ALTANAL
Abstraction Layer for Task bAsed NumericAl Libraries

Rabab Alomairy

Department of Computer, Electrical and Mathematical Sciences & Engineering
King Abdullah University of Science and Technology
rabab.omairy@kaust.edu.sa

IXPUG Middle East, 2018
Outline

1. **Motivation**
   - Computing Hardware and Software Evolution
   - Task-based Runtime Systems
   - Overview of Task-based ECRC Projects
   - ALTANAL

2. **Literature Review**
   - Overview of Existing Asynchronous Task-based Runtime Systems
   - DARMA

3. **ALTANAL**
   - ALTANAL Layer
   - ALTANAL Interface

4. **Test Cases and Experiments**
   - Dense Cholesky Factorization
   - Tile Low Rank Cholesky Factorization

5. **Conclusion and Future Work**
Table of Contents

1 Motivation
   • Computing Hardware and Software Evolution
   • Task-based Runtime Systems
   • Overview of Task-based ECRC Projects
   • ALTANAL

2 Literature Review
   • Overview of Existing Asynchronous Task-based Runtime Systems
   • DARMA

3 ALTANAL
   • ALTANAL Layer
   • ALTANAL Interface

4 Test Cases and Experiments
   • Dense Cholesky Factorization
   • Tile Low Rank Cholesky Factorization

5 Conclusion and Future Work
Table of Contents

1 Motivation
   • Computing Hardware and Software Evolution
     • Task-based Runtime Systems
     • Overview of Task-based ECRC Projects
     • ALTANAL

2 Literature Review
   • Overview of Existing Asynchronous Task-based Runtime Systems
     • DARMA

3 ALTANAL
   • ALTANAL Layer
   • ALTANAL Interface

4 Test Cases and Experiments
   • Dense Cholesky Factorization
   • Tile Low Rank Cholesky Factorization

5 Conclusion and Future Work
Computing Hardware Evolution

- Frequency barrier
  - Processing units cannot run faster for reasons of energy efficiency
- Concurrency makes up for frequency
  - Several processing units run together such as:
    - Multicore and manycore
    - Vector processing extensions
    - Accelerators
    - Heterogeneous architectures
- Deep memory hierarchy
- More capabilities, more complexity
With this architectural evolution, the burden to design intrinsic algorithmic parallelism rests on software developers.

Parallel programming languages such as MPI+X, which require programmers to:
- expose parallelism from algorithm
- manage computational recourses and communication

Asynchronous task-based runtime systems, which:
- relieve developers from managing low-level resources and let them focus on developing parallel applications
- enhance user productivity
Task-based Runtime Systems

- Task-based runtime systems conceptually similar to out-of-order processor scheduling
- They provide an automatic parallelization by tracking data dependencies and resolving data hazards at runtime
- Task-based runtimes logically operate with a directed acyclic graph (DAG) that:
  - captures data dependencies between application tasks
  - captures tasks read/write data
- Task-based runtimes provide additional information such as task priority, load balancing, and data locality
- Profiling and tracing
- Main challenge of these recent runtime systems is language expressivity
# Table of Contents

1. **Motivation**
   - Computing Hardware and Software Evolution
   - Task-based Runtime Systems
   - Overview of Task-based ECRC Projects
   - ALTANAL

2. **Literature Review**
   - Overview of Existing Asynchronous Task-based Runtime Systems
   - DARMA

3. **ALTANAL**
   - ALTANAL Layer
   - ALTANAL Interface

4. **Test Cases and Experiments**
   - Dense Cholesky Factorization
   - Tile Low Rank Cholesky Factorization

5. **Conclusion and Future Work**
HiCMA: Hierarchical Computations on Manycore Architectures

HiCMA (Hierarchical Computations on Manycore Architectures) is a library designed to address the needs of manycore architectures. It leverages hierarchical computations to improve the efficiency and effectiveness of linear algebra operations. Hierarchical computations are particularly useful in scenarios where data sparsity plays a significant role, such as in large-scale simulations and data analytics. The HiCMA library aims to redesign existing linear algebra libraries to better exploit the data sparsity of the matrices involved. This approach is particularly effective in reducing computational overhead and improving performance in manycore environments.
ExaGeoStat: Exascale GeoStatistics

The Exascale GeoStatistics project (ExaGeoStat) is a parallel high performance unified framework for computational geostatistics on many-core systems. The project aims at optimizing the likelihood function for a given spatial data to provide an efficient way to predict missing observations in the context of climate/weather forecasting applications. This machine learning framework proposes a unified simulation code structure to target various hardware architectures, from commodity x86 to GPU accelerator-based shared and distributed-memory systems. ExaGeoStat enables statisticians to tackle computationally challenging scientific problems at large-scale, while abstracting the hardware complexity, through state-of-the-art high performance linear algebra software libraries.

**ExaGeoStat Dataset Generator**
- Generate 2D spatial locations using uniform distribution.
- Model covariance function:
  \[ \Sigma(h) = \exp(-h^2 / a^2) \]
- Cholesky factorization of the covariance matrix \( \Sigma(h) \) of \( V \times V \).
- Measurement vector generation (Z):
  \[ Z = V \cdot \mathbf{Z} \]
- ExaGeoStat Maximum Likelihood Estimator:
  - Maximum Likelihood Estimator (MLE) learning function:
    \[ \hat{\theta} = \arg\max_{\theta} \log L(\theta) \]
  - Where \( L(\theta) \) is a covariance model with entries
    \[ L_{ij} = \exp(-|x_i - x_j|^2 / a^2) \]

**ExaGeoStat Predictor**
- MLE prediction problem:
  \[ \hat{Z} = X \cdot \hat{\theta} \]
- Wired \( L \in \mathbb{R}^{n \times n} \) and \( X \in \mathbb{R}^{n \times m} \).
- The estimated conditional distribution where \( \hat{Z} \) represents a set of unknown measurements.

**ExaGeoStat Current Research**
- ExaGeoStat Application package
  - For Linear Regression (LR) approximation
  - Out-of-Core support
  - PARSEC runtime system
  - IoV processing

**Performance Results (MLE)**

---

StarPU
MOAO: Multi-Object Adaptive Optics systems

The Multi-Object Adaptive Optics (MOAO) framework provides a comprehensive toolbox for high-performance computational astronomy. In particular, the European Extremely Large Telescope (EELT) is one of today's most challenging projects in ground-based astronomy and will make use of a MOAO instrument based on turbulence tomography. The MOAO framework uses a novel computer-intense pseudoanalytical approach to achieve real-time data processing on massive architectures. The scientific goal of the MOAO simulation package is dimension future EELT instruments and to assess the qualitative performance of tomographic reconstruction of the atmospheric turbulence on real datasets.

OmpSs
KSVD: KAUST Singular Value Decomposition

A QDWH-Based SVD Software Framework on Distributed-Memory Manycore Systems

KSVD

The KAUST SVD (KSVD) is a high-performance software framework for computing a dense SVD on distributed-memory manycore systems. The KSVD solver relies on the polar decomposition using the QR Dynamically-Weighted Halley algorithm (QDWH), introduced by Nakatsuka and Higham [SIAM Journal on Scientific Computing, 2013]. The computational challenge resides in the significant amount of extra floating-point operations required by the QDWH-based SVD algorithm, compared to the traditional one-stage bidiagonal SVD. However, the inherent high level of concurrency associated with Level 3 BLAS compute-bound kernels ultimately compensates the arithmetic complexity overhead and makes KSVD a competitive SVD solver on largescale supercomputers.

The Polar Decomposition

- \( A = U \Sigma V^T \) in \( \mathbb{R}^{m \times n} \), where \( U \) is orthogonal, \( \Sigma \) is a symmetric positive semi-definite matrix.

Application to SVD

- \( A = U \Sigma V^T = U (D_1 V^T) + U (D_2 V^T) \)

Performance Results

- Current Research Directions:
  - Approximate, Task-Based QDWH
  - Dynamic Scheduling
  - Hardware Acceleration
  - Distributed Memory Machines
  - Approximate, Task-Based QDWH
  - QDWH-Based Eigensolvers
  - Integration into PLASMA/MAGMA

Advantages:
- Performs extra loops but nice loops
- Relies on compute intensive kernels
- Exposes high concurrency
- Maps well to GPU architectures
- Minimizes data movement
- Wireline resource synchronizations

KSVD 1.0 Compilable STAGE:
- QDWH-Based Polar Decomposition
- Single Value Decomposition
- Double Precision
- Support to EUPA Symmetric Eigensolvers
- Support to ScalAPACK GESV and MRP Symmetric Eigensolvers
- ScalAPACK Interface / Native Interface
- ScalAPACK-Compliant Error Handling
- ScalAPACK-Compliant Accuracy

Download the software at http://github.com/ksvd/ksvd

Quark
Async-AMG: Asynchronous Algebraic Multigrid

- Asynchronous task-based parallelization of additive algebraic multi-grid for solving $Ax=b$
- It exploits the parallelism between levels of the grid hierarchy in additive AMG

Hybrid MPI+OmpSs (Barcelona Supercomputer Center)
- Cray XC40 supercomputer
Table of Contents

1 Motivation
   - Computing Hardware and Software Evolution
   - Task-based Runtime Systems
   - Overview of Task-based ECRC Projects
   - ALTANAL

2 Literature Review
   - Overview of Existing Asynchronous Task-based Runtime Systems
   - DARMA

3 ALTANAL
   - ALTANAL Layer
   - ALTANAL Interface

4 Test Cases and Experiments
   - Dense Cholesky Factorization
   - Tile Low Rank Cholesky Factorization

5 Conclusion and Future Work
Ringing Bell

API Standardization
ALTANAL: Abstraction Layer for Task Based Numerical Libraries

Although the task-based model is quite active, lack of API standardization makes it difficult for application or library developers to switch runtimes.

A thin layer of abstraction, making the user experience oblivious to the underneath run-time systems.

ALTANAL Goals:
- Providing a set of abstractions to facilitate the expression of tasks that map to a variety of run-time systems such as StarPU, Quark, OmpSs, PaRSEC, OpenMP, and Kokkos.
- Enabling exploration of a variety of underlying runtime system technologies and architectures without changing the application code.
- Enhancing user productivity.
Motivation
- Computing Hardware and Software Evolution
- Task-based Runtime Systems
- Overview of Task-based ECRC Projects
- ALTANAL

Literature Review
- Overview of Existing Asynchronous Task-based Runtime Systems
- DARMA

ALTANAL
- ALTANAL Layer
- ALTANAL Interface

Test Cases and Experiments
- Dense Cholesky Factorization
- Tile Low Rank Cholesky Factorization

Conclusion and Future Work
Table of Contents

1 Motivation
   - Computing Hardware and Software Evolution
   - Task-based Runtime Systems
   - Overview of Task-based ECRC Projects
   - ALTANAL

2 Literature Review
   - Overview of Existing Asynchronous Task-based Runtime Systems
   - DARMA

3 ALTANAL
   - ALTANAL Layer
   - ALTANAL Interface

4 Test Cases and Experiments
   - Dense Cholesky Factorization
   - Tile Low Rank Cholesky Factorization

5 Conclusion and Future Work
Quark: QUEuing And Runtime for Kernels

- ICL, University of Tennessee Knoxville.
- Enable dynamic asynchronous execution of tasks
- Targets multi-core, multi-socket shared memory systems
- It is highly optimized for PLASMA library
- Data locality, priority, and region access.
- Data dependencies: INPUT, INOUT, OUTPUT

<table>
<thead>
<tr>
<th>Algorithm 1: Quark STF Tile Cholesky</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUARK_New(NUM_THREADS);</td>
</tr>
<tr>
<td>QUARK_Sequence_Create (quark);</td>
</tr>
<tr>
<td>for k = 0; k &lt; nt; k++ do</td>
</tr>
<tr>
<td>QUARK_Insert_Task(quark, POTRF, flg, ...);</td>
</tr>
<tr>
<td>for m = k + 1; m &lt; nt; m++ do</td>
</tr>
<tr>
<td>QUARK_Insert_Task(quark, TRSM, flg, ...);</td>
</tr>
<tr>
<td>for n = k + 1; n &lt; nt; n++ do</td>
</tr>
<tr>
<td>QUARK_Insert_Task(quark, SYRK, flg, ...);</td>
</tr>
<tr>
<td>for m = n + 1; m &lt; nt; m++ do</td>
</tr>
<tr>
<td>QUARK_Insert_Task(quark, GEMM, flg, ...);</td>
</tr>
<tr>
<td>QUARK_BARRIER( quark );</td>
</tr>
<tr>
<td>QUARK_Sequence_Wait (quark, seq);</td>
</tr>
<tr>
<td>QUAR_Delete(quark);</td>
</tr>
</tbody>
</table>
QUARK_Insert_Task(quark, POTRF, flags, sizeof(int), &uplo, VALUE, sizeof(int), &n, VALUE, sizeof(double)*nb*nb, &A, INOUT, sizeof(int), &lda, VALUE, sizeof(int), &iinfo, VALUE, 0);

void POTRF (Quark* quark) {
    quark_unpack_args_5(quark, &uplo,&n, &A, &lda, &iinfo );
    potrf(uplo, n, A, lda, iinfo);
}
INRIA Bordeaux Sud-Ouest
Enable dynamic task-based implementation.
Multicore, Manycore shared/distributed memory, and heterogeneous architecture
StarPU provides task scheduling and memory management mechanisms
Based on three principle:
- Registering its buffers, to get one handle per buffer
- Defining codelets to CPU and GPU implementations
- Applying codelets on some handles
Data dependencies: STARPU_R, STARPU_W, STARPU_RW

Algorithm 2: StarPU STF Tile Cholesky

```
starpu_init(NULL);
starpu_data_handle_t handle;
starpu_matrix_data_register(&handle, ..)
for k = 0; k < nt; k++ do
    starpu_insert_task(&POTRF, ..);
    for m = k + 1; m < nt; m++ do
        starpu_insert_task(&TRSM, ..);
    for n = k + 1; n < nt; n++ do
        starpu_insert_task(&SYRK, ..);
    for m = n + 1; m < nt; m++ do
        starpu_insert_task(&GEMM, ..);
starpu_task_wait_for_all();
starpu_data_unregister (handle);
starpu_shutdown();
```
starpu_insert_task(&dpotrf_starpu,
    STARPU_VALUE, &uplo, sizeof(int),
    STARPU_VALUE, &n, sizeof(int),
    STARPU_RW, &Ahandle,
    STARPU_VALUE, &lda, sizeof(int),
    STARPU_VALUE, &iinfo, sizeof(int),
    ,0);

void POTRF(void *descr[], void *cl_args) {
    A = STARPU_MATRIX_GET_PTR(descr[0]);
    starpu_codelet_unpack_args(cl_args,
        &uplo,&n, &lda, &iinfo );
    potrf(uplo, n, A, lda, iinfo);
}

struct starpu_codelet
dpotrf_starpu = {
    .type = STARPU_SEQ,
    .cpu_funcs = {POTRF},
    .nbuffers = 1,
    .modes ={ STARPU_RW} } ;
PaRSEC: Parallel Runtime Scheduling and Execution Controller

- ICL, University of Tennessee
- Dynamic Task Discovery and Parametrized Task Graph
- Multicore, Manycore
- Shared/distributed memory, and heterogeneous architecture
- Data dependencies: INPUT, OUTPUT, INOUT
- PTG has symbolic DAG and no need to build and store it in memory.
- DTD inserts tasks sequentially, and the DAG is built dynamically during run-time and stores it in memory.

**Algorithm 3: PaRSEC STF Tile Cholesky**

```c
parsec=setup_parsec(argc, argv, iparam);
dtd_tp = parsec_dtd_taskpool_new ();
parsec_dtd_data_collection_init(&A);
parsec_enqueue( parsec, dtd_tp );
parsec_context_start(parsec);

for k = 0; k < nt; k++ do
    parsec_dtd_taskpool_insert_task(&POTRF, ...);
    for m = k + 1; m < nt; m++ do
        parsec_dtd_taskpool_insert_task(&TRSM, ...);
    for n = k + 1; n < nt; n++ do
        parsec_dtd_taskpool_insert_task(&SYRK, ...);
        for m = n + 1; m < nt; m++ do
            parsec_dtd_taskpool_insert_task(&GEMM, ...);
    parsec_dtd_taskpool_wait( parsec, dtd_tp );
    parsec_context_wait( parsec );
    parsec_taskpool_free( dtd_tp );
```

Rabab Alomairy (KAUST)
parse_dtd_taskpool_insert_task(dtd_task, POTRF, priority, "potrf"
    sizeof(int),    &uplo,    VALUE,
    sizeof(int),    &n,        VALUE,
    sizeof(double)*nb*nb, &A,        INOUT,
    sizeof(int),    &lda,     VALUE,
    sizeof(int),    &iinfo,    VALUE,
    ,PARSEC_DTD_ARG_END);

void POTRF(parsec_execution_stream_t *es, parsec_task_t *this_task){
    parsec_dtd_unpack_args(this_task, &uplo,&n, &A, &lda, &iinfo);
    potrf(uplo, n, A, lda, iinfo);
}
OpenMP: Open Multi-Processing

- OpenMP Architecture Review Board (or OpenMP ARB)
- OpenMP 1.x (1997-98), OpenMP 2.x (2000-02)
  - Thread-based fork-join programming model
- OpenMP 3.x (2008-11)
  - Independent tasks
- OpenMP 4.x (2013-15)
  - Task with dependencies
- Multicore, Manycore shared memory systems
- Data dependencies: in, out, inout

Algorithm 4: OpenMP STF Tile Cholesky

```c
#pragma omp parallel
#pragma omp master
{
  for (k = 0; k < nt; k++)
    #pragma omp task depend(inout:A[0:nb])
    POTRF(A[k][k]);
  for (m = k + 1; m < nt; m++)
    #pragma omp task depend(in:A[0:nb]) depend(inout:A[0:nb])
    TRSM( A[k][k], A[m][k]);
  for (n = k + 1; n < nt; n++)
    #pragma omp task depend(in:A[0:nb]) depend(inout:A[0:nb])
    SYRK(A[n][k], A[n][n]);
  for (m = n + 1; m < nt; m++)
    #pragma omp task depend(in:A[0:nb], A[0:nb]) depend(inout:A[0:nb])
    GEMM(A[n][k], A[m][k], A[n][m]);
}
```
OmpSs

- Bercelona Supercomputing Center (BSC)
- Name originally comes from: OpenMP and StarSs
- OmpSs-2 is second generation of the OmpSs
- Multicore, Manycore shared/distributed memory, and heterogeneous architecture
- It allows nesting of tasks
- Data dependencies: in, out, inout

Algorithm 5: OmpSs STF Tile Cholesky

```plaintext
for k = 0; k < nt; k++ do
    #pragma omp inout(A[0:nb])
    POTRF(A[k][k]);
    for m = k + 1; m < nt; m++ do
        #pragma omp in(A[0:nb]) inout(A[0:nb])
        TRSM( A[k][k], A[m][k]);
    for n = k + 1; n < nt; n++ do
        #pragma omp task in(A[0:nb]) inout(A[0:nb])
        SYRK(A[n][k], A[n][n]);
    for m = n + 1; m < nt; m++ do
        #pragma omp task in(A[0:nb], A[0:nb]) inout(A[0:nb])
        GEMM(A[n][k], A[m][k], A[n][m]);
```
## Summary Table of Existing Task-based Runtime Systems

<table>
<thead>
<tr>
<th>Task-based Runtime</th>
<th>Developer group</th>
<th>Distributed Memory</th>
<th>GPU</th>
<th>Scheduler Features</th>
<th>Task Dependency</th>
<th>Programming Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMP</td>
<td>OpenMP ARB</td>
<td></td>
<td></td>
<td>standard in GNU, pragma directive</td>
<td>STF, implicit DAG, fork-join model</td>
<td>C, C++, Fortran</td>
</tr>
<tr>
<td>OmpSs/OmpSs-2</td>
<td>Barcelona Supercomputing Center</td>
<td>✓</td>
<td>✓</td>
<td>Breadth First, Work First, Socket-aware scheduler, Bottom level-aware scheduler</td>
<td>STF, implicit DAG</td>
<td>C, C++, Fortran</td>
</tr>
<tr>
<td>StarPU</td>
<td>INRIA Bordeaux Sud-Ouest</td>
<td>✓</td>
<td>✓</td>
<td>prio, dm, dmda, eager scheduler, work stealing, priority</td>
<td>STF, implicit DAG</td>
<td>C</td>
</tr>
<tr>
<td>PaRSEC</td>
<td>UTK</td>
<td>✓</td>
<td>✓</td>
<td>PBQ, ePBQ schedulers, work stealing, priority</td>
<td>STF, implicit DAG, PTG, explicit DAG</td>
<td>C, Fortran JDF compiler</td>
</tr>
<tr>
<td>Quark</td>
<td>UTK</td>
<td></td>
<td></td>
<td>priority and locality hinting, accumulator and gatherv tasks</td>
<td>STF, implicit DAG</td>
<td>C</td>
</tr>
<tr>
<td>Kokkos</td>
<td>Sandia National Laboratories</td>
<td>✓</td>
<td></td>
<td>execution policy and pattern, memory space and layout</td>
<td>STF, implicit DAG</td>
<td>C++</td>
</tr>
<tr>
<td>Task-based Runtime</td>
<td>Developer group</td>
<td>Distributed Memory</td>
<td>GPU</td>
<td>Scheduler Features</td>
<td>Task Dependency</td>
<td>Programming Interface</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------</td>
<td>--------------------</td>
<td>-----</td>
<td>--------------------</td>
<td>----------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>Legion</td>
<td>Stanford University</td>
<td>✓</td>
<td>✓</td>
<td>logical regions, spawning child task, mapping interface</td>
<td>STF, implicit DAG</td>
<td>C++, Regent compiler</td>
</tr>
<tr>
<td>HPX</td>
<td>STEllAR Group</td>
<td>✓</td>
<td>✓</td>
<td>work-queuing model, message driven computation using tasked based</td>
<td>explicit parallelism, fork-join model</td>
<td>C, C++</td>
</tr>
<tr>
<td>Charm++</td>
<td>University of Illinois</td>
<td>✓</td>
<td>✓</td>
<td>shares, entry methods, migratability, asynchrony, structure dagger</td>
<td>explicit DAG</td>
<td>C++, custom</td>
</tr>
<tr>
<td>OCR</td>
<td>Universities Laboratories</td>
<td></td>
<td></td>
<td>asynchronous event-driven, work-stealing, priority, work-sharing</td>
<td>explicit, DAG</td>
<td>C</td>
</tr>
<tr>
<td>Cilk</td>
<td>Intel</td>
<td></td>
<td></td>
<td>spawn and sync, reducers and hyperobjects, and work-stealing</td>
<td>Divide and conquer (recursive tasks), fork-join model</td>
<td>C, C++</td>
</tr>
<tr>
<td>TBB</td>
<td>Intel</td>
<td></td>
<td></td>
<td>nested parallelism, work-stealing, ranges and partitioners, memory allocators</td>
<td>parallel algorithms, explicit flow graphs</td>
<td>C++</td>
</tr>
</tbody>
</table>
Table of Contents

1 Motivation
   • Computing Hardware and Software Evolution
   • Task-based Runtime Systems
   • Overview of Task-based ECRC Projects
   • ALTANAL

2 Literature Review
   • Overview of Existing Asynchronous Task-based Runtime Systems
   • DARMA

3 ALTANAL
   • ALTANAL Layer
   • ALTANAL Interface

4 Test Cases and Experiments
   • Dense Cholesky Factorization
   • Tile Low Rank Cholesky Factorization

5 Conclusion and Future Work
DARMA is a C++ abstraction layer for asynchronous many-task (AMT) runtimes.

Sandia National Laboratories.

Executing applications with multiple different run-time systems to take advantage of the strengths and weaknesses of each without modifying user code.

DARMA software provides two levels: front-end API, and back-end-API.

DARMA currently supports OpenMP and Kokkos runtime systems in the backend.
DARMA: Distributed Asynchronous Resilient Models and Applications

- DARMA follows the Sequential Task Flow (STF) model to express the main building blocks.
- Data interactions occur using special handles called AccessHandle.
- Asynchronous tasks can be defined using `create_work()` function.
  - `create_work()` can be called with either a C++11 lambda or a functor.
- AccessHandle has well-defined deterministic permissions on the underlying data: Modify, Read, None.
Table of Contents

1 Motivation
   - Computing Hardware and Software Evolution
   - Task-based Runtime Systems
   - Overview of Task-based ECRC Projects
   - ALTANAL

2 Literature Review
   - Overview of Existing Asynchronous Task-based Runtime Systems
   - DARMA

3 ALTANAL
   - ALTANAL Layer
   - ALTANAL Interface

4 Test Cases and Experiments
   - Dense Cholesky Factorization
   - Tile Low Rank Cholesky Factorization

5 Conclusion and Future Work
Table of Contents

1 Motivation
   • Computing Hardware and Software Evolution
   • Task-based Runtime Systems
   • Overview of Task-based ECRC Projects
   • ALTANAL

2 Literature Review
   • Overview of Existing Asynchronous Task-based Runtime Systems
   • DARMA

3 ALTANAL
   • ALTANAL Layer
   • ALTANAL Interface

4 Test Cases and Experiments
   • Dense Cholesky Factorization
   • Tile Low Rank Cholesky Factorization

5 Conclusion and Future Work
ALTANAL Layer

- Application
  - Unified API
  - Abstraction Layer
- Runtime
  - OpenMP, OmpSs, ParSEC, Kokkos, Quark, StarPU
- Hardware
  - Multicore, Manycore
  - Shared/distributed memory system
  - CPU/GPU
ALTANAL Layer

High-Level API

- control
- context
- task
- async
- options

Low-Level API

- runtime_control
- runtime_context
- runtime_task
- runtime_async
- runtime_options
Table of Contents

1 Motivation
   • Computing Hardware and Software Evolution
   • Task-based Runtime Systems
   • Overview of Task-based ECRC Projects
   • ALTANAL

2 Literature Review
   • Overview of Existing Asynchronous Task-based Runtime Systems
   • DARMA

3 ALTANAL
   • ALTANAL Layer
     • ALTANAL Interface

4 Test Cases and Experiments
   • Dense Cholesky Factorization
   • Tile Low Rank Cholesky Factorization

5 Conclusion and Future Work
ALTANAL Interface

High-Level API

- ALTANAL_Init
- ALTANAL_Finalize
- ALTANAL_Enable
- ALTANAL_Disable
- ALTANAL_Sequence_Create
- ALTANAL_Sequence_Destroy
- ALTANAL_Insert_Task
- ALTANAL_Unpack_Arg
- ALTANAL_Options_Init

Low-Level API

- ALTANAL_Runtime_init
- ALTANAL_Runtime_finalize
- ALTANAL_Runtime_enable
- ALTANAL_Runtime_disable
- ALTANAL_Runtime_sequence_create
- ALTANAL_Runtime_sequence_destroy
- ALTANAL_Runtime_insert_task
- ALTANAL_Runtime_unpack_arg
- ALTANAL_Runtime_options_init
ALTANAL_Init

ALTANAL

control.c

ALTANAL_Init( ncpus, ngpus);

context.c

allocate altanal context struct
ALTANAL_Runtime_context_create(altanal);

Runtimes

Runtime_context.c

void ALTANAL_Runtime_context_create
(ALTANAL_context_t*)
RUNTIME_SCHED_OMPSS
RUNTIME_SCHED_OPENMP
RUNTIME_SCHED_QUARK
RUNTIME_SCHED_STARPU

Runtime_control.c

int ALTANAL_Runtime_init_scheduler()
QUARK_New
starpu_init
starpu_mpi_init
starpu_cublas_init
parsec_init

1- altanal_context_create()

2-ALTANAL_Runtime_init_scheduler(
altanal, ncpus, ncudas,
nthreads_per_worker );
ALTANAL_Finalize

```
control.c
  ALTANAL_Finalize()
  ALTANAL_Runtime_barrier(altanal);
  ALTANAL_Runtime_finalize(altanal);
  altanal_context_destroy();

context.c
  ALTANAL_Runtime_context_destroy(altanal)

Runtimes

Runtime_control.c
  void ALTANAL_Runtime_barrier(altanal)
  QUARK_BARRIER()
  starpu_task_wait_for_all();
  parsec_context_wait(parsec);

Runtime_control.c
  void ALTANAL_runtime_finalize(altanal)
  QUARK_Delete()
  starpu_mpi_shutdown()
  starpu_cublas_shutdown()
  starpu_task_wait_for_all()
  parsec_fini(&parsec);

Runtime_context.c
  void ALTANAL_Runtime_context_destroy_destroy(altanal)
  free(altanal->schedopt);
```

Rabab Alomairy  (KAUST)
**Algorithm 6:** ALTANAL STF Tile Cholesky

```c
ALTANAL_Init(ncpus, ngpus);
ALTANAL_context_t *altanal;
ALTANAL_sequence_t *sequence;
ALTANAL_request_t *request;
ALTANAL_Sequence_Create(altanal, &sequence);

for k = 0; k < nt; k++ do
    ALTANAL_Insert_Task(&ALTANAL_CODELETS_NAME(POTRF), options, ...);
    for m = k + 1; m < nt; m++ do
        ALTANAL_Insert_Task(&ALTANAL_CODELETS_NAME(TRSM), options, ...);
    for n = k + 1; n < nt; n++ do
        ALTANAL_Insert_Task(&ALTANAL_CODELETS_NAME(SYRK), options, ...);
            for m = n + 1; m < nt; m++ do
                ALTANAL_Insert_Task(&ALTANAL_CODELETS_NAME(GEMM), options, ...);

ALTANAL_Sequence_Wait(altanal, sequence);
ALTANAL_Sequence_Destroy(altanal, sequence);
ALTANAL_Finalize();
```
ALTANAL Main API

ALTANAL_CODELETS(POTRF, POTRF_CPU)

ALTANAL_Insert_Task(ALTANAL_CODELETS_NAME(POTRF), options,
    ALTANAL_VALUE, &uplo, sizeof(int),
    ALTANAL_VALUE, &n, sizeof(int),
    ALTANAL_INOUT, &A, sizeof(double)*nb*nb
    ALTANAL_VALUE, &lda, sizeof(int),
    ALTANAL_VALUE, &iinfo, sizeof(int),
    ,ALTANAL_PARAM_END);

void POTRF_CPU(ALTANAL altanal) {
    ALTANAL_Unpack_Args(altanal, &uplo,&n, &lda, &iinfo );
    potrf(uplo, n, A, lda, iinfo);
}
Table of Contents

1 Motivation
   - Computing Hardware and Software Evolution
   - Task-based Runtime Systems
   - Overview of Task-based ECRC Projects
   - ALTANAL

2 Literature Review
   - Overview of Existing Asynchronous Task-based Runtime Systems
   - DARMA

3 ALTANAL
   - ALTANAL Layer
   - ALTANAL Interface

4 Test Cases and Experiments
   - Dense Cholesky Factorization
   - Tile Low Rank Cholesky Factorization

5 Conclusion and Future Work
Table of Contents

1 Motivation
   ● Computing Hardware and Software Evolution
   ● Task-based Runtime Systems
   ● Overview of Task-based ECRC Projects
   ● ALTANAL

2 Literature Review
   ● Overview of Existing Asynchronous Task-based Runtime Systems
   ● DARMA

3 ALTANAL
   ● ALTANAL Layer
   ● ALTANAL Interface

4 Test Cases and Experiments
   ● Dense Cholesky Factorization
   ● Tile Low Rank Cholesky Factorization

5 Conclusion and Future Work
Dense Cholesky Factorization

Figure: Dual-socket 18-core Intel(R) Xeon(R) Haswell CPU E5-2699 v3 @ 2.3 GHz with 256GB of main memory.
Dense Cholesky Factorization

**Figure:** Dual-socket 28-core Intel(R) Xeon(R) Platinum 8176 CPU @ 2.10GHz 38.5MB with 256GB of main memory.
1 Motivation
- Computing Hardware and Software Evolution
- Task-based Runtime Systems
- Overview of Task-based ECRC Projects
- ALTANAL

2 Literature Review
- Overview of Existing Asynchronous Task-based Runtime Systems
- DARMA

3 ALTANAL
- ALTANAL Layer
- ALTANAL Interface

4 Test Cases and Experiments
- Dense Cholesky Factorization
- Tile Low Rank Cholesky Factorization

5 Conclusion and Future Work
Tile Low Rank Cholesky Factorization

Figure: Dual-socket 18-core Intel(R) Xeon(R) Haswell CPU E5-2699 v3 @ 2.3 GHz with 256GB of main memory, shared memory system
Tile Low Rank Cholesky Factorization

**Figure:** Dual-socket 28-core Intel(R) Xeon(R) Platinum 8176 CPU @ 2.10GHz 38.5MB with 256GB of main memory, shared memory system
Table of Contents

1 Motivation
   - Computing Hardware and Software Evolution
   - Task-based Runtime Systems
   - Overview of Task-based ECRC Projects
   - ALTANAL

2 Literature Review
   - Overview of Existing Asynchronous Task-based Runtime Systems
   - DARMA

3 ALTANAL
   - ALTANAL Layer
   - ALTANAL Interface

4 Test Cases and Experiments
   - Dense Cholesky Factorization
   - Tile Low Rank Cholesky Factorization

5 Conclusion and Future Work
Conclusion and Future Work

- ALTANAL is designed to provide run-time oblivious interface that can be mapped to many backend run-times (OpenMP, StarPU, Quark, OmpSs, Kokkos, PaRSEC)
- It tackles many challenges such as the ability of studying different runtimes for best standards and practices
- ALTANAL API is used in the interface of compute-bound and memory-bound workloads and enable them to switch between Quark, StarPU, and PaRSEC
- ALTANAL currently abstracts Quark, StarPU, and PaRSEC
- This research will support many of today's scientific applications based on leading asynchronous dynamic runtime systems
- We are targeting other run-time systems such as OpenMP, OmpSs, Kokkos, and TBB
- ALTANAL will be extended to benefit from C++ abstraction mechanism
Thanks