9-2011

Non-uniform Memory Affinity Strategy in Multi-Threaded Sparse Matrix Computations

Avinash Srivinasa
Iowa State University

Masha Sosonkina
Iowa State University

Non-uniform Memory Affinity Strategy in Multi-Threaded Sparse Matrix Computations

Abstract
As the core counts on modern multi-processor systems increase, so does the memory contention with all the processes/threads trying to access the main memory simultaneously. This is typical of UMA (Uniform Memory Access) architectures with a single physical memory bank leading to poor scalability in multi-threaded applications. To palliate this problem, modern systems are moving increasingly towards Non-Uniform Memory Access (NUMA) architectures, in which the physical memory is split into several (typically two or four) banks. Each memory bank is associated with a set of cores enabling threads to operate from their own physical memory banks while retaining the concept of a shared virtual address space. However, accessing shared data structures from the remote memory banks may become increasingly slow. This paper proposes a way to determine and pin certain parts of the shared data to specific memory banks, thus minimizing remote accesses. To achieve this, the existing application code has be supplied with the proposed interface to set-up and distribute the shared data appropriately among memory banks. Experiments with NAS benchmark as well as with a realistic large-scale application calculating ab-initio nuclear structure have been performed. Speedups of up to 3.5 times were observed with the proposed approach compared with the default memory placement policy.

Keywords
Memory affinity, Non-Uniform Memory Access (NUMA) node, Multi-threaded execution, Shared array, Sparse matrix-vector multiply

Disciplines
Systems Architecture

Non-uniform Memory Affinity Strategy in Multi-Threaded Sparse Matrix Computations

Technical Report #11-07,
Compute Science Department,
Iowa State University,
Ames, IA 5011,
Sep. 2011.

Avinash Srinivasa
Ames Laboratory/DOE
Iowa State University
Ames, IA 50011, USA
avinashs@scl.ameslab.gov

Masha Sosonkina
Ames Laboratory/DOE
Iowa State University
Ames, IA 50011, USA
masha@scl.ameslab.gov

Abstract

As the core counts on modern multi-processor systems increase, so does the memory contention with all the processes/threads trying to access the main memory simultaneously. This is typical of UMA (Uniform Memory Access) architectures with a single physical memory bank leading to poor scalability in multi-threaded applications. To palliate this problem, modern systems are moving increasingly towards Non-Uniform Memory Access (NUMA) architectures, in which the physical memory is split into several (typically two or four) banks. Each memory bank is associated with a set of cores enabling threads to operate from their own physical memory banks while retaining the concept of a shared virtual address space. However, accessing shared data structures from the remote memory banks may become increasingly slow. This paper proposes a way to determine and pin certain parts of the shared data to specific memory banks, thus minimizing remote accesses. To achieve this, the existing application code has be supplied with the proposed interface to set-up and distribute the shared data appropriately among memory banks. Experiments with NAS benchmark as well as with a realistic large-scale application calculating *ab-initio* nuclear structure have been performed. Speedups of up to 3.5 times were observed with the proposed approach compared with the default memory placement policy.

Keywords Memory affinity, Non-Uniform Memory Access (NUMA) node, Multi-threaded execution, Shared array, Sparse matrix-vector multiply.

1. Introduction

Transistor densities have been growing in accordance with Moore's law resulting in more and more cores being put on a single processor chip. With the increasing core counts on modern multi-processor systems, main memory bandwidth becomes an important consideration for high performance applications. The main memory sub-system can be of two types nowadays: Uniform Memory Access (UMA) or Non-Uniform Memory Access (NUMA). UMA machines consist of a single physical memory bank for the main memory, which may lead to the memory bandwidth contention when there are many application threads trying to access the main memory simultaneously. This problem of scalability may be alleviated by NUMA architectures wherein the main memory is physically split into several memory banks, with each bank associated to a set of cores, the combination of which is called a NUMA node. Hence, the memory contention may be reduced among the threads.

However, accesses to remote memory banks as in the case of large shared arrays, for example, may become painstakingly slow and may negatively affect the application scalability for higher thread counts [10]. Thus, it is imperative to carefully consider which parts of the shared data should be attributed to which physical memory bank based on the data access pattern or on other considerations. Such an attribution of data to physical main memory is often called *memory affinity* [2, 9]. This notion goes hand in hand with the CPU affinity, as noted in [6], such that the threads are being bound to specific cores for the application start and their context switches are disabled. Once threads are bound, the memory may be pinned too. On multi-core NUMA plat-

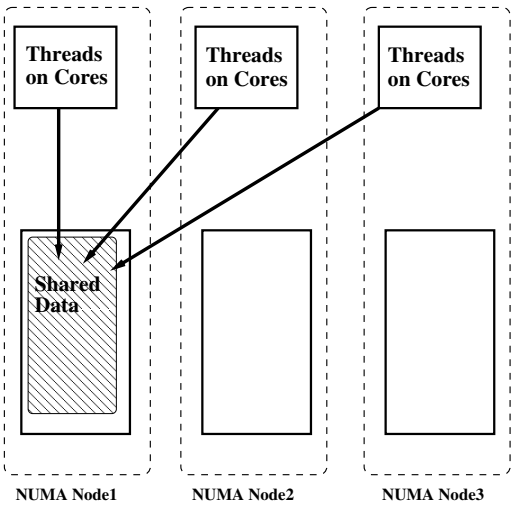

Figure 1. Shared data access pattern with the default *first-touch* policy on a NUMA architecture. A dashed curved-corner rectangle represents NUMA node.

forms, the ability to pin the memory in the application code becomes important since it is generally most beneficial for a data portion local to a thread to be placed on the memory bank local to the core it is executing on[1], so as to ensure the fastest access [1].

Conversely, the default memory affinity policy — used in most Linux-type operating systems — is enforced system-wide for all the application. This policy, called *first-touch*, ensures that there is fast access to at least one memory bank regardless of the shared data access pattern within application threads [7]. Specifically, the data is placed in the memory bank local to the thread writing to it first, which is typically done by the master thread. Thus, the downside of the first-touch policy is that all the threads accessing this shared data converge to this NUMA node, as shown in Fig. 1, causing bandwidth contention in the memory bank servicing the master thread. The problem may be exacerbated since the master thread typically initializes multiple shared data structures. Since the threads have to go out of their local NUMA node for accessing the data, the remote access latencies are also incurred, which causes the application performance overhead increase. Thus, the default first-touch memory placement policy calls for improvement to achieve better scalability, which may be obtained using already existing software libraries to work with NUMA nodes [9].

The motivation for the present work was the need for improvements in sparse matrix-vector multiplications (SpMV), which constitute the bulk of computational load in large-scale applications modeling physical phenomena using structured or unstructured matrices [15]. In particular, a nuclear physics application Many Fermion Dynamics for nuclear structure (MFDn) [17] handles very large sparse unstructured matrices arising in the solution of the underlying Schrödinger equation.

The paper is organized as follows. Section 2 describes the proposed memory placement strategy followed by the outline of its implementation and usage within sparse matrix computations (Section 3). Section 4 presents the applications tested while Section 5 provides the experimental results. In Section 6, the concluding remarks are given.

2. Proposed memory placement strategy

The goal of the proposed memory placement strategy is to minimize the data transfer overhead between main memory and the application code when accessing shared data. Hence, the default (*first-touch*) placement has to be changed according to certain application and system considerations [5]. In a nutshell, the following general steps need to be taken to study the application at hand to determine the memory placement for its shared data structures:

Step 1: Identify all the shared data structures in the application

Step 2: Classify them as having *deterministic* and *non-deterministic* access pattern by threads.
- For deterministic: Find a *chunk*-to-thread correspondence; Pin each chunk to the memory bank local to the corresponding thread.
- For non-deterministic: Spread the data across all the memory banks.

The classification step (Step 2) may be performed based on a definition of the *deterministic* and *non-deterministic* accesses to a data structure. In the former, portions of the structure is accessed by a thread exclusively, while several thread may access a portion in the latter case. This definition is rather general and is featured, for example, in the case of multi-threaded loop parallelization, such that a block of loop iterations is dedicated to a thread. If the loop index corresponds to a data portion (called *chunk*), such as that of a shared array, then each thread accesses its own array chunk exclusively. Such an array may be classified as having deterministic access and then distributed among specific memory banks. Fig. 2 presents the obtained distribution to the local NUMA nodes, such that vertical arrows emphasize the local access patterns, that minimizes the access latency. Since, for the non-deterministically accessed data structures, their thread access pattern and timing may not be known in advance, they are spread out in a fine-grain fashion across all the memory banks, as sketched in Fig. 3, in an attempt to alleviate the memory bandwidth contention. Algorithm 1 specifies array chunk sizes attributed to each thread and, consequently, to each NUMA node by accepting the following inputs:

▷ Total array dimension dim_total;
▷ Total number of threads $nthreads$;
▷ Number of NUMA nodes $mnodes$ (system parameter);

[1] Here and throughout the paper, it is assumed that only one thread is executing per core and there is no oversubscription of cores as has been studied, e.g., in [16].

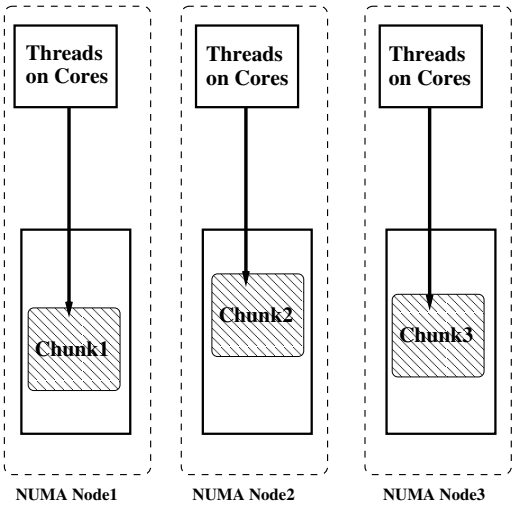

Figure 2. Proposed placement of the shared data accessed deterministically A dashed curved-corner rectangle represent NUMA node.

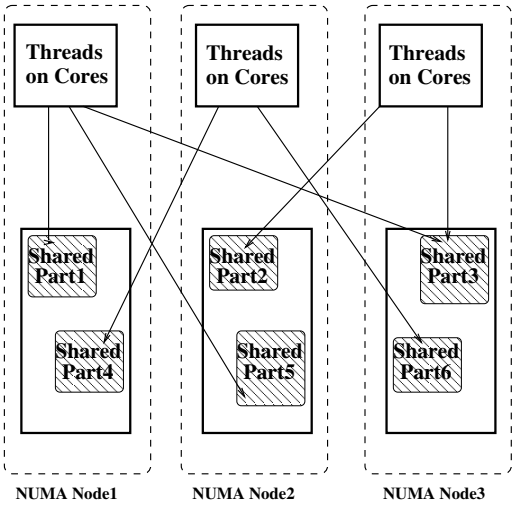

Figure 3. Interleaved placement of the shared data accessed non-deterministically. A dashed curved-corner rectangle represents NUMA node.

▷ Number of cores $lcores$ per NUMA node $lcores$ (system parameter).

and producing two outputs:

◁ Chunk size $dim\_per\_thread(i)$ attributed to thread i, $(i = 1, \ldots, nthreads)$.

◁ Chunk size $dim\_per\_node(j)$ attributed to NUMA node j, $(j = 1, \ldots, mnodes)$.

Note that each NUMA node is typically associated with several cores — thus, with a group of threads (one thread per core).

Algorithm 1 splits the data structure into chunks in accordance with the *exact assignment* thread access pattern, in which each thread is assigned an (almost) equal contiguous portion of the data structure. This pattern is common among multi-threaded programming models, such as OpenMP [3], with the default assignment size to ensure contiguous data in each chunk. Additionally, the thread scheduling (also called work-sharing) is assumed to be *static*, so that it is known before the loop execution. Thus, once the contiguous chunk sizes are determined by Algorithm 1, the actual chunk attribution is accomplished by providing a mapping of chunk number to NUMA node number, where array chunks and NUMA nodes are numbered consecutively, as in Fig. 2, for example.

Algorithm 1 Determine chunk size per NUMA node.

for $j = 1$ to $mnodes$ **do**
  $dim\_per\_node(j) \leftarrow 0$
end for
$per\_thread\_dim \leftarrow ceiling(dim\_total/nthreads)$
$virtual\_dim \leftarrow per\_thread\_dim \times nthreads$
$offset \leftarrow virtual\_dim - dim\_total$
for $i = 1$ to $(nthreads - offset)$ **do**
  $dim\_per\_thread(i) \leftarrow per\_thread\_dim$
end for
for $i = (nthreads - offset + 1)$ to $nthreads$ **do**
  $dim\_per\_thread(i) \leftarrow per\_thread\_dim - 1$
end for
for $j = 1$ to $mnodes$ **do**
  for $i = lcores \times (j-1) + 1$ to $lcores \times j$ **do**
    $dim\_per\_node(j) \leftarrow dim\_per\_node(j) + dim\_per\_thread(i)$
  end for
end for

3. Implementation details

The NUMA application programming interface (API) [9] available for Linux is used in this work to control the data placement for shared arrays, overriding the default *first-touch* memory affinity policy employed by the operating system. This API offers two principal memory placement policies called *bind* and *interleave*. The former places (binds) memory of an application on a selected memory bank or set of banks whereas the latter spreads (interleaves) data on a page-by-page basis over the memory banks of a NUMA machine. If applied throughout the entire application, each policy may be too restrictive since it is often necessary to tailor the memory attribution to a particular access pattern of a data structure [14]. For the fine-tuning purposes, the NUMA API provides a system call `mbind()` which may be used to apply these affinity polices selectively to certain regions of the memory. The `mbind()` interface has been used in this work to implement the proposed shared data placement in which certain portions of shared arrays are to be assigned to memory in accordance with their access pattern within the multi-threaded application at hand. To benefit from the selective and intelligent data placement on the memory banks, the

thread migration or their context switch have to be disabled. In other words, the CPU affinity must be observed, which may be accomplished with the `sched_setaffinity()` system call also available on Linux systems.

An important aspect to consider when using `mbind()` is that it is designed on work on large chunks of data which are aligned on a page boundary i.e., the starting address of the chunk should be an integral multiple of the system page size. So, once the shared array chunks have been determined, it becomes necessary to check whether each such chunk is page aligned before consigning it to a NUMA node. Otherwise, the nearest page-aligned address to the chunk is to be determined, starting from which the chunk can be pinned to the appropriate NUMA node. Such a pinning may cause an address mismatch between the page-aligned and the actual chunk boundaries as determined in Algorithm 1. To minimize the occurrence of the mismatches, all the shared arrays are allocated starting on a page boundary using the C function `valloc()`. With this set-up, the maximum difference between the mismatched addresses per NUMA node is estimated to be half of the system page size. Note that a typical system page is of the order of 10^3 bytes. Since high performance applications routinely work with the data in the order of gigabytes, this mismatch has no serious impact on the effectiveness of the strategies described in this work.

3.1 Shared arrays in sparse matrix-vector multiply

The sparse matrix-vector multiply (SpMV) forms an important computational core in many scientific applications. Hence, it is highly beneficial to employ its efficient implementation. Its naive implementations, however, may suffer from poor performance on multi-core NUMA architectures mainly due to the memory contention and latency problems as will be evident from the experiments described in Section 5. Therefore, the strategies described in Section 2 are being applied to sparse matrix data structures, such that the most common matrix storage formats are considered. Specifically, sparse matrices are characterized by a very large percentage — often as much as 95% — of zero entries, which are not stored for performance and space reasons. As a rule of thumb, a $n \times m$ sparse matrix is represented by three one-dimensional arrays:

- ▷ `A`, for all the non-zero values
- ▷ `jA` for their positions in the in each row or column.
- ▷ `ptrA` for the pointers to the beginning of each column or row.

Such a storage format is called Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC) depending on whether column or row indices are being stored in `jA`, respectively. Then, a multiplication of the sparse matrix stored in CSR or CSC by a vector x of size m may be performed to obtain a vector y of size n, as shown in the top and bottom code segments, respectively, in Fig. 4.

Sparse matrices are shared among the threads involved in the SpMV computation and need to be bound to local memory banks to ensure minimal data transfer overhead. As is evident from the code segments in Fig. 4, the outer loop is over the rows (for CSR) or columns (for CSC) of the matrix. For a typical multi-threaded SpMV, this loop is parallelized such that each thread gets a certain number of rows/columns to work with. However, several threads may access the same vector components to read or write their values. Since the precise timing of the read/write operations may vary dynamically, for coherent results, the sequencing of the write accesses in the CSC format (line 3 of the bottom code segment in Fig. 4) must be enforced in some way. On the contrary, in the CSR format, the SpMV has no shared arrays with simultaneous write accesses by threads, so no sequencing is needed.

```
1: for i = 1 to n do
2:     for k = ptrA(i) to ptrA(i + 1) − 1 do
3:         y(i) ← y(i) + x(jA(k)) × A(k)
4:     end for
5: end for
```

```
1: for i = 1 to m do
2:     for k = ptrA(i) to ptrA(i + 1) − 1 do
3:         y(jA(k)) ← y(jA(k)) + x(i) × A(k)
4:     end for
5: end for
```

Figure 4. Pseudo-code for sparse matrix-vector multiplication in the CSR (top) and CSC (bottom) format after the output vector initializations are performed.

Table 1. Shared array access and pinning for CSR.

Array	Access		Policy
`A`	Deterministic	Read	Bind
`jA`	Deterministic	Read	Bind
`ptrA`	Deterministic	Read	Bind
x	Non-deterministic	Read	Interleave
y	Deterministic	Write	Bind

The presence of shared arrays and parallelizable loops makes SpMV an ideal candidate for testing the proposed memory affinity policies. Once these shared arrays have been identified, they are assigned to the deterministic or non-deterministic category based on the thread access patterns. The bind and interleave strategies are then applied in accordance with the two-step strategy from Section 2. Tables 1 and 2 provide information regarding the shared array access patterns which are part of the CSR and CSC multiplications, respectively, along with the NUMA policy used for each.

It may be observed that sparse matrices are shared among the threads having exclusive access to their portions in the SpMV computation, and thus, need to be bound to local memory banks to ensure minimal data transfer overhead. On the other hand, the vectors x and y may be shared with ei-

Table 2. Shared array access and pinning for CSC.

Array	Access		Policy
`A`	Deterministic	Read	Bind
`jA`	Deterministic	Read	Bind
`ptrA`	Deterministic	Read	Bind
x	Deterministic	Read	Bind
y	Non-deterministic	Write	Interleave

ther deterministic or non-deterministic access depending on the type of the storage format considered. Thus, to effectively distribute the shared arrays with the deterministic access pattern, it becomes necessary to select specific portions (chunks) of these arrays which are accessed by each thread. To accomplish this task, the output of Algorithm 1, i.e., the chunk size of each NUMA node, is used to determine the array staring and ending indices that delineate each chunk boundaries.

Application interface. To facilitate the usage of proposed memory placement strategies, a high-level interface set, termed MASA-SpMV (Memory Affinity for Shared Arrays-Sparse Matrix Vector multiply) has been developed for sparse matrix-vector multiply in CSC or CSR formats. This interface, encapsulating the implementation of Algorithm 1, determination of the contiguous chunk start and end positions within the arrays and the memory pinning function calls from [9], is composed of the C function signatures as follows:

▷ `void masa_preprocess()`
- Determines the system constants $mnodes$ and $lcores$ required by Algorithm 1, maintained as global variables.
- Catches environment variables which deal with multi-threading, such as OMP_SCHEDULE in OpenMP.
- Disables the proposed strategies if the thread scheduling differs from static.

▷ `void masa_allocate(void **sharedarray)`
- Allocates the shared array as aligned to a page boundary.

▷ `void masa_compute_chunks(int dim_total, int nthreads, void *ptrA)`
- Computes the chunk size per NUMA node (Algorithm 1) for each shared array used in the SpMV computation.
- Finds chunk-to-node mapping for these shared arrays and stores it using global data structures.

▷ `void masa_distribute(void *A, void *jA, void *ptrA, void *x, void *y)`
- Determines the nearest page-aligned address for each shared array chunk.
- Pins the shared arrays according to the mapping calculated in `masa_compute_chunks` by calling `mbind()`.

4. Test applications

Armed with the proposed strategies and their incarnation in the MASA API interface, the proposed strategy may be employed in realistic settings of scientific application codes. Two applications have been selected for their reliance on parallel SpMV: CG (Conjugate Gradient) NAS benchmark code and an *ab-initio* nuclear structure calculation code MFDn (Many Fermion Dynamics nuclear). Both codes exploit multi-threaded parallelism using OpenMP, in which the SpMV loop is parallelized as shown in Fig. 4(top).

4.1 CG: NAS parallel benchmark

NAS Parallel Benchmarks (NPBs) is a suite derived mainly from computational fluid dynamics (CFD) codes and is composed of both entire applications and computational kernels [8]. In particular, the CG kernel been selected for this work. It consists of an iterative solution of a linear system of equations with a sparse symmetric matrix and is performed as part of a "outer" eigenvalue computation. The most computationally intensive stage of the CG iterative method is sparse-matrix vector multiplication. As implemented in the CG of the NAS suite, this multiplication stores the full matrix — without regard for its symmetry — in the CSR format. Its main loop features multi-threaded parallelism with OpenMP[2].

4.2 MFDn: Many Fermion Dynamics nuclear

MFDn is a large scale parallel code developed at Iowa State University and is used for *ab-initio* nuclear physics calculations [17]. In MFDn, the large sparse symmetric Hamiltonian matrix is evaluated in a large harmonic oscillator basis. It is then diagonalized by an iterative Lanczos procedure to obtain the low-lying eigenvalues and eigenvectors. The eigenvectors are then used to obtain a suite of experimental quantities to test accuracy and convergence of the nuclear structure. Each Lanczos iteration spends most time in SpMV with the Hamiltonian matrix, only the lower half of which is stored (in the CSC format) to save memory.

MFDn has been shown to have good scaling properties using the Message Passing Interface (MPI) [4] on existing supercomputing architectures due to the recent algorithmic improvements that significantly improved its overall performance. In [11], the use of a hybrid MPI/OpenMP approach has been presented to take advantage of the current multi-core supercomputing platforms [13]. Here, the sparse matrix data is partitioned among the available compute nodes and is being exchanged by using the MPI distributed communication library. Then, the local portions of the data are being accessed also but, this time, by using multi-threaded programming tools such as OpenMP. Fig. 5 shows the MFDn sparse matrix distribution across the available MPI processes, which are organized in the 2×2 grid. The off-diagonal processors (numbered 6 – 15 in Fig. 5) have more work to do during the SpMV phase since they have to work with the upper half of the matrix as well (for computing the transpose output vector) which is not stored in memory. The code segment in Fig. 6 describes the SpMV for MFDn

[2] This implementation has been provided by the OMNI compiler group [12].

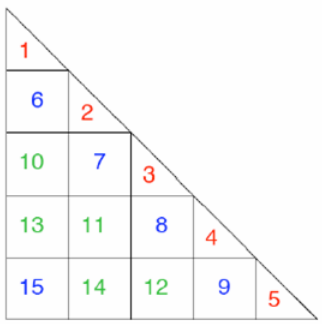

Figure 5. Two dimensional distribution of the lower triangle of the Hamiltonian matrix (Diagonal processors are numbered 1 – 5 and marked in red).

```
1: for i = 1 to m do
2:    for k = ptrA(i) to ptrA(i + 1) − 1 do
3:       y(jA(k)) ← y(jA(k)) + x(i) × A(k)
4:       y^t(i) ← y^t(i) + x^t(jA(k)) × A(k)
5:    end for
6: end for
```

Figure 6. Pseudo-code for sparse matrix-vector multiplication in MFDn for the off-diagonal processors.

on an off-diagonal MPI processor with x^t and y^t referring to the components of the input and output vectors, respectively, used with the transposed matrix (i.e., upper matrix half). In essence, this SpMV morphs the two loops shown in Fig. 4 into one, such that its multiplication operation in line 3 is the same as the one in Fig. 4(bottom). Therefore, during its multi-threaded execution, the write operation on y has to be also performed in sequence by the threads involved.

Using OpenMP, the serialization of the write operation has been implemented by way of a "critical section", which is entered one thread at a time and, thus, exhibits no parallelism hurting the code scaling at higher thread counts. Since the experiments presented here aim to test the proposed memory placement strategy for a large number of threads working in parallel, a workaround to circumvent the need for serialization has been developed for the testing purposes. Specifically, in MFDn, the SpMV has been augmented with the code to perform the multiplication in the CSR as well as CSC matrix formats, such that CSC is used for the computation of y^t and CSR for y, respectively. Fig. 7 presents the modified SpMV for an off-diagonal processor. Note that this code requires additional storage for the CSR representation of the matrix, which is reprsented by the arrays `Ar`, `jAr`, and `ptrAr`. These stand to the value, position and pointer arrays in CSR.

```
1: for i = 1 to m do
2:    for k = ptrA(i) to ptrA(i + 1) − 1 do
3:       y^t(i) ← y^t(i) + x^t(jA(k)) × A(k)
4:    end for
5: end for
6: for j = 1 to n do
7:    for k = ptrAr(j) to ptrAr(j + 1) − 1 do
8:       y(j) ← y(j) + x(jAr(k)) × Ar(k)
9:    end for
10: end for
```

Figure 7. Pseudo-code for the modified sparse matrix-vector multiplication in MFDn for the off-diagonal processors.

5. Experimental results

Experiments were conducted for the two test applications with the following problem and execution parameters:
CG: Class C with matrix size 150,000; single MPI process.
MFDn: Carbon-12 (^{12}C) nucleus with the quantum oscillation number $N_{\max} = 4$ resulting in the matrix of size 1,118,926; six MPI processes (one per compute node).

The tests were performed on the Hopper supercomputer at NERSC. Hopper is a Cray XE6 with 6,384 compute nodes. Each compute node has a cache coherent Non-Uniform Memory Access (ccNUMA) architecture with two twelve-core AMD 'MagnyCours' 2.1 GHz processors and 32 GB of RAM. The RAM is split into 4 memory banks of 8 GB each with each group of 6 cores having a direct link to one memory bank. Thus, one NUMA node is associated with six cores. All the results are reported by timing SpMV (wall-clock time) on a single compute node on Hopper. For MFDn, the maximum time is taken over all the compute nodes running off-diagonal MPI processes (Fig. 5) since they appear to be more compute-intensive with regard to SpMV.

First, a scaling study was conducted to observe the impact of the default *first touch* policy on the performance of the two applications when the number of threads is increased. Fig. 8 and 9 show the speed-ups obtained with varying thread numbers per node, normalized to the smallest considered number of threads, for CG and MFDn, respectively. Note that the case of one thread is skipped in exposition due to its triviality when investigating remote memory contention and latency in multi-threaded environments. A linear speed-up has been observed in SpMV when increasing the number of threads from one to three. From Fig. 8 and Fig. 9, it can be clearly observed that there is good scaling in moving from three to six threads. Beyond that, however, the scaling is erratic and poor, which may be be explained by remote access latencies and bandwidth contention.

Next, a performance comparison was conducted by applying the proposed memory placement strategy during the SpMV in the two applications. Fig. 10 and 11 illustrate the gains obtained with the proposed strategy compared with the

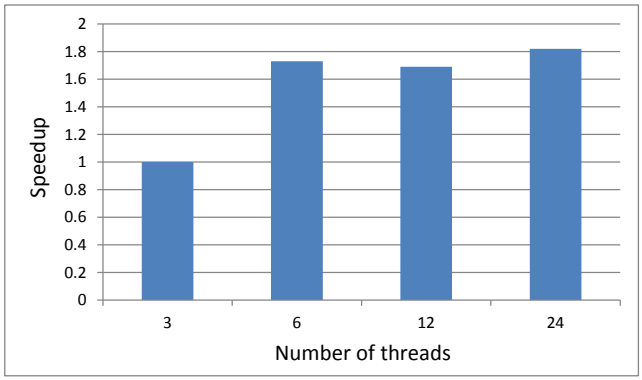

Figure 8. CG performance with *first touch* policy and increasing thread count.

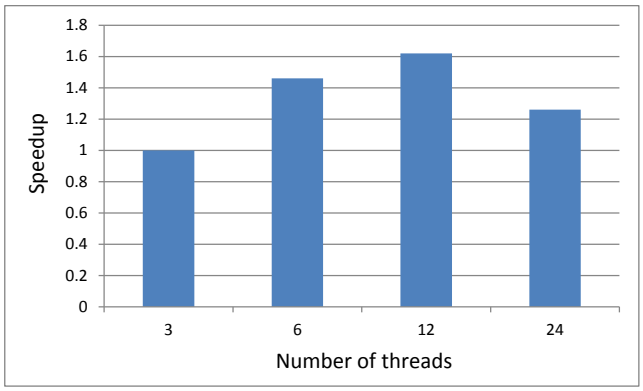

Figure 9. MFDn performance with *first touch* policy and increasing thread count.

default policy for CG and MFDn, respectively, while Fig. 12 and 13 present the "raw" speed-ups as calculated with respect to the lowest thread count used in the experiments.

From the comparison, it is clear that CG is benefiting from the proposed strategy to a much higher extent than MFDn. Additionally, the scaling for CG is almost ideal whereas it suffers for MFDn moving all the way up to 24 threads. The absence of parallelism in the critical section of SpMV hinders the performance of MFDn at higher thread counts. Hence, the SpMV as in Fig. 7 has been considered in the experiments. Fig. 14 and 15 present the obtained results for the performance gains with the proposed strategy and for the scaling, respectively. Near perfect scaling has been encountered for the modified SpMV, which proves the effectiveness of the proposed strategies within multi-threaded applications with high degree of parallelism.

6. Conclusions

This work investigates the impact of the memory affinity on multi-threaded applications when executing on multi-core NUMA architectures with multiple physical memory banks. A strategy is proposed to place the shared data into specific memory banks based on the access pattern within the

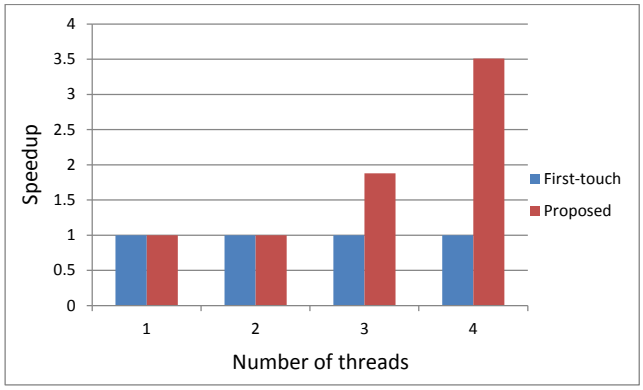

Figure 10. Performance gains of CG when the proposed strategy (red bars) is used. For each thread count, the result is normalized by the performance with the first-touch policy (blue bars).

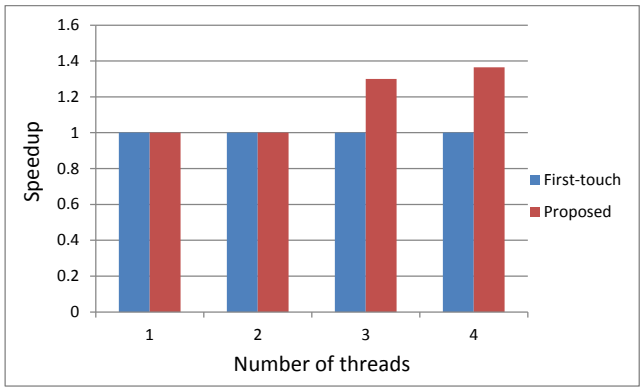

Figure 11. Performance gains of MFDn when the proposed strategy (red bars) is used. For each thread count, the result is normalized by the performance with the first-touch policy (blue bars).

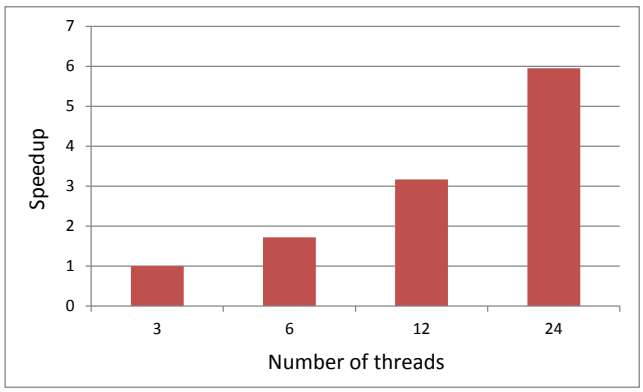

Figure 12. CG performance with proposed policy and increasing thread count.

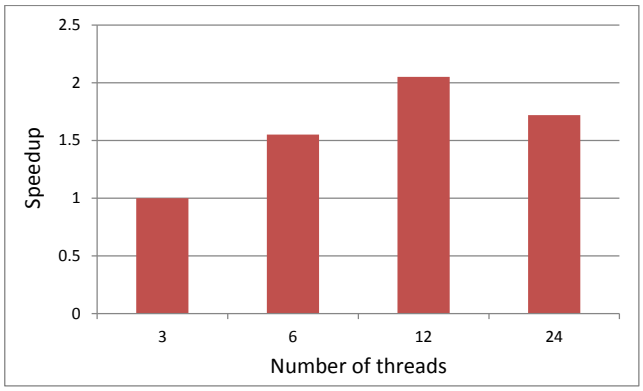

Figure 13. MFDn performance with proposed policy and increasing thread count.

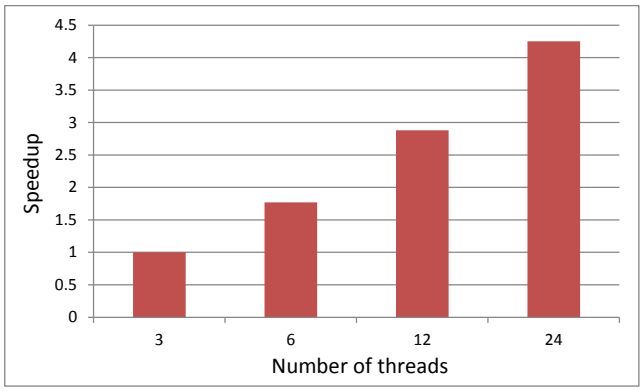

Figure 14. Performance gains of modified MFDn (SpMV) when the proposed strategy (red bars) is used. For each thread count, the result is normalized by the performance with the first-touch policy (blue bars).

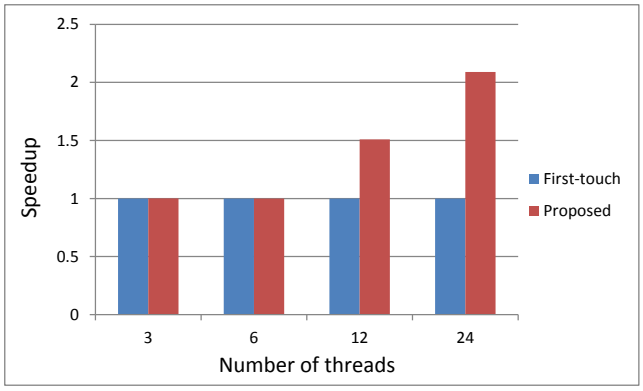

Figure 15. Modified MFDn (SpMV) performance with proposed policy and increasing thread count.

application. Specifically, the shared data is first categorized as being deterministically or non-deterministically accessed. Then, for the former, the chunk sizes are computed for the distribution to the memory banks local to the threads accessing the chunk. The data accessed non-deterministically, on the other hand, is to be interleaved across the memory banks on the uniform fine-grain (page) bases. A way of tailoring this general strategy to a computation has been also provided by considering sparse matrix-vector multiplication as a case study. For this purpose, two widely-used sparse matrix representations have been selected and an API proposed to employ the strategies within the multiplication code. By using this API, other multi-threaded computations that access shared data may be enhanced with the proposed strategy.

The new strategy overcomes the shortcomings of the default operating system placement policy that may cause remote access latencies and bandwidth contention in NUMA architectures. The experiments demonstrate a significant advantage of the careful shared data memory placement in the case of two different applications that rely on multi-threaded multiplication of a large sparse matrix by a vector. Both the CG computational kernel from the NAS parallel benchmark suite and the *ab-initio* nuclear structure calculation MFDn benefited greatly from the proposed strategies. Improvements of up to 3.5 times were observed compared with the default memory placement. For any increase in the number of computational threads on a multi-core node, an almost perfect performance scaling has been achieved.

In the future, the proposed strategy may be expanded to the hierarchical NUMA architectures as they come on-board with the advent of exascale computing platforms.

Acknowledgments

This work was supported in part by Iowa State University under the contract DE-AC02-07CH11358 with the U.S. Department of Energy, by the U.S. Department of Energy under the grants DE-FC02-09ER41582 (UNEDF SciDAC-2) and DE-FG02-87ER40371 (Division of Nuclear Physics), by the Director, Office of Science, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract number DE-AC02-05CH11231, and in part by the National Science Foundation grant NSF/OCI – 0749156, 0941434, 0904782, 1047772.

References

[1] J. Antony, P. Janes, and A. Rendell. Exploring Thread and Memory Placement on NUMA Architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/HyperTransport. pages 338–352. 2006. doi: 10.1007/11945918_35. URL `http://dx.doi.org/10.1007/11945918_35`.

[2] F. Bellosa and M. Steckermeier. The performance implications of locality information usage in shared-memory multiprocessors. *Jornal of Parallel and Distributed Computing*, 37:113–121, August 1996.

ISSN 0743-7315. doi: 10.1006/jpdc.1996.0112. URL http://dl.acm.org/citation.cfm?id=241170.241180.

[3] L. Dagnum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998. doi: http://doi.ieeecomputersociety.org/10.1109/99.660313.

[4] M. P. I. Forum. *MPI: A message-passing interface standard*, 1994.

[5] B. Goglin and N. Furmento. Enabling high-performance memory migration for multithreaded applications on linux. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–9, may 2009. doi: 10.1109/IPDPS.2009.5161101.

[6] R. E. Grant and A. Afsahi. A comprehensive analysis of openmp applications on dual-core intel xeon smps. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, march 2007. doi: 10.1109/IPDPS.2007.370682.

[7] R. Iyer, H. Wang, and L. N. Bhuyan. Design and analysis of static memory management policies for cc-numa multiprocessors. *Journal of Systems Architecture*, 48:59–80, September 2002. ISSN 1383-7621. doi: http://dx.doi.org/10.1016/S1383-7621(02)00066-8. URL http://dx.doi.org/10.1016/S1383-7621(02)00066-8.

[8] H. Jin, H. Jin, M. Frumkin, M. Frumkin, J. Yan, and J. Yan. The openmp implementation of nas parallel benchmarks and its performance. Technical report, 1999.

[9] A. Kleen. A NUMA API for LINUX. Technical report, apr 2008. URL http://whitepapers.zdnet.co.uk/0,1000000651,260150330p,00.htm.

[10] C. Lameter. Local and Remote Memory: Memory in a Linux/NUMA System. Technical report, 2009. URL http://www.kernel.org/pub/linux/kernel/people/christoph/pmig/numamemory.pdf.

[11] P. Maris, M. Sosonkina, J. P. Vary, E. G. Ng, and C. Yang. Scaling of ab-initio nuclear physics calculations on multicore computer architectures. *Procedia Computer Science*, 1(1):97–106, 2010, ICCS 2010.

[12] omni. Omni compiler software. URL http://www.hpcs.cs.tsukuba.ac.jp/omni-openmp/.

[13] R. Rabenseifner, G. Hager, and G. Jost. Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In *17th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 427–436, Los Alamitos, CA, USA, 2009. IEEE Computer Society. doi: http://doi.ieeecomputersociety.org/10.1109/PDP.2009.43.

[14] C. Ribeiro, J.-F. Mehaut, A. Carissimi, M. Castro, and L. Fernandes. Memory affinity for hierarchical shared memory multiprocessors. In *Computer Architecture and High Performance Computing, 2009. SBAC-PAD '09. 21st International Symposium on*, pages 59–66, oct. 2009. doi: 10.1109/SBAC-PAD.2009.16.

[15] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003. ISBN 0898715342.

[16] A. Srinivasa, M. Sosonkina, P. Maris, and J. P. Vary. Dynamic adaptations in ab-initio nuclear physics calculations on multicore computer architectures. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1332–1339, may 2011. doi: 10.1109/IPDPS.2011.288.

[17] P. Sternberg, E. G. Ng, C. Yang, P. Maris, J. P. Vary, M. Sosonkina, and H. V. Le. Accelerating full configuration interaction calculations for nuclear structure. In *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2008, November 15-21, 2008, Austin, Texas, USA*, pages 1–12. IEEE/ACM, 2008. ISBN 978-1-4244-2835-9.