Experiences with using different OpenMP 4.5 programming styles to bring DMRG++ to Exascale

Arghya Chatterjee (ORNL / Georgia Tech.)
Oscar Hernandez (ORNL)
Vivek Sarkar (Georgia Tech.)

Application Support Team
Dr. E. D’Azevedo (CNMS)
Dr. G. Alvarez (CNMS)
Dr. Wael Elwasif (ORNL)

OpenMP Con 2017
September 18, 2017
INTRODUCTION
INTRODUCTION

• Rapidly changing microprocessor design and heterogeneous architectures
INTRODUCTION

- Rapidly changing microprocessor design and heterogeneous architectures
- Applications must adapt to exploit parallelism
INTRODUCTION

• Rapidly changing microprocessor design and heterogeneous architectures
• Applications must adapt to exploit parallelism
• Mini-application -- will serve as a foundation for Exascale-ready implementation
INTRODUCTION

- Rapidly changing microprocessor design and heterogeneous architectures
- Applications must adapt to exploit parallelism
- Mini-application -- will serve as a foundation for Exascale-ready implementation
- Co-designing with asynchronous programming models (Habanero C++, Kokkos, MAGMA)
OUTLINE
OUTLINE

DMRG++

Target Application

MOTIVATION

Physics / Nano-science & Programming Models
OUTLINE

DMRG++

Target Application

MOTIVATION

Physics / Nano-science & Programming Models

MINI-APPLICATION

Design, Implementation & Challenges encountered
OUTLINE

DMRG++
- Target Application

MOTIVATION
- Physics / Nano-science & Programming Models

MINI-APPLICATION
- Design, Implementation & Challenges encountered

STRATEGY
- Programming framework support
DMRG ++
DMRG ++

- DMRG++ $\rightarrow$ Density Matrix Renormalization Group
DMRG++

• DMRG++ → *Density Matrix Renormalization Group*

• DMRG++ algorithm -- deeper understanding of nanoscale material properties
DMRG ++

• DMRG++ → *Density Matrix Renormalization Group*

• DMRG++ algorithm -- deeper understanding of nanoscale material properties

• *Sparse matrix algebra* computational motif
DMRG ++

- DMRG++ \(\rightarrow\) *Density Matrix Renormalization Group*

- DMRG++ algorithm -- deeper understanding of nanoscale material properties

- *Sparse matrix algebra* computational motif

- Actively developed application by Material Applications group @ORNL
PROFILING : Shared Memory (DMRG++)
PROFILING: Shared Memory (DMRG++)

- Profiled on a Bulldozer AMD Opteron processor (TITAN)
- 80% of execution time → calculating the Hamiltonian

*Standard deviation of all execution times per parallel region instance over time. Execution uses 8 threads on a single TITAN node (8 blue dots per instance).*
PROFILING: Shared Memory (DMRG++)

- Profiled on a Bulldozer AMD Opteron processor (TITAN)
- 80% of execution time $\rightarrow$ calculating the Hamiltonian
- Application runs in phases (graph shows 5 phases)

*Standard deviation of all execution times per parallel region instance over time. Execution uses 8 threads on a single TITAN node (8 blue dots per instance).*
PROFILING: Shared Memory (DMRG++)

- Profiled on a Bulldozer AMD Opteron processor (TITAN)
- 80% of execution time → calculating the Hamiltonian
- Application runs in phases (graph shows 5 phases)
- Shows significant load imbalance

*Standard deviation of all execution times per parallel region instance over time. Execution uses 8 threads on a single TITAN node (8 blue dots per instance).*
• Profiled on a Bulldozer AMD Opteron processor (TITAN)

• 80% of execution time → calculating the Hamiltonian

• Application runs in phases (graph shows 5 phases)

• Shows significant load imbalance

• Dynamic Application → problem size grows → greater load imbalance

Standard deviation of all execution times per parallel region instance over time. Execution uses 8 threads on a single TITAN node (8 blue dots per instance).
MOTIVATION — Physics
MOTIVATION — Physics

• Current DMRG++ — limited to 1-Dimensional Problems
MOTIVATION — Physics

• Current DMRG++ — limited to 1-Dimensional Problems

• Scaling up — enables:
MOTIVATION — Physics

- Current DMRG++ — limited to 1-Dimensional Problems
- Scaling up — enables:
  - Practical solution for 2-D and 3-D problems
MOTIVATION — Physics

- Current DMRG++ — limited to 1-Dimensional Problems
- Scaling up — enables:
  ✤ Practical solution for 2-D and 3-D problems
  ✤ First principle (almost) models without approximating electron-electron interaction
MOTIVATION — Programming Models
MOTIVATION — Programming Models

- OpenMP
  - Nested parallelism
  - OpenMP tasks — address load imbalance
MOTIVATION — Programming Models

- OpenMP
  - Nested parallelism
  - OpenMP tasks — address load imbalance

- Exploring asynchronous task based programming models
  - New ideas could be candidates for future OpenMP extensions

↑ Productivity  ↑ Performance
MOTIVATION — Programming Models

• OpenMP
  ❖ Nested parallelism
  ❖ OpenMP tasks — address load imbalance

• Exploring asynchronous task based programming models
  ❖ New ideas could be candidates for future OpenMP extensions

  🔺 Productivity    🔺 Performance

• Compiler optimizations
EXECUTION MODEL
EXECUTION MODEL

- DMRG++ → compute intensive: calculating the sparse matrix Hamiltonian
EXECUTION MODEL

- DMRG++ → *compute intensive*: calculating the sparse matrix *Hamiltonian*
- Two-dimensional dense/sparse matrix multiplication
EXECUTION MODEL

- DMRG++ → *compute intensive*: calculating the sparse matrix *Hamiltonian*
- Two-dimensional dense/sparse matrix multiplication
- Programming styles (using OpenMP 4.5 constructs):
EXECUTION MODEL

- DMRG++ → *compute intensive*: calculating the sparse matrix *Hamiltonian*
- Two-dimensional dense/sparse matrix multiplication
- Programming styles (using OpenMP 4.5 constructs):
  - Nested Parallelism
EXECUTION MODEL

- DMRG++ → *compute intensive*: calculating the sparse matrix *Hamiltonian*
- Two-dimensional dense/sparse matrix multiplication
- Programming styles (using OpenMP 4.5 constructs):
  - Nested Parallelism
  - Multi-level Tasking
EXECUTION MODEL

• DMRG++ → compute intensive: calculating the sparse matrix Hamiltonian

• Two-dimensional dense/sparse matrix multiplication

• Programming styles (using OpenMP 4.5 constructs):
  ✤ Nested Parallelism
  ✤ Multi-level Tasking
  ✤ Multi-level Tasking with Nested Parallelism
KRONECKER PRODUCT ($y = H^* x$)
Kronecker Product \((y = H^*x)\)

Hamiltonian Matrix
Kronecker Product ($y = H^*x$)

CIJ [npatches : npatches]
KRONECKER PRODUCT \( (y = H^*x) \)

Hamiltonian Matrix

CIJ \([\text{npatches} : \text{npatches}]\)
KRONECKER PRODUCT ($y = H^*x$)

CIJ [npatches : npatches]

Hamiltonian Matrix

$A[k]$  $B[k]$
Kronecker Product \( (y = H^*x) \)
KRONECKER PRODUCT \((y = H \times x)\)

CIJ \([npatches : npatches]\)

Hamiltonian Matrix
KRONECKER PRODUCT \((y = H^*x)\)

CIJ [npatches : npatches]

Hamiltonian Matrix
KRONECKER PRODUCT \( (y = H^*x) \)

**Hamiltonian Matrix**

\[
\begin{align*}
\text{y} & \quad \rightarrow \quad \text{x} \\
C_{IJ} & \quad \left[ \text{npatches} : \text{npatches} \right]
\end{align*}
\]
PSEUDO CODE — \( y = H^*x \): Sequential
**PSEUDO CODE — (y = H*x): Sequential**

```plaintext
for ( i in C_Rows )
    Y[i] = 0.0;

for ( j in C_Cols )
    for ( k in C[i][j].list() )
        Y[i] += C[i][j].A[k] * C[i][j].B[k] * X[i]
        (dgemm call — IBM ESSL)
```
PSEUDO CODE — $(y = H^*x)$ : Parallel
PSEUDO CODE — \( y = H^*x \) : Parallel

```
parallel_for (i in C_Rows)
    Y[i] = 0.0;

parallel_reduce (j in C_Cols)
    parallel_reduce (k in C[i,j].list())
        (dgemm call — IBM ESSL)
```
PROGRAMMING STYLES

_nested Parallelism_
PROGRAMMING STYLES

- Nested Parallelism
- Multi-Level Tasking
PROGRAMMING STYLES

- Nested Parallelism
- Multi-Level Tasking
- Multi-Level Tasking with Nested Parallelism
1. Nested Parallelism
1. Nested Parallelism

1: **INPUT:** C Matrix, X Vector
2: **OUTPUT:** Y Vector
3: Enumerate over the entire C Matrix
4: **omp parallel for:** i in C_ROWS
5:  Reduction Variable: YI[i] = 0.0
6: **omp parallel for w/ reduction(YI):** j in C_COLS
7:  Reduction Variable: YIJ[i] = 0.0
8: **omp parallel for w/ reduction(YIJ):** k in C[i,j].list()
9:  **Update YIJ:** += C[i][j].A[k] ⊗ C[i][j].B[k] * X[]
10: **Update YI:** YI += YIJ
11: **Update Y:** Y = YI
12: Return vector Y
1. Nested Parallelism

1. **INPUT:** C Matrix, X Vector
2. **OUTPUT:** Y Vector
3. Enumerate over the entire C Matrix
4. `omp parallel for:` i in C_Rows
   5. Reduction Variable: YI[i] = 0.0
5. `omp parallel for w/ reduction(YI):` j in C_Cols
   6. Reduction Variable: YIJ[i] = 0.0
7. `omp parallel for w/ reduction(YIJ):` k in C[i,j].list()
8. **Update YIJ:** += C[i][j].A[k] $\otimes$ C[i][j].B[k] * X[]
9. **Update YI:** YI += YIJ
10. **Update Y:** Y = YI
11. Return vector Y
1. Nested Parallelism

1. **INPUT:** C Matrix, X Vector
2. **OUTPUT:** Y Vector
3. Enumerate over the entire C Matrix
4. **omp parallel for:** i in C_Rows
5. Reduction Variable: YI[i] = 0.0
6. **omp parallel for w/ reduction(YI):** j in C_Cols
7. Reduction Variable: YIJ[i][j] = 0.0
8. **omp parallel for w/ reduction(YIJ):** k in C[i,j].list()
9. **Update YIJ:** += C[i][j].A[k] ⊗ C[i][j].B[k] * X[]
10. **Update YI:** YI += YIJ
11. **Update Y:** Y = YI
12. Return vector Y
1. Nested Parallelism: Challenges
1. Nested Parallelism: Challenges

- Lack of support for nested work-sharing loops (Spec. Restriction)
1. Nested Parallelism: Challenges

- Lack of support for nested work-sharing loops (Spec. Restriction)

- Creating / destroying parallel regions — affects scalability
1. Nested Parallelism: Challenges

- Lack of support for nested work-sharing loops (Spec. Restriction)
- Creating / destroying parallel regions — affects scalability
- Lack of debugging support within work-sharing loops
  ✤ `omp single / omp master / omp critical`
1. Nested Parallelism: Challenges

- Lack of support for nested work-sharing loops (Spec. Restriction)
- Creating / destroying parallel regions — affects scalability
- Lack of debugging support within work-sharing loops
  - `omp single / omp master / omp critical`
- Thread affinity (using OpenMP Places / proc-bind) — depth 3
2. Multi-level Tasking: using taskloop
2. Multi-level Tasking: using taskloop

1. **INPUT:** C Matrix, X Vector
2. **OUTPUT:** Y Vector
3. Enumerate over the entire C Matrix
4. **Create OpenMP Parallel region**
   5. **omp taskloop:** i in C_Rows with grain-size
   6. **Reduction Variable:** YI[i] = 0.0
   7. **omp taskloop:** j in C_Cols with grain-size
   8. **Reduction Variable:** YIJ[i] = 0.0
   9. **omp taskloop:** in C[i,j].list() with grain-size
   10. **Update YIJ:** YIJ += C[i][j].A[k] \(\otimes\) C[i][j].B[k] \(\times\) X[k]
   11. **Update YI:** YI += YIJ
   12. **Update Y:** Y = YI
13. Return vector Y
2. Multi-level Tasking: using taskloop

1: INPUT: C Matrix, X Vector
2: OUTPUT: Y Vector
3: Enumerate over the entire C Matrix
4: Create OpenMP Parallel region
5: \textbf{omp taskloop}: i in C\_Rows with grain-size
6: \hspace{1cm} Reduction Variable: YI[i] = 0.0
7: \textbf{omp taskloop}: j in C\_Cols with grain-size
8: \hspace{1cm} Reduction Variable: YIJ[i] = 0.0
9: \textbf{omp taskloop}: in C[i,j].list() with grain-size
10: \hspace{2cm} Update YIJ: YIJ += C[i][j].A[k] \otimes C[i][j].B[k] \times X[]
11: \hspace{2cm} Update YI: YI += YIJ
12: \hspace{2cm} Update Y: Y = YI
13: Return vector Y
2. Multi-level Tasking: using taskloop

1: **INPUT:** C Matrix, X Vector
2: **OUTPUT:** Y Vector
3: Enumerate over the entire C Matrix
4: **Create OpenMP Parallel region**

5: `omp taskloop: i in C_Rows with grain-size`
   - **Reduction Variable:** $YI[i] = 0.0$

6: `omp taskloop: j in C_Cols with grain-size`
   - **Reduction Variable:** $YJ[i] = 0.0$

7: `omp taskloop: in C[i,j].list() with grain-size`
   - **Update YIJ:** $YIJ += C[i][j].A[k] \otimes C[i][j].B[k] \ast X[]$
   - **Update YI:** $YI += YIJ$
   - **Update Y:** $Y = YI$

8: Return vector Y

---

User-defined Reductions
2. Multi-level Tasking: Challenges
2. Multi-level Tasking: Challenges

- Nested task-loops — behavior is implementation specific
2. Multi-level Tasking: Challenges

- Nested task-loops — behavior is implementation specific
2. Multi-level Tasking: Challenges

- Nested task-loops — behavior is implementation specific
- Lack of support for task-level reductions
2. Multi-level Tasking: Challenges

- Nested task-loops — behavior is implementation specific
- Lack of support for task-level reductions
- No support for OpenMP constructs within task-loops
  - `omp single / omp master / omp critical`
2. Multi-level Tasking: Challenges

- Nested task-loops — behavior is implementation specific

- Lack of support for task-level reductions

- No support for OpenMP constructs within task-loops
  - `omp single / omp master / omp critical`

- Task-elasticity — no support for dynamic resource allocation
  - grain-size — thread to task-mapping
3. Multi-level Tasking with Nested Parallelism
3. Multi-level Tasking with Nested Parallelism

1. **INPUT:** C Matrix, X Vector
2. **OUTPUT:** Y Vector
3. Enumerate over the entire C Matrix
4. **OpenMP Parallel region**
   for i in C_Rows
   **Create OpenMP Tasks:** adjust granularity
   **OpenMP Parallel region w/ Reduction (YI)**
   Reduction Variable: YI[i] = 0.0
   for j in C_Cols
   **Create OpenMP Tasks:** adjust granularity
   **OpenMP Parallel region w/ Reduction (YIJ)**
   Reduction Variable: YIJ[i] = 0.0
   for k in C[i,j].list()
   YIJ += C[i][j].A[k] \( \otimes \) C[i][j].B[k] * X[]
   **Reducing YIJ in Parallel region**
   **Reducing YI in parallel region:** YI += YIJ
5. **Update Y:** Y = YI
6. Return vector Y
3. Multi-level Tasking with Nested Parallelism

1. **INPUT**: C Matrix, X Vector
2. **OUTPUT**: Y Vector
3. Enumerate over the entire C Matrix
4. **OpenMP Parallel region**
5. for i in C_Rows
6.   **Create OpenMP Tasks**: adjust granularity
7.   **OpenMP Parallel region w/ Reduction (YI)**
8.   Reduction Variable: YI[i] = 0.0
9.   for j in C_Cols
10.  **Create OpenMP Tasks**: adjust granularity
11.  **OpenMP Parallel region w/ Reduction (YIJ)**
12.  Reduction Variable: YIJ[i] = 0.0
13.  for k in C[i,j].list()
15.     **Reducing YIJ in Parallel region**
16.    **Reducing YI in parallel region**: YI += YIJ
17. **Update Y**: Y = YI
18. Return vector Y
3. Multi-level Tasking with Nested Parallelism

1. **INPUT**: C Matrix, X Vector
2. **OUTPUT**: Y Vector
3. Enumerate over the entire C Matrix
4. **OpenMP Parallel region**
5. for i in C_Rows
6.   **Create OpenMP Tasks**: adjust granularity
7.   **OpenMP Parallel region w/ Reduction (YI)**
8.     Reduction Variable: YI[i] = 0.0
9.     for j in C_Cols
10.    **Create OpenMP Tasks**: adjust granularity
11.    **OpenMP Parallel region w/ Reduction (YIJ)**
12.     Reduction Variable: YIJ[i] = 0.0
13.     for k in C[i,j].list()
15.      **Reducing YIJ in Parallel region**
16.     **Reducing YI in parallel region**: YI += YIJ
17.     **Update Y**: Y = YI
18. Return vector Y

User-defined Reductions
3. Multi-level Tasking with Nested Parallelism

1. **INPUT**: C Matrix, X Vector
2. **OUTPUT**: Y Vector
3. Enumerate over the entire C Matrix
4. **OpenMP Parallel region**
5. for i in C_ROWS
6. **Create OpenMP Tasks**: adjust granularity
7. **OpenMP Parallel region w/ Reduction (YI)**
8. Reduction Variable: YI[i] = 0.0
9. for j in C_COLS
10. **Create OpenMP Tasks**: adjust granularity
11. **OpenMP Parallel region w/ Reduction (YIJ)**
12. Reduction Variable: YIJ[i] = 0.0
13. for k in C[i,j].list()
15. **Reducing YIJ in Parallel region**
16. **Reducing YI in parallel region**: YI += YIJ
17. **Update Y**: Y = YI
18. Return vector Y
3. Multi-level Tasking with Nested Parallelism

1. INPUT: C Matrix, X Vector
2. OUTPUT: Y Vector
3. Enumerate over the entire C Matrix
4. OpenMP Parallel region
   for i in C_Rows
   Create OpenMP Tasks: adjust granularity
   OpenMP Parallel region w/ Reduction (YI)
   Reduction Variable: YI[i] = 0.0
   for j in C_Cols
   Create OpenMP Tasks: adjust granularity
   OpenMP Parallel region w/ Reduction (YIJ)
   Reduction Variable: YIJ[i] = 0.0
   for k in C[i,j].list()
   YIJ += C[i][j].A[k] ⊗ C[i][j].B[k] * X[]
   Reducing YIJ in Parallel region
   Reducing YI in parallel region: YI += YIJ
5. Update Y: Y = YI
6. Return vector Y
3. Tasking with Nested Parallelism: Challenges
3. Tasking with Nested Parallelism: Challenges

- Creating / destroying parallel regions — affects scalability
3. Tasking with Nested Parallelism: Challenges

• Creating / destroying parallel regions — affects scalability

• Task-loops — variable are first private and NOT thread private
  ✤ reduction becomes complicated
3. Tasking with Nested Parallelism: Challenges

- Creating / destroying parallel regions — affects scalability
- Task-loops — variable are *first private* and **NOT thread private**
  - reduction becomes complicated
3. Tasking with Nested Parallelism: Challenges

- Creating / destroying parallel regions — affects scalability

- Task-loops — variable are *first private* and **NOT thread private**
  - reduction becomes complicated

- Task-affinity — within OpenMP nested parallel region
  - extensions to *tied-tasks* (depth 3)
EXPERIMENTAL SETUP
Experimental Setup

- Compilers used: GCC, XLC++, CLANG++
EXPERIMENTAL SETUP

• Compilers used: **GCC**, XLC++, CLANG++

• Evaluated on Summit-dev (OLCF):
  ✦ 2, 10-core IBM Power8 CPUs (8 h/w threads per core)
  ✦ 4 NVIDIA Tesla P100 GPUs
EXPERIMENTAL SETUP

• Compilers used: GCC, XLC++, CLANG++

• Evaluated on Summit-dev (OLCF):
  ✦ 2, 10-core IBM Power8 CPUs (8 h/w threads per core)
  ✦ 4 NVIDIA Tesla P100 GPUs

• DGEMM call: IBM ESSL (and ESSL-smp)
EXPERIMENTAL DATASET
EXPERIMENTAL DATASET

CIJ [npatches : npatches]

Hamiltonian Matrix
EXPERIMENTAL DATASET

Sparsity of the Hamiltonian Matrix

Sparse Matrix

Dense Matrix
EXPERIMENTAL DATASET

Sparsity of the Hamiltonian Matrix

• Data (A[k]'s and B[k]'s)
  → mostly principal diagonal

Sparse Matrix

Dense Matrix
EXPERIMENTAL DATASET

- Data (A[k]'s and B[k]'s) → mostly principal diagonal
- Density increases → towards the center

Sparsity of the Hamiltonian Matrix
EXPERIMENTAL DATASET

Sparsity of the Hamiltonian Matrix

- Data (A[k]'s and B[k]'s) → mostly principal diagonal
- Density increases → towards the center
- Sparsity increases → away from the principal diagonal
EXPERIMENTAL DATASET

Sparsity of the Hamiltonian Matrix

- Data (A[k]’s and B[k]’s) → mostly principal diagonal
- Density increases → towards the center
- Sparsity increases → away from the principal diagonal
EXPERIMENTAL EVALUATION
EXPERIMENTAL EVALUATION

- Each loop is parallelized separately
Each loop is parallelized separately

Inner loop (jPatch) performs vector reduction
Each loop is parallelized separately

Inner loop (jPatch) performs vector reduction

Parallelizing k-loop → threaded DGEMM (IBM ESSL)
EXPERIMENTAL EVALUATION: Projection
EXPERIMENTAL EVALUATION: Projection

- Nested parallel-for’s → iPatch and jPatch

- Processor bindings
  - *Outer*: Spread
  - *Inner*: Close
EXPERIMENTAL EVALUATION: Projection

- Nested parallel-for’s → iPatch and jPatch
- Processor bindings
  - Outer: Spread
  - Inner: Close
- Dynamic scheduling
• Nested parallel-for’s  
→ iPatch and jPatch

• Processor bindings
  ✤ *Outer*: Spread
  ✤ *Inner*: Close

• Dynamic scheduling

• Significant overhead  
→ creating / destroying || regions
USING OpenMP WITH LIBRARIES
for ( i in C_Rows )
Y[i] = 0.0;

for ( j in C_Cols )

for ( k in C[i][j].list() )
Y[i] += C[i][j].A[k] * C[i][j].B[k] * X[i]
(dgemm call — IBM ESSL)
for ( i in C_Rows )
Y[i] = 0.0;
for ( j in C_Cols )
for ( k in C[i][j].list() )
Y[i] += C[i][j].A[k] * C[i][j].B[k] * X[i]

*IBM ESSL – SMP (Threaded version)*
OpenMP WITH LIBRARIES: Challenges
OpenMP WITH LIBRARIES: Challenges

• Current thread support:
  ✤ OMP_SET_NUM_THREADS <num of threads>
OpenMP WITH LIBRARIES: Challenges

• Current thread support:
  ✤ OMP_SET_NUM_THREADS <num of threads>

• Lack of support for dynamic thread assignment
OpenMP WITH LIBRARIES: Challenges

- Current thread support:
  - OMP_SET_NUM_THREADS <num of threads>

- Lack of support for dynamic thread assignment
OpenMP WITH LIBRARIES: Challenges

• Current thread support:
  ✷ OMP_SET_NUM_THREADS <num of threads>

• Lack of support for dynamic thread assignment

• Interoperability with external libraries:
  ✷ Support to extract task-level/thread-level information
  ✷ Within nested parallel region — undefined behavior
HabaneroUPC++: a Compiler-free PGAS Library. V. Kumar, Y. Zheng, V. Cavé, Z. Budimlić, and V. Sarkar, PGAS 2014
Habanero C/C++ Library (HCLib)

- Developed at Rice University as part of the Habanero Extreme Scale Software Research Project
Habanero C/C++ Library (HCLib)

- Developed at Rice University as part of the Habanero Extreme Scale Software Research Project
- HCLib — Task-based parallel programming model
HABANERO C/C++ LIBRARY (HCLib)

- Developed at Rice University as part of the Habanero Extreme Scale Software Research Project
- HCLib — Task-based parallel programming model
- HCLib runtime — light-weight, work-stealing & locality-aware

HabaneroUPC++: a Compiler-free PGAS Library. V. Kumar, Y. Zheng, V. Cavé, Z. Budimlić, and V. Sarkar, PGAS 2014
HABANERO C/C++ LIBRARY (HCLib)

- Developed at Rice University as part of the Habanero Extreme Scale Software Research Project
- HCLib — Task-based parallel programming model
- HCLib runtime — light-weight, work-stealing & locality-aware
- HCLib — path to Exascale programming system
  → intra-node: resource management & scheduling
  → inter-node: integration w/ communication models (MPI, UPC++ or OpenSHMEM)

HabaneroUPC++: a Compiler-free PGAS Library. V. Kumar, Y. Zheng, V. Cavé, Z. Budimlic’, and V. Sarkar, PGAS 2014
FINISH ACCUMULATORS

• Construct for pre-defined / user-defined reductions
FINISH ACCUMULATORS

- Construct for pre-defined / user-defined reductions
- Large flexibility for implementations:

*Finish Accumulators: a Deterministic Reduction Construct for Dynamic Task Parallelism*. J. Shirako, V. Cavé, J. Zhao, & V. Sarkar. WODET 2013
FINISH ACCUMULATORS

• Construct for pre-defined / user-defined reductions
• Large flexibility for implementations:
  ✤ Eager reduction policy — portable implementation

Finish Accumulators: a Deterministic Reduction Construct for Dynamic Task Parallelism. J. Shirako, V. Cavé, J. Zhao, & V. Sarkar. WODET 2013
FINISH ACCUMULATORS

- Construct for pre-defined / user-defined reductions
- Large flexibility for implementations:
  - Lazy reduction policy — customized for target runtime task scheduler

Lazy: Reduction at end-finish

Finish Accumulators: a Deterministic Reduction Construct for Dynamic Task Parallelism. J. Shirako, V. Cavé, J. Zhao, & V. Sarkar. WODET 2013
PSEUDO CODE: HABANERO C++
PSEUDO CODE: HABANERO C++

1: INPUT: C Matrix, X Vector  
2: OUTPUT: Y Vector  
3: Create array of Accumulators: accs <SUM> (npatches)  
4: Create OpenMP Parallel region  
5: Create OpenMP Single region  
6: for i in C_Rows do  
7:   Create OpenMP Tasks  
8:    for j in C_Cols do  
9:      Create OpenMP Tasks  
10:     YIJ[i] = 0.0  
11:    for k in C[i,j].list() do  
12:      Update YIJ: $YIJ += C[i][j] \odot C[i][j].B[k] \times X[]$  
13:     end for  
14:   end for  
15: end for  
16: Retrieve Accumulator: accs[patch]->put(YIJ)  
17: Return vector Y
1: **INPUT**: C Matrix, X Vector  
2: **OUTPUT**: Y Vector  
3: **Create array of Accumulators**: accs <SUM> (npatches)  
4: **Create OpenMP Parallel region**  
5: **Create OpenMP Single region**  
6: for i in C_Rows do  
7: **Create OpenMP Tasks**  
8: for j in C_Cols do  
9: **Create OpenMP Tasks**  
10: YIJ[i] = 0.0  
11: for k in C[i,j].list() do  
12: **Update YIJ**: \( YIJ^+ = C[i][j].A[k] \otimes C[i][j].B[k] \times X \)  
13: **Update Accumulator**: accs[ipatch] -> put(YIJ)  
14: end for  
15: end for  
16: end for  
17: Retrieve Accumulator: accs[...]->get()  
18: Return vector Y
PSEUDO CODE: HABANERO C++

1: **INPUT**: C Matrix, X Vector
2: **OUTPUT**: Y Vector
3: **Create array of Accumulators**: accs <SUM> (npatches)
4: **Create OpenMP Parallel region**
5: **Create OpenMP Single region**
6:   for i in C_Rows do
7:     **Create OpenMP Tasks**
8:       for j in C_Cols do
9:         **Create OpenMP Tasks**
10:      YIJ[i] = 0.0
11:     for k in C[i,j].list() do
12:       **Update YIJ**: \( YIJ^+ = C[i][j] \odot C[i][j] \odot B[k] \odot X[] \)
13:     end for
14:   end for
15: end for
16: **Update Accumulator**: accs[ipatch]->put(YIJ)
17: Retrieve Accumulator: accs[...]->get()
18: Return vector Y
**PSEUDO CODE: HABANERO C++**

1. **INPUT:** C Matrix, X Vector
2. **OUTPUT:** Y Vector
3. **Create array of Accumulators:** accs <SUM> (npatches)
4. **Create OpenMP Parallel region**
5. **Create OpenMP Single region**
   - for i in C_Rows do
     - **Create OpenMP Tasks**
     - for j in C_Cols do
       - **Create OpenMP Tasks**
       - for k in C[i,j].list() do
         - Update **YIJ:** $YIJ^+ = C[i][j] \otimes C[i][j].B[k] \times X[]$
       - Update Accumulator: accs[ipatch]->put(YIJ)
     - end for
   - end for
   - Retrieve Accumulator: accs[...]->get()
18. Return vector Y

**User-defined Reduction**
Candidates for Future OpenMP Support
Candidates for Future OpenMP Support

- Support for Task-level reductions
Candidates for Future OpenMP Support

- Support for Task-level reductions
- Task-inflation: dynamic resource allocation to tasks
Candidates for Future OpenMP Support

- Support for Task-level reductions
- Task-inflation: dynamic resource allocation to tasks
- Task-affinity: binding tasks to cores (extending tied-tasks)
Candidates for Future OpenMP Support

- Support for Task-level reductions
- Task-inflation: dynamic resource allocation to tasks
- Task-affinity: binding tasks to cores (extending tied-tasks)
- Thread persistence in nested parallelism (Depth 3)
Candidates for Future OpenMP Support

- Support for Task-level reductions
- Task-inflation: dynamic resource allocation to tasks
- Task-affinity: binding tasks to cores (extending tied-tasks)
- Thread persistence in nested parallelism (Depth 3)
- Dynamic OpenMP Places
Candidates for Future OpenMP Support

- Support for Task-level reductions
- Task-inflation: dynamic resource allocation to tasks
- Task-affinity: binding tasks to cores (extending tied-tasks)
- Thread persistence in nested parallelism (Depth 3)
- Dynamic OpenMP Places
- Addressing debugging Challenges
ONGOING WORK: DMRG++
ONGOING WORK: DMRG++

- OpenMP target directives — GPUs on Summit-dev
ONGOING WORK: DMRG++

- OpenMP target directives — GPUs on Summit-dev
- OpenMP parallel regions — create abstractions on CPUs
ONGOING WORK: DMRG++

• OpenMP target directives — GPUs on Summit-dev

• OpenMP parallel regions — create abstractions on CPUs

• Co-designing:
  ✤ Using Kokkos (Sandia National Lab)
  ✤ Using MAGMA — DGEMM batched kernel (Univ. of Tennessee, Knoxville)
  ✤ Habanero C/C++ Accumulators (Georgia Tech. / Rice University)
    ○ Original implementation of finish accumulators was done in Habanero-Java
ACKNOWLEDGEMENTS

• We would like to thank our collaborators:
  – ORNL: Dr. Wael Elwasif (CSMD)  
    Dr. E. D’Azevedo (CNMS)  
    Dr. G. Alvarez (CNMS)  
  – Rice University : Dr. Jun Shirako

• This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.
Bringing the DMRG++ Scientific Application to Exascale

Arghya “Ronnie” Chatterjee
Research Collaborator, CSMD, ORNL
Ph.D. Student, Georgia Tech
arghya@gatech.edu
September 18th, 2017