CP2K

Optimising the libgrid library for CP2K

Intel Parallel Computing Centre
Introduction

CP2K [1] is a freely available atomistic simulation program, able to study a wide range of molecular and bulk materials with methods from classical potentials through Density Function Theory (DFT) and QM/MM to Hartree-Fock and Møller-Plesset 2nd order perturbation theory (MP2). The code is becoming increasingly popular and is heavily used on many large European systems including “ARCHER” in the UK and “Piz Daint” in Switzerland.

Four different versions of CP2K can be compiled from a single source tree. These will subsequently be referred to as the SOPT, SSMP, POPT and PSMP versions. These correspond respectively to, a serial version, an OpenMP version, an MPI version and a mixed mode MPI/OpenMP version.

Baseline performance

We have used the H2O-64 benchmark to investigate the baseline performance of CP2K as this is representative of the types of calculations that many users run on CP2K. Figure 1 (left) shows the performance of the H2O-64 benchmark run on ARCHER using the POPT version of CP2K. We also used the Cray perftools-lite tool to generate a breakdown of the time spent in user code, libraries and MPI calls, these results can be found in Figure 1 (right).

![Figure 1: Performance of CP2K on ARCHER as obtained using the H2O-64 benchmark. The left hand plot shows the total runtime (CP2K) and also the time spent in some of the routines previously identified for optimization in [2]. These timings are reported by the codes own inbuilt timing routines. The right hand plot shows the performance breakdown as obtained via the Cray perftools-lite tool.](image)

Examining Figure 1 (left) we see that for the H2O-64 benchmark CP2K scales well to 192 processors after which the runtime remains almost constant. From the right hand plot we can see that as the number of processors increases the time spent in MPI begins to dominate which is largely to be expected, as the H2O-64 benchmark is a relatively small system and thus unlikely to scale to a very large number of processes. As our aim is to improve the vectorisation of the code we actually want to see which particular routines (functions or subroutines) the code spends its time in.
We therefore now focus on the performance of CP2K when running on a single node of ARCHER. The CRAY perftools-lite tool has been used to generate a sampling profile. The following snippet shows the output from profiling on 24 cores.

From the profile we can see that on 24 cores 57.2% of the runtime is spent in ETC, 27.2% is spent in USER code and 13.0% is spent in MPI. The time spent in ETC represents the time spent in library calls, either to libraries that are part of the CP2K source tree or to external libraries such as FFTW, BLAS, LAPACK, MKL or similar. The time spent in USER code describes the time spent in routines that are part of the main CP2K source, i.e. source code that is not contained inside a library.

The profile is dominated by the time spent in external libraries (ETC). In particular, the routines collocate_core_* and integrate_core_* amount to over 25% of the runtime. These routines (highlighted in yellow) are all located inside the libgrid library.

Libgrid is an optional library that can be linked with CP2K. It provides system specific tuned versions of the collocate and integrate routines. These routines are involved in the mapping between the sparse matrix representation of the electron

---

1 We also obtained profiling data using Intel Vtune on the EPCC Hydra/Phi machine and found that the same key routines were the most costly.
density (coefficients of the 3D Gaussian basis functions) and the real space grid. The integrate routines carry out integration of the Gaussian products whereas the collocate routines carry out the inverse operation. If libgrid is not available on a particular HPC system then CP2K uses generic versions of these routines.

Libgrid uses an auto-tuning framework to create optimal versions of the collocate and integrate routines for your particular architecture and compiler version. The library is generated by timing several source variants of the `collocate_fast.F` and `integrate_fast.F` source files. Dedicated routines are generated for products of the Gaussian functions of a given quantum number, \( l_p \). Each variant is then tested for correctness and the variant that runs fastest is retained for inclusion in the final version of the libgrid library. The user can control the number of different optimisations that will be tested, the maximum value of \( l_p \) to compute up to and the number of times a test is run when timing the code. When running with the default settings (6 optimisations, quantum numbers up to \( l_{\text{max}}=7 \) and 3 runs of each test) the library takes around 7.5 hours to build.

Once the library is built the `out_best/` directory contains the optimal `collocate_fast.F` and `integrate_fast.F` routines as obtained for your particular architecture and compiler. We have opted to use these optimal routines as our starting point and from there we attempt to improve upon the vectorisation of these routines. To speed up the process of building the libgrid library we have set \( l_{\text{max}}=1 \). This is possible as the H2O-64 benchmark actually doesn’t use angular momentum values, \( l_p \), greater than 1.

All of our subsequent experiments have been carried out on the EPCC Hydra cluster (comprised of Intel(R) Xeon(R) CPU E5-2650 0 @ 2.00GHz chips) using version 14.0.2, 14.0.3 or 15.0.1 of the Intel compiler. These tests have also been repeated using the ARCHER test machine (comprised of Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz) using ifort version 15.0.0. Similar performance, taking into account the different CPU speeds, was obtained on both machines.

### Vectorising collocate_core_0

We begin by looking at the vectorisation of the routines in `collocate_fast.F` and considering the performance and vectorisation of each of the `collocate_core_*` routines separately. We do this by taking a copy of `collocate_fast.F` and then apply our changes to this file new source file.

The `-vec-report` compiler flag can produce a large amount of output especially when the reporting level is set to 5 or above. In order to make interpreting the output easier we carried out much of our analysis using a stripped down version of `collocate_fast_opt.F` which contains only a single `collocate_core_*` routine at a time with empty stubs for the other routines (e.g. when working on `collocate_core_0`, `collocate_core_default`, `collocate_core_1`, etc would be empty). This makes it considerably easier to interpret the `-vec-report` output as we only get information regarding the routine we are currently optimising. The downside is that this stripped down file can’t actually be compiled into a valid binary meaning two versions of the code have to be maintained. This was
only found to be an issue when working with the 14.0.X versions of the compiler. With version 15.0.X the output for each function/subroutine is separated (rather than being interleaved) which makes interpretation of the output much easier.

For `collocate_core_0` the initial vectorisation report at level 2 gives us:
Increasing the reporting level to 6 gives us a lot more information:

If we now look at lines 73-76 of `collocate_core_0` we can see that the compiler thinks that there is some dependence between different entries of the grid array.

However, we know that no such dependence exists and can use the `!DIR$ IVDEP` directive before the loop over `ig` (line 61) to tell the compiler this. Adding this directive gets rid all the vector dependence messages. The output from `-vec-report=6` is now:

Now we need to address the alignment issues. If data is not aligned in memory any benefits from increasing the amount of vectorisation can be negated by the overheads resulting from the data not being properly aligned in memory. The map variable is actually defined in the source file `qs_collocate_density.F` and thus the alignment directive, `!DIR$ ATTRIBUTES ALIGN: 64 :: map` for map must to be added to this file. We also need to `tell` the compiler that the data is aligned within `collocate_core_0` using the directive `!DIR$ ASSUME_ALIGNED` map : 64. As these two directives are in separate files we may not actually see any
improvement when compiling test_collocate.F with `-vec-report=6` and thus we need to build the binary and see if the performance has improved or not. On testing the binary we find that the performance has not improved at all. Our suspicion is that the loop over `ig` is still not being vectorised correctly.

At this stage we noticed that the loop over `ig` could be simplified in several ways. For `collocate_core_0` the value of `lp` is 0 and thus the loop over `lxp` is unnecessary and can be removed. The compiler should be capable of performing this type of optimisation. Also we can store `s01, s02, s03` and `s04` in a one dimensional array, `s(4)`, which enables us to use Fortran array syntax which should in theory help with vectorising the code. After making these changes our loop over `ig` becomes:

```
DO ig=igmin,igmax
   i=map(ig,1)
   s(:)=0.0_wp
   s(:)=s(:)+coef_x(:,0)*pol_x(0,ig)
   grid(i,j,k) = grid(i,j,k) + s(1)
   grid(i,j2,k) = grid(i,j2,k) + s(3)
   grid(i,j,k2) = grid(i,j,k2) + s(2)
   grid(i,j2,k2) = grid(i,j2,k2) + s(4)
END DO
```

We now re-examine the resulting vectorisation report:
Basically, the loop cannot be vectorised because the loop index, ig, is being to set the value of i (via i = map(ig,1)) which is then subsequently used to update the elements of the grid array. We tried creating a vector ivec(-cmax:cmax) and pre-computing the i values outside the loop over ig but this doesn’t help us as the compiler doesn’t know that the i values are always being accessed in a contiguous manner and that there are no overlapping values.

Examining the algorithm we see that for each igmin to igmax loop the map(:,1) array contains one or more blocks of contiguous i values with jumps between the blocks of contiguous values. Within a given loop over ig there are no repeated i values. E.g. map(:,1) might contain something like (24,25,26,27,28,29,30,31,5,6,7,8). Here we have two consecutive blocks of i values, 24-31 and 5-8 which should be vectorisable. Therefore, if we can re-write the loop such that we can make the innermost loop a loop over contiguous i values then it should be possible to vectorise the loop over ig.

This can be achieved by working out how many blocks of contiguous i values there are and then replacing the loop over ig with a loop over blocks where each block loops over a contiguous set of i values. This approach requires us add a new loop over ig where we pre-compute the starting i value and final i value for each block along with the starting ig value for each block as this is required to compute pol_x(0,ig). The code for doing the pre-compute is then as follows:
The loop over \( ig \) can then be replaced by a loop over \( \text{countblocks} \) blocks inside which we have a loop over contiguous \( i \) values e.g. we now have:

```plaintext
DO ib = 1, \text{countblocks}
  DO i = starti(ib), stopi(ib)
    ig = starti(ib) + i - starti(ib)
    s(:)=coef_x(:,0)*pol_x(0,ig)
    grid(i,j,k) = grid(i,j,k) + s(1)
    grid(i,j2,k) = grid(i,j2,k) + s(3)
    grid(i,j,k2) = grid(i,j,k2) + s(2)
    grid(i,j2,k2) = grid(i,j2,k2) + s(4)
  END DO
END DO
```

The inner most loop over \( i \) can now be vectorised because we have removed the dependence on \( \text{map}(ig,1) \) and the compiler now knows that consecutive values will be accessed. After making these changes we had expected to see that the inner loop would be vectorised, however, we still see dependencies between \( i \) and \( ig \). If we add the \#DIR$ SIMD directive directly before the loop over \( i \) then we can force the compiler to vectorise this loop but the resulting performance is still very poor.

Essentially the problem is that the loop still contains references to both \( i \) and \( ig \) and these depend on each other. Although the references to \( i \) are contiguous and vectorisable, \( ig \) is required to look up which part of \( \text{pol}_x \) should be used to compute \( s() \). We therefore attempted to split the loop such that we have a loop over \( ig \) and a loop over \( i \). To do this a new two-dimensional array, \( \	ext{sar} \), was created such that we compute the contribution to this array for \( ig_{\text{min}} \) to \( ig_{\text{max}} \). The code for doing this looks like:

```plaintext
do ig = igmin,igmax  ! pre-calculate all the contributions to sar for
  ! the whole range of ig
  sar(:,ig)=coef_x(:,0)*pol_x(0,ig)
end do
DO ib = 1, \text{countblocks}
  offset = startig(ib) - starti(ib)
  DO i=starti(ib),stopi(ib)
    grid(i,j,k) = grid(i,j,k) + sar(1,i+offset)
    grid(i,j2,k) = grid(i,j2,k) + sar(3,i+offset)
    grid(i,j,k2) = grid(i,j,k2) + sar(2,i+offset)
    grid(i,j2,k2) = grid(i,j2,k2) + sar(4,i+offset)
  END DO
END DO
```

On looking at the final output from the vectoriser, we now see that the loop over \( i \) can now be vectorised however the loop over \( ig \) cannot. When testing the algorithm in isolation all went well and both loops were vectorised. In our test code we had set
igmin to be a typical value as used in the code, e.g. -9. The problem with the actual code is that igmin is computed at runtime inside the outer loop over jg which in turn depends on the loop over kg and thus creates a dependency, which simply cannot be unravelled.

We also tried experimenting with aligning various data structures, e.g. map, pol_z and grid but this invariably makes the code run even slower than it does without any alignment. The alignment issues with grid cannot be removed by just using the align directives because of how the data is being accessed. However, if the loop over ib is split into two parts as shown below we can ensure that access to grid is aligned, however, again this makes the code run even slower.

```fortran
do i=starti(ib),stopi(ib)
    grid(i,j,k) = grid(i,j,k) + sar(1,i+offset)
    grid(i,j,k2) = grid(i,j,k2) + sar(2,i+offset)
end do

do i=starti(ib),stopi(ib)
    grid(i,j2,k) = grid(i,j2,k) + sar(3,i+offset)
    grid(i,j2,k2) = grid(i,j2,k2) + sar(4,i+offset)
end do
```

Finally, after discussing our problem with Xinmin Tian (Intel) at SC14 we also tried experimenting with the OpenMP SIMD directive, !$omp simd, which is part of OpenMP 4.0 and has been implemented in Intel compiler versions 15.0.0 and above. The OpenMP SIMD directive can be applied to a loop in order to indicate that it can be transformed into a SIMD loop, i.e. multiple iterations of the loop can be executed concurrently using SIMD instructions. This directive appears very similar to the Intel specific SIMD directive !DIR$ SIMD/ #pragma simd and takes many similar clauses etc. There are some subtle differences in the implementation, e.g. the handling of memory copies and the alignment of arrays etc. However, the advantage of using the OpenMP directive is that the resulting code will be portable to other OpenMP 4.0 compliant compilers.

A summary of the main optimisations applied and their impact on the performance of the code can be found in Table 1. The fastest result of 10 runs was taken and the results verified for correctness each time.

<table>
<thead>
<tr>
<th>Code version</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original code</td>
<td>2.423632</td>
</tr>
<tr>
<td>Adding !DIR$ IVDEP to loop over ig</td>
<td>2.472624</td>
</tr>
<tr>
<td>Attempt 1: Array syntax</td>
<td>2.438629</td>
</tr>
<tr>
<td>Attempt 1: Array syntax + !DIR$ IVDEP on loop over ig</td>
<td>2.437631</td>
</tr>
<tr>
<td>Attempt 1: Array syntax + !DIR$ VECTOR ALWAYS on loop over ig</td>
<td>2.430630</td>
</tr>
<tr>
<td>Attempt 1: Array syntax + !DIR$ SIMD on loop over ig</td>
<td>2.463625</td>
</tr>
<tr>
<td>Attempt 1: Array syntax + !$OMP SIMD private(i,s) on loop over ig</td>
<td>2.484623</td>
</tr>
<tr>
<td>Attempt 1: Array syntax + align map and pol_x</td>
<td>2.479622</td>
</tr>
<tr>
<td>Attempt 2: use ivec(ig) array and array syntax</td>
<td>2.524676</td>
</tr>
<tr>
<td>Attempt 2: use ivec(ig) array and array syntax + !DIR$ IVDEP on loop over ig</td>
<td>2.477632</td>
</tr>
<tr>
<td>Attempt 2: use ivec(ig) array and array syntax + !DIR$ SIMD on loop over ig</td>
<td>2.473624</td>
</tr>
<tr>
<td>Attempt 2: use ivec(ig) array and array syntax + !$OMP SIMD private(i,s) on loop over ig</td>
<td>2.580608</td>
</tr>
<tr>
<td>Attempt 2: use ivec(ig) array and array syntax + !$OMP SIMD private(i,s) on loop over ig</td>
<td>2.620602</td>
</tr>
<tr>
<td>Attempt</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>2</td>
<td>use ivec(ig) array and array syntax + localmap 1d array used to compute i</td>
</tr>
<tr>
<td>3</td>
<td>replace the ig loop with loops over countblocks and starti(ib) to stopi(ib) for each block of contiguous iterations</td>
</tr>
<tr>
<td>3</td>
<td>replace the ig loop with loops over countblocks and starti(ib) to stopi(ib) for each block of contiguous iterations + !DIR$ IVDEP on loop over i</td>
</tr>
<tr>
<td>3</td>
<td>replace the ig loop with loops over countblocks and starti(ib) to stopi(ib) for each block of contiguous iterations + !DIR$ VECTOR ALWAYS on loop over i</td>
</tr>
<tr>
<td>3</td>
<td>replace the ig loop with loops over countblocks and starti(ib) to stopi(ib) for each block of contiguous iterations + !DIR$ SIMD on loop over i</td>
</tr>
<tr>
<td>3</td>
<td>replace the ig loop with loops over countblocks and starti(ib) to stopi(ib) for each block of contiguous iterations + !$OMP SIMD private(s) on loop over i</td>
</tr>
<tr>
<td>4</td>
<td>as per attempt 3 but now split into two loops, one over ig and one over countblocks etc</td>
</tr>
<tr>
<td>4</td>
<td>as per attempt 3 but now split into two loops, one over ig and one over countblocks etc + !DIR$ IVDEP on loop over i</td>
</tr>
<tr>
<td>4</td>
<td>as per attempt 3 but now split into two loops, one over ig and one over countblocks etc + !DIR$ VECTOR ALWAYS on loop over i</td>
</tr>
<tr>
<td>4</td>
<td>as per attempt 3 but now split into two loops, one over ig and one over countblocks etc + !DIR$ SIMD on loop over i</td>
</tr>
<tr>
<td>4</td>
<td>as per attempt 3 but now split into two loops, one over ig and one over countblocks etc + !$OMP SIMD private(s) on loop over i</td>
</tr>
</tbody>
</table>

Table 1: Summary of different optimisations attempted with Intel compiler version 15.0.1. This includes the OpenMP SIMD directive results. The * indicates that the compiler couldn’t vectorize the code due to data dependencies that it couldn’t unravel.

Although we’ve now successfully managed to vectorise the innermost loop of the collocate_core_0 routine the vectorised code does not go any faster than the original code (see Table 1). In fact rather disappointingly, it actually goes slower due to the overheads involved in the re-factoring of the code or the application of the SIMD directive or OpenMP SIMD directives.

Our inability to improve the performance is essentially because the data access pattern on the grid array is very complex and depends on the variables computed in the outer loops. By adding the directives or re-factoring the code we can access blocks of contiguous ig values, however, the other two indices of grid are not accessed contiguously and thus the code doesn’t go any faster.

The structure of the other routines within libgrid is very similar to that of collocate_core_0 with and thus it is not expected that any performance improvement will be obtained by attempting to vectorise the inner most loops of these routines either and thus we are now looking into optimising other parts of the code.

References
