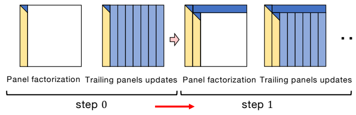
Task-parallel algorithms for matrix factorizations

Tomohiro Suzuki (stomo@yamanashi.ac.jp), University of Yamanashi, Japan

Introduction

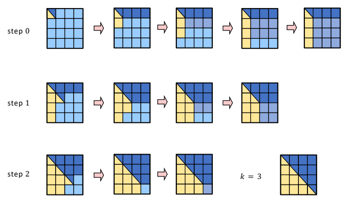
- Goal: Improve the resource usage of the highly-parallel system
- OpenMP : Thread parallel programming in Shared memory env.
  - Past: Data parallel ⇨ Present: Task parallel
  - `task` construct with `depend` and `priority` clause
- Matrix factorization
  - One-sided: Cholesky, LU, QR
  - Flop counts: $O(N^3)$
- In a highly-parallel environment, the 1D/2D block algorithm with task parallel fashion is effective for matrix factorization?

- (1D) block algorithm



- Trailing matrix is split into multiple panels and updated for each panel

- (2D block) tile algorithm ⇨ PLASMA https://bitbucket.org/icl/plasma/



- Target matrix is divided into $p \times q$ tiles
- Factorize and update each one or a couple of tiles
- Asynchronous execution of many fine-grained tasks

Pseudo code

- 1D block matrix factorization with OpenMP `task` construct

```
#pragma omp parallel {
    #pragma omp single {
        for (int i=0; i<p; i++) {
            #pragma omp task depend(inout: A[i*nb*m:m*nb]) prioity(P1)
            Panel_Factorization(m-i*nb, nb, A+(i*nb+i*nb*lda));

            for (int j=i+1; j<p; j++)
                #pragma omp task depend(in: A[i*nb*m:m*nb])
                                 depend(inout: A[j*nb*m:m*nb]) priority(P2)
                Panel_Update(m-i*nb, nb, A+(i*nb+j*nb*lda));
        }
    } // End of single
} // End of parallel
```

- `p`: # of panels
- `nb`: panel width

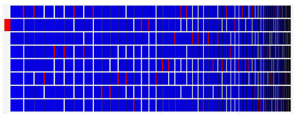
- Priority variant
  1. P1: none, P2: none
  2. P1: `p`, P2: none
  3. P1: `p`, P2: `max(p/2,p-j)`

- 2D block QR factorization (PLASMA)

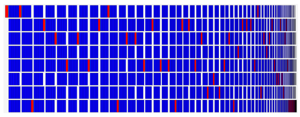
```
#pragma omp parallel {
    #pragma omp single {
        for (int k=0; k<p; k++) {
            #pragma omp task depend(inout:A(k,k)) depend(out:T(k,k))
            GEQRT( A(k,k), T(k,k) );

            for (int j=k+1; j<p; j++)
                #pragma omp task depend(in:A(k,k), T(k,k)) depend(inout:A(k,j))
                LARFB( A(k,k), T(k,k), A(k,j) );

            for (int i=k+1; i<p; i++) {
                #pragma omp task depend(inout:A(k,k), A(i,k)) depend(out:T(i,k))
                TSQRT( A(k,k), A(i,k), T(i,k) );

                for (int j=k+1; j<p; j++)
                    #pragma omp task depend(in:A(i,k),T(i,k)) depend(inout:A(k,j),A(i,j))
                    SSRFB( A(i,k), T(i,k), A(k,j), A(i,j) );
}}}}
```

Execution trace

1D block LU variant 1

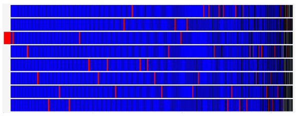


1D block QR variant 1

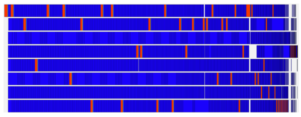


factorizaton
update

1D block LU variant 2

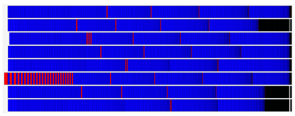


1D block QR variant 2

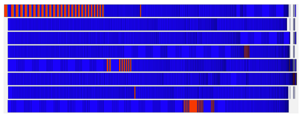


factorizaton
update

1D block LU variant 3

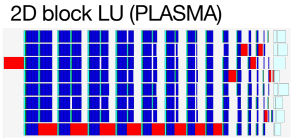


1D block QR variant 3



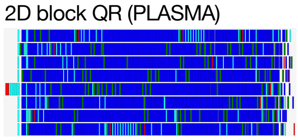
factorizaton
update

2D block LU (PLASMA)



panel factorizaton | gemm | swp1 | swp2

2D block QR (PLASMA)



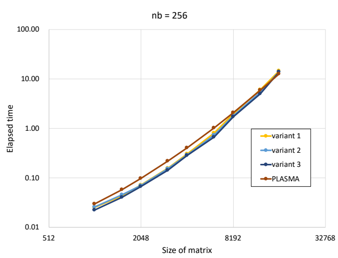
GEQRT | TSQRT | LARFB | SSRFB

Experimental env.

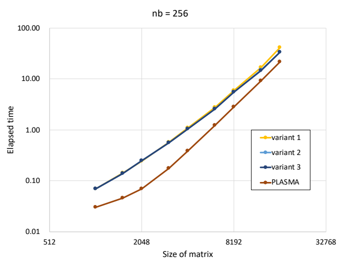
- CPU: Intel Core i7-6900K (8 core, @3.2GHz)
- Compiler: GNU C++ 9.2.1
- BLAS, LAPACK: MKL 2019.5.281 (core, lp64, sequential)
- OpenMP: libgomp

Performance results

LU factorization



QR factorization



Remarks

- 1D block algorithm
  - Sequential code + task & depend = `task parallel code` (variant1),
  - However, data dependency analysis is required.
  - Look-ahead does not deepen even if prioritizing only decomposition (variant2)
  - To achieve deep look-ahead, update tasks must also be properly prioritized. (variant3)
  - It lacks inherent parallelism.
- 2D block algorithm (tile algorithm)
  - In general, many fine-grained tasks that can be executed in parallel can be generated.
  - Without improving the pivoting strategy, the performance improvement of LU factorization cannot be achieved.
  - QR factorization shows high performance.
  - It is mandatory to tune the tile size `nb`.