Resolutions of the Coulomb Operator:*

VIII. Parallel implementation using the modern programming language X10

Taweetham Limpanuparb †‡§
Josh Milthorpe §¶
Alistair P. Rendell§

23 August 2014

Abstract

Use of the modern parallel programming language X10 for computing long-range Coulomb and exchange interactions is presented. By using X10, a partitioned global address space language with support for task parallelism and the explicit representation of data locality, the resolution of the Ewald operator can be parallelized in a straightforward manner including use of both intra- and inter-node parallelism. We evaluate four different schemes for dynamic load balancing of integral calculation using X10’s work stealing runtime, and report performance results for long-range HF energy calculation of large molecule/high quality basis running on up to 1024 cores of a high performance cluster machine.

*Previous papers in the series can be found in Refs. [1–7].
†Corresponding author: taweetham.lim@mahidol.ac.th
‡Mahidol University International College, Mahidol University, Nakhonpathom 73170, Thailand
§Research School of Computer Science, Australian National University, ACT 0200, Australia
¶IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598 USA
A key bottleneck in the Hartree–Fock method is exchange matrix calculation. Using the Ewald resolution, this step can be split as shown in the picture to multiple computing nodes. At each node, the standard linear algebra subroutines DGEMM and DSYRK perform the matrix-matrix multiplications necessary to compute exchange matrix elements.
1 Introduction

The partitioning of the Coulomb operator \(1/r_{12}\) into a short-range part and a long-range part\(^{8,9}\) is central to the accuracy and computational performance of many quantum chemistry methods, including long-range corrected density functional theory (LRC-DFT) methods\(^{10–18}\), and a number of new MP2 methods\(^{19,20}\). While the short-range (local) component can be evaluated at a relatively small computational cost by various techniques\(^{13,21–25}\), evaluation of the long-range (non-local) part is still a relatively expensive task.

Resolutions of the Coulomb operator (RO) are mathematical identities that may provide a potent new route to reduce the computational complexity of quantum chemical methods\(^{26}\). It has previously been demonstrated to reduce sequential computation time for long-range Hartree–Fock energy compared to conventional calculation of two-electron integrals\(^{7}\). Even so, for practical problems it is advantageous to have a method that is efficiently parallelized over a large computer cluster. This requires making full use of the available memory, balancing load between processing elements and minimizing communication costs.

The X10 programming language\(^{27}\) is a modern asynchronous partitioned global address space (APGAS) language, which supports task parallelism and the explicit representation of data locality. It presents a particularly attractive programming model for irregular problems, due to its use of work stealing for load balancing parallel tasks. Communication constructs in X10 map to modern high-performance interconnects and have been demonstrated to provide high performance scaling out to tens of thousands of cores\(^{28}\).

In the next section, we outline relevant theory, resolutions of the Coulomb operator, application of the resolution to Hartree–Fock and X10 programming language. Our parallel implementation details and results are discussed in \(\S3\) and \(\S4\) respectively. Finally, we present concluding remarks and future development in \(\S5\). Unless otherwise stated, atomic units, Einstein summation convention, real Cartesian Gaussian basis functions and an Ewald partition parameter \(\omega = 0.3\) are used throughout.

2 Theory

2.1 Resolutions of the Coulomb Operator

In our previous paper\(^{7}\), we demonstrated that the evaluation of Hartree–Fock Coulomb and exchange interactions for the long-range Ewald operator

\[
L(r_{12}) = \frac{\text{erf}(\omega r_{12})}{r_{12}} \tag{1}
\]

can be sped up by resolving \((1)\) into a sum of products of one-electron functions

\[
L(r_{12}) = \sum_{n=0}^{\infty} \sum_{l=0}^{\infty} \sum_{m=-l}^{l} \phi_{nlm}(r_1)\phi_{nlm}(r_2) \equiv \sum_{k=1}^{\infty} \phi_k(r_1)\phi_k(r_2) \tag{2}
\]

and truncating the resolution \((2)\) to a \(K\)-term finite sum,

\[
L(r_{12}) \approx \sum_{n=0}^{N} \sum_{l=0}^{L} \sum_{m=-l}^{l} \phi_{nlm}(r_1)\phi_{nlm}(r_2) \equiv \sum_{k=1}^{K} \phi_k(r_1)\phi_k(r_2) \tag{3}
\]
where \( K = (N' + 1)(L + 1)^2 \) and \( k = n(L + 1)^2 + l(l + 1) + m + 1 \). The resolution function, \( \phi_{nlm} \) in \(^2\) can also take a myriad of forms. In this paper, we continue to use functions of the form

\[
\phi_{nlm}(\mathbf{r}) = q_n j_l(\lambda_n r) Y_{lm}(\mathbf{r}),
\]

where \( j_l \) is a spherical Bessel function and \( Y_{lm} \) is a real spherical harmonic. Thus in the Hartree–Fock method using the truncated resolution \(^3\), the traditional calculation of two-electron integrals \( \langle \mu \nu | \lambda \sigma \rangle \) is replaced by calculation of the auxiliary integrals

\[
(\mu \nu | nlm) = \int \chi_\mu(\mathbf{r}) \chi_\nu(\mathbf{r}) \phi_{nlm}(\mathbf{r}) \, d\mathbf{r}.
\]

The resolution technique has some connections to two long established methods, Resolution of the Identity (RI) or Density Fitting (DF)\(^29\)\(^32\) and Cholesky Decomposition (CD)\(^33\)\(^38\). The common approach to performance improvement is factorization of the two-electron integrals. Comprehensive discussion of RO in relation to the RI/DF and CD can be found in Limpanuparb’s thesis\(^26\). Parallel implementations of RI/DF are being developed by various groups. For example, Neese et al. have reported in 2009 that their implementation of RI in ORCA for Hartree–Fock exchange has a speedup of 8.6 observed for 10 processes\(^39\).

### 2.2 Application of the Resolution to Hartree–Fock Method

The closed-shell Hartree–Fock Coulomb and exchange matrices are given by:

\[
J_{\mu \nu} = 2 \sum_{a}^{\text{occ}} \sum_{\lambda \sigma}^{N^2} c_{\lambda a} c_{\sigma a} (\mu \nu | \lambda \sigma)
\]

\[
K_{\mu \nu} = 2 \sum_{a}^{\text{occ}} \sum_{\lambda \sigma}^{N^2} c_{\lambda a} c_{\sigma a} (\mu \lambda | \sigma \nu),
\]

where \( N \) is the number of basis functions, \( c_{j a} \) are the molecular orbital coefficients and \( \text{occ} \) represents the set of \( O \) occupied orbitals. Substituting the resolution into above expressions yields the RO formulae for \( J \) and \( K \):

\[
J_{\mu \nu} \approx \sum_{nlm}^{K} (\mu \nu | nlm) D_{nlm}^{nlm}
\]

\[
D_{nlm}^{nlm} = \sum_{\lambda \sigma}^{N^2} D_{\lambda \sigma} (nlm | \lambda \sigma)
\]

\[
K_{\mu \nu} \approx 2 \sum_{a}^{\text{occ}} \sum_{nlm}^{K} (\mu a | nlm) (nlm | a\nu)
\]

\[
(\mu a | nlm) = \sum_{\lambda}^{N} c_{\lambda a} (\mu \lambda | nlm)
\]

where \( D_{\lambda \sigma} = 2 \sum_{a}^{\text{occ}} c_{\lambda a} c_{\sigma a} \) is the density matrix.
For $n = 0 ... N'$

\[ A_{\mu,\nu lm} \leftarrow (\mu \nu | nlm) \quad \text{Eq. (5)} \]

// Coulomb matrix

\[ D^{lm} \leftarrow P_{\mu \nu} \times A_{\mu,\nu lm} \quad \text{Eq. (9)} \]

\[ J_{\mu \nu} \leftarrow D^{lm} \times A_{\mu,\nu lm} \quad \text{Eq. (8)} \]

// Exchange matrix

\[ B_{\nu lm, a} \leftarrow A_{\mu,\nu lm} \times C_{\mu, a} \quad \text{Eq. (11)} \]

\[ K_{\mu \nu} \leftarrow B_{\mu, lma} \times B_{\nu, lma} \quad \text{Eq. (10)} \]

Next $n$

**Figure 1:** RO Pseudocode for construction of Coulomb and exchange matrices

The resolution of the Coulomb and exchange operators for a set of $N$ basis functions requires the computation of auxiliary integrals $(\mu \nu | nlm)$ for basis functions $\mu, \nu$ where $n \in \{0, 1, 2, \ldots, N'\}$ describes the radial component and $l \in \{0, 1, 2 \ldots \mathcal{L}\}$, $m \in \{-l, -l + 1, -l + 2, \ldots l\}$ describe the angular components of the resolution. Constructing the Fock matrix requires a total of $N^2(N' + 1)(\mathcal{L} + 1)^2$ auxiliary integrals; however, as the contributions for the different radial components of the resolution are independent, the integrals may be computed and their contributions to the Coulomb matrix $J$ and the exchange matrix $K$ accumulated in a loop over $n$, thereby reducing the storage required for integrals by a factor of $N'+1$. Dividing by $n$ (rather than by angular component $l, m$) allows the use of recurrence relations to efficiently compute the resolution for higher angular momenta.

In a previous paper, we presented a sequential implementation of RO as shown in the pseudocode in Figure 1. For parallel implementation, it may be noted that while the elements of both the $J$ matrix (lines 4–5) and the $K$ matrix (lines 7–8) depend on the auxiliary integrals (line 2), $J$ and $K$ may be computed independently of each other. Each $J_{\mu \nu}$ depends on all values of $D^{lm}$ (a reduction dependency), which in turn depends on the auxiliary integrals $A$ and the density matrix $P$. Each $K_{\mu \nu}$ depends on two blocks of $B_{\nu lm, a}$, which in turn depends on the auxiliary integrals $A$ and the molecular orbital coefficients matrix $C$. The most expensive operations are those to produce the exchange matrix at lines 7 and 8, costing $O(N^2 K)$ operations, but these can be formulated as matrix multiplications and efficiently computed using optimised (DGEMM) library calls. The memory requirements for matrices $A$ and $B$ are $O(N^2 (\mathcal{L} + 1)^2)$ and $O(N (\mathcal{L} + 1)^2)$ respectively. It was shown that the computational cost of HF $J$ and $K$ matrix calculation using high quality basis sets can be significantly reduced by using the resolution. RO algorithm scales only quadratically with respect to basis set size (for a fixed molecule) and works best for compact globular molecules, for which traditional cut-off strategies are relatively ineffective.

We note that the discussion in this subsection and the previous subsection are also true for full Coulomb and short-range Coulomb operators but we choose to apply them to the long-range Ewald operator.
2.3 X10 Programming Language

The X10 programming language\(^{22}\) is an asynchronous partitioned global address space (APGAS) language. The sequential core of the language is similar to and largely interoperable with Java, which allows X10 applications to make use of the existing rich ecosystem of Java tools and libraries. The main differences with Java are the explicit representation of data locality in the form of \textit{places}, and the expression of task parallelism through asynchronous \textit{activities}. A place in X10 corresponds to one or more co-located processing elements with attached local storage, for example a single node of a compute cluster. X10 supports the \textit{active message} idiom, in which data transfer and remote computation are combined in a single message; this allows for the terse expression of many distributed algorithms using localized patterns of communication and synchronization.

Task parallelism is supported in X10 through the \texttt{async} statement, which starts a new asynchronous activity at the current place. Activities may change place using the \texttt{at(p)} construct, which evaluates a statement or expression at a remote place and returns the result to the originating place. The \texttt{finish} statement causes the current activity to await the termination of all activities contained within the finish scope. Arbitrary patterns of distributed activity creation and termination are supported, with optimizations for commonly occurring patterns\(^{25}\).

X10 provides two compilation paths using source-to-source compilation: \textit{Managed X10}, which generates Java code to run on a standard Java Virtual Machine with an X10 runtime library; and \textit{Native X10}, which translates to C++ and is compiled to a binary application using a standard C++ compiler. Both compilation paths support integration with native code libraries. Communication between places is handled by \texttt{X10RT}, a C++ library implementation of the X10 runtime layer. There are several implementations of \texttt{X10RT} which use different communication methods including sockets, MPI, PAMI\(^{1}\) and a stand-alone implementation to support multiple places on a single host. These implementations make it possible to run multi-place X10 programs on a wide range of computer systems.

The X10 Global Matrix Library (GML)\(^{11}\) was developed to support distributed linear algebra in X10. It provides a variety of single-place and distributed matrix formats for dense and sparse matrices, and implements linear algebra operations for these matrix formats. Operations for single-place dense matrices are implemented as wrappers around the basic linear algebra subroutines (BLAS)\(^{42}\) and Linear Algebra Package (LAPACK)\(^{43}\) routines; more complex operations are implemented in X10 using the simple operations as building blocks.

3 Parallel Implementation

In Figure\(^{1}\) the auxiliary integrals (5) are calculated by calling a C++ routine and stored in a dense matrix \(A_{\mu,\nu,lm}\) (line 2). In comparison to our previous paper\(^{22}\), we insert an additional step 2a which stores a copy of the integrals in sparse format as a ragged array \(A_{\mu,\nu,lm}^{s}\). This dual storage approach allows the contribution of matrix \(B\) (line 7) to be computed using an

\(^{1}\)a proprietary communications API developed by IBM for use on high-performance interconnects including Blue Gene and Infiniband
efficient DGEMM operation, while the contraction of the auxiliary integrals with the density matrix and $D^{lm}$ (line 4-5) reads integral values with unit stride in memory.

The computations of the auxiliary integrals and their contributions to the $J$ and $K$ matrices permit multiple levels of parallelism. The most natural levels at which to divide the computation are by shell, shell pair and radial component $n$.

When decomposing by shell (a block of $\mu$), the $J$ and $K$ matrix contributions for each block of $\mu$ depend on the full sets $D^{lm}$ and $B_{\nu lma}$, which must therefore also be reduced across all places involved in computing integrals for a given $n$. The auxiliary integrals $A_{\mu,\nu lm}$ and intermediate matrix $B_{\mu lma}$ may be divided between places, to allow larger problems to be treated by using the available memory on multiple cluster nodes. Parallelism within a place is also readily exposed by dividing by shell; as each shell contributes to a unique block of both $J$ and $K$ matrices, threads may proceed independently without the need for mutual exclusion.

Integral calculation may be further divided by shell pair (a block of $\mu\nu$) to increase the available parallelism; however, this means that each activity at line 7 in Figure 1 contributes to very narrow blocks of data. The coarser division by shell allows the computation of a larger block of $B$ to be performed by a call to an efficient BLAS DGEMM operation, which is the approach used in this implementation.

When decomposing by the radial component, each value of $n$ requires all spherical harmonics $Y_{lm}^{\mu\nu}$ in (4), so distributing different values of $n$ to different processes requires that the full set of $Y_{lm}^{\mu\nu}$ must be either replicated or recomputed at each place. As the full matrices $A_{\mu\nu lm}$, $B_{\nu lma}$ are required for each value of $n$, there is no reduction in the memory requirement per node, and therefore dividing by radial component does not by itself permit the treatment of a larger problem than would fit in the memory of a single cluster node. Distribution of $n$, however, may offer time saving when the SCF requires many cycles and auxiliary integrals are not recalculated after the first cycle. As our experiment is concerned with one SCF cycle, the current implementation uses only shell decomposition for place parallelism.

For large problem sizes it is not possible to store all auxiliary integrals in cache. For example, for a cluster of 5 water molecules using the cc-pVQZ basis set where $N = 700$, $L = 13$, there are $700^2 \times (13 + 1)^2 \approx 96$ M auxiliary integrals requiring 733 MiB, which is around two orders of magnitude larger than a typical last-level cache. To achieve performance it is necessary to make efficient use of cache by optimizing for locality; this will be considered in §3.1.3.

The time required to compute auxiliary integrals in (3) depends on the angular momenta of the basis functions and the degree of contraction, and may differ widely between shells within a given basis set. Figure 2 shows the time to compute auxiliary integrals for the different shells of a cluster of 10 water molecules using the cc-pVQZ basis set. The time required to compute a single shell (a block of $\mu$ paired with all $\nu$) varies from 11 ms to more than 70 ms. This presents a load balancing problem as it is necessary to divide the set of integrals into evenly sized portions for computation by different processing elements. A static load balancing approach - carefully mapping integrals to processing elements so that each processing element performs roughly the same amount of computational work - has been successfully used for integral evaluation on GPUs[13]. Even so, static schemes cannot account for unpredictable changes in the compute capacity of different cores that may arise when, for
example, the frequency of a core is automatically reduced in response to local environmental factors. Alternatively, load may be balanced dynamically by ordering integral calculations in a centralized task queue\(^4\), such an approach has previously been considered for use with X10\(^{46,47}\).

In §3.1 we explore an alternative solution which makes use of X10’s work stealing runtime for dynamic load balancing within a place, without the need for a centralized task queue. In §3.2 we describe the distribution of key data structures needed for multi-place computation. Finally, in §3.3 we consider the load imbalance between places due to the choice of data distribution.

### 3.1 Auxiliary Integral Calculation with a Work Stealing Runtime

Although work stealing can be an effective way of load balancing parallel activities between worker threads, it also raises three potential problems: i) how to avoid synchronization so as to allow activities to compute independently; ii) the overhead of creating and stealing activities; and iii) the impact of stealing patterns on data locality. The following subsections consider each problem in the context of auxiliary integral calculation.

#### 3.1.1 Use of Worker-Local Data to Avoid Synchronization

The WorkerLocalHandle class in X10 allows parallel activities to compute contributions to the $J$ and $K$ matrices independently of other activities. Figure 3 shows the use of a WorkerLocalHandle to allocate memory for a separate set of auxiliary integrals for each worker thread. The X10 Rail class used for the integrals is a zero-based, one-dimensional

---

**Figure 2:** Variation in time to compute auxiliary integrals for different shells on *Raijin* ($[\text{H}_2\text{O}]_{10}$, cc-pVQZ, $N' = 9, L = 16$)
(C-style) array. The parallel loop on line 5 starts a separate activity (async) for each shell,

```scala
def main(args: Array[String]): Unit = {
  val aux_wlh = new WorkerLocalHandle[Rail[Double]](
    () => new Rail[Double](maxam1*N*roK));
  val dlm_wlh = new WorkerLocalHandle[Rail[Double]](
    () => new Rail[Double](roK));

  finish for (shell in shells) async {
    val aux = aux_wlh();
    val dlm = dlm_wlh();
    for (shellPair in shell.pairs) {
      Genclass(shellPair, ..., aux);
      for ((mu, nu) in shell.basisFunctions) {
        for (l in 0..roL) {
          dlm(l, m) += density(mu, nu) * aux(mu, nu, l, m);
        }
      }
    }
    DGEMM(mos, aux, b);  // Single thread version
  }
  dlm_wlh.reduceLocal(dlm_complete, ...);
}
```

**Figure 3:** X10 code to compute auxiliary integrals, $D^{lm}$ and $B$ using `WorkerLocalHandle` and waits for all activities to terminate (finish); the activities are scheduled for execution on the available worker threads. The worker threads also use a `WorkerLocalHandle` to store partial contributions to $D^{lm}$, which are summed together after the computation of all auxiliary integrals for a given $n$. Thus each iteration of the parallel loop in Figure 3 is independent and synchronization is only required in the reduction of $D^{lm}$ on the final line.

### 3.1.2 Overhead of Activity Management

For efficient scheduling, work stealing relies on over-decomposition — a division of work into significantly more activities than there are units of hardware parallelism. It is possible that the cost of creating many activities and scheduling them for execution by worker threads could outweigh the benefit gained by balancing load. To evaluate the costs and benefits for this application requires examination of the pattern of activity creation and stealing exhibited at runtime.

For the basic parallel loop shown in Figure 3 (line 5), the worker thread that executes the enclosing `finish` statement (the master thread) creates a separate activity for each iteration of the loop. When an activity is created, if any threads are currently idle then the activity is immediately run on one of the idle threads. Otherwise the activity is placed on the `deque` (double-ended queue) of the worker thread that created it. The majority of activities are therefore placed on the `deque` of a single thread, to be stolen by other threads. In other
words, the steal ratio — the proportion of activities that are stolen rather than executed by the worker which created them — is high.

To evaluate the benefit of work stealing in this application, we compare it with the static load balancing approach, in which a single activity is created per thread. Figure 4 shows code for such a static load balancing approach using a cyclic decomposition of shells.

```c
1 finish for (th in 0..(Runtime.NTHREADS-1)) async {
2    for (shellIdx = th; shellIdx < shells.size; shellIdx += maxTh) {
3        ... // Line 6--18 in Figure 3
4    }
5 }
```

**Figure 4:** Parallel loop to compute auxiliary integrals: single activity per thread with cyclic decomposition

Using a single long-running activity per worker thread as in Figure 4 eliminates the overhead of activity management, however, there may be a load imbalance between worker threads due to the differing times to compute shell pairs as shown in Figure 2.

### 3.1.3 Work Stealing and Locality

The most natural order in which to store the auxiliary integrals (using either sparse or dense matrix representation) is by shell pair (a block of $\mu\nu$). Thus there is a relationship between `shellIdx` and the memory locations to which the corresponding auxiliary integrals must be written. Using the basic parallel loop shown in Figure 3, the order of stealing is effectively random, which means that there is no relationship between the loop index `shellIdx` for an activity and the worker thread which executes it. In comparison, the cyclic decomposition in Figure 4 means that each thread’s accesses are roughly evenly distributed throughout memory, in small contiguous blocks separated by irregular stride.

A single activity per worker thread may also use a block decomposition, as shown in Figure 5. This divides the shell pairs into contiguous blocks of approximately equal size. The block decomposition has the additional advantage of locality: each worker thread writes to a compact portion of the auxiliary integral matrix $A_{\mu,\nu lm}$, whereas with the cyclic decomposition each thread writes to widely separated portions of the matrix.

The basic parallel loop entails a high steal rate and therefore stealing overhead, whereas the static load balancing approaches may lead to load imbalance. Ideally, we would prefer a loop decomposition that provides load balancing with a low steal rate, while also maintaining good locality between the tasks processed by each worker thread. A divide-and-conquer loop transformation promises such a combination. In this approach, the loop is recursively bisected into approximately equal ranges, with each range constituting an activity. Bisection continues until a certain minimum grain size is reached. Figure 6 shows example code for this transformation applied to the loop over shell pairs.

With the recursive bisection loop transformation, if a worker thread’s deque contains any activities, then the activity at the bottom of the deque will represent at least half of the loop range held by that worker. Thus idle workers tend to steal large contiguous chunks of the
val chunk = shells.size / Runtime.NTHREADS;
val remainder = shells.size % Runtime.NTHREADS;
finish for (th in 0..(Runtime.NTHREADS-1)) async {
  val start = th < remainder ? ((chunk+1) * th)
    : (remainder + chunk*th);
  val end = (th < remainder ? ((chunk+1) * (th+1))
    : (remainder + chunk*(th+1))) - 1;
  for (shellIdx in start .. end) {
    ...
  }
}

Figure 5: Parallel loop to compute auxiliary integrals: single activity per thread with block decomposition

struct RecursiveBisection1D(start:Long, end:Long, grainSize:Long){
  public def this(start:Long, end:Long, grainSize:Long) {
    property(start, end, grainSize);
  }

  public def execute(body:(idx:Long)=> void) {
    if ((end-start) > grainSize) {
      val secondHalf =
        RecursiveBisection1D((start+end)/2L, end, grainSize);
      async secondHalf.execute(body);
      val firstHalf =
        RecursiveBisection1D(start, (start+end)/2L, grainSize);
      firstHalf.execute(body);
    } else {
      for (i in start..(end-1)) {
        body(i);
      }
    }
  }
}

finish RecursiveBisection1D(0, shells.size, grainSize).execute(
  (shellIdx:Long)=> {
    ...
  });

Figure 6: Parallel loop to compute auxiliary integrals: recursive bisection transformation
loop range, preserving locality. A further advantage is that activities are executed in order of shell index, rather than randomly in the basic parallel loop (Figure 3). The bisection approach is not without cost: for a loop of \( N \) iterations, an additional \( \log_2(N/\text{grainSize}) \) activities are created to divide the work.

3.2 Distributed and Replicated Data Structures

When implementing the SCF for execution on a distributed architecture, an important question is how to distribute the key matrices. Certain matrices such as the density matrix and molecular orbital coefficients must be accessible to all places, as their elements are used in the calculation of widely separated elements of the Fock matrix. As these matrices are of size \( N^2 \) and \( ON \), it is generally feasible to replicate them to every place.

```
1 public class ROFockMethod {
2   ...
3   public def compute(density: Density, mos: MolecularOrbitals) {
4       finish ateach(place in Place.places()) {
5           for (ron in 0..roN) {
6               computeAuxBDlm(density, mos, ron,
7                  /* output */ auxJ, bMat, dlm);
8               computeJ(ron, auxJ, dlm,
9                  /* output */ localJ);
10               computeK(ron, bMat,
11                  /* output */ localK);
12               Team.WORLD.barrier();
13           }
14           // gather J, K to place 0
15       }
16   }
17   ...
18 }
```

**Figure 7:** High level structure of X10 code to compute Fock matrix, showing broadcast of density and molecular orbital matrices using `finish/ateach`
3.2.1 Distributing Dense Matrices for Linear Algebra

The computation of $K$ matrix contributions from auxiliary integrals can be cast in the form of dense linear algebra operations, which enables the use of a highly-optimized BLAS implementation exposed through GML. The contraction of auxiliary integrals with molecular orbitals (step 7 in Figure 1) for a given $n$ is performed using a call to DGEMM, which uses a single thread to allow multiple such calls in parallel for different shells, each of which creates a unique block of the matrix $B_{\nu lma}$. Once $B$ has been computed in full, the contribution to matrix $K$ (step 8 in Figure 1) may be computed using a symmetric rank-K update. For a single place, all of $B$ is held locally, which allows the use of the BLAS DSYRK operation. For multiple places, $B$ is divided into blocks of rows, meaning that a distributed rank-K update is required.

Matrices $B$ and $K$ (as well as $J$) are represented using the DistDenseMatrix class from the GML. In a DistDenseMatrix, the matrix is block-distributed between places; each place maintains its own local block as a DenseMatrix, and also holds a Grid which defines the extent of the local block held at each place in the distribution. For our purposes, matrix $B$ is divided into blocks of rows according to the values of $\mu$ for which that place has computed auxiliary integrals. At each place, an upper-triangular block of $K$ is computed from the local block of rows using DSYRK. The remaining off-diagonal blocks of $K$ are computed using DGEMM on a combination of local and non-local data.

Figure 8 shows the pattern of distributed computation of blocks of the $K$ matrix for 1–4 places. For odd numbers of places, all DGEMM operations compute roughly square blocks of $K$. For even numbers of places, to ensure load balance the last block in each row is divided between pairs of places, therefore the final DGEMM for each place computes a rectangular block of $K$ (dimension ratio of about 2:1).

After each place has computed its partial contributions to $J$ and $K$, these matrices are gathered to Place 0 for further processing using the Rail.asyncCopy method which asynchronously transfers array data to a remote place. For large problems, computing these matrices (density and molecular orbital) entirely at Place 0 may present an undesirable sequential bottleneck; however for the problems considered in this paper the cost of this step is small relative to other parts of the computation.

3.3 Load Balancing Between Places

The computational cost attributable to a given place is estimated as being proportional to the total number of functions from shells assigned to the place; shells are divided so that the total number of functions is approximately even between places. A whole shell is always assigned to one place because the integral calculation code does the job in a batch of shell pairs.

Given the data distribution scheme described above, the simplest measurement of computational cost at each place is the Fock matrix cost, $\text{cost}_{\text{Fock}}$, which represents the cost of computing $J$ and $K$ matrix elements from auxiliary integrals (lines 4–8 in Figure 1). This cost is proportional to the number of basis functions of the shells assigned to place $p$. The
Figure 8: RO contributions to $K$ matrix: block pattern of distributed computation for different numbers of places
ideal value of \( \text{cost}_{\text{Fock}} \) is approximately \( N/p \) but the actual value is calculated by:

\[
\text{cost}_{\text{Fock}}(p) = \sum_{\text{shell } i \text{ in } p} \# \text{ basis functions in } i
\]  

(12)

A second measurement is the cost of computing auxiliary integrals (line 2 in Figure [1]),

\[
\text{cost}_{\text{aux}}(p) = \sum_{\text{significant shellpair } ij \text{ in } p} \# \text{ integrals in } ij.
\]  

(13)

The second measurement arises because a shell pair cut-off eliminates the need to compute some integrals, thereby providing computational savings that are unevenly distributed across places.

From these costs, two different measurements of load imbalance can be calculated: the Fock matrix load imbalance

\[
L_{\text{Fock}} = \frac{\max_p \text{cost}_{\text{Fock}}(p)}{\sum_{p=0}^{P-1} (\text{cost}_{\text{Fock}}(p))/P},
\]  

(14)

and the auxiliary integral load imbalance

\[
L_{\text{aux}} = \frac{\max_p \text{cost}_{\text{aux}}(p)}{\sum_{p=0}^{P-1} (\text{cost}_{\text{aux}}(p))/P}.
\]  

(15)

Assuming a perfectly parallel code with no sequential sections or parallel overhead, the load imbalance is the inverse of parallel efficiency. For example, an imbalance of 4 would mean a parallel efficiency of 25% and an imbalance of 1 would mean perfect parallel efficiency.

4 Numerical Results

To evaluate the X10 implementation of long-range energy calculation using RO, its performance was first compared against that of a conventional SCF calculation using Q-Chem version 4.0.0.1\(^{48}\). Shared-memory scaling was then considered, with particular regard to the effects on locality and performance of the different methods for dividing loops for integral and \( J \) matrix calculation previously described in §3.1. Finally, distributed-memory scaling was assessed with regard to load balance and communication required between places.

All timing experiments were conducted on the \textit{Raijin} Fujitsu Primergy cluster installed at the NCI National Facility at the Australian National University. Each node of \textit{Raijin} contains two eight-core 2.6 GHz Intel Xeon Sandy Bridge CPUs, 32 GB DDR3-1600 memory and onboard FDR InfiniBand. The Intel Math Kernel Library version 12.1.9.293 implementation of the BLAS library was used for the X10 code.

Our full source code\(^{49}\) is freely available in \textit{Pumja Rasaayani} (Sanskrit for “quantum chemistry”), a program in ANUChem package\(^{47}\). All accuracy and timing results given below use X10 version 2.4 and the corresponding tagged version of ANUChem.

Our test cases comprise of water clusters\(^{50}\) and polypeptides\(^{51}\). Geometries are described in the references given here. The test cases represent typical requirements for simulations of
biological systems: clusters of five to twenty water molecules, and one-dimensional (chain-like) and three-dimensional (globular) alanine polypeptides. We ran a single SCF cycle for calculation of long-range energy using Ewald partition parameter $\omega = 0.3$ and accuracy threshold $\text{THRESH} = 6$ for various molecules with basis sets of different quality. We use molecular orbitals from diagonalization of the core Hamiltonian and Cartesian orbitals for all calculations in this section. Our timings include only the calculation of long-range Fock matrix components from one cycle and do not include diagonalization or any other associated steps in HF.

4.1 Single-Threaded Performance

Table 1 shows the time to compute Fock matrix, including auxiliary integral, $J$ and $K$ matrix calculations, for various molecules on a single core of *Raijin*. In addition to computation time, the table reports characteristics of each calculation including number of occupied orbitals $O$, number of basis functions $N$, molecular radius $R$, resolution radial truncation $\mathcal{N}$, and angular truncation $\mathcal{L}$. For each molecule, basis sets are ordered down the table from smallest to largest number of basis functions. To compare the performance of *Pumja Rasaayani* against conventional SCF calculation, the time for Fock matrix calculation using Q-Chem is also reported.2

As discussed earlier in §2.2, the RO computational cost is described by $\mathcal{O}(ON^2K)$. Figure 9 shows that the relationship between $\mathcal{O}(ON^2K)$ and SCF time is approximately linear. The deviation from linearity is due to integral screening which plays a significant role in

2Although we use only a single SCF cycle for timing purposes, the code is capable of performing a full SCF calculation.

3The reported time for Q-Chem is the “AOints” time for the full Coulomb operator. This is an approximation to the conventional long-range $J$ and $K$ matrix calculation time, as the number of integrals required to be calculated is approximately the same.
Table 1: Time to compute Fock matrix components for different molecules and basis sets using RO on Sandy Bridge (one thread).

<table>
<thead>
<tr>
<th>Molecule</th>
<th>Basis set</th>
<th>O</th>
<th>N</th>
<th>R</th>
<th>N'</th>
<th>L</th>
<th>SCF time/s*</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RO on Pumja Rasaayani</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Aux</td>
</tr>
<tr>
<td>(H₂O)₅</td>
<td>cc-pVDZ</td>
<td>25</td>
<td>125</td>
<td>16.19</td>
<td>9</td>
<td>13</td>
<td>0.71</td>
</tr>
<tr>
<td></td>
<td>cc-pVTZ</td>
<td></td>
<td>325</td>
<td></td>
<td></td>
<td></td>
<td>3.67</td>
</tr>
<tr>
<td></td>
<td>cc-pVQZ</td>
<td></td>
<td>700</td>
<td></td>
<td></td>
<td></td>
<td>16.8</td>
</tr>
<tr>
<td>(H₂O)₁₀</td>
<td>cc-pVDZ</td>
<td>50</td>
<td>250</td>
<td>18.97</td>
<td>9</td>
<td>16</td>
<td>3.62</td>
</tr>
<tr>
<td></td>
<td>cc-pVTZ</td>
<td></td>
<td>650</td>
<td></td>
<td></td>
<td></td>
<td>20.5</td>
</tr>
<tr>
<td></td>
<td>cc-pVQZ</td>
<td></td>
<td>1400</td>
<td></td>
<td></td>
<td></td>
<td>90.8</td>
</tr>
<tr>
<td>(H₂O)₂₀</td>
<td>cc-pVDZ</td>
<td>50</td>
<td>500</td>
<td>31.07</td>
<td>14</td>
<td>25</td>
<td>51.1</td>
</tr>
<tr>
<td></td>
<td>cc-pVTZ</td>
<td></td>
<td>1300</td>
<td></td>
<td></td>
<td></td>
<td>314.1</td>
</tr>
<tr>
<td></td>
<td>cc-pVQZ</td>
<td></td>
<td>2800</td>
<td></td>
<td></td>
<td></td>
<td>1768</td>
</tr>
<tr>
<td>1D-alanine₄</td>
<td>6-311G</td>
<td>77</td>
<td>326</td>
<td>34.52</td>
<td>15</td>
<td>27</td>
<td>23.4</td>
</tr>
<tr>
<td></td>
<td>cc-pVDZ</td>
<td></td>
<td>410</td>
<td></td>
<td></td>
<td></td>
<td>43.4</td>
</tr>
<tr>
<td></td>
<td>cc-pVTZ</td>
<td></td>
<td>1030</td>
<td></td>
<td></td>
<td></td>
<td>280.0</td>
</tr>
<tr>
<td></td>
<td>cc-pVQZ</td>
<td></td>
<td>2170</td>
<td></td>
<td></td>
<td></td>
<td>1317</td>
</tr>
<tr>
<td>3D-alanine₄</td>
<td>6-311G</td>
<td>77</td>
<td>326</td>
<td>22.91</td>
<td>11</td>
<td>19</td>
<td>9.88</td>
</tr>
<tr>
<td></td>
<td>cc-pVDZ</td>
<td></td>
<td>410</td>
<td></td>
<td></td>
<td></td>
<td>18.4</td>
</tr>
<tr>
<td></td>
<td>cc-pVTZ</td>
<td></td>
<td>1030</td>
<td></td>
<td></td>
<td></td>
<td>95.2</td>
</tr>
<tr>
<td></td>
<td>cc-pVQZ</td>
<td></td>
<td>2170</td>
<td></td>
<td></td>
<td></td>
<td>413.3</td>
</tr>
<tr>
<td>1D-alanine₈</td>
<td>6-311G</td>
<td>153</td>
<td>646</td>
<td>61.64</td>
<td>24</td>
<td>46</td>
<td>502.2</td>
</tr>
<tr>
<td></td>
<td>cc-pVDZ</td>
<td></td>
<td>810</td>
<td></td>
<td></td>
<td></td>
<td>843.9</td>
</tr>
<tr>
<td></td>
<td>cc-pVTZ</td>
<td></td>
<td>2030</td>
<td></td>
<td></td>
<td></td>
<td>†</td>
</tr>
<tr>
<td></td>
<td>cc-pVQZ</td>
<td></td>
<td>4270</td>
<td></td>
<td></td>
<td></td>
<td>†</td>
</tr>
<tr>
<td>3D-alanine₈</td>
<td>6-311G</td>
<td>153</td>
<td>646</td>
<td>30.56</td>
<td>13</td>
<td>24</td>
<td>82.6</td>
</tr>
<tr>
<td></td>
<td>cc-pVDZ</td>
<td></td>
<td>810</td>
<td></td>
<td></td>
<td></td>
<td>142.4</td>
</tr>
<tr>
<td></td>
<td>cc-pVTZ</td>
<td></td>
<td>2030</td>
<td></td>
<td></td>
<td></td>
<td>946.7</td>
</tr>
<tr>
<td></td>
<td>cc-pVQZ</td>
<td></td>
<td>4270</td>
<td></td>
<td></td>
<td></td>
<td>†</td>
</tr>
</tbody>
</table>

* We used Q-CHEM default THRESH of 8 and RO default THRESH of 6 as it has been shown previously[7] that these default setup give comparable level of accuracy. To confirm the reliability of our RO method, we compute \( \epsilon = -\log_{10}[|E/E_{REF} - 1|] \) for 1D-alanine₄/6-311G, 1D-alanine₄/cc-pVDZ, 3D-alanine₈/6-311G, 3D-alanine₈/6-311G, (H₂O)₅/cc-pVQZ, (H₂O)₁₀/cc-pVTZ and (H₂O)₂₀/cc-pVDZ. The \( \epsilon_J \) are 8.56, 8.87, 8.39, 8.32, 9.15, 7.92, 8.80 and \( \epsilon_K \) are 7.39, 7.41, 6.94, 6.65, 8.51, 7.08, 7.60, respectively. These are in line with the earlier result.[7]

† Fock matrix for 1D-alanine₈ with cc-pVTZ and cc-pVQZ basis sets, and 3D-alanine₈ with cc-pVQZ could not be computed using RO on a single place due to lack of memory.

‡ Computation failed with Q-Chem due to a numerical stability problem with ‘negative overlap matrix’ reported.
relatively large and non-globular molecules.

For each molecular system in Table 1, we can treat $O$ and $K$ as constants. Therefore, we observe that the computational time for RO increases approximately quadratically with the number of basis functions $N$, as expected. Q-Chem computation time rises much more rapidly, as conventional calculation is quartic in the number of basis functions for a fixed molecule.

By comparison, for a fixed basis set, the relationship between system size and computation time is not clear for either RO or conventional methods. In Table 1, the observed increase in computation time for both conventional and RO methods is more than quadratic. Doubling the number of atoms (i.e. from $(H_2O)_5$ to $(H_2O)_{10}$) doubles both the number of occupied orbitals $O$ and the number of basis functions $N$. It also tends to increase the resolution parameters $K$ which depends on the molecule radius. However, the integral screening mentioned earlier may substantially reduce the actual computational cost when there are more atoms in the molecules. The conventional approach which is based on four-center two-electron integrals are likely to benefit from screening more than the RO approach which is based on three-center overlap integral. We therefore elect to use dense linear algebra operations for $K$ matrix computation for reasons of efficient implementation.

A comparison of computation times for 1D- versus 3D-alanine shows that RO is more effective for 3D molecules, because the required values of $N'$ and $L$ increase with the molecular radius. This is in contrast to traditional cutoff strategies, which are more effective for 1D chain-like molecules. RO’s effectiveness for 3D molecules is a major advantage for biological applications, where 3D structures are the norm.

For all molecules tested, RO is slower than Q-Chem with small, low-quality basis sets, and faster than Q-Chem with large, high-quality basis sets due to its superior scaling with the number of basis functions.

In the next two subsections, we consider the parallel speedup achievable by using multiple cores of a shared memory system with particular focus on auxiliary integral and $J$ matrix calculations for parallel execution §4.2 and by using multiple places with a particular focus on distribution of $K$ matrix calculations §4.3.

### 4.2 Shared-Memory Scaling

To measure parallel scaling, we selected a single test case from Table 1: the 3D-alanine$_4$ polypeptide with the cc-pVQZ basis set. Figure 10(a) shows the scaling with number of threads of the major components of Fock matrix construction for RO long-range energy calculation on a single node of Raijin, using the basic parallel loop shown in Figure 3. Figure 10(b) presents the same data in terms of parallel efficiency, showing the total thread time (elapsed time × number of threads). A component that exhibits perfect linear scaling shows constant total thread time, while an increase in total thread time represents a loss of parallel efficiency.

In Figure 10, total Fock matrix construction time reduces substantially from 608.9 s on one thread to a minimum of 56.5 s on 15 threads, increasing slightly to 60.3 s on 16 threads. A node of Raijin has 16 physical cores, so ideal scaling would see linear speedup from 1 to 16 threads. Measured speedup reduces substantially above 8 threads; this appears to be largely due to poor scaling of $K$ matrix computation. While the time for $K$ matrix computation
drops from 78.9 s on a single thread to 10.8 s on 10 threads, it increases again to 15.8 s on 16 threads. The bulk of \textit{K} matrix computation for a single place is a call to a multithreaded BLAS DSYRK operation with input matrices of dimension $2170 \times 2170$. The increase in computation time suggests that the multithreaded implementation of DSYRK may be unsuitable for use on multiple sockets of \textit{Raijin}. Therefore we restricted all subsequent runs to 8 threads per place, and for multi-place tests we ran two places per node, one bound to each socket. \textit{J} matrix computation time also increases above 8 threads and total thread time increases substantially from 1 to 16 threads, which equates to a drop in parallel efficiency to just 22.8%. The poor scaling of \textit{J} matrix computation is due to the high overhead of activity management relative to the cost of computation, as described in \S 3.1.2. Auxiliary integral computation scales much better, with computation time reducing steadily from 1 to 16 threads.

We next consider the overhead associated with work stealing. We measured the performance of the static load balancing approaches using either block or cyclic division of the auxiliary and \textit{J} loops, as well as the recursive bisection loop transformation. Table 2 compares component timings for Fock matrix construction on a single socket of \textit{Raijin} (8 cores/threads) for the four different methods of dividing the integral and \textit{J} matrix loops that were described in \S 3.1. Auxiliary integral computation is fastest using the basic parallel loop; followed by recursive bisection, then the static partitioning approaches. The \textit{J} matrix computation is slowest using the basic parallel loop; block static partitioning is slightly faster, due to the reduced cost of stealing; cyclic partitioning is twice as fast as the basic loop, due to improved load balance between worker threads. Recursive bisection was by far the fastest approach for \textit{J} matrix calculation, more than four times faster than the basic parallel loop.
Table 2: Multithreaded component scaling of RO long range energy calculation on *Raijin* with different methods of dividing integral and $J$ matrix loops between worker threads (8 threads, 3D-alanine$_4$, cc-pVQZ, $N^\prime = 11$, $L = 19$)

<table>
<thead>
<tr>
<th>Component</th>
<th>Basic (Figure 3)</th>
<th>Cyclic (Figure 4)</th>
<th>Block (Figure 5)</th>
<th>Bisection (Figure 6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auxiliary integrals</td>
<td>60.35</td>
<td>69.51</td>
<td>68.35</td>
<td>61.24</td>
</tr>
<tr>
<td>$J$ matrix</td>
<td>3.60</td>
<td>2.35</td>
<td>2.81</td>
<td>1.13</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>63.95</strong></td>
<td><strong>71.86</strong></td>
<td><strong>71.16</strong></td>
<td><strong>62.37</strong></td>
</tr>
</tbody>
</table>

Overall, recursive bisection is the fastest; both dynamic partitioning approaches (bisection and basic) are significantly faster than the static partitioning approaches (cyclic and block).

To assist in understanding the performance of the different loop partitioning approaches, the locality of the activities executed during auxiliary integral computation was recorded for visualization. Figure 11(a) shows the activities to compute integrals for functions $\mu, \nu$ over 8 worker threads using the basic parallel loop shown above, for a smaller problem size of five water molecules with the cc-pVQZ basis set. Activities are plotted in different colors according to which worker thread executed the activity. From the plot it is apparent that the activities for higher values of $\mu$ are all executed by the master thread, while the other activities are randomly distributed between the remaining threads. Figure 11(b) shows that with the cyclic decomposition, activities are dealt evenly to all threads, while Figure 11(c) shows that with the block decomposition, each thread receives a roughly even-sized contiguous portion of $\mu, \nu$. Figure 11(d) shows that with recursive bisection, the bulk of the activities are divided into large, irregularly-sized, contiguous chunks.

For this problem, static cyclic decomposition (Figure 11(b)) results in a relatively even distribution of work, however, the locality of the activities displayed in Figure 11(d) explains the superior performance of recursive bisection.

We now return to the scaling and efficiency of multithreaded Fock matrix calculation for the same polyalanine system previously measured in Figure 10. Figures 12(a) and 12(b) display the parallel scaling and efficiency achieved with recursive bisection of loops to compute auxiliary integrals and $J$ matrix contributions.

The overall speedup is improved; the total time on 16 threads is 58.8 s in Figure 12(a) using recursive bisection compared to 60.3 s in Figure 10(a) with the basic parallel loop. The greatest difference is in $J$ matrix calculation; the increase in total thread time (loss of parallel efficiency) is much smaller with the recursive bisection approach. Given the superior performance of the recursive bisection approach, we use it in all subsequent measurements.

### 4.3 Distributed Memory Scaling

Figure 13 shows the strong scaling with number of X10 places on *Raijin* of Fock matrix construction, for 3D-alanine$_4$ with the cc-pVQZ basis set. Total Fock matrix construction
Figure 11: Locality of auxiliary integral calculation on *Raijin* with different methods of dividing integral loop between worker threads (8 threads, \([\text{H}_2\text{O}]_5\), cc-pVQZ, \(N' = 9, L = 13\)). Activities executed by each worker thread are shown in a different color. White space indicates screened integrals which are not calculated.
Figure 12: Multithreaded component scaling and efficiency of RO long range energy calculation on Raijin with recursive bisection of loops for integral and $J$ matrix calculation (1–16 threads, 3D-alanine$_4$, cc-pVQZ, $N'$ = 11, $L$ = 19).

time drops from 86.5 s on a single place (one socket) to 6.9 s on 64 places. Auxiliary integral calculation decreases with increasing number of places. $K$ matrix calculation time decreases from 1 to 32 places, but increases above 64 places. Total computation time increases above 128 places.

Table 3 shows both Fock matrix and auxiliary integral load imbalance with varying numbers of places for Fock matrix construction using RO for 3D-alanine$_4$. Both measures of load imbalance tend to rise with the number of places. Auxiliary integral imbalance $L_{aux}$ is larger than Fock matrix imbalance $L_{Fock}$ for all numbers of places. Imbalance for 32 places is already quite severe, and limits the maximum parallel efficiency achievable to 72% assuming no parallel overhead or sequential bottlenecks. This suggests that the data distribution and load balancing schemes described in §3.2–§3.3 alone cannot balance load sufficiently evenly to achieve strong scaling and we should further explore decomposition by $n$ in future work.

As well as load imbalance, communication overhead represents a barrier to strong scaling. To compute the local portion of the $K$ matrix, each place performs a number of DGEMM operations to multiply different sized blocks of $B$ as described in §3.2.1. For each DGEMM, an $m \times k$ block of matrix $B$ is transferred from another place and multiplied (transposed) with the local $n \times k$ block of $B$; the resulting $m \times n$ block is accumulated to matrix $K$. Thus each DGEMM requires the transfer of $mk$ word-length matrix elements and performs $2(mnk)$ floating-point operations. Table 4 shows the floating-point and communication intensity for DGEMM operations in $K$ matrix construction for the same problem on different numbers of X10 places. For each number of places, the table shows: the calculated range of floating-point intensities (in FLOPs to words), in other words the ratio of the number of FLOPs performed by DGEMM to the number of words required to be transferred be-
Figure 13: Multi-place component scaling of RO long range energy calculation (8 threads per place, 3D-alanine$_4$, cc-pVQZ, $N' = 11, L = 19$)

Table 3: Distributed Fock matrix construction using RO: calculated load imbalance between different numbers of places on *Raijin* (3D-alanine$_4$, cc-pVQZ, $N' = 11, L = 19$)

<table>
<thead>
<tr>
<th>Places</th>
<th>Load imbalance $L_{\text{Fock}}$</th>
<th>$L_{\text{aux}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.000</td>
<td>1.002</td>
</tr>
<tr>
<td>3</td>
<td>1.002</td>
<td>1.098</td>
</tr>
<tr>
<td>4</td>
<td>1.014</td>
<td>1.055</td>
</tr>
<tr>
<td>5</td>
<td>1.007</td>
<td>1.064</td>
</tr>
<tr>
<td>6</td>
<td>1.009</td>
<td>1.163</td>
</tr>
<tr>
<td>8</td>
<td>1.032</td>
<td>1.166</td>
</tr>
<tr>
<td>12</td>
<td>1.051</td>
<td>1.174</td>
</tr>
<tr>
<td>16</td>
<td>1.069</td>
<td>1.287</td>
</tr>
<tr>
<td>32</td>
<td>1.106</td>
<td>1.379</td>
</tr>
<tr>
<td>64</td>
<td>1.298</td>
<td>1.530</td>
</tr>
</tbody>
</table>
between places; the measured range of floating-point performance of DGEMM in GFLOP/s; and the measured range of transfer rates in Gword/s. Theoretical peak FLOP/s on an eight-core Sandy Bridge socket of *Raijin* is 166 GFLOP/s, and MPI bandwidth is approximately 0.75 Gword/s. Therefore to achieve peak FLOP/s on *Raijin* requires a floating-point intensity of more than 220 FLOP/word.

**Table 4:** Distributed $K$ matrix construction using RO: DGEMM floating-point and communication intensity with different numbers of places on *Raijin* (3D-alanine$_4$, cc-pVQZ, $N' = 11$, $L = 19$)

<table>
<thead>
<tr>
<th>Places</th>
<th>Floating-point intensity (calculated) in FLOP/word</th>
<th>Computation rate (measured) in GFLOP/s</th>
<th>Transfer rate (measured) in Gword/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2170 – 2170</td>
<td>102.4–150.4</td>
<td>0.041–0.072</td>
</tr>
<tr>
<td>4</td>
<td>1070 – 1100</td>
<td>110.2–145.4</td>
<td>0.051–0.138</td>
</tr>
<tr>
<td>8</td>
<td>530 – 548</td>
<td>90.4–134.6</td>
<td>0.070–0.342</td>
</tr>
<tr>
<td>16</td>
<td>260 – 286</td>
<td>48.7–106.6</td>
<td>0.073–0.665</td>
</tr>
<tr>
<td>32</td>
<td>110 – 150</td>
<td>12.1– 84.1</td>
<td>0.044–1.253</td>
</tr>
<tr>
<td>64</td>
<td>52 – 88</td>
<td>2.4– 68.7</td>
<td>0.017–1.206</td>
</tr>
</tbody>
</table>

The FLOP/word ratio drops approximately linearly with number of places, as the average size of blocks of $B$ drops linearly with the number of places. The measured performance of DGEMM and minimum transfer rate also drop with shrinking block size. The decreasing floating-point intensity means that strong scaling hits a limit for this problem size at 32 places; after this, no further reduction in $K$ matrix computation time is possible.

### 5 Concluding Remarks

Our implementation of the resolution [1] reduces the cost of long-range Hartree–Fock energy calculations compared to conventional calculation of two-electron integrals. Calculations of long-range Coulomb and exchange energies of dense 3D molecular systems in high-quality basis sets – e.g. polyalanine in the correlation-consistent polarized valence basis sets (cc-pVXZ) – benefit the most from this new development. We have shown how RO may be effectively parallelized both to reduce computation time (strong scaling) and allow the treatment of larger problems (weak scaling).

The computation of Fock matrix elements is an irregular problem, and the X10 programming language provides significant support for irregular parallelism on distributed memory computers. X10’s work stealing runtime provides load balancing between worker threads within a place, while data decomposition and transfer between places is supported by X10’s explicit representation of locality.

In future, we hope to exploit additional parallelism by further dividing integral calculation by the radial component $n$. We expect this to help the RO calculation to scale beyond 1000 cores of CPUs. The auxiliary integral load imbalance $L_{aux}$ will be less important as the
auxiliary integrals are computed once and stored for use in later SCF cycles. We also hope to study the application of RO to a wider range of molecule types, in addition to the polypeptides and water clusters used here.

ACKNOWLEDGMENTS

This work was partially supported by the Australian Research Council and IBM through Linkage grant LP0989872, the NCI National Facility through NCMAS grant y42, and the Mahidol University International College through Seed Grant 008/2014. We would like to thank Dr. David Grove at IBM Watson Research Center, Dr. Ching-Yeh (Leaf) Lin at the NCI National Facility, and Prof. Peter Gill and Dr. Andrew Gilbert at the ANU for their academic and technical supports.

References


[50] The Cambridge Cluster Database, *ab initio* optimized (H$_2$O)$_n$ clusters.