Porting DMRG++ Scientific Application to OpenPOWER

Arghya Chatterjee[1,3](✉), Gonzalo Alvarez[2], Eduardo D'Azevedo[1], Wael Elwasif[1], Oscar Hernandez[1], and Vivek Sarkar[3]

[1] Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA
{chatterjeea,dazevedoef,elwasif,oscar}@ornl.gov

[2] Computational Chemical and Material Sciences, Oak Ridge National Laboratory, Oak Ridge, TN 38932, USA
alvarezcampg@ornl.gov

[3] School of Computer Science, Georgia Institute of Technology, Atlanta, GA 30332, USA
{arghya,vsarkar}@gatech.edu

Abstract. With the rapidly changing microprocessor designs and architectural diversity (multi-cores, many-cores, accelerators) for the next generation HPC systems, scientific applications must adapt to the hardware, to exploit the different types of parallelism and resources available in the architecture. To get the benefit of all the in-node hardware threads, it is important to use a single programming model to map and coordinate the available work to the different heterogeneous execution units in the node (e.g., multi-core hardware threads (latency optimized), accelerators (bandwidth optimized), etc.).

Our goal is to show that we can manage the node complexity of these systems by using OpenMP for in-node parallelization by exploiting different "programming styles" supported by OpenMP 4.5 to program CPU cores and accelerators. Finding out the suitable programming-style (e.g., SPMD style, multi-level tasks, accelerator programming, nested parallelism, or a combination of these) using the latest features of OpenMP to maximize performance and achieve performance portability across heterogeneous and homogeneous systems is still an open research problem.

We developed a mini-application, Kronecker Product (KP), from the original DMRG++ application (*sparse matrix algebra*) computational

G. Alvarez—Author contribution consisted in explaining the DMRG algorithm and its implementation, and not in the OpenMP use and evaluation.

R. Yokota et al. (Eds.): ISC 2018 Workshops, LNCS 11203, pp. 418–431, 2018.
https://doi.org/10.1007/978-3-030-02465-9_29

motif to experiment with different OpenMP programming styles on an OpenPOWER architecture and present their results in this paper.

Keywords: Power8 · OpenMP · OpenMP 4.5 · Nested parallelism
Task parallelism · Data parallelism

1 Introduction

Our goal is to learn how to experiment with OpenMP for in-node parallelization incrementally by exploiting different "programming styles" supported by OpenMP 4.5 to program CPU cores and accelerators available in the OpenPOWER architecture. The heterogeneous architectures on these systems provide different types of parallelism, data locality and memory management, which is an added level of complexity when addressing an efficient use of an architecture. Mapping work efficiently onto the execution units (e.g. GPU, team of threads) while hiding data access latencies, data movement, synchronization points (e.g. across devices, etc.) requires asynchronous programming with multiple levels of parallelism and different "types" of parallelism. Expressing all of this, with a single programing model like OpenMP using a single programming style, is a challenge.

We explore ways to program the OpenPOWER architecture using an incremental approach where we can first express the different types of parallelism and map them first to the CPU cores and then to the accelerators. How to write performance portable code across multiple architectures (heterogeneous and homogeneous), is still an open research problem. A given solution of programming style may be specific to the application algorithm or computational motif. We first find ways to express multi-level parallelism using an OpenMP programming style while minimizing significant changes to the code and incrementally porting the application on-to accelerators.

To accomplish this, we have developed a mini-application, Kronecker Product (KP), from the original DMRG++ application (*sparse matrix algebra* computational motif, developed at ORNL). This is one of the key computational kernel in DMRG which computes the lowest eigenvector by evaluating the matrix-vector product of the Hamiltonian operator, in an iterative method such as the Lanczos algorithm. The new KP formulation can lead to an order of magnitude reduction in memory compared to storing the operators as sparse matrices in compressed sparse row storage. Exploiting the property of Kronecker product and with adaptive conversion from sparse to dense matrix computations can lead to significant speedups on large problems. This KP formulation has been implemented in DMRG++. Numerical experiments using DMRG++ show the new KP formulation can lead to 15X speedup in some cases.

Our mini-application uses different types of parallelism that can be implemented via different OpenMP 4.5 constructs to express multi-level parallelism and map it to multi-core threads. This work-in-progress paper primarily focuses on porting the DMRG++ mini-application to the Power-8 processors, is a first step towards expressing and mapping all the available parallelism, that we can later map to accelerators.

2 Motivation

The increasing complexity and power in modern day high performance computing platforms has resulted in an explosive growth in available hardware threads of execution to be exploited by application codes. It is expected that the upcoming Exascale platforms will provide O(1 Billion) hardware execution threads [5,8,9]. Efficient exploitation of this massive parallelism presents a major challenge for programming models and application codes. Historically, the "MPI+X" methodology has been adopted as the de-facto standard for large scale distributed application. In this model, inter-node message passing based on the MPI [3,11] standard providing the foundation for Multiple Program Multiple Data (MPMD) *top level* programming model, while a separate programming model is used to exploit on-node hardware parallelism. Recent trends in HPC node design have resulted in the increased adoption of *fat nodes* where each node exposes a significant number of hardware threads [7,10], organized around complex and deep memory hierarchies. Efficient execution on such architectures require application codes to expose a significant level of parallelism, while simultaneously avoiding synchronous programming styles that rely on global on-node barriers which introduce significant overheads.

Increased node parallelism and the need to support more irregular algorithms have driven the programming model community to explore options beyond single level parallel constructs (exemplified by a variant of `parallel for` loop and/or SIMD data-parallel vector operations). Hierarchical or nested parallelism has emerged as a major focus for modern programming model development. Mapping multi-level parallel algorithm onto hierarchical or nested programming model constructs enable a more *natural* representation of the underlying algorithm, alleviating the requirement to artificially transform the algorithm into another formulation that better matches single level parallel programming models. This however requires careful implementation and/or introduction of resource management and load balancing primitives into the underlying programming model that increases its complexity. Another challenge involves *heterogeneous hierarchy* where different nested levels implement different parallelism constructs (e.g. a higher level irregular task-parallel invoking lower level data-parallel operations).

In this paper, we outline our experience optimizing the core computational kernel of the DMRG++ application on Power8 platform using different permutations of hierarchical parallel constructs of the OpenMP 4.5 standard. We investigate the performance of a mini-application designed to faithfully represent the multi-level parallelism of the underlying patched matrix vector multiplication kernel. The work illustrates several challenges in using hierarchical parallelism in OpenMP under load imbalance conditions and issues with mixing task and data parallelism (particularly when leveraging external libraries).

3 Density Matrix Renormalization Group

3.1 The Application

The density matrix renormalization group (DMRG) is the preferred method to study low-dimensional strongly correlated electrons. It was developed to overcome the problems arising in the application of the Numerical Renormalization Group (NRG) to quantum lattice many-body system. Strongly correlated materials are a wide class of materials that show unusual - often technologically useful - electronic and magnetic properties, such as metal- insulator transitions or half-metalicity. The term "correlated" refers to the way electrons behave in these materials, which precludes relying on simple one-electron approximations.

We have used DMRG++ (developed at the Oak Ridge National Lab), a fully developed application that uses a *sparse matrix algebra* computational motif for the simulation of Hubbard-like models and spin systems [1]. In this work, we present insights into the acceleration of the DMRG++ algorithm by making use of the inherent Kronecker Product (KP) structure of the problem. The compact storage property of Kronecker Product allows us to write efficient algorithms to compute matrix-products [12]. For example, if matrices A and B are $n \times n$ matrices, then the Kronecker product $C = A \otimes B$ is $n^2 \times n^2$. Moreover, computing matrix vector product $vec(Y) = C * vec(X)$ takes $O(n^4)$ operations if matrix C were used explicitly, but can be evaluated as $Y = B * X * At$ that requires only $O(n^3)$ operations.

As we explore the opportunity to port the DMRG++ application to OpenPOWER, a mini-application capturing the core algorithmic and computational structure of the application (Kronecker Product) will serve as the foundation for the exascale-ready implementation of DMRG++. One goal of DMRG++ is to compute the lowest eigenvalue λ (which is related to the "ground-state" energy of the system) and the eigenvector Ψ of the full Hamiltonian (H_{full}) with N sites

$$H_{\text{full}}\Psi = \lambda\Psi, \text{ or } \lambda = \text{minimize}_{v \neq 0} \frac{v' H_{\text{full}} v}{v'v} \tag{1}$$

where the unit norm vector attaining the minimum value of Rayleigh quotient λ is eigenvector Ψ. Because the full Hamiltonian matrix is conceptually a very large $4^N \times 4^N$ matrix, we can only approximate this within a limited subspace of M vectors. The DMRG++ algorithm is a systematic process to find this subspace of vectors that approximates well the lowest eigenvector. The algorithm partitions the sites on the 1D lattice into the left part (called "system") and the right part (called "environment").

The full Hamiltonian can then be written as Kronecker product of operators on left and right

$$H_{\text{full}} = H_L \otimes I_R + I_L \otimes H_R + \sum_{k=0}^{K} C_L^k \otimes C_R^k \tag{2}$$

where $H_L(H_R), I_L(I_R), C_L(C_R)$ are the Hamiltonian, identity, and interaction operators on the left (right).

By bringing DMRG++ to exascale, condensed matter theorists, will be able to solve problems such as correlated electron models of *ladder geometries* as opposed to just *chain geometries*, and *multi-orbital* models as opposed to just *one-orbital* models.

3.2 Baseline Performance Characteristics of the Application

The DMRG++ application[1] spends most of its computational time calculating the sparse matrix *Hamiltonian*. We profiled the existing implementation to identify the performance bottlenecks and use this as a baseline for comparison with the implementation of our novel algorithm. We modified the original Pthreads version of DMRG++ to use OpenMP to help us quantify load imbalances.

We used this version to measure the time spent in the parallel regions and on each individual task, in-order to quantify useful work versus time spent on synchronizations as a result of load imbalances, task creation, or parallel region creation overheads.

We observed that 80% of the time was spent on the OpenMP parallel region responsible for calculating the sparse matrix *Hamiltonian*, when using eight threads on a Bulldozer AMD Opteron processor (Titan node). As seen in Fig. 1 there is a significant load imbalance across the different phases, executed by the threads for the different instances of the parallel region. The application runs in phases, and due to the dynamic nature of the problem as the application progresses the Hamiltonian matrix grows in size, and the load imbalance problem gets worse over time.

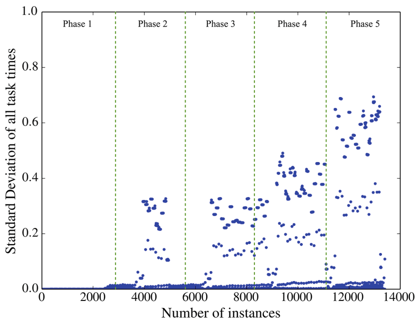

Fig. 1. Standard deviation of all execution times per parallel region instance over time. As the application progresses over time these points become more disperse. Execution uses 8 threads on a single TITAN node (8 blue dots per instance). (Color figure online)

These preliminary profiling results suggest that an asynchronous *task-based execution* model that leverages *multi-level parallelism* as exposed by the new algorithm for Kronecker Product (KP) using the latest OpenMP 4.5 constructs (see, Algorithm 1) is important to address the load imbalances by breaking down the computational workload into smaller and more uniform units of work.

[1] DMRG++ is used as a convergence algorithm to compute the lowest *eigenvector* by evaluating the matrix vector product of the Hamiltonian operator in an iterative method (Lanczos algorithm).

3.3 Hamiltonian Matrix

The key computational kernel in DMRG++ for computing the lowest eigenvector is the evaluation of matrix-vector product of the Hamiltonian matrix (H_{full}) in an iterative method such as the Lanczos algorithm. The Kronecker Product formulation expresses the Hamiltonian matrix as sum of smaller KP matrices.

The H_{full} has a special property that it "conserves" quantum number such that H_{full} matrix can reordered (or permuted) to consist of diagonal blocks. Each diagonal block is associated with a particular $(M_\uparrow, M_\downarrow)$ quantum number.

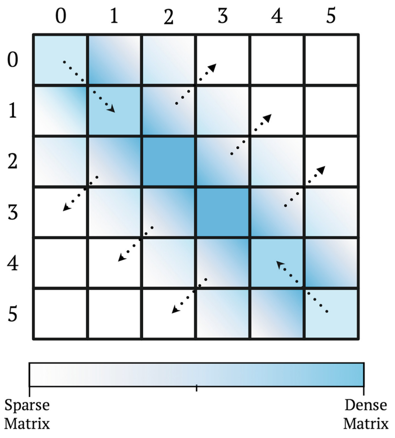

Fig. 2. Each cell is made up of smaller matrices with different sparsity/density. The arrows show how the sparsity/density, increases/ decreases.

Due to the special block diagonal structure of H_{full}, the eigen-decomposition of the full system can be obtained by separately computing the eigen-decomposition of each diagonal block. Moreover, the lowest eigen-pair will be attained at one of the diagonal blocks. For many systems, the diagonal block associated with $(M_\uparrow = N/2, M_\downarrow = N/2)$ at "half-filling" is of special interest and commonly contains the lowest eigen-pair. The use of $(M_\uparrow = N/2, M_\downarrow = N/2)$ will be assumed in the rest of the paper and the corresponding diagonal block is called H_{target}.

The two-dimensional block diagonal Hamiltonian matrix is made up of smaller operators (matrices) that are of varying weights (density). This matrix is the primary workload for our Kronecker Product application. The sparsity/density of the matrix contributes to our data layout challenges that we must overcome and the types of parallelism we can use to exploit the underlying hardware. As shown in Fig. 2, the cells get denser as we traverse towards the center of the 2-D matrix, and the sparsity of the cell increases as we traverse away from the primary diagonal.

3.4 Pseudo Code: Apply Hamiltonian Target

In order to motivate this paper and show the need for multi-level parallelism, data- and task-level parallelism, in this section we summarize the algorithm (see Algorithm 1) that we use as our research vehicle. Figure 3 shows the structure of the data-layout and the computation to evaluate the Kronecker Product. We use this algorithm to show a use-case of a triple-nested loop that can benefit from using nested-parallelism or task-based parallelism without compromising the time-complexity due to the added synchronization overheads.

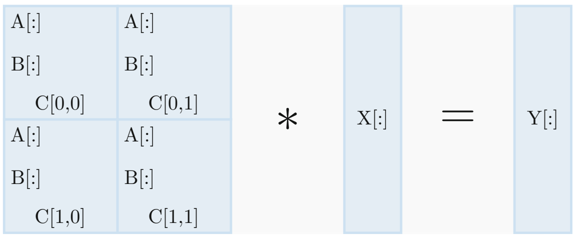

Fig. 3. Visual representation of the data layout and computation for Algorithm 1

3.5 Types of Available Parallelism in the Kronecker Product Algorithm

In this section we will discuss the potential of using two different parallelism techniques to exploit maximum benefits to the Kronecker Product algorithm (see Algorithm 1). Based on the growth aspect of the Hamiltonian matrix over time (as discussed in Sect. 3.2), the nature of the workload (as discussed in Sect. 3.3), and the dependencies between the cells in the Hamiltonian matrix, we believe, to gain the most out of the underlying hardware, we must use two different parallelism techniques in conjunction. We briefly discuss how each of these technique can benefit our algorithm.

- **Task Parallelism.** There exist no data dependencies between any of the adjacent cells in the two-dimensional Hamiltonian matrix. Each cell in the matrix can be viewed as an isolated task which computes the KP for that cell. As seen in Fig. 1, the work at each instance is sparse, to effectively use the underlying hardware threads, we must aggregate smaller workloads together. Since each task can be handled asynchronously, we can reduce the load-balancing challenges, there by reducing the sparsity of the work, per instance, of the Kronecker product.
- **Data Parallelism.** As seen in Fig. 2, the two-dimensional matrix is neither completely sparse nor dense. Just dividing the computation into a number tasks and allocating equal resources to each task will not be beneficial. Since each cell (or task) has a variable workload, we must use data parallelism with variable thread count (based on the work) in conjunction with task-parallelism.

The granularity of the amount of tasks to be created, and the variable thread count (resource allocation) per task, is of the prime importance and the key to getting maximum benefit of using machines with significant number of hardware threads on a single node.

Algorithm 1. Pseudo code to compute Kronecker multiplication

```
 1: procedure HTarget(C[][], LPatch[], RPatch[], X[])
 2:     $NPatches \leftarrow Size(C)$
 3:     $VSize \leftarrow PatchSize(LPatch, RPatch, NPatches)$
 4:     for $i \leftarrow 1, C.rows$ do
 5:         $YI \leftarrow zeros(VSize[i])$
 6:         for $j \leftarrow 1, C.cols$ do
 7:             $YIJ \leftarrow zeros(VSize[i])$
 8:             $ElemInC \leftarrow Size(C[i][j])$
 9:             for $k \leftarrow 1, ElemInC$ do
10:                 $[MatA, MatB] \leftarrow GetMat(C[i][j], k)$
11:                 $YIJ[i] \leftarrow YIJ[i] + (MatA \otimes MatB * X[])$
12:             end for
13:             for $l \leftarrow 1, VSize[i]$ do
14:                 $YI[l] \leftarrow YI[l] + YIJ[l]$
15:             end for
16:         end for
17:         for $m \leftarrow 1, VSize[i]$ do
18:             $Y[m] \leftarrow YI[m]$
19:         end for
20:     end for
21:     return Y
22: end procedure
23:
24: procedure PatchSize(LPatch[], RPatch[], NPatches)
25:     for $i \leftarrow 1, NPatches$ do
26:         $LPatchRows \leftarrow LPatch[i]$
27:         $RPatchRows \leftarrow RPatch[i]$
28:         $VSize[i] \leftarrow LPatchRows * RPatchRows$
29:     end for
30:     return VSize
31: end procedure
32:
33: procedure GetMat(c, k)
34:     $MatrixA \leftarrow c.A[k]$
35:     $MatrixB \leftarrow c.B[k]$
36:     return MatrixA, MatrixB
37: end procedure
```

4 Problem Statement

OpenMP offers different ways to express multi-level parallelism and how this parallelism can be mapped to the architectures [2,4,6]. One of the approaches is to specify nested parallelism, which increases the number of threads available to the program, to exploit more parallelism, which can be used to break the amount of work to a hierarchy of teams of threads. The other approach is to decompose work into units of work that can be schedule to teams of threads.

Both approaches have pros and cons. Using OpenMP nested parallelism increases dynamically the number of threads available to an OpenMP program which is good for load balancing, but it comes at the cost of thread creation and data locality. Nested threads may be destroyed and re-created again. This is an expensive operation if the amount of work in the nested region is small compared to the thread creation overhead. It also affects data locality as different instances of the threads touches data, affecting implicit data placement.

Using tasking improves asynchronous execution, can more easily mapped to accelerators, and improves load balancing on the application. The challenge in using tasks is that OpenMP 4.5 does not support task reduction (which are being discussed to be included in OpenMP 5.0) on certain groups of tasks. Another challenge with the tasking approach is when tasks contain data parallel work (e.g. matrix multiplications) that maps better to OpenMP work-sharing directives (e.g. parallel loops). A given task may need different resources and this becomes a scheduling and nested parallelism challenge. Given these limitations in the OpenMP programming model, if we want to explore tasks, we have to use both tasks and nested parallelism to allow the synchronization among threads, to perform reductions among a group of threads.

5 Implementation and Experimental Evaluations

5.1 Experimental Setup

For our evaluation we used the OLCF's early access system, SummitDev, with each node running on a 2 10-core IBM POWER8 CPUs (IBM S822LC) with

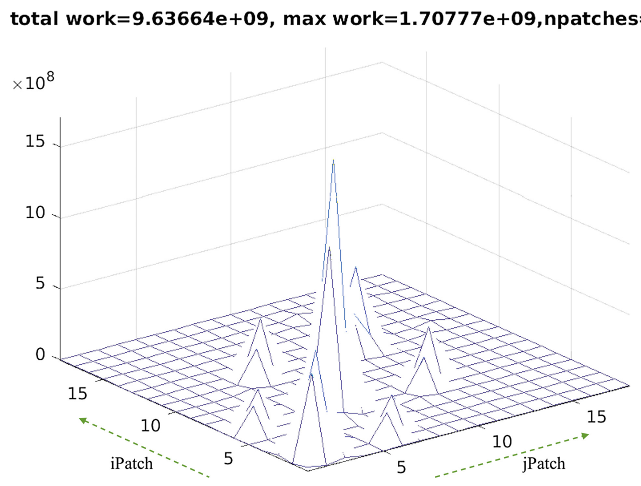

Fig. 4. Shows a three-dimensional line plot to show one of the workloads we have used for our evaluation, per cell in the Hamiltonian matrix (workload). x- and y- axes shows the matrix cells and the z-axis depicts the amount of work available per cell. One must note that the structure of the workload corresponds to the structure of the Hamiltonian matrix as discussed in Sect. 3.3.

each core supporting up-to 8 hardware threads with 256 GB DDR4 memory per node.

We tested our mini-application with different workloads, synthetically generated datasets that mimic the original structural complexity of the Hamiltonian Matrix. Figure 4 shows the distribution of work across the two dimensional matrix. The workload distribution stays the same for larger Hamiltonian matrices as well. For our evaluation purposes we have used only 20 threads (1 hardware thread per core) so that we don't oversubscribe. This setup yielded the best performance. We have used clang/clang++ compilers (version 4.0.0) for all our evaluations.

5.2 Pseudo Codes for Evaluation

For evaluation purposes we have used different parallelism styles with OpenMP constructs and IBM ESSL for the DGEMM operations in our Kronecker Product kernel (Algorithm 1). Some of the techniques used, are as follows:

Algorithm 2. Pseudo code for nested parallel work-sharing loops with OpenMP

```
1: procedure HTarget(C[][], LPatch[], RPatch[], X[])
2:     $NPatches \leftarrow Size(C)$
3:     $VSize \leftarrow PatchSize(LPatch, RPatch, NPatches)$
4:     #pragma omp parallel num_threads(numZero)
5:                          proc_bind(levelZero)
6:     #pragma omp for schedule(dynamic,1)
7:     for $i \leftarrow 1, C.rows$ do
8:         #pragma omp parallel num_threads(numOne)
9:                              proc_bind(levelOne)
10:                             reduction(YI)
11:        #pragma omp for schedule(dynamic,1)
12:        for $j \leftarrow 1, C.cols$ do
13:            for $k \leftarrow 1, ElemInC$ do
14:                $YIJ[i] \leftarrow YIJ[i] + (MatA \otimes MatB * X[])$
15:            end for
16:            for $l \leftarrow 1, VSize[i]$ do
17:                $YI[l] \leftarrow YI[l] + YIJ[l]$
18:            end for
19:        end for
20:        for $m \leftarrow 1, VSize[i]$ do
21:            $Y[m] \leftarrow YI[m]$
22:        end for
23:    end for
24:    return Y
25: end procedure
```

5.2.1 Nested OpenMP Work-Sharing Loops (2 Levels)

Algorithm 2 shows the use of nested OpenMP work-sharing constructs in two levels. For the matrix multiplication we use the IBM ESSL (non-threaded) DGEMM kernel. For experimental evaluation we used up-to 20 hardware threads on a single node (using OMP_THREAD-_LIMIT set to 20) to account for, no over-subscription of threads, which might account for higher execution time. We have used OMP_PROC_BIND at each level of the parallel region to account for thread bindings. For all experimental results we have used *spread* for the outer region and *close* for the inner region.

5.2.2 Nested OpenMP Work-Sharing with Tasking

Algorithm 3 shows the use of the tasking constructs of OpenMP with the nested OpenMP parallel regions. Ideally, we would want to use nested tasks or OpenMP 4.5 taskloop construct to exploit the tasking model, but due to no current support of task-reductions, all reductions are being performed in the OpenMP par-

Algorithm 3. Pseudo code for tasking within OpenMP parallel regions

```
1: procedure HTARGET(C[][], LPatch[], RPatch[], X[])
2:     $NPatches \leftarrow Size(C)$
3:     $VSize \leftarrow PatchSize(LPatch, RPatch, NPatches)$
4:     #pragma omp parallel num_threads(numZero)
5:                                   proc_bind(levelZero)
6:     for $i \leftarrow 1, C.rows$ do
7:         #pragma omp single
8:         #pragma omp task
9:         $YI \leftarrow zeros(VSize[i])$
10:        #pragma omp parallel num_threads(numOne)
11:                                  proc_bind(levelOne)
12:                                  reduction(YI)
13:        for $j \leftarrow 1, C.cols$ do
14:            #pragma omp single
15:            #pragma omp task
16:            for $k \leftarrow 1, ElemInC$ do                ▷ Data Parallel Loop
17:                $YIJ[i] \leftarrow YIJ[i] + (MatA \otimes MatB * X[])$
18:            end for
19:        end for
20:        for $l \leftarrow 1, VSize[i]$ do
21:            $YI[l] \leftarrow YI[l] + YIJ[l]$
22:        end for
23:        // End Parallel Region for j iteration
24:        for $m \leftarrow 1, VSize[i]$ do
25:            $Y[m] \leftarrow YI[m]$
26:        end for
27:    end for
28:    return Y
29: end procedure
```

allel regions. Due to this restriction, and using OpenMP parallel regions with OpenMP tasks, we don't observe the complete benefits of using nested tasking. Future OpenMP constructs will support task-reductions and we plan to modify our code accordingly.

5.2.3 Threaded ESSL Without Any OpenMP Constructs

Algorithm 4 shows the use the IBM threaded ESSL version for computing the DGEMM operations. Since ESSL-smp with nested OpenMP is currently not supported (or undefined), we wrapped the DGEMM operation in a single OpenMP parallel region to control the thread count for the threaded ESSL. As seen in Fig. 4, since the work is not uniformly divided, calling a threaded ESSL for each cell in the Hamiltonian matrix creates a massive overhead. We do not observe any performance benefits of using the threaded ESSL version because it currently has no support for dynamically allocating the threads (number of threads used in the ESSL-smp must be defined using the OMP_NUM_THREADS environment variable during compile time).

5.3 Evaluation

Figure 5 shows the execution time of computing the Kronecker Product over the total number of OpenMP threads used in the calculation. The bar chart shows two bars, the blue bar corresponds to the version with nested OpenMP work-sharing loops (see pseudo code Algorithm 2), and the orange bar corresponds to

Algorithm 4. Pseudo code with BLAS (using multi-threaded IBM ESSL)

```
1: procedure HTARGET(C[][], LPatch[], RPatch[], X[])
2:     $NPatches \leftarrow Size(C)$
3:     $VSize \leftarrow PatchSize(LPatch, RPatch, NPatches)$
4:     #pragma omp parallel num_threads(numZero)
5:     for $i \leftarrow 1, C.rows$ do
6:         for $j \leftarrow 1, C.cols$ do
7:             for $k \leftarrow 1, ElemInC$ do                ▷ Data Parallel Loop
8:                 Using threaded ESSL (ESSL-smp)
9:                 $YIJ[i] \leftarrow \text{DGEMM}(MatA, MatB, *X)$
10:            end for
11:        end for
12:        for $l \leftarrow 1, VSize[i]$ do
13:            Custom reduction with accumulators
14:            $YI[l] \leftarrow YI[l] + YIJ[l]$
15:        end for
16:        for $m \leftarrow 1, VSize[i]$ do
17:            $Y[m] \leftarrow YI[m]$
18:        end for
19:    end for
20:    return Y
21: end procedure
```

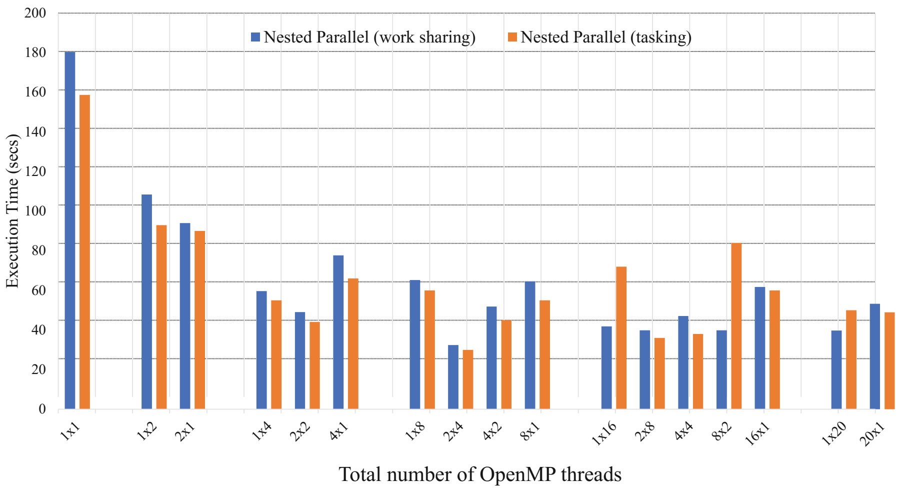

Fig. 5. Shows the execution time of computing the Kronecker product using two parallelism strategies as discussed in Sects. 5.2.1 and 5.2.2. The x-axis shows the total number of OpenMP threads used. Processor bindings for each execution is, spread for the outer region and close for the inner parallel region. Total number of threads in x-axis is shown by outer threads × inner threads. (Color figure online)

the version with nested OpenMP tasks (see pseudo code Algorithm 3). As discussed in the problem statement, challenges with using nested OpenMP tasks, is the lack of support of task-reductions across the group of threads. This forces us to use OpenMP Parallel regions to compute the reductions thereby accounting for added overhead, due to the creation/deletion of parallel regions. One might note that for most of the cases, nested parallel version gives us a better execution time than the tasking, although a pure tasking model (without nesting OpenMP tasks in parallel regions) with dynamic resource allocation would be the optimal strategy for applications like DMRG++ (dynamic data-structure with a high load imbalance across the workload). As discussed in Sect. 5.2.3, using the threaded version to compute the Kronecker product incurred a higher overhead than any of the other parallelism strategies, hence we are not providing any evaluation data-points for the threaded ESSL method.

Acknowledgment. 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. Research sponsored by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory, managed by UT-Battelle, LLC, for the U.S. Department of Energy.

References

1. Alvarez, G.: The density matrix renormalization group for strongly correlated electron systems: a generic implementation. Comput. Phys. Commun. **180**, 1572–1578 (2009)
2. Ayguade, E., Martorell, X., Labarta, J., Gonzalez, M., Navarro, N.: Exploiting multiple levels of parallelism in OpenMP: a case study. In: Proceedings of the 1999 International Conference on Parallel Processing, pp. 172–180 (1999)
3. Barker, B.: Message passing interface (MPI). In: Workshop: High Performance Computing on Stampede, vol. 262 (2015)
4. Broquedis, F., Diakhaté, F., Thibault, S., Aumage, O., Namyst, R., Wacrenier, P.-A.: Scheduling dynamic OpenMP applications over multicore architectures. In: Eigenmann, R., de Supinski, B.R. (eds.) IWOMP 2008. LNCS, vol. 5004, pp. 170–180. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-79561-2_15
5. Department of Energy, Office of Science. ECP: Exascale Computing Project, addressing challenges, March 2017
6. Duran, A., Gonzàlez, M., Corbalán, J.: Automatic thread distribution for nested parallelism in OpenMP. In: Proceedings of the 19th Annual International Conference on Supercomputing, ICS 2005, pp. 121–130. ACM, New York (2005)
7. NERSC, Lawrence Berkley National Laboratory. CORI: Cray XC40, November 2017
8. NNSA, US Department of Energy: Office of Science. ECP: Exascale Computing Project, addressing challenges (2017)
9. Oak Ridge National Lab. Stepping up software for Exascale, May 2017
10. OLCF, Oak Ridge National Laboratory. Summit: Scale new heights. Discover new solutions, November 2017
11. OpenMPI developers. OpenMPI: Open Source High Performance Computing, May 2017
12. Van Loan, C.F.: The ubiquitous Kronecker product. J. Comput. Appl. Math. **123**, 85–100 (2000)