Dynamic Load Balancing in CP2K

Pradeep Shivadasan

August 19, 2014

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2014

Abstract

CP2K is a widely used atomistic simulation package. Unlike other similar packages, CP2K uses the Quickstep algorithm for molecular dynamics simulation. Quickstep is an improvement over plane-wave implementation of DFT and uses a dual-basis approach to store data in different representation $-$ wave functions are stored as sparse matrix and electronic density stored on distributed regular grids. A frequent conversion between the different representation is needed to find the ground state energy. A load balancing module is available in CP2K which optimizes the mapping of tasks to processes during the conversion from matrix to regular grids format. However, the module allocates $O(P^2)$ memory, where P is the number of MPI processes, and uses a serial algorithm to optimize load on all processes. The high memory requirement and serial load balancing task limits the scalability of the algorithm to high number of processors. This document describes a solution to the high memory requirement problem and shows the issues in parallelizing the serial load balancing task.

Contents

List of Figures

List of Algorithms

Acknowledgements

If I have seen further it is by standing
on the shoulders of giants

Newton, 1675

I would like to express my deepest gratitude to my supervisor, Iain Bethune (EPCC), for his excellent guidance, patience, and encouragement in carrying out this project work.

1 Introduction

CP2K [1] is an open-source tool for atomistic simulation available under the GNU General Public License (GPL). It provides a broad range of models and simulation methodologies for large and condensed phase systems. The software is written in Fortran95 and parallelized for distributed memory architectures using MPI augmented with OpenMP for shared memory architectures. It uses various libraries like dense linear algebra packages (BLAS, LAPACK, ScaLAPACK); fast fourier transforms (FFTW); specialized chemical libraries like electron repulsive integrals (libint) and exchange correlation functionals (libxc) to decrease the complexity and enchance the efficiency and robustness of the code.

CP2K is mainly used to explore properties of complex systems. The methods available in CP2K to explore the potential energy surface are *Stationary Points*- to optimize atomic positions and cell vectors based on various algorithms like cubically scaling methods, Hessian and linear scaling methods; *Molecular Dynamics*- DFT based molecular dynamics simulation to simulate atoms and molecules in a system; *Monte Carlo*- alternative to MD for sampling purposes; *Ehrenfest dynamics*- to study the time dependent evolution of electric fields in a system and the response of the electronic structure to perturbation; and *Energy and force methods*- to model materials using *Classic Force Fields* and *Electronics Structure Methods*. See [2] for examples of outstanding science performed with CP2K.

CP2K uses the Quickstep [3] algorithm, an efficient implementation of density function theory (DFT) based molecular simulations method. It uses a dual-basis approach where data is stored in two distinct representations - wave-functions stored in a sparse matrix and electronic density stored on distributed regular grids. Sparse matrix storage uses less memory and is faster to process while distributed regular grid helps in computing Hartree potential efficiently. However, the disadvantage of this representation is that to find the ground state energy of the system, the data needs to be converted between these two representations in every iteration of the Self-Consistent field (SCF) loop.

Figure 1 shows the conversion between various representations. The conversion from matrix to regular grids is called Collocation and the reverse conversion is called Integration.

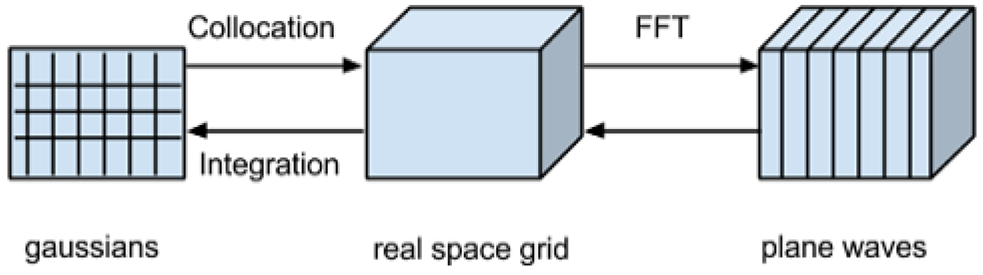


Figure 1: Conversion between representations.

As mentioned earlier, the conversion between various representation is carried out in every iteration of the SCF loop hence it needs to be very efficient. A number of performance improvements, including matrix to regular grid conversion; FFT methods; load balancing of tasks transferred from real-space to planewave representation, were made to improve the performance of the Quickstep algorithm in dCSE [4] project.

The scope of this project is to improve the scalability of the existing load balancing algorithm by reducing its memory requirements and parallelizing the algorithm.

The rest of the document describes the efforts to improve the existing load balancing module. Section 2 provides an overview of the tools and technologies used to analyze and develop an improved solution for the problem. Section 3 describes the data-structures and algorithms used to implement the load balancing module. Section 4 lists the limitations of the existing implementation followed by a discussion of the new improved solutions in Section 5. Section 6 summarizes the lessons learned during the project development. Finally, Section 7 summarizes the work carried out during the project.

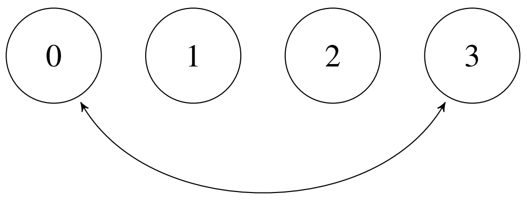
2 Background

CP2K version 2.4.0 was used for the development of the solution. The solution was developed on ARCHER [5] (Advanced Research Computing High End Resource), the UK National Supercomputing Facility. The ARCHER facility is based around a Cray XC30 supercomputer with 1.56 petaflops of theoretical peak performance. It consists of 3,008 nodes with each node having two 12-core 2.7 GHz Ivy Bridge multicore processors for a total of 72,192 cores. Each node contains 64 GB of DDR3 main memory, giving 2.6 GB of main memory per core. The nodes are interconnected via an Aries Network Interface Card. See [6] for full details of the system.

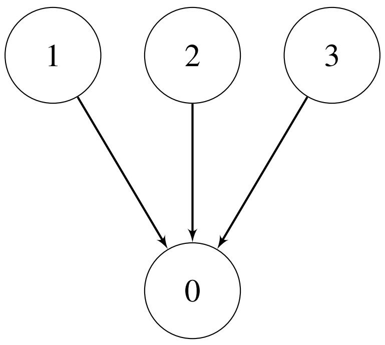
A brief overview of the tools and techniques used to develop the solution follows.

2.1 MPI

The Message Passing Interface (MPI) [7] is a message passing library used to program parallel computers mostly distributed memory system. MPI is a SPMD (Single Program Multiple Data) style of parallel programming where data is communicated between processes using messages. MPI provides two modes of communication: *a*) Point-to-point; and *b*) Collective . Point-to-point operation involves message passing between two different processes. One performs a send operation and the other process performs a matching receive operation. A collective communication involves all processes within the system. Collective operations are used for synchronization, data-movement (broadcast, scatter/gather, all to all), and reduction operations (min, max, add, multiply, etc). Figure 2 shows modes of MPI communication diagrammatically. Parallel algorithms in CP2K are based on message passing using MPI.



(a) Point-to-point communication



(b) Collective communication

Figure 2: MPI communication modes. a) is a point-to-point communication between two processes. b) is a collective communication where the final result is available on a single process. Another form of collective communication is available where the result is made available on all processes in the group.

2.2 Algorithm analysis framework

2.2.1 Levels of parallelization

A modern processor provides support for different levels of parallelism at instruction level, data-level, and task level. The tools and techniques to exploit parallelism are different at different levels. The next sections provide a brief overview of parallelism support in a modern processor and techniques to exploit the available parallelism.

Instruction Level Parallelism or Pipeline Parallelism
A modern processor provides multiple ways to execute program instructions in parallel to reduce the program execution-time. Instruction Level Parallelism [8] (ILP) or Pipeline Parallelism is a technique where multiple independent instructions are executed simultaneously in a pipelined manner. Modern processors contain multi-stage pipelines where micro-instruction are executed in parallel. Figure 3 shows a 5 stage pipeline where a new independent instruction can start execution every cycle (provided the resources needed by the instruction are available). It is imperative that the processor pipeline is issued new instructions on every clock cycle for efficient utilization of the processor. Independent instructions keep the pipeline busy and increase the processor performance, while dependent instructions stall the pipeline and reduce the performance. Optimizing compilers are good at exploiting ILP by finding independent instructions and scheduling them efficiently to execute in parallel. Another method to exploit ILP in software is loop unrolling which directly exposes independent operations to the compiler.

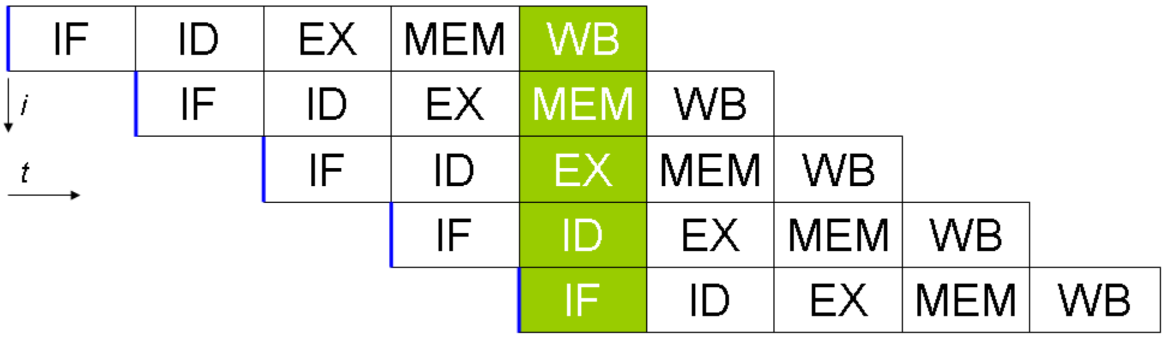


Figure 3: A 5 stage pipeline where an instruction is executed in stages and a new instruction can start execution as soon as the resource used by the last instruction becomes available. The stages shown in the diagram are Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (MEM), and Write Back (WB). Source [13].

Data-Parallelization or Vectorization
Modern hardware provides vector units to exploit data-level parallelism [9]. In data-parallelization, a single instruction is used to perform the same operation on multiple

data points simultaneously. Vector units uses special instruction set called SIMD [10] (Single Instruction Multiple Data) instructions. SIMD instructions include instructions to perform arthimetic, data-movement, and bit-wise operations for both integer and floating point values. Some examples of SIMD instructions from different vendors are SSE, AVX from Intel; 3DNow! from AMD; AltiVec, SPE from IBM; MAX from HP; and NEON from ARM.

The vector instructions perform better than a SISD (Single Instruction Single Data) instruction due to the fact that only a single instruction is decoded and executed for multiple data points. The data access pattern also exhibits a strong locality of reference. The data is accessed from consequent memory locations. This predictable pattern helps the hardware to prefetch data before the instruction requires it. Many optimizing compilers are capable of producing automatic vectorized code. These compilers checks for data dependencies between instructions and vectorize parts of code that are independent. Figure 4 shows a SIMD instruction working on multiple data points.

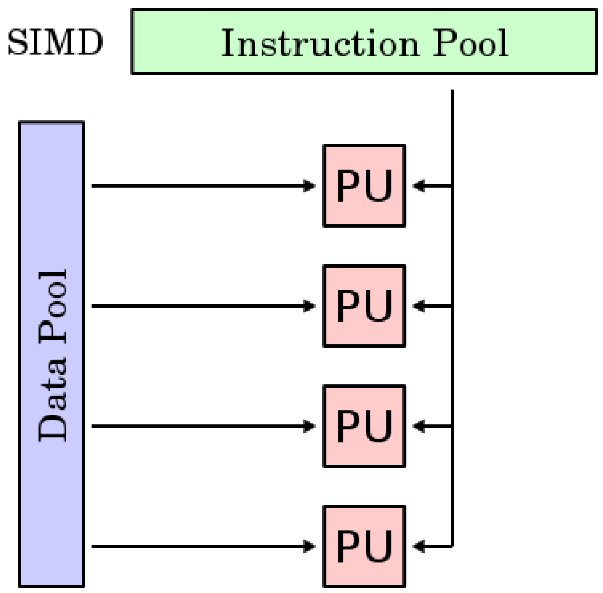


Figure 4: A SIMD architecture with multiple Processing Units (PU) processing multiple data points using a single instruction. Source [14].

Task Parallelism
Task parallelism [11] is used to execute independent tasks in parallel. Independent task could be a print job working in the background and windows updating in the foreground. Independent task could also be iterations of a loop executing in parallel. Independent task could be executed on the same machine or on distributed machines. Tasks are created by Threads or Processes, abstractions provided by the operating system. They are optimally scheduled by the operating system to utilize the available resources. Hardware support for tasks are provided by multiple or multi-core processors. Operating

systems uses the concept of time-slicing where a single CPU is multiplexed between multiple Threads and Processes.

In performance programming, tasks are mostly iterations of loop running in parallel or program running on distributed systems. On a shared memory machine, OpenMP [12] is mostly used to parallelize iterations of a loop. Other commonly used threading libraries are pthreads, Win32 threads, Boost etc. On a distributed cluster environment, the two typical approaches to communication between cluster nodes have been PVM, the Parallel Virtual Machine and MPI, the Message Passing Interface. However, MPI has now emerged as the de facto standard for message passing on a distributed environment.

Compared to SIMD instruction, a thread or a process executes a MIMD (Multiple Instruction Multiple Data) type of instruction. Figure 5 shows a MIMD instruction working on multiple instructions and multiple data points.

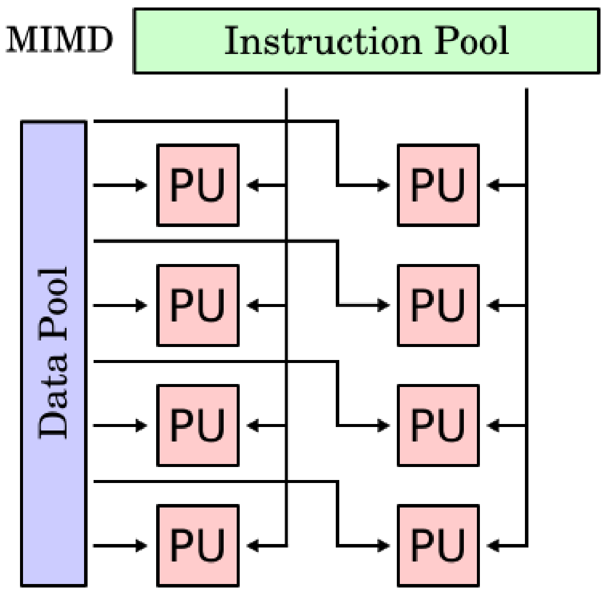


Figure 5: A MIMD architecture (Task parallel) with multiple Processing Units (PU) processing multiple data points using multiple instructions. Source [15].

2.2.2 Dependency Analysis

As shown in the Section 2.2.1, modern processors executes programs faster by executing instructions in parallel. Instructions can only be executed in parallel if there are no execution-order dependency between statements/instructions. Dependent instructions inhibit parallelism and may introduce bubbles in the processor pipelines or serialize loop iterations. Dependency analysis techniques are available to check for dependencies between instructions and loops. The next sections discuss techniques available to check for *data dependencies* and *loop dependencies* in a program.

Data Dependence Analysis

Understanding data dependencies is fundamental in implementing parallel algorithms. Data Dependence Analysis [17] determines whether or not it is safe to re-order or parallelize statements. Bernstein's conditions [18] describe when the instructions are independent and can be executed in parallel.

$$O(i) \cap I(j) = \emptyset \tag{1}$$
$$I(i) \cap O(j) = \emptyset \tag{2}$$
$$O(i) \cap O(j) = \emptyset \tag{3}$$

Where:

$O(i)$:	is the set of (output) locations altered by instruction i
$I(i)$:	is the set of (input) locations read by instruction i
$\emptyset$:	is an empty set.

A potential data-dependency exists between instruction i and a subsequent instruction j when at least one of the conditions fails.

A *flow dependency* (statement using results produced by its preceding statements) is introduced by the violation of the first condition. An *anti-dependency* is represented by the second condition, where the statement j produces a variable needed by statement i. The third condition represents an *output dependency*: When two statements write to the same location, the result comes from the logically last executed statement.

Figure 6 demonstrate several kinds of dependencies. In Figure 6(a), statement S2 cannot be executed before (or even in parallel with) statement S1, because statement S2 uses a result from statement S1. It violates condition 1, and thus introduces a flow dependency. Similarly, Figure 6(b) and Figure 6(c) violates condition 2 and condition 3 respectively.

Table 1 shows the notations used in this document to describe data dependencies between statements.

Flow dependence	δ
Anti-dependence	$\bar{\delta}$
Output dependence	δ°

Table 1: Data Dependency Notations

$$\begin{aligned} S1: X &= \\ S2: &= X \end{aligned}$$

(a) Flow or true dependency

$$\begin{aligned} S1: &= X \\ S2: X &= \end{aligned}$$

(b) Anti dependency

$$\begin{aligned} S1: X &= \\ S2: X &= \end{aligned}$$

(c) Output dependency

Figure 6: Data Dependency Scenarios. a) shows X cannot be read in statement 2 until X is written to in statement 1. b) shows X cannot be written to in statement 2 until statement 1 has read the value in X. c) shows X cannot be written to in statement 2 until X is written to in statement 1.

Loop Dependency Analysis

A numerical application spends majority of its time executing instructions inside a loop. Hence, loops are potential candidates for parallelization. A well-designed loop can produce operations that can all be performed in parallel. However, a single misplaced dependency in the loop can force it all to be run in serial. A Loop Dependency Analysis [17] is the task of determining whether statements within a loop body form a dependence, with respect to array access and modification. If the analysis can determine that no data dependencies exist between the different iterations of a loop then the iterations can be executed in parallel. The types of dependencies in loops are:

1. Loop-independent dependence.
2. Loop-carried dependence.

Loop-independent dependence occurs between accesses in the same loop iteration, whereas *loop-carried* dependence occurs between accesses across different loop iterations. Listing 1 shows example of loop dependencies.

```
    DO I = 1, N
S1:       A(I) = A(I-1) + 1
S2:       B(I) = A(I)
    ENDDO
```

Listing 1: Loop Dependency Example

In the given example, there is loop-carried dependence from statement $S1_i$ to statement $S1_{i+1}$. That is, the instruction that uses A(i-1) can only access the element after the previous one computes it. Every iteration depends on the result of its previous iteration. Clearly, this is not a problem if the loop is executed serially, but a parallel execution may produce a wrong result. Similarly, there is a loop-independent dependency between statements $S1_i$ and $S2_i$. The statement $S2_i$ depends on the output of statement $S1_i$ hence the statements cannot be re-ordered or executed in parallel.

Another concept that is important in loop analysis is the *Data Dependency Direction Vector*. As the name suggests, direction vectors return the direction of the dependency: forward, backward or equal. In forward dependency the value is computed in iteration *i* and used in iteration *i+k*. In backward dependency the vale is computed in iteration *i* and used in iteration *i-k*. This is only possible in nested loops. Finally, in equal dependency the value is computed in iteration *i* and used in iteration *i*. These techniques will be examined in greater detail in the coming sections.

Table 2 shows the notations used in this document to show direction of loop dependency.

Forward	$<$
Backward	$>$
Equal	$=$

Table 2: Data dependency direction notations

2.3 Test data

The two test inputs used to check the performance and correctness of the modified algorithms are H2O-32 and W216. H2O-32 is a small benchmark which performs 10 MD steps on a system of 32 water molecules in a 9.85 Angstrom cubic cell. This small benchmark is used to quickly check the performance and correctness of the modified algorithms.

W216 is a large system of 216 molecules in a non-periodic 34 Angstrom cell. The atoms in the system are clustered in the center of the simulation cell. Because of large molecules some systems will have fewer tasks than others. This creates a load imbalance and is a good test case for testing the load balancing algorithm.

3 An overview of load balancing in CP2K

As discussed in the introduction section, CP2K uses Quickstep algorithm for molecular simulations. This algorithm uses dual basis approach and represents data in two different formats: sparse matrix and regular grid. The data needs to be converted between these two representations in every iteration of the SCF loop. Figure 1 shows the conversion between different representations. During the conversion, the gaussian basis function stored in the sparse matrix form is distributed to the real-space grid (shown as the collocation step in Figure 1). The gaussian basis function is stored as a task list in the source process. Once distributed, the task may be executed by other process in the system. The function of the load balancing module is to optimize the choice of the destination process to balance the load of task across all processes in the system.

The load balance is achieved by migrating tasks between the least loaded neighboring processes. The key information to migrate tasks between processes are the default destination process of a task, its alternative destinations, and the cost of the task. The default destination of a task is the process where it will be executed. The alternate destinations are the processes where the task may be migrated for load balancing. Figure 7 shows the mapping of task destination and its alternate destinations on the real-space grid.

The task is represented by the `task` data-structure in CP2K. Listing 2 shows the fields of this data-structure.

```
 task(1,i) #default destination
 task(2,i) #source
 task(3,i) #compressed type (iatom, jatom,...)
 task(4,i) #(0: replicated,1: distributed local,2:
distributed generalised)
 task(5,i) #cost
 task(6,i) #alternate destinations, 0 if none
available
```

Listing 2: Fields of the task data-structure. The alternate destinations field also contains the process ID of the default destination process.

As shown in Listing 2, the default destination is stored in `task(1,i)` field, the alternate destinations are stored in `task(6,i)` field, and the cost of the task is stored in `task(5,i)` field. The destination and the alternate destinations of the task are computed in the `rs_find_node` routine. The cost of a task is computed as

$$((lmax + v1) * (cmax + v2)^3 * v3 * fraction + v4 + v5 * lmax^7)/1000$$

Where:

$cmax$:	measure of the radius in grid points of the gaussian
$lmax$:	angular momentum quantum orbital for the particular gaussian
$fraction$:	factor to split cost between processes for the generalized task type
v_i:	constant to account for other costs

Figure 8 shows accuracy of prediction of the cost model.

Figure 9 shows the flowchart summarizing the steps to optimize load balance between all processes. Every process maintains a list of tasks that it is responsible to execute, along with a list of alternate destinations where the task can be migrated for load balancing. The idea of load balancing is to shift tasks to their least loaded alternate destination

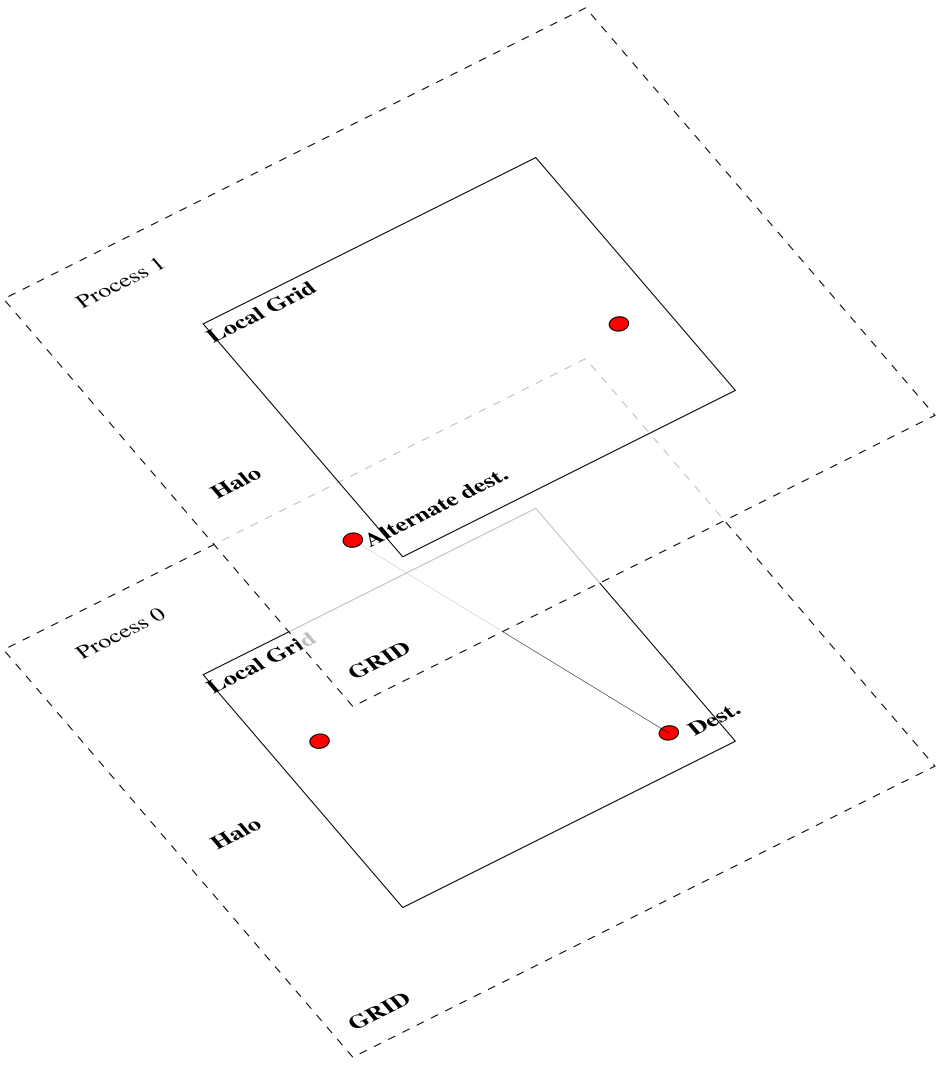


Figure 7: Mapping of tasks on grid. Tasks are shown as red circles. A tasks default process is the one where it falls within the local grid. Alternate destinations are the process where the entire task falls within the halo region.

and even out load among all processes. The load of a process is calculated by summing the cost of the tasks that is assigned to it. Once the load on each process is calculated, the optimum load of all processes is found by iteratively shifting the load between the destination and alternate destination processes. Finally, the task is re-assigned to its least loaded possible destination.

The load balancing module is implemented in the `task_list_methods.f90` file. `load_balance_distributed` is the driver routine that implements the load balancing process. The driver routines calls `create_destination_list` to allocate memory for the `list` data-structure (discussed in Section 3.2), `compute_load_list` to add task load to the `list` data-structure (discussed in Section 3.2) and update task destination after load optimization (discussed in Section 3.4), and `optimize_load_list` to optimize the global load of all processes (discussed in Section 3.3).

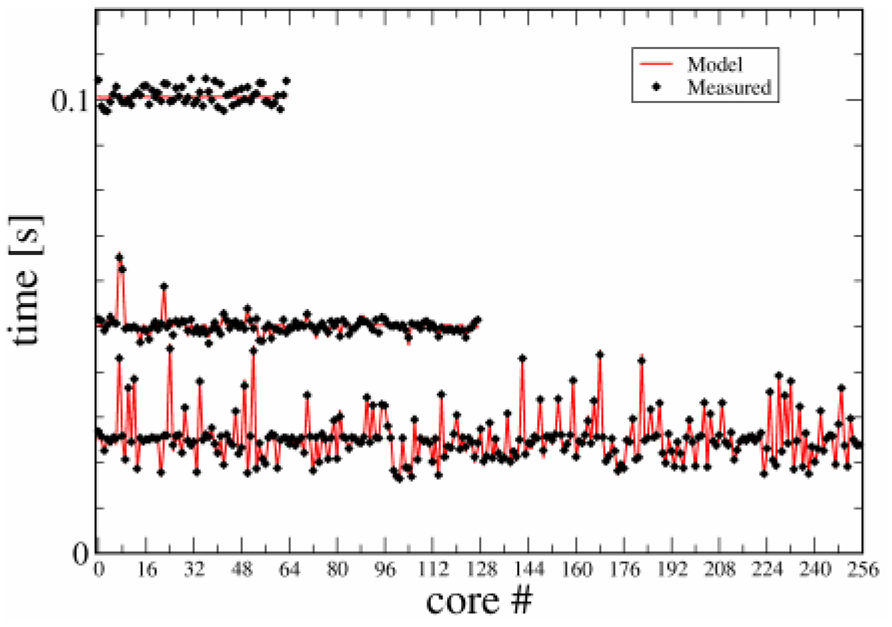


Figure 8: Cost model accuracy. Source [16].

The next sections show details of the implementation begining with the key data-structures used in the implementation.

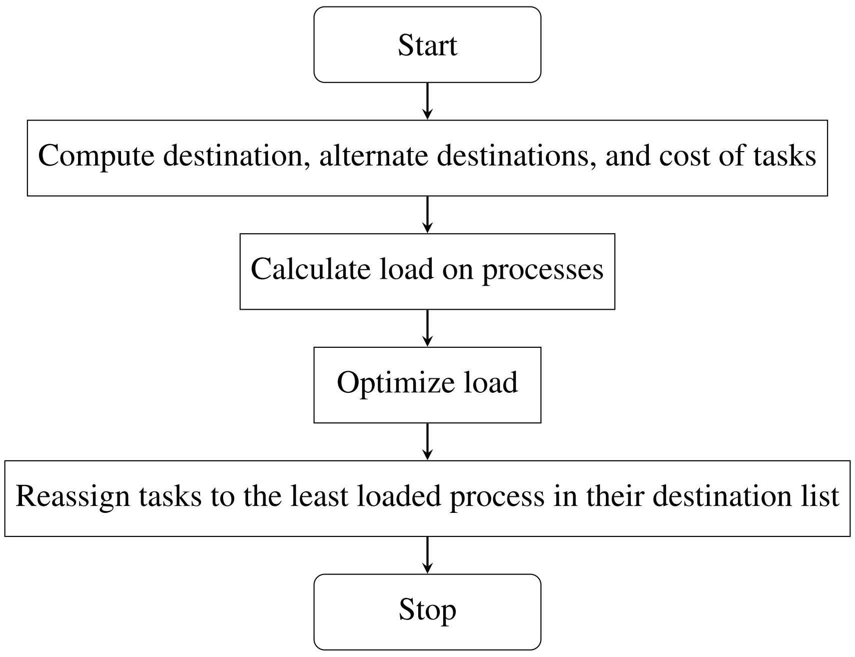

Start
Compute destination, alternate destinations, and cost of tasks
Calculate load on processes
Optimize load
Reassign tasks to the least loaded process in their destination list
Stop

Figure 9: Flowchart of the load balance optimization process.

3.1 task data-structure initialization

As discussed in the previous section, the `task` data-structure contains the key information to migrate tasks between processes to optimize the load on every process. The task's default destination and alternate destinations are found on the real space grid. The real space grid is decomposed into chunks (Grid) handled by different MPI processes. The default destination of the task is the process which have sections of the grid, including halos, that fully contain the grid-points required to process the current gaussian (task). Then the 6 neighbour nodes (+/- x,y,z) are checked to see if they fit the bounds of the task. If so these processes are set as the alternate destinations of the task. See Figure 7 for a 2D view of the task mapping process. See [4] for a detailed discussion of the grid layout.

The information stored in the `task` data-structure is re-organized and stored as a map of task destination, alternate destinations and its cost. This new re-organized information is stored in a data-structure called `list`. The next section describes the `list` data-structure in detail.

3.2 The list data-structure

As discussed in the previous section, the `list` data-structure stores the re-organized information of the `task` data-structure. From now onwards, the default task destination will be referred to as the source process and the alternate task destinations will be referred to as the destination process. The term cost and load will be used interchangeably to refer to the load of the process.

The `list` data-structure is a 3 dimensional array. Figure 10 shows the layout of this data-structure. The Rank (process ID is referred to as Rank in MPI) of the destinations are stored in the first dimension of the array and the cost of the tasks that can be shifted to these destinations are stored in the second dimension. The third dimension is used to index the array for all processes in the system. For example, in an 8 process system, the statement `dest = list(1,2,3)` returns the rank of the 2nd destination of the 3rd source. Similarly, the statement `load=list(2,2,3)` returns the cost of task of the same example process. The total load of the source process is the sum of the cost of the task transferable to its destination process.

list initialization
The `list` array is created in the routine `create_destination_list`. The size of the first and second dimension of the array is determined by the number of destinations a process has: 3 for 1D decomposition, 5 for 2D decomposition, and 7 for 3D decomposition. The different possible decompositions are shown in Figure 11. The size of the third dimension is determined by the number of processes in the system.

After the array is created, the `create_destination_list` routine initializes the

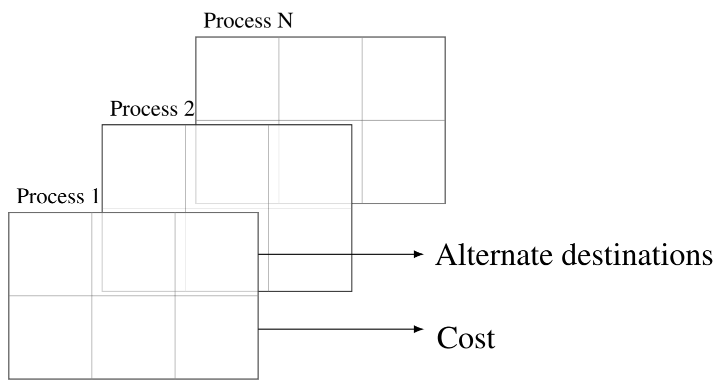


Figure 10: Example layout of the 3 dimensional list array with the alternate task destinations of the process stored in the first dimension of the array and the corresponding task cost stored in the second dimension. The index of the third dimension (shown in the vertical direction) is used as the source process identifier.

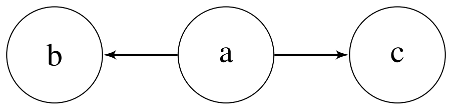
first dimension of the `list` data-structure with the Ranks of the task destination processes. The destination process information is found in the `rs_descs` data-structure. The destination processes are the neighboring process of the source process under consideration. Since the information stored in `rs_descs` is global and same in all processes, the first dimension values are same in all processes. A detailed discussion of `rs_descs` data-structure is beyond the scope of this document.

The second dimension of the `list` is initialized with the cost of the tasks. The cost of the task stored in the `task` data-structure is added to its destination process cost in the `list` data-structure. The initialization is done in the `compute_load_list` routine. The routine handles two cases: initialize `list` array with cost of the task, and reassign task to optimum process after load optimization.

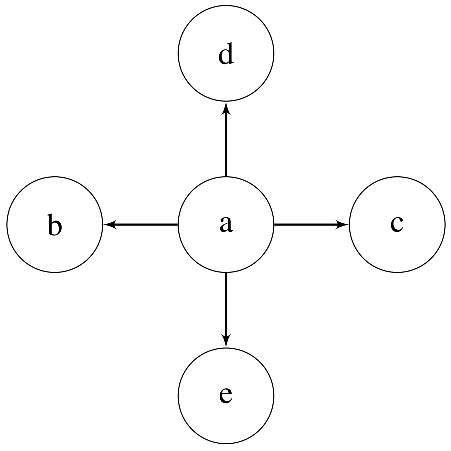
The pseudo-code for computing process load is shown in Algorithm 1. The algorithm groups all tasks into blocks that handles the computation of same atom pairs and thus depend on the same matrix block. A list of destination for every task in the block is created and for each task in the block, its alternative destinations are checked in the list. If alternative destinations are found in the list, the task is assigned to the least loaded destination in the list. Otherwise, the task is assigned to its default destination. This completes the initialization of the `list` data-structure. Next, using this information, the global load of all process is computed and optimized for load balancing.

3.3 Load optimization

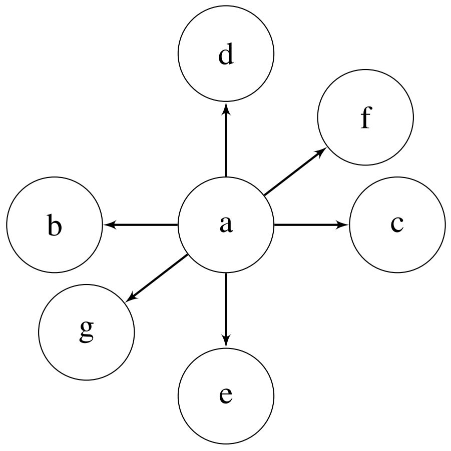
CP2K uses an iterative algorithm for load optimization. The load of processes are optimized and tasks are migrated to their possible destinations to balance the work load across processes. Figure 12 shows the load balancing process diagrammatically. The diagram shows two processes, Process A and Process B as load sharing processes. The process begins by Process A sharing its work load with Process B, Process B in-turn



(a) 1D arrangement



(b) 2D arrangement



(c) 3D arrangement

Figure 11: Possible network arrangements. a) shows 1D arrangement where load can be shared between processes arranged in x dimension only. b) shows 2D arrangement where load can be shared between processes arranged in x and y dimensions. c) shows 3D arrangement where load can be shared between processes arranged in x, y, and z dimensions.

Algorithm 1 Compute load list

```
Begin
    for all $blocks$ in $tasks$ do                                ▷ task blocks with same atom pairs
        for all $i$ in $blocks$ do
            $dests_i \leftarrow task_i.dest$
        end for
        for all $i$ in $blocks$ do
            for all $d$ in $task_i.alt\_dests$ do
                if $d \in ANY(dests)$ then
                    $alt\_opts \leftarrow d$
                end if
            end for
            if $alt\_opts \neq \emptyset$ then
                $rank \leftarrow \text{LEAST\_LOADED}(alt\_opts)$
            else
                $rank \leftarrow task_i.dest$
            end if
            $li \leftarrow list\_index(list, rank)$
            if create_list then                                   ▷ Case 1: init list
                $list(2, li, task_i.dest) = list(2, li, task_i.dest) + task_i.cost$
            else                                                  ▷ Case 2: update task dest
                if $list(1, li, task_i.dest) \neq task_i.dest$ then
                    if $list(2, li, task_i.dest) \geq task_i.cost$ then
                        $list(2, li, task_i.dest) = list(2, li, task_i.dest) - task_i.cost$
                        $task_i.dest \leftarrow rank$
                    end if
                end if
            end if
        end for
    end for
End
```

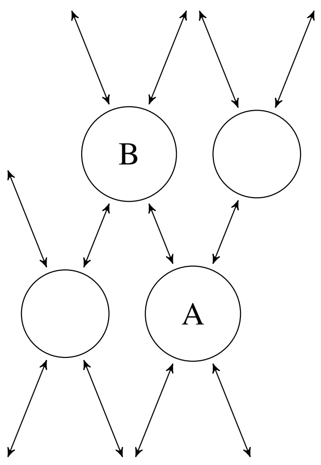


Figure 12: Iterative load sharing

shares its work load with its neighbors. The neighbors of Process B shares their work load with their neighbors and so on. This process is iteratively executed optimizing the load across the system. Once optimized, the tasks are migrated from overloaded process to the least loaded process.

The pseudo-code for load optimization is shown in Algorithm 2. The process is implemented in the `optimize_global_list` routine. The algorithm has three parts: calculate global load, optimize global load, and update local load (the `list` data-structure). Each part is explained next.

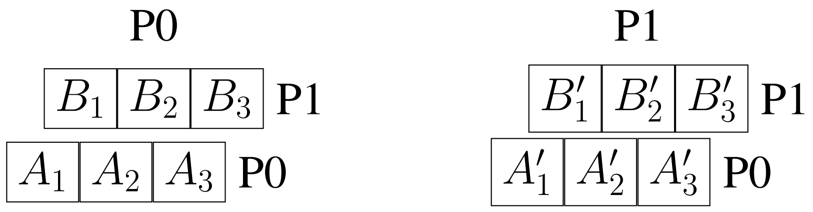
Calculating global load

All process store their task information locally in the `list` data-structure. These fragmented local load information is accumulated on a single process (Rank 0) and processed to create the global load information. The global load information shows the total load on each process globally.

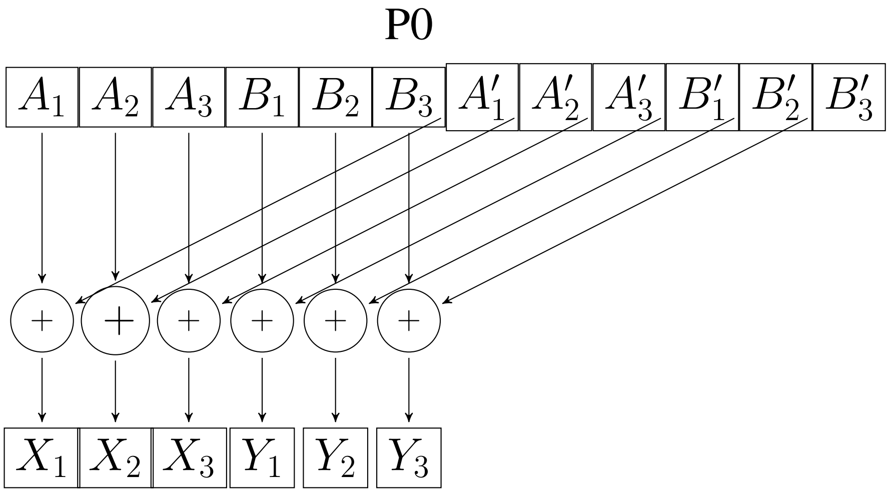
Algorithm 2 shows the pseudo-code to compute global load of each process. The local load information from all process is gathered on Rank 0. A temporary array, `load_per_source`[1], sufficient enough to store information from all processes is allocated to gather this information. The global load on each process is calculated by summing the local load gathered in the last step. The global load is stored in the `list_global` array.

Figure 13 shows the steps to calculate global load diagrammatically. In the diagram, load of Process P0 and Process P1 is gathered on Process P0 and stored in the temporary array `load_per_source`. The values in the temporary array are added together to find the global load of each process.

[1]The load_per_source array is declared as a 1D array in the implementation. Here the array is shown as a 2D array for ease of explaination.



(a) Local load information in the list data-structure. Alternate destinations not shown.


P0
A_1 A_2 A_3 B_1 B_2 B_3 A'_1 A'_2 A'_3 B'_1 B'_2 B'_3
+
X_1 X_2 X_3 Y_1 Y_2 Y_3

(b) Load gathered from all processes to Rank 0. Process wise load added to compute global load

Figure 13: Calculating global load - in serial. a) shows load information stored in the list data-structure (alternate neighbors fields excluded from the diagram). b) shows load information gathered on Rank 0 and global load computed serially on Rank 0.

Algorithm 2 Optimize load list

```
Begin
    MPI_GATHER($list.load, load\_per\_source, \ldots$)        ▷ Gather Local loads
    if $RANK = 0$ then
        for all $i$ in $nProc$ do
            for all $j$ in $nProc$ do
                $load\_all_j \leftarrow load\_all_j + load\_per\_source_{i,j}$  ▷ Calculate global load
            end for
        end for
        $list\_global.dest \leftarrow list.dest$
        $list\_global.load \leftarrow load\_all$
        BALANCE_GLOBAL_LIST($list\_global$)        ▷ Optimize global load
        for all $i$ in $nProc$ do
            for all $j$ in $nProc$ do
                $tmp \leftarrow$ MIN($list\_global_j.load, load\_per\_source_{i,j}$)
                $load\_per\_source_{i,j} \leftarrow tmp$
                $list\_global_j.load \leftarrow list\_global_j.load - tmp$
            end for
        end for
    end if
    MPI_SCATTER($load\_per\_source, list.load, \ldots$)        ▷ Update local load
End
```

Optimizing global load
The load optimization algorithm is implemented in the `balance_global_list` routine. This algorithm works on the `list_global` data-structure described previously. It finds the optimum flux (amount of work load) that could be shifted between the source and the destination process. The calculated optimum flux is used to normalize the load stored in the `list_global` data-structure.

Similar to the `list` data-structure, the `list_global` array stores values in pairs of alternate destinations and task load. However, the load information stored in the `list_global` array is that of the global load of all processes in the system. This values also sets a limit on the amount of load that could be shifted between the pairs of process. The limit is called the *flux limits*. Another concept used in the load balancing module is that of a *flux connections*. The pair of the source and the destination process in the `list_global` array forms a *flux connections*. The load optimization algorithm balances the load on processes by optimizing the load between the pairs in the *flux connections*. See Section 5.2.1 for a detailed discussion on the load optimization process.

Updating local load list
The optimized global list, computed in the last step, is used to update the local load stored in the `list` array. The new local load is computed by the Rank 0 process and

scattered to their respective processes. Processes gather this information and update their `list` array with the new load information. The pseudo-code for the algorithm is shown in Algorithm 2. The algorithm reassigns local load array (`load_per_source`) with the minimum of existing values or the newly computed load. The updated load is scattered to the respective processes which updates the load values in their `list` array. The steps to update the local load information is shown diagrammatically in Figure 14. The diagram shows the updated local load information calculated in Rank 0 and scattered to their respective processes.

3.4 Update task destination

Finally, the updated `list` is used to reassign tasks to its optimum destinations. The pseudo-code for the task reassignment is shown in Case 2 of Algorithm 1. The `list` data-structure contains optimized load from the last step. The algorithm searches for the least loaded process in the task's alternate destination list. If the destination process can accommodate load of the task, the task is assigned to the process otherwise the task remains with its default process.

This concludes the discussion on existing implementation of the load optimization process. The next section discusses issues with the current implementation followed by solutions to the identified issues.

4 Issues with current implementation

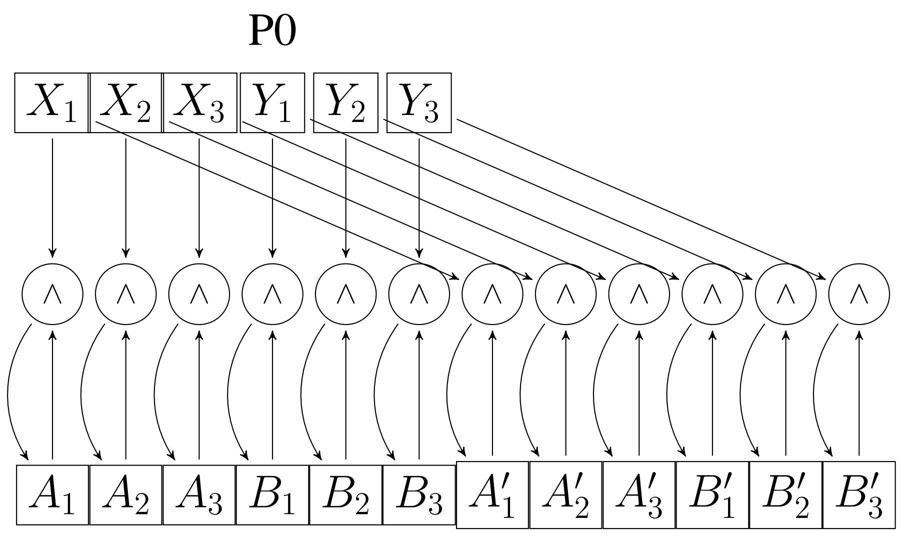
4.1 High memory requirement

The first major issue with the current implementation is the amount of memory required to implement the load balancing module. The most expensive data-structure in terms of memory requirement is the temporary array `load_per_process`. Memory for this data-structure is allocated in the `optimize_load_list` routine. The variable is used to gather local load information from all processes in the system. The load information is retrieved from the `list` data-structure. `list` uses *O(P)* memory to store load information of all task in a process. In order to store this information centrally, `load_per_process` allocates *O(P^2)* amount of memory. The amount of memory needed is calculated as

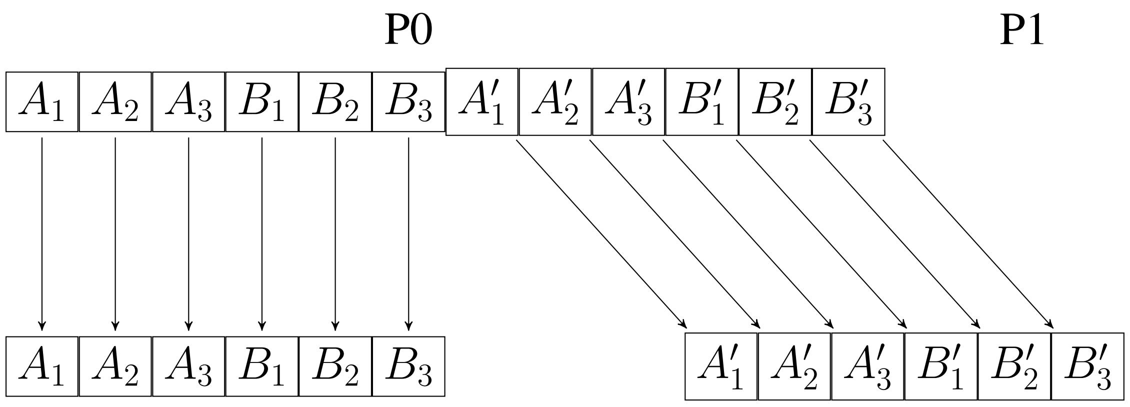
$$sizeofint * max\_dest * ncpu^2$$

Where:

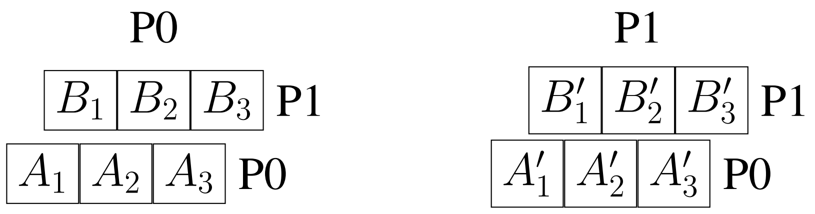
$sizeofint$:	is the word size on the executing machine
max_dest:	is maximum alternate destinations for a task
$ncpu$:	is the number of processes in the system.



(a) Updating load_per_source_array



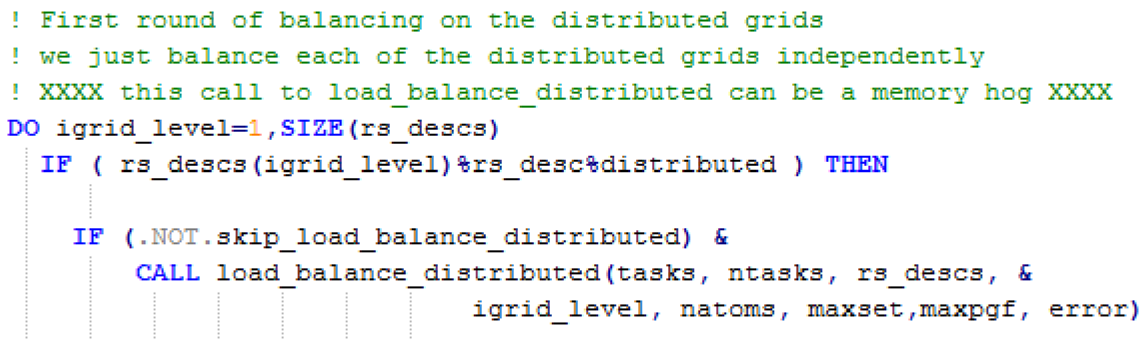
(b) Scatter updated load_per_source_array to respective processes



(c) Update list data-structure with updated load information

Figure 14: Updating list array. a) shows distribution (using minimum ($\wedge$) operation) of optimized list_global to load_per_source array. b) shows the updated load_per_source scattered to the respective processes. c) shows the list data-structure updated with the new load information.

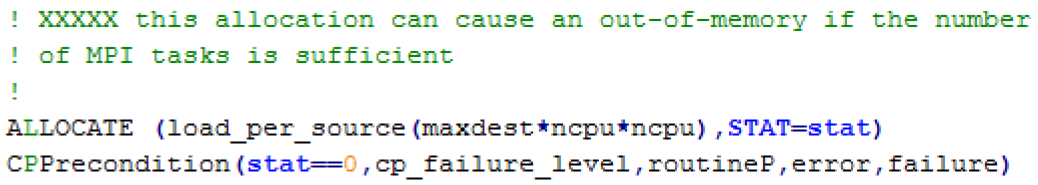
The usage of this variable is shown in Algorithm 2. Figure 15 and 16 shows warnings in the source code related to high memory usage.



```
! First round of balancing on the distributed grids
! we just balance each of the distributed grids independently
! XXXX this call to load_balance_distributed can be a memory hog XXXX
DO igrid_level=1,SIZE(rs_descs)
  IF ( rs_descs(igrid_level)%rs_desc%distributed ) THEN

    IF (.NOT.skip_load_balance_distributed) &
        CALL load_balance_distributed(tasks, ntasks, rs_descs, &
                                      igrid_level, natoms, maxset,maxpgf, error)
```

Figure 15: High memory requirement warning in distribute_tasks routine



```
! XXXXX this allocation can cause an out-of-memory if the number
! of MPI tasks is sufficient
!
ALLOCATE (load_per_source(maxdest*ncpu*ncpu),STAT=stat)
CPPrecondition(stat==0,cp_failure_level,routineP,error,failure)
```

Figure 16: Out of memory warning in optimize_load_list routine

Figure 17 shows the memory requirement for `load_per_source` array versus number of processes. The memory requirement increases quadratically with increase in the number of processors. The high memory requirement limits the algorithm scalability beyond a certain number of processes. For example, on ARCHER, the amount of memory available per node is 64GB. The number of processors per node is 24, this gives 2.6GB of available memory per processor. The Operating system does not support Virtual Memory hence 2.6GB is the hard limit. This limits the algorithm to scale beyond 10,112 processes on ARCHER.[2] The load balancing function can be turned on or off through the `SKIP_LOAD_BALANCE_DISTRIBUTED` switch provided in the input file. By default, the load balancing function is turned off. The function is also turned off for programs running on more than 1024 processes. Figure 15 shows the code snippet checking this flag to run the load balancing routine.

4.2 Serial task

The second major issue with the current implementation is the serial processing of the `balance_global_list` routine. As discussed in Section 3.3, this routine optimizes the load of the system by iteratively shifting load between processes. The time complexity of the algorithm is $O(N^2)$ as `nproc*nproc` load shift operations are performed by this algorithm. These operations are computed iteratively and could take

[2]In practice CP2K is not scalable beyond 10,000 processes due to memory requirements of other modules.

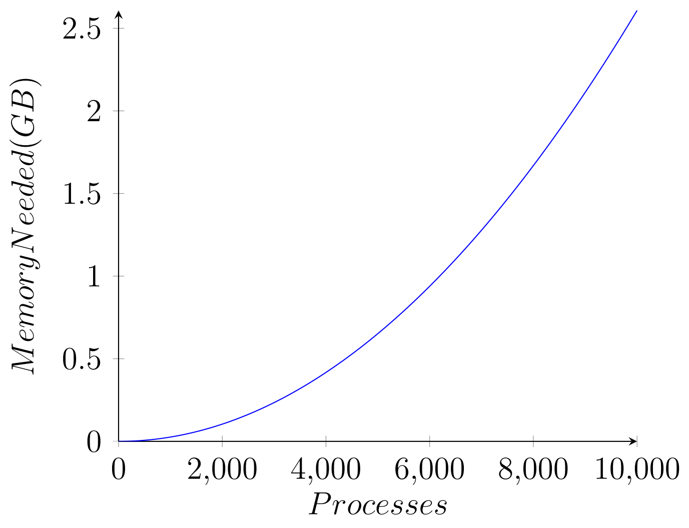


Figure 17: Memory needed vs number of processes

many steps to converge making this a time consuming task. This is a good candidate task for parallelization. However, in the current implementation, the task is executed only by the Rank 0 process effectively making this a serial task. This limits the scalability of the algorithm because no matter how many processors are used only Rank 0 process executes the task.

The efforts to remove the issues identified in this section are described next.

5 Improvements

The two major issues with the current implementation identified in Section 4 are

1. High memory requirement
2. Serial processing of load optimization task

This section describes the solutions to the above identified issues. Section 5.1 describes the solution to resolve memory issues and Section 5.2 describes the solution to parallelize the serial load optimization task.

5.1 Resolving memory issues

The `optimize_load_list` routine is the most memory consuming routine in the load balancing module. This routine is used to optimize the global load of all processes. As described in Section 4, the routine allocates $O(P^2)$ memory to gather local load data from all processes in the system. The gathered data is stored in the variable called `load_per_source`. This variable is used to compute two pieces of information:

a) global load of all processes in the system; and *b*) distribution of optimized load to source processes .

The computed global load is sent to the `balance_global_list` routine which optimizes the global load values. The optimized global load is distributed to the `list` array through the `load_per_source` variable. A detailed discussion of this process is given in Section 3.3.

The need for this large array arises because the global load and optimum load values are computed sequentially on a single process (Rank 0). To compute this values the load information stored in other processes are required on Rank 0. The situation can be improved if the information can be calculated locally by all processes without transferring and storing large amount of system-wide global information.

Section 5.1.1 and Section 5.1.2 describes the attempt to compute the above mentioned information locally by all processes thus eliminating the need for the large memory for load balancing.

5.1.1 Computing global load

As mentioned in Section 5.1, the global load of all process is required to balance the work load between processes. The pseudo-code of existing implementation to compute the global load is shown in Algorithm 3. Algorithm 3 is a snippet from Algorithm 2. In order to compute the global load, the local loads from other processes are gathered on Rank 0 using the MPI collective routine *MPI_GATHER*. See Section 2.1 for information on MPI collective routines. The gathered values are stored in the large array `load_per_source`. The values in the array are added together to compute the global load on all processes. See Section 3.3 for details on computation of global load.

Algorithm 3 Computation of global load

```
Begin
    MPI_GATHER($list.load, load\_per\_source, \dots$)        ▷ Gather on RANK 0
    if $RANK = 0$ then
        for all $i$ in $nProcess$ do
            for all $j$ in $nProcess$ do
                $load\_all_j \leftarrow load\_all_j + load\_per\_source_{i,j}$
            end for
        end for
        ...
    end if
    ...
End
```

In order to eliminate the use of the large array, the computation needs to be done collectively by all process and the final result needs to be communicated back to Rank 0.

The MPI collective operation MPI_REDUCE can be used to compute the global load collectively by all processes. This operation applies a global reduction operation on data from all processes in the system and places the result on a single process. The reduction operation is a polymorphic function that can be used to parallelize computation of associative operations like min, max, sum, product, bit-wise operation etc. See Section 2.1 for information on MPI collective operations.

Algorithm 4 shows the parallel implementation of the global load computation using the MPI_REDUCE collective routine. Compared to the existing solution, the new implementation computes the value collectively by all processes without the overhead of the large temporary array load_per_source. This eliminates the large array needed to store the global information required in serial computation.

Algorithm 4 Parallel computation of global load

```
Begin
    MPI_REDUCE($list.load, load\_all, MPI\_SUM, \dots$)    ▷ Result on RANK 0
    ...
End
```

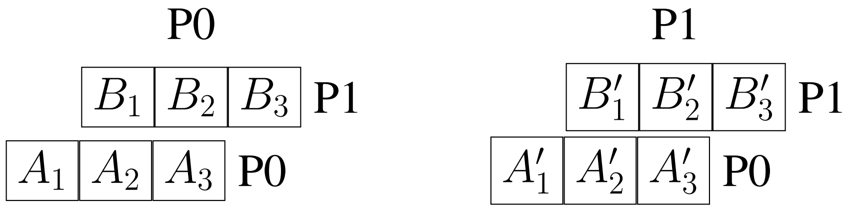
Figure 18 shows the process diagrammatically where process P0 and P1 collectively add the load of each process stored in the list data-structure and stores the result in process P0.

5.1.2 Update local load list

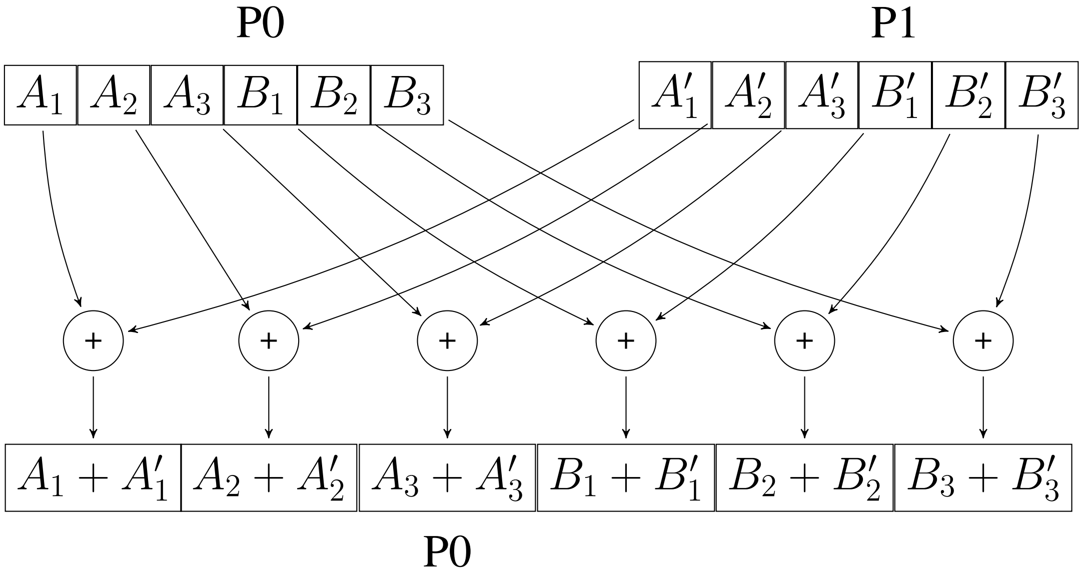
As described in Section 5.1, the second usage of the large array is to distribute the optimized global load information to source processes. Source process uses this information to update the list array and migrate their tasks to the optimum process. Algorithm 5 is a snippet from Algorithm 2 showing the distribution of global load. The list_global array contains the optimized global load values computed by the balance_global_list routine. The optimized values are distributed to the list data-structure through the load_per_source array. See Section 3.3 for a discussion on the global load optimization process. Similar to the computation of the global load values, this is also a sequential process and requires the large temporary array load_per_source for computation.

Similar to the solution presented in the previous section, the temporary array can be eliminated by computing the optimum local load values locally on every process. Unlike the previous solution, there are no ready made routines available to parallelize the algorithm. Hence, the algorithm needs to be analyzed and parallelized manually.

Before parallelizing a serial algorithm, the algorithm needs to be analyzed for fitness of parallel execution. An algorithm can only be parallelized without any overhead if there are no dependencies between potential parallel tasks. Dependency analysis methods like the data dependency analysis and loop-dependency analysis are used to analyze the



(a) Load information in the list data-structure. Alternate destination information not shown.

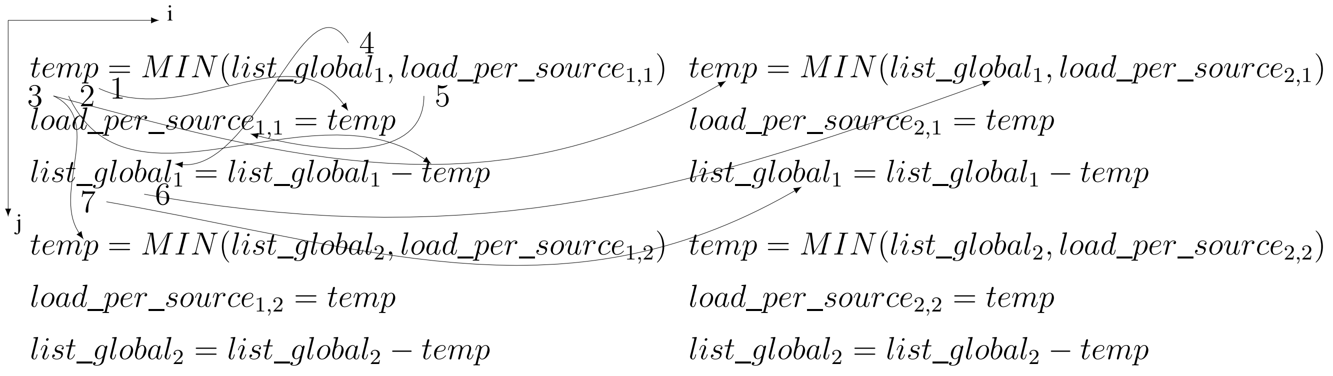

P0
A_1 A_2 A_3 B_1 B_2 B_3
P1
A'_1 A'_2 A'_3 B'_1 B'_2 B'_3
+
$A_1 + A'_1$ $A_2 + A'_2$ $A_3 + A'_3$ $B_1 + B'_1$ $B_2 + B'_2$ $B_3 + B'_3$
P0

(b) Collective computation of global load values

Figure 18: Computing global load - in parallel. a) shows the load values in the list data-structure used to compute the global load. b) shows the collective computation of global load values using MPI_REDUCE operation. The result is available in P0 only.

Algorithm 5 Global load distribution

```
Begin
    if $RANK = 0$ then                                  ▷ compute sequentially on RANK 0
        ...
        for all $i$ in $nProc$ do
            for all $j$ in $nProc$ do
                $temp \leftarrow$ MIN($list\_global_j.load, load\_per\_source_{i,j}$)
                $load\_per\_source_{i,j} \leftarrow temp$
                $list\_global_j.load \leftarrow list\_global_j.load - temp$
            end for
        end for
        ...
    end if
    MPI_SCATTER($load\_per\_source, list.load, \dots$)        ▷ scatter from RANK 0
End
```


i
j
temp = MIN(list_global1, load_per_source1,1)
load_per_source1,1 = temp
list_global1 = list_global1 − temp
temp = MIN(list_global2, load_per_source1,2)
load_per_source1,2 = temp
list_global2 = list_global2 − temp
temp = MIN(list_global1, load_per_source2,1)
load_per_source2,1 = temp
list_global1 = list_global1 − temp
temp = MIN(list_global2, load_per_source2,2)
load_per_source2,2 = temp
list_global2 = list_global2 − temp
1
2
3
4
5
6
7

1: δ ==; 2: δ ==; 3: δ° <<; 4: $\bar{\delta}$ ==; 5: $\bar{\delta}$ ==; 6: δ <=; 7: δ° <=

Figure 19: Data Dependency Analysis of Algorithm 5. Dependency #3 inhibits inner-loop parallelism and dependency #3,6,7 inhibits outer-loop parallelism.

candidate parallel tasks for dependencies and opportunities for parallelization. Section 2.2 gives a background information on various dependency analysis methods.

The next section shows the analysis of Algorithm 5 using the framework described in Section 2.2. The information gathered during the analysis of the algorithm is used to safely convert the serial implementation to a parallel implementation.

Algorithm Analysis

The following section uses the dependency analysis framework described in Section 2.2. Figure 19 shows the dependency analysis of Algorithm 5. To understand the dependencies between loop iterations, the loop is unrolled twice in both dimensions. The outermost loop index *i* is shown as increasing from left to right and the innermost index *j* is shown as increasing from top to bottom. For each iteration the scalar variable `temp` is assigned a minimum of `list_global` or `load_per_source`. Then, array `load_per_source` is assigned the value of `temp` which is then subtracted from `list_global`. Dependencies between statements and iterations are shown using arrows in the figure. The type and direction of dependencies are annotated using the notation shown in Table 1 and 2. Figure 20 shows the dependency graph of the loop under investigation.

The graph shows dependency from statement $S1 \to S2$, from $S1 \to S3$ and from $S3 \to S1$. There is also a self dependency on statement $S1$ and$S3$. The type of data dependency from $S1 \to S2$ is *flow dependence* and *anti-dependence*. The value of `temp` is assigned to `load_per_source` in $S2$ which is computed in $S1$, hence there is a *flow dependence* between statement $S1$ and $S2$. Similarly, the array `load_per_source` is read in statement $S1$ and written to in $S2$, hence there is also a *anti-dependence* dependency between statement $S1$ and $S2$. The data dependence direction (=,=) shows that both the dependencies occur in the same iteration of the loop hence the loop depen-

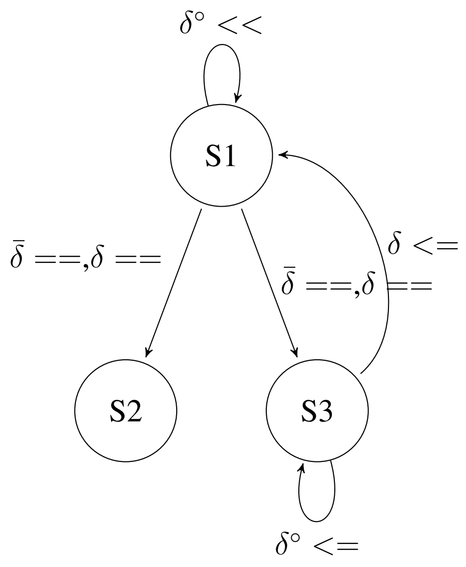


Figure 20: Data Dependency Graph of Algorithm 5.

dency is of type *loop-independent*. Clearly, the statements could be executed simultaneously in different iterations of the loop.

The type of data dependency from $S1 \rightarrow S3$ is *anti-dependence*. The value of array `list_global` is read in statement $S1$ and written to in statement $S3$. Similar to the previous dependency, the data dependence direction (=,=) shows a *loop-independent equal dependence* and hence the statements could be executed simultaneously in different iterations of the loop.

The type of data dependence from $S3 \rightarrow S1$ is *flow dependence*. The data dependence direction (<,=) shows a *forward dependence* in the outer loop level *i*. There is a *loop-carried* dependence from iteration *i* to iteration *i+k*. The value of `list_global` is calculated in iteration *i* and used in iteration *i+k*. The *loop-carried* dependency prohibits the outer loop to run in parallel.

The type of self loop dependency in $S3$ is *output dependence*. Similar to the previous dependency, the type of data dependence direction (<,=) is *loop-carried forward dependence* in the outer loop level. The variable `list_global` is assigned a value in all iterations of the loop and hence the statement cannot be run in parallel.

The type of self loop dependency in $S1$ is *output dependence* and the type of *data dependency* direction (<,<) is *loop-carried forward dependency*. The variable `temp` is assigned value in all iterations of the loop. However, as `temp` is a temporary scalar variable, the dependence can be removed by making it an auto or thread private variable. An auto variable is a variable which is allocated and deallocated in the program's stack memory when the program flow enters and leaves the variable's context. Variables passed as function arguments, declared as local variables inside a function are classified as auto variables. Variables on the stack however are local to threads, because each thread has its own stack residing in a different memory location. Sometimes it is

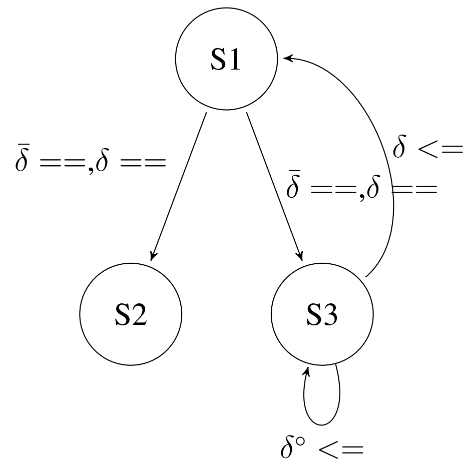


Figure 21: Data Dependency Graph with self dependency on S1 removed

desirable that two threads referring to the same variable are actually referring to different memory locations, thereby making the variable thread-private, a canonical example being the *OpenMP private clause* creating the loop index variable private to the thread.

The next section describes the methods to remove the dependencies identified during the analysis of the algorithm to help parallelize the algorithm.

Loop transformation
In this section, the loop transformation process is illustrated using OpenMP. The transformed loop is parallelized using MPI in the final implementation of the algorithm.

As discussed in the previous section, the inner-loop could be parallelized by making the temporary variable `temp` an auto or thread private variable. Algorithm 5a shows the pseudo-code parallelizing the inner-loop of of Algorithm 5 using the *OpenMP directive*. The variable `temp` is made thread private using the *openMP private clause*. Figure 21 shows the new dependency graph after the modification.

However, parallelization of the inner-loop is not sufficient to parallelize the algorithm for a distributed environment. The inner-loop effectively updates elements of per process data (`list` values stored in `load_per_source`) in parallel but for a distributed parallel algorithm per process data, not elements, needs to be updated in parallel. For example, the inner-loop updates the elements of array of process 0 in parallel then elements of process 1 and so on. In order for this algorithm to run efficiently in a distributed environment, both process 0 and process 1 needs to process their arrays in parallel. This could only be done through parallelization of the outer-loop. But the dependency constraints identified in the previous section prevents the outer-loop from running in parallel.

The outer-loop can be parallelized by transforming the loop into a suitable form that can be parallelized without breaking the dependency constraints. Some of the methods that can transform a loop into a more suitable form required by the algorithm are *loop*

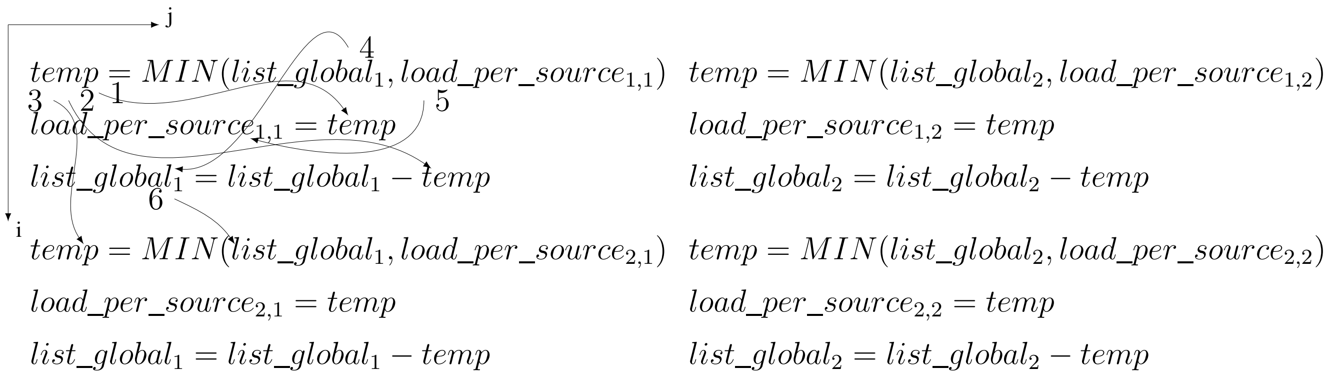
Algorithm 5a Parallel inner-loop

```
Begin
    ...
    if $RANK = 0$ then
        for all $i$ in $nProc$ do
            #pragma omp parallel for private(temp)                    ▷ C notation
            for all $j$ in $nProc$ do
                $temp \leftarrow$ MIN($list\_global_j.load, load\_per\_source_{i,j}$)
                $load\_per\_source_{i,j} \leftarrow temp$
                $list\_global_j.load \leftarrow list\_global_j.load - temp$
            end for
        end for
        ...
    end if
    ...
End
```

unrolling, *loop fission*, *loop fusion*, and *loop interchange*. Loop unrolling limits the loop overhead by increasing the number of instructions executed per iteration. Loop fusion also limits the loop overhead by combining multiple loops into a single loop. Loop fission is used to split loop into multiple loops to expose more parallelism. Loop interchange exchanges the order of two iteration variables used by a nested loop. The variable used in the inner loop switches to the outer loop, and vice versa. It is often done to improve the memory access pattern of a multidimensional array. However, here it can be used to remove the loop carried dependency and parallelize the outer-loop. Figure 22 shows the data dependency analysis of the loop after application of the loop interchange transformation. Figure 23 shows the data dependency graph of the newly transformed loop.

It can be seen from the dependency graph shown in Figure 23 that the loop-carried dependency in the outermost loop no longer exists. Also, Figure 22 shows that the data dependencies are preserved and array elements are accessed and updated in correct order. Algorithm 5b shows pseudo-code for the parallel implementation of global load computation using OpenMP.

Though the array access pattern is not suitable for a shared memory machine, the algorithm can be made to run efficiently on a distributed memory machines. Section 5.1.3 discuss the implementation of the parallel algorithm for a distributed environment using MPI. In the MPI implementation the `list` array is transposed to get the same effect of loop transformation in a distributed environment.


j
i
4
3
2
1
5
6
$temp = MIN(list\_global_1, load\_per\_source_{1,1})$
$temp = MIN(list\_global_2, load\_per\_source_{1,2})$
$load\_per\_source_{1,1} = temp$
$load\_per\_source_{1,2} = temp$
$list\_global_1 = list\_global_1 - temp$
$list\_global_2 = list\_global_2 - temp$
$temp = MIN(list\_global_1, load\_per\_source_{2,1})$
$temp = MIN(list\_global_2, load\_per\_source_{2,2})$
$load\_per\_source_{2,1} = temp$
$load\_per\_source_{2,2} = temp$
$list\_global_1 = list\_global_1 - temp$
$list\_global_2 = list\_global_2 - temp$

1: δ ==; 2: δ ==; 3: δ° =<; 4: $\bar{\delta}$ ==; 5: $\bar{\delta}$ ==; 6: δ =<

Figure 22: Data Dependency Analysis after loop transformation. Note the loop index variables are swapped. The temp variable is private hence no dependency on temp variable on the outer-loop. Compared to Figure 19, the outer-loop dependencies no longer exists.

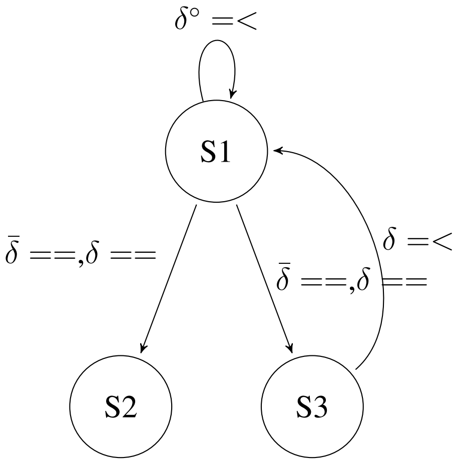


Figure 23: Data Dependency Graph after loop transformation. The first dimension of the data direction vectors shows an equals dependency - the dependency is in the inner loop, the outer loop can now be parallelized safely.

Algorithm 5b Parallel outer-loop

```
Begin
    ...
    if $RANK = 0$ then
        #pragma omp parallel for private(temp)                    ▷ C notation
        for all $j$ in $nProc$ do
            for all $i$ in $nProc$ do
                $temp \leftarrow \text{MIN}(global\_load_i, load\_per\_source_{i,j})$
                $load\_per\_source_{i,j} \leftarrow temp$
                $global\_load_i \leftarrow global\_load_i - temp$
            end for
        end for
        ...
    end if
    ...
End
```

5.1.3 Parallel solution

The section began with the goal to eliminate the large array `list_per_source` and in the process transformed various parts of Algorithm 2 from serial implementation to parallel implementation. The motivation of the parallel implementation was to process the data locally on every process avoiding the need for the large temporary array. This section puts together all the parallel algorithms developed in the previous sections and develops the final parallel solution for Algorithm 2 using MPI.

Figure 24 shows the steps of the new parallel algorithm diagrammatically. The below list walks through the steps shown in diagram.

1. The local load of each process stored in the `list` data-structure is collectively added together to compute the global load of each process. Having eliminated the need for the `list_per_source` array, the collectively computed values are stored in the `list_global` array. The process is shown diagrammatically in Figure 24(a). Algorithm 4 is used to compute this global sum in parallel. See Section 5.1.1 for a detailed discussion on parallel computation of global load.

2. The local load from all process is transposed to gather load information of individual process in one place. For example, load of process 0 stored in the `list` array in all processes is gathered on process 0. Similarly, load of process 1 is gathered in process 1 and so on. This process is shown diagrammatically in Figure 24(b).

3. The global load is optimized for load balancing by the `balance_global_list` routine. See Section 5.2.1 for a detailed discussion of the existing implementation (serial) of the global load optimization process. A parallel implementation of the algorithm is developed in Section 5.2.3

4. The optimized global list is divided into chunks and scattered to other processes to update the load values locally in parallel. The process is shown diagrammatically in Figure 24(c).
5. The local load list is optimized using the global list chunk from step 4. The process is shown diagrammatically in Figure 24(d).
6. The optimized values are restored to their original process by transposing the array again. The process is shown diagrammatically in Figure 24(e).

Algorithm 6 shows the final implementation of parallel algorithm in MPI. The next section discusses the correctness of the new algorithm.

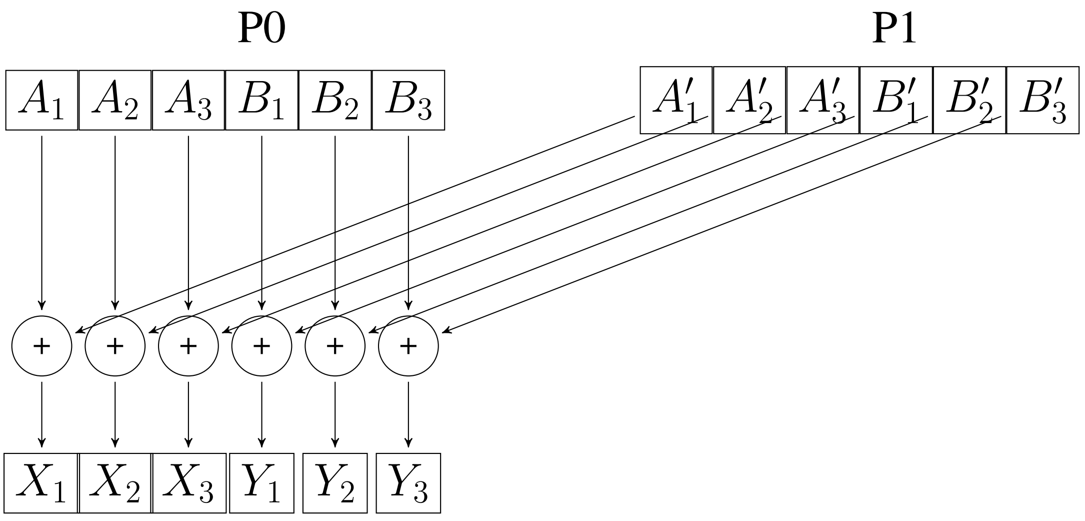
Algorithm 6 Final implementation

```
Begin
    MPI_REDUCE($list.load, list\_global.load, MPI\_SUM, \ldots$)    ▷ on RANK 0
    MPI_ALLTOALL($list.load, load\_t, \ldots$)    ▷ Transpose list
    if $RANK = 0$ then
        $list\_global.dest \leftarrow list.dest$
        BALANCE_GLOBAL_LIST($list\_global$)    ▷ Still serial
    end if
    MPI_SCATTER($list\_global.load, list\_local$)    ▷ from RANK 0
    for all $i$ in $nProcess$ do
        $new\_load \leftarrow$ MIN($list\_local_i, load\_t_i$)
        $load\_t_i \leftarrow new\_load$
        $list\_local_i \leftarrow list\_local_i - new\_load$
    end for
    MPI_ALLTOALL($list\_t, list.load, \ldots$)    ▷ Restore list
End
```

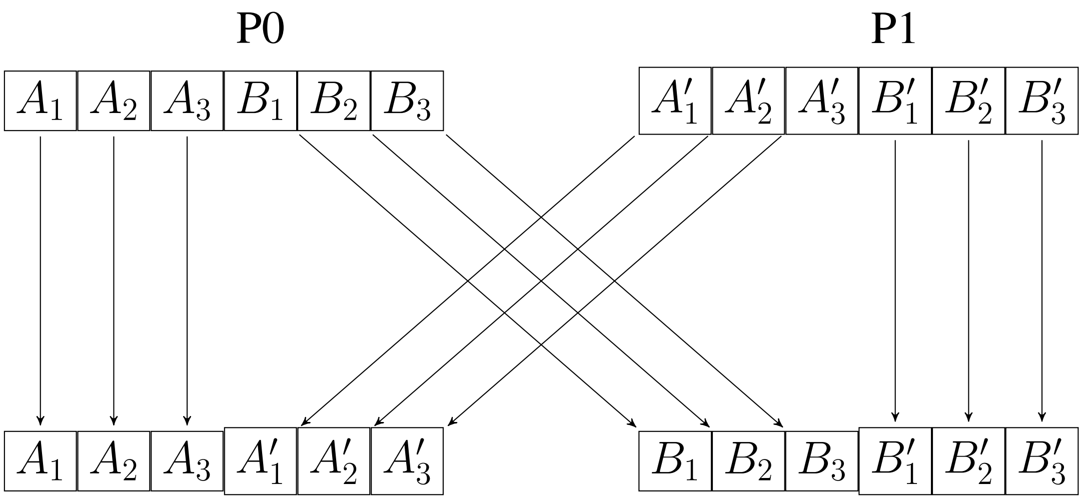
5.1.4 Checking program correctness

A good way to check program correctness is to run the modifications against a set of unit tests. Unfortunately, no automated unit tests were available for this module in CP2K. However, the software provides a function to assert run-time state of the program using the `CPPrecondition` routine. The routine takes a predicate as its argument and evaluates the condition during run-time. If the predicate evaluates to false, the program aborts and prints useful diagnostic information like the program call stack, name of the routine where the assertion failed, name of the source file containing the failed function, line number in the source file where the assertion failed, and the process Rank where the assertion failed. Figure 25 shows output of an assertion failure in process 0 in `optimize_load_list` routine at line 1066 in task_list_methods file.

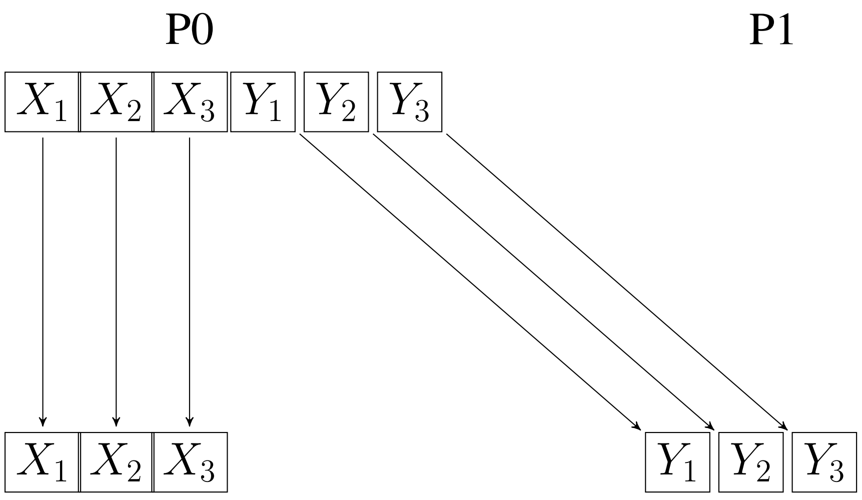
In order to check for the program correctness, output from both the existing serial implementation and the newly developed parallel implementation were compared using



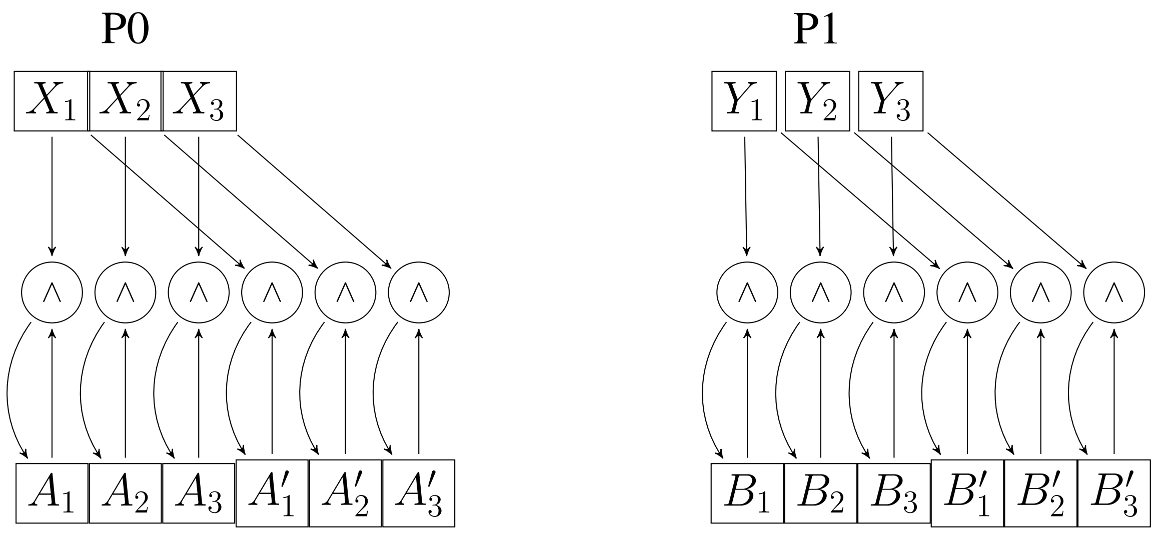
(a) Step 1: Compute global load collectively



(b) Step 2: Transpose load in the list array to collect data of corresponding processes in one location



(c) Step 4: Scatter optimized global load


P0
P1
X_1 X_2 X_3
Y_1 Y_2 Y_3
$\wedge$
A_1 A_2 A_3 A'_1 A'_2 A'_3
B_1 B_2 B_3 B'_1 B'_2 B'_3
(d) Step 5: Optimize the local load locally

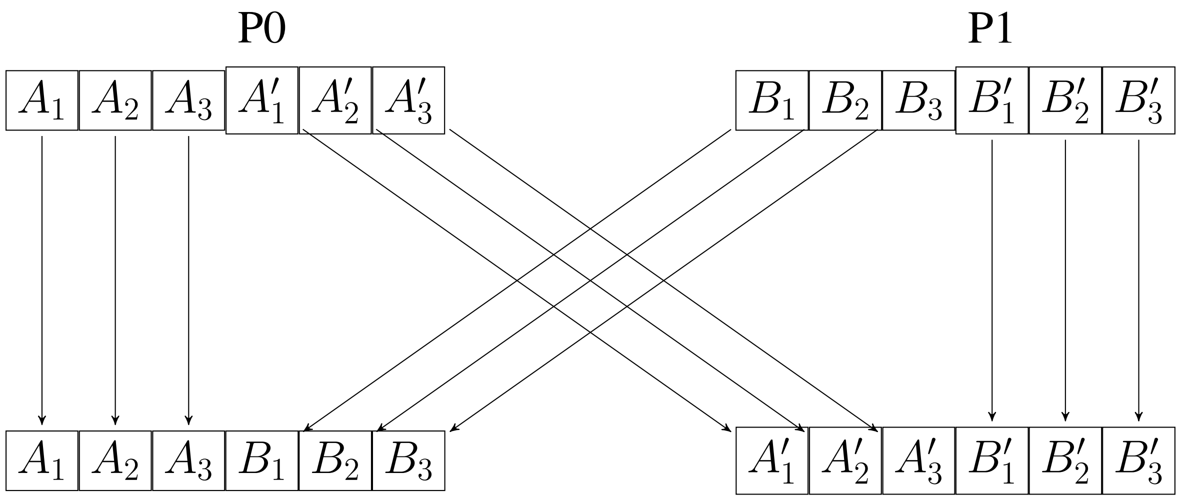
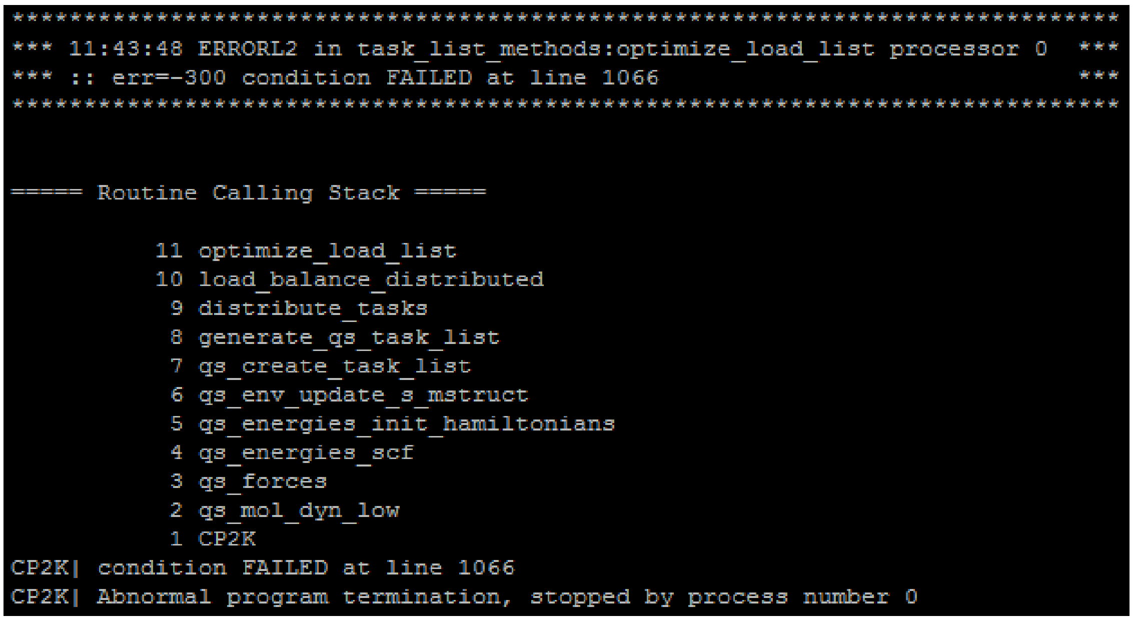

P0
P1
A_1 A_2 A_3 A'_1 A'_2 A'_3
B_1 B_2 B_3 B'_1 B'_2 B'_3
A_1 A_2 A_3 B_1 B_2 B_3
A'_1 A'_2 A'_3 B'_1 B'_2 B'_3
(e) Step 6: Restore transposed load back to orignal process

Figure 24: Final parallel algorithm. a) shows the collective computation of global load. MPI_REDUCE operation is used in the final implementation. b) shows collection of load of corresponding process on one location. MPI_ALLTOALL is used for this operation. The third step - optimizing global load is not shown here. c) shows optimized global load values scattered to other processes. MPI_SCATTER is used for this operation. d) shows the optimization of local load locally in parallel. e) shows the optimized local load values restored back to its original process.



```
******************************************************************************
*** 11:43:48 ERRORL2 in task_list_methods:optimize_load_list processor 0  ***
*** :: err=-300 condition FAILED at line 1066                              ***
******************************************************************************


===== Routine Calling Stack =====

          11 optimize_load_list
          10 load_balance_distributed
           9 distribute_tasks
           8 generate_qs_task_list
           7 qs_create_task_list
           6 qs_env_update_s_mstruct
           5 qs_energies_init_hamiltonians
           4 qs_energies_scf
           3 qs_forces
           2 qs_mol_dyn_low
           1 CP2K
CP2K| condition FAILED at line 1066
CP2K| Abnormal program termination, stopped by process number 0
```

Figure 25: Sample assertion failure report produced by CPPrecondition function.

the `CPPrecondition` routine. Listing 3 shows the code snippet comparing the output of the global load calculation using both the collective operation and the existing serial operation. Listing 4 shows the code snippet comparing the output of parallel and serial local load normalization process.

```
   load_global_sum = load_all
   CALL mp_sum(load_global_sum, 0, group) !Parallel
 summation

   IF (my_pos==0) THEN
      load_all=0
      DO icpu=1,ncpu !Serial summation
         load_all=load_all+ &
            load_per_source(maxdest*ncpu*(icpu-1) + &
            1:maxdest*ncpu*icpu)
      ENDDO
      !** Assert old and new values are matching **!
      DO icpu=1,ncpu*maxdest
         CPPrecondition( &
              load_global_sum(icpu)==load_all(icpu), &
              cp_failure_level, routineP, &
              error, failure)
      ENDDO
```

Listing 3: Testing correctness of parallel global load computation. Output of both the serial and parallel implementation is compared using the CPPrecondition function.

```
 i=1
!task performed by all process in parallel
 DO icpu=1, ncpu
   DO idest=1,maxdest
        itmp=MIN(load_local_sum(idest),load_t(i))
        load_t(i)=itmp
        load_local_sum(idest)=load_local_sum(idest)-itmp
        i=i+1
   ENDDO
 ENDDO

 !load_t is output of parallel task
 CALL mp_alltoall(load_t, load_tt, maxdest, group)
 !load_per_source is output of serial task
 CALL mp_scatter(load_per_source,load_all,0,group)

 !** Assert old and new values are matching **!
 DO icpu=1,ncpu*maxdest
    CPPrecondition(load_tt(icpu)==load_all(icpu),
  cp_failure_level,routineP,error,failure)
 ENDDO
```

Listing 4: Testing global load distribution (Serial calculation not shown).

The modifications passed all tests confirming that the new parallel implementation result confirms to that of the existing serial implementation.

5.1.5 Performance measurement

Figure 26 compares the execution timings of the new and the old implementation of the `optimize_load_list` routine . The timings are collected using the CP2K in-built profiler.

The W216 test input was used to generate the timings. It can be observed from the graph that as the number of processes increases the performance of the algorithm also increases. It can be noted that for 1024 processes the algorithm achieved a 4.75X speedup. The reason being the overhead to allocate heap memory is eliminated and the computation to calculate the global load, and optimization of the local load are collectively done by all processes. In the new implementation process time is utilized in computation rather than waiting for synchronization with Rank 0.

This completes the discussion of solution to the high memory requirement issue. The next section begins the discussion of solution to the second problem: serial task.

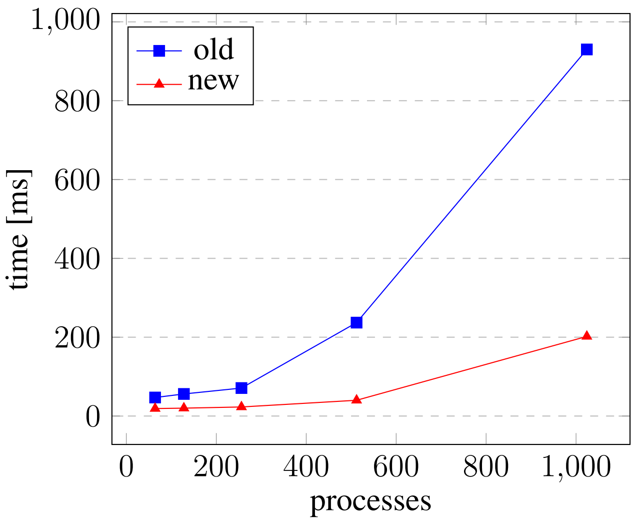


Figure 26: Comparison of old vs new implementation

5.2 Resolving serialization issues

The previous section dealt with the problem of high memory requirement for optimizing the load balance. This section shows the solution to the second problem: the serial load optimization task. The load optimization task is implemented in the routine `balance_global_list`. This is a serial routine taking $O(N^2)$ time to balance load on N processes. The serial algorithm also limits the scalability of the algorithm to more than one process.

This routine uses information of alternate task destinations and global load of all process to optimize the work load. Both the information is stored in the `list_global` data-structure. The calculation of global load is discussed in Section 5.1.1.

This section begin with an overview of the existing serial implementation and discusses various steps of the algorithm in detail. Next, a parallel implementation of the algorithm is shown along with the performance comparison of the serial and parallel implementation.

5.2.1 Existing implementation

Figure 27 shows the flowchart of the existing implementation. The algorithm begins by calculating the *flux connections* and *flux limits*. *Flux connections* are pairs of source and destination processes that can share their work load. *Flux limits* set the maximum load that could be transferred between the pairs of processes in *flux connections*. Once the *flux limits* and *flux connections* are computed, the actual optimization process begins. The process iteratively calculates the optimum flux (the load that the source can offload to the destination process) that could be transferred between the processes in the *flux connections*. The global load stored in the `list_global` is then updated with the optimum flux calculated in the previous step. This optimized `list_global` is used to update each process' `list` data-structure with the new optimum load values. The steps in the flowchart are elaborated next.

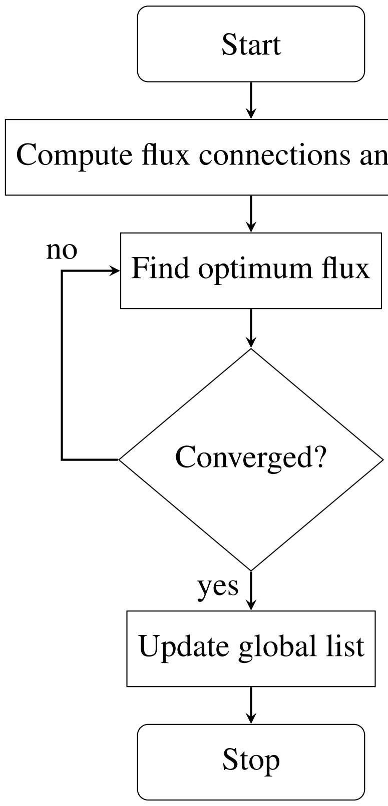


Figure 27: Flowchart for global load optimization process

Computation of flux connections and limits
This step reorganizes the data stored in the `list_global` array and arranges them in pairs of source and destination processes with their flux limits. The pairs of processes are stored in the `flux_connections` array and their flux limits are stored in the `flux_limits` array. Algorithm 7 shows the pseudo-code for computing the *flux connections* and *flux limits*. The next step uses these data-structures to iterate over the source-destination pairs and calculate the optimum flux that should be shifted for load balancing.

`flux_connections` is a two dimensional array. Its values are calculated from the source and destination information stored in the `list_global` array. For every source process, its Rank is stored in the first dimension of the `flux_connections` array and its destinations Rank are stored in the second dimension. The source-destination combination is stored only for those pairs where the destination Rank is higher than that of the source. The reason for such an arrangement is to avoid multiple execution of the load balancing task by both the source and the destination process. For example, if two processes agree to share their load then both the processes are eligible to run the load

Algorithm 7 Compute flux limits

```
Begin
    for all $src$ in $procs$ do
        for all $dest$ in $list\_global.dest$ do
            if $dest > src$ then
                $flux\_limits.src \leftarrow list\_global_{src,dest}.load$
                $flux\_connections.src \leftarrow src$
                $flux\_connections.dest \leftarrow dest$
            else
                for all $connection$ in $flux\_connections$ do
                    if $connection.dest = src$ AND $connection.src = dest$ then
                        $flux\_limits.dest \leftarrow -list\_global_{src,dest}.load$
                        $exit$
                    end if
                end for
            end if
        end for
    end for
End
```

sharing task. But that would duplicate the task execution. By putting the constraint that only the process with the lowest rank of the two processes can run the task, the duplication is eliminated.

The `flux_limits` is also a two dimensional array that stores the maximum limit of the load that can be shifted between the source and destination pairs. The flux limit for the source to destination is stored in the second dimension of the array and the flux limit for destination to source is stored in the first dimension. For ease of computation the values in the first dimension are stored as negative values. During load optimization, the source and the destination can only transfer load up-to the minimum of these two limits.

Find optimum flux

Optimum flux is the optimum load value that can be shifted from the source to its alternate destination process. It is calculated for the pairs stored in the `flux_connections` array. Algorithm 7a shows the pseudo-code for calculating optimum flux values.

The algorithm begins by calculating the amount of load that can be shifted between the pairs in the `flux_connections` array. The amount of load that can be shifted between the pairs is calculated as the minimum of the average of the difference in the global load of the source and destination process and their flux limits. For example, the amount of load that can be shift from process P0 to P1 can be calculated as $(load_{P0} - load_{P1})/2$ where $load$ is the global load of the processes. The minimum of the three values (computed load shift, source flux limit, destination flux limit) is the optimum load that can be shifted from the source to its destination. The calculated values are stored in the `optimum_flux` array which is used in the next step to optimize the global load in `list_global`.

Algorithm 7a Find optimum flux

```
Begin
    ...
    for all $connection$ in $flux\_connections$ do
        $load\_shift \leftarrow (load(flux\_connections.src) - load(flux\_connections.dest))/2$
        $load\_shift \leftarrow \text{MAX}(flux\_limits.dest - shifted_{src,dest}, load\_shift)$
        $load\_shift \leftarrow \text{MIN}(flux\_limits.src - shifted_{src,dest}, load\_shift)$
        $load(flux\_connections.src) \leftarrow load(flux\_connections.src) - load\_shift$
        $load(flux\_connections.dest) \leftarrow load(flux\_connections.dest) + load\_shift$
        $shifted_{src,dest} \leftarrow shifted_{src,dest} + load\_shift$
    end for
    ...
End
```

Update global load
Algorithm 7b shows the pseudo-code to update the global load in the `list_global` array using the optimum flux values calculated in the last step. The `src` and `dest` variables in the pseudo-code refers to the source process and its destination processes stored in the `list_global` array. As discussed earlier, process with low Rank in the source destination pair is considered the source and other the destination. The algorithm adjusts this relationship and begins the optimization process. If the optimum flux between the pair is higher than the already assigned load to the destination process, the difference in value is added to the source process and the destination process load is set the optimum flux value. Otherwise, the destination process is considered to be overloaded and nothing is offloaded to the destination process. The load previously assigned to the destination is recalled and added to the load of source process.

The output of this process, the optimized `list_global`, is used to optimize the `list` data-structure. The algorithm to update the `list` data-structure is discussed in Section 5.1.2.

5.2.2 Algorithm analysis

Algorithm 7a is analyzed in this section to identify the scope of parallelization. The Dependency analysis framework described in Section 2.2 cannot be used to analyze this algorithm because the `load` array, which is both read and written inside the loop, is indexed using a non loop control variable. However, an eyeball analysis shows that

Algorithm 7b Update global load

```
Begin
    for all $dest$ in $src.alt\_dest$ do
        if $dest.rank < src.rank$ then
            $src \leftarrow dest$
            $dest \leftarrow src$
        end if
        if $optimum\_flux > 0$ then
            $src.load \leftarrow src.load + dest.load - optimum\_flux$
            $dest.load \leftarrow optimum\_flux$
        else
            $src.load \leftarrow src.load + dest.load$
            $dest.load \leftarrow 0$
        end if
    end for
End
```

there may be a dependency on the `load` array in different iterations of the loop. For example, in every iteration of the loop a source process computes an optimum flux value using the load of its destination processes. However, the destination process itself may become a source process in some other iteration and update its load with new flux values. If this loop is executed in parallel without synchronization there will be a race condition on the `load` variable. The result will be incorrect and non-deterministic producing a hard to find bug. This imposes a dependency between the source and the destination processes calculating optimum load. Both the processes cannot be executed in parallel without interlocking.

The order of loop execution is also important. In the loop, optimum load values are calculated for all process sequentially. For example, values for process 1 is calculated first, then for process 2 and so on. A parallel execution will produce non-deterministic output.

Figure 28(a) shows the process dependency diagrammatically. The diagram shows the load optimization process between the pair of processes stored as flux connections. The example in Figure 28(a) shows the load balancing process between three processes P1, P2, and P3. The diagram shows that the optimum flux is calculated in lock-step between all the processes. The example shows P1 (source) calculating the optimum flux between itself and its destination process P2 followed by P2 (now source) calculating the optimum flux with its destination P3 in lock-step. In every step, the load of the source and the destination process is updated with new values and the next step uses the new value for computing optimum flux. This process repeats till convergence. It can be seen from the example that there is an execution-order dependency between the source and destination processes. For example, Process P2 cannot initiate its task until Process P1 has completed its. Process P2 begins its task with the updated value calculated by its source process (P1).

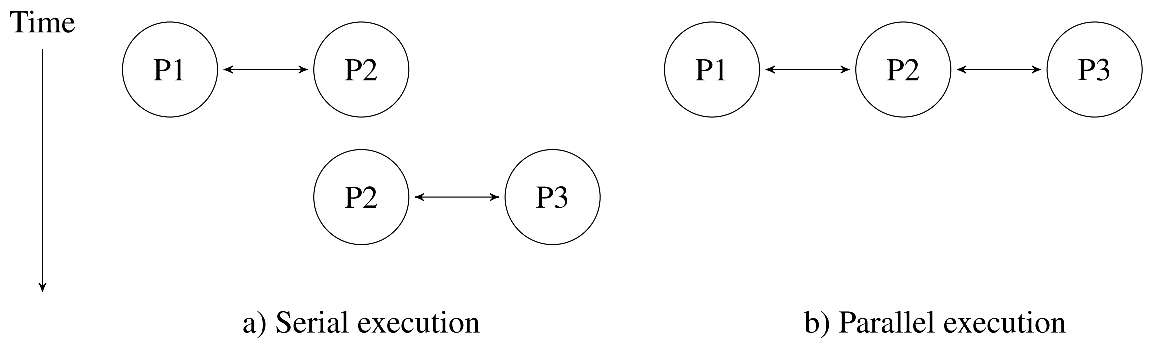


Figure 28: Algorithm execution patterns. a) shows the serial execution pattern preserving dependencies and taking more execution time compared to the parallel execution pattern. b) shows the parallel execution pattern taking less time without preserving the dependencies

As described above, the dependencies between the source and the destination processes inhibits parallelization of the algorithm. In other words, the algorithm cannot be parallelized without breaking the dependencies between the source and the destination processes. The next section describes an attempt to parallelize the algorithm without preserving the dependencies.

5.2.3 Parallel Implementation

Figure 28 compares the serial and parallel implementation of the load balancing algorithm. Figure 28(a) shows the serial implementation and Figure 28(b) shows the parallel implementation. Figure 28(b) shows that the load optimization task is executed in parallel and completes faster than the serial tasks. However, due to the dependencies between the processes some form of synchronization is required to prevent race-conditions between competing processes. A race-condition occurs when two parallel tasks updates the same memory location simultaneously. The value stored in the memory will be that of the task that last writes to the memory. Depending on the order of execution, the value may vary between different executions of the task producing incorrect results. The synchronization overhead may add additional delays in the program execution and may increase the program execution time.

The serial routine `balance_global_list` is parallelized using MPI point-to-point operations. See Section 2.1 for a brief discussion on MPI communication types. The implementation uses `MPI_ISEND` to send messages to other processes and `MPI_IREC` to receive messages from other processes. These functions executes asynchronously with the computation effectively overlapping computation and communication. The `MPI_WAIT` family of routines are used to synchronize the completion of asynchronous operations. A variation of this function `MPI_WAITALL` is used to synchronize completion of multiple asynchronous operations. The parallel routine uses `MPI_WAITALL` to synchronize various send and receive operations.

The new parallel implementation is called `balance_global_list_distributed`. Various functions of the new implementation is described next.

Calculate global load revisited
Section 5.1.1 described a parallel implementation to calculate global load. In that implementation the global load was collectively computed and the result was made available on Rank 0. The reason being the global load optimization was done serially on Rank 0. However, to parallelize the global load optimization process the global load needs to be made available on all processes. Like the previous implementation, reduction operations can be used to calculate the global load collectively. MPI provides three types of reduction operation: *MPI_REDUCE* - where the result of the operation is made available on a single process; *MPI_ALLREDUCE* - where the result is made available on all processes; and *MPI_REDUCE_SCATTER* - where operation is executed on elements of vector type and the result vector is split and distributed across processes. The *MPI_REDUCE_SCATTER* collective routine fits the task as results are made available on all processes. Algorithm 8 shows the pseudo-code for the new way of computing global load.

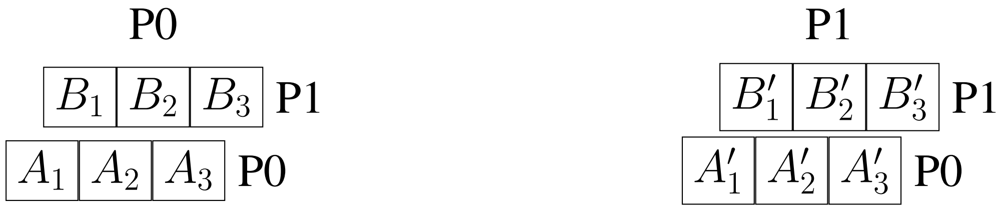
Algorithm 8 Parallel computation of global load

```
Begin
    ...
    MPI_REDUCE_SCATTER($list.load, load\_all, MPI\_SUM, \dots$)    ▷ Result on all processes
    ...
End
```

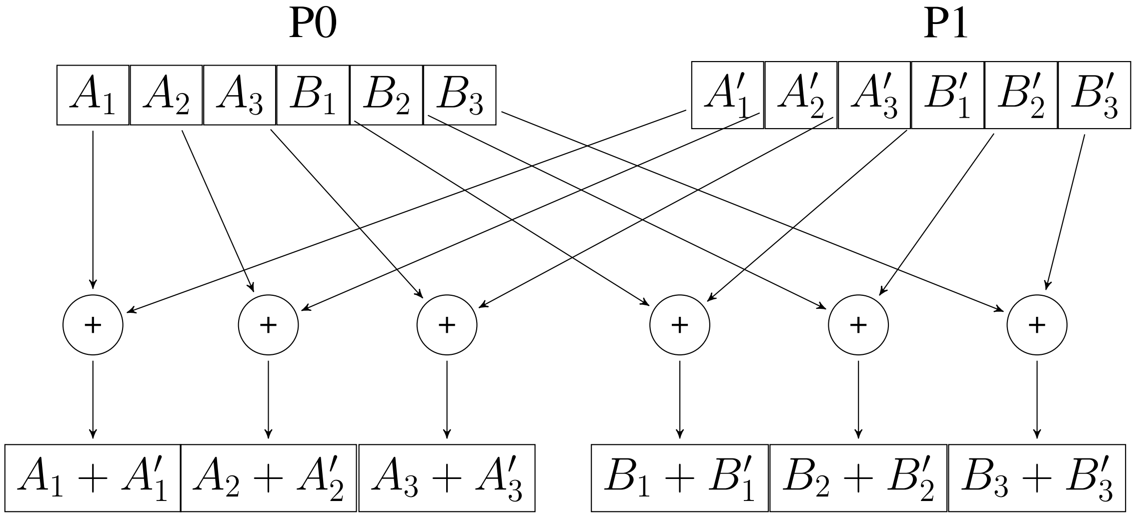
Figure 29 shows the process diagrammatically. The diagram shows the global load of respective process calculated collectively and result made available on all processes.

Calculate flux limits
With the new parallel implementation, the load balancing is done by all processes in parallel. Also, the use of the `list_global` array is eliminated and the global load information is no more accessible to the load balancing algorithm. The algorithm needs the flux limits of both the source and the destination processes. The flux limits of the destination processes are accessed by the source using the MPI message passing routines. Algorithm 9 shows the pseudo-code to transfer flux limits between the source and the destination process. As described in Section 5.2.1, the source gets the *flux limits* only from the processes with Ranks higher than its own. The *flux limits* are used in the calculation of optimum flux shown next.

Find optimum flux
Algorithm 9a shows the pseudo-code for the parallel implementation of Algorithm 7a.



(a) Load information in the list data-structure. Alternate destination information not shown.


P0
P1
A_1 A_2 A_3 B_1 B_2 B_3
A'_1 A'_2 A'_3 B'_1 B'_2 B'_3
+
$A_1 + A'_1$ $A_2 + A'_2$ $A_3 + A'_3$
$B_1 + B'_1$ $B_2 + B'_2$ $B_3 + B'_3$

(b) Collective computation of global load using MPI_REDUCE_SCATTER operation

Figure 29: Global load calculation - New method. a) shows the values of load in the list array of all processes. b) shows the collective computation of the global load values and available of the result vector on all processes.

Algorithm 9 Compute parallel flux limits

```
Begin
    for all $alt\_dest$ in $list_{dest}$ do
        if $alt\_dest > src$ then
            MPI_IRECV($limit, \ldots, alt\_dest, \ldots$)
        end if
        if $alt\_dest < src$ then
            MPI_ISEND($list_{load,dest}, \ldots, alt\_dest, \ldots$)
        end if
        if $Sending$ then
            MPI_WAITALL($\ldots$)
        end if
        if $Receiving$ then
            MPI_WAITALL($\ldots$)
        end if
        if $Receiving$ then
            $flux\_limits_{dest} \leftarrow limit$
        end if
    end for
End
```

The routine calculates the load shift values called optimum flux which are used to optimize the global load of all processes. An optimum flux is calculated between the source and destination pairs of processes stored in the `flux_connections` array. The source calculates the optimum flux and sends it to the destination processes to update their load.

The optimum flux values are calculated iteratively by shifting the load between the source and the destination processes and recording the amount of load shifted in each iteration. The amount of shift calculated is the minimum of the average of the difference in the load of both the source and the destination process and their flux limits. For every iteration, the load on both source and destination are updated and exchanged between the source and the destination process. The data is exchanged using the `MPI_ISEND` and `MPI_IRECEIVE` routines. It can be noted that access to the load variable is synchronized by the send/receive operation. The race condition identified in Section 5.2.2 is eliminated by this synchronization. The algorithm iterates for `Max_iter` times, which is set to 100, or till the maximum amount of load shift calculated in the iteration is less then the tolerance which is set to to 0.1% of the average load of entire system.

Optimize global load

The optimum flux calculated in the previous section is used to update the global load. Algorithm 9b shows the parallel implementation of the Algorithm 7b. As discussed earlier, the source process (process having Rank less than its destination process) calculates the optimum flux and updates the load on both process. In parallel implementation, the

Algorithm 9a Parallel load optimization

```
Begin
    for $m \leftarrow 1$ to 100 do                                    ▷ iterate max 100 times
        for all $alt\_dest$ in $list_{dest}$ do
            if $alt\_dest > src$ then
                MPI_IRECV($load_{dest}, \ldots, alt\_dest, \ldots$)
            end if
            if $alt\_dest < src$ then
                MPI_ISEND($load_{src}, \ldots, alt\_dest, \ldots$)
            end if
            if $Sending$ then
                MPI_WAITALL(...)
            end if
            if $Receiving$ then
                MPI_WAITALL(...)
            end if
            if $alt\_dest > src$ then
                $load\_shift \leftarrow (load_{src} - load_{dest})/2$
                $load\_shift \leftarrow$ MAX($flux\_limits_{dest} - shifted_{src,dest}, load\_shift$)
                $load\_shift \leftarrow$ MIN($flux\_limits_{src} - shifted_{src,dest}, load\_shift$)
                $max\_shift \leftarrow$ MAX($max\_shift, load\_shift$)
                $load_{src} \leftarrow load_{src} - load\_shift$
                $shifted_{src,dest} \leftarrow shifted_{src,dest} + load\_shift$
                MPI_ISEND($load\_shift, \ldots, alt\_dest, \ldots$)
            end if
            if $alt\_dest < src$ then
                MPI_IRECV($load\_shift, \ldots, alt\_dest, \ldots$)
            end if
            if $Sending$ then
                MPI_WAITALL(...)
            end if
            if $Receiving$ then
                MPI_WAITALL(...)
            end if
            if $alt\_dest < src$ then
                $load_{src} \leftarrow load_{src} - load\_shift$
            end if
        end for
        MPI_ALLREDUCE($max\_shift, MPI\_MAX, \ldots$)
        if $max\_shift < tolerance$ then
            $exit$
        end if
    end for
End
```

calculated optimum flux needs to be communicated with the source and the destination process. The source process updates its global load using the optimum flux calculated by both the source and the destination process.

Algorithm 9b Optimize global load

```
Begin
    ...
    for all $dest$ in $src.alt\_dest$ do
        if $dest.rank < src.rank$ then
                                            ▷ Append to optimum flux
            MPI_IRECV($\ldots, optimum\_flux, \ldots, alt\_dest, \ldots$)
        end if
        if $dest.rank > src.rank$ then
            MPI_ISEND($optimum\_flux, \ldots, alt\_dest, \ldots$)
        end if
    end for
    for all $dest$ in $src.alt\_dest$ do
        if $dest.rank < src.rank$ then
            $src \leftarrow dest$
            $dest \leftarrow src$
        end if
        if $optimum\_flux > 0$ then        ▷ optimum flux between src and dest pair
            $src.load \leftarrow src.load + dest.load - optimum\_flux$
            $dest.load \leftarrow optimum\_flux$
        else
            $src.load \leftarrow src.load + dest.load$
            $dest.load \leftarrow 0$
        end if
    end for
    ...
End
```

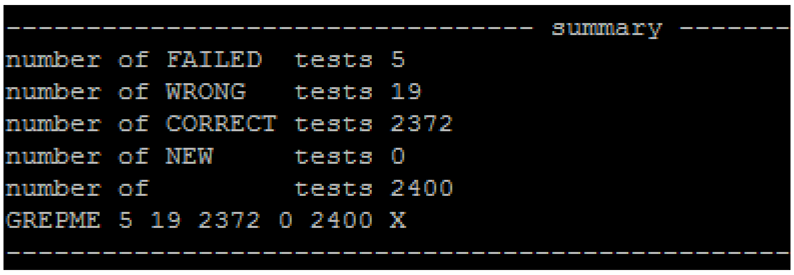
5.2.4 Checking program correctness

As discussed in Section 5.2.2 the serial loop is parallelized without respecting the dependency constraints. As a result, the result of parallel execution is non-deterministic and does not match with that of the serial execution. Table 3 shows the variations in output of serial and parallel execution. The output is generated using the aforementioned H2O-32 input file using 8 process and the values are shown from the first step of the simulation. The first column shows the values of `optimum_flux` computed in serial and the second column shows the values computed in parallel. The Delta column shows the difference in the output between serial and parallel computation. To further check the effect of the variations in the result, the program was tested using the regression test suite that comes with CP2K.

Serial implementation	Parallel implementation	Delta
4167.3203125	4189.83644104	22.51612854
15590.33984375	15579.08177948	11.25806427
21992.16015625	21983.2381896973	8.9219665527
18329	18329	0
-16150	-16150	0
10868.34375	10796.181640625	72.162109375
11489.671875	11469.896484375	19.775390625
-16640	-16640	0

Table 3: Values of optimum_flux variable using serial and parallel execution.

CP2K comes with over 2400 test inputs regression test suite that ensure that all parts of the code are working correctly. The final parallel implementation was tested using this utility on ARCHER. Figure 30 shows the output of the regression tests. 2372 regression tests passed out of 2400 tests, 5 failed and 19 were reported as wrong. However, the tests of unmodified code also failed in same places which confirms that the failures are not related to the modifications done in this project. The main reason for the failed tests are due to some issues with the Intel Math Kernel Library implementation and the wrong results reported are due to precision errors (at 10^{-14} decimal digit) in calculation which is within the acceptable limits.

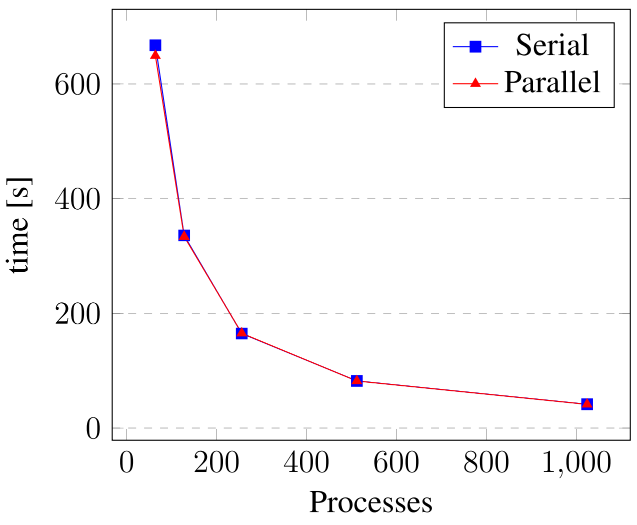


```
-------------------------------- summary -------
number of FAILED  tests 5
number of WRONG   tests 19
number of CORRECT tests 2372
number of NEW     tests 0
number of         tests 2400
GREPME 5 19 2372 0 2400 X
------------------------------------------------
```

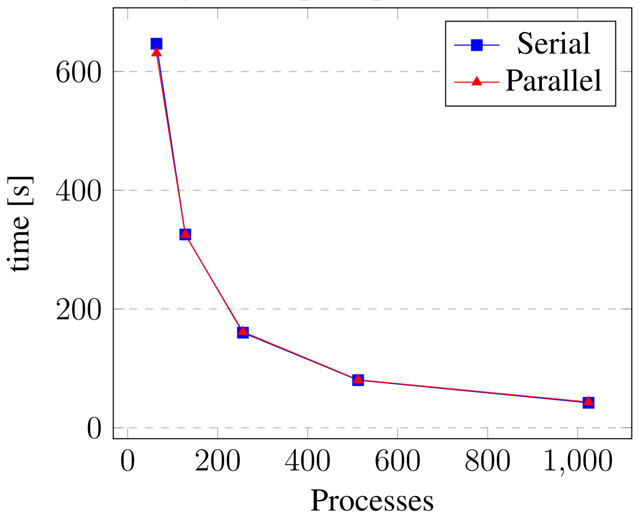
Figure 30: Output of CP2K regression test utility

5.2.5 Result comparison

As discussed in Section 3, the goal of the load balancer is to optimize the choice of the task destination in the collocation step. So a better test would be to check for the performance of the serial and the parallel load balancing algorithm in the collocation step. Figure 31 compares the result of load balance on the collocation and integration steps using the serial and parallel load balancing algorithms.Figure 31(a) shows the comparison of integration step and Figure 31(b) shows the comparison of collocation step. The execution timings of `calculate_rho_elec` and `integrate_v_rspace` routines are used for comparison. The output was generated using the W216 test input file. Figure 31 confirms that both serial and parallel algorithms produces similar results.



(a) Integration step comparison



(b) Collocation step comparison

Figure 31: Performance comparison of the serial and the parallel load balancing algorithm.

5.2.6 Performance measurement

The metrics used to evaluate performance of the parallel implementation are the *Speedup ratio* and *Parallel Efficiency*. Speedup gives the relative performance improvement of the serial task when executed in parallel and Parallel Efficiency shows the scalability of the algorithm. Speedup is given as

$$S_p = \frac{T_s}{T_p}$$

Where:

p:	is the number of processors
T_s:	is the execution time of the sequential algorithm
T_p:	is the execution time of the parallel algorithm with p processors

The Parallel Efficiency is

$$E = \frac{S}{P}$$

Where:

S:	is the Speedup
P:	is the number of processors

An algorithm is said to be scalable if the parallel efficiency is an increasing function of N/P, the problem size per node.

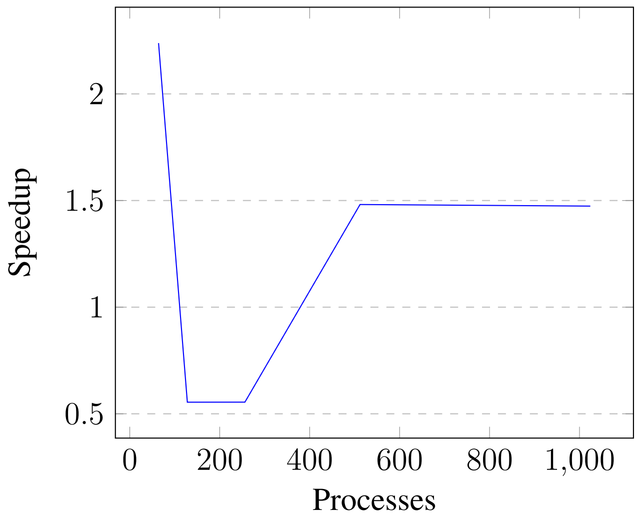


Figure 32: Speedup of parallel implementation

Algorithms showing linear Speedup are considered good as the algorithm scales well with increase in the number of processes. Figure 32 shows Speedup of the parallel load optimization routine implemented in the `balance_global_load_distributed` routine. It can be observed that the maximum Speedup achieved by the algorithm is 2.5X and with increase in the number of processes the Speedup further dropped down.

The performance penalty in parallel implementation is due to the overhead of communication and synchronization between the source and the destination processes. In the serial implementation, the algorithm executes seamlessly without any interruption while in the parallel implementation there is an explicit synchronization between processes after the end of every iteration. After every iteration, processes block in wait state to send and receive updated load values. The algorithm also uses a collectively reduction operation to calculate the maximum load shifted in the iteration. The amount of data sent between processes per transaction is also very low (4 bytes). As a result the start-up cost (latency) dominates the communication cost and the achieved bandwidth is also very low. Also, the communicated data is used only in a single arithmetic operation which implies a very high communication overhead for a single operation.

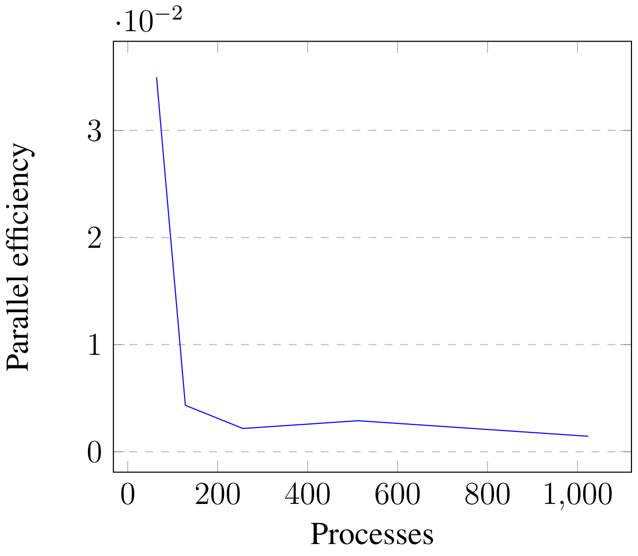
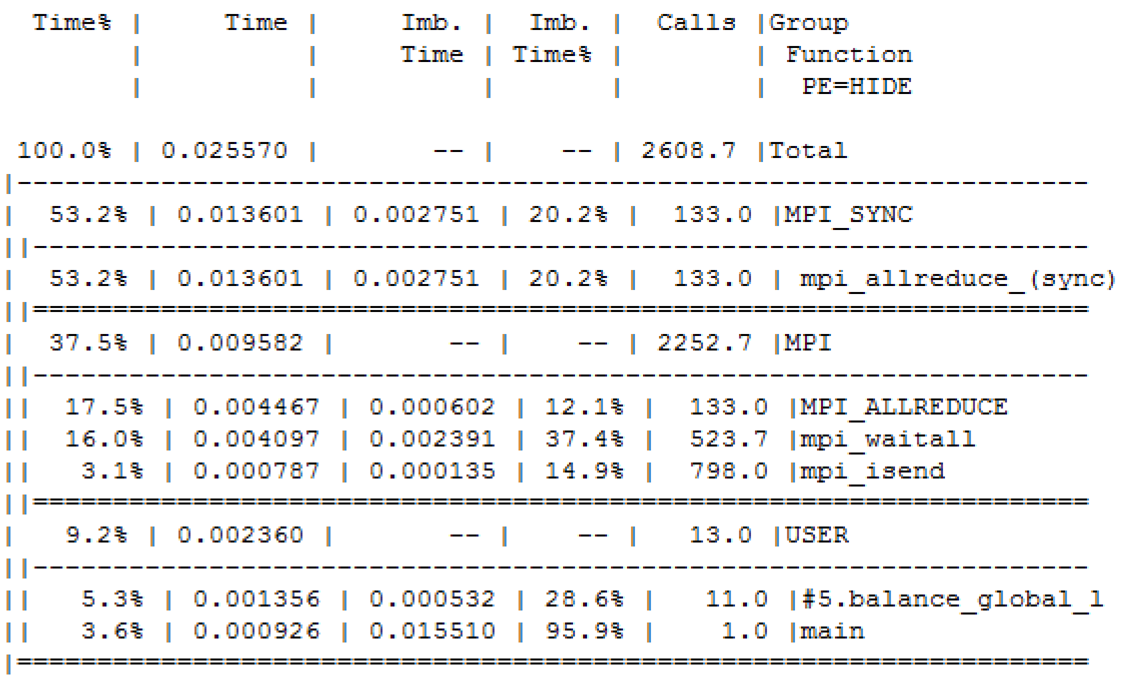


Figure 33: Parallel efficiency

Figure 33 shows the Parallel Efficiency of the new algorithm. An algorithm that scales linearly has a parallel efficiency 1. It can be observed that the new algorithm is not scalable and the efficiency drops as more number of processors are added.

In order to find the bottleneck, the code was instrumented with CrayPAT (Cray Performance Analysis Tool) region. The resulting output is shown in Figure 34. It can be noted that 53% of the total execution time is spent in the blocked state (synchronizing collective reduction operation). As described earlier, the synchronization happens in the MPI_ALLREDUCE collective which is used to calculate the maximum load shifted by all processes in an iteration. The value is used to check for convergence to terminate loop. A possible solution (to reduce the synchronization wait time) would be to reduce the number of calls to MPI_ALLREDUCE. The convergence could be checked on every other iteration or after some fixed number of iterations. However, the modifications were not carried out in this project and the option is kept open for future improvements.

The next expensive operation after MPI_ALLREDUCE is MPI_WAITALL used to synchronize the completion of the asynchronous send and receive operations. 16% of the execution time is spent waiting for completion of the asynchronous operation. Moreover, the collective routine call is highly imbalanced at 37%. The implementation also incurred a 38% overhead for parallelizing the algorithm using MPI. Clearly, due to the overheads of parallel implementation and high communication over computation ratio the parallel implementation did not performed well.



```
  Time% |     Time |     Imb. |  Imb. |  Calls |Group
        |          |     Time | Time% |        | Function
        |          |          |       |        |  PE=HIDE

 100.0% | 0.025570 |       -- |    -- | 2608.7 |Total
|-----------------------------------------------------------------
|  53.2% | 0.013601 | 0.002751 | 20.2% |  133.0 |MPI_SYNC
||----------------------------------------------------------------
|  53.2% | 0.013601 | 0.002751 | 20.2% |  133.0 | mpi_allreduce_(sync)
||================================================================
|  37.5% | 0.009582 |       -- |    -- | 2252.7 |MPI
||----------------------------------------------------------------
||  17.5% | 0.004467 | 0.000602 | 12.1% |  133.0 |MPI_ALLREDUCE
||  16.0% | 0.004097 | 0.002391 | 37.4% |  523.7 |mpi_waitall
||   3.1% | 0.000787 | 0.000135 | 14.9% |  798.0 |mpi_isend
||================================================================
|   9.2% | 0.002360 |       -- |    -- |   13.0 |USER
||----------------------------------------------------------------
||   5.3% | 0.001356 | 0.000532 | 28.6% |   11.0 |#5.balance_global_1
||   3.6% | 0.000926 | 0.015510 | 95.9% |    1.0 |main
|=================================================================
```

Figure 34: Output of CrayPAT Profiler. The output is generated for H2O-32 input file using 64 processes.

6 Retrospective

6.1 Project Goals and Objectives

The scope of this project was to:

1. Resolve high memory requirement of the load balancing module
2. Parallelize the load optimization algorithm

The project has successfully achieved the set objectives.

6.2 Major Milestone Achievement

Appendix A shows the project Gantt chart prepared during the project preparation stage. The project was started earlier than planned. Planned tasks were also completed before the estimated duration. Overall the project completed before the estimated finish date. Table 4 shows major milestone information.

Milestone	**Planned completion date**	**Revised completion date**	**Actual completion date**
Design Review	07/07/2014	07/02/2014	06/21/2014
Implementation Review	07/19/2014	07/14/2014	06/27/2014
Report Review	08/11/2014	08/06/2014	08/04/2014

Table 4: Major milestone achievement

6.3 Risk Management

Figure 35 shows the Risk Impact Matrix for the key risks identified in the project preparation stage. The highly probable and high impact risk "Insufficient domain background" never materialized due to execution of proper risk mitigation strategies developed in the project preparation stage. Strategy of getting timely help from the project supervisor helped to keep the project on track without any major issues. Similarly, the medium probability risk "Insufficient experience in programming language used" was mitigated by referring to Fortran class notes and online tutorials. The low probability risk "Code loss due to hardware failure" also never materialized. However, the risk mitigation strategy of using a distributed version control software (GIT was used) was already in place.

Some of the risks that materialized during the project execution stage were "Supervisor unavailability" and "ARCHER unavailable". Both the risks were identified as low priority risks. The project supervisor was unavailable for two weeks during the project execution but the impact on the project was negligible as most of the coding work was already completed. Had he not been available during the initial phase of the project the impact on project schedule would have been profound. Similarly, ARCHER was not available on a weekend due to insufficient time budget. This occurred during collection of program run-time data. However, the impact of the risk on the project schedule was negligible as the task was rescheduled and another non-dependent task was scheduled instead.

Refer Appendix A for the Risk Register developed during the project preparation stage.

6.4 Lessons Learnt

Table 5 list out few activities which could have been carried out differently for smooth execution of the project. One item that stands out in particular is the approach taken

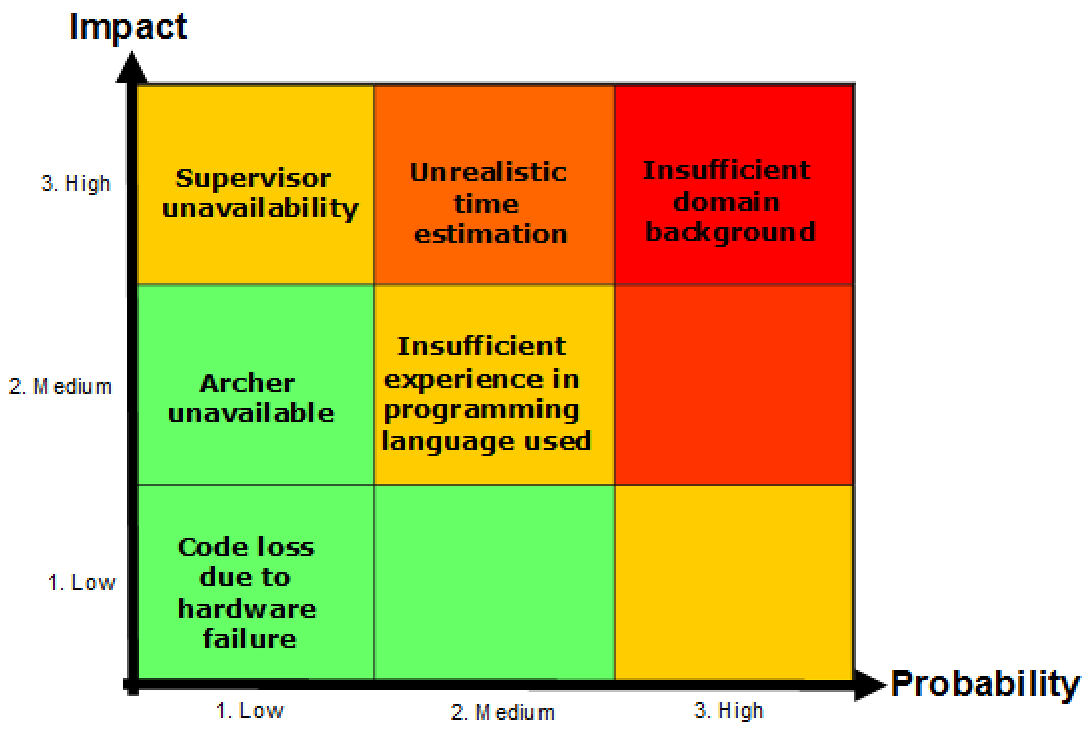


Figure 35: Risk impact matrix

to understand the existing code. The taken approach was to understand the solution by working through the code. A better approach would have been a top-down approach where the entire algorithm and flow of information was understood before working with the code. This would have further reduced the development time.

Lesson No	Lesson Description	Suggested Future Action
1	Cultural difference - No communications on weekends and holidays	Have communication policy in place
2	Resource Constraint - low priority job queue on ARCHER	Give high priority queue for time consuming tasks like regression testing, code profiling, etc.
3	Understand the problem top-down instead of bottom-up	Start with high level overview of the existing solution and then move into code details.

Table 5: Could do better

Table 6 list out few activities which helped in smooth execution of the project. Project meetings were effective in monitoring and mitigating project risks like "Insufficient domain background" through constant discussions and feedbacks. Project preparation stage helped in understanding the project in greater detail before the start of the execution stage.

Lesson No	**Lesson Description**	**Suggested Future Action**
1	Weekly meetings	Increase duration - settling time needs to be accomodated
2	Project Preparation	Increase meeting frequency - project initiation phase needs new information to be explored through discussions
3	Use of version control tool to manage code changes	Use proper change descriptions (in log file) for ease of restoration and merge task

Table 6: Worked well

7 Conclusions

The project set out to reduce the memory footprint of the load balancing module and parallelize the load balancing algorithm.

The module used large memory to gather global load information for optimizing the load on all processes. The high memory requirement limited the scalability of the algorithm as the memory requirement increased quadratically with the number of processes. This resulted in turning the load balancing module off on high number of processes where the load balancing would be critical for efficient use of all computing resources. The algorithms were modified to calculate the global information locally and collectively thus eliminating the need for the large memory. As a side effect of the modification, the performance of the algorithm also improved due to parallelization of operations computing global information.

The next improvement in the list was to parallelize the algorithm optimizing the load on all processes. The analysis of the algorithm showed an execution order dependency in computation of optimum load values. It was shown that non-deterministic order of parallel process execution will result in variation of output generated using the parallel and serial implementation. The algorithm was parallelized and as proved during the analysis the output of parallel and serial execution were different. However, the variation was not very significant and was also shown to be giving same load balancing effect on test input.

The performance measurement of the parallel implementation showed very poor performance due to high number of communication of small size message (4 Bytes) and low computation. Moreover, there was also synchronization overhead in every iteration that further degraded the performance.

Finally, both the goals were achieved but due to the poor performance of the parallel implementation of the load balancing algorithm it is recommended to use the serial implementation till a better solution is not found.

A Project Plan

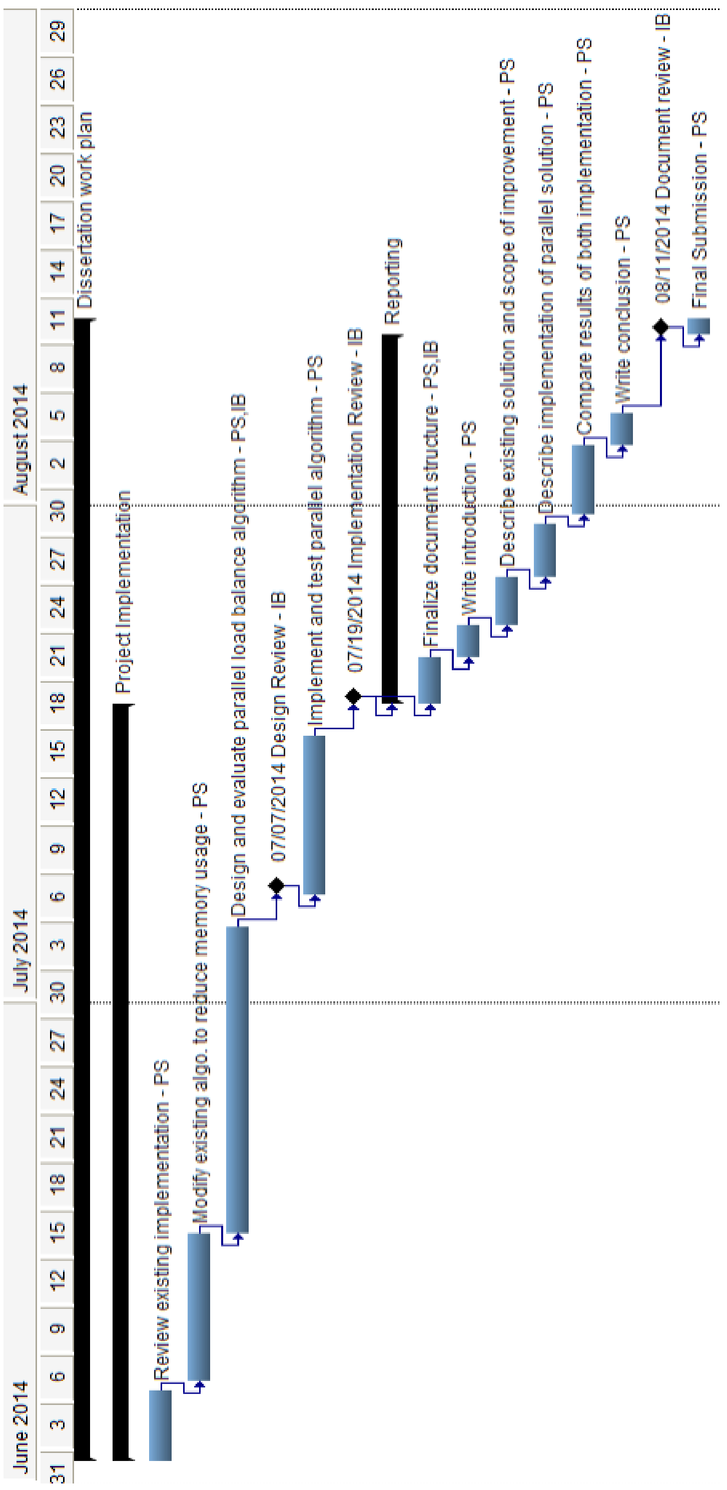


Figure 36: Project Gantt chart

Risk Register

Sr. No	Risk	Response	Probability	Impact
1.	Unclear requirement	Properly document the requirement and get approved by the Supervisor	V. LOW	HIGH
2.	High level of technical complexity	Get help from supervisor	HIGH	HIGH
3.	Unrealistic time estimation	Get project plan approved by the Supervisor	LOW	HIGH
4.	Poor project planning	Get the plan approved by the Supervisor	LOW	HIGH
5.	Developing the wrong software function	Get design reviewed by the Supervisor	V. LOW	V. HIGH
6.	CP2K coding standard not followed	Get code reviewed by Supervisor	V. LOW	V. LOW
7.	Supervisor unavailable	Communicate using emails, phone, Skype etc.	LOW	V. HIGH
8.	Student unfamiliar with problem domain	Get help from Supervisor	V. HIGH	V. HIGH
9.	Student not experienced in programming language used	Refer to Fortran class materials and lab exercises	LOW	MEDIUM
10.	Student absent from the project due to illness	Get extension from University	LOW	V. HIGH

References

[1] CP2K website, http://www.cp2k.org

[2] CP2K science showcase, http://cp2k.org/science

[3] J. VandeVondele, M. Krack, F. Mohamed, M.Parrinello, T. Chassaing and J. Hutter. *Quickstep: fast and accurate density functional calculations using a mixed Gaussian and plane waves approach* Comp. Phys. Comm. 167, 103 (2005)

[4] I. Bethune. *Improving the performance of CP2K on HECToR, A dCSE Project*, http://www.hector.ac.uk/cse/distributedcse/reports/cp2k/cp2k_final_report.pdf

[5] ARCHER website, http://www.archer.ac.uk

[6] ARCHER hardware, http://archer.ac.uk/about-archer/hardware/

[7] MPI website, http://www.mpi-forum.org/

[8] Patterson, David A. and Hennessy, John L., *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers Inc., 1990

[9] Blelloch, Guy E, *Vector Models for Data-Parallel Computing*, MIT Press, 1990

[10] David A. Patterson and John L. Hennessey, *Computer Organization and Design: the Hardware/Software Interface*, Morgan Kaufmann Publishers Inc., 1998

[11] Quinn Michael J, *Parallel Programming in C with MPI and OpenMP*, McGraw-Hill Inc., 2004

[12] OpenMP website, http://openmp.org/wp/

[13] Parallel Computing, http://en.wikipedia.org/wiki/Parallel_computing

[14] SIMD architecture, http://en.wikipedia.org/wiki/SIMD

[15] Task parallelism,
wiki.expertiza.ncsu.edu/index.php/CSC/ECE_506_Spring_2010/
ch_2_aj/Data_Parallel_Programming

[16] Joost VandeVondele, *CP2K: parallel algorithms*, http://www.cscs.ch/uploads/media/CP2K_parallel.pdf

[17] Kennedy, Ken; Allen, Randy, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*, Morgan Kaufmann, 2001.

[18] Paul Feautrier, *Encyclopedia of Parallel Computing*, Springer, 2011