dCSE Project on Improving Scaling in Conquest for HECToR Phase 3—OpenMP-MPI Hybrid Implementation

Lianheng Tong

Thursday, 2012/09/27

Abstract

This report describes the work done in the distributed Computational Science and Engineering (dCSE) project aimed to improve the scaling performances of the linear scaling ab initio Density Function Theory code CONQUEST by implementing the hybrid OpenMP-MPI parallelisation architecture. After profiling the most time consuming part of CONQUEST was found to be the matrix multiplication subroutines, and OpenMP work-sharing schemes were implemented in these subroutines. Several different implementations using OpenMP, including tasks had been tried and compared. Tests showed the hybrid parallelisation implementation significantly improved weak-scaling performances of CONQUEST running on HECToR.

1 Introduction

There are various classes of important problems in material science and biochemistry where ab initio quantum mechanical studies on a large-scale system is required for the understanding the observed properties of a given material or molecule. For example, to understand the formation of hut-cluster self-assembly of Germanium (Ge) on Silicon (Si) 001 surface[6], we need to simulate at least one complete Ge hut-cluster with several layers of the Si surface in a simulation cell. The hut-cluster is about 20nm $\times$ 20nm in size, and including the Si substrate this amounts to over 20,000 atoms in total. For studies involving bio-chemical molecules, the non-crystalline nature of the system, and the importance of including several different action groups and solvants means that a small repeating simulation cell is never satisfactory. A dozen DNA base pairs would already involve a few thousand atoms.

CONQUEST[1] is a linear scaling density functional theory (DFT)[5] code ideally suited for ab initio studies on such large systems. A good scaling property is essencial for codes like CONQUEST running on HPC platforms such as HECToR, so that the size of a given system can be increased by increasing the number of cores used accordingly while keeping the amount of work on each core constant. This way we could hopefully simulate a very large system with use of many cores without using more time. This property is referred to in the literature as *weak-scaling*.

CONQUEST used to have almost perfect weak-scaling on HECToR Phase 2a (see table 1), where there are four cores per each node; however since Phase 2b—24 cores per node—the scaling became worse (see table 2). Scaling can be improved on Phase 2b if one runs calculations with undersaturated nodes. For example, as seen from table 2 if we limit the number of cores per node to 4, then weak-scaling improves significantly. This suggested there is a penalty for saturating the newer

Table 1: Weak-Scaling test for Conquest running on HECToR Phase 2a. The test system is bulk Si.

Atoms	Time/Core (s)	Cores
4096	7069	8
32768	6894	64
262144	6931	512
2097152	7032	4096

generation of CPUs with MPI processes, caused by the fact that there are a limited number of inter-node communication channels. When all of the cores tries to communicate with each other between the nodes there will inevitably be congestion. By under-saturating the nodes we are effectively increasing the number of inter-node communication channels per MPI process.

Table 2: Weak-Scaling test for Conquest running on HECToR Phase 2b. The test system is bulk Si. PPN stands for MPI processes per node.

Atoms	Time/Core (s)	Cores	Nodes/PPN	Time/Core (s)	Cores	Nodes/PPN
4096	1400	144	6/24	1132	128	32/4
8192	1416	264	11/24	1126	256	64/4
16384	1655	528	22/24	1391	512	128/4
32768	1776	1032	43/24	1197	1024	256/4

HECToR Phase 3 now has nodes each having two 16 core CPUs, in other words 32 cores. We therefore expect scaling of Conquest to be even worse on the new system. This is a problem not unique to Conquest, but common to nearly all pure MPI codes on the new generation of multi-core CPUs. In addition to the issues related to scaling performances, another important limitation for a pure MPI code is memory usage. The new generation of CPUs tend to have more cores per node, but less memory per core. Each node on HECToR Phase 3 has a total of 32 GB of RAM organised in 4 NUMA regions, and this means if running fully saturated, each MPI process could only have 1 GB of memory available. This is to be compared with 8 GB of memory in total for each quad-core node on Phase 2a, and 32 GB of total memory for 24 cores on each node on Phase 2b. This means if we want to use all cores on HECToR Phase 3, we would need to reduce the size of the simulation system per core or use a coarser integration grid. Neither solution is satisfactory. The former introduces computation inefficiencies as the amount of continuous work for each core will be reduced—leading to more cache misses; and the latter reduces the accuracy of a calculation.

The simplest solution to the above limitations without any modification to the code is of course to run calculations with fewer MPI processes per node. This is very expensive and highly inefficient as a large number of cores will be left unused. It follows therefore many originally pure MPI codes on HECToR are now migrating to a hybrid parallelisation model mixing the distributed memory parallelisation (DMP) structure of MPI with shared memory parallelisation (SMP) solutions such as OpenMP. This way we can run fewer MPI processes per node, thus reducing the amount of competition in inter-node communications and making more memory available per process; and at

the same time utilising the remaining cores on each node by using OpenMP threads to share the work load of each MPI master processes. Note that in this case only the master threads handle the MPI communications.

The aim of this dCSE project is to first analyse CONQUEST's scaling performance on HECToR Phase 3, find the hottest part (i.e. the most time consuming part) of the code and then implement the hybrid SMP-DMP parallelisation (OpenMP-MPI) architecture in CONQUEST.

In this report, I will first present performance profiling results of the current CONQUEST code—which from now on will be referred to as "/vanilla/" version—on HECToR 3. Then I will discuss possible approaches to add multi-threaded OpenMP parallelism into CONQUEST, and finally I will show the performance improvements of the code after the required work had been done.

2 Scaling Properties of Vanilla Conquest on HECToR Phase 3

Test calculations on a system of bulk Silicon were done to test performances of CONQUEST on HECToR Phase 3. The calculations were ab initio static (the Si ions are fixed in space) self-consistent density functional ground-state energy calculations, with integration grid of 0.32 Bohrs. The basis used was single zeta (SZ), which gives each Silicon atom 4 basis functions.

The code was compiled by Cray compiler suite 4.0.46, with Cray LibSci 11.0.06. I have also tried PGI compiler and ACML library, however I found the Cray compiler and science library gave the fastest code.

The most important scaling property for CONQUEST concerning the developer and researchers is weak-scaling, that is how the amount of time it takes for a calculation to finish varies with respect to size of the simulation system, with amount of work allocated to each core being fixed. The size of the system is increased by increasing the number of cores used in the calculation.

There is a technical complication when comparing the wall times for the calculations. There are three main iteration loops in CONQUEST: loop for calculating the inverse of the overlap matrix using Hoteling's method[4], loop for McWeeny initialisation[7] and the combined loop for energy minimisation[2] and charge self-consistency. As the system size increases the number of iterations for each of the loops for a given accuracy tolerance also increases. Hence the raw wall times obtained from the calculations cannot be compared directly. It is not possible to fix the number of iterations for each calculation, because if a calculation reaches the maximum allowed iterations while still not achieving the required tolerance CONQUEST would exit abruptly, and any operations that follows will *not* be performed. The recorded wall time in this case would still be different from the case when a calculation successfully finished the calculation in the given iteration steps.

To calculate a wall time that could be compared between the different sized systems, I timed each of the iteration steps in the calculations, then by counting the number of iterations done in each calculation, I took away from the total wall time the time taken for the extra iterations done by a calculation. This way I was ensuring that the final wall times were the time taken for a complete ab initio self-consistent DFT calculation with exactly the same number of iterations in each stage of the calculations.

The table below shows the results

Atoms	Nodes	MPI Processes	Wall Time (s)	Ratio
512	1	32	855.039	1.000
4096	8	256	1114.189	1.303
32768	64	2048	1788.672	2.092

The "Ratio" column gives the ratio of wall time taken for the calculations compared with respect to that of the 512 atoms calculation.

If I undersaturated the nodes, the amount of wall time did reduce significantly, and the scaling also improved albeit by not as significant amount. In the following result I fixed the number of MPI processes per node to be 4:

Atoms	Nodes	MPI Processes	Wall Time (s)	Ratio
512	8	32	552.090	1.000
4096	64	256	617.986	1.119
32768	512	2048	1034.916	1.875

It is interest to note that the time taken for a calculation done on a saturated node was significantly greater than that done on a set of unsaturated nodes. This was true even for the 512 atoms case, where there were no inter-node communications for the saturated case. Cray PAT sampling results also seemed to suggest percentage of time spend on MPI communications for the two 512 atoms calculations were roughly the same. The following are the sampling results for the calculation with 1 node and 32 MPI processes:

```
 Samp%  |   Samp   | Imb.   |  Imb.   |Group
        |          | Samp   | Samp%   | Function
        |          |        |         |  PE=HIDE
 100.0% | 53401.8 |    -- |     -- |Total
|-------------------------------------------------------------------------------
|  96.0% | 51286.7 |    -- |     -- |USER
||------------------------------------------------------------------------------
||  47.6% | 25432.5 |  80.5 |   0.3% |m_kern_min$multiply_module_
||  39.4% | 21029.7 |  62.3 |   0.3% |m_kern_max$multiply_module_
||   1.5% |   791.5 |  35.5 |   4.4% |daxpy_k
||   1.2% |   627.8 |  26.2 |   4.1% |act_on_vectors_new$calc_matrix_elements_module_
||==============================================================================
|   3.3% |  1750.9 |    -- |     -- |MPI
||------------------------------------------------------------------------------
|   1.9% |  1030.8 |  66.2 |   6.2% | mpi_recv
||==============================================================================
|   0.7% |   364.2 |    -- |     -- |ETC
|==============================================================================
```

and below are the sampling results of same calculation but using 8 nodes with 4 MPI processes per node:

```
 Samp%  |   Samp   | Imb.   |  Imb.   |Group
        |          | Samp   | Samp%   | Function
        |          |        |         |  PE=HIDE
 100.0% | 34440.5 |    -- |     -- |Total
|-------------------------------------------------------------------------------
|  96.0% | 33073.8 |    -- |     -- |USER
||------------------------------------------------------------------------------
```

```
||  49.8% | 17161.5 |  55.5 |   0.3% |m_kern_min$multiply_module_
||  38.2% | 13158.1 |  42.9 |   0.3% |m_kern_max$multiply_module_
||   1.3% |   453.8 |  32.2 |   6.8% |act_on_vectors_new$calc_matrix_elements_module_
||   1.2% |   399.9 |  38.1 |   9.0% |daxpy_k
||   1.0% |   356.8 |  22.2 |   6.1% |get_matrix_elements_new$calc_matrix_elements_module_
||=============================================================================================
|   3.5% |  1193.5 |    -- |     -- |MPI
||---------------------------------------------------------------------------------------------
|   2.1% |   735.9 | 148.1 |  17.3% | mpi_recv
||=============================================================================================
|   0.5% |   173.2 |    -- |     -- |ETC
|=============================================================================================
```

Therefore when running unsaturated on HECToR Phase 3 the code simply ran more efficient overall.

Never-the-less, as I increased system size, we can see that MPI communications starts to take more time for the saturated calculations. The following is the CrayPAT sampling report on the 4095 atoms calculation with 8 nodes and 32 MPI processes per node (total 256 MPI processes):

```
 Samp%  |   Samp  |  Imb.  |  Imb.  |Group
        |         |  Samp  | Samp%  | Function
        |         |        |        |  PE=HIDE
 100.0% | 76646.5 |     -- |     -- |Total
|-------------------------------------------------------------
|  74.1% | 56780.6 |     -- |     -- |USER
||------------------------------------------------------------
||  37.2% | 28489.2 |  214.8 |   0.8% |m_kern_min$multiply_module_
||  29.7% | 22748.4 |  367.6 |   1.6% |m_kern_max$multiply_module_
||   1.1% |   863.7 |   57.3 |   6.2% |daxpy_k
||============================================================
|  25.3% | 19418.8 |     -- |     -- |MPI
||------------------------------------------------------------
||  16.6% | 12711.1 | 3233.9 |  20.4% |mpi_recv
||   7.1% |  5415.4 | 4300.6 |  44.4% |MPI_BARRIER
||============================================================
|   0.6% |   447.2 |     -- |     -- |ETC
|============================================================
```

And this is the report for the same calculation running on 64 nodes and 4 MPI processes per node:

```
 Samp%  |   Samp  |  Imb.  |  Imb.  |Group
        |         |  Samp  | Samp%  | Function
        |         |        |        |  PE=HIDE
 100.0% | 42299.6 |     -- |     -- |Total
|-----------------------------------------------------------------------------
|  87.5% | 37015.6 |     -- |     -- |USER
||----------------------------------------------------------------------------
```

```
||  45.4% | 19203.9 |   99.1 |   0.5% |m_kern_min$multiply_module_
||  34.5% | 14596.5 |  143.5 |   1.0% |m_kern_max$multiply_module_
||   1.2% |   503.7 |   42.3 |   7.8% |act_on_vectors_new$calc_matrix_elements_module_
||   1.0% |   432.6 |   53.4 |  11.0% |daxpy_k
||=============================================================================
|  12.0% |  5084.6 |     -- |     -- |MPI
||-----------------------------------------------------------------------------
||   6.3% |  2652.1 | 1701.9 |  39.2% |mpi_recv
||   4.8% |  2043.0 |  829.0 |  29.0% |MPI_BARRIER
||=============================================================================
|   0.5% |   199.4 |     -- |     -- |ETC
|=============================================================================
```

As we can observe the amount of time spent in MPI communications was over 25% of the total run time for the saturated calculation, compared with 12% in the unsaturated calculation. This trend continues as we increase system size. For 32768 atoms case, if running saturated on 64 nodes the MPI communication time took 53% of the total run time (1788.672 s), and became the main bottleneck of the calculation; however if running on 512 nodes, the time spent in MPI communications reduced to 47.0% of the total time (1034.916 s), with the user (computation) routines still being the most time consuming using over 52% of the total time.

All of the above results suggests that a SMP-DMP hybrid parallelisation architecture would be very likely to improve performance of CONQUEST on HECToR Phase 3.

By looking at the CrayPAT results we can see that in all cases the CONQUEST matrix multiplication kernels `m_kern_min` and `m_kern_max` are by far the most time consuming part of the code. This is not surprising, as the two major computations in the code are matrix multiplication and integration on grid. The profiling tool revealed to us that the code spends much longer in matrix multiplication (over 70% of total run time on average) than integration[1] (less than 3% of the total time on average). It follows that the work for implementing shared memory parallelisation should be concentrated on the two matrix kernels `m_kern_min` and `m_kern_max` of the `multiply_module`.

3 Structure of Conquest Matrix Multiplication Subroutines

Before we try to implement shared memory parallelisation such as OpenMP, we need to first understand the structure of the matrix multiplication routines. A detailed description of the parallel sparse matrix multiplication algorithm used by CONQUEST can be found in the work[3] by Bowler et al. For the purpose of explaining my OpenMP implementations in the following sections of this this report I will give a brief description of the algorithm here.

3.1 Conquest Matrix Data Structure

To maximise efficiency both in terms of cache usage and MPI communications, CONQUEST organises matrix data in several intermediate levels of groupings.

CONQUEST divides the simulation cell input by the user—will be referred to as the *fundamental simulation cell* (FSC)—into many spacial *partitions*. Each partition contains zero or more atoms,

[1] The subroutines associated with integrations are `act_on_vectors_new` and `daxpy_k`.

and they are assigned to the MPI processes either automatically via Hilbert curve method or manually by the user using a mapping file. Ideally one would want to assign the partitions in such a way so that the atoms are evenly distributed to the MPI processes and the atoms that are spatially close together are assigned to the same MPI process. This is to ensure good load balancing and minimise communications. The set of atoms in the FSC in charge of by a MPI process is referred to as the *primary set* of that process.

Because CONQUEST uses a representation basis set—referred to as *support functions*—that are centred around each atom, the matrices in CONQUEST thus have the label $A_{i\alpha j\beta}$, where both row index and the column index each has two sub-indices: i (or j) corresponding to the atom, and α (or β) corresponding to the support function centred on the given atom. The row atomic indices range over all atoms in the fundamental simulation cell; while the column indices ranges over *all* neighbours of the given row index atom, including those in the periodic images of the FSC. CONQUEST matrices does not store elements corresponding to the cases where i and j are non-neighbours (i.e. outside interaction range). The values of these ignored matrix elements are assumed to be zero. The matrices are distributed across the MPI processes by rows. In other words each matrix is stored on each MPI node as a continuous block of memory ranged in the four-level hierarchy of partitions–row atomic indices–column atomic index (neighbour list)–(α, β) support functions block. Please see figure 1.

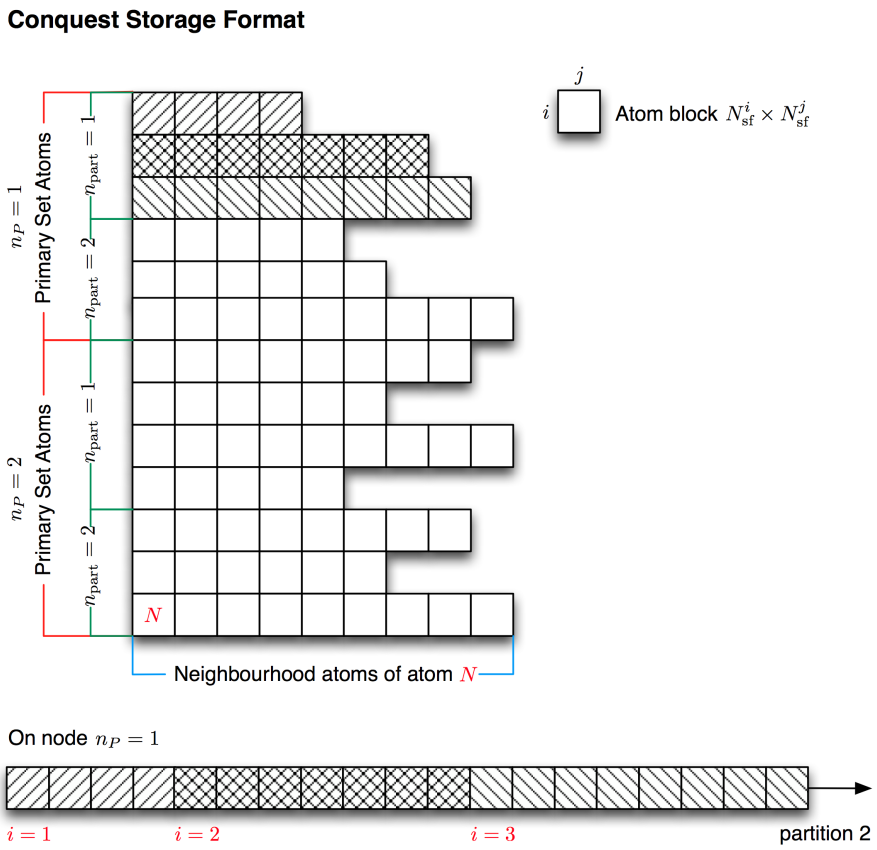

Figure 1: Conquest matrix storage scheme

The list of all atoms being a neighbour to at least one atom in the primary set is called a *halo*. In other words the halo contains all atoms that interact with atoms belonging to a given MPI process. Any atom in a halo is called a *halo atom*. A partition that contains at least one halo atom is called a *halo-partition*. And the set of all halo-partitions for a given interaction range is called the *covering set* and the union of all covering sets (of all interaction ranges) is called the *grand covering set*.

Since the concept of neighbourhood depends on interaction range, the following groupings are not only local to each MPI process but also distinct for each defined interaction range:

- Neighbour list
- Halo
- Halo-partition
- Covering set

And the following groupings are local to each MPI process but independent of interaction ranges:

- Partition
- Primary set
- Grand covering set

A halo-partition, if containing a halo atom not a member of the primary set, belongs to a remote MPI process. Since matrix elements are stored in continuous blocks of memory in terms of partitions (see figure 1), in CONQUEST partitions forms the most basic unit for MPI communications.

3.2 Matrix multiplication

The formula for matrix multiplication concerned here is

$$C_{i\alpha j\beta} = \sum_{k\gamma} A_{i\alpha k\gamma} B_{k\gamma j\beta} \tag{1}$$

Each matrix is assumed to have a distinct interaction range, and the computation of a matrix element of **C** is done by the MPI process which this element belongs to. Thus, in order to calculate $C_{i\alpha j\beta}$ for i in the primary set, and all of the j in **C**-neighbour list of i, the MPI process needs only to fetch data $B_{k\gamma j\beta}$, where k is not part of the primary set (but is member of **A**-neighbour list of i) from the remote MPI processes. By the nature of CONQUEST matrix storage, the local MPI process would already have all the required data from **A** stored. For efficiency of MPI communication, remote data are fetched in continuous blocks of halo-partitions.

The matrix multiplication in CONQUEST is done in the following scheme (the existence or two ways of doing the final multiply and add—˜m_kern_max˜ and `m_kern_min` will be explained later):

```
! send data, note if a remote process has a neighbour to an
! atom in the local process, then the local process will
! of course have a halo-partition of the remote process
do inode in list of remote nodes in charge of A-halo-partitions
   non-blocking send B data for each partitions to remote inode
```

```
end do
do kpart = 1, number of A-halo-partitions
   ! receive data
   fetch B data for A-halo-partition kpart
   ! do the actual computation
   if (C range >= A range + B range) then
      call m_kern_max
   else
      call m_kern_min
   end if
end do
```

Generally for matrix multiplication there must be three levels of nested loops: the loop over row indices of **A** (or **C**, they are the same, all primary set indices), the loop over column indices of **A** and loop over column indices of **C** (or **B**, which ever fewer). Since one needs to minimise the amount of communications, the loop over column indices of **A**, or more precisely over the k-partitions (i.e. partitions which contains primary set atoms (of a MPI process) corresponding to k index in equation (1)) must be the outermost.

Once a halo-partitions is fetched, each MPI process will have enough data to compute the partial sum (for the maximum case—see paragraphs below)

$$C^{\text{kpart}}_{i\alpha j\beta} = \sum_{k\in\text{kpart}} \sum_{\gamma} A_{i\alpha k\gamma} B_{k\gamma j\beta} \tag{2}$$

locally. And this work is done by the matrix multiplication kernel subroutines.

```
subroutine m_kern_max ! or m_kern_min
  do atoms k in kpart A-halo-partition
     Transcribe j indexing from B partition to C-halo labeling
     do atoms i in A neighbour list of k
        work out indices needed for atom i
        do atoms j in B neighbour list of k
           if (j is also in C halo) then
              do n2 in j support functions
                 calculate array addresses
                 do n1 in i support functions
                    calculate array addresses
                    do n3 in k support functions
                       multiply and add matrix elements
                    end do ! support fns for k
                 end do ! support fns for i
              end do ! support fns for j
           end if
        end do ! atoms j
     end do ! atoms i
  end do ! atoms k
end subroutine m_kern_max ! or m_kern_min
```

CONQUEST recognises two possible situations: the first is the case when $R(\mathbf{C}) \geq R(\mathbf{A}) + R(\mathbf{B})$, where R corresponds to the range of a given matrix this is referred to as the *maximum* case; and

the second is when $R(\mathbf{C}) < R(\mathbf{A}) + R(\mathbf{B})$. In the extreme of the second case we could have $R(\mathbf{C}) < |R(\mathbf{A}) - R(\mathbf{B})|$, which is referred to as the *minimum* case. The only difference between the two cases are the fact that the role of the matrices $\mathbf{A}$ and $\mathbf{C}$ are swapped and $\mathbf{B}^{\mathsf{T}}$ is used in place of $\mathbf{B}$. This is done in order to reuse cache as much as possible (see [3]). This means the order of inner multiplication and addition of matrix elements in the matrix multiplication kernel differs: For the maximum case (`m_kern_max`) we have

```
C(n1, n2, ncbeg) = C(n1, n2, nabeg) + &
                   A(n3, n1, ncbeg) * B(n3, n2, nbbeg)
```

and for the minimum case

```
A(n3, n1, ncbeg) = A(n3, n1, ncbeg) + &
                   C(n1, n2, nabeg) + B(n3, n2, nbbeg)
```

where `nabeg`, `nbbeg` and `ncbeg` are the addresses of the beginning of the support function blocks corresponding to the particular (i, k), (k, j) or (i, j) neighbour pairs in $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ respectively.

For the purpose of adding shared memory parallelisation to matrix multiplication subroutines, the distinction between the maximum and minimum case is not very important. The implementation in both cases is very similar with only subtle differences.

4 Linking Multi-Threaded BLAS Library Using DGEMM

There are a second version of `m_kern_max` and `m_kern_min` already available in CONQUEST, which can be be used instead of the default versions (described above) by setting an input flag. The modified version uses the fact that the sum over γ in equation (2) can be regarded as a matrix multiplication on its own-right:

$$C^k_{i\alpha j\beta} = \sum_{\gamma} A_{i\alpha k\gamma} B_{k\gamma j\beta}$$

Hence if instead of looping over the support function blocks and do the multiplication and summation explicitly, but store $A_{i\alpha k\gamma}$ and $B_{k\gamma j\beta}$ in temporary arrays in FORTRAN matrix format, then level 3 BLAS library routine `DGEMM` can be used to calculate the required $C^k_{i\alpha j\beta}$. The overall partial sum may be calculated by summing over k later on:

$$C^{\text{kpart}}_{i\alpha j\beta} = \sum_{k \in \text{kpart}} C^k_{i\alpha j\beta}$$

The use of `DGEMM` in the optional matrix multiplication kernels means that if we use these kernels and link with a multi-threaded BLAS library, then we would have a shared-memory parallelisation implementation with minimum work and without any modifications to the code.

4.1 Test Results

I ran test calculations by running total of 32 MPI processes on systems of 512 and 2048 Si atoms. I also tested calculations with the single zeta (SZ) basis (4 support functions per atom) and the single

zeta with polarisation (SZP) basis (9 support functions per atom) for the 512 atoms case, so that I could look at the case when the support-functions blocks are larger. For multi-threaded calculations, I increased the number of nodes (via threads) while keeping the number of MPI processes constant at 32. I made sure that the MPI processes are equally distributed across each node, and across each NUMA region. For example if running with 2 threads, the code would be using 2 nodes, with 16 MPI processes per node, 4 MPI processes per NUMA region and 2 threads per MPI process; if running with 4 threads, the code would be running on 4 nodes, with 8 MPI processes per node, 2 MPI processes per NUMA region and 4 threads per MPI process; and so on. The maximum number of threads in the tests was 8, as this was the largest number of threads each NUMA region could support. If we went beyond this number, then data would need to be shared between the NUMA regions, and there would be communications, which would lead to a dramatical reduction in multi-threaded performance.

The code was compiled with Cray compiler suite 4.0.46 and Cray LibSci 11.0.06.

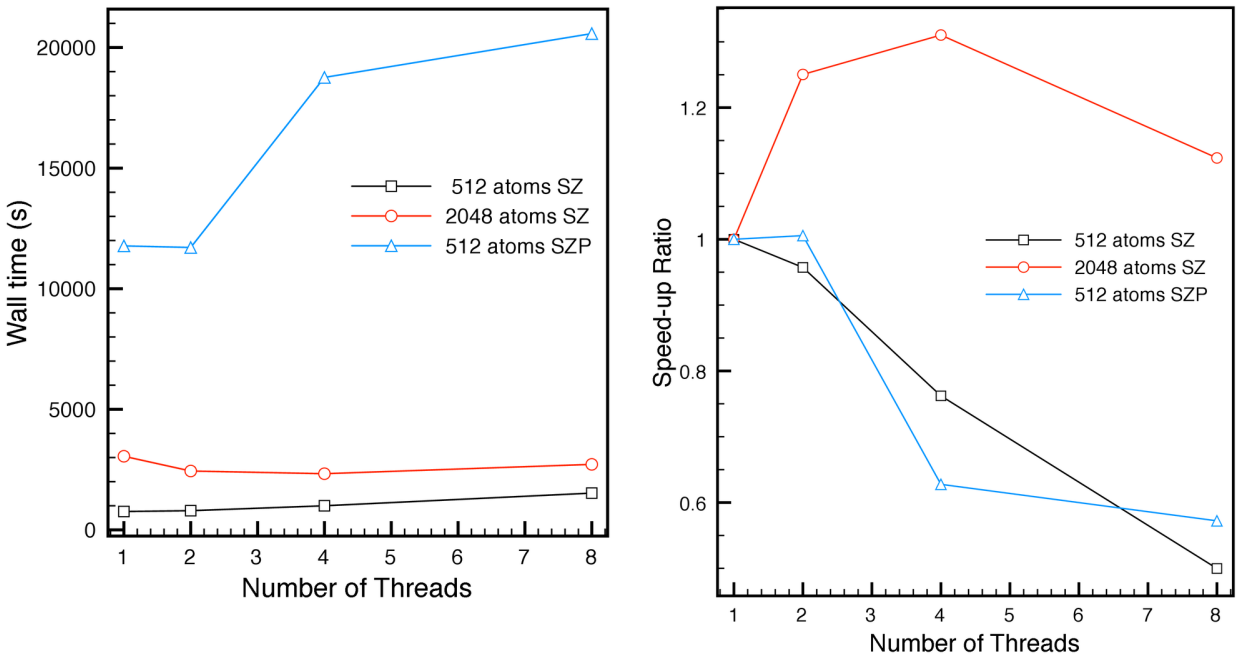

Figure 2: Strong scaling: Wall time vs. increasing number of threads used by `DGEMM`

Figure 2 shows the total wall time vs. the number of threads used by the multi-threaded `DGEMM`, and the corresponding speed-up (wall time of 1 threaded calculation divided by wall time of n threaded calculation) vs. the number of threads. To my surprise, the multi-threaded calculations in general performed worse than the single threaded version. A quick profiling by CrayPAT revealed that in all cases (except for the single threaded calculations) much of the time were spent in a function call called `_new_slave_entry`, for example see table below (for 512 atoms, SZ basis):

N Threads	Total Time (s)	Most Time Consuming Part	% Total Time
1	796.822	`m_kern_maxgemm`	29.3%
2	795.209	`_new_slave_entry`	65.3%
4	988.843	`_new_slave_entry`	77.0%
8	1526.096	`_new_slave_entry`	66.7%

It seemed that most of the time spent in the multi-threaded `DGEMM` cases were in the overhead to create threads. This might be caused by the fact that `DGEMM` call was nested in the inner most loop of the matrix kernel routines.

One would expect however that by increasing the basis size, we increase the amount of work for each `DGEMM` call, and hence the multi-threaded `DGEMM` kernels should perform better. The surprising result shows that this is not true. Again the bottle-necks for the calculations were in `_new_slave_entry`, which on average used over 65% of the total calculation time.

I had also tried PGI compilers and multi-threaded ACML libraries, however I found PGI compiler on HECToR generally produced a slower code than Cray, and when compiled with ACML library, the code ran too slow (> 4 hours for 8 threaded calculations). And CrayPAT profiling showed again the bottle-necks were in thread-creation.

Although using multi-threaded `DGEMM` version for the kernel failed to improve performance of CONQUEST, the single threaded `DGEMM` version was in fact faster than the original matrix multiplication kernels, where the inner most calculations are done explicitly (`DGEMM` version finished with wall time of 796.822 s, compared with the original version which used 855.039 s).

5 Implementing OpenMP in Matrix Multiplication Kernels

With the failure of using multi-threaded BLAS libraries to improve performance of CONQUEST, the next logical step was to implement explicit OpenMP regions in CONQUEST matrix multiplication subroutines.

We want the master threads to handle all of the MPI communications, and the slave threads to help-out the master thread in the heavy computational loops. Therefore any OpenMP parallel region must be created *after* data from remote **A**-halo-partitions have been fetched from the remote MPI processes, and hence be inside the matrix multiplication kernel subroutines.

Inside the multiplication kernel, there are three main loops, the outer most loops over the atoms k in each of the **A**-halo-partitions, the middle loops over primary set atoms i which are **A**-neighbours of k and the inner loop loops over atoms j, which are both **C**-neighbours of i and **B**-neighbours of k. In general, we want to use create OpenMP parallel regions on the outer most loop possible, so that we can minimise the amount of overhead related to thread creation at `!$omp parallel` and the implicit OpenMP barrier at `!$omp end parallel`.

It is perfectly okay to make the parallel region to enclose the entire multiplication kernel subroutine. Most of the passed variables of the multiplication kernel are input (i.e. read) only, these are safe to be defined as `shared` in the parallel region. The matrices must be defined as `shared` as they are the main data the threads must work on as a team. For the maximum case, the matrix elements of **A** and **B** will be read only, only elements of **C** will be accumulated with respect to $k\gamma$ component. And for the minimum case, **C** and **B** will be read only, and only **A** will be accumulated, this time with respect to $j\beta$ component. We only have to be careful about avoiding racing conditions in components where accumulation are done.

To obtain the correct data from the matrix arrays for each support-function blocks the original implementation uses simple incrementation of address indices after each of the i and j loops. However, since loops are going to be work-shared in the OpenMP implementation, the original book-keeping method no longer works. Therefore extra i and j loops need to be done to work out the starting positions of the support-function blocks before the work-sharing starts, and store these array indices in temporary arrays.

5.1 Maximum Case

For the maximum case (`m_kern_max`), upon the input of arrays **A**, **B** and **C** the kernel calculates the equation

$$C^{\text{kpart}}_{i\alpha j\beta} = \sum_{k\in\text{kpart}} \sum_{\gamma} A_{i\alpha k\gamma} B_{k\gamma j\beta}$$

the partial sum computed in the kernel is accumulated over both γ and k. This means if a OpenMP work-sharing construct is to be implemented on the outer-most k-loop, then matrix **A** must be included in the list for `reduction`. While reduction is allowed since OpenMP 2.0 for FORTRAN arrays (mainly because the intrinsic operators "+", "-", "*", "/" etc. can operate on arrays), using reduction in this case would be inefficient and dangerous. The reason is that with reduction clause, private copies of matrix **C** (maximum case) or **A** (minimum case) are created (intrinsically) on each participating thread, and after the work-sharing these private matrices must then be summed over, one full matrix summation for every thread. More importantly, due to the sparse nature of the matrices, the number of neighbours of each atom k could be different, resulting in different data structures of **C** (or **A**) matrices on each participating thread, and the final "+" operation for the reduction would be too primitive to handle the different data structures in the matrices, hence causing race conditions. It follows therefore the work-sharing construct should be done either on i-loop or j-loop. The inner support-function loops (that associated with `n1`, `n2` and `n3`) should not be work-shared, as it would be more efficient to apply loop-un-rolling and vectorisation using compiler optimisations. In general since work-sharing (`!omp do` in this case) in OpenMP also has an overhead, it is good practice to implement work-sharing on the outer most loop possible. It therefore follows the most logical loop to work-share is the i-loop.

The OpenMP implementation to the maximum multiplication kernel is hence done as follows:

```
subroutine m_kern_max(k_off, kpart, ib_nd_acc, ibaddr, nbnab,       &
                      ibpart, ibseq, bndim2, a, b, c, ahalo, chalo, &
                      at, mx_absb, mx_part, mx_iprim, lena, lenb,   &
                      lenc, debug)

  use datatypes
  use matrix_module
  use basic_types,    only: primary_set
  use primary_module, only: bundle

  implicit none

  ! Passed variables
  type(matrix_halo)  :: ahalo, chalo
```

```
  type(matrix_trans) :: at
  integer            :: mx_absb, mx_part, mx_iprim, lena, lenb, lenc
  integer            :: kpart, k_off
  real(double)       :: a(lena)
  real(double)       :: b(lenb)
  real(double)       :: c(lenc)
  integer, optional  :: debug
  ! Remote indices
  integer(integ) :: ib_nd_acc(mx_part)
  integer(integ) :: ibaddr(mx_part)
  integer(integ) :: nbnab(mx_part)
  integer(integ) :: ibpart(mx_part*mx_absb)
  integer(integ) :: ibseq(mx_part*mx_absb)
  integer(integ) :: bndim2(mx_part*mx_absb)
  ! Local variables
  integer :: jbnab2ch(mx_absb)  ! Automatic array
  integer :: nbkbeg, k, k_in_part, k_in_halo, j, jpart, jseq
  integer :: i, nabeg, i_in_prim, icad, nbbeg, j_in_halo, ncbeg
  integer :: n1, n2, n3, nb_nd_kbeg
  integer :: nd1, nd2, nd3
  integer :: naaddr, nbaddr, ncaddr
  ! OpenMP required indexing variables
  integer :: nd1_1st(at%mx_halo), nd2_1st(mx_absb)

!$omp parallel default(none)                                              &
!$omp          shared(kpart, ibaddr, ib_nd_acc, nbnab, ibpart, ibseq,     &
!$omp                 k_off, bndim2, mx_absb, mx_part, at, ahalo, chalo,  &
!$omp                 a, b, c)                                            &
!$omp          private(i, j, k, j_in_halo, k_in_halo, k_in_part, nbkbeg,  &
!$omp                  nb_nd_kbeg, nd1, nd2, nd3, jpart, jseq, jbnab2ch,  &
!$omp                  nabeg, nbbeg, ncbeg, i_in_prim, icad, naaddr,      &
!$omp                  nbaddr, ncaddr, n1, n2, n3, nd1_1st, nd2_1st)
  ! Loop over atoms k in current A-halo partn
  do k = 1, ahalo%nh_part(kpart)
     k_in_halo = ahalo%j_beg(kpart) + k - 1
     k_in_part = ahalo%j_seq(k_in_halo)
     nbkbeg = ibaddr(k_in_part)
     nb_nd_kbeg = ib_nd_acc(k_in_part)
     nd3 = ahalo%ndimj(k_in_halo)
     ! for OpenMP sub-array indexing
     nd1_1st(1) = 0
     do i = 2, at%n_hnab(k_in_halo)
       i_in_prim = at%i_prim(at%i_beg(k_in_halo)+i-2)
       nd1_1st(i) = nd1_1st(i-1) + nd3 * ahalo%ndimi(i_in_prim)
     end do
     nd2_1st(1) = 0
     do j = 2, nbnab(k_in_part)
        nd2_1st(j) = nd2_1st(j-1) + nd3 * bndim2(nbkbeg+j-2)
     end do
     ! transcription of j from partition to C-halo labelling
     do j = 1, nbnab(k_in_part)
```

```
        jpart = ibpart(nbkbeg+j-1) + k_off
        jseq = ibseq(nbkbeg+j-1)
        jbnab2ch(j) = chalo%i_halo(chalo%i_hbeg(jpart)+jseq-1)
     end do
!$omp do schedule(runtime)
     ! Loop over primary-set A-neighbours of k
     do i = 1, at%n_hnab(k_in_halo)
        ! nabeg = at%i_beg(k_in_halo) + i - 1
        i_in_prim = at%i_prim(at%i_beg(k_in_halo)+i-1)
        nd1 = ahalo%ndimi(i_in_prim)
        icad = (i_in_prim-1) * chalo%ni_in_halo
        ! nabeg index for openMP is calculated inside loop, here
        nabeg = at%i_nd_beg(k_in_halo) + nd1_1st(i)
        ! Loop over B-neighbours of atom k
        do j = 1, nbnab(k_in_part)
           ! nbbeg = nbkbeg + j - 1
           nd2 = bndim2(nbkbeg+j-1)
           nbbeg = nb_nd_kbeg + nd2_1st(j)
           j_in_halo = jbnab2ch(j)
           if (j_in_halo /= 0) then
              ncbeg = chalo%i_h2d(icad+j_in_halo)
              !nd2 = chalo%ndimj(j_in_halo)
              if (ncbeg /= 0 ) then ! multiplication of ndim x ndim blocks
                 do n2 = 1, nd2
                    nbaddr = nbbeg + nd3 * (n2 - 1)
                    ncaddr = ncbeg + nd1 * (n2 - 1)
                    do n1 = 1, nd1
                       naaddr = nabeg + nd3 * (n1 - 1)
                       do n3 = 1, nd3
                          c(ncaddr+n1-1) = c(ncaddr+n1-1) + &
                                           a(naaddr+n3-1) * b(nbaddr+n3-1)
                       end do
                    end do
                 end do
              end if
           end if ! End of if(j_in_halo.ne.0)
        end do ! End of j = 1, nbnab
     end do ! End of i = 1, at%n_hnab
!$omp end do
  end do ! End of k = 1, nahpart
!$omp end parallel
  return
end subroutine m_kern_max
```

5.2 Minimum Case

For the minimum case (`m_kern_min`), upon input of arrays **A**, **B**, and **C** the kernel calculates the equation

$$A_{i\alpha k\gamma} = \sum_{j} \sum_{\beta} C_{i\alpha j\beta} * B_{k\gamma j\beta}$$

where $k \in$ kpart. The important distinction between `m_kern_min` and `m_kern_max` is that this time it is the j and β indices that are summed over. Following the same arguments as that of the maximum case, OpenMP work-sharing region could be setup at either the outermost k-loop, or the medium level i-loop. Initially for sake of symmetry between the minimum case and the maximum case I choose the i-loop for the `!$omp do` workshare. The comparison between the performances resulting from work-sharing the different loops will be shown in section 6.1.1.

The OpenMP implementation to minimum multiplication kernel is thus given as:

```
subroutine m_kern_min(k_off, kpart, ib_nd_acc, ibaddr, nbnab,       &
                      ibpart, ibseq, bndim2, a, b, c, ahalo, chalo, &
                      at, mx_absb, mx_part, mx_iprim, lena, lenb,   &
                      lenc)
  use datatypes
  use matrix_module
  use basic_types,    only: primary_set
  use primary_module, only: bundle

  implicit none

  ! Passed variables
  type(matrix_halo)  :: ahalo, chalo
  type(matrix_trans) :: at
  integer            :: mx_absb, mx_part, mx_iprim, lena, lenb, lenc
  integer            :: kpart, k_off
  ! Remember that a is a local transpose
  real(double)       :: a(lena)
  real(double)       :: b(lenb)
  real(double)       :: c(lenc)
  ! dimension declarations
  integer :: ibaddr(mx_part)
  integer :: ib_nd_acc(mx_part)
  integer :: nbnab(mx_part)
  integer :: ibpart(mx_part*mx_absb)
  integer :: ibseq(mx_part*mx_absb)
  integer :: bndim2(mx_part*mx_absb)
  ! Local variables
  integer :: jbnab2ch(mx_absb)
  integer :: k, k_in_part, k_in_halo, nbkbeg, j, jpart, jseq
  integer :: i, nabeg, i_in_prim, icad, nbbeg, j_in_halo, ncbeg
  integer :: n1, n2, n3, nb_nd_kbeg
  integer :: nd1, nd2, nd3
  integer :: naaddr, nbaddr, ncaddr
  ! For OpenMP
```

```
  integer :: nd1_1st(at%mx_halo), nd2_1st(mx_absb)

!$omp parallel default(none)                                                &
!$omp          shared(kpart, ibaddr, ib_nd_acc, nbnab, ibpart, ibseq,       &
!$omp                 k_off, bndim2, mx_absb, mx_part, at, ahalo, chalo,    &
!$omp                 a, b, c)                                              &
!$omp          private(j_in_halo, k_in_halo, k_in_part, nbkbeg,             &
!$omp                  nb_nd_kbeg, nd1, nd2, nd3, i, j, k, jpart, jseq,     &
!$omp                  jbnab2ch, icad, nabeg, nbbeg, ncbeg, naaddr, nbaddr, &
!$omp                  ncaddr, n1, n2, n3, i_in_prim, nd1_1st, nd2_1st)
  ! Loop over atoms k in current A-halo partn
  do k = 1, ahalo%nh_part(kpart)
     k_in_halo = ahalo%j_beg(kpart) + k - 1
     k_in_part = ahalo%j_seq(k_in_halo)
     nbkbeg = ibaddr(k_in_part)
     nb_nd_kbeg = ib_nd_acc(k_in_part)
     nd3 = ahalo%ndimj(k_in_halo)
     ! for OpenMP sub-array indexing
     nd1_1st(1) = 0
     do i = 2, at%n_hnab(k_in_halo)
        i_in_prim = at%i_prim(at%i_beg(k_in_halo)+i-2)
        nd1_1st(i) = nd1_1st(i-1) + nd3 * ahalo%ndimi(i_in_prim)
     end do
     nd2_1st(1) = 0
     do j = 2, nbnab(k_in_part)
        nd2_1st(j) = nd2_1st(j-1) + nd3 * bndim2(nbkbeg+j-2)
     end do
     ! transcription of j from partition to C-halo labelling
     do j = 1, nbnab(k_in_part)
        jpart = ibpart(nbkbeg+j-1) + k_off
        jseq = ibseq(nbkbeg+j-1)
        jbnab2ch(j) = chalo%i_halo(chalo%i_hbeg(jpart)+jseq-1)
     end do
!$omp do schedule(runtime)
     ! Loop over primary-set A-neighbours of k
     do i = 1, at%n_hnab(k_in_halo)
        !nabeg=at%i_beg(k_in_halo)+i-1
        i_in_prim = at%i_prim(at%i_beg(k_in_halo)+i-1)
        nd1 = ahalo%ndimi(i_in_prim)
        icad = (i_in_prim-1) * chalo%ni_in_halo
        nabeg = at%i_nd_beg(k_in_halo) + nd1_1st(i)
        ! Loop over B-neighbours of atom k
        do j = 1, nbnab(k_in_part)
           !nbbeg = nbkbeg + j - 1
           nd2 = bndim2(nbkbeg+j-1)
           nbbeg = nb_nd_kbeg + nd2_1st(j)
           j_in_halo = jbnab2ch(j)
           if (j_in_halo /= 0) then
              !nd2 = chalo%ndimj(j_in_halo)
              ncbeg = chalo%i_h2d(icad+j_in_halo)
              if (ncbeg /= 0) then  ! multiplication of ndim x ndim blocks
```

```
                do n2=1, nd2
                   nbaddr = nbbeg + nd3 * (n2 - 1)
                   ncaddr = ncbeg + nd1 * (n2 - 1)
                   do n1 = 1, nd1
                      naaddr = nabeg + nd3 * (n1 - 1)
                      do n3 = 1, nd3
                         a(naaddr+n3-1) = a(naaddr+n3-1) + &
                                          c(ncaddr+n1-1) * b(nbaddr+n3-1)
                      end do
                   end do
                end do
             end if
          end if
       end do
    end do
!$omp end do
  end do
!$omp end parallel
  return
end subroutine m_kern_min
```

6 Scaling Properties of Conquest After OpenMP Implementation

I tested the OpenMP-MPI hybrid implementation by running full ab initio self-consistent static DFT ground state energy calculations of the same bulk Si system with different number of atoms and basis sizes. The code was compiled with Cray compiler suite 4.0.46 and Cray LibSci 11.0.06.

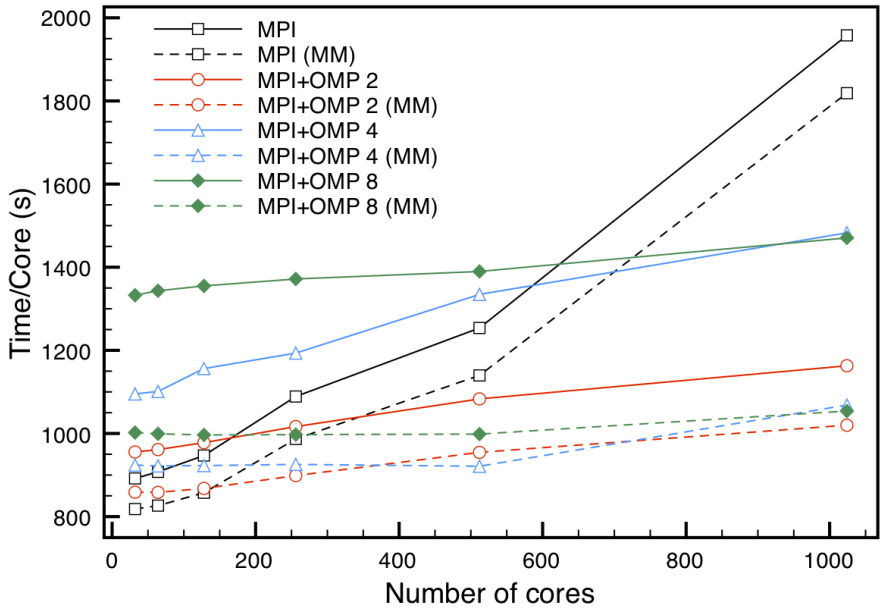

Figure 3: Weak Scaling: fixed work-load 16 atoms per core, bulk Si with SZ basis. (MM) = time spent matrix multiplication

Figure 3 shows the weak scaling property of CONQUEST using pure MPI and MPI with 2, 4 and 8 OpenMP threads per MPI process. The average amount of work allocated to each *core* was the same, at 16 atoms per core. More precisely 512 atoms were assigned to each node (consisting of 32 cores). In all of the calculations all cores were utilised. For the pure MPI calculations each node were saturated by running 32 MPI processes per node. For OpenMP-MPI hybrid calculations, the number of MPI processes per node times the number of threads per MPI process was always 32. So 2 thread calculations used 16 MPI processes per node and 2 threads per MPI process; and 4 threaded calculations used 8 MPI processes per node and 4 threads per MPI process; and so on. I made sure the MPI processes were distributed evenly across the NUMA regions using the `-S` option of `aprun`. The maximum number of threads I had tested was 8, again due to the NUMA regions. The system size was increased by increasing the number of cores in the calculation. Ideally we want the wall time to be similar for calculations of any system size. The solid lines in the figure corresponds to total wall time, and the dashed lines corresponds to the time taken in matrix-multiplication subroutines.

Due to the fact that the larger systems takes more iterations to reach the desired accuracy tolerance, the times I present in figure 3 are corrected in the same manor as that described in section 2.

As we can see from figure 3 pure MPI parallelisation is faster overall for smaller systems. This is expected, since the only multi-threaded part in the code is the matrix multiplication kernels, and therefore having less MPI processes running means other parts of the code would run slower. This is confirmed by the increase in difference in time between the solid lines of total wall time and the dashed lines of total matrix multiplication time as the number of threads per MPI processes increase. More threads of course means less total number of MPI processes given the same number of cores. The hybrid implementations however have a much better weak-scaling (more flat) than the pure MPI implementation, so as the size of system increase, we can see that the performances of hybrid implementations starts to overtake the pure MPI implementation.

We notice that weak scaling improves as the number of threads per MPI processes increase (and the number of MPI processes per node decrease). This is especially evident in matrix multiplication where OpenMP has been implemented, albeit this is off-set by other parts of code in the total wall time.

A compromise between the number of MPI processes and the number of threads to use has to be found. In the example of bulk Si, I found using 16 MPI processes per node and 2 threads per MPI process may be the best choice for systems with over 10000 atoms, while the pure MPI calculations are still faster for smaller systems.

Figure 4 shows the amount of speed up when using the same number of MPI processes but increasing number of OpenMP threads per MPI process. The number of MPI processes was fixed at 32. For single thread calculations, a single node of 32 cores were used; for 2 threads calculation, 2 nodes were used and 16 MPI processes were assigned to each node, with 8 per NUMA region; for 4 threads calculation, 4 nodes were used with 8 MPI processes assigned to each node, 4 per NUMA region; and so on. The dashed lines again shows the time or speed-up in matrix multiplication subroutines, while the solid line corresponds to the total wall time. The scaling for larger system size is better because there are more atoms per MPI process resulting in larger loops on each process for the OpenMP work-sharing constructs and more work-load for each work-sharing threads to work on.

Figure 5 shows the amount of speed up with increasing number of OpenMP threads per MPI process for different basis sizes. The number of MPI processes was again fixed at 32. The SZ basis

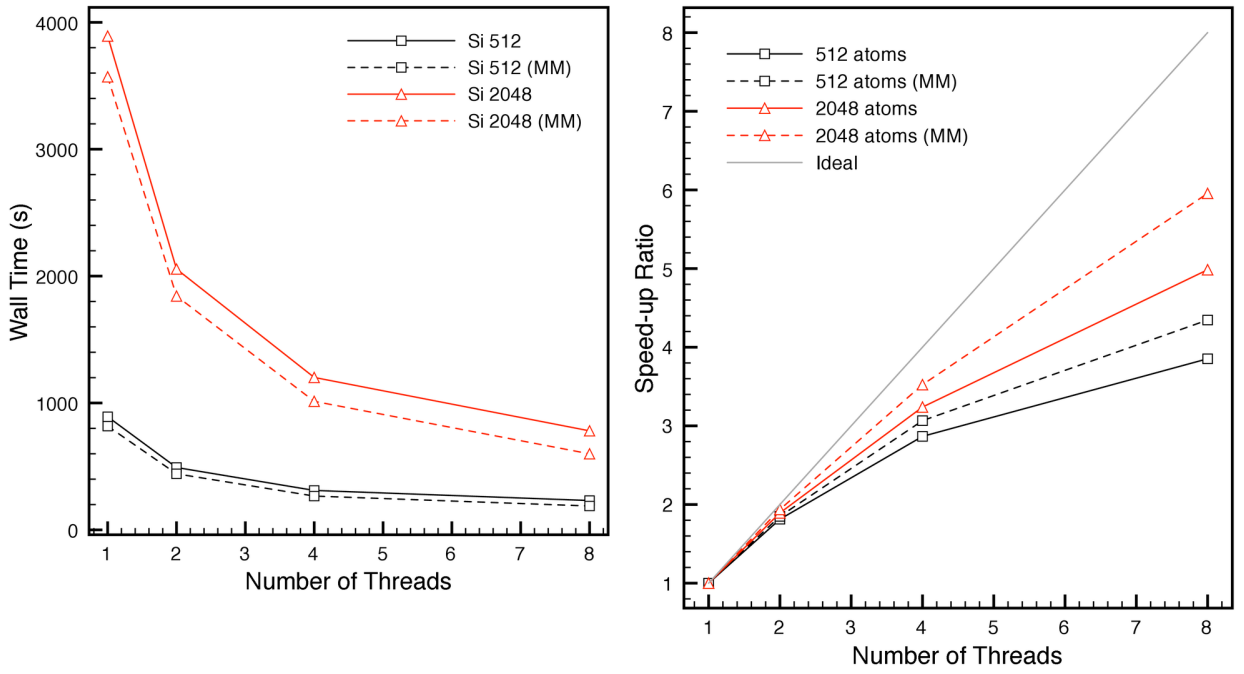

Figure 4: Strong Scaling: Comparing different system sizes

has 4 support functions per Si atom, while SZP basis has 9 per atom. The result however is puzzling. We would expect that as the basis size increases there will be more work for each support-function blocks in the matrix kernel, and hence this should mean larger chunks to work on for each of the work-sharing threads. So in theory calculations with larger basis set should scale better. The results shows the opposite, and this trend is also observed in the multi-threaded DGEMM case as show in figure 2.

I have also looked at the effect of different scheduling schemes to the performances of the calculations. First to note is the choice of chunk sizes. The upper-limits of the worked-shared loops are not known a priori, these depends on the neighbour-lists of the different matrix types calculated during a calculation. The work-load in the loop also varies and depends on the neighbour lists calculated using data inside each loop. Hence the work-load cannot be determined before the work-sharing construct. Without more knowledge before hand, for `STATIC` scheduling the most obvious choice is to evenly divide the loop into chunks of similar sizes for each thread; for `DYNAMIC` scheduling without the chunk size should be 1; and for `GUIDED` scheduling the default chunk sizes—sets the chunk sizes dynamically by dividing the remaining work to be done by the available threads. For bulk Si systems, the different choices of scheduling schemes had minimal effect to the performance of the calculations (see figure fig:omp_strong_scaling_scheduling). This is unsurprising, since the system is a perfect lattice evenly distributed across the fundamental simulation cell and this means the size and work-load of j-loops in the multiplication kernel are going to be very similar for each i and k, and hence the work-load on the work-sharing threads are evenly divided unless one choses a chunk size that is too large. The `STATIC` scheduling gives the best results by a small margin, this is probably because it requires the least amount of overhead.

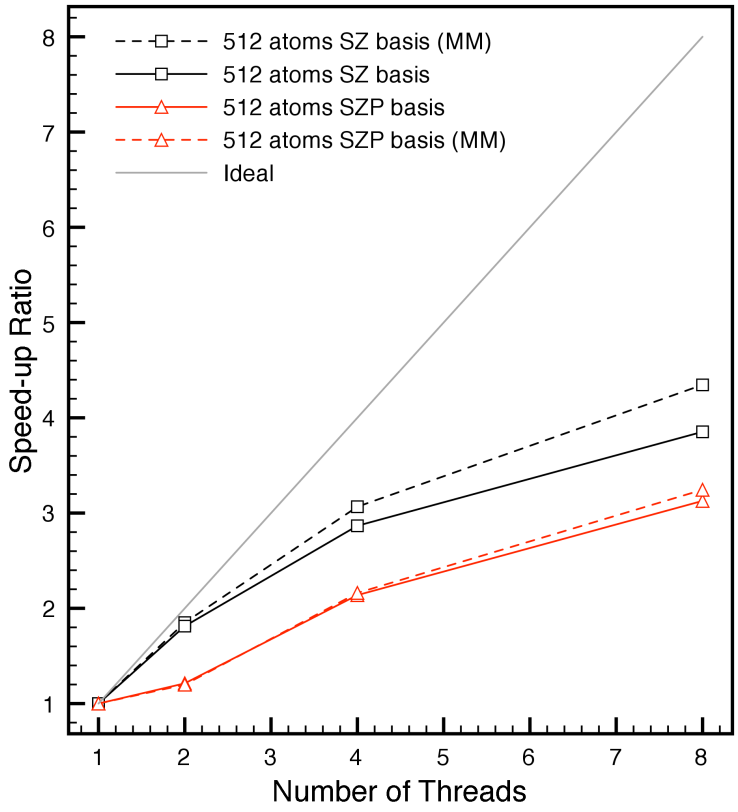

Figure 5: Strong Scaling: Comparing different basis sizes

6.1 Scaling Properties for Calculations of Ice

As mentioned above, bulk Si is a perfectly uniform system, and is thus a very good to use a test system to check the best scaling performances of the code. However in practice, a code such as CONQUEST will be used to calculate systems such as biological molecules which are more sparsely and less uniformly distributed across the fundamental simulation cell. The complication for such systems are two fold: non-uniform distribution of atoms in space and interaction involving different species each with different size of basis/support-functions. The first problem is to do with load-balancing, that is how to partition the space and allocate the partitions to MPI processes in such a way so that each process gets roughly the same amount of work-load. This is beyond the scope of this dCSE project. The second problem introduces uneven sized loops and work-load for the work-sharing OpenMP threads.

To test the performance of CONQUEST on such systems, I timed the calculations on bulk ice. Ice is a particularly good test system to use because it still had a crystalline structure, this means it is easy to divide space into partitions with roughly similar number of atoms and atom types distributed to each MPI process. This avoided the complications introduced by poor load-balancing. On the other hand the structure has two species types: Hydrogen (H) and Oxygen (O), each with different basis sizes (for SZ basis, H has 1 support function per atom and O as 4). The ice structure is sparse and introduces non-uniformity in the neighbour lists. I had chosen a basis set with a particular short cut-off range (4.8 Bohrs for H and 3.9 Bohrs for O) to exacerbate the unevenness in the neighbour

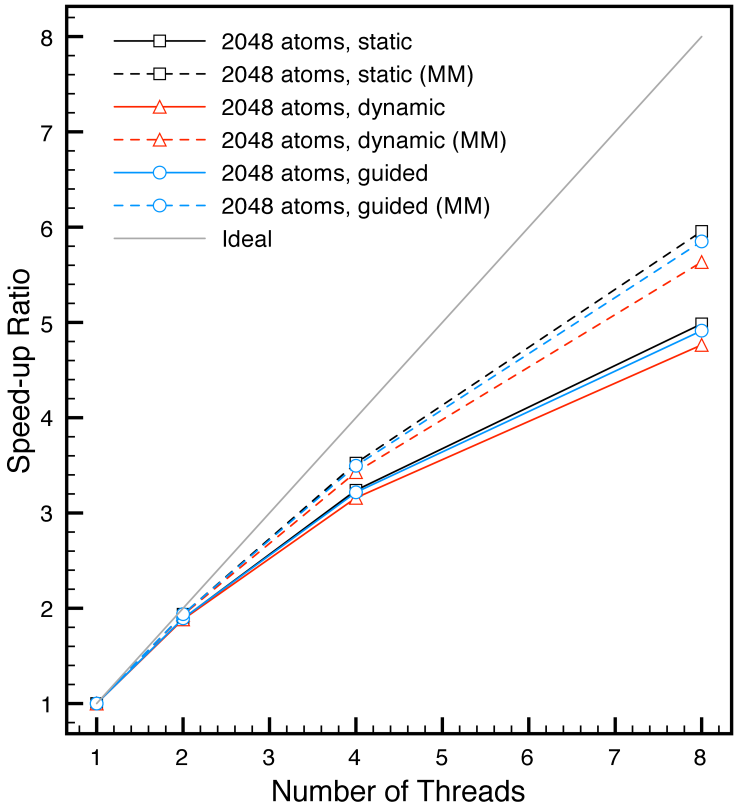

Figure 6: Strong Scaling: Comparing different scheduling schemes

lists.

Table 3: Strong scaling: Ice 1536 atoms with SZ basis, using MPI+OpenMP hybrid scheme with loop work-sharing at i-loop for both matrix multiplication kernels, 32 MPI processes

Number of Threads	Wall Time (s), STATIC	DYNAMIC	GUIDED
1	976.290	980.669	977.957
2	564.746	599.262	544.354
4	397.290	526.171	452.203
8	320.667	415.751	416.387

Table 3 shows the strong scaling result (with respect to the number of OpenMP threads) obtained from calculations on 1536 atoms bulk ice, and fixed 32 MPI processes. We can see clearly that for all OpenMP scheduling schemes, the performance improvements flattens out after number of threads increased beyond 4.

After some investigation I found that the apparent poor scaling performance for more than 4 threads was caused by the fact that the ice structure was sparse which results in some neighbour lists being very short. That is, some of the i-loops in the matrix multiplication kernel I was trying

Table 4: Top most occurring loop limits of k, i and j-loops in the matrix multiplication kernel for a typical full self-consistent DFT calculation on bulk ice (1536 atoms) with SZ basis.

k-loop limits	occurrences	i-loop limits	occurrences	j-loop limits	occurrences
48	503168	48	6149664	61	5980608
1	121984	1	4333248	62	5971648
14	84384	2	2764416	70	5904832
18	77280	3	2579296	67	5836576
16	66048	5	2019424	65	5784736
24	63360	4	1802752	59	5736160
17	55680	6	1688608	293	2891168
20	55680	7	1502624	290	2884320
31	52704	9	1398720	291	2859488
26	50880	10	1266496	294	2848448
28	49440	8	1217792	287	2828384
34	48576	13	1161312	277	2755232
36	45024	16	1136352	12	88288
2	44640	11	1067296	15	39680

to parallelise simply were too small and we ran out of neighbours to share between the treads (see table 4).

6.1.1 Strong Scaling of Work-Sharing on Different Loops in Multiplication Kernel

As mentioned in section 5 for `m_kern_max` we can work-share either the i-loop or j-loop and for `m_kern_min` either the i-loop or k-loop. Table 4 clearly shows the reason for the poor strong scaling of the hybrid OpenMP implementation based on work sharing on the i-loop for both kernels: the top most occurring limits in the calculation for the i-loops were mostly less than 8. Therefore the first thing we should try to improve the strong scaling properties of the OpenMP implementation is to try to work-share the different loops in the two matrix multiplication kernels.

Table 5: Strong scaling: Wall times (s) of calculations for Ice 1536 atoms with SZ basis, OpenMP workshare at i, j or k-loops in multiplication kernels, "max: i" means work sharing i-loop in `m_kern_max`. All calculations used `STATIC` scheduling with default chunks, and 32 MPI processes.

Number of Threads	max: i, min: i	max: i, min: k	max: j, min: i
1	976.290	961.908	1095.035
2	564.746	547.537	616.610
4	397.290	369.831	437.063
8	320.667	272.060	415.787

Tabel 5 shows the total wall time for the self-consistent DFT calculation of bulk ice with 1536 atoms in the fundamental simulation cell, with SZ basis, and applying `!$omp do` work-sharing to the

different loops. The number of MPI processes was fixed at 32. Since the matrix kernels `m_kern_max` and `m_kern_min` are independent, we only need to compare three cases: work-sharing i-loops in both kernels, work-sharing i-loop in `m_kern_max` and k-loop in `m_kern_min`, and work-sharing j-loop in `m_kern_max` and i-loop in `m_kern_min`. The results indicate that for ice, work-sharing the i-loop in `m_kern_max` and k-loop in `m_kern_min` gives the best performances. This is perhaps unsurprising in the light of tabel 4 and the fact that in general work-sharing the outer most loops possible will give the least amount of overhead.

6.1.2 Weak Scaling

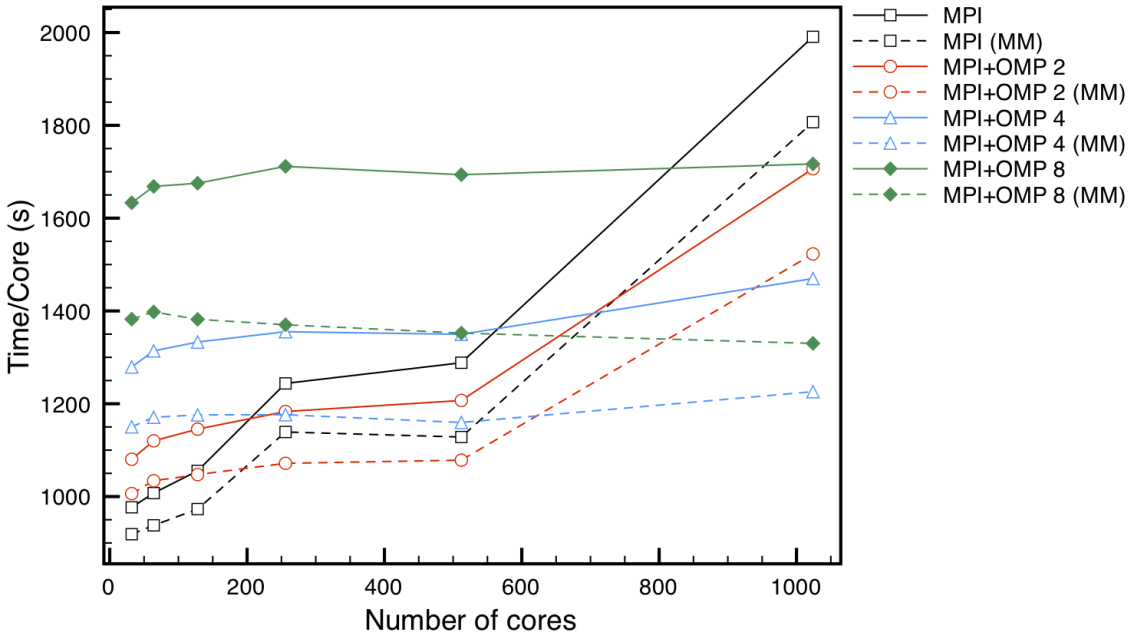

Figure 7: Weak scaling: fixed work-load with 48 atoms per core, bulk Ice with SZ basis.

Figure 7 shows the weak scaling of calculations on bulk ice using pure MPI, and MPI with 2, 4 and 8 OpenMP threads per MPI process. 1536 atoms were assigned to each node on HECToR Phase 3, and in all test cases all cores were used: pure MPI calculation would have 32 MPI processes distributed per core, and for OpenMP-MPI hybrid calculations, the product of number of MPI process with the number of OpenMP threads was always 32. The version of hybrid implementation used in the test had work-sharing on the i-loop in `m_kern_max` and k-loop in `m_kern_min`. As we can see from the figure, for smaller ice systems, the pure MPI calculation is faster, and the more MPI processes per node the faster the calculation. However the OpenMP-MPI implementation has better scaling when the number of threads increase, and hence becomes faster when system becomes large. The key result here is that just as with the bulk Si case, the OpenMP-MPI hybrid implementation significantly improves the weak-scaling of CONQUEST for cases where the neighbours are more sparse.

7 Further Improvements

7.1 Attempt at using OpenMP Tasks

Being aware of the fact that in some cases, such as the case with ice in the previous example, there may be occasions that there are simply too few atoms in a particular neighbour list to give a large enough loop for OpenMP to work-share efficiently, I considered changing the OpenMP implementation to use concept of tasks introduced since OpenMP 3.0.

By using OpenMP tasks, we can effectively combine work-sharing of i and j-loops in `m_kern_max` and k, i-loops in `m_kern_min`, so that in theory all threads should have something to do during the execution of the multiplication kernels.

My OpenMP task implementation for `m_kern_max` is given as follows

```
subroutine m_kern_max(k_off, kpart, ib_nd_acc, ibaddr, nbnab,       &
                      ibpart, ibseq, bndim2, a, b, c, ahalo, chalo, &
                      at, mx_absb, mx_part, mx_iprim, lena, lenb,   &
                      lenc, debug)

  use datatypes
  use matrix_module
  use basic_types,    only: primary_set
  use primary_module, only: bundle
  use omp_lib

  implicit none

  ! Passed variables
  type(matrix_halo)  :: ahalo, chalo
  type(matrix_trans) :: at
  integer            :: mx_absb, mx_part, mx_iprim, lena, lenb, lenc
  integer            :: kpart, k_off
  real(double)       :: a(lena)
  real(double)       :: b(lenb)
  real(double)       :: c(lenc)
  integer, optional  :: debug
  ! Remote indices
  integer(integ) :: ib_nd_acc(mx_part)
  integer(integ) :: ibaddr(mx_part)
  integer(integ) :: nbnab(mx_part)
  integer(integ) :: ibpart(mx_part*mx_absb)
  integer(integ) :: ibseq(mx_part*mx_absb)
  integer(integ) :: bndim2(mx_part*mx_absb)
  ! Local variables
  integer :: nbkbeg, k, k_in_part, k_in_halo, j, jpart, jseq
  integer :: i, nabeg, i_in_prim, icad, nbbeg, j_in_halo, ncbeg
  integer :: n1, n2, n3, nb_nd_kbeg
  integer :: nd1, nd2, nd3
  integer :: naaddr, nbaddr, ncaddr
  ! The OpenMP firstprivate clause for some reason crashes (at least
  ! for gcc) whenever there is automatic arrays in its
  ! list. Everthing works fine with allocatable arrays. The
```

```
  ! allocation of allocatable arrays may be slightly slower than
  ! automatic arrays, but it is the only option at the moment.
  integer, dimension(:), allocatable :: nd1_1st, nd2_1st, jbnab2ch

  allocate(nd1_1st(at%mx_halo), nd2_1st(mx_absb), jbnab2ch(mx_absb))

!$omp parallel default(none)                                          &
!$omp          shared(kpart, ibaddr, ib_nd_acc, nbnab, ibpart, ibseq,    &
!$omp                 k_off, bndim2, mx_absb, mx_part, at, ahalo, chalo, &
!$omp                 a, b, c)                                          &
!$omp          private(i, j, k, j_in_halo, k_in_halo, k_in_part, nbkbeg, &
!$omp                  nb_nd_kbeg, nd1, nd2, nd3, jpart, jseq, jbnab2ch, &
!$omp                  nabeg, nbbeg, ncbeg, i_in_prim, icad, naaddr,     &
!$omp                  nbaddr, ncaddr, n1, n2, n3, nd1_1st, nd2_1st)
    ! run on single thread for generating tasks
!$omp single
    ! task for generating computation tasks
!$omp task untied                                                    &
!$omp      default(none)                                             &
!$omp      shared(kpart, ibaddr, ib_nd_acc, nbnab, ibpart, ibseq,    &
!$omp             k_off, bndim2, mx_absb, mx_part, at, ahalo, chalo, &
!$omp             a, b, c)                                          &
!$omp      private(i, j, k, j_in_halo, k_in_halo, k_in_part, nbkbeg, &
!$omp              nb_nd_kbeg, nd1, nd2, nd3, jpart, jseq, jbnab2ch, &
!$omp              nabeg, nbbeg, ncbeg, i_in_prim, icad, naaddr,     &
!$omp              nbaddr, ncaddr, n1, n2, n3, nd1_1st, nd2_1st)
  ! Loop over atoms k in current A-halo partn
  do k = 1, ahalo%nh_part(kpart)
     k_in_halo = ahalo%j_beg(kpart) + k - 1
     k_in_part = ahalo%j_seq(k_in_halo)
     nbkbeg = ibaddr(k_in_part)
     nb_nd_kbeg = ib_nd_acc(k_in_part)
     nd3 = ahalo%ndimj(k_in_halo)
     ! for OpenMP sub-array indexing
     nd1_1st(1) = 0
     do i = 2, at%n_hnab(k_in_halo)
       i_in_prim = at%i_prim(at%i_beg(k_in_halo)+i-2)
       nd1_1st(i) = nd1_1st(i-1) + nd3 * ahalo%ndimi(i_in_prim)
     end do
     nd2_1st(1) = 0
     do j = 2, nbnab(k_in_part)
        nd2_1st(j) = nd2_1st(j-1) + nd3 * bndim2(nbkbeg+j-2)
     end do
     ! transcription of j from partition to C-halo labelling
     do j = 1, nbnab(k_in_part)
        jpart = ibpart(nbkbeg+j-1) + k_off
        jseq = ibseq(nbkbeg+j-1)
        jbnab2ch(j) = chalo%i_halo(chalo%i_hbeg(jpart)+jseq-1)
     end do
     ! Loop over primary-set A-neighbours of k
     do i = 1, at%n_hnab(k_in_halo)
```

```
        i_in_prim = at%i_prim(at%i_beg(k_in_halo)+i-1)
        nd1 = ahalo%ndimi(i_in_prim)
        icad = (i_in_prim-1) * chalo%ni_in_halo
        nabeg = at%i_nd_beg(k_in_halo) + nd1_1st(i)
        ! Loop over B-neighbours of atom k
        do j = 1, nbnab(k_in_part)
           nd2 = bndim2(nbkbeg+j-1)
           nbbeg = nb_nd_kbeg + nd2_1st(j)
           j_in_halo = jbnab2ch(j)
!$omp task if(j_in_halo /= 0)                                          &
!$omp      default(none)                                               &
!$omp      shared(nbnab, bndim2, chalo, a, b, c)                       &
!$omp      firstprivate(nd1, nd2, nd3, nbkbeg, nb_nd_kbeg, nd2_1st,  &
!$omp                   jbnab2ch, nabeg, icad, k_in_part, j_in_halo, &
!$omp                   nbbeg)                                       &
!$omp      private(ncbeg, n1, n2, n3, naaddr, nbaddr, ncaddr)
           if (j_in_halo /= 0) then
              ncbeg = chalo%i_h2d(icad+j_in_halo)
              if (ncbeg /= 0 ) then ! multiplication of ndim x ndim blocks
                 do n2 = 1, nd2
                    nbaddr = nbbeg + nd3 * (n2 - 1)
                    ncaddr = ncbeg + nd1 * (n2 - 1)
                    do n1 = 1, nd1
                       naaddr = nabeg + nd3 * (n1 - 1)
                       do n3 = 1, nd3
                          c(ncaddr+n1-1) = c(ncaddr+n1-1) + &
                                           a(naaddr+n3-1) * b(nbaddr+n3-1)
                       end do
                    end do
                 end do
              end if
           end if ! End of if(j_in_halo.ne.0)
!$omp end task
        end do ! End of j = 1, nbnab
     end do ! End of i = 1, at%n_hnab
!$omp taskwait
  end do ! End of k = 1, nahpart
  ! need to wait for all the computation tasks to finish before the
  ! generating task is to end.
!$omp end task
!$omp end single
!$omp end parallel

  deallocate(nd1_1st, nd2_1st, jbnab2ch)

  return
end subroutine m_kern_max
```

The parallel region is generated as soon as the code enters the subroutine, a single thread is then assigned the job of generating an untied task for generating the computation tasks. For every iteration in i and j loop, a computation task is generated and put into the task pool for any idle

threads to pick-up. A `taskwait` must be present at the end of every iteration of k-loop to avoid race-conditions, since all matrix elements of $\mathbf{A}$ (`a`) associated with i and j must be calculated before we start to accumulate the new products onto them. The reason for using an outer task region to generate the computation tasks is because if the computation task generating thread at some stage is assigned a long computation task it-self, then some other idle thread should be able to take over the task generating job, so that the task-pool is never empty with idle threads waiting.

In the original `!$omp do` implementation, the arrays for holding the sub-matrix indices are allocated as automatic arrays on the stack. However it seemed that if used as a `firstprivate` variable for an explicit task-region, these arrays must either be static or `allocatable`, and automatic arrays caused the code to crash. I therefore changed all the offending automatic arrays to allocatable arrays, and this seemed to have solved the problem. I did not found specific mention of the FORTRAN automatic arrays in the OpenMP 3.1 specification documents.

For the `m_kern_min` case the implementation is similar, only that in this case the computation task region *include* the entire j-loops:

```
subroutine m_kern_min(k_off, kpart, ib_nd_acc, ibaddr, nbnab,       &
                      ibpart, ibseq, bndim2, a, b, c, ahalo, chalo, &
                      at, mx_absb, mx_part, mx_iprim, lena, lenb,   &
                      lenc)
  use datatypes
  use matrix_module
  use basic_types,    only: primary_set
  use primary_module, only: bundle

  implicit none

  ! Passed variables
  type(matrix_halo)  :: ahalo, chalo
  type(matrix_trans) :: at
  integer            :: mx_absb, mx_part, mx_iprim, lena, lenb, lenc
  integer            :: kpart, k_off
  ! Remember that a is a local transpose
  real(double)       :: a(lena)
  real(double)       :: b(lenb)
  real(double)       :: c(lenc)
  ! dimension declarations
  integer :: ibaddr(mx_part)
  integer :: ib_nd_acc(mx_part)
  integer :: nbnab(mx_part)
  integer :: ibpart(mx_part*mx_absb)
  integer :: ibseq(mx_part*mx_absb)
  integer :: bndim2(mx_part*mx_absb)
  ! Local variables
  integer :: k, k_in_part, k_in_halo, nbkbeg, j, jpart, jseq
  integer :: i, nabeg, i_in_prim, icad, nbbeg, j_in_halo, ncbeg
  integer :: n1, n2, n3, nb_nd_kbeg
  integer :: nd1, nd2, nd3
  integer :: naaddr, nbaddr, ncaddr
  ! The OpenMP firstprivate clause for some reason crashes (at least
  ! for gcc) whenever there is automatic arrays in its
```

```
  ! list. Everthing works fine with allocatable arrays. The
  ! allocation of allocatable arrays may be slightly slower than
  ! automatic arrays, but it is the only option at the moment.
  integer, dimension(:), allocatable :: nd1_1st, nd2_1st, jbnab2ch

  allocate(nd1_1st(at%mx_halo), nd2_1st(mx_absb), jbnab2ch(mx_absb))

!$omp parallel default(none)                                          &
!$omp          shared(kpart, ibaddr, ib_nd_acc, nbnab, ibpart, ibseq,    &
!$omp                 k_off, bndim2, mx_absb, mx_part, at, ahalo, chalo, &
!$omp                 a, b, c)                                          &
!$omp          private(i, j, k, j_in_halo, k_in_halo, k_in_part, nbkbeg, &
!$omp                  nb_nd_kbeg, nd1, nd2, nd3, jpart, jseq, jbnab2ch, &
!$omp                  nabeg, nbbeg, ncbeg, i_in_prim, icad, naaddr,     &
!$omp                  nbaddr, ncaddr, n1, n2, n3, nd1_1st, nd2_1st)
  ! run on single thread for generating tasks
!$omp single
  ! task for generating computation tasks
!$omp task untied                                                   &
!$omp      default(none)                                            &
!$omp      shared(kpart, ibaddr, ib_nd_acc, nbnab, ibpart, ibseq,    &
!$omp             k_off, bndim2, mx_absb, mx_part, at, ahalo, chalo, &
!$omp             a, b, c)                                          &
!$omp      private(i, j, k, j_in_halo, k_in_halo, k_in_part, nbkbeg, &
!$omp              nb_nd_kbeg, nd1, nd2, nd3, jpart, jseq, jbnab2ch, &
!$omp              nabeg, nbbeg, ncbeg, i_in_prim, icad, naaddr,     &
!$omp              nbaddr, ncaddr, n1, n2, n3, nd1_1st, nd2_1st)
  ! Loop over atoms k in current A-halo partn
  do k = 1, ahalo%nh_part(kpart)
     k_in_halo = ahalo%j_beg(kpart) + k - 1
     k_in_part = ahalo%j_seq(k_in_halo)
     nbkbeg = ibaddr(k_in_part)
     nb_nd_kbeg = ib_nd_acc(k_in_part)
     nd3 = ahalo%ndimj(k_in_halo)
     ! for OpenMP sub-array indexing
     nd1_1st(1) = 0
     do i = 2, at%n_hnab(k_in_halo)
        i_in_prim = at%i_prim(at%i_beg(k_in_halo)+i-2)
        nd1_1st(i) = nd1_1st(i-1) + nd3 * ahalo%ndimi(i_in_prim)
     end do
     nd2_1st(1) = 0
     do j = 2, nbnab(k_in_part)
        nd2_1st(j) = nd2_1st(j-1) + nd3 * bndim2(nbkbeg+j-2)
     end do
     ! transcription of j from partition to C-halo labelling
     do j = 1, nbnab(k_in_part)
        jpart = ibpart(nbkbeg+j-1) + k_off
        jseq = ibseq(nbkbeg+j-1)
        jbnab2ch(j) = chalo%i_halo(chalo%i_hbeg(jpart)+jseq-1)
     end do
     ! Loop over primary-set A-neighbours of k
```

```
     do i = 1, at%n_hnab(k_in_halo)
        i_in_prim = at%i_prim(at%i_beg(k_in_halo)+i-1)
        nd1 = ahalo%ndimi(i_in_prim)
        icad = (i_in_prim-1) * chalo%ni_in_halo
        nabeg = at%i_nd_beg(k_in_halo) + nd1_1st(i)
!$omp task default(none)                                          &
!$omp      shared(nbnab, bndim2, chalo, a, b, c)                  &
!$omp      firstprivate(nd1, nd3, nbkbeg, nb_nd_kbeg, nd2_1st,  &
!$omp                   jbnab2ch, nabeg, icad, k_in_part)        &
!$omp      private(j, nd2, nbbeg, j_in_halo, ncbeg, n1, n2, n3, &
!$omp              naaddr, nbaddr, ncaddr)
        ! Loop over B-neighbours of atom k
        do j = 1, nbnab(k_in_part)
           nd2 = bndim2(nbkbeg+j-1)
           nbbeg = nb_nd_kbeg + nd2_1st(j)
           j_in_halo = jbnab2ch(j)
           if (j_in_halo /= 0) then
              ncbeg = chalo%i_h2d(icad+j_in_halo)
              if (ncbeg /= 0) then  ! multiplication of ndim x ndim blocks
                 do n2=1, nd2
                    nbaddr = nbbeg + nd3 * (n2 - 1)
                    ncaddr = ncbeg + nd1 * (n2 - 1)
                    do n1 = 1, nd1
                       naaddr = nabeg + nd3 * (n1 - 1)
                       do n3 = 1, nd3
                          a(naaddr+n3-1) = a(naaddr+n3-1) + &
                                           c(ncaddr+n1-1) * b(nbaddr+n3-1)
                       end do
                    end do
                 end do
              end if
           end if
        end do
!$omp end task
     end do
  end do
  ! need to wait for all the computation tasks to finish before the
  ! generating task is to end.
!$omp taskwait
!$omp end task
!$omp end single
!$omp end parallel

  deallocate(nd1_1st, nd2_1st, jbnab2ch)

  return
end subroutine m_kern_min
```

The new implementations were first tested on a Sun Microsystems work-station with a single AMD Opteron quad-core processor before being used on HECToR. While the new implementation produced similar or better performances for both bulk Si and ice compared to the original

`!$omp do` implementation on the work-station using a single MPI process with 2 or 4 threads, on HECToR the task implementations performed significantly worse (using the same strong scaling set-up as for the calculations described in previous sections in this report). I tested calculations with `PSC_OMP_AFFINITY=TRUE` or `PSC_OMP_AFFINITY=FALSE` but in all cases the calculations went beyond 4 hours limit, which were significantly longer than the `!$omp do` implementations. Therefore I concluded that the new implementations using OpenMP tasks is not successful on HECToR. Never-the-less I cannot rule out the possibility that the implementation would work on other HPC systems.

7.2 Implementation of OpenMP+DGEMM

As one might notice from section 4 that the single threaded `DGEMM` version of the matrix multiplication kernels is actually faster than the vanilla version. For 512 atoms bulk Si calculation with SZ basis, the vanilla pure MPI CONQUEST finished after 855.039 seconds (see section 2), while the `DGEMM` version finished in 796.822 seconds (see section 4.1). I therefore looked at the possibility of implementing an OpenMP multi-threaded version of the matrix multiplication kernels using `DGEMM`.

The original implementation of the `DGEMM` version of multiplication kernels had memory allocation and deallocation calls inside loops, this was done because the array sizes has to be determined during run-time and changes for each iteration. I improved this by first calculating the upper limit of each array, and allocating the required arrays at the start of the subroutines. This avoided allocation and deallocation inside loops at the expense of slightly larger memory foot-print. This increase in memory usage is only temporary however, since the arrays are deallocated at the end of the subroutine anyway.

The source code is given as follows:

For `m_kern_max`:

```
subroutine m_kern_max(k_off, kpart, ib_nd_acc, ibaddr, nbnab,       &
                      ibpart, ibseq, bndim2, a, b, c, ahalo, chalo, &
                      at, mx_absb, mx_part, mx_iprim, lena, lenb,   &
                      lenc, debug)
  use datatypes
  use matrix_module
  use basic_types,    only: primary_set
  use primary_module, only: bundle
  use numbers,        only: zero, one

  implicit none

  ! Passed variables
  type(matrix_halo)  :: ahalo, chalo
  type(matrix_trans) :: at
  integer      :: mx_absb, mx_part, mx_iprim, lena, lenb, lenc
  integer      :: kpart, k_off
  real(double) :: a(lena)
  real(double) :: b(lenb)
  real(double) :: c(lenc)
  integer, optional :: debug
  ! Remote indices
  integer(integ) :: ib_nd_acc(mx_part)
```

```
  integer(integ) :: ibaddr(mx_part)
  integer(integ) :: nbnab(mx_part)
  integer(integ) :: ibpart(mx_part*mx_absb)
  integer(integ) :: ibseq(mx_part*mx_absb)
  integer(integ) :: bndim2(mx_part*mx_absb)
  ! Local variables
  integer :: jbnab2ch(mx_absb)  ! Automatic array
  integer :: nbkbeg, k, k_in_part, k_in_halo, j, jpart, jseq
  integer :: i, nabeg, i_in_prim, icad, nbbeg, j_in_halo, ncbeg
  integer :: n1, n2, n3, nb_nd_kbeg
  integer :: nd1, nd2, nd3
  integer :: naaddr, nbaddr, ncaddr
  real(double), allocatable, dimension(:,:) :: tempb, tempa, tempc
  integer :: sofar, maxnd1, maxnd2, maxnd3, maxlen
  external :: dgemm
  ! OpenMP required indexing variables
  integer :: nd1_1st(at%mx_halo), nd2_1st(mx_absb)

!$omp parallel default(none)                                          &
!$omp          shared(kpart, ibaddr, ib_nd_acc, nbnab, ibpart, ibseq,    &
!$omp                 k_off, bndim2, mx_absb, mx_part, at, ahalo, chalo, &
!$omp                 a, b, c)                                          &
!$omp          private(i, j, k, j_in_halo, k_in_halo, k_in_part, nbkbeg, &
!$omp                  nb_nd_kbeg, nd1, nd2, nd3, jpart, jseq, jbnab2ch, &
!$omp                  nabeg, nbbeg, ncbeg, i_in_prim, icad, naaddr,     &
!$omp                  nbaddr, ncaddr, n1, n2, n3, nd1_1st, nd2_1st,     &
!$omp                  tempa, tempb, tempc, maxnd1, maxnd2, maxnd3,      &
!$omp                  maxlen, sofar)
  maxnd1 = maxval(ahalo%ndimi)
  maxnd2 = maxval(bndim2)
  maxnd3 = maxval(ahalo%ndimj)
  maxlen = maxval(nbnab) * maxnd2
  allocate(tempa(maxnd1,maxnd3), tempc(maxnd1,maxlen), tempb(maxnd3,maxlen))
  tempa = zero
  tempb = zero
  tempc = zero
  ! Loop over atoms k in current A-halo partn
  do k = 1, ahalo%nh_part(kpart)
     k_in_halo = ahalo%j_beg(kpart) + k - 1
     k_in_part = ahalo%j_seq(k_in_halo)
     nbkbeg = ibaddr(k_in_part)
     nb_nd_kbeg = ib_nd_acc(k_in_part)
     nd3 = ahalo%ndimj(k_in_halo)
     ! for OpenMP sub-array indexing
     nd1_1st(1) = 0
     do i = 2, at%n_hnab(k_in_halo)
       i_in_prim = at%i_prim(at%i_beg(k_in_halo)+i-2)
       nd1_1st(i) = nd1_1st(i-1) + nd3 * ahalo%ndimi(i_in_prim)
     end do
     nd2_1st(1) = 0
     do j = 2, nbnab(k_in_part)
```

```
        nd2_1st(j) = nd2_1st(j-1) + nd3 * bndim2(nbkbeg+j-2)
     end do
     ! transcription of j from partition to C-halo labelling
     do j = 1, nbnab(k_in_part)
        jpart = ibpart(nbkbeg+j-1) + k_off
        jseq = ibseq(nbkbeg+j-1)
        jbnab2ch(j) = chalo%i_halo(chalo%i_hbeg(jpart)+jseq-1)
     end do
!$omp do schedule(runtime)
     ! Loop over primary-set A-neighbours of k
     do i = 1, at%n_hnab(k_in_halo)
        i_in_prim = at%i_prim(at%i_beg(k_in_halo)+i-1)
        nd1 = ahalo%ndimi(i_in_prim)
        nabeg = at%i_nd_beg(k_in_halo) + nd1_1st(i)
        do n1 = 1, nd1
           naaddr = nabeg + nd3 * (n1 - 1)
           do n3 = 1, nd3
              tempa(n1,n3) = a(naaddr+n3-1)
           end do
        end do
        icad = (i_in_prim - 1) * chalo%ni_in_halo
        sofar = 0
        ! Loop over B-neighbours of atom k
        do j = 1, nbnab(k_in_part)
           nd2 = bndim2(nbkbeg+j-1)
           nbbeg = nb_nd_kbeg + nd2_1st(j)
           j_in_halo = jbnab2ch(j)
           if (j_in_halo /= 0) then
              ncbeg = chalo%i_h2d(icad+j_in_halo)
              if (ncbeg /= 0) then  ! multiplication of ndim x ndim blocks
                 do n2 = 1, nd2
                    nbaddr = nbbeg + nd3 * (n2 - 1)
                    do n3 = 1, nd3
                       tempb(n3,sofar+n2) = b(nbaddr+n3-1)
                    end do
                 end do
                 sofar = sofar + nd2
              end if
           end if ! End of if (j_in_halo /= 0)
        end do ! End of j = 1, nbnab
        if (sofar > 0) then
           call dgemm('n', 'n', nd1, sofar, nd3, one, tempa, &
                      maxnd1, tempb, maxnd3, zero, tempc, maxnd1)
        end if
        sofar = 0
        ! Loop over B-neighbours of atom k
        do j = 1, nbnab(k_in_part)
           nd2 = bndim2(nbkbeg+j-1)
           j_in_halo = jbnab2ch(j)
           if (j_in_halo /= 0) then
              ncbeg = chalo%i_h2d(icad+j_in_halo)
```

```
              if (ncbeg /= 0) then ! multiplication of ndim x ndim blocks
                 do n2 = 1, nd2
                    ncaddr = ncbeg + nd1 * (n2 - 1)
                    do n1 = 1, nd1
                       c(ncaddr+n1-1) = c(ncaddr+n1-1) + tempc(n1,sofar+n2)
                    end do
                 end do
                 sofar = sofar + nd2
              end if
           end if
        end do ! end of j = 1, nbnab(k_in_part)
     end do ! end of i = 1, at%n_hnab
!$omp end do
  end do ! end of k = 1, nahpart
  deallocate(tempa, tempb, tempc)
!$omp end parallel
  return
end subroutine m_kern_max
```

And for `m_kern_min`:

```
subroutine m_kern_min(k_off, kpart, ib_nd_acc, ibaddr, nbnab,       &
                      ibpart, ibseq, bndim2, a, b, c, ahalo, chalo, &
                      at, mx_absb, mx_part, mx_iprim, lena, lenb,   &
                      lenc)
  use datatypes
  use matrix_module
  use basic_types,    only: primary_set
  use primary_module, only: bundle
  use numbers,        only: one, zero

  implicit none

  ! Passed variables
  type(matrix_halo)  :: ahalo, chalo
  type(matrix_trans) :: at
  integer :: mx_absb, mx_part, mx_iprim, lena, lenb, lenc
  integer :: kpart, k_off
  ! Remember that a is a local transpose
  real(double) :: a(lena)
  real(double) :: b(lenb)
  real(double) :: c(lenc)
  ! dimension declarations
  integer :: ibaddr(mx_part)
  integer :: ib_nd_acc(mx_part)
  integer :: nbnab(mx_part)
  integer :: ibpart(mx_part*mx_absb)
  integer :: ibseq(mx_part*mx_absb)
  integer :: bndim2(mx_part*mx_absb)
  ! Local variables
  integer :: jbnab2ch(mx_absb)
```

```
  integer :: k, k_in_part, k_in_halo, nbkbeg, j, jpart, jseq
  integer :: i, nabeg, i_in_prim, icad, nbbeg, j_in_halo, ncbeg
  integer :: n1, n2, n3, nb_nd_kbeg
  integer :: nd1, nd2, nd3
  integer :: naaddr, nbaddr, ncaddr
  integer :: sofar, maxnd1, maxnd2, maxnd3, maxlen
  real(double), allocatable, dimension(:,:) :: tempb, tempc
  external :: dgemm
  ! OpenMP required indexing variables
  integer :: nd1_1st(at%mx_halo), nd2_1st(mx_absb)

!$omp parallel default(none)                                          &
!$omp          shared(kpart, ibaddr, ib_nd_acc, nbnab, ibpart, ibseq,    &
!$omp                 k_off, bndim2, mx_absb, mx_part, at, ahalo, chalo, &
!$omp                 a, b, c)                                          &
!$omp          private(i, j, k, j_in_halo, k_in_halo, k_in_part, nbkbeg, &
!$omp                  nb_nd_kbeg, nd1, nd2, nd3, jpart, jseq, jbnab2ch, &
!$omp                  nabeg, nbbeg, ncbeg, i_in_prim, icad, naaddr,     &
!$omp                  nbaddr, ncaddr, n1, n2, n3, nd1_1st, nd2_1st,     &
!$omp                  tempb, tempc, maxnd1, maxnd2, maxnd3, maxlen,     &
!$omp                  sofar)
  maxnd1 = maxval(ahalo%ndimi)
  maxnd2 = maxval(bndim2)
  maxnd3 = maxval(ahalo%ndimj)
  maxlen = maxval(nbnab) * maxnd2
  allocate(tempb(maxnd3,maxlen), tempc(maxlen,maxnd1))
  tempb = zero
  tempc = zero
  ! Loop over atoms k in current A-halo partn
  do k = 1, ahalo%nh_part(kpart)
     k_in_halo = ahalo%j_beg(kpart) + k - 1
     k_in_part = ahalo%j_seq(k_in_halo)
     nbkbeg = ibaddr(k_in_part)
     nb_nd_kbeg = ib_nd_acc(k_in_part)
     nd3 = ahalo%ndimj(k_in_halo)
     ! for OpenMP sub-array indexing
     nd1_1st(1) = 0
     do i = 2, at%n_hnab(k_in_halo)
       i_in_prim = at%i_prim(at%i_beg(k_in_halo)+i-2)
       nd1_1st(i) = nd1_1st(i-1) + nd3 * ahalo%ndimi(i_in_prim)
     end do
     nd2_1st(1) = 0
     do j = 2, nbnab(k_in_part)
        nd2_1st(j) = nd2_1st(j-1) + nd3 * bndim2(nbkbeg+j-2)
     end do
     ! transcription of j from partition to C-halo labelling
     do j = 1, nbnab(k_in_part)
        jpart = ibpart(nbkbeg+j-1) + k_off
        jseq = ibseq(nbkbeg+j-1)
        jbnab2ch(j) = chalo%i_halo(chalo%i_hbeg(jpart)+jseq-1)
     end do
```

```
!$omp do schedule(runtime)
     ! Loop over primary-set A-neighbours of k
     do i = 1, at%n_hnab(k_in_halo)
        i_in_prim = at%i_prim(at%i_beg(k_in_halo)+i-1)
        nd1 = ahalo%ndimi(i_in_prim)
        nabeg = at%i_nd_beg(k_in_halo) + nd1_1st(i)
        icad = (i_in_prim-1) * chalo%ni_in_halo
        sofar = 0
        ! Loop over B-neighbours of atom k
        do j = 1, nbnab(k_in_part)
           nd2 = bndim2(nbkbeg+j-1)
           nbbeg = nb_nd_kbeg + nd2_1st(j)
           j_in_halo = jbnab2ch(j)
           if (j_in_halo /= 0) then
              ncbeg = chalo%i_h2d(icad+j_in_halo)
              if (ncbeg /= 0) then ! multiplication of ndim x ndim blocks
                 do n2 = 1, nd2
                    nbaddr = nbbeg + nd3 * (n2 - 1)
                    ncaddr = ncbeg + nd1 * (n2 - 1)
                    do n3 = 1, nd3
                       tempb(n3,sofar+n2) = b(nbaddr+n3-1)
                    end do
                    do n1 = 1, nd1
                       tempc(sofar+n2,n1) = c(ncaddr+n1-1)
                    end do
                 end do
                 sofar = sofar + nd2
              end if
           end if
        end do
        if (sofar > 0) then
           call dgemm('n', 'n', nd3, nd1, sofar, one, tempb, &
                      maxnd3, tempc, maxlen, one, a(nabeg:), nd3)
        end if
     end do
!$omp end do
  end do
  deallocate(tempb, tempc)
!$omp end parallel
  return
end subroutine m_kern_min
```

The implementation was tested on HECToR using bulk Si system with 2048 atoms in the fundamental simulation cell. The code crashed when using Cray compiler (version 4.0.46) and linking with single-threaded version of the Cray `LibSci` (version 11.0.06). When linking with multi-threaded version of `LibSci` the code did not crash and did finish with correct results. However, the performance is very poor (see table 7).

Further investigation showed that in fact the performance of the multiplication kernels were actually better than that of my original OpenMP implementation (see tabel 6), with good scaling performances as the number of OpenMP threads increase. However the bulk of the time spent

Table 6: Strong scaling: Bulk Si 2048 atoms with SZ basis, 32 MPI processes, using original OpenMP (`!$omp do` at i-loops and no `DGEMM`) using Cray compiler 4.0.46 and single threaded `LibSci` 11.0.06

Number of Threads	Wall Time (s)	Multiplication Time (s)
1	3891.475	3570.430
2	2055.713	1842.980
4	1201.578	1013.300
8	780.665	599.511

Table 7: Strong scaling: Bulk Si 2048 atoms with SZ basis, 32 MPI processes, using Cray compiler 4.0.46 and `LibSci` 11.0.06

Number of Threads	Wall Time (s)	Multiplication Time (s)
1	3343.269	3021.270
2	1754.533	1540.130
4	3303.531	850.356
8	5313.146	516.028

in the 4 and 8 threads case was in integration in which there were also BLAS library calls. The BLAS library call inside an OpenMP parallel region is automatically single threaded by default behaviour, while outside the region the multi-threaded version is used. It appeared from the tests that CONQUEST ran very slow if linked with the multi-threaded version of Cray `LibSci`, this is consistent also with the results shown in section 4.

If instead of linking with Cray `LibSci` but with single threaded version of ACML, then CONQUEST did not crash, and the results (see table 8) showed very good strong-scaling. However, the ACML library appeared to be significantly slower than the `LibSci` (when working).

Table 8: Strong scaling: Bulk Si 2048 atoms with SZ basis, 32 MPI processes, using Cray compiler 4.0.46 and single threaded ACML

Number of Threads	Wall Time (s)	Multiplication Time (s)
1	4283.321	3874.730
2	2272.824	2018.400
4	1355.185	1130.590
8	922.253	707.000

8 Miscellaneous Changes to Conquest

Since the I have written several versions of OpenMP implementations of the matrix multiplication kernels, and these all required some temporary storage of sub-matrix indices that are not required

in the vanilla version, it would be wise to implement the OpenMP versions as separate subroutines. After discussions with the main developer of CONQUEST, we decided not to use C preprocessors for switching between the different versions (this would make the code almost unreadable), but instead moved the matrix multiplication kernel subroutines out of the original `multiply_module` and into its own module `multiply_kernel_module`, and each version of the matrix multiplication kernel would have its own source code file. The choice of which version of kernel to use would be made at compilation time through a flag in `system.make` (the makefile include file).

To implement this, I used a simple method of using variable substitution in makefile. The source and object file name for the kernel module was defined as `multiply_kernel_${MULT_KERN}` and each version of the kernel were given a name according to the following table:

Kernel Type	`MULT_KERN` value	File Name
Vanilla	`default`	`multiply_kernel_default.f90`
`DGEMM` without OMP	`gemm`	`multiply_kernel_gemm.f90`
OMP do, max: i, min: i	`ompDoii`	`multiply_kernel_ompDoii.f90`
OMP do, max: i, min: k	`ompDoik`	`multiply_kernel_ompDoik.f90`
OMP do, max: j, min: i	`ompDoji`	`multiply_kernel_ompDoji.f90`
OMP do, max: j, min: k	`ompDojk`	`multiply_kernel_ompDojk.f90`
OMP with `DGEMM`	`ompGemm`	`multiply_kernel_ompGemm.f90`
OMP with tasks	`ompTsk`	`multiply_kernel_ompTsk.f90`

Hence the choice of which kernel to use can be made by setting the correct value to `MULT_KERN`. `MULT_KERN` has a default value of `default`.

To make this selection method to work, I also modified the original automatic dependency generator script supplied with CONQUEST to make it only to scan a given list of source files instead of all files in the source directory. This had to be done because all the different kernel source files contained the same module name, which would confuse the automatic dependency generation process if the generator scanned all of the source files.

The new kernel selection method was tested on several different systems with different compilers and is working. This replaced the original run-time selection method between vanilla kernel and `DGEMM` kernel in CONQUEST (prior this work), and in the process removed the conditional construct inside the matrix multiplication subroutine associated with making such choices, hence further improves the efficiency of the subroutine. The CONQUEST input variable `MM.UseGemm` for making the run-time choice is thus obsolete.

CONQUEST allows the user to replace its matrix diagonalisation module used for non-order-N calculations with a dummy module if the user only wishes to do calculations using linear ordering method and does not have Scalapack on his/er system. This originally had to be done manually, by renaming the appropriate dummy module file and moving the original diagonalisation module to somewhere else. The developer and I realised the method of choosing the multiplication kernels could also be applied to this case, and hence implemented a new makefile variable `DIAG_DUMMY` so that if `DIAG_DUMMY` is set to `DUMMY` then the dummy file, named `DiagModuleDUMMY.f90` is used, and if left undefined, then the original `DiagModule.f90` is used.

I have also added a CONQUEST input flag `IO.DumpL` to control whether the auxiliary (density) matrix should be dumped to files after every self-consistent step. The vanilla CONQUEST always dumps the matrix to files, and if running on thousands of MPI processes this would produce thousands of large files (several or even dozens of MBs each), slowing the code down and clogging the IO

of the system significantly. The dumped auxiliary matrix is there to allow a follow-up calculation starting using to start results from previous calculation.

9 Summary and Future Work

In this dCSE project I have successfully implemented several versions of MPI-OpenMP hybrid architecture in CONQUEST. The original CONQUEST was profiled and the hottest part of the code was found to be in the matrix multiplication kernels. It followed that the OpenMP implementations were implemented in the corresponding multiplication kernels. The main objective of this dCSE project is to restore the linear weak-scaling of CONQUEST running on HECToR Phase 3, and this has being achieved through the OpenMP implementations.

Six versions of OpenMP implementation in the matrix multiplication kernels were implemented, which could be chosen by the user through defining an appropriate variable in the makefile input `system.make`. Four of the versions implemented correspond to work-sharing the different levels of `do` loops in the two matrix kernels (`ompDoii`, `ompDoji`, `ompDoik` and `ompDojk`), one version corresponds to OpenMP implementation using `DGEMM` for the support-function block level multiplications (`ompGemm`) and one corresponds to using OpenMP Tasks instead of `do` loop work-sharing (`ompTsk`).

The `ompGemm` version should be the fastest, however due to the issue of CONQUEST not working well with multi-threaded BLAS libraries (both Cray `LibSci` and ACML) on HECToR, the single threaded version of Cray `LibSci` not working when called inside a parallel region with more than 2 threads, and the fact that ACML library seems to be less optimised than `LibSci`, the `DGEMM` version did not work as well as it should have in practice on HECToR. The problems associated with the libraries are not well understood. The `ompTsk` version did not work well on HECToR at all. This may be caused by too much overhead associated with task creation and assignment, and also because the threads take up the computation tasks in an unpredictive manner cache re-usage may be poor. The current OpenMP task implementation should therefore be avoided when running CONQUEST on HECToR Phase 3. It follows therefore the best OpenMP implementations to use on HECToR Phase 3 at the moment are the `do` loop work-sharing versions. There are two main factors affecting performance: the neighbour-lists corresponding to the i, j and k atoms and the overhead associated with work-sharing the inner loops. The user may need to run some tests to find out which loop work-sharing version is the most optimum.

I found for the particular test systems used (bulk silicon and bulk ice) the static scheduling gave the best performances for the OpenMP implementation. In any case the user should be able to change scheduling and chunk sizes through the environment variable `OMP_SCHEDULE` without needing to modify or recompile the source code.

A mechanism allowing the user to choose a particular version of a source code file at compilation time has been implemented in CONQUEST. This was done without relying on any C-preprocessors, which was an explicit requirement by the CONQUEST developers.

For future work, it may be beneficial to achieve a better understanding of why CONQUEST performs poorly when linked with multi-threaded BLAS libraries available on HECToR Phase 3. Load-balancing in the MPI processes will be a major factor in the actual scaling performance of the code, and in the next dCSE project starting following the completion of the current work, I will be looking at ways to improve the work distribution algorithms (space filling curves and weighted distribution of partitions) that are used to automatically balance the work-load on each MPI process.

10 Acknowledgment

This project was funded under the HECToR Distributed Computational Science and Engineering (CSE) Service operated by NAG Ltd. HECToR—A Research Councils UK High End Computing Service—is the UK's national supercomputing service, managed by EPSRC on behalf of the participating Research Councils. Its mission is to support capability science and engineering in UK academia. The HECToR supercomputers are managed by UoE HPCx Ltd and the CSE Support Service is provided by NAG Ltd. http://www.hector.ac.uk

References

[1] http://www.order-n.org.

[2] D. R. Bowler. Linear-scaling density matrix minimisation and electron number conservation. Technical report, University College London, 2005.

[3] D. R. Bowler, T. Miyazaki, and M. J. Gillan. Parallel sparse matrix multiplication for linear scaling electronic structure calculations. *Comput. Phys. Commun.*, 137(2):255–273, June 2001.

[4] H. Hotelling. Some new methods in matrix calculation. *Ann. Math. Stat.*, 14(1):1–34, 1943.

[5] R. M. Martin. *Electronic Structure: Basic Theory and Practical Methods*. Cambridge University Press, 2004.

[6] T. Miyazaki, D. R. Bowler, M. J. Gillan, and T. Ohno. The energetics of hut-cluster self-assembly in ge/si(001) from linear-scaling dft calculations. *J. Phys. Soc. Jpn.*, 77:123706, 2008.

[7] A. H. R. Palser and D. E. Manolopoulos. Canonical purification of the density matrix in electronic-structure theory. *Phys. Rev. B*, 58(19):12704–12711, Nov 1998.