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

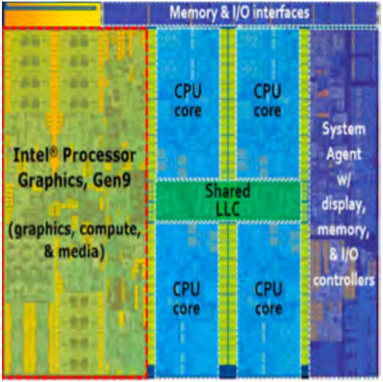
Outline

Table of Contents

Table of Contents

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



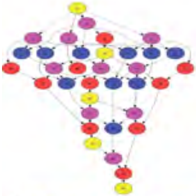
Software Evolution

- 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

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



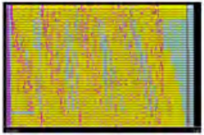
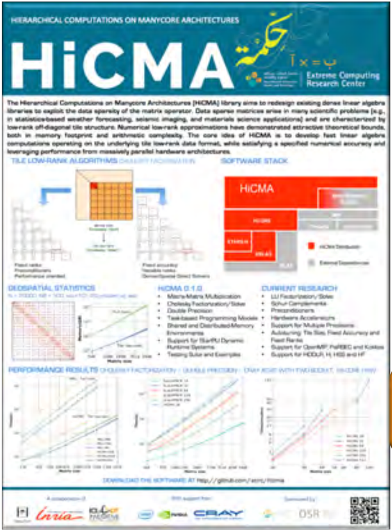


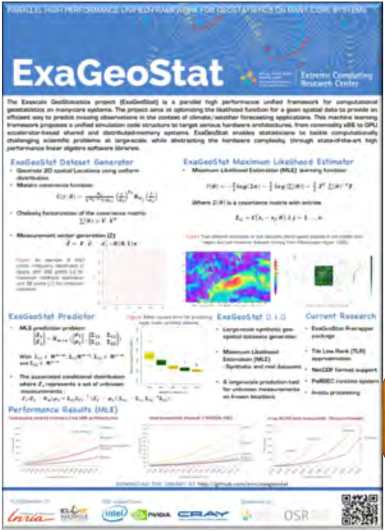
Table of Contents

HiCMA: Hierarchical Computations on Manycore Architectures



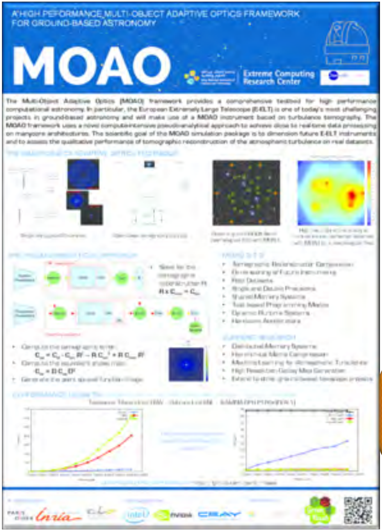
PaRSEC

ExaGeoStat: Exascale GeoStatistics



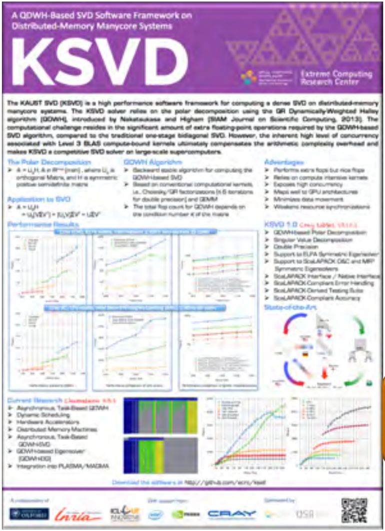
StarPU

MOAO: Multi-Object Adaptive Optics systems



OmpSs

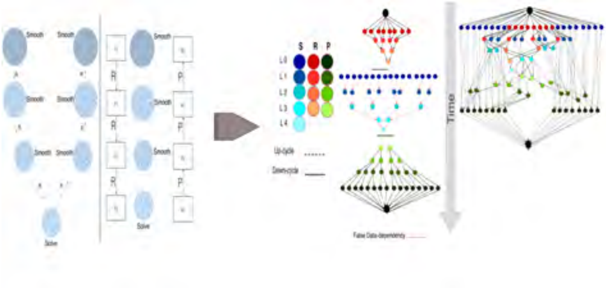
KSVD: KAUST Singular Value Decomposition



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

API Standardization

ALTANAL

- 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 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

Table of Contents

Quark: **QU**euing **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: QUeuing And Runtime for Kernels

```
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);
}
```

StarPU

- 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

```
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]);
    }
```

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

```
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++

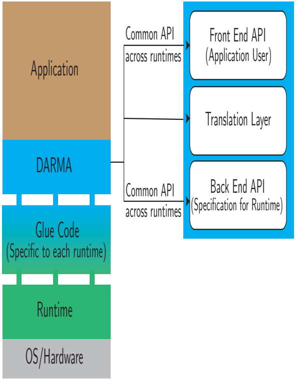
Summary Table of Existing Task-based Runtime Systems

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++, cus-tom
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

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`
- Asynchrous 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`.

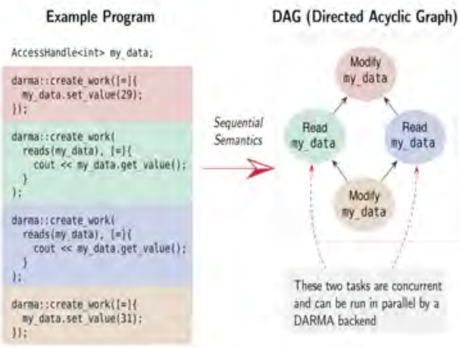
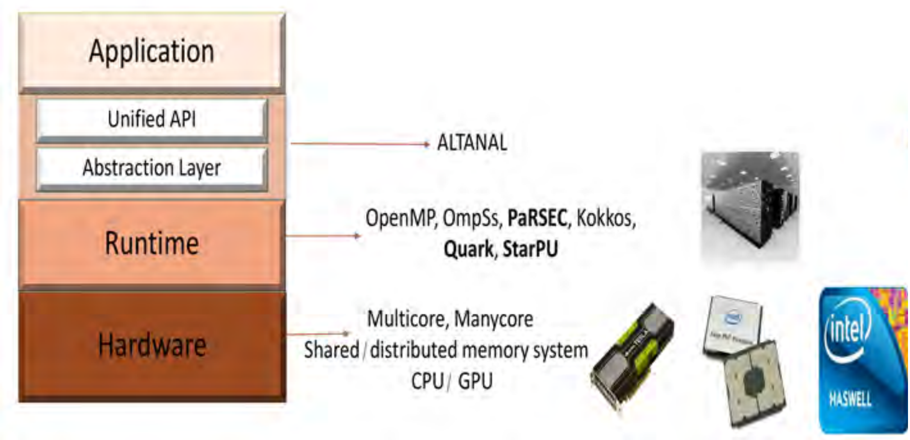


Table of Contents

Table of Contents

ALTANAL Layer



ALTANAL Layer

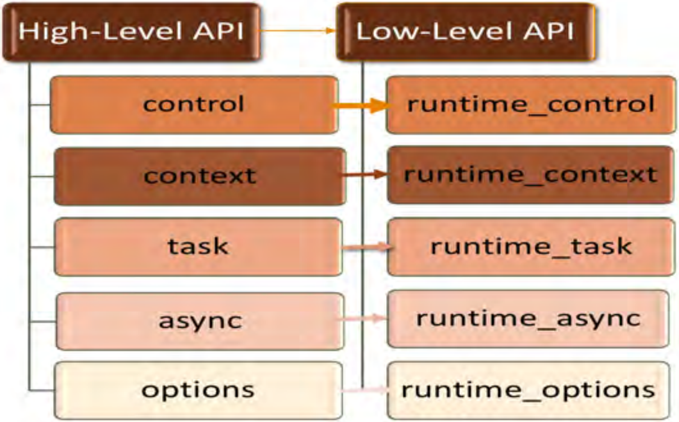


Table of Contents

ALTANAL Interface

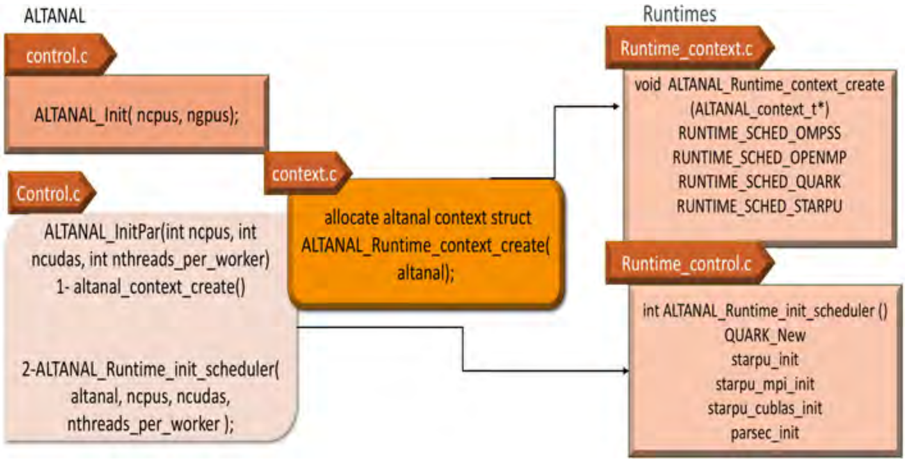
High-Level API

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

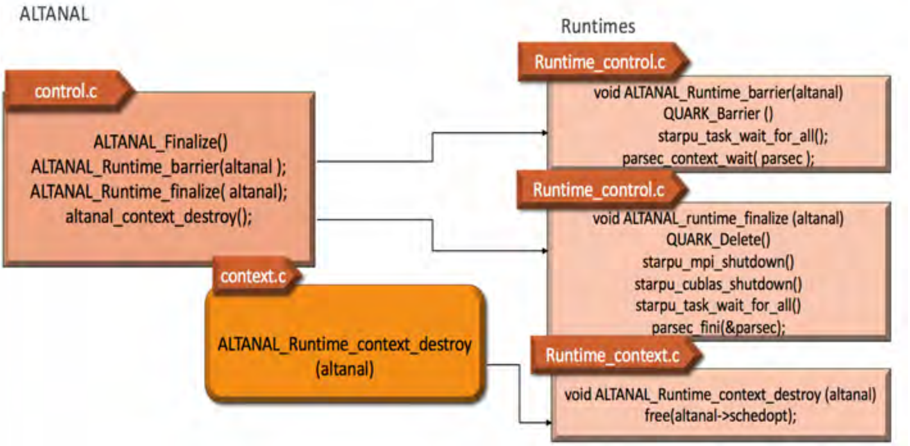
Low-Level API

- 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_Finalize



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

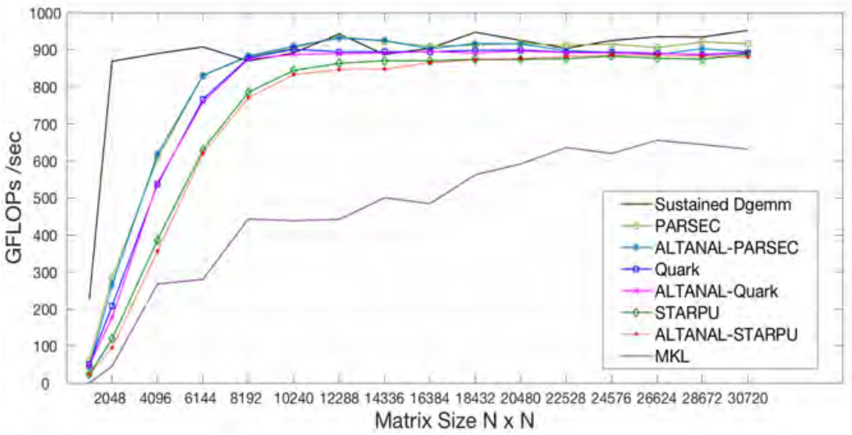
```
ALTANAL_CODELETS(POTRF, POTRF_CPU)

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

Table of Contents

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.

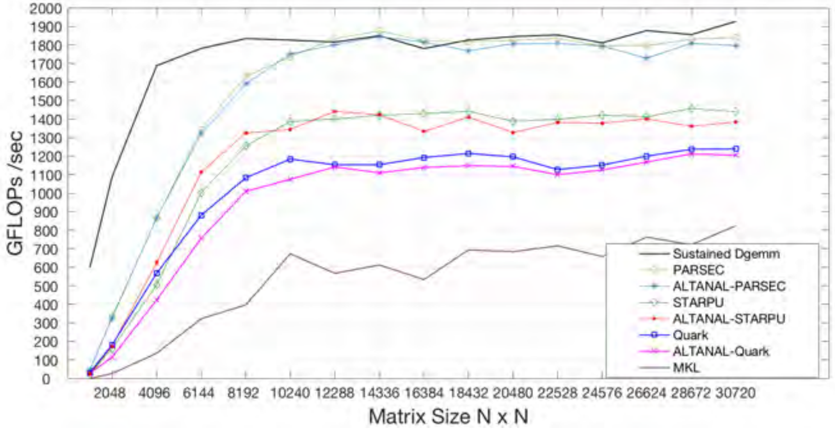
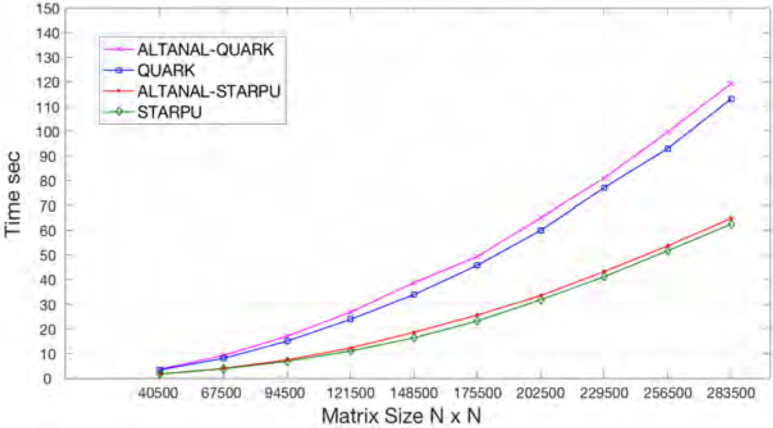


Table of Contents

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

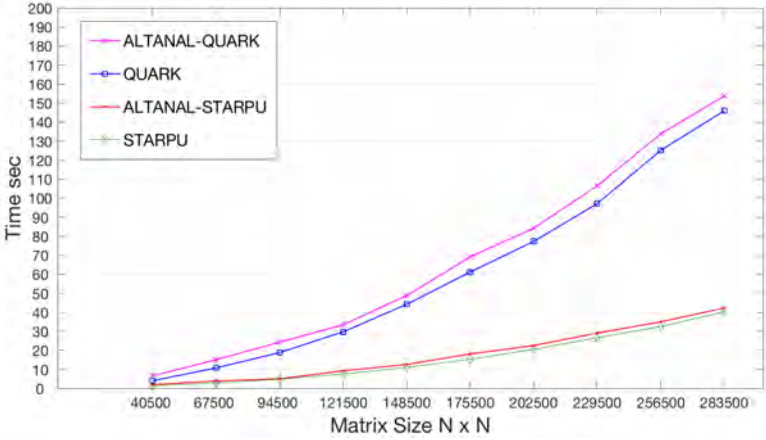


Table of Contents

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