A Locality-based Threading Algorithm for the Configuration-Interaction Method

Hongzhang Shan, Samuel Williams
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, CA 94720, USA
{*hshan, swwilliams*}*@lbl.gov*

Calvin Johnson
Department of Physics
San Diego State University
San Diego, CA 92182
cjohnson@mail.sdsu.edu

Kenneth McElvain
Department of Physics
University of California, Berkeley
Berkeley, CA 94720
kenmcelvain@me.com

***Abstract*—The Configuration Interaction (CI) method has been widely used to solve the non-relativistic many-body Schrödinger equation. One great challenge to implementing it efficiently on manycore architectures is its immense memory and data movement requirements. To address this issue, within each node, we exploit a hybrid MPI+OpenMP programming model in lieu of the traditional flat MPI programming model. In this paper, we develop optimizations that partition the workloads among OpenMP threads based on data locality, which is essential in ensuring applications with complex data access patterns scale well on manycore architectures. The new algorithm scales to 256 threads on the 64-core Intel Knights Landing (KNL) manycore processor and 24 threads on dual-socket Ivy Bridge (Xeon) nodes. Compared with the original implementation, the performance has been improved by up to 7× on the Knights Landing processor and 3× on the dual-socket Ivy Bridge node.**

I. Introduction

An important problem with applications to astrophysics, nuclear science, and investigations into fundamental symmetries and detection of neutrinos and other fundamental particles, including dark matter, is the nuclear many-body problem. For the ground state and low-lying excited states, the Configuration Interaction (CI) [7], [16] method has been widely used, with protons and neutrons as the degrees of freedom. In CI, the non-relativistic many-body nuclear Schrödinger equation is cast as a very large sparse matrix eigenpair problem with matrices whose dimension size can exceed ten billion. With even a typical sparsity (between 1 and 100 nonzeros per million matrix elements) such large-scale sparse matrix eigenpair problems place high demands on memory capacity and memory bandwidth. In reality, a matrix with dimension of one billion with a typical sparsity has at least $10^9 \times 10^9 \times 10^{-6} =$ one trillion nonzeros and will require several terabytes for storage. Some CI problems may require two orders of magnitude more memory requiring petabytes for a stored matrix representation.

To reduce the memory pressure, the BIGSTICK code [8], written in Fortran 90/95, implements the Configuration Interaction method by 1) factorizing both the basis and the interaction into two subsystems (protons and neutrons), and 2) reconstructing the nonzero matrix elements on the fly. Compared with explicit stored matrix formats, such

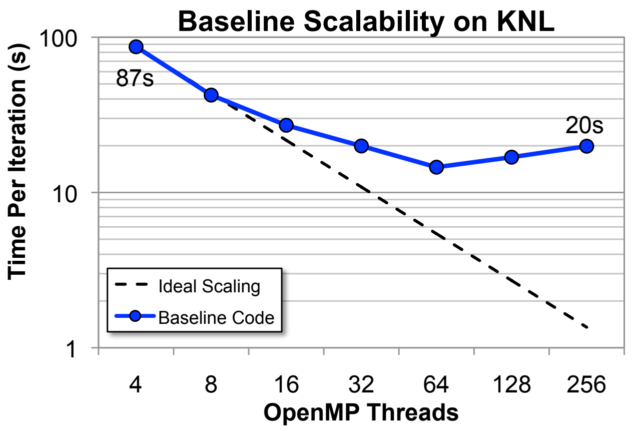


Figure 1. Time per Lanczos iteration when strong scaling OpenMP parallelism (1 MPI process) on the Knights Landing (KNL) architecture in *quadcache* for the b10nmax6 test problem. Note, the last two data points represent running multiple threads per core. Observe performance rapidly departs from the linear scaling trend line.

factorizations can reduce the memory requirements by one or two orders of magnitude. Nevertheless, the memory requirements can still exceed 1TB.

To further reduce the memory requirement, BIGSTICK may be run in a hybrid MPI+OpenMP mode with MPI for the inter-node parallelism and OpenMP for intra-node parallelism. Typically, memory requirements are independent of OpenMP parallelization, while for MPI more memory is needed as the number of tasks increases [14]. Thus, using OpenMP may save a significant amount of memory when running on emerging manycore architectures that have O(100) cores. Specifically, in BIGSTICK, OpenMP enables the sharing (rather than duplication) of both vector data as well as matrix reconstruction information. Furthermore, with only one MPI process per node, BIGSTICK can completely avoid the intra-node reduction operation needed by the sparse matrix vector multiplication across multiple MPI processes.

Figure 1 shows the strong scaling OpenMP performance of the baseline BIGSTICK implementation running on a 64-core Knights Landing processor for a small (single node class) ^{10}B test problem. Clearly, baseline performance rapidly departs from the ideal (linear) scaling line. Analysis showed performance suffers heavily from poor data locality and load imbalance at high thread concurrency. This is mainly related to the complex data access patterns of BIGSTICK that exhibit random characteristics. To address these two issues, we develop a scalable OpenMP implementation that partitions the workloads directly based on data locality.

The rest of the paper is organized as follows. In Section II, we will discuss some related work. Section III will describe the overview of the BIGSTICK algorithm followed by a discussion of both the original and optimized OpenMP implementations in Section IV and V respectively. Section VI introduces the experimental platform and two data sets, while Section VII examines the strong scaling performance on both conventional multicore and emerging manycore processor architectures. Moreover it includes a discussion of the effects of both hybrid programming models and hierarchical memory (DDR+MCDRAM) on performance. Finally, we will summarize our findings, insights, and future work in Section VIII.

II. RELATED

In our previous work [15], we have developed a weighted load balancing strategy to balance the Sparse Matrix Vectorization (SpMV) workload across MPI processes. In this work, we continue to apply this approach to partition the workload among OpenMP threads. Our goal is to develop an efficient OpenMP implementation which can scale up to all the threads on a manycore node.

Based on the treatment of the nonzero matrix elements, configuration interaction codes can be divided into two categories — those that explicitly store the nonzero matrix elements as in OXBASH [3] and MFDn code [17], or those that reconstruct the nonzero matrix elements on the fly as in ANTOINE [6], NATHAN [5], NuShellX [4], EICODE [12], and BIGSTICK [8]. BIGSTICK and MFDn are the only two codes that have been ported to large-scale distributed memory platforms [2], [13]. Compared with MFDn, BIGSTICK can solve comparable problems with an order of magnitude less memory.

Ultimately, scatter-add is the core computational challenge faced when threading the application of the Hamiltonian operator (matvec) in BIGSTICK. That is, random memory locations are incremented with the corresponding contribution from the Hamiltonian. In stored matrix representations, straightforward implementations of SpMV with CSR data layouts completely sidestep this problem. However, in MFDn, where symmetry is exploited, a sparse matrix transpose-vector multiplication is required (effectively a CSC computation). Previous work transformed the scatter-add challenges of synchronization and data locality associated with CSC through the use of compressed sparse blocks (CSB) [1].

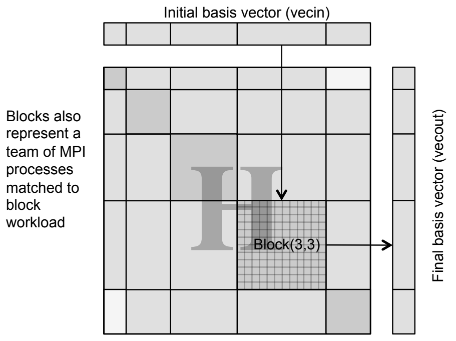

Initial basis vector (vecin)
Blocks also represent a team of MPI processes matched to block workload
H
Block(3,3)
Final basis vector (vecout)

Figure 2. A schematic diagram of the SpMV operation. The application of the Hamiltonian matrix to the initial basis vector to produce the final basis vector, is organized as operations from initial fragments to final fragments.

Similar challenges manifest in the particle-in-cell method in which particles deposit mass or charge onto a grid. As two particles may be bounded by the same grid points, there is a race condition in threaded environments when particle-to-grid interpolation is threaded over the list of particles. A number of efforts have attempted to mitigate these challenges through particle binning, particle redistribution, atomic updates, transactional memory, and brute force replication of grids [11], [10], [9]. Unfortunately, the fine-grained scatter-increments associated with BIGSTICK disqualify atomic updates while the sheer size of the vectors prohibits replication on manycore architectures.

III. BIGSTICK ALGORITHM OVERVIEW

BIGSTICK, following other CI codes, uses the iterative Lanczos algorithm to solve the matrix form of the Schrödinger equation. Its dominant computation is the sparse matrix-vector multiplication between the Hamiltonian matrix (H) and the basis vectors. To efficiently parallelize this operation, one is motivated to evenly distribute both the nonzero Hamiltonian matrix elements and the vectors across the MPI processes. To fulfill this purpose, the basis vector is divided into fragments based upon the proton substate eigenvalue M_p. Once the basis state vectors are divided into n *fragments*, the Hamiltonian matrix will be divided into $n \times n$ *blocks* correspondingly. Figure 2 illustrates this process. Each block (i, j) includes all the jump operations from fragment j of the input basis state vector to fragment i of the output vector. Because of physical constraints, mostly quantum selection rules, the nonzero matrix elements, as well as the data for reconstructing those nonzero matrix elements on the fly, is not uniformly distributed with the basis elements. To aid in distributing work and memory, the control information for reconstructing matrix elements is contained in data structures we call *bundles*.

Although generalizations to three nucleons are possible, in our experiments, the Hamiltonian operator can affect at most two nucleons at a time. Thus we can classify the elements of the Hamiltonian as PP (two protons), NN (two neutrons), or PN (one proton and one neutron); a fourth kind of element are the single-particle energies or *SPE* which only contribute elements along the diagonal. Equation 1 shows the relationship between the Hamiltonian operator and its subtypes.

$$\hat{H} = \hat{H}_{spe} + \hat{H}_{pp} + \hat{H}_{nn} + \hat{H}_{pn} \quad (1)$$

Correspondingly, the bundles which orchestrate the application of the SpMV, can be classified into four types: PP, NN, PN, and SPE. Each type has its own unique quantum characteristics, leading to significant differences in computational cost when computing the nonzero matrix elements on the fly. Therefore, empirically-derived weights are assigned to different bundle types to affect load balance [15]. Some bundle types can be further classified into backward and forward subtypes.

A. SpMV workload partition

We can exactly predict the *number* of operations in a block of Figure 2 (by operation we mean reconstruction and application of a matrix element) from the bundle information. MPI processes are assigned to blocks based on the amount of associated work required for the blocks. All processes assigned to the same block form a team. The operations and associated data will be evenly distributed among the team members. Figure 2 shows the MPI processes divided into a two dimensional array of teams. Each MPI process now owns a set of unique bundles and stores one copy of a fragment of the input and output vectors.

B. Organization of Basis Vectors

BIGSTICK uses an *M-scheme basis*, which means every many-body basis state has the same definite value of M (the z-component of the angular momentum, an example of a *quantum number*, that is, an eigenvalue which often represents a conserved quantity). BIGSTICK allows two species of fermions, typically protons and neutrons for nuclear cases. Each basis state is a simple tensor product of a proton Slater determinant (SD) and a neutron Slater determinant; a Slater determinant is the anti-symmetrized product of single particle states, i.e. protons or neutrons. All Slater determinants of a given species have the same number of particles, but may have different M, parity, and W (weighting, used for truncating the many-body basis).

As M quantum numbers are additive, the total M is M_p+M_n (the sum of proton and neutron M-values). Absent of other constraints, every proton SD with M_p not only can, but must be combined with every neutron SD with M_n where $M_n = M - M_p$. BIGSTICK applies this constraint to construct and access the many-body basis vectors. For a product basis state constructed from proton SD ip and neutron SD in, its index $ibasis$ in the basis vector can be derived using two arrays, $pstart$ and $nstart$:

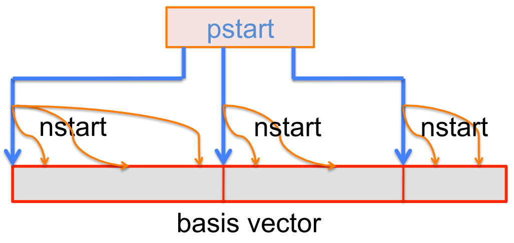


Figure 3. The relationship between basis vector, $pstart$, and $nstart$. $Pstart(ip)$ is the offset of proton SD ip in basis vector while $nstart(in)$ is the offset of neutron SD in relative to its matching conjugate proton SD. $Pstart(ip) + nstart(in)$ determine the position of a combined basis in basis vector.

$$ibasis = pstart(ip) + nstart(in) \quad (2)$$

where $pstart(ip)$ is the offset of proton SD ip in basis vector while $nstart(in)$ is the offset of neutron SD in relative to its matching conjugate proton SD. The relationship between many-body basis vector, $pstart$, and $nstart$ is shown in Figure 3.

IV. Original OpenMP Implementation: Threading Inside Bundles

In the previous section, we discussed how the SpMV workload is partitioned across MPI processes. Now, each MPI process owns a unique set of bundles that are used to orchestrate the SpMV operation, and a fragment of the initial and final basis vectors. For each bundle type, the process loops over all its assigned bundles and performs the corresponding SpMV operations. Due to different quantum characteristics, the method by which one computes nonzero matrix elements and the resultant memory access patterns to the basis vectors are different across bundle types. Nevertheless, the partitioning of work among OpenMP threads is similar.

Algorithm IV.1 shows in detail the OpenMP implementation for bundle type PN backward operations. Every PN bundle is partitioned by all OpenMP threads. A pre-computed partitioning is stored in the $bundle$ array. The workload is partitioned based on neutron SDs. For thread $mythread$, the work from bundle i is bound by $bundle(i).startn(mythread)$ and $bundle(i).endn(mythread)$ (i.e. neutron start and end). The access pattern to the basis vector $vecout$ and $vecin$ tends to be effectively random.

Figure 4(top) illustrates the results of this partitioning. There are some potential problems with this approach. First, if the partitioned loop length (Line 9 in Algorithm IV.1) is short, it may not be efficient to partition the loop across all available OpenMP threads. Moreover, if the loop length is comparable to the number of threads, there will be some

Algorithm IV.1 Original SpMV for type PN (threading inside bundle)

```
1: Partition all $PN$ backward bundles among OpenMP threads based on neutron SDs and save results into $bundle$ data
   structure
2: for $i$ = $startBundle$ to $endBundle$ do                ▷ Loop through all bundles assigned to this process
3:     if bundle($i$).type $\neq$ 'PN' then cycle end if                                        ▷ Do nothing
4:     if bundle($i$).dir $=$ backward then                                              ▷ Backward direction
5:         !$OMP Parallel
6:         mythread = omp_get_thread_num()
7:         istart = bundle($i$).startn($mythread$)
8:         iend = bundle($i$).endn($mythread$)
9:         for njmp = istart to iend do              ▷ Mythread partition based on neutron slater determinant
10:            $nsdi$ = n1b_isd(njmp)                                    ▷ Initial neutron slater determinant
11:            $nsdf$ = n1b_fsd(njmp)                                      ▷ Final neutron slater determinat
12:                                                        ▷ n1b_isd, n1b_fsd obtained through $nstart$
13:            for pjmp = bundle(i).pxstart to bundle(i).pxend do        ▷ Across proton slater determinant
14:                $psdi$ = p1b_isd(pjmp)                                 ▷ Initial proton slater determinant
15:                $psdf$ = p1b_fsd(pjmp)                                   ▷ Final proton slater determinant
16:                                                        ▷ p1b_isd, p1b_fsd obtained through $pstart$
17:                Compute corresponding matrix element xme
18:                vecout($psdf + nsdf$) += xme * vecin($psdi + nsdi$)
19:            end for
20:        end for
21:        !$OMP End Parallel
22:    end if
23: end for
```

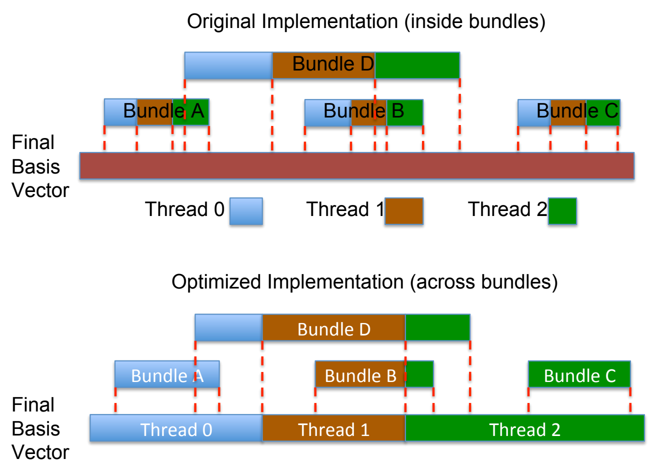


Figure 4. Example in which three threads with four bundles (A, B, C, D) access the basis vector. (top)Baseline code wherein threads within each bundle and suffers load imbalance and poor cache locality. (bottom)Optimized code has restructured the algorithm to improve data locality and facilitate load balancing

degree of load imbalance (increasingly likely on manycore processors). Second, the data accessed by an OpenMP thread may range over the entire final basis vector. This can result in cache and TLB performance issues. Finally, for the PN backward bundles, the OpenMP partitioning is based on neutron SDs, causing the loop in Line 12 (loop over proton SDs) to access the basis vector with high strides (through *pstart*) as shown in Figure 3.

Although one might suggest changing the OpenMP scheduling policy (e.g. `dynamic`), it will have no effect as the workload is statically partitioned inside the bundles to avoid data hazards. Moreover, bundles cannot be executed concurrently due to data hazards. Using locking or atomic operations causes application performance slowdown. Fundamentally, to achieve scalable performance, we must change the algorithm so that it can balance the loads across hundreds of threads and exploit the data locality so that each thread can work on its own independent data sets to avoid expensive synchronization operations.

V. Locality-based OpenMP Implementation: Threading Across Bundles

To address the problems with the original OpenMP implementation, our first step is to apply standard optimizations to improve the data locality. This includes switching the loops to avoid the high stride data access as shown in Algorithm IV.1; applying the blocking technology to improve cache reuse. Such changes have been reflected in the new OpenMP Algorithm V.1. Lines 14-23 show that we avoid the high stride access to both *vecout* and *vecin* by switching the proton SD loops to the outside while Lines 13 and 17 show the blocking technology.

Algorithm V.1 Optimized SpMV Algorithm (threading across bundles)

```
Divide the dimension of final basis vector $vecout$ into $n$ regions
Go over all the bundles to compute the weighted access frequency of each region
Based on the frequency, divide $vecout$ into $nth$ regions        ▷ $nth$ is the number of OpenMP threads
Adjust the region boundaries so that they fall on the boundaries of proton slater determinant ($pstart$)
Based on regions, partition all the bundles among threads and store the results into threadStart and threadStop arrays
!$OMP Parallel        ▷ Start OpenMP parallel
mythread = omp_get_thread_num()
for $i$ = $startBundle$ to $endBundle$ do        ▷ Loop through all bundles assigned to this process
    switch bundle($i$).type do
        case SPE, NN, PP
            Perform corresponding SpMV based on threadStart($i$, mythread) and threadStop($i$, mythread)
        case PN        ▷ Perform SpMV for type PN
            for njmpblock = bundle($i$).nxstart to bundle($i$).nxend, blocksize do        ▷ Across neutron slater determinant
                for $pjmp$ = threadStart($i$, mythread) to threadStop($i$, mythread) do
                    $psdi$ = p1b_isd(pjmp)        ▷ Initial proton slater determinant
                    $psdf$ = p1b_fsd(pjmp)        ▷ Final proton slater determinat
                    for njmp = njmpblock to min(njmpblock+blocksize, bundle($i$).nxend) do
                        $nsdi$ = n1b_isd(njmp)        ▷ Initial neutron slater determinant
                        $nsdf$ = n1b_fsd(njmp)        ▷ Final neutron slater determinant
                        Compute corresponding matrix element xme
                        vecout($psdf + nsdf$) += xme * vecin($psdi + nsdi$)
                    end for
                end for
            end for
    end switch
end for
!$OMP End Parallel
```

To address the load balancing problem and concentrate the writes, we developed a new OpenMP algorithm to partition the SpMV workload based on data locality as shown in Figure 4(bottom). The final basis vector is partitioned among the OpenMP threads so that each thread will be responsible for all the write operations to its region. To balance the SpMV workload, we cannot simply partition the final basis vector based on its dimension. Instead, we must partition based upon the number of write operations and the cost of each operation. The cost of each update operation is the weight associated with each bundle type.

To balance the workload, we first partition the index space of the final basis vector into a fixed number of regions. We then scan through all bundles to estimate the weighted write frequencies for these regions. Based on the estimated frequencies, we divide the basis vectors among OpenMP threads with equal weighted update operations. In addition, we need to adjust the region boundaries so that they fall exactly on the $pstart$ position, which points to the position of proton SDs in the basis vector. Otherwise, the loop at Line 17 of Algorithm V.1 may need to be split across OpenMP threads, increasing the complexity of the algorithm.

As long as we calculate the index space for each OpenMP thread, we can revisit all the bundles to compute the corresponding fragment for each OpenMP thread and store the results into the $threadStart(bundles, threads)$ and $threadStop(bundles, threads)$ data structures. This time, the partition is based on proton SDs.

Line 6 creates the OpenMP parallel region. For each OpenMP thread, it will scan over all the bundles and work only on its own part as shown in Line 14 of Algorithm V.1 and Figure 4. Therefore, during the whole SpMV operation, each OpenMP thread will write its own partition of the final basis vector and no longer spread all over. Furthermore, short loops may no longer need to be partitioned as long as they update the same thread's basis region.

VI. Platforms and Data Sets

In this paper, we examine techniques to improve strong scaling on a node via OpenMP. To that end, we examined two architectures (multicore and manycore) and two data sets (small and large).

A. Ivy Bridge

The Ivy Bridge (IVB) node contains two Xeon sockets each with 12 out-of-order superscalar cores running at 2.4 GHz with 256b wide vector units and two hardware threads. Each core includes private 32KB L1 and 256KB L2 caches, and each processor includes a shared 30MB L3 cache with over 300GB/s of bandwidth. The node has 64GB of DDR3-1866 memory. The nominal STREAM [18] bandwidth to DRAM is roughly 103 GB/s.

B. Knights Landing

The Knights Landing (KNL) node contains a single, self-hosted Intel Xeon Phi processor with 64 out-of-order superscalar (but to a lesser degree than Ivy Bridge) cores running at speed of 1.4GHz. Each core has a 32KB L1 data cache, two 512b vector units, and four hardware threads. Each tile (2 cores) shares a 1MB L2 cache. The node contains 16GB of MCDRAM and 96GB DDR4 2133 memory providing about 80 GB/s of bandwidth. For most experiments in this paper, we have configured the MCDRAM as a direct mapped L3 cache and configure the directory in quadrant mode (*quadcache*). This provides about 350 GB/s of STREAM bandwidth for arrays that fit in cache. Overall, the memory hierarchies look very similar to the IVB node with the caveat that the KNL node's L3 cache is 25$\times$ larger but half the bandwidth. Nevertheless, KNL cache bandwidth is nearly 4$\times$ higher than IVB main memory bandwidth.

C. Data Sets

We select two problems, b10nmax6 (^{10}B) and fe52 (^{52}Fe) that present different computational challenges. The first problem, b10nmax6 is an *ab initio* calculation (also called a *no-core shell model* calculation) that has 5 protons and 5 neutrons (^{10}B); the designation $N_{\max} = 6$ describes the model space and signifies the maximum excitation in units of harmonic oscillator energies. There are 176,844 Slater Determinants (SD) for both species. There are about 18,143 bundles used to reconstruct the nonzero elements for the sparse Hamiltonian matrix. The basis vector size is around 12 million and the number of nonzero elements is about 13 billion. Nevertheless, in BIGSTICK, the total memory footprint is less than 16GB memory and thus should fit in the MCDRAM cache.

Fe52 has 6 valence protons and 6 neutrons, with a frozen ^{40}Ca core. There are 38,760 Slater Determinants (SD) for both species, but only 484 bundles are needed to reconstruct the nonzero elements for the Hamiltonian matrix. The basis vector size is 110 million and there are approximately 152 billion nonzeros — more than 10$\times$ the number in b10nmax6. Nevertheless, the nonzero density is significantly lower (10^{-6} for Fe52 versus 10^{-4} for b10nmax6). The total memory footprint exceed 16GB memory thus resulting in MCDRAM capacity misses. These two data sets represent two common types of configuration interaction calculations for nuclear structure.

VII. Performance Results and Analysis

In this section, we will study the strong scaling performance on both the Ivy Bridge and Knights Landing architectures. We show the performance improvement in two steps. The first step focuses on the data locality which shows the performance improvement due to avoiding the long stride data access and cache blocking (labeled as "Locality") and the next step shows the cumulative improvement with the new OpenMP workload partition (labeled as "Balanced").

A. Strong Scaling Performance on Ivy Bridge

Figure 5(left) shows the strong scaling performance for b10nmax6 as a function of the number of OpenMP threads using scatter affinity (sockets, then cores, then hardware threads). The original implementation does not scale well beyond 16 OpenMP threads. Improving the data locality can reduce the run time by about 25% across all concurrencies, but does not improve scalability.

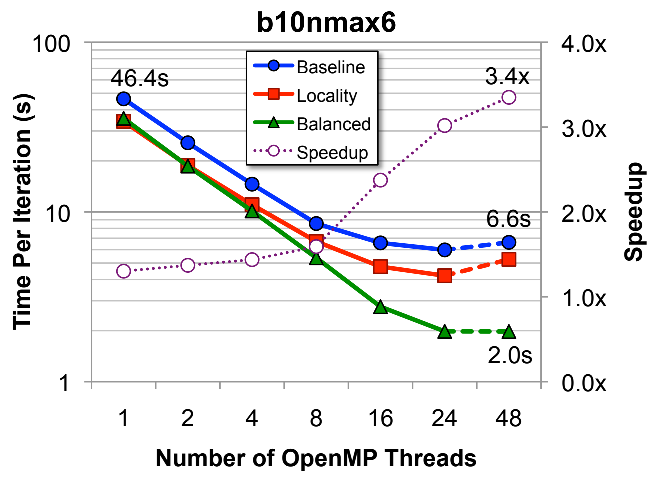
To understand why scalability was unaffected, we show the time spent in SpMV on each OpenMP thread when running with a total of 24 threads in Figure 5(right). If the original SpMV implementation were perfectly load balanced the line would be flat. However, it is clearly imbalanced with the slowest cores requiring nearly 1.3$\times$ more time than the average. The optimizations for locality improved SpMV time on each thread, but did not affect load balance.

The original algorithm partitions each bundle among all available OpenMP threads synchronizing after each bundle. Due to certain quantum characteristics of b10nmax6, its bundles are dominated by short loops and are thus a challenge to partition among ever increasing numbers of threads. Figure 5 (right) shows that the application of the new partitioning algorithm substantially improves load balance at 24-way threading. As a result, overall scalability is substantially improved at high thread concurrency (Figure 5 left).

The overall picture is somewhat different for Fe52 (see Figure 6 left). Optimizations for data locality improved performance by 45% — far more than on b10nmax6, but not surprising given the larger basis vectors. However, the improved load balancing algorithm had almost no effect. As shown in Figure 6 (right), execution was relatively balanced (albeit slow) to begin with. Note, compared with b10nmax6, the Fe52 basis size and the total number of nonzero matrix elements has increased by over 10$\times$. However, the total number of bundles was reduced by over 35$\times$. As the work within each bundle increased by over 350$\times$, it was much easier to load balance across large numbers of threads.

B. Strong Scaling Performance on Knights Landing

Figure 7 shows strong scaling performance on Knights Landing as a function of the number of OpenMP threads. We use 64 cores on this architecture and each core supports 4 hardware threads. For b10nmax6 data set (left figure), improving the data locality reduces the wallclock running times up to 64 threads (64 cores). Beyond 64, where multiple threads contend for resources on a core and on a tile (L2), performance begins to degrade. Although the



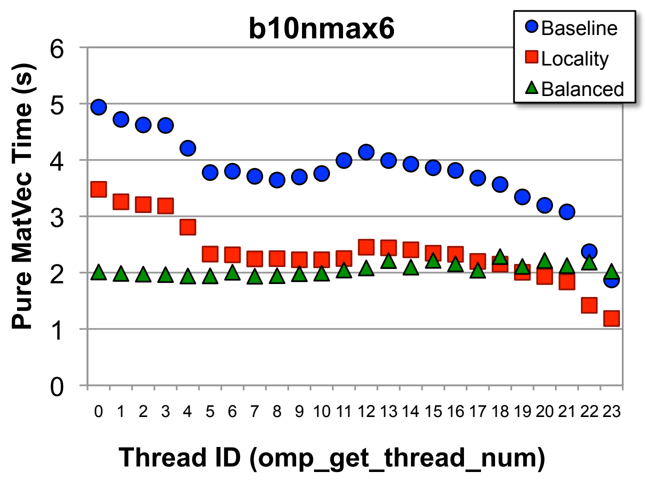
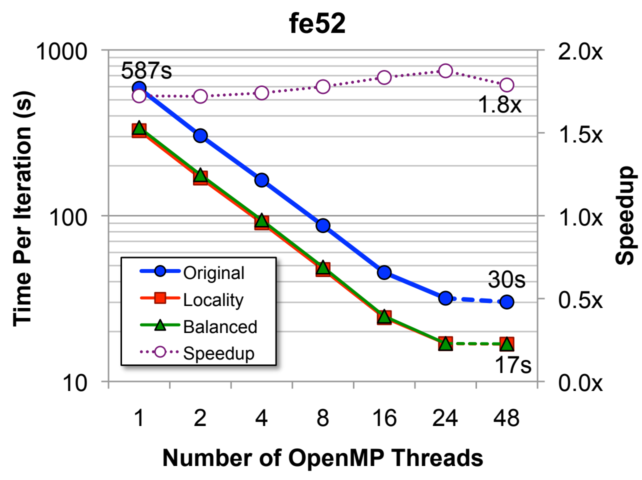


Figure 5. The OpenMP strong scaling performance on the 24-core Ivy Bridge node for b10nmax6 (left). The dashed line segments represent the HyperThreading results. And the load balancing across on the 24-core Ivy Bridge node for b10nmax6 (right). Note, each data point represents the time spent by that particular thread. Observe the large speedup from load balancing.



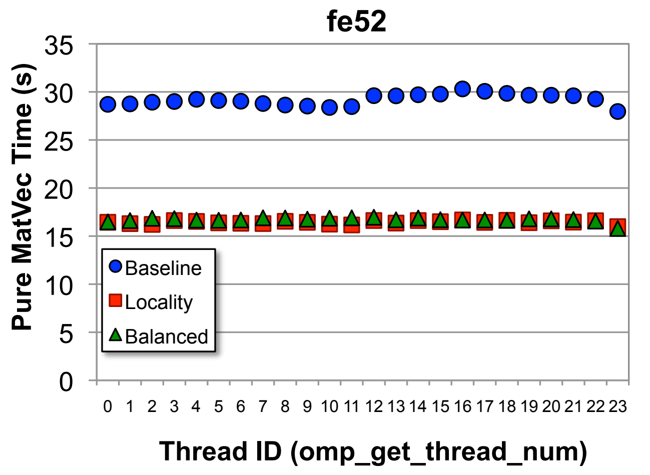


Figure 6. The OpenMP strong scaling performance on the 24-core Ivy Bridge node for fe52 (left). The dashed line segments represent the HyperThreading results. And the load balancing across on the 24-core Ivy Bridge node for fe52 (right). Note, each data point represents the time spent by that particular thread. Observe the large speedup from data locality.

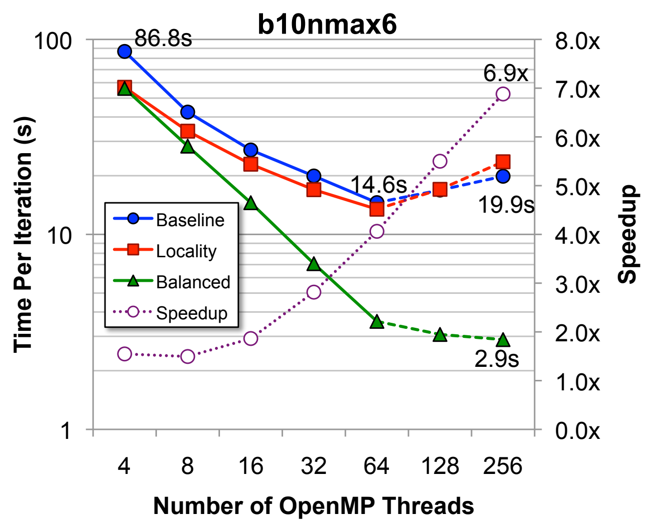
locality optimizations have some benefit at small scale, as on Ivy Bridge, algorithmic changes to affect load balancing substantially improved performance on Knights Landing delivering near linear scaling through 64 cores with 4 hardware threads per core further improving performance.

For fe52, even the baseline OpenMP implementation scales well up to 64 threads and provides some continued improvements through 256 threads. Improving the locality can reduce the running times around 40% across all concurrencies except the 256 thread case. Similar to Ivy Bridge on fe52, the new load balancing algorithm generates no performance improvement up to 128 threads (in fact some degradation in performance), but does provide further benefit at 256 thread concurrencies.

Compared with the Ivy Bridge platform, the performance effect of HyperThreading on Knights Landing is much more pronounced. This is probably related with how many threads are needed to saturate the memory and cache bandwidth (hide latency). Further analysis is left for future work.

C. *Hybrid Programming Models*

Nominally, motivated by memory capacity, it is often most efficient to run BIGSTICK with 1 MPI process per node (and thus maximize OpenMP concurrency on a node). Nevertheless, it is possible, for some configurations, to run multiple MPI processes on a node. In this subsection, we will study how the optimized load balanced implementation performs as one trades OpenMP parallelism for increased



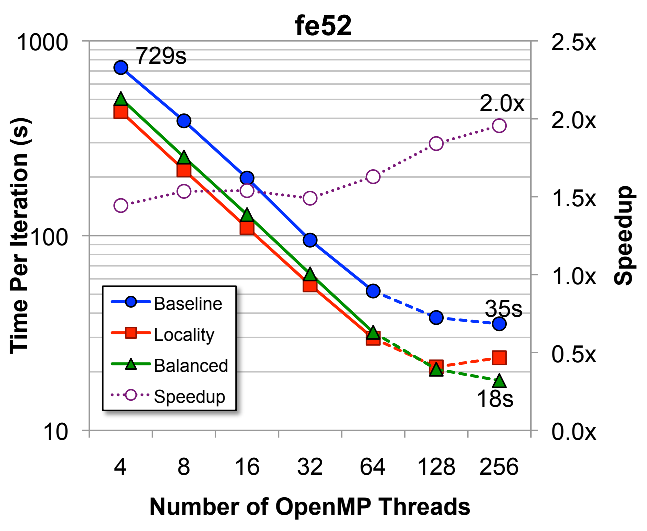
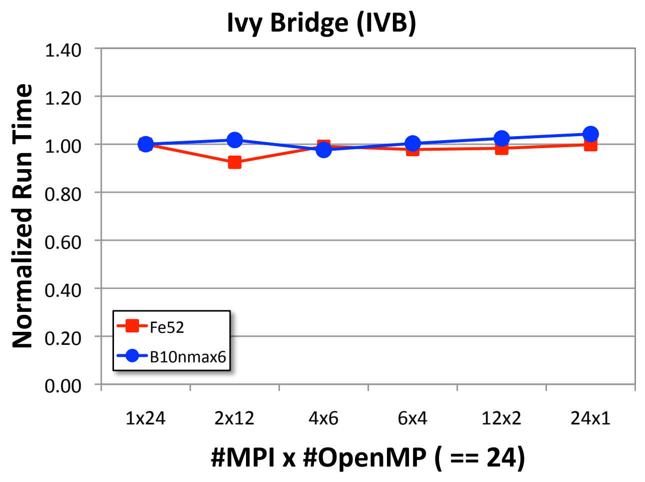


Figure 7. The OpenMP strong scaling performance on the 64-core Knights Landing architecture for b10nmax6 (left) and fe52 (right). The dashed line segments represent the SMT results.



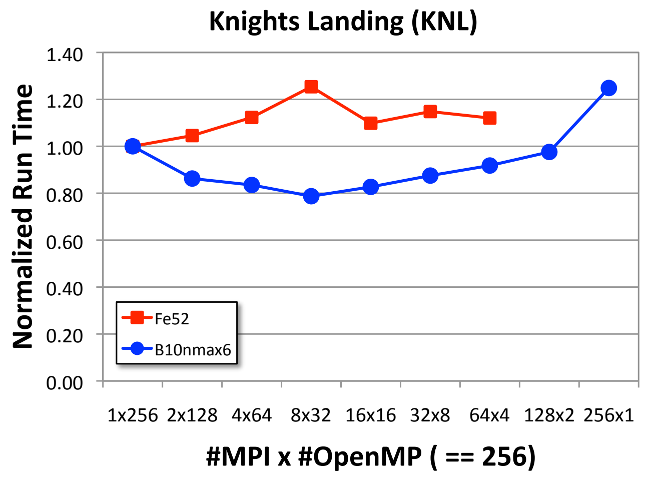


Figure 8. The benefit of a hybrid programming model on performance on the Ivy Bridge (left) and Knights Landing (right) systems. The right most data represents a traditional flat MPI mode. Note, for data set fe52 on Knights Landing, there is insufficient memory capacity in cache mode for the 128x2 and 256x1 data points.

process parallelism.

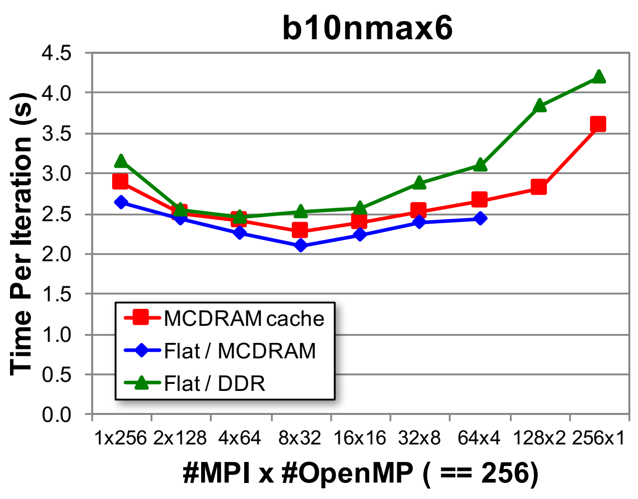
Figure 8 shows the running times in a hybrid execution mode normalized to pure OpenMP as one varies the number of MPI processes and OpenMP threads while fixing total concurrency to 24 threads on the Ivy Bridge and 256 threads on the Knights Landing. Overall, the performance effect on the Ivy Bridge is quite small for both data sets.

The performance effect on the Knights Landing is more nuanced. For the fe52 data set, the best performance is obtained when one MPI process and 256 OpenMP threads are used. Analysis showed that although load balancing is easily managed for this configuration, the time spent in MPI collectives on vectors (shown in Table I and required for 2D parallelizations of SpMV) generally correlated with overall run time. Future work will examine techniques to mitigate these effects in a multi-node KNL environment.

Table I
THE MPI REDUCTION TIME PER ITERATION ON KNIGHTS LANDING. NOTE, ORDERING IS #MPI×#OPENMP.

	1x256	2x128	4x64	8x32	16x16	32x8	64x4	128x2	256x1
Fe52	0	0.72	1.14	3.58	0.97	1.22	1.10	N/A	N/A
B10nmax6	0	0.05	0.14	0.10	0.11	0.12	0.03	0.20	0.21

The best run time for the smaller data set, b10nmax6,



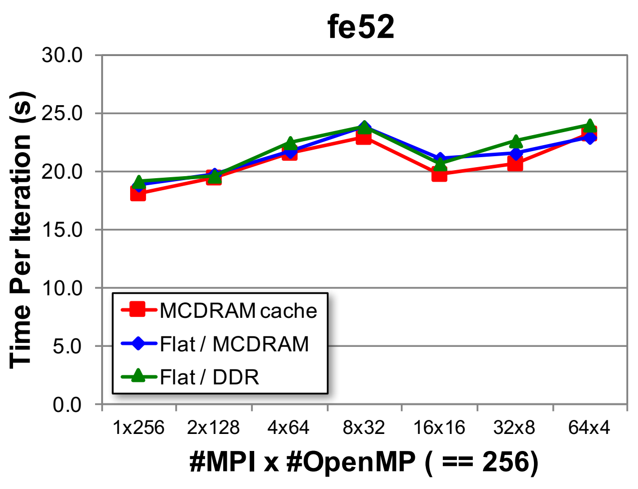


Figure 9. The effect of memory architecture configuration (*quadcache* vs. *quadflat*) on KNL for b10nmax6 (left) and fe52 (right). Note, there is insufficient MCDRAM memory capacity in *quadflat* for the 128x2 and 256x1 data points for b10nmax6 and insufficient memory for them in any mode for fe52.

is obtained with 8 processes of 32 threads — the result of minimizing load imbalance (at 256 threads, load imbalance was around 70%). As we described in Section III and IV, the bundles are first partitioned among the MPI processes and each process gets its own set of the bundles. Then, the bundles will be partitioned among OpenMP threads. For these two partitionings, the workload is estimated using the same approach, the number of SpMV operations and its associated costs. Currently, we use one cost for each bundle type. This simple approach causes more load imbalance when parallelism is dominated by either OpenMP or MPI. Clearly, a more accurate cost estimation method is necessary for high OpenMP concurrency for relatively small data sets like b10nmax6.

D. Memory Configuration

On Knights Landing, the 16GB of MCDRAM can be configured at boot time to operate in one of three modes. First, one can configure MCDRAM to be a giant L3 cache on the DDR memory (all data presented thus far used this *quadcache* mode). Alternatively, one can configure MCDRAM as a second NUMA node ("Flat" memory mode). In such a regime some addresses (MCDRAM) are fast and others (DDR) are slow. It is the responsibility of the programmer to allocate data in the appropriate NUMA node, or, as we did, rely on `numactl` to either bind memory or preferably allocate memory in the desired NUMA node ("Flat / MCDRAM") or in DDR ("Flat / DDR").

Figure 9 shows the run times for b10nmax6 and fe52 on Knights Landing as a function of MPI/OpenMP and hierarchical memory configuration/usage. Note, "MCDRAM Cache" represents the baseline data previously presented in the paper, while "Flat / MCDRAM" and "Flat / DDR" represents pinning data to either the MCDRAM or DDR NUMA nodes in the flat memory mode. For b10nmax6 (left), best performance is obtained when memory is allocated in MCDRAM while the worst performance is obtained when memory is allocated from DDR memory. Cache mode delivers the performance between Flat/MCDRAM and Flat/DDR. Although the results fit well with our expectation, the performance differences are far smaller than the differences in bandwidths indicating we are far from the bandwidth limit for this code. (n.b., due to smaller MCDRAM capacity we could not run all the test cases when allocating only in MCDRAM memory and thus use `--preferred` allocation via `numactl`). For fe52 (right), all three cases deliver similar performance with "Cache" being slightly better.

VIII. Conclusions

Thread parallelism is a key technology of manycore architectures. Developing scalable algorithms that can make effective use of all threading resources on a manycore node is critical in harnessing the massive threading parallelism provided by the manycore architectures. Generally speaking, we face two contending forces in the manycore era: data locality and load balancing in highly threaded environments. The baseline implementation of BIGSTICK attempted to address these challenges by organizing computations around bundles wherein all threads would collaborate on one bundle at a time. On architectures with large shared (and obviously coherent) caches multiple threads could realize constructive locality. Similarly, when loop parallelism greatly exceed thread parallelism, load balancing was easily attained.

Unfortunately, on manycore processors like Knights Landing, both of these underlying assumptions are invalid. There are no large, low-latency, on-chip caches, but rather an archipelago of tiles with a private cache and a few threads.

Moreover, thread parallelism can match, if not greatly exceed, loop parallelism. As a result, a major restructuring of the computations in BIGSTICK was required.

The implementation we developed restructures computations so that threads are loosely coupled and work on independent data sets (obviating the need for fine-grained synchronization or atomic operations) spanning multiple bundles in BIGSTICK. We demonstrate scalability to 256 OpenMP threads on the Knights Landing architecture and 24 threads on a dual-socket Ivy Bridge node. On two very different configuration interaction nuclear structure calculations, the resultant implementation outperforms the original by up to 7× and 3× on KNL and IVB respectively. The results indicate that improving the data locality by restructuring the codes is critical to the viability of manycore architectures. However, we observe that hierarchical memory architectures had little effect on performance (cores likely underperform due to a lack of vectorization).

In the future, we will migrate our KNL-based optimization efforts to the "Cori" XC-40 supercomputer at NERSC. There, we will continue to improve vectorization on KNL, study the effects of different KNL clustering modes (e.g. SNC4) on performance, improve the performance of `MPI_Allreduce` vector operations on the Knights Landing, and examine ways to more accurately estimate the workload to enable effective load balancing.

Acknowledgements

This material is based upon work supported by the Advanced Scientific Computing Research Program in the U.S. Department of Energy, Office of Science, under Award Number DE-AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] H. M. Aktulga, A. Buluc, S. Williams, and C. Yang. Optimizing sparse matrix-multiple vector multiplication for nuclear configuration interaction calculations. *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, 2014.

[2] H. M. Aktulga, C. Yang, E. G. Ng, P. Maris, and J. P. Vary. Improving the scalability of a symmetric iterative eigensolver for multi-core platforms. *Concurrency and Computation: Practice and Experience*, 26:2631–2651, 2014.

[3] B. Brown, A. Etchegoyen, and W. Rae. Computer code OXBASH: the Oxford University-Buenos Aires-MSU shell model code. *Michigan State University Cyclotron Laboratory Report No. 524*, 1985.

[4] B. A. Brown and W. D. M. Rae. The Shell-Model Code NuShellX@MSU. *Nuclear Data Sheets*, 120:115–118, 2014.

[5] E. Caurier, G. Martinez-Pinedo, F. Nowacki, A. Poves, J. Retamosa, and A. P. Zuker. Full $0\hbar\omega$ shell model calculation of the binding energies of the 1f7/2 nuclei. *Phys. Rev. C*, 59:2033–2039, 1999.

[6] E. Caurier and F. Nowacki. Present status of shell model techniques. *Scopus Preview*, 30:705–714, 1999.

[7] C. J. Christopher. *Essentials of Computational Chemistry*, pages 191–232. John Wiley & Sons, Ltd., ISBN 0-471-48552-7, 2002.

[8] C. W. Johnson, W. E. Ormand, and P. G. Krastev. Factorization in large-scale many-body calculations. *Computer Physics Communications*, 184:2761–2774, 2013.

[9] K. Madduri, E.-J. Im, K. Z. Ibrahim, S. Williams, S. Ethier, and L. Oliker. Gyrokinetic particle-in-cell optimization on emerging multi- and manycore platforms. *Parallel Computing Journal*, 2011.

[10] K. Madduri, J. Su, S. Williams, L. Oliker, S. Ethier, and K. Yelick. Optimization of parallel particle-to-grid interpolation on leading multicore platforms. *Transactions on Parallel and Distributed Systems (TPDS)*, 2012.

[11] K. Madduri, S. Williams, S. Ethier, L. Oliker, J. Shalf, E. Strohmaier, and K. Yelicky. Memory-efficient optimization of gyrokinetic particle-to-grid interpolation for multicore processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 48:1–48:12, New York, NY, USA, 2009. ACM.

[12] https://www.jyu.fi/fysiikka/en/research/accelerator/nuctheory/Research/Shellmodel.

[13] H. Potter, D. Oryspayev, P. Maris, M. Sosonkina, and et. al. Accelerating Ab Initio Nuclear Physics Calculations with GPUs. *Proc. 'Nuclear Theory in the Supercomputing Era - 2013' (NTSE-2013)*, 2014.

[14] H. Shan, H. Jin, K. Fuerlinger, A. Koniges, and N. J. Wright. Analyzing the Effect of Different Programming Models Upon Performance and Memory Usage on Cray XT5 Platforms. In *Cray User Group Conference (CUG)*, 2010.

[15] H. Shan, S. Williams, C. Johnson, K. McElvain, and E. Ormand. Parallel Implementation and Performance Optimization of the Configuration-Interaction Method. In *SC '15 Proceedings of 2015 International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2015.

[16] I. Shavitt. The history and evolution of configuration interaction. *Molecular Physics*, 94:3–17, 1998.

[17] P. Sternberg, E. Ng, C. Yang, P. Maris, J. Vary, M. Sosonkina, and H. V. Le. Accelerating configuration interaction calculations for nuclear structure. *The Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008.

[18] www.cs.virginia.edu/stream/ref.html.