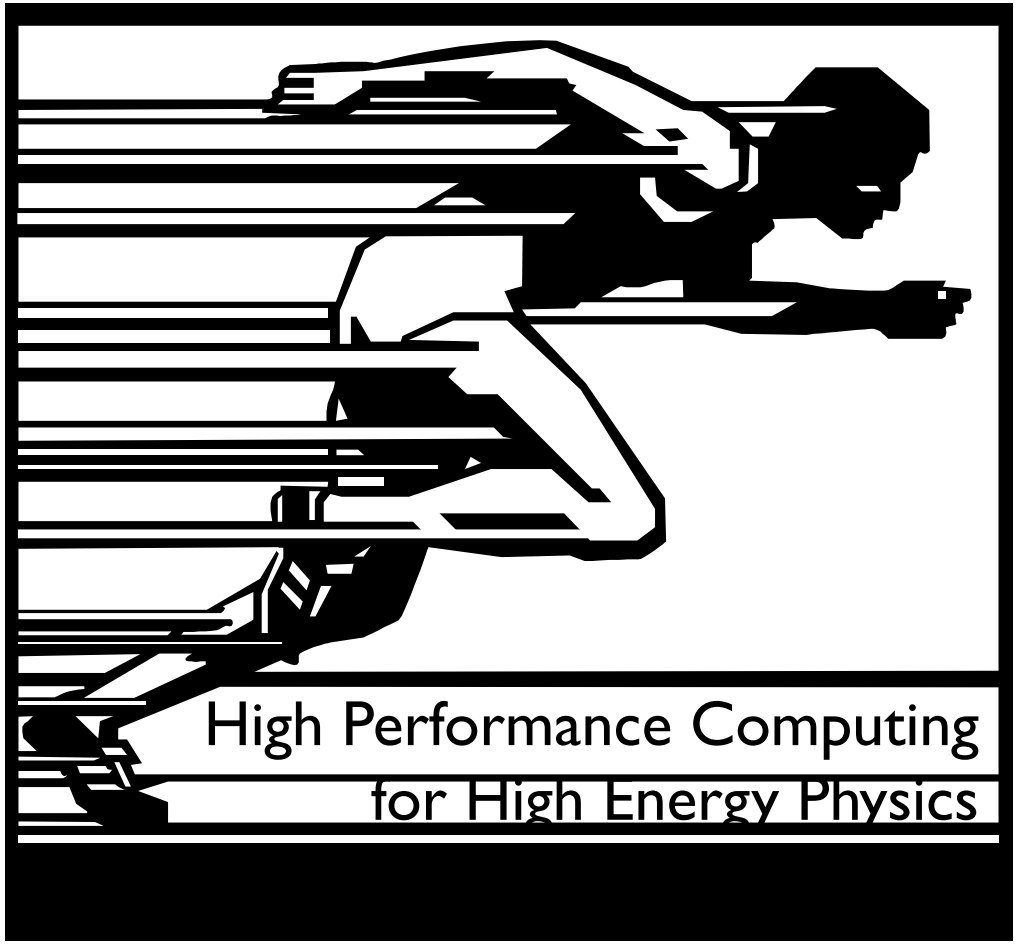
Improving Performance in Concurrency: Software Tools & Techniques



preparation for the
ECFA HL-LHC workshop
CERN, September 5th, `14

Vincenzo Innocente
CERN
CMS Experiment

Paying for Lunch

- Total cost of ownership
  - » Hardware investment
  - » Software development
  - » Operation & maintenance
- Value of Physics outcome

- Recurrent issues
  - » Memory size (and latency)
  - » Cost of Energy
  - » Burden of workflow & dataflow management (number of jobs&files)
  - » Complexity of parallel implementations
  - » Missing person-power to migrate code
  - » Slowness in validation of results
  - » Readiness of physics results (Conference Driven)

I will not cover

- Workflow and dataflow management
  - » Critical for an efficient use of parallel resources
  - » 70% efficiency of batch system often reported…
- Quality Assurance
  - » 10% of resources allocated to release validation?
  - » Rerun because of a bug == 50% efficiency
  - » Will become even more critical in presence of heterogeneous resources and concurrent applications
- Will limit to "single host" not covering cluster and cloud computing

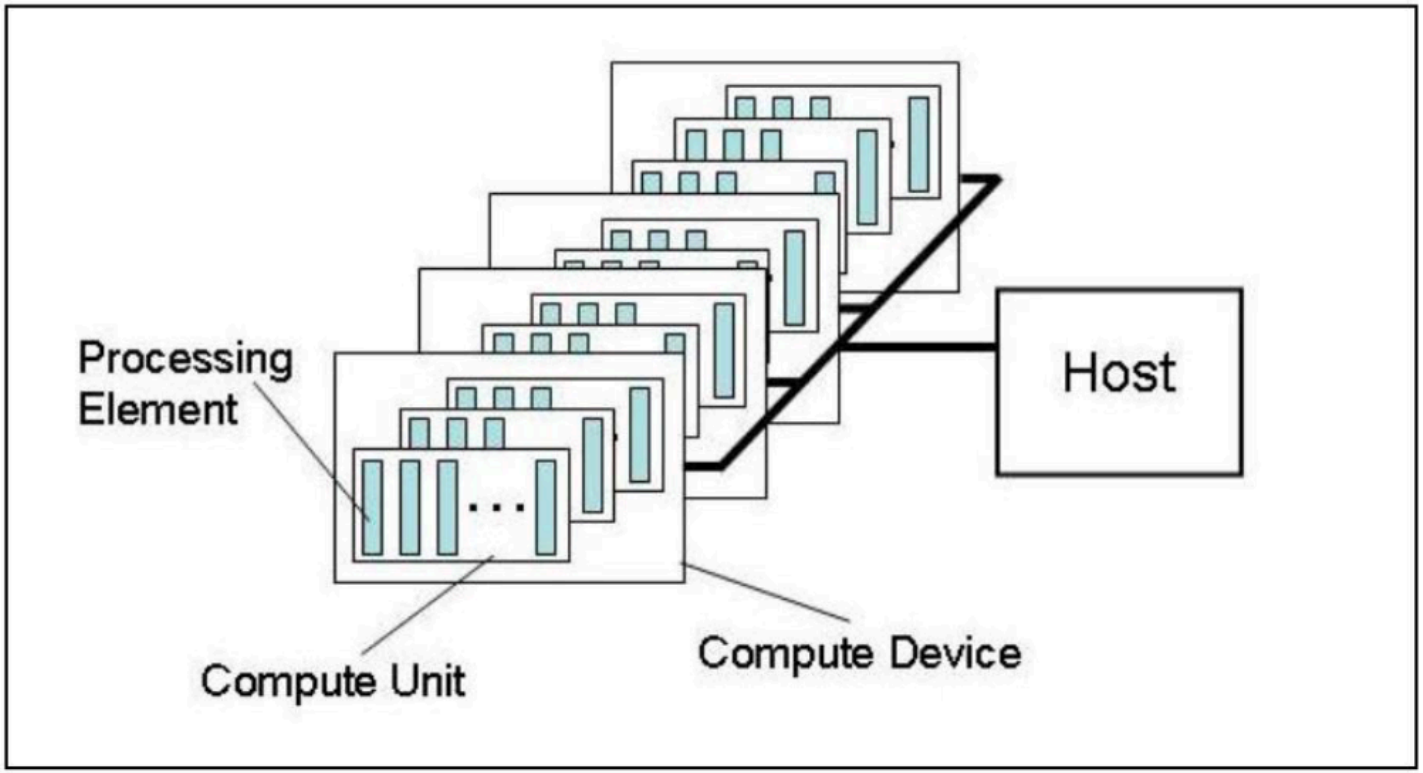
Hardware landscape

- Computers' manufactures are moving on two distinct roads
  - » Energy efficient generic computing (multi-core)
  - » High performance computational-intensive engine (many-core, SIMD)
- Packaging-wise the two will most probably coexist
  - » ARM cores + NVIDIA (or AMD) GPUs
  - » Some CPU (such as Intel-Xeon) can be operated in both modes
- The memory wall is higher than ever
  - » The cost of data access will continue to be a dominant factor
  - » Discrete accelerators and wide-SIMD will only exacerbate the issue

ISA Landscape

- i386/X86_64 dominance is over
- In the next years we will be confronted with at least 3, if not more, mayor ISAs
  - » X86_64 (with many variants in particular in the width of the SIMD)
  - » ARM64 (aarch64)
  - » NVIDIA gpu (CUDA)
  - » AMD gpu, IBM PowerPC,…?
- Non “algorithmic” code may be restricted to the first two
- Computational-intensive code will most probably be required to be fully portable among all of them

OpenCL Platform Model

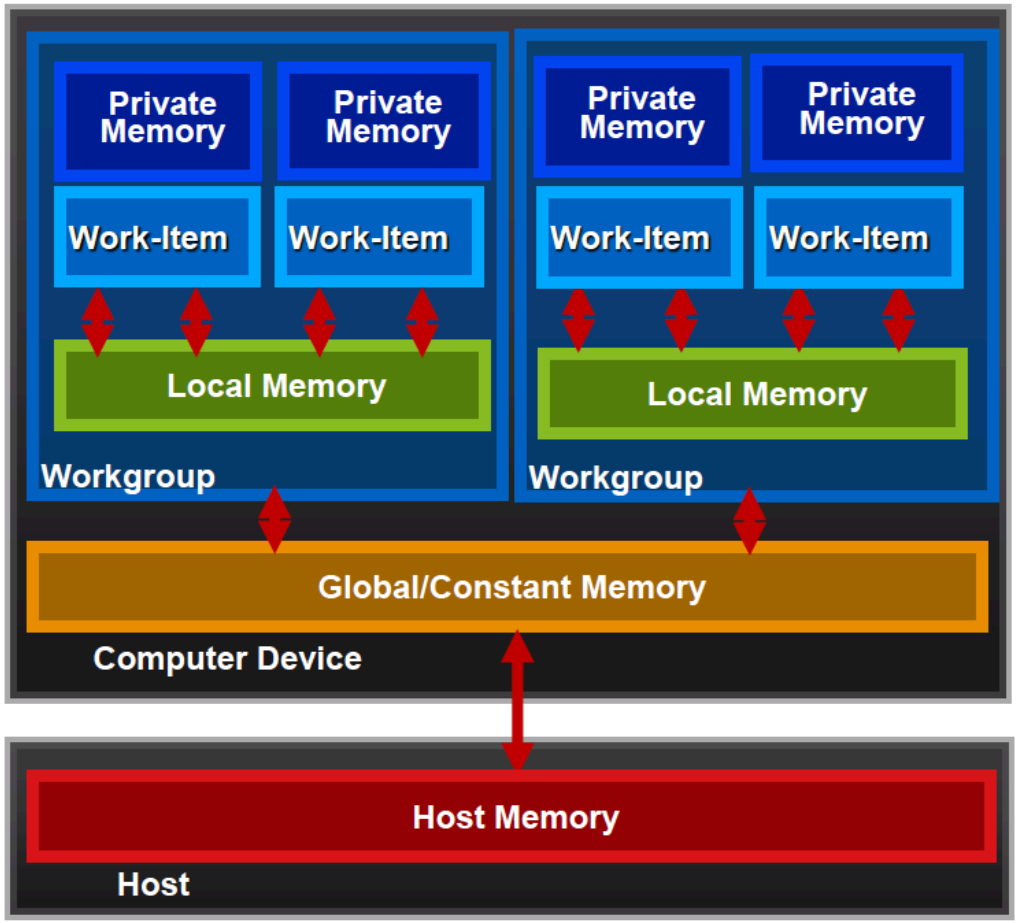


- **One Host + one or more Compute Devices**
  - Each Compute Device is composed of one or more Compute Units
    - Each Compute Unit is further divided into one or more Processing Elements

Source: SC09 OpenCL tutorial

OpenCL Memory Model

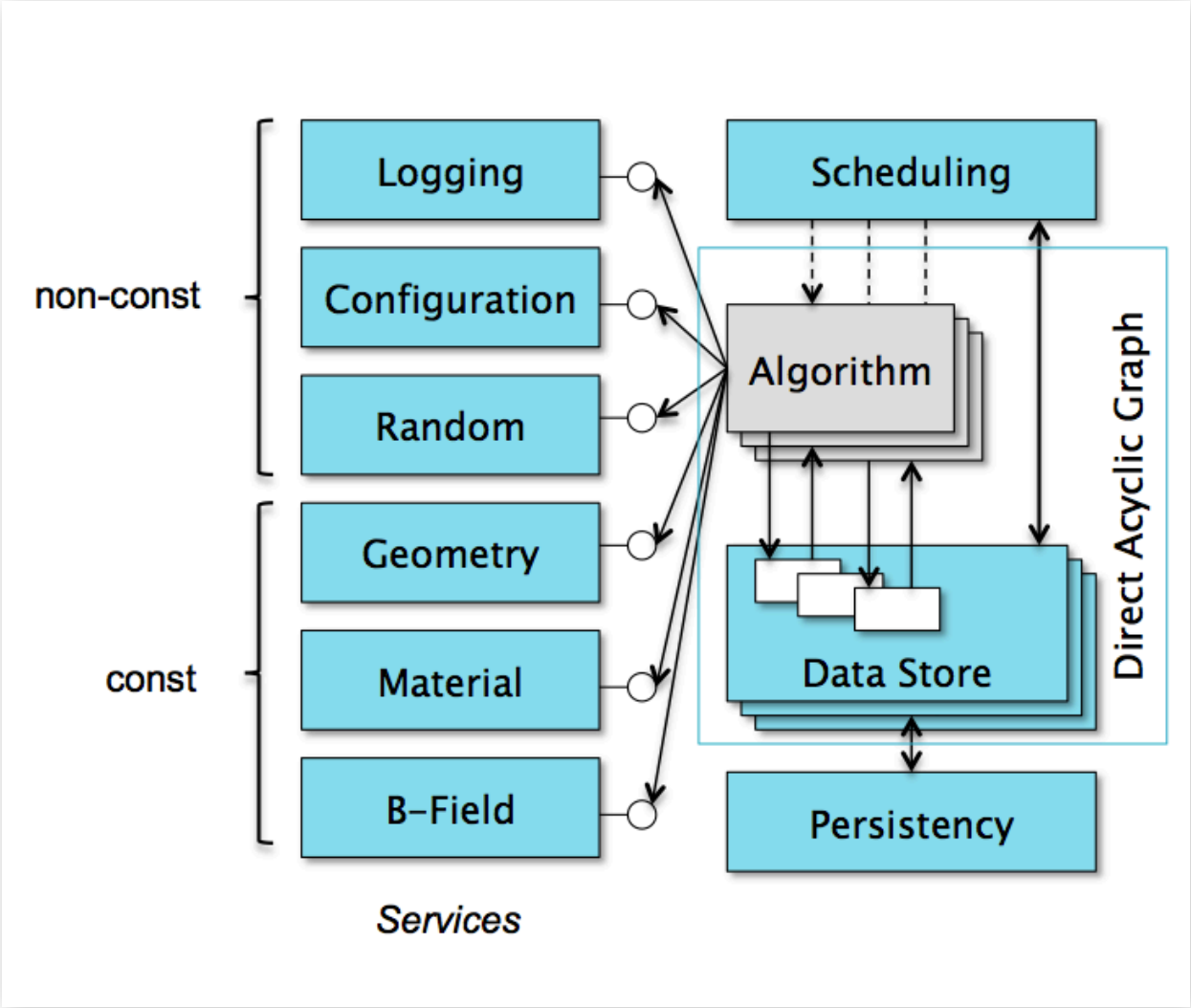
- **Private Memory**
  - Per work-item
- **Local Memory**
  - Shared within a workgroup
- **Local Global/Constant Memory**
  - Visible to all workgroups
- **Host Memory**
  - On the CPU



- **Memory management is explicit**
  You must move data from host -> global -> local *and* back

Source: SC09 OpenCL tutorial

HEP Applications

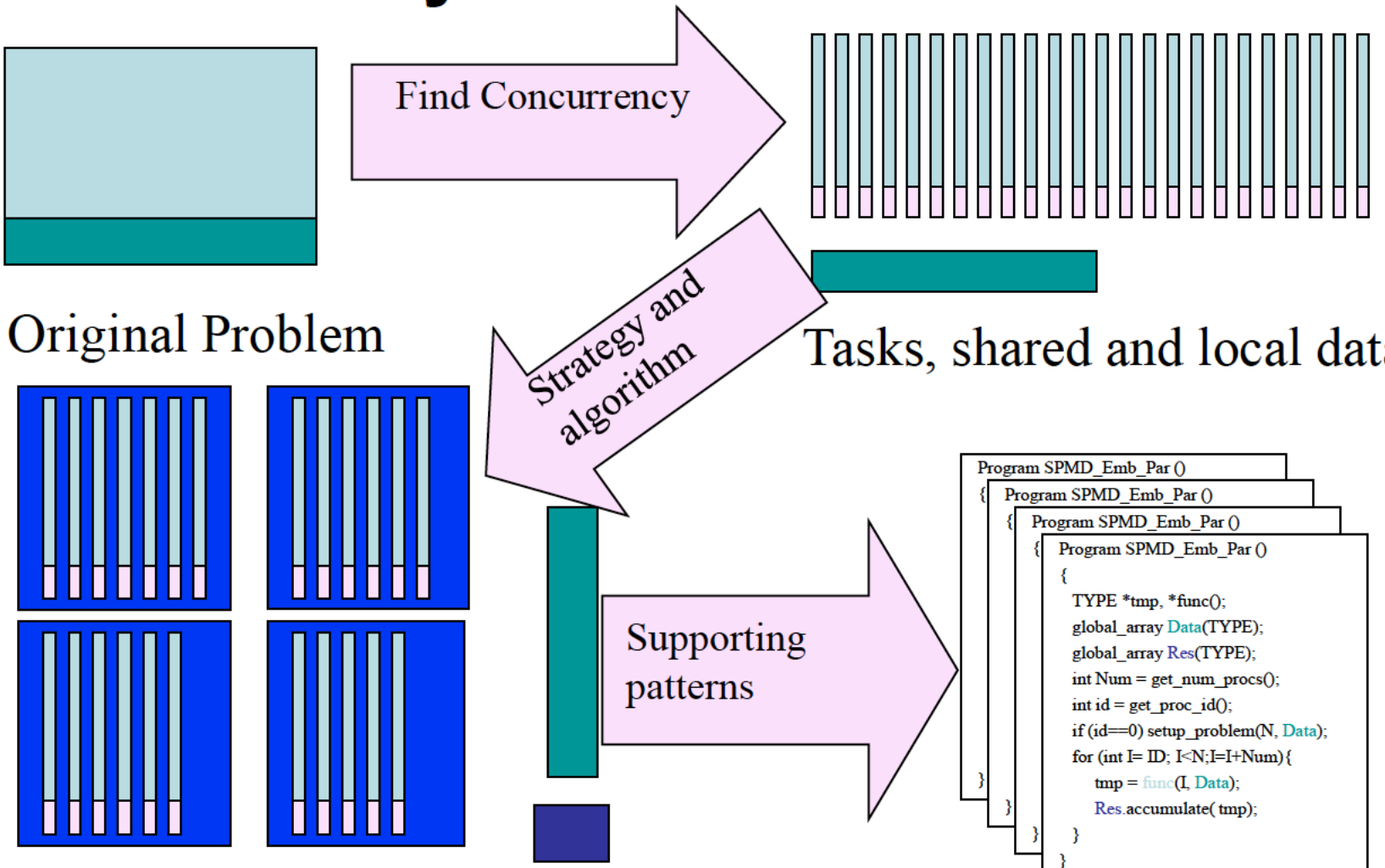


Algorithms read and write from/to the event-data store and the "services"

Only interfaces are defined (with no "cost" associated)

Algorithms are in turn based on a large set of utilities and foundation libraries

Concurrency in Parallel software:



Software Architecture

- Computational problems in HEP (simulation, reconstruction, analysis) are complex and dynamic
  - » Many different detector elements whose occupancy may vary widely with events
  - » Hundreds of algorithms and filters
- Static decomposition is improbable to be cost-effective beyond L1-Trigger
  - » (from a computational point-of-view L1-Trigger is very inefficient!)
- The application framework will be called to manage heterogeneous resources, a-priori unspecified, in a dynamic fashion
- Algorithms, utilities and foundation libraries should be able to run on any hardware and in any concurrency environment
  - » To be efficient variants/specialization may be needed
  - » Results may differ: we need to cope with that
- Efficient management of the memory hierarchy will be a key for the success

Concurrent Design Patterns

- Naïve OO Design used in the current generation of HEP applications does not fit anymore available hardware
  - » Memory unfriendly
    - Large network of objects "scattered around"
  - » Large cost from encapsulation and abstraction
    - Many tiny virtual functions (run-time resolution of what actually run!)
  - » Often intrinsically sequential, thread-unsafe, not vectorizable
    - Implicit dependencies, lazy evaluation, nested recursive branches, global states…
- Need to move to a Data Oriented Design (DoD) centered on data-collections and algorithms acting on collections
  - » Emphasis on data locality and on cache-friendliness
  - » Move abstractions few levels higher
  - » Decompose algorithms in simpler kernels acting on a clearly specified set of data

Application Frameworks

- Several HEP application frameworks have been successfully adapted to handle a concurrent scheduling even beyond the even level
  - » Able to manage resources (threads!) even for concurrent algorithms
  - » Not difficult to extend to manage heterogeneous resources
- Data Model still based on a central whiteboard as collection of collections of "directly addressable" objects (same for "services")
  - » Top level abstraction may survive
  - » Interface and implementation need to be revised to match a cache-friendly Data-Driven approach
- More R&D is required to identify the best way to support DoD at central framework/utility level

Concurrent Framework Landscape

- Native (pthreads, atomics, simd-intrinsics, language extensions)
  - » Useful to build next layer, not for end-users
- Pragma based (openMP, openACC)
  - » Easy to use
  - » Language independent
  - » Supported by compilers (comes with the system)
  - » Not obvious to use in a large, component based, application
  - » (pragmas useful to provide hints to compilers beyond language syntax)
- Library based (OpenCl, TBB, Cuda, std (c++17), …)
  - » Steeper learning and deployment curve
  - » Require a minimal "user" software infrastructure
  - » Flexible, rich in features (scheduling, memory management)
  - » Match the architecture of component based HEP applications

Do not even think to mix frameworks in the same application

Heterogeneous Concurrent Algorithms

- The only promising framework supporting heterogeneous concurrent algorithms is OpenCl
  - » C++17 may provide in a (distant?) future an alternative
  - » Deployment based either on JIT or on fat-library technologies
- Very limited experience on actual coded algorithms able to run on heterogeneous hardware
  - » Minimum common denominator approach will not improve efficiency much
  - » Achieving the maximal efficiency requires specialization w/r/t memory hierarchy, SIMD width, scheduling
- More specific R&D is required to understand the ability of OpenCl to serve HEP use-cases
- Short/Medium term solutions will be based on a mixture of C++ (tbb,std) and CUDA
  - » multiple implementations of the same algorithm
  - » deployment based on target-specific releases with a limited use of fat-libraries

Tools to support development

- Developing correct, efficient concurrent algorithm is not easy
- We need a new set of libraries to support basic parallel algorithms and data structure to be used in a heterogeneous environment
  - » Standard will eventually come (C++17 and beyond)
  - » Need to fill the gap with shopping + in house development
- Essential to provide to developers tools to verify
  - » Correctness (Data races)
  - » Effective parallelism
  - » Memory, cpu and energy efficiency
- Use of safe and efficient design and implementation patterns is an essential starting point
  - » C++ provides already syntactic and library elements to ensure const-correctness, atomicity and proper memory management
  - » A static code analyzer can easily verify their correct usage
  - » Library implementations are welcome
- Integration with the build system is vital

Conclusions

- Free lunch is over
  - » To improve the efficiency of software we need to increase the granularity of parallelism, optimize data access patterns and make use of heterogeneous resources
- Waiting for the definitive standard to emerge we need to develop our own infrastructure to support the implementation of concurrent algorithms able to exploit parallelism on heterogeneous hardware
- Recent work shows that
  - » An efficient concurrent schedule of algorithms is feasible
  - » With huge effort it is possible to make current algorithm implementations free from data-race (thread safe)
  - » Making use of parallelism in algorithms requires a total re-implementation
- More R&D is required to tackle the challenges of
  - » Exploiting heterogeneity
  - » Efficient parallelize algorithms
  - » Efficient utilization of memory hierarchy
  - » Efficient utilization of the few developers left