Programming heterogeneous, manycore machines: a runtime system's perspective

Raymond Namyst
University of Bordeaux

Journées scientifiques SEPIA
November 12th, 2015
Multicore is a solid trend

- Multicore chips are commonplace
  - More performance = more cores
  - Complex memory hierarchies
  - Clusters can no longer be considered as “flat sets of processors”

- Designing scalable multicore architectures
  - Non-coherent cache architectures?
    • Intel SCC
    • IBM Cell/BE
  - Different memory technologies
    • Intel KNL
      High throughput (smaller) memory for stream processing
What Programming Models for such machines?

Widely used, standard programming models

- **MPI**
  - Communication Interface
  - Scalable implementations exist already
    - Was actually designed with scalability in mind

- **OpenMP**
  - Directive-based, incremental parallelization
  - Well suited to symmetric machines

- Several efforts to make these models multicore-ready
  - **MPI**
    - NUMA-aware buffer management
    - Efficient collective operations
    - Memory footprint
  - **OpenMP**
    - NUMA-awareness, memory affinity
    - Tasks
Mixing OpenMP with MPI

Makes sense even on shared-memory machines

- MPI address spaces must fit the underlying topology
- Experimental platforms for tuning hybrid applications
  - Topology-aware process allocation
  - Customizable core/process ratio
  - # of OpenMP tasks independent from # of cores
- MPC software (CEA/DAM)
  - “Threadization” of MPI +OpenMP programs

Impact of Thread distribution

Impact of thread/process ratio
Then came heterogeneous computing

And it seems to be a solid trend...

- **Accelerators**
  - Nvidia & AMD GPUs
    - De facto adoption
    - Concrete success stories
      Speedups > 50
  - Intel Xeon Phi
  - ARM Mali

- **Accelerators get more integrated**
  - Intel Ivy Bridge, Haswell
  - AMD APUs

- **They all raise the same issues**
  - Cost of moving data
    - Even for so-called APUs
  - Programming model
The Quest for Programming Models

How shall we program hybrid machines?

- Use well-known libraries when available
  - BLAS routines, FFT kernels, stencils

- Use directive-based languages
  - OpenACC, HMPP, OpenMP 4.0
  - They can save you from:
    - Learning which type of memory coalescing is supported by a given card
    - Thinking about setting some arrays dimensions to 17 instead of 16
    - Searching for tradeoffs between cluster occupancy and register usage

- Otherwise, fall back to CUDA, OpenCL, …
The Quest for Programming Models

How shall we program heterogeneous clusters?

• The hard hybrid way
  – Combine different paradigms by hand
    • MPI + {OpenMP/TBB} + {OpenCL/OpenACC}
  – Portability is hard to achieve
    • Work distribution depends on #GPU & #CPU per node…
    • Needs aggressive autotuning
  – Currently used for building distributed parallel numerical kernels
    • MAGMA, D-PLASMA, FFT kernels
The Quest for Programming Models

How shall we program heterogeneous clusters?

- The uniform way
  - Use high-level programming languages to deal with network + multicore + accelerators
  - Increasing number of directive-based languages
    - Use simple directives… and good compilers!
    - XcalableMP PGAS approach
    - OpenACC, OpenMP 4.0, OmpSs
  - Much better potential for composability…
    - If compiler is clever!
The Quest for Programming Models

Current trend: directive-based programming

- OpenMP 4.0 introduces new directives for accelerators
  - Task-based parallelism

```c
#pragma omp target device(0) map(tofrom:A)
#pragma omp parallel for
  for (i=0; i<N; i++)
    A[i] = f(A[i]);
```

- But
  - We start from a sequential code, and ask the compiler to generate efficient parallel code…
  - Data transfers and tasks’ targets are mostly explicit
    - A clever runtime system could do much better
What dynamic runtime systems can do for you

Toward “portability of performance”

• Do dynamically what can’t be done statically
  – Load balance
  – React to hardware feedback
  – Autotuning, self-organization

• Extract knowledge from upper layers
  – Structure of parallelism
  – Let the runtime system take control!
Illustration of a widespread approach: StarPU

A runtime system for heterogeneous architectures

- Rational
  - Dynamically schedule tasks on all processing units
    - See a pool of heterogeneous processing units
  - Avoid unnecessary data transfers between accelerators
    - Software VSM for heterogeneous machines

\[ A = A + B \]
Overview of StarPU

Maximizing PU occupancy, minimizing data transfers

• Accept tasks that may have multiple implementations
  – Potential inter-dependencies
  • Leads to a directed acyclic graph of tasks
  • Data-flow approach

• Open, general purpose scheduling platform
  – Scheduling policies = plugins
Dealing with heterogeneous architectures

- Task completion time estimation
  - History-based
  - User-defined cost function
  - Parametric cost model

- Can be used to improve scheduling
  - E.g. Heterogeneous Earliest Finish Time

Performance prediction
Dealing with heterogeneous architectures

- Data transfer time estimation
  - Sampling based on off-line calibration
- Can be used to
  - Better estimate overall execution time
  - Minimize data movements
Mixing PLASMA and MAGMA with StarPU
With University of Tennessee & INRIA HiePACS

- Cholesky decomposition
  - 5 CPUs (Nehalem) + 3 GPUs (FX5800)
  - Efficiency > 100%
Mixing PLASMA and MAGMA with StarPU
With University of Tennessee & INRIA HiePACS

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

![Graph showing performance comparison between different configurations of CPUs and GPUs.]

MAGMA

+12 CPUs
~200 GFlops

(although 12 CPUs alone
~150 Gflops)
Mixing PLASMA and MAGMA with StarPU

« Super-Linear » efficiency in QR?

• Kernel efficiency
  - sgeqrt
    • CPU: 9 Gflops
    • GPU: 30 Gflops Ratio: x3
  - stsqrt
    • CPU: 12 Gflops
    • GPU: 37 Gflops Ratio: x3
  - somqr
    • CPU: 8.5 Gflops
    • GPU: 227 Gflops Ratio: x27
  - Sssmqr
    • CPU: 10 Gflops
    • GPU: 285 Gflops Ratio: x28

• Task distribution observed on StarPU
  - sgeqrt: 20% of tasks on GPUs
  - Sssmqr: 92.5% of tasks on GPUs

• Heterogeneous architectures are cool! 😊
Scheduling for Power Consumption

- Power consumption information can be retrieved from OpenCL devices
  - `clGetEventProfilingInfo`
    - `CL_PROFILING_POWER_CONSUMED`
  - Implemented by Movidius multicore accelerator

- Autotuning of power models similar to performance

- Scheduling heuristic inspired by MCT
  - Actually, a subtle mix with MCT is possible

![Diagram showing power consumption for different CPUs and GPUs]
Theoretical bound

Can we perform a lot better?

- Express set of tasks (and dependencies) as Linear Problem
  - With heterogeneous task durations, and heterogeneous resources

\[
\begin{align*}
\text{minimize} & \quad t_{\text{max}} \\
\forall w \in W, \sum_{t \in T} n_{t,w} t_{t,w} & \leq t_{\text{max}} \\
\forall t \in T, \sum_{w \in W} n_{t,w} & = n_t.
\end{align*}
\]
Theoretical bound

Can we perform a lot better?

- Express set of tasks (and dependencies) as Linear Problem
  - With heterogeneous task durations, and heterogeneous resources
So, is there room for improvement?

• The quest for “good” granularity
  – Adaptive granularity
    • Divisible tasks
    • Autotuning

• Automatic selection of code variants
  – Database of various algorithms
  – Different algorithms for different problem sizes

• All the magic done by the scheduler (scheduling, prefetching, flushing) makes it harder to understand (bad) performance
  – Where does this disappointing behavior come from?

• Need for tighter compiler/runtime system integration!
Integrating runtime systems in languages

 Directive-based languages on top of dynamic runtime systems
  - OpenMP 4
    - Via KaStar Inria initiative
  - OpenACC

 Idea
  - Use a compiler to generate accelerator code
  - Use runtime system to achieve dynamic load balancing

// Get rid of such explicit data transfers!
//
#pragma acc region copy(A[0:N-1][0:M-1])
{
  for (int i=0; i<N;++i) { ... }
}
Evolution of Parallel Hardware

Exascale Parallel Machines

- SuperComputers expected to reach Exascale ($10^{18}$ flop/s) by 2020

- From the programmer point of view, the biggest change will probably come from node architecture
  - High number of cores
  - Powerful SIMD units
  - Hybrid systems will be commonplace

- Extreme parallelism
  - Total system concurrency is estimated to reach $O(10^9)$
    - Including $O(10)$ to $O(100)$ to hide latency
  - Embarrassingly parallel hardware
    - Do we really want to assign different tasks to each individual computing unit?
How will we program these machines?

From a programming point of view

• Billions of threads will be necessary to occupy exascale machines
  - Express massive parallelism
  - SIMD/vectorization is essential

  • Toward new extreme programming models?
    • Note: OpenCL-like approaches are good at expressing extreme parallelism
      Start from pure, extreme parallelism…
      then map over a restricted set of resources
      E.g. cores, vector units, streaming processors
    • Reflects constraints of manycore architectures
      No global synchronization
      Need to feed large vectorization units/numerous HW threads
How will we program these machines?

From a runtime system’s perspective

- No global, consistent view of node’s state
  - Local algorithms
  - Hierarchical coordination/load balance
  - Spawn coarse grain tasks over subsets of cores

- Reuse and couple existing codes/algorithms
  - Multi-scale, Multi-physics applications are welcome!
    - Great opportunity to exploit multiple levels of parallelism

- It’s all about composability!
  - Probably the biggest upcoming challenge for runtime systems
    - Hybridization will mostly be indirect (linking libraries)
How will we program these machines?

From a pragmatic point of view

- Cluster nodes will contain many cores…
  - But not every algorithm/problem resolution can scale that far 😞
- Co-scheduling parallel tasks simultaneously can help

![Graph showing performance of DPOTRF on Xeon Phi 7120p](image)
Co-scheduling multiple parallel tasks

- Code composability and tasks co-scheduling raise similar issues
- Libraries are typically not aware of each other
  - Resource over-subscription
    - Each library typically allocates and binds one thread per core
    - Inter-thread cache interference
- Even if a common runtime system is used underneath
  - Resource negotiation between libraries is not a trivial task
Scheduling contexts

• Lightweight virtualization
  – Multiple parallel libraries can run simultaneously
  – Each context features its own scheduler

• Contexts may expand and shrink
  – Hypervised approach
    • Maximize overall throughput
    • Minimize completion time

• Successfully used inside linear algebra solvers
  – QR-MUMPS
Towards a common BASIS for runtime systems?

- There is currently no consensus on a common runtime system…
  - Seen from outer space, many runtime systems offer similar functionalities!
- OCR initiative, INTERTWINE EU Project
- But we could certainly agree on a common “micro-runtime” basis
Major Challenges are ahead…

We are living exciting times!

more information at
http://www.inria.fr/en/teams/storm