Towards a hybrid multi-core implementation of MAQUIS Exact Diagonalization (ED)

Sergei Isakov, Will Sawyer, Adrian Tineo, Gilles Fourestey, Neil Stringfellow, Matthias Troyer
Overarching goals of our group’s work
Overarching goals of our group’s work

Use *scientifically relevant* mini-apps to:
Overarching goals of our group’s work

Use *scientifically relevant* mini-apps to:

- Evaluate emerging architectures
  - AMD Interlagos
  - Intel Sandybridge
  - IBM BG/Q, GPUs, if possible
Overarching goals of our group’s work

Use *scientifically relevant* mini-apps to:

- Evaluate emerging architectures
  - AMD Interlagos
  - Intel Sandybridge
  - IBM BG/Q, GPUs, if possible
- Evaluate programming paradigms
  - MPI + OpenMP hybrid programming
  - MPI-2 one-sided communication
  - SHMEM
  - UPC (as implemented in Cray compiler)
  - OpenACC, if possible
Overarching goals of our group’s work

Use *scientifically relevant* mini-apps to:

- Evaluate emerging architectures
  - AMD Interlagos
  - Intel Sandybridge
  - IBM BG/Q, GPUs, if possible
- Evaluate programming paradigms
  - MPI + OpenMP hybrid programming
  - MPI-2 one-sided communication
  - SHMEM
  - UPC (as implemented in Cray compiler)
  - OpenACC, if possible
- Compare performance across platforms
  - out-of-the-box performance
  - evaluate optimization effort
  - socket-for-socket, node-for-node comparisons
## CSCS Testbed Platforms

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

<table>
<thead>
<tr>
<th>System name</th>
<th>Rosa</th>
<th>Todi</th>
<th>Rothorn</th>
<th>Grotius</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product name</td>
<td>Cray XE6</td>
<td>Cray XT6</td>
<td>SGI UV1000</td>
<td>IBM BG/Q</td>
</tr>
<tr>
<td>Interconnection</td>
<td>Gemini</td>
<td>Gemini</td>
<td>NUMAlink</td>
<td>5D Torus</td>
</tr>
<tr>
<td>Processor</td>
<td>AMD 6272</td>
<td>AMD 6272</td>
<td>Intel E7-8837</td>
<td>IBM A2</td>
</tr>
<tr>
<td>Proc. nickname</td>
<td>Interlagos</td>
<td>Interlagos</td>
<td>Westmere</td>
<td></td>
</tr>
<tr>
<td>Clock (GHz)</td>
<td>2.1</td>
<td>2.1</td>
<td>2.66</td>
<td>1.60</td>
</tr>
<tr>
<td>Sockets/Node</td>
<td>2</td>
<td>1</td>
<td>32</td>
<td>1</td>
</tr>
<tr>
<td>Cores/Socket</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>NUMA/Socket</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>DP GFlops/Socket</td>
<td>134.4</td>
<td>134.4</td>
<td>85.1</td>
<td>204.8</td>
</tr>
<tr>
<td>Memory/Socket</td>
<td>16 GB</td>
<td>16 GB</td>
<td>64 GB</td>
<td>16 GB</td>
</tr>
<tr>
<td>DDR3 mem. speed</td>
<td>1600</td>
<td>1600</td>
<td>1333</td>
<td>1333</td>
</tr>
<tr>
<td>L1 cache (excl.)</td>
<td>16KB</td>
<td>16KB</td>
<td>32KB</td>
<td>32KB</td>
</tr>
<tr>
<td>L2 cache/# cores</td>
<td>2MB/2</td>
<td>2MB/2</td>
<td>256KB/1</td>
<td>32MB/16</td>
</tr>
<tr>
<td>L3 cache/# cores</td>
<td>8MB/8</td>
<td>8MB/8</td>
<td>24MB/8</td>
<td>N/A</td>
</tr>
<tr>
<td>Hyperthreading?</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes (4)</td>
</tr>
<tr>
<td>TPA/Socket (W)</td>
<td>115</td>
<td>115</td>
<td>130</td>
<td>60</td>
</tr>
</tbody>
</table>
Fundamental ED problem

\[ H = J \sum S_i^z S_j^z + \Gamma \sum S_i^x \]
Fundamental ED problem

\[ H = J \sum S_i^z S_j^z + \Gamma \sum S_i^x \]
Fundamental ED problem

\[ H = J \sum S_i^z S_j^z + \Gamma \sum S_i^x \]
Fundamental ED problem

\[ H = J \sum S_i^z S_j^z + \Gamma \sum S_i^x \]
Fundamental ED problem

\[ H = J \sum S_i^z S_j^z + \Gamma \sum S_i^x \]
Fundamental ED problem

\[ H = J \sum S_i^z S_j^z + \Gamma \sum S_i^x \]
**Fundamental ED problem**

\[
H = J \sum_i S_i^z S_j^z + \Gamma \sum_i S_i^x
\]
Fundamental ED problem

\[ H = J \sum S_i^z S_j^z + \Gamma \sum S_i^x \]

- Any lattice with \( n \) sites, \( 2^n \) states
Fundamental ED problem

\[ H = J \sum S_i^Z S_j^Z + \Gamma \sum S_i^x \]

- Any lattice with \( n \) sites, \( 2^n \) states
- Lanczos eigensolver

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: \( w_j \leftarrow H v_j - \beta_j v_{j-1} \)
6: \( \alpha_j \leftarrow (w_j, v_j) \)
7: \( \beta_{j+1} \leftarrow \| w_j \| \)
8: \( v_{j+1} \leftarrow w_j / \beta_{j+1} \)
9: end for
Fundamental ED problem

\[ H = J \sum S_i^z S_j^z + \Gamma \sum S_i^x \]

- Any lattice with \( n \) sites, \( 2^n \) states
- Lanczos eigensolver
- Large, sparse symmetric mat-vec

**Algorithm 1** Lanczos iteration

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

\[ H = J \sum S_i^z S_j^z + \Gamma \sum S_i^x \]

- Any lattice with \( n \) sites, \( 2^n \) states
- Lanczos eigensolver
- Large, sparse symmetric mat-vec
- Operator has integer operations

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: \( w_j \leftarrow Hv_j - \beta_j v_{j-1} \)
6: \( \alpha_j \leftarrow (w_j, v_j) \)
7: \( \beta_{j+1} \leftarrow \|w_j\| \)
8: \( v_{j+1} \leftarrow w_j / \beta_{j+1} \)
9: end for
Fundamental ED problem

\[ H = J \sum S_i^z S_j^z + \Gamma \sum S_i^x \]

- Any lattice with \( n \) sites, \( 2^n \) states
- Lanczos eigensolver
- Large, sparse symmetric mat-vec
- Operator has integer operations
- Very irregular sparsity, but

**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: \( w_j \leftarrow Hv_j - \beta_j v_{j-1} \)
6: \( \alpha_j \leftarrow (w_j, v_j) \)
7: \( \beta_{j+1} \leftarrow \|w_j\| \)
8: \( v_{j+1} \leftarrow w_j / \beta_{j+1} \)
9: end for
Fundamental ED problem

\[ H = J \sum S_i^z S_j^z + \Gamma \sum S_i^x \]

- Any lattice with \( n \) sites, \( 2^n \) states
- Lanczos eigensolver
- Large, sparse symmetric mat-vec
- Operator has integer operations
- Very irregular sparsity, but
- Limited number of process neighbors (new to this work)
Fundamental ED problem

\[ H = J \sum S_i^z S_j^z + \Gamma \sum S_i^x \]

- Any lattice with \( n \) sites, \( 2^n \) states
- Lanczos eigensolver
- Large, sparse symmetric mat-vec
- Operator has integer operations
- Very irregular sparsity, but
- Limited number of process neighbors (*new to this work*)
- Symmetries considered in some models: smaller complexity at cost of more communication
Benchmark code: simplest “SPIN” model
Benchmark code: simplest “SPIN” model

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)
Benchmark code: simplest “SPIN” model

Loop for MAX_ITER

Reduction B (MPI_Reduce)
L3 (local work, normalize v1)
Loop over rounds of msgs
  MSG_NB (MPI_Isend,upc_memput(_nbj))
  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)

Loops:
• L3: Initialize array
• L4: Local mat-vec
• L6/7: Off process mat-vec
• L8: Alpha calculation
• L9: Beta calculation
Benchmark code: simplest “SPIN” model

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)

Loops:
• L3: Initialize array
• L4: Local mat-vec
• L6/7: Off process mat-vec
• L8: Alpha calculation
• L9: Beta calculation
SPIN single core/socket/node comparisons

- Loop-based OMP directives: performance worse than MPI-only
- Task-based OpenMP/MPI implementation by Fourestey/Stringfellow
**SPIN single core/socket/node comparisons**

- Loop-based OMP directives: performance worse than MPI-only
- Task-based OpenMP/MPI implementation by Fourestey/Stringfellow

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

Double buffering
Multi-buffering concept

Double buffering

- Send data to other processor
- Send data to other processor
- Send data to other processor

Processor x in step i
- \( v_1 \) (read-only data, L6-7)
- \( v_2 \) (accumulate results)
- \( v_1 \) (buffer 1)
- \( v_2 \) (buffer 2)

Receive data from other processor

Processor x in step i+1
- \( v_1 \) (read-only data, L6-7)
- \( v_2 \) (accumulate results)
- \( v_2[i] = g^i v_1[i] \)
- \( v_1 \) (buffer 1)
- \( v_2 \) (buffer 2)

Receive data from other processor

Processor x in step i+2
- \( v_1 \) (read-only data, L6-7)
- \( v_2 \) (accumulate results)
- \( v_2[i] = g^i v_2[i] \)
- \( v_1 \) (buffer 1)
- \( v_2 \) (buffer 2)

Receive data from other processor
Multi-buffering concept

Double buffering

Multi-buffering
Multi-buffering concept

Double buffering

Multi-buffering

REFERENCE
1 private buffer per pe

Local work
Loop
2-sided MPI_Isend/Irecv (single round)
MPI_Wait
Remote work (in order)

OPTIMIZED
k shared buffers per pe
(limited by on-node mem)

Loop
1-sided non-blocking put (round of k msgs)
Local work (if any)
Sync (e.g. barrier, fence, or notification flags)
Remote work (out of order)
UPC “Elegant” Implementation
UPC “Elegant” Implementation

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

for (iter = 0; iter < ed->max_iter; ++iter) {
    upc_barrier(0);
    /* matrix vector multiplication */
    upc_forall (s = 0; s < ed->nlstates; ++s; &(ed->v1[s]) ) {
        /* diagonal part */
        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];
        }
    }
    /* Calculate alpha */
    /* Calculate beta */
}
```
Inelegant UPC versions
Inelegant UPC versions

Inelegant 1

```c
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)];
upc_barrier(2);
```

Inelegant 2

```c
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) );
upc_barrier(2);
```
UPC *Inelegant3*: use double buffers and upc_put
UPC \textit{Inelegant3}: use double buffers and \texttt{upc\_put}

\begin{verbatim}
shared[NBLOCK] double vtmp1[THREADS*NBLOCK];
shared[NBLOCK] double vtmp2[THREADS*NBLOCK];
:
upc_memput( &vtmp1[ed->to_nbs[0]*NBLOCK], ed->v1, NBLOCK*sizeof(double) );
upc_barrier(1);
:
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) );
}
:
if ( mode == 0 ) {
  for (i = 0; i < ed->nlstates; ++i) { ed->v2[i] += ed->gamma * vtmp1[i+MYTHREAD*NBLOCK]; }
  mode = 1;
} else {
  for (i = 0; i < ed->nlstates; ++i) { ed->v2[i] += ed->gamma * vtmp2[i+MYTHREAD*NBLOCK]; }
  mode = 0;
}
upc_barrier(2);
\end{verbatim}
Other message passing paradigms

**MPI-2: One-sided PUT**

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

MPI_Win_fence( 0, win1);

**SHMEM: non-blocking PUT**

```c
vtmp1 = (double *) shmalloc(ed->nlstates*sizeof(double));
:
    shmem_barrier_all();
    shmem_double_put_nb(vtmp1, ed->v1, ed->nlstates, ed->from_nbs[neighb], NULL);
```

**SHMEM “fast”: non-blocking PUT, local wait only**

```c
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);
}
:
    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[1*(ed->nlstates+1)+ed->nlstates]=((double)-1); /*reset*/
```
SPIN strong scaling: Cray XE6, n=22,24; 10 iter.
SPIN weak scaling: Cray XE6/Gemini, 10 iterations
Can UPC perform better than MPI two-sided?
Can UPC perform better than MPI two-sided?

- **Work**: original MPI two-sided version with double buffering
Can UPC perform better than MPI two-sided?

- **Work**: original MPI two-sided version with double buffering
- **Ref_MPI**: naive single buffered version
Can UPC perform better than MPI two-sided?

- **Work**: original MPI two-sided version with double buffering
- **Ref_MPI**: naive single buffered version
- **Opt_MPI**: multiple round-robin buffers utilizing MPI_Isend/Irecv
Can UPC perform better than MPI two-sided?

- **Work**: original MPI two-sided version with double buffering
- **Ref_MPI**: naive single buffered version
- **Opt_MPI**: multiple round-robin buffers utilizing MPI_Isend/Irecv
- **Opt_UPC_Fence**: blocking upc_memput with single fence
Can UPC perform better than MPI two-sided?

- **Work**: original MPI two-sided version with double buffering
- **Ref_MPI**: naive single buffered version
- **Opt_MPI**: multiple round-robin buffers utilizing MPI_Isend/Irecv
- **Opt_UPC_Fence**: blocking upc_memput with single fence
- **Opt_UPC_Fence_each**: blocking upc_memput with fence for each message
Can UPC perform better than MPI two-sided?

- **Work**: original MPI two-sided version with double buffering
- **Ref_MPI**: naive single buffered version
- **Opt_MPI**: multiple round-robin buffers utilizing MPI_Isend/Irecv
- **Opt_UPC_Fence**: blocking upc_memput with single fence
- **Opt_UPC_Fence_each**: blocking upc_memput with fence for each message
- **Opt_UPC_Fence_nbi**: Cray-specific implicit non-blocking memput with a single fence
Can UPC perform better than MPI two-sided?

- **Work**: original MPI two-sided version with double buffering
- **Ref_MPI**: naive single buffered version
- **Opt_MPI**: multiple round-robin buffers utilizing MPI_Isend/Irecv
- **Opt_UPC_Fence**: blocking upc_memput with single fence
- **Opt_UPC_Fence_each**: blocking upc_memput with fence for each 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 fence for each message
Optimized SPIN normed performance: Cray XE6
Optimized SPIN normed performance: Cray XE6
Optimized SPIN normed performance: SGI UV1000
Optimized SPIN normed performance: SGI UV1000

- Shared-memory, extensible to 256 sockets
- Fat node NUMAlink interconnect
Optimized SPIN normed performance: SGI UV1000

- Shared-memory, extensible to 256 sockets
- Fat node NUMAlink interconnect
Optimized SPIN normed performance: SGI UV1000

- Shared-memory, extensible to 256 sockets
- Fat node NUMAlink interconnect
Optimized SPIN normed performance: SGI UV1000

- Shared-memory, extensible to 256 sockets
- Fat node NUMAlink interconnect
Optimizing task placement (XE6, Magny Cours)
Optimizing task placement (XE6, Magny Cours)
Optimizing task placement (XE6, Magny Cours)
Optimizing task placement (XE6, Magny Cours)
Optimizing task placement (XE6, Magny Cours)
SPIN experiences
SPIN experiences

- SPIN is not computationally intensive
SPIN experiences

- SPIN is not computationally intensive
- Contain integer operations, e.g., bit shifting
SPIN experiences

- SPIN is not computationally intensive
- Contain integer operations, e.g., bit shifting
- Fundamentally a benchmark of the interconnection network
SPIN experiences

- SPIN is not computationally intensive
- Contain integer operations, e.g., bit shifting
- Fundamentally a benchmark of the interconnection network
- MPI two-sided performing better than (nearly) all other communication paradigms, including SHMEM, MPI-2
SPIN experiences

- SPIN is not computationally intensive
- Contain integer operations, e.g., bit shifting
- Fundamentally a benchmark of the interconnection network
- MPI two-sided performing better than (nearly) all other communication paradigms, including SHMEM, MPI-2
- Work by A. Tineo showed that UPC+optimizations can attain better performance in some cases
SPIN experiences

- SPIN is not computationally intensive
- Contain integer operations, e.g., bit shifting
- Fundamentally a benchmark of the interconnection network
- MPI two-sided performing better than (nearly) all other communication paradigms, including SHMEM, MPI-2
- Work by A. Tineo showed that UPC+optimizations can attain better performance in some cases
- Simplistic OpenMP/MPI hybrid performed not better than MPI
SPIN experiences

- SPIN is not computationally intensive
- Contain integer operations, e.g., bit shifting
- Fundamentally a benchmark of the interconnection network
- MPI two-sided performing better than (nearly) all other communication paradigms, including SHMEM, MPI-2
- Work by A. Tineo showed that UPC+optimizations can attain better performance in some cases
- Simplistic OpenMP/MPI hybrid performed not better than MPI
- Task-based OpenMP/MPI implementation by Fourestey/Stringfellow did show slightly better performance (n=28 test case)
SPIN experiences

- SPIN is not computationally intensive
- Contain integer operations, e.g., bit shifting
- Fundamentally a benchmark of the interconnection network
- MPI two-sided performing better than (nearly) all other communication paradigms, including SHMEM, MPI-2
- Work by A. Tineo showed that UPC+optimizations can attain better performance in some cases
- Simplistic OpenMP/MPI hybrid performed not better than MPI
- Task-based OpenMP/MPI implementation by Fourestey/Stringfellow did show slightly better performance (n=28 test case)

<table>
<thead>
<tr>
<th>MPI Processes</th>
<th>MPI-only (s.)</th>
<th>2 Threads (s.)</th>
<th>4 Threads (s.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096</td>
<td>17.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2048</td>
<td>28.1</td>
<td>16.6</td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>48.9</td>
<td>25.1</td>
<td>14.4</td>
</tr>
</tbody>
</table>
Exact Diagonalization: HP2C Implementation
Exact Diagonalization: HP2C Implementation

Full ED application implemented within the High Performance and High Productivity Computing Initiative (www.hp2c.ch)
Exact Diagonalization: HP2C Implementation

Full ED application implemented within the High Performance and High Productivity Computing Initiative (www.hp2c.ch)

- Symmetries (reduces complexity at cost of more communication)
Exact Diagonalization: HP2C Implementation

Full ED application implemented within the High Performance and High Productivity Computing Initiative (www.hp2c.ch)

- Symmetries (reduces complexity at cost of more communication)
- Supports multiple one- and two-dimensional lattices
Exact Diagonalization: HP2C Implementation

Full ED application implemented within the High Performance and High Productivity Computing Initiative (www.hp2c.ch)

- Symmetries (reduces complexity at cost of more communication)
- Supports multiple one- and two-dimensional lattices
- Multiple quantum models
Exact Diagonalization: HP2C Implementation

Full ED application implemented within the High Performance and High Productivity Computing Initiative (www.hp2c.ch)
- Symmetries (reduces complexity at cost of more communication)
- Supports multiple one- and two-dimensional lattices
- Multiple quantum models
  - Heisenberg
Exact Diagonalization: HP2C Implementation

Full ED application implemented within the High Performance and High Productivity Computing Initiative (www.hp2c.ch)
- Symmetries (reduces complexity at cost of more communication)
- Supports multiple one- and two-dimensional lattices
- Multiple quantum models
  - Heisenberg
  - Fendley (computationally intensive, unlike SPIN benchmark)
Exact Diagonalization: HP2C Implementation

Full ED application implemented within the High Performance and High Productivity Computing Initiative (www.hp2c.ch)

- Symmetries (reduces complexity at cost of more communication)
- Supports multiple one- and two-dimensional lattices
- Multiple quantum models
  - Heisenberg
  - Fendley (computationally intensive, unlike SPIN benchmark)
  - FQHE (computationally intensive)
Exact Diagonalization: HP2C Implementation

Full ED application implemented within the High Performance and High Productivity Computing Initiative (www.hp2c.ch)

- Symmetries (reduces complexity at cost of more communication)
- Supports multiple one- and two-dimensional lattices
- Multiple quantum models
  - Heisenberg
  - Fendley (computationally intensive, unlike SPIN benchmark)
  - FQHE (computationally intensive)

- OMP/MPI Implementation
  (simple loop-based directives)
Exact Diagonalization: HP2C Implementation

Full ED application implemented within the High Performance and High Productivity Computing Initiative (www.hp2c.ch)

- Symmetries (reduces complexity at cost of more communication)
- Supports multiple one- and two-dimensional lattices
- Multiple quantum models
  - Heisenberg
  - Fendley (computationally intensive, unlike SPIN benchmark)
  - FQHE (computationally intensive)

- OMP/MPI Implementation (simple loop-based directives)

```c
// local offdiagonal part
for (unsigned j = 0; j < asubspace.size(); ++j) {
    index_type lastk = bspace.last(j);
    typename bspace_type::biterator it = bspace.begin(j);
    #pragma omp parallel for
    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;
    }
}
```
Cray and PGI C++ compiler optimizations issues
Cray and PGI C++ compiler optimizations issues

On Cray XE6/XK6 expect CCE to generate adequate code
Cray and PGI C++ compiler optimizations issues

On Cray XE6/XK6 expect CCE to generate adequate code

- CCE 8.0.2 vs. GNU 4.6.2 performance comparison
Cray and PGI C++ compiler optimizations issues

On Cray XE6/XK6 expect CCE to generate adequate code

- CCE 8.0.2 vs. GNU 4.6.2 performance comparison
- Various thread-core pinnings
Cray and PGI C++ compiler optimizations issues

On Cray XE6/XK6 expect CCE to generate adequate code
- CCE 8.0.2 vs. GNU 4.6.2 performance comparison
- Various thread-core pinnings

<table>
<thead>
<tr>
<th>Threads</th>
<th>Pinning (-cc)</th>
<th>Cray</th>
<th>GNU</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0,1,2,3</td>
<td>1011</td>
<td>667</td>
</tr>
<tr>
<td>4</td>
<td>0,2,4,6</td>
<td>1635</td>
<td>592</td>
</tr>
<tr>
<td>8</td>
<td>cpu</td>
<td>960</td>
<td>433</td>
</tr>
</tbody>
</table>
Cray and PGI C++ compiler optimizations issues

On Cray XE6/XK6 expect CCE to generate adequate code
- CCE 8.0.2 vs. GNU 4.6.2 performance comparison
- Various thread-core pinnings

<table>
<thead>
<tr>
<th>Threads</th>
<th>Pinning (-cc)</th>
<th>Cray</th>
<th>GNU</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0,1,2,3</td>
<td>1011</td>
<td>667</td>
</tr>
<tr>
<td>4</td>
<td>0,2,4,6 cpu</td>
<td>1635</td>
<td>592</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>960</td>
<td>433</td>
</tr>
</tbody>
</table>
Cray and PGI C++ compiler optimizations issues

On Cray XE6/XK6 expect CCE to generate adequate code
- CCE 8.0.2 vs. GNU 4.6.2 performance comparison
- Various thread-core pinnings

<table>
<thead>
<tr>
<th>Threads</th>
<th>Pinning (-cc)</th>
<th>Cray</th>
<th>GNU</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0,1,2,3</td>
<td>1011</td>
<td>667</td>
</tr>
<tr>
<td>4</td>
<td>0,2,4,6</td>
<td>1635</td>
<td>592</td>
</tr>
<tr>
<td>8</td>
<td>cpu</td>
<td>960</td>
<td>433</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PAPI Event</th>
<th>Cray</th>
<th>GNU</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI.TOT_CYC</td>
<td>2’900762’613’118</td>
<td>1’512844’977’806</td>
</tr>
<tr>
<td>PAPI.FPU_IDL</td>
<td>1’130’511’978’612</td>
<td>242’206’647’131</td>
</tr>
<tr>
<td>PAPI.L2_TCM</td>
<td>1’422’885’720</td>
<td>74’482’188</td>
</tr>
</tbody>
</table>
Cray and PGI C++ compiler optimizations issues

On Cray XE6/XK6 expect CCE to generate adequate code

- CCE 8.0.2 vs. GNU 4.6.2 performance comparison
- Various thread-core pinnings

<table>
<thead>
<tr>
<th>Threads</th>
<th>Pinning (-cc)</th>
<th>Cray</th>
<th>GNU</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0,1,2,3</td>
<td>1011</td>
<td>667</td>
</tr>
<tr>
<td>4</td>
<td>0,2,4,6</td>
<td>1635</td>
<td>592</td>
</tr>
<tr>
<td>8</td>
<td>cpu</td>
<td>960</td>
<td>433</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PAPI Event</th>
<th>Cray</th>
<th>GNU</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI.TOT_CYC</td>
<td>2′900′762′613′118</td>
<td>1′512′844′977′806</td>
</tr>
<tr>
<td>PAPI.FPU_IDL</td>
<td>1′130′511′978′612</td>
<td>242′206′647′131</td>
</tr>
<tr>
<td>PAPI.L2_TCM</td>
<td>1′422′885′720</td>
<td>74′482′188</td>
</tr>
</tbody>
</table>
Cray and PGI C++ compiler optimizations issues

On Cray XE6/XK6 expect CCE to generate adequate code
- CCE 8.0.2 vs. GNU 4.6.2 performance comparison
- Various thread-core pinnings
Cray and PGI C++ compiler optimizations issues

On Cray XE6/XK6 expect CCE to generate adequate code
  - CCE 8.0.2 vs. GNU 4.6.2 performance comparison
  - Various thread-core pinnings

Vast performance difference!

<table>
<thead>
<tr>
<th>Threads</th>
<th>Pinning (-cc)</th>
<th>Cray</th>
<th>GNU</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0,1,2,3</td>
<td>1011</td>
<td>667</td>
</tr>
<tr>
<td>4</td>
<td>0,2,4,6</td>
<td>1635</td>
<td>592</td>
</tr>
<tr>
<td>8</td>
<td>cpu</td>
<td>960</td>
<td>433</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PAPI Event</th>
<th>Cray</th>
<th>GNU</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI.TOT.CYC</td>
<td>2'900'762'613'118</td>
<td>1'512'844'977'806</td>
</tr>
<tr>
<td>PAPI.FPU.IDL</td>
<td>1'130'511'978'612</td>
<td>242'206'647'131</td>
</tr>
<tr>
<td>PAPI.L2.TCM</td>
<td>1'422'885'720</td>
<td>74'482'188</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PAPI Event</th>
<th>Cray</th>
<th>GNU</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI.TOT.CYC</td>
<td>2'269'187'342'972</td>
<td>775'493'075'990</td>
</tr>
<tr>
<td>PAPI.FPU.IDL</td>
<td>1'311'686'430'846</td>
<td>141'553'650'048</td>
</tr>
<tr>
<td>PAPI.L2.TCM</td>
<td>821'641'494</td>
<td>80'416'142</td>
</tr>
</tbody>
</table>
Cray and PGI C++ compiler optimizations issues

On Cray XE6/XK6 expect CCE to generate adequate code
- CCE 8.0.2 vs. GNU 4.6.2 performance comparison
- Various thread-core pinnings

Vast performance difference!
- GNU uses full SSE width; unrolls loops, pipelines
Cray and PGI C++ compiler optimizations issues

On Cray XE6/XK6 expect CCE to generate adequate code
- CCE 8.0.2 vs. GNU 4.6.2 performance comparison
- Various thread-core pinnings

Vast performance difference!
- GNU uses full SSE width; unrolls loops, pipelines
- CCE does not

<table>
<thead>
<tr>
<th>Threads</th>
<th>Pinning (-cc)</th>
<th>Cray</th>
<th>GNU</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0,1,2,3</td>
<td>1011</td>
<td>667</td>
</tr>
<tr>
<td>4</td>
<td>0,2,4,6</td>
<td>1635</td>
<td>592</td>
</tr>
<tr>
<td>8</td>
<td>cpu</td>
<td>960</td>
<td>433</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PAPI Event</th>
<th>Cray</th>
<th>GNU</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI.TOT.CYC</td>
<td>2’900’762’613’118</td>
<td>1’512’844’977’806</td>
</tr>
<tr>
<td>PAPI.FPU.IDL</td>
<td>1’422’885’720</td>
<td>242’206’647’131</td>
</tr>
<tr>
<td>PAPI.L2.TCM</td>
<td></td>
<td>74’482’188</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PAPI Event</th>
<th>Cray</th>
<th>GNU</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI.TOT.CYC</td>
<td>2’269’187’342’972</td>
<td>775’493’075’990</td>
</tr>
<tr>
<td>PAPI.FPU.IDL</td>
<td>1’311’686’430’846</td>
<td>141’553’650’048</td>
</tr>
<tr>
<td>PAPI.L2.TCM</td>
<td>821’641’494</td>
<td>80’416’142</td>
</tr>
</tbody>
</table>
Cray and PGI C++ compiler optimizations issues

On Cray XE6/XK6 expect CCE to generate adequate code

- CCE 8.0.2 vs. GNU 4.6.2 performance comparison
- Various thread-core pinnings

Vast performance difference!

- GNU uses full SSE width; unrolls loops, pipelines
- CCE does not
- Cray bug reports filed, Cray has reproduced problem, working on optimizer

<table>
<thead>
<tr>
<th>Threads</th>
<th>Pinning (-cc)</th>
<th>Cray</th>
<th>GNU</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0,1,2,3</td>
<td>1011</td>
<td>667</td>
</tr>
<tr>
<td>4</td>
<td>0,2,4,6</td>
<td>1635</td>
<td>592</td>
</tr>
<tr>
<td>8</td>
<td>cpu</td>
<td>960</td>
<td>433</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PAPI Event</th>
<th>Cray</th>
<th>GNU</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI.TOT_CYC</td>
<td>2'900'762'613'118</td>
<td>1'512'844'977'806</td>
</tr>
<tr>
<td>PAPI.FPU_IDL</td>
<td>1'430'511'987'612</td>
<td>242'206'647'131</td>
</tr>
<tr>
<td>PAPI.L2.TCM</td>
<td>1'422'885'720</td>
<td>74'482'188</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PAPI Event</th>
<th>Cray</th>
<th>GNU</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI.TOT_CYC</td>
<td>2'269'187'342'972</td>
<td>775'493'075'990</td>
</tr>
<tr>
<td>PAPI.FPU_IDL</td>
<td>1'311'686'430'846</td>
<td>141'553'650'048</td>
</tr>
<tr>
<td>PAPI.L2.TCM</td>
<td>821'641'494</td>
<td>80'416'142</td>
</tr>
</tbody>
</table>
Optimal # threads / socket
Optimal # threads / socket

Performance comparison:
• Cray XK6: GNU 4.6.2
• SGI UV1000: Intel 11.1
• Various thread, pinning & socket configurations tried
Optimal # threads / socket

Performance comparison:
• Cray XK6: GNU 4.6.2
• SGI UV1000: Intel 11.1
• Various thread, pinning & socket configurations tried
Optimal # threads / socket

Performance comparison:
• Cray XK6: GNU 4.6.2
• SGI UV1000: Intel 11.1
• Various thread, pinning & socket configurations tried
Optimal # threads / socket

Performance comparison:
• Cray XK6: GNU 4.6.2
• SGI UV1000: Intel 11.1
• Various thread, pinning & socket configurations tried

Conclusions:
Optimal # threads / socket

Performance comparison:
• Cray XK6: GNU 4.6.2
• SGI UV1000: Intel 11.1
• Various thread, pinning & socket configurations tried

Conclusions:
• Cray XK6: 1 MPI process / socket, 16 threads
Optimal # threads / socket

Performance comparison:
• Cray XK6: GNU 4.6.2
• SGI UV1000: Intel 11.1
• Various thread, pinning & socket configurations tried

Conclusions:
• Cray XK6: 1 MPI process / socket, 16 threads
• SGI UV1000: 2 sockets / MPI Process, 16 threads
Performance Comparison Cray XE6 / SGI UV1000

- Cray XE6: AMD Interlagos, Gemini, GNU 4.6.2
- SGI UV1000: Intel Westmere, NUMAlink, Intel 11.1
ED performance, scientifically relevant test case
(4x4 lattice, Fendley model, no symmetries)
ED performance, scientifically relevant test case (4x4 lattice, Fendley model, no symmetries)
ED performance, scientifically relevant test case (4x4 lattice, Fendley model, no symmetries)
Take-home messages
Take-home messages

- SPIN benchmark on single-node:
  - **Intel Sandybridge**: best core/socket/node performance, but hyperthreading brings little improvement
  - **AMD Interlagos**: tricky to optimize thread placement, Open64 best single-node performance
  - **Intel Westmere**: competitive with Interlagos
Take-home messages

- SPIN benchmark on single-node:
  - **Intel Sandybridge**: best core/socket/node performance, but hyperthreading brings little improvement
  - **AMD Interlagos**: tricky to optimize thread placement, Open64 best single-node performance
  - **Intel Westmere**: competitive with Interlagos

- Cray Gemini: MPI two-sided is competitive with all other message passing paradigms, slight improvements with UPC
Take-home messages

- SPIN benchmark on single-node:
  - **Intel Sandybridge**: best core/socket/node performance, but hyperthreading brings little improvement
  - **AMD Interlagos**: tricky to optimize thread placement, Open64 best single-node performance
  - **Intel Westmere**: competitive with Interlagos

- Cray Gemini: MPI two-sided is competitive with all other message passing paradigms, slight improvements with UPC

- ED: Cray and PGI C++ compilers generate inferior code compared to GNU (bug reports filed)
Take-home messages

- **SPIN benchmark on single-node:**
  - **Intel Sandybridge:** best core/socket/node performance, but hyperthreading brings little improvement
  - **AMD Interlagos:** tricky to optimize thread placement, Open64 best single-node performance
  - **Intel Westmere:** competitive with Interlagos

- **Cray Gemini:** MPI two-sided is competitive with all other message passing paradigms, slight improvements with UPC

- **ED:** Cray and PGI C++ compilers generate inferior code compared to GNU (bug reports filed)

- **ED nearest-neighbor feature ensures scaling to large configurations**
Take-home messages

- **SPIN benchmark on single-node:**
  - **Intel Sandybridge:** best core/socket/node performance, but hyperthreading brings little improvement
  - **AMD Interlagos:** tricky to optimize thread placement, Open64 best single-node performance
  - **Intel Westmere:** competitive with Interlagos

- **Cray Gemini:** MPI two-sided is competitive with all other message passing paradigms, slight improvements with UPC

- **ED:** Cray and PGI C++ compilers generate inferior code compared to GNU (bug reports filed)

- **ED nearest-neighbor feature ensures scaling to large configurations**

- **Hybrid OMP/MPI viable even for simplest model**
Take-home messages

- SPIN benchmark on single-node:
  - **Intel Sandybridge**: best core/socket/node performance, but hyperthreading brings little improvement
  - **AMD Interlagos**: tricky to optimize thread placement, Open64 best single-node performance
  - **Intel Westmere**: competitive with Interlagos

- Cray Gemini: MPI two-sided is competitive with all other message passing paradigms, slight improvements with UPC

- ED: Cray and PGI C++ compilers generate inferior code compared to GNU (bug reports filed)

- ED nearest-neighbor feature ensures scaling to large configurations

- Hybrid OMP/MPI viable even for simplest model
Thank you for your attention!
wsawyer@cscs.ch