Applications parallèles, le défi des architectures manycoeurs
Parallel applications, the challenges of manycore architectures

Forum TER@TEC 2010 Sponsors

Ter@tec est membre du pôle de compétitivité mondial
www.teratec.eu
Agenda

• Introduction aux challenges des manycoeurs
  o François Bodin, CAPS entreprise

• Issues in parallel languages for future HPC systems
  o Jesus LABARTA, Barcelona Supercomputing Center

• Debugging large scale and hybrid parallel programs
  o David LECOMBER, ALLINEA

• Des Multi-Cores aux Clouds avec ProActive Parallel Suite
  o Denis CAROMEL, INRIA

• Les enjeux du parallélisme pour les développeurs
  o Eric VERNIE, MICROSOFT
Introduction

• Main stream applications will rely on new multicore / manycore architectures
  o It is about performance not parallelism
  o Currently, it is about heterogeneous computing on CPU & GPU

• Numerous legacy applications can benefit from GPU computing
  o Hybrid computing
  o Fully rewriting applications is not an option

• Question is: “how to migrate software to manycore?”
The Past of Parallel Computing, is not the Future of Manycore

• The Past
  o Scientific computing focused
  o Microprocessor or vector based, homogeneous architectures
  o Trained programmers willing to pay effort for performance
  o Fixed execution environments

• The Future
  o New applications (multimedia, medical, …)
  o Thousands of heterogeneous systems configurations
  o Unfriendly execution environments
  o Blurry picture on hardware
Current Heterogeneous Hardware

- GPU Memory
- Shared Memory
- CORE
- network
- GPU Memory

GPU1

GPU2

CORE

GPU1

GPU2

GPU Memory

Shared Memory

Current Heterogeneous Hardware
• Mix of programming models
  • Multiple interfaces between (part of) the architectures and the application
    o PGAS, Thread, Vector, Message, Transactions, Streaming, Futur, …
• Examples
  o PGAS + threads
  o OpenMP + MPI
  o Vector instructions and scalar instructions
  o OpenMP + HMPP
  o CUDA + MPI
  o Transactional + …
  o Vector + MPI + OpenMP + HMPP
Why Hybrid/Heterogeneous May be Unavoidable?

- Shares some goals with embedded systems that leads to the same technological issues in HPC
  - Optimizing energy consumption leads to specialized architectures
  - HPC and embedded mainly differ from the acceptable degree of hardware specialization (driven by application development cost)
- No common programming API
  - API always makes some underlying architecture assumptions
  - Fixing API makes hypothesis on the future of architectures
  - No low level programming API common to all devices
  - An API addresses a specific hardware component as a consequence we need many
- No hope in virtualization
  - Virtualization is not about performance
Fundamental Issues with Hybrid

- High risk - huge legacy codes sets
  - Wrong choices may have a very strong economical impact (dev. cost)
  - Lack of parallel algorithms
  - Training, porting methodology critical
  - Standardization effort important (e.g. OpenMP)

- Resource sharing
  - Spatial sharing of memory and computing resources

- Huge optimization space
  - Difficult to explore

- Performance of parallelism is not the sum of local parallelization
  - A global state (data location), long term performance effect (affinity)

- Language specific solutions are limited
  - Parallelism is ubiquitous

- No benchmark to drive technology development
  - Anyway most public benchmarks are outdated

- Classical compilers hitting the limitation of static analysis
  - Auto-parallelization moving to more runtime techniques

- Amdahl’s law
Manycore = Numerous Configurations - 1

- Heterogeneity brings a lot of configurations
  \[ \text{Proc.} \times \text{Nb Cores} \times \text{HWA} \times \text{Mem. Sys.} = 1000^s \text{ of configurations} \]
- Code optimization strategy may differ from one configuration to another

*Is it possible to make a single (a few) binary that will run efficiency on a large set of configurations?*
Manycore = Numerous Configurations - 2

- You don’t have to program one machine but many
  - Changing runtime context, spatial resource management
  - Input data / problem size
  - Data localization
Peak Performance is Not Always the Goal

- Maximizing the Return on Investment
Why GPU Computing?

• Microprocessor frequency is not increasing anymore
  o Superscalar microarchitectures are not scaling
• Parallelism provides the performance increase
  o With a large increase in the number of cores
• Operation/Watt is the efficiency scale
  o More flops per watt
• GPUs as accelerators
  o Provides high potential computing power and memory bandwidth
  o Are ubiquitous
  o Have high level programming environments
  o Have aggressive roadmap
  o A path to prepare applications to future manycores
Applications Benefiting from GPUs

- Many applications show performance benefits when using GPUs
  - Linear Algebra
  - Signal processing
  - Bio informatics
  - Molecular dynamics
  - Magnetic resonance imaging, tomography
  - Reverse time migration
  - Electrostatic
  - …

- In some cases, the underlying algorithm/method has to be re-designed to exhibit fine grain data parallelism
A Hybrid/Heterogeneous Compute Node

- General purpose cores
  - Share a main memory
  - Core ISA provides fast SIMD instructions
  - Large cache memories

- Streaming engines (e.g. GPU)
  - Application specific architectures ("narrow band")
  - Vector/SIMD
  - Can be extremely fast

Teratec - June 2010
What is Stream Computing?

- A similar computation is performed on a collection of data (stream)
  - There is no data dependence between the computation on different stream elements
- Stream programming is well suited to GPU

```
kernel void Fct(float a<>, float b<>, out float c<>) {
    c = a + b;
}
int main(int argc, char** argv) {
    int i, j;
    float a<10, 10>, b<10, 10>, c<10, 10>;
    float input_a[10][10], input_b[10][10], input_c[10][10];
    for(i=0; i<10; i++) {
        for(j=0; j<10; j++) {
            input_a[i][j] = (float) i;
            input_b[i][j] = (float) j;
        }
    }
    streamRead(a, input_a);
    streamRead(b, input_b);
    Fct(a, b, c);
    streamWrite(c, input_c);
    ...
}
```

Brook+ example
What is a GPU Thread?

• A GPU thread is characterized by
  o A set of statements
  o A unique identifier

• All threads have the same code (SPMD)

• Two GPU threads belonging to a WARP (i.e. a set of threads) are executed in a lock step manner
How Does a GPU Work?

- GPUs achieve high performance by exploiting massive thread parallelism
  - Threads are distributed over the numerous cores
  - Thread execution is pipeline on a core to avoid waiting for memory accesses
  - Memory accesses of multiple threads are grouped (coalesced) into faster accesses by exploiting spatial locality
Application Development Issues

- Parallelism and data locality
  - Locality aware programming
- Resources management
  - Runtime libraries
- Performance portability
  - What happens with next platform generation
- Debugging and performance profiling
  - Code tuning issues
  - Ensuring correctness
- A binary has to run on many platforms
  - Forward scalability or “write once, run faster on new hardware”
  - Loosing performance is not an option
Methodology Example to Port Applications

• **Prerequisite**
  o Understand your performance goal
    • Memory bandwidth needs are a good potential performance indicator
  o Know your hotspots
    • Beware of Amdahl’s law
  o Ensure you know how to validate the output of your application
    • Rounding may differs on GPUs
  o Determine if your goal can be reached
    • How many CPUs and GPUs are necessary?
    • Is there similar existing codes for GPUs (in CUDA, OpenCL or HMPP)?

• **Define an incremental approach**
  o Check the results at each step, backtracking is very expensive

• **Ensure that performance are stable across platforms**

• **What tools and languages to support a methodology?**
  o Why don’t we have this tools yet? Heterogeneity was not anticipated
Tools

- How to address a large number of heterogeneous cores not necessarily sharing the same address space
  - Multiple “fat” computing nodes
  - APIs and tools make many hypothesis on the underlying
- Fundamental contradiction
  - Users want unifying abstraction – performance analysis requires exposing the hardware
- Parallel programming API and IDE
  - Many paradigms and implementations of the paradigms
- Parallel programming tools
  - Parallelism discovery
  - Tuning, profiling
  - Debugging
  - Libraries
- Different classes of applications, different tools?
  - Embarrassingly parallel
  - Static dependence flow
  - Dynamic data dependencies
- Scope of the tools
  - Intra nodes
  - Inter nodes
Parallelism Discovery Tools (for Legacy)

- Automatic parallelization
  - Compiler – limited by static analysis
  - Limited by the lack of accurate hardware performance model

- Speculative, adaptative, auto-tuning techniques
  - Inspector/executor techniques
  - Parasol, Nemalabs, …
Tuning and Profiling

• **Issues**
  - Vast amount of tracing/profile data to deal with
  - Non uniform view of performance (GPU Threads ≠ pThreads)

• **Example of tracing tools**
  - TAU, Vampire, Paraver, …

• **Example of profiling tools**
  - Nexus, Acumen ThreadSpotter, Vtune, VisualStudio, …
• **Issues**
  o Library are mostly designed with an egocentric view
    • FFTW, MKL, LAPACK, …
  o Composability is a difficult issue (data distribution/layout long term effect)

• **Parallel scientific math libraries**
  o NAG, PETSc, SCALAPACK, MKL

• **Data structures and thread libraries**
  o PGAS libraries, TBB, …

• **GPU Libraries**
  o CULA, Magma, Gatlas, …

• **More interaction between libraries and compilers needed**
  o JIT in libraries, autotuning libraries, …
Debugging

• Heterogeneity should not mean multiple debuggers
  o Show what necessary but hide low level details
  o Do not multiply “abstract views”

• A set of debugging levels
  o From a simple program view to a detailed view
  o Execution flow inspections: Allinea-DDT, Totalview, VisualStudio, …
  o Memory inconsistencies: Intel Thread Checkers, …

• CPU-GPU memory inconsistencies debugging
API for Adaptative Programming

- API that provides a map of available hardware as well as its characteristics
  - OpenCL has many of these tools

- API that allows to select/generate implementation in reaction to input data and available hardware

- OpenCL provides many of previous points but is low level

- The hard questions is “how to program decisions?”
Conclusion

• Existing parallel applications may need to be re-engineered
  o Adaptation of code necessary for nodes with 1000s of cores
  o Which parallel architecture?

• Progress is slowed by the lack of benchmarks
  o Porting large applications is a key activity for preparing the future
  o Requires strong collaboration between computer science and other scientific/technical domains

• Manycore is going to push application development to a new organization
  o Application programmers
  o Performance programmers
  o Standard are APIs are needed for this
CAPS
Innovative Software for Manycore Paradigms