Scalable Dynamic Dag Building for data-flow task-based Runtime Systems

Reazul Hoque
9TH JLESC WORKSHOP
April 15, 2019, Knoxville, TN, USA
Need for a Runtime

• Parallel programming is complex

• Application developers need to:
  • Express large degree of parallelism
  • Consider data placement and movement (distributed)
  • Manage heterogeneity
  • Tackle communication bottleneck
  • Manage various memory types
- **Parallel Runtime Scheduling and Execution Controller**
- Executes a graph of tasks generated based on a dataflow representation of a program (high level runtime)
- Task-based stateful (re-execution, delay, migration, stealing, ...)
- Supports heterogeneous architecture and distributed memory machines
- **Communications are implicit, overlapped**
- How tasks are presented to the runtime is critical from many aspects
Motivation

- Building dynamic DAG offers flexibility
- Apriori information about task dependencies is not required
- Analysis of overhead of dynamic and static approach to DAG building
- Promise of composition of multiple interfaces (static and dynamic)
Dynamic Task Discovery (DTD)

• Task-graph (DAG) is built dynamically during runtime (similar to OpenMP but using an API)
• Easy to express algorithms (API based)
• Allows users to write sequential code
• Shares the entire software infrastructure with other DSL/Interface (such as PTG):
  • Schedulers
  • Communication Engine
  • Data Interface
  • Heterogeneous and Architecture-aware support
for( k = 0; k < total; k++ ) {
    parsec_insert_task( "Potrf", TILE_OF(A, k, k), INOUT | AFFINITY);
    for( m = k+1; m < total; m++ ) {
        parsec_insert_task( "Trsm",
            TILE_OF(A, k, k), INPUT,
            TILE_OF(A, m, k), INOUT | AFFINITY);
    }
    for( m = k+1; m < total; m++ ) {
        parsec_insert_task( "Herk",
            TILE_OF(A, m, k), INPUT,
            TILE_OF(A, m, m), INOUT | AFFINITY);
        for( n = m+1; n < total; n++ ) {
            parsec_insert_task( "Gemm",
                TILE_OF(A, n, k), INPUT,
                TILE_OF(A, m, k), INPUT,
                TILE_OF(A, n, m), INOUT | AFFINITY);
        }
    }
}
for( k = 0; k < total; k++ ) {
    parsec_insert_task( "Potrf", TILE_OF(A, k, k), INOUT | AFFINITY);
    for( m = k+1; m < total; m++ ) {
        parsec_insert_task( "Trsm",
            TILE_OF(A, k, k), INPUT,
            TILE_OF(A, m, k), INOUT | AFFINITY);
    }
    for( m = k+1; m < total; m++ ) {
        parsec_insert_task( "Herk",
            TILE_OF(A, m, k), INPUT,
            TILE_OF(A, m, m), INOUT | AFFINITY);
        for( n = m+1; n < total; n++ ) {
            parsec_insert_task( "Gemm",
                TILE_OF(A, n, k), INPUT,
                TILE_OF(A, m, k), INPUT,
                TILE_OF(A, n, m), INOUT | AFFINITY);
        }
    }
}
for( k = 0; k < total; k++ ) {
    parsec_insert_task( "Potrf", TILE_OF(A, k, k), INOUT | AFFINITY);
    for( m = k+1; m < total; m++ ) {
        parsec_insert_task( "Trsm",
            TILE_OF(A, k, k), INPUT,
            TILE_OF(A, m, k), INOUT | AFFINITY);
    }
    for( m = k+1; m < total; m++ ) {
        parsec_insert_task( "Herk",
            TILE_OF(A, m, k), INPUT,
            TILE_OF(A, m, m), INOUT | AFFINITY);
        for( n = m+1; n < total; n++ ) {
            parsec_insert_task( "Gemm",
                TILE_OF(A, n, k), INPUT,
                TILE_OF(A, m, k), INPUT,
                TILE_OF(A, n, m), INOUT | AFFINITY);
        }
    }
}
Inserting $2 \times \text{HERK} + 1 \times \text{GEMM}$

\[
\begin{align*}
\text{for} (k = 0; k < \text{total}; k++) \{ \\
\quad \text{parsec\_insert\_task} ("\text{Potrf}", \text{TILE\_OF}(A, k, k), \text{INOUT} \mid \text{AFFINITY}); \\
\quad \text{for} (m = k+1; m < \text{total}; m++) \{ \\
\quad\quad \text{parsec\_insert\_task} ("\text{Trsm}", \\
\quad\quad\quad \text{TILE\_OF}(A, k, k), \text{INPUT}, \\
\quad\quad\quad \text{TILE\_OF}(A, m, k), \text{INOUT} \mid \text{AFFINITY}); \\
\quad\quad \} \\
\quad \} \\
\text{for} (m = k+1; m < \text{total}; m++) \{ \\
\quad \text{parsec\_insert\_task} ("\text{Herk}", \\
\quad\quad \text{TILE\_OF}(A, m, k), \text{INPUT}, \\
\quad\quad \text{TILE\_OF}(A, m, m), \text{INOUT} \mid \text{AFFINITY}); \\
\quad \text{for} (n = m+1; n < \text{total}; n++) \{ \\
\quad\quad \text{parsec\_insert\_task} ("\text{Gemm}", \\
\quad\quad\quad \text{TILE\_OF}(A, n, k), \text{INPUT}, \\
\quad\quad\quad \text{TILE\_OF}(A, m, k), \text{INPUT}, \\
\quad\quad\quad \text{TILE\_OF}(A, n, m), \text{INOUT} \mid \text{AFFINITY}); \\
\quad\quad \} \\
\quad \} \\
\}
\end{align*}
\]
for( k = 0; k < total; k++ ) {
    parsec_insert_task( parsec_dtd_handle, parsec_core_potrf, priority,"Potrf",
    sizeof(int), &uplo, VALUE,
    ...
    PASSED_BY_REF, TILE_OF(A, k, k, 0), INOUT | REGION_FULL | AFFINITY, 0 );
    for( m = k+1; m < total; m++ ) {
        parsec_insert_task( parsec_dtd_handle, parsec_core_trsm, priority, "Trsm",
        ...
        PASSED_BY_REF, TILE_OF(A, k, k, 0), INPUT | REGION_FULL,
        PASSED_BY_REF, TILE_OF(A, m, k, 0), INOUT | REGION_FULL | AFFINITY, 0 );
    }
    for( m = k+1; m < total; m++ ) {
        parsec_insert_task( parsec_dtd_handle, parsec_core_herk, priority, "Herk",
        ...
        PASSED_BY_REF, TILE_OF(A, m, k, 0), INPUT | REGION_FULL,
        PASSED_BY_REF, TILE_OF(A, m, m, 0), INOUT | REGION_FULL | AFFINITY, 0 );
        for( n = m+1; n < total; n++ ) {
            parsec_insert_task( parsec_dtd_handle, parsec_core_gemm, priority, "Gemm",
            ...
            PASSED_BY_REF, TILE_OF(A, n, k, 0), INPUT | REGION_FULL,
            PASSED_BY_REF, TILE_OF(A, m, k, 0), INPUT | REGION_FULL,
            PASSED_BY_REF, TILE_OF(A, n, m, 0), INOUT | REGION_FULL | AFFINITY, 0 );
        }
    }
}
Higher is better

QR Factorization on Stampede, 2048 cores, Tile Size = 320, Block Size = 64, 128 nodes

- More complicated communication pattern
- Less opportunity for an explosion in parallelism
- More complex dependencies

Higher is better

PaRSEC-PTG
PaRSEC-DTD
ScaLAPACK
Higher is better

Weak Scaling - DPOTRF and DGEQRF, (20k x 20k)/Node, Tile Size 320, up to 2304 Cores

~75% of Practical Peak
Overall execution time for DTD paradigm (lower bound):

\[ T_{DTD} = \frac{N \times C_T}{P \times n} + N \times C_D + \frac{N \times C_R}{P} \]

- \( T_{DTD} \): Overall time
- \( N \): Total number of tasks
- \( C_T \): Cost/duration of each task
- \( P \): Total number of nodes/process
- \( n \): Total number of cores
- \( C_D \): Cost of discovering
- \( C_R \): Cost of building DAG/relationship
\[ T_{DTD} = \frac{N \times C_T}{P \times n} + \frac{N \times C_D}{P} + \frac{N \times C_R}{P} \]

\( T_{DTD/PTG} \): Overall time

\( N \): Total number of tasks

\( C_T \): Cost/duration of each task

\( P \): Total number of nodes/process

\( n \): Total number of cores

\( C_D \): Cost of discovering a task

\( C_R \): Cost of building DAG/relationship
Higher is better

Cholesky(double) on Haswell, 8 nodes, 20 cores

Gflops

Size(N)/No. of Tasks(millions), Tile size: 320
Cholesky(double) on Haswell, 8 nodes, 20 cores

Higher is better
Overall execution time for DTD paradigm (lower bound):

\[ T_{DTD} = \frac{N \times C_T}{P \times n} + N \times C_D \]

- \( T_{DTD} \): Overall time
- \( N \): Total number of tasks
- \( C_T \): Cost/duration of each task
- \( P \): Total number of nodes/process
- \( n \): Total number of cores
- \( C_D \): Cost of discovering
- \( C_R \): Cost of building DAG/relationship

We are trying to see the upper bound of the best we can do:

\[ T_{DTD} = \frac{N \times C_T}{P \times n} + \frac{N \times C_D}{P} \]
Initial Experiment

- Chain of tasks
- Partitioned per process/rank
- No remote dependency
Proposed Solution (Future Work)

- Distributed Dynamic Task Discovery
- Owner Tracks Future:
  - Each rank will discover all local tasks (task that rank will execute)
  - Each rank will also discover all the remote tasks that uses a data residing in it
  - The owner(temporary) rank will be responsible for tracking the Future of a data
  - The correct ordering will be coordinated by the current owner of each data
Data Placement

Current Scheme

Future Scheme: Owner Tracks Future
Overhead

\[ T_{DTD} = \frac{N \times C_T}{P \times n} + N \times C_D + \frac{N \times C_R}{P} \]

\[ N \times C_{smaller} \]

**\( T_{DTD/PTG} \):** Overall time

- **\( N \):** Total number of tasks
- **\( C_T \):** Cost/duration of each task
- **\( P \):** Total number of nodes/process
- **\( n \):** Total number of cores
- **\( C_D \):** Cost of discovering a task
- **\( C_R \):** Cost of building DAG/relationship
Open Question

• Simple paradigm is not scalable
• What could be an alternate interface that allows *easy expression and is also scalable?