1 Motivation and Aims

This document presents the ongoing work undertaken towards defining and developing a suite consisting of four mini-applications (hereafter mini-apps), each of which pertains to one of the four ab initio Computational Materials Science (aiCMS) applications involved in the NOMAD CoE - namely ABINIT, exciting, FHI-aims, and GPAW.
Benchmarking of the aforementioned (and similar) aiCMS applications is routinely used to inform the implementation of new high performance computer (HPC) systems. Ideally, this would be a continuous process not only throughout the implementation, but also in the initial design phase of the system. The complexity of the software, however, often demands that the full scope of HPC features be available, and thus benchmarking is typically only practicable once the development of the system approaches production capabilities. It is this problem that mini-apps primarily address; a mini-app is, by definition, a small fraction of the code length and complexity of its parent application, and yet retains the primary performance-intensive aspects [1–3].

Our mini-apps suite will be used as a benchmarking tool for the computationally demanding features specific to each the four aforementioned applications. This will support our efforts towards co-design as the performance metrics will be transferable between (selected) HPC systems and (pre-exascale) prototypes. We also intend the mini-app suite to inform the developers of the parent applications of the feasibility of potential directions for future developments. We hence intend both the benchmark performance metrics and mini-app suite itself to be of utility primarily in the following ways:

(a) Early node architecture studies – Our mini-apps will provide opportunities to study node performance early in the design cycle of the processor node architecture, a strongly determining factor in scalable system performance.

(b) Network scaling studies – Our mini-apps are developed from to run on a wide ranging number of processes, and will be an easily deployable tool to provide early insight into network scaling issues, as well as subsequently to test network scalability in production-ready systems.

(c) New language and programming models – Our mini-apps are designed from inception to be easily refactored, and even completely rewritten in new languages and programming models. This will inform the development groups of each parent application when considering extending or rewriting production applications.

In order to ensure that these aims are achieved, we impose the following requirements on each mini-app (letters in parentheses refer to the utility defined in the list above):

1. consist of section(s) of its parent application selected from the kernel(s) identified during the performance evaluation undertaken in support of NOMAD WP8 Task 1 (a,b,c)

2. be able to run real world inputs precipitating a range of data sizes (a,b,c)

3. adhere to software engineering standards including sustainability, extensibility,
With these requirements in mind, the remainder of this document is organised as follows. Section 2 briefly introduces the theoretical background of the overarching focus of interest of our mini-apps development, and therefore also of the performance evaluations which inform it. This is intended to orient the reader (as necessary) and also provide a quick reference for this and future work. Section 3 outlines the programming models considered to be of relevance to the suite, and summarises the technical aspects of each of the four parent applications. Section 4 describes the salient findings of our performance evaluations with respect to each parent application and identifies kernels to be implemented in the suite. Section 5 summarises the current state of the development of suite.

2 Theoretical Background

Development of each mini-app is informed by analysing the performance of its respective parent application executing GW calculations, specifically using the so-called single-shot GW approach. Both the general GW formalism and this particular flavour are therefore sketched here for completeness and ease of reference.

2.1 Computational materials science beyond the ground state

Ab initio computational materials science - *in silico* modelling from first principles of the quantum mechanical properties of systems such as bulk materials, solvates, surfaces, clusters, and molecules - has become a distinct third axis in the practice of materials science, complimenting its experimental and theoretical counterparts. By far the most widely-used approach among the hierarchy of methods in this field is Kohn-Sham density functional theory (DFT) [4, 5], which reformulates the many-body quantum mechanical problem as a set of loosely-coupled single-particle problems, and thereby offers a computationally tractable formalism. DFT permits both the physics and chemistry communities to study a diverse range of many-electron systems with acceptable accuracy at a relatively modest computational cost in both its cubic- and linear-scaling forms; it is often the only possible approach to obtain useful *ab initio* results for relevant system sizes.

A fundamental limitation of the DFT formalism, however, is that it does not strictly extend to the study of excited states, i.e. those other than that of the lowest energy. This limitation is particularly significant in the immensely technologically relevant field of semiconductor physics, where many of the phenomena of interest, including electronic band gaps and (inverse) photoemission spectra, emerge from the properties of
excited states. Although differences between single-particle energies obtained from DFT are often interpreted as physical excitation energies, and sometimes to good effect, a formal justification for this practice does not generally exist [6, 7]. Furthermore, the accuracy of quantities derived from these energies is sensitive to various underlying characteristics of the system; results may consequently be unusable, and the efficacy of DFT as the theoretical foundation of a predictive model is generally rendered system-dependent. Higher levels of theory are thus frequently required in order both to maintain first principles and reliably attain greater accuracy.

2.2 The GW approximation

The GW approximation [8] is a well-established and systematic approach to improving upon the single-particle excitation spectra obtained from implementations of DFT, Hartree-Fock, and related hybrid approaches. Central to this improvement is the introduction of the self-energy, $\Sigma$, which in principle contains all electron–electron interactions beyond the Hartree energy. The GW approximation derives its name from the definition of the self-energy in the formalism, i.e. in terms of an expansion of the screened Coulomb interaction to first-order only,

$$\Sigma(r, r'; \omega) = \frac{i}{2\pi} \int d\omega' G(r, r'; \omega + \omega') W(r, r'; \omega') e^{i\omega'\delta}. \quad (1)$$

The two terms in the integrand of (1), besides the exponential term, in which $\delta$ is an infinitesimal positive number, are as follows. $G$ is the interacting single-particle Green’s function, the poles of which are associated with the single-particle excitation energies. This Green’s function itself depends on $\Sigma$ through the Dyson equation,

$$G^{-1} = G_0^{-1} [\Sigma - v_{\text{XC}} + \Delta v_{\text{H}}], \quad (2)$$

where $G_0$ is the Green’s function of an independent particle system (see Subsection 2.3), $v_{\text{XC}}$ is the exchange-correlation potential of the DFT system (for example) corresponding to $G_0$, and $\Delta v_{\text{H}}$ accounts for changes in the Hartree potential due to density differences between $G_0$ and $G$.

$W$ of (1) is the screened Coulomb interaction,

$$W(r, r'; \omega) = v(r, r') + \int \int dr_1 dr_2 v(r, r_1) P(r_1, r_2; \omega) W(r_2, r'; \omega), \quad (3)$$

where $v$ is the bare Coulomb interaction, and $P$ is the polarizability (taken as that
defined in the random-phase approximation),

$$P(r, r'; \omega) = -\frac{i}{2\pi} \int d\omega' G(r, r'; \omega + \omega') G(r', r; \omega') e^{i\omega'\delta}. \quad (4)$$

The key objects of the formalism are thus $G$, $W$, $P$, and $\Sigma$; their inter-dependence demands that, in principle, each be determined self-consistently.

### 2.3 Single-shot GW

The difficulty inherent to progressing with the complex inter-dependence of the self-consistent GW formalism is circumvented in a scheme commonly referred to as single-shot GW, or simply $G_0W_0$ [9]. Single-shot calculations are by far the most commonly-performed variant of GW due to the scheme being relatively straightforward to implement compared with fully self-consistent approaches, while often retaining acceptable accuracy.

$G$ is approximated by $G_0$, the non-interacting Green’s function obtained (typically) at the DFT level,

$$G_0(r, r'; \omega) = \sum_{nk} \frac{\psi_{nk}(r)\psi^*_{nk}(r')}{\omega - \epsilon_{nk} - i\eta}, \quad (5)$$

where $\eta$ is a positive (negative) infinitesimal for occupied (unoccupied) one-particle states. $\psi$ and $\epsilon$ are respectively the DFT eigenfunctions and eigenvalues, whose band and $k$-point indices are respectively $n$ and $k$.

The screened Coulomb interaction, $W$, is correspondingly approximated by $W_0$,

$$W_0(r, r'; \omega) = \int dr_1 v(r_1, r') \epsilon^{-1}(r, r_1; \omega). \quad (6)$$

$\epsilon$ in (6) is the dielectric function; this is derived from (4) but with $G_0$ replacing $G$. This gives the corresponding approximation for the polarizability, $P_0$, in terms of the DFT eigenfunctions,

$$P_0(r, r'; \omega) = \sum_{n,m} \sum_k F(k, q; \omega) \psi_{nk}(r)\psi^*_{mk-q}(r)\psi^*_{nk}(r')\psi_{mk-q}(r'). \quad (7)$$

where $q$-point $q \equiv k - k'$ and the factor $F$ depends on the frequency of $G_0$, as well as the occupation of the DFT states entering (5).

The self-energy is of course subsequently re-defined within the $G_0W_0$ approximation by making the obvious substitutions in (1). Approximations to the quasi-particle energies are henceforth obtained via a linearised Taylor expansion of the self-energy around the
DFT eigenvalues,
\[ \epsilon_{n\mathbf{k}}^{QP} = \epsilon_{n\mathbf{k}} + \langle \psi_{n\mathbf{k}}(\mathbf{r}) | \text{Re} [\Sigma(\mathbf{r}, \mathbf{r}'; \epsilon_{n\mathbf{k}})] - v_{XC}(\mathbf{r}) \delta(\mathbf{r} - \mathbf{r}') | \psi_{n\mathbf{k}}(\mathbf{r}') \rangle . \] (8)

The matrix elements of the self-energy entering (8) are non-zero only on the diagonal; neglect of the off-diagonal elements accounts considerably for the relative computational efficiency of the single-shot approach.

The scaling of the number of operations needed to perform any flavour of GW calculations, including single-shot, is typically \( O(N^4) \) with respect to the system size (i.e. number of electrons). This is true of planarwave implementations, and also of localised bases if certain techniques, such as resolution of the identity (see e.g. [10] and references therein), are employed. A formalism for cubic-scaling GW was pioneered by the space-time approach of Rojas, Needs, and Godby in 1995 [11], and attempts at practical implementations now abound, including within NOMAD (see WP2 for details). It should be noted that such attempts are not considered in relation to development of the mini-apps, however; our sole focus is the quartic-scaling single-shot implementation extant in each parent application.

3 Technical Background

We have provided, or will provide, support in the mini-apps suite for several programming models. Each has been selected because it is either already utilised, to varying extents, in a parent application, or because it offers potential optimisation solutions to a bottleneck identified therein. Hence both the brief description of the programming models that follows, and the accompanying overview of the technical specifications of each parent application, inform the discussions of the performance analyses of the parent applications in Section 4 and the development of the mini-apps suite in Section 5.

3.1 Programming models

- MPI (The Message Passing Interface) is the de facto standard for developing parallel applications for platforms which support the distributed memory model, such as HPC clusters. MPI is implemented through library functions called from within existing sequential programming languages. Both intra- and internode parallelism is supported, processes usually communicating via optimised shared memory constructs and explicit message passing respectively. Communications are most commonly two-sided, including both point-to-point and collectives. This mode necessitates remote process synchronisation, since both sender and receiver must participate in order to complete the data movement. One-sided
communications are also possible, whereby a process directly accesses a remote address space without the explicit participation of the remote process. This mode can be more efficient than two-sided communications in some contexts, but can also be more challenging to implement safely.

- OpenMP (Open Multi-Processing) is the de facto standard in shared memory programming. It is implemented via a set of pragmas (or directives) that are added to source code to express parallelism. Only intranode parallelism is typically possible by definition, since the model requires a shared memory-space to be addressable by all threads. OpenMP for multi-core CPUs is supported by all the major compiler vendors. Support for offloading OpenMP regions to accelerators was only introduced in OpenMP 4.0, and this has subsequently been extended (most recently in OpenMP 5).

- CUDA (Compute Unified Device Architecture) is essentially a proprietary extension to C/C++ (and a corresponding extension to the Fortran language) developed and controlled by NVIDIA. It is specifically designed to enable GPU calculations and manage memory between the CPU and GPU. The model requires applications to be restructured in order to most efficiently use the GPU architecture and exploit the massive parallelism therein. It can only target NVIDIA GPU hardware, however, and therefore has implications for application portability.

- OpenACC (Open ACCelerator) is a pragma-based programming framework which, unlike OpenMP, is specifically designed to enable computation to be offloaded to accelerators. It is intended to minimise the modifications required to existing source code and provides a portable, open, standards-based solution. It currently has well-established support from NVIDIA compilers, and is also increasingly supported by Open Source GNU (from version 10 onwards in practical terms).

### 3.2 Technical overview of the applications

Each of the four parent applications uses a different basis set in which to expand the wavefunctions (and other objects) throughout a calculation. The choice of basis naturally has relevant implications for important computational aspects specific to each code, such as the treatment of the atomic potential, the available dimensions of parallelism, and possible performance bottlenecks. A detailed description of each basis is beyond the scope of this document; Table 1 simply states them, and the atomic potential treatment, for future reference.

Table 2 summarises the relevant technical details of each application.
Table 1: Summary of the atomic potential method and wavefunction basis set expansion of each of the four applications.

<table>
<thead>
<tr>
<th>Wavefunction basis</th>
<th>ABINIT</th>
<th>exciting</th>
<th>FHI-aims</th>
<th>GPAW</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Augmented) planewave</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linearised augmented planewave + local orbitals</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Numeric atom-centred orbitals</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Projector augmented wave</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Atomic potential</th>
<th>Pseudo-potential</th>
<th>All electron</th>
<th>All electron</th>
<th>All electron</th>
</tr>
</thead>
</table>

Table 2: Summary of some of the computationally relevant technical specifications of the four applications.

<table>
<thead>
<tr>
<th>URL</th>
<th>ABINIT</th>
<th>exciting</th>
<th>FHI-aims</th>
<th>GPAW</th>
</tr>
</thead>
<tbody>
<tr>
<td>abinit.org</td>
<td>abinit.org</td>
<td>exciting.wikidot.com</td>
<td>fhi-aims.org</td>
<td>gitlab.com/gpaw</td>
</tr>
<tr>
<td>Version</td>
<td>9.5.4</td>
<td>Oxygen</td>
<td>200112</td>
<td>21.6.0</td>
</tr>
<tr>
<td>Lines of code</td>
<td>900k</td>
<td>265k</td>
<td>485k</td>
<td>130k</td>
</tr>
<tr>
<td>Language, ✓=primary</td>
<td>Fortran</td>
<td>Python</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Programming model, ✓=primary</td>
<td>MPI</td>
<td>OpenMP</td>
<td>CUDA</td>
<td>OpenACC</td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Compiler support</td>
<td>GNU</td>
<td>Intel</td>
<td>NVIDIA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

4 Performance Analysis and Kernel Identification

This section identifies kernels and/or features of each parent application to be implemented and developed in the mini-apps suite. Identification is based on salient points of the preliminary profiling and performance analysis completed thus far for each application. This was conducted by a member of the Performance Optimisation and Productivity CoE (pop-coe.eu) assigned to each application, in collaboration with the developer(s) of the respective parent application and the current author.

All profiling was performed on BSC MareNostrum 4 (MN4). Each MN4 node consists of 2 × 24 Intel Xeon Platinum 8160 Cores at 2.1 GHz, a 100Gb Intel Omni-Path Full-Fat Tree and 10Gb Ethernet network interconnects.
4.1 ABINIT

4.1.1 Test case parameters

Two test cases have been developed. One is a relatively small calculation in order to obtain an initial profile and assessment of the application; the other is a larger, more challenging system which may be more representative of real-world frontier calculations in some aspects. Only the small test case has been profiled to date, and hence this is the sole focus of the discussion below. Profiling and performance analysis of the larger test case will of course further inform future development of the mini-app.

- Small test case: ZrO$_2$; 3 atom FCC primitive unit cell
- K-points: 8×8×8 MP grid, 29 k-points in the irreducible set
- Bands to be used in the screening calculation: 1024
- Cut-off energy of the planewave set to represent the dielectric matrix: 8 Ha

4.1.2 Discussion

There are four distinct steps involved in a standard one-shot GW calculation in ABINIT (see docs.abinit.org/tutorial/gw1/ for details). The first two steps are respectively to obtain the ground state density of the system via a standard DFT (or hybrid) calculation and subsequently compute the corresponding single-particle eigenpairs. One can then proceed to Step 3 and compute the independent-particle susceptibility, $\chi_0$, of which the dielectric (see (6)) is a functional; the required inverse dielectric is obtained via matrix inversion. Finally, Step 4 is to compute the self-energy matrix elements, $\Sigma_0$ (see (1)), for a given set of k-points in order to obtain the GW quasi-particle energies of (8). While the second step can be expensive, mostly due to the inclusion of a number of unoccupied states, it is the third step which is the primary bottleneck in the calculation and is hence the present focus.

Profiling of Step 3 using the small test case with up to 8 MN4 nodes reveals that almost all of the time is spent in gw/m_chio.F90:cchi0, the routine to calculate the independent-particle susceptibility at q-points other than the origin. While some parts of this routine scale well (such as wfdesc/m_oscillators.F90:rbo_tw_g), the scaling of procedure wfdesc/m_wfd.F90:wfd%distribute_kb_kpbp is particularly poor; this is where most of the cchi0 time lies with 384 processes.

This routine distributes pairs of bands for which the susceptibility matrix elements are to be computed. The distribution is task-scheduled by a greedy algorithm which determines the MPI process(es) currently idle, taking into account the MPI distribution of the periodic (i.e. k-point dependent) part of the Bloch wavefunction in reciprocal space. This routine is called inside a loop over q-points; a separate routine, cchi0q0,
is used exclusively at \( q = (0,0,0) \) as this requires additional terms. The algorithm causing the bottleneck is not MPI-parallelised and hence the parallel efficiency is significantly affected when the number of MPI processes increases. A parallelised version is in active development and will be included, and further developed as necessary, in the mini-apps suite.

Figure 1 compares executions of Step 3 for the single molecule testcase on 1 (48), 4 (192) and 8 (384) MN4 ‘highmem’ nodes (processes). A timeline for each execution is shown, ordered from top to bottom with increasing number of nodes (colour-coded white (1), magenta (4), and light green (8)); corresponding function summaries are the below the timelines. In order to facilitate analysis, the timelines are filtered such that the \( \text{cchi0} \) function (green in the timelines) does not include \( \text{wfd\_distribute\_kb\_kpbp} \) (purple) or \( \text{MPI\_Allreduce} \) (red).

The wall-time duration of the three executions is 899, 427 and 472 seconds respectively. 4 and 8 nodes are therefore both about twice as fast as the single node but use roughly twice and four times the CPU time (core-seconds) respectively; clearly, the 8 node execution is beyond the strong-scaling limit of the application for this input configuration.

In the timelines, which are all on the same scale, one can distinguish the initial \( \text{cchi0}q0 \) (darker green) from the subsequent 28 \( \text{cchi0} \) ‘substeps’ of varying duration. Each \( \text{cchi0} \) substep starts with \( \text{wfd\_distribute\_kb\_kpbp} \) and concludes with calls to \( \text{MPI\_Allreduce} \). The longer substeps involve more \( q \)-points than the shorter ones; the workload is \( q \)-point dependent because symmetries in the Brillouin zone are exploited to reduce the number of matrix elements that must be computed. Comparing 1 and 8 nodes, the total time spent in \( \text{cchi0} \) increases \( 1.8 \times \) from 38222s to 67477s, but is scaling much better than \( \text{wfd\_distribute\_kb\_kpbp} \), which increases \( 54 \times \) from (a relatively negligible) 1802s to (a dominant) 98410s.

A \( \text{cchi0} \) load imbalance is clear when the \( \text{MPI\_Allreduce} \) at the end of each substep is examined. This is not due to \( \text{wfd\_distribute\_kp\_kpbp} \) but rather the rest of \( \text{cchi0} \), which manifests as more waiting time in the concluding \( \text{MPI\_Allreduce} \) for those processes with higher MPI ranks in each case. This load imbalance arises from the block-wise partitioning of the 1012 active bands: the lowest 4 ranks of the 48-process execution, 52 of the 192-process execution, and 244 of the 384-process execution each receive an additional residual band relative to their peers. Note that these limitations of the implementation are also discussed in the ABINIT GW tutorial (see Section 2 of docs.abinit.org/tutorial/paral_mbt). This partitioning scheme will be included and optimised in the mini-apps suite.
Figure 1: Comparison of timelines (top) and most expensive function calls (bottom) for the single molecule test case on 1 (white), 4 (magenta) and 8 (green) MN4 nodes.
4.2 exciting

4.2.1 Test case parameters

All calculations fully populated four MN 4 nodes (192 cores). The developers of the parent application advised that k-points are currently the only memory-distributed objects and that parallelisation is mostly implemented in OpenMP. Therefore 4 MPI processes (1 per node) were used, with 48 OpenMP threads available per node.

- System: ZrO$_2$; 3 atom FCC primitive unit cell
- K-points: 2×2×2 MP grid
- Quasi-particle corrections applied to 12 (all) bands in the reduced (full) test case

4.2.2 Discussion

The total wall-times for the reduced and full calculations are 225s and 3636s respectively. Figure 2 compares the two calculations in terms of time spent in MPI calls and OpenMP regions. For both test cases, MPI parallelisation is limited to initialisation (region 1 in the figure) and finalisation (region 3) phases; this of course verifies the information provided by the developers. These phases are negligible in the full run (except for an imbalance at finalisation). The main computation (region 2 in the figure) in both calculations runs independently only with OpenMP parallelisation.

Figure 2: The exciting application structure in terms of the supported MPI and OpenMP programming models for the reduced (top) and full (bottom) calculation. MPI functions are colour differentiated; black indicates an absence of MPI calls, and is therefore serial or OpenMP only. The three regions are 1) initialisation, 2) OpenMP parallelisation only, and 3) MPI reduction and final synchronisation (red indicates MPI_barrier).
The reduced calculation identified two phases for which optimisations may be possible, but the weight of these phases changes considerably in the full configuration. Figure 3 shows the useful computation (i.e. time spent outside MPI functions) of the reduced (top) and full (bottom) runs. The reduced calculation is dominated by two functions, `src_gw/calcmicm.f90:calcmicm` implemented in the user code and `src_gw/.../gemm_omp_driver_v2` implemented in the Intel MKL library. These are colour-coded cyan and gray respectively. In the full calculation, however, the weight of the MKL library is significantly reduced, and thus the initial focus for implementation in the mini-apps suite is `calcmicm`.

![Figure 3: A zoomed in view of useful computation in the main OpenMP parallel region (see region 2 in Figure 2) for the reduced (top) and full (bottom) calculation.](image)

The `calcmicm` routine calculates the matrix elements $M_{cm}(\vec{k}, \vec{q})$, where $c$ is a core-state. The routine is essentially a multi-nested do-loop, within some of which are conditionals. OpenMP parallelism is implemented via a single `$OMP DO$ directive applied to the outer-most loop. This appears to offer opportunities for optimisation. Additional consultation with the application developers is scheduled in order to provide further context.

Figure 4 shows a clustering analysis of the main OpenMP region of the full calculation in terms of the average number of instructions executed per clock cycle (IPC), and the correspondingly colour-coded timeline. The analysis classifies the main computations in three clusters. The IPC for clusters 2 (yellow) and 3 (red) is higher than 2, but is only 0.46 for Cluster 1 (green). As a reference, an average IPC for applications on MN4 is between 1.5 and 2. Thus, Cluster 1, which corresponds to `calcmicm` and represents 97% of the computing time, suggests that current IPC performance is well below that which might be achievable. This analysis hence supports the previous findings and, further, suggests a direction for development in the mini-apps suite.
4.3 FHI-aims

4.3.1 Test case parameters

- Test case: ZrO$_2$; 3 atom FCC primitive unit cell
- K-points: $4 \times 4 \times 4$ MP grid
- Number of frequencies: 20

4.3.2 Discussion

Efficiency metrics were collected for a region of interest instrumented by the application developer. This region is within `evaluate_periodic_gw_selfenergy.f90`, specifically in the `evaluate_periodic_gw_selfenergy` routine. This calculates the polarizability (see (7)) inside the main loop of the GW computation. The region involves nested do-loops containing multiple calls to the ScaLAPACK `pzgemm` routine to perform matrix-matrix products of a double precision complex (general) matrix.
Figure 5: Useful time per process over the instrumented region of FHI-aims GW calculation (top) and (bottom). The black line is the mean useful time calculated over all processes.

Figure 6: Time spent in MPI calls (excluding point-to-point and collectives) over the instrumented region of FHI-aims GW calculation (top) and the total bytes exchange during communications (bottom). Blue (green) indicates longer (shorter) duration of communication (top), and more (fewer) bytes (bottom).

The wall-time for the instrumented region is 1454s, 53% of the total 2728s runtime of the calculation. A pattern is observed with respect to the useful time measured for each process over the duration, as can be seen in Figure 5. Every 64th process, and to a lesser extent 8th process, spends significantly more time in useful computation than its peers; the time ranges from \(\sim1000s\) for the highest to a low of \(\sim700s\). This may be due to the number of processes assigned to sub-communicators; the code automates sub-communicator population based upon the number of available processes and the parameters of the parallelised objects as determined by the input file.

Additional analysis of the the MPI calls throughout the instrumented region reveals that a specific communication pattern is performed periodically by two alternating groups of 64 processes. This pattern is manifest in the top timeline of Figure 6; blocks of blue and green repeatedly occupy the timeline of processes of increasing rank as the runtime proceeds. In this top figure, blue (green) indicates a longer (shorter) time spent in MPI calls other than point to point and collectives - the exact functions called have yet to be determined. The 64 processes blocks which are of higher MPI duration are not the regions with higher bytes exchange, however, as evident from the bottom figure, where
the number of sent and received bytes in all MPI calls is illustrated (colour-coding as previously, black indicating zero-bytes exchanged).

The communication patterns described, and the aforementioned sub-communicator hypothesis, are currently under further investigation; the related routines will in due course most probably form the basis of the implementation of this application in the mini-apps suite.

4.4 GPAW

4.4.1 Test case parameters

- System: ZrO$_2$; 6 atoms in total (3 atom FCC primitive unit cell, repeated (2,1,1))
- K-points: $2 \times 2 \times 2$ MP grid, 6 k-points in irreducible set
- GW planewave cutoff ($E_{\text{cut}}$): 300 eV
- Quasi-particle corrections applied to 6 bands per k-point
- ScaLAPACK blocking parameter ($N_{\text{blocks}}$): 8

4.4.2 Discussion

Figure 7 shows the global strong scaling behaviour of the application for the test case running on 1 (48), 2 (96), 3 (144), and 4 (192) MN4 nodes (processes). The speed-up effectively plateaus immediately for process counts greater than 48 (i.e. 1 node), indicating that there is most probably a significant number of optimisations which could potentially be developed.

Table 3 shows the POP efficiency metrics for the application; unity is ideal, values higher than 0.8 are considered good, less than 0.6 implies clear need for improvement. The application performs well on 1 node, with 0.92 global efficiency, but performance is poor with higher node counts; the metric deteriorates monotonically from 0.56 (2 nodes) to 0.32 (4 nodes).

Table 3 also shows that the instruction scaling is by far the worst performing POP metric, and further profiling concludes that replication of instructions is the main cause for the poor strong scaling behaviour. Examination of the non-scaling regions of the application, as shown in Table 4, reveals that the most expensive of these is the Dyson Equation (see (2) in 2.2).

The containing module for the Dyson Equation region is (perhaps not surprisingly) /response/g0w0.py, which will be adapted and developed within the mini-app suite.
The call hierarchy is

```
g0w0.py
  \rightarrow calculate(l:361)
    \rightarrow calculate_screened_potential(l:704)
      \rightarrow calculate_w(l:896).
```

Within `calculate_w`, the Dyson Equation calculation starts at line 1003 and ends at line 1168. That part of the code has no explicit calls to MPI. It is essentially a double
for-loop containing multiple conditionals, and multiple calls to numpy functions, specifically numpy.dot and numpy.linalg.inv between arrays. The functions called are also not parallelised, but could be potentially, e.g. kernels.py:get_coulomb_kernel.

5 Development of the Mini-Apps Suite

5.1 Framework

With reference to the points made in the previous sections, the overall intention of the mini-apps suite, in simple terms, is to provide a software package which is as straightforward and as efficient as possible to compile, install, and run within typical HPC environments. Thus far we have established a framework (see gitlab.bsc.es/afarres/nomad-mini-apps) with the essential features necessary to support further development. These include support for numeric libraries, unit testing, command line argument parsing, and automatic documentation. The following briefly describes some of these features as developed to date in the framework.

• Language support
  We support Fortran, Python, C, and C++. The application can be written in a mix of these languages seamlessly interacting with each other. The main program and the utilities provided are written in C++, but the numeric kernels can be written in any of the languages supported.

• Programming model support
  We currently support MPI and OpenMP; with CUDA and OpenACC support in development.

• Architecture support
  We have tested the framework application in several environments available at BSC, including
  - Intel Skylake using both GNU and Intel toolchains
  - IBM Power9 using both GNU and IBM toolchains
  - NVIDIA Volta GPU using the MAGMA interface.

• Compilation system
  The compilation system uses CMake to generate all the Makefiles. We include support to generated binaries for debug and release builds. A configure script is available to automatically compile for known environments (such as those found at BSC); this script can easily be adapted and extended to new environments.
• Library support
  We currently have support for dense algebra libraries BLAS, LAPACK and ScaLAPACK. Several implementations are supported, from reference implementations, such as OpenBLAS, to architecture specific ones including Intel MKL and IBM ESSL.

• Testing
  Tests can be defined through rules in the compilation system. A test usually consists of calls to the main program using specific arguments or a parameters file. Additionally, unit tests can be also defined using the Catch2 framework (github.com/catchorg/Catch2), which provides basic micro-benchmarking features and simple BDD macros.

• Documentation
  Documentation is generated automatically using Doxygen. This is the de facto standard tool for generating documentation from annotated C++ sources, and also supports all the other languages of the mini-apps suite, as well as many others.

5.2 Language selection

Both the main program and the utilities of the mini-apps suite are being developed in C++. There are many benefits to choosing C++ in this context, besides the obvious ones of mature support for interfacing with the languages of the parent applications, object-oriented programming, and provision of polymorphism. Here, we highlight some of the most relevant.

Greater exception-safety is innate to the language as it is based upon the Resource Acquisition Is Initialization model, which ensures that usage of any finite resource (e.g. memory) is scope-bound. Templates allow manipulation and generation of types before the final compilation; this promotes error detection at compilation, rather than execution time. Furthermore, the C++ Standard Library is almost entirely built from templates; library support is thus only generated as explicitly required, which facilitates compiler optimisation.

In brief, C++ transfers much of the work from runtime to compilation. This is appropriate because the mini-apps suite is intended to be compiled just once (say) but executed several times. Additionally, error management, which remains in execution time, is simplified.
References