Engineering with Computers (2024) 40:1927–1942
https://doi.org/10.1007/s00366-023-01890-z

ORIGINAL ARTICLE

General resource manager for computationally demanding scientific software (MARE)

Xinchen Guo[1,2] · **James Charles**[1,2] · **Namita Narendra**[1,2] · **Gerhard Klimeck**[1,2,3] · **Tillmann Kubis**[1,2,3,4,5]

Received: 3 October 2022 / Accepted: 21 August 2023 / Published online: 3 October 2023
© The Author(s), under exclusive licence to Springer-Verlag London Ltd., part of Springer Nature 2023

Abstract
Today's supercomputers power scientific calculations in very different areas, ranging from nanotechnology to climate studies and astronomy. Research groups in nascent areas of science and engineering develop their own scientific software, since it cannot be obtained otherwise. These groups are typically not funded to spend time on performance optimization and software design, but to address their original research questions. In more established fields, several accepted research applications emerge as a standard and get adopted by research groups that are users, with little software development experience. The variability of applications yields a hard to optimize and sub-optimal supercomputer resource usage. Large computing times or large memory requirements often limit the scope of the research exploration and a more optimal use can reduce overall compute time or increase the scientific scope of questions being asked. This work introduces a resource manager that requires minimal intrusion and only superficial understanding of the scientific code, but still schedules the execution of the calculations to optimally utilize memory, central processing unit (CPU) or both. This is exemplified with Quantum Espresso and recursive open boundary and interfaces (ROBIN) calculations on regional and national computer infrastructures. The resource manager presented here reduces compute times by 13–30% in those two scientifically relevant computational codes.

Keywords High-performance computing · CPU · Memory · Scientific software · Resource management

1 Introduction

Since the first top 500 listing, the processing power of supercomputers has increased by 5–6 orders of magnitude [1, 2] and computer simulations have become important research tools [3, 4]. Efficient design of scientific simulation tools has become increasingly challenging as the complexity and hardware has changed significantly. The community has evolved from adapting codes from single processor vector machines [5], to shared memory supercomputers [6], to small Beowulf clusters [7], massive clusters [8], and eventually to massive clusters with significant co-processors [9]. Recently, the memory systems have also become more complex with high-bandwidth memory [10] and non-volatile memory [11]. Adapting research code to specific hardware environments is extremely labor intensive and is not a viable option for most scientific codes, given the variability of existing and future systems. Even well-established codes, like Quantum Espresso, do not have a software team that is dedicated to performance optimization on every computing cluster a user of the code has access to. We believe that self-adapting resource management will be the path forward to fully utilize the available hardware resources without altering the actual research code.

✉ Xinchen Guo
guo351@purdue.edu

1 Elmore Family School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907, USA

2 Network for Computational Nanotechnology, Purdue University, West Lafayette, IN 47907, USA

3 Purdue Center for Predictive Materials and Devices, Purdue University, West Lafayette, IN 47907, USA

4 Purdue Institute of Inflammation, Immunology and Infectious Disease, Purdue University, West Lafayette, IN 47907, USA

5 Purdue Quantum Science and Engineering Institute, Purdue University, West Lafayette, IN 47907, USA

1.1 State-of-the-art cluster-level and node-level resource management

The resource management in supercomputers has two levels: the cluster level and the node level. On the cluster level, job

scheduling tools (e.g., PBS [12] and SLURM [13]) allocate computing resources (e.g., individual nodes) to each simulation task and focusing, e.g., on maximal overall system utilization. On each node, the operating system (OS) manages the computing and storage resources.

The OS manages memory and CPU usage [14–16]. However, scientific software developers and users are expected to explicitly limit the peak memory usage below the node memory [17]. Explicit and flexible memory management is challenging given the wide range of memory per CPU core, e.g., 1.4 GB/core on the Stampede 2 KNL sub-cluster [18] and 6.4 GB/core on the Halstead cluster at Purdue University [19].

The OS manages the CPU resources using a time-sharing method [20–22]: time slots are allocated among active processes to ensure fair sharing [23–25]. Context switches, switching between different processes for the same CPU core, can happen regularly which causes overhead. Most of the large clusters we are aware of, and certainly the most massive clusters assign a CPU or a full node to one specific user with one specific executable to minimize the context switching. When a node gets exclusively assigned to a specific research software, that software is expected to match the number of active threads to the available CPU cores to maximize CPU utilization while minimizing context switching.

1.2 Motivation and requirements for a general resource management tool

Scientific simulations often have memory usage variations [26, 27] and load imbalance [28–31]. Adding dynamic memory and CPU management to existing software would typically require fundamental code modifications. Such modifications are time intensive and error prone. Resource management can be difficult for non-specialist scientific software developers [32–34]. Therefore, we introduce a general resource manager that treats existing software as black boxes and requires only minimal code intrusions.

There are many workflow management tools to combine multiple programs and coordinate their execution such as SWIFT [35], Pegasus [36], HTCondor [37], ANSYS workbench [38], Galaxy [38], etc. These high-level tools analyze the input–output dependencies and generate a directed acyclic graph and execute programs accordingly. However, they lack the dependency management on the iteration level which is fundamentally required to control memory usage. The capability to adjust CPU usage is not in the scope of the existing tools.

We envision and demonstrate a different approach which uses available OS facilities, does not require expertise in performance optimization, nor access to administrative privileges. The OS provides a tool set that in general allows to manage resources in high detail. To make this available to computational researchers, these tools have to get combined into a usable application. This application, the general manager of resources (MARE) has to be flexible to accommodate any resource scenario computational scientists may encounter. Users of MARE must be able to program the resource management control.

This paper presents MARE, the resource manager that combines abstractions of the resource management with a programmable workflow management. MARE allows researchers who are not specialized in performance optimization to easily optimize computational resource usage. MARE treats software as black boxes. This allows users to optimize the resource usage of scientific software without knowing its details. One key innovation of MARE is ensuring optimal CPU usage with dynamic and task-adaptive multithreading. Our MARE requires no system administration privilege.

The reminder of the paper is organized as follows: Sect. 2 presents the design details of MARE. Sections 3 and 4 show two application examples to demonstrate several concrete and typical resource usage problems that can be solved with MARE. Conclusions are presented in Sect. 5.

2 Framework design and key features

The MARE framework consists of three major components: manager, agent, and client. First, the MARE manager starts on a single node. Second, on each available compute node one MARE agent is started. During the agent start, the location of the MARE manager is provided to the agent. The last step of the initialization has each application software register an associated client to the respective MARE agent. During runtime, each client monitors its application software status by instrumenting the scientific software and communicates the status repeatedly via its agent to the MARE manager (see Fig. 1). Depending on the management policy and the received information, the MARE manager dynamically adjusts the assigned resources for each application.

2.1 Manager design and implementation

MARE requires only a single manager that runs on either a compute node or a login node. The manager receives application software status updates from all agents via text files stored in the virtual file system of the RAM. These text files are monitored by inotify of Linux, which notifies the manager about new status files available to be processed. The MARE manager processes concurrent status files in the order it received the inotify notification. Scientific application software often requires multiple compute nodes. Shared network filesystems like NFS [39], Lustre [40], GPFS [41],

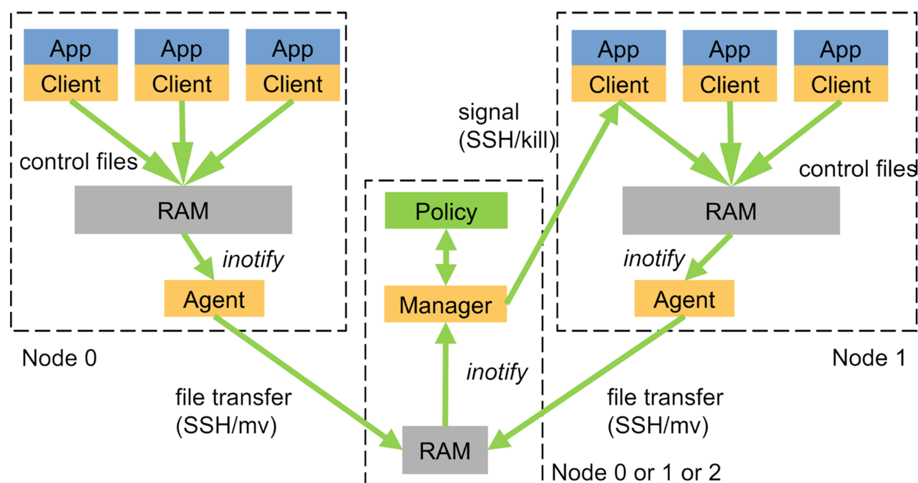


Fig. 1 The software architecture of the MARE resource management framework, containing clients, agents, and a manager. The manager can share a computing node (0 or 1 in the figure) with the simulation code or be hosted on a dedicated node (2 in the figure). The manager communicates with each client but only one arrow to the first client on Node 1 is shown in the figure for cleanness

etc. are mounted to compute nodes. However, MARE agents do not move the status files to a network filesystem, since the inotify feature is not sufficiently supported on network filesystems [42, 43]. Instead, the widely available SSH tool is used to transfer application software status files to the local filesystem of the manager. With proper caching of the SSH authentication, the latency of sending one status file is as low as 2 ms. This enables a management granularity (solver/iteration) of the MARE framework in the order of seconds.

The management policy is contained in a function that is executed every time a new status file is detected. Any kind of status information or metadata of the application software process can be input to the policy function. This can include from CPU or memory load, process information, application data, or even user input. The manager also provides adjustment APIs for the policy function that can wrap OS details that would require advanced OS knowledge. Application software status updates direct the manager via the policy function to make adjustments. Example policies for synchronization and dynamic CPU usage adjustments are provided in MARE.

2.2 Agent design

An agent is a thin layer needed on each compute node to handle the network communication with the MARE manager. The agent uses the inotify feature to monitor application software status files. The event-driven design of inotify avoids unnecessary file observation which saves CPU resources. The agent forwards new status files to the MARE manager.

2.3 Client design and implementation

Each application software process has an exclusive associated client. The client connects the application software process with the agent. The client interacts via two interfaces. The first interface consists of control points in the source code of the application software. These control points allow the client to monitor the application status. Code locations around MPI function calls are natural candidates of control points. The client writes the status updates and process metadata into the status files that are forwarded by the agent to the manager.

The second interface provides dynamic CPU resource adjustment. During the client's initialization, a signal handler in the framework is registered to receive SIGUSR1 and SIGUSR2 signals from the OS. The signal handler will automatically adjust the number of threads (CPU cores) used by the application software according to the received signals from the manager.

Since the framework is written in C++ and expected to work with various software, a version of client-side APIs without name mangling [44] is created for C and Fortran. Additionally, a special wrapper interface (MEX) is also created for the popular engineering software Matlab and Octave.

2.4 Design considerations

The MARE framework stores and transfers application software status between the different MARE components as text files. Using the file system rather than direct communication (pipeline or socket) decouples each component of the MARE framework for the highest flexibility. This design makes the client an optional component in the MARE framework. An application software can avoid the client by generating the MARE status files by itself.

2.5 Example policies and control points

Different software applications have different resource usage patterns. The MARE resource manager is designed to be

Table 1 Potential control point location for different type of scientific simulation

Simulation	Control point	Potential sources of load imbalance
Computational fluid dynamics	End of subdomain	E.g. grid refinement, inner iterative solver [45]
N-body	End of subdomain	E.g. inherent limitation of geometry based domain decomposition method [46]
Weather forecast	End of subdomain	E.g. localized weather phenomena, different physics [47] solvers [48]

highly customizable and programmable via user-defined policies. A few policy examples are provided that address typical resource usages. The following application examples illustrate the management of CPU load and the simultaneous management of memory and CPU load. Similar policies can be imagined to control the memory bandwidth, network bandwidth, energy consumption, allocation budget on community clusters, etc.

Many scientific simulations are done iteratively. Within one iteration, the problem is divided into smaller chunks for parallel processing. An ideal location for control points is the end of each chunk before the synchronization preparing for the next iteration. Table 1 shows potential control point locations for different types of simulations in addition to the nanoscale simulation examples demonstrated in the later sections.

3 Policy example: CPU load

To exemplify a policy that manages dynamic multithreading, the ab-initio code Quantum Espresso (QE) [49–51] is considered when applied on a heterojunction of two 2D materials. The code is run on Purdue University's Halstead cluster [19, 52] for performance testing. Halstead consists of two 10-core CPUs with 128 GB of memory per node on a 100 Gbps Infiniband network.

3.1 Application example: structure optimization with Quantum Espresso

Figure 2 illustrates the application example of a monolayer MoS_2/WS_2 heterostructure. The electronic properties of the heterostructure are calculated in the density functional theory framework of the plane wave-based QE. The QE software is written in Fortran language and had been augmented to utilize the MARE client APIs following Sect. 2.3.

Figure 3 shows the simplified software flow of a typical electronic-structure calculation with QE which consists of an outer structural relaxation and an inner electronic relaxation loop [53]. The ions are moved according to the forces on the ions that depends on the electronic density, which in turn depends on the ion positions. Due to their mutual dependence, ionic positions and electronic density have to

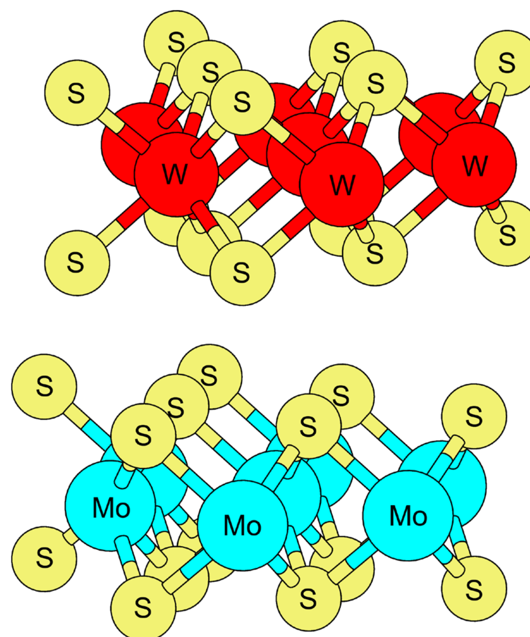


Fig. 2 Illustration of the monolayer MoS_2/WS_2 heterostructure system considered in the Quantum Espresso example

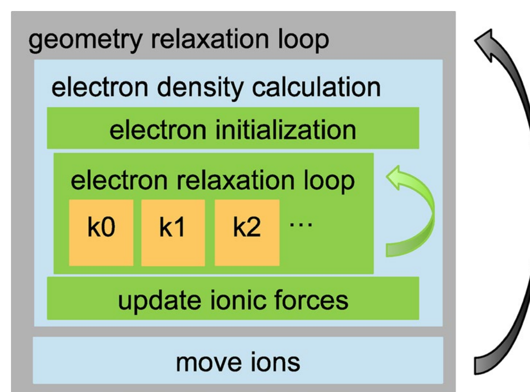


Fig. 3 Schematic of the program flow for the Quantum Espresso structure optimization. The inner self-consistent loop performs the electronic density calculation that involves solving for the mutually independent momentum point (labeled with "k0", "k1", etc.) contribution to the density. The calculation of the k-point contributions get repeated until a converged electronic density is found. The outer loop iterates between ion positions and electronically mediated forces. The outer loop continues until global convergence is reached

be solved iteratively until convergence is achieved. Most of the simulation time is spent in the solution of the momentum (k) point contributions to density, which gets iterated often due to the self-consistency loops. For this example, a single electronic density calculation with fixed ionic positions is used. The resource management would be the same for relaxation of ionic positions.

3.2 Without MARE: load imbalance

The solution of k-points is independent from each other and can run trivially parallel. However, since the number of required k-points is determined by the modeling problem, it can easily be incommensurate with the number of available CPU cores per node. This leads to load imbalance, as also illustrated in Fig. 4 for this example with 13 independent k-points and 20 CPU cores per compute node. In this case, ideal CPU usage cannot be achieved when the number of threads for each process is statically set by environmental variables.

3.3 Multithreading management by the operating system

One way to improve the CPU usage is increasing the number of threads per process from 1 to 2, which leads to an oversubscription of 26 computing threads for the 20-core CPU. Modern multi-use and multi-program operating systems are capable of handling such oversubscriptions with context switching [54].

The CPU usage with context switching shown in Fig. 5 is indeed higher than in the situation without shown in Fig. 4. Consequently, the end-to-end time improved by 197 s. However, the context switching also causes a very frequent oscillation of CPU usage per process as expected (see Fig. 6), which creates some CPU load by itself.

This is illustrated in the CPU usage of 1 process using 2 threads shown in Fig. 6. The OS-controlled context switching causes the CPU usage to oscillate between 1 and 2.

Typically, to minimize context switching, the number of running processes or threads has to match the number CPU cores. The amount of processes is typically determined prior to the simulation by specifying the amount of MPI ranks. The amount of threads is determined either prior to the simulation by environmental variables or during the simulation by explicit API calls.

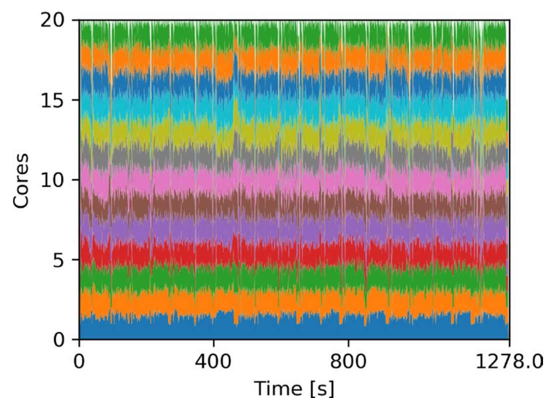


Fig. 5 CPU usage of the example simulation of Fig. 4 with 2 computing threads per process and context switching of the OS. 13 k-points correspond to the 13 colors and their computation is spread out over 20 cores. The CPU usage and simulation time improve by 197 s down to 1278 s from 1474 s shown in the single threaded case of Fig. 4, but a highly frequent oscillation of CPU usage is found

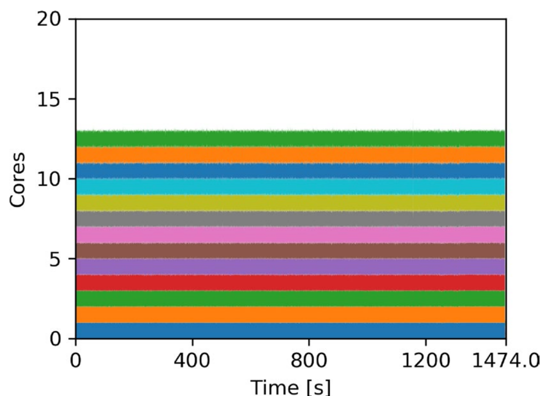


Fig. 4 CPU usage of the Quantum Espresso example of Fig. 2 with 13 k-points and 13 MPI processes. Multithreading is not enabled to avoid oversubscription of the 20 available CPU cores. This setup leaves 7 CPU cores unused. Here, each color corresponds to one k-point and one MPI process

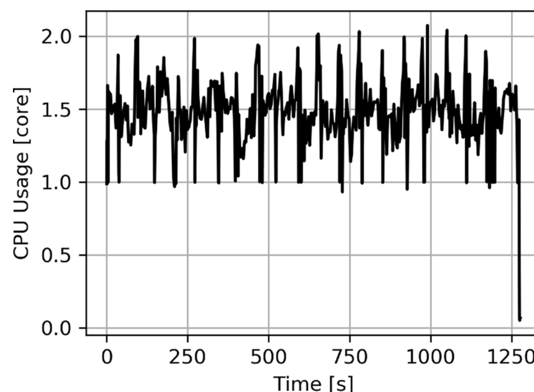


Fig. 6 CPU usage of 1 process with a 2 threads per process in the oversubscription situation of the example calculation in Fig. 5 averaged over subsequent time intervals of 1.7 s. The usage frequently oscillating between 1 and 2 due to frequent context switching by the OS

3.4 Management policy for CPU load

The overhead created by this oscillation can be avoided when information of the application software is used to dynamically adjust the number of threads by the MARE framework following a CPU load management policy. Processes with smaller and/or fewer computational tasks complete the computation sooner. The CPU resources of those processes are then reclaimed by the MARE manager and redistributed to processes with remaining tasks. Since threads share their memory, making the required data available to the newly assigned CPU cores does not require explicit data movement. As a result of this policy, tasks with higher computational load get effectively assigned a larger number of cores which improves the overall CPU load balance. To enable core reassignment, the application software needs to be augmented with a control point that (1) reports to the manager the computational task is done (see Fig. 1) and (2) puts the corresponding process into sleep mode. When all processes reach that barrier, the CPU resource assignment is then reset to default and the processes are signaled to proceed. In the example of Quantum Espresso, “proceeding” means to continue calculation of further self-consistency iterations.

Many large-scale software face similar load imbalance that this policy of dynamic multithreading can address. Even unforeseen scenarios can be improved due to the dynamic nature of MARE’s thread control. This policy essentially requires multiple processes being hosted on the same compute node to benefit from the shared memory situation. Therefore, it becomes more effective with a larger number of cores per compute node are available.

3.5 MARE control points in Quantum Espresso

The example of Fig. 2 requires Quantum Espresso to trigger a global collective MPI communication each after electron initialization, k-point computation, and update of the force field. In addition to the multithreaded computation of k-points discussed above, also the electron initialization and the update of the force field can be solved with multiple threads. Thereby, electron initialization, k-point computation, and update force field are logic segments that are independently CPU-load-optimized by MARE. Accordingly, at the end of each of these segments and right before the MPI communication, an MARE control point is inserted (see Fig. 7). It is worth to mention the additions of these control points, including corresponding interfaces to the numerical library, are the only code alteration needed in Quantum Espresso to use MARE. This minimal intrusion makes it easy for simulation developers to adapt MARE for their needs.

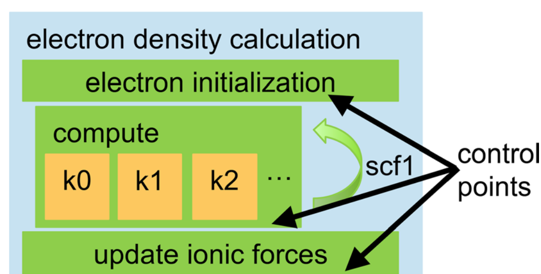


Fig. 7 Control points required for the electron density calculation. For each electron density iteration, MPI collective communication is required after initialization, computation of k-points and update of the force field. The MARE manager dynamically optimizes the CPU usage for each code segment individually

3.6 Dynamic multithreading

The goal of this CPU load example of the MARE resource manager is to dynamically adjust the number of threads assigned to each process to ensure that the available CPU cores are always fully utilized. Here, the 13 processes that handle the 13 considered k-points are initialized with 1 thread each. Then, 7 out of the 13 processes are assigned one additional thread to fully utilize the total 20 CPU cores. These 7 processes with 2 threads finish their calculations earlier than the other 6 processes that have only 1 thread each. Once the 7 double-threaded processes reach their MARE control points, they signal the MARE manager which in turn redistributes the overall 20 CPU cores to the remaining 6 processes as equally as possible. This redistribution of CPU cores gets repeated until all processes have reached the same control point (as detailed in Table 2). Then, the thread assignment is reset by the MARE manager for the next computational segment.

3.7 Assessment of performance improvement

Figure 8 shows the CPU usage of the same Quantum Espresso simulation with the MARE manager optimizing the CPU utilization. All available 20 CPU cores are almost fully utilized during the simulation time. Compared to the case with no multithreading (in Fig. 4), the end-to-end timing reduces from 1470 to 1177 s with a speedup factor of 1.25×. MARE’s dynamic thread control still accelerate the end-to-end calculation by 1.08× compared to the case of constant multithreading (in Fig. 5).

3.8 MARE in Quantum Espresso with multiple parallelization levels

A Quantum Espresso example that simulates the energy barrier of vacancy diffusion in Si [55] is used to demonstrate

Table 2 Number of threads assigned to each of the 13 processes which each solve one specific k-point

Step	k0	k1	k2	k3	k4	k5	k6	k7	k8	k9	k10	k11	k12
Step 1	2	2	2	2	2	2	2	1	1	1	1	1	1
Step 2	0	0	0	0	0	0	0	4	4	3	3	3	3
Step 3	0	0	0	0	0	0	0	0	0	5	5	5	5

The MARE resource manager dynamically adjusts the assigned number of threads for each k-point according to the work load. In this specific example, MARE adjusts the thread numbers for the k-point calculation in three steps

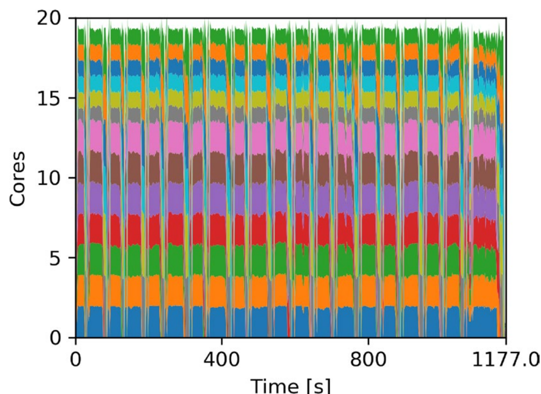


Fig. 8 CPU usage of the same simulation example and setup as Fig. 4 with the MARE resource manager. The bottom 7 colors represent k-points with 2 initial threads, whereas the top 6 colors represent k-points with 1 initial thread. The available 20 CPU cores are fully utilized. The spike-like CPU usage changes indicate reassignment of idling CPU resources to the remaining active k-points close to the end of each simulation iteration (illustrated in Table 2). The simulation time improves by 297 s down to 1177 s from 1474 s shown in the single threaded case of Fig. 4

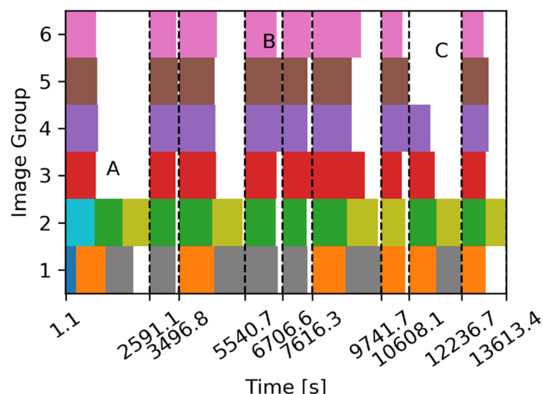


Fig. 9 Workload distribution among image groups without resource control by MARE. Each of the 10 colors represents the calculation of a specific image. The completion of iterations of all images are marked with dashed lines. White areas represent idling ranks. The letters A, B, and C highlight the 3 different types of load imbalance encountered in this example

the effectiveness of MARE in scenarios with multiple parallelization levels. A CPU load management policy is applied to an example of Nudged Elastic Band (NEB) [56, 57] calculation in QE. The NEB simulation tool is built on top of the electronic structure computing engine exemplified in Fig. 3. The NEB method divides the simulation into independent images in one iteration. One image consists of the electronic structure computation discussed in Sect. 3.1. This simulation example contains 10 independent images with the image freezing optimization [58] enabled. In this demonstration, the simulation is allowed to run for 9 iterations. Some images are calculated fewer than 9 times as discussed in detail below. At minimum, two MARE control points per MPI ranks are utilized. One control point signals the start, while the other signals the completion of an iteration. Concretely, in this example, 12 MPI ranks are used to solve 6 images simultaneously with each image supported by 2 ranks. More control points could be used for each individual image calculation which is equivalent to the example in Sect. 3.5. However, for simplicity in demonstration, only two control points at the image level are applied. The Gilbreth cluster [59] at Purdue University with 24 CPU cores and 196 GB memory per node is used for the demonstration.

This simulation setup exemplifies different types of load imbalance in NEB simulations.

- Uneven distribution of images among image groups: In this example, some image groups have to solve 2 or 3 images, while the rest solve only 1. This causes idling ranks and gaps in the workload distribution of type A in Fig. 9. While this type of imbalance is under user control, it too often is inevitable due to simulation setup needs.
- Different workload of the images of each iteration: Different images cause different numerical load until their converge (see B in Fig. 9). This difference is unpredictable and varies between different iterations.
- Frozen images in one iteration: The freezing image optimization avoids iterations for images with small errors. While this reduces the numerical workload of an image group, it causes load imbalance between image groups (see C in Fig. 9).

Once the MARE control points at the end of each iteration signal the MARE manager the completion of an image

group, the MARE manager can redistribute the available CPU cores to other image groups still busy with the electronic structure calculation. This CPU core distribution is reset at the beginning of each iteration. This method is applicable to the 3 types of load imbalances discussed above. As a result of the MARE management, Fig. 10 shows 1.19× speedup for this example.

4 Policy example: memory and CPU load

Simulations of the recently developed ROBIN method [60] are used to exemplify a policy that optimizes memory and CPU load on two different HPC systems: (1) the Rice cluster [61] at Purdue University with 20 CPU cores and 64 GB memory per node and (2) the Intel KNL partition on the Stampede 2 cluster [18, 62] at Texas Advanced Computing Center (TACC) with 68 cores and 96 GB per node.

4.1 ROBIN simulations of materials

State-of-the-art material simulations require periodicity at the simulation domain boundaries. While these boundary conditions are typically appropriate for ideal solid-state materials, realistically fabricated material samples typically host defects, dislocations, and irregularly strained interfaces. The recently developed Recursive Open Boundary and INterfaces (ROBIN) method allows the explicit discretization of millions of atoms and solution of material properties within a center region of the discretized atom pool (see circle and zoom-in in Fig. 11). ROBIN does not constrain the simulation area with any symmetry. This allows accurate prediction of disordered materials and interfaces and avoids artificial periodicity assumptions. It was shown in Ref. [60] the periodicity assumption can even yield an erroneous band gap when the actual disordered material is gapless (see Fig. 4 in Ref. [60]). Solving the impact of the environment on a center simulation region requires partitioning the atoms and iterating quantum transport equations over the segments (see Fig. 11).

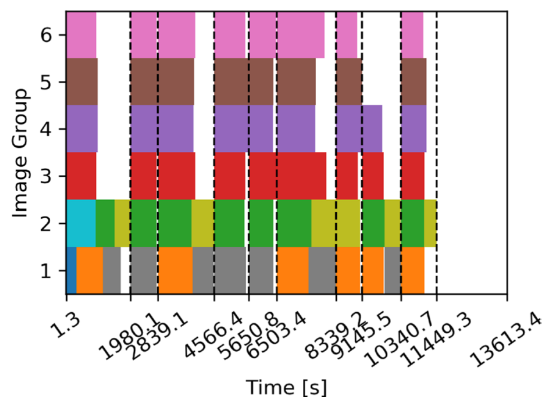


Fig. 10 Workload distribution among the image groups of Fig. 9 with resource control by MARE. All colors and dashed lines have the same meaning as in Fig. 9. Significant reduction of idling ranks (white space) is achieved. The total simulation with MARE takes 11449.3 s, which is 1.19× faster than the 13613.4 s of simulation time without MARE

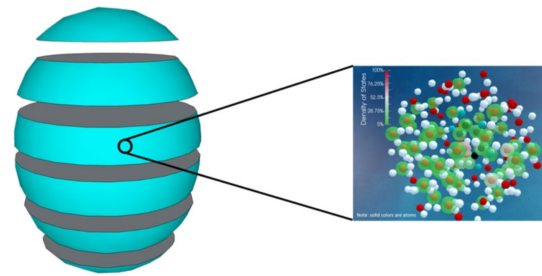


Fig. 11 Schematics of the simulation region partitioning, which is modeled as a round sphere with a large number of atoms. Recursive Green's function method compatible partitioning requires every segment to have at most two neighbor segments. The memory load of each segment scales squared with the number of atoms in it

4.2 Peak memory usage

Depending on the overall distribution of discretized atoms and the partition strategy, the number of atoms in each segment can vary significantly (see Fig. 11). The memory load of the quantum transport equations scales squared with the number of atoms in each respective segment. Since the quantum transport equations are iterated over the segments, the peak memory usage varies a lot with simulation time (see Fig. 12).

The example of Fig. 12 solves the quantum transport equations of ROBIN for two different energies in sequence. The calculation of each energy point yields a maximum

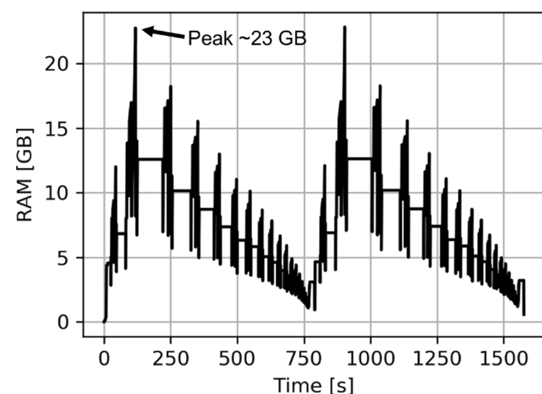


Fig. 12 Memory usage of an example calculation of the ROBIN method for two electronic energies solved in sequence with 20 threads used

memory usage of 23 GB and sees the highest usage plateau at 13 GB. As common for quantum transport simulations, different electronic energies can be solved independently in parallel. In production run ROBIN simulations, hundreds to thousands of energy points [63, 64] are solved that scale embarrassingly.

The parallelism at the level of energy points is preferred to the lower level parallel computation within energy points because of better resource utilization efficiency and throughput discussed in the later section. Given the computational behavior of each energy point is similar regardless of the specific energy value, computing two energy points (with 10 threads each on a node of 20 CPU cores) simultaneously results in almost doubling the peak memory usage of Fig. 12 (i.e., above 32 GB), as shown in Fig. 13a. The more efficient CPU commensurate calculation scenario of ROBIN would involve four simultaneous energy point calculations (with 5 threads each), which then exceeds the node's memory limit.

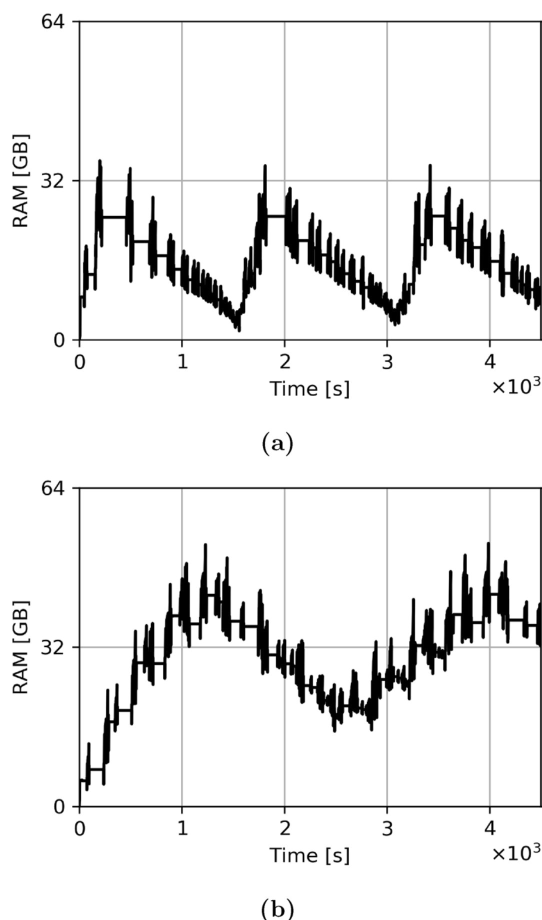


Fig. 13 Memory usage of the calculation of 8 energy points with multithreading to fully utilize 20 CPU cores. (**a**) 2 parallel processes with 10 threads each. (**b**) 4 parallel processes with 5 threads each enabled by the MARE manager. The MARE framework enables the simultaneous calculation of 4 energy points with the memory consumption under the system limit

Swap space is designed to absorb the memory usage bursts. However, swapping, even briefly, is typically turned off on HPC systems to avoid expensive page faults [17]. Frequent page faults lead to memory trashing which significantly slows down application software [17]. The users of HPC systems are expected to explicitly limit the memory usage to the amount of available RAM. Therefore, keeping the peak memory usage of the ROBIN simulation under the system limit may cause reduced parallelism and CPU underutilization. Otherwise, the simulation may have reliability (crashes due to out-of-memory) issues.

4.3 Management policy for memory and CPU load

This example management policy provides a synchronization mechanism (semaphore) to manage the dependencies among software processes. Two functions are provided for processes to acquire and release resources. The wait function allows a process to sleep and is controlled via signals from the manager. In this example, wait is applied as long as required resources are unavailable. The signal function reports the application progress to the resource manager. In this example, the signal function reports the completion of a task which triggers the manager to redistribute resources with the wait function. Both wait and signal are executed at control points added to the respective application software. This allows pipelining MPI processes to control the application software progress and make it utilize all available machine resources constantly.

It is generally important that the application software is free from race conditions [65, 66]. For the MARE management tool, it is even essential that all application software is free of race conditions. The concept of MARE ensures that it does not introduce any race conditions by itself.

4.4 Control points with the MARE framework

To enable MARE's control of ROBIN, two control points are added to the ROBIN source code: the first control point is added at the beginning of the simulation. At this point, the MARE manager gets signaled via the agent that the code execution has reached this position and the software process is put into sleep state. It will wait for a wake-up signal from the MARE manager. The second control point is added to the source code position of peak memory usage (see Fig. 12). Depending on the MARE manager, this procedure can yield a delay between two consecutive processes. Such delay can be desired to avoid peak memory usage beyond the machine's limit. The client monitors whether its corresponding host software reaches the second control point and reports the software status to the manager. Then, the manager sends the wake-up signal to the next process to enter the execution pipeline (see Fig. 14). The pipelined execution

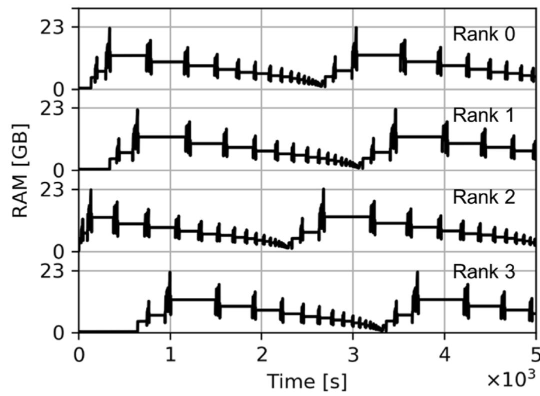
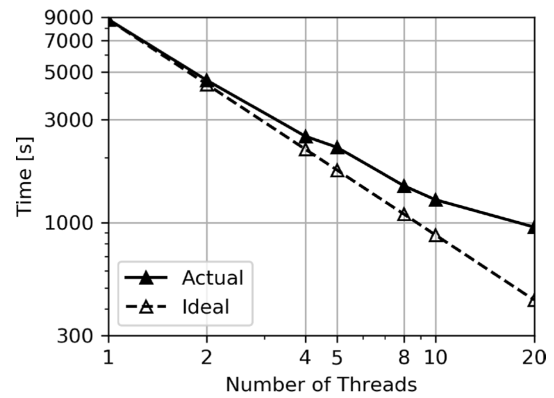
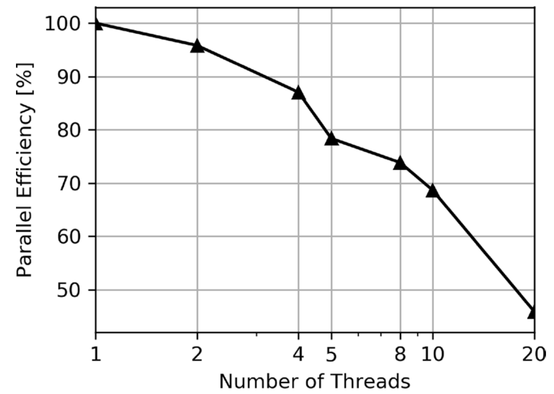


Fig. 14 Memory usage of 4 processes running the ROBIN application simultaneously. The control points added to the ROBIN code enable the delayed start to pipeline the peak memory usage



(a)



(b)

Fig. 15 Multithreading (**a**) strong scaling and (**b**) parallel efficiency of the ROBIN simulation for one energy point on a compute node with 20 CPU cores on the Rice cluster

enables smoother and under-the-limit memory consumption with the parallel computation of 4 energy points on a computer node shown in Fig. 13b, which results in faster simulation because of higher resource utilization efficiency.

4.5 Memory management to enable efficient simulation setup

MARE can enable running 4 processes in parallel: the example in Fig. 13b uses the MARE resource manager to pipeline the parallel calculation of four energy points. By properly pipelining the 4 processes, the MARE resource manager successfully limits the overall peak memory usage below the system capacity of 64 GB. Additionally, the MARE resource manager enables less memory usage oscillation to achieve better overall resource utilization.

The multithreading parallel efficiency is another optimization factor that impacts the end-to-end application timing. Figure 15a shows the strong scaling of one energy point of ROBIN with respect to the number of parallel threads on one Rice node with 20 CPU cores. The end-to-end timing deviates the more from the projected ideal scaling line the more threads are used.

A more efficient CPU usage would prefer running the ROBIN code with as few threads as possible but as many processes per compute node as possible. MARE's pipelining of processes discussed in the previous section enables more parallel processes which results in more efficient CPU usage and shorter end-to-end timing. Since the solution of different energy points scales embarrassingly parallel, we consider in the following the time-to-solution per energy point.

The change of this time-to-solution per energy point is shown in Fig. 16. The black (gray) line in Fig. 16 shows the memory usage over time when four (two) processes run in parallel per node of the Rice cluster. It is worth to repeat that only MARE enables running 4 processes simultaneously

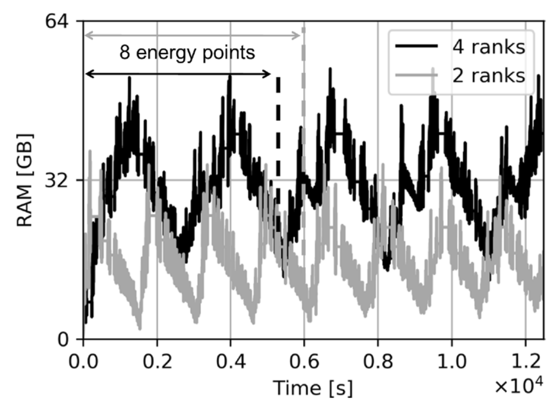


Fig. 16 Memory usage on a Rice compute node of the example ROBIN simulation (multithreading enabled). The black line shows the memory usage of 4 simulation processes with 5 threads each using the MARE framework. The gray line shows the memory usage of 2 simulation processes with 10 threads each, i.e., the optimal setting without the MARE framework. MARE enables more efficient resource usage and a shorter time-to-solution

below the system's limit. With 4 processes per compute node, the number of threads per process needed to maximally utilize the Rice CPU reduces from 10 to 5. The parallel efficiency improves from 68.7 to 78.4%. That is the reason the MARE framework effectively enables processing four energy points every 2937 s (734 s per energy point). In contrast, without MARE, one compute node can process two energy points every 1588 s (794 s per energy point). Using this scheme, MARE improves the calculation throughput per node by 1.08×.

The maximum performance improvement MARE can achieve depends on the hardware configuration and with it the specific strong scaling behavior. Figure 17 shows the strong scaling of ROBIN on the Stampede 2 cluster at the Texas Advance Computing Center with respect to the number of threads.

Accordingly, Fig. 18 illustrates a stronger performance improvement on Stampede 2 than on Rice (shown in Fig. 16). Figure 18 shows the memory usage of ROBIN when run on Stampede 2. The gray line shows three processes with 22 threads per process which is the optimal setting without the MARE manager. The black line shows five processes with 13 threads per process managed by MARE. Since the parallel efficiency improves with MARE from 50.4% with 22 threads to 66.2% with 13 threads (see Fig. 17) the processing throughput of one Stampede 2 node improves from 3 energy points every 1583 s to 5 energy points every 2174 s. That means, the per energy point execution time improves from 528 to 435 s (1.21× speedup). Figure 18 also shows the MARE framework enables a more efficient usage of the available node memory.

4.6 CPU management to automatically balance computing resource distribution

The scheduling of computational processes entails a ramping of the number of active threads per node. This can be seen in Fig. 19 which shows the usage of a single Rice node's CPUs in the case of the 4-process simulation of Fig. 16. The delaying of MPI processes needed to limit the memory load below the node's maximum memory (see the memory usage management discussion of Sect. 4.2) causes the CPU usage to increment by steps of 5 at the beginning and decrement by 5 at the end of the simulation. Since each process is set here by environmental variables to spawn 5 threads, the CPU core usage increments and decrements in steps of 5. In addition to optimizing the memory usage, MARE can improve the CPU usage with the dynamic multithreading discussed in Sect. 3.6.

MARE's multithreading policy dynamically adjusts the number of computing threads (CPU cores) used by each process during the simulation as shown for process 1 in

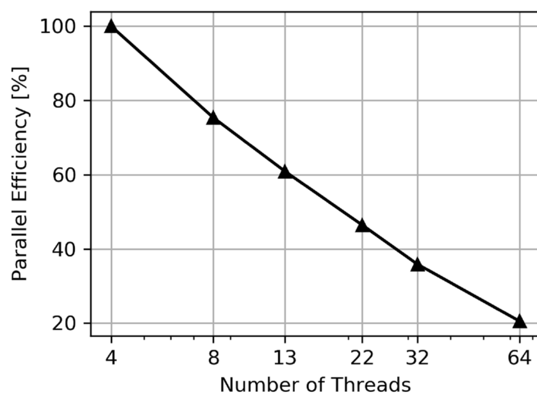


Fig. 17 The multithreading parallel efficiency of the ROBIN simulation for one energy point solved on the Stampede 2 cluster

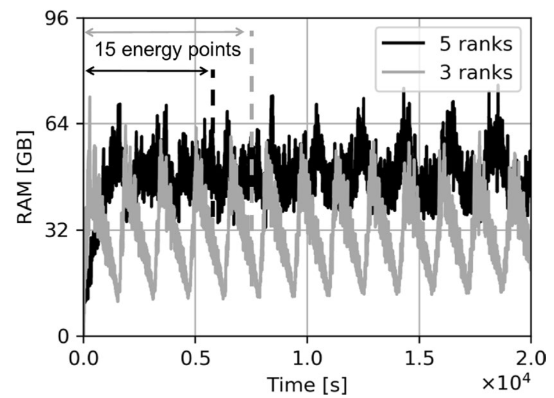


Fig. 18 Memory usage on a compute node of an example simulation with 60 energy points (multithreading enabled). The black line shows the memory usage of 5 simulation processes (12 energy points per process) with the MARE framework. The gray line shows the memory usage of 3 simulation processes (20 energy points per process) without the MARE framework. The black line with the MARE framework is faster than the gray line without the MARE framework

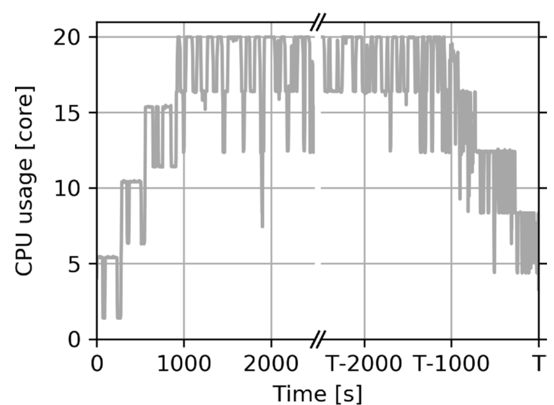


Fig. 19 CPU usage over time of the simulation of Fig. 16 with 4 simulation processes and the MARE framework without additional adjustment. The first and last approximate 1000 s show CPU ramping in increments of 5 cores

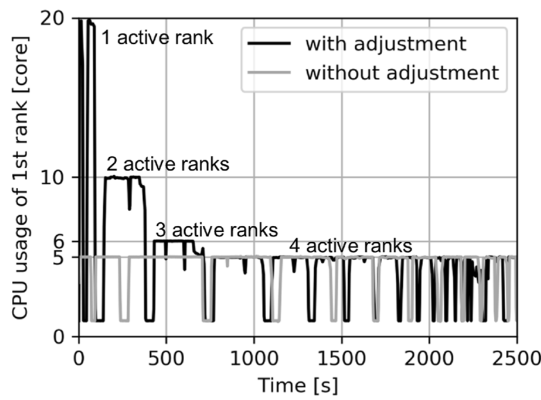


Fig. 20 CPU usage of the first process of the simulation of Fig. 19 with and without the dynamic adjustment of the number of spawned threads. The gray line shows the constant setting of 5 threads as in Fig. 19. The black line shows the dynamic spawning which starts with 20 threads when all other processes are paused. It reduces in steps whenever another process becomes active to avoid oversubscription of the node

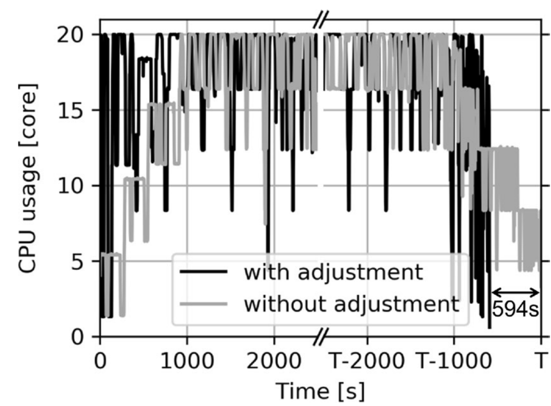
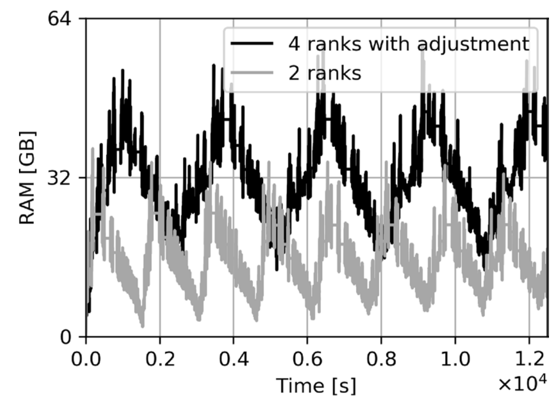


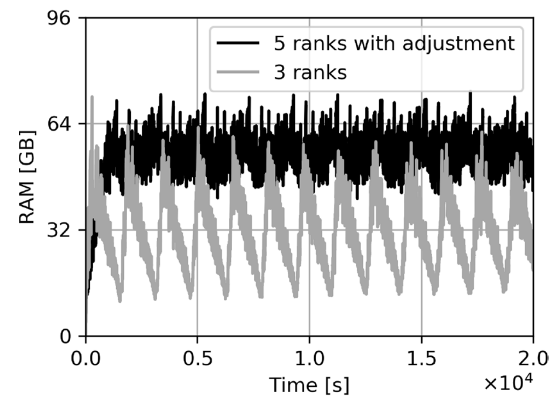
Fig. 21 Total CPU usage of all processes of the simulation in Fig. 20. The black line shows the CPU usage with MARE's dynamic multithreading. It shows full CPU usage at the begin and at the end of the simulation. For comparison, the CPU usage without MARE's dynamic thread handling of Fig. 19 is shown in gray

Fig. 20 (black line). For comparison, Fig. 20 shows the number of threads for the same process without MARE's dynamic multithreading (gray line). The dynamic policy assigns in the beginning the whole 20 CPU cores to process 1, since it is the first to enter the scheduling pipeline. When the second process enters the pipeline, the 20 cores are distributed among the two processes. Accordingly, the number of threads for process 1 drops to 10. When the third process is scheduled to start, processes 1 and 2 retain 6 threads, and process 3 receives the remaining 8 available threads. Once the forth process becomes active, all processes are assigned 5 threads. The reverse of this assignment procedure happens near the end of the simulation, whenever another process finishes its simulation tasks. Note that both dynamic (black) and static (gray) multithreading scenarios face oscillations in the CPU usage when the code iterates between serial and multithreaded sections. The overall CPU usage of 4 processes are shown in Fig. 21. The black line represents higher CPU usage at the begin and end of simulation with the dynamic CPU adjustment.

Figure 22 shows the memory usage of the same example on the Rice cluster. Similarly, it also shows the memory usage of the example on the Stampede 2 cluster. The additional CPU management enables processing four energy points every 2818 s on the Rice cluster (compared with 2937 s with memory management only). The speedup factor is 1.13× with the CPU management. The processing time of 5 energy points on one Stampede 2 node improves from 2174 to 2037 s with the additional CPU management as well (1.30× speedup).



(a)



(b)

Fig. 22 **a** The memory usage of the same example in Fig. 16 on the Rice cluster. **b** The memory usage of the same example in Fig. 18 on the Stampede 2 KNL cluster. Dynamic multithreading is enabled in this figure in addition to memory usage pipelining shown in Figs. 16 and 18. Additional speedup (see the detailed comparison in Table 3) and smoother memory usage can be observed

Table 3 Average time-to-solution (s) per energy point for the ROBIN example calculation on the Rice cluster and the Stampede 2 cluster with different levels of optimization

System	Without MARE	With MARE (memory)	With MARE (memory + CPU)
Rice	794 s	734 s (1.08×)	705 s (1.13×)
Stampede 2	528 s	435 s (1.21×)	407 s (1.30×)

Speedup factor relative to the "Without MARE" baseline is also shown

4.7 Overall performance improvement

Table 3 summarizes the calculation throughput on the Rice cluster and the Stampede 2 cluster with and without the MARE framework. When memory management schedules the processes (labeled "With MARE (memory)"), more processes can run simultaneously and fewer threads per process are needed to utilize all CPUs. This improves the parallel efficiency and thus the throughput. The dynamic adjustment of multithreading reduces the CPU underutilization during pipeline filling and draining (labeled "With MARE (memory + CPU)"), which improves the throughput further.

5 Conclusion

The majority of supercomputers are dedicated to scientific simulations. Scientific simulation software typically allows users varying applications with hard-to-predict computational load. In consequence, the supercomputer hardware tends to be incompletely utilized during the scientific simulation runs. This paper presents the MARE framework which solves this problem with a few control points to the scientific simulation codes. With these observational points, MARE schedules the computation of each parallel process to optimize memory and CPU load during the simulation's runtime. In this way, MARE enables even inexperienced users to optimally utilize the available infrastructure without significantly altering the scientific software. MARE accepts user-defined policies for, e.g., optimizing memory usage, CPU usage, or both. The applications of MARE on Quantum Espresso and ROBIN demonstrate resource efficiency improvement with only superficial knowledge of the simulation software.

Appendix: Necessary code changes to interface with MARE

The MARE framework requires superficial knowledge of the scientific simulation software rather than an actual understanding of the algorithm. The following commands are provided to interface with the MARE framework.

- mare_init: initialize the MARE client and register the SIGUSR1 and SIGUSR2 signal handlers.
- mare_wait: check with the MARE manager if continuing execution is allowed. Suspend the process if not.
- mare_signal: inform the MARE manager to wake up a suspended process if there is any.
- mare_barrier: suspend the current process and inform the MARE manager.

In case of ROBIN simulations, all that is needed is the location of the actual code, i.e., the respective module source files. The initialization of the MARE framework is added to the initialization function of the ROBIN module:

```
MARE_once := false;
if MARE_once ≡ false
  if simulation_condition1
    call mare_init(null, mkl_set_num_threads, program_name);
    call mare_wait(control_point_1);
  fi
  if simulation_condition2
    call mare_signal(control_point_1);
    MARE_once := true;
  fi
fi
if simulation_condition3
  call mare_signal(control_point_2);
fi
```

Here, ⟨*simulation_condition*1⟩ is only true during the simulation initialization phase. ⟨*simulation_condition*2⟩ is only true when a delay of execution is desired to, e.g., shift the peak memory usage point. ⟨*simulation_condition*3⟩ is only true when the calculation of one energy point (out of typically several hundred) finishes. These simulation conditions can be identified by examining the simulation logs in a benchmark simulation scenario. The MARE framework then uses such signals from all simulation processes to adjust pipelining and the distribution of CPU cores.

The calculations in the Quantum Espresso example all start at the same time, while CPU resources are adjusted for each parallel process toward the end of each iteration to ensure full utilization. Explicitly using the wait and signal commands of

semaphores are, therefore, not needed. A simplified barrier command is sufficient to instrument the end of calculation of a process in one iteration.

```
call mare_init();
call mare_barrier(control_point_1);
while simulation_condition1 do
      call mare_barrier(control_point_2);
      call mare_barrier(control_point_3);
od
call mare_barrier(control_point_4);
```

⟨*simulation_condition*1⟩ is true when the self-consistent calculation does not meet the convergence criteria. The ⟨*control_point*_1⟩ represents the end of initialization of each parallel process. The ⟨*control_point*_2⟩ and ⟨*control_point*_3⟩ represent the end of two dependent steps in one iteration to calculate a momentum point. The ⟨*control_point*_4⟩ represents the end of updating ionic forces after the convergence of self-consistent calculations. The numerical FFT library also picks up the updated threads assignment at the beginning of calculations.

Note that a management policy file is required as part of the MARE framework. Besides parsing command line arguments, the policy file implements the logic of a standard semaphore and a method to distribute CPU cores among active processes. Both the semaphore and the distribution method are agnostic to the scientific simulation. This management policy file is provided as part of the MARE framework. The Purdue Research Foundation can be contacted via OTCIP@prf.org to obtain the MARE framework source code.

Acknowledgements This work was supported in part by Intel Parallel Computing Center at Purdue. J.C. and T.K. acknowledge support from Silvaco. This research was supported in part through computational resources provided by Information Technology at Purdue University, West Lafayette, Indiana. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation under Grant No. ACI-1548562. This work used the Extreme Science and Engineering Discovery Environment (XSEDE) Stampede 2 at the Texas Advanced Computing Center (TACC), The University of Texas at Austin through allocation TG-MCA08X012. The use of nanoHUB.org computational resources operated by the Network for Computational Nanotechnology funded by the US National Science Foundation under Grant Nos. EEC-0228390, EEC-1227110, EEC-0228390, EEC-0634750, OCI-0438246, OCI-0832623, and OCI-0721680 is gratefully acknowledged. The authors would like to thank valuable discussions with Kuang-Chung Wang and Yuanchen Chu.

Funding This work was supported in part by Intel Parallel Computing Center at Purdue. J.C. and T.K. acknowledge support from Silvaco. This research was supported in part through computational resources provided by Information Technology at Purdue University, West Lafayette, Indiana. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation under Grant No. ACI-1548562. This work used the Extreme Science and Engineering Discovery Environment (XSEDE) Stampede 2 at the Texas Advanced Computing Center (TACC), The University of Texas at Austin through allocation TG-MCA08X012. The use of nanoHUB.org computational resources operated by the Network for Computational Nanotechnology funded by the US National Science Foundation under Grant Nos. EEC-0228390, EEC-1227110, EEC-0228390, EEC-0634750, OCI-0438246, OCI-0832623, and OCI-0721680 is gratefully acknowledged.

Data availability The data are available upon reasonable request.

Declarations

Conflict of interest The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

1. Performance development. https://www.top500.org/statistics/perfdevel/. Accessed 28 Feb 2022
2. Robert Y, Shende S, Malony AD, Morris A, Spear W, Biersdorff S, Smith B, Wang D, Ricciuto D, Post W, Berry MW, Irigoin F, Yelick K, Graham SL, Hilfinger P, Bonachea D, Su J, Kamil A, Datta K, Colella P, Wen T, Dongarra J, Luszczek P, Bhatele A, Freudenberger SM, Diekert V, Muscholl A, Herlihy M, Moss JEB (2011) TOP500. Encyclopedia of parallel computing. Springer, Boston, pp 2055–2057. https://doi.org/10.1007/978-0-387-09766-4_157
3. Kaufmann WJ, Smarr LL (1992) Supercomputing and the transformation of science. W. H. Freeman & Co., New York
4. Council NR, Sciences DEP, Board CST, Supercomputing CF, Patterson CA, Snir M, Graham SL (2004) Getting up to speed. National Academies Press, Washington. https://doi.org/10.17226/11148
5. Watanabe, T.: The NEC SX-3 supercomputer system. In: COMPCON Spring '91 Digest of Papers. IEEE Comput. Soc. Press, pp 303–308. https://doi.org/10.1109/CMPCON.1991.128822p
6. Gostin, G., Collard, J.-F., Collins, K.: The architecture of the HP Superdome shared-memory multiprocessor. In: Proceedings of the 19th annual international conference on supercomputing—ICS '05. ACM Press, New York, p 239 (2005). https://doi.org/10.1145/1088149.1088181
7. Sterling T, Becker DJ, Savarese D, Dorband JE, Ranawake UA, Packer CV (1995) Beowulf: a parallel workstation for scientific computation. In: Proceedings of the 24th international conference on parallel processing. CRC Press, pp 11–14
8. Kahle JA, Moreno J, Dreps D (2019) 2.1 Summit and Sierra: Designing AI/HPC Supercomputers. In: 2019 IEEE international solid- state circuits conference—(ISSCC), vol 2019-Febru. IEEE, San Francisco, CA, USA, pp 42–43. https://doi.org/10.1109/ISSCC.2019.8662426
9. Tanabe N, Ichihashi Y, Nakayama H, Masuda N, Ito T (2009) Speed-up of hologram generation using ClearSpeed accelerator board. Comput Phys Commun 180(10):1870–1873. https://doi.org/10.1016/j.cpc.2009.06.001
10. Sodani A (2015) Knights landing (KNL): 2nd Generation Intel® Xeon Phi processor. In: 2015 IEEE hot chips 27 symposium (HCS). IEEE, Cupertino, CA, USA, pp 1–24. https://doi.org/10.1109/HOTCHIPS.2015.7477467

11. Vetter JS, Mittal S (2015) Opportunities for nonvolatile memory systems in extreme-scale high-performance computing. Comput Sci Eng 17(2):73–82. https://doi.org/10.1109/MCSE.2015.4
12. Nitzberg B, Schopf JM, Jones JP (2004) PBS pro: grid computing and scheduling attributes, pp 183–190. https://doi.org/10.1007/978-1-4615-0509-9_13
13. Yoo AB, Jette MA, Grondona M (2003) SLURM: simple Linux utility for resource management, pp 44–60. https://doi.org/10.1007/10968987_3
14. Zivanovic D, Pavlovic M, Radulovic M, Shin H, Son J, Mckee SA, Carpenter PM, Radojković P, Ayguadé E (2017) Main memory in HPC. ACM Trans Arch Code Optim 14(1):1–26. https://doi.org/10.1145/3023362
15. Silberschatz A, Galvin PB, Gagne G (2003) Operating system concepts. Wiley, Hoboken
16. Bovet DP, Cesati M (2006) Understanding the Linux Kernel. O'Reilly, Beijing
17. Out-of-Memory (OOM) or Excessive Memory Usage. https://www.osc.edu/documentation/knowledge_base/out_of_memory_oom_or_excessive_memory_usage. Accessed 02 Mar 2022
18. STAMPEDE2. https://www.tacc.utexas.edu/systems/stampede2. Accessed 02 Mar 2022
19. Overview of Halstead. https://www.rcac.purdue.edu/compute/halstead/. Accessed 02 Mar 2022
20. Ritchie DM, Thompson K (1974) The UNIX time-sharing system. Commun ACM 17(7):365–375. https://doi.org/10.1145/361011.361061
21. Ritchie DM (1980) The evolution of the unix time-sharing system, pp 25–35. https://doi.org/10.1007/3-540-09745-7_2
22. Pellegrini A, Quaglia F (2017) A fine-grain time-sharing time warp system. ACM Trans Model Comput Simul 27(2):1–25. https://doi.org/10.1145/3013528
23. Wong CS, Tan IKT, Kumari RD, Lam JW, Fun W (2008) Fairness and interactive performance of O(1) and CFS Linux kernel schedulers. In: 2008 international symposium on information technology. IEEE, Kuala Lumpur, Malaysia, pp 1–8. https://doi.org/10.1109/ITSIM.2008.4631872
24. Kim C, Huh J (2016) Fairness-oriented OS scheduling support for multicore systems. In: Proceedings of the 2016 international conference on supercomputing—ICS '16. ACM Press, New York, pp 1–12. https://doi.org/10.1145/2925426.2926262
25. Love R (2005) Linux kernel development. Developer's library. Pearson Education, UK
26. Juve G, Chervenak A, Deelman E, Bharathi S, Mehta G, Vahi K (2013) Characterizing and profiling scientific workflows. Futur Gener Comput Syst 29(3):682–692. https://doi.org/10.1016/j.future.2012.08.015
27. Balaprakash P, Buntinas D, Chan A, Guha A, Gupta R, Narayanan SHK, Chien AA, Hovland P, Norris B (2013) Exascale workload characterization and architecture implications. In: 2013 IEEE international symposium on performance analysis of systems and software (ISPASS). IEEE, pp 120–121. https://doi.org/10.1109/ISPASS.2013.6557153
28. Hendrickson B, Devine K (2000) Dynamic load balancing in computational mechanics. Comput Methods Appl Mech Eng 184(2–4):485–500. https://doi.org/10.1016/S0045-7825(99)00241-8
29. Frachtenberg E, Feitelson DG, Petrini F, Fernandez J (2003) Flexible coscheduling: mitigating load imbalance and improving utilization of heterogeneous resources. In: Proceedings international parallel and distributed processing symposium. IEEE Comput. Soc, p 10. https://doi.org/10.1109/IPDPS.2003.1213191
30. Chen W, Ferreira da Silva R, Deelman E, Sakellariou R (2015) Using imbalance metrics to optimize task clustering in scientific workflow executions. Future Gener Comput Syst 46:69–84. https://doi.org/10.1016/j.future.2014.09.014
31. Tallent NR, Adhianto L, Mellor-Crummey JM (2010) Scalable identification of load imbalance in parallel executions using call path profiles. In: 2010 ACM/IEEE international conference for high performance computing, networking, storage and analysis. IEEE, New Orleans, pp 1–11. https://doi.org/10.1109/SC.2010.47
32. Wilson G (2014) Software carpentry: lessons learned. F1000Research 3:62. https://doi.org/10.12688/f1000research.3-62.v2
33. Johanson A, Hasselbring W (2018) Software engineering for computational science: past, present and future. Comput Sci Eng 20(2):90–109. https://doi.org/10.1109/MCSE.2018.021651343
34. Schmidberger M, Brugge B (2012) Need of software engineering methods for high performance computing applications. In: 2012 11th international symposium on parallel and distributed computing. IEEE, Munich, pp 40–46. https://doi.org/10.1109/ISPDC.2012.14
35. Zhao Y, Hategan M, Clifford B, Foster I, von Laszewski G, Nefedova V, Raicu I, Stef-Praun T, Wilde M (2007) Swift: fast, reliable, loosely coupled parallel computation. In: 2007 IEEE congress on services (Services 2007). IEEE, Salt Lake City, pp 199–206. https://doi.org/10.1109/SERVICES.2007.63
36. Deelman E, Blythe J, Gil Y, Kesselman C, Koranda S, Lazzarini A, Mehta G, Papa MA, Vahi K (2004) Pegasus and the pulsar search: from metadata to execution on the grid, pp 821–830. https://doi.org/10.1007/978-3-540-24669-5_107
37. Litzkow MJ, Livny M, Mutka MW (1988) Condor-a hunter of idle workstations. In: Proceedings of the 8th international conference on distributed. IEEE Computer Society Press, pp 104–111. https://doi.org/10.1109/DCS.1988.12507
38. Chimakurthi SK, Reuss S, Tooley M, Scampoli S (2018) ANSYS workbench system coupling: a state-of-the-art computational framework for analyzing multiphysics problems. Eng Comput 34(2):385–411. https://doi.org/10.1007/s00366-017-0548-4
39. Pawlowski B, Juszczak C, Staubach P, Smith C, Lebel D, Hitz D (1994) NFS Version 3: design and implementation. In: USENIX summer 1994 technical conference (USENIX Summer 1994 Technical Conference). USENIX Association, Boston
40. Braam P (2019). The Lustre storage architecture. https://doi.org/10.48550/arXiv.1903.01955
41. Schmuck F, Haskin R (2002) GPFS: a shared-disk file system for large computing clusters. In: Proceedings of the 1st USENIX conference on file and storage technologies. FAST '02. USENIX Association, USA, p 19
42. Inotify(7)-Linux manual page. https://man7.org/linux/man-pages/man7/inotify.7.html. Accessed 03 May 2022
43. Fisher C (2022) Linux filesystem events with inotify. https://www.linuxjournal.com/content/linux-filesystem-events-inotify. Accessed 05 Mar 2022
44. Meyers S (2012) Effective C++ digital collection: 140 ways to improve your programming. Pearson Education, Upper Saddle River
45. Streng M (1996) Load balancing for computational fluid dynamics calculations, pp 145–172. https://doi.org/10.1007/978-94-009-0271-8_4
46. Pearce O, Gamblin T, de Supinski BR, Arsenlis T, Amato NM (2014) Load balancing n-body simulations with highly non-uniform density. In: Proceedings of the 28th ACM international conference on supercomputing. ACM, New York, pp 113–122. https://doi.org/10.1145/2597652.2597659
47. Straka M (2023) Application scaling case study. https://bluewaters.ncsa.illinois.edu/liferay-content/document-library/DocumentationDocuments/Workshops/New UserWorkshopMay2013/Straka-WRF_Scaling.pdf. Accessed 17 Aug 2023
48. Ouermi TAJ, Kirby RM, Berzins M (2018) Performance optimization strategies for WRF physics schemes used in weather modeling. Int J Network Comput 8(2):301–327. https://doi.org/10.15803/ijnc.8.2_301

49. ...Giannozzi P, Andreussi O, Brumme T, Bunau O, Buongiorno Nardelli M, Calandra M, Car R, Cavazzoni C, Ceresoli D, Cococcioni M, Colonna N, Carnimeo I, Dal Corso A, de Gironcoli S, Delugas P, DiStasio RA, Ferretti A, Floris A, Fratesi G, Fugallo G, Gebauer R, Gerstmann U, Giustino F, Gorni T, Jia J, Kawamura M, Ko H-Y, Kokalj A, Küçükbenli E, Lazzeri M, Marsili M, Marzari N, Mauri F, Nguyen NL, Nguyen H-V, Otero-de-la-Roza A, Paulatto L, Poncé S, Rocca D, Sabatini R, Santra B, Schlipf M, Seitsonen AP, Smogunov A, Timrov I, Thonhauser T, Umari P, Vast N, Wu X, Baroni S (2017) Advanced capabilities for materials modelling with Quantum ESPRESSO. J Phys Condens Matter 29(46):465901. https://doi.org/10.1088/1361-648X/aa8f79
50. ...Giannozzi P, Baroni S, Bonini N, Calandra M, Car R, Cavazzoni C, Ceresoli D, Chiarotti GL, Cococcioni M, Dabo I, Dal Corso A, de Gironcoli S, Fabris S, Fratesi G, Gebauer R, Gerstmann U, Gougoussis C, Kokalj A, Lazzeri M, Martin-Samos L, Marzari N, Mauri F, Mazzarello R, Paolini S, Pasquarello A, Paulatto L, Sbraccia C, Scandolo S, Sclauzero G, Seitsonen AP, Smogunov A, Umari P, Wentzcovitch RM (2009) QUANTUM ESPRESSO: a modular and open-source software project for quantum simulations of materials. J Phys Condens Matter 21(39):395502. https://doi.org/10.1088/0953-8984/21/39/395502
51. Giannozzi P, Baseggio O, Bonfà P, Brunato D, Car R, Carnimeo I, Cavazzoni C, de Gironcoli S, Delugas P, Ferrari Ruffino F, Ferretti A, Marzari N, Timrov I, Urru A, Baroni S (2020) Q <scp>uantum</scp> ESPRESSO toward the exascale. J Chem Phys 152:15. https://doi.org/10.1063/5.0005082
52. McCartney G, Hacker T, Yang B (2014) Empowering faculty: a campus cyberinfrastructure strategy for research communities. Educause Rev 20:20
53. Affinito F (2016) QE, main strategies of parallelization and levels of parallelisms. https://hpc-forge.cineca.it/files/CoursesDev/public/2016/Bologna/Material_Science_codes_on_innovative_HPC_architectures/CorsoMAX-PRACE-QE-Parallelization.pdf. Accessed 05 Mar 2022
54. Iancu C, Hofmeyr S, Blagojevic F, Zheng Y (2010) Oversubscription on multicore processors. In: 2010 IEEE international symposium on parallel and distributed processing (IPDPS). IEEE, Atlanta, pp 1–11. https://doi.org/10.1109/IPDPS.2010.5470434
55. QuantumNerd: Quantum-Espresso-Tutorial-2019. https://github.com/quantumNerd/Quantum-Espresso-Tutorial-2019-Projects/blob/master/16_Si_vacancy_diffusion/neb_cal_1_noCI/si.supercell_3_neb.in. Accessed 17 Aug 2023
56. Henkelman G, Jónsson H (2000) Improved tangent estimate in the nudged elastic band method for finding minimum energy paths and saddle points. J Chem Phys 113(22):9978–9985. https://doi.org/10.1063/1.1323224
57. Henkelman G, Uberuaga BP, Jónsson H (2000) A climbing image nudged elastic band method for finding saddle points and minimum energy paths. J Chem Phys 113(22):9901–9904. https://doi.org/10.1063/1.1329672
58. Input File Description Program: neb.x / NEB / Quantum ESPRESSO (version: 7.2). https://www.quantum-espresso.org/Doc/INPUT_NEB.html#idm88. Accessed 17 Aug 2023
59. Overview of Gilbreth. https://www.rcac.purdue.edu/knowledge/gilbreth/overview. Accessed 17 Aug 2023
60. Charles J, Kais S, Kubis T (2020) Introducing open boundary conditions in modeling nonperiodic materials and interfaces: the impact of the periodicity assumption. ACS Mater Lett 2(3):247–253. https://doi.org/10.1021/acsmaterialslett.9b00523
61. Overview of Rice. https://www.rcac.purdue.edu/compute/rice/. Accessed 05 Mar 2022
62. Towns J, Cockerill T, Dahan M, Foster I, Gaither K, Grimshaw A, Hazlewood V, Lathrop S, Lifka D, Peterson GD, Roskies R, Scott JR, Wilkins-Diehr N (2014) XSEDE: accelerating scientific discovery. Comput Sci Eng 16(5):62–74. https://doi.org/10.1109/MCSE.2014.80
63. Andrawis R, Bermeo JD, Charles J, Fang J, Fonseca J, He Y, Klimeck G, Jiang Z, Kubis T, Mejia D, Lemus D, Povolotskyi M, Rubiano SAP, Sarangapani P, Zeng L (2015) NEMO5: achieving high-end internode communication for performance projection beyond Moore's law. arXiv:1510.04686
64. Steiger S, Povolotskyi M, Park H-H, Kubis T, Klimeck G (2011) NEMO5: a parallel multiscale nanoelectronics modeling tool. IEEE Trans Nanotechnol 10(6):1464–1474. https://doi.org/10.1109/TNANO.2011.2166164
65. Quinn MJ (2003) Parallel programming in C with MPI and OpenMP. McGraw-Hill Education Group, Dubuque
66. Lu S, Park S, Seo E, Zhou Y (2008) Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: Proceedings of the 13th international conference on architectural support for programming languages and operating systems. ASPLOS XIII, pp 329–339. ACM, New York, NY, USA (2008). https://doi.org/10.1145/1346281.1346323

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.