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

Abdullah Alperen[†], Md Afibuzzaman[†], Fazlay Rabbi[†], M. Yusuf Özkaya[‡], Ümit V Çatalyürek[‡], Hasan M. Aktulga[†]

[†]Computer Science & Engineering, Michigan State University

[‡]School of Computational Science & Engineering, Georgia Institute of Technology

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

Introduction

- Sparse matrix computations comprise the core component of a broad base of scientific applications.
- They become challenging in the presence of large-scale data due to the memory-bound nature of the computations.
- These challenges are not well addressed by bulk synchronous parallel (BSP) approaches where poor cache performance and high synchronization costs become the limiting factors.
- This validates the emergence and increased use of asynchronous many-task (AMT) programming models.

Introduction

- OpenMP's task parallelism has been used since 2013 allowing extracting parallelism via asynchronous execution of fine-grained tasks [1].

- HPX [2] is an advanced runtime system and a programming API that conforms to the C++ standards while supporting lightweight task scheduling to expose parallelism.

- Regent [3], a programming language and compiler designed for HPC. Regent runtime system discovers implicit dataflow parallelism in the code.

Introduction

- Recently, using the AMT model in OpenMP has been shown to offer important advantages over its BSP model within the DeepSparse framework.

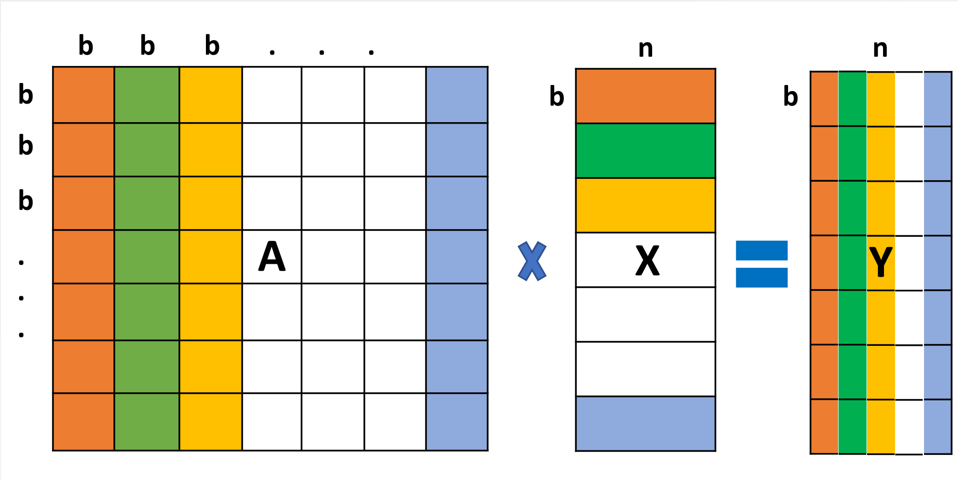
- DeepSparse [4] automatically generates and expresses the entire computation as a task dependency graph (TDG) and relies on OpenMP for the execution of this TDG.

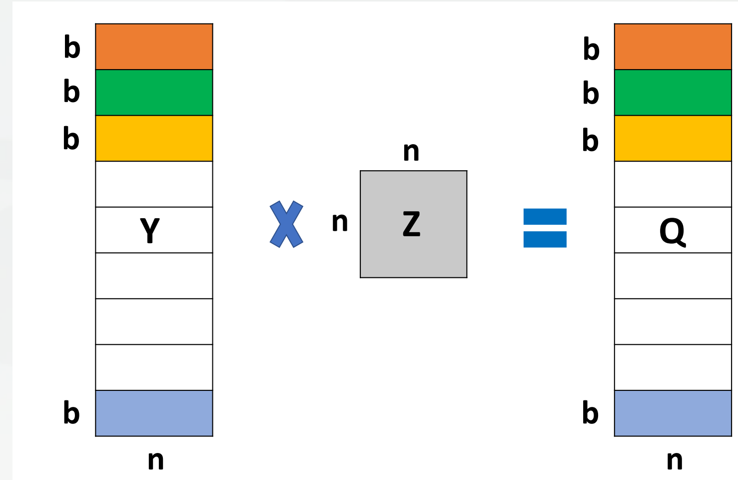
- We aimed to discern how OpenMP, HPX and Regent compare as well as what they offer over BSP models by providing
  - a task-parallel implementation Lanczos and LOBPCG using the HPX and Regent
  - an evaluation of AMT models on multicore and manycore architectures

Implementation

- In all three frameworks (DeepSparse, HPX and Regent), tasks are defined based on the decomposition of sparse matrices.
- We adapt a 2D partitioning scheme using Compressed Sparse Block (CSB) [5] representation of the sparse matrix, which also dictates the decomposition of other data structures involved.
- Consider the following code snippet:

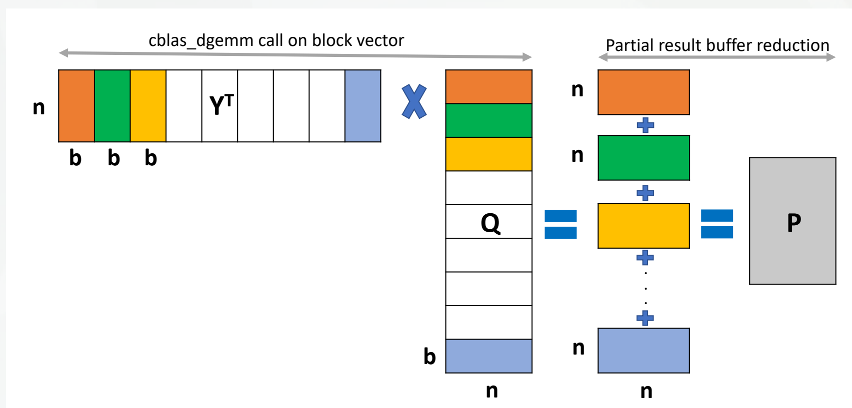
```
SpMM(A, X, Y, m, n); // A*X = Y
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
      m, n, n, 1.0, Y, n, Z, n, 0, Q, n); // Y*Z = Q
cblas_dgemm(CblasRowMajor, CblasTrans, CblasNoTrans, n,
      n, m, 1.0, Y, n, Q, n, 0, P, n); // Y'*Q = P
```





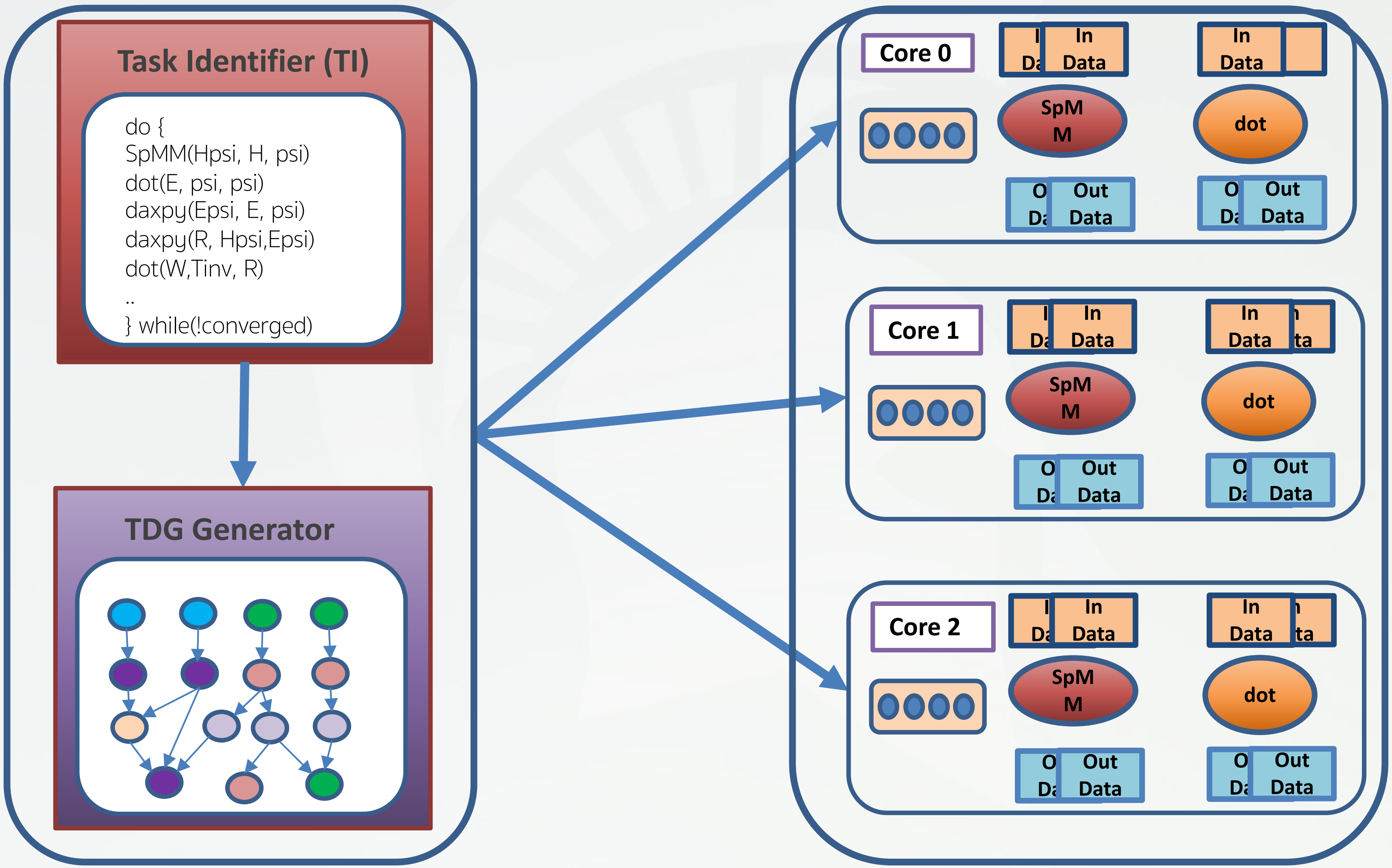
SpMM Kernel

Linear Combination Kernel (XY)



Inner Product Kernel (XTY)

DeepSparse Overview



Primitive Conversion Unit (PCU)

Task Executor

[4]

HPX Overview

- HPX attains asynchronous parallelism through asynchronous function execution and future instances.

- A dataflow object triggers a predefined function when a set of futures become ready

```cpp
std::vector<hpx::shared_future<void>> Y(np);
std::vector<hpx::shared_future<void>> Q(np);
std::vector<hpx::shared_future<void>> P_prtl_ftr(np);
hpx::shared_future<void> P_rdcd_ftr;
// np (number of partitions) = ceil(m/blocksize)
for(int i = 0; i != np; ++i)
    Y_ftr[i] = hpx::make_ready_future();
// to unwrap futures passed to functions
auto OpSpMM = hpx::util::unwrapping(&SpMM);
auto OpDGEMV = hpx::util::unwrapping(&f_dgemm);
auto OpDGEMV_T = hpx::util::unwrapping(&f_dgemm_t);
auto OpRed = hpx::util::unwrapping(&reduce_buf);
// Y = A * X
for(i = 0; i != np; ++i)
    for(int j = 0; j != np; ++j)
        if(A[i * np + j].nnz > 0)
            Y_ftr[i] = hpx::dataflow(hpx::launch::async
                , OpSpMM, Y_ftr[i], A, X, Y, i, j);
// Q = Y * Z
for(i = 0; i != np; ++i)
    Q_ftr[i] = hpx::dataflow(hpx::launch::async,
        OpDGEMV, Y_ftr[i], Y, Z, Q, i);
// P = Y' * Q
for(i = 0; i != np; ++i)
    P_prtl_ftr[i] = dataflow(hpx::launch::async,
        OpDGEMV_T, Y_ftr[i], Q_ftr[i], Y, Q, Pbuf, i);
P_rdcd_ftr = dataflow(hpx::launch::async, OpRed,
    P_prtl_ftr, Pbuf, P);
```

Regent Overview

- Regent exerts implicit dataflow parallelism through two key abstractions: tasks and regions
- Privileges describe how tasks interact with regions(read, write…)

```
fspace csb_entry{
    {rloc, cloc}: uint16, val: double,
}
task SpMM(rA: region(ispace(int1d), csb_entry),
            rX: region(ispace(int1d), double),
            rY: region(ispace(int1d), double),
            s: int, e: int)
where reads(rA, rX), reads writes(rY) do
    -- ... (SpMM implementation)
end
-- ... (other tasks)
task main()
    -- ... np (num partitions) = ceil(m/blksize)
    var sparse_matrix_is = ispace(int1d, nnz)
    var vector_block_is = ispace(int1d, m * n)
    var Alr = region(sparse_matrix_is, csb_entry)
    var Xlr = region(vector_block_is, double)
    -- ... (other region defs, Alr & blkptrs init)
    var part = ispace(int1d, np)
    var Xlp = partition(equal, Xlr, part)
    -- ... (Y and Q partitionings, etc.)
    -- Y = A * X
    for i = 0, np do
        for j = 0, np do
            if blkptrs[i*np+j] < blkptrs[i*np+j+1] then
                SpMM(Alr, Xlp[j], Ylp[i], blkptrs[i*np+
                     j], blkptrs[i*np+j+1])
            end
        end
    end
    -- Q = Y * Z
    __demand(__index_launch)
    for i = 0, np do
        f_dgemm(Ylp[i], Zlr, Qlp[i], m, n, blksize, i)
    end
    -- P = Y' * Q
    __demand(__index_launch)
    for i = 0, np do
        f_dgemm_t(Ylp[i], Qlp[i], Plr, m, n, blksize,i)
    end
```

Performance Evaluation

- Test Applications:
  - **Lanczos** - one SpMV and one inner product kernel at each iteration
  - **LOBPCG** - SpMM based complex algorithm with several kernels

- Two systems in HPC Center @ MSU:
  - Intel **Broadwell** - two 14-core Intel Xeon E5-2680v4 2.4 GHz processors
  - AMD **EPYC** - two 64-core AMD EPYC 7H12 2.6 GHz processors
  - Using an entire node – 28 cores on Broadwell and 128 cores on EPYC

- Baseline versions:
  - **libcsr** - CSR storage format + Intel MKL routines.
  - **libcsb** - CSB storage format + Intel MKL routines.

Performance Evaluation

- Matrices with varying sizes, sparsity patterns, and domains.
- Performance data from the solver iteration parts, averaged over 20 iterations for Lanczos and 10 for LOBPCG.
- Comparison criteria are L1, L2, LLC (L3) misses and execution times, normalized wrt libcsr
- Results from the experiments with optimal block size

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

Table: Matrices used in our evaluation

Lanczos Evaluation

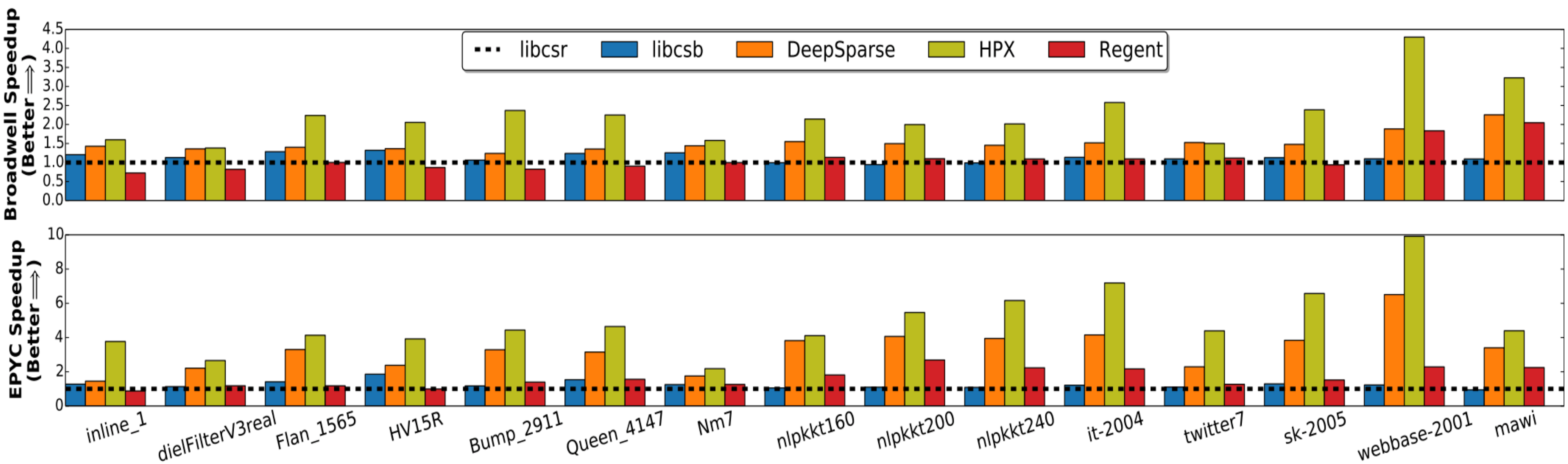


Figure: Speedup of different Lanczos versions on Broadwell (top) and EPYC (bottom) over libcsr.

- DeepSparse, HPX and Regent achieve up to 2.3x, 4.3x and 2.0x improvement, respectively, on Broadwell (1.5x, 2.2x and 1.1x on average).
- Even better, they achieve up to 6.5x (DeepSparse), 9.9x (HPX) and 2.7x (Regent) speedup on EPYC (3.3x, 4.9x and 1.6x speedup on average)

Lanczos Evaluation

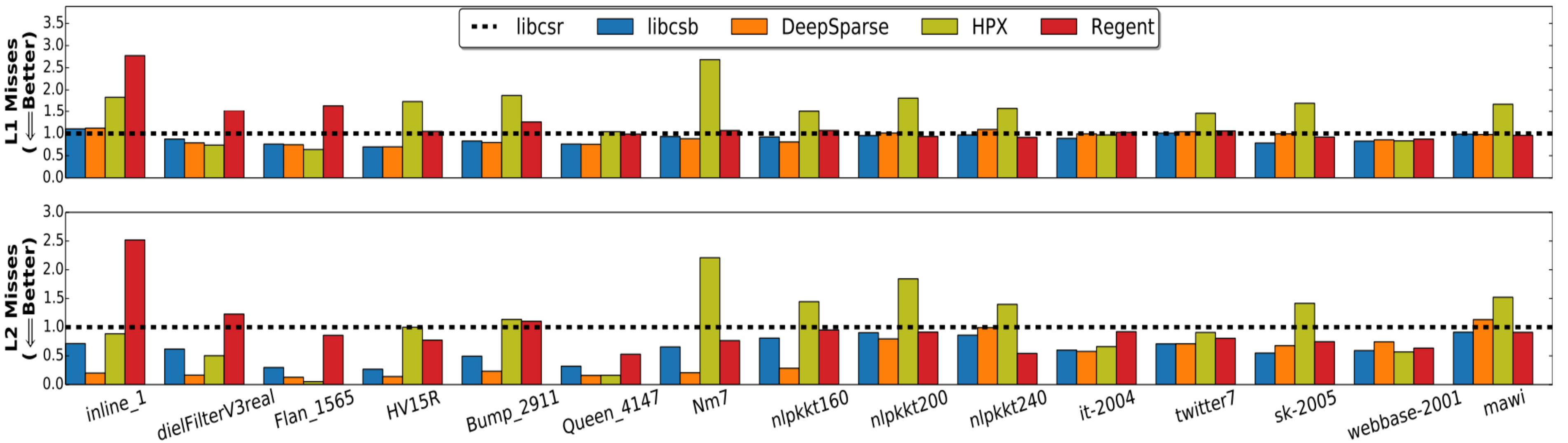


Figure: L1 and L2 misses of different Lanczos versions on EPYC normalized wrt libcsr.

- Lanczos is relatively simple with only few data reuse opportunities so no improvement in terms of cache misses.
- No consistent reduction on L1 level whereas improvements on L2 level can be attributed to the CSB format (L3 misses unavailable due to root access).

Lanczos Evaluation

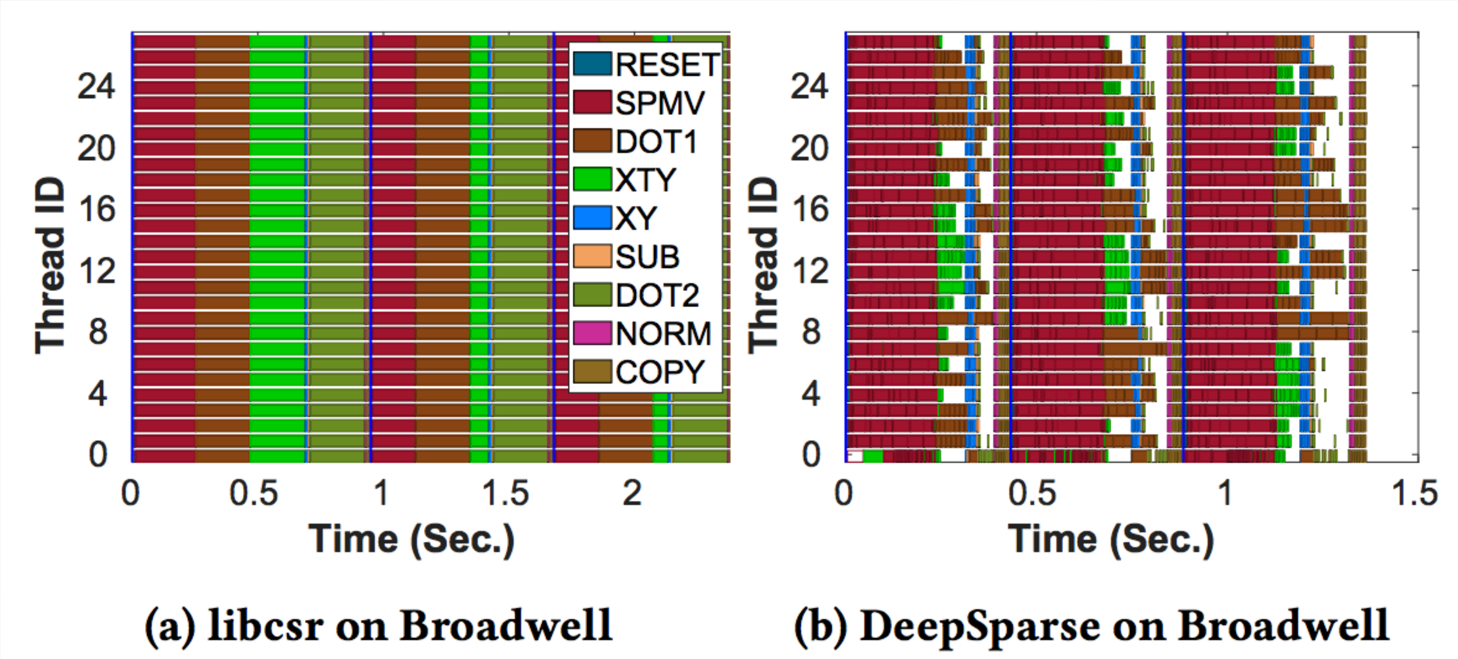


Figure: Execution flow graph of nlpkkt240 from first three iterations of Lanczos.

- We attribute the speedups to the increased parallelism with tasking and reduced synchronization overheads.
- Task parallel systems can fill the gap resulting from load imbalances of SpMV with the succeeding tasks

LOBPCG Evaluation

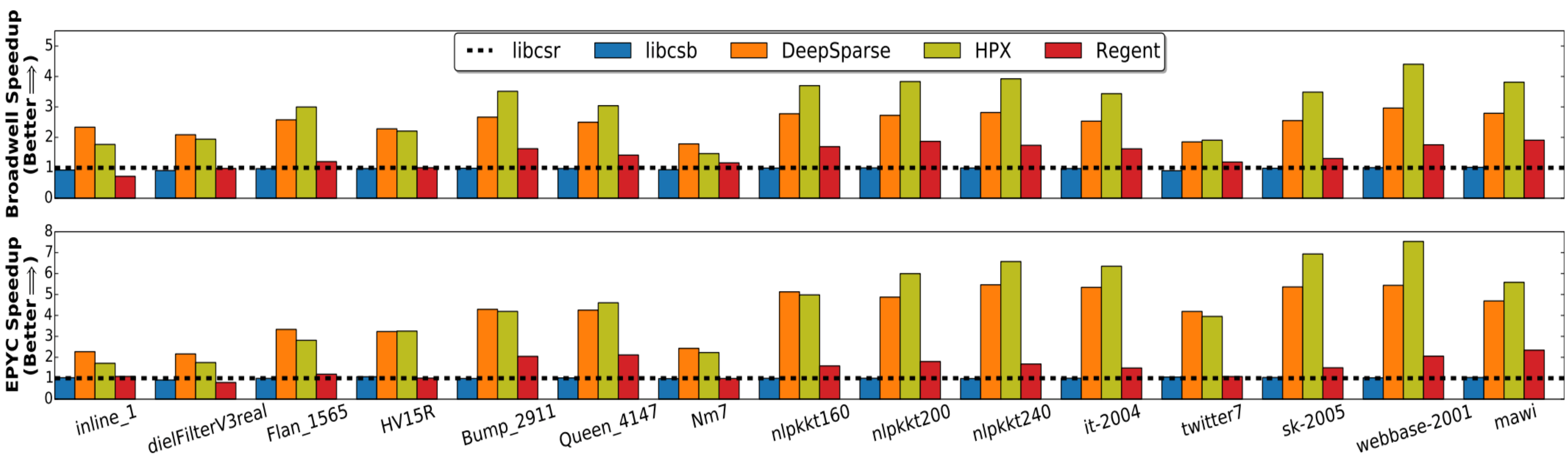


Figure: Speedup of different LOBPCG versions on Broadwell (top) and EPYC (bottom) over libcsr.

- On Broadwell, the speedup numbers are 1.8x - 3.0x for DeepSparse, 1.5x - 4.4x for HPX and 0.8x - 1.9x for Regent (slowdown on smaller matrices).
- They achieve 1.2x - 5.5x (DeepSparse), 1.7x - 7.5x (HPX) and 0.8x – 2.3x (Regent) speedup on EPYC, improving further compared to Broadwell

LOBPCG Evaluation

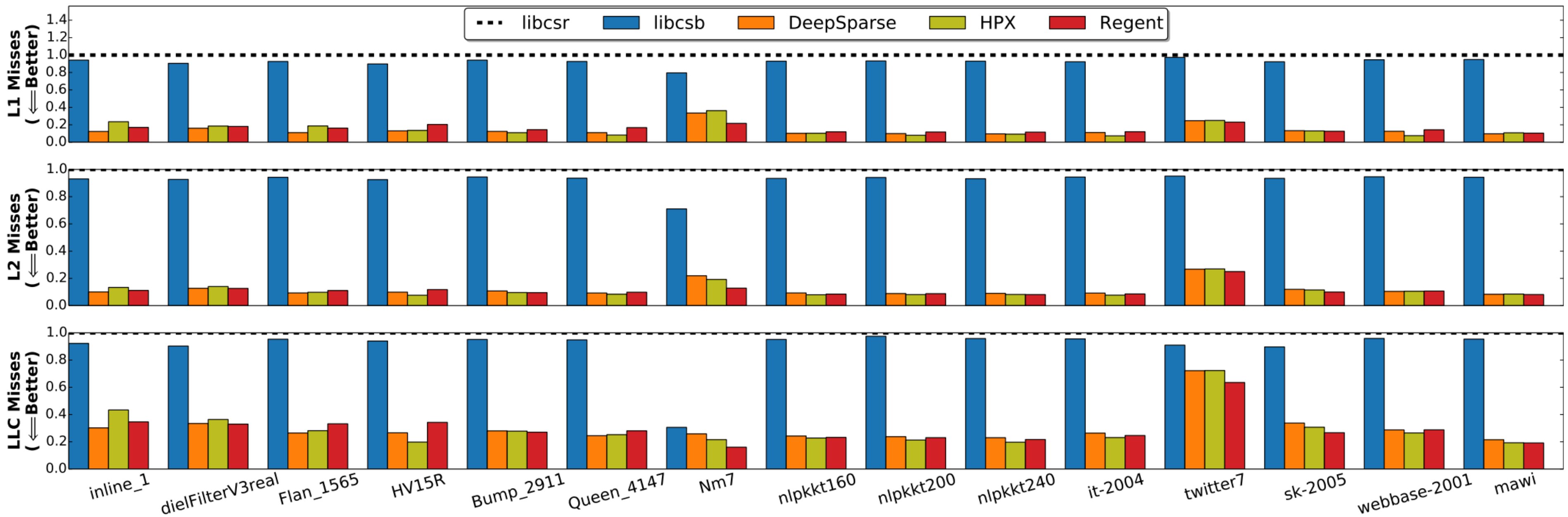


Figure: L1, L2 and LLC (L3) misses of different LOBPCG versions on Broadwell normalized wrt libcsr.

- LOBPCG requires several vector operations consecutively so plenty data reuse opportunities.
- Task parallel versions show outstanding cache miss performance. Besides, they achieve up to 99% L1 hits compared to 85-90% of BSP versions.

Conclusion

- Several AMT frameworks emerged but there is a lack of comparative studies in the context of sparse solvers.
- We introduce optimized implementations of LOBPCG and Lanczos eigensolvers using the task-parallel paradigm in OpenMP (through DeepSparse), HPX and Regent.
- Future work will be testing AMT models in a distributed case using large-scale sparse solvers and graph analytics kernels.
- Please refer to the paper for implementation details, optimization efforts, and for the heuristic to determine the ideal task granularity through the block size.

References

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

[2] 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.

[3] 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.

[4] 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.

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