Evaluation of Programming Models to Address Load Imbalance on Distributed Multi-Core CPUs: A Case Study with Block Low-Rank Factorization

Yu Pei[1], George Bosilca[1], Ichitaro Yamazaki[2],
Akihiro Ida[3], and Jack Dongarra[1]

Nov 17th, 2019

PAW-ATW Workshop@SC19, Denver Colorado

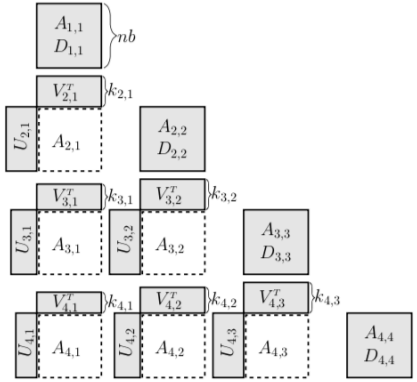
University of Tennessee[1], Sandia National Laboratory[2], University of Tokyo[3]

ICL INNOVATIVE COMPUTING LABORATORY

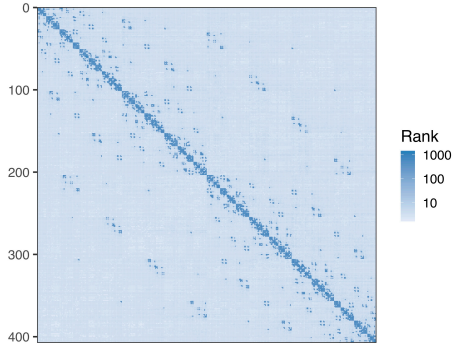
THE UNIVERSITY OF TENNESSEE KNOXVILLE

Motivation

- Flat-MPI (one process per core) can be inefficient when dealing with irregular workload (long wait time in blocking collective calls)
- One such case is Block Low Rank (BLR) factorization
- Compared with dense, less memory and computation
- But compared with hierarchical formats, less complicated to implement, since the data distribution will be regular
- MPI+OMP or use alternative programming models?



Compressed off-diagonal blocks



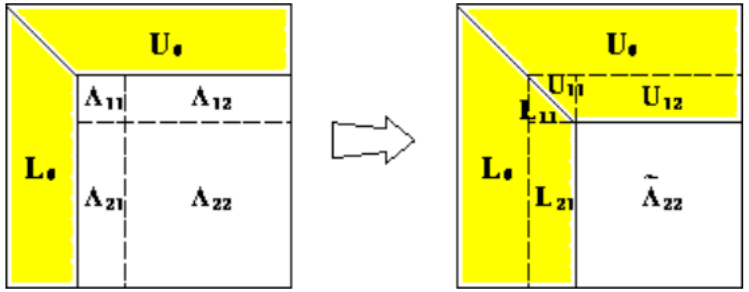
Ranks of a BLR matrix,
Block sizes from 500 to 1000

BLR LU Background

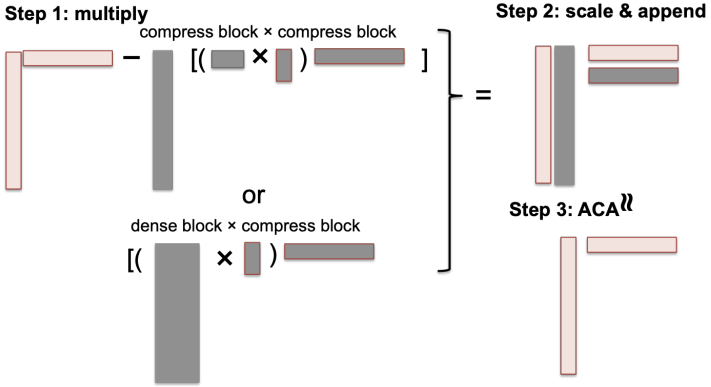
- Same steps as dense LU, with pivots only within the diagonal blocks
- After diagonal and panel factorization, depending on whether the blocks are dense or low rank, there are total of 8 configurations.
- Avoid rank increase and maintain accuracy, Adaptive Cross Approximation (ACA) for recompression

Several features needed for efficient BLR algorithm:

- handle load imbalance
- dynamic data reallocation
- variable send/receive data size
- communication computation overlap



Block LU factorization steps



Updating a low rank block
order the ops to minimize FLOPS

Programming Models Background

1. MPI – the de facto approach (with one process per core)
2. MPI+OpenMP - nested level of parallelism with OpenMP tasks to manage local dependencies. MPI thread multiple for global communications
3. Adaptive MPI/Charm++ - oversubscribe the cores, MPI implemented on top of Charm++, overlap the long blocking collectives
4. PaRSEC DTD – Dynamic Task Discovery, a sequential task insertion model within the PaRSEC runtime
5. PaRSEC PTG – Parameterized Task Graph, a dataflow model with its domain specific language JDF

All use the two-dimensional block cyclic distribution for fair comparison

Flat MPI

- ScaLAPACK approach (standard for dense linear algebra)
- 2-dimensional block cyclic distribution
- Row and column communicators on the process grid for broadcast
- Two messages for the broadcast (1. the shape of the block 2. actual data)
- Synchronization on each step with imbalanced workload (Motivates us to try AMPI)

Adaptive MPI (AMPI)

- With minimum changes to the MPI code (program -> submodule mpi_main)
- Run with a wrapper ***charmrun***
- Oversubscribe the cores in the hope of overlapping computation and communication

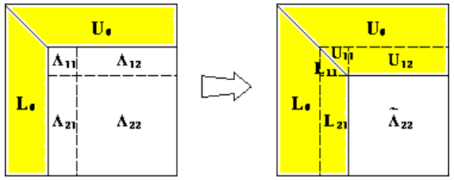
MPI+OpenMP

- Relax the synchronization, task on the block panel, which will spawn nested parallel works on the panel
- Since data can be reallocated, to track the tasks, separate integer array used
- Priority to panel computation (prioritize critical path)
- Need to track temporary buffers used to store non-local blocks and deallocate

```
#pragma omp parallel
#pragma omp master
{
  // start pipeline (factor 1st panels)
  factorPanel(0, A);
  for (int k = 1; k < A.getNt(); k++) {
    lookaheadUpdateA(k-1, A);

    // factor next panel
    factorPanel(k, A);

    // update remaining submatrix
    // using current (k-1)th panel
    remainingUpdateA(k-1, A);
  }
}
```



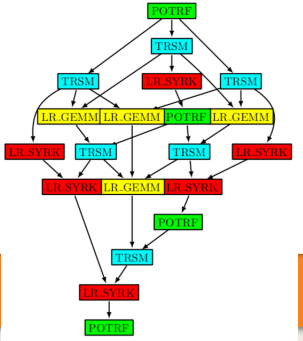

```
#pragma omp task priority(1) \
                 depend(in:tileB[0:1]) \
                 depend(inout:tileA[0:1])
{
 // factor diagonal
 if (A.isLocalRow(k) || A.isLocalCol(k)) {
     A.factorDiagBlock( k );
 }
 // compute off-diagonal L
 if (A.isLocalCol( k )) {
     for (int i = k+1; i < A.getMt(); i++) {
         if (A.isLocalRow( i )) {
             #pragma omp task priority(1)
             {
                 A.computeL(i, k);
             }
         }
     }
     #pragma omp taskwait

     if (!A.isLocal(k, k)) {
         A.freeBuffer(k, k);
     }
 }
 // broad cast tiles in panel along the rows
 A.iBcastL(k);
}
```

Left: main loop
Right: Diagonal factor body

PaRSEC DTD

- Describe the data distribution to PaRSEC
- Using DTD API to implement the algorithm is straightforward, like sequential code
- The runtime system builds the DAG of Tasks in the background, and schedules tasks execution and data movements, in parallel
- Challenge for BLR: data size changes, need to reallocate, and send data of variable sizes. Supported in PaRSEC runtime




```
for(k = 0; k < NT; k++){
  // diagonal DGETRF
  insert_task(taskpool, parsec_dgetrf,
    1, "getrf",
    sizeof(int)   , &k                  ,VALUE,
    PASSED_BY_REF, TILE_OF(A, k, k) ,INOUT | AFFINITY,
    PASSED_BY_REF, TILE_OF(IP, k, 0),OUTPUT,
    PARSEC_DTD_ARG_END);
  if(k < NT-1){
    for(int i = k+1; i < NT; i++){
      insert_task(taskpool, parsec_dtrsm_l,
                  ...);
      insert_task(taskpool, parsec_dtrsm_u,
                  ...);
    }
    data_flush(dtd_tp, TILE_OF(A, k, k));
    data_flush(dtd_tp, TILE_OF(IP, k, 0));

    for(int i = k+1; i < NT; i++){
      for(int j = k+1; j < NT; j++){
        insert_task(taskpool, parsec_dgemm,
                    ...);
      }
    }
  }
}
```

DTD interface of the LU algorithm

Update data size to send in the body

PaRSEC PTG

- With the distributed data collection in place, halfway there!
- Need to specify the data flows between the tasks in JDF format
- No need for task discovery, expressed explicitly and avoid exploring the whole task graph at once
- Different parallelization philosophy, driven by data dependencies and no artificial synchronization points
- Require decent amount of effort to convert existing algorithms written in MPI+OMP

```
dgetrf(k)
k = 0 .. NT

: descA(k, k) //locality

RW A  <- (FIRST) ? descA(k,k)
      <- (!FIRST) ? C dgemm(k_prev, DIAG, DIAG)

      -> (END>=START) ? A dtrsm_l(k, START..END)
      -> (END>=START) ? A dtrsm_u(k, START..END)

RW IP <- IP ipiv_in(k)  [type = PIVOT count = NB]
      -> IP ipiv_out(k) [type = PIVOT count = NB]

/* Priority */
;1

BODY
{
  // Factorizing diagonal block (k, k)
  int mb = descA.nbi[k];
  double *dA = &(((double*)A)[1]);
  iinfo = LAPACKE_dgetrf(LAPACK_COL_MAJOR,
                         mb, mb, dA, mb, ipiv);
}
```

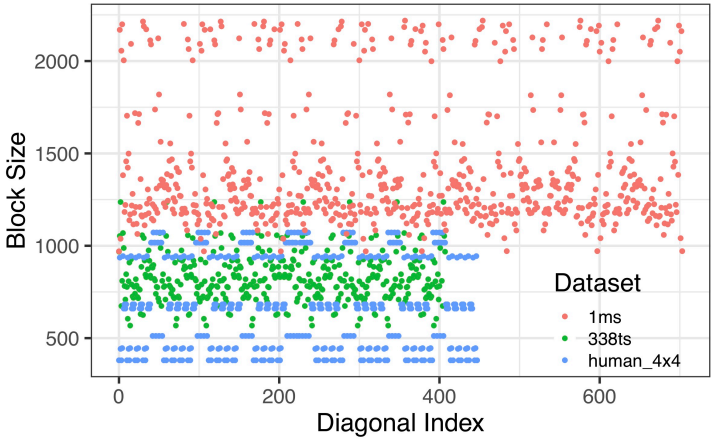
JDF of the DGETRF task flow specification, modify data size to send in BODY

Results – Experiment Setup

- Implemented using software package ppohBEM, in particular, HACApK
- XSEDE PSC Bridge cluster (2 Intel Haswell E5-2695 v3 per node, 14 cores per CPU)
- 3 datasets of different sizes and shapes

Best Configurations for each approach

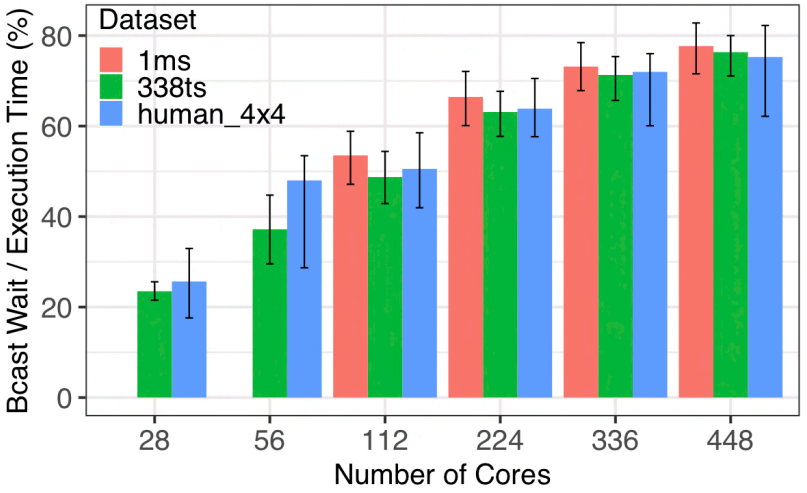
- One MPI process per core, oversubscribe of factor 3 for AMPI
- Two MPI processes per socket, 4 per node for MPI+OMP (NUMA issue for OMP)
- One process per node for PaRSEC, with one core dedicated to communication



Datasets we used, number of blocks and the diagonal blocks size. Matrix sizes are 1M, 338K and 310K respectively

Results – Imbalance

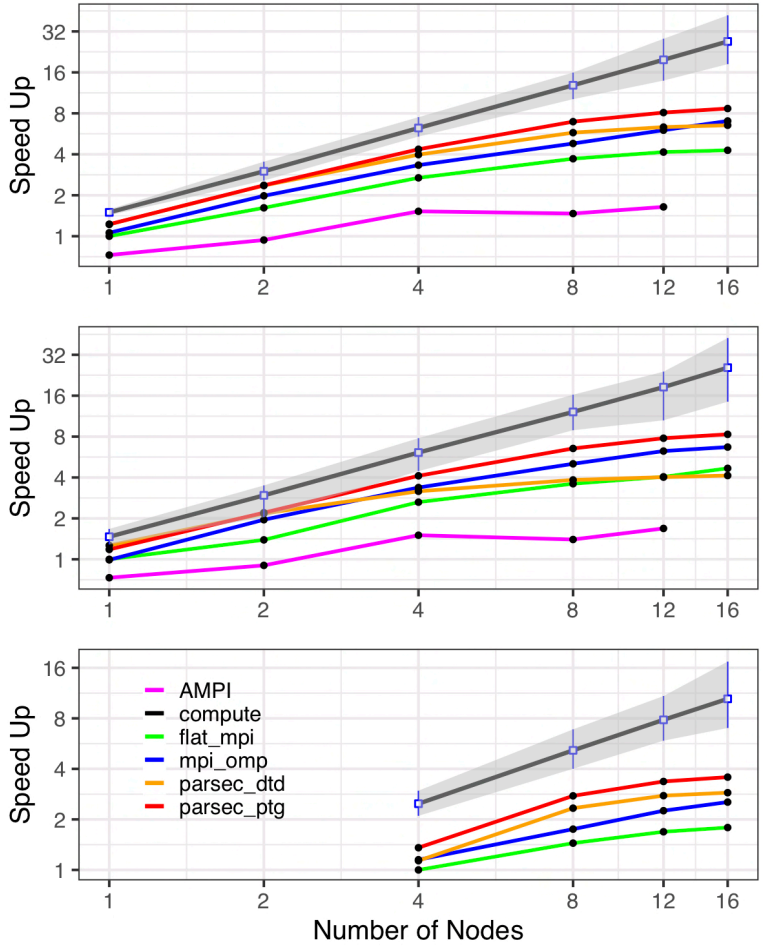
- Imbalance workload among the processes during each step of the factorization
- But the error bar indicates that the total computation among the processes are reasonably balanced
- More than 70% execution time is idling for some cases!



Flat MPI run with wait before broadcast, sum the wait time for each process. Average, with min and max as error bar

Results – Performance

- Performance result for the 5 models, flat MPI performance on 1 node used as baseline
- Black line shows the average total compute time for each process (from flat MPI runs)
- AMPI not doing well, all other models improve on flat MPI, with relaxed synchronization
- DTD performs well, then degrades, task discovery overhead? (a better strategy for task-trimming might help)
- MPI+OMP relaxed the strong control in flat MPI, and PTG removed that



Top: 338ts Middle: human_4x4 Bottom: 1ms

Conclusion

- Several features needed for efficient BLR algorithm:
  - handle load imbalance
  - dynamic data reallocation
  - variable send/receive data size
  - communication computation overlap

- Provide a guide to transition into task-based runtime system
- Performance can be further improved (e.g. a new data distribution)

- Future works
  - Use Task-aware MPI (TAMPI)
  - Quantify the temporary memory usage of PaRSEC runtime
  - GPU/accelerator which will change the comp/comm dynamics