Optimization of Condensed Matter Physics Application with OpenMP Tasking Model *

Joel Criado[1], Marta Garcia-Gasulla[1], Jesús Labarta[1],
Arghya Chatterjee[2], Oscar Hernandez[2], Raül Sirvent[1], and Gonzalo Alvarez[2] **

[1] Barcelona Supercomputing Center
{joel.criado,marta.garcia,raul.sirvent,jesus.labarta}@bsc.es
[2] Oak Ridge National Laboratory, US
{chatterjeea,oscar,alvarezcampg}@ornl.gov

Abstract. The Density Matrix Renormalization Group (DMRG++) is a condensed matter physics application used to study superconductivity properties of materials. It's main computations consist of calculating hamiltonian matrix which requires sparse matrix-vector multiplications. This paper presents task-based parallelization and optimization strategies of the Hamiltonian algorithm. The algorithm is implemented as a mini-application in C++ and parallelized with OpenMP. The optimization leverages tasking features, such as dependencies or priorities included in the OpenMP standard 4.5. The code refactoring targets performance as much as programmability. The optimized version achieves a speedup of 8.0× with 8 threads and 20.5× with 40 threads on a Power9 computing node while reducing the memory consumption to 90 MB with respect to the original code, by adding less than ten OpenMP directives.

Keywords: OpenMP · Tasks · Dependencies · Optimization · Analysis

1 Introduction and Related Work

Nowadays the High Performance Computing (HPC) community is focusing on the Exascale race. To succeed in this race, efforts are needed from all the actors, i.e., more powerful and efficient systems from architects, more flexible and scalable programming models and system software and, last but not least, applications that can exploit all the parallelism and computing power.

From the system architecture point of view, and looking at the current top systems in the top500 list, they are pushing into two clear directions: heterogeneous accelerator-based (e.g., GPUs) and many-core systems. Also, from the

** Author contributed in explaining the DMRG algorithm and its implementation, not in the OpenMP use and evaluation.

programming models and system software point of view, the efforts go to more flexible approaches [9, 12], e.g., the tasking model in OpenMP.

Looking into scientific applications, their development pushes towards two directions: their scientific field and their performance [7]. For this reason, programmability is crucial, since applications cannot be written from scratch each time the architecture where they run changes. To avoid this, they must rely on programming models and runtime systems [10].

In this paper, we will focus on optimizing a critical computational kernel, a Hamiltonian sparse matrix-vector multiplication, of the Density Matrix Renormalization Group (DMRG++) application parallelized with OpenMP, which is currently a directive-based de-facto standard to program a shared memory programming model. We present an alternative parallelization with OpenMP using the tasking model to improve its performance and memory consumption, and at the same time maintain its programmability.

The optimization has been an iterative process of performance analysis, code optimization, and evaluation. This process ensures that we target the main source of inefficiency and we improve the performance with each change. We prove the benefits of our approach evaluating it on a POWER9 cluster hosted at Barcelona Supercomputing Center (BSC) since the objective of this research is to improve the performance of DRMG++ on the Summit supercomputer at the Oak Ridge Leadership Facility (OLCF) [3], that has the same architecture.

The main contribution of this paper is not only the optimization of the DMRG++ mini-application using the OpenMP tasking model, but also, the demonstration that the tasking model has huge benefits with very irregular applications concerning their load imbalance, offering a flexible, powerful and performant yet easy approach to parallelize code. The work presented in this paper can be considered a best practice or guide for programmers when dealing with similar problems.

The remaining of this document is organized as follows: Section 2 introduces the DMRG++ application and its scientific background, how the mini-application has been extracted, the original code and its main performance issues. In Section 3 we describe the environment in which the experiments have been conducted both in hardware and software terms and explain step by step the optimizations performed in the code and their impact. Finally, in Section 5 we will summarize the conclusions we extract from this work.

2 Application Context and Background

The Density Matrix Renormalization Group (DMRG) algorithm, used in this work, is the preferred method to study quasi-one-dimensional systems. Strongly correlated materials are at the heart of current scientific and technological interest. These are a wide class of materials that show unusual, often technologically

[3] World's fastest and smartest supercomputer with a theoretical performance of 200 petaflops at Oak Ridge National Laboratory as of Novemeber 2018.

useful, electronic and magnetic properties, such as metal- insulator transitions or half-metalicity.

DMRG++ is a fully developed application that has been written entirely at Oak Ridge National Laboratory [4–6], and uses a sparse-matrix algebra computational motif for the simulation of Hubbard-like models and spin systems. 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 instead of just one-orbital models.

As an on-ramp to porting the DMRG++ application to OpenPOWER, a mini-application capturing the core algorithmic and computational structure of the application (Kronecker Product) was developed as the foundation for the exascale-ready implementation of DMRG++. In [8], the authors use OpenMP for on-node parallelization to manage the node complexity, by exploiting various "programming styles" in OpenMP 4.5 [13] (such as, SPMD style, multi-level tasks, accelerator programming and nested parallelism).

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. 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).

The critical computational kernel in DMRG++ for computing the lowest eigenvector is the evaluation of matrix-vector products of the Hamiltonian matrix (H_{full}) in an iterative method such as the Lanczos algorithm.

2.1 Mini-application Code Structure and Initial Analysis

The DMRG++ mini-application (Kronecker Product) consists of 12k lines of C++ code parallelized with OpenMP. The mini-application comes with three input sets, each one representing a typical problem size (small, medium and large) of the real application (solving real science). The original parallelization of the mini-application is shown in Listing 1.1, which consisted of three OpenMP nested loops.

Figure 1 shows the data layout and main computations performed in the Kronecker Product. The Hamiltonian matrix is a 2-D matrix with each cell consisting of two, 1-D vector of vectors (A's and B's). The length of each of the vectors in a cell will be same, but will differ across the cells. The length of each element in vector's A and B, determines the sparsity or the density of the cell in the Hamiltonian Matrix. By property of the Hamiltonian Matrix in DMRG++, the data is primarily associated in the principal axis of the matrix and the density of the cells increase as we move closer to the center of the matrix, and the sparsity of the cells increase as we move away from the primary diagonal. This data layout gives rise to a significant load imbalance across the entirety of the matrix.

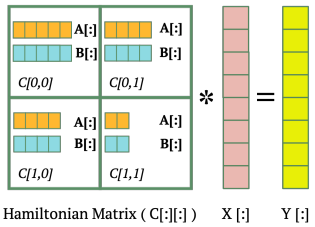

Fig. 1. Data layout in the Hamiltonian Matrix and computation for DMRG++

In Figure 2, we can see a trace obtained from an execution of the mini-application with the second parallel pragma active (corresponding to jpatch). The x axis represents the time, and the y axis OpenMP threads, 40 in this case. The color indicates the duration of useful computation bursts; dark blue represents high values, whereas light green shows low computation, and the white areas represent idle time due to lack of parallelism or load imbalance. The bottom plot outlines the total number of actives threads as a function line, with values ranging between 1 and 40 in this figure and all the following ones.

```
#pragma omp parallel for schedule(dynamic,1)
for(int ipatch=0; ipatch<npatches; ipatch++){
  std::vector<double> YI(vsize[ipatch], 0.0);
  #pragma omp parallel for schedule(dynamic,1) reduction(vec_add:YI)
  for(int jpatch=0; jpatch<npatches; jpatch++){
    std::vector<double> YIJ(vsize[ipatch], 0.0);
    #pragma omp parallel for schedule(dynamic,1) reduction(vec_add:YI)
    for(int k=0; k<CIJ.cij[ipatch][jpatch].size()){
      std::vector<double> Y_tmp(vsize[ipatch], 0.0);
      Matrix A = CIJ.cij[ipatch][jpatch]->A[k];
      Matrix B = CIJ.cij[ipatch][jpatch]->B[k];
      int has_work = (A->nnz() && B->nnz());
      if(!has_work) continue;
      A->kron_mult('n','n', *A, *B, &X[j1], &Y_tmp[0]);
      for(int i=0; i<vsize[ipatch]; i++) YIJ[i] += Y_tmp[i];
    }
    for(int i=0; i<vsize[ipatch]; i++) YI[i] += YIJ[i];
  }
  for(int i=i1; i<i2; i++) Y[i] = YI[i-i1];
}
```

Listing 1.1. Original code

In this trace, one must note that the workload in different parallel loops (arranged in columns) is not uniform across the execution. We also observe that the main bottleneck is the load imbalance (marked as the white space on each column of threads) since the variability of the workload happens within the parallel loop, too. The important load imbalance within each loop results in very poor overall efficiency, while in reality, we know there is potential concurrency between many of these loops. The core of the problem lies at the too synchronous

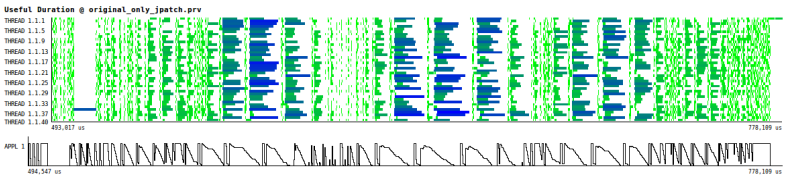

Fig. 2. Original code time line showing useful duration

structure of the *parallel do* OpenMP construct. We will explore code refactoring based on medium or coarse grain tasks to expose the potential concurrency.

3 Code Optimization and Evaluation

In this section, we are going to explain the different steps we have taken to improve the performance of the Kronecker Product mini-app. For each stage, we include the proposed source code, explain the modifications together with their motivation, and show the performance evaluation and memory consumption of that version. The optimization process has been iterative and incremental, and for this reason, all the new versions are based on modifications from the previous one and their performance is also compared with it.

3.1 Environment and Methodology

All the experiments have been performed on the CTE-Power cluster [3,15] hosted at BSC. The cluster consists of 2 login and 52 compute nodes, each of them with 2 IBM Power9 8335-GTH 2.4 GHz processors (20 cores per processor, 4 SMT per core adding 160 SMTs per node), 512 GB of main memory at 2666 MHz distributed in 16 dimms and 4 NVIDIA V100 GPUs with 16 GB HBM2 memory. In all our experiments we have not used the GPUs nor the SMT, therefore we will use maximum 40 threads per core.

We have used GCC 8.1.0 as C and C++ compiler and its OpenMP runtime implementation and linked with IBM ESSL 5.4 library. Traces have been obtained using Extrae 3.5.4 [1,11] and visualized with Paraver [2,14].

All numbers reported (both for time and memory) are the average of 5 independent runs of 10 consecutive iterations, to be able to compare all versions to each other. In all cases, the relative error is below 5%, therefore, we do not show error bars on the charts for the sake of clarity. All experiments have been performed in one compute node of CTE-Power.

3.2 First Taskification

As we have seen from Figure 2, the main performance issue is load imbalance. The current parallelization using nested parallelism worsens this problem due to the implicit barrier necessary at the end of each parallel loop. For this reason,

the first modification of the code consists of removing the nested parallelism and using a task approach instead. In previous work, A. Chatterjee et al. [8] already explored a task version of this Kernel, but there is minimal overlap with this new version, illustrated in Listing 1.2.

```
1 #pragma omp parallel
2 #pragma omp single
3 for(int ipatch=0; ipatch<npatches; ipatch++){
4   for(int jpatch=0; jpatch<npatches; jpatch++){
5     for(int k=0; k<CIJ.cij[ipatch][jpatch].size(); k++){
6       double* Y_tmp = new double[vsize[ipatch]]();
7       Matrix A = CIJ.cij[ipatch][jpatch]->A[k];
8       Matrix B = CIJ.cij[ipatch][jpatch]->B[k];
9       int has_work = (A->nnz() && B->nnz());
10      if(has_work){
11        //Tasks in charge of dgemms. Red.
12 #pragma omp task depend(inout:Y_tmp[0:vsize[ipatch]]) firstprivate(A,B)
13        A->kron_mult('n','n', *A, *B, &X[j1], &Y_tmp[0]);
14        //Reduction tasks. Green.
15 #pragma omp task depend(inout: Y[i1:i2]) depend(in: Y_tmp[0:vsize[ipatch]])
16        {
17        int ilocal=0;
18        for(int i=i1; i<i2; i++) Y[i] += Y_tmp[ilocal++];
19        delete[] Y_tmp;
20 } } } } }
```

Listing 1.2. First Taskification

We keep only one parallel region (line 1) and add a single region (line 2) where the tasks will be created. We define two kinds of tasks: a **computation task**, that perform *dgemm* operations, and a **reduction task**, that accumulates partial results from the first one into the return array. To pass intermediate results from a *dgemm* task to its corresponding reduction task, we use temporal arrays which are allocated sequentially by the same thread that creates the tasks. To guarantee the correctness of the program the first task has an *out* dependence on the temporary array and the second one has an *in* dependence. Additionally, we define an *inout* dependence on a fraction of the return array to avoid several tasks reducing on the same portion of Y simultaneously.

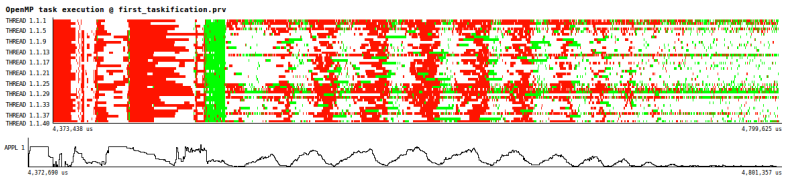

Fig. 3. First Taskification: Task Execution Timeline

Figure 3 shows a trace of this version. In this case, the color represents which task is being executed by each thread: the computing task is labeled as red and the reduction task as green. This version allows exploiting more parallelism, with many pink tasks running concurrently and avoiding periodic barriers. Nevertheless, we can still see other problems: a) the duration of *computation tasks* has a considerable variability, with some tasks taking 18 microseconds while the average is 150 microseconds, and b) a single thread must allocate all the buffers

before creating the corresponding task, which adds significant overhead. The second situation can be better appreciated with the pulsations of the bottom function plot in the figure, that shows the total number of pending tasks generated. When it starts executing regions with fine grain tasks, the number of ready tasks decrease quickly, and there are not enough tasks to fill every thread.

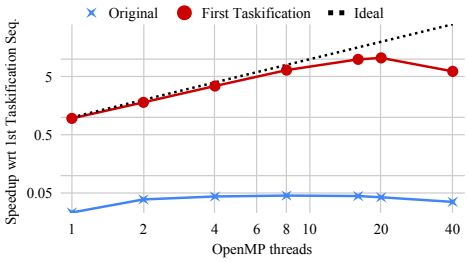

Fig. 4. Original vs First Taskification

In Figure 4, we can observe the speedup of the *Original* version with the nested work sharing and the taskified code. On the x axis we plot the number of OpenMP threads used, and the y axis shows the speedup with respect to the *First Taskification* version executed sequentially (i.e., with no OpenMP pragmas). Comparing one version to the other, we can see a speedup of 41.65× with one thread when using the taskified version, which reveals the huge impact on the performance introduced by the nested worksharings. We can see the biggest difference at 20 OpenMP threads, with a value of 457.82×. Comparing the performance of the *First Taskification* version with respecto to ideal, it goes up to 10.55× using 20 OpenMP threads, which indicates that there is still margin to improve the performance in subsequent versions. On the other hand, the speedup of the original code is around 0.045× for 4 threads and above.

Memory usage [GB]	**OpenMP Threads**						
Code version	1	2	4	8	16	20	40
Original	1.39	1.40	1.40	1.42	1.46	1.47	1.55
First Taskification	1.39	1.43	1.44	1.47	1.62	1.64	1.61
Three Tasks	1.39	1.43	1.45	1.52	1.64	1.61	1.61
Priorities & Buffer Reuse	1.39	1.40	1.41	1.42	1.44	1.46	1.50
Overlap Iterations	1.39	1.41	1.41	1.47	1.47	1.48	1.49
Nested Tasks	1.39	1.39	1.40	1.41	1.43	1.41	1.46

Fig. 5. Total memory usage in GB for each version and different number of threads

In Figure 5, we observe the total memory used in GB by each version depending on the number of OpenMP threads used. As we can see, the memory consumption is not a critical factor in our situation, but we want to demonstrate that this techniques don't increase the memory usage. In addition, memory usage is indeed a critical factor to scientifics using DMRG++, so porting this changes to the original application will allow them to use bigger inputs. In the *First Taskification* version since all the buffers are allocated at the beginning, the memory usage is higher than in the original one, and when using 20 threads the memory increases from 1.47 GB to 1.64 GB. The rest of versions will be presented in the following sub-sections.

3.3 Tasks Distinction Based on Grain Size

In the *First Taskification* version, we see an issue with the very fine-grained tasks, which introduce a relevant overhead. To address this, in *Tasks' Size Distinction* version we define 3 kinds of tasks: **Fine grain tasks** with a low computational

load, will do both the *computation* and the *reduction* (line 11); **Coarse grain compute task** (line 16); and the corresponding **reduction task** (line 19). The decision if a task has a high or a low load is taken based on a threshold that can be set by the user. The code corresponding to this version can be seen in Listing 1.3.

```
1 #pragma omp parallel
2 #pragma omp single
3 for(int ipatch=0; ipatch<npatches; ipatch++){
4   for(int jpatch=0; jpatch<npatches; jpatch++){
5     for(int k=0; k<CIJ.cij[ipatch][jpatch].size(); k++){
6       Matrix A = CIJ.cij[ipatch][jpatch]->A[k];
7       Matrix B = CIJ.cij[ipatch][jpatch]->B[k];
8       if(A->nnz() && B->nnz()){
9         if(vsize[ipatch] <= Threshold){
10          //Create single task for small pieces of work
11 #pragma omp task depend(inout: Y[i1:i2]) firstprivate(A, B)
12          A->kron_mult('n','n', *A, *B, &X[j1], &Y[i1]);
13        }else{
14          //Create compute task and reduction task for larger pieces
15          double* Y_tmp = new double[vsize[ipatch]]();
16 #pragma omp task depend(inout: Y_tmp[0:vsize[ipatch]]) firstprivate(A, B)
17          A->kron_mult('n','n', *A, *B, &X[j1], &Y_tmp[0]);
18 #pragma omp task depend(inout: Y[i1:i2]) depend(in: Y_tmp[0:vsize[ipatch]])
19          {
20          int ilocal=0;
21          for(int i=i1; i<i2; i++) Y[i] += Y_tmp[ilocal++];
22          delete[] Y_tmp;
23 } } } } } }
```

Listing 1.3. Tasks' Size Distinction

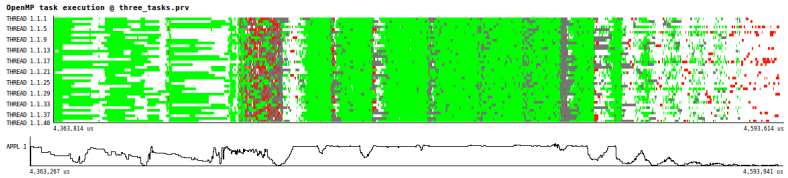

Fig. 6. Tasks' Size Distinction: Task Execution Timeline

In Figure 6, we plot a trace showing the behavior of this version. In this case, fine grain tasks are represented in red, compute tasks in green and reduction tasks in grey. Using this strategy, the application can exploit more parallelism, and there is less overhead of task creation, since the average task size has increased and the total number of tasks has decreased. The function at the bottom shows how this version can make better usage of the threads, reducing the number of pulsations from Figure 3 of *First Taskification*.

In Figure 7 (left), we plot the speedup obtained by the *Tasks' Size Distinction* version, which is computed with respect the *First Taskification* version when executed sequentially. As it can be seen, the previous version has a better performance when using a single thread, due to the if-else structure introduced in *Tasks' Size Distinction* version. Nevertheless, this fact allows for a better scaling, reaching a speedup of 1.19× and 1.6× when using 20 and 40 OpenMP threads, respectively.

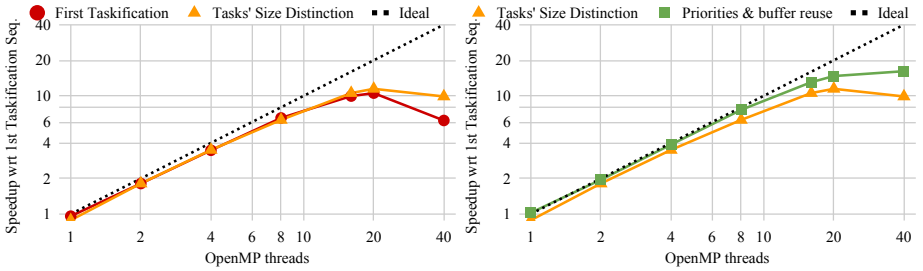

Fig. 7. First Taskification vs Tasks' Size Distinction (Left). Tasks' Size Distinction vs Priorities (Right)

3.4 Priorities and Buffer Reuse

In this version, we are going to address two problems from the *Tasks' Size Distinction* version: a) the scheduling of the tasks to improve the performance, and b) reusing buffer to improve the memory consumption. The new code is shown in Listing 1.4. To decrease memory consumption, instead of allocating one Y_tmp array for each task, we allocate a buffer of N arrays (line 1) that are reused by the different tasks. Each buffer establishes the dependence between compute and reduction tasks (previously Y_tmp), and also creates an anti-dependence between two compute tasks that use the same buffer.

```
1  double* buffers[NBUFF]; //Set of buffers to limit memory usage
2  #pragma omp parallel
3  #pragma omp single
4  for(int ipatch=0; ipatch<npatches; ipatch++){
5    for(int jpatch=0; jpatch<npatches; jpatch++){
6      for(int k=0; k<CIJ.cij[ipatch][jpatch].size(); k++){
7        Matrix A = CIJ.cij[ipatch][jpatch]->A[k];
8        Matrix B = CIJ.cij[ipatch][jpatch]->B[k];
9        if(A->nnz() && B->nnz()){
10         if(vsize[ipatch] <= Threshold){
11 #pragma omp task depend(inout: Y[i1:i2]) firstprivate(A, B) priority(0)
12           kron_mult('n','n', A, B, &X[j1], &Y[i1]); //New kron_mult call
13         }else{
14           mybuff = next = (next+1)%NBUFF;
15           int prio = vsize[ipatch] > PrioThreeshold; //Dynamic priority
16 #pragma omp task depend(inout:buffers[mybuff]) \
17                  firstprivate(mybuff,ipatch,A,B) priority(prio)
18           {
19           double* Y_tmp = new double[vsize[ipatch]]();
20           buffers[mybuff] = Y_tmp;
21           kron_mult('n','n', A, B, &X[j1], Y_tmp);
22           }
23 #pragma omp task depend(inout:Y[i1:i2],buffers[mybuff]) \
24                  firstprivate(mybuff) priority(10)
25           {
26           double* Y_tmp=buffers[mybuff];
27           int ilocal=0;
28           for(int i=i1; i<i2; i++) Y[i] += Y_tmp[ilocal++];
29           delete[] Y_tmp;
30 } } } } } }
```

Listing 1.4. Priorities and Buffer Reuse

Task priorities have also been added to help on improving the schedule of the tasks (i.e., schedule the bigger tasks first, followed by the smaller ones). Reduction tasks (line 23) have been assigned with the highest priority, to free

buffer positions as soon as possible. The coarse grain compute tasks are assigned a variable priority depending on their workload (line 16).

Figure 8 shows a task execution timeline of this version (top), where the task coloring corresponds to the one in Figure 6, and the task execution order (middle), where green stands for older tasks (instantiated early) and blue for younger ones. We can see the execution of tasks instantiated "late" (dark blue) are intermixed with the execution of tasks instantiated "early" (light green). Coalescing the priorities of the tasks as explained earlier, the execution timeline is now more compact, thereby leveraging more parallelism and almost removing pulsations of tasks (bottom function plot).

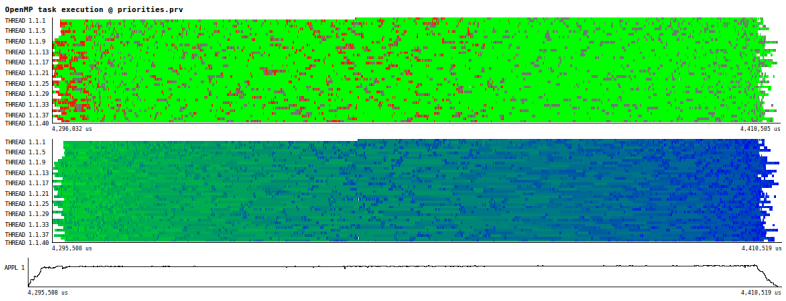

Fig. 8. Priorities and Buffer Reuse: Task Execution and Task Order Timelines

In Figure 7(right), we plot the speedup obtained of the *Priorities and Buffer Reuse* version compared with the *Tasks' Size Distinction*. The speedup has been computed in both cases with respect to the *First Taskification* version executed sequentially. One should note that this new version performs better than the previous one, in particular for a high number of thread count. With 16 OpenMP threads, the execution is 1.28× faster than the *Tasks' Size Distinction* version. We can also see that the performance improves for 20 and 40 threads, although it is still far from the ideal one. Despite this fact, this is the first version which performance improves when using 40 OpenMP threads. Regarding the memory usage, it has been reduced achieving values equal to the *Original* version, as shown in Figure 5.

3.5 Overlap Iterations

One of the issues detected in the *Priorities and Reuse Buffer* version is the imbalance at the end of the iteration, produced by the lack of parallelism and coarse grained tasks that need to be executed at that moment. Taking into account that the real application performs several iterations of this kernel, overlapping different iterations can reduce the impact of the imbalance. To achieve this, we move the parallel region up to include several iterations, as can be seen in Listing 1.5.

```
#pragma omp parallel
#pragma omp single
for(int its=0; its<NITS; its++){
  //same code from Listing 1.4
}
```

Listing 1.5. Overlap iterations

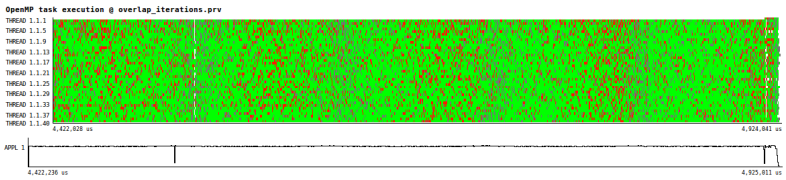

Fig. 9. Overlap Iterations: Task Execution Timeline

In Figure 9 we can see a trace of this version including 5 iterations. The color represents the task being executed: red for **fine grain tasks** including computation and reduction, green for **coarse grain compute tasks**, and grey for **reduction tasks**. Although we can visually detect the five iterations, we can see that the tasks belonging to different iterations are executed concurrently, thereby increasing the parallelism and reducing the imbalance.

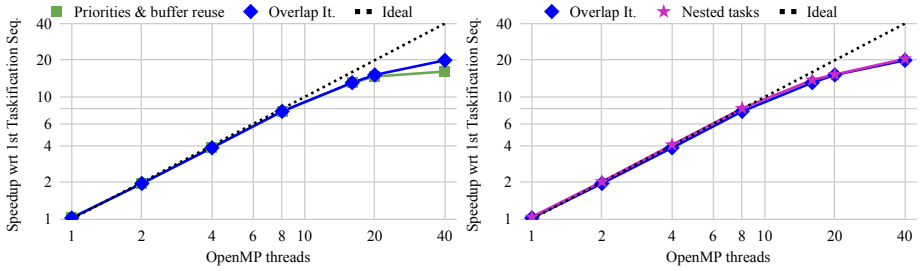

Fig. 10. Priorities vs Overlap It. (Left). Overlap It. vs Nested Tasks (Right)

Figure 10 (left) shows the speedup for the *Overlap Iterations* version. We can see that its performance is slightly better than the *Priorities* version, except in the case of 40 threads, where it obtains an improved gain of 1.24×. Because the unbalance increase as we add more threads, we will have a better benefit from overlapping iterations.

3.6 Nested Tasks

Upon further analysis of the *Overlap Iterations* version, we observe that grey tasks, with an average duration of few microseconds, are limiting the scalability.

To address this issue, we implement a new task decomposition, with two levels of tasks. This strategy takes into account that the load only depends on ipatch.

```
1  char* sentinel = new char[npatches](); //Dependence token
2  #pragma omp parallel
3  #pragma omp single
4  for(int its=0; its<NITS; its++){
5    for(int ipatch=0; ipatch<npatches; ipatch++){
6      int fine_grain = vsize[ipatch] <= Threshold;
7      //New external task. It will generate more tasks based on size of ipatch
8  #pragma omp task depend(inout: sentinel[ipatch]) priority(10)
9      for(int jpatch=0; jpatch<npatches; jpatch++){
10       for(int k=0; k<CIJ.cij[ipatch][jpatch].size(); k++){
11         Matrix A = CIJ.cij[ipatch][jpatch]->A[k];
12         Matrix B = CIJ.cij[ipatch][jpatch]->B[k];
13         if(A->nnz() && B->nnz()){
14           //Fine_grain branch. Each task will always take the same path
15           if(fine_grain){
16             kron_mult('n','n', A, B, &X[j1], &Y[i1]);
17           }else{
18             double** buffer = new double*;
19  #pragma omp task depend(out:buffer)firstprivate(A,B,buffer,ipatch)priority(0)
20             {
21             double* Y_tmp = new double[vsize[ipatch]]();
22             buffer[0] = Y_tmp;
23             kron_mult('n','n', A, B, &X[j1], Y_tmp);
24             }
25  #pragma omp task depend(inout:Y[i1:i2])depend(in:buffer)firstprivate(buffer)
       priority(5)
26             {
27             double* Y_tmp=buffer[0];
28             int ilocal=0;
29             for(int i=i1; i<i2; i++) Y[i] += Y_tmp[ilocal++];
30             delete[] Y_tmp;
31             delete[] buffer;
32       } } } }
33       #pragma omp taskwait
34 } } }
```

Listing 1.6. Nested Tasks

For each ipatch, a single task is created, and inside this task, there are two paths depending on the threshold of the ipatch size set by the user. If the ipatch is considered fine grain, then the computation and reduction are computed (line 16). On the other hand, if the ipatch is deemed to be coarse grain, then two tasks are created: the compute task (line 19) and the reduction task (line 25). To guarantee that ipatches from different iterations are executed in the correct order (i.e., they do not overtake each other) a sentinel is used to generate a dependence (line 8).

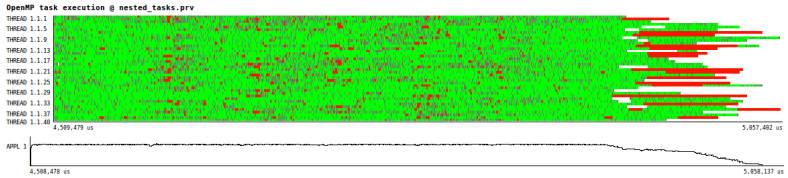

Fig. 11. Nested Tasks: Task Execution Timeline

In Figure 11, we can see a task execution timeline of this version with five iterations overlapped. Here, the **external task** is red, while grey and green are

the **reduction** and **compute tasks** respectively, like in the previous versions. This version has reduced the number of tasks created and parallelized the tasks' creation, which reduces the overhead from *Overlap Iterations* version. However, it presents a severe imbalance at the end, caused by the creation of tasks near the end, which limits its performance and will be addressed in future work.

Figure 10 (right), shows the speedup obtained with the *Nested Tasks* version over the *Overlap Iterations*. We can see a slight gain of performance of 1.06× and 1.05× using 8 and 16 OpenMP threads respectively; with 40 OpenMP threads the gain is even smaller, reaching 1.03×, caused by the big unbalance at the end of the execution shown in Figure 11. Besides, this version reduces the memory usage for all the number of threads, with the higher difference from *Original* version at 90 MB when using 40 OpenMP threads, being the best one both in terms of execution time and memory usage, as illustrated in Figure 5.

4 Summary and Best Practices

In this section, we summarize all the results obtained by the different optimizations and we present the lessons learned with this work as some best practices and guidelines for developers facing similar challenges.

In Figure 12 we can see the speedup of the different versions presented with respect to the sequential execution of the *First Taskification* code. The performance of the *Original* code is not shown because its performance is too far from the optimized versions to be displayed on the same scale; nevertheless, it can be found in Figure 4.

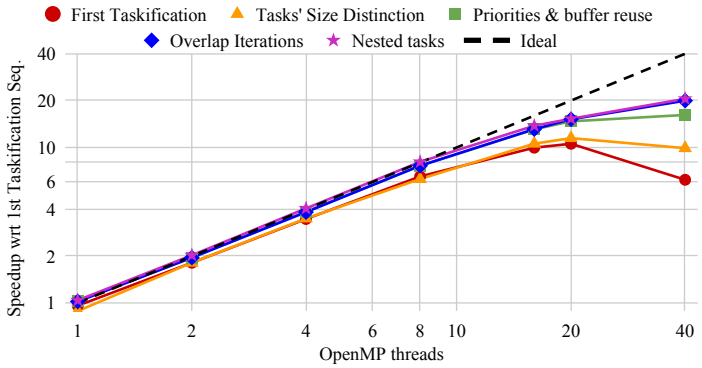

Fig. 12. Performance summary of the different versions

The most important improvement was obtained when adding the tasking model instead of the nested loop parallelism (585× with 40 threads). The following optimizations provided incremental gain, less spectacular but significant in global and especially when scaling to a high number of threads. Comparing

the *First Taskification* and *Nested Tasks* versions we observe a speedup of 1,24× with 8 threads and 3,32× with 40 threads. We conclude that these fine grain optimizations are necessary when scaling applications to a high number of cores.

The limiting factor of the last version (*Nested Tasks*) seems to be the NUMA effect when using two sockets and the late instantiation of some "big" tasks, leaving a significant load imbalance at the end. As future work, we can try to mitigate this effect by using higher priorities for tasks that will create more tasks.

The main lesson learned with this work is the potential of the tasking model to address irregular problems, even for codes with a regular structure with loops, where a `parallel loop` construct can be used straight forward. Also, we have seen the high impact on the performance of synchronizations imposed by the parallel construct. We highlight how using clauses like `priorities` or `dependences` to fine tune the parallelization are crucial to achieving good scalability to a high number of threads while keeping the flexibility of the runtime to schedule them.

5 Conclusions

In this study, we have presented the modifications done to the Kronecker Product mini-application with the OpenMP tasking model. We have demonstrated the benefits of using this model, both in terms of performance and programmability for algorithms with such irregular computation. Besides, this work can be considered as a best practice for other researchers dealing with similar algorithms, including uneven workloads, huge imbalances or granularity problems.

Applying the described changes to the mini-application, we report a speedup of 8.0× with 8 OpenMP threads of the *Nested Tasks* version with respect to the serial code and 20.5× with 40 threads. Also, the memory usage decreases 90 MB, from *Original* version. The optimization has been done keeping the number of changes to the source code to a minimum. Moreover, the number of pragmas has been reduced increasing the programmability and maintainability of the code.

We consider this kind of work, not only an optimization and best practice programming guidelines, but also useful for co-design effort to the OpenMP community. For example, the if-else structure to generate a different kind of tasks depending on its load is not as elegant as one would want. The compiler could generate the code for the two branches given the corresponding syntax.

As future work, imbalances from *Nested Tasks* version will be addressed. Also, some features from OpenMP 5.0 may be used, like the *mutexinoutset* dependence type. Finally, a hybrid approach with MPI may help to reduce the NUMA effect detected when scaling from 20 to 40 cores.

Acknowledgments

This work is partially supported by the Spanish Government through Programa Severo Ochoa (SEV-2015-0493), by the Spanish Ministry of Science and Technology (project TIN2015-65316-P), by the Generalitat de Catalunya (contract 2017-SGR-1414) and by the BSC-IBM Deep Learning Research Agreement, under JSA "Application porting, analysis and optimization for POWER and POWER AI". This work was partially supported by the Scientific Discovery through Advanced Computing

(SciDAC) program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research and Basic Energy Sciences, Division of Materials Sciences and Engineering. 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.

References

1. Extrae website. https://tools.bsc.es/extrae - Last accessed May. 2019
2. Paraver website. https://tools.bsc.es/paraver - Last accessed June 2019
3. Power9 CTE User's Guide. https://www.bsc.es/support/POWER_CTE-ug.pdf - Last accessed May. 2019
4. Alvarez, G.: DMRG++ website. https://g1257.github.com/dmrgPlusPlus
5. Alvarez, G.: Implementation of the SU(2) Hamiltonian symmetry for the DMRG algorithm. Computer Physics Communications **183**, 2226–2232 (2012)
6. Alvarez, G.: The density matrix renormalization group for strongly correlated electron systems: A generic implementation. Computer Physics Communications **180**(9), 1572–1578 (2009)
7. Cajas, J.C., Houzeaux, G., Vázquez, M., Garcia, M., Casoni, E., Calmet, H., Artigues, A., Borrell, R., Lehmkuhl, O., Pastrana, D., et al.: Fluid-structure interaction based on hpc multicode coupling. SIAM Journal on Scientific Computing **40**(6), C677–C703 (2018)
8. Chatterjee, A., Alvarez, G., D'Azevedo, E., Elwasif, W., Hernandez, O., Sarkar, V.: Porting DMRG++ Scientific Application to OpenPOWER. In: Yokota, R., Weiland, M., Shalf, J., Alam, S. (eds.) High Performance Computing. pp. 418–431. Springer International Publishing, Cham (2018)
9. Garcia, M., Labarta, J., Corbalan, J.: Hints to improve automatic load balancing with lewi for hybrid applications. Journal of Parallel and Distributed Computing **74**(9), 2781–2794 (2014)
10. Garcia-Gasulla, M., Mantovani, F., Josep-Fabrego, M., Eguzkitza, B., Houzeaux, G.: Runtime mechanisms to survive new hpc architectures: A use case in human respiratory simulations. The International Journal of High Performance Computing Applications p. 1094342019842919 (2019)
11. Llort, G., Servat, H., González, J., Giménez, J., Labarta, J.: On the usefulness of object tracking techniques in performance analysis. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. p. 29. ACM (2013)
12. Martineau, M., McIntosh-Smith, S.: The Productivity, Portability and Performance of OpenMP 4.5 for Scientific Applications Targeting Intel CPUs, IBM CPUs, and NVIDIA GPUs. In: International Workshop on OpenMP. pp. 185–200. Springer (2017)
13. OpenMP Architecture Review Board: OpenMP 4.5 Specification. Tech. rep. (November 2015), https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf
14. Pillet, V., Labarta, J., Cortes, T., Girona, S.: Paraver: A tool to visualize and analyze parallel code. In: Proceedings of WoTUG-18: transputer and occam developments. vol. 44, pp. 17–31. IOS Press (1995)
15. Sadasivam, S.K., Thompto, B.W., Kalla, R., Starke, W.J.: IBM Power9 Processor Architecture. IEEE Micro **37**(2), 40–51 (2017)