High-Performance Exact Diagonalization Techniques

Sergei Isakov
Institute for Theoretical Physics
ETH Zurich
CH-8093 Zurich, Switzerland
isakov@itp.phys.ethz.ch

William Sawyer
Swiss National Supercomputing Centre (CSCS)
CH-6900 Lugano, Switzerland
wsawyer@cscs.ch

Gilles Fourestey
Swiss National Supercomputing Centre (CSCS)
CH-6900 Lugano, Switzerland
fourestey@cscs.ch

Adrian Tineo
Swiss National Supercomputing Centre (CSCS)
CH-6900 Lugano, Switzerland
atineo@cscs.ch

Neil Stringfellow
Swiss National Supercomputing Centre (CSCS)
CH-6900 Lugano, Switzerland
nstring@cscs.ch

Mattias Troyer
Institute for Theoretical Physics
ETH Zurich
CH-8093 Zurich, Switzerland
troyer@phys.ethz.ch

***Abstract*—In this work we analyze performance and scalability of Exact Diagonalization (ED) techniques for correlated quantum many-body systems. Typical quantum models give rise to a real symmetric or complex Hermitian Hamiltonian matrix H, which is sparse with tens to hundreds of non-zeros per row, depending on a model. We choose one of the simplest as a prototype benchmark case: a one-dimensional Ising model with periodic boundary conditions, where quantum dynamics has been introduced through a transverse magnetic field.**

The ED technique uses the Lanczos algorithm to determine a few eigenstates of H. The sparsity pattern is irregular, and the underlying matrix-vector operator, Hx, generally exhibits only limited data locality, leading in a naive implementation to all-to-all communication among p compute nodes. Classically this has proved to be an impediment to running on large machine configurations.

Grouping the basis states in a smart way, guided by the structure of the underlying lattice, one can reduce the communication requirements so that each node communicates with only an order O(log(p)) subset of nodes. The price paid for this data locality is the need to copy most of the node-local vector entries to the immediate neighbors. In other words, a large amount of data is communicated to only a limited number of neighbors. In the resulting C++ implementation, communication is performed with MPI_Isend/Irecv primitives. Within each node, OpenMP multi-threading is employed. The resulting hybrid implementation scales well to large CPU-configurations, and we report results from the 1492-node Interlagos-based Cray XE6 at CSCS.

We have also investigated alternative paradigms for the communication, such as one-way MPI-2 and SHMEM primitives. Moreover, a UPC implementation was derived from the above-mentioned prototype Ising model written in C. By utilizing Cray-specific functionality along with UPC one-sided communication primitives, the overall execution time of the MPI two-sided communication can be exceeded, albeit only by a modest amount (10-15%). In this paper, we present the results of various communication paradigms on Cray XE6, SGI UV1000, Intel Westmere and Sandybridge.

Depending on the model chosen, the Hx operation can be computationally expensive. This opens the possibility of offloading the local computation to graphical processing units (GPUs). We briefly discuss the possibilities for achieving this with the current state of the OpenACC standard implemented in the PGI and Cray C++ compilers.

***Index Terms*—High performance computing, quantum mechanics, computer simulation**

I. Introduction

Understanding the properties of correlated quantum many-body systems is a very challenging task. The full analytical treatment is usually not possible and one has to resort to numerical methods. One of the widely used numerical methods is exact diagonalization (ED) of the Hamiltonian matrix of the quantum system [1]. Various bosonic, fermionic and spin models, e.g. Heisenberg and Hubbard models, can be studied on different lattices. Hamiltonian matrices are typically sparse with tens to hundreds of non-zeros per row. We calculate the matrix on the fly as the memory requirement is very high for large problem sizes, even though the matrix is sparse. In this paper we mostly focus on diagonalizing a transverse field Ising model, probably the simplest quantum spin model. This model still can have non-trivial physics, especially on frustrated lattices [2].

The underlying kernel in ED is the Lanczos tridiagonalization [5], given in Algorithm 1, to determine a small number of eigenstates of H. After r iterations a tridiagonal matrix is generated:

Algorithm 1 *Lanczos iteration*

1: $v_1 \leftarrow$ random vector with norm 1
2: $v_0 \leftarrow 0$
3: $\beta_1 \leftarrow 0$
4: **for** $j = 1, \ldots, r$ **do**
5: $\quad w_j \leftarrow Hv_j - \beta_j v_{j-1}$
6: $\quad \alpha_j \leftarrow (w_j, v_j)$
7: $\quad \beta_{j+1} \leftarrow \|w_j\|$
8: $\quad v_{j+1} \leftarrow w_j / \beta_{j+1}$
9: **end for**

$$T_H = \begin{bmatrix} \alpha_1 & \beta_2 & & \\ \beta_2 & \alpha_2 & \ddots & \\ & \ddots & \ddots & \beta_r \\ & & \beta_r & \alpha_r \end{bmatrix}$$

The eigenvalues of T_H can either be an approximation for the true eigenvalues of H, or they can be spurious because of the round-off errors. Ways to distinguish the latter can be found in the literature [4]. The algorithm contains two potentially expensive kernels:

- a matrix-vector multiplication Hv_j,
- a dot-product operation (w_j, v_j), which is also used to calculate norms, $\|w_j\|$.

We investigate first a highly simplified benchmark version of ED, which is coded in C and employs a simple transverse field Ising model [2]. We refer to this simplified spin model as "SPIN" in order to avoid confusion with the full C++ implementation supporting multiple models, referred to simply as "ED".

Our prime interest in both SPIN and ED is to investigate high performance programming paradigms for message passing, thread parallelization and acceleration on multicore processors such as GPUs, and to compare the efficacy of these paradigms on a variety of computer platforms. Section II proposes a methodology for the comparison of a spectrum of current and emerging architectures. Various message-passing paradigms will be analyzing in Section III, and evaluated on Cray XE6 and SGI UV1000 platforms. In Section IV the performance of a hybrid OpenMP-MPI implementation of SPIN is evaluated, while in Section V the hybrid implementation of the full ED application is considered. We draw some conclusions about the efficacy of these paradigms in Section VI, in which we also discuss potential for acceleration of SPIN and ED using OpenACC [3] directives.

II. Methodology for comparative benchmarking

Comparing hardware platforms is fraught with difficulty. Since different architectures provide widely varying numbers of sockets, dies and cores per node, have different power requirements, and have a wide range of non-uniform access, comprehensive comparisons are virtually impossible. Our methodology to compare architectures is a compromise: we first compare single core, single socket and single node performance, taking into account the best performance for any thread count, as the architecture might or might not support hyperthreading. We use the most performant compiler on the given architecture, which is generally the vendor-supplied compiler, e.g., `opencc` for AMD, `craycc` for Cray or `icc` for Intel, and report on cases where another compiler (usually GNU 4.6.2) performs better.

The platforms available (Tab. I) for single socket/node comparisons were the following:

- **Rivera:** Single-node AMD Interlagos testbed, 2 sockets
- **Sandy:** Single-node Intel Sandybridge testbed, 2 sockets
- **Castor:** Intel Westmere cluster with 32 nodes, each with 2 sockets

System name	Rivera	Sandy	Castor
Processor	AMD 6274	Intel E5-2680	Intel X5650
Proc. nickname	Interlagos	Sandybridge	Westmere
Clock (GHz)	2.2	2.7	2.66
Sockets/Node	2	2	2
Cores/Socket	16	8	6
NUMA/Socket	2	1	1
DP GFlops/Socket	140.8	172.8	63.8
Memory/Socket	16 GB	16 GB	12 GB
DDR3 mem. speed	1600	1333	1333
L1 cache (excl.)	16KB	32KB	32KB
L2 cache/# cores	2MB/2	256KB/1	256KB/1
L3 cache/# cores	8MB/8	20MB/8	12MB/6
Hyperthreading?	no	yes (2)	unenabled
TPA/Socket (W)	115	130	95

TABLE I
CSCS TESTBED PLATFORMS

After the single node evaluation, we evaluate the performance of the hybrid MPI/OpenMP implementation over multiple nodes. The parallel platforms (Tab. II) available for the comparison where the following:

- **Rosa:** Cray XE6 with 1496 nodes (2x16-core AMD Interlagos socket) and a Gemini interconnect.
- **Todi:** Cray XT6 with 176 nodes (1x16-core AMD Interlagos socket); each node also has an NVIDIA M2090 graphical processing unit, which was not used in these tests.
- **Rothorn:** SGI UV1000 with 1 (virtual) node consisting of 32 8-core Intel Westmere sockets and a NUMAlink interconnect. This machine has highly non-uniform access within this single node, so that multi-threading is best performed over one, or very few, sockets.

For the comparison the optimal number of MPI processes per node (generally not just one) was selected, as well as the optimal number of threads per MPI process (which was not always equal to the number of cores assigned to a process). In view of the wide range of sockets per node, the timing comparison is made on a *socket-for-socket* basis. However, this comparison still cannot be considered fair if other metrics are considered, e.g., energy-to-solution, or the price of hardware.

III. Message-passing implementation of the SPIN benchmark

In this simplest case, a lattice with n sites, each with two possible spin states and no lattice symmetries is considered. There are therefore 2^n total possible states to the system Vectors of length 2^n are distributed over 2^m processes, with each local segment containing 2^{n-m} entries. Scientifically interesting values for n would be in the range of $35-50$; for testing typically values $20-30$ are used. The novelty in this technique is the partitioning of H such that the each local process requires only data from a small neighborhood of processes, which increases in size as $O(m)$ rather than $O(2^m)$. Within this neighborhood, however, all the vector entries must be exchanged. The communication pattern of an idealized case is depicted in Fig. 1, indicating a banded structure. The size of neighborhood is just $\log_2(\text{number of processes}) = m$. The algorithm has a computational intensity of roughly 1 floating point operation per one double word (8 bytes) and is thus network- and memory-bandwidth limited.

Fig. 2 depicts the code structure of the Lanczos iteration. In this benchmark version a double buffer (`vv1, vv2`) is employed (see Fig. 3) to overlap the current matrix-vector product with the subsequent data exchange.

The above discussion applies only to a highly-idealized SPIN problem, using the simplest possible quantum model. The full C++ exact diagonalization (ED) application currently under development supports a wide range of 1D, 2D, and 3D lattices, and several quantum models [1] including fermionic and spin models, as well as the quantum Hall effect. The size of neighborhood depends on the locality of the interaction terms in the Hamiltonian and the coordination number of the lattice. For local interactions, the size of neighborhood is usually bounded by the logarithm of the total number of processes. The full application can also take symmetries into account to reduce the Hilbert space size: the size of the neighborhood stays more or less the same, but the overall problem size is reduced roughly by a factor equal to the number of symmetry operations.

The initial version of the SPIN benchmark utilized MPI-

System name	Rosa	Todi	Rothorn
Product name	Cray XE6	Cray XT6	SGI UV1000
Interconnection	Gemini	Gemini	NUMAlink
Processor	AMD 6272	AMD 6272	Intel E7-8837
Proc. nickname	Interlagos	Interlagos	Westmere
Clock (GHz)	2.1	2.1	2.66
Sockets/Node	2	1	32
Cores/Socket	16	16	8
NUMA/Socket	2	2	1
DP GFlops/Socket	134.4	134.4	85.1
Memory/Socket	16 GB	16 GB	64 GB
DDR3 mem. speed	1600	1600	1333
L1 cache (excl.)	16KB	16KB	32KB
L2 cache/# cores	2MB/2	2MB/2	256KB/1
L3 cache/# cores	8MB/8	8MB/8	24MB/8
Hyperthreading?	no	no	no
TPA/Socket (W)	115	115	130

TABLE II
CSCS parallel platforms

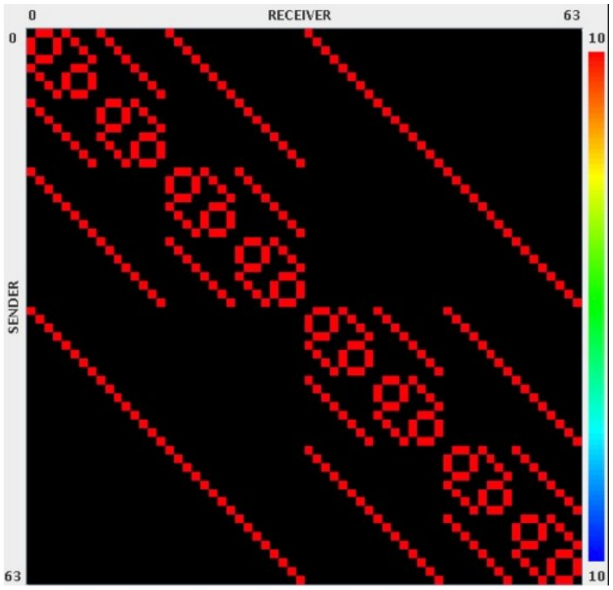

Fig. 1. Communication matrix for an idealized test problem with $m = 6$ and $n = 20$. Each process sends 2^{14} entries to its neighbors.

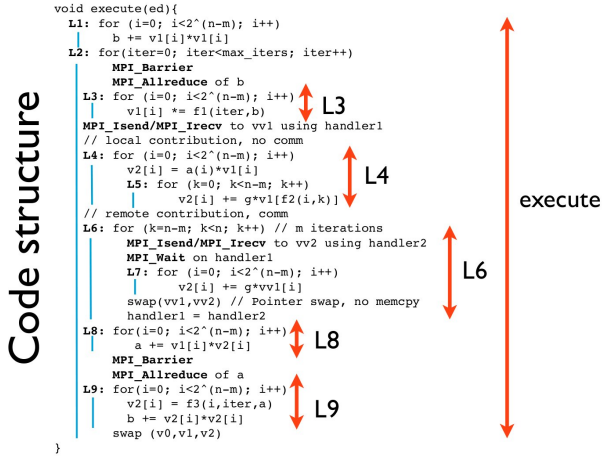

Fig. 2. MPI implementation of Lanczos iteration for SPIN model

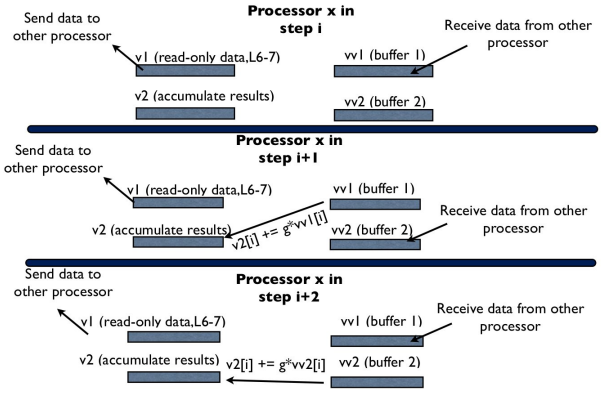

Fig. 3. Double buffering scheme in the SPIN prototype

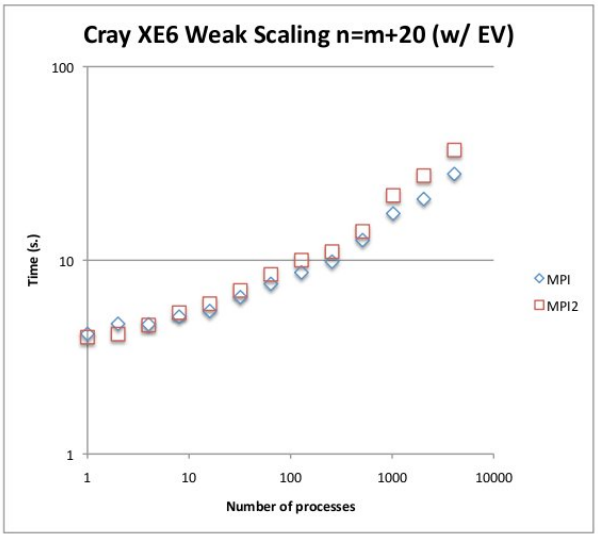

Fig. 4. Cray XE6 weak scaling with $n = m + 20$ for unthreaded MPI-1 (two-sided) and MPI-2 (one-sided) communication paradigms.

1 functionality and a double-buffering scheme as depicted in Figs. 2 and 3. We refer to this subsequently as the "Work" version. This scheme was replaced by one-way communication from MPI-2 (Sec. III-A), SHMEM (Sec. III-B) and UPC (Sec. III-C) as supported in the Cray C compiler. The implementations and their resulting performance are discussed in the following subsections.

A. MPI one-way

One straightforward way to reimplement the algorithm with one-sided communication entails replacing Isend/Irecv pairs, e.g.,

```
MPI_Isend(ed->v1,ed->nlstates,MPI_DOUBLE,ed->to_nbs[0],
          ed->nm - 1, MPI_COMM_WORLD, &req_send1);
MPI_Irecv(ed->vv1,ed->nlstates,MPI_DOUBLE,ed->from_nbs[0],
          ed->nm - 1, MPI_COMM_WORLD, &req_recv1);
```

with corresponding `MPI_Put` and fence operations:

```
MPI_Put(ed->v1, ed->nlstates, MPI_DOUBLE, ed->to_nbs[0],
        0, ed->nlstates, MPI_DOUBLE, win1);
MPI_Win_fence( 0, win1);
```

Separate windows `win1, win2` are created to represent the `vv1` and `vv2` buffers. The global reductions remain the same.

Performance results (Fig. 4) reveals a slight degradation in performance with this naive one-sided communication design with respect to the MPI two-sided version.

B. SHMEM

In a fashion similar to MPI-2 one-sided, SHMEM can be used to implement the double buffering scheme. However, since the SHMEM paradigm is actually a simple implementation of partitioned global address space (PGAS), it is necessary to create aligned temporary buffers which replace the original `vv1` and `vv2`:

```
vtmp1 = (double *) shmalloc(ed->nlstates*sizeof(double));
vtmp2 = (double *) shmalloc(ed->nlstates*sizeof(double));
```

The put operation needs to be prefaced by a global barrier to make sure that the buffer can be accessed, for example:

```
shmem_barrier_all();
if (k < ed->n - 1) {
   neighb = k - ed->nm + 1;
   shmem_double_put_nb(vtmp2, ed->v1, ed->nlstates,
                       ed->from_nbs[neighb], NULL);
}
```

The global reductions are implemented with SHMEM primitives, e.g.,

```
local_b[0]=b;
shmem_double_sum_to_all(shared_b,local_b,1,0,0,
                        ed->nprocs,pWrk,pSync);
ed->beta[iter] = sqrt(fabs(shared_b[0]));
```

As for the MPI-2 implementation, the results (Figs. 5 and 6) indicate a performance degradation. As the global synchronization comes at a high price, we attempt to optimize the algorithm by performing only a local wait for the buffer to be filled. One potential implementation is to allocate space for a sentinel at the end of the message, e.g.,

```
vtmp = (double *)
   shmalloc(ed->m*(ed->nlstates+1)*sizeof(double));
for (l = 0; l < numbuf; ++l)
   vtmp[l*(ed->nlstates+1)+ed->nlstates] = ((double)-1);
```

and set the sentinel in the source buffer, e.g.,

```
ed->v1[ed->nlstates] = ((double) ed->rank); /* sentinel */
for (l = 0; l < ed->m; ++l) {
  offset = l*(ed->nlstates+1); /* Offset into buffer */
  shmem_double_put_nb(&vtmp[offset],ed->v1,
                      ed->nlstates+1,ed->to_nbs[l],NULL);
}
```

the idea being that if the sentinel changes state, the buffer can then be consumed:

```
tag = vtmp[offset+ed->nlstates];
while (tag != (double) ed->from_nbs[k-ed->nm]) { /* spin */
  tag = vtmp[offset+ed->nlstates];
}
for (i = offset, j=0; i < offset+ed->nlstates; ++i, ++j) {
   ed->v2[j] += ed->gamma * vtmp[i];
}
vtmp[l*(ed->nlstates+1)+ed->nlstates]=((double)-1); /*reset*/
```

The performance results of **SHMEM_fast** (Figs. 5 and 6) for small numbers of processes tend to indicate slightly better performance than for MPI two-sided. However, for larger numbers of process the execution fails. The algorithm is built on the assumption that the message blocks arrive in order, and unfortunately this is not always the case with the default DMAPP environment. If the environment variable `SHMEM_ROUTING_MODE` is set to 0 (default: 2), the routine mode is `DMAPP_ROUTING_IN_ORDER` and the execution problem disappears. However, the performance (not shown) becomes poorer than the MPI two-sided version.

C. UPC

Unified Parallel C or UPC [7] offers a partitioned global address space (PGAS) which, conceptually at least, can make the data transfer between processes more elegant and possibly more efficient. The basis for the implementation is the sequential code version, with only minor revisions. For example, the vectors become shared data structures:

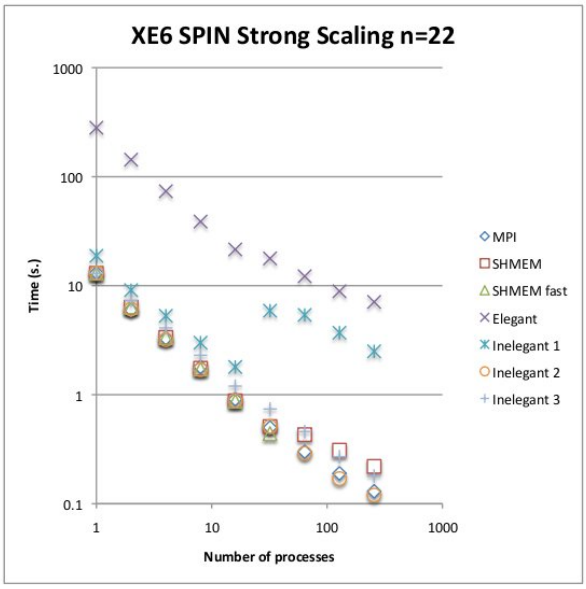

Fig. 5. Cray XE6 strong scaling of $n = 22$ for unthreaded version with various communication paradigms: MPI two-sided reference version, two SHMEM implementations, and four UPC implementations ("elegant" and "inelegant1/2/3").

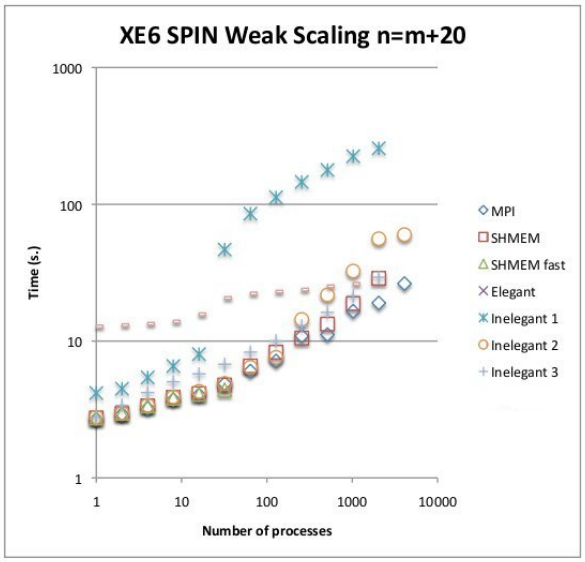

Fig. 6. Weak scaling of $n = m + 20$ for unthreaded version with various communication paradigms: MPI two-sided reference version, two SHMEM implementations, and four UPC implementations ("elegant" and "inelegant1/2/3").

```
struct ed_s {
   shared double *v0, *v1, *v2;  /* vectors */
   shared double *swap;   /* for swapping vectors */
};
```

Next, the main loops are rewritten as `upc_forall` loops, e.g.,

```
upc_forall (s = 0; s < ed->nlstates; ++s; &(ed->v1[s]) ) {
   /* v2 = A * v1, over all threads */
   ed->v2[s] = diag(s, ed->n, ed->j) * ed->v1[s];
   /* offdiagonal part */
   for (k = 0; k < ed->n; ++k) {
     s1 = flip_state(s, k);
     ed->v2[s] += ed->gamma * ed->v1[s1];
   }
}
```

Communication occurs in the off-diagonal part of the calculation, when `s1` does not reside on the local process. Furthermore, a global reduction is required for the vector dot product. This is performed with UPC primitives:

```
shared double shared_a[THREADS];
                :
for (iter = 0; iter < ed->max_iter; ++iter) {
   :
   a = 0.0;  /* Calculate local conjugate term */
   upc_forall (i = 0; i < ed->nlstates; ++i; &(ed->v1[i]) ) {
     a += ed->v1[i] * ed->v2[i];
   }
   shared_a[MYTHREAD] = a;
   upc_all_reduceD( dotprod, shared_a, UPC_ADD, THREADS,
                1, NULL, UPC_IN_ALLSYNC | UPC_OUT_ALLSYNC );
   :
}
```

Unfortunately, the performance (Fig. 5) of this elegant formulation is very poor. Studies have shown [8], [9] that PGAS compilers are not able to aggregate communication in an optimal way to minimize message latencies. Moreover, many PGAS implementations implement communication with a message-passing library, e.g., ARMCI, which then is unlikely to be more efficient than using the message-passing library directly. The Cray UPC compiler, however, uses DMAPP directly, namely the fundamental communication layer of the Gemini network.

In order to properly test the potential of UPC, we programmed three *inelegant* UPC implementations, all of which mimic MPI message-passing version.

1) **Inelegant1:** This version utilizes normal array syntax to move source array to a single shared buffer (i.e., double buffering has been removed):

```
shared[NBLOCK] double vtmp[THREADS*NBLOCK];
  :
for (i = 0; i < NBLOCK; ++i)
   vtmp[i+MYTHREAD*NBLOCK] = ed->v1[i];
  :
upc_barrier(1);
for (i = 0; i < NBLOCK; ++i)
   ed->vv1[i] = vtmp[i+(ed->from_nbs[0]*NBLOCK)];
```

2) **Inelegant2:** Almost identical to the previous coding, in this version an explicit `upc_memput` or `upc_memget` operations in place of the array syntax:

```
shared[NBLOCK] double vtmp[THREADS*NBLOCK];
  :
upc_memput( &vtmp[MYTHREAD*NBLOCK],
            ed->v1, NBLOCK*sizeof(double) );
  :
upc_barrier(1);
upc_memget( ed->vv1, &vtmp[ed->from_nbs[0]*NBLOCK],
            NBLOCK*sizeof(double) );
```

3) **Inelegant3:** In this implementation the double buffering is reimplemented. This is more complicated than in the original version, since shared pointers cannot be swapped in the same way as in the original double-buffered version.

```
shared[NBLOCK] double vtmp1[THREADS*NBLOCK];
shared[NBLOCK] double vtmp2[THREADS*NBLOCK];
```

Loop for MAX_ITER
- **Reduction B** (MPI_Reduce)
- **L3** (local work, normalize v1)
- **Loop over rounds of msgs**
  - **MSG_NB** (MPI_Isend,upc_memput(_nbi))
  - **L4** (work on local matrix, only 1st iteration)
  - **SYNC** (no-op, upc_fence)
  - **L7** (manage msg reception and do remote work)
- **L8** (local work,A norm)
- **Reduction A** (MPI_Reduce)
- **L9** (local work, B norm)

Fig. 7. A simplified framework for single-buffering

```
  :
if ( mode == 0 ) {
   upc_memput( &vtmp2[ed->to_nbs[neighb]*NBLOCK],
                ed->v1, NBLOCK*sizeof(double) );
} else {
   upc_memput( &vtmp1[ed->to_nbs[neighb]*NBLOCK],
               ed->v1, NBLOCK*sizeof(double) );
}
```

In all cases `upc_all_reduceD` is used for the reductions. The results can be seen in Figs. 5 and 6. Interestingly the double-buffering concept does not necessarily provide the best performance. The performance of **Inelegant2** is competitive with MPI two-sided, so this approach was pursued and refined. We subsequently investigate whether the best possible UPC performance can exceed the optimal MPI two-sided performance. To this end, a simplified single-buffer version was derived for subsequent development (see Fig. 7).

This version led to a generalized implementation utilizing multiple buffers, arranged in a round-robin fashion, as depicted in Fig. 8.

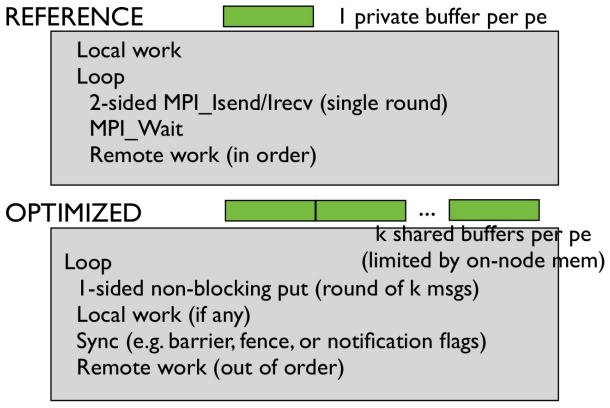

Fig. 8. The single- and multi-buffered implementations of SPIN

We have tested the following benchmark versions of SPIN:

- **Work:** The original MPI two-sided version with double buffering
- **Ref_MPI:** The naive single buffered version as depicted in Fig. 7
- **Opt_MPI:** Multiple round-robin buffers utilizing `MPI_Isend/Irecv` operations
- **Opt_UPC_Fence:** Blocking `upc_memput` with single fence
- **Opt_UPC_Fence_each:** Blocking `upc_memput` with fence for every message
- **Opt_UPC_Fence_nbi:** Cray-specific implicit non-blocking `memput` with a single fence
- **Opt_UPC_Fence_each_nbi:** Cray-specific implicit non-blocking `memput` with a fence for every message

Depending on the size of m (which is equal to the number of messages generated on each PE for a given iteration), there can be considerable benefit to multi-buffering. On the Cray XE6 and the SGI UV1000, multi-buffering becomes worthwhile after roughly $m = 7$, that is, 128 MPI processes.

Performance results for the Cray XE6 and SGI UV1000 are given in Fig. 9 and 10 for $n = 22$ and $n = 28$, respectively. It is clear that the best overall performer is **Opt_UPC_Fence_nbi**, however, it is very sensitive to the values of m and n. Particularly on the UV1000, the reference MPI version (single buffering) performs surprisingly well.

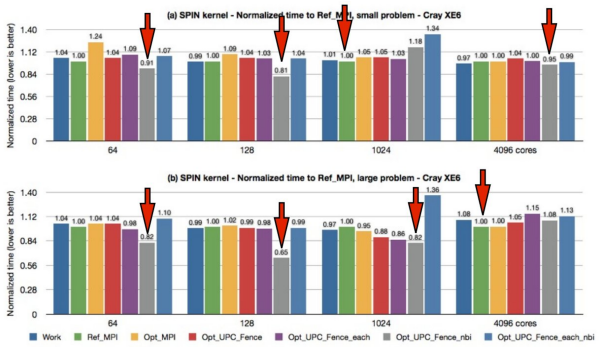

Fig. 9. Comparative run times (s.) for 100 iterations for $n = 22$ (top) and $n = 28$ (bottom). The Cray-specific NBI memput performs well in many cases.

The seemingly erratic performance of the variants can be better understood by timing the individual code sections in Fig. 7. These timings are vastly different in the variants. Figs. 11 and 12 illustrate the component timings on Cray XE6, and Fig. 13 for SGI UV1000. The timings for *MPI_NB* and *SYNC* can be taken as the overall message overhead. This is, effectively, hidden in the multi-buffered version **Opt_MPI** due to the overlapping of communication and computation.

D. Process placement within NUMA node

As mentioned in the introduction, each process in SPIN communicates with m neighbors, a key feature in this algorithm. The communication pattern (Fig. 1) is sparse and irregular. A 24×24 section of the pattern is given in Fig. 14.

If these processes are mapped to the cores of a NUMA node, it is clear that some configurations will better distribute the communication traffic than others. In Fig. 15 the mapping is given which would be used by default if the pattern were mapped to a 24-core Magny-Cours node. This mapping arises from a simple sequential assignment of MPI processes to the

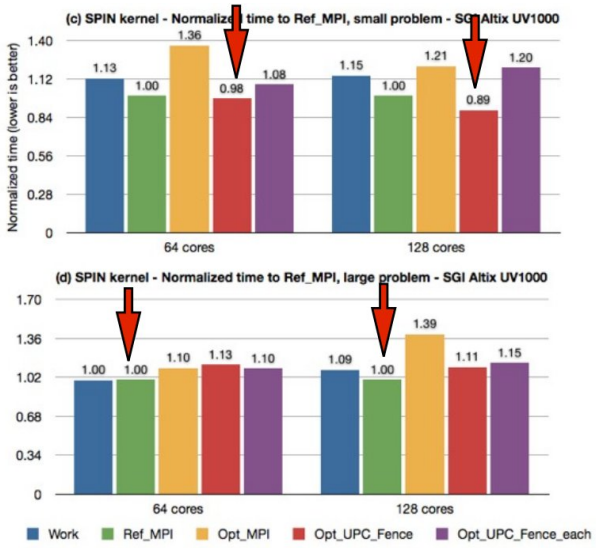

Fig. 10. Comparative run times (s.) for 100 iterations on the SGI UV1000 for $n = 22$ (top) and $n = 28$ (bottom). The simple single-buffered MPI two-sided reference version is highly competitive.

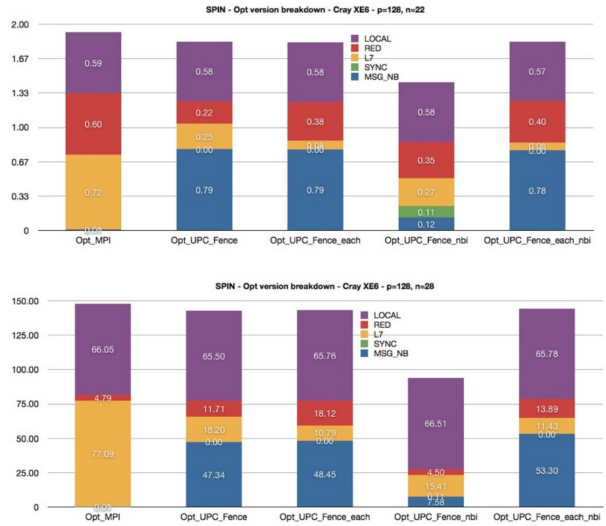

Fig. 11. Timings (s.) of individual code components for $m = 7$ (128 MPI processes) on the Cray XE6 for $n = 22$ (top) and $n = 28$ (bottom).

natural numbering of the cores. The ratio of on-die to off-die messages is 10:10.

If the nodes are distributed to the dies in a round-robin fashion, however, a more localized communication pattern (Fig. 16) can be achieved.

Fig. 17 compares the performance of the reference version with round-robin mapping to the default mapping for the original version (**Work**) and the reference.

IV. Hybrid MPI-OpenMP parallelization of SPIN

All of the SPIN code variants discussed up to this point have been single-threaded distributed memory versions. This code is however, a potential candidate for hybrid parallelization due to the presence of inner loops operating over the vectors `v0, v1, v2`. In the most naive implementation, these loops can be multithreaded, first by applying a first touch, e.g.,

```
#pragma omp parallel for private(i)
for (i = 0; i < ed->nlstates; ++i)
  ed->v1[i] = 1.0;
```

and then multithreading the corresponding vector operation, e.g., the process-local calculation of the matrix-vector multiplication (diagonal and off-diagonal matrix entries):

```
#pragma omp parallel for private(i,s1,k)
for (i = 0; i < ed->nlstates; ++i)
{
  s = loc_index2state(i, ed->nm, ed->rank);
  /* diagonal */
  ed->v2[i] = diag(s, ed->n, ed->j) * ed->v1[i];
  /* offdiagonal part */
  for (k = 0; k < ed->nm; ++k)
  {
    s1 = flip_state(s, k);
    ed->v2[i] += ed->gamma *
```

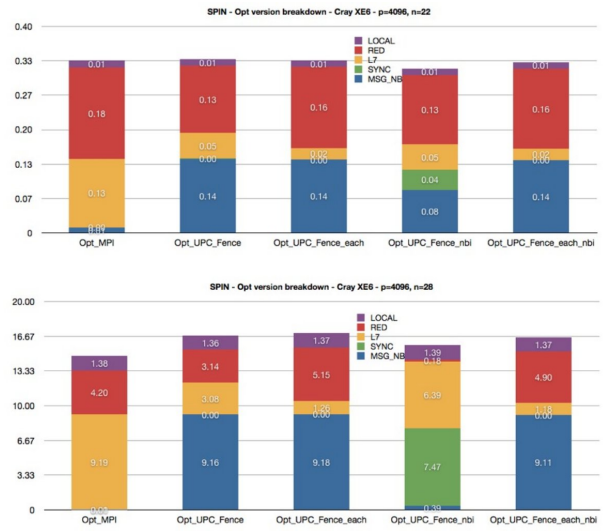

Fig. 12. Timings (s.) of individual code components for $m = 12$ (4096 MPI processes) on the Cray XE6 for $n = 22$ (top) and $n = 28$ (bottom).

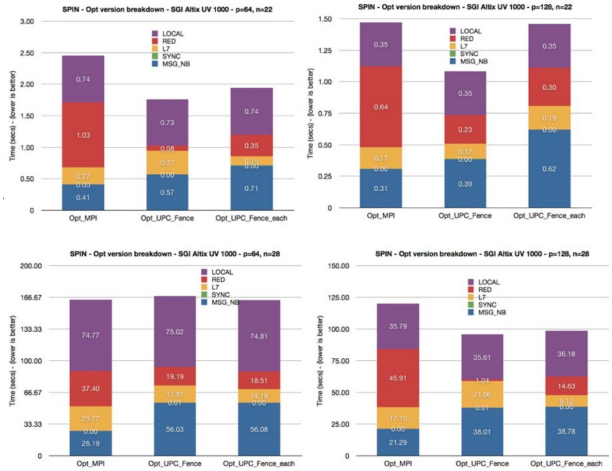

Fig. 13. Timings (s.) of individual code components for $m = 6, 7$ (64 and 128 MPI processes) on the SGI UV1000 for $n = 22$ (top) and $n = 28$ (bottom).

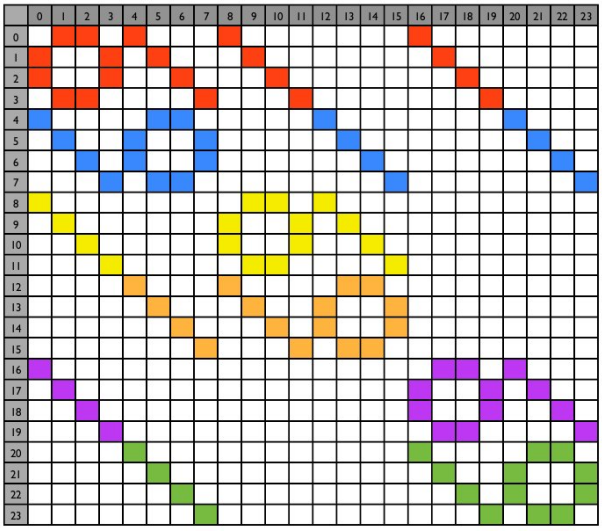

Fig. 14. A 24×24 cross-section of the communication pattern $m = 8$ communication pattern

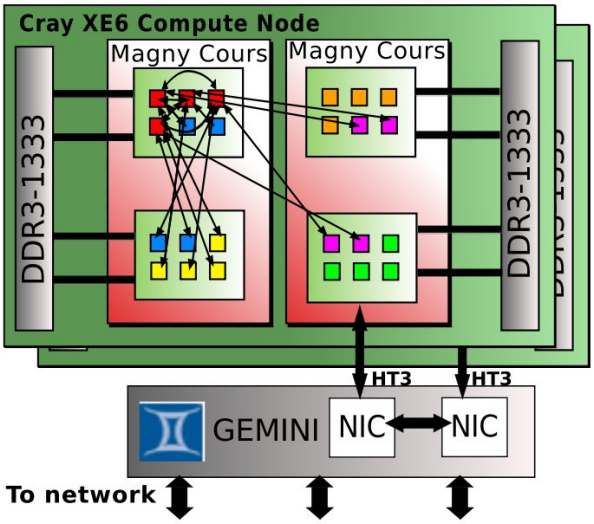

Fig. 15. Default task mapping, based on sequential assignment to the natural numbering of cores

```
            ed->v1[state2loc_index(s1, ed->nlstates)];
        }
    }
```

This naive hybrid parallelization leads in all cases to slower code than the MPI-only in a core-for-core comparison. The central problem is the paucity of work for the needed communication, as shown in Figs. 11 and 12. Increasing the number of MPI processes reduces the message size inversely, while only increasing the number of neighbors ($= m$) instead of 2^m. The MPI implementation itself is therefore quite scalable (Figs. 5 and 6).

A more effective approach seems to be to use the *task* paradigm added in the OMP3 standard [6]. We utilize a work-sharing concept which allows overlapping communication with computation, similar to the technique used in [10]. Given

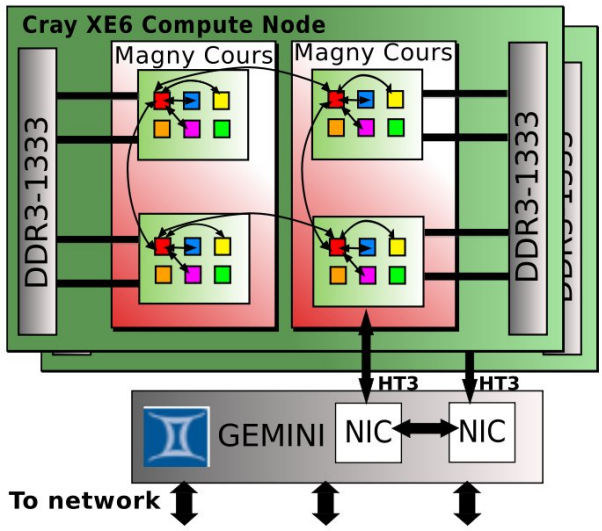

Fig. 16. Task mapping based on a round-robin assignment with a period of 4 (to alternate between the dies)

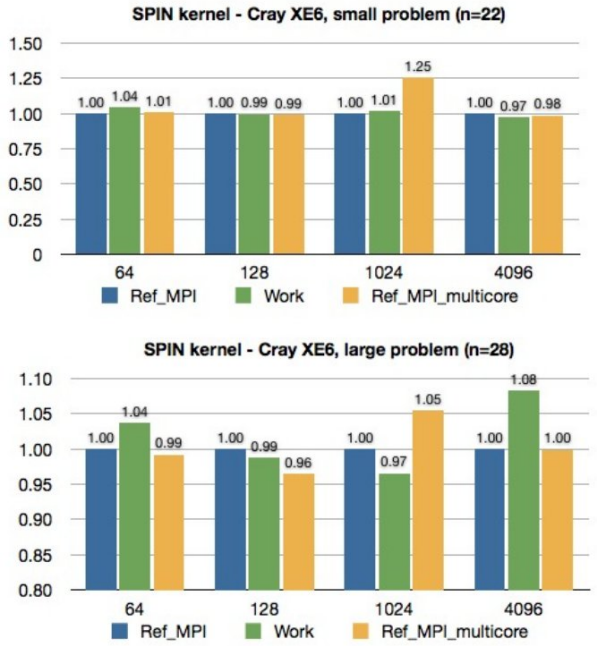

Fig. 17. The round-robin distribution of processes yields a performance improvement in many cases.

t available cores, a master task distributes $t - 1$ threads to awaiting cores, each one assigned to a different segment of the target vector. The master then proceeds to the communication section, in which the vectors are exchanged between neighbors. After the communication has completed, the master waits for all the slave tasks to complete. This approach is, conceptually at least, more efficient than the simple loop multi-threading described previously, in which computation and communication are sequentialized. The code is, however, conceived for UMA architectures, and its performance on NUMA nodes may not be ideal.

```
#pragma omp parallel
{
#pragma omp master
  {
    int j;
    for (j = 0; j < omp_get_num_threads() - 1; ++j)
    {
#pragma omp task firstprivate(j) private (i, k, s, s1)
      {
        int n         = ed->nlstates;
        int num_thrds = omp_get_num_threads() - 1;
        int thrd_id   = j;
        int start     = n*thrd_id/num_thrds;
        int end       = n*(thrd_id + 1)/num_thrds;
        for (i = start; i < end; ++i)
        {
          s = loc_index2state(i, ed->nm, ed->rank);
          /* diagonal part */
          ed->v2[i] = diag(s, ed->n, ed->j) * ed->v1[i];
          /* offdiagonal part */
          for (k = 0; k < ed->nm; ++k)
          {
            s1 = flip_state(s, k);
            ed->v2[i] += ed->gamma *
              ed->v1[state2loc_index(s1, ed->nlstates)];
          }
        }
      }
    }
    //
    // Communications part, e.g., MPI_Isend/Irecv
    //
     :
  }
}
```

Preliminary scalability analysis (Tab. III) on **Rosa** indicated the viability of this approach for the SPIN benchmark for the $n = 28$ test case.

MPI processes	MPI-only (s.)	2 Threads (s.)	4 Threads (s.)
4096	17.4		
2048	28.1	16.6	
1024	48.9	25.1	14.4

TABLE III
SPIN HYBRID MPI/OPENMP TIMININGS (S.) FOR 100 ITERATIONS WITH $n = 28$.

A. SPIN single-node performance comparison

For the single core/socket/node intercomparison (Tab. IV) a non-trivial problem size, $n = 24$ with 100 iterations, was chosen, which is still small enough to provide tractable execution times.

System name	Rivera	Castor	Sandy
Processor	AMD 6274	Intel E5-2680	Intel X5650
Nickname	Interlagos	Westmere	Sandybridge
Cores/Socket	16	6	8
Sockets/Node	2	2	2
Hyperthreading	no	unenabled	yes (2)
Compiler	Open64	Intel	Intel
Core time (s.)	754 (1T)	280 (1T)	227 (1T)
Socket time (s.)	74 (15T)	51 (6T)	29 (16T)
Node time (s.)	38 (31T)	26 (12T)	15 (32T)

TABLE IV
TESTBED TIMINGS (S.) OF 100 ITERATIONS OF SPIN WITH $n = 24$.

These results are illustrated pictorially in Fig. 18.

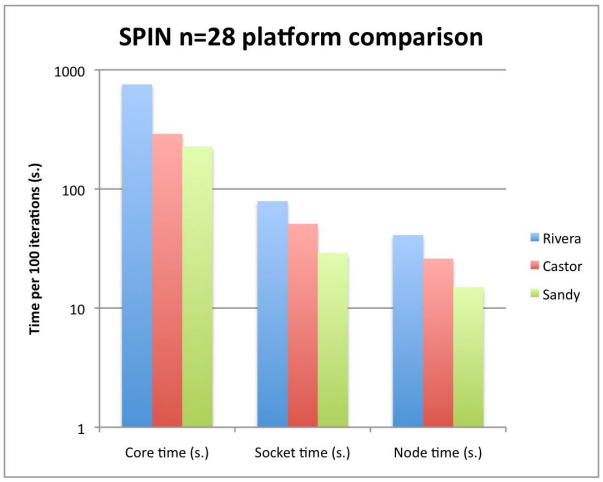

Fig. 18. Comparison of single core, single socket, and single node performance between platforms.

V. HYBRID MPI-OPENMP PARALLELIZATION OF ED

The full ED implementation is conceptually an extension of the SPIN benchmark: it supports multiple lattices and multiple models, where SPIN used one each. However, the code is quite different: ED is programmed in C++ and utilizes templating, data encapsulation and object orientation.

The matrix-vector multiplication is generic, and includes the model type as an object. This allows the selection of virtually any model. Currently a few fermionic and spin models are implemented. Communication is essentially hidden in a `comm` object:

```
// send data
std::vector<send_s> const& se = t.se;
for (unsigned i = 0; i < se.size(); ++i) {
  value_type const* pvec = vspace.slice(vec, se[i].l_job);
  index_type size = bspace.size(se[i].l_job);
  comm.isend(se[i].r_worker, se[i].l_job, pvec, size).wait();
}

// recv data
if (t.r_worker != id && t.do_recv) {
  value_type* pvec = &buf[0];
  index_type size = bspace.size(t.r_worker, t.r_job);
  comm.irecv(t.r_worker, t.r_job, pvec, size).wait();
}
```

which allows future implementation of communication paradigms other than the current MPI two-sided.

The code structure of the matrix-vector multiplication rootine can be depicted as follows

```
// local part
foreach lterm in local_terms
  foreach state in states
    me, nstate = lterm.apply(state)
    v2[state2index(state)] += me * v1[state2index(nstate)]

// nonlocal part
foreach nlterm in nonlocal_terms
  buf = fetch_vector_part_from_node(node(nlterm))
  foreach state in states
    me, nstate = nlterm.apply(state)
  v2[state2index(state)] += me * buf[state2index(nstate)]
```

In the local part, we loop over local terms in the Hamiltonian, i.e., terms that result in states that are local. In nonlocal

part, we loop over non local terms in the Hamiltonian, i.e. terms that result in states that are non-local. In the latter case, we need to fetch vector parts from remote nodes. The loop over states runs over all the local states and the function `state2index` calculates the corresponding (local) index of the vector element. The remote vector part is fetched from the node given by the function `node`. The full algorithm will be described elsewhere [11].

Using the SPIN benchmark as a guideline, multi-threading was implemented in the first cut by parallelizing the internal loops with OpenMP directives, e.g. for the local off-diagonal contributions:

```
// local offdiagonal part
for (unsigned j = 0; j < asubspace.size(); ++j) {
  as_elem const& ae = aspace[asubspace[j]];
  state_type a_state = ae.state;
  value_type const* pvec = vspace.slice(vec, j);
  index_type lastk = bspace.last(j);
  typename bspace_type::biterator it = bspace.begin(j);

#ifdef _OPENMP
#       pragma omp parallel for
#endif
  for (index_type k = 0; k < lastk; ++k) {
    if (it[k].size == 0) continue;
    state_type state = it[k].state | a_state;
    value_type nv =
     matrix.def.b_apply(state, pvec, *this, ae, it[k].size);
    vspace.slice(rvec, j)[k] += nv;
  }
}
```

While this technique was not beneficial in the SPIN benchmark, there is a crucial difference in ED, in that most of the quantum models available there are more computationally intense than in the former. In fact, it became quickly apparent that loop multi-threading for, e.g., a complicated model with multi-spin interactions, which we refer to as a *Fendley* model [12] in the next subsections, yielded much more performant results than the MPI-only version on several platforms.

A. Compiler Optimization Issues

With the change from C to C++ in ED, came an unexpected performance issue. The Cray CCE 8.0.2 compiler, which we would have expected to produce the best code for the Cray XE6/XK6, turned out to generate significantly less performant code than the GNU 4.6.2 compiler. In this section, we try to investigate the Cray C++ compiler issues. Tab. V shows comparative runs of ED on a 4×3 lattice using the GCC and Cray compilers and different pinning options. One MPI process was used. We restricted ourselves to one NUMA domain to avoid any hypertransport interference when running on one socket.

Threads	Pinning (-cc)	Cray	GNU
4	0,1,2,3	1011	667
4	0,2,4,6	1635	592
8	cpu	960	433

TABLE V
4X3 ED TIMINGS (S.) ON A SINGLE NUMA REGION

The Cray compiler's behavior can be explained by the fact that packing threads on the two first module will enable the system to increase their freqency around 3.0+ Ghz. This makes the 4 threads performance on par with the 8 threads configuration, as fewer threads compete with each other for the available memory bandwidth. Using the CPU pinning 0,2,4,6 makes the use of the Interlagos turbo boost impossible (the turbo boost granularity is module-based). However, this behavior is not seen using GNU compiler. In fact, the general behavior is the opposite of the Cray compiler's: using one thread per module is faster than packing for the 4-thread case, and the 8-thread case is much faster. Furthermore, overall performance is much greater than that of the code produced by the Cray compiler. In order to explain this difference between the two compilers, we checked several PAPI events for both executables: PAPI_TOT_CYC, which represents the total number of cycles used to compute the kernel, PAPI_FPU_IDL, which is the total number of cycles when the FPU is idle, and PAPI_L2_TCM, which represents the total number of L2 cache misses. Results are displayed in the tables VI, VII and VIII.

PAPI Event	Cray	GNU
PAPI_TOT_CYC	2'900'762'613'118	1'512'844'977'806
PAPI_FPU_IDL	1'130'511'978'612	242'206'647'131
PAPI_L2_TCM	1'422'885'720	74'482'188

TABLE VI
PAPI EVENTS RESULTS, 4 THREADS, -CC=0,1,2,3

PAPI Event	Cray	GNU
PAPI_TOT_CYC	3'955'167'462'807	1'327'738'681'779
PAPI_FPU_IDL	2'259'595'470'876	219'174'779'204
PAPI_L2_TCM	2'714'833'450	73'238'731

TABLE VII
PAPI EVENTS RESULTS, 4 THREADS, -CC=0,2,4,6

PAPI Event	Cray	GNU
PAPI_TOT_CYC	2'269'187'342'972	775'493'075'990
PAPI_FPU_IDL	1'311'686'430'846	141'553'650'048
PAPI_L2_TCM	821'641'494	80'416'142

TABLE VIII
PAPI EVENTS RESULTS, 8 THREADS, -CC=CPU

Obviously, the Cray compiler version is not properly optimizing: the FPU is idle 30% to 60% of the time, whereas the FPU is idle around 16% of the time using the GNU compiler. Furthermore, the L2 cache misses are much higher with the Cray compiler. An analysis of the generated assembly code revealed that GNU was making use of the full SSE width, unrolling loops and pipelining, while Cray was not, even when used with the most aggressive compilation flags. The difference in the resulting performance was so significant that it was reported to Cray as a compiler bug.[1] In contrast, there

[1]Cray was able to reproduce these results, and is taking steps to further optimize their compiler.

was little difference in the performance of the code generated by GNU 4.6.2 and Intel 12.0.0 on **Rothorn**, although the Intel codes almost always yielded slightly better performance.

B. Hybrid MPI/OMP comparison

For the inter-platform comparison, a scientifically representative problem was chosen: the Fendley model [12] on a 4x4 square lattice with symmetries. This model has complicated multi-spin interactions, and the spins live on the links of the square lattice. There are 32 spins on a 4×4 square lattice. We do use symmetries and the problem size is 272367616. The peculiarity of the model is that the Hamiltonian matrix is less sparse due to multi-spin interactions — there are roughly 2^{16} matrix elements per row. This makes the computation of the matrix elements quite intense. While the problem size is not a breakthrough, a large number of matrix elements makes this problem rather hard. This model should also indicate the algorithm behavior for larger scientifically significant problems, while still having an execution time, which is small enough for significant benchmarking efforts.

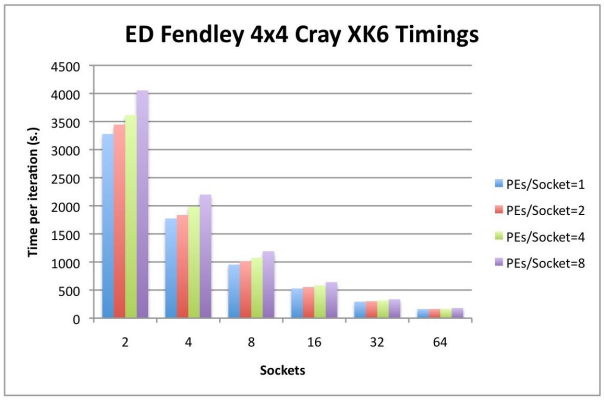

Fig. 19. Time per iteration (s.) of the Fendley 4x4 test case with symmetries on multi-socket configurations of Cray XK6 **Todi**. As expected, the best performance is generally achieved by mapping one MPI process to a NUMA domain, i.e., 2 per socket.

As discussed in the methodology (Section II), we believe a socket-for-socket comparison of platforms to be the fairest. The effort in the single-node benchmarking yielded the optimal thread configurations for socket performance. The check this work, distributed memory runs were performed on 2 – 64 sockets on the **Todi** and 2 – 30 on **Rothorn**, with varying numbers of MPI processes per socket. While this problem size can run on as little as one socket, the execution times for such runs were restrictive.

Fig. 19 illustrates the overall performance for 2 and 4 MPI processes per AMD Interlagos socket. As suspected, the optimal configuration is almost always to assign one process per NUMA domain, i.e., 2 processes per socket. On the SGI UV1000 **Rothorn**, the logical options were to assign one process per socket or one per blade (2 sockets). Fig. 20 clearly shows that the better choice is the to assign one process per blade. This is mildly surprising, since this introduces

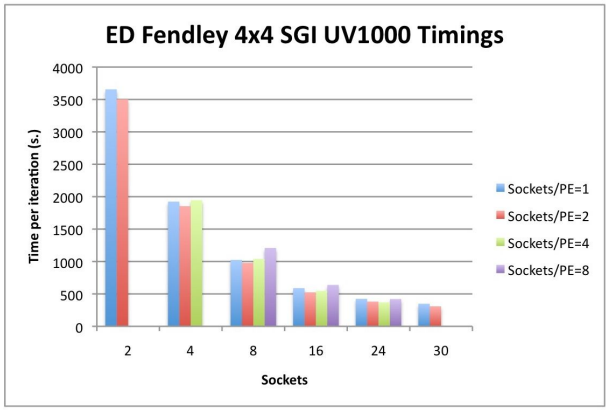

Fig. 20. Time per iteration (s.) of the Fendley 4x4 test case with symmetries on multi-socket configurations of SGI UV1000 **Rothorn**. Interestingly the best performance is achieved by mapping one MPI process to two sockets (or one blade) of the SGI. Assigning a process to more than one blade (introducing two levels of message latencies) yielded considerably poorer performance (not shown).

non-uniform memory access. Assigning a process to multiple blades, however, consistently yielded poorer performance.

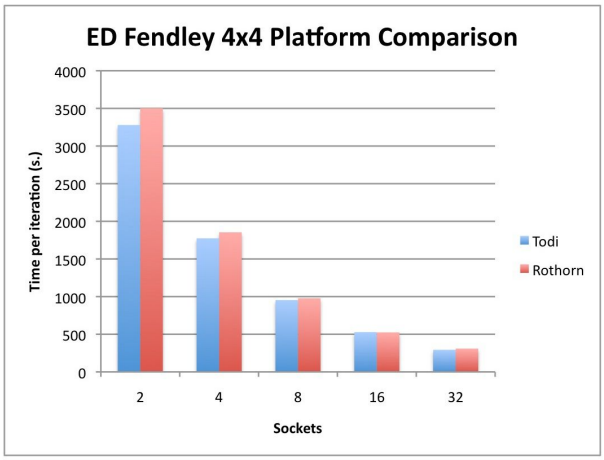

Fig. 21. Time per iteration (s.) of the Fendley 4x4 test case with symmetries compared on Cray XK6 **Todi** and SGI UV1000 **Rothorn**. The timing for "32" sockets on **Rothorn** is actually only 30 sockets, to avoid contention with operating system processes.

The best configurations for the respective sockets are then compared in Fig. 21 for 2 – 30 sockets. The Cray XK6 yields the best absolute performance in this socket-to-socket comparison. However, the scaling is arguably better on the SGI UV1000, which is surprising since the its NUMAlink interconnect is from a previous generation. A further discussion of these results from several perspectives is offered in Section VI.

C. ED Scalability

Also of interest is the scalability of the algorithm to very large machine configurations. Naive parallelizations of exact diagonalization have a performance bottleneck because it is difficult to define a limited neighborhood of interaction in the

calculation of outcomes of applying the Hamiltonian operator. The ED code elegantly removes this limitation, suggesting that it will scale well. In order to get performances for a scientifically meaningful test case, the Fendley model was chosen [12] *without symmetries*. The problem size is 2^{32}. The memory requirements are such that this problem requires approximately 256 cores or more to run. The timing results are shown in Tab. IX.

MPI Processes	Total Cores	Time / iteration (s.)
100	3200	218.0
512	16384	65.4
1024	32768	35.0
1476	47232	25.4

TABLE IX
TIME PER ITERATION (S.) FOR THE FENDLEY 2D MODEL WITHOUT SYMMETRIES ON A 4×4 LATTICE.

These results are visualized in Fig. 22, indicating the code indeed scales to the full extent of the CSCS platform.

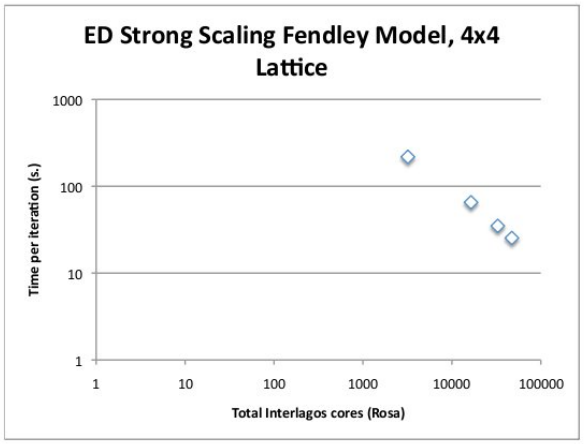

Fig. 22. The time per iteration for the multi-spin model [12] on a 4x4 square lattice as a function of the number of AMD Interlagos cores on the Cray XE6 at CSCS.

VI. CONCLUSIONS

In this paper we have evaluated a variety of different parallel programming paradigms applied to the exact diagonalization technique, which ultimately reduces to the Lanczos tridiagonalization using matrix-vector operators employing both integer and floating-point operations. These paradigms were: MPI two-sided, MPI one-sided, SHMEM and UPC for inter-process communication, and OpenMP for multi-threading. Variants of these were tested comparatively on a number of different architectures, including the Cray XE6, SGI UV1000 and testbeds with Intel Westmere and Sandybridge.

For message-passing, it seems clear that MPI two-sided communication is probably the best overall approach to achieve parallel performance with the XE6's Gemini interconnection network. MPI one-sided communication could not achieve comparable performance, and SHMEM may achieve similar performance, if global barrier synchronization can be avoided. UPC offers some hope for higher performance on the XE6, however the performance increase is relatively small, and depends on the test case. It is unclear whether the improvement warrants a paradigm shift to UPC.

Several approaches for multi-threading the code were investigated. Simple loop multitasking with OpenMP did not offer performance improvement for the simplest test case (the SPIN model), but a paradigm utilizing MPI + OpenMP 3.1 tasks yielded better performance than the MPI-only version. In the more general case of ED with a more computationally intensive quantum model [12], MPI plus simple OpenMP loop multitasking usually scaled much better that MPI-only.

The performance of SPIN and ED has been compared between several different architectures, including Cray XE6, SGI UV1000, along with testbeds with Intel Sandybridge and Interlagos nodes. While such comparisons are hard to make objectively, the best single socket performance is clearly achieved by the Intel Sandybridge. Not only that, Intel Sandybridge provided the best out-of-the-box performance, with only a few viable configurations to consider for optimal performance. The worst processor in this sense was the AMD Interlagos, which required considerable effort to find the correct process/thread-to-core mappings to achieve the best performance. The SPIN/ED codes would be good candidates for auto-tuning to help determine the optimal configurations.

When distributed memory parallelism is considered, the story becomes less definitive. The hybrid MPI-OpenMP ED code yields comparable performance and scaling on the Cray XE6/XK6 and the SGI UV1000. We did not consider the price of hardware to the performance or the energy-to-solution. Those metrics might yield different conclusions.

A significant handicap in this work was our lack of access to a Intel Sandybridge cluster with a state-of-the-art commodity interconnection network, such as FDR. We hope to obtain access to such as system in the near future, and plan to extend the comparison. We also hope to have results from the IBM BG/Q in a subsequent publication.

Since the efficacy for OMP 3.1 task parallelism in the SPIN benchmark to overlap computation and communication was clearly illustrated, we look forward to introducing this paradigm into the full ED code as well.

While introducing OpenMP directives into ED was illustrated to be easy, the introduction of OpenACC directives for GPUs was not. Our attempts at a mirror image GPU implementation met with immediate difficulties. These were related to the data encapsulation employed in the ED C++ code, which refers to separation of data allocation/transfers and calculation, and to the lack of possibility to transfer data that are accessed via a pointer (even via the implicit `this` pointer) to GPUs. OpenACC, on the other hand, requires that the data to be transferred to GPUs are clearly visible to the calculation part of the code (to the code that will be executed on GPUs). We have communicated these issues to the OpenACC community, and hope to work together with vendors to achieve a GPU-capable implementation. One possibility

would be to work with the SPIN benchmark, which is written in C and whose data structures are simple arrays. However, our preference is to work with the full ED code exclusively in the future.

ACKNOWLEDGMENT

The authors would like to thank the High Performance and High Productivity Computing (`www.hp2c.ch`) Initiative for funding this project.

REFERENCES

[1] A. Weisse and H. Fehske in *Computational Many-Particle Physics* Lecture Notes in Physics, Vol. 739, Springer Verlag, 2008.

[2] R. Moessner and S. L. Sondhi, *Ising models of quantum frustration* Phys. Rev. B 63, 224401 (2001).

[3] *The OpenACC Application Programming Interface, Version 1.0.0* `http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf`, 2011.

[4] J. Cullum and R. A. Willoughby, *Computing eigenvalues of very large symmetric matrices–an implementation of a Lanczos algorithm with no reorthogonalization* J. Computat. Phys. 44, 329 (1981).

[5] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, third edition, 1996.

[6] *OpenMP Application Program Interface Version 3.1* `http://www.openmp.org/mp-documents/OpenMP3.1.pdf`, 2011.

[7] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, Center for Computing Sciences, Bowie, MD, May 1999.

[8] C. Maynard. Comparing UPC and One-sided MPI: A distributed hash table for GAP `http://pgas11.rice.edu/papers/Maynard-Distributed-Hash-Table-PGAS11.pdf`, 2011.

[9] W. Chen, C. Iancu and K. Yelick. Communication Optimizations for Fine-grained UPC Applications Lawrence Berkeley National Lab Tech Report LBNL-58382, 2005.

[10] A. Koniges, R. Preissl, J. Kim, D. Eder, A. Fisher, N. Masters, V. Mlaker, S. Ethier, W. Wang, M. Head-Gordon, and N. Wichmann *Application Acceleration on Current and Future Cray Platforms* CUG 2010 Proceedings, Edinburgh `http://cug.org`

[11] S. V. Isakov, M. Troyer, in preparation.

[12] P. Fendley, S. V. Isakov, M. Troyer, in preparation.