

# 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.omaairy@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

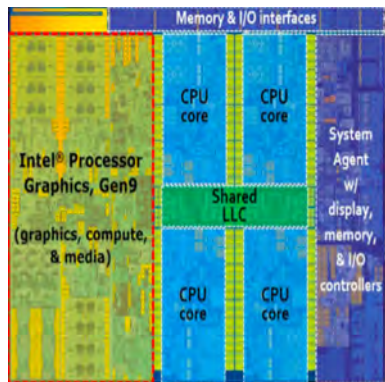
## 4 Test Cases and Experiments

- 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

# 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

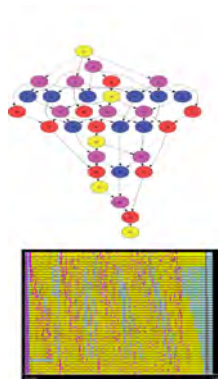
## 4 Test Cases and Experiments

- Dense Cholesky Factorization
- Tile Low Rank Cholesky Factorization

## 5 Conclusion and Future Work

# Task-based Runtime Systems

- Task-based runtime systems conceptually similar to out-of-order processor scheduling
- They provides 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

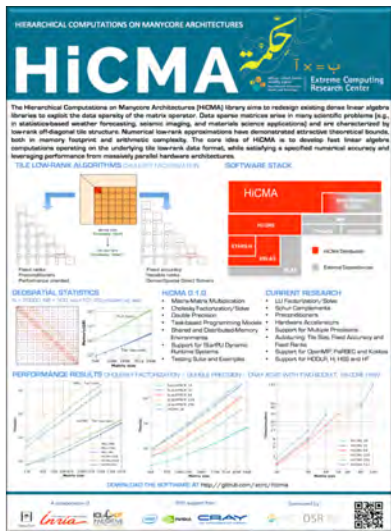
- ALTANAL Layer
- ALTANAL Interface

## 4 Test Cases and Experiments

- Dense Cholesky Factorization
- Tile Low Rank Cholesky Factorization

## 5 Conclusion and Future Work

# HiCMA: Hierarchical Computations on Manycore Architectures



**HIERARCHICAL COMPUTATIONS ON MANYCORE ARCHITECTURES**

# HiCMA

Extreme Computing Research Center

The Hierarchical Computations on Manycore Architectures (HiCMA) library aims to redesign existing dense linear algebra libraries to exploit the data sparsity of the matrix operator. Data sparse matrices arise in many scientific problems (e.g., in statistics-based weather forecasting, seismic imaging, and materials science applications) and are characterized by low-rank off-diagonal life structure. Numerical low-rank approximations have demonstrated attractive theoretical bounds, both in memory footprint and arithmetic complexity. The core idea of HiCMA is to develop fast linear algebra computations operating on the underlying the low-rank data format, while satisfying a specified numerical accuracy and leveraging performance from massively parallel hardware architectures.

**TILE LOW-RANK ALGORITHMS ON SPARSE MATRICES**

**SOFTWARE STACK**

**DISCRETE STATISTICS**

- Memory-Matrix Multiplication
- Cholesky Factorization/Update
- Double Precision
- Task-based Programming Model
- Shared and Distributed Memory Environments
- Support for GPU/Dynamic Runtime Systems
- Testing Suite and Examples

**CURRENT RESEARCH**

- LU Factorization/Update
- Sparse Conjugate Gradient
- Preconditioners
- Hardware Accelerators
- Support for Multiple Precision
- Addressing Tile Size, Fixed Accuracy and Fixed Rank
- Support for OpenMP/PARSEC and Kokkos
- Support for HEDRA, H-HB and H-F

**PERFORMANCE RESULTS (SPEEDUP FACTORS) - DOUBLE PRECISION - DATA ADJ. WITH ENHANCED BLOCKS (HW)**

Download the software at <http://github.com/wy/hicma>

Advertisement by: **Crucial** | Sponsors: **Intel**, **ARM**, **CRAY**, **DSR**



# ExaGeoStat: Exascale GeoStatistics

FORAL: HIGH PERFORMANCE UPGRADE FRAMEWORK FOR GEOSTATISTICS ON EMV CORE SYSTEMS

## ExaGeoStat

Extreme Computing Research Center

The Exascale GeoStatistics project (ExaGeoStat) is a parallel high performance unified framework for computational geostatistics on manycore 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
- Matrix covariance function:
 
$$C(r, \theta) = \frac{\sigma^2}{2\pi} \exp\left(-\frac{\sqrt{r^2 + \theta^2}}{\lambda}\right) K_0\left(\frac{r}{\lambda}\right)$$
- Cholesky factorization of the covariance matrix:
 
$$\Sigma(r) = P^T L^2$$
- Measurement vector generation [2]:
 
$$Z = P \cdot L \cdot \epsilon + \theta \cdot \mathbf{1}$$

**ExaGeoStat Maximum Likelihood Estimator**

- Maximum Likelihood Estimator (MLE) (learning function):
 
$$\hat{\theta} = -\left[\text{diag}(\Sigma^{-1}) - \frac{1}{\sigma^2} \Sigma^{-1} \mathbf{1} \mathbf{1}^T \Sigma^{-1}\right]^{-1} \Sigma^{-1} \mathbf{1}^T Z$$
- Where  $\Sigma^{-1}(R)$  is a covariance matrix with entries:
 
$$\Sigma_{ij}^{-1} = C(\|x_i - x_j\|) \delta_{ij} - C(\|x_i\|) \delta_{ij} - C(\|x_j\|) \delta_{ij} + C(0)$$

**ExaGeoStat Predictor**

- MLI prediction profiles:
 
$$\hat{Z}_i = \hat{\theta} + \frac{\sigma^2}{\sigma^2 + \Sigma_{ii}^{-1}} \left( \frac{\Sigma_{ii}^{-1}}{\sigma^2 + \Sigma_{ii}^{-1}} Z_i + \frac{\sigma^2}{\sigma^2 + \Sigma_{ii}^{-1}} \hat{\theta} \right)$$
- The associated conditional distribution where  $Z_i$  represents a set of unknown measurement vectors:
 
$$Z_i | Z_{-i} = \hat{\theta} + \frac{\sigma^2}{\sigma^2 + \Sigma_{ii}^{-1}} \left( \frac{\Sigma_{ii}^{-1}}{\sigma^2 + \Sigma_{ii}^{-1}} Z_i + \frac{\sigma^2}{\sigma^2 + \Sigma_{ii}^{-1}} \hat{\theta} \right)$$

**ExaGeoStat 0.1.0**

- Large-scale synthetic geo-spatial datasets generation
- Maximum Likelihood Estimation (MLE) Synthetic and real datasets
- A large-scale production tool for unknown measurements on known locations

**Current Research**

- ExaGeoStat manager package
- File Line Rank (FLR) approximation
- NetCDF format support
- ParMETIS runtime system
- Event processing

**Performance Results (MLE)**


Performance speed results for 4000000 locations

Performance speed of MLE (MLE)

Time (M) and Accuracy (Theoretical)

StarPU

© 2018-2020 ExaGeoStat. All rights reserved. For more information, please visit <http://github.com/exageostat>





# KSVD: KAUST Singular Value Decomposition

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

# KSVD

Extreme Computing Research Center

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 (GDWH), introduced by Nakatawase and Higham (SIAM Journal on Scientific Computing, 2013). The computational challenge resides in the significant amounts of extra floating-point operations required by the GDWH-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 large-scale supercomputers.

**The Polar Decomposition:**  $A = U_1 H V^T$ , where  $U_1$  is orthogonal Matrix, and  $H$  is a symmetric positive semidefinite matrix.

**GDWH Algorithm:**

- Sechwarz-like algorithm for computing the GDWH-based SVD
- Based on conventional computational kernels, i.e. Cholesky/QR factorizations (1/2 iterations for double precision) and GEMM
- The total flop count for GDWH depends on the condition number of the matrix

**Advantages:**

- Performs extra flops but less flops
- Relies on compute intensive kernels
- Enables high concurrency
- Maps well to GPU architectures
- Minimizes data movement
- Utilizes resource approximations

**Application to SVD**

- $A = U_1 H V^T$
- $A = U_1 (U_2 V^T) = U_1 U_2 V^T = U V^T$

**Performance Results:**

**KSVD 1.0 Core Tables (K11.0)**

- GDWH-based Polar Decomposition
- Singular Value Decomposition
- Double Precision
- Support to CUDA Symmetric Eigenvalue
- Support to SciLAPACK DSC and MPI Symmetric Eigensolvers
- SciLAPACK interface / Native Interface
- SciLAPACK Compiler Error Handling
- SciLAPACK Compiler Testing Suite
- SciLAPACK-Compliant Accuracy

**State-of-the-Art:**

**Current Research Challenges (K14.0)**

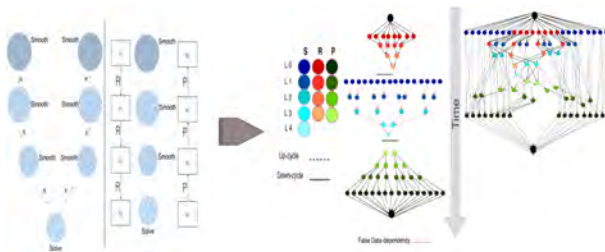
- Asynchronous, Task-Based GEMM
- Dynamic Scheduling
- Hardware Accelerators
- Checkpoint Memory Mappings
- Asynchronous, Task-Based GDWH-based SVD
- GDWH-based Eigensolver (GDWH-ES)
- Integration into PLASMA/MAEMA

Download the software in <http://github.com/ksvd/ksvd>



# 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

## API Standardization





- ALTANAL: Abstraction Layer for Task based NumericAI 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 facilitates the expression of tasking 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

# 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

# Quark: **Q**Ueuing **A**nd **R**untime for **K**ernels

- 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

---

**Algorithm 1:** Quark STF Tile Cholesky

---

```
QUARK_New(NUM_THREADS);
QUARK_Sequence_Create (quark);

for  $k = 0$ ;  $k < nt$ ;  $k++$  do
    QUARK_Insert_Task(quark, POTRF, flg, ...);
    for  $m = k + 1$ ;  $m < nt$ ;  $m++$  do
        QUARK_Insert_Task(quark, TRSM, flg, ...);
        for  $n = k + 1$ ;  $n < nt$ ;  $n++$  do
            QUARK_Insert_Task(quark, SYRK, flg, ...);
            for  $m = n + 1$ ;  $m < nt$ ;  $m++$  do
                QUARK_Insert_Task(quark, GEMM, flg, ...);

QUARK_Barrier( quark );
QUARK_Sequence_Wait (quark, seq);
QUAR_Delete(quark);
```

---

## Quark: **Q**ueuing **A**nd **R**untime for **K**ernels

```
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

---

```
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 );
```

---



# PaRSEC: Parallel Runtime Scheduling and Execution Controller

```
parsec_dtd_taskpool_insert_task(dtd_tp, 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

---

```
#pragma omp parallel
#pragma omp master
{
  for k = 0; k < nt; k++ do
    #pragma omp task depend(inout:A[0:nb])
    POTRF(A[k][k]);
  for m = k + 1; m < nt; m++ do
    #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++ do
    #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++ do
    #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]);
}
```

---

- Barcelona 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

---

```
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

Task-based Runtime	Developer group	Distributed Memory	GPU	Scheduler Features	Task Dependency	Programming Interface
OpenMP	OpenMP ARB			standard in GNU, pragma directive	STF, implicit DAG, fork-join model	C, C++, Fortran
OmpSs /OmpSs-2	Bercelona Supercomputing Center	✓	✓	Breadth First, Work First, Socket-aware scheduler, Bottom level-aware scheduler	STF, implicit DAG	C, C++, Fortran
StarPU	INRIA Bordeaux Sud-Ouest	✓	✓	prio, dm , dmda, eager scheduler, work stealing, priority	STF, implicit DAG	C
PaRSEC	UTK	✓	✓	PBQ , ePBQ schedulers, work stealing, priority	STF, implicit DAG, PTG, explicit DAG	C, Fortran JDF compiler
Quark	UTK			priority and locality hinting, accumulator and gatherv tasks	STF, implicit DAG	C
Kokkos	Sandia National Laboratories		✓	execution policy and pattern, memory space and layout	STF, implicit DAG	C++

# Summary Table of Existing Task-based Runtime Systems

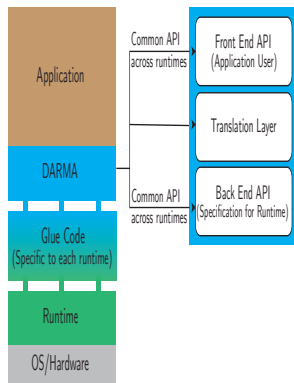
Task-based Runtime	Developer group	Distributed Memory	GPU	Scheduler Features	Task Dependency	Programming Interface
Legion	Stanford University	✓	✓	logical regions, spawning child task, mapping interface	STF, implicit DAG	C++, Regent compiler
HPX	STELLAR Group	✓	✓	work-queuing model, message driven computation using tasked based	explicit parallelism, fork-join model	C, C++
Charm++	University of Illinois	✓	✓	chares, entry methods, migratability, asynchrony , structure dagger	explicit DAG	C++, custom
OCR	Universities \ Laboratories			asynchronous event-driven, work-stealing, priority, work-sharing	explicit, DAG	C
Cilk	Intel			spawn and sync, reducers and hyperobjects, and work-stealing	Divide and conquer (recursive tasks), fork-join model	C, C++
TBB	Intel			nested parallelism, work-stealing, ranges and partitioners, memory allocators	parallel algorithms, explicit flow graphs	C++

# 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

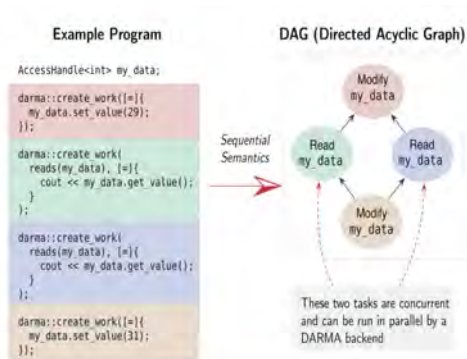
# DARMA: Distributed Asynchronous Resilient Models and Applications

- **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 occurs using special handle called an `AccessHandle`
- Asynchronous task 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`.





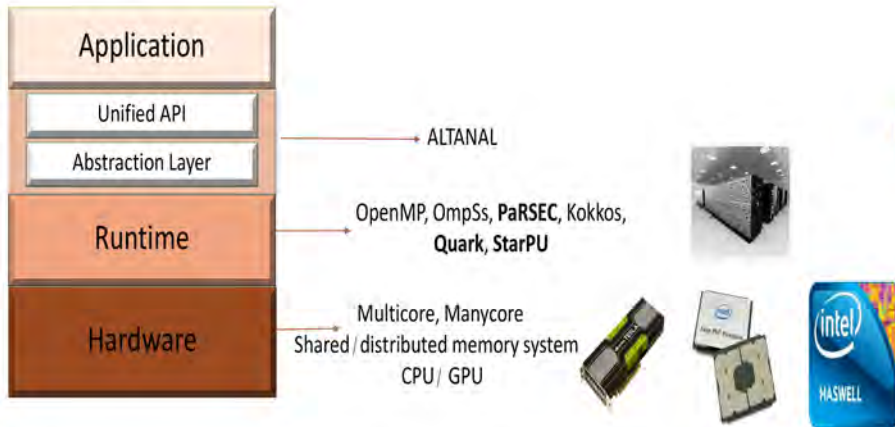
# Table of Contents

- 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

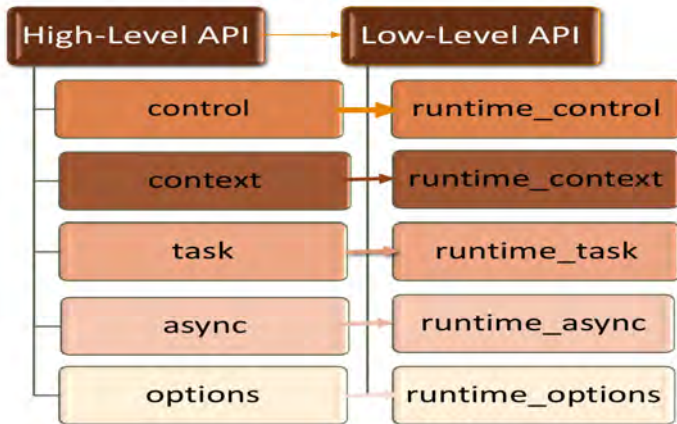
# Table of Contents

- 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



# ALTANAL Layer



# 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

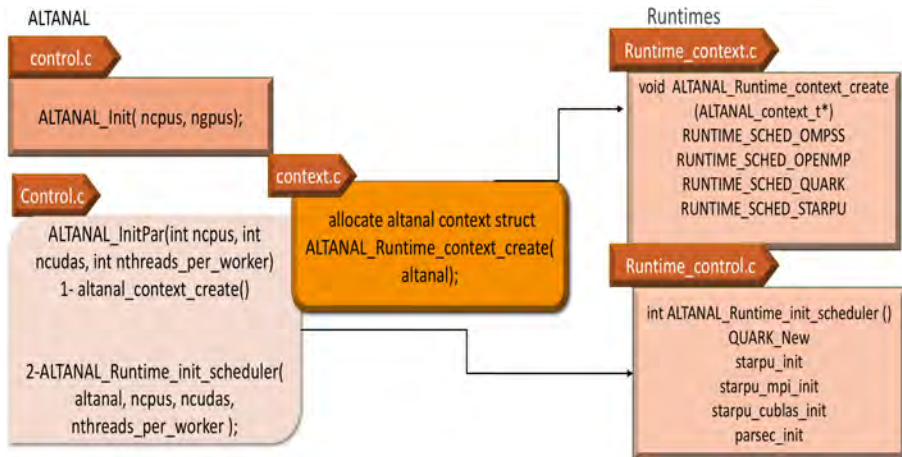
## High-Level API

- 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\_Finalize

ALTANAL

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(altanal)  
free(altanal->schedopt);
```



# ALTANAL Main API

---

**Algorithm 6:** ALTANAL STF Tile Cholesky

---

```
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

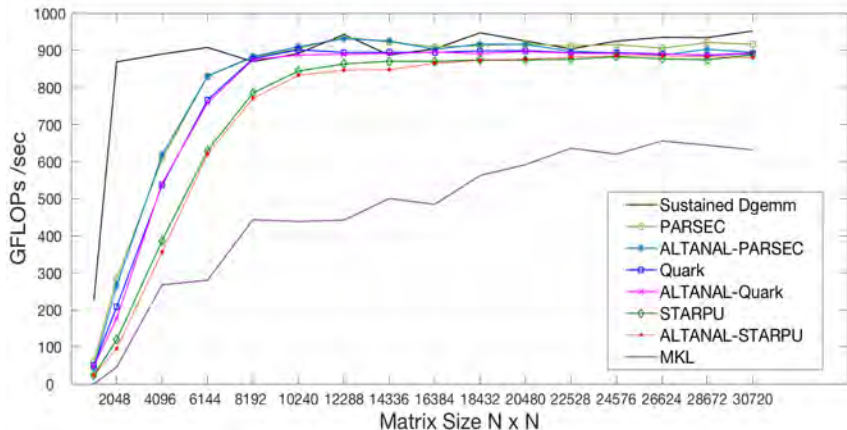
- 1 Motivation
  - Computing Hardware and Software Evolution
  - Task-based Runtime Systems
  - Overview of Task-based ECRC Projects
  - ALTANAL
- 2 Literature Review
  - Overview of Existing Asynchronous Task-based Runtime Systems
  - DARMA
- 3 ALTANAL
  - ALTANAL Layer
  - ALTANAL Interface
- 4 Test 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

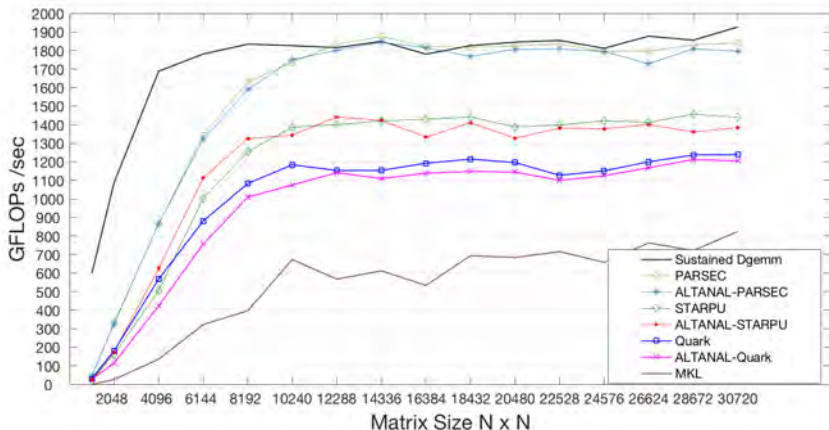
# Dense Cholesky Factorization

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.

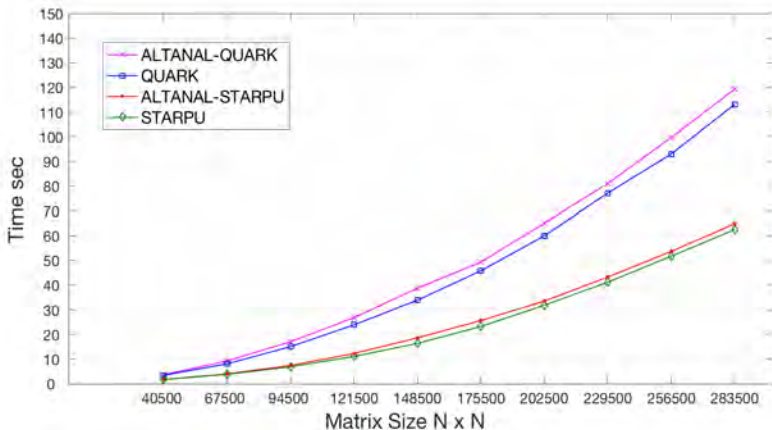


# Table of Contents

- 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

# Tile Low Rank Cholesky Factorization

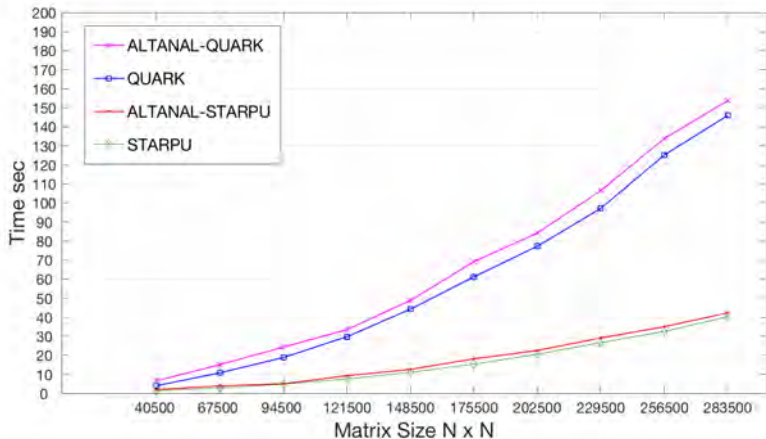
Figure: Dual-socket 18-core Intel(R) Xeon(R) Haswell CPU E5-2699 v3 @ 2.3 GHz with 256GB of main memory, shared memory system





# 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

