A Reproducible Research Methodology for Designing and Conducting Faithful Simulations of Dynamic Task-based Scientific Application

Luka Stanisic

Inria, Bordeaux Sud-Ouest, France

MPCDF seminar
Garching
February 24, 2017
**Background**

**Bachelor**
(CS specialty)
EE faculty
*Belgrade, Serbia*

**PhD**
(supervisors A. Legrand & J.F. Mehaut)
*Grenoble, France*
- Modeling and simulation of dynamic task-based applications
- Methodology for reproducible research
- Statistical analysis, trace visualizations

**Research Master**
(parallelism specialty)
*Grenoble, France*
- Benchmarking
- CPU cache modeling
- ARM vs Intel

**PostDoc**
*Bordeaux, France*
- Performance optimization
- Large scale simulations
- Modeling complex kernels
- Simulating openQCD
Background

**Bachelor**

(CS specialty)  
EE faculty  
Belgrade, Serbia

**Phd**

(supervisors A. Legrand & J.F. Mehaut)  
Grenoble, France

Modeling and simulation of dynamic task-based applications  
Methodology for reproducible research  
Statistical analysis, trace visualizations

**Research Master**

(parallelism specialty)  
Grenoble, France  
Benchmarking  
CPU cache modeling  
ARM vs Intel

**PostDoc**

Bordeaux, France  
Performance optimization  
Large scale simulations  
Modeling complex kernels  
Simulating openQCD
Parallel Programming Challenges

- Communications and data placement
- Synchronization of the workers
- Computation duration variability
  \(\rightsquigarrow\) scalability
- Exploiting hybrid machines
- Choosing granularity
  \(\rightsquigarrow\) portability of code and performance
Different Programming Approaches

- Traditional, explicit programming models
  (MPI, CUDA, OpenMP, pthreads, . . .)
    - Perfect control $\Rightarrow$ maximal achievable performance
    - Efficient granularity $\Rightarrow$ advanced numerical features
Different Programming Approaches

- Traditional, explicit programming models (MPI, CUDA, OpenMP, pthreads, ...)
  - Happy face: Perfect control \(\Rightarrow\) maximal achievable performance
  - Happy face: Efficient granularity \(\Rightarrow\) advanced numerical features
  - Sad face: Monolithic codes \(\Rightarrow\) hard to develop and maintain
  - Sad face: Fixed scheduling \(\Rightarrow\) sensitive to variability
  - Sad face: Hard and long to optimize \(\Rightarrow\) performance portability
Different Programming Approaches

- **Traditional, explicit programming models**
  (MPI, CUDA, OpenMP, pthreads, ...)
  - Perfect control $\Rightarrow$ maximal achievable performance
  - Efficient granularity $\Rightarrow$ advanced numerical features
  - Monolithic codes $\Rightarrow$ hard to develop and maintain
  - Fixed scheduling $\Rightarrow$ sensitive to variability
  - Hard and long to optimize $\Rightarrow$ performance portability

- **Recent task-based programming models**
  (PaRSEC, OmpSs, Charm++, StarPU, ...)
  - Single, abstract programming model based on DAG
  - Runtime system responsible for dynamic scheduling
  - Portability of code and performance
Different Programming Approaches

- **Traditional, explicit programming models (MPI, CUDA, OpenMP, pthreads, ...)**
  - Perfect control $\leadsto$ maximal achievable performance
  - Efficient granularity $\leadsto$ advanced numerical features
  - Monolithic codes $\leadsto$ hard to develop and maintain
  - Fixed scheduling $\leadsto$ sensitive to variability
  - Hard and long to optimize $\leadsto$ performance portability

- **Recent task-based programming models (PaRSEC, OmpSs, Charm++, StarPU, ...)**
  - Single, abstract programming model based on DAG
  - Runtime system responsible for dynamic scheduling
  - Portability of code and performance
  - Introducing runtime system overhead
  - Developing omnipotent runtime $\leadsto$ new challenges
Tiled Cholesky Algorithm
using Sequential Task Flow (STF)

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]);
    }
} wait();
Task-based Programming Paradigm

Tiled Cholesky Algorithm
using Sequential Task Flow (STF)

```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]);
    }
}
wait();
```
Task-based Programming Paradigm

Tiled Cholesky Algorithm using Sequential Task Flow (STF)

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]);
    }
}
wait();
Task-based Programming Paradigm

Tiled Cholesky Algorithm
using Sequential Task Flow (STF)

```cpp
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]);
        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]);
    }
}
wait();
```
Tiled Cholesky Algorithm using Sequential Task Flow (STF)

```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]);
        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]);
    }
} wait();
```
Task-based Programming Paradigm

Tiled Cholesky Algorithm using Sequential Task Flow (STF)

```
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]);
    }
}  
wait();
```
Task-based Programming Paradigm

Tiled Cholesky Algorithm
using Sequential Task Flow (STF)

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]);
}

wait();
Tiled Cholesky Algorithm using Sequential Task Flow (STF)

```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]);
}
}
wait();
```
Task-based Programming Paradigm

Tiled Cholesky Algorithm
using Sequential Task Flow (STF)

```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]);
}
wait();
```
Task-based Programming Paradigm

Tiled Cholesky Algorithm
using Sequential Task Flow (STF)

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]);
}
wait();
Tiled Cholesky Algorithm
using Sequential Task Flow (STF)

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]);
        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]);
    }
} wait();
Tiled Cholesky Algorithm using Sequential Task Flow (STF)

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]);
    }
  }
} wait();
Tiled Cholesky Algorithm
using Sequential Task Flow (STF)

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]);
        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]);
    }
} wait();
Need Regular Performance Evaluation

Native experiments
- Complex systems
- Wide variety of setups
- Faithful but expensive

Model, equations, theory
- PRAM, BSP, DAG
- Scheduling bounds
- Quick trends but simplistic
Need Regular Performance Evaluation

Native experiments
- Complex systems
- Wide variety of setups
- Faithful but expensive

Model, equations, theory
- PRAM, BSP, DAG
- Scheduling bounds
- Quick trends but simplistic

Simulation: running real code with machine abstraction

Advantages:
- Reproducible executions (performance, bugs)
- Predictions on unavailable architectures (extrapolation)
- Richer experimental design possible

Difficulties:
- Implementing more than a simple prototype
- Hard to validate its reliability
Is it possible to perform a clean, coherent, reproducible study of HPC applications executed on top of dynamic task-based runtime systems, using simulation?
Is it possible to perform a clean, coherent, reproducible study of HPC applications executed on top of dynamic task-based runtime systems, using simulation?
Is it possible to perform a clean, coherent, reproducible study of HPC applications executed on top of dynamic task-based runtime systems, using simulation?
Outline

1. Simulating Task-based Applications
   - Coupling StarPU Runtime System and SimGrid Simulator
   - Tackling Irregular Numerical Codes
   - Use Cases
   - Summary

2. Methodology for Reproducible Research

3. Conclusion
StarPU and SimGrid

StarPU (Inria Bordeaux)
- Dynamic runtime system for hybrid architectures (CPU, GPU, MPI)
- Opportunistic scheduling of a task graph guided by performance models
- Features dense, sparse and FMM applications

SimGrid (Inria Grenoble, Lyon, Rennes, ...)
- Scalable simulation framework for distributed systems
- Sound fluid network models accounting for heterogeneity and contention
- Modeling with threads rather than only trace replay
  → ability to simulate dynamic applications
- Portable, open source and easily extendable

Same approach could be applicable to any task-based runtime system
Devised Workflow: StarPU + SimGrid

Calibration

Run once!

StarPU

Performance Profile
Devised Workflow: StarPU + SimGrid

Calibration

Simulation

Quickly Simulate Many Times

Performance Profile

Run once!

Quickly Simulate Many Times
Implementation Principles

**Emulation** executing real applications in a synthetic environment

**Simulation** replace process execution by delays using performance models

- StarPU applications and runtime system are *emulated*
  - $\Rightarrow$ similar scheduling
- Thread synchronizations, actual computations, memory allocations and data transfers are *simulated*
  - $\Rightarrow$ need for a good computational kernel and communication models
- Control part of StarPU is modified to inject into SimGrid runtime system, communication and computation delays
Simulation delays (increasing simulated time)

- Process synchronizations
- Memory allocations of CPU or GPU
- Submission of data transfer requests

Example for CUDA memory allocation in StarPU

```c
...  
#ifdef STARPU_SIMGRID
   MSG_process_sleep((float) dim * alloc_cost_per_byte);
#else
   if (_starpu_can_submit_cuda_task()) {
      cudaError_t cures;
      cures = cudaHostAlloc(A, dim, cudaHostAllocPortable);
...  
```
Components of hybrid platforms have differing characteristics.
Correctly modeling their communication is of primary importance.
Built on exhaustively validated existing SimGrid network models.
Components of hybrid platforms have differing characteristics.
Correctly modeling their communication is of primary importance.
Built on exhaustively validated existing SimGrid network models.
- Actual computation results irrelevant \(\leadsto\) only computation time matters
- Task = Kernel for task-based paradigm
- Execution of tasks replaced by simulation delays
- Average duration for regular kernels
- Additional techniques to optionally account for variability
Overview of Simulation Accuracy

<table>
<thead>
<tr>
<th>Experiment Type</th>
<th>Native</th>
<th>SimGrid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hannibal: 3 QuadroFX5800</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attila: 3 TeslaC2050</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mirage: 3 TeslaM2070</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conan: 3 TeslaM2075</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Matrix dimension

GFlop/s

Experiment

Type

Native

SimGrid

Publications


Overview of Simulation Accuracy

Publications


1 Simulating Task-based Applications
   • Coupling StarPU Runtime System and SimGrid Simulator
   • Tackling Irregular Numerical Codes
     • Use Cases
     • Summary

2 Methodology for Reproducible Research

3 Conclusion
- Regular kernels use always the same block size
  $\Rightarrow$ duration is (relatively) stable
- Irregular kernel durations depend on their input parameters
  $\Rightarrow$ need more than simple average values
Modeling Duration of Complex Computation Kernels

- Using statistical analysis and **multiple linear regression**
- Extended StarPU to automatically support such models
- Kernel duration estimations useful for both simulation and native executions (scheduling)
Simulating Irregular Numerical Libraries

- Chameleon solver: dense linear algebra library
- qr_mumps solver: MUMPS multi-frontal factorization
- ScalFMM library: simulating N-body interactions using the FMM
- QDWH solver: QR-based Dynamically Weighted Halley (ongoing)

Publication


Outline

1. Simulating Task-based Applications
   - Coupling StarPU Runtime System and SimGrid Simulator
   - Tackling Irregular Numerical Codes
   - Use Cases
   - Summary

2. Methodology for Reproducible Research

3. Conclusion
Comparing Different Schedulers

- Differences between schedulers performances faithfully predicted
- DMDAR and DMDAS locality aware schedulers
  \[\Rightarrow\] less transfers between GPUs

![Graph showing performance comparison between DMDA, DMDAR, and DMDAS for different matrix dimensions. The graph plots GFlop/s against matrix dimension for each experiment type (Native and SimGrid).]
Minimizing memory footprint is very important for such applications. Remember scheduling is dynamic so consecutive Native experiments have different output.

Experiment number 1

Experiment number 2

Experiment number 3

Experiment number 4
Minimizing memory footprint is very important for such applications.
Remember scheduling is dynamic so consecutive Native experiments have different output.
Extrapolating to Larger Machines

- Predicting performance in idealized context
- Studying the parallelization limits of the problem

![Graph showing extrapolation](image)
Outline

1. Simulating Task-based Applications
   - Coupling StarPU Runtime System and SimGrid Simulator
   - Tackling Irregular Numerical Codes
   - Use Cases
   - Summary

2. Methodology for Reproducible Research

3. Conclusion
Achievements

- Works great for small hybrid setups with dense, sparse and FMM StarPU applications
- Not only a prototype, already used by other researchers
- Our solution allows to:
  - Debug applications on a commodity laptop in a reproducible way
  - Detect problems with real experiments using reliable comparison
  - Test different scheduling alternatives
  - Evaluate memory footprint
  - Quickly and accurately evaluate the impact of various scheduling/application parameters:

<table>
<thead>
<tr>
<th>qr_mumps</th>
<th>Cores</th>
<th>RAM</th>
<th>Evaluation</th>
<th>Makespan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native</td>
<td>40</td>
<td>58.0GiB</td>
<td>157s</td>
<td>141s</td>
</tr>
<tr>
<td>SimGrid</td>
<td>1</td>
<td>1.5GiB</td>
<td>57s</td>
<td>142s</td>
</tr>
</tbody>
</table>
Simulating Task-based Applications
- Coupling StarPU Runtime System and SimGrid Simulator
- Tackling Irregular Numerical Codes
- Use Cases
- Summary

Methodology for Reproducible Research

Conclusion
Challenges of Experimental Studies in HPC

- Large, hybrid, prototype hardware/software (hard to control)
- Costly experiments with numerous parameters
- Non-deterministic executions (overall duration, traces, . . .)
- Workflows specific to the studies (hardly applicable in general)

→ difficult to make research results reproducible
Inspired by Roger D. Peng’s lecture on reproducible research, May 2014
Our approach: use a lightweight combination of existing generic tools
Experiments

- Full control of design of experiments
- Automatize process
- Gather as much useful meta-data as possible for each experiment

Publication

Analysis

- Write papers/reports with completely reproducible analysis
- Rely on literate programming tools (IPython/Jupyter, Orgmode)
- Modular scripting approach (shell + R)

Publication

Workflow for the Whole Research Process

- Documentation and experimentation journal (laboratory notebook)
- Unique Git branching system for better project history

Publications


Achievements

Design:

- **Original approach** based on well-known tools
- Helps **filling the author/reader gap** in our context
- Applicable and extendable to other research fields

Application:

- Used this approach for many **studies, presentations and papers**
- **Efficiently handled** ~10,000 experiments (40GiB) and ~2,000 commits

Evangelism:

- Our closest colleagues **successfully adopting** this approach
- Presented our methods on numerous occasions (RR webinar, conferences, workshops, ANR project meetings, . . .)
Outline

1. Simulating Task-based Applications
   - Coupling StarPU Runtime System and SimGrid Simulator
   - Tackling Irregular Numerical Codes
   - Use Cases
   - Summary

2. Methodology for Reproducible Research

3. Conclusion
Experience

- Modeling, simulation and performance evaluation
- Methodology for reproducible research
- Statistical analysis, visualizations

- Code and performance debugging and optimizations
- Working with large, hybrid, prototype hardware and software
- Contributions to many large code projects:

  StarPU (C)      SimGrid (C/C++)
  qr_mumps (C/Fortran)  ScalFMM (C++)
  Chameleon (C/Fortran)
Summary

**Regular algorithms**
- Dynamic task-based HPC applications
- Research methodology

**Real-life applications**
- Collaboration with other domain experts

**Benchmarks**
- Basic modeling

**Numerical (irregular) libraries**
- Performance optimization
- Large scale executions
Summary

Regular algorithms
- Dynamic task-based HPC applications
- Research methodology

Real-life applications
- Collaboration with other domain experts

Benchmarks
- Basic modeling

Numerical (irregular) libraries
- Performance optimization
- Large scale executions

Thank you!

http://mescal.imag.fr/membres/luka.stanisic/
Ongoing Research: Multiple Nodes

- StarPU-MPI + SimGrid for large scale distributed memory studies
- Requires combining two modules of SimGrid framework
  ⟷ technically challenging, need to rewrite internals
- Large number of resources, kernels and communications in parallel
  ⟷ need to optimize simulator performance
- Multiple network models (PCI bus and Ethernet/Infiniband)
  ⟷ contention harder to model