

# Pipelining Computation and Optimization Strategies for Scaling GROMACS on the Sunway Many-Core Processor

Yang Yu<sup>1(✉)</sup>, Hong An<sup>1(✉)</sup>, Junshi Chen<sup>1</sup>, Weihao Liang<sup>1</sup>, Qingqing Xu<sup>1</sup>,  
and Yong Chen<sup>2</sup>

<sup>1</sup> University of Science and Technology of China, Hefei, Anhui, China  
{yy130611, cjuns, lwh, tsqua}@mail.ustc.edu.cn,  
han@ustc.edu.cn

<sup>2</sup> Department of Computer Science, Texas Tech University, Lubbock, TX, USA  
yong.chen@ttu.edu

**Abstract.** The increasing gap between plentiful computing elements and limited memory bandwidth makes it increasingly difficult and sometimes even infeasible for HPC community to port more applications onto many-core processor architectures. The Sunway many-core processor SW26010 used to build the Sunway TaihuLight System contains a total of 260 heterogeneous cores. All these cores can be divided into 4 core groups (CGs). Each CG includes a Management Processing Element (MPE) core and 64 Computing Processing Elements (CPEs) cores. In this paper, we refactor an important molecular dynamics (MD) application GROMACS on the Sunway TaihuLight system. By rewriting the compute-intensive kernel of GROMACS, we exploit a suitable parallelism for CPE cluster and implement pipelining computation between MPE and CPE cluster. Optimization strategies including the efficient use of scratchpad, the software-emulated cache and a hybrid parallel algorithm are adopted to solve the challenging memory bandwidth limitation. When comparing the refactored version using MPE and 64 CPEs with the original ported version using only MPE, we achieve a 16x speedup for the compute-intensive kernel. For simulating a molecule with 3 million atoms, we currently have managed to scale to 798,720 cores. Moreover, we analyze the adaptability of our mapping and optimization strategies for solving the memory bandwidth limitation when refactoring a real-world application on the Sunway heterogeneous many-core processor system.

**Keywords:** Sunway TaihuLight system · GROMACS · Parallel model · Performance optimization · Bandwidth competition · Adaptability

## 1 Introduction

The Sunway TaihuLight System, developed by the National Research Center of Parallel Computer Engineering and Technology (NRCP) of China, with a Linpack score of 93 petaflops, captured the number-one spot on the latest Top 500 list of supercomputers released at the conference SC16. As the only 100-PFLOPS system in the world, it

contains about ten million heterogeneous cores in total. 40,960 computing nodes, each of which has an extremely high peak performance of 3 TFLOPS. However, given the specific heterogeneous many-core architecture, it is challenging to utilize such plentiful computing resource effectively to achieve high performance.

Molecular Dynamics (MD) simulation [1] plays an important role in theoretical studies of biomolecular systems. The fundamental algorithm includes computing the interactions between particles and updating the coordinates and velocities via Newton's second law [2] in finite time. GROMACS is a widely-used software package for MD simulation. It is computationally intensive but highly efficient at calculating the nonbonded interactions. In recent years, several research efforts have focused on refactoring GROMACS for heterogeneous system architectures and analyzing its scalability. Typical examples include refactoring GROMACS for the Cell processor [3] and to GPU accelerators [4]. Recently, a study focusing on the challenges for exascale simulation [5] was performed by the official GROMACS developers.

The main contributions of this paper are as follows.

- We exploit a suitable parallelism to refactor GROMACS onto the Sunway TaihuLight System. By adopting a task-level pipeline, we solve the load imbalance and data dependency problems that were exposed during parallelization of the compute-intensive kernel without introducing additional execution time.
- We introduce effective optimization strategies to reuse data and reduce memory-access delays, thus solving the challenging bandwidth limitation problem on the Sunway TaihuLight System. A detailed analysis is presented concerning the implementation and benefits of each optimization strategy.
- We compare the refactored GROMACS code using both MPE and CPE clusters to the serial official GROMACS code on MPE. We achieve up to a 16x speed improvement for the main compute-intensive kernel.
- We discuss the adaptability of our parallel model and optimization strategies that are not only suitable for GROMACS but also potentially apply to other HPC applications and benchmarks as they are refactored onto the Sunway heterogeneous many-core architecture.

## 2 The Sunway TaihuLight System

### 2.1 Overview

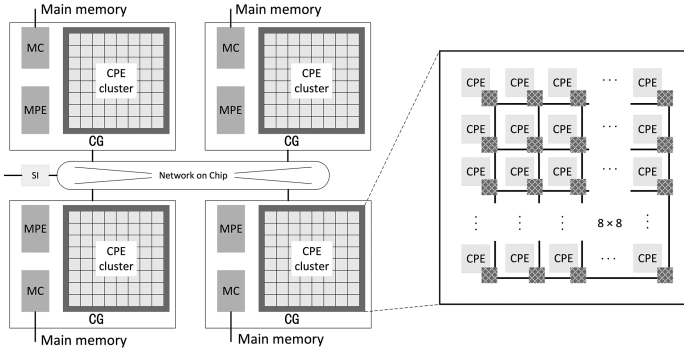
As the fastest supercomputer in the world, The Sunway TaihuLight System [8] has a theoretical peak performance of 125.4 PFLOPS. The full system adopts a multilevel tree topology composed of 40 cabinets, each of which contains 4 supernodes. Every supernode contains 256 nodes, each of which has 260 heterogeneous cores and 32 GB of main memory. In total, this system includes 10,649,600 heterogeneous cores and 1.31 PB of main memory.

The software environment includes a customized 64-bit Linux operating system and compiler components that support C/C++ and Fortran. In addition, a targeted thread

library called Athread and some other parallel programming libraries, including MPI and OpenACC, are also provided.

## 2.2 The SW26010 Processor

The processor chip of the Sunway Taihulight System is the SW26010, manufactured by the Shanghai High Performance IC Design Center, running at a frequency of 1.45 GHz. The SW26010 is composed of four core groups (CGs) which are connected via the network on chip (NoC). Each CG includes one management processing element (MPE), one computing processing element (CPE) cluster, and one memory controller (MC). The MPE—a 64-bit RISC core with 256-bit vector instructions—includes a 32 KB L1 instruction cache, a 32 KB L1 data cache and a 256 KB L2 cache. The CPE cluster contains 64 CPEs which are integrated in an  $8 \times 8$  mesh structure. Each CPE includes a 16 KB L1 instruction cache and a 64 KB Scratch Pad Memory (SPM), and also supports 256-bit vectorization. Each CG has its own memory space, which can be accessed by the MPE and the CPE cluster through the MC (Fig. 1).



**Fig. 1.** The general architecture of the SW26010 processor

Some features of the SW26010 chip that serve as the prerequisites of our optimization strategies should be explained. In one core group, the MPE can continue to run after offloading compute tasks to the CPE cluster. Unlike commercial hardware accelerators like Xeon phi and Graphic Processing Unit, the CPE cluster has no shared memory. Each CPE can access main memory and its own SPM. Continuous data blocks in the main memory can be transferred to the SPM efficiently by Direct Memory Access (DMA). Given the limited size and the low access latency of the SPM, like an L1 cache, it can be configured as a software-emulated cache for data reuse or as a read-only buffer for resident data. Moreover, data transmission is supported among the CPEs in one CPE cluster. Each CPE can transfer data at the register level to the other CPEs located in the same row or column. This capability is significant in fostering cooperative computing and CPE synchronization.

### 3 GROMACS

#### 3.1 Application Introduction

As one of the mainstream MD applications, the GROMACS package has been substantially optimized from the MD algorithm and computing platform aspects. It supports all the usual algorithms of modern molecular-dynamics implementations and employs efficient parallel models for different task-granularity levels [6, 7]. Both single- and double-precision floating point operations are supported in GROMACS. However, GROMACS has not yet provided effective support on unique many-core architectures such as the Sunway architecture. As a baseline, we adopted the official GROMACS 5.0.3 release, which supports coarse-grained parallelism using interfaces such as MPI and OpenMP and fined-grained parallelism for use with platforms such as Compute Unified Device Architecture (CUDA) and Single Instruction Multiple Data (SIMD).

The entire simulation work can be divided into two procedures: calculating interactions between atoms and updating spatial information. The interactions consist of both bonded forces between bonded atoms and nonbonded forces such as electrostatic and Van der Waals forces. The spatial information consists of coordinates and velocities. Through numerous iterations of these two procedures, GROMACS can simulate the physical motions of molecules in finite time. In addition to these two procedures, the software calculates some other molecular features such as temperature and energy. It can output the results of all the calculations to files.

#### 3.2 The Nonbonded Kernel

Calculating the nonbonded interactions is the most time-consuming part of GROMACS; consuming more than eighty-five percent of the total simulation time. Several dozen modules, distinguished by different methods for calculating electrostatic and Van der Waals forces, are integrated in the nonbonded kernel. We chose this kernel as the target for execution on the CPEs. The code framework of this hotspot can be simplified into the two nested loops shown in Algorithm 1.

---

**Algorithm 1** The nonbonded kernel
 

---

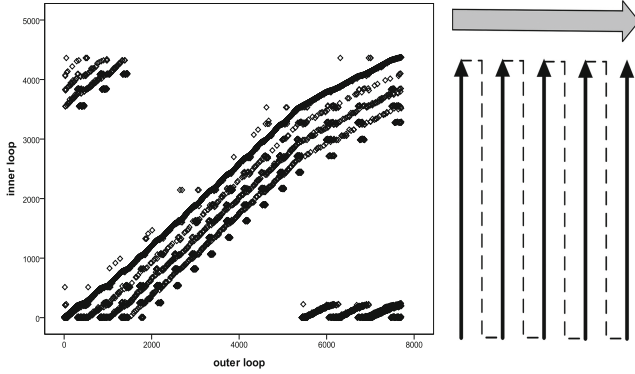
```

1:  for every atomic cluster  $i$  in the pair list
2:      data preprocessing
3:      for every atomic cluster  $j$  in  $i$ 's neighbor list
4:          calculate nonbonded forces between  $i$  and  $j$ 
5:          update force array via accumulate
6:      end for
7:  end for
    
```

---

The inner loop body that calculates the nonbonded forces will be executed numerous times in this kernel. Figure 2 shows the data-access feature requirements and the execution order of the inner loop body in a molecule with 17,089 atoms. The X-axis indicates the atomic cluster id of each outer iteration, and the Y-axis indicates the atomic cluster

id in the neighbor list required by the given outer iteration. Each point represents the interaction between two atomic clusters. Given that the data of atomic cluster are stored in id sequence, the coordinates of each point correspond to the data required by the two atomic clusters during one execution of the inner loop body. Therefore, the distribution of the scattered points in Fig. 2 reflects the memory-access behavior of the nonbonded kernel to some degree. We notice that adjacent atomic clusters are accessed in adjacent outer iterations or inner iterations and almost all the data of each atomic cluster are reused several times during the full iterative process. These two features inspire us to focus on data locality and reuse, which are the key factors affecting the design of our optimization strategies.



**Fig. 2.** The memory-access requirements and execution order of the inner loop body in the nonbonded kernel

## 4 Refactoring GROMACS for the Sunway System

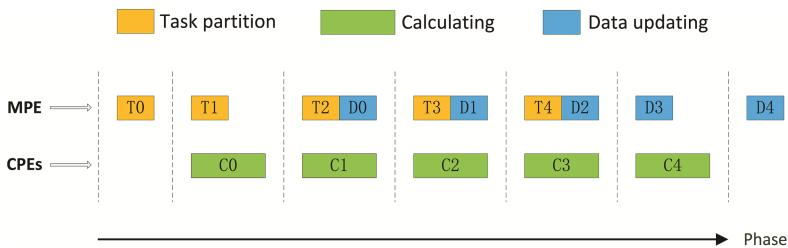
The MPE can be used as a general-purpose CPU. To begin, we refactor the official GROMACS 5.0.3 code for the Sunway system with all the CPEs inoperative. This procedure is similar to compilation and installation on a commercial server running a Linux OS. Then, we use the CPEs to accelerate the nonbonded kernel.

### 4.1 Parallelization Using CPEs

Considering the task granularity, we choose to split the outer loop for parallelization. This partition of tasks leads to two deficiencies. First, the neighbor lists of different atomic clusters potentially have common members, which causes concurrency problem when writing data. Second, the neighbor lists of the different atomic clusters vary in length, which usually leads to load imbalances among the CPEs. Two methods are proposed to solve these two problems. Before initializing the CPE threads, the computing tasks should be partitioned evenly. The results must be stored redundantly and updated serially after the 64 CPE threads complete. However, implementing these two methods would necessarily increase the execution time. Thus, we introduce the

targeted pipeline strategy, shown in Fig. 3. The three components in our pipeline are as follows.

- *Task partitioning*: By accumulating the neighbor lists in the serial outer iterations until their lengths reach a finite threshold, this component specifies the computing task for each CPE thread. The total length and iteration id of each task are recorded when accessing the data array of atomic cluster and allocating the result array.
- *Calculating*: This component is responsible for the main computing load of the nonbonded kernel according to the given task. It is also responsible for recording intermediate outputs to the newly allocated result array in the form of index and value.
- *Data updating*: This component use the intermediate results obtained from different CPE threads to update the force array serially, which avoids concurrency problem of writing data.



**Fig. 3.** A pipeline with three components: *task partitioning*, *calculating* and *data updating*

These three components are dispatched in phases. In each phase, the *task partitioning* component and the *data updating* component are executed serially on the MPE, and the *calculating* component is executed on the CPEs in parallel. The computing tasks are partitioned evenly in the *task partitioning* component, and the force array is updated serially in the *data updating* component. The execution time of the *task partitioning* component and the *data updating* component are almost entirely hidden. In summary, we handle the load imbalance and data dependency problems successfully without introducing additional execution time. Moreover, the *task partitioning* component is useful for data reuse, as will be explained in Sect. 4.3.

## 4.2 The Efficient Use of SPM

By introducing the pipeline strategy, we solve the above problems without increasing the time overhead. However, the memory bandwidth competition among 64 CPEs is extremely intense because of the frequent main-memory accesses. Consequently, the execution time of the nonbonded kernel running on the CPEs is much longer than running on the MPE. Therefore, we consider moving the repeatedly accessed data on SPM to reduce the main-memory access frequency of each CPE thread. The SPM is a 64 KB memory that is not large enough to completely store the data required by the nonbonded kernel. For example, storing three-dimensional coordinates for 10,000 atoms in single-precision floating point form requires at least 120 KB of memory. Therefore,

to use the SPM efficiently, we need to analyze the data-access characteristics in the nonbonded kernel and evaluate the advantages that could accrue from transferring data into the SPM in different ways.

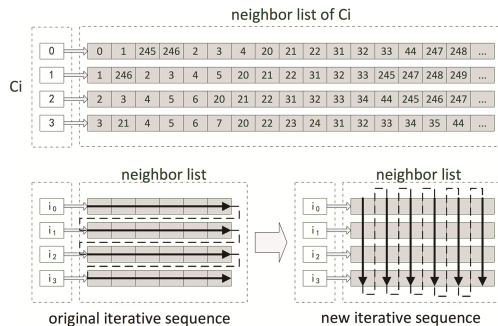
In our work, we move some frequently reused data such as the constants and small arrays into the SPM, which serve as resident data until the CPE threads finish. For large arrays such as the coordinate and force arrays, we choose to transfer the data that are useful for the current iteration to the SPM and then replace them during the next iteration to avoid address-space overflow. Therefore, we just have to transfer several small data blocks into the SPM in each iteration instead of buffering all these large arrays. All these data movements mentioned above are implemented through DMA. By doing this, we make almost all the required data available as either resident or temporary data in the local SPM. Each CPE can fetch most of its data from the local SPM instead of the main memory, which reduces the main memory access frequency. Consequently, the execution time of the nonbonded kernel is shortened obviously through the efficient use of SPM.

### 4.3 The Software-Emulated Cache

After implementing efficient SPM use, we notice that the performance of the nonbonded kernel is still dominated by the data-transmission time, indicating that the bandwidth competition is still intense. In the pipeline's *task partitioning* component described in Sect. 4.1, the outer iterations are partitioned successively and the intrinsic data features are maintained. Therefore, we can capitalize on the locality and reuse of the atomic clusters analyzed above to further reduce the frequency of main-memory accesses. Moreover, the SPM is not that limited after efficient use described in Sect. 4.2 and there is still some free memory space in the SPM. To implement more intensive data reuse, it is necessary to design a software-emulated cache.

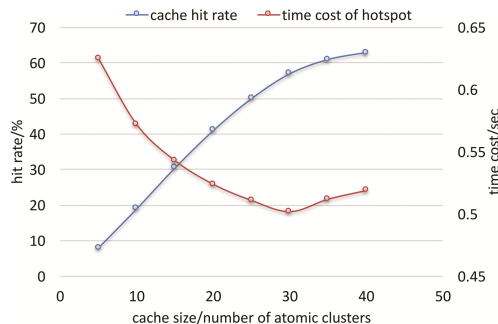
In our work, we implement a cache with a software-controlled cache size and cache line size. Continuous data block can be prefetched in a cache line and we adopt a FIFO algorithm as a cache replacement policy. Considering the memory-access behavior revealed in Fig. 2, we exchange the outer loop and inner loop for a new iterative sequence to utilize data locality sufficiently. As shown in Fig. 4, the reuse distance of some atomic cluster data is shortened and adjacent atomic cluster data can be accessed in adjacent loop iterations. This means that the prefetched data are more likely to be used before replacement, which is highly beneficial for the cache hit rate. We then set various cache sizes to discover the most suitable value for optimal performance. Moreover, we implement an output buffer to store intermediate results. Some intermediate results can be accumulated in the local SPM instead of in the main memory. By doing this, we further decrease the main-memory access frequency.

As shown in Fig. 5, the cache size is determined by the number of atomic clusters it contains. A larger cache size always leads to a higher cache hit rate and a longer execution time for maintaining the cache mechanism. When the overhead of maintaining a large cache eclipses the benefits gained by the high hit rate, the performance of the nonbonded kernel will decline. We obtain the maximum performance when the cache is set to contain thirty atomic cluster, which yields a sixty-percent cache hit rate, saving approximately sixty



**Fig. 4.** Alteration of the iterative sequence according to the locality of memory access

percent of the data transmissions in the nonbonded kernel. By implementing the software-emulated cache, we exploit the data locality and reuse that are implicit in the memory-access behavior, achieving a noticeable performance improvement.



**Fig. 5.** The influence of cache size on execution time and hit rate. The best performance is achieved with a soft-emulated cache where 30 atomic clusters can be stored

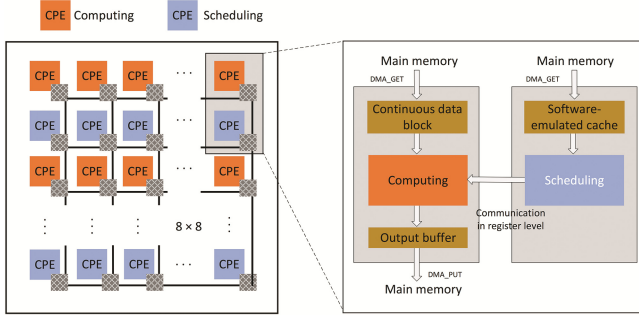
#### 4.4 A Hybrid Parallel Algorithm for Computing and Scheduling

The strategies presented in Sects. 4.2 and 4.3 speed up the nonbonded kernel by decreasing the main-memory access frequency. However, the bandwidth competition introduced by 64 CPE threads is still intense; consequently, the performance improvement achieved by using 64 CPE threads is no better than that from using only 32 CPE threads. Given this result, the other 32 CPE threads can be freed from the main computing load to act as helpers, performing data-transmission tasks and supportive computations. Our strategies are illustrated below.

We divide the 64 CPEs into two blocks: 32 CPEs for computing and 32 CPEs for scheduling. As shown in Fig. 6, one computing CPE thread collaborates with one scheduling CPE thread to undertake the tasks that is previously assigned to one CPE thread. The scheduling CPE thread is responsible for maintaining a software-emulated cache and transmitting required data to the corresponding computing CPE thread at the register



level after scheduling. The scheduling process includes a query algorithm that finds all the reusable records in the given data through array traversal—regardless of the data locality. The computing CPE thread is responsible for the main computing load of the nonbonded kernel after receiving the required data and for updating the intermediate results in the output buffer before sending them to main memory. Moreover, data transmitted earlier will always be received earlier, which guarantees the sequence of the cooperation operations between the computing thread and scheduling thread.



**Fig. 6.** A logical mapping of the hybrid parallel algorithm between a scheduling CPE and a computing CPE. The arrowheads indicate the data transmission directions

This approach mitigates the bandwidth competition by halving the number of threads attempting to access main memory. The cooperation between the computing thread and scheduling thread can be regarded as a two-level pipeline. One level includes the transmission of the required data from main memory and a scheduling algorithm; the other level includes computing the nonbonded kernel and sending the intermediate results to main memory. As a computing thread executes one iteration, the data required for the next iteration is prepared by the corresponding scheduling thread in parallel. By doing this, the required data have already been transmitted into local registers when the computing thread begins to execute a new iteration. The execution time of data-transmission, the software-emulated cache and the query algorithm in the scheduling thread are effectively hidden by the computing thread.

## 5 Results and Analysis

In this section, we provide data showing the performance of the nonbonded kernel after implementing our parallel model and optimization strategies on the Sunway TaihuLight System. We adopt a membrane protein sample with 70,960 atoms as the benchmark for the single CG performance test. This simulation spans 2,000 time steps. Moreover, we test the scalability and parallel efficiency using a large benchmark with 3 million atoms.

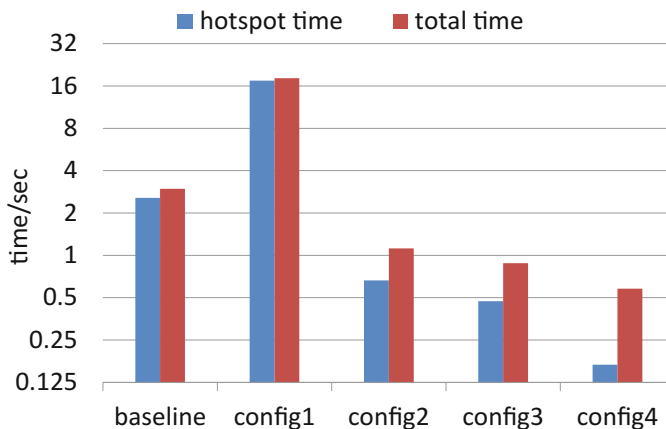
## 5.1 Single CG Performance

In Table 1, the parallel model and optimization strategies are combined into five configurations. The specific data illustrate the speedup contribution by each strategy appear in Fig. 7. In Config 1, the data required for the CPE task have to be fetched from main memory frequently which leads to extremely intense bandwidth competition. consequently, the execution time of the pipeline framework is much longer than the baseline. The efficient use of SPM in Config 2 utilizes explicit data reuse based on the code structure, which significantly reduces the main-memory access frequency. In contrast, the software-emulated cache in Config 3 capitalizes on the implicit data reuse of the nonbonded kernel, which is hidden in the data-access feature. A high cache hit rate leads to substantial reductions in main-memory accesses. After implementing the strategies described above, the bandwidth competition is still intense; the performance achieved by 64 CPEs is no better than that of 32 CPEs. Therefore, we choose to use 32 CPEs as assistant cores. These 32 CPEs analyze data reuse to lower main memory access and feed data to the 32 computing CPEs. By doing this, we manage to exploit the computing power of the 32 computing CPEs efficiently and achieve higher performance.

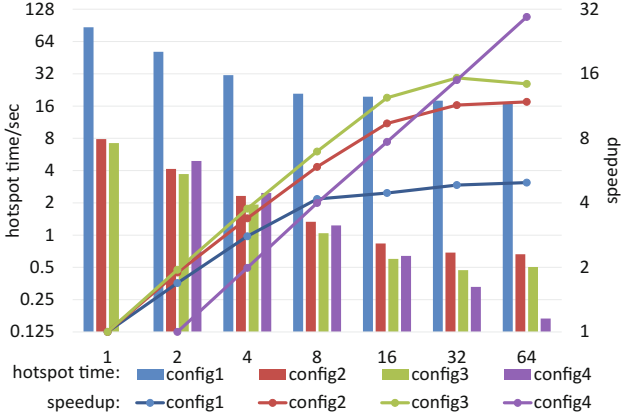
**Table 1.** Different configurations using different optimization strategies

Name	Description
Baseline	Execution of serial GROMACS code on MPE
Config 1	Parallelization of the nonbonded kernel using the pipeline strategy on an MPE + 64CPEs
Config 2	Config 1 + The efficient use of SPM
Config 3	Config 2 + The software-emulated cache
Config 4	Config 3 + Hybrid parallelism among CPEs

In Fig. 7, the hotspot refers to the Algorithm 1—the nonbonded kernel that we use 64 CPEs to accelerate. After implementing our parallel model and optimization



**Fig. 7.** Performance of GROMACS under various configurations



**Fig. 8.** The execution time and speedup of the nonbonded kernel using different numbers of CPEs in different configs;

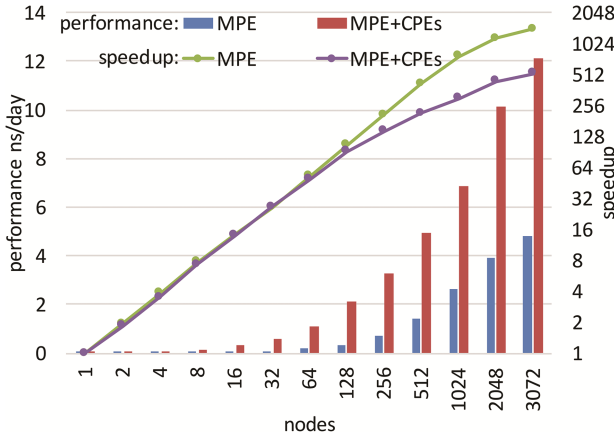
strategies, the execution time of the nonbonded kernel decreases from 2.56 s to 0.16 s, which means that our 64-CPE implementation achieves a 16X speedup compared to the baseline. Specifically, through efficient use of the SPM, we achieve a 26.62X speedup compared to Config 1. A software-emulated cache tuned for a sixty-percent hit rate speeds up the nonbonded kernel by 1.41X compared to Config 2. Introducing the cooperative computing among the CPEs brings about a 2.83X speedup compared to Config 3. After that, the overall time of our optimized code in config 4 has a 5x speedup compared to the baseline due to the restriction of Amdahl's law. Given that further optimizations of the main compute kernel could bring only limited speedup of the overall time, we plan to focus on parallelizing other computational functions in future work.

The execution time and speed improvement for each configuration obtained by increasing the number of CPEs are shown in Fig. 8. In Config 1, the bandwidth competition is extremely intense; consequently, we cannot obtain an obvious speedup in hotspot performance using more than 8 CPEs. In Config 2 and Config 3, the bandwidth competition is mitigated by decreasing the main-memory access frequency. Obvious performance improvements can be achieved in these two configs when we use no more than 32 CPEs and the performance decreases in Config 3 when we use 64 CPEs due to the saturation of parallelism. Owing to the implementation of a software-emulated cache, the performance in Config 3 is always higher than that in Config 2. In Config 4, we always halve the number of CPEs for computing and scheduling no matter how many CPEs are used. A higher performance can be achieved compared to Config 3 when we use more than 32 CPEs. Overall, we solve the challenging bandwidth-constrained problem on the Sunway architecture and achieve a near-linear growth in hotspot performance, which means that the bandwidth competition is effectively mitigated and data-transmission operations no longer dominates the nonbonded kernel.

## 5.2 Strong Scaling

We test the scalability and parallel efficiency of our code using a large benchmark with 3 million atoms. Each node of the Sunway TaihuLight System includes 4 core groups, with 260 heterogeneous cores in total. Each core group corresponds to one process, which means that a task undertaken by one core in the Intel architecture is undertaken by 65 heterogeneous cores in the Sunway architecture. In this way, scalability is significantly improved. Thus far, GROMACS on the Sunway TaihuLight System has been scaled to 798,720 heterogeneous cores.

Simulation performance and parallel efficiency are shown below. In general, GROMACS, when executed on MPEs and CPEs, has higher performance than using only MPEs at any scale. In small node scale (1, 2, 4, 8 nodes), 5x speedup can be achieved. In large node scale (3072 nodes), roughly 2.5x speedup can be achieved. Although the total execution time when using MPE + CPEs is better, the parallel efficiency using MPE + CPEs saturates faster than that using only MPE. In view of this behavior, we think that higher performance on single node always increases the portion of communication among nodes and therefore leads to the faster saturation in parallel efficiency (Fig. 9).



**Fig. 9.** Performance and parallel efficiency of GROMACS in strong scaling test. We demonstrate both the results for running on only MPE and the results for running on MPE + CPEs.

## 5.3 Analysis

In this paper, we present a parallel pipeline model to solve data dependency and load imbalance problems without increasing the execution time. This pipeline, which can be considered as cooperative computing, is implemented based on parallelism between the MPE and the CPE clusters. This approach is efficient not only in the Sunway architecture but also for other master-slave architectures such as the “Xeon CPU + Xeon Phi” combination. Given the 8 GB of shared memory and the high bandwidth of the Xeon Phi [9], our pipeline can be extended to multi-level configurations based on the

requirements, and the main computing load can be distributed over multiple levels, instead of concentrating the computing component implementation as is done in this case on the Sunway architecture.

Among our optimization strategies, the efficient use of SPM serves as a conventional optimization step for the memory bandwidth limitation. Many works on application optimization on the Sunway TaihuLight System adopt this strategy to handle bandwidth competition, which always leads to high performance improvements. However, this approach is not enough for some applications which requires high memory bandwidth, such as GROMACS. Even after this optimization step, bandwidth competition might still be intense. To address this situation, we introduce the strategies of the software-emulated cache and cooperative computing among the CPEs, which are the main innovation points in this paper. The hit rate of the software cache is based on the locality of memory access. For the memory-access feature of GROMACS, optimal performance is achieved with a sixty-percent hit rate. This strategy is suitable for applications with good data locality. Using the hybrid parallel algorithm among CPEs, the bandwidth competition is mitigated by using 32 CPEs solely to access memory. We actualize a scheduling algorithm that schedules CPEs for data reuse regardless of locality. This strategy is widely suitable for applications with good data reuse and—unlike the strategy of software-emulated cache—it has no requirement for data locality. Moreover, the computing load can be distributed by the scheduling CPEs to best use the available computing power under the condition of maintaining the original computation sequence.

## 6 Related Work

GROMACS is a classical software package in the MD field that supports multi-level parallelism. From version 4.6 onward, GROMACS has supported efficient GPU acceleration [6]. In the GPU version code, the nonbonded kernel is offloaded to a GPU device; meanwhile, the CPU calculates bonded forces and lattices. Moreover, GROMACS can be executed on MIC in a native and symmetric model.

In recent years, we have started to see projects that refactor GROMACS for heterogeneous many-core architectures such as refactoring GROMACS on the Cell architecture [3]. The Cell processor contains 1 Power Processor Element (PPE) and 8 Synergistic Processing Elements (SPE) and the memory bandwidth seemed not to be a bottleneck in their optimization procedure. Compared to their work, we explore the new challenge—the memory bandwidth limitation when using much more on-chip computing elements to accelerate GROMACS and gave multiple optimization steps to overcome this challenge. Moreover, we use a different way to solve the data dependencies. When compared to other MD packages such as AMBER [10], NAMD [11], CHARMM [12] and LAMMPS [13], GROMACS has good support for the x86 vector instruction sets. By grouping 4 or 8 atoms into one cluster, the nonbonded kernel is quite suitable for execution in the SIMD model [6]. The current scalability is not very ideal compared to NAMD. Using the same benchmark, NAMD can scale to more computing nodes. An extremely large benchmark is needed for possible utilization of the entire computing nodes on the Sunway TaihuLight System.

Before our work on the Sunway Taihulight architecture, the idea of using multiple cores in a heterogeneous way has already been applied in some other multiple/many core architectures. A technique named Speculative Precomputation aiming to improve single-thread performance on a multi-threaded architecture was explored in Jamison D. Collins et al.'s work [14]. This technique utilized otherwise idle hardware thread contexts to execute speculative threads on behalf of the non-speculative thread. These speculative threads attempted to trigger future cache miss events far enough in advance of access by the non-speculative thread that the memory miss latency was avoided entirely. Besides, Daniele Buono et al. [15] investigated message-passing supports on many-core architecture (notably Intel Xeon Phi). In their work, part of threads were used to execute support activities like point-to-point communications so as to overlap communications with calculation. Moreover, the idea of exploiting register-to-register communications between cores has also been used in Daniele Buono's another work [16]. In this work, efficient run-time mechanisms for inter-thread synchronization/communication were developed for fine-grained parallelism on network processors.

## 7 Conclusions and Future Work

GROMACS is a typical scientific application for high performance computing. It is highly optimized to use fine-grained parallelism such as SIMD and SIMT and exhibits irregular data-access features in hotspots. These characteristics form large challenges when migrating applications to the Sunway TaihuLight System architecture. In this paper, we present an up-to-date parallel pipeline model and several optimization strategies, including efficient use of the SPM, a software-emulated cache, a hybrid parallel algorithm among CPEs to remove the bottlenecks in the source code and to better utilize the hardware architecture in the parallelization procedure. All these strategies play well for solving the challenging bandwidth-constrained problem on the Sunway architecture. In one CG, we achieve a 16X speed improvement in the nonbonded kernel using 64 CPEs. The performance and scalability of this simulation are also significantly improved in strong scaling tests.

To further discuss the adaptability, we also plan to undertake the task of refactoring GROMACS for the Xeon Phi using an offload model. Furthermore, the performance of GROMACS on the Sunway TaihuLight System has the potential to be improved through further vectorization.

## References

1. Allen, M.: Introduction to molecular dynamics simulation. *Comput. Soft Matter-From Synthet. Polym. Prot.* **23**, 1–28 (2004)
2. Haile, J.M.: *Molecular Dynamics Simulation: Elementary Methods*. Wiley, New York (1992)
3. Olivier, S., Prins, J., Derby, J., Vu, K.: Porting the gromacs molecular dynamics code to the cell processor. In: *21st International Parallel and Distributed Processing Symposium*, pp. 1–8. IEEE, California (2007)

4. Elsen, E., Houston, M., Vishal, V., Darve, E., Hanrahan, P., Pande, V.S.: N-Body simulation on GPUs. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, p. 188. ACM, New York (2006)
5. Páll, S., Abraham, M.J., Kutzner, C., Hess, B., Lindahl, E.: Tackling exascale software challenges in molecular dynamics simulations with GROMACS. In: Markidis, S., Laure, E. (eds.) EASC 2014. LNCS, vol. 8759, pp. 3–27. Springer, Cham (2015). doi: [10.1007/978-3-319-15976-8\\_1](https://doi.org/10.1007/978-3-319-15976-8_1)
6. Abraham, M.J., Murtola, T., Schulz, R., Páll, S., Smith, J.C., Hess, B., Lindahl, E.: Gromacs: high performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX* **1**, 19–25 (2015)
7. Hess, B., Kutzner, C., van der Spoel, D., Lindahl, E.: GROMACS 4: algorithms for highly efficient, load-balanced, and scalable molecular simulation. *J. Chem. Theory Comput.* **4**(3), 435–447 (2008)
8. Fu, H., Liao, J., Yang, J., Wang, L., Song, Z., Huang, X., Zhao, W.: The Sunway TaihuLight supercomputer: system and applications. *Sci. China Inf. Sci.* **59**(7), 1–16 (2016)
9. Chrysos, G.: Intel® Xeon Phi™ coprocessor-the architecture. Intel Whitepaper (2014)
10. Pearlman, D., Case, D., Caldwell, J., Ross, W., Cheatham III, T., DeBolt, S., Ferguson, D., Seibel, G., Kollman, P.: AMBER, a package of computer programs for applying molecular mechanics, normal mode analysis, molecular dynamics and free energy calculations to simulate the structural and energetic properties of molecules. *Comput. Phys. Commun.* **91**, 1–41 (1995)
11. Nelson, M.T., Humphrey, W., Gursoy, A., Dalke, A., Kale, L.V., Skeel, R.D., Schulten, K.: NAMD: a parallel, object oriented molecular dynamics program. *Int. J. High Perform. Comput. Appl.* **10**(4), 251–268 (1996)
12. Brooks, B.R., Bruccoleri, R.E., Olafson, B.D., States, D.J., Swaminathan, S., Karplus, M.: CHARMM—a program for macromolecular energy, minimization, and dynamics calculations. *J. Comput. Chem.* **4**(2), 187–217 (1983)
13. Plimpton, S., Crozier, P., Thompson, A.: LAMMPS-Large-Scale Atomic/Molecular Massively Parallel Simulator, vol. 18. Sandia National Laboratories (2007)
14. Collins, J.D., Wang, H., Tullsen, D.M., Hughes, C., Lee, Y.F., Lavery, D., Shen, J.P.: Speculative precomputation: long-range prefetching of delinquent loads. In: 28th Annual International Symposium on Computer Architecture, pp. 14–25. ACM, Göteborg (2001)
15. Buono, D., De Matteis, T., Mencagli, G., Vanneschi, M.: Optimizing message-passing on multicore architectures using hardware multi-threading. In: 22nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing, pp. 262–270. IEEE, Torino (2014)
16. Buono, D., Mencagli, G.: Run-time mechanisms for fine-grained parallelism on network processors: the tilepro64 experience. In: International Conference on High Performance Computing Simulation, pp. 55–64. IEEE, Bologna (2014)

Algorithms and Architectures for Parallel Processing  
17th International Conference, ICA3PP 2017, Helsinki,  
Finland, August 21-23, 2017, Proceedings  
Ibrahim, S.; Choo, K.-K.R.; Yan, Z.; Pedrycz, W. (Eds.)  
2017, XXIII, 829 p. 330 illus., Softcover  
ISBN: 978-3-319-65481-2