Comparing performance of s-step and pipelined GMRES

on distributed-memory multicore CPUs

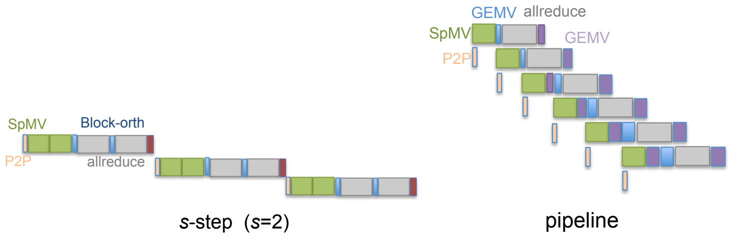
Ichitaro Yamazaki[*], Mark Hoemmen[†], Piotr Luszczek[*], Jack Dongarra[*]

[*]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 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

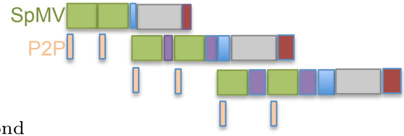
- distributed CPUs with multicores on node

Programming paradigm

- performance
  - thread-parallelism on multicores
  - non-blocking collective to progress in background
- productivity, maintainablity (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**



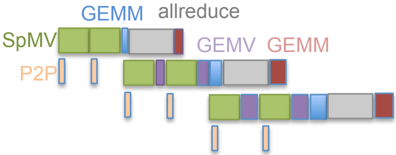
- **pipelined**

- ***s*-step** with standard SpMV+Precond
  - P2P for each SpMV, instead of Matrix Power Kernel (MPK)
  - in our experiement, 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" ($\sim$ 50% of Orth)
    ▷ improvement only on large number of nodes
- combine the two?



pipelined t-step GMRES with MPI (step size t, pipeline depth ℓ)

for $j = 1, 1 + t \cdot \ell, \ldots, m$ **do**

1 generate t basis vectors

  for $k = 1, 2, \ldots, t$ **do**

   SpMV with P2P and change-of-basis, $i := j + k - t \cdot \ell + 1$

   $\mathbf{v}_{j+k} := AM^{-1}\mathbf{v}_{j+k-1}$ (MPI_Isend and MPI_Irecv with neighbors)

   generate $\mathbf{h}_{1:i-1,i}$

   $\mathbf{v}_{j+k} := (\mathbf{v}_{j+k} - V_{i:i+k-1}\mathbf{h}_{1:i-1,i-1})/h_{i,i-1}$ (BLAS-2)

 if $j > t \cdot \ell$ **then**

  $k := j - t \cdot \ell + 1$

2 finish block-ortho $Q_{k:k+t-1}$ with MPI_Wait

  2.1 update $R_{1:k+t,k:k+t-1}$

  2.2 block orthogonalize (BLAS-3)

   $Q_{k:k+t-1} := (V_{k:k+t-1} - Q_{1:k-1}R_{1:k-1,k:k+t-1})R^{-1}_{k:k+t-1,k:k+t-1}$

  2.3 apply change-of-basis to next vector (extra computation)

   generate $h_{1:k,k}$

   $\mathbf{v}_{j+1} := \mathbf{v}_{j+1} - V_{k:k+t-1}h_{1:k-1,k-1})/h_{k,k-1}$ (BLAS-2)

 end if

3 start block-ortho $Q_{j+1:j+t}$ against $Q_{1:j}$ with non-block reduce

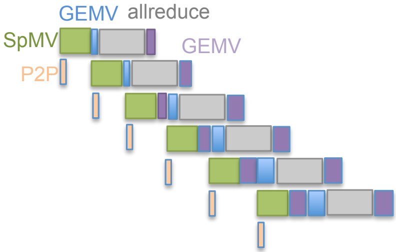
 $R_{1:j+t,j:j+t} := Q^T_{1:j+1}Q_{j+1:j+s}$ (BLAS-3 and MPI_Iallreduce)

end for

- 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

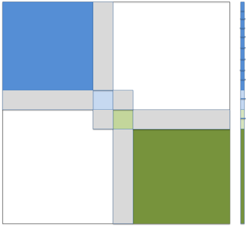
- setup data dependencies
- one task per communication

```
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`

```
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

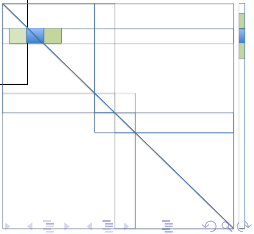
QUARK wrapper: SpMV + GEMV

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

```
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)

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

- 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

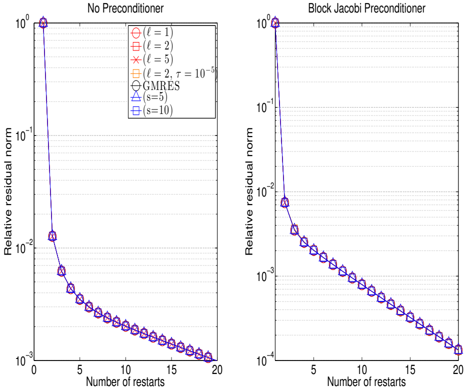
MPI benchmark: pipelining all-reduces

#bytes	80	160	240	320	400	480	560	640
10 calls MPI_Iallreduce followed by MPI_Waitall, progress threads								
$n_p = 60$	4.62	4.86	5.55	6.02	6.10	6.83	6.62	6.45
120	4.22	4.81	6.32	5.98	6.43	6.76	7.11	6.48
10 calls to MPI_Allreduce from n_t threads per process, $n_p = 20$.								
$n_t = 2$	9.74	9.66	9.77	9.42	9.75	9.32	9.61	9.25
5	8.79	8.97	8.72	9.26	8.50	10.58	10.87	10.50

- 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 pipline
  (using different communicator per thread)
- MPI_Iallreduce seems to do a bit better

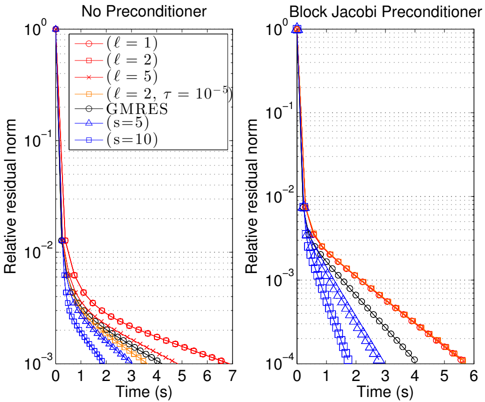
Convergence rate on 12 processes: 5-pts 2D Laplace ($n_x = 512$)
(2 nodes, six processes per node, one thread per process)



- 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)

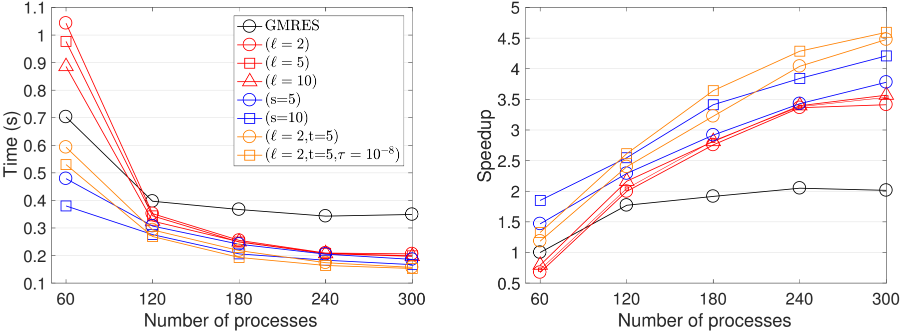
Convergence rate on 12 processes: 5-pts 2D Laplace ($n_x = 512$)

(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 ($n_x = 128$)

ℓ	s	60	120	180	240
		number of processes			
GMRES					
–	–	2.10 (1.00)	1.25 (1.00)	0.88 (1.00)	0.64 (1.00)
pipelined					
2	–	2.36 (0.89)	1.36 (0.92)	0.88 (1.00)	0.68 (1.00)
5	–	2.32 (0.91)	1.27 (0.98)	0.84 (1.05)	0.65 (1.05)
10	–	2.20 (0.95)	1.19 (1.05)	0.83 (1.06)	0.61 (1.11)
s-step					
–	5	1.85 (1.14)	1.06 (1.18)	0.74 (1.19)	0.49 (1.38)
–	10	1.75 (1.20)	1.04 (1.20)	0.70 (1.26)	0.47 (1.45)
pipelined s-step					
2	5	2.03 (1.03)	1.13 (1.11)	0.78 (1.13)	0.51 (1.33)

- Time in seconds (speedups over GMRES) -

- lower speedups compared to 2D problems (heavier SpMV)

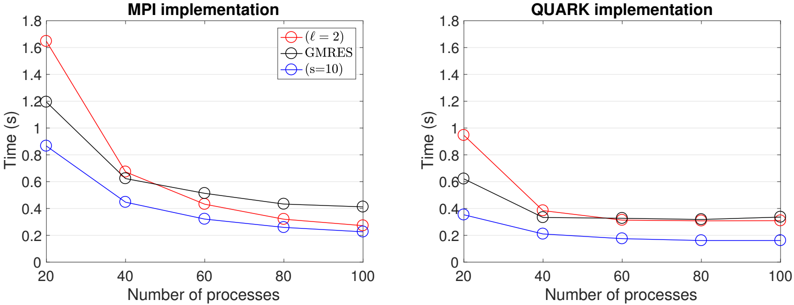
Performance comparison: U. of Florida Matrix collection

	n (M)	$\frac{nnz}{n}$	time	pipelined	s-step	pipelined s-step
G3_Circuit	1.6	4.8	0.43	1.31	1.48	1.55
thermal2	1.2	7.0	0.43	1.54	1.60	1.65
atmosmodd	1.3	6.9	0.74	1.78	1.95	1.99

- 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
- Implementaion
  - 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!!