Comparing performance of \( s \)-step and pipelined GMRES on distributed-memory multicore CPUs

Ichitaro Yamazaki*, Mark Hoemmen†, Piotr Luszczek*, Jack Dongarra*

*University of Tennessee, Knoxville, USA
†Sandia National Laboratories, Albuquerque, New Mexico, USA

SIAM Annual Meeting
Pittsburgh, Pennsylvania, 07/14/2017
Avoid or Hide Communication in Krylov (inter-process)

- Krylov is a powerful method for solving large-scale linear systems
  - is based on subspace projection
  - generates a basis vector at each iteration

- Krylov uses SpMV (+Precon) and Orth to generate each basis vector
  - P2P of SpMV and all-reduce of Orth can become bottleneck

- s-step aims to “avoid” them by generating s vectors at a time
  - latency reduced by a factor of $s \times$

- pipeline tries to “hide” them by pipeline iterations
  - max speedup of $2 \times$, but maybe more through pipelining
Performance comparison

- distributed CPUs with multicores on node

Programming paradigm

- performance
  - thread-parallelism on multicores
  - non-blocking collective to progress in background

- productivity, maintainability (and hopefully “portability”)
  - hide details of thread-parallelization
  - no application thread to ensure non-blocking collective

- two implementations
  1. MPI’s progress thread for non-blocking collective
     + threaded comp kernels (i.e., MKL)
  2. insert-task (using shared-memory QUARK runtime)
GMRES solvers

- standard

- pipelined
  - s-step with standard SpMV+Precond
    - P2P for each SpMV, instead of Matrix Power Kernel (MPK)
    - in our experiment, main improvement from block-orth
    - MPK has overheads, e.g., redundant store/comp and preconditioning
      → focus on reducing global collectives, and not on P2P
    - pipelined focuses on hiding global all-reduce for Orth
    - nice comparison between s-step and pipelined

- pipelined s-step
  aka, pipelining with block ortho, or s-step with pipelined block orth.
Why combine pipeline and s-step?

- **s-step (without MPK):**
  - improvement even on small number of nodes when latency is significant
  - also reduces intra-proc comm using BLAS-3
  - still block synchronous

- **pipeline**
  - hide latency
  - additional computation for “Change-of-basis” (∼ 50% of Orth)
    - improvement only on large number of nodes

- combine the two?
pipelined $t$-step GMRES with MPI (step size $t$, pipeline depth $\ell$)

```latex
\begin{algorithm}
\begin{algorithmic}
  \State \text{for $j = 1, 1 + t \cdot \ell, \ldots, m$ do}
  \State \hspace{1em} \text{generate $t$ basis vectors}
  \State \hspace{2em} \text{for $k = 1, 2, \ldots, t$ do}
  \State \hspace{3em} \text{SpMV with P2P and change-of-basis, $i := j + k - t \cdot \ell + 1$}
  \State \hspace{3em} \text{$v_{j+k} := AM^{-1}v_{j+k-1}$ (MPI.Isend and MPI.Irecv with neighbors)}
  \State \hspace{3em} \text{generate $h_{1:i-1,i}$}
  \State \hspace{3em} \text{$v_{j+k} := (v_{j+k} - V_{i:i+k-1}h_{1:i-1,i-1})/h_{i,i-1}$ (BLAS-2)}
  \State \hspace{1em} \If{$j > t \cdot \ell$}
  \State \hspace{2em} $k := j - t \cdot \ell + 1$
  \State \hspace{2em} \text{finish block-ortho $Q_{k:k+t-1}$ with MPI.Wait}
  \State \hspace{3em} \text{2.1 update $R_{1:k+t,k:k+t-1}$}
  \State \hspace{3em} \text{2.2 block orthogonalize (BLAS-3)}
  \State \hspace{3em} $Q_{k:k+t-1} := (V_{k:k+t-1} - Q_{1:k-1}R_{1:k-1,k:k+t-1})R_{1:k+t-1,k:k+t-1}^{-1}$
  \State \hspace{3em} \text{2.3 apply change-of-basis to next vector (extra computation)}
  \State \hspace{3em} \text{generate $h_{1:k,k}$}
  \State \hspace{3em} $v_{j+1} := v_{j+1} - V_{k:k+t-1}h_{1:k-1,k-1}/h_{k,k-1}$ (BLAS-2)
  \State \hspace{1em} \EndIf
  \State \hspace{1em} \text{3 start block-ortho $Q_{j+1:j+t}$ against $Q_{1:j}$ with non-block reduce}
  \State \hspace{2em} $R_{1:j+t,j:j+t} := Q_{1:j+1}^{T}Q_{j+1:j+s}$ (BLAS-3 and MPI.Iallreduce)
  \EndFor
\end{algorithmic}
\end{algorithm}
```

- BLAS-3 for orthogonalization
- pipelined to hide all-reduces over $t\ell$ iterations
- extra computation to maintain stability (pipeline depth $t \cdot \ell$)
Why tasks?

- fork-join in standard, and also in $s$-step potential for scheduling local and boundary tasks from different steps in MPK
- pipeline may provide opportunity for runtime
  - parallel execution of independent tasks
  - overlap/pipeline computation and communication

- SpMV, GEMV, GEMM are distributed and threaded
**QUARK implementation**

- shared-memory runtime based on “insert-task” model (similar to OpenMP)

- each process uses QUARK to schedule its **comp** and **comm** tasks on shared-memory multicores
  
  - **comp task**: implicitly split local submatrix into “tiles” (1D block row) each task works on tiles on a separate core
  
  - **comm task**: calls “blocking” MPI P2P (MPI_Isend/MPI_Irecv, then MPI_Wait) for SpMV and all-reduce (MPI_Allreduce) for Orth are wrapped into tasks

- some cores may be idle, but
  
  - “priority” tag to reduce the idel time
  
  - may be non-significant on manycores or with GPUs

- **comm** and **comp** should overlap, and

- parallel execution of independent tasks
  
  - block size as a tuning parameter
QUARK P2P Comm wrapper for SpMV

- setup data dependencies
- one task per communication

```c
void quark_SpMV_Gather(sparse_desc A, Complex64_t *g) {
    Task *task = Task_Init(quark, core_SpMV_Gather_quark, task_flags);
    ...

    // INPUT on local “underlap” tiles with vector elements to be sent
    for (int k=0; k<num_send_blocks; k++)
        Pack_Arg(task, sizeof(Complex64_t)*A.mb, &g[send_blocks[k+1]], INPUT);

    // OUTPUT on non-local “ghost” tiles with vector elements to be received
    for (int k=0; k<num_recv_blocks; k++)
        Pack_Arg(task, sizeof(Complex64_t)*A.mb, &g[recv_blocks[k+1]], OUTPUT);
}
```

- data access types for process
  (INPUT, OUTPUT, INOUT)
- define data-dependencies with for-loop
  based on the sparsity pattern of the matrix
QUARK P2P Core routine for SpMV

- prepare buffer, MPI_Isend and MPI_Irecv, and then MPI_Wait

```c
void core_SpMV_Gather(int iter, sparse_desc A, Complex64_t *g) {
    for (each neighbor process, p) {
        // pack local vector elements to be send
        int count = num_send_vecs[p];
        for (i=0; i<count; i++)
            send_buffer[send+i] = g[A.send_vecs[p][i]];

        // start MPI_Isend
        MPI_Isend(&send_buffer[send], count, MPI_DOUBLE, p,
                   iter, MPI_COMM_WORLD, &(A.send[p][request_id]));
        send += count;
    }
    // set up MPI_Irecv
    ...

    // wait for MPI_Isend
    for (each neighbor process, p)
        MPI_Wait(&(A.send[p][request_id]), &status);

    // wait for MPI_Irecv and unpack message
    for (each neighbor process, p) {
        MPI_Wait(&(A.recv[p][request_id]), &status);
        for (i=0; i<count; i++)
            g[A.recv_vecs[p][i]] = recv_buffer[send+i];
    }
}
```

- same as MPI implementation
- for all-reduce: we pack, MPI_Allreduce, and unpack
QUARK wrapper: SpMV + GEMV

- each task work on tiles (multiple comp tasks per SpMV)
- neighborhood data dependencies (local or ghost) for tile

```c
void quark_SpMV_Gemv( . . ) {
    // subroutine to be executed
    Task *task = Task_Init(quark, CORE_zspmv_gemv_quark, task_flags );
    // arguments for SpMV, y = A*x
    ...
    i-th local tile of output vector
    Pack_Arg(task, sizeof(Complex64_t)*mb, y, INOUT | LOCALITY);
    // dependency for i-th input tile on neighboring tiles
    for (each neighbor tiles, \( k \)) {
        int offset = neighbors[i][k+1];
        Pack_Arg(task, sizeof(Complex64_t)*mbk, &x[offset], INPUT);
    } // arguments for GEMV, \( w = Z^*y \)
    Pack_Arg(task, sizeof(Complex64_t)*mb*n,Z, INPUT);
    Pack_Arg(task, sizeof(Complex64_t)*mb, w, INOUT);
    ...
}
```

- data locality is crucial for performance
  - “locality” tag to schedule on core close to data
  - computational kernels are fused into one task
    also to reduce scheduling overhead
GMRES with QUARK

for (j = 0; j < restart; j++) {
    // neighborhood comm for SpMV
    quark_SpMV_gather(...);

    // SpMV: Q(:, j+1) := A*Q(:, j)
    // GEMV: H(:, j) := Q(:, 0:j)'*Q(:, j+1));
    for each local tiles do
        quark_SpMV_Gemv(...);

    // Orth: local and global reduce, H(1:j, j) := \sum_{k=0}^{mt-1} T(k)
    quark_GeAdd_reduce(...);

    // GEMV: Q(:, j+1) -= Q(:, 1:j)'*H(1:j, j)
    // DOT: T(i) := Q(i, j+1)'*Q(i, j+1)
    for each local tiles do
        quark_Gemv_Dot(...);

    // normalize: local and global reduce, H(j+1, j) := \sum_{k=0}^{mt-1} T(i)
    quark_GeAdd_reduce(...);

    for each local tile do
        quark_laScal_copy(...);
end for

- looks similar to MPI implementation
  but is task based (parallel execution of independent tasks)
- block size as tuning parameter
2nd implementation:
non-blocking MPI collective + threaded MKL

- converted QUARK implementation
  - some changes e.g., `MPI_Iallreduce` with `MPI_Wait`, draining pipeline
  - directly call core routines without wrapper,
    i.e., threaded MKL, no specialized kernels
Experiment setups

- Tsubame supercomputer at Tokyo Tech.
  - two six-core Intel Xeon CPUs per node
  - 80Gbps QDR InfiniBand

- threaded MKL (BLAS, LAPACK, Sparse BLAS)
  MKL_NUM_THREADS=1 with QUARK

- MPICH 3.2 (for overlap, and may not for performance)
  - MPI_Iallreduce (implemented using TCP/IP) for MPI implementation
  - thread support (configured with --enable-threads=multiple)
  - MPI_THREAD_MULTIPLE support for QUARK and MPI implementations

- bind process to specific cores for both QUARK and MKL threads

- leave one spare core per process for MPI’s progress thread with MPI implementation

- mostly simple model problems just to understand their performance
### MPI benchmarks: overlap of MPI_Iallreduce with comp (IMB)

<table>
<thead>
<tr>
<th>#bytes</th>
<th>$t_{ovrl}[\mu\text{sec}]$</th>
<th>$t_{pure}[\mu\text{sec}]$</th>
<th>$t_{CPU}[\mu\text{sec}]$</th>
<th>overlap [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>312.37</td>
<td>242.53</td>
<td>272.48</td>
<td>74.37</td>
</tr>
<tr>
<td>16</td>
<td>268.53</td>
<td>225.00</td>
<td>254.62</td>
<td>82.91</td>
</tr>
<tr>
<td>32</td>
<td>264.67</td>
<td>222.07</td>
<td>251.30</td>
<td>83.05</td>
</tr>
<tr>
<td>64</td>
<td>281.10</td>
<td>237.46</td>
<td>249.84</td>
<td>82.53</td>
</tr>
<tr>
<td>128</td>
<td>267.30</td>
<td>227.92</td>
<td>253.52</td>
<td>84.47</td>
</tr>
<tr>
<td>256</td>
<td>278.94</td>
<td>227.63</td>
<td>265.70</td>
<td>80.69</td>
</tr>
</tbody>
</table>

- good overlap (may be slower, and may not reflect solver)
- progress thread is enabled with one spare core per process
- GMRES reduces $1 \times 1 \sim 10 \times 30$ numerical values
  - $8 \sim 2400$ bytes
**MPI benchmark:** pipelining all-reduces

<table>
<thead>
<tr>
<th>#bytes</th>
<th>80</th>
<th>160</th>
<th>240</th>
<th>320</th>
<th>400</th>
<th>480</th>
<th>560</th>
<th>640</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 calls MPI_Iallreduce followed by MPI_Waitall, progress threads</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n_p = 60$</td>
<td>4.62</td>
<td>4.86</td>
<td>5.55</td>
<td>6.02</td>
<td>6.10</td>
<td>6.83</td>
<td>6.62</td>
<td>6.45</td>
</tr>
<tr>
<td>$n_p = 120$</td>
<td>4.22</td>
<td>4.81</td>
<td>6.32</td>
<td>5.98</td>
<td>6.43</td>
<td>6.76</td>
<td>7.11</td>
<td>6.48</td>
</tr>
</tbody>
</table>

10 calls to MPI_Allreduce from $n_t$ threads per process, $n_p = 20$.  

<table>
<thead>
<tr>
<th>$n_t =$</th>
<th>2</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.74</td>
<td>8.79</td>
<td>8.72</td>
</tr>
<tr>
<td>9.66</td>
<td>8.97</td>
<td>9.26</td>
</tr>
<tr>
<td>9.77</td>
<td>8.72</td>
<td>8.50</td>
</tr>
<tr>
<td>9.42</td>
<td>9.26</td>
<td>10.58</td>
</tr>
<tr>
<td>9.75</td>
<td>8.50</td>
<td>10.87</td>
</tr>
<tr>
<td>9.32</td>
<td>10.58</td>
<td>10.50</td>
</tr>
<tr>
<td>9.61</td>
<td>10.87</td>
<td>10.50</td>
</tr>
<tr>
<td>9.25</td>
<td>10.50</td>
<td>10.50</td>
</tr>
</tbody>
</table>

- Time over one all-reduce (12 cores per node) -

- 1.00 means “perfect” pipeline (not possible due to bandwidth)  
- $\geq 10.00$ means “no” pipeline

- **MPI_Allreduce** does not seem to pipeline (using different communicator per thread)  
- **MPI_Iallreduce** seems to do a bit better
Convergence rate on 12 processes: 5-pts 2D Laplace ($n_x = 512$) (2 nodes, six processes per node, one thread per process)

- all solvers converge equivalently in term of iteration counts even with preconditioner
- for remaining slides, 20 restart cycles of GMRES(30) (Newton basis, no precond)
Convergence rate on 12 processes: 5-pts 2D Laplace \((n_x = 512)\)
(2 nodes, six processes per node, one thread per process)

- all solvers converge equivalently in term of iteration counts even with preconditioner
- for remaining slides,
  20 restart cycles of GMRES(30) (Newton basis, no precond)
**Performance comparison:** 5-pts 2D Laplace \((n_x = 1024)\)  
(six processes per node, one thread per process)

- \(s\)-step reduces both intra and inter **comm**
- pipeline improves GMRES and is expected to improves \(s\)-step at a larger scale
- combining two may obtain the best performance at a large-scale
Performance comparison: 27-pts 3D problems ($n_x = 128$)

<table>
<thead>
<tr>
<th>$\ell$</th>
<th>$s$</th>
<th>number of processes</th>
<th>60</th>
<th>120</th>
<th>180</th>
<th>240</th>
</tr>
</thead>
<tbody>
<tr>
<td>GMRES</td>
<td>–</td>
<td>–</td>
<td>2.10 (1.00)</td>
<td>1.25 (1.00)</td>
<td>0.88 (1.00)</td>
<td>0.64 (1.00)</td>
</tr>
<tr>
<td>pipelined</td>
<td>2</td>
<td>–</td>
<td>2.36 (0.89)</td>
<td>1.36 (0.92)</td>
<td>0.88 (1.00)</td>
<td>0.68 (1.00)</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>–</td>
<td>2.32 (0.91)</td>
<td>1.27 (0.98)</td>
<td>0.84 (1.05)</td>
<td>0.65 (1.05)</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>–</td>
<td>2.20 (0.95)</td>
<td>1.19 (1.05)</td>
<td>0.83 (1.06)</td>
<td>0.61 (1.11)</td>
</tr>
<tr>
<td>s-step</td>
<td>–</td>
<td>5</td>
<td>1.85 (1.14)</td>
<td>1.06 (1.18)</td>
<td>0.74 (1.19)</td>
<td>0.49 (1.38)</td>
</tr>
<tr>
<td></td>
<td>–</td>
<td>10</td>
<td>1.75 (1.20)</td>
<td>1.04 (1.20)</td>
<td>0.70 (1.26)</td>
<td>0.47 (1.45)</td>
</tr>
</tbody>
</table>

- Time in seconds (speedups over GMRES) -

- lower speedups compared to 2D problems (heavier SpMV) -
**Performance comparison:** U. of Florida Matrix collection

<table>
<thead>
<tr>
<th></th>
<th>$n$ (M)</th>
<th>$\frac{nnz}{n}$</th>
<th>time</th>
<th>pipelined</th>
<th>s-step</th>
<th>pipelined s-step</th>
</tr>
</thead>
<tbody>
<tr>
<td>G3_Circuit</td>
<td>1.6</td>
<td>4.8</td>
<td>0.43</td>
<td>1.31</td>
<td>1.48</td>
<td>1.55</td>
</tr>
<tr>
<td>thermal2</td>
<td>1.2</td>
<td>7.0</td>
<td>0.43</td>
<td>1.54</td>
<td>1.60</td>
<td>1.65</td>
</tr>
<tr>
<td>atmosmodd</td>
<td>1.3</td>
<td>6.9</td>
<td>0.74</td>
<td>1.78</td>
<td>1.95</td>
<td>1.99</td>
</tr>
</tbody>
</table>

- Speedups over GMRES (240 processes) –

- $s$-step reduces both intra and inter **comm**

- pipeline improves GMRES and is expected to improves $s$-step at a larger scale

- combining two may obtain the best performance at a large-scale
Thread-parallelization: threaded MKL+MPI or QUARK? (1 process/socket)

- QUARK could utilize cores better obtained higher performance on small number of processes
- but seems to lose its advantage on a larger number of processes scheduling overhead, pipelining?
Final slide

- Studied two implementations of pipelined $s$-step GMRES

Current work: DOE ECP PEEKS project

- ECP applications on Exascale architectures
  much heavier SpMV, running with manycores/accelerators

- Implementation
  - Trillinos components (Tpetra, Teuchos, Kokkos)
    collaboration with Sandia’s solver group
  - Other solvers (CG, BiCGStab, and Lanczos)

- Performance
  - Other MPIs (e.g., Intel MPI, OpenMPI)
  - Other machines with GPUs/manycores on a node
    (e.g., Titan, Cori, Theta)
Acknowledgements

**ECP PEEKS:** This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, in support of the nations exascale computing imperative.

Thank you!!