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:

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

\[ \begin{array}{ccc}
 b & b & b \\
 b & b & b \\
 b & b & b \\
\end{array} \begin{array}{ccc}
 A & Q & P \\
 X & Y & Z \\
 Y & T & X \\
\end{array} \begin{array}{ccc}
 b & b & b \\
 b & b & b \\
 b & b & b \\
\end{array} \]

Linear Combination Kernel (XY)

\[ \begin{array}{ccc}
 b & b & b \\
 b & b & b \\
 b & b & b \\
\end{array} \begin{array}{ccc}
 n & n & n \\
 Y & X & Z \\
 X & Y & Q \\
\end{array} \begin{array}{ccc}
 b & b & b \\
 b & b & b \\
 b & b & b \\
\end{array} \]

Inner Product Kernel (XTY)

\[ \begin{array}{ccc}
 n & b & b \\
 b & b & b \\
 b & b & b \\
\end{array} \begin{array}{ccc}
 Y^T & Q & P \\
 Q & X & Y \\
 X & Y & Z \\
\end{array} \begin{array}{ccc}
 n & n & n \\
 b & b & b \\
 b & b & b \\
\end{array} \]

\[ \begin{array}{ccc}
 n & n & n \\
 n & n & n \\
 n & n & n \\
\end{array} \begin{array}{ccc}
 Q & P & Z \\
 P & X & Y \\
 X & Y & Z \\
\end{array} \begin{array}{ccc}
 n & n & n \\
 b & b & b \\
 b & b & b \\
\end{array} \]
DeepSparse Overview

Task Identifier (TI)

do {
    SpMM(Hpsi, H, psi)
    dot(E, psi, psi)
    daxpy(Epsi, E, psi)
    daxpy(R, Hpsi, Epsi)
    dot(W, Tinv, R)
    ..
} while(!converged)

TDG Generator

Core 0

Core 1

Core 2

Primitive Conversion Unit (PCU)

Task Executor
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
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);
```
Regent Overview

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

```plaintext
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

Table: Matrices used in our evaluation

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

- 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

- 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

- 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.

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

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

- 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.

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