 14.6 GFLOPS, a difference of more than an order of magnitude.

5 for dense matrix multiplication [5], but it remains to be seen if these modifications will materialise. This leads to our first motivating point: *commodity architectures tend to chase commodity applications*. While it is a probable that several architectural variants of the Cell may emerge, these are likely to be motivated by high-volume commodity applications such as media and network processing, video games, and consumer electronics which are adequately served by single-precision arithmetic. It seems unlikely that additional functionality on the Cell will be directed at double-precision aritmetic.

Unlike the Cell BE, the ClearSpeed CSX accelerators are targeted squarely in the scientific computation market with a claimed peak performance of 25 GFLOPS double-precision [6]. It is our view, however, that the 64bit wide memory interface on these multithreaded array processors may be limiting. The fixed architecture of ClearSpeed family does still require programmers make the application fit the processor.

The sheer number of ERIs presents challenges, but coupled with the total parallelism between individual ERIs, there are also opportunities. One such opportunity is the capacity for latency hiding. This may be easily exploited with a multithreaded architecture such as TERA MTA [30]. The TERA architecture has its origins in the dataflow class of computer architectures that are known theoretically to be able to exploit all concurrency in an application and are latency tolerant. To achieve this, such architectures require an abundance of concurrency in their workloads. The overall architecture is a symmetric shared memory multiprocessor with deep pipelines and no conventional cache structure.

For the TERA architecture the ERI computations may be readily expressed as independent threads and sub-threads of computation. As the computation of the numerous ERIs does not proceed *lock step*, contention on relatively infrequent memory accesses is spread over time as the TERA automatically adapts to the applied load.

There has been relatively little recent success in exploiting vector processing. The amount of data-level parallelism inherent in each ERI computation is limited, frustrating vectorization of ERI. This appears to have been demonstrated by recent studies. Yahiro and Gondo [31] report speedup due to vectorization slightly higher than 2, while Obara and Saika [32] report speedups of around 5 utilizing their ERI method on a vector machine. This is not to say that these claims are contradictory, but rather that speedup is relative to the nature of the computation and its implementation. In any case, it is apparent that the problem does not immediately vectorize very well [33], though certain transformations such as integral sorting do seem to provide better results [34], and these shall be the subject of further work.

Nevertheless, there may still be a place for vector processing in the system. Matrix diagonalization is also an important part of the overall SCF workflow and vector processing is highly applicable there, as noted in [35,36]. Matrix diagonalization is outside our scope of interest but in light of this, a vector processing machine may be an ideal candidate for a host system, with the ERIs farmed out to slave nodes. This would make a substantial difference for small to medium sized computations.

The mapping of the target application to a clasic MIMD system based on the T800 *transputer* processor is presented in [23]. The evaluation of each ERI executes sequentially on each node. In addition, each node computes an element of the G-matrix, which is subsequently transferred back to the host system. This straight-forward arrangement is something we will emulate. The main point differentiating our proposal against other conventional MIMD systems is the design of the individual *node*, wherein we intend to exploit more concurrency (specific to the needs of the particular application) than previously designed mainstream architectures are capable of. This goal was also shared by the workers of the ERIC project. We review their efforts in the next section.

5.2 The ERIC special-purpose processor

The ERI Computer (ERIC) is the only known system specifically constructed to compute the ERIs and thus deserves mention. This processor was developed at Kyushu University, in collaboration with industry and government partners, and is comprehensively documented in [18,37,38]. ERIC was implemented with 0.13 μm process technology and operates at 200 MHz—a similar clock rate to last generation FPGAs. The Obara-Saika (OS) ERI algorithm was selected, which is similar to the Gauss-Rys quadrature algorithm.

As presented in [37], OS consists of two stages: an *initial integral calculation* stage (the bootstrap stage), and a *recurrence calculation* stage. The bootstrap stage has limited instruction level parallelism (ILP) and requires evaluation of complex floating-point arithmetic functions. The recurrence stage has a high degree of ILP, and basically only requires floating-point multiplication and addition. Since there are two distinct stages with distinct computational requirements, ERIC consists of two distinct parts, namely an *Initial Integral Calculation Engine* (IICE) and a *Recurrence Calculation Engine* (RCE). Both parts share a bus to memory and communications ports.

The IICE is basically a MIPS RISC core augmented with specialized arithmetic units. These include units for evaluation of:

- Square root.
- Exponential.
- "Error function" required for a Taylor series.

as well as division, addition, and multiplication. Unfortunately, for this stage of the computation, the authors find that

their current implementation is substantially slower than a Pentium 4 [38]. However, they estimate the power efficiency of their implementation is 1.5 times higher than the Pentium 4 [38].

Ideally, the RCE would consist of a large number of floating-point multipliers and adders to leverage the large degree of ILP available in this stage. However, the authors argue that this would not be practical mainly because very wide multiport register files are inefficient in terms of both area and delay [37]. This problem is overcome by introducing the notion of *subengines*. A subengine consists of a register file, a load-store unit, and arithmetic units. The RCE consists of multiple subengines, and resembles a VLIW architecture, and the authors prescribe 4 subengines within the RCE [38]. This means the RCE has a maximum parallelism of 4 multiply and add functions. Here too they find that their implementation is substantially slower than the Pentium 4 [38].

The authors have identified the bottlenecks in their implementation [38]. A chief cause turns out to be the memory bandwidth available to the RCE, which limits the IPC of a single subengine to 0.07 on average. The authors propose several improvements to the memory system that would improve the power efficiency of the ERIC to 4.58 times over the Pentium 4.

For a range of test computations, the authors report that their implementation is >35 times slower than the Pentium 4 [38]. Although the authors express disappointment with these results, important ground-breaking work has been illustrated such as algorithmic modifications. More importantly, the authors have yet to report the performance of their overall parallel system—the cost efficiency of their solution has not yet been reported: if the ERIC is favorable in this regard, as opposed to the Pentium 4, it would be more appropriate to compare a network of ERICs to a network of Pentiums of equivalent cost, which may result in a more practically accurate, and possibly more positive, assessment.

6 Architecture exploration

In Sect. 5 the overall computation was mapped onto several general-purpose systems. One exception to this was the ERIC processor that in principle involved a special-purpose design. As far as we are aware, thus far (1) there has not been a comprehensive architectural analysis of the overall computation, and (2) there has been no special-purpose work touching specifically on the Rys quadrature ERI algorithm. In this section we consider each computational component of the ERIs and G matrix construction in a hierarchical manner, and we attempt to propose architectural designs that are appropriate for the computation yet practical from an implementation stand-point [4].

It seems prudent to first consider the behavior of the overall HF program formally in the context of Ahmdal's well known *Speedup Law*:

$$\text{Speedup} \leq \frac{100}{\text{Serial } \% + \frac{\text{Parallel } \%}{\text{Parallel nodes}}} \tag{9}$$

The target subset of the overall algorithm, i.e. the parallel/enhanced fraction of the program, grows approximately with basis set size at $O(N^4)$, while the matrix diagonalization component which we suggest be run on a host system generally scales as $O(N^3)$—this means that as problem size increases, "Serial %" decreases and "Parallel %" increases,[5] which means the maximum potential speedup increases as well. This hypothesis will be validated with a "back-of-the-envelope" analysis.

The two dominant aspects of the computation are the generation of the Fock matrix—$O(N^4)$—and the diagonalization of the Fock matrix—$O(N^3)$. The two-electron component aspect of the Fock matrix, i.e. the G-matrix, dominates the Fock matrix construction time, and is in fact the sole $O(N^4)$ component (i.e. $O(N^4)$ ERIs are computed). Thus, the "Serial %" as a function of basis set size N may be expressed as:

$$\text{Serial } \% = \frac{100 \times O(N^3)}{O(N^4) + O(N^3)} \tag{10}$$

In order to determine the upper bound on potential speedup, let there be infinite resources available: i.e. let "Parallel nodes" $\rightarrow \infty$. Speedup then becomes:

$$\text{Max. Speedup} = \frac{100}{\text{Serial } \%} \tag{11}$$

Substituting 10 into 11, it is apparent that the maximum speedup roughly scales linearly with basis set size.

Recall Table 1. The maximum speedup for the 91 basis function Nitrobenzene computation is 7.7 and the maximum speedup for the 124 basis function Morphine computation is 50. The findings presented in Table 1 of [18] indicate a maximum speedup of 155 for a 427 basis problem. Extrapolating these results linearly would clearly lead to very positive results about the potential gains on the table, though one would perhaps be over-optimistic in doing so, since one would be neglecting the effect of practical optimizations such as symmetries and cut-offs. However, it does appear that at least 2 orders-of-magnitude speedup is available just by focusing on the construction of the G-matrix.

Nevertheless, the rest of the computation ought not to be taken for granted. Fortunately some of the "Serial %" of the algorithm may be hidden by overlapping communication and computation between the host and the ASP. This will be elaborated on in Sect. 6.1.

[5] This notion is empirically supported by Table 1.

The first task at hand is to partition/allocate the computation between the host system and the application specific processor (ASP), and a top-level architectural qualitative discussion. Then consideration of the individual components of the computation, with a particular emphasis on the Rys quadrature recurrence relations, will be made.

6.1 Partitioning

There are many automated hardware/software co-design and partitioning tools/methodologies available—far more than can be cited to do such a thriving research area justice, though [39,40] are examples of a well-established work and a relatively recent work. In the present case, because we are willing to go "back to the drawing board" as far as the implementation of the algorithms are concerned the partitioning can be tackled from an analytical level, thereby avoiding encumbrances of any existing code-base that could frustrate automated tools and result in a sub-optimal solution. Similar frustrations have been known to impede automatic vectorizing compilers [33].

Consider the overall computation of the G matrix (Fig. 3). The data required to perform this computation are the $O(N^2)$ density matrix and the $O(N)$ basis set,[6] and the output is the $O(N^2)$ G matrix. This involves $O(N^4)$ ERI computations, each of which yields a scalar value, which are then reduced by way of a dot product computation with the density matrix to yield a single component of the G matrix. First of all, due to the sheer number of ERIs, all individual ERI computations are done completely on the ASP, and therefore we may elect to implement the complete program in Fig. 3 on the ASP. Analytically one can see this requires $O(N^4)$ computation per iteration and the data transfer requirements are limited to $O(N^2)$ per iteration from host to ASP (the density matrix) and $O(N^2)$ per iteration from ASP to host (the G matrix), yielding a *computation vs communication* ratio of $O(N^4)/O(N^2) \approx O(N^2)$.[7] This is similar to the *computation vs communication* ratio of the MD-GRAPE solution.

The data transfers can be scheduled in a manner that hides communication latency as well as a portion of the sequential computation performed on the host system. We can stream the density matrix from the host to the ASP while computations proceed on the ASP. Conversely, we can stream G matrix elements from the ASP to the host as they are computed; doing so allows the construction and subsequent diagonalization of the Fock matrix to proceed on the host concurrently while the rest of the G matrix is computed on the ASP.

Table 3 G matrix nested loop partitioning Comp./Comm

ASP loops	Data in	Data out	ERI calls	Computation/ communication ratio
i,j,k,l	$O(N^2)$	$O(N^2)$	$O(N^4)$	$\approx O(N^2)$
j,k,l	$O(N^2)$	$O(N)$	$O(N^3)$	$\approx O(N)$
k,l	$O(N)$	$O(1)$	$O(N^2)$	$\approx O(N)$
l	$O(N)$	$O(1)$	$O(N)$	$\approx O(1)$
inner kernel	$O(1)$	$O(1)$	$O(1)$	$\approx O(1)$

It is important to consider each of five possible options:

1. Computation of a single ERI—corresponding to the inner-most code kernel.
2. Computation of a partial value of a G matrix element—corresponding to the {l} loop.
3. Computation of a single G matrix element—corresponding to the {k,l} loops.
4. Computation of a single G matrix column—corresponding to the {j,k,l} loops.
5. Computation of the entire G matrix—corresponding to the {i,j,k,l} loops.

We summarize our findings in Table 3. "Data in" is limited only to the volume of density matrix data that is required in each iteration; since the basis set data may just be loaded onto the ASP once, it is not included in the "Data in" tally. "Data out" refers to the volume of G matrix data that can be returned to the host. When it comes to the computation cost, only the number of ERI operations is counted because ERIs dominate the cost of the G matrix computation. The metric of interest is the "Computation/Communication" score which provides a simple measure of the amount of work done relative to the amount of data required to do the work: it is desirable to maximize this score since it is far easier to provision more memory than it is to provision high performance data transfer between the host and the ASP.

Therefore assuming one is not subject to any other constraints such as memory availability, or the desire to run some other computation within some loop level, the entire code in Fig. 3 is computed on the ASP.

6.2 Top-level system view

As noted by other works reviewed in Sect. 5, the high degree of function-level parallelism may be exploited through a MIMD architecture. One might argue that a SIMD architecture is applicable; indeed with the vast majority of computational resources being required for multiple instances of

[6] The basis set is only loaded onto the ASP at the start of the HF procedure.

[7] This has been confirmed experimentally by conducting a memory trace analysis on the relevant computational partition in GAMESS [19]; details on our specific methodology will be provided in a future communication.

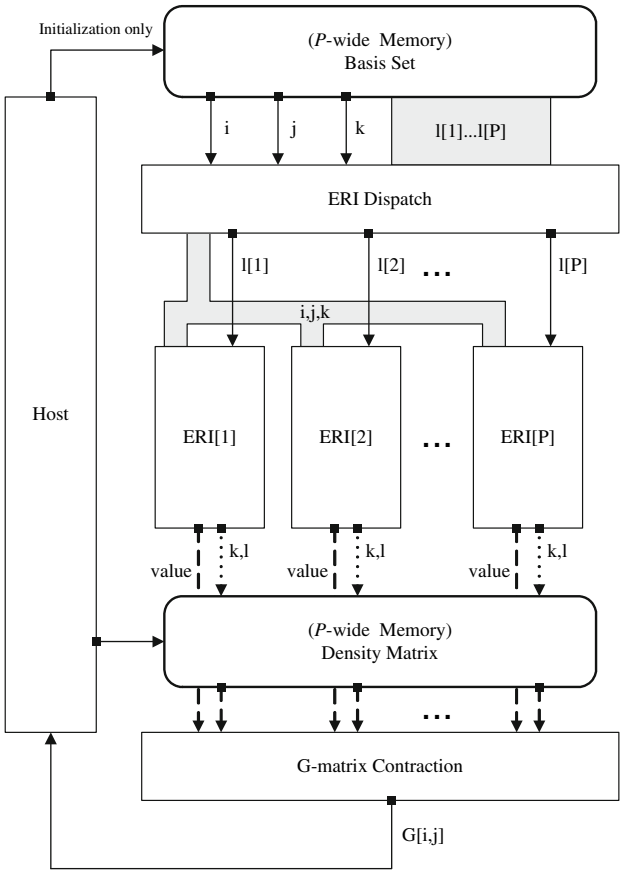

Fig. 7 Two electron component computer, top-level view. *Dotted arrows* indicate address buses, *dashed arrows* indicate data buses, *solid arrows* indicate data/address buses, *thick gray lines* indicate wide (multiple-word) data/address buses

```
for (i=0;i<N;i++){
  for (j=0;j<N;j++){
    G[i,j]=0;
    for (k=0;k<N;k++){
      for (l=0;l<N;l+=P){
        for (proc=0;proc<P;proc++){
          //compute
          temp[proc]=
                2*ERI(i,j,k,l+proc)-
                0.5*ERI(i,k,j,l+proc)-
                0.5*ERI(i,l+proc,k,j);
          tG[proc]+=
                temp[proc]*D[k,l+proc];
        }
        //reduce
        for (proc=0 to P){
          G[i,j]+=tG[p];
```

Fig. 8 Two electron component construction, inner-loop parallel. *P* ERIs are computed in parallel

the ERI program it would appear that a single instruction stream is characteristic of the computational load. However, one problem with ERI computation is the variability in computation time for ERIs based on the workload (specifically the angular momenta of the input GTO), and handling this imbalance efficiently requires that each computational unit be able to sequence the program execution autonomously based on the particular ERI workload. Being faithful to the design-space framework that we have adopted [4], systems which incorporate local sequencing control are categorized as MIMD.

As previously alluded to, we have chosen the same assumptions as the *Protein Explorer* researchers [3], in that the host system will consist of multiprocessor nodes interconnected to form a distributed-memory MIMD system and that some number of the multiprocessor nodes will have local access to the ASP.

We propose the top-level design depicted in Fig. 7. The high degree of function-level parallelism is supported by replicating ERI pipelines, with P replicated ERI computation pipelines in the ASP. There are several practicalities that limit P, particularly memory bandwidth, therefore $P \ll N$, and P ERI computations can be issued in parallel. A loop blocking/partitioning that is cache-friendly in the sense of a conventional processor is not required in this case, so one may simply break up the l-loop, as in Fig. 8.

MIMD systems come in two flavors: distributed memory and shared memory. Bear in mind that in this context MIMD concerns the ASP itself, and not the configuration of the host system. In this context, the memory access issue concerns the streaming of basis set data into each ERI pipeline and the reduction of ERI values with density matrix values into the G-matrix. Both flavors are workable solutions, however a decision regarding which approach is "better" requires more information—at the very least one would need to know how many ERI pipelines will be instantiated. Fortunately, this decision need not influence the design of the ERI pipeline itself, and one may guess (based on initial data) that the number of ERI pipelines will be reasonably small (the order of 10s at most). Furthermore, while feeding each ERI pipeline requires significant bandwidth due to the fact that the CGTO quartet are quite large, this cost is mitigated by the fact that one may feed identical CGTOs to each of the pipelines—i.e. one may employ a shared broadcast bus, as the MD-GRAPE does. The bandwidth required to contract ERI values with density matrix elements into G-matrix elements does scale with the number of ERI pipelines, but here the data in question are just scalar real values. We will tentatively (and marginally) favour a shared-memory implementation, as it affords us a bit more flexibility and adaptability. We may switch our preference to a distributed memory design later when we have more information available; doing so will not cost us anything.

Contemplating some aspects of the shared-memory MIMD design space [4]:

- Memory access: High throughput is required and high latency is tolerable. In addition, a priori knowledge of memory access patterns are available. As we wish to maximize P, uniform memory access (UMA) would not provide sufficient data bandwidth. However, one may easily distribute memory accesses in a uniform manner across memory banks to manage contention. Cache-coherent non-uniform memory access (ccNUMA) and cache-only memory access (COMA) are not necessary. With properly organized data structures, NUMA should provide the necessary scalability and performance required.
- Interconnection scheme: There are several layers in our proposed system:
  1. Basis set memory → ERI Dispatch.
  2. ERI Dispatch → ERI processors.
  3. ERI processors → density matrix memory and G-matrix-Contraction.

  We will first consider layers 1 and 2. Going on the assumption that due to practicalities the inequality $P \ll N$ holds true, one expects that in general any one of the ijkl indices will generate enough parallelism to comfortably utilize the P-way replicated parallelism on the system for N/P cycles. Therefore one may provision a shared bus for 3 CGTOs, and P point-to-point buses (or perhaps clusters of wide shared buses) to feed a unique CGTO to each of the P ERI processors. As for 3, each ERI pipeline produces a unique ERI result, therefore P point-to-point buses ought to be provisioned to address the density matrix memory to obtain the corresponding density matrix element D_{kl}, and subsequently for the ERI result and D_{kl} to be fed into the G-matrix Contraction unit.
- Cache coherency: Host processors will stream the ERI parameters to the ASP with G-matrix elements being streamed back, after the ASP pipeline delays, for the host processors to construct the Fock matrix and compute the new density matrix. In practice this will most likely be implemented through direct memory access (DMA) transfers from memory. The host processor is thus responsible for cache coherency which may be minimised using appropriate access privileges to regions of virtual memory.

Besides the ERI Processor pipelines, there are two noteworthy components depicted in Fig. 7:

- *ERI Dispatch* (ERID): This unit issues CGTO quartet to each ERI pipe. In order to reduce bandwidth usage, the i, j and k CGTOs are issued to a shared bus that is read by all P ERI pipes, while P unique l CGTOs are issued directly to the P ERI pipes. Contracted basis functions are "uncontracted" here; the ERI Dispatch unit implements the functionality of the code in Fig. 6. This unit will be semi-programmable so as to be able to support HF variations (i.e. UHF, RHF, etc.), which correspond to differences in the exact procedure that ERI are combined to form a G-matrix element. The unit programming remains unchanged for the duration of the entire SCF computation.
- *G-matrix Contraction* (GmC): This unit sums and scales the generated ERIs to form the G-matrix, as in the inner-loop of Fig. 8. A note-worthy aspect of this unit is that due to the fact that ERI computation times are not uniform, the G-matrix contraction unit must implement some manner of asynchronous/wavefront/dataflow functionality.

These two components will receive more attention in a future communication.

To fully saturate the ERI processor pipelines implies a large number of of outstanding memory requests and consequent high latency. An effective approach to overcome this latency is to adopt a multithreaded approach to the ERI computation. Multithreading tolerates large memory latencies as long as the memory system has sufficient throughput. Similar characteristics have been noted in high-performance network processing, where a heavily pipelined memory hierarchy was deemed appropriate [41]. In our case memory accesses patterns are predictable and sequential, permitting a straight forward pipelined interleaved memory [42]. The main limitation is cost.

Furthermore, as each ERI is independent, there is no inter-ERI-pipe synchronization/communication required. However, there is an implied "barrier" at the GmC stage where the ERI values are contracted, but we do not need to stall ERI pipelines to enforce the barrier, instead we can provision buffers at the GmC processor and allow the ERI pipelines to proceed. We expect that over many cycles the imbalance of individual ERI computations would average across all the ERI pipelines—a similar observation was noted with the TERA MTA [30] which we reviewed in Sect. 5.1. The issue of variation in the cost of evaluating different ERI was reviewed by Harrison et al. [29], where loss of efficiency was ascribed to imbalances stemming from different individual ERI execution times. However, Harrison et al. observe, as do Bolding et al. [30], that the efficiencies improve as molecule size grows. This is due to the fact that with a sufficiently large population of ERI tasks, and assuming that $N_{ERI} \gg P$, there would be a balance in the distribution of ERI classes to each ERI Processor. If adequate buffering is employed at the ERI pipeline inputs and outputs there would be no need to stall pipelines as a consequence of execution imbalance between parallel pipelines waiting at the implied "barrier". A complementary approach would be for the ERI Dispatch unit to perform explicit load balancing to reduce the aggregate imbalance between ERI Processors, thus reducing the required buffer capacity. Performing this load balancing

would require the mapping of P ERI tasks to P ERI Processors, based on the current load of each ERI Processor and the cost of each ERI task.

The width of the buffers on both ends of the ERI Processor is manageable: each GmC input buffer entry needs to hold the ERI result and a density matrix element (2× real values), while each ERI pipe input buffer entry need to hold four GTOs (4× {5× real values + 3× integers}). Having said that, the latter while manageable is certainly not trivial, so we should strive to minimize the imbalance, or variance, in computation time for each ERI because greater imbalance results in larger buffering requirements. We need to illustrate where and how the imbalances arise—we therefore turn our attention to the ERI Processor.

6.3 Top-level ERI processor view

To achieve order-of-magnitude speedup over mainstream general-purpose systems, at least one (preferably both) of these two conditions must be satisfied:

- P must be large.
- The throughput of each ERI pipeline T_{pipe} must be large.

Both P and T_{pipe} would of course be constrained by resource limits. The total throughput of the ASP is dictated by both factors, i.e. $T_{\text{ASP}} \propto P \times T_{\text{pipe}}$, therefore as clearly anticipated we seek to maximize both P (i.e. we want to maximize the number of ERI pipelines) and T_{pipe} (i.e. we want to maximize the throughput of each ERI pipeline). Because P is bandwidth limited, let us initially consider how to maximize T_{pipe}.

We proceed by revisiting the chosen ERI algorithm—Rys quadrature—which we discussed in Sect. 4.3. The computational workflow is depicted in Fig. 9, and the individual components have the following function and requirements:

1. *Preliminary bootstrapping* (bP)—Several constants and other values are computed here. This stage is a static graph; there are no control requirements. The Gaussian product rule is applied here, and the distance between the two virtual centers XX is subsequently determined. The number of required polynomials N' (which is related to the angular momentum of the input GTOs) is also determined here. A quantitative analysis of this stage is presented in Sect. 7.
2. *Roots and Weights* (RaW)—This is the first of the two stages that is responsible for the variance in the computation time for an ERI, because N' roots and weights are generated (where N' ranges from 1 to 13). One may elect to use a read-only look-up table (that may be shared by multiple pipes) and interpolate the exact values required. This will be subject to further investigation.

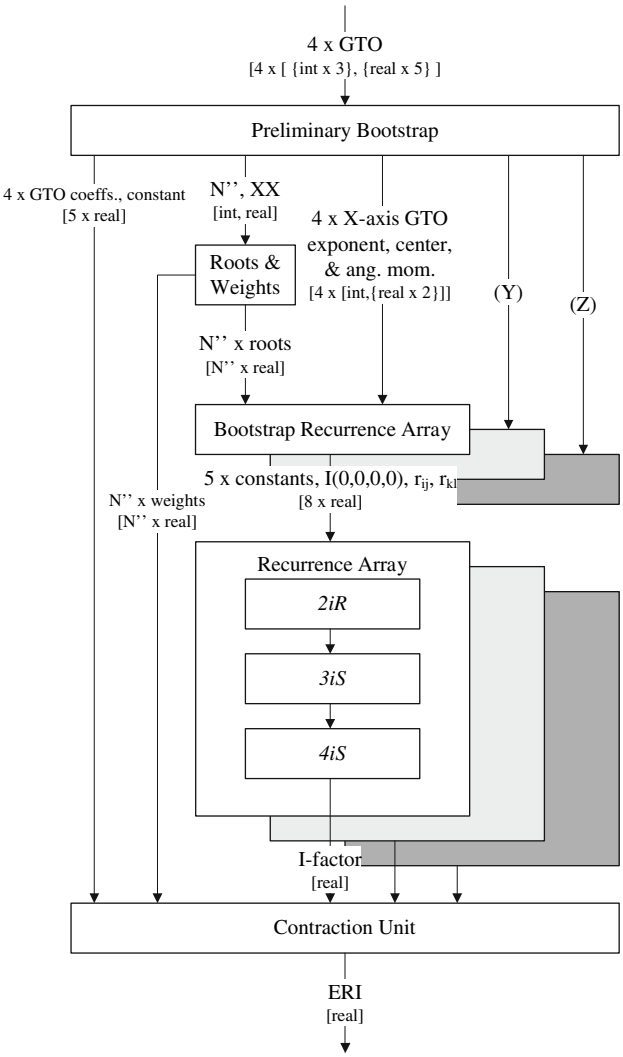

Fig. 9 Rys quadrature ERI workflow

3. *Bootstrap Recurrence Array* (bRA)—Several constants and other values based on the quadrature roots are computed here. This stage computes several values including the initial value $I(0, 0, 0, 0)$ required by the recurrence relations part of the routine. Computation of an exponential term is required here. This stage is a static graph; there are no control requirements. This stage is required for each of the three axes, and computes different values for each of the N' roots.
4. *Recurrence Array* (RA)—This is the second and more dominant source of variance in the computation time for an ERI. Here, the recurrence relations (Eqs. 4 – 7) are computed. Once again, the computation time variance is due to angular momentum: the required "depth" of the recurrence relations is due to angular momentum. The computation of the recurrence relations is super-quadratically dependent on angular momenta, as graphically expressed in Fig. 5. This stage is subject to further elaboration later in this section.

5. *Contraction Unit* (CU)—Contract the three axis I-factors (with GTO coefficients and other constants) to form a single primitive Gaussian ERI, and then contract primitive ERI into contracted ERI (i.e. the functionality depicted in Fig. 6, though the iterations over pGTO contained within a CGTO is effectively handled by ERID). This stage is almost a static graph; there is a minimal control requirement, specifically the contraction accumulation register needs to be reset before the start of a new contraction.

A characteristic common to all stages is the reliance on floating-point arithmetic.

Maximizing the throughput of each ERI pipeline T_{pipe} means maximizing the number of ERI computations completed within a given period of time. This may be achieved by doing one or both of the following:

- Minimize the average ERI computation time, $\bar{t}_{\text{ERI}}$.
- Maximize the average number of ERI computations in-flight in the ERI pipeline in any given cycle, $\bar{n}_{\text{ERI}}$.

Let us consider the variance of the ERI computation time. The upper-bound of the computation time varies with angular momenta. The lower-bound of the computation time is due to minimum angular momenta, and is basically fixed. If a hard lower-bound is established, then minimizing $\bar{t}_{\text{ERI}}$ increases T_{pipe}, and it also effectively minimizes the variance in ERI computation time, which reduces the required buffer lengths in the ERI Processor inputs and GmC inputs. The variance is due to the two components of the algorithm that have a workload-dependent amount of computation—RaW and RA.

We first pursue the minimization of $\bar{t}_{\text{ERI}}$. We start by considering the aspects of the algorithm that have a workload-dependent amount of computation—specifically RaW and RA.

Analytically one expects that the RA component would be a greater computational hotspot than RaW (this assumption has been validated empirically through performance profiling). The RA workload scales quadratically with the nature of the input (specifically the angular momenta) for a single scalar result—i.e. $O(N^2)$ work for 1 data output. One's efforts into $\bar{t}_{\text{ERI}}$ minimization should therefore be initially focused on the RA component. This will be discussed in Sect. 6.4.

Several arguments could be made for a multithreaded architecture in the context of the application. First of all, through sheer number of independent ERI, the application inherently has much thread-level parallelism. Secondly, though memory latencies are tolerable in the application (and furthermore the access patterns are known a priori), there are other forms of latency, such as pipeline data dependencies. These are particularly relevant in the application because of the floating-point computational intensity, and floating-point arithmetic units typically have deep pipelines. Through thread-level parallelism, one can avoid data dependency hazards and achieve better utilization of floating-point arithmetic units and other logic. This point is demonstrated quantitatively in Sect. 7.

Contemplating some aspects of the multithreaded design space [4]:

- Granularity: *Fine-grained thread-interleaving* implies that a high number of different threads are active in various stages of the pipeline at any cycle. Potential pitfalls with this are:
  - Poor single-thread performance; this is of little concern, as the main goal is high overall throughput.
  - Low processor utilization when there are insufficient active threads; this is highly unlikely to be the case with the application, where there is a huge number of ERI, and issuing the ERI computations should not prove problematic because the memory requirements seem tractable, as discussed in Sect. 6.2.
- Number of threads: because there are a huge number of threads available, it makes sense to exploit as many of these as is convenient, thus enjoying the maximum degree of latency hiding.

The second point leads us to the question: *how many threads per processor*? At this stage this remains an open question, but it is likely that the number of threads will be closely related to the number of pipeline stages, constrained by logic space. In order to achieve high memory-system throughput the number of concurrent outstanding memory references must exceed the product of bandwidth and memory latency [42]; this would set the lower limit on the number of active threads that must be in-flight at any instance in order to make effective use of the memory system.

There is, however, one danger with a multithreaded architecture. Efficient precise exception handling is somewhat complex with heavily multithreaded architectures. With the target application there may be the need to handle arithmetic exceptions. At this point it is unclear how critical a factor this will be.

Although our discussion of multithreading has been conducted within the confines of the ERI Processor specifically, the architecture cannot be applied to this stage in isolation. The success of a massively multithreaded processor hinges on the ability of the rest of the system to keep the processor fed with threads. Therefore the design of the entire system should accommodate multithreaded architecture.

Another approach to maximize $\bar{n}_{\text{ERI}}$ is to embrace pipelining. Since individual thread performance is not an issue and a large number of threads is available, a heavily pipelined architecture would be synergistic with a heavily multithreaded architecture, though a large number of pipeline stages and

large number of concurrent threads both increase the *state storage* overhead of the system. Nevertheless, heavy pipelining in the ERI Processor is a means to both increase $\bar{n}_{\mathrm{ERI}}$ as well as increase the clock rate, and therefore decrease $\bar{t}_{\mathrm{ERI}}$. Contemplating some aspects of the pipelining design-space [4]:

- Dependency resolution: One may leverage application specific insight to statically resolve dependencies, or one may rely on large thread-level parallelism to automatically resolve dependency hazards. In either case, dependencies are effectively resolved statically.
- Number of stages: At the sub-task level there are a moderate number of stages (bP → RaW → bRA → RA → CU), each of which exhibits differing degrees of parallelism. Each of these stages in turn would be internally pipelined, thus yielding a very deep overall pipeline.
- Stage sequence: The computation proceeds sequentially, however the RA and RaW stages introduce a degree of variability; with the RA stage in particular, handling this variability may require cycled operation. The downside of this is that while the RA is cycling, all the stages would be stalled—which is extremely undesirable with a very deep pipeline.

Pipeline depth is an important consideration, especially in light of the proposed cyclic operation of the RA stage (elaborated in Sect. 6.4). This architectural aspect requires care, and will be subject of further work.

6.4 ERI recurrence array

The graphical expression of the recurrence relations computation presented in Fig. 5 exposes the varying size of the computation as well as the varying level of parallelism—note the parallelism along the diagonal hyperplane on the 2iR plane. This variance reveals itself in the computation time of the entire ERI procedure, and is undesirable because buffers need to be used to counter the imbalance. Fortunately, as was illustrated in Sect. 6.3 minimizing $\bar{t}_{\mathrm{ERI}}$ – which is desirable in it's own right because doing so maximizes T_{pipe}—also minimizes the variance. The trade-off would be logic space.

When implemented on a scalar processor the recurrence relations exhibit $O(N'^2)$ time scaling, where N' has a maximum value of 13. One may reduce the time scaling to $O(N')$ by provisioning $O(N')$ processing elements (PEs). Alternatively, if one provisions $O(N'^2)$ PEs, one would still require $O(N')$ processing time per thread, however one would be able to compute $O(N')$ threads concurrently, thus improving $\bar{n}_{\mathrm{ERI}}$ and correspondingly the overall throughput T_{pipe}.

The "multiple PEs" concept fits within the framework of virtually every architecture. Some characteristics/requirements of the recurrence relations are as follows:

- Limited parallelism—i.e. up to 13-wide.
- There are three functions that each PE may need to perform, corresponding to the 2iR, 3iS, and 4iS functions (see Sect. 4.3).
- The computation time for each of the three functions does not vary with data, however the 2iR computation is different than the 3iS and 4iS computations (though 3iS and 4iS are identical; the only difference being the constants used). Global sequencing may be used.
- Regular dependencies—each PE has data dependencies to just two other nearest-neighbor PEs. This is true for all three functions.

With these traits in mind, the SIMD architecture[8] appears to be a suitable candidate. Contemplating some aspects of the SIMD design-space [4]:

1. Complete Fine-Grained Parallelism: Granularity in this context refers to the size of the data-set processed by each PE. Since the data-set is limited to 13-wide, assigning one data element to each PE is practical.
2. Near-neighbor connectivity: This simple fixed mesh connectivity scheme is sufficient for the recurrence relations.
3. Local Algorithm Autonomy: The PEs need to be able to switch between the three functions. Also, while the interconnection links remain unchanged for all three functions, the direction of data flow does depend on the function.

This SIMD array proposal is arguably the most novel aspect of our proposal.

7 Fine-grained parallelism

At the coarse grained thread level there is a huge amount of parallelism available. The downside to coarse-grained, in particular thread-level, parallelism is that in general, a moderate amount of state information has to be maintained for each thread, and keeping the processor fed with threads (as opposed to the processor being "thread starved" and therefore idle) requires large bandwidth. In this sense, fine-grained parallelism is "cheaper". Therefore, one ought to consider the degree of fine-grained parallelism exhibited by a single thread.

Let us consider just the *Preliminary Bootstrap* stage in detail. This stage is invariant to the angular momentum of the CGTO quartet. This stage requires short-word integer

[8] Another apparently likely candidate is the systolic architecture, but once again being faithful to the design-space approach [4] we accept the definition that systolic arrays cannot have autonomous PEs (i.e. PEs cannot execute different functions).

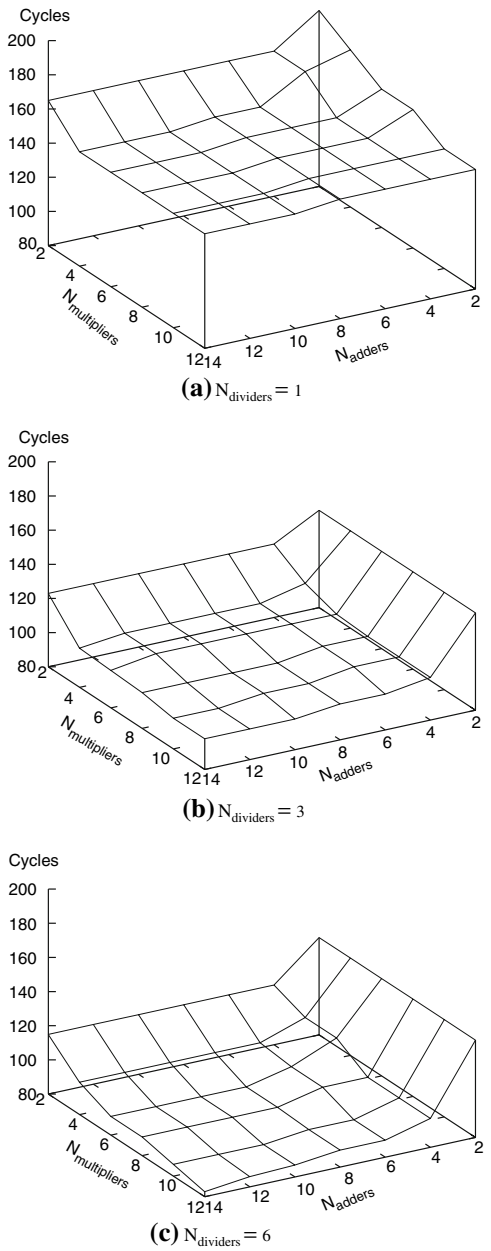

Fig. 10 Preliminary Bootstrap, available file grained parallelism. The number of available arithmetic units are varied as indicated on each of the three subplots, and the execution time (i.e. required cycles) of a single ERI is simulated

addition, floating point addition, floating point multiplication, and floating point division. The impact of varying the number of arithmetic units on the total required computation time has been estimated through discrete event simulation. Integer addition was omitted from our analysis because that part of the computation was not the critical section of the computation; varying the number of integer adders has no impact whatsoever on the computation time of a single thread. Our findings are illustrated in Fig. 10.

It may appear premature to draw any conclusions about fine-grained parallelism based on just one part of the program. This is not the case—the limits of fine-grained operation level parallelism inherent in the program is already observable. Recall Ahmdal's speedup law (Eq. 9): there is an *enhanced* (parallel) portion of the program and a *fixed* (serial) portion of the program. Let the Preliminary Bootstrap stage be the fixed portion and let the rest of the computation be the enhanced portion. Even if doubling the number of each functional unit results in doubling the throughput of the enhanced portion the overall throughput will still be limited by the throughput of the serial portion, i.e. the Preliminary Bootstrap stage.

In addition, this analysis can direct efforts towards designing the architecture of the system by indicating the "path of steepest descent". Armed with such plots, one has an idea of how best to utilise available logic area. This in effect extends the longevity of the architectural analysis because it provides a plan for leveraging increasing logic densities.

The main point we wish to emphasise is this: while there is some fine-grained parallelism inherent in the computation, one would achieve large speedup only be exploiting thread-level parallelism.

However, exploiting thread-level parallelism requires a significant logic investment for state storage and bandwidth. Therefore, one should not rule out fine-grained parallelism. Figure 11 reflects the effect of running two threads consecutively with the same allocation of functional units. Notice that the additional thread results in better performance, due to better utilisation of resources. It is likely that there exists an ideal balance between thread-level parallelism and operation-level parallelism, and we will investigate this issue further, especially by expanding the simulation to encompass the entire Rys quadrature computation.

8 Numerical analysis

One factor that has not been considered at all by works on general-purpose systems is the issue of arithmetic representation. When constructing a special-purpose computer, arithmetic representation is an added dimension of flexibility/customizability. One may specialize the numerical representation in two ways:

1. Representation system—e.g. one may opt for logarithmic arithmetic representation (which has been applied to the quantum chromodynamics problem [11], for example). Other possibilities worthy of consideration include fixed-point representation, and of course IEEE format floating-point.
2. Bitwidth optimization for the selected representation, e.g. increase the number of mantissa bits. In general,

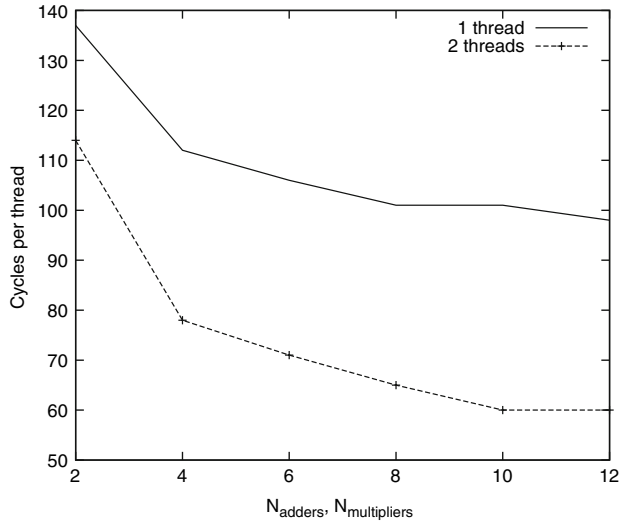

Fig. 11 Performance with 1 and 2 threads. $N_{\text{dividers}} = 3$, and $N_{\text{multipliers}} = N_{\text{adders}}$

increasing the bitwidth results in higher precision while lowering the bithwidth results in smaller datapaths and consequently we can instantiate more functional units for a given amount of logic space, resulting in higher parallelism.

Some initial analysis has been completed on this matter. Observing some of the value ranges typical of these computations, it seems that a fixed point implementation is unsuitable—it has been observed from some run-time number ranges that it would take a fixed point bitwidth as wide as 200 bits or more to match the precision of 64 bit floating point. This leaves us with the IEEE floating point format, and logarithmic arithmetic. Recall that with logarithmic arithmetic, in terms of computational expense/complexity multiplication is simple and addition is complex. Underwood et al. present a comparison of floating point and logarithmic arithmetic in [43], and their analysis specially targets FPGAs. They prescribe a set of operation ratio thresholds (e.g. percentage of multiplications vs. additions) as an indicator of whether for a particular workload logarithmic arithmetic is advantageous over IEEE floating point—as one might expect, a computation with a large number of multiplications (>70%) is a good candidate for logarithmic arithmetic. It was found that this is not the case with the target subset/partition; the ratio is closer to 50%. Therefore, there seems to be no compelling reason to strongly favour any numerical representation over standard IEEE format floating point. Furthermore, sticking with IEEE floating point has several anciliary advantages such as the availability of floating point hardware libraries (even bitwidth parameterisable libraries such as [44]), widespread availability of stable precision analysis tools, and—at least in the early stages of development—repeatability of existing results.

The developers of the ERIC processor observe in [16] that in order to achieve an accuracy of 0.01 kcal/mol (a commonly adopted standard), the required numerical precision grows with the basis set size to compensate for the accumulation of errors. For a basis size of 10k, a 52-bit floating point representation is prescribed [18]. We will build upon the work in [16] by employing a more robust methodology: *interval arithmetic*.[9] With interval arithmetic, individual scalar values are represented by a pair of values: an upper bound and a lower bound. As floating-point *roundoff* occurs, there is some uncertainty about the computed value; this uncertainty is captured by the upper and lower bounds. These ranges are propagated through the computation. Interval arithmetic is expensive; we do not propose that the processor have interval arithmetic capabilities. Rather, we propose to use interval arithmetic "offline" as a design-space exploration tool, to experiment with different arithmetic unit configurations. This general approach has proven to be successful in the past, as evidenced by the work of Luk et al. [45,46].

The work presented in [16] is very complete in the sense that the error is propagated all the way to the evaluation of the energy value. In the present work, however, we will only consider the error propagation within the partition of interest (i.e. only up till G-matrix generation). Our aim is to investigate the possibility of mixing different levels of arithmetic precision in the hopes of yielding better result precision (i.e. tighter intervals) with modest resource expense.

Our baseline is IEEE double precision (64-bit) floating point. The basis set and density matrix values are assumed to be exact (i.e. their interval width is zero), and the entire candidate partition is instrumented with interval arithmetic. The intervals of each G-matrix element may then be observed. This analysis was performed with a single Restricted-HF iteration (only up to the computation of the G-matrix) on a H_2O molecule with $6{-}31G^{**}$ basis. Our findings are summarized in Table 4. These results show that while the resulting precision will be very close to that of the weakest precision operations in the overall computation, it is possible to improve the overall precision by "promoting" certain sensitive operations to greater levels of precision.

Our next step would be to introduce completely arbitrary bitwidths so that we may experimentally observe the impact of various non-standard encodings; for instance instead of standard double precision 64-bit encoding, one may utilise a

[9] We are aware that error bounds calculated with interval arithmetic are sometimes pessimistic; in the near future we will use *affine arithmetic* instead, as it is regarded as being a slightly more reasonable (and not overly pessimistic) method of computing error bounds. The initial analysis was conducted with interval arithmetic due to our familiarity with and confidence in an existing library.

Table 4 Interval widths for different arithmetic configurations

Arithmetic configuration	Mean width	Max width
All single precision	1.267 D-4	1.259 D-3
All double precision	1.810 D-13	2.089 D-12
Mostly double precision, but with single precision Recurrence array	8.300 D-5	1.144 D-3
Mostly single precision, but with double precision Recurrence array	6.64 D-5	5.360 D-4

non-standard 70-bit encoding—the slight boost in precision may prove vital for large-scale computations.

9 Future work

We have indicated many points for immediate future investigation in the preceding sections, and most of these were of the ilk of detailed analysis and specification, or expanding our analysis to cover additional areas of interest. However, there are other aspects of our proposal that require the kind of "big picture" analysis that we have presented in this paper.

Both the ERI Dispatch (ERID) and G-matrix Contraction (GmC) units need in-depth treatment. The GmC unit may be required to support variable data arrival times due to imbalances in the individual ERI computation—this seems to suggest that *dataflow* architectural characteristics would be desirable, though we will not commit to that conclusion without further investigation. Part of our desire to minimize $\bar{t}_{\mathrm{ERI}}$ is due to the reasoning that this would reduce the variance in the computation time of individual ERI, so there is still the unlikely possibility that we will elect (after initial logic area estimates are obtained) to dedicate enough logic to the individual ERI pipelines to completely eliminate the imbalances, thus allowing a SIMD GmC unit.

As we mentioned earlier, cut-offs and symmetries may be exploited to reduce the number of ERI to be computed. Such optimizations may be made in the inner loop of the code depicted in Fig. 3. Similar functionality ought to be built into the hardware accelerator. This functionality is not overly complex; checking for cutoff conditions requires *Schwarz inequality* checking. This checking can be implemented in the *ERI Dispatch* unit in Fig. 7. Depending on the nature of the eventual solution, one could perform condition checking before the ERI computation is issued, or alternatively one may perform the checking while the ERI is being computed and then flush the pipeline if the ERI is deemed redundant and/or negligible.

10 Conclusion

From the perspective of physical significance, electron repulsion integrals and Coulombic charge repulsion may have some similarity, however from the perspective of computation they are very different beasts. ERI computation algorithms are complex, making the task of designing a special-purpose processor far more challenging. Fortunately, the emergence of System Level Design tools greatly eases the development process, making incremental architecture exploration and design possible, while also producing reasonably high quality hardware layout.

We have conducted a qualitative architectural analysis of the Hartree-Fock method (and the bottleneck ERI computation). An overview of our architectural mapping is presented in Fig. 12. We have identified that the massive thread-level parallelism inherent in the computation, combined with significant pipeline data-dependency hazards, should be exploited through fine-grained thread-interleaving. Specifically regarding the individual ERI pipelines, we have observed that pipelining is apparent at the high-level subtask layer of abstraction and because (1) one may eliminate data dependency hazards and (2) one may employ massive fine-grained multithreading, one may employ deep pipelines, constrained

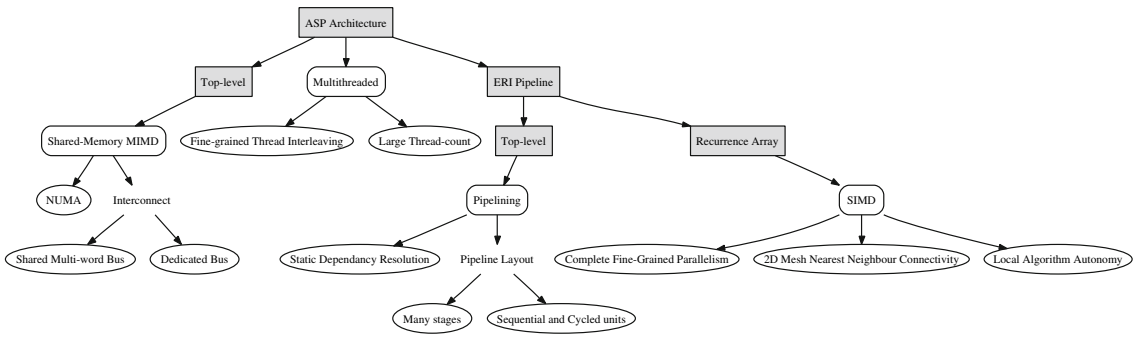

Fig. 12 Top-level Design-Space. Shaded boxes indicate stages/components, rounded boxes indicate architectural classes and ellipses are chosen design characteristics corresponding to the architecture. This is based on the *design-space approach* espoused in [4]

by logic space. We have prescribed a set of architectural specifications for the Recurrence Array SIMD processor, the most novel aspect of our proposal.

We demonstrated the approach we will employ to carry out detailed simulation of the system to parametrically optimize our design, and in doing so we have indicated the possible gains that may be obtained by leveraging fine-grained parallelism. Our analysis supports the idea of exploiting both fine-grained and coarse-grained parallelism, towards the ends of achieving a balance of high speed and practical implementation.

Finally we demonstrated that there are precision gains that one may obtain by tinkering with the precision of floating point functional units, towards the ends of pursuing larger capability computation. Alternatively, one may relax precision specification to allow the instantiation of a greater number of functional units and therefore provide a larger computing capacity.

Armed with this "roadmap" of the various components of the computation and various analysis and design methods, we may confidently pursue detailed specification, design, and implementation of the various sub-systems. Furthermore, our analysis has indicated that a "one-size-fits-all" computer architecture cannot optimally address the varied requirements and characteristics of the complete computation. Our application-specific approach allows us to produce a computer that is well-matched to the specific requirements of the Hartree–Fock method, and in doing so provide the infrastructure for higher-capacity and higher-capability computations.

References

1. Simon H, Kramer W, Saphir W, Shalf J, Bailey D, Oliker L, Banda M, William McCurdy C, Hules J, Canning, A, Day M, Colella P, Serafini D, Wehner M, Nugent P (2005) J Earth Simulat 2
2. Makino J (2006) Comput Sci Eng 8(1):30–40
3. Taiji M, Narumi T, Ohno Y, Futatsugi N, Suenaga A, Takada N, Konagaya A (2003) Protein Explorer: a Petaflops Special-Purpose Computer System for Molecular Dynamics Simulation; In: Proceedings of Supercomputing 2003
4. Sima D, Fountain T, Kacsuk P (1997) Advanced computer architectures: a design space approach; Addison-Wesley, Reading
5. Williams S, Shalf J, Oliker L, Kamil S, Husbands P, Yelick K (2006) The potential of the cell processor for scientific computing; In: CF '06: Proceedings of the 3rd conference on Computing frontiers, New York, ACM, New York, pp 9–20
6. CSX Processor Architecture Whitepaper (2006)
7. Emer J (2000) Relaxing constraints: thoughts on the evolution of computer architecture. In: Keynote Speech at the 6th International Symposium on High Performance Computer Architecture
8. Xilinx Virtex 4 Family Overview, June (2005)
9. Boyle PA, Chen D, Christ NH, Clark anbd MA, Cohen SD, Cristian C, Dong Z, Gara A, Joo B, Jung C, Kim C, Levkova LA, Liao X, Liu G, Mawhinney RD, Ohta S, Petrov K, Wettig T, Yamaguchi A (2005) IBM J Res and Dev 49(2):351–365
10. McCurdy CW, Stevens R, Simon H, Kramer W, Bailey D, Johnston W, Catlett C, Lusk R, Morgan T, Meza J, Banda M, Leighton J, Hules J (2002) Creating science-driven computer architecture: a new path to scientific leadership, LBNL/PUB-5483; Technical report, Computing Science Directorate, Ernets Orlando Lawrence Berkeley National Laboratory
11. Callanan O, Nisbet A, Ozer E, Sexton J, Gregg D (2005) FPGA implementation of a lattice quantum chromodynamics algorithm using logarithmic arithmetic. In: IPDPS '05: Proceedings of the 19th IEEE international parallel and distributed processing symposium (IPDPS'05), Workshop 3, Washington, IEEE Computer Society, p 146.2
12. Gu Y, Court TV, Herbordt MC (2005) Accelerating molecular dynamics simulations with configurable circuits. In: Proceedings of FPL 2005, pp 475–480
13. Underwood K (2004) Fpgas vs. cpus: trends in peak floating-point performance. In: FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays, pp 171–180, New York, ACM, New York
14. Wain R, Bush I, Guest M, Deegan M, Kozin I, Kitchen C (2006) An overview of FPGAs and FPGA programming: Initial experiences at Daresbury; Technical report, Computational Science and Engineering Department, CCLRC Daresbury Laboratory, Daresbury
15. HP Presents Alternate Strategy for Chip Improvement (2007)
16. Takashima H, Kitamura K, Tanabe K, Nagashima U (1999) J Comput Chem 20:443–454
17. Swan S (2001) An introduction to system level modeling in systemC 2.0
18. Hashimoto K, Tomita H, Inoue K, Metsugi K, Murakami K, Inabata S, Yamada S, Miyakawa N, Takashima H, Kitamura K, Obara S, Amisaki T, Tanabe K, Nagashima U (1999) MOE: a special-purpose parallel computer for high-speed, large-scale molecular orbital calculation; In: Supercomputing '99: proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM), New York, 1999. ACM, New York, p 58
19. Schmidt MW, Baldridge KK, Boatz JA, Elbert ST, Gordon MS, Jensen JH, Koseki S, Matsunaga N, Nguyen KA, Su S, theresa L. Windus, Dupuis M, Montgomery JA Jr (1993) J Comput Chem 14(11):1347–1363
20. Baumgartner G, Auer A, Bernholdt DE, Bibireata A, Choppella V, Cociorva D, Gao X, Harrison RJ, Hirata S, Krishnamoorthy S, Krishnan S, Lam C-C, Lu Q, Nooijen M, Pitzer RM, Ramanujam J, Sadayappan P, Sibiryakov A (2005) Proc IEEE 93(2):276–292
21. Almlöf J, Faegri K, Korsell K (1982) J Comp Chem 3:385–399
22. Cioslowski J (1993) vol 4 of Reviews in Computational Chemistry VCH Publishers, pp 1–33
23. Cooper MD, Hillier IH (1991) J Comput-Aid Mol Des 5:171–185
24. Dupuis M, Rys J, King HF (1976) J Chem Phys 65(1):111–116
25. Rys J, Dupuis M, King HF (1983) J Comput Chem 4(2):154–157
26. Auspurger JD, Bernholdt DE, Dykstra CE (1990) J Comput Chem 11(8):972–977
27. Sagar RP Jr, Smith VH (1992) Int J Quant Chem 42:827–836
28. Wilson S (1987) vol. 1 of Methods in Computational Chemistry, Chap 3, Plenum Press, New York, pp 251–350
29. Harrison RJ, Shepard R (1994) Annu Rev Phys Chem 45:623–658
30. Bolding B, Baldridge K (2000) Comput Phys Commun 128:55–66
31. Yahiro S, Gondo Y (1992) J Comput Chem 13(10):1246–1254
32. Obara S, Saika A (1986) J Chem Phys 84(7):3963–3974
33. Agarwal PA, Alexander RA, Apra E, Balay S, Bland AS, Colgan J, D'Azevedo EF, Dongarra JJ, Dunigan TH Jr, Fahey MR, Fahey RA, Geist A, Gordon M, Harrison RJ, Kaushik D, Krishnakumar M, Luszczek P, Mezzacappa A, Nichols JA, Nieplocha J, Oliker L, Packwood T, Pindzola MS, Schulthess TC, Vetter JS, White JB III, Windus TL, Worley PH, Zacharia T (2004) ORNL/TM-2004/13: Cray X1 evaluation status report; Technical report, Oak Ridge National Laboratory

34. Taylor PR Jr, Bauschlicher CW, Schwenke DW (1989) vol 3 of Methods in computational chemistry, Chap 2. Plenum, New York, pp 63–146
35. Shirsat RN, Limaye AC, Gadre SR (1993) J Comput Chem 14(4):445–451
36. Gan Z, Harrison RJ (2005) Calibrating quantum chemistry: a multi-teraflop, parallel-vector, full-configuration interaction program for the Cray-X1. In: Supercomputing, p 22
37. Nakamura K, Hatae H, Harada M, Kuwayama Y, Uehara M, Sato H, Obara S, Honda H, Nagashima U, Inadomi Y, Murakami K (2002) Eric: a special-purpose processor for ERI calculations in quantum chemistry applications; In HPC-Asia
38. Nakamura K, Honda H, Inoue K, Sato H, Uehara M, Komatsu H, Umeda H, Inadomi Y, Araki K, Sasaki T, Obara S, Nagashima U, Murakami K (2005) A high-performance, low-power chip multiprocessor for large scale molecular orbital calculation. In: Proceedings of the workshop on unique chips and systems (UCAS), pp 87–94
39. Madsen J, Grode J, Knudsen P, Petersen M, Haxthausen A (1997) Lycos: the lyngby cosynthesis system
40. Vanmeerbeeck G, Schaumont P, Vernalde S, Engels M, Bolsens I (2001) Hardware/software partitioning of embedded system in ocapi-xl. In: CODES '01, New York, ACM, New York, pp 30–35
41. Sherwood T, Varghese G, Calder B (2003) A pipelined memory architecture for high throughput network processors
42. Ahn JH, Erez M, Dally WJ (2006) The design space of data-parallel memory systems. In: SC2006: proceedings of supercomputing 2006
43. Haselman M, Beauchamp M, Underwood K, Hemmert KS (2005) A comparison of floating point and logarithmic number systems for FPGAs. In: FCCM'05: proceedings of the 13th annual IEEE symposium on field-programmable custom computing machines. pp 181–190
44. Belanović P, Leeser M (2002) A library of parameterized floating point modules and their use. In: 12th international conference on field programmable logic and application, FPL 2002. Montpellier, France, pp 657–666
45. Lee D-U, Gaffar AA, Mencer O, Luk W (2005) MiniBit: bit-width optimization via affine arithmetic. In: DAC '05: proceedings of the 42nd annual conference on design automation, New York, ACM, New York, pp 837–840
46. Gaffar AA, Mencer O, Luk W, Cheung PYK (2004) Unifying bit-width optimisation for fixed-point and floating-point designs. In: FCCM2004: proceedings of the 12th annual IEEE symposium on field-programmable custom computing machines. pp 79–88