An Evaluation of Task-Parallel Frameworks for Sparse Solvers on Multicore and Manycore CPU Architectures

Abdullah Alperen
alperena@msu.edu
Michigan State University
East Lansing, Michigan, USA

MD Afibuzzaman
afibuzza@msu.edu
Michigan State University
East Lansing, Michigan, USA

Fazlay Rabbi
rabbimd@msu.edu
Michigan State University
East Lansing, Michigan, USA

M. Yusuf Özkaya
myozka@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

Ümit V. Çatalyürek
umit@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

Hasan Metin Aktulga
hma@msu.edu
Michigan State University
East Lansing, Michigan, USA

ABSTRACT

Recently, several task-parallel programming models have emerged to address the high synchronization and load imbalance issues as well as data movement overheads in modern shared memory architectures. OpenMP, the most commonly used shared memory parallel programming model, has added task execution support with dataflow dependencies. HPX and Regent are two more recent runtime systems that also support the dataflow execution model and extend it to distributed memory environments. We focus on parallelization of sparse matrix computations on shared memory architectures. We evaluate the OpenMP, HPX and Regent runtime systems in terms of performance and ease of implementation, and compare them against the traditional BSP model for two popular eigensolvers, Lanczos and LOBPCG. We give a general outline in regards to achieving parallelism using these runtime systems, and present a heuristic for tuning their performance to balance tasking overheads with the degree of parallelism that can be exposed. We then demonstrate their merits on two architectures, Intel Broadwell (a multicore processor) and AMD EPYC (a modern manycore processor). We observe that these frameworks achieve up to 13.7× fewer cache misses over an efficient BSP implementation across L1, L2 and L3 cache layers. They also obtain up to 9.9× improvement in execution time over the same BSP implementation.

CCS CONCEPTS

• **Computing methodologies** → **Shared memory algorithms**; **Parallel programming languages**.

KEYWORDS

sparse solvers, task parallelism, asynchronous many-task programming, performance optimization, runtime systems, OpenMP, HPX, Regent.

ICPP '21, August 9–12, 2021, Lemont, IL, USA

ACM ISBN 978-1-4503-9068-2/21/08…$15.00
https://doi.org/10.1145/3472456.3472476

ACM Reference Format:
Abdullah Alperen, MD Afibuzzaman, Fazlay Rabbi, M. Yusuf Özkaya, Ümit V. Çatalyürek, and Hasan Metin Aktulga. 2021. An Evaluation of Task-Parallel Frameworks for Sparse Solvers on Multicore and Manycore CPU Architectures. In *50th International Conference on Parallel Processing (ICPP '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3472456.3472476

1 INTRODUCTION

Sparse matrix computations manifest themselves in many forms such as the solution of systems of linear equations, matrix factorizations, linear least squares problems, and eigenvalue problems [10]. As such, they comprise the core component of a broad base of scientific applications in fields ranging from molecular dynamics and nuclear physics to data mining and signal processing. In the presence of large-scale data, sparse matrix computations become quite challenging as they demand massive parallelism but cannot effectively utilize compute resources. This underutilization stems from the memory-bound nature of the computations, which is not only the result of low arithmetic intensity but also irregular data access patterns. The latter factor becomes more evident in modern computer architectures where cache performance holds a greater significance within deep memory hierarchies with the accumulated cost of going farther away from the processor.

Challenges posed by large scale sparse matrix computations are not particularly well addressed by a bulk synchronous parallel (BSP) approach, which is a type of coarse-grained parallelism, and imposes a barrier synchronization at the end of each computational kernel. The two main factors that limit the performance of the BSP approaches are (i) poor cache performance that can be attributed to coarse-grain tasks which does not fit into the last level cache (LLC) and (ii) high synchronization costs that are exacerbated by load imbalances given the skewed distribution of nonzero values within the matrices. Therefore, a fundamentally new approach is needed to tackle these issues, which validate the emergence and increased use of asynchronous many-task (AMT) programming models.

OpenMP's task parallelism has been commonly used and well studied since 2013 [21] as it allows extracting parallelism via scheduling and asynchronous execution of fine-grained tasks. This model has the potential to remedy both deficiencies of the BSP model with regard to cache performance and load balancing issues.

There are other runtime systems that enable fine-grained task parallelism such as HPX [12]. HPX is an advanced runtime system and a programming API that conforms to the C++11/14/17/20 standards while supporting lightweight task scheduling to expose new levels of parallelism. It also extends the standard to the distributed case by employing a global address space, which renders efficient utilization of inter-node parallelism when combined with runtime adaptive resource management. HPX has demonstrated promising results in many projects as diverse as astrophysics simulations, n-body problems and storm surge forecasting [8, 11].

Another runtime system that adopts the AMT model is Regent [24], a programming language and compiler designed for HPC. Regent programs appear to be sequential codes with calls to *tasks*, i.e., functions eligible for parallel execution. Regent runtime system discovers implicit dataflow parallelism in the code by internally computing the task dependency graph, eliminating the need for explicit synchronization. Moreover, through its ability to schedule and run tasks on distributed machines, Regent frees programmers from low level distributed memory programming. Regent is shown to achieve performance comparable to OpenMP and MPI+X for a variety of applications [24, 27].

Recently, using the AMT model in OpenMP has been shown to offer important advantages over its BSP model in the context of sparse solvers with the DeepSparse framework [1]. DeepSparse adopts a fully integrated task-parallel approach that targets all computational steps in a sparse solver rather than a single kernel such as SpMV or SpMM. DeepSparse automatically generates and expresses the entire computation as a task dependency graph (TDG) and relies on OpenMP for the execution of this TDG. Despite the larger number of tasks that must be generated and managed, DeepSparse achieved significant improvements in terms of cache misses with little overheads through pipelined execution of tasks. DeepSparse is publicly available at [1].

Having seen the success of OpenMP's task parallelism on sparse solvers, and the lack of work on evaluation of other AMT models in this area, this paper aims to discern how OpenMP, HPX and Regent compare as well as what they offer over BSP models. Our main contributions can be summarized as follows:

- a novel task-parallel implementation of two sparse solvers with different characteristics, i.e., Lanczos and LOBPCG algorithms, using the HPX and Regent runtime systems by highlighting key factors to obtain an optimized code,
- an extensive performance evaluation of DeepSparse, HPX and Regent on multicore and manycore CPU architectures using a variety of sparse matrices from different domains,
- empirical demonstration of the significant cache miss reduction across all cache levels and the execution time improvement by up to 9.9× and 7.5× compared to highly optimized library implementations for Lanczos and LOBPCG, respectively.
- presentation of a practical rule of thumb for determining the ideal task granularity for each runtime system.

After reviewing the related work in Section 2, we discuss how we leverage the dataflow model in each framework in the context of sparse matrix computations and point out the key factors for optimized implementations in Secton 3. Section 4 describes the solvers used for benchmarking. Section 5 shows the impact of the optimizations applied, evaluates the frameworks in terms of execution time and cache performance, and provides a heuristic for choosing the task granularity.

[1] https://gitlab.msu.edu/SParTA/deepsparse

2 RELATED WORK

There are several other AMT frameworks as we try to review below. Unfortunately, we cannot examine all of them in this paper, but to the best of our knowledge, cross-examination of end-to-end sparse solver performances of some important AMT frameworks constitutes a unique aspect of our work.

PaRSEC is a framework and a runtime system that aims to manage tasks through architecture aware scheduling [5]. D-PLASMA library [6] is developed using PaRSEC, and shows that PaRSEC can improve the performance of *dense linear algebra algorithms* by expressing them as a directed acyclic graph (DAG) of tasks. However, PaRSEC provides a conservative data-flow model in both shared and distributed memory as the runtime system works on the data distribution and task graph information specified by the user [26]. It also offers limited work stealing on distributed memory as inter-node scheduling relies on remote completion notifications.

StarPU is a runtime system with a unified execution model at high level [3]. Its main goal is to facilitate the generation and execution of parallel tasks on heterogeneous architectures using multiple scheduling algorithms. Nevertheless, it requires the explicit data distribution and task creation by the developer and depends on MPI communications on distributed memory [26].

Legion runtime system extracts parallelism by dynamically identifying nested parallelism and independent tasks on account of logical (definition of objects) and physical regions (actual copies of objects) [4]. Regent compiler is essentially built on top of Legion, making it simpler to program without sacrificing performance [24].

Charm++ is a C++-based, message-driven, and portable parallel programming framework and language [13]. It can expose both task and data level parallelism through the execution of parallel processes called "chares". Charm++ is similar to HPX in that they both (i) mitigate load imbalances in a distributed system by migrating part of the data between nodes, (ii) prefer to execute the code close to where the data resides, (iii) adopt message&data-driven approach, and (iv) employ a global address space (GAS) environment.

There are several other task-based parallel programming models such as Intel Threading Building Blocks (TBB) [16], Qthreads [30] and Intel Cilk Plus [22]. What makes HPX and Regent more appealing and convenient than these models listed above is that both can employ the same task parallel approach on distributed memory systems with automatic data migration, which improves the programmability on exascale architectures.

Thoman *et. al.* [26] provide a task-focused taxonomy for HPC technologies, including HPX, Charm++, Legion, OpenMP, StarPU, Intel Cilk Plus and TBB. Kulkarni and Lumsdaine [17] theoretically compare AMT runtimes along programming model, execution model, and implementation characteristics bases. Stpiczyński [25] evaluates the performance of OpenMP, TBB and Cilk Plus and advises how to improve performance using the Belman-Ford algorithm as an example. Wang [28] provides a guideline to help programmers select an appropriate task model between Cilk, OpenMP and HPX

by drawing conclusions from six benchmarks: *Fibonacci, Knight, Pi, Sort, N-Queens, and Unbalanced-Tree-Search*. To our knowledge, our work is the first to compare the empirical performance of three AMT models; OpenMP, HPX and Regent to an optimized BSP approach within the context of sparse solvers on both manycore and multicore architectures.

3 IMPLEMENTATION AND OPTIMIZATIONS

In many large-scale scientific applications, sparse matrix computations are the most expensive kernels. As such, in all three frameworks (DeepSparse, HPX and Regent), we define tasks based on the decomposition of the input sparse matrices. Compared to a 1D (block row) partitioning, a 2D (sparse block) partitioning is known to expose a higher degree of parallelism while potentially reducing data movement [23]. Therefore, we adapt a 2D partitioning scheme where tasks are defined based on the Compressed Sparse Block (CSB) [7] representation of the sparse matrix. All three task-parallel versions start by partitioning the sparse matrix into CSB blocks, which also dictates the decomposition of all other data structures involved, such as input/output vectors and/or vector blocks.

Consider the simple code snippet in Listing 1, which we will utilize to illustrate the salient technical details of the Lanczos and LOBPCG algorithms and our optimization ideas. This code snippet includes three of the most common sparse solver kernels. Suppose that the input matrix A is of size m×m, block size is denoted by b, and the vector width by $n \geq 1$:

- SpMM kernel is partitioned into tasks where each operates on a b×b block of the matrix A, and b×n block of X and Y as shown in Fig. 1. Tasks are created only for non-empty blocks.
- In the second kernel, which is a linear combination operation and will be referred to as the XY kernel, each task operates on a b×n block of Y, the entire Z matrix (a single n×n block) and b×n block of Q.
- The third one is the inner product kernel, which will be referred to as XTY kernel, spawns tasks as shown in Fig. 2, computing partial results from the multiplication of n×b block of Y^T and b×n block of Q. A final task reduces the partial results.

Listing 1: An example pseudocode

```
1 || SpMM(A, X, Y, m, n); // A*X = Y
2 || cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
  ||     m, n, n, 1.0, Y, n, Z, n, 0, Q, n); // Y*Z = Q
3 || cblas_dgemm(CblasRowMajor, CblasTrans, CblasNoTrans, n,
  ||     n, m, 1.0, Y, n, Q, n, 0, P, n); // Y'*Q = P
```

Notice that there are two different ways to implement the task-parallel SpMM kernel: (i) to launch all SpMM tasks asynchronously and keep a partial output vector on each thread/core, or (ii) to setup dependencies between tasks to ensure no two threads/cores access the same portion of the output vector. With the latter option, the maximum degree of concurrency in SpMM is still equal to the number of blocks in the output vector. Therefore, as long as the number of blocks in the output vector is greater than the number of threads, it avoids the memory cost and processing overhead of the reduction required for the first option. As shown in Section 5, the degree of parallelism yielded by the optimal block sizes exceeds the thread count for DeepSparse and HPX. Experimental results on

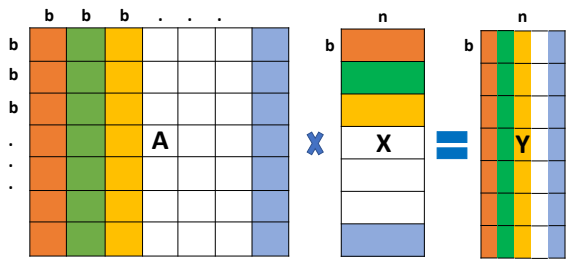


Figure 1: Task partitioning of the SpMM kernel.

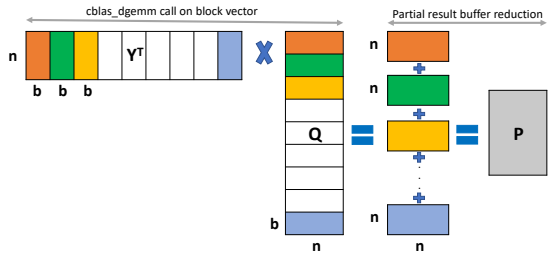


Figure 2: Task partitioning of the inner product kernel.

Figure 3: Task graph for the pseudocode in listing 1.

Regent pointed out that the dependency based solution achieves better performance than the buffer based solution even when there is not enough tasks to keep all threads occupied. As a result, we adopt the latter approach in all three frameworks.

With this crucial detail in mind, for $b = m/3$, the computational DAG for Listing 1 is shown in Fig. 3. Regardless of the underlying representation of the DAG by the runtime system, the correctness of computation depends on a valid execution order with respect to the DAG topology, whereas the performance relies on exploiting the maximum parallelism available while determining a schedule that reduces data movement. Next, we discuss how each framework accomplishes these conflicting goals in detail.

3.1 DeepSparse

DeepSparse [1] consists of two major components:

Primitive Conversion Unit (PCU). PCU essentially provides a high level front-end scientific application development. Task Identifier (TI), the first subcomponent of PCU, parses GraphBLAS [14] and BLAS/LAPACK [2, 19] function calls, which are similar to those in Listing 1, expressed through DeepSparse API. The output of TI is a dependency graph at the function call level whereas tasks must be created at a much finer granularity to expose parallelism and

to allow control over data movement. Task Dependency Graph Generator (TDGG), the second subcomponent, accomplishes that by going over the input/output data information generated by TI for each function call and decomposing corresponding data structures. TDGG then generates the dependencies between individual fine-granularity tasks by examining the function call dependencies determined by TI, while taking into consideration the non-zero pattern of the sparse matrix.

Task Executor. DeepSparse provides the OpenMP task-based implementation of all computational kernels it supports. As such, Task Executor picks each node from the output of TDGG one by one and extracts the corresponding task information. Based on the kernel id, partition id of the input/output data structures and other required parameters, Task Executor calls the corresponding function found in the DeepSparse library, effectively spawning an OpenMP task. In DeepSparse, the master thread spawns all OpenMP tasks in a depth-first topological order, and relies on OpenMP's default task scheduling algorithms for execution of these tasks.

DeepSparse will explicitly generate the task dependency graph for each algorithm and input sparse matrix combination but the overhead of this graph generation is negligible for two reasons. First, each vertex in the graph corresponds to a task operating on a large set of data in the original problem. Secondly, sparse solvers are typically iterative, and the same task dependency graph is used for several iterations. Therefore, such overhead would be insignificant in comparison to the actual problem size.

3.2 HPX

HPX attains asynchronous parallelism through asynchronous function execution (*async*) and *future* instances: Asynchronous execution of a function will result in scheduling of it as a new HPX thread and return a new *future* instance as HPX threads in the queue are dynamically managed by the runtime system [12]. A *dataflow* object, on the other hand, triggers a predefined function when a set of futures become ready. Combining a *dataflow* object with asynchronous execution provides a powerful mechanism for maintaining data dependencies and constructing an execution tree. We show HPX's dataflow model in Listing 2, which implements the pseudocode in Listing 1.

Listing 2: HPX code for the pseudocode in Listing 1

```
1  std::vector<hpx::shared_future<void>> Y(np);
2  std::vector<hpx::shared_future<void>> Q(np);
3  std::vector<hpx::shared_future<void>> P_prtl_ftr(np);
4  hpx::shared_future<void> P_rdcd_ftr;
5  // np (number of partitions) = ceil(m/blocksize)
6  for(int i = 0; i != np; ++i)
7      Y_ftr[i] = hpx::make_ready_future();
8  // to unwrap futures passed to functions
9  auto OpSpMM = hpx::util::unwrapping(&SpMM);
10 auto OpDGEMV = hpx::util::unwrapping(&f_dgemm);
11 auto OpDGEMV_T = hpx::util::unwrapping(&f_dgemm_t);
12 auto OpRed = hpx::util::unwrapping(&reduce_buf);
13 // Y = A * X
14 for(i = 0; i != np; ++i)
15     for(int j = 0; j != np; ++j)
16         if(A[i * np + j].nnz > 0)
17             Y_ftr[i] = hpx::dataflow(hpx::launch::async
                   , OpSpMM, Y_ftr[i], A, X, Y, i, j);
18 // Q = Y * Z
19 for(i = 0; i != np; ++i)
20     Q_ftr[i] = hpx::dataflow(hpx::launch::async,
           OpDGEMV, Y_ftr[i], Y, Z, Q, i);
21 // P = Y' * Q
22 for(i = 0; i != np; ++i)
23     P_prtl_ftr[i] = dataflow(hpx::launch::async,
           OpDGEMV_T, Y_ftr[i], Q_ftr[i], Y, Q, Pbuf, i);
24 P_rdcd_ftr = dataflow(hpx::launch::async, OpRed,
       P_prtl_ftr, Pbuf, P);
```

Each *future* defined (line 1-4) indicates whether a task of *void* is executed, and thus the futures are of *void* type as well. We have four functions that are not given in the code snippet: (i) `SpMM` for a single block of the matrix, (ii) `f_dgemm` and (iii) `f_dgemm_t` that are wrapper functions for `cblass_dgemm` calls to execute XY and XTY tasks, and (iv) `reduce_buffer` to accumulate partial results of P. We define a proxy function for each of these four functions (line 9-12) whose sole purpose is to unwrap the futures when ready before passing to the actual function. That enables programmers to write functions with the same ease as the equivalent sequential code.

We launch an asynchronous SpMM task (line 17) that operates on sparse matrix block $A_{i,j}$ and block X_j to update block Y_i. The *dataflow* returns a *future* to the result of the SpMM task, which is assigned to $Y\_ftr[i]$ (line 17). Since the use of a buffer for the output Y is avoided through a dependency based approach, this *future* depends on itself. To compute block Q_i within the XY kernel, the computation of Y_i should be finished, which is indicated by the readiness of $Y\_ftr[i]$ (line 20). Likewise, in the XTY kernel, the task responsible for ith buffer of P will be triggered when both $Y\_ftr[i]$ and $Q\_ftr[i]$ are ready (line 23). Note that checking $Y\_ftr[i]$ is redundant there as $Q\_ftr[i]$ already depends on $Y\_ftr[i]$. HPX allows a vector of futures being provided as a parameter in *dataflow* to set the dependencies; we use this feature for P: `reduce_buffer` will be invoked once every *future* in vector P_prtl_ftr is ready (line 24). Last but not least, we skip the empty matrix blocks (line 16) since they do not contribute to the output, in order to lighten the burden on the runtime system and improve the performance.

3.3 Regent

Regent is a language, runtime system, and a compiler that exerts implicit dataflow parallelism through two key abstractions: *tasks* and *logical regions* (or simply *regions*) [24]. *Tasks* are functions that are marked as eligible for parallel execution by the programmer and *regions* are collections of structured objects that can be recursively partitioned to render parallel execution possible. *Tasks* in Regent are forced to describe how they interact with each *region* they take as argument by declaring *privileges*: *read, write, read/write* or *reduce*, which in return allows Regent to discover parallelism in seemingly sequential code. To illustrate this, in Listing 3, we provide the Regent code of the pseudocode presented in Listing 1.

Listing 3: Regent code for the pseudocode in Listing 1

```
1  fspace csb_entry{
2      {rloc, cloc}: uint16, val: double,
3  }
4  task SpMM(rA: region(ispace(int1d), csb_entry),
5            rX: region(ispace(int1d), double),
```

```
6  ||             rY: region(ispace(int1d), double),
7  ||             s: int, e: int)
8  || where reads(rA, rX), reads writes(rY) do
9  ||     -- ... (SpMM implementation)
10 || end
11 || -- ... (other tasks)
12 || task main()
13 ||     -- ... np (num partitions) = ceil(m/blksize)
14 ||     var sparse_matrix_is = ispace(int1d, nnz)
15 ||     var vector_block_is = ispace(int1d, m * n)
16 ||     var Alr = region(sparse_matrix_is, csb_entry)
17 ||     var Xlr = region(vector_block_is, double)
18 ||     -- ... (other region defs, Alr & blkptrs init)
19 ||     var part = ispace(int1d, np)
20 ||     var Xlp = partition(equal, Xlr, part)
21 ||     -- ... (Y and Q partitionings, etc.)
22 ||     -- Y = A * X
23 ||     for i = 0, np do
24 ||         for j = 0, np do
25 ||             if blkptrs[i*np+j] < blkptrs[i*np+j+1] then
26 ||                 SpMM(Alr, Xlp[j], Ylp[i], blkptrs[i*np+
   ||                     j], blkptrs[i*np+j+1])
27 ||             end
28 ||         end
29 ||     end
30 ||     -- Q = Y * Z
31 ||     __demand(__index_launch)
32 ||     for i = 0, np do
33 ||         f_dgemm(Ylp[i], Zlr, Qlp[i], m, n, blksize, i)
34 ||     end
35 ||     -- P = Y' * Q
36 ||     __demand(__index_launch)
37 ||     for i = 0, np do
38 ||         f_dgemm_t(Ylp[i], Qlp[i], Plr, m, n, blksize,i)
39 ||     end
```

A *region* (line 16-17) is the cross-product between an *index space* (lines 14-15) and a field space (lines 1-3, like *structs* in C). A *region* can be disjointly partitioned into *subregions* with a simple use of *partition* function (line 20). An *index space* (line 19) must be provided to name the respective *subregions*. CSB format requires each block (*subregion* in Regent) to dynamically allocate memory based on their number of non-zero entries, but Regent does not allow such allocation. As a workaround, we create a *region* that contains all entries (line 14&16) in advance where the entries falling into the same block are kept contiguous to better utilize the cache.

SpMM task implementation is similar to that of HPX, but here, rather than passing the pointer to the entire X and Y data and making sure we access and update the appropriate portion, we directly pass the corresponding *subregion* (i.e., X_j and Y_i in line 26). By analyzing the *privileges* defined on each passed *region/subregion* (line 8), the runtime system extracts parallelism for SpMM tasks as shown in Fig. 3. Moreover, the index launch represents a loop of tasks that are non-interfering and is a compiler-level optimization. This concept helps the Regent to launch those tasks without any dependency checks. It is not required, but the programmer can use it to ensure the implementation is sound (line 31&36), for `f_dgemm` and `f_dgemm_t` tasks (line 33&38), for instance. Although we do not share it due to space constraints, `f_dgemm` declares *read privilege* on Y_i and Z and *write privilege* on Q_i. Slightly different from HPX, `f_dgemm_t` declares *reduce privilege* on P, which is convenient in contrast to using *reduce* on Y_i for SpMM since P is a much smaller matrix with a lower overhead.

4 BENCHMARK APPLICATIONS

We evaluate the performance of the task parallel frameworks on two popular eigensolvers with different characteristics: Lanczos [18], which is SpMV based, and Locally Optimal Block Preconditioned Conjugate Gradient (LOBPCG) [15], which is SpMM based.

Lanczos computes the k algebraically largest (or smallest) eigenvalues of a symmetric matrix by building the Krylov subspace Q, a block of orthogonal vectors. As $k \ll m$, it is a relatively simple algorithm (see Alg. 1) where SpMV is the main kernel. We give the

Algorithm 1: Lanczos Algorithm

b = *initial vector*, $Q_0 = b/||b||_2$
for $i = 1$ *to* k **do**
  $z = AQ_{i-1}$
  $\alpha_i = {Q_{i-1}}^T z$
  $q = [Q_0, \ldots, Q_{i-1}]$
  $z = z - qq^T z$
  $\beta_i = ||z||$
  $Q_i = z/\beta_i$
end

pseudocode for the LOBPCG solver in Alg. 2. It involves kernels with much higher arithmetic intensities (such as SpMM and several level-3 BLAS calls) compared to Lanczos. The total memory needed for block vectors Ψ, R, Q and others can easily exceed the space matrix $\widehat{H}$ takes up. Figure 4 [1] shows a sample task graph for LOBPCG for a toy problem, which demonstrates the difficulty of creating a schedule to attain an efficient execution.

Algorithm 2: LOBPCG Algorithm solving $\hat{H}\Psi = M\Psi$

Input: $\hat{H}$, matrix of dimensions $m \times m$
Input: Ψ_0, a block of vectors of dimensions of $m \times n$
Output: Ψ and M such that $\|\hat{H}\Psi - \Psi M\|_F < \epsilon$, and $\Psi^T\Psi = I_n$

Orthonormalize the columns of Ψ_0
$Q_0 = 0$
for i = *0, 1, …, until convergence* **do**
  $M_i = \Psi_i^T \hat{H} \Psi_i$
  $R_i = \hat{H}\Psi_i - \Psi_i M_i$
  Apply the Rayleigh–Ritz procedure on span$\{\Psi_i, R_i, Q_i\}$
  $\Psi_{i+1} = \underset{V \in \text{span}\{\Psi_i . R_i, Q_i\},\ V^TV = I_n}{\text{argmin}} \text{trace}(V^T \hat{H} V)$
  $Q_{i+1} = \Psi_{i+1} - \Psi_i$
end
$\Psi = \Psi_{i+1}$

DeepSparse uses the DAG constructed for a single iteration with barriers in between because the number of iterations until convergence is unknown. HPX and Regent form the DAG internally on-the-fly, so they might proceed between iterations without a barrier, but this is hard to achieve in practice due to the convergence check at each iteration. Critical path lengths in Lanczos and LOBPCG are 5 and 29, respectively. Number of tasks depends on block and matrix sizes, and ranges from 56 to 6,570,446 per iteration.

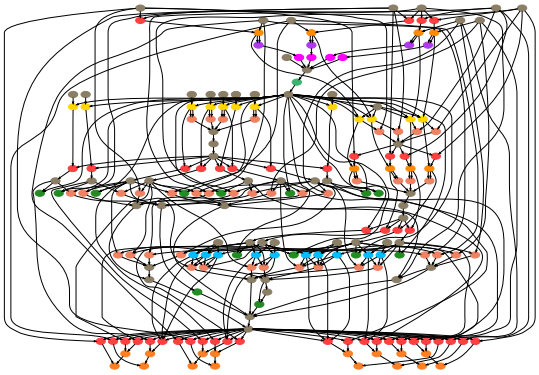


Figure 4: A sample task graph of the LOBPCG algorithm [1].

5 PERFORMANCE EVALUATION

Performance evaluations were carried out on two systems, an Intel Broadwell-based multicore cluster and an AMD EPYC-based manycore cluster. Each Broadwell cluster node has two 14-core Intel Xeon E5-2680v4 Broadwell 2.4 GHz processors with a 64 KB L1 cache (32 KB instruction, 32 KB data) and a 256 KB L2 cache per core, in addition to a 35 MB shared L3 cache. Each EPYC cluster node has two 64-core AMD EPYC 7H12 2.6 GHz processors. Each EPYC core has a 64 KB L1 cache (32 KB instruction, 32 KB data), a 512 KB L2 cache and 16 MB L3 cache is shared between every four cores.

We compare the performance of DeepSparse, HPX and Regent implementations with two library-based BSP versions: (i) **libcsr** is the implementation of the benchmark solvers using thread-parallel Intel MKL Library calls (including SpMV/SpMM) with CSR storage of the sparse matrix, (ii) **libcsb** also uses Intel MKL calls, but with the matrix being stored in the CSB format. We use MKL calls within each task of the AMT models whenever possible for a fair comparison. Finally, we do not claim to have the best possible implementation for all solver versions, although we did our best to optimize them.

We utilized an entire node for our runs on both clusters, i.e., 28 cores on Broadwell and 128 cores on EPYC. For DeepSparse, libcsr and libcsb, we bind OpenMP threads to cores. For HPX, the number of OS threads spawned through *–hpx:threads* argument is the same as the number of cores. For Regent, the number of cores to be used for executing the application tasks is specified using *-ll:cpu*, and *-ll:util* determines the number of cores allocated to the runtime system. Empirically we found that *-ll:cpu 24 -ll:util 4* on Broadwell and *-ll:cpu 110 -ll:util 18* on EPYC yield (near-)optimal results on both benchmark applications.

We selected 14 matrices with varying sizes, sparsity patterns, and domains from the SuiteSparse Matrix Collection in addition to the Nm7 matrix, which is from a nuclear shell model code (see Tab. 1) [9]. Since both solvers require the input matrix to be symmetric, the matrices that are not symmetric (shown in bold in Tab. 1) are made so by copying the transpose of the lower triangular part over the upper triangular part: $A_{new} = L + L^T - D$. Matrices shown in italics were originally binary matrix, and hence were filled with random values without breaking the symmetry.

Table 1: Matrices used in our evaluation.

Matrix	#Rows	#Non-zeros
inline1	503,712	36,816,170
dielFilterV3real	1,102,824	89,306,020
Flan_1565	1,564,794	117,406,044
HV15R	2,017,169	281,419,743
Bump_2911	2,911,419	127,729,899
Queen4147	4,147,110	329,499,284
Nm7	4,985,422	647,663,919
nlpkkt160	8,345,600	229,518,112
nlpkkt200	16,240,000	448,225,632
nlpkkt240	27,993,600	774,472,352
it-2004	41,291,594	1,120,355,761
twitter7	41,652,230	868,012,304
sk-2005	50,636,154	1,909,906,755
webbase-2001	118,142,155	1,013,570,040
mawi_201512020130	128,568,730	270,234,840

All presented performance data come solely from the solver iteration parts, excluding any I/O, initialization and setup parts. Performance data were averaged over multiple iterations (20 for Lanczos, 10 for LOBPCG). For the last two matrices, number of iterations was 10 for Lanczos and 5 for LOBPCG due to their size. Our comparison criteria are L1, L2, LLC (L3) misses (unavailable on EPYC due to root access requirement) and execution times for both solvers and architectures. Cache misses were normalized with respect to that of libcsr, and speedups were calculated over libcsr. Cache miss data was obtained using "perf stat" command.

We note that performance data from where a single socket is used on both architectures (14 cores on Broadwell and 64 cores on EPYC) are similar to the case presented here, where both sockets are used. The only difference is, on EPYC, the task parallel frameworks seem to be affected less by the NUMA-related performance issues as their speedup numbers improve going from a single socket to the entire node. The single socket results are not presented due to space constraints.

The impact of the degree of parallelism is measured by conducting tests for several different block sizes. However, for a given block size, all AMT models are essentially presented the same DAG, i.e., the available degree of parallelism are identical across all runtimes. Since all runtimes are executing the same DAG, we believe their performance differences are due to the different scheduling algorithms employed as this directly impacts thread idling and cache utilization.

Scheduling policies of the runtimes studied here are either opaque or are not well documented, making a detailed comparison of the impact of the different scheduling policies difficult. For instance, OpenMP's task scheduling is left to the implementation and not well documented; task priorities are only ignorable hints. For HPX, NUMA-aware scheduling made a big difference in cache utilization and performance, but their scheduling algorithms are not well documented either. Regent gives task mapping options, which is, however, mainly recommended for heterogeneous computing.

In Sect. 5.2 and 5.3, we share the results from the experiments where the optimal block size is employed, which depends on the solver, architecture, runtime system and matrix type. Then in Sec. 5.4, we present a practical rule of thumb for determining the ideal task

granularity by choosing the ideal block size for each runtime system.

5.1 The Effect of Optimizations

We tried to optimize the code at the same level for each task parallel system using the techniques discussed below. Although each framework benefits from all optimizations, due to space constraints we only share the results of every optimization for a certain framework, solver and architecture combination where the impact is the most evident. In all following optimization plots, compared implementations incorporate all other optimization techniques so that the only variable is the selected optimization at hand.

First-Touch Placement Policy. This policy refers to allocation of a data page in the memory closest to the thread accessing it first. When a single thread initializes all data structures, the data ends up residing in the memory of a single NUMA node, which increases access times, consequently hurting the performance. Leveraging this policy would simply require the initialization of vector blocks and the sparse matrix in parallel in the case of sparse solvers. The 128 cores on an EPYC node are internally organized into 8 NUMA subregions, 4 per socket. As shown in Fig. 5 for DeepSparse, this optimization is vital for good performance (up to 2.5 fold) for the small and mid-sized matrices on the EPYC system. We also utilize it in the BSP versions (libcsr and libcsb) for a fair comparison.

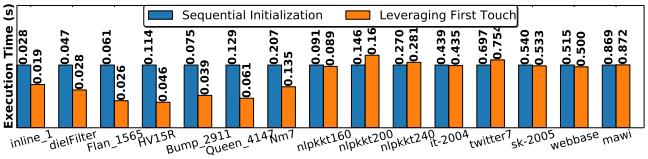


Figure 5: Execution time of DeepSparse for Lanczos on EPYC wrt first-touch policy.

Skipping Empty Tasks. Depending on the sparsity pattern of a matrix and chosen CSB block size, there are empty blocks which do not contribute to the result of SpMV/SpMM operation. Spawning tasks for those blocks creates scheduling overheads for the runtime system. Figure 6 shows that skipping such tasks may speed up the execution time by 30% on average, albeit not as effective on some matrices. We attribute the lack of improvement in those cases to the fact that both implementations use the optimal block size for each matrix, which in general does not yield too many empty blocks.

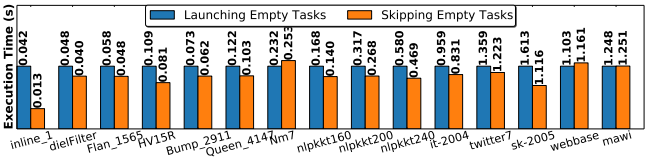


Figure 6: Execution time of HPX for Lanczos on Broadwell wrt skipping empty tasks.

Eliminating Reduction for SpMV/SpMM Output. As discussed in Section 3, the dependency based approach is used on all frameworks to eliminate the reduction overheads. In Fig. 7, we show the empirical advantage of this decision on Regent for the LOBPCG algorithm. We observe that the reduce-based approach yields an extremely poor performance on large matrices, and we believe this is due to large buffers that need to be allocated by each core. Furthermore, Regent runtime system manages the reduce operation internally (recall that one of the *region* privileges was *reduce*). Given the problematic scaling behavior of Regent with regard to the number of tasks (discussed in Sect. 5.4), poor performance of the reduce based approach on Regent is not a surprise.

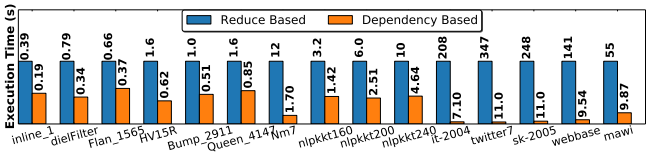


Figure 7: Execution time of Regent for LOBPCG on Broadwell wrt two SpMV/SpMM computation approaches.

Other Attempts. We also attempted several framework-specific optimizations. For instance, HPX allows passing scheduling hints to their scheduler in an effort to create executors that target a specific NUMA domain or to pin HPX threads to a particular core. We employed scheduling hints to achieve a locality-aware scheduling for both solvers. This technique improved HPX's both Lanczos and LOBPCG performance significantly on EPYC, where there exist 8 NUMA domains (16 cores each). Regent provides a technique called *dynamic tracing* [20] to reduce the task management overhead for iterative solvers. This technique relies on capturing the task graph in the first iteration and replaying it for subsequent iterations through memoization to avoid the dependence analysis. However, this last attempt did not yield any significant performance improvement.

5.2 Lanczos Evaluation

Lanczos algorithm is a relatively simple algorithm in the sense that it has much fewer types and number of tasks than LOBPCG because it essentially consists of one SpMV and one inner product kernel at each iteration. As such, scheduling decisions are simpler and there are fewer data reuse opportunities. Consequently, we observe that the task parallel systems often lead to little to no improvement in terms of cache misses. This can be seen in Fig. 8 where the cache misses comparison on EPYC for different Lanczos versions is shown. No framework achieves consistent reduction in cache misses on L1 level. Moreover, the improvements on L2 level can be attributed to the matrices being stored in the CSB format since libcsb, the other BSP version, yields similar improvements.

Most importantly, all three task-parallel versions give decent speedups on both architectures as shown in Fig. 9. On Broadwell (the top subplot), DeepSparse, HPX and Regent achieve up to 2.3×, 4.3× and 2.0× improvement although on average, the speedup achieved is somewhat modest (1.5×, 2.2× and 1.1×, respectively). Task parallel versions perform better when we go from a multicore (Broadwell) to a manycore (EPYC) architecture. DeepSparse

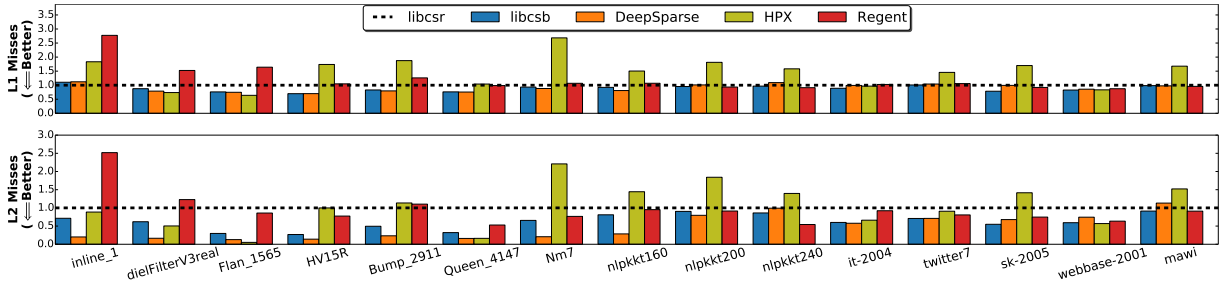


Figure 8: L1 and L2 misses of different Lanczos versions on EPYC normalized wrt libcsr.

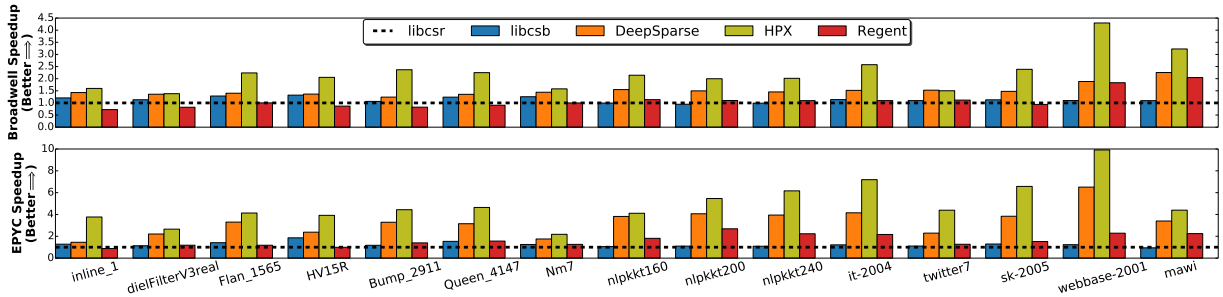


Figure 9: Speedup of different Lanczos versions on Broadwell (top) and EPYC (bottom) over libcsr.

achieves as high as 6.5× speedup, HPX up to 9.9× speedup and Regent up to 2.7× speedup. On average, DeepSparse, HPX and Regent achieve 3.3×, 4.9× and 1.6× speedup, the majority of which comes from the large matrices.

We attribute the speedups observed across the board to the increased parallelism with tasking and reduced synchronization overheads. In fact, a further investigation of the execution flow graph of tasks in Fig. 10 shows that the manycore architecture provides a greater level of parallelism for the task parallel systems to fill the gap resulting from load imbalances of SpMV with the succeeding tasks. Therefore, each iteration is completed not long after the execution of the last SpMV task on EPYC, providing the AMT approaches with a greater success.

5.3 LOBPCG Evaluation

LOBPCG is a complex algorithm with several different kernel types; its task graph may result in millions of tasks depending on the block size. In LOBPCG, vector blocks have only 8-16 columns, hence there is no tiling in the column dimension. Block sizes refer to the number of rows in each chunk, they ranged from 1K to 16M. Since LOBPCG requires several vector operations consecutively, there are plenty of data reuse opportunities for vector chunks.

In Fig. 11, we show the cache misses and speedup comparison on Broadwell for all five LOBPCG versions. The libcsr and libcsb versions achieve similar number of cache misses, while the task-parallel versions demonstrate an outstanding cache performance:

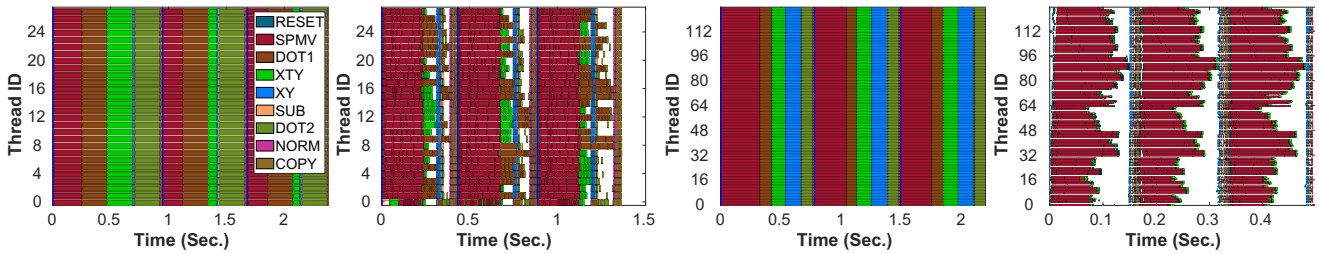
- DeepSparse yields a consistent improvement throughout all cache levels: It achieves 3.0× - 10.4× fewer L1 misses, 3.8× - 12.0× fewer L2 misses and 1.4× - 4.7× fewer L3 cache misses than libcsr.
- HPX's cache performance is on par with DeepSparse: It achieves 2.8× - 13.7× fewer L1 misses, 3.7× - 13.1× fewer L2 misses and 1.4× - 5.2× fewer L3 cache misses than libcsr.
- Regent has competitive cache utilization too, as it produces 4.3× - 9.6× fewer L1 misses, 4.0× - 12.3× fewer L2 misses and 1.6× - 6.2× fewer L3 cache misses compared to libcsr.

As the top subplot of Fig. 12 shows that on Broadwell, even with the implicit task graph creation and execution overheads of the runtime systems, this significant reduction in cache misses leads to 1.8× - 3.0× speedup for DeepSparse, 1.5× - 4.4× speedup for HPX and 0.8× - 1.9× speedup for Regent (slowdown occurring on a few smaller matrices) over the execution times of libcsr. Given the highly complex underlying task dependency graph of LOBPCG and abundant data re-use opportunities available, we attribute these improvements to the pipelined execution of tasks which belong to different computational kernels but use the same data structures.

AMT models continue their superior performance in terms of execution time on EPYC as shown in the bottom subplot of Fig. 12. As a matter of fact, DeepSparse and HPX improve their performance further compared to Broadwell: DeepSparse achieves 1.2× - 5.5× speedups and HPX achieves 1.7× - 7.5× speedups over libcsr. However, Regent demonstrate a similar performance on this architecture achieving 0.8× - 2.3× speedup where the performance degradation is again being observed on the smaller matrices.

AMT models achieve up to 99% L1 hits for LOBPCG, compared to the 85-90% hit ratio of loop-parallel versions. Considering AMT models' outstanding cache miss performance as well, we conclude that cache utilization is an important factor with LOBPCG due to data reuse opportunities. The improved performance observed in HPX by switching to NUMA-aware scheduling, which is around 50%, also supports this view.

The pipelined execution of tasks in DeepSparse and HPX in comparison to libcsr can be observed in Fig. 13. The performance on XTY kernel accounts for the main difference in timing (see Fig. 13a & Fig. 13b). Data parallel execution of this kernel in the



(a) libcsr on Broadwell (b) DeepSparse on Broadwell (c) libcsr on EPYC (d) HPX on EPYC

Figure 10: Execution flow graph of nlpkkt240 from first three iterations of Lanczos for different versions and architectures.

BSP model considerably hurts the performance, which seems to be avoided in task parallel execution to a great extent through the re-use of involved matrix blocks in kernels such as XY or SpMM after the execution of XTY tasks. There also exist characteristic differences within the task parallel models as seen in Fig. 13c & Fig. 13d: Both DeepSparse and HPX give the best result on the nlpkkt240 matrix with 256K CSB block size whereas their execution flow graphs do not show much resemblance to each other. HPX in general seems to place less value on prioritization of the tasks that are launched earlier, which consequently produces a more shuffled graph where the overlap in each kernel's start and finish time increases. Regardless of that difference, their execution times are similar in this example ($\approx 3.0sec$).

5.4 Block Size Selection

The CSB block size has a significant effect on the performance of task parallel models as the factors such as task granularity, degree of parallelism, and scheduling overhead of the runtime systems are directly shaped by the block size for a given matrix and solver type. This is because we use this block size as a uniform partitioning factor whether it is for a 2D (e.g., SpMV and SpMM) or 1D (e.g., all vector operations) kernel. Therefore, by the "block count", we simply mean the number of tiles/blocks in each dimension, which is determined by the row count of matrices/vectors and the CSB block size. Choosing a small block size creates a large number of small tasks, which is preferable on a parallel architecture, but the large number of tasks may lead to significant scheduling overheads. Increasing the block size reduces such overheads, but this may then lead to increased thread idle times and load imbalances. Therefore, finding the sweet spot between these two extremes is important.

We found the optimal block size, which differs for each matrix, solver architecture, and runtime system combination by trying a variety of numbers from 2^{10} to 2^{24}. This brute-force search may not always be practical. Thus, analyzing the data further, we have come to notice that the optimal block size would always yield a block count between 8 and 511 regardless of the case. As such, picking the optimal one boils down to the comparison of the six block sizes, the ones that result in 8 to 15, 16 to 31, 32 to 63, 64 to 127, 128 to 255 or 256 to 511 block counts.

We show the performance profiles in Fig. 14 to compare these six block counts for each runtime system and architecture. Note that our findings on LOBPCG solver match the ones from Lanczos solver. So, to avoid redundancy, we only share LOBPCG results here. In the performance profiles, we plot the percentages of the instances in which a block count yields an execution time on a matrix that is no longer than τ times the best execution time found by any block count for that matrix. Therefore, the higher a profile at a given τ, the better a heuristic is. We observe the following:

- For DeepSparse, 32-63 block count is the best option and it is always within 1.15× the best option on Broadwell. On EPYC, 64-127 block count have the top spot and 32-63 block count provides a comparable performance.
- For HPX, 64-127 block count gives the optimal performance in general on both Broadwell and EPYC. However, 32-63 and 128-255 block count configurations perform similarly well regardless of the architecture.
- Regent prefers more coarse-grained tasks as 16-31 block count performs the best on both architectures. Considering that bottom three spots belong to highest three options suggests that Regent has scaling issues with regard to creation or scheduling of large number of tasks. In fact, going beyond 64 block count can cause 5× - 10× slowdowns although we cannot see it here as τ is shown for between 1.0 and 2.0 for all systems.

As a practical rule of thumb, we can say that 32-63 block count on Broadwell and 64-127 block count on EPYC for DeepSparse and HPX are good choices. Even though we have slightly less than a task per thread per kernel with the 64-127 block count, there are many kernels in LOBPCG to be executed in parallel. In fact, the execution flow graphs verify that this kernel-level parallelism might be just enough to keep all threads occupied by exposing more than a task per thread.

We know from the speedup plots that DeepSparse and HPX are the best two versions by far. Taking into account these block counts, they achieve high performance by over-decomposing the work to yield more than one task per thread for load balancing purposes while limiting that task to thread ratio to avoid scheduling overhead.

Tuning the block size is very important for best results. However, this is a complicated choice that depends on the specific problem, architecture and compiler. We note that even for the easier-to-characterize dense linear algebra kernels, auto-tuning is necessary (e.g., ATLAS library [29]). Hence, for sparse solvers this is a very difficult problem. Nevertheless, we tried to illustrate the trade-offs and give some insights which we hope could be helpful to others.

6 CONCLUSION

Several AMT frameworks emerged as we move towards exascale, and there is a lack of comparative studies, by third party users in particular. We believe a fair evaluation on various application domains would benefit readers in helping them make a well-informed decision in preparing for exascale. This is precisely the motivation

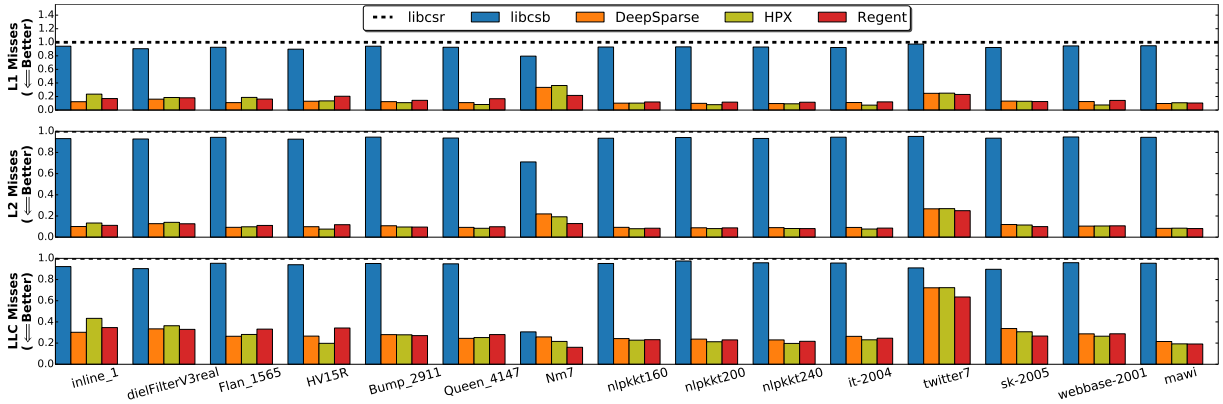


Figure 11: L1, L2 and LLC (L3) misses of different LOBPCG versions on Broadwell normalized wrt libcsr.

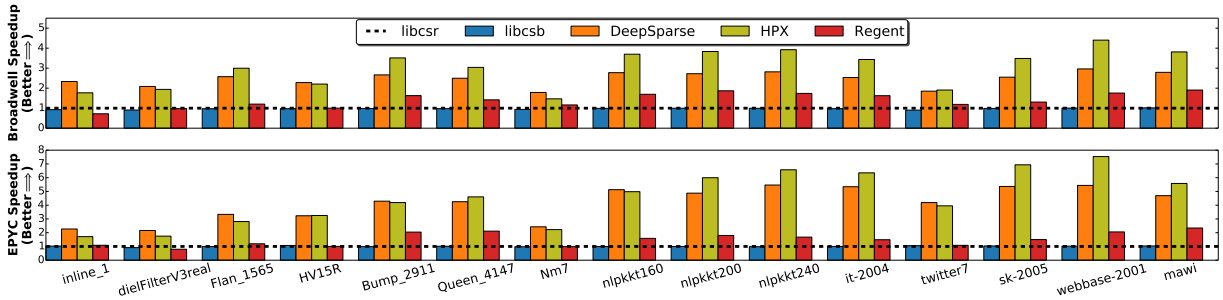
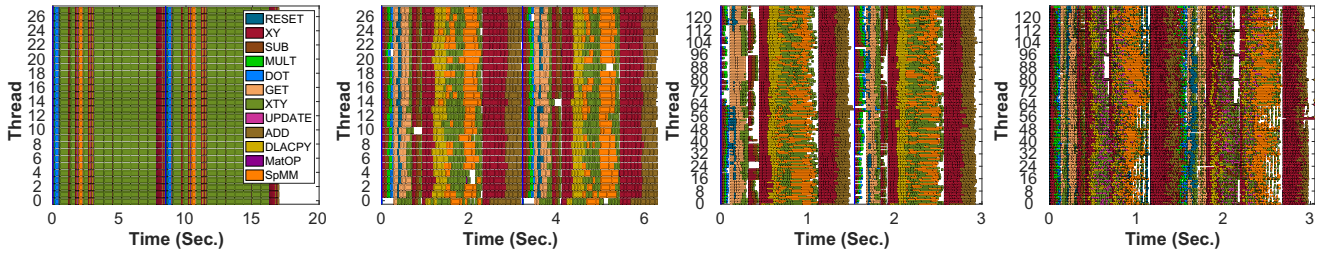


Figure 12: Speedup of different LOBPCG versions on Broadwell (top) and EPYC (bottom) over libcsr.



(a) libcsr on Broadwell **(b) DeepSparse on Broadwell** **(c) DeepSparse on EPYC** **(d) HPX on EPYC**

Figure 13: Execution flow graph of nlpkkt240 from two iterations of LOBPCG for different versions and architectures.

for our evaluation of three runtime systems in the context of sparse solvers. To our knowledge, this is the first such comparative work.

We introduced optimized implementations of LOBPCG and Lanczos eigensolvers using the task-parallel paradigm using three runtime systems: OpenMP (through the DeepSparse framework), HPX and Regent. We show that these task-parallel systems achieve significantly fewer cache misses across different cache layers for LOBPCG, a fairly complex solver. They also provide promising improvements in the execution time over a traditional optimized BSP implementation for both solvers on two different architectures: Broadwell, an Intel-based multicore system, and EPYC, an AMD-based manycore system. Moreover, they allow achieving such performance improvements without sacrificing the ease of high-level programming.

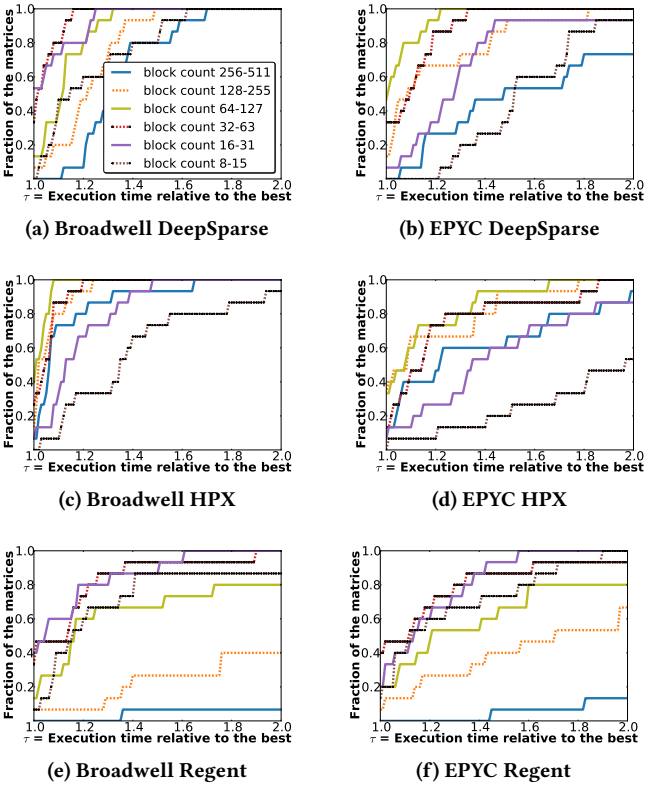
We conclude that OpenMP tasking and HPX are setting themselves apart from others. OpenMP's great performance is commendable, but so is HPX's because it generates the DAG itself as it goes along and is extensible to distributed memory architectures. Future work will be in the direction of testing HPX in a distributed memory environment using large-scale sparse solvers and graph analytics kernels, and comparing these to hybrid MPI+OpenMP solutions.

ACKNOWLEDGMENTS

This work was in part supported by the NSF under awards CCF-1822932, OAC-1845208, and CCF-1919021, as well as the US Department of Energy, Office of Science under the award DE-SC0018083 (NUCLEI SciDAC-4 Collaboration). Computational resources were provided by the High Performance Computing Center (HPCC) at Michigan State University.

REFERENCES

[1] Md Afibuzzaman, Fazlay Rabbi, M Yusuf Özkaya, Hasan Metin Aktulga, and Ümit V Çatalyürek. 2019. DeepSparse: A Task-Parallel Framework for Sparse-Solvers on Deep Memory Architectures. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 373–382.



(a) Broadwell DeepSparse (b) EPYC DeepSparse

(c) Broadwell HPX (d) EPYC HPX

(e) Broadwell Regent (f) EPYC Regent

Figure 14: The relative performance of block counts for DeepSparse, HPX and Regent on LOBPCG solver.

[2] Edward Anderson, Zhaojun Bai, Christian Bischof, Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, S Hammerling, Alan McKenney, et al. 1999. LAPACK Users' guide, vol. 9. *Society for Industrial Mathematics* 39 (1999).

[3] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2009. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Euro-Par - 15th International Conference on Parallel Processing (Lecture Notes in Computer Science, Vol. 5704)*. Springer, Delft, The Netherlands, 863–874. https://doi.org/10.1007/978-3-642-03869-3_80

[4] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing locality and independence with logical regions. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11.

[5] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J Dongarra. 2013. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering* 15, 6 (2013), 36–45.

[6] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pitior Luszczek, and Jack Dongarra. 2012. *Dense linear algebra on distributed heterogeneous hardware with a symbolic DAG approach*. Technical Report. Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States).

[7] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. 2009. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. ACM, 233–244.

[8] Gregor Daiß, Parsa Amini, John Biddiscombe, Patrick Diehl, Juhan Frank, Kevin Huck, Hartmut Kaiser, Dominic Marcello, David Pfander, and Dirk Pflüger. 2019. From piz daint to the stars: simulation of stellar mergers using high-level abstractions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–37.

[9] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.

[10] M. T. Heath. 1984. Sparse matrix computations. In *The 23rd IEEE Conference on Decision and Control*. 662–665. https://doi.org/10.1109/CDC.1984.272092

[11] Thomas Heller, Patrick Diehl, Zachary Byerly, John Biddiscombe, and Hartmut Kaiser. 2017. Hpx–an open source c++ standard library for parallelism and concurrency. *Proceedings of OpenSuCo* (2017), 5.

[12] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. 2014. Hpx: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. 1–11.

[13] Laxmikant V Kale and Sanjeev Krishnan. 1993. Charm++ A portable concurrent object oriented system based on C++. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*. 91–108.

[14] Jeremy Kepner, David Bade, Aydın Buluç, John Gilbert, Timothy Mattson, and Henning Meyerhenke. 2015. Graphs, matrices, and the GraphBLAS: Seven good reasons. *arXiv preprint arXiv:1504.01039* (2015).

[15] Andrew V Knyazev. 2001. Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method. *SIAM journal on scientific computing* 23, 2 (2001), 517–541.

[16] Alexey Kukanov and Michael J Voss. 2007. The Foundations for Scalable Multi-core Software in Intel Threading Building Blocks. *Intel Technology Journal* 11, 4 (2007).

[17] Abhishek Kulkarni and Andrew Lumsdaine. 2019. A Comparative Study of Asynchronous Many-Tasking Runtimes: Cilk, Charm++, ParalleX and AM++. *arXiv preprint arXiv:1904.00518* (2019).

[18] Cornelius Lanczos. 1950. *An iteration method for the solution of the eigenvalue problem of linear differential and integral operators*. United States Governm. Press Office Los Angeles, CA.

[19] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. 1979. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software (TOMS)* 5, 3 (1979), 308–323.

[20] Wonchan Lee, Elliott Slaughter, Michael Bauer, Sean Treichler, Todd Warszawski, Michael Garland, and Alex Aiken. 2018. Dynamic tracing: Memoization of task graphs for dynamic task-based runtimes. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 441–453.

[21] ARB OpenMP. 2013. OpenMP application program interface version 4.0.

[22] Arch D Robison. 2012. Cilk plus: Language support for thread and vector parallelism. *Talk at HP-CAST* 18 (2012), 25.

[23] Erik Saule, Hasan Metin Aktulga, Chao Yang, Esmond G Ng, and Ümit V Çatalyürek. 2015. An out-of-core task-based middleware for data-intensive scientific computing. In *Handbook on Data Centers*. Springer, 647–667.

[24] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. 2015. Regent: a high-productivity programming language for HPC with logical regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[25] Przemysław Stpiczyński. 2018. Language-based vectorization and parallelization using intrinsics, OpenMP, TBB and Cilk Plus. *The Journal of Supercomputing* 74, 4 (2018), 1461–1472.

[26] Peter Thoman, Kiril Dichev, Thomas Heller, Roman Iakymchuk, Xavier Aguilar, Khalid Hasanov, Philipp Gschwandtner, Pierre Lemarinier, Stefano Markidis, Herbert Jordan, et al. 2018. A taxonomy of task-based parallel programming technologies for high-performance computing. *The Journal of Supercomputing* 74, 4 (2018), 1422–1434.

[27] Hilario Torres, Manolis Papadakis, and Lluís Jofre Cruanyes. 2019. Soleil-X: turbulence, particles, and radiation in the Regent programming language. In *SC'19: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–4.

[28] Chun-Kun Wang. 2017. Selection of Parallel Runtime Systems for Tasking Models. In *2017 International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE, 1091–1096.

[29] R Clint Whaley, Antoine Petitet, and Jack J Dongarra. 2001. Automated empirical optimizations of software and the ATLAS project. *Parallel computing* 27, 1-2 (2001), 3–35.

[30] Kyle B Wheeler, Richard C Murphy, and Douglas Thain. 2008. Qthreads: An API for programming with millions of lightweight threads. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 1–8.