EXA2PRO Runtime System : StarPU

Samuel Thibault
INRIA STORM Team
Introduction
Toward heterogeneous multi-core architectures

• Multicore is here
  • Hierarchical architectures
  • Manycore
  • Heterogeneous systems

• Architecture specialization
  • Now
    – Accelerators (GPGPUs, FPGAs)
    – Coprocessors (Xeon Phi)
    – All of the above
  • In the near Future
    – Many simple cores
    – A few full-featured cores

Mixed Large and Small Cores
Introduction
Toward heterogeneous multi-core clusters

- Multicore is here
  - Hierarchical architectures
  - Manycore
  - Heterogeneous systems

- Clusters thereof
  - High-speed network
  - Network topology
  - Towards exascale
How to program these architectures?

• Multicore programming
  • pthreads, OpenMP, TBB, ...
How to program these architectures?

- Multicore programming
  - pthreads, OpenMP, TBB, ...

- Accelerator programming
  - CUDA, OpenCL, FPGA ?
  - OpenMP 5.0?
  - (Often) Pure offloading model
How to program these architectures?

• Multicore programming
  • pthreads, OpenMP, TBB, ...

• Accelerator programming
  • CUDA, OpenCL, FPGA?
  • OpenMP 5.0?
  • (Often) Pure offloading model

• Network support
  • MPI / PGAS
How to program these architectures?

- Multicore programming
  - pthreads, OpenMP, TBB, ...

- Accelerator programming
  - CUDA, OpenCL, FPGA ?
  - OpenMP 5.0?
  - (Often) Pure offloading model

- Network support
  - MPI / PGAS

- Hybrid models?
  - Take advantage of all resources 😊
  - Complex interactions and distribution 😞
Task graphs

- Well-studied for scheduling parallelism (since 60’s!)
- Departs from usual sequential programming

Really?
Task management
Implicit task dependencies

• Right-Looking Cholesky decomposition (from PLASMA)

```c
for (j = 0; j < N; j++) {
    POTRF (RW, A[j][j]);
    for (i = j+1; i < N; i++)
        TRSM (RW, A[i][j], R, A[j][j]);
    for (i = j+1; i < N; i++)
        SYRK (RW, A[i][i], R, A[i][j]);
    for (k = j+1; k < i; k++)
        GEMM (RW, A[i][k],
              R, A[i][j], R, A[k][j]);
}
task_wait_for_all();
```
Task management
Implicit task dependencies

- Right-Looking Cholesky decomposition (from PLASMA)

```c
for (j = 0; j < N; j++) {
    POTRF (RW, A[j][j]);
    for (i = j+1; i < N; i++) {
        TRSM (RW, A[i][j], R, A[j][j]);
        for (k = j+1; k < i; k++)
            SYRK (RW, A[i][k], R, A[i][j]);
        GEMM (RW, A[i][k], R, A[i][j], R, A[k][j]);
    }
} task_wait_for_all();
```
Task management
Implicit task dependencies

- Right-Looking Cholesky decomposition (from PLASMA)

```c
for (j = 0; j < N; j++) {
    POTRF (RW, A[j][j]);
    for (i = j+1; i < N; i++)
        TRSM (RW, A[i][j], R, A[j][j]);
    for (i = j+1; i < N; i++)
        SYRK (RW, A[i][i], R, A[i][j]);
    for (k = j+1; k < i; k++)
        GEMM (RW, A[i][k],
              R, A[i][j], R, A[k][j]);
}
task_wait_for_all();
```
Task management

Implicit task dependencies

- Right-Looking Cholesky decomposition (from PLASMA)

```c
for (j = 0; j < N; j++) {
    POTRF (RW, A[j][j]);
    for (i = j+1; i < N; i++)
        TRSM (RW, A[i][j], R, A[j][j]);
    for (i = j+1; i < N; i++)
        SYRK (RW, A[i][i], R, A[i][j]);
    for (k = j+1; k < i; k++)
        GEMM (RW, A[i][k],
            R, A[i][j], R, A[k][j]);
}
task_wait_for_all();
```
Task management
Implicit task dependencies

• Right-Looking Cholesky decomposition (from PLASMA)

```c
for (j = 0; j < N; j++) {
    POTRF (RW, A[j][j]);
    for (i = j+1; i < N; i++)
        TRSM (RW, A[i][j], R, A[j][j]);
    for (i = j+1; i < N; i++)
        SYRK (RW, A[i][i], R, A[i][j]);
    for (k = j+1; k < i; k++)
        GEMM (RW, A[i][k],
              R, A[i][j], R, A[k][j]);
}
task_wait_for_all();
```
Task management
Implicit task dependencies

- Right-Looking Cholesky decomposition (from PLASMA)

```c
for (j = 0; j < N; j++) {
    POTRF (RW, A[j][j]);
    for (i = j+1; i < N; i++)
        TRSM (RW, A[i][j], R, A[j][j]);
    for (i = j+1; i < N; i++)
        SYRK (RW, A[i][i], R, A[i][j]);
    for (k = j+1; k < i; k++)
        GEMM (RW, A[i][k],
               R, A[i][j], R, A[k][j]);
}
task_wait_for_all();
```
Task management
Implicit task dependencies

• Right-Looking Cholesky decomposition (from PLASMA)

```c
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++)
    SYRK (RW,A[i][i], R,A[i][j]);
  for (k = j+1; k < i; k++)
    GEMM (RW,A[i][k],
        R,A[i][j], R,A[k][j]);
}
task_wait_for_all();
```
Task management

Implicit task dependencies

- Right-Looking Cholesky decomposition (from PLASMA)

```c
for (j = 0; j < N; j++) {
    POTRF (RW, A[j][j]);
    for (i = j+1; i < N; i++)
        TRSM (RW, A[i][j], R, A[j][j]);
    for (i = j+1; i < N; i++)
        SYRK (RW, A[i][i], R, A[i][j]);
    for (k = j+1; k < i; k++)
        GEMM (RW, A[i][k],
              R, A[i][j], R, A[k][j]);
}
task_wait_for_all();
```
Task management
Implicit task dependencies

- Right-Looking Cholesky decomposition (from PLASMA)

```c
for (j = 0; j < N; j++) {
    POTRF (RW, A[j][j]);
    for (i = j+1; i < N; i++)
        TRSM (RW, A[i][j], R, A[j][j]);
    for (i = j+1; i < N; i++)
        SYRK (RW, A[i][i], R, A[i][j]);
    for (k = j+1; k < i; k++)
        GEMM (RW, A[i][k],
             R, A[i][j], R, A[k][j]);
}
task_wait_for_all();
```
Task management
Implicit task dependencies

- Right-Looking Cholesky decomposition (from PLASMA)

```c
for (j = 0; j < N; j++) {
    POTRF (RW, A[j][j]);
    for (i = j+1; i < N; i++)
        TRSM (RW, A[i][j], R, A[j][j]);
    for (i = j+1; i < N; i++)
        SYRK (RW, A[i][i], R, A[i][j]);
    for (k = j+1; k < i; k++)
        GEMM (RW, A[i][k],
              R, A[i][j], R, A[k][j]);
}
task_wait_for_all();
```
Task management
Implicit task dependencies

• Right-Looking Cholesky decomposition (from PLASMA)

```c
for (j = 0; j < N; j++) {
    POTRF (RW, A[j][j]);
    for (i = j+1; i < N; i++)
        TRSM (RW, A[i][j], R, A[j][j]);
    for (i = j+1; i < N; i++)
        SYRK (RW, A[i][i], R, A[i][j]);
    for (k = j+1; k < i; k++)
        GEMM (RW, A[i][k],
             R, A[i][j], R, A[k][j]);
}
task_wait_for_all();
```
Task management
Implicit task dependencies

- Right-Looking Cholesky decomposition (from PLASMA)

```c
for (j = 0; j < N; j++) {
    POTRF (RW,A[j][j]);
    for (i = j+1; i < N; i++)
        TRSM (RW,A[i][j], R,A[j][j]);
    for (i = j+1; i < N; i++) {
        SYRK (RW,A[i][i], R,A[i][j]);
        for (k = j+1; k < i; k++)
            GEMM (RW,A[i][k],
                R,A[i][j], R,A[k][j]);
    }
}
task_wait_for_all();
```
Task management

Implicit task dependencies

• Right-Looking Cholesky decomposition (from PLASMA)

```c
for (j = 0; j < N; j++) {
    POTRF (RW, A[j][j]);
    for (i = j+1; i < N; i++)
        TRSM (RW, A[i][j], R, A[j][j]);
    for (i = j+1; i < N; i++)
        SYRK (RW, A[i][i], R, A[i][j]);
    for (k = j+1; k < i; k++)
        GEMM (RW, A[i][k], R, A[i][j], R, A[k][j]);
}
task_wait_for_all();
```
Write your application as a task graph

Even if using a sequential-looking source code
➔ Portable performance

Sequential Task Flow (STF)

• Algorithm remains the same on the long term
• Can debug the sequential version.
• Only kernels need to be rewritten
  • BLAS libraries, multi-target compilers
• Runtime will handle parallel execution
Task-based programming

• Needs code restructuring
  • Split computation into tasks
    – BLAS, typically
    – Supposed to have “stable” performance

• Constraining
  • No global variables
    – Mandatory for GPUs

• Actually… functional programming
So a good move, in the end 😊

• Have to accept constraints and losing control
Just like we did when moving from assembly to high-level languages
EXA2PRO stack

HPC Application (C, C++, Fortran, MPI, OpenMP)
- Skeletons API
- Multi-Variant Components API
- Smart Data-Containers API
- Platform Query API
- Fault Tolerance API

Technical Debt Management Tools

Composition Framework
- ComPU Composition Control
- Component Metadata IR (COMIR)
- Source-to-Source Compiler IR = Mercurium compiler IR
- ComPU plugins
- Common IR Access Layer
- mcxx plug-ins
- Composition Plug-ins
  - Granularity/Local Optimization
  - Performance Modeling
  - Fault Tolerance (pragma syntax)

Platform Modeling Framework

Platform Toolchain
- CPU Toolchain
- CUDA Toolchain
- OpenCL Toolchain
- MAxeler Toolchain

Component Metadata IR (COMIR)

Runtime System (StarPU)
- Fault Tolerance
- Load Balancing
- Multi-criteria Scheduling

HPC Cluster with Accelerators
- CPU
- GPU
- CPU
- GPU
- CPU
- DFE
- ...

https://exa2pro.eu
Overview of StarPU
Overview of StarPU

Rationale

Task scheduling
- Dynamic
- On all kinds of PU
  - General purpose
  - Accelerators/specialized

Memory transfer
- Eliminate redundant transfers
- Software VSM (Virtual Shared Memory)
The StarPU runtime system

The need for runtime systems

• “do dynamically what can’t be done statically anymore”

• Compilers and libraries generate (graphs of) tasks
  • Additional information is welcome!

• StarPU provides
  • Task scheduling
  • Memory management
Data management

• StarPU provides a Virtual Shared Memory (VSM) subsystem (aka DSM)
  • Replication
  • Consistency
  • Single writer
    – Or reduction, ...

• Input & output of tasks = reference to VSM data
The StarPU runtime system

Task scheduling

- **Tasks =**
  - Data input & output
    - Reference to VSM data
  - Multiple implementations
    - E.g. CUDA + CPU implementation
  - Non-preemptible
  - Dependencies with other tasks

- StarPU provides an **Open Scheduling platform**
  - Scheduling algorithm = plug-ins

---

The StarPU runtime system

Task scheduling

- **Tasks =**
  - Data input & output
    - Reference to VSM data
  - Multiple implementations
    - E.g. CUDA + CPU implementation
  - Non-preemptible
  - Dependencies with other tasks

- StarPU provides an **Open Scheduling platform**
  - Scheduling algorithm = plug-ins
The StarPU runtime system

Task scheduling

• Who generates the code?
  • StarPU Task ~= function pointers
  • StarPU doesn't generate code

• Libraries era
  • PLASMA + MAGMA
  • FFTW + CUFFT…
  • Variants management

• Rely on compilers
The StarPU runtime system

HPC Applications

High-level data management library

Execution model

Scheduling engine

Specific drivers

CPUs  GPUs  SPUs  ...

Mastering CPUs, GPUs, SPUs … *PUs → StarPU
The StarPU runtime system
Execution model

Application

Memory Management (DSM)

Scheduling engine

GPU driver

CPU driver #k

RAM

GPU

CPU#k

https://exa2pro.eu
The StarPU runtime system

Execution model

Submit task « A += B »
The StarPU runtime system

Execution model

Schedule task

A += B

Memory Management (DSM)

Application

Scheduling engine

GPU driver

CPU driver #k

CPU#k

RAM

GPU

A

B

A

B

A

B

https://exa2pro.eu
The StarPU runtime system
Execution model

Application

Scheduling engine

Memory Management (DSM)

A

B

A += B

GPU driver

Fetch data

CPU driver

CPU #k

RAM

GPU

CPU #k
The StarPU runtime system

Execution model

- Memory Management (DSM)

RAM

A

B

A+= B

A

B

Fetch data

CPU driver #k

GPU driver

GPU

CPU #k

https://exa2pro.eu
The StarPU runtime system
Execution model

Application

Scheduling engine

Memory Management (DSM)

A += B

CPU driver #k

CPU#k

Fetch data

GPU driver

GPU

RAM

A

B

A

B

A

B

https://exa2pro.eu
The StarPU runtime system
Execution model

Application

Scheduling engine

Memory Management (DSM)

RAM

GPU driver

CPU driver #k

Offload computation

A += B
The StarPU runtime system

Execution model

Application

Scheduling engine

Memory Management (DSM)

CPU driver

GPU driver

GPU

RAM

Notify termination
The StarPU runtime system

Development context

• **History**
  • Started about 9 years ago
    – PhD Thesis of Cédric Augonnet
  • StarPU main core ≈ 70k lines of code
  • Written in C

• **Open Source**
  • Released under LGPL
  • Sources freely available
    – git repository and nightly tarballs
    – See https://starpu.gitlabpages.inria.fr/
  • Open to external contributors

• [HPPC'08]
• [Europar'09] – [CCPE'11],... >1500 citations
Task-based programming actually makes things easier!

- **QR-Mumps** (sparse linear algebra)
  - Non-task version: only 1D decomposition
  - Task version: 2D decomposition, flurry of parallelism
    - With seamless memory control
- **H-Matrices** (compressed linear algebra, AirBus)
  - Out-of-core support
    - Could run cases unachievable before
    - e.g. 1600 GB matrix with 256 GB memory
  - Shipped to AirBus customers
- Implemented CFD, FMM, CG, stencils, …
The StarPU runtime system

Supported platforms

• **Supported architectures**
  - Multicore CPUs (x86, PPC, ...)
  - NVIDIA GPUs
  - OpenCL devices (eg. AMD cards)
  - Intel Xeon Phi (MIC)
  - FPGA (ongoing)
  - Intel SCC, Kalray MPPA, Cell (decommissioned)

• **Supported Operating Systems**
  - Linux
  - Mac OS
  - Windows
Task-based support

Then all of this comes “for free”:

- Task/data scheduling
  - Pipelining
  - Load balancing
  - GPU memory limitation management
  - Data prefetching
- Performance bounds
- Distributed execution through MPI
- High-level performance analysis
- Out-of-core: optimized swapping to disk
- Debugging sequential execution
- Reproducible performance simulation
Task Scheduling
Why do we need task scheduling?

Blocked Matrix multiplication

Things can go (really) wrong even on trivial problems!

- Static mapping?
  - Not portable, too hard for real-life problems
- Need Dynamic Task Scheduling
  - Performance models

2 Xeon cores
Quadro FX5800
Quadro FX4600
Runtime-based task scheduling

When a task is submitted, it first goes into a pool of “frozen tasks” until all dependencies are met.

Then, the task is “pushed” to the scheduler.

Idle processing units poll for work (“pop”).

Various scheduling policies, can even be user-defined.

Scheduler

Push

Pop

Pop

CPU workers

GPU workers
Runtime-based task scheduling

When a task is submitted, it first goes into a pool of “frozen tasks” until all dependencies are met.

Then, the task is “pushed” to the scheduler.

Idle processing units poll for work (“pop”).

Various scheduling policies, can even be user-defined.
Runtime-based task scheduling

When a task is submitted, it first goes into a pool of “frozen tasks” until all dependencies are met.

Then, the task is “pushed” to the scheduler.

Idle processing units poll for work (“pop”).

Various scheduling policies, can even be user-defined.
Runtime-based task scheduling

When a task is submitted, it first goes into a pool of “frozen tasks” until all dependencies are met.

Then, the task is “pushed” to the scheduler.

Idle processing units poll for work (“pop”).

Various scheduling policies can even be user-defined.
Prediction-based scheduling
Load balancing

- Task completion time estimation
  - History-based
  - User-defined cost function
  - Parametric cost model
  - [HPPC'09]
- Can be used to implement scheduling
  - E.g. Heterogeneous Earliest Finish Time

![Diagram showing CPU and GPU tasks with time轴]
Prediction-based scheduling
Load balancing

- Task completion time estimation
  - History-based
  - User-defined cost function
  - Parametric cost model
  - [HPPC'09]
- Can be used to implement scheduling
  - E.g. Heterogeneous Earliest Finish Time
Prediction-based scheduling

Load balancing

- Task completion time estimation
  - History-based
  - User-defined cost function
  - Parametric cost model
  - [HPPC'09]
- Can be used to implement scheduling
  - E.g. Heterogeneous Earliest Finish Time
Prediction-based scheduling

Load balancing

- Task completion time estimation
  - History-based
  - User-defined cost function
  - Parametric cost model
  - [HPPC'09]
- Can be used to implement scheduling
  - E.g. Heterogeneous Earliest Finish Time
Predicting data transfer overhead

Motivations

- Hybrid platforms
  - Multicore CPUs and GPUs
  - PCI-e bus is a precious resource

- Data locality vs. Load balancing
  - Cannot avoid all data transfers
  - Minimize them

- StarPU keeps track of
  - data replicates
  - on-going data movements
Prediction-based scheduling
Load balancing

• Data transfer time
  • Sampling based on off-line calibration
• Can be used to
  • Better estimate overall exec time
  • Minimize data movements
• Further
  • Power overhead
• **dmda** [ICPADS'10]
Mixing PLASMA and MAGMA with StarPU

- QR decomposition
  - Mordor8 (UTK) : 16 CPUs (AMD) + 4 GPUs (C1060)
Mixing PLASMA and MAGMA with StarPU

- QR decomposition
  - Mordor8 (UTK) : 16 CPUs (AMD) + 4 GPUs (C1060)
Mixing PLASMA and MAGMA with StarPU

- QR decomposition
  - Mordor8 (UTK) : 16 CPUs (AMD) + 4 GPUs (C1060)

+12 CPUs
~200GFlops
vs measured
~150Gflops!
Thanks to heterogeneity
Mixing PLASMA and MAGMA with StarPU

• QR decomposition
  • Mordor8 (UTK) : 16 CPUs (AMD) + 4 GPUs (C1060)

![Graph showing performance comparison between MAGMA and StarPU configurations.](https://exa2pro.eu)

+12 CPUs
~200GFlops

vs measured
~150Gflops!

Thanks to heterogeneity
Mixing PLASMA and MAGMA with StarPU

• « Super-Linear » efficiency in QR?
  • Kernel efficiency
    - sgeqrt
      - CPU: 9 Gflops  GPU: 30 Gflops (Speedup : ~3)
    - stsqrt
      - CPU: 12 Gflops  GPU: 37 Gflops (Speedup: ~3)
    - somqr
      - CPU: 8.5 Gflops  GPU: 227 Gflops (Speedup: ~27)
    - Sssmqr
      - CPU: 10 Gflops  GPU: 285 Gflops (Speedup: ~28)
  • Task distribution observed on StarPU
    - sgeqrt: 20% of tasks on GPUs
    - Sssmqr: 92.5% of tasks on GPUs
  • Taking advantage of heterogeneity!
    - Only do what you are good for
    - Don't do what you are not good for
Cluster support
How to scale over MPI?

(StarPU handles intra-MPI node scheduling fine)

- Splitting graph by hand
  - Complex, not flexible
- Master-Slave does not scale
  - Each node should determine its duty by itself
- Algebraic representation of e.g. Parsec
  - Difficult to write
  - Not flexible enough for any kind of application
- Recursive task graph unrolling
  - Complex
  - Rather just unroll the whole task graph on each node
Automatic generation of Send/Recv
MPI VSM

- Application decides data distribution over MPI nodes
- But data coherency extended to the MPI level
  - Automatic starpu_mpi_send/recv calls for each task
- Similar to a DSM, but granularity is whole data and whole task
  - All nodes process the whole algorithm
    - Actual task execution according to data being written to Sequential-looking code!
For (k = 0 .. tiles – 1) {
  POTRF(A[k,k])
  for (m = k+1 .. tiles – 1)
    TRSM(A[k,k], A[m,k])
  for (m = k+1 .. tiles – 1) {
    SYRK(A[m,k], A[m,m])
    for (n = m+1 .. tiles – 1)
      GEMM(A[m,k], A[n,k], A[n,m])
  }
}
MPI VSM

• Data mapping (e.g. 2D block-cyclic)

int get_rank(int m, int n) { return ((m%p)*q + n%q); }

For (m = 0 .. tiles – 1)
  For (n = m .. tiles – 1)
    set_rank(A[m,n], get_rank(m,n));

For (k = 0 .. tiles – 1) {
  POTRF(A[k,k])
  for (m = k+1 .. tiles – 1)
    TRSM(A[k,k], A[m,k])
  for (m = k+1 .. tiles – 1) {
    SYRK(A[m,k], A[m,m])
    for (n = m+1 .. tiles – 1)
      GEMM(A[m,k], A[n,k], A[n,m])
  }
}
MPI VSM

• Each node unrolls the whole task graph
  • Data ↔ node mapping
    • Provided by the application
      – E.g. 2D block-cyclic
    • Can be modified during submission
      starpu_mpi_data_migrate()
  • Task ↔ node mapping
    • Tasks move to data they modify
  • Separation of concerns: graph vs mapping
  • MPI transfers
    • Automatically queued
  • Local view of the computation
    • No synchronizations
    • No global scheduling
MPI VSM

- Right-Looking Cholesky decomposition (from PLASMA)
Cholesky cluster performance

@CEA: 144 nodes with 8 CPU cores (E5620) + 2 GPUs (M2090)
Simulation
Simulation with SimGrid

Calibration

From A. Legrand and L. Stanisic

Run once!
Simulation with SimGrid

From A. Legrand and L. Stanisic

Run once!

Quickly Simulate Many Times
Simulation with SimGrid

- Run application natively on target system
  - Records performance models
- Rebuild application against simgrid-compiled StarPU
- Run again
  - Uses performance model estimations instead of actually executing tasks

- Way faster execution time
- Reproducible experiments
- No need to run on target system
- Can change system architecture
Simulation with SimGrid

- Way faster execution time
- Reproducible experiments
- No need to run on target system
- Can change system architecture
Conclusion

Task graphs

- Nice programming model
  - Keep sequential program!
- Optimized execution
- Playground for research
  - Scheduling
  - Fault Tolerance
  - Statistics

- Used for various real-world computations
  - Cholesky/QR/LU (dense/sparse/compressed), stencil, CG, CFD, FMM…

http://starpu.gitlabpages.inria.fr/tutorials/
StarPU Tutorial on February 24h

- To be run in a docker container
- Please follow the EXA2PRO Getting Started Guide
  - See attachment in the timetable of the event
  - Section 2 « Installation »
  - Takes 1/2h - 1h