The Journal of Supercomputing (2021) 77:13584–13600
https://doi.org/10.1007/s11227-021-03823-3

Multilevel parallelism optimization of stencil computations on SIMDlized NUMA architectures

Kaifang Zhang[1] · Huayou Su[1] · Yong Dou[1]

Accepted: 17 April 2021 / Published online: 28 April 2021

Abstract
Stencil computations within a single core or multicores of an SMP node have been over-investigated. However, the demands on HPC's higher performance and the rapidly increasing number of cores in modern processors pose new challenges for program developers. These cores are typically organized as several NUMA nodes, which are characterized by remote memory across nodes and local memory with uniform memory access within each node. In this paper, we conducted experiments of stencil computations on NUMA systems based on the two most typical processors, ARM and Intel Xeon E5. We leverage a hybrid programming approach by combining MPI and OpenMP to exploit the potential benefits among NUMA nodes and within a NUMA node. Optimizations of the two selected 3D stencil computations involve four-level parallelism: block decomposition for NUMA nodes and processes, thread-level parallelism within a NUMA node, and data-level parallelism within a thread based on SIMD extension. Experimental results show that we obtain a maximum speedup of 7.27× compared to the pure OpenMP implementations on the ARM platform and 11.68× on the Intel platform.

Keywords Stencil computation · Parallelism optimization · Hybrid programming · NUMA

✉ Kaifang Zhang
zhangkaifang18@nudt.edu.cn

Huayou Su
shyou@nudt.edu.cn

Yong Dou
yongdou@nudt.edu.cn

[1] College of Computer, National University of Defense Technology, Changsha, China

1 Introduction

Modern processors' great tendency is the increasing number of cores integrated on chips, which avoids the ascends of clock frequency and power. Large numbers of HPC systems on the TOP-500 list [1] are composed of such processors that provide high performance. Ideally, the performance should scale with the number of cores built in. The basis of such a composition block is the multicore chips sharing memory within a single node. Programming for such parallel architectures consisting of NUMA (nonuniform memory access) [2] nodes is hugely challenging for the programmers have to explicitly control the memory accesses to take full advantage of the memory access characteristics. On the other hand, the enormous amount of cores is an immense potential for thread-level parallelism.

Practically, the programmers are apt to ignore the hybrid memory system and utilize a naive OpenMP or MPI (Message Passing Interface) implementations for convenience. However, the former strategy tends to behave poorly due to its applicability to SMPs (Symmetric Multi-Processing). It is advisable for the MPI model since most MPI libraries take advantage of the shared memory within a node and optimize the inter-node communications.

Nonetheless, in this paper, we leverage both MPI and OpenMP to investigate the multicore scaling, which is a hybrid programming model for the NUMA architectures. It is not the first time that this approach has been utilized for parallel optimizations of applications. However, we employ a core grouping strategy to excavate the CPU affinity. We also propose an outermost dimension decomposition scheme to integrate the data-level parallelism scheme—SIMD (Single Instruction Multiple Data)—into the multicore scaling to exploit the best whole platform performance.

Our main contributions can be summarized as follows:

- Proposing a four-level parallelism scheme, including NUMA node decomposition, block decomposition, thread-level parallelism, and data-level parallelism, to exploit the potential benefits of the multicore systems.
- Implementing the above techniques using a hybrid MPI+OpenMP programming model on two SIMDlized NUMA systems with SIMD intrinsics supported by each specific architecture, one ARM NEON Extension and the other Intel AVX.
- Evaluating the proposed multilevel parallelism scheme on two NUMA platforms with two typical 3D stencil computations. Comparisons to naive implementations of OpenMP show that it obtains a better performance as well as scalability.

The rest of the paper is organized as follows: Sect. 2 is a brief introduction of OpenMP and MPI models, NUMA architecture, and our motivation for the hybrid programming method. Section 3 illustrates our multilevel parallelism scheme we propose. Experimental evaluations of our proposed scheme are conducted and demonstrated in Sect. 4. Section 5 describes some related work. Finally, we conclude the whole paper in Sect. 6 and provide with some future work hints.

2 Background and motivation

Some preliminary basis about OpenMP, MPI, and NUMA architecture and our hybrid programming approach is described in this section. We also illustrate the motivation for hybrid programming.

2.1 OpenMP programming model

OpenMP is detailedly discussed in [3], and much latest features about it can be found at https://www.openmp.org/. It is characterized by its compiler directives as well as enormous library calls. As a portable approach for parallel programming, the high-level abstraction of the parallel description provided by OpenMP reduces the difficulty and complexity of parallel programming so that programmers can devote more energy to the parallel algorithm itself, rather than its specific implementation details. For multithreaded programming based on data diversity, OpenMP is generally the right choice. Simultaneously, the use of OpenMP also provides greater flexibility, and it can quickly adapt to different parallel system configurations. Thread granularity and load balancing are challenging problems in traditional multi-threaded programming, but in OpenMP, the OpenMP library takes over part of these two aspects of work from programmers. However, as a high-level abstraction, OpenMP is not suitable for situations that require complex synchronizations and mutual exclusion between threads. Another disadvantage of OpenMP is that it cannot be used on non-shared memory systems (such as computer clusters), where MPI is utilized much more.

2.2 MPI programming model

MPI is further described in [4], and [5]. As a de facto standard for parallel programming on the distributed memory systems, it has two advantages: achievable performance and portability. Specifically, it improves communication efficiency by measures, including avoiding multiple repeated copies from memory to memory, allowing overlap of calculation and communication. The standard API and libraries contribute to the portability. All in all, MPI provides a language and platform-independent standard that can be widely used to write message-passing programs. It is practical, portable, efficient, and flexible and has not much change from the current existing implementation when ported from one machine to another. Challenges also exist. Carefully thought-out strategies are required when programming. The data portions and correlated communications have to be considered seriously. Other challenges include memory requirements and fault tolerance [6].

2.3 Hybrid programming approach

For the appearance of large numbers of multicore platforms with shared memory nodes or clusters, it is quite natural to combine those mentioned above two parallel programming models together, which is a kind of hybrid programming

approach [7]. This model can leverage the advantages of MPI among nodes with distributed memory systems and OpenMP within a node. In other words, we distinguish between the memory hierarchies and can maintain cross-node performance with MPI and reduce the number of processes needed simultaneously. The OpenMP is then in charge of the inner-node performance optimization with thread-level and other level parallelisms.

2.4 NUMA architecture

NUMA architecture is a shared memory architecture that describes the placement of main memory modules with respect to processors in a multiprocessor system [8]. Like almost every other processor architectural feature, ignorance of NUMA can result in a sub-par application memory performance. In the NUMA shared memory architecture, each processor has the local memory module that it can access directly with a distinctive performance advantage. At the same time, it can also access any memory module belonging to another processor using a shared bus [8]. However, the two various memory accesses' costs are distinctly shown in Table 1 for the ARM platform we work on. A NUMA-node always has a distance of 10 to itself, which is the lowest possible value. In contrast to UMA, which provides a centralized memory pool, it has to traverse the interconnect and connect to the remote memory controller if a memory controller accesses remote memory. Thus, accessing remote memory adds additional latency overheads to local memory access. Because of the different memory locations, a NUMA system experiences nonuniform memory access time.

There exist two critical notions in managing performance within the NUMA shared memory architecture: processor affinity and data placement [8]. It is based on the NUMA distances listed in Table 1 that we can deduce the following core grouping of the eight NUMA nodes in Table 2. Further, we conduct the multicore scaling based on the core grouping. For instance, we would prefer to deploy nodes 0, 1, 3, 5 rather than 0, 1, 2, 3, when four NUMA nodes are required for the former scheme has a better memory access behavior.

Table 1 NUMA node distances (ARM)

Node	0	1	2	3	4	5	6	7
0	10	20	40	30	20	30	50	40
1	20	10	30	20	30	20	40	30
2	40	30	10	20	50	40	20	30
3	30	20	20	10	40	30	30	20
4	20	30	50	40	10	20	40	30
5	30	20	40	30	20	10	30	20
6	50	40	20	30	40	30	10	20
7	40	30	30	20	30	20	20	10

Table 2 Core grouping according to node distances (ARM)

Node/dis.	20	30	40	50
0	1, 4	3, 5	2, 7	6
1	0, 3, 5	2, 4, 7	6	- -
2	3, 6	1, 7	0, 5	4
3	1, 2, 7	0, 5, 6	4	–
4	0, 5	1, 7	3, 6	2
5	1, 4, 7	0, 3, 6	2	–
6	2, 7	3, 5	1, 4	0
7	3, 5, 6	1, 2, 4	0	–

2.5 Motivation

We demonstrate a simple example in this section to illustrate the necessity of the hybrid programming model. Table 3 is the performance scalability for a 3D-7pt stencil computation on one of our experimental platforms. The platform has eight NUMA nodes, with each consisting of eight cores, and we only utilize OpenMP for the multicore scalability. It can be observed that within a node when the cores utilized is less than 8, approximate linear scalability is presented. However, when much more cores across multi-NUMA nodes are utilized, no further performance benefits are obtained. A performance regression phenomenon even occurs due to the memory bandwidth competitions.

3 Multilevel parallelism scheme

In this section, we demonstrate the proposed four-level parallelism scheme in detail. The overview of our suggested parallel scheme is illustrated in Fig. 1.

To be specific, the first two levels of parallelisms are the block decomposition of the original data grid according to the number of NUMA nodes and the processes

Table 3 Performance scaling using only OpenMP (ARM)

#Cores	GFlops	Speedup	Ideal speedup
1	0.52	1.00×	1.00×
2	0.95	1.83×	2.00×
3	1.40	2.69×	3.00×
4	1.87	3.60×	4.00×
5	2.36	4.54×	5.00×
6	2.81	5.40×	6.00×
7	3.28	6.31×	7.00×
8	3.53	6.79×	8.00×
16	4.35	8.37×	16.00×
32	2.57	4.94×	32.00×
64	2.68	5.15×	64.00×

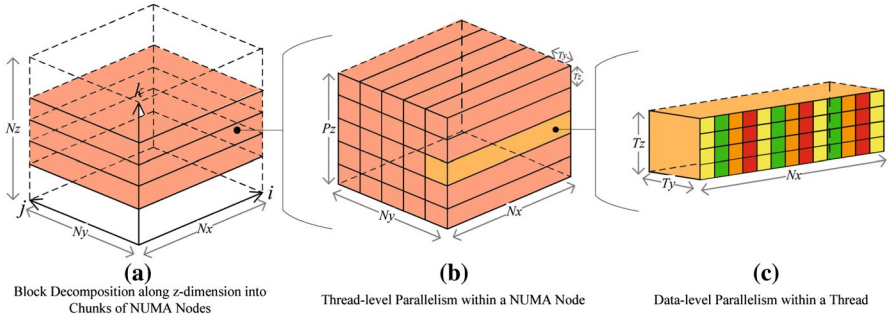

Fig. 1 Overview of the four level parallelisms we utilize

utilized. Figure 1a shows the decomposition along z-dimension without cutting the other two dimensions. When it comes to Fig. 1b, we depict the third-level parallelism—thread-level parallelism—a further decomposition among threads within a NUMA node along the outer two dimensions. Finally, Fig. 1c is the last-level parallelism we study the data-level parallelism-which is usually implemented through SIMD ISAs. We provide much more details of the techniques mentioned above in the remaining section, respectively.

3.1 Block decomposition

As is depicted in Fig. 2, the first two levels of parallelisms are the block decomposition among NUMA nodes and processes. We conduct the block composition among NUMA nodes and processes along the outermost z-dimension, for it has a longer reuse distance compared to the two inner dimensions. For instance, for an $Nz \times Ny \times Nx$ input grid, the reuse distance of the 3D-27pt stencil computation along the innermost

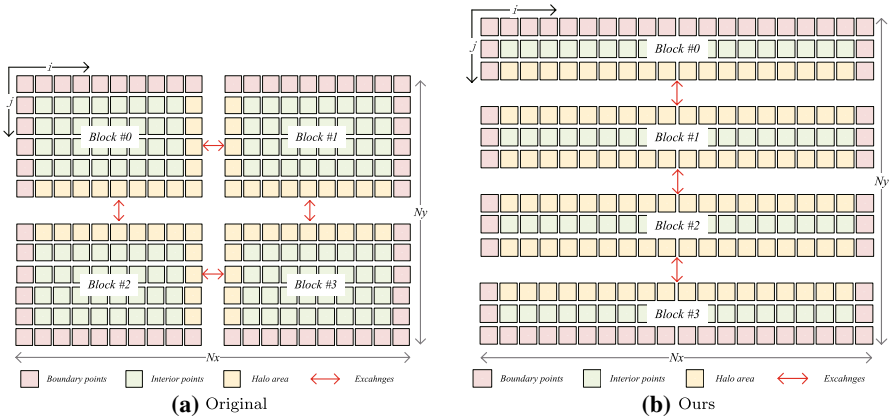

Fig. 2 The first two level parallelisms—block decomposition

x-dimension is 1. However, the reuse distances along y- and z-dimension are Nx and $Nx \times Ny$ separately, which are more extensive than that of the x-dimension. We utilize such a decomposition scheme to avoid the ruins of data locality.

The halo areas for exchanges and asynchronous communications are also colored in Fig. 2. It is an illustration for a 2D stencil computation input grid, and we cut along the outermost y-dimension correspondingly. Based on the decomposition scheme mentioned above, we have to do the halo exchanges of diverse blocks for the next time step updates represented by the red double arrows. Three kinds of points exist in the scheme: the boundary points, the interior points, and the halo area points. The two former kinds of points are without necessities for exchanges, while the third kind of points has to do the asynchronous exchanges.

What is worth noting is that our outermost dimension splitting scheme in Fig. 2b spends less time for halo exchanges compared to the naive *c*ross splits along both dimensions. Generally speaking, for an input grid of size $Nx \times Ny$, and a decomposition scheme of $Bx \times By$ of the naive implementation, a total number of exchanges of $2[(Nx/Bx - 1) \times Ny/By + (Ny/By - 1) \times Nx/Bx]$ are required. Meanwhile, for our scheme, the number of exchanges necessary is $2[NxNy/(BxBy) - 1]$ for the same number of blocks. The difference between the two numbers is:

$$\begin{aligned} \Delta &= 2\left[\left(\frac{Nx}{Bx} - 1\right)\frac{Ny}{By} + \left(\frac{Ny}{By} - 1\right)\frac{Nx}{Bx}\right] - 2\left(\frac{Nx}{Bx}\frac{Ny}{By} - 1\right) \\ &= 2\left(\frac{Nx}{Bx} - 1\right)\left(\frac{Ny}{By} - 1\right) \end{aligned} \tag{1}$$

In practice, we have $(Nx/Bx - 1) > 0$ and $(Ny/By - 1) > 0$ and as a result we have $\Delta > 0$. Consider the two schemes in Fig. 2: four blocks are decomposed from the original input data grid. Eight exchanges are required for the four decomposition boundaries for each block has to receive and send at the same time for the naive decomposition. As to our decomposition scheme, only three decomposition boundaries exist, and a total number of six exchanges are necessary for the same number of blocks.

Besides, we evaluate the communication overheads for the two schemes above, and the results are listed in Table 4. The problem size is 512^3 and time steps = 100. The left column *x-y-z* represents various block decomposition schemes (eight blocks in total), and the *total time* column the corresponding total time including the halo exchange overheads. We also list the percentage of exchange overheads. It can be observed that our outermost splitting scheme *1-1-8* has a minimum overhead of 1.70s, and it accounts for 8.60% of the total time, while the innermost decomposition scheme *8-1-1* has a maximum overhead of 10.05s and a percentage of 21.88 of its overall time. Other block decomposition scheme overheads are between them.

3.2 Thread-level parallelism

For the third-level parallelism, we refer to the thread-level parallelism within a NUMA node. The parallel scheme is outlined in Algorithm 1. For the original

Table 4 Halo exchanges overheads for diverse decomposition schemes

x-y-z	Exchange time (sec.)	Total time (sec.)	Overhead percentage (%)
2-2-2	2.72	24.85	10.95
1-1-8	1.70	19.77	8.60
1-8-1	1.74	17.50	9.94
8-1-1	10.05	45.93	21.88
1-2-4	1.88	17.67	10.64
1-4-2	1.92	18.34	10.47
2-4-1	2.88	21.98	13.10
2-1-4	3.02	24.33	12.41
4-1-2	5.34	28.44	18.78
4-2-1	5.30	31.25	16.96

stencil computation depicted in Line 8, there exist three nested loops. The original input array is referred as *old*, while the new updated one as *new*. The kernel update formulation is in Line 8, in which 26 neighbor elements and the center element itself are weighted by their separate coefficients.

To parallel the code, we add the *parallel* directive to it in Line 1. Line 2 is the workload scheduling approach supported by OpenMP. In Line 3, we assign the parallelism granularity by setting the threads utilized. Line 4 is the parallel depth. We set it as 2 by combining the outer two dimensions and leave the innermost dimension for data-level parallelism described in the next section.

Algorithm 1 Thread-level parallelism with OpenMP

Input: z_s, z_e, y_s, y_e, x_s, x_e, $threads$, $depth$, $schedule$, $Stencil()$;

```
1: Procedure OMP();
2: OMP_NUM_THREADS(threads);
3: Schedule(schedule);
4: Collapse(depth);
5: for k = z_s → z_e do
6:    for j = y_s → y_e do
7:       for i = x_s → x_e do
8:          Stencil();
9:       end for
10:   end for
11: end for
12: End Procedure;
```

3.3 Data-level parallelism

The last-level parallelism we investigate is the data-level parallelism, which is the SIMD extension supported on each platform. We omit the specific principles and regulations

for much has been discussed in [9–13], and [14]. We make use of a 1D-3pt stencil, for instance, to demonstrate the SIMD intrinsic implementations on the two platforms. Two diverse implementations of the two specific platforms are listed in Fig. 3 briefly:

For the ARM platform, the NEON extension has a 128-bit vector length, and the V corresponding to it is 2 for double precision. The Intel platform supports a vector length of 256-bit, and its V equals 4. It can be observed that they have a similar implementation pattern. Note that the data types of *old* and *new* are corresponding SIMD data types on separate platforms rather than double precision.

3.4 Put it all together

Taking all the aforementioned techniques into consideration, the pseudocode for the kernel combining both MPI and OpenMP would be in Algorithm 2:

Algorithm 2 Multi-level parallelism algorithm

Input: T, z_s, z_e, y_s, y_e, x_s, x_e, $numa$, $processes$, $threads$, $depth$, $schedule$, V, $Stencil()$;

```
Procedure MULTI_PARALLEL();
· · · ;
Decomposition(numa);
Decomposition(processes);
for t = 1 → T do
    MPI(processes);
    OMP(threads, schedule, depth);
    for k = z_s → z_e do
        for j = y_s → y_e do
            for i = x_s → x_e, V do
                Stencil();
            end for
        end for
    end for
end for
· · · ;
End Procedure;
```

```
1. for(i = 1; i < I; i += V)
2.     new[i] = vaddq_f64(old[i-1], vaddq_f64(old[i], old[i+1]));
```

(a) SIMD Implementation using ARM NEON Extension

```
1. for(i = 1; i < I; i += V)
2.     new[i] = _mm256_add_pd(old[i-1], _mm256_add_pd(old[i], old[i+1]));
```

(b) SIMD Implementation using Intel AVX

Fig. 3 Two Intrinsic Implementations using SIMD Extension

The first two-level parallelisms are initially conducted before the time step iterations in Lines 1-2. Note that for the MPI instances of multiprocesses in Line 5, we add *parallel* to indicate that they are performing concurrently. It also contains the halo exchanges and asynchronous communications mentioned earlier, which we omit in the pseudocode. Line 5 is the OpenMP parallel region mentioned above. In Line 9, we express the kernel updates of SIMD implementations in brief.

4 Performance evaluation

4.1 Experimental test bed

Details about the core, socket, system, and programming of the experimental platforms are in Table 5. The experimental ARM processor integrates 64 ARMv8 instruction set compatible processor cores and adopts a parallel system on chip (PSoC) architecture. The NEON extension of the platform supports a 128-bit length vector length, with double precision support. The 64 cores are divided into eight NUMA nodes, each with eight cores. As to the Intel platform, it is formerly the products of Broadwell. The SIMD extension of it has a vector length of 256-bit with double precision supported. Its 20 cores are grouped into two NUMA nodes evenly. Both platforms have multilevel memory hierarchies and support hardware prefetching mechanism. Other specific architectural parameters are in Table 5. All benchmarks are evaluated using *-O3* compiler options with no other unique options deployed.

4.2 Stencil overview

One of the benchmarks we work on is shown in Fig. 4. In this paper, we take two typical 3D stencils, for instance, to evaluate the platforms' performance. Some general characteristics of the stencils can be found in Table 6.

The 27-pt stencil loads twenty-seven points first and then does the update before writing the updated value into another 3D array, which means that two separate 3D arrays are required. To be specific, the 27 points are portioned into one central point, six face points, eight corner points, and 12 edge points according to their locations and distances from the center point. The 125-pt stencil has a similar memory accessing behavior. Besides, we assume that each point has a diverse coefficient for the two stencils.

4.3 Architectural specific exceptions

Due to the hardware infrastructures and software compiler configurations, not all the benchmarks are evaluated under the same parameters. For the ARM platform, we only have 16 GB main memory, and the input grid size of 1024^3 failed as a result.

Table 5 Architectural summary of experimental platforms

Type	Superscalar out-of-order	Superscalar out-of-order
Core architecture		
SIMD	NEON	AVX
Threads/core	1	1
Clock (GHz)	2.2–2.4	2.4
DP (GFlops)	8.8	19.2
L1 data cache	32KB	32KB
Socket architecture		
Cores/socket	4	10
L2 data cache	2 MB/4 cores	256KB
Shared L3 data cache	–	25MB
primary memory parallelism paradigm	HW prefetch	HW prefetch
System architecture		
Sockets/SMP	2	1
DP (GFlops)	563.2 @ 2.2 GHz	384
DRAM BW (GB/s)	204.8	68.3
DP flop: cyte ratio	2.75	5.62
DRAM capacity (GB)	16	64
DRAM type	DDR4-2666	DDR4-2133
System power (W)	100	90
Threading	MPI + OpenMP	MPI + OpenMP
Compiler	gcc 8.3	gcc4.8

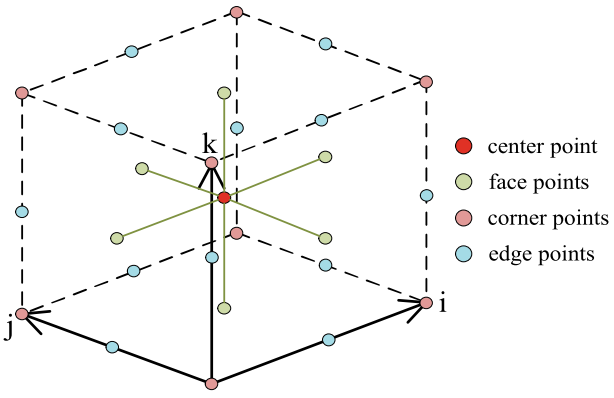

Fig. 4 3D-27pt stencil computation we concern

Besides, we only have GCC (GNU Compiler Collection) on the platform, the NEON and other related options for the ARM compiler are neglected. For the Intel platform with only two NUMA nodes, the scalability experiments are not like that on the ARM platform with eight NUMA nodes. However, it does have a broader range of problem sizes with a 64 GB main memory. Besides, we conduct single-precision experiments on the ARM platform and double-precision ones on the Intel platform separately for the NEON and AVX vector length difference. Other differences include the OpenMP scalability within a node for one of them has eight cores in a NUMA node and the other ten.

4.4 Results and analysis

We conduct the experiments according to the aforementioned configurations. Results for the ARM and Intel platforms are demonstrated in Figs. 5 and 6, respectively. We demonstrate three versions of the two stencil computations under diverse input grid sizes: the pure OpenMP version *OMP*, the hybrid MPI + OpenMP version *MPI+OMP*, and the SIMDlized version *MPI + OMP + NEON + /AVX*. The speedup is illustrating the effectiveness of the SIMD Extensions on each platform and is calculated between the *MPI+OMP* version and the *MPI+OMP+NEON/AVX* version.

4.4.1 ARM platform

Experimental results of the ARM platform present the scalability of the NUMA nodes. What can be observed is that the *OMP* version utilizing only OpenMP can only scale with cores within a NUMA node. Meanwhile, the *MPI+OMP* version presents scalability both within and among NUMA nodes. The best performance for the 3D-27pt stencil is 123.41 GFlops and the 125-pt one 117.89 GFlops. The NEON speedup for the two benchmarks is also prominent with an average improvement of 1.40× and a maximum of 2.00× for the 27-pt stencil and 2.04× and 2.38× for the 125-pt one, respectively.

To be specific, the two stencils of diverse sizes show the same conclusion. For the processes utilized from 1, 2, 4, 8, and 16 to 32, the MPI+OMP version outperforms the OMP version with better performance, while the OMP version performs no scalability. It is specially obvious when the processor cores used are among diverse NUMA nodes. For the ARM platform, one NUMA node consists of eight processor cores. Taking the 27-pt stencil of size 128 in Fig. 5a for example, the OMP version performance almost has no change from 16 cores to 32

Table 6 Average stencil characteristics (ARM and Intel)

Stencil	Total points	Read	Write	Flops per iteration	AI (DP)
27-pt	28	27	1	53	0.237
125-pt	126	125	1	249	0.247

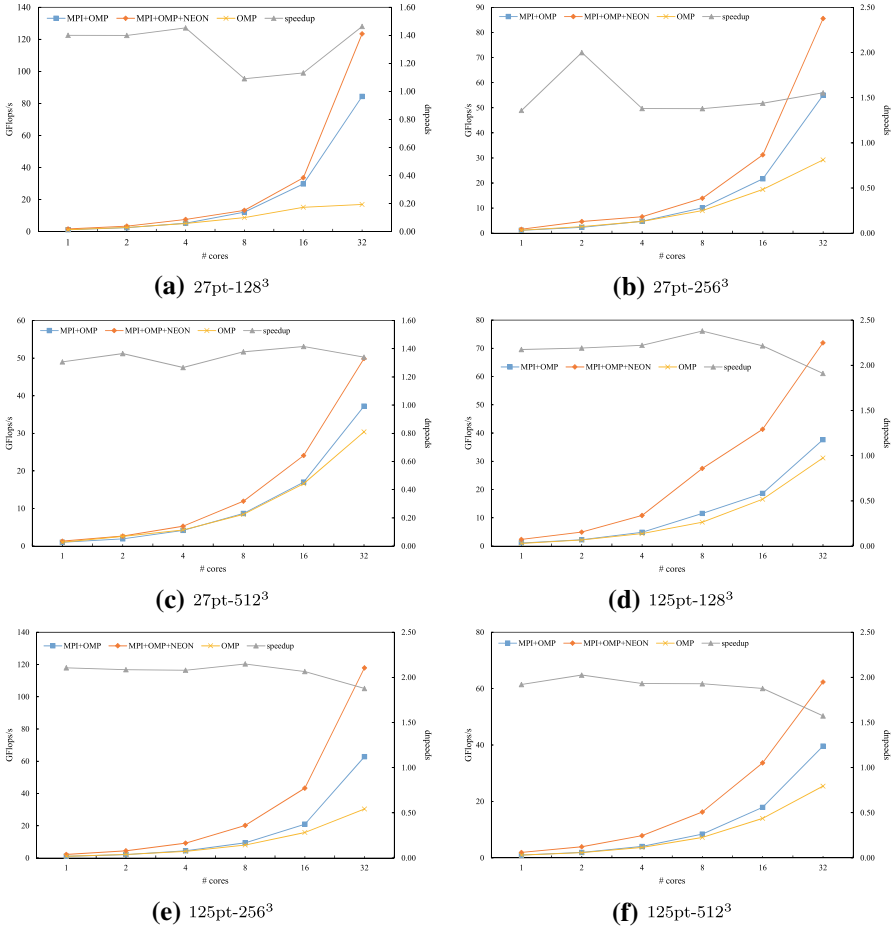

Fig. 5 Experimental results for ARM

cores. The MPI+OMP version has a speedup of about 2.0 with all the other configurations the same.

4.4.2 Intel platform

We also obtain good scalability for the two NUMA nodes and within a NUMA node by combining OpenMP and MPI for the Intel platform. We obtain the best performance for the 3D-27pt stencil at 113.06 GFlops and 159.26 GFlops for the 3D-125pt one, 29.44% and 41.47%, respectively, of the peak performance of the platform. The AVX speedup for all the benchmarks and input grid sizes behaves well with an average speedup of 1.97× and a maximum speedup of 2.17× for the 27-pt stencil and 3.51× and 4.17× for the 125-pt one, respectively.

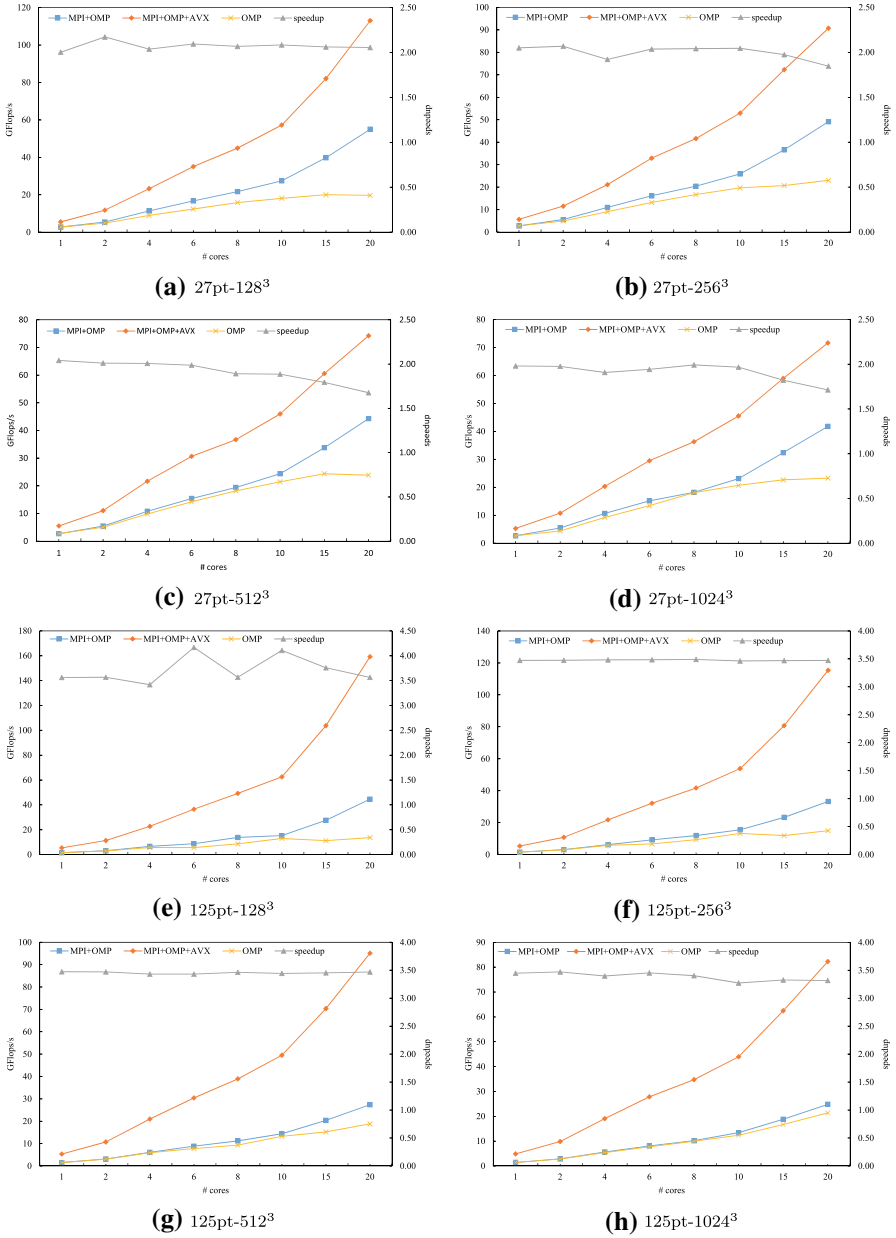

Fig. 6 Experimental results for Intel

Similar results occur on the Intel platform. For the Intel platform, one NUMA node consists of ten processor cores. Taking the 27-pt stencil of size 128 in Fig. 6a for example, the OMP version performance almost has no change from ten cores to 15, and 20 cores. The MPI+OMP version has a speedup of about 2.0 with all the

other configurations the same. These results show the effectiveness of our proposed multilevel parallelism optimization scheme.

5 Related work

To our knowledge, some correlated work was conducted. [15], and [16] proposed a DSL-based framework that automatically generates parallel high-performance stencil codes with both productivity and performance benefits within the ORWL paradigm, which is an intertask synchronization model for iterative data-oriented parallel and distributed algorithms that uses strict FIFO ordering for the access to all resources. Experiments were based on Grid'5000 [17], which is a large-scale and flexible test bed for experiment-driven research. [18] researched a multilevel autotuning framework that includes data allocation, domain decomposition, low level, and autotuning techniques. Some of their strategies, especially the decomposition schemes, are similar to ours. However, their experiments are conducted on SMP platforms using Pthreads. [19] discussed the motivation of combining OpenMP and MPI, and coarse-grained OpenMP parallelism is used to facilitate overlapping MPI communication and computation for stencil-based grid programs on an IBM SP. [20], and [21] studied the data movement strategies for heterogeneous architectures with distributed memory with the Polyhedral model on an AMD CPU and an NVIDIA GPU, which immensely reduced the volumes of communication between computing devices. [22] investigated on the hybrid MPI/OpenMP parallel programming on clusters of multicore SMP nodes. Specific cases for pure MPI, pure OpenMP, or hybrid programming model of MPI + OpenMP are analyzed separately. [6] demonstrated the benefits of the hybrid approach for performance and resource usage on three multicore-based parallel systems, including one IBM, and two SGI Altix systems, with two real-world applications. It is the first time we fuse the hybrid programming model and multilevel parallelisms on NUMA and SIMDlized architectures. We make full use of the whole platform's potential benefits, including the core, socket, and system-level architectural characterizes. Putting all the considerations together, we evaluate two commonly utilized stencil computations and exploit their best performances.

6 Conclusions and future work

We propose and implement as well as evaluate a multilevel parallelism scheme for stencil computation optimizations on NUMA architectural multicore platforms with SIMD supported. It is motivated by the fact that the systems we work on are organized by diverse memory systems, including local memory and remote memory. A hybrid programming approach of MPI and OpenMP is employed. We utilize a four-level parallelism optimization strategy including the block decomposition for NUMA nodes and processes combined with MPI, the thread-level parallelism using OpenMP, and the data-level parallelism based on SIMD Extensions. Experimental

results of two ubiquitous 3D stencil computations demonstrate the effectiveness of our proposed scheme. Future work may include the extensions of many more hardware platforms as well as stencil computations. Multinode or clusters of processors should also be considered for the performance scalability. Other optimizations including cache blocking [23–25], and [26], register blocking as well as instruction-level parallelism [18] could also be integrated into our scheme to better improve the performances.

Acknowledgements The work is supported by National Key Research and Development Program of China (2018YFB0204301) and Open Fund of PDL (6142110190201). We would also like to thank Chaorun Liu (NUDT), Peng Zhang (CAEP), Song Liu (XJTU), and the reviewers for their remarkable comments on the work.

References

1. The top-500 list of supercomputer sites. Available from: http://www.top500.org/lists/
2. Lameter C (2006) Local and remote memory: memory in a Linux or NUMA System. Linux symposium
3. Chandra R, Menon R, Dagum L, Kohr D, Maydan D, McDonald J (2000) Parallel Programming in OpenMP. Morgan Kaufman, San Francisco
4. Pacheco PS (1997) Parallel Programming with MPI. Morgan Kaufman, San Francisco
5. Gropp W, Lusk E, Skjellum A (1994) Using MPI: Portable Parallel Programming with the Message-Passing Interface. MPI Press, Cambridge
6. Jin Haoqiang, Jespersen Dennis, Mehrotra Piyush, Biswas Rupak, Huang Lei, Chapman Barbara (2011) High performance computing using MPI and OpenMP on multi-core parallel systems. Parallel Comput. 37:562–575. https://doi.org/10.1016/j.parco.2011.02.002
7. Uday Bondhugula (2013) Compiling affine loop nests for distributed-memory parallel architectures. International Conference for High Performance Computing, Networking, Storage and Analysis, SC. https://doi.org/10.1145-2503210.2503289
8. Optimizing applications for NUMA. Development topics and technologies, Intel. Published: 11/02/2011, Last Updated: 11/02/2011. https://software.intel.com/content/www/us/en/develop/articles/optimizing-applications-for-numa.html
9. Kaushik Datta, Shoaib Kamil, Samuel Williams, Leonid Oliker, John Shalf, Katherine A (2009) Yelick: optimization and performance modeling of stencil computations on modern microprocessors. SIAM Rev. 51(1):129–159
10. Henretty T, Veras R, Franchetti F, Pouchet L-N, Ramanujam J, Sadayappan P (2013) A stencil compiler for short-vector SIMD architectures. In: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing (ICS'13). Association for Computing Machinery, New York, NY, USA, pp 13–24. https://doi.org/10.1145/2464996.2467268
11. Tom Henretty (2014) Performance optimization of stencil computations on modern SIMD architectures[Ph.D]
12. Armejach Adriá, Caminal Helena, Cebrian Juan, Langarita Rubèn, González-Alberquilla Rekai, Adeniyi-Jones Chris, Valero Mateo, Casas Marc, Miquel Moreto (2019) Using Arm's scalable vector extension on stencil codes. J Supercomput. https://doi.org/10.1007/s11227-019-02842-5
13. Jang M, Kim K, Kim K (2011) The performance analysis of ARM NEON technology for mobile platforms. In: RACS'11: Proceedings of the 2011 ACM symposium on research in applied computation, pp 104–106. https://doi.org/10.1145/2103380.2103401
14. Elena L, Arseny T, Dmitry N, Vladimir A (2020) Fast implementation of morphological filtering using ARM NEON extension. arXiv:2002.09474
15. Mariem Saied (2016) Jens Gustedt. Automatic Code Generation for Iterative Multi-dimensional Stencil Computations. HiPC, Gilles Muller, pp 280–289
16. Saied, Mariem (2018) Automatic code generation and optimization of multi-dimensional stencil computations on distributed-memory architectures

17. Grid5000. Available from: https://www.grid5000.fr/
18. Kaushik Datta, Samuel Williams, Vasily Volkov, et al (2009) Auto-tuning the 27-point stencil for multicore[J]. Proc Iwapt the Fourth International Workshop on Automatic Performance Tuning
19. Kaiser Timothy, Baden Scott (2001) Overlapping communication and computation with OpenMP and MPI. Sci. Programm. 9:73–81. https://doi.org/10.1155/2001/712152
20. Dathathri Roshan, Reddy Chandan, Ramashekar Thejas, Bondhugula Uday (2013) Generating efficient data movement code for heterogeneous architectures with distributed-memory. Parallel Architectures and Compilation Techniques-Conference Proceedings, PACT. 375–386. https://doi.org/10.1109/PACT.2013.6618833
21. Bondhugula Uday, Baskaran Muthu, Krishnamo Orthy Sriram, Ramanujam J., Rountev Atanas, and Sadayappan Ponnuswamy (2008) Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. International Conference on Compiler Construction. 4959. 132-146
22. Rabenseifner Rolf, Hager Georg, and Jost Gabriele (2009) Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. Euromicro International Conference on Parallel, Distributed and Network-based Processing 427-436
23. Kamil DK et al (2009) Optimization and performance modeling of stencil computations on modern microprocessors. SIAM Rev 51(1):129–159
24. Bandishti V., Pananilath I., Bondhugula Uday (2012) Tiling stencil computations to maximize parallelism[C]// International Conference on High Performance Computing. IEEE Computer Society Press
25. Wellein G., Hager G., Zeiser T., et al (2009) Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization[C]// IEEE International Computer Software and Applications Conference. IEEE
26. Datta K, Murphy M, Volkov V et al (2008) Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC'08). IEEE Press, Article 4, pp 1–12