# Task-Farming Parallelisation of Py-ChemShell for Nanomaterials: an ARCHER eCSE Project


## Abstract

We have implemented a two-level task-farming parallelisation framework in the new Python-based version of the ChemShell multiscale computational chemistry environment. Compared to the non–task-farmed code, we have achieved speedup factors of up to 5.7 times by carrying out a range of benchmark calculations at various QM/MM theory levels. We have also parallelised the code used for setting up solid state embedded cluster models. As part of this work we have enabled new science through a range of new functionality, including an interface to the highly scalable DL_POLY 4 classical molecular mechanics package, support for the Global Arrays (GA) library required by the NWChem quantum code, and a numerical gradients module. Through the developments in this project we now are able to model nanoparticles of over $1.6\times10^5$ atoms at the QM/MM level on the ARCHER supercomputer.



You Lu[1], Matthew R. Farrow[2], Alexey A. Sokol[2],
C. Richard A. Catlow[2], Paul Sherwood[1] and Thomas W. Keal[1]

[1] Scientific Computing Department, STFC Daresbury Laboratory, Daresbury, Warrington WA4 4AD
[2] Department of Chemistry, University College London, London WC1E 6BT



31 January 2018

## Table of Contents

## 1. Introduction

ChemShell (http://www.chemshell.org) is a computational chemistry environment for multiscale modelling. While it supports standard quantum chemical or force field calculations, its main strength lies in hybrid quantum mechanical/molecular mechanical (QM/MM) calculations. The concept of ChemShell is to leave the time-consuming energy evaluation to external specialised codes, while ChemShell takes over higher-level tasks, communication and data handling. The QM/MM approach, acknowledged in the awarding of the 2013 Nobel Prize to its original inventors, is particularly useful in the study of heterogeneous catalysis. ChemShell supports embedded cluster calculations [1, 2] in which the bulk material or surface is represented by a finite MM cluster model, optionally with additional surrounding point charges to mimic the effect of bulk electrostatics. The active site is modelled by a high-level QM calculation with electrostatic embedding, where the cluster environment is represented by point charges in the QM Hamiltonian. This setup is intended to give an optimal balance between accuracy at the active site and computational expense.

Py-ChemShell is a Python-based redevelopment of the original Tcl-based ChemShell code, which is a well-established module on ARCHER. The switch to Python has been planned since 2012, motivated by the wide appeal of Python in the scientific community and the power of the language and its libraries. Unlike the closed-source Tcl-ChemShell package, Py-ChemShell is free and open-source software released under the GNU Lesser General Public License version 3 (LGPLv3). Py-ChemShell development work began in 2014 and the first alpha release was made in December 2017, incorporating the work carried out in this eCSE. A first full release is planned in Spring 2018, after which Py-ChemShell will be deployed as a module on ARCHER for general use.

The current project *Task-Farming Parallelisation of Python-ChemShell for Nanomaterials* (eCSE08-14) was funded under the embedded CSE programme of the ARCHER UK National Supercomputing Service (http://www.archer.ac.uk). It aims at introducing a task-farming parallel framework, similar to that of Tcl-ChemShell which we implemented in a previous dCSE project, to the Py-ChemShell codebase for parallelising tasks that are commonly used in nanomaterial simulations. The eCSE proposal was written by Thomas Keal, Paul Sherwood (STFC Daresbury Laboratory), Alexey Sokol, and Richard Catlow (University College London). The project work was carried out by You Lu (STFC Daresbury Laboratory) and Matthew Farrow (University College London). 14 months of development effort was deployed over 12 chronological months from August 2016 to July 2017. The code development objectives for the project were:

**A.** Implementation of a task-farming parallel framework in the Python-based version of ChemShell using workgroups defined by MPI communicators.

Sharing of the MPI workgroup environment with external codes such as GAMESS-UK and GULP to perform multiple energy and gradient calculations simultaneously, to allow task-farming of common chemical tasks such as finite-difference gradients and nudged elastic band (NEB) optimisation in DL_FIND. Demonstration of a speedup factor of over 4 for finite-difference gradient evaluation in typical QM/MM calculations.

**B.** Parallelisation of the ChemShell routines that set up QM/MM model clusters from a periodic input structure, including parallel computation of electrostatic potential, field and field gradients on centres using Ewald summation and fitting of point charges around the cluster using this data to reproduce the missing periodic electrostatic interactions. Support for task-farming the cluster set up process to enable QM/MM calculations with multiple QM regions. Demonstration of parallel scaling on a target system containing 100,000 atoms, with a speedup factor of over 2 compared to the sequential calculation.

**C.** Support for task-farming with codes that use the Global Arrays (GA) library such as NWChem, implemented by creating GA processor groups corresponding to the MPI workgroups and passing these to NWChem as a directly-linked library. Demonstration of a comparable speedup factor (>4) to GAMESS-UK for finite-difference gradient evaluation using NWChem.

**D.** Creation of an interface to DL_POLY 4 in ChemShell including the facility to directly link the code and share the MPI environment. Demonstration of successful task-farming using ChemShell/NWChem/DL_POLY 4 with speedup factor comparable to ChemShell/NWChem/GULP above.

**E.** Enabling of QM/MM catalysis studies of nanomaterials using ChemShell/NWChem/DL_POLY 4. Demonstration of benchmark task-farming calculations beyond the capabilities of the previous Tcl-based version of ChemShell (systems of larger than 100,000 atoms in the MM environment), with single and multiple QM regions.

We have fully completed objectives A, C, and D, and carried out a series of benchmark calculations to assess the improvement in performance resulting from the task-farming framework. The best resulting gain is 5.7-fold speedup which surpasses our expectation in the project proposal. We have also parallelised the code to setup QM/MM model cluster in objective B. For objective E, we have managed to carry out calculations on nanomaterials of more than 160,000 atoms using the newly implemented interface to DL_POLY 4, well above our target. We have also designed a protocol to perform multi–QM-region calculations and finished associated code development of charge fitting which is necessary for this type of calculation.

# 2. Methodology

## 2.1 Task-farming parallelisation

Task-farming parallelism is useful in situations where a set of independent calculations have to be performed. [3,4] The *task farm* consists of all available processors, which are then divided into subsets of processors called *workgroups*. The tasks are split between the workgroups which work on them in parallel. As the calculations are independent, no information needs to be exchanged between workgroups during this time, and sharing of results can be postponed until all the tasks have completed. In computational chemistry, it is common to carry out calculations containing a number of single-point energy evaluations that are independent from each other. Typical types of applications include the nudged elastic band (NEB) method for energy minimisation, [5,6] finite-difference numerical gradients and Hessians, and population-based optimisation (e.g. genetic algorithms).

Figure 1 illustrates how the MPI parallel environment of Py-ChemShell was extended by the work of the current eCSE project. In the original framework shown in panel (a), all processes are grouped into a single MPI communicator (`MPI_COMM_WORLD`), which can be used for parallel execution of external codes. The Python interpreter, which parses the user input file, runs on a single master node. In the task-farming framework shown in panel (b), the processes are evenly grouped into `Workgroup 0` to $n_{\texttt{workgroups}}-1$, each containing a *Master* process and *Replica* process(es). Each Master instantiates a Python interpreter, which parses the user input script and invokes the external programs to execute the actual computational chemistry tasks. We will discuss more details of Py-ChemShell's mechanisms in the following part of this chapter. There is no lower limit for the size of workgroup (i.e., $n_{\text{processes}}$ per workgroup), so that there is no Replica process if the workgroups contain only one process each. The new task-farming framework allows Py-ChemShell to execute multiple energy evaluations simultaneously.

(a)

(b)

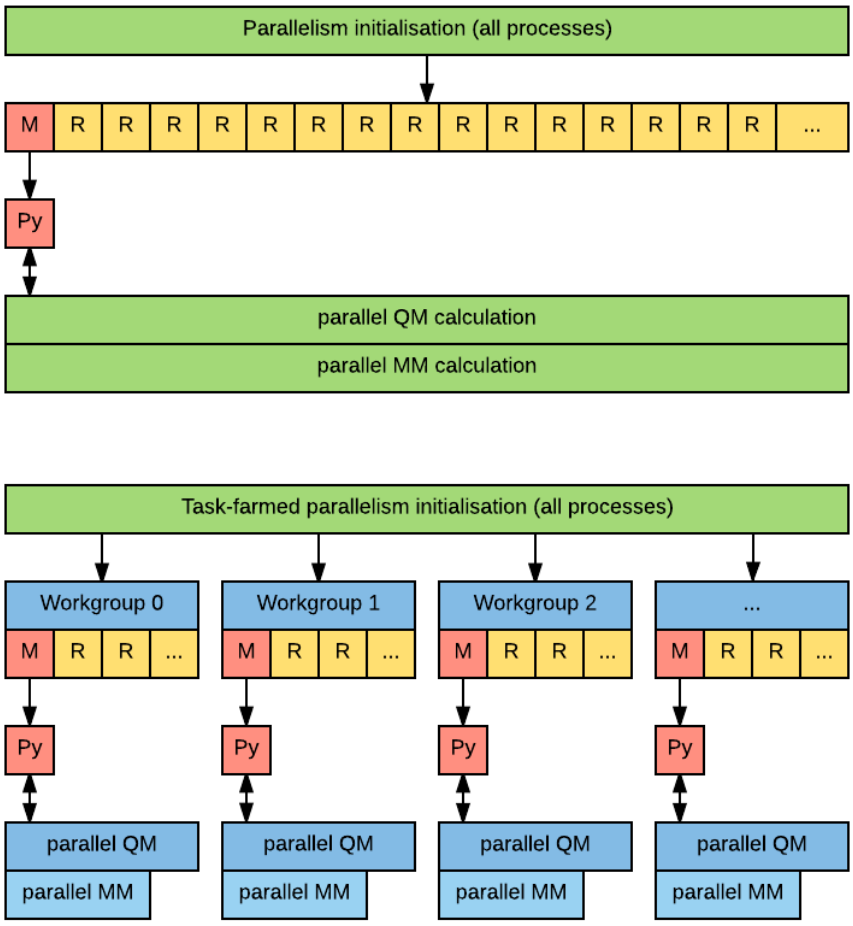


**Figure 1.** The parallel MPI framework in Py-ChemShell (a) before the work of the current eCSE (non-task-farmed parallelism) and (b) following implementation of the task-farming framework. M: Master; R: Replica; Py: Python interpreter.

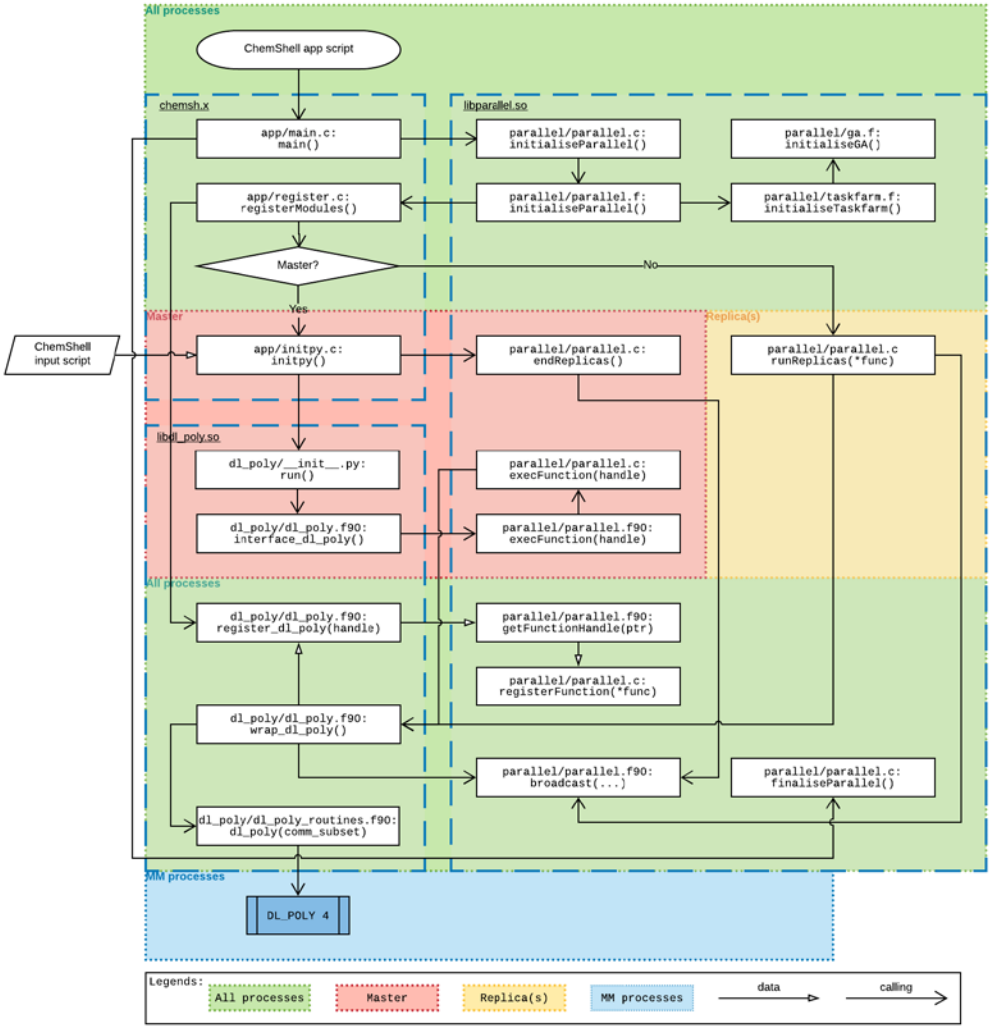


**Figure 2.** Flowchart of Py-ChemShell parallelism: the diagram's width represents the processes.

To implement the task-farming parallelisation in Py-ChemShell, a number of substantial changes to the code and the external programs were necessary. To relate how the mechanism works, let us walk through the flowchart of Py-ChemShell parallelism presented in Figure 2. There are three built-in ChemShell modules depicted in the dashed boxes: the executable binary `chemsh.x`, the main parallel infrastructural module `libparallel.so`, and an application module interfacing the external code (here we take DL_POLY 4 as an example). In the flowchart, we also use the dotted boxes' widths (and colours) to indicate the processes on which a method (namely, subroutine or function) is executed. According to the principle explained above, the processes are divided into Master and Replica at the level of task-farming parallelisation. Then the whole procedure of executing a task-farmed parallel ChemShell calculation is as follows.

I. The user executes the Py-ChemShell application script `chemsh.py`, which is pure Python, with command line arguments provided to specify the number of processes, workgroups, etc., for example:

```
chemsh.py --platform archer --account my_user_account --nprocs 24 --nworkgroups 2 --walltime 01:00:00 --jobname 2pfd_gamess mgo_2pfd_gamess_gulp.py
```

for a job to be submitted to the ARCHER standard queue requesting 24 CPU cores (1 node) and 1.0 h of wall time. In this example, the job will be task-farmed as `nworkgroups` is greater than 1, while it is non–task-farmed if `nworkgroups` is 1 (default) or left unspecified. All the command line arguments are available in short forms for the users' convenience, for example `-nwg` is equivalent to `--nworkgroups`. Getting the help information with argument `-h` or `--help` will print a full list of them.

II. When the submitted job starts to run, the executable binary `chemsh.x` invokes the C `main` routine, which initialises the root MPI parallel environment by calling the (Fortran) parallelisation initialisers encapsulated within `libparallel.so`. In the example of a task-farmed job, $n_{\texttt{workgroups}}$ communicators associated to the respective workgroups are created by the MPI function `MPI_COMM_SPLIT` acting on the MPI global communicator `MPI_COMM_WORLD`.

III. Then it is necessary to call the function `registerModules` to keep a reference (i.e., procedure pointer) to the application module, namely DL_POLY 4 in our example, in `libparallel.so` during the job's lifetime, because the application is not directly visible to the latter, which serves as a generic infrastructural library. A more detailed discussion is available in section 2.5.

IV. In the next step, the Master processes instantiate Python interpreters. The user input script containing the information of chemical structures and task specifications will be parsed in this step. The Python procedure is directly interoperable with the application's Fortran 2003 interface via Py-ChemShell's internal DL_PY2F library [7], by means of both function calling and memory sharing. The realisation is based on Fortran 2003's intrinsic module `iso_c_binding`, but we will not discuss the details here as it is not affected by the implementation of task-farming parallelisation. In the meantime, the Replica processes do nothing until the Master processes have done the application initialisation.

V. Both Master and Replica processes run the application wrapper function `wrap_dl_poly` using the previously saved reference, through the functions `execFunction` and `runReplicas`, respectively.

VI. The external program DL_POLY 4 should be separately compiled with the `-fPIC` flag as a static archive (`.a`) to be linked against the Py-ChemShell binaries. As part of the current project we have introduced a new feature into Py-ChemShell that allows the MM code being executed on just a subset of the workgroup processes rather than all of them in a workgroup (as shown in Figure 2, the processes for MM can have a width different from all processes). This is achieved by creating a separate MPI communicator solely for the use of the MM program, based on a number of processes specified by the user. This feature

provides the users the extra flexibility to run QM and MM codes on different numbers of processes. This is useful as MM programs have been observed to scale differently on typical QM/MM systems, where the MM region is relatively small compared to typical classical MD simulations, and the MM program can therefore suffer from unnecessarily large overheads at high core counts. Also thanks to this implementation, interfacing to DL_POLY 4 has become feasible in Py-ChemShell as the former currently has a hard-coded limit for maximum processes when computing non-periodic systems.

VII. Finally, after the calculation has completed, `main` issues a command to end all processes through `finaliseParallel`.

## 2.2 Parallelisation of cluster set-up routines

The parallel capability has been first implemented and is under optimisation and testing in a prototype CONSTRUCT code (version 90), which includes support for computationally intensive Ewald summations over 3D and 2D periodic systems. A QM/MM cluster set-up requires a calculation of the electrostatic potential and its derivatives over a set of atomic centres in an active region where response of all atoms is modelled explicitly in single point micro-iterative, geometry optimisation and Hessian calculations. This set is augmented, in particular on 2D models, by a regular 3D grid spanning a predefined box, or parallelepiped. Typically, the former includes ~1,500-2,000 centres and the latter ~$10^4$–$10^5$ sites. Parallelisation here is performed over the sites of interest using a standard MPI implementation with replicated data. Scalability tests are in progress. A bottleneck in fitting of point charges is removed by using appropriate linear algebra routines from ScaLAPACK, and the code is further parallelised at the stage of calculation of nanoparticle electrostatic potential and its derivatives at sites of interest, where the size of the largest nanoparticle tested > $1.6\times10^5$ guarantees the good scalability within one node or cluster. Further work will be undertaken in future to allow for the distributed memory model for particularly large systems. This work enables a QM/MM model set-up for systems with massive unit cells and/or large active QM and MM regions. The code will be transferred to Py-ChemShell on completion of validation tests.

## 2.3 Task-farm parallelisation of cluster set-up and QM/MM calculation routines for multiple QM regions

We have built up a QM/MM $ZrO_2$ nanoparticle containing over $1.6\times10^5$ atoms and successfully run a hybrid QM/MM calculation including the electrostatic potential of the whole particle at the NWChem/DL_POLY 4 level of theory. Our parallel benchmarks against the sequential calculation show that this type of calculation is highly scalable. The work is in progress to test task-farming parallelism of the hybrid QM/MM model set-up to support models with multiple independent and interacting QM regions, using the CONSTRUCT prototype at present**.** We have additionally

implemented a RESP (restrained electrostatic potential) charge fitting procedure into Py-ChemShell for use with multiple QM region calculations. This will enable efficient calculations of one QM region under the influence of another (going beyond the scope of the current project) following the incorporation of the CONSTRUCT prototype into Py-ChemShell.

## 2.4 Task-farming with Global Arrays using Py-ChemShell/NWChem

Global Arrays (GA) is a library developed by Pacific Northwest National Laboratory for high-performance shared-memory parallel computing with multidimensional arrays. A variety of popular quantum chemistry packages, including GAMESS-UK, NWChem, Molpro, Molcas, and TURBOMOLE, have support for GA and also have been interfaced to Tcl-ChemShell, so the support for GA in Py-ChemShell is a crucial target for achieving high sustainability in the long-term development.

The GA support has been incorporated into Py-ChemShell as part of the programming work in this project. Currently, the GA source code comes with NWChem if the user opts it in for the Py-ChemShell build. The GA static archive is compiled as a separate shared library `libga.so` to be linked against ChemShell's `libparallel.so` and it acts as an embedded subsystem in `ChemShellParallelModule`.

For executing a task-farmed parallel job with the support of GA, the GA environment and the necessary facility library Memory Allocator (MA) are first of all initialised by the intrinsic functions `GA_INITIALIZE_LTD` and `MA_INIT`. Function `GA_NODEID` is used to obtain the global rank information that is equivalent to `MPI_WORLD_RANK`. And the task-farming parameters, such as $n_{\text{processes}}$ per workgroup (obtained from function `GA_NNODES`), are applied to create GA process groups. The resulted group identities and node identities of each group, being dealt with by different physical processors, are then used by the ChemShell task-farm initialiser to split `MPI_COMM_WORLD` into workgroups with the function `MPI_COMM_SPLIT`, as described above in Section 2.1. Note that although the GA library does not in principle have to be built on top of MPI, the only parallel framework ChemShell supports is MPI and so we can be assured that the MPI workgroups will align correctly with the GA process groups.

The modular coupling of the GA library to Py-ChemShell secures the usability at compile time for users, the maintainability and sustainability for developers, and the accessibility by other external packages such as GAMESS-UK that are possibly driven by GA.

## 2.5 Integration of DL_POLY 4 into ChemShell for high performance QM/MM calculations on nanoparticles

The general-purpose parallel molecular dynamics simulation package DL_POLY has several variants, out of which DL_POLY 4 [8] is the newest and most highly scalable

distribution, being based on domain decomposition. However, only the older replicated-data version DL_POLY Classic has been incorporated and interfaced into Tcl-ChemShell. As part of this project we have implemented a DL_POLY 4 interface for Py-ChemShell, providing significantly enhanced MM scalability compared to Tcl-ChemShell.

Similar to other software packages used by ChemShell, DL_POLY 4 also yields a few static library archives (`.a` extension) when successfully built, which are then compiled into the shared library `libdl_poly.so` of Py-ChemShell. The invoking of `libdl_poly.so` functions in Py-ChemShell is described in Figure 2. The Fortran function `interface_dl_poly`, in the Py-ChemShell interface source file `dl_poly.f90` contained in the shared library, is of `bind(c)` attribute, thanks to the intrinsic module `iso_c_binding` of Fortran 2003 standard, and directly visible to the Python code as an accessible C function. The main routine of DL_POLY 4 with modifications for Py-ChemShell is adapted in the source file `dl_poly_routines.f90` and it overrides the native DL_POLY 4 main routine. When a DL_POLY 4 type job is executed with Py-ChemShell, the Python interface generates the required `CONTROL`, `CONFIG`, and `FIELD` files according to the user's requests and specifications, then invokes the DL_POLY 4 procedures, and finally collects the computed results, such as energies and gradients, by directly accessing the memory (rather than reading the text output file).

The parallelism of DL_POLY 4 in Py-ChemShell is also similar to those of other interfaced external programs. Figure 1 explains how the handle of (i.e., the pointer to) DL_POLY 4 main routine is managed and passed for the execution on Master and Replica processes. Moreover, when Py-ChemShell is initialised before any specific computational chemistry task is run, all installed modules with linked-in libraries are requested, by function `c_registerModules()` of the executable binary `chemsh.x`, to be registered with their respective handles which are saved in memory by function `c_registerFunction()`. At a later stage, after the DL_POLY 4 calculation task is set up, both the Master and Replica processes will execute the DL_POLY 4 main routine via the saved function handle. Such design is necessary in ChemShell and the Replica(s) will not be able to access the DL_POLY 4 main routine otherwise, because only the Master, whether task-farmed or not, initialises the Python interpreter to reduce the overhead.

### 2.6 Compilation on ARCHER

Py-ChemShell, GAMESS-UK 8.0, NWChem 6.6, GULP 4.5, and DL_POLY 4 were compiled on ARCHER phase 2 using the Cray compiler wrapper `ftn` with the Intel Composer XE 15.0.2.164 Fortran and C compilers. Additional tools and libraries used for the compiling include CMake 3.5.2, Anaconda Python3, Cray MPICH 7.2.6, and Intel MKL 15.0.2.164. The benchmark calculations in this report were performed on the ARCHER phase 2 supercomputer—a Cray XC30 MPP system. Each compute

node contains two 12-core Intel Ivy bridge processors and 64 GB of memory. The large nanoparticle containing over $1.6\times10^5$ atoms was computed on the large memory node which has 128 GB memory. The test cases have been chosen to represent realistic calculations that ChemShell users typically run on ARCHER.

## 3. Results and Discussion

### 3.1 GAMESS-UK/GULP task-farmed finite-difference gradients calculation

Numerical gradient calculations based on finite-difference methods are an ideal case for task-farming because they involve a large number of single-point energy evaluations that are independent from each other. With this method, single-point calculations are carried out with respect to each minor displacement ($1.0\times10^{-5}$ Å by default) that is made on every atomic centre along all *X*, *Y*, and *Z* directions. In a one-point finite difference method (1PFD) the energies are compared with that of the original structure and the gradient quantity on each centre is estimated by

$$\mathbf{g}_{\mathbf{1pfd}} = (E_{+} - E_0)/\Delta\mathbf{r}_{+}$$

or

$$\mathbf{g}_{\mathbf{1pfd}} = (E_{-} - E_0)/\Delta\mathbf{r}_{-}$$

depending on whether the displacements are positive or negative. In a two-point finite difference method (2PFD) the displacements are bidirectional (both + and −), and the energy differences are evaluated between the two displaced points:

$$\mathbf{g}_{\mathbf{2pfd}} = (E_{+} - E_{-})/\Delta\mathbf{r}$$

Therefore, if numerical gradients of $n_{\text{atoms}}$ centres are to be evaluated, the total numbers of independent single-point calculations of 1PFD and 2PFD are $n_{\text{atoms}}\times3+1$ and $n_{\text{atoms}}\times6$, respectively.

In Py-ChemShell, the functionality of performing a finite-difference gradients calculation has been implemented in a fully automatic manner rather than requiring the user to write a script. The user switches on the method by just specifying, for example, `gradients='2pfd'`, which stands for two-point finite-difference, in the task definition in a Py-ChemShell input script, for example

```
job1 = SP(theory=mgo_gamess_gulp, gradients='2pfd').run()
```

where `SP` is the Python class in Py-ChemShell to do single-point calculations.

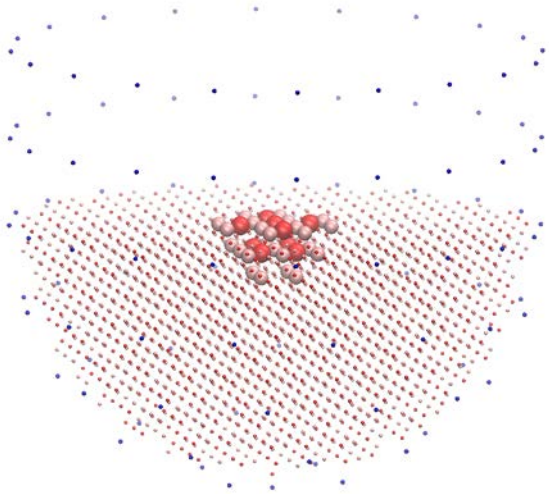

**Figure 3.** QM/MM embedded MgO system for the two-point finite-difference (2PFD) numerical gradients calculations: 2,370 atoms in total; 25 Mg (pink VDW representation) and 9 O (red VDW representation) atoms in the QM region; 2,326 MM atoms including 107 background point charges (blue dot representation). Graph produced with VMD 1.9.3.

To validate the code developed during this eCSE project, we have performed a series of 2PFD benchmark calculations. We set up an embedded QM/MM MgO cluster containing 2,370 atoms and 1,133 shell centres (the latter for the polarisable MM description). The numerical gradients of only the 34 QM atoms, namely 102 degrees of freedom or 204 QM/MM single-point calculations, are computed, which are enough to demonstrate the performance of Py-ChemShell's task-farming parallelisation. In each single-point calculation, a full relaxation of the shell centres is undertaken until converged (tolerance: $1.0\times10^{-4}$ a.u.) before the formal energy evaluation for estimating the gradients, therefore in total there are substantially more than 204 QM and MM calculations actually performed.

The first set of benchmark was performed at the theory level of GAMESS-UK (BLYP functional) and GULP, for which the Py-ChemShell interfaces were implemented before the current eCSE project. The task-farming scalability has been benchmarked against a range of $n_{\text{workgroups}}$, on a fixed number of 48 ARCHER phase 2 cores. Table 1 lists the wall time of these calculations with respect to the changing $n_{\text{workgroups}}$ and Figure 4 visualises the profiles. We have achieved 5.7-fold speedup compared to the non–task-farmed reference ($n_{\text{workgroups}}$ equals 1) by dividing the tasks into 24 workgroups, which exceeds our target of a 4-fold speedup and fulfils the objective of the project. The result has proven that the code development during the project results in a great advance over the original basic parallelisation scheme in Py-ChemShell. It is easy to find that the scaling profile of the task-farmed calculations is non-linear as $n_{\text{workgroups}}$ grows and it always has a minimum point. This is due to the increase of computational time for a single-point GAMESS-UK or GULP execution, when $n_{\text{workgroups}}$ becomes greater while $n_{\text{cores}}$ per workgroup decreases (see the red dashed line with circles in Figure 4).

It is noticeable that the total numbers of single-point calculations vary with the alteration of $n_{\text{workgroups}}$. This is because in each workgroup the shells relaxation begins with a new start and all the following points within the current workgroup are based on the relaxed shell positions and should take less iterations to converge.

**Table 1.** Benchmark of task-farmed two-point finite-difference (2PFD) numerical gradients calculation on QM/MM embedded MgO system against the number of workgroups on a fixed total number of 48 ARCHER cores (2 nodes): the total wall time ($t_{\text{total}}$ in h) averaged over workgroups and the average time for single-point (SP) calculations ($t_{\text{SP}}$ in sec). Only the QM region (102 degrees of freedom) is active for the 2PFD evaluation. System size: 2,370 atoms, of which 34 are QM, plus 1,133 shell centres; QM theory: GAMESS-UK at BLYP level with customised basis set and pseudo potential (effective core potential or ECP); MM theory: GULP. Shell relaxation tolerance: $1.0\times10^{-4}$ a.u.

| $n_{\text{workgroups}}$ | 1 | 2 | 4 | 8 | 16 | 24 | 48 |
|---|---|---|---|---|---|---|---|
| $n_{\text{cores}}$ | 48 | 48 | 48 | 48 | 48 | 48 | 48 |
| $n_{\text{cores/workgroup}}$ | 48 | 24 | 12 | 6 | 3 | 2 | 1 |
| $n_{\text{SP}}$ | 412 | 416 | 424 | 440 | 472 | 504 | 600 |
| $t_{\text{SP}}$/s (GAMESS-UK) | 2.9 | 4.2 | 5.7 | 9.8 | 18.0 | 25.8 | 57.6 |
| $t_{\text{SP}}$/s (GULP) | 2.1 | 2.5 | 2.6 | 3.0 | 4.1 | 5.0 | 8.1 |
| $t_{\text{total}}$/h | 1.48 | 0.85 | 0.50 | 0.33 | 0.27 | 0.26 | 0.32 |
| Speedup vs. 1 workgroup | 1.0 | 1.7 | 2.9 | 4.5 | 5.4 | 5.7 | 4.6 |

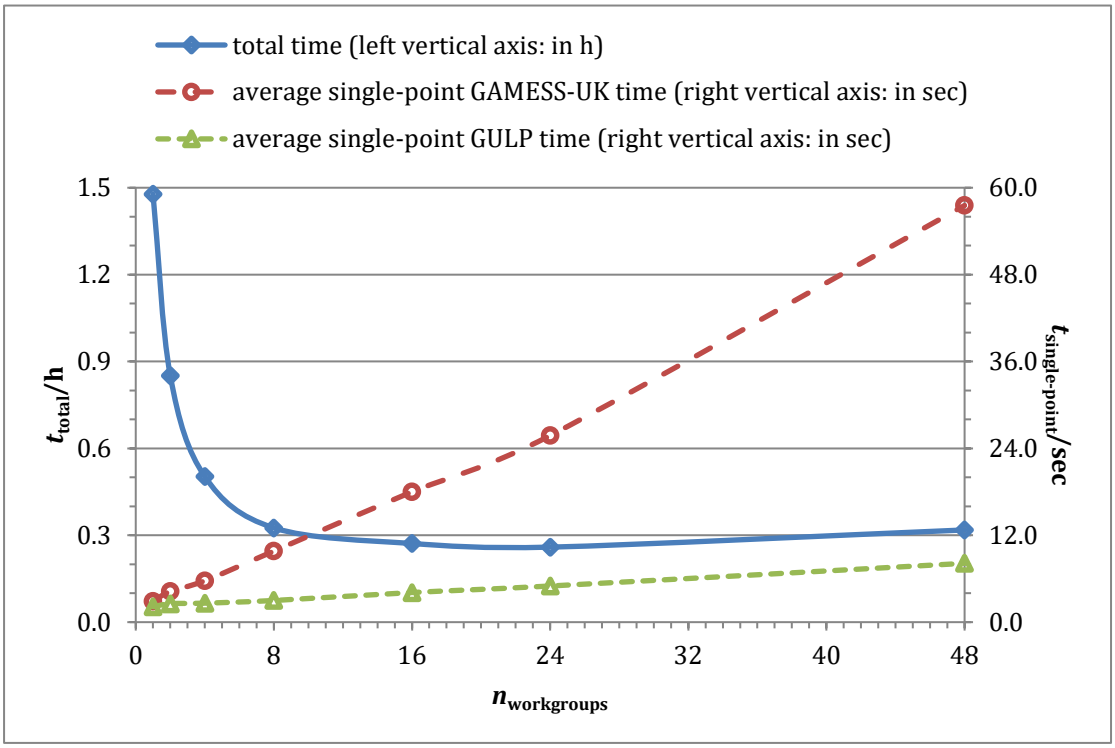


**Figure 4.** Benchmark of task-farmed two-point finite-difference (2PFD) numerical gradients calculation on QM/MM embedded MgO system against the number of workgroups on a fixed total number of 48 ARCHER cores (2 nodes): the total wall time ($t_{\text{total}}$ in h) averaged over workgroups and the average time for single-point (SP) calculations ($t_{\text{SP}}$ in s). Only the QM region (102 degrees of freedom) is active for the 2PFD evaluation. System size: 2,370 atoms, of which 34 are QM, plus 1,133 shell centres; QM theory: GAMESS-UK at BLYP level with customised basis set and pseudo potential (effective core potential or ECP); MM theory: GULP. Shell relaxation tolerance: 1.0E−4 a.u.

**Table 2.** Benchmark of task-farmed two-point finite-difference (2PFD) numerical gradients calculation on QM/MM embedded MgO system against the number of cores and workgroups: each workgroup consists of 48 ARCHER cores. The total wall time ($t_{total}$ in h) averaged over workgroups and the average time for single-point (SP) calculations ($t_{SP}$ in s) are with respect to the left and right axes, respectively. Only the QM region (102 degrees of freedom) is active for the 2PFD evaluation. System size: 2,370 atoms, of which 34 are QM, plus 1,133 shell centres; QM theory: GAMESS-UK at BLYP level with customised basis set and pseudo potential (effective core potential or ECP); MM theory: GULP. Shell relaxation tolerance: $1.0\times10^{-4}$ a.u.

| $n_{workgroups}$ | 1 | 2 | 4 | 8 |
|---|---|---|---|---|
| $n_{cores}$ | 48 | 96 | 192 | 384 |
| $n_{cores/workgroup}$ | 48 | 48 | 48 | 48 |
| $n_{SP}$ | 412 | 416 | 424 | 440 |
| $t_{SP}$/s (GAMESS-UK) | 2.9 | 2.6 | 2.8 | 3.1 |
| $t_{SP}$/s (GULP) | 2.1 | 1.9 | 2.0 | 2.0 |
| $t_{total}$/h | 1.48 | 0.72 | 0.37 | 0.20 |
| Parallel scaling | 1.00 | 2.05 | 4.04 | 7.26 |

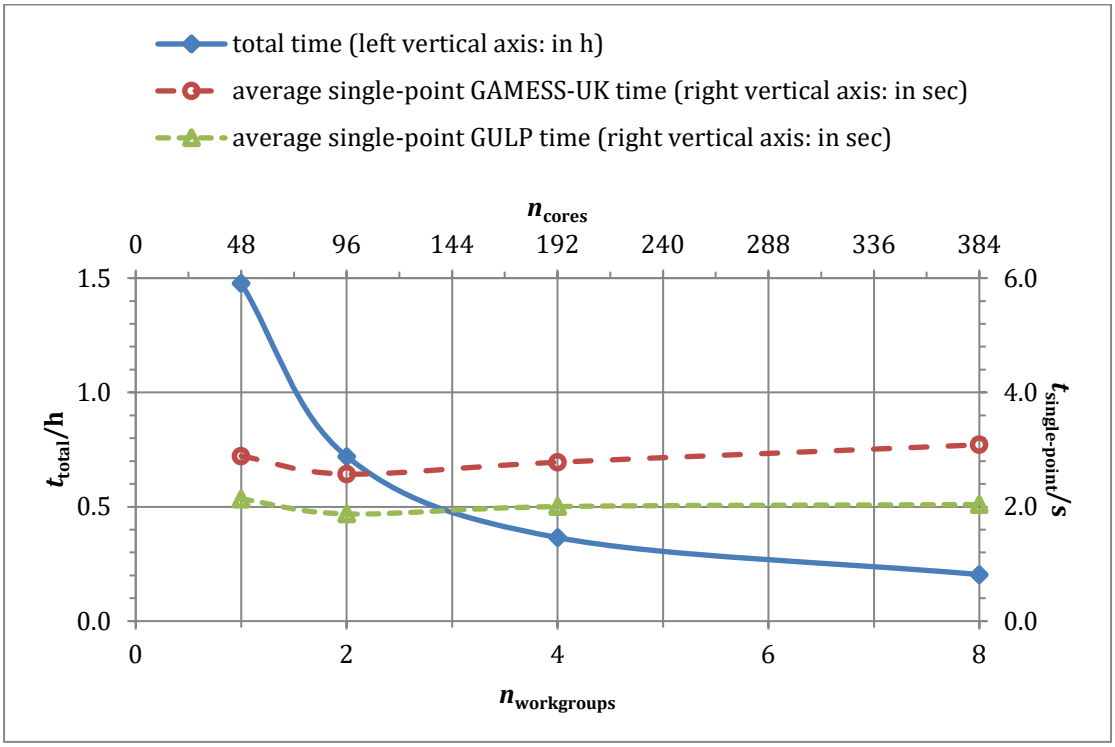


**Figure 5.** Benchmark of task-farmed two-point finite-difference (2PFD) numerical gradients calculation on QM/MM embedded MgO system against the number of cores and workgroups: each workgroup consists of 48 ARCHER cores. The total wall time ($t_{total}$ in h) averaged over workgroups and the average time for single-point (SP) calculations ($t_{SP}$ in s) are with respect to the left and right axes, respectively. Only the QM region (102 freedoms) is active for the 2PFD evaluation. System size: 2,370 atoms, of which 34 are QM, plus 1,133 shell centres; QM theory: GAMESS-UK at BLYP level with customised basis set and pseudo potential (effective core potential or ECP); MM theory: GULP. Shell relaxation tolerance: $1.0\times10^{-4}$ a.u.

To further demonstrate the power of task-farming parallelisation, we also carried out similar 2PFD calculations with increasing $n_{workgroups}$ and total $n_{cores}$ while keeping

$n_{cores}$ per workgroup, which is 48 here, unchanged (see Table 2 and Figure 5). The speedup factor obtained in this way is 7.26 by employing 384 ARCHER cores (16 nodes) and 8 workgroups. In this case the time used for a single-point GAMESS-UK or GULP execution remains the same level because the available $n_{cores}$ per workgroup is a constant number. Note also that super-linear scaling is possible because of subtle differences in the shell relaxation starting points as discussed above.

### 3.2 NWChem/GULP task-farmed finite-difference gradients calculation

**Table 3.** Benchmark of task-farmed two-point finite-difference (2PFD) numerical gradients calculation on QM/MM embedded MgO system against the number of workgroups on a fixed total number of 48 ARCHER cores (2 nodes): the total wall time ($t_{total}$ in h) averaged over workgroups and the average time for single-point (SP) calculations ($t_{SP}$ in s). Only the QM region (102 degrees of freedom) is active for the 2PFD evaluation. System size: 2,370 atoms, of which 34 are QM, plus 1,133 shell centres; QM theory: NWChem at BLYP level with customised basis set and pseudo potential (effective core potential or ECP); MM theory: GULP. Shell relaxation tolerance: $1.0\times10^{-4}$ a.u.

| $n_{workgroups}$ | 1 | 2 | 4 | 8 | 16 | 24 | 48 |
|---|---|---|---|---|---|---|---|
| $n_{cores}$ | 48 | 48 | 48 | 48 | 48 | 48 | 48 |
| $n_{cores/workgroup}$ | 48 | 24 | 12 | 6 | 3 | 2 | 1 |
| $n_{SP}$ | 412 | 416 | 424 | 440 | 472 | 504 | 600 |
| $t_{SP}$/s (NWChem) | 34.1 | 48.4 | 83.8 | 159.8 | 273.8 | 370.8 | 608.6 |
| $t_{SP}$/s (GULP) | 2.6 | 2.6 | 2.4 | 2.9 | 3.8 | 5.0 | 8.3 |
| $t_{total}$/h | 4.68 | 3.23 | 2.73 | 2.61 | 2.54 | 2.43 | 2.58 |
| Speedup vs. 1 workgroup | 1.0 | 1.4 | 1.7 | 1.8 | 1.8 | 1.9 | 1.8 |

The second set of benchmark calculations was carried out at the NWChem/GULP level, for which NWChem was GA-driven. The specifications of density functional, basis set, effective core potential (ECP), shell centres, $n_{workgroups}$, $n_{cores}$, and $n_{cores}$ per workgroup are the same as in the previous GAMESS-UK/GULP benchmark to conduct a fair comparison. However, the achieved speedup is significantly less compared to GAMESS-UK/GULP (1.9 vs. 5.7, see Table 3 and Figure 6). This can be explained by the overwhelmingly dominant time used for NWChem and the several-times slower DFT energy convergence with NWChem for this specific MgO cluster. Then the overall task-farming scaling becomes determined by the rapidly increasing time of the QM calculation. It is beyond the scope of this project to investigate why the convergence behaviour of NWChem is so much poorer than GAMESS-UK for this system, but we will look into this closely in future work.

Nevertheless the total time for 2PFD calculation can be greatly shortened by utilising more cores because the task-farming parallelisation in Py-ChemShell is extremely scalable. And in this case the scaling for this system is just as good for NWChem as for GAMESS-UK: with 8 workgroups on 384 ARCHER cores the calculation a speedup of 7.33 times was observed (see Table 4 and Figure 7).

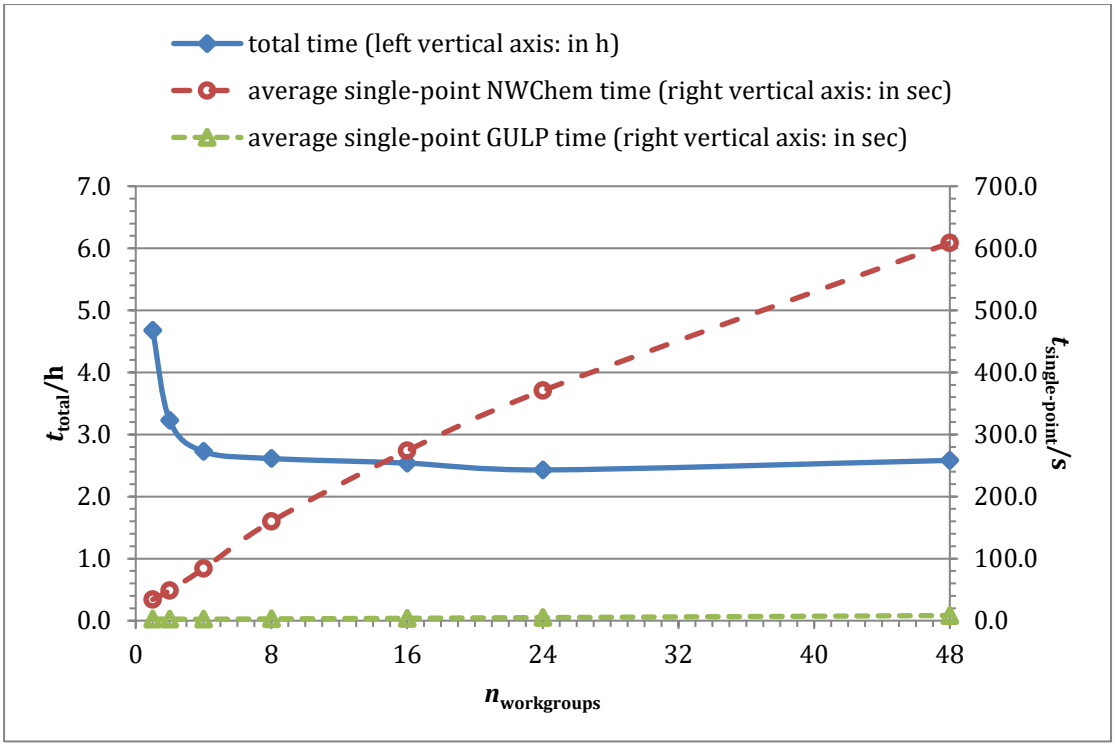


**Figure 6.** Benchmark of task-farmed two-point finite-difference (2PFD) numerical gradients calculation on QM/MM embedded MgO system against the number of workgroups on a fixed total number of 48 ARCHER cores (2 nodes): the total wall time ($t_{total}$ in h) averaged over workgroups and the average time for single-point (SP) calculations ($t_{SP}$ in s). Only the QM region (102 freedoms) is active for the 2PFD evaluation. System size: 2,370 atoms, of which 34 are QM, plus 1,133 shell centres; QM theory: NWChem at BLYP level with customised basis set and pseudo potential (effective core potential or ECP); MM theory: GULP. Shell relaxation tolerance: $1.0\times10^{-4}$ a.u.

**Table 4.** Benchmark of task-farmed two-point finite-difference (2PFD) numerical gradients calculation on QM/MM embedded MgO system against the number of cores and workgroups: each workgroup consists of 48 ARCHER cores. The total wall time ($t_{total}$ in h) averaged over workgroups and the average time for single-point (SP) calculations ($t_{SP}$ in s) are with respect to the left and right axes, respectively. Only the QM region (102 degrees of freedoms) is active for the 2PFD evaluation. System size: 2,370 atoms, of which 34 are QM, plus 1,133 shell centres; QM theory: NWChem at BLYP level with customised basis set and pseudo potential (effective core potential or ECP); MM theory: GULP. Shell relaxation tolerance: $1.0\times10^{-4}$ a.u.

| $n_{workgroups}$ | 1 | 2 | 4 | 8 |
|---|---|---|---|---|
| $n_{cores}$ | 48 | 96 | 192 | 384 |
| $n_{cores/workgroup}$ | 48 | 48 | 48 | 48 |
| $n_{SP}$ | 412 | 416 | 424 | 440 |
| $t_{SP}$/s (NWChem) | 34.1 | 33.1 | 33.5 | 34.3 |
| $t_{SP}$/s (GULP) | 2.6 | 2.7 | 2.8 | 2.8 |
| $t_{total}$/h | 4.68 | 2.36 | 1.22 | 0.64 |
| Parallel scaling | 1.00 | 1.99 | 3.82 | 7.33 |

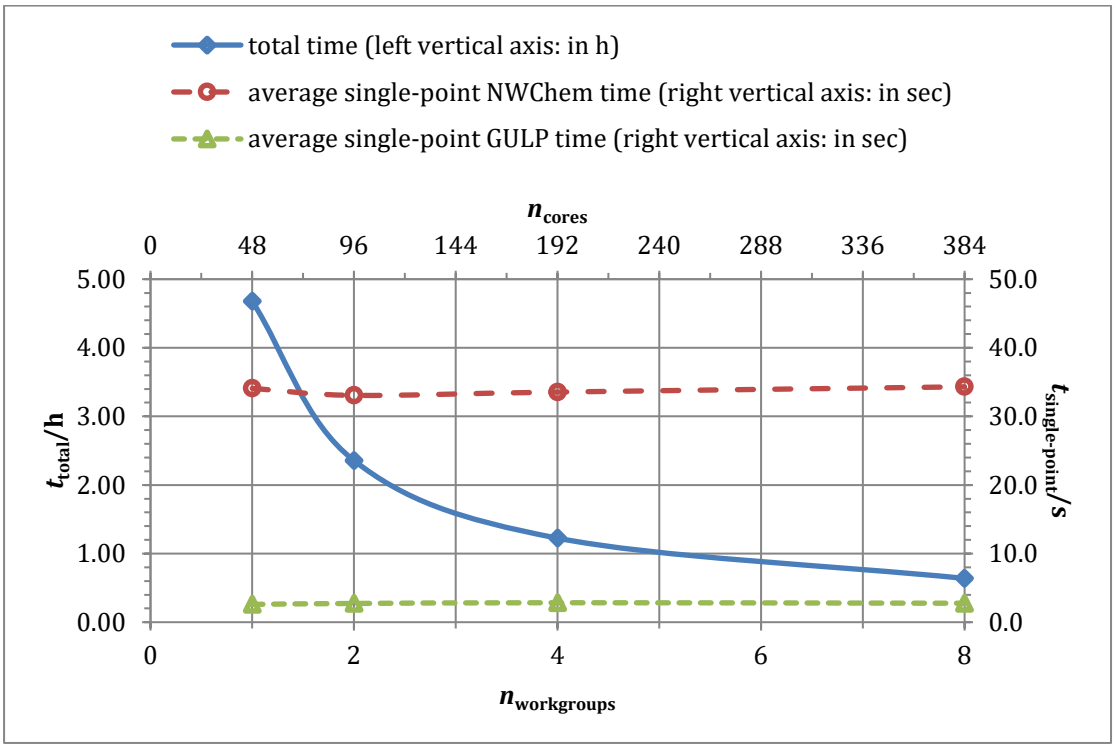


**Figure 7.** Benchmark of task-farmed two-point finite-difference (2PFD) numerical gradients calculation on QM/MM embedded MgO system against the number of cores and workgroups: each workgroup consists of 48 ARCHER cores. The total wall time ($t_{total}$ in h) averaged over workgroups and the average time for single-point (SP) calculations ($t_{SP}$ in s) are with respect to the left and right axes, respectively. Only the QM region (102 degrees of freedom) is active for the 2PFD evaluation. System size: 2,370 atoms, of which 34 are QM, plus 1,133 shell centres; QM theory: NWChem at BLYP level with customised basis set and pseudo potential (effective core potential or ECP); MM theory: GULP. Shell relaxation tolerance: $1.0\times10^{-4}$ a.u.

## 3.3 NWChem/DL_POLY 4 task-farmed finite-difference gradients calculation

**Table 5.** Benchmark of task-farmed two-point finite-difference (2PFD) numerical gradients calculation on QM/MM embedded MgO system against the number of workgroups on a fixed total number of 48 ARCHER cores (2 nodes): the total wall time ($t_{total}$ in h) averaged over workgroups and the average time for single-point (SP) calculations ($t_{SP}$ in s). Only the QM region (102 degrees of freedom) is active for the 2PFD evaluation. System size: 2,370 atoms, of which 34 are QM, without shell centres; QM theory: NWChem at BLYP level with customised basis set and pseudo potential (effective core potential or ECP); MM theory: DL_POLY 4.

| $n_{workgroups}$ | 1 | 2 | 4 | 8 | 16 | 24 | 48 |
|---|---|---|---|---|---|---|---|
| $n_{cores}$ | 48 | 48 | 48 | 48 | 48 | 48 | 48 |
| $n_{cores/workgroup}$ | 48 | 24 | 12 | 6 | 3 | 2 | 1 |
| $n_{cores}$ (DL_POLY 4) | 8 | 8 | 8 | 4 | 2 | 2 | 1 |
| $n_{SP}$ | 205 | 206 | 208 | 212 | 220 | 228 | 252 |
| $t_{SP}$/s (NWChem) | 87.8 | 150.7 | 271.5 | 495.7 | 857.6 | 1185.6 | 1768.8 |
| $t_{SP}$/s (DL_POLY 4) | 0.4 | 0.4 | 0.4 | 0.4 | 0.4 | 0.5 | 0.5 |
| $t_{total}$/h | 5.19 | 4.47 | 4.12 | 3.82 | 3.68 | 3.73 | 3.69 |
| Speedup vs. 1 workgroup | 1.0 | 1.2 | 1.3 | 1.4 | 1.4 | 1.4 | 1.4 |

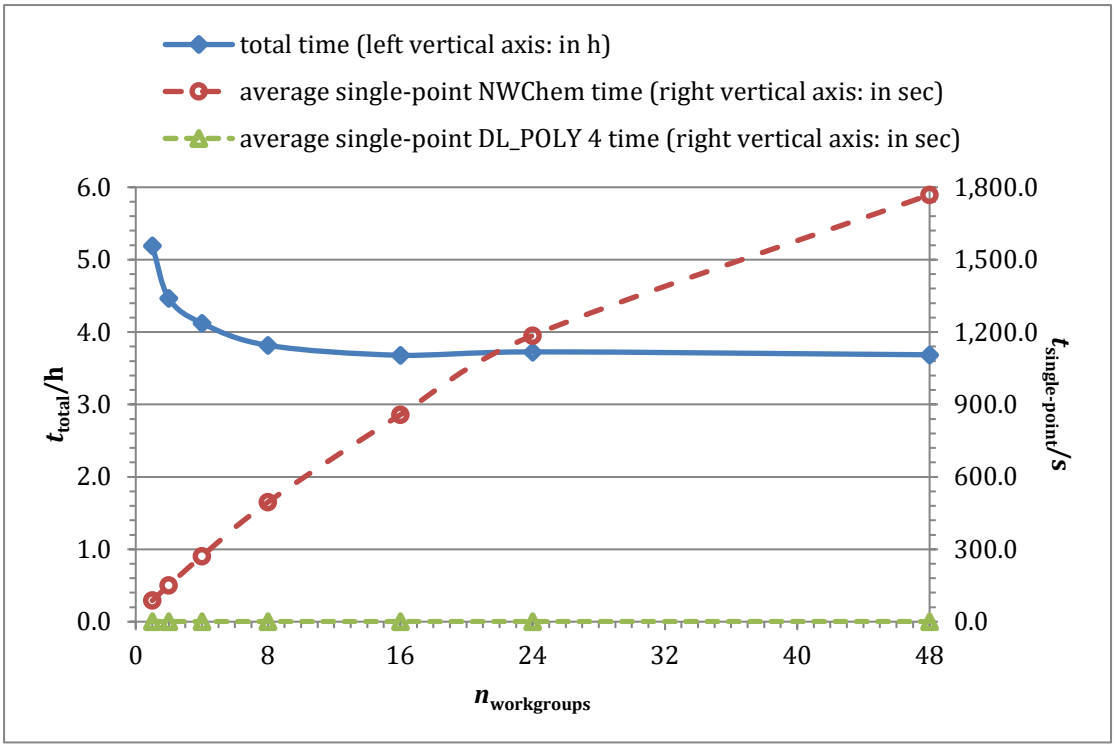


**Figure 8.** Benchmark of task-farmed two-point finite-difference (2PFD) numerical gradients calculation on QM/MM embedded MgO system against the number of workgroups on a fixed total number of 48 ARCHER cores (2 nodes): the total wall time ($t_{total}$ in h) averaged over workgroups and the average time for single-point (SP) calculations ($t_{SP}$ in s). Only the QM region (102 degrees of freedom) is active for the 2PFD evaluation. System size: 2,370 atoms, of which 34 are QM, without shell centres; QM theory: NWChem at BLYP level with customised basis set and pseudo potential (effective core potential or ECP); MM theory: DL_POLY 4.

With the newly implemented interface to DL_POLY 4, we have been for the first time able to benchmark task-farmed NWChem/DL_POLY 4 calculations. However, due to the current lack of support for shells in the DL_POLY interface, the calculations were carried out without shell centres. Despite this, the conclusion is similar to the NWChem/GULP benchmark, i.e. that due to the very slow DFT energy convergence the speedup gained by task-farming the 2PFD gradients on a fixed number of cores (see Table 5 and Figure 8) is relatively small compared to GAMESS-UK/GULP, while the parallel scaling is nevertheless very good when increasing $n_{workgroups}$ and $n_{cores}$ at the same time (see Table 6 and Figure 9). It should be pointed out that the number of cores deployed for DL_POLY 4 in the benchmark was a constant number (see Table 5 and Figure 8), because the current release of DL_POLY 4 is limited to 8 processes for non-periodic structures. Nevertheless, this does not affect the overall scaling behaviour for this benchmark calculation.

**Table 6.** Benchmark of task-farmed two-point finite-difference (2PFD) numerical gradients calculation on QM/MM embedded MgO system against the number of workgroups on a fixed total number of 48 ARCHER cores (2 nodes): the total wall time ($t_{total}$ in h) averaged over workgroups and the average time for single-point (SP) calculations ($t_{SP}$ in s). Only the QM region (102 degrees of freedom) is active for the 2PFD evaluation. System size: 2,370 atoms, of which 34 are QM, without shell centres; QM theory: NWChem at BLYP level with customised basis set and pseudo potential (effective core potential or ECP); MM theory: DL_POLY 4.

| $n_{workgroups}$ | 1 | 2 | 4 | 8 |
|---|---|---|---|---|
| $n_{cores}$ | 48 | 96 | 192 | 384 |
| $n_{cores/workgroup}$ | 48 | 48 | 48 | 48 |
| $n_{SP}$ | 205 | 206 | 208 | 212 |
| $n_{cores}$ (DL_POLY 4) | 8 | 8 | 8 | 8 |
| $t_{SP}$/s (NWChem) | 87.8 | 86.6 | 87.7 | 86.2 |
| $t_{SP}$/s (DL_POLY 4) | 0.4 | 0.4 | 0.4 | 0.4 |
| $t_{total}$/h | 5.19 | 2.56 | 1.35 | 0.69 |
| Speedup | 1.00 | 2.03 | 3.84 | 7.53 |

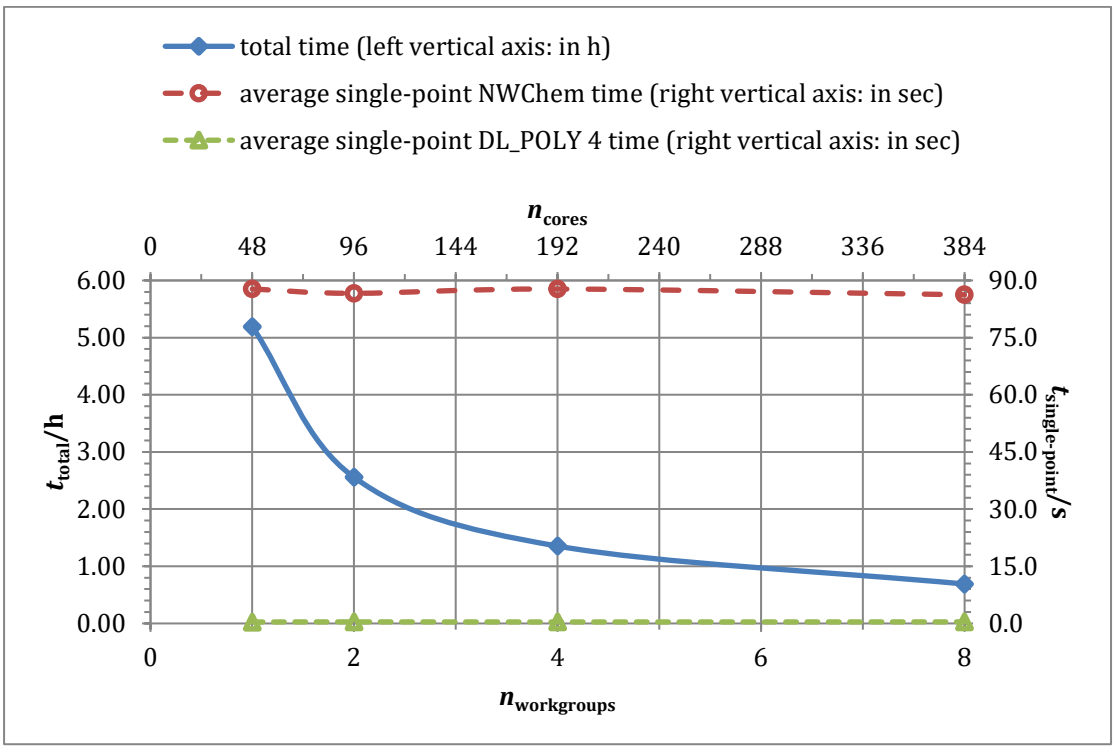


**Figure 9.** Benchmark of task-farmed two-point finite-difference (2PFD) numerical gradients calculation on QM/MM embedded MgO system against the number of workgroups on a fixed total number of 48 ARCHER cores (2 nodes): the total wall time ($t_{total}$ in h) averaged over workgroups and the average time for single-point (SP) calculations ($t_{SP}$ in s). Only the QM region (102 degrees of freedom) is active for the 2PFD evaluation. System size: 2,370 atoms, of which 34 are QM, without shell centres; QM theory: NWChem at BLYP level with customised basis set and pseudo potential (effective core potential or ECP); MM theory: DL_POLY 4.

### 3.4 NWChem/DL_POLY 4 modelling of $ZrO_2$ nanoparticle

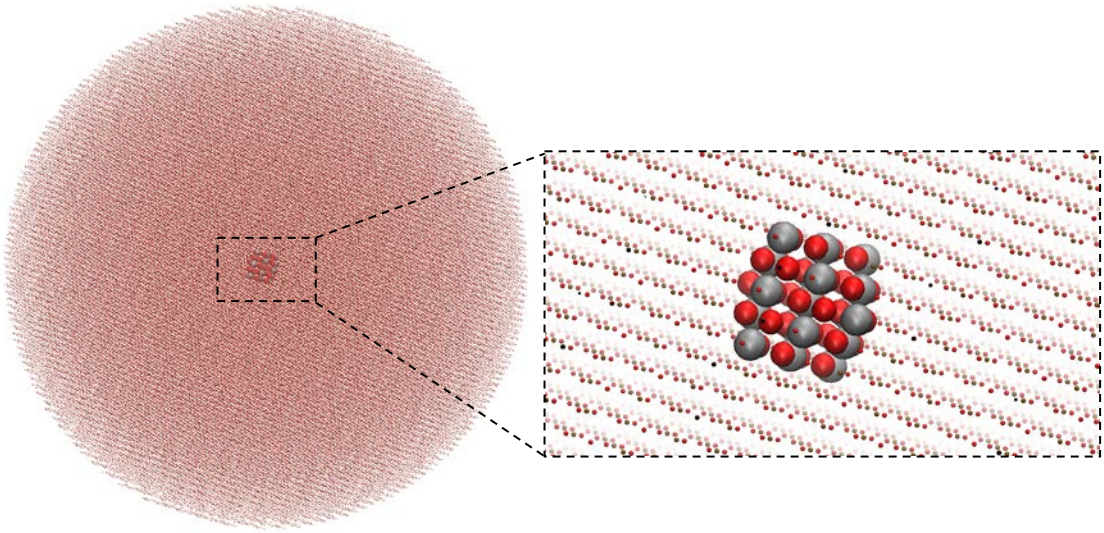

**Figure 10.** $ZrO_2$ nanoparticle: 162,994 atoms including 19 Zr (grey VDW representation) and 32 O (red VDW representation) atoms in the QM region. Graphs generated using VMD 1.9.3.

Due to the embedded cluster model we use for QM/MM solid state calculations, in ChemShell we do not impose any point group symmetry or periodic boundary condition (PBC) on a chemical structure. However, the current release of DL_POLY 4 is only able to optimise its own memory usage for periodic systems. As a result, a large amount of memory is required when we calculate a bulky cluster comprising over a hundred thousand atoms. Considering that DL_POLY 4 needs to allocate an N×N array, 162,994 atoms, namely 26,567,044,036 32-bit integers, requires 101,345.23 MB (~98.97 GB) space in memory. Fortunately, in the ARCHER phase 2 system there are 64 large-memory nodes, possessing 128 GB memory per node, which is adequate for handling our largest $ZrO_2$ setup. The user should use the Py-ChemShell executable script's argument `--node largemem` or `-N largemem` on ARCHER to submit the job to the large-memory queue. On the other hand, according to the memory limit available on the ARCHER machine, we can deduce that the largest non-periodic system we can calculate at present using DL_POLY 4 is of about $1.85\times10^5$ atoms. It is beyond the scope of this project to optimise DL_POLY's memory use but we will work with the DL_POLY developers to remove this limitation in future work.

By contrast, NWChem uses much less memory for the QM region, which contains 51 atoms (19 Zr and 32 O). The number of background point charges ($n_{bqs}$) 162,943 is as many as the number of MM atoms but these have a low memory requirement in NWChem. However, there is an internal parameter limiting the number of background point charges in NWChem: the default maximum $n_{bqs}$ is only 25,000, which is inadequate for our embedding system. To overcome this limit, we let Py-

ChemShell detect $n_{bqs}$ for the system to compute and if the number is greater than 25,000 automatically insert a line

```
set bq:max_nbq 200000
```

into the ChemShell-generated NWChem input script to raise the default limit. The new limit parameter is the round number that is larger than and closest to the actual $n_{bqs}$: for example, it is 200,000 for our system of over $1.6\times10^5$ point charge centres. Another potential issue is the analytical gradients calculation with forces on point-charge centres for such a system of a great number of point charges, because in this case NWChem needs much more scratch space than hard-coded to write out temporary data. To prevent NWChem from crashing due to this reason (error message: `HF1D: Insufficient scratch space`), we have to modify the NWChem source code file `src/NWints/api/exactd_mem.F` manually to increase the value for the variable `memdum` and recompile NWChem. [9]

To assess the performance, we have carried out single-point energy evaluations, at the level of NWChem Hartree-Fock (HF) and DL_POLY 4 molecular mechanics, for the $ZrO_2$ nanoparticle of radius 75.0 Å containing 162,994 atoms on a range of $n_{cores}$. The resulting time used for the calculations are listed in Table 7, which shows that the Py-ChemShell code scales very well for this system. The time elapsed for DL_POLY 4 energy evaluation slightly fluctuates but within a reasonable extent for a production environment.

**Table 7.** Single-point energy evaluations of QM/MM embedded $ZrO_2$ nanoparticle of radius 75.0 Å (162,994 atoms) computed on various number of "largemem" node cores. The number of processes for DL_POLY 4 has been kept as a constant 8. Theory level: NWChem Hartree-Fock and DL_POLY 4 force field.

| $n_{cores}$ | $n_{nodes}$ | $n_{cores}$ (DL_POLY 4) | $t_{NWChem}$/h | $t_{DL\_POLY}$/s | $t_{total}$/h | speedup |
|---|---|---|---|---|---|---|
| 24 | 1 | 8 | 11.99 | 50.1 | 12.21 | 1.00 |
| 48 | 2 | 8 | 6.15 | 49.3 | 6.38 | 1.91 |
| 96 | 4 | 8 | 3.26 | 58.8 | 3.47 | 3.52 |
| 192 | 8 | 8 | 1.71 | 50.8 | 1.92 | 6.36 |

## 4. Conclusion and Outlook

The task-farming parallelism introduced into Py-ChemShell has resulted in significant performance and scalability improvements for tasks consisting of multiple independent energy evaluations, which will greatly improve the performance on ARCHER of numerical gradient and Hessian calculations, nudged elastic band (NEB) optimisations, population-based optimisations and so on. The benchmarked finite-difference gradients calculations at various QM/MM theory levels show that the performance gains in terms of the speedup factor can be as great as 5.7, which is in excess of the target in the original proposal of this eCSE project. More objectives achieved during the project include:

- An interface to DL_POLY 4 which enables calculations of large nanomaterials of over 160,000 atoms, well beyond the capabilities of Tcl-ChemShell.
- Support for task-farming with the Global Arrays (GA) library required for calculations with NWChem.
- Parallelisation of the embedded cluster setup routines.
- The functionality of charge fitting that is required for performing multi-QM-region calculations.

The achievements during this project and the data in this technical report will be prepared for publication in the peer-reviewed literature. An initial alpha release of Py-ChemShell was made in December 2017 including the code developed for this project, with source available on request for testing purposes. A first formal release is scheduled for Spring 2018 and a Py-ChemShell module will then be installed on ARCHER phase 2 machine for general use.

The main aim of follow-up work will be to exploit the task-farming approach on ARCHER phase 2 and its successor for scientific applications. Ongoing collaborations with other scientists will use task-farmed parallel QM/MM calculations for the study of nanomaterials and heterogeneous catalysis, on systems similar to those in the benchmark calculations in this report. The task-farmed approach will also be useful in other areas where large-scale calculations are required, such as biomolecular modelling. And it could be extended to other types of chemical calculations, such as multiple-trajectory molecular dynamics simulations.

Further technical developments will also be required to maximise performance of the code, for example:

- The linear algebra library ScaLAPACK was applied in our benchmark calculations with NWChem but found to be less efficient than the intrinsic GA linear algebra solvers. However the latter are not compatible with the task-farming framework. We will work with the NWChem developers to settle this

issue. We will also carry out benchmark calculations to assess the performance of NWChem driven with different linear algebra libraries, including ELPA, on the ARCHER phase 2 machine.

- The current ChemShell interface to DL_POLY 4 does not yet support shells relaxation in the QM/MM context, and we will carry out the implementation work that is necessary for this. We also find that DL_POLY 4 has limitations in employing more processes and it quickly reaches the memory limit with non-periodic systems. Further technical effort on this issue, working together with the developers of DL_POLY, will obviously benefit the material science community by allowing them to compute even larger QM/MM chemical systems.

We will also incorporate the prototype parallel cluster setup code in CONSTRUCT directly into Py-ChemShell once validation is complete.

**Acknowledgements**

This work was funded under the embedded CSE programme of the ARCHER UK National Supercomputing Service (http://www.archer.ac.uk).

## References


[1] Sherwood P, de Vries AH, Guest MF, Schreckenbach G, Catlow CRA, French SA, Sokol AA, Bromley ST, Thiel W, Turner AJ, Billeter S, Terstegen F, Thiel S, Kendrick J, Rogers SC, Casci J, Watson M, King F, Karlsen E, Sjøvoll M, Fahmi A, Schäfer A, Lennartz C (2003) J Mol Struct (Theochem) 632:1–28

[2] Sokol AA, Bromley ST, French SA, Catlow CRA, Sherwood P (2004) Int J Quantum Chem 99:695–712

[3] van Dam HJJ, Guest MF, Sherwood P, Thomas JMH, van Lenthe JH, van Lingen JNJ, Bailey CL, Bush IJ (2006) J Mol Struct (Theochem) 771:33–41

[4] Keal TW, Sherwood P, Dutta G, Sokol AA, Catlow CRA (2011) Proc R Soc A 467: 1900–1924

[5] Henkelman G, Jónsson H (2000) J Chem Phys 113:9978–9985

[6] Kästner J, Carr JM, Keal TW, Thiel W, Wander A, Sherwood P (2009) J Phys Chem A 113:11856–11865

[7] Lu Y, Keal TW, "DL_PY2F: a library for interfacing Python with modern Fortran" (in preparation). Not to be confused with the F2PY project for interfacing Python with legacy Fortran libraries.

[8] Todorov IT, Smith W, Trachenko K, Dove MT (2006) J Mater Chem 16:1911–1918

[9] For details, please refer to the discussion at: http://www.nwchem-sw.org/index.php/Special:AWCforum/st/id2048/HF1D%3A_Insufficient_scratch_space.html