2021

PsmArena: Partitioned Shared Memory for NUMA-Awareness in Multithreaded Scientific Applications

Zhang Yang  
*Laboratory of Computational Physics, Institute of Applied Physics and Computational Mathematics, Beijing 100088, China*

Aiqing Zhang  
*Laboratory of Computational Physics, Institute of Applied Physics and Computational Mathematics, Beijing 100088, China*

Zeyao Mo  
*Laboratory of Computational Physics, Institute of Applied Physics and Computational Mathematics, Beijing 100088, China*

Follow this and additional works at: https://tsinghuauniversitypress.researchcommons.org/tsinghua-science-and-technology

Part of the Computer Sciences Commons, and the Electrical and Computer Engineering Commons

**Recommended Citation**

Zhang Yang, Aiqing Zhang, Zeyao Mo. PsmArena: Partitioned Shared Memory for NUMA-Awareness in Multithreaded Scientific Applications. Tsinghua Science and Technology 2021, 26(3): 287-295.

This Research Article is brought to you for free and open access by Tsinghua University Press: Journals Publishing. It has been accepted for inclusion in Tsinghua Science and Technology by an authorized editor of Tsinghua University Press: Journals Publishing.
PsmArena: Partitioned Shared Memory for NUMA-Awareness in Multithreaded Scientific Applications

Zhang Yang*, Aiqing Zhang, and Zeyao Mo

Abstract: The Distributed Shared Memory (DSM) architecture is widely used in today's computer design to mitigate the ever-widening processing-memory gap, and it inevitably exhibits Non-Uniform Memory Access (NUMA) to shared-memory parallel applications. Failure to adapt to the NUMA effect can significantly downgrade application performance, especially on today’s manycore platforms with tens to hundreds of cores. However, traditional approaches such as first-touch and memory policy fall short in false page-sharing, fragmentation, or ease of use. In this paper, we propose a partitioned shared-memory approach that allows multithreaded applications to achieve full NUMA-awareness with only minor code changes and develop an accompanying NUMA-aware heap manager which eliminates false page-sharing and minimizes fragmentation. Experiments on a 256-core cc-NUMA computing node show that the proposed approach helps applications to adapt to NUMA with only minor code changes and improves the performance of typical multithreaded scientific applications by up to 4.3 folds with the increased use of cores.

Key words: partitioned shared memory; Non-Uniform Memory Access (NUMA); heap manager; multithread; manycore

1 Introduction

The Non-Uniform Memory Access (NUMA) as a Distributed Shared Memory (DSM) architecture is becoming increasingly popular in computer design, especially in the manycore era. The Knights Landing generation of Intel Xeon Phi processors introduces sub-NUMA cluster modes for best possible performance, and the EPYC family of AMD processors uses NUMA as the major approach to maintain high memory bandwidth. ARM-based manycore processors such as Phytium 2000+ and Cavium ThunderX2 also incorporate NUMA, either intra- or intersockets. Furthermore, the computing nodes of most supercomputers are configured as multiple NUMA domains.

Failure to adapt to the NUMA architecture, i.e., achieve NUMA-awareness, can significantly downgrade the performance of multithreaded parallel applications, which can be as large as 200% for memory-bounded applications on multi-socket cc-NUMA systems[1]. However, although special care has been taken in areas such as the Message Passing Interface (MPI) runtime[2] and graph analytics[3], the area of scientific applications still largely relies almost solely on Operating System (OS)-based approaches such as first-touch and memory policy. We argue in this paper that these approaches are far from optimal, in the sense of both performance and ease of use. To be short, these approaches suffer from both false page-sharing and fragmentation, because the memory management granularity is stuck to a fixed-size page. Working around these shortcomings would require an application-specific redesign of both data structures and memory management schemes[1], which would soon be a burden for application developers and features poor ease of use.

To address the above problems, we propose a
partitioned shared memory approach in this paper. Our approach consists of an application-level conceptual abstraction of partitioned shared memory and an accompanying NUMA-aware multithreaded heap manager named PsmArena. The abstraction provides applications with a thread-partitioned view of shared memory and allows them to take full control of which thread one memory block resides, thus enabling the applications to adapt to NUMA with only minor code changes. PsmArena then manages these memory blocks to ensure no page-sharing happens across NUMA nodes and reduces fragmentation with advanced segregated storage schemes, thus ensuring memory blocks reside where they intend to reside. Compared with first-touch, our approach can improve the performance of real-world scientific applications on a 256-core cc-NUMA system by up to 4.3 folds, with only minor code changes. In summary, this paper presents the following contributions:

(1) Identification of pitfalls of OS-provided NUMA-awareness approaches. We use both theoretical analysis and experiments to show that first-touch and memory policy would suffer from false page-sharing and fragmentation.

(2) Proposing of a partitioned shared-memory NUMA-awareness approach. The proposed approach is conceptually simple and yet adequate for full NUMA-awareness of multithreaded scientific applications.

(3) Design of an accompanying multithreaded heap manager. We design and implement a general-purpose NUMA-aware multithreaded heap manager, which eliminates false page-sharing and reduces fragmentation.

The rest of this paper is structured as follows. We analyze the pitfalls of OS-based NUMA-awareness approaches in Section 2 and present the proposed partitioned shared-memory approach in Section 3 and the PsmArena heap manager in Section 4. The experimental results are shown in Section 5, and related works are presented in Section 6. Finally, the conclusion and outlook are presented in Section 7.

2 Pitfalls of OS-Based NUMA-Awareness Approaches

In this section, we analyze the pitfalls of OS-based NUMA-awareness approaches. Parallel scientific applications are often built upon the MPI+Thread model and exhibit Bulk Synchronous Parallel (BSP) execution patterns similar to Fig. 1. The application executes in locksteps and each lockstep often consists of an inter-thread exchange phase and a thread-local owner-computation phase. The combination of these two phases and multiple locksteps can make the memory access pattern highly irregular as those in multi-block applications\cite{4] and sweepings\cite{5].

The recommended approach for scientific applications to achieve NUMA-awareness is the “first-touch” approach, where the application arranges the owner of a data block to write to the pages first\cite{6]. When the first-touch is inadequate, explicit memory policy mechanisms such as “numactl”\cite{7] are used to instruct the OS where a range of pages will be placed. Behind the scene, the OS may introduce statistics-based automatic page migration to further “optimize” locality\cite{8,9]. These OS-based NUMA-awareness approaches suffer from both performance and ease of use.

First, since OS-based approaches can only work with fixed-size pages and applications usually work with variable-size blocks, fragmentation can become significant. For example, structured-grid scientific applications such as Uintah\cite{10] and JASMIN\cite{11] usually partition the computational domain into small blocks called “patches” and memory is allocated per-patch. For a patch of size 20 × 20, an array to store 20 × 20 double-precision number will be allocated, which utilizes
3.2 KB memory. Given a 4 KB page size, at least 0.8 KB memory will be wasted and a fragmentation rate of 21.9% is produced. We computed the fragmentation rate of typical patch sizes considering different page sizes and present them in Table 1. Even with the 4 KB page, one can waste 21.9% of the memory in extreme cases, and for huge pages such as the 2 MB page on x86_64, the fragmentation is unacceptably high.

Second, to mitigate the fragmentation, heap managers will kick in and share the page among threads. Since a page can reside on only one NUMA node, this creates a phenomenon that we call “false page-sharing”, where although each thread instructs the heap manager the requested memory block shall be local, at least one thread will get remote blocks. This causes unnecessary remote memory access. In the case of automatic page migration, false page-sharing can cause significant performance degradation similar to those in cache false-sharing, where the page is constantly migrated back and forth among accessing threads.

Third, for real-world applications, simple approaches such as first-touch can be impractical. For example, in multi-block applications and algebraic multigrid methods, the first thread written to a page is often not the most frequent writer and first-touch simply would not work. The only option is to explicitly mark the memory location with the numactl Application Programming Interface (API), which can be extremely costly as indicated in Section 5. Even in cases where first-touch does work, wrong decisions can often be made during automatic page migration, as regards page shall be moved, and this causes unnecessary migration overheads, as described in our experiments in Section 5.

To address the above pitfalls, applications would retreat to carefully arranging their memory access or taking control of heap management in an application-specific way, which would soon become an unacceptable burden on application developers, and thus reduces the ease of use. These facts motivate us to seek a reliable and yet general method for NUMA-awareness in multithreaded scientific applications.

### Table 1: Fragmentation rate of structured-grid applications on typical patch sizes considering different page sizes.

| Patch size Data size (Patch size × 8) (KB) | Fragmentation rate (%) |
|--------------------------------------------|------------------------|
| 4 KB page 64 KB page 2 MB page             |                        |
| 20×20                                      | 3.2 21.9 95.1 99.8     |
| 50×50                                      | 4.0 2.3 69.5 99.0      |
| 10×10×10                                   | 8.0 2.3 87.8 99.6      |
| 30×30×30                                   | 21.6 0.5 17.6 89.6     |

![Fig. 2](image)

**Fig. 2** Illustration of the partitioned shared memory approach. Threads partition the shared memory and each thread’s local part is bound to its running NUMA node.
is managed using variable-size blocks instead of fixed-size pages, thus fragmentation is reduced. This approach is also conceptually simple for application developers. For applications using the “owner-compute” rule, initializing thread simply allocates data blocks in the owner thread’s TLM. This is similar to first-touch but much more flexible, since the owner does not need to be the first writer now. This directly enables NUMA-awareness in applications such as multi-block applications and algebraic multigrid methods where “owner-compute” still holds but the first writer is not the owner. Even for those that do not use “owner-compute”, providing one can infer which thread would be the most frequent consumer of a data block, NUMA-awareness can still be achieved by allocating the memory in that consumer’s TLM.

The partitioned shared-memory software stack as illustrated in Fig. 2c addresses other pitfalls of OS-based approaches. First, unlike OS-based approaches, it works at the runtime level and serves memory as variable-size blocks instead of fixed-size pages, thus reduces fragmentation. Second, since pages can be shared among threads on the same NUMA node but never shared among threads on different NUMA nodes, false page-sharing is eliminated while keeping the fragmentation low. Third, this approach remains easy to use besides its ability to achieve full NUMA-awareness in complex real-world applications, since the only changes to application code are memory allocation calls where the allocator supplies on which thread shall the memory block resides. Lastly, the proposed approach is portable, since it only requires the OS to ensure a page is bound to the specified NUMA node.

The key ingredient of partitioned shared-memory is that the heap management will be location-aware and able to handle allocation requests with location constraints, which will be addressed by our PsmArena heap manager as described in Section 4.

4 PsmArena: A NUMA-Aware Multithreaded Heap Manager

To support the partitioned shared-memory abstraction, as well as to eliminate false page-sharing, reduce fragmentation, and mitigate overhead caused by page allocation and binding, we design and develop a location-aware multithreaded heap manager, PsmArena. PsmArena provides two fundamental memory management APIs:

1. Location-aware allocation: void* psm_alloc (size_t bytes, int owner).

2. Location-free deallocation: void psm_free (void* p).

The APIs provided by PsmArena are very low-level and are similar to the malloc and free APIs provided in C standard library. Thus PsmArena can support memory requests both directly from the application and indirectly through standard library calls (such as C++ containers). In fact, users of C++ standard containers can derive a custom allocator, which is implemented upon PsmArena, to achieve NUMA-awareness in these containers.

The PsmArena heap manager is able to address the following challenges. First, the manager will always return blocks from correct locations; second, the manager will eliminate false page-sharing and reduce fragmentation; third, the implementation will be thread-safe and scalable with tens to hundreds of cores.

4.1 Location-aware and multithread-optimized design of PsmArena

PsmArena stems from the TCMalloc heap manager\cite{14}. TCMalloc is a heap manager optimized for multicore systems, with the ability of low fragmentation from the carefully-designed segregated storage scheme.

To improve scalability, TCMalloc introduces per-core block caches and a two-level block management scheme consisting of “thread cache + central free list”. However, TCMalloc is not location-aware and suffers from both false page-sharing and returning remote blocks. For example, TCMalloc treats the global shared memory equally and migrates memory blocks back and forth among per-thread caches and central free-lists; thus, unexpected remote blocks can be frequently introduced.

We extend the TCMalloc design to that in Fig. 3. Instead of treating the heap as a whole, we divide it into several independent NUMA node heaps, whereby each manages memory blocks belonging to one NUMA node the same way as in TCMalloc. Threads are bound to cores and the pcm_alloc(bytes, threadId) call will be satisfied by the heap on whose NUMA node threadId resides. This design eliminates false page-sharing and reduces fragmentation by utilizing the advanced segregated storage scheme in TCMalloc and thus achieves NUMA-awareness. This design also improves the scalability, since except for the location-aware page allocator which allocates pages on a certain NUMA node, all locks are local to a NUMA node heap.
Together with the multicore optimization of TCMalloc, this design can scale to tens to hundreds of cores.

4.2 Location-aware block serving and recycling

With the NUMA-aware design, the remaining problem is how to ensure the blocks are served correctly with the multi-level block caching. This property is ensured by the location-aware block serving and recycling algorithm, which is an extension of the corresponding algorithms in TCMalloc\cite{14}.

To serve an allocation request $psm\_alloc(\text{bytes, threadId})$, PsmArena first identifies the core on which thread $\text{threadId}$ resides, then locks and asks the corresponding core cache to serve the block (the lock is eliminated if local). If the core cache cannot serve the block, it will lock the connected central free list and ask for a run of blocks matching the requested size, which will in turn lock and ask the page allocator to allocate and bind new pages if not able to fulfill the above request. Since many threads may request blocks from the same thread, the locks on core caches are mandatory. However, in the case where these concurrent requests are sparse, these locks will not have a significant impact on performance.

To recycle a memory block, PsmArena first queries all NUMA node heaps for which heap owns the block, which is done by checking the address against a two-level page map in “Page cache”. Then if the block belongs to the same NUMA node heap the requesting thread resides, the block is simply returned to the corresponding core cache for recycling. Otherwise, the block is returned to the central free list of the owning NUMA node heap for recycling. Further recycling is done by each NUMA node heap using the same algorithm as TCMalloc.

With the above location-aware block serving and recycling and the above NUMA-aware multithread-optimized design, PsmArena addresses the challenges as mentioned in the beginning of this section.

5 Experimental Evaluation

In this section, we evaluate the performance of the proposed partitioned shared memory NUMA-awareness approach. We first verify that the design and algorithms in PsmArena ensure correct location-aware block serving. Then we compare our partitioned shared memory against the widely used first-touch approach, considering real-world multithreaded scientific applications.

Details of our experimental platform are presented in Table 2. This platform exhibits similar NUMA effects to modern manycore processors with a cc-NUMA architecture with 32 NUMA nodes. To bind threads and avoid performance jitters, all threads were bound to the CPU cores with $KMP\_AFFINITY=\text{compact}$ in all our experiments. Thus thread $i$ is always bound to core $i$ and NUMA node $i/8$.

5.1 Verification of NUMA-awareness in PsmArena

We designed a synthetic benchmark to check if heap managers ensure local allocation as List 1. To emulate the complex but common case where threads other than the owner free the memory (for example, modern C++ applications using smart pointers), each thread will allocate sixty-four 1MB blocks, write to them, and free those of the left neighbor. The benchmark kernel is run once to warm-up the heap manager, then repeated

| Table 2 | Details of the experimental platform (a 256-core ccNUMA node). |
|---------|-------------------------------------------------------------|
| Processor | Intel Xeon 7550 (8 Cores)                                   |
| Number of processors | 32 Sockets                                                |
| Memory | DDR3 1600, 1 TB (32 GB per socket)                           |
| Number of NUMA nodes | 32                                                        |
| NUMA interconnect | SGI NumaLink 5                                           |
| Min/max NUMA distance | 1.0/6.8 (reported by the OS)                             |
| OS kernel | Linux 2.6.32                                          |
to verify the NUMA-awareness of heap managers.

```c
char* a[NTHREADS][64];
#pragma omp parallel num_threads(NTHREADS)
{
    int tid = omp_get_thread_num();
    for (int i = 0; i < 64; i++)
        a[tid][i] = alloc_memory_at(1024*1024, tid);
    // check location of pages to verify
    // correctness with get_mempolicy
    // ... get_mempolicy(b, n)...
    // write to the page to verify writing
    // performance.
    for (int i = 0; i < 64; i++)
        memset(a[tid][i], '1', 1024);
#pragma omp barrier
    for (int i = 0; i < 64; i++)
        free_memory(a[myid]);
}
```

5 times, where remote pages are checked using the `get_mempolicy` system call or the page writing time is measured. We want to check if the heap manager can retain memory pages as local. A NUMA-aware heap manager shall contain zero remote pages. We compare PsmArena with the default heap manager in GLIBC (namely ptmalloc2) and TCMalloc.

The remote page counts for GLIBC, TCMalloc, and PsmArena are presented in Table 3. The table shows that PsmArena correctly returned local pages and was NUMA-aware, while TCMalloc returned more remote pages with the increase of threads due to its NUMA-unnawareness and block caching design. Surprisingly, GLIBC returns several remote pages, even when it retreat to system mmap and first-touch for the 1 MB block size. This reveals that spurious remote page allocation can happen on such cc-NUMA systems even when first-touch is used.

| Number of threads | GLIBC | TCMalloc | PsmArena |
|-------------------|-------|----------|----------|
| 8                 | 0     | 0        | 0        |
| 16                | 0     | 112079   | 0        |
| 32                | 5     | 323038   | 0        |
| 64                | 389   | 779228   | 0        |
| 128               | 1047  | 1684069  | 0        |
| 192               | 1962  | 2598901  | 0        |
| 256               | 2317  | N.A.     | 0        |

The page writing time as presented in Table 4 shows a similar trend. Memory blocks from PsmArena take the least time to write since pages are local, while those from TCMalloc take four times more since there are many remote pages. Those from GLIBC take a surprisingly long time (100 times more with 256 threads) because the page allocation under the scene dominates the overall time. This fact indicates that naive first-touch or memory policy can be extremely costly on such cc-NUMA systems and block-caching heap managers are necessary.

### 5.2 Application performance with partitioned shared memory

We also tested the effectiveness of the proposed partitioned shared memory approach with real-world multithreaded scientific applications. We chose two applications: a 2D/3D linear advection application sample\[11\] and an electromagnetic package J Electromagnetic Solver with Finite Difference Time Domain (JEMS-FDTD)\[15\] with FDTD scheme. Both applications are memory-bounded and adhere to the “owner-compute” rule and static load balancing. We modified the memory allocation in these applications so that either the owner thread of a patch first-touches the memory or `psm_alloc` is utilized to locally allocate memory; therefore, we can compare first-touch with PsmArena. Introducing `psm_alloc` only cost renaming `malloc` to `psm_alloc(patch, bytes, patch.owner)` since the owner is known in advance.

We first show the result of the linear advection application sample in Tables 5 and 6, where the wall time of numerical computing for 100 time steps is aggregated. With the increase of NUMA nodes, the application performed increasingly better than first-touch. PsmArena started to gain an advantage at 64 threads and achieved maximum improvements of 3.3 and 2.8 folds for 2D and 3D cases, respectively. Further investigation showed that in the case where
Table 5 Accumulated kernel time (in seconds) with different NUMA-awareness approaches for the 2D linear advection application sample.

| Number of threads | Accumulated kernel time | Improvement |
|-------------------|------------------------|-------------|
|                   | First-touch | PsmArena |             |
| 8                 | 89.6       | 90.4     | -0.01       |
| 16                | 44.8       | 45.2     | -0.01       |
| 32                | 23.7       | 22.7     | 0.04        |
| 64                | 16.0       | 11.2     | 0.43        |
| 128               | 11.9       | 5.6      | 1.1         |
| 256               | 17.7       | 4.1      | 3.3         |

Table 6 Accumulated kernel time (in seconds) with different NUMA-awareness approaches for the 3D linear advection application sample.

| Number of threads | Accumulated kernel time | Improvement |
|-------------------|------------------------|-------------|
|                   | First-touch | PsmArena |             |
| 8                 | 59.6       | 60.1     | -0.01       |
| 16                | 29.8       | 30.1     | -0.01       |
| 32                | 15.6       | 15.1     | 0.03        |
| 64                | 10.6       | 7.5      | 0.41        |
| 128               | 6.9        | 3.8      | 0.82        |
| 256               | 9.1        | 2.4      | 2.8         |

multiple halo-exchange and computation locksteps were involved, under first-touch, the OS either placed pages in remote NUMA nodes or migrated pages at times; thus, it is necessary for approaches such as the proposed partitioned shared-memory.

We presented the result of JEMS-FDTD solving a plane-wave scattering problem with 100 time steps in Table 7. The data show trends similar to that of the linear advection application sample, where PsmArena performs increasingly better than first-touch with the increase of NUMA nodes. PsmArena started to gain an advantage at 64 threads and achieved maximum improvements of 4.3 folds at 256 threads. The spurious increase of computation with 256 threads needs further investigation but may be caused by overhead in the ccNUMA protocols.

Table 7 Accumulated kernel time (in seconds) with different NUMA-awareness approaches for JEMS-FDTD application.

| Number of threads | Accumulated kernel time | Improvement |
|-------------------|------------------------|-------------|
|                   | First-touch | PsmArena |             |
| 8                 | 47.5       | 46.8     | 0.01        |
| 16                | 23.7       | 23.3     | 0.02        |
| 32                | 12.4       | 12.0     | 0.03        |
| 64                | 7.3        | 6.4      | 0.14        |
| 128               | 8.4        | 4.2      | 1.0         |
| 256               | 28.1       | 5.3      | 4.3         |

6 Related Work

First-touch has long been recommended as the major approach for multithreaded scientific applications, as suggested in Refs. [6, 16], and has been practiced by applications such as stencils[17]. Beyond first-touch, advanced page placements such as the numacl API[7] and automatic page migration[8] and balancing[9] are introduced in the Linux kernel. However, these OS-based approaches can suffer from false page-sharing and fragmentation, as well as imperfect OS optimization as suggested by this paper and in algorithmic skeletons by Ref. [18].

To hide the complexity of NUMA while optimizing the performance, runtime-based approaches have been widely exploited, such as NUMA-aware OpenMP[19] and TBB[20]. Task-based runtimes and frameworks also make extensive NUMA-aware optimizations as those in PaRSEC[21], Charm++[22], and Boxlib[23]. Except for domain-specific frameworks such as Boxlib and Uintah, these optimizations cannot achieve full NUMA-awareness in complex applications since they rely on assumptions of applications’ data access patterns, and domain-specific frameworks can be restricted in their application domains.

Similar to the co-design idea in this paper, Ribeiro et al.[24] proposed Minas to allow an application to explicitly manage or automatically tune its memory affinity. Their approach does not introduce the partitioned shared-memory abstraction, and thus leaves all complexity to the application. Other approaches based on NUMA-aware heap managers include a NUMA-aware TCMalloc[25] and memkind[26]. However, the former emulates first-touch and the latter requires users to explicitly manage NUMA nodes; thus, they fall short of simple conceptual abstractions as partitioned shared memory. The partitioned shared-memory approach borrows from the PGAS idea in parallel languages such as UPC[13].

7 Conclusion and Outlook

To address the drawbacks of traditional OS-based NUMA-awareness approaches and help multithreaded parallel scientific applications to adapt to NUMA, we propose a partitioned shared-memory approach in this paper, that consists of an application-level conceptual abstraction of partitioned shared-memory and an accompanying NUMA-aware multithreaded
heap manager called PsmArena. The partitioned shared-memory abstraction enables parallel applications to explicitly reason which thread will own the allocated block, and thus maximizes their ability to achieve NUMA-awareness and preserve locality. PsmArena then ensures allocated blocks are pinned to the owner’s local memory. PsmArena manages memory at the block granularity which eliminates false page-sharing and minimizes fragmentation. Owing to the full NUMA-awareness possibility of this approach, the performance of real-world multithreaded scientific applications such as JEMS-FDTD can be improved by up to 4.3 folds on a 256-core cc-NUMA system.

One shortcoming of the proposed approach is that the application explicitly labels which thread owns the block to be allocated. We would like to investigate the possibility to infer this information with the help of a task-based runtime such as Legion, StarPU, and PaRSEC. We also plan to optimize the heap manager algorithms for better performance and scalability.

Acknowledgment
The authors would like to thank Dr. Linping Wu from High Performance Computing Center of Institute of Applied Physics and Computational Mathematics for his help on understanding the OS interferences on cc-NUMA systems. Dr. Xu Liu and Dr. Xiaowen Xu contributed several key ideas to the refinement of this paper. This work was supported by the National Key Research and Development Program of China (No. 2016YFB0201300). The authors thank the reviewers for their helpful comments.

References
[1] Z. Majo and T. R. Gross, (Mis)understanding the NUMA memory system performance of multithreaded workloads, in Proc. 2013 IEEE Int. Symp. Workload Characterization (ISW C), Portland, OR, USA, 2013, pp. 11–22.
[2] S. G. Li, T. Hoeﬂer, and M. Snir, NUMA-aware shared-memory collective communication for MPI, in Proc. 22nd Int. Symp. High-Performance Parallel and Distributed Computing, New York, NY, USA, 2013, pp. 85–96.
[3] K. Y. Zhang, R. Chen, and H. B. Chen, NUMA-aware graphstructured analytics, ACM SIGPLAN Not., vol. 50, no. 8, pp. 183–193, 2015.
[4] H. Guo, Z. Y. Mo, and A. Q. Zhang, A parallel module for the multiblock structured mesh in JASM IN and its applications, (in Chinese), Computer Eng. Sci., vol. 34, no. 8, pp. 69–74, 2012.
[5] Z. Y. Mo, A. Q. Zhang, and Z. Yang, A new parallel algorithm for vertex priorities of data flow acyclic digraphs, J. Supercomput., vol. 68, no. 1, pp. 49–64, 2014.
[6] G. Hager and G. Wellein, Introduction to High Performance Computing for Scientists and Engineers. Boca Raton, FL, USA: CRC Press, 2010.
[7] A. Kleen, A NUMA API for Linux, http://halobates.de/ numaapi3.pdf, 2005.
[8] B. Goglin and N. Furmento, Enabling high-performance memory migration for multithreaded applications on LINUX, in Proc. 2009 IEEE Int. Symp. Parallel & Distributed Processing (IPDPS09), Washington, DC, USA, 2009, pp. 1–9.
[9] M. Gorman, Automatic NUMA balancing v4, https://lwn.net/Articles/526097/, 2012.
[10] M. Berzins, J. Beckvermit, T. Harman, A. Bezdjian, A. Humphrey, Q. Y. Meng, J. Schmidt, and C. Wight, Extending the Uintah framework through the petascale modeling of detonation in arrays of high explosive devices, SIAM J. Sci. Comput., vol. 38, no. 5, pp. S101–S122, 2016.
[11] Z. Y. Mo, A. Q. Zhang, X. L. Cao, Q. K. Liu, X. W. Xu, H. B. An, W. B. Pei, and S. P. Zhu, JASM IN: A parallel software infrastructure for scientiﬁc computing, Front. Computer Sci. China, vol. 4, no. 4, pp. 480–488, 2010.
[12] R. D. Falgout and U. M. Yang, hypre: A library of high performance preconditioners, in Proc. Int. Conf. Computational Science, Amsterdam, Netherlands, 2002, pp. 632–641.
[13] T. El-Ghazawi and L. Smith, UPC: Uniﬁed parallel C, in Proc. 2006 ACM/IEEE Conf. Supercomputing, Tampa, FL, USA, 2006, p. 27.
[14] S. Ghasawat and P. Menage, TCmalloc: Thread-caching malloc, https://gperftools.github.io/gperftools/tcmalloc.html, 2009.
[15] H. Y. Li, H. J. Zhou, Y. Liu, X. F. Bao, and Z. G. Zhao, Massively parallel FDTD program JEMS-FDTD and its applications in platform coupling simulation, in Proc. 2014 Int. Symp. Electromagnetic Compatibility, Gothenburg, Sweden, 2014, pp. 229–233.
[16] R. Rabenseifner, G. Hager, and G. Jost, Hybrid MPI/OpenMP parallel programming on clusters of multicore SMP nodes, in Proc. 2009 17th Euromicro Int. Conf. Parallel, Distributed and Network-Based Processing, Weimar, Germany, 2009, pp. 427–436.
[17] M. Shaheen and R. Strzodka, NUMA aware iterative stencil computations on many-core systems, in Proc. 2012 IEEE 26th Int. Parallel and Distributed Processing Symp., Washington, DC, USA, 2012, pp. 461–473.
[18] P. Metzger, M. Cole, and C. Fensch, NUMA optimizations for algorithmic skeletons, in Euro-Par 2018: Parallel Processing, M. Aldinucci, L. Padovani, and M. Torquati, eds. Springer, 2018, pp. 590–602.
[19] F. Broquedis, N. Furmento, B. Goglin, P. A. Wacrenier, and R. Namyst, ForestGOMP: An efﬁcient OpenMP environment for NUMA architectures, Int. J. Parallel Program., vol. 38, nos. 5&6, pp. 418–439, 2010.
[20] Z. Majo and T. R. Gross, A library for portable and composable data locality optimizations for NUMA systems, in Proc. 20th ACM SIGPLAN Symp Principles and Practice of Parallel Programming, New York, NY, USA, 2015, pp. 227–238.
Zhang Yang et al.: \textit{PsmArena: Partitioned Shared Memory for NUMA-Awareness in Multithreaded Scientific ...} 295

[21] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra, PaRSEC: Exploiting heterogeneity to enhance scalability, \textit{Comput. Sci. Eng.}, vol. 15, no. 6, pp. 36–45, 2013.

[22] L. L. Pilla, C. P. Ribeiro, D. Cordeiro, and J. F. Méhaut, Charm++ on NUMA platforms: The impact of SMP optimizations and a NUMA-aware load balancer, in \textit{Proc. 4th Workshop of the INRIA-Illinois Joint Laboratory on Petascale Computing}, Urbana, IL, USA, 2010.

[23] W. Q. Zhang, A. Almgren, M. Day, T. Nguyen, J. Shalf, and D. Unat, Boxlib with tiling: An adaptive mesh refinement software framework, \textit{SIAM J. Sci. Comput.}, vol. 38, no. 5, pp. S156–S172, 2016.

[24] C. P. Ribeiro, M. Castro, J. F. Méhaut, and A. Carissimi, Improving memory affinity of Geophysics applications on NUMA platforms using Minas, in \textit{High Performance Computing for Computational Science-VECPAR 2010}, J. M. L. M. Palma, M. Daydé, O. Marques, and J. C. Lopes, eds. Springer, 2011, pp. 279–292.

[25] P. Kaminski, NUMA aware heap memory manager, \url{https://developer.amd.com/wordpress/media/2012/10/NUMA_aware_heap_memory_manager_article_final.pdf}, 2009.

[26] C. Cantalupo, V. Venkatesan, J. Hammond, K. Czurlyo, and S. D. Hammond, \textit{Memkind: An Extensible Heap Memory Manager for Heterogeneous Memory Platforms and Mixed Memory Policies}. Albuquerque, NM, USA: Sandia National Lab., 2015.

\begin{figure}[h]
\centering
\includegraphics[width=0.4\textwidth]{author_photo.png}
\caption{Zhang Yang} received the PhD degree from Chinese Academy of Sciences in 2011. He is currently an associate professor in Institute of Applied Physics and Computational Mathematics. His current research interests include high performance computing, runtime systems, and parallel programming frameworks.

\begin{figure}[h]
\centering
\includegraphics[width=0.4\textwidth]{author_photo.png}
\caption{Zeyao Mo} received the PhD degree from National University of Defense Technology in 1997. He is currently a professor and vice director in Institute of Applied Physics and Computational Mathematics. His current research interests include high performance computing and parallel computing.

\begin{figure}[h]
\centering
\includegraphics[width=0.4\textwidth]{author_photo.png}
\caption{Aiqing Zhang} received the PhD degree from China Academy of Engineering Physics in 2009. She is currently a professor in Institute of Applied Physics and Computational Mathematics. Her current research interests include high performance computing, software architecture, and numerical software developments.
