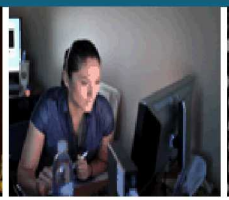
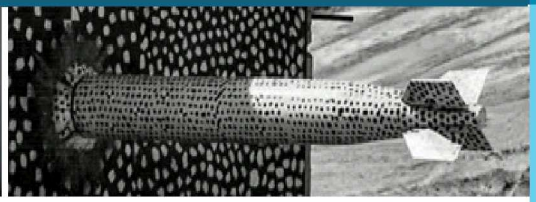
SAND2019-4996C

Development of parallel sparse CP-APR tensor decomposition solvers





Kokkos User Group Meeting, 04/23/2019

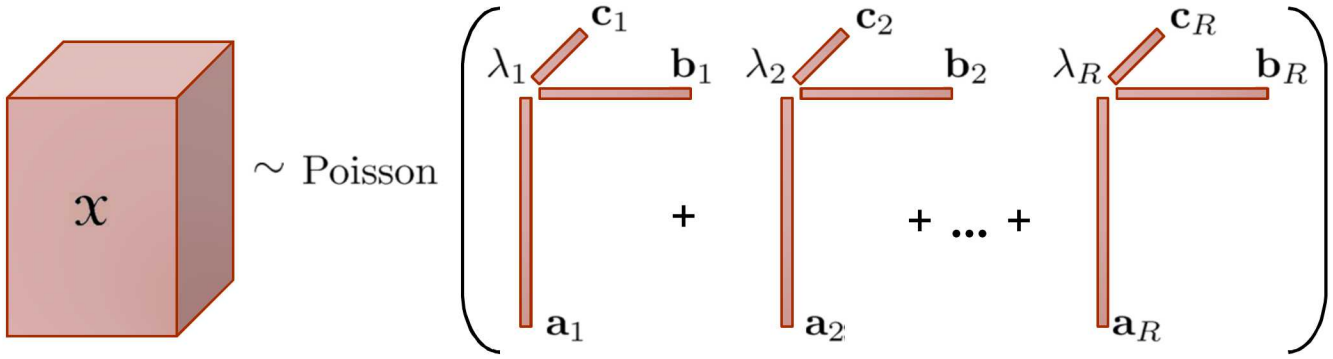
PRESENTED BY

Keita Teranishi

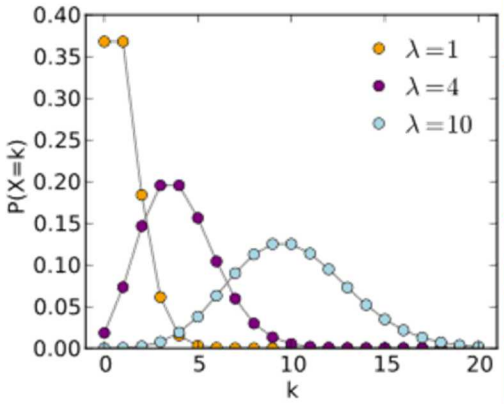
David Hollman, Richard Barrett, Daniel Dunlavy, and Tamara Kolda

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Tensor Decomposition with Poisson Model



$$x_{ijk} \sim \text{Poisson}(m_{ijk}) \text{ where } m_{ijk} = \sum_r \lambda_r \, a_{ir} \, b_{jr} \, c_{kr}$$



$$P(X = x) = \frac{\exp(-\lambda)\lambda^x}{x!}$$

- CANDECOMP/PARAFAC (CP) decomposition
- CP decomposition for **Poisson regression** is suitable to analyze count (non-negative integer) data
  - Involves **nontrivial nonlinear (and non-convex) constraint optimization** problems
- We discuss how to implement the algorithm with Kokkos.

CP-APR

Algorithm 1: CPAPR, Alternating Block Framework

1 CPAPR $(\mathcal{X}, \mathcal{M})$;
  Input : Sparse N-mode Tensor $\mathcal{X}$ of size $I_1 \times I_2 \times \ldots I_N$ and the number of components R
  Output: Kruskal Tensor $\mathcal{M} = [\lambda; A^{(1)} \ldots A^{(N)}]$
2 **Initialize**
3 **repeat**
4 **for** $n = 1, \ldots, N$ **do**
5 Let $\Pi^{(n)} = (A^{(N)} \odot \cdots \odot A^{(n+1)} \odot A^{(n-1)} \odot \ldots A^{(1)})^T$
6 Compute $\bar{A}^{(n)}$ that minimize $f(\bar{A}^{(n)})$ s.t. $\bar{A}^{(n)} \geq 0$
7 $A^{(n)} \leftarrow \bar{A}^{(n)}$
8 **end**
9 **until** *all mode subproblems converged*;

Iterate over modes

Minimization problem is expressed as:

$$\min_{\bar{A}^{(n)} > 0} f(\bar{A}^{(n)}) = e^T [\bar{A}^{(n)} \Pi^{(n)} - X_{(n)} * \log(\bar{A}^{(n)} \Pi^{(n)})] e$$

CP-APR

⊙ is called Khatori-Rao product
(Column wise Kronecker product)

$C = [C_1|C_2|C_3]$

$D = [D_1|D_2|D_3]$

$C \odot D = [C_1 \otimes D_1 \mid C_2 \otimes D_2 | C_3 \otimes D_3]$

Algorithm 1: CPAPR

1 CPAPR $(\mathcal{X}, \mathcal{M})$;
Input : Sparse N-mode Tensor $\mathcal{X}$ of size $I_1 \times I_2 \times \ldots I_N$ and the number of components R
Output: Kruskal Tensor $\mathcal{M} = [\lambda; A^{(1)} \ldots A^{(N)}]$
2 **Initialize**
3 **repeat**
4 **for** $n = 1, \ldots, N$ **do** ← Iterate over modes
5 Let $\Pi^{(n)} = (A^{(N)} \odot \cdots \odot A^{(n+1)} \odot A^{(n-1)} \odot \ldots A^{(1)})^T$
6 Compute $\bar{A}^{(n)}$ that minimize $f(\bar{A}^{(n)})$ s.t. $\bar{A}^{(n)} \geq 0$
7 $A^{(n)} \leftarrow \bar{A}^{(n)}$
8 **end**
9 **until** *all mode subproblems converged*;

Minimization problem is expressed as:

$$\min_{\bar{A}^{(n)} > 0} f(\bar{A}^{(n)}) = e^T [\bar{A}^{(n)} \Pi^{(n)} - X_{(n)} * \log(\bar{A}^{(n)} \Pi^{(n)})] e$$

CP-APR

⊙ is called Khatori-Rao product
(Column wise Kronecker product)

$C = [C_1|C_2|C_3]$

$D = [D_1|D_2|D_3]$

$C \odot D = [C_1 \otimes D_1 \mid C_2 \otimes D_2 | C_3 \otimes D_3]$

Algorithm 1: CPAPR

1 CPAPR $(\mathcal{X}, \mathcal{M})$;
Input : Sparse N-m
number of components R
Output: Kruskal Tensor $\mathcal{M} = [\lambda; A^{(1)} \dots A^{(N)}]$
2 **Initialize**
3 **repeat**
4 **for** $n = 1, \dots, N$ **do**
5 Let $\Pi^{(n)} = (A^{(N)} \odot \cdots \odot A^{(n+1)} \odot A^{(n-1)} \odot \dots A^{(1)})^T$
6 Compute $\bar{A}^{(n)}$ that minimize $f(\bar{A}^{(n)})$ s.t. $\bar{A}^{(n)} \geq 0$
7 $A^{(n)} \leftarrow \bar{A}^{(n)}$
8 **end**
9 **until** *all mode subproblems converged*;

Π is expressed in sparse matrix (indices and values).

Iterate over modes

Minimization problem is expressed as:

$$\min_{\bar{A}^{(n)} > 0} f(\bar{A}^{(n)}) = e^T [\bar{A}^{(n)} \Pi^{(n)} - X_{(n)} * \log(\bar{A}^{(n)} \Pi^{(n)})] e$$

CP-APR

⊙ is called Khatori-Rao product
(Column wise Kronecker product)

$C = [C_1|C_2|C_3]$

$D = [D_1|D_2|D_3]$

$C \odot D = [C_1 \otimes D_1 \mid C_2 \otimes D_2 | C_3 \otimes D_3]$

Algorithm 1: CPAPR

1 CPAPR $(\mathcal{X}, \mathcal{M})$;
Input : Sparse N-m
number of components R
Output: Kruskal Tensor $\mathcal{M} = [\lambda; A^{(1)} \dots A^{(N)}]$
2 **Initialize**
3 **repeat**
4 **for** $n = 1, \dots, N$ **do**
5 Let $\Pi^{(n)} = (A^{(N)} \odot \dots \odot A^{(n+1)} \odot A^{(n-1)} \odot \dots A^{(1)})^T$
6 **Compute** $\bar{A}^{(n)}$ **that minimize** $f(\bar{A}^{(n)})$ **s.t.** $\bar{A}^{(n)} \geq 0$
7 $A^{(n)} \leftarrow \bar{A}^{(n)}$
8 **end**
9 **until** *all mode subproblems converged*;

Π is expressed in sparse matrix (indices and values).

Iterate over modes

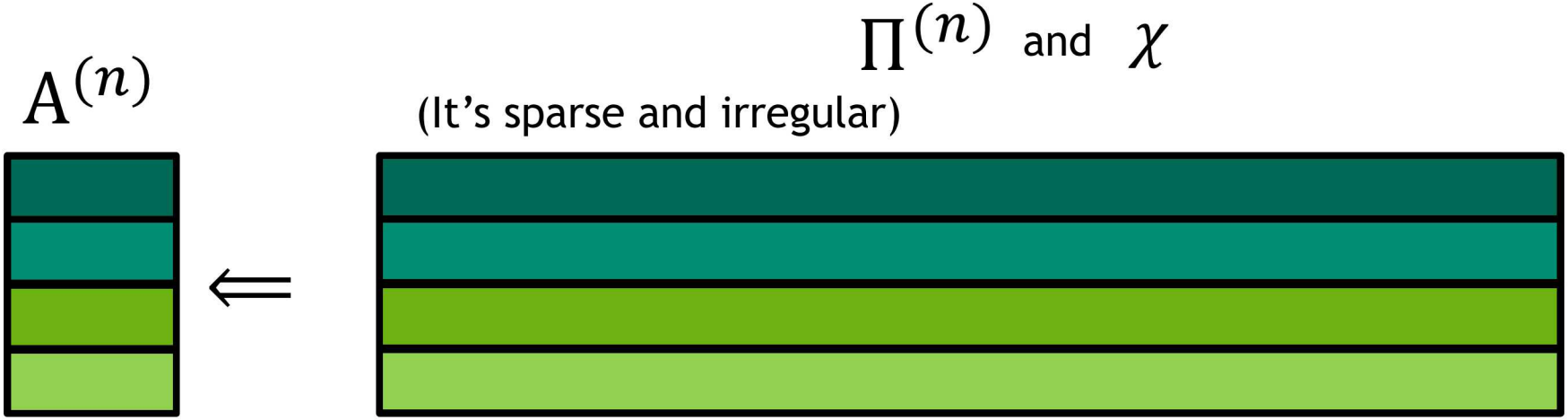
ion

95%+ of Total Execution Time

Minimization problem is expressed as:

$$\min_{\bar{A}^{(n)} > 0} f(\bar{A}^{(n)}) = e^T [\bar{A}^{(n)} \Pi^{(n)} - X_{(n)} * \log(\bar{A}^{(n)} \Pi^{(n)})] e$$

Parallelism in the key computation



- Algorithm #1: Data Parallel Computation
  - Steepest Descent-like algorithm to update the all rows of $A^{(n)}$
  - Requires many iterations to converge

- Algorithm #2: Task Parallel Computation
  - Constraint Newton-based optimization for the individual rows of $A^{(n)}$
  - For each row, the optimization solver is executed independently
  - Better convergence property

Introducing Data Parallelism CP-APR Multiplicative Update

- Relatively Straightforward
  - Kokkos::View to express all data structures
    - Factor Matrices $A^{(n)}$
    - Unfolded Tensor $\Pi^{(n)}$
    - Sparse Tensor χ
  - Data parallelism
    - parallel_for
    - parallel_reduce
  - Works like OpenMP

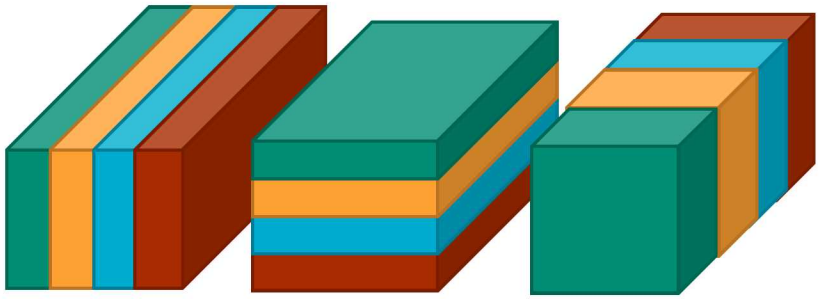
```
Kokkos::parallel_for (policy, KOKKOS_LAMBDA (Policy::member_type team)
{
  const auto iNonz_start = team.league_rank()*TeamSize;
  const auto iNonz_end = iNonz_start + TeamSize < sparse_nElement ? iNonz_start + TeamSize : sparse_nElement;

  Kokkos::parallel_for (Kokkos::TeamThreadRange(team, iNonz_start, iNonz_end), [&] (ElemIdx iNnz)
  {
    const auto index = indices(iDim,iNnz);
    KruskalValue dVal = 0;
    Kokkos::parallel_reduce (Kokkos::ThreadVectorRange(team, kruskal_nComponent), [=] (SubIdx iComp, KruskalValue &ldVal)
    {
      ldVal += kData(index, iComp) *pi(iNnz, iComp);
    }, dVal);

    dVal = static_cast<KruskalValue>(spData(iNnz)) /
      static_cast<KruskalValue>(max(static_cast<KruskalValue>(eps), static_cast<KruskalValue>(dVal)));
    Kokkos::parallel_for (Kokkos::ThreadVectorRange(team, kruskal_nComponent), [&] (SubIdx iComp)
    {
      Kokkos::atomic_add(&phi(index, iComp), dVal * pi(iNnz, iComp ));
    });
  });
});
```

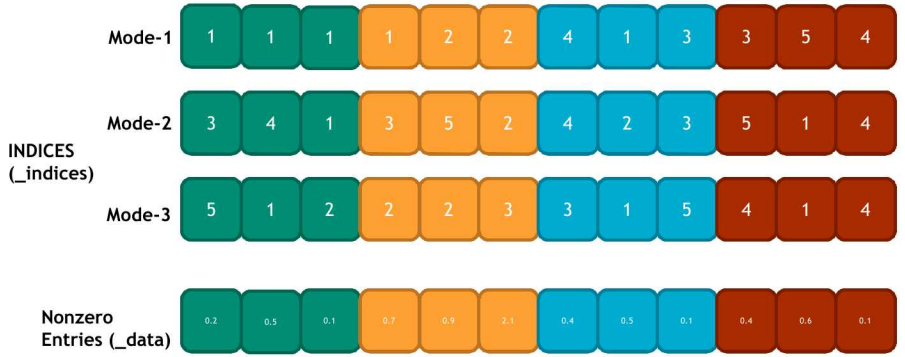
Introducing GPUs: CP-APR, Multiplicative Update

- Poor performance with UVM option
  - Order of magnitude slow down
  - **Chose explicit data movement**

- Extra care is taken to introducing math functions inside parallel_for/parallel_reduce
  - max, min, fabs and log for GPUs

Introducing GPUs: CP-APR, Multiplicative Update (Data Partitioning)





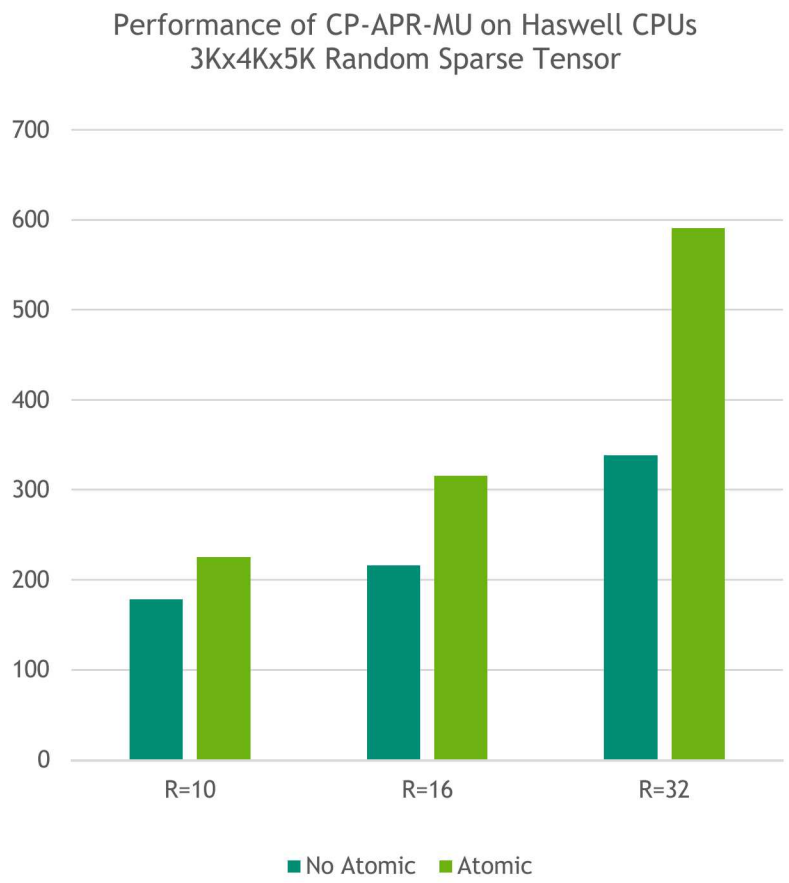
- CPU version exploits row-wise data partitioning
  - All thread makes exclusive access to individual rows (no atomics)
  - GPU leads poor utilization of computing resources for rows with few nonzero entries
  - Require extra indexing to access tensor elements by row in each mode
    - Non-contiguous memory access to sparse tensor data χ

- GPU version exploits direct partitioning the sparse tensor (COO) data format
  - Multiple threads accesses the same rows
  - Contiguous memory access to $\Pi^{(n)}$ and χ
  - Exploit GPU Atomics to avoid race conditions

CP-APR-MU: Performance on GPUs (10 inner, 10 outer iterations, 10 components)

Data	Haswell CPU 1-core		2 Haswell CPUs 14-cores		2 Haswell CPUs 28-cores		Intel KNL (Cache Mode) 68-core CPU		NVIDIA P100 GPU		NVIDIA V100 GPU	
	Time(s)	Speedup	Time(s)	Speedup	Time(s)	Speedup	Time(s)	Speedup	Time(s)	Speedup	Time(s)	Speedup
Random	185	1	22	8.4	13	14.11	8.4	22.01	4.47	41.31	3.01	61.53
LBNL	39	1	19	2.05	13	3.0	33	1.18	2.99	13.04	2.09	18.66
NELL-2	1157	1	137	8.44	87	13.29	100	11.02	47.17	24.52	28.80	40.17
NELL-1	3365	1	397	16.62	258	20.9	257	10.86	OOM		OOM	
Delicious	4170	1	2183	1.91	1872	2.23	3463	1.41	OOM		OOM	

Performance Comparison: Atomic vs Non-Atomic

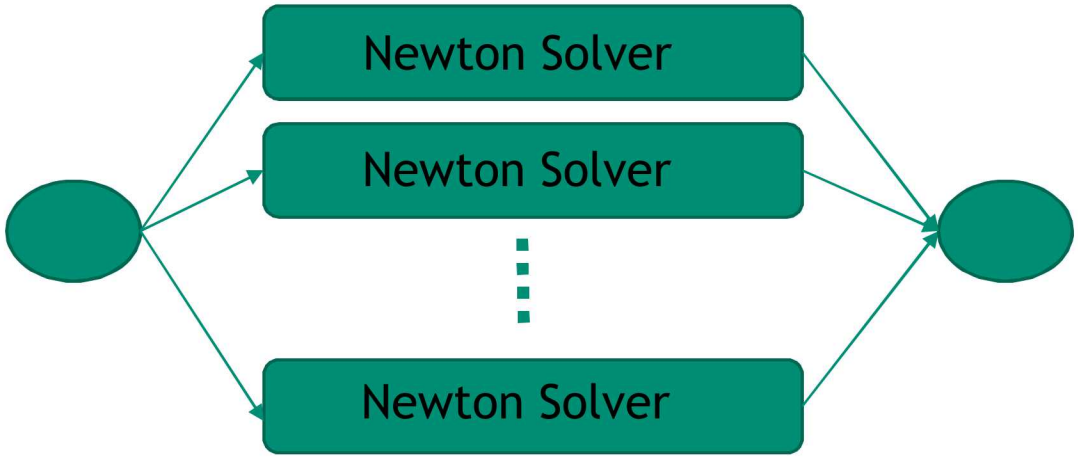


Lower is better

Intel CPUs: Software-based atomic operations

NVIDIA GPUs: Hardware-based atomic operations

Adapting Kokkos to CP-APR-Newton for Row Subproblems



- ***Task*** parallelism using Parallel_for/parallel_reduce
  - Run the same Newton algorithm
- Two types of Newton Algorithms
  - Projected Damped Newton (PDNR)
  - Projected Quasi Newton (PQNR)
- PDNR involves:
  - Dense linear algebra for computing Hessian
    - DOT, GEMM, GEMV, TRSV and POSV (Cholesky factorization and solve)
- PQNR
  - More iterations to converge
  - Vector dot product
  - Less temporary storage

Adapting Kokkos to CP-APR-Newton for Row Subproblems

```
Kokkos::parallel_reduce(
  team_policy.set_scratch_size(1, Kokkos::PerTeam(scratch_per_team)),
  DampedNewtonKernel<team_policy_t, NumericTypes>{
    /* pi = */ pi,
    /* dKdata = */ kruskalOutput.get_factor_matrix(iDim),
    /* nonz_loc = */ nonzLocs[iDim],
    /* nonz_loc_idx = */ nonzLocsIdx[iDim],
    /* sparse_data = */ sparseInput.get_data_view(),
    /* configuration = */ _config,
    /* input_info = */ sparse_tensor_info,
    /* kruskal_info = */ kruskal_info
  },
  reducer_type{modeValues}
);
```

- ***Task*** parallelism using Parallel_for/parallel_reduce
  - Run the same Newton algorithm
- Two types of Newton Algorithms
  - Projected Damped Newton (PDNR)
  - Projected Quasi Newton (PQNR)
- PDNR involves:
  - Dense linear algebra for computing Hessian
    - DOT, GEMM, GEMV, TRSV and POSV (Cholesky factorization and solve)
- PQNR
  - More iterations to converge
  - Vector dot product
  - Less temporary storage

Inside "Newton Method Kernel"

- Create RowSubProblem Solver class to manage
  - Scratch space allocation
  - Execution of Newton Iteration
- The solver is called inside parallel_reduce
  - The solver is assigned to single "team"
  - We have had hard time to understand the league, team, thread, vector

```
KOKKOS_INLINE_FUNCTION
void operator()(
  team_member_t const& team_member,
  reducer_value_t& output
) const
{

  // Readability aliases:
  auto iRow = team_member.league_rank();

  // Construct the row subproblem Solver
  auto row_solver = RowSubProblemDampedNewton<TeamPolicy, NumericalTypes>{
    /* iRow = */ sub_index_t(iRow),
    /* kernel = */ *this,
    /* team_member = */ team_member
  };

  if(not row_solver.has_nonzero_entries())
  {
    sparten::deep_copy(team_member, dKdata, 0);
  }
  else
  {
    auto result = row_solver._solve();

    output.row_vars_modified = output.row_vars_modified || result.first;
    if(result.second > output.max_kkt_violation) {
      output.max_kkt_violation = result.second;
    }

    // TODO function evaluation count and inner iterations
  }

}
```

Inside "Row Subproblem Solver"

- Utilize TeamThread or ThreadVector Range to parallelize vector operations.
- Team-wise synchronization
- Class methods for
  - Gradient computation
  - Search direction computation
  - Evaluation of the objective function
  - Line Search

```
// Perform iterations to solve the row subproblem.
for (sub_index_t iIts = 0; iIts < _config.max_inner_iterations; ++iIts)
{
  if( iIts == 0 )
  {
      // set the initial error
    dInitialKktError = dKktError;
  }
  // Gradient is 1 - phi, where
  // phi_j = sum i=1:nnz X[i] / (sum r=1:R m[r] Pi[i,r]), for j=1:R
  // Save phi to use later in computing the Hessian.
  this->_compute_phi();


  // Compute the gradient and maximum KKT error for the row subproblem.
  dKktError = 0.0;
  Kokkos::parallel_reduce(
    Kokkos::ThreadVectorRange(_team_member, nComps),
    [&] KOKKOS_FUNCTION (sub_index_t iComp, kruskal_value_t& dKktErrorLocal) {
      _daGrad(iComp) = 1.0 - _daPhi(iComp);
      kruskal_value_t d = min(_daVars(iComp), _daVars(iComp));
      dKktErrorLocal = max(abs(d), dKktError);
    },
    Kokkos::Max<kruskal_value_t>{dKktError}
  );

  if (dKktError < _config.row_tolerance) {...}


  // Compute a search direction based active and free variables,
  // using a damped Newton step for the free variables.
  _compute_search_dir( dMuDamping, dPredictedReduction );
  _team_member.team_barrier();


  // Perform a line search.
  kruskal_value_t dAred; // is this unused?!??
  this->_line_search(
    /* daRowVars = */ _daVarsOld,
    /* daRowGrad = */ _daGradOld,
    /* dNewRowVars = */ _daVars,
    /* dObjNew = */ dObj,
    /* dUnitStepAred = */ dAred,
    /* nRetCode = */ nLineSearchCode
  );

  if (dPredictedReduction == 0.0) {...}
  else {...}
}
```

Inside Search Direction Computation

```
spartenBlas::gemm('T', 'N', nNumFree, nNumFree, _nNonz, tempOne,
      daTmpMat1, _nNonz, daTmpMat2, _nNonz, tempOne, daFreeHessian,
      nNumFree );

Kokkos::parallel_for (
  Kokkos::ThreadVectorRange( _team_member, nNumFree ), [&]  KOKKOS_FUNCTION (sub_index_t iNum  ) { ...
});
_team_member.team_barrier();
auto& daFreeSearch  = _daWork1;

if (spartenBlas::posv(nNumFree, daFreeHessian, daFreeSearch)) {...}

auto& daSolution  = _daWork3;
Kokkos::parallel_for ( Kokkos::ThreadVectorRange( _team_member, nNumFree), [&]  KOKKOS_FUNCTION (sub_index_t iNum ) {...
});
_team_member.team_barrier();

spartenBlas::trmv('L', 'T', 'N', nNumFree, daFreeHessian, nNumFree,  daSolution, static_cast<sub_index_t>(1));
spartenBlas::trmv('L', 'N', 'N', nNumFree, daFreeHessian, nNumFree,  daSolution, static_cast<sub_index_t>(1));
```

- Hessian Computation of PDNR involves multiple BLAS like computation
  - Team-based
  - One of the dimensions is small (5-200) and fixed all across the rows
    - GEMM, POSV, DOT, TRMV
  - Another dimension depends on number of nonzero entries in the row (1-1000)
    - GEMM, GEMV
- We need optimized kernels executed by a single team.
  - Opportunity for KokkosKernels!

Conclusion

- Development of Portable on-node Parallel CP-APR Solvers
  - Data parallelism for MU method
  - Performance on CPU, Manycore and GPUs
  - Atomic operations boost the performance of GPUs, which eliminates the need for the extra indexing (reordering)
- PDNR/PQNR solver is under development
  - We have struggled to introduce the advanced features of Kokkos
    - Scratch space
    - Nested parallelism
  - For performance tuning, we need more help from Kokkos and KokkosKernels developers

Extra Slides CP-APR-MU Performance

Performance Test

Strong Scalability
- Problem size is fixed

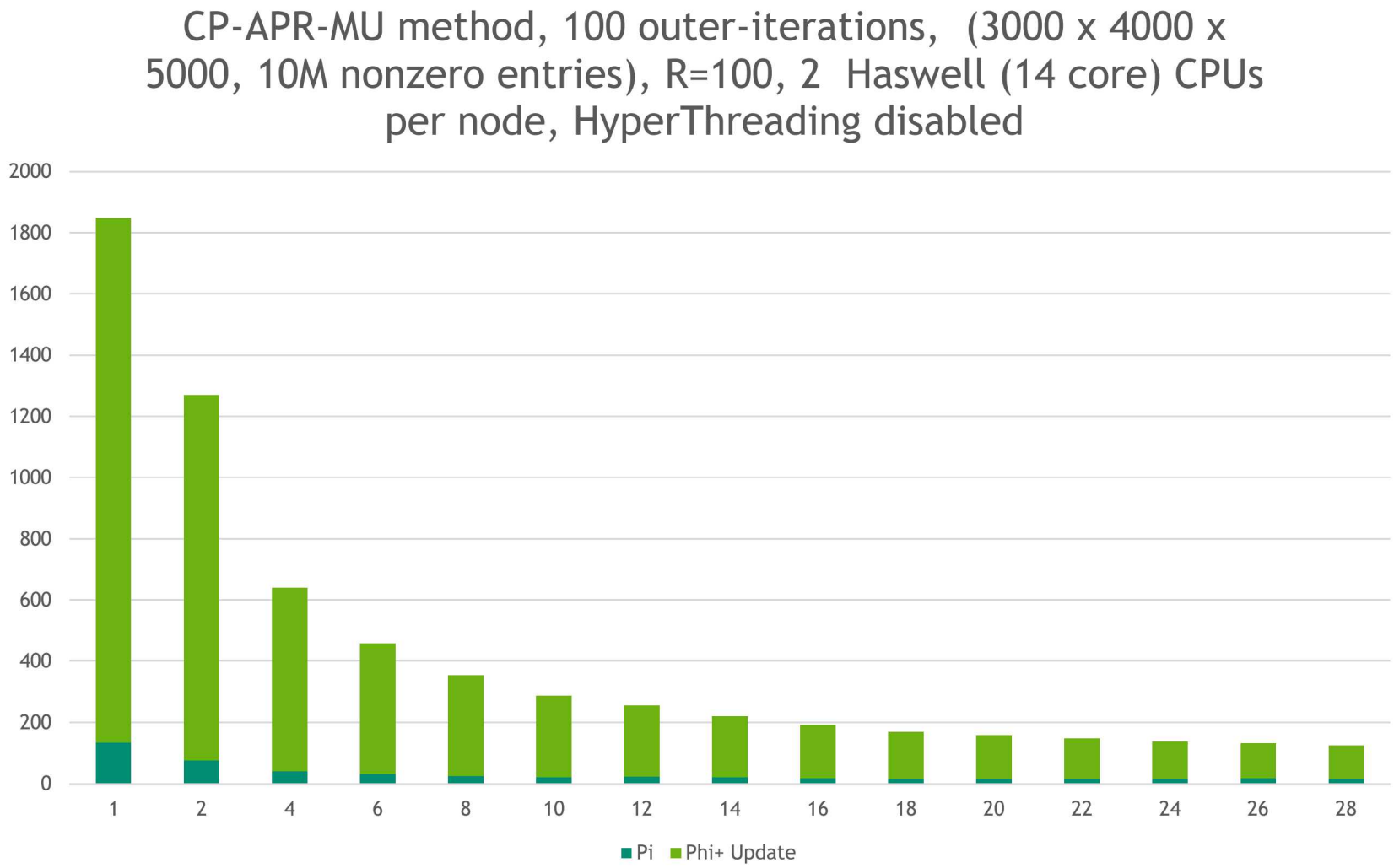
Random Tensor
- 3K x 4K x 5K, 10M nonzero entries
- **100 outer iterations**

Realistic Problems
- Count Data (Non-negative)
- Available at http://frostt.io/
- **10 outer iterations**

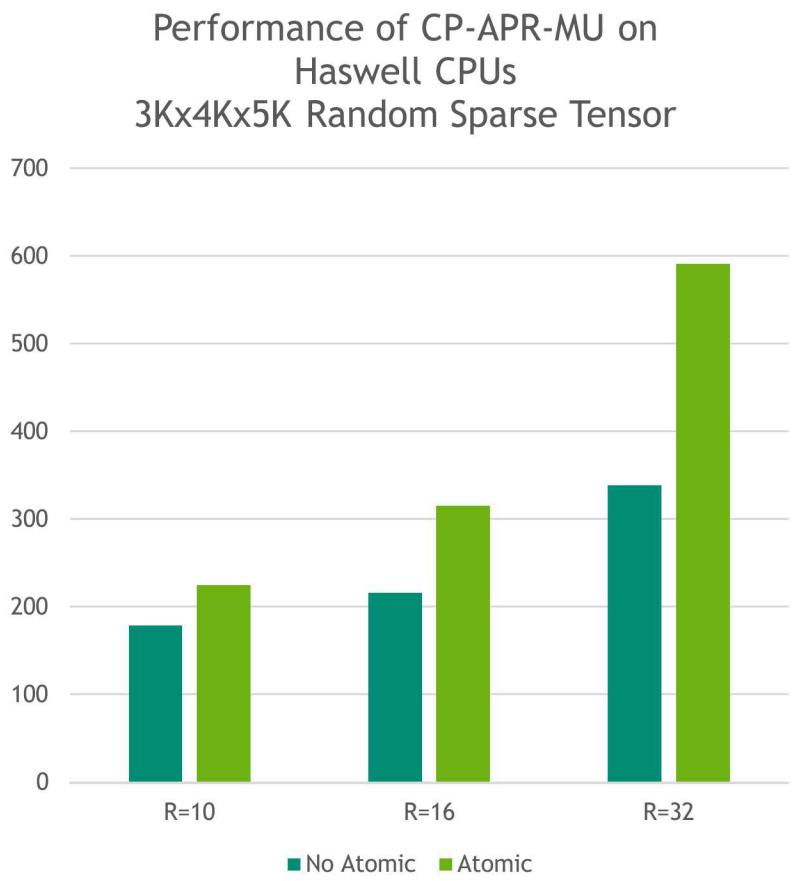
Data	Dimensions	Nonzeros	Rank (*)
LBNL	2K x 4K x 2K x 4K x **866K**	1.7M	10
NELL-2	12K x 9K x 29K	77M	10
NELL-1	3M x 2M x 25M	144M	10
Delicious	500K x **17M** x 3M x **1K**	140M	10

(*) if not indicated.

Scalability of CPAPR-MU on CPU (Random)



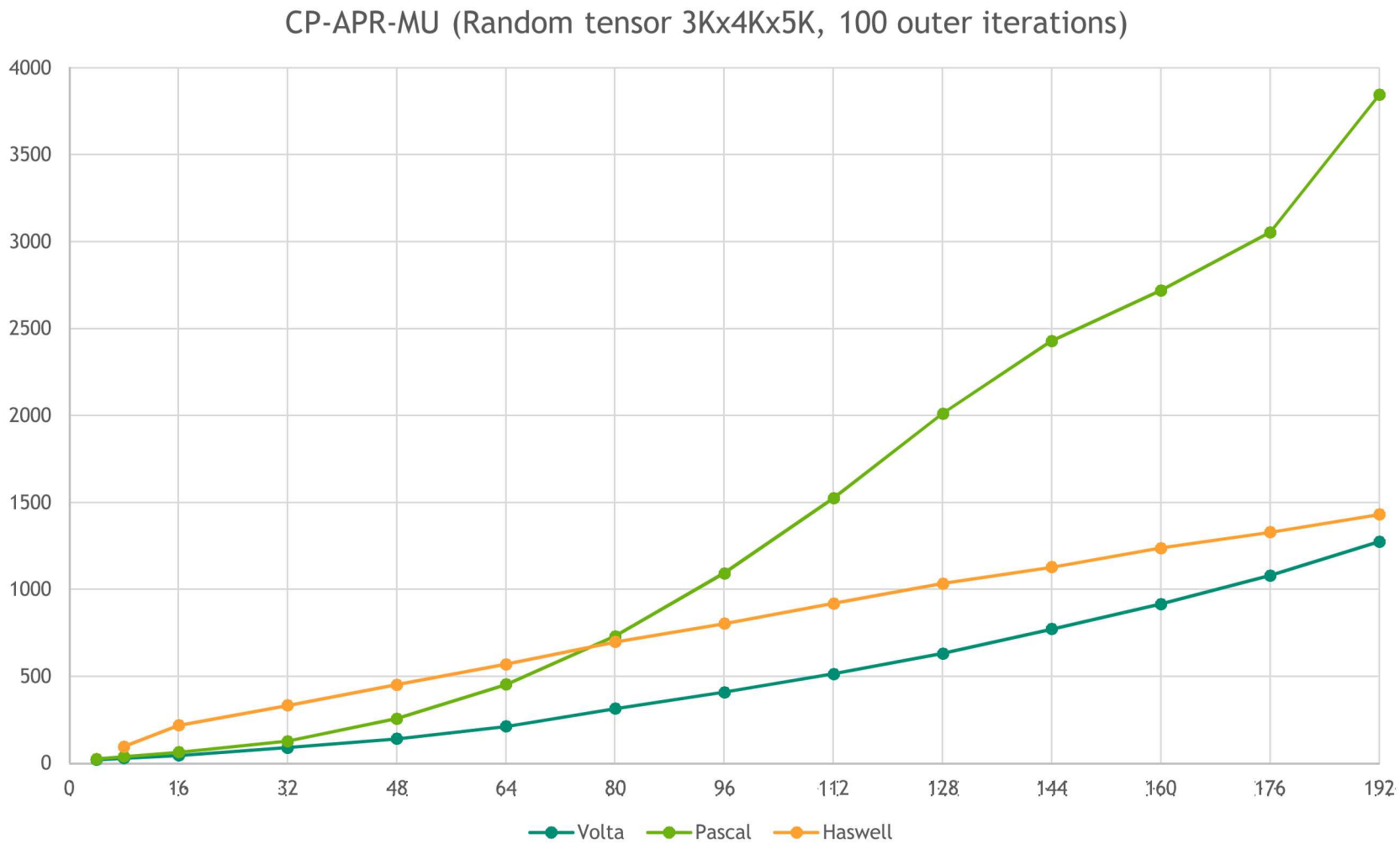
Performance Comparison: Atomic vs Non-Atomic



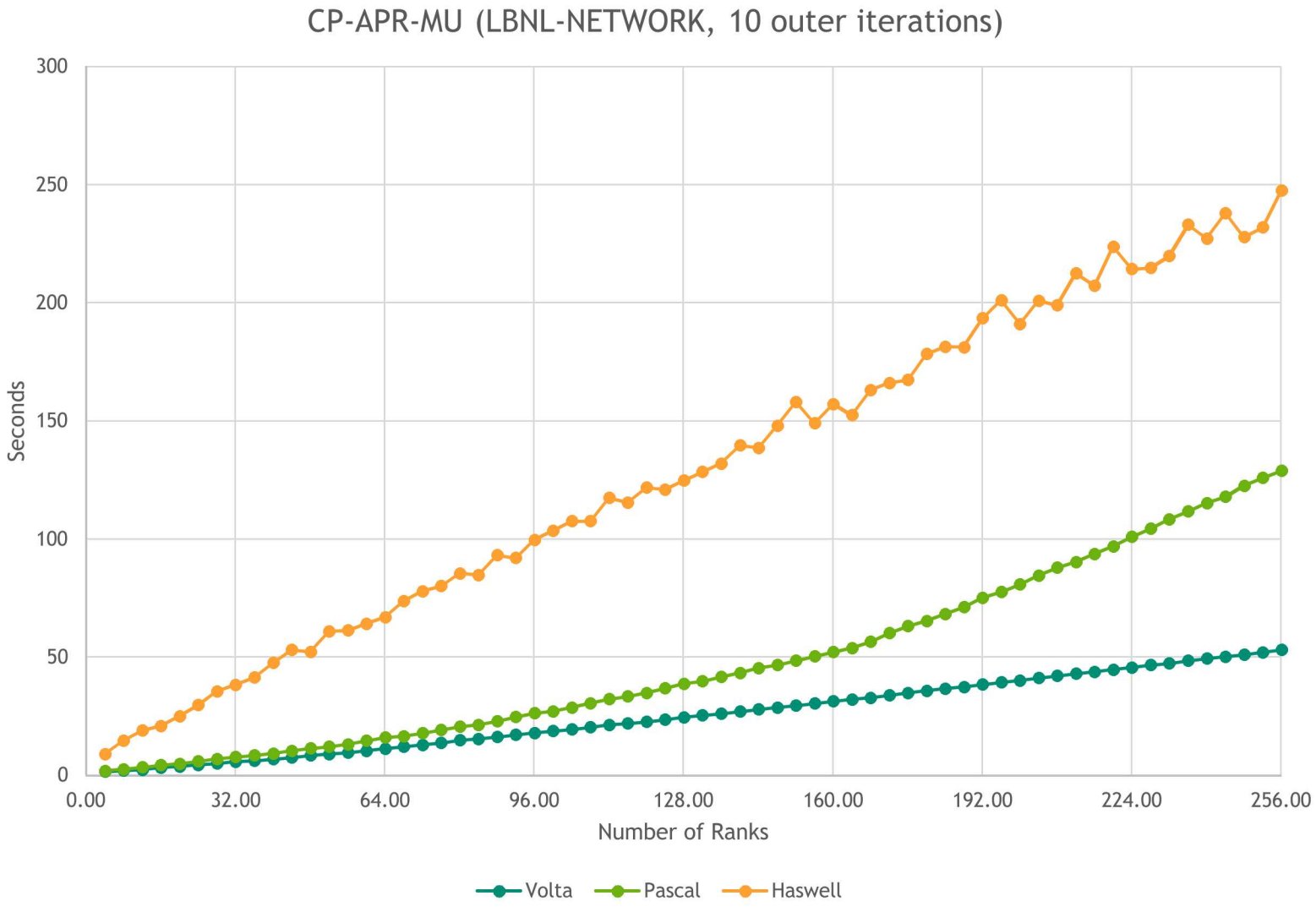
Intel CPUs: Software-based atomic operations

NVIDIA GPUs: Hardware-based atomic operations

Performance of CPU-APR-MU with respect to different rank size



Performance of CP-APR-MU (LBNL-Network) with respect to different rank sizes



CPAPR-PDNR on CPU(Random)