Challenges for Compiler Support for Exascale Computing

Programming Languages and Compiler Workshop

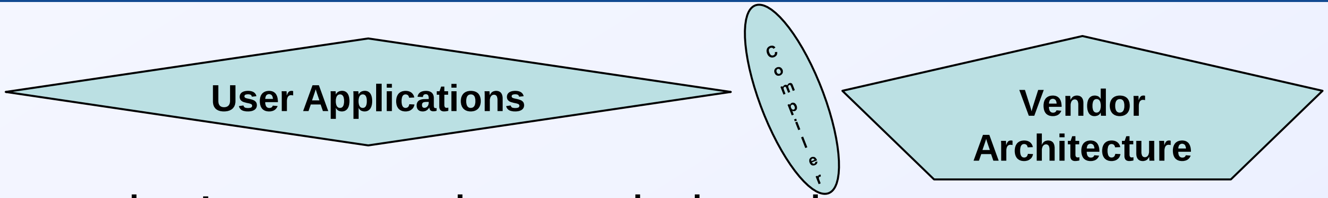
Concentrate on the challenges advantages and disadvantages of the various approaches

July 2011

Daniel Quinlan

Lawrence Livermore National Laboratory, P. O. Box 808, Livermore, CA 94551
Operated by Lawrence Livermore National Security, LLC, or the U.S. Department of Energy,
National Nuclear Security Administration under Contract DE-AC52-07NA27344

The Exascale compiler is between a rock and a hard place (economics)



- Users don't want to change their code
- The architecture is unknown, but it will be different, maybe very different, e.g. scratch space instead of cache.
- Users write their code using general purpose (GP) languages
  - They are the only languages the vendors support…
  - They are the languages that new talent knows…will learn…
  - And its where the tools are…(leverage)
- Parallelization History:
  - Vectorization hardware demanded local analysis, so the transition was relatively smooth once the compilers caught up.
  - Distributed memory parallelism is a global optimization, so out of bounds using program analysis on GP languages

How do we get out of this mess… (preview)

- Make the software *more* globally analyzable
  - Use restrictions to avoid practices that are unsafe or unanalyzable
  - Use abstractions with well defined semantics
  - *Define runtime and/or compiler support for your abstractions…*
  - Abstractions could have multiple levels of APIs (users, compiler, …)
- Also packaged as DSLs, programming models, new languages

Insert favorite compiler here

- Good news: you can do this with existing languages (source-to-source)
  - The appropriate restrictions and extensions are domain-specific research topics
  - Don't get hung up on syntax…

Modern parallel programming models have a larger design space; they are more complex and often lag behind hardware nowadays

Diverse Algorithms (App. Domains)

Conflicting Design Goals

- **Expressiveness**
- **Performance**
- **Portability**
- **Programmability**
- **…**

Complex Software Stack

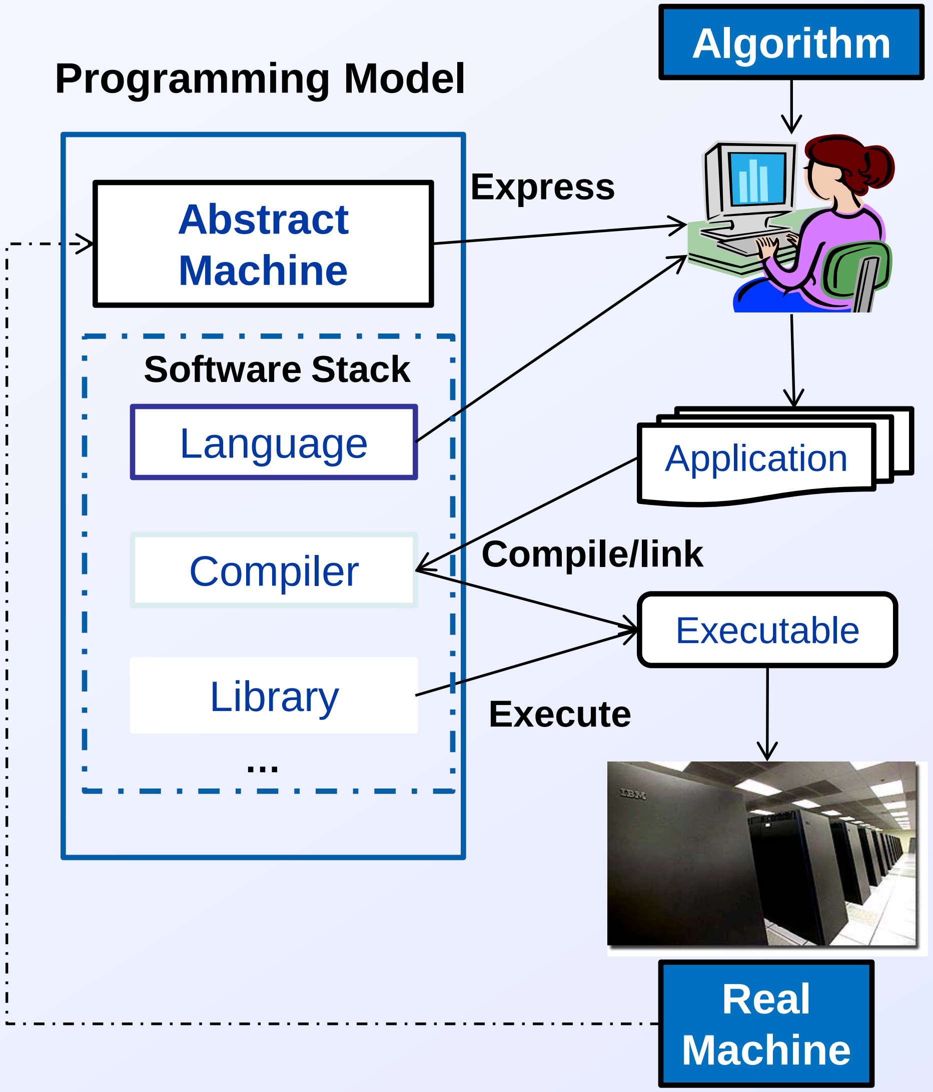
- **Language**
- **Compiler**
- **Library**
- **…**

Fast-changing Parallel Machines

Numerous
- **choices/options/tradeoff**
- **combinations**
- **interactions**

Today's parallel programming models are already behind today's machines. (e.g. multithreading CPU+GPU)

Programming models bridge algorithms and machines and are implemented through components of software stack

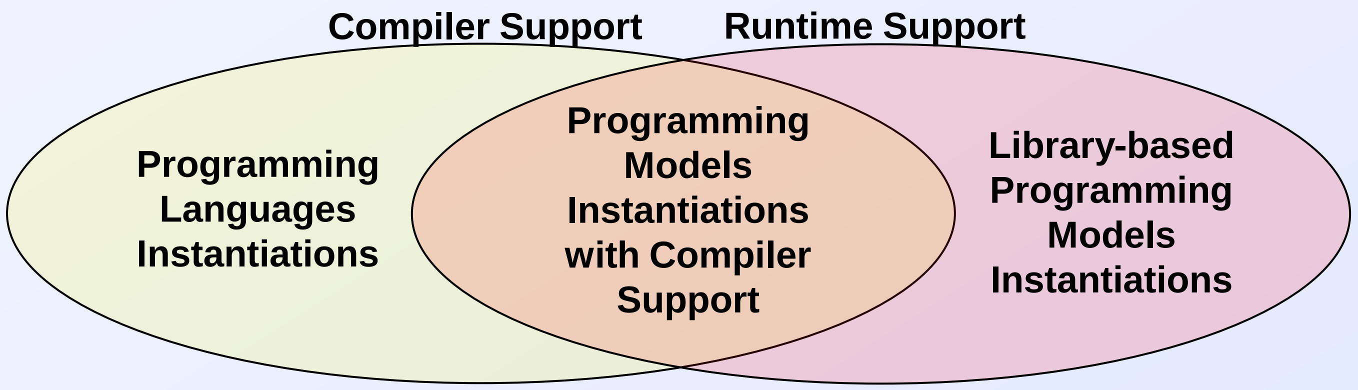


Measures of success:

- Expressiveness
- Performance
- Programmability
- Portability
- Efficiency
- …

Challenges for Compilers and Programming Languages

- Programming Models often have compiler requirements
  - Programming model instantiations are supported using a range from libraries (MPI) to compilers (OpenMP)
  - Always a runtime level of support
  - Often includes compiler support
- Programming Languages require compiler support
- If you give a mouse a cookie… make the HPC community build a programming model…


Compiler Support
Runtime Support
Programming Languages Instantiations
Programming Models Instantiations with Compiler Support
Library-based Programming Models Instantiations

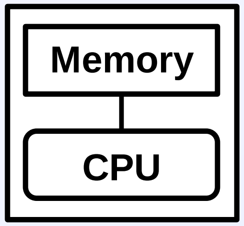
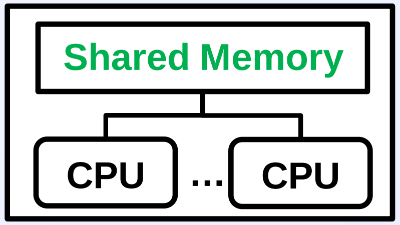
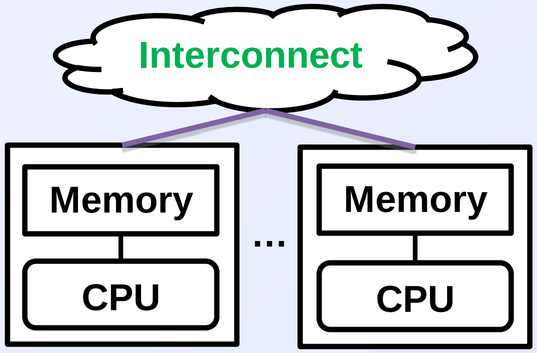
Exascale will make demands on compiler technology

- Unique one off solutions for specific hardware
- Unique one off solutions for Exascale…
- Demanding schedules will drive manual solutions first
  - Compiler technology can only backfill with automated solutions where possible
  - Automated and semi-automated techniques will lag
- Economics will drive different solutions at different levels

- But the codes will be the same…until users have to optimize the performance

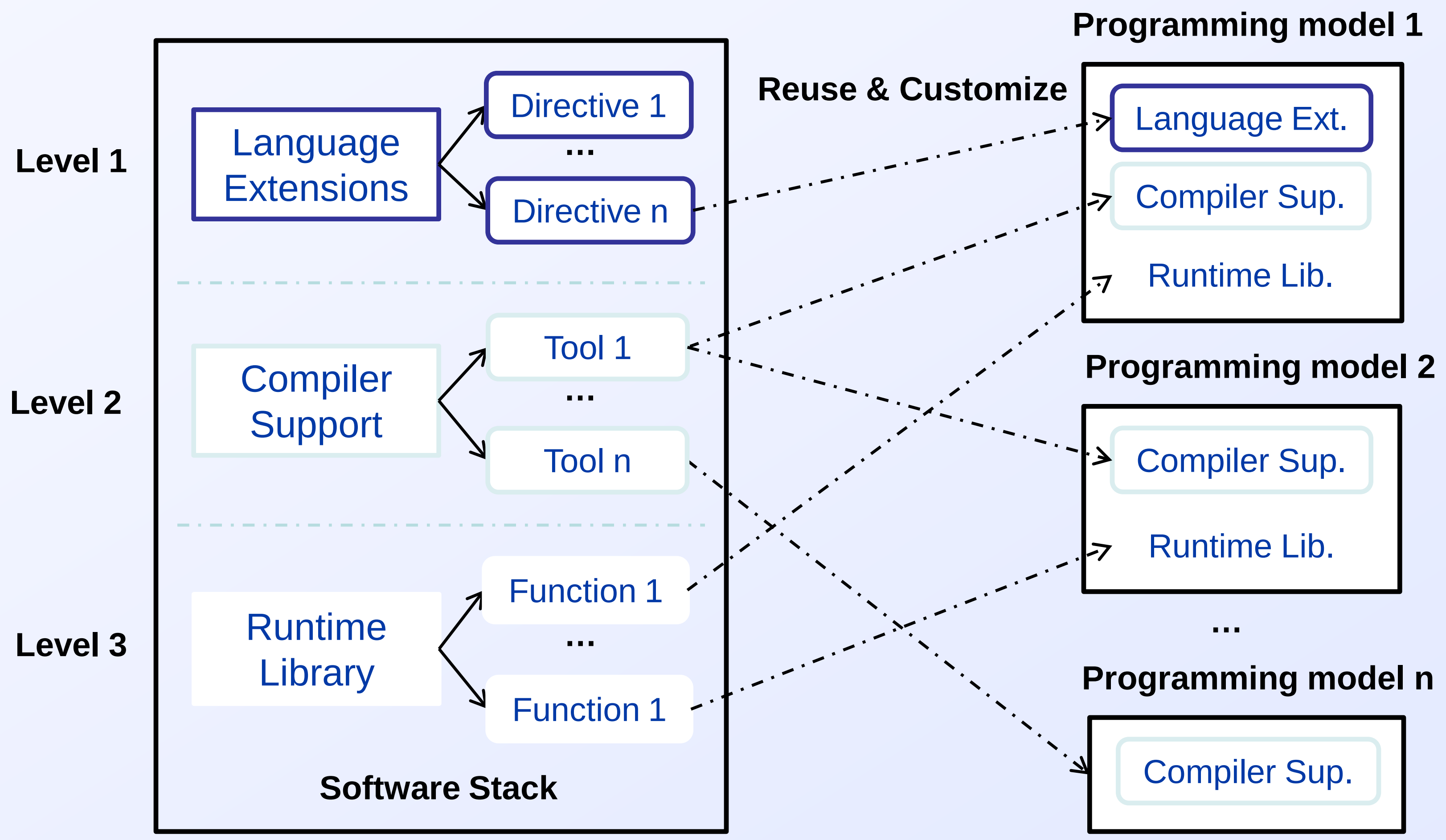
- Resiliency as an example of Exascale specific compiler work

Parallel programming models are built on top of sequential ones and use a combination of language/compiler/library support

Programming Model	Sequential	Parallel	
		Shared Memory (e.g. OpenMP)	Distributed Memory (e.g. MPI)
Abstract Machine (overly simplified)			
Software Stack	General purpose Languages (GPL) C/C++/Fortran	GPL + Directives	GPL + Call to MPI libs
	Sequential Compiler	Seq. Compiler + OpenMP support	Seq. Compiler
	Optional Seq. Libs	OpenMP Runtime Lib	MPI library

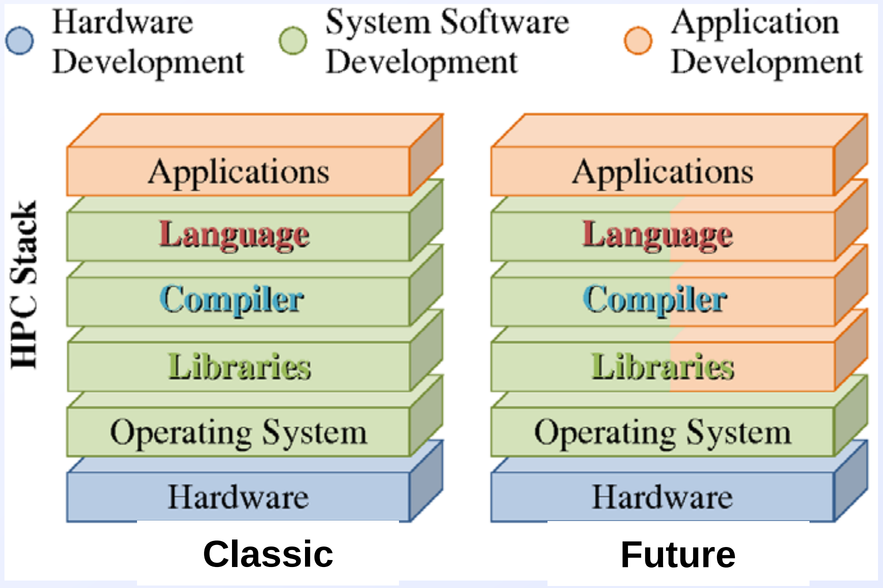
We could define a programming model framework to address exascale challenges and beyond

A three-level, open framework to facilitate building node-level programming models for exascale architectures



Serve both researchers and developers, engage HPC applications, and targets heterogeneous architectures

- Users:
  - Programming model researchers: explore design space
  - Experienced application developers: build custom models targeting current and future machines
- Scope is a research topic



The programming model framework vastly increases the flexibility in how the HPC stack can be used for application development.

- HPC applications: scientific computing
- Heterogeneous architectures: CPUs + GPUs
- Building blocks: parallelism, locality, power efficiency, resilience

It is a challenging research & development problem to provide building blocks in order to address exascale challenges

Building blocks: essential, widely applicable, reusable, customizable
Framework: easy combination of building blocks to explore the design space

	Parallelism	Data Locality	Power Efficiency	Resilience

Language Extension	**#task**
	#device
	#depend_on
		#distribution
		#location
		#mem_pattern
			#turn_off(FPU)
			#cpu_freq()
			#cache(n-way)
				#check_sum
				#TMR
				#checkpoint

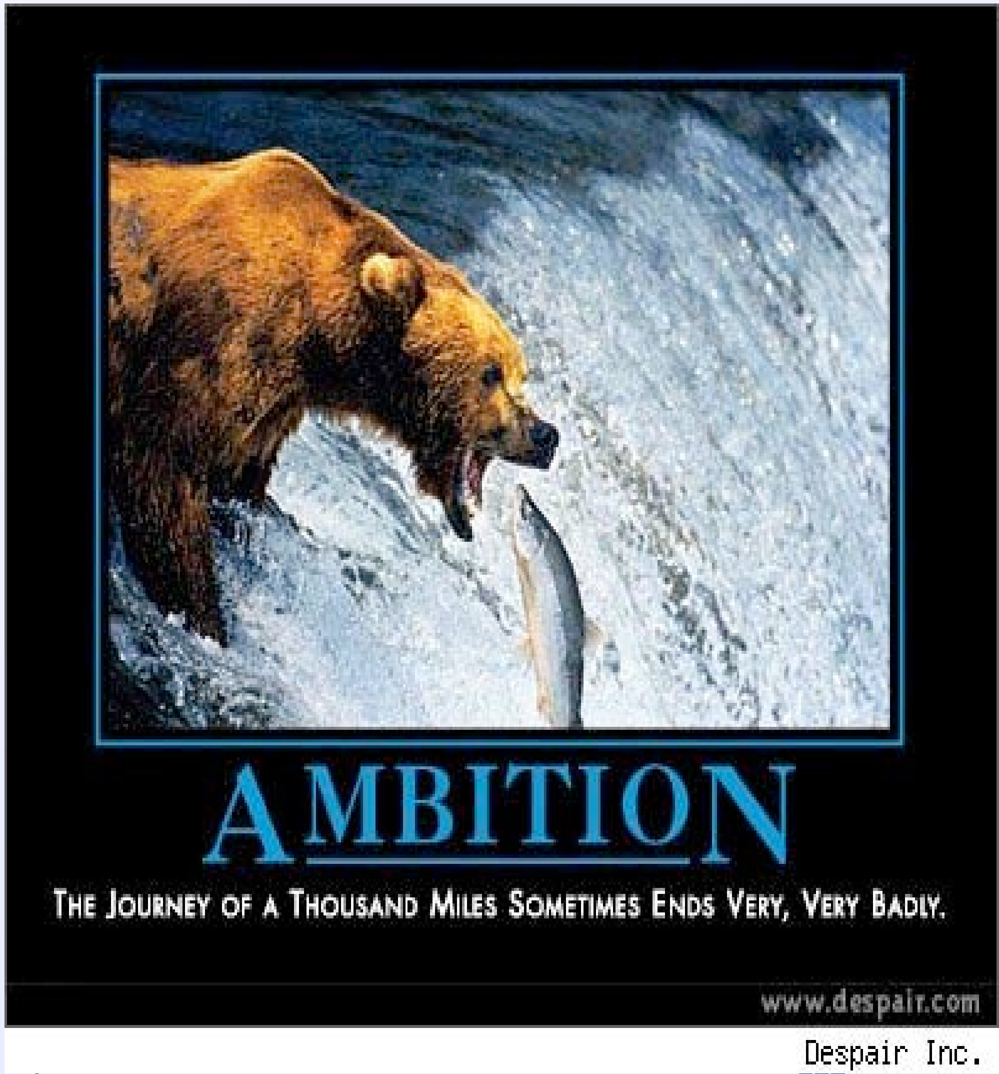
Compiler Support	**outliner** **instrumentor** **depAnalyzer**	**dataPartitioning** *reuseDistance* *arrayAccessPattern*	**resourceAnalysis** **loopTranslation** *worstCaseExe*	**faultDetection** *faultInjection* *InCacheTMR*

Runtime Library	**threadCreate();**
	barrier();
	taskSchedule();
		set_affinity();
		set_mempolicy();
		data_redist();
			power_off();
			get_enegy_metric();
			set_mem_freq();
				check_sum();
				generate_fault();
				checkpoint();

Notes:
- Building blocks in a bold font: planned R&D in this proposal
- Others in an italic font: long term research goals
- #task is used instead of #pragma task for brevity in the table
- #TMR: Triple Modular Redundancy

Summary: Building Blocks Approach

- Leverage the existing languages
- Build Programming model building blocks
  - Compiler support
  - Runtime support
- Enable research to instantiate specific programming models
- Target evolving architectures quickly…
- Challenges:
  - Selection of abstractions
  - Description of abstractions semantics
  - Generating transformations using abstraction semantics


AMBITION
THE JOURNEY OF A THOUSAND MILES SOMETIMES ENDS VERY, VERY BADLY.
www.despair.com
Despair Inc.

Compiler technology has to be easy to use…

END

What makes the compiler and runtime support useful?

- Accessibility of compiler support
  - Is the compiler support required available?
  - Can there be a community to support this?
- Maturation
  - It takes many years for compiler support to mature
  - How can such work be tested and maintained
- Adaptability
  - How can it be extended to suit the needs of HPC (for Exascale and beyond)

Resiliency via Compiler Transformations (soft errors only)

- Processor checking:
  - Introduction of Triple Modular Redundancy (TMR)
  - Different granularities of synchronization

- Data Integrity
  - Communication via noisy channel
  - Redundancy of data is unreasonable

Exascale will make demands on compiler technology

- Accessible (open source availability)
- Easy to use (documented)
- Robust (must handle full scale DOE applications)

- Maybe this is asking too much…

We propose to build a framework for creating node-level parallel programming models for exascale

- Problem:
  - Parallel programming models: important but increasingly lag behind node-level architectures
  - Exascale machines more challenges to programming models
- Goal:
  - Speedup designing/evolving/adopting programming models for exascale
- Approach:
  - Identify and implement common building blocks of node-level programming models so both researchers and developers can quickly construct or customize their own models
- Deliverables:
  - A **programming model framework** (PMF) with building blocks at language, compiler, and library levels
  - Example node-level programming models built using the PMF

Programming models will mostly likely become a limiting factor for exascale computing if no drastic measures are taken

- Future exascale architectures
  - Clusters of many-core nodes
  - Abundant threads, deep memory hierarchy, CPU+GPU, …
  - Power and resilience constraints, …
- (Node level) programming models
  - Increasingly complex design space
- Current situation:
  - Programming model researchers: struggle to design/build *individual models* to find the right one in the huge design space
  - Application developers: stuck with *stale models*: insufficient high-level models and tedious low-level ones
- Exascale computing may be well behind schedule because of lengthy design and adoption of exascale programming models!

The 1st level of the framework provides building blocks for directive-based language extensions of programming models

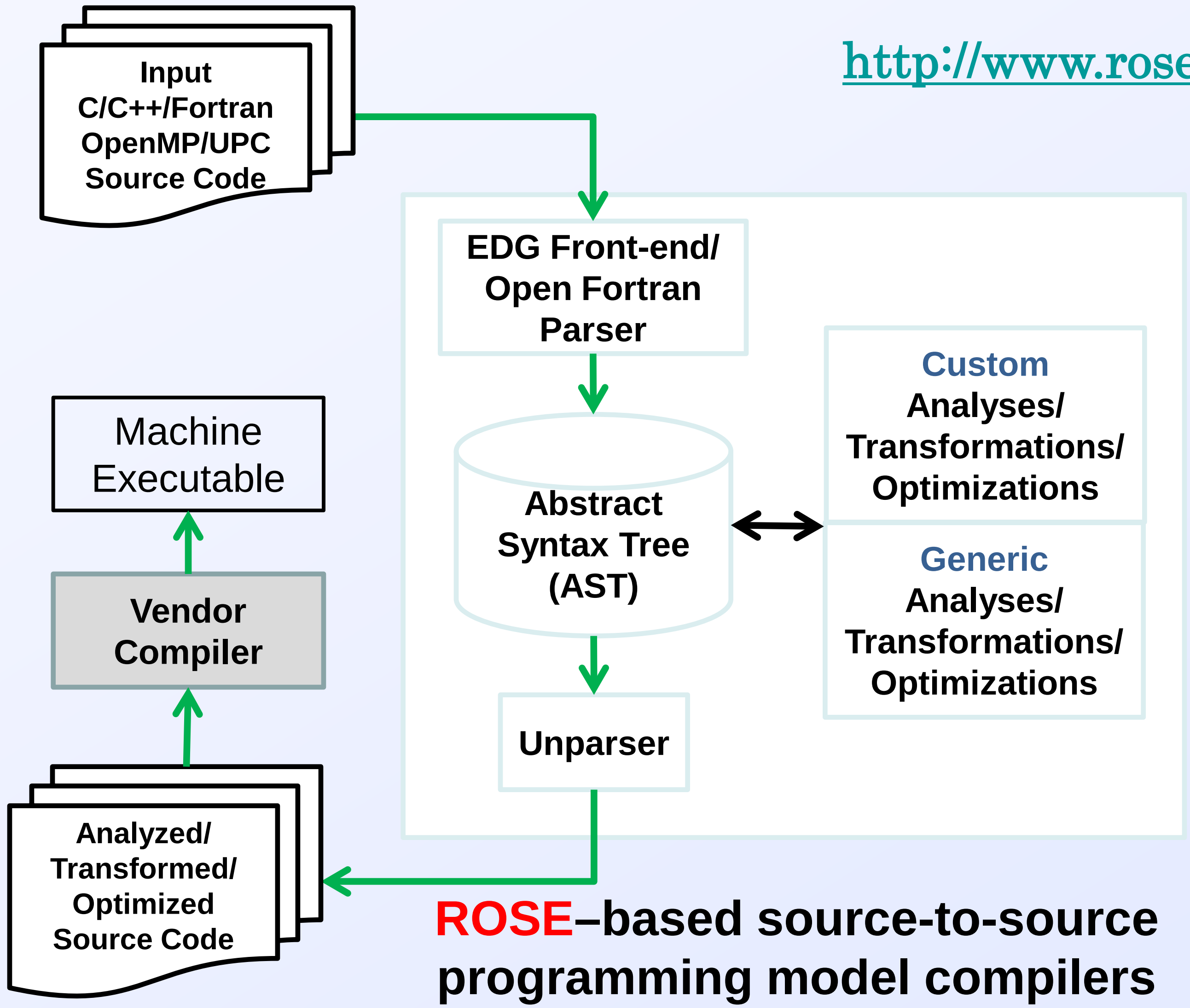
- Language level building blocks:
  - Compiler directives that express additional semantics to address exascale challenges (parallelism, locality, power, resilience,…)
- Compiler directives: source code annotations that provide additional information to compilers
  - C/C++: #pragma omp parallel ; Fortran: !$omp parallel
  - #pragma task, #pragma device(CPU|GPU)
- Research and development issues:
  - Unify existing directives
  - New directives (what to express, at what granularity, and how?)
- Benefits
  - Quick experiment with various language features
  - Minimal footprint to existing general purpose languages
    - Provide a fast avenue for migrating legacy code
    - Separate algorithms from implementation details

The 2nd level of the framework provides building blocks for compiler support of programming models

- Compiler level building blocks:
  - Composable software tools with application programming interfaces (APIs) for implementing compiler support of various programming models
  - Parsing customized directives: parse_expression()…
  - Analyses: dependence, resource usage, …
  - Transformations: instrumentation, outlining, …
  - Optimizations: loop unrolling, auto parallelization,…
- Research and development issues:
  - Identify and encapsulate existing common complier support
  - Develop new compiler analyses/optimizations for upcoming challenges

The compiler support will be implemented using the ROSE compiler infrastructure (developed at LLNL)

http://www.roseCompiler.org



Input C/C++/Fortran OpenMP/UPC Source Code
EDG Front-end/ Open Fortran Parser
Abstract Syntax Tree (AST)
Custom Analyses/ Transformations/ Optimizations
Generic Analyses/ Transformations/ Optimizations
Unparser
Machine Executable
Vendor Compiler
Analyzed/ Transformed/ Optimized Source Code

2009 Award

ROSE–based source-to-source programming model compilers

The 3rd level of the framework will provide building blocks for runtime libraries of programming models

- Runtime Library building blocks: generic interface functions that support the implementation and execution of programming models
  - Thread management, data locality
  - Power management, resilience support
  - E.g. threadCreate(), taskScheduling(), data_redist(), power_off()…
- A thin layer on top of existing runtime library functions
  - Share same compiler support with multiple libraries (GOMP, StarPU, etc)
  - Provide an actual functionality only if it is not available otherwise
- R&D issues:
  - Unify common runtime support, develop new functions

Our framework makes it simpler to evolve existing programming models (use case #1)

- E.g : evolve the OpenMP programming model
  - OpenMP: the most popular node-level model
- We will provide an OpenMP implementation using our framework
  - Building blocks of language directives, compiler, runtime library support
- Users:
  - Insert locality, energy or resilience building blocks into the OpenMP implementation
  - Experiment with combinations and interactions of building blocks from three levels

“Fault Resilience for HPC Applications on Exascale Systems” – Dan Quinlan, LLNL

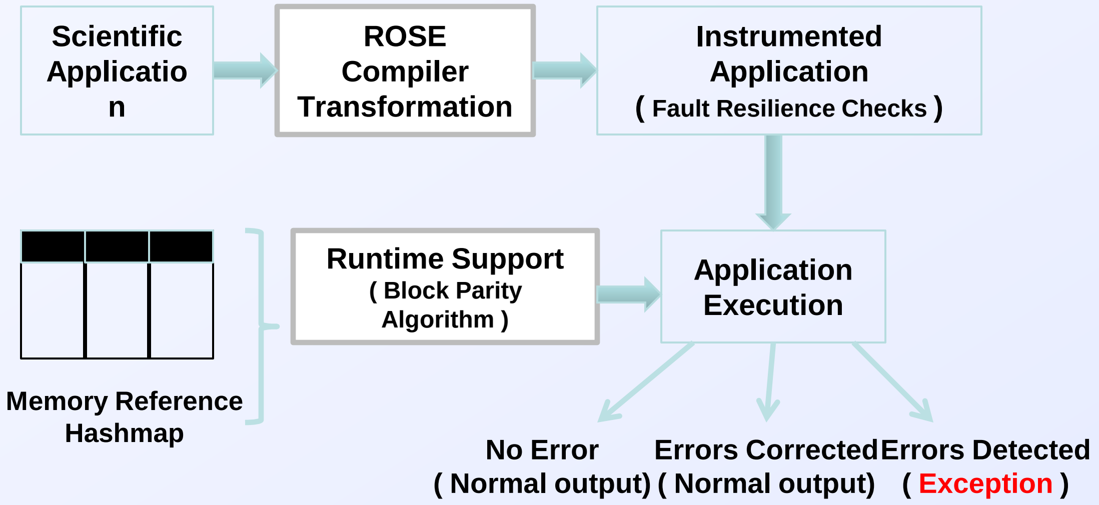
ASCR- Computer Science Highlight

Objectives

- Create an automated compiler transformation to assist programmers in DOE for integrating memory-related fault resilience in their applications :
  - Creating memory efficient fault resilience technique at compiler level
  - Automatically introduce runtime fault resilience checks with some support for error correction capability

Impact

- Automated approach to addressing the resilience challenge of exascale computing
- Assist application sustainability in ExaScale environments where memory failures may occur every 2 hours [DARPA ExaScale Study 2008 Report]


Scientific Application
ROSE Compiler Transformation
Instrumented Application (Fault Resilience Checks)
Memory Reference Hashmap
Runtime Support (Block Parity Algorithm)
Application Execution
No Error (Normal output)
Errors Corrected (Normal output)
Errors Detected (Exception)

Accomplishments 2011

- **Developed compiler transformation for instrumenting memory references in scientific kernels with fault resilience checks**
- **Designed a library to support runtime detection of memory errors**
- **Implemented a fault resilience technique with block parity algorithm**

Our framework enables fast prototyping of new programming models (use case #2)

- E.g. : a multithreading programming model for both CPUs & GPUs
  - Concurrent execution on both processors
  - Work-queue threading strategy
- Language (directives):
  - C++ with pragmas to identify tasks
  - Highly parallel algorithms (kernels) written using CUDA
- Compiler (tools):
  - Outline tasks and add them onto a queue
  - Transform CUDA kernels into code suitable for x86 machines using vector extensions
- Runtime library (functions):
  - Scheduler dispatches tasks in the queue to CPUs or GPUs

Implementing a work-queue threading strategy for CPUs & GPUs (use case #2 continued)

Language (level 1): Application source code annotated with pragmas

```
//'globalData' is divided into patches.  For each patch, perform some work.
          #pragma threadqueue task for shared(globalData)
 for (int idxPatch = 0; idxPatch < numPatch; ++idxPatch) {
               #pragma threadqueue task device(CPU) label("doMeFirst")
   {
     function_1(globalData[idxPatch]);
     function_2(globalData[idxPatch]);
   }
               #pragma threadqueue task device(GPU) depend_on("doMeFirst")
   function_3(numBlock, numThread, globalData[idxPatch]);
 }
```

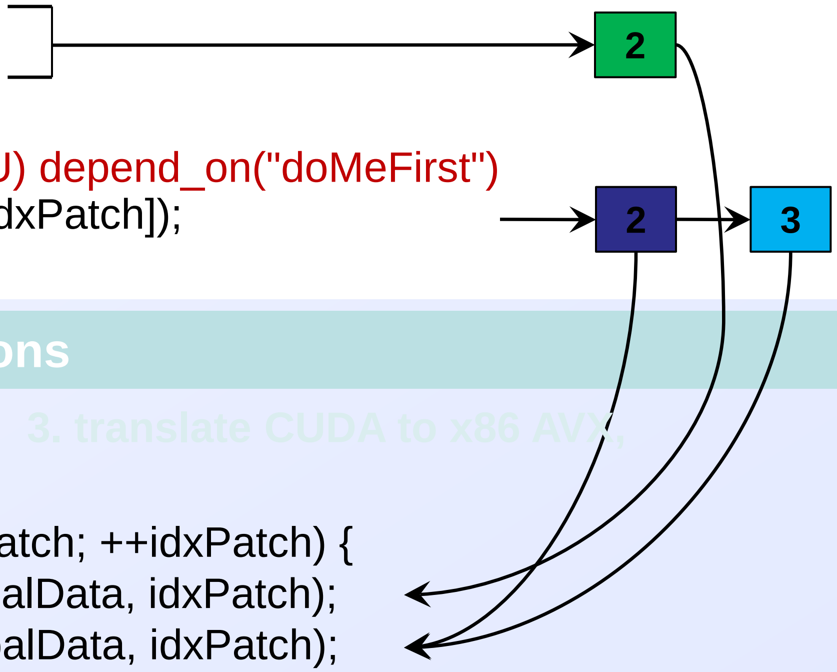
- Tell compiler a parallelizable loop and shared data
- Identify tasks for CPUs
- Identify tasks for GPUs and dependence

Implementing a work-queue threading strategy for a CPU & GPU (use case #2 continued)

Language (level 1): Application source code annotated with pragmas

//'globalData' is divided into patches. For each patch, perform some work.

```
        #pragma threadqueue task for shared(globalData)
for (int idxPatch = 0; idxPatch < numPatch; ++idxPatch) {
            #pragma threadqueue task device(CPU) label("doMeFirst")
  {
    function_1(globalData[idxPatch]);
    function_2(globalData[idxPatch]);
  }
            #pragma threadqueue task device(GPU) depend_on("doMeFirst")
  function_3(numBlock, numThread, globalData[idxPatch]);
}
```



Compiler (level 2): Compiler transformations

1. parse pragma statements, 2. outline tasks, 3. translate CUDA to x86 AVX, and 4. push onto queue.

```
for (int idxPatch = 0; idxPatch < numPatch; ++idxPatch) {
  CPUQueue.add(outlined_task1(globalData, idxPatch);
  GPUQueue.add(outlined_task2(globalData, idxPatch);
}
```

Implementing a work-queue threading strategy for a CPU & GPU (use case #2 continued)

Language (level 1): Application source code annotated with pragmas

//'globalData' is divided into patches. For each patch, perform some work.

```
        #pragma threadqueue task for shared(globalData)
 for (int idxPatch = 0; idxPatch < numPatch; ++idxPatch) {
            #pragma threadqueue task device(CPU) label("doMeFirst")
   {
     function_1(globalData[idxPatch]);
     function_2(globalData[idxPatch]);
   }
            #pragma threadqueue task device(GPU) depend_on("doMeFirst")
   function_3(numBlock, numThread, globalData[idxPatch]);
```

Runtime (level 3): Runtime library

//A worker thread on the CPU, if idle, obtains a new task from the thread scheduler

```
while (outlined_task *myTask = CPUQueue.get()) {
 myTask->exec();
};
```

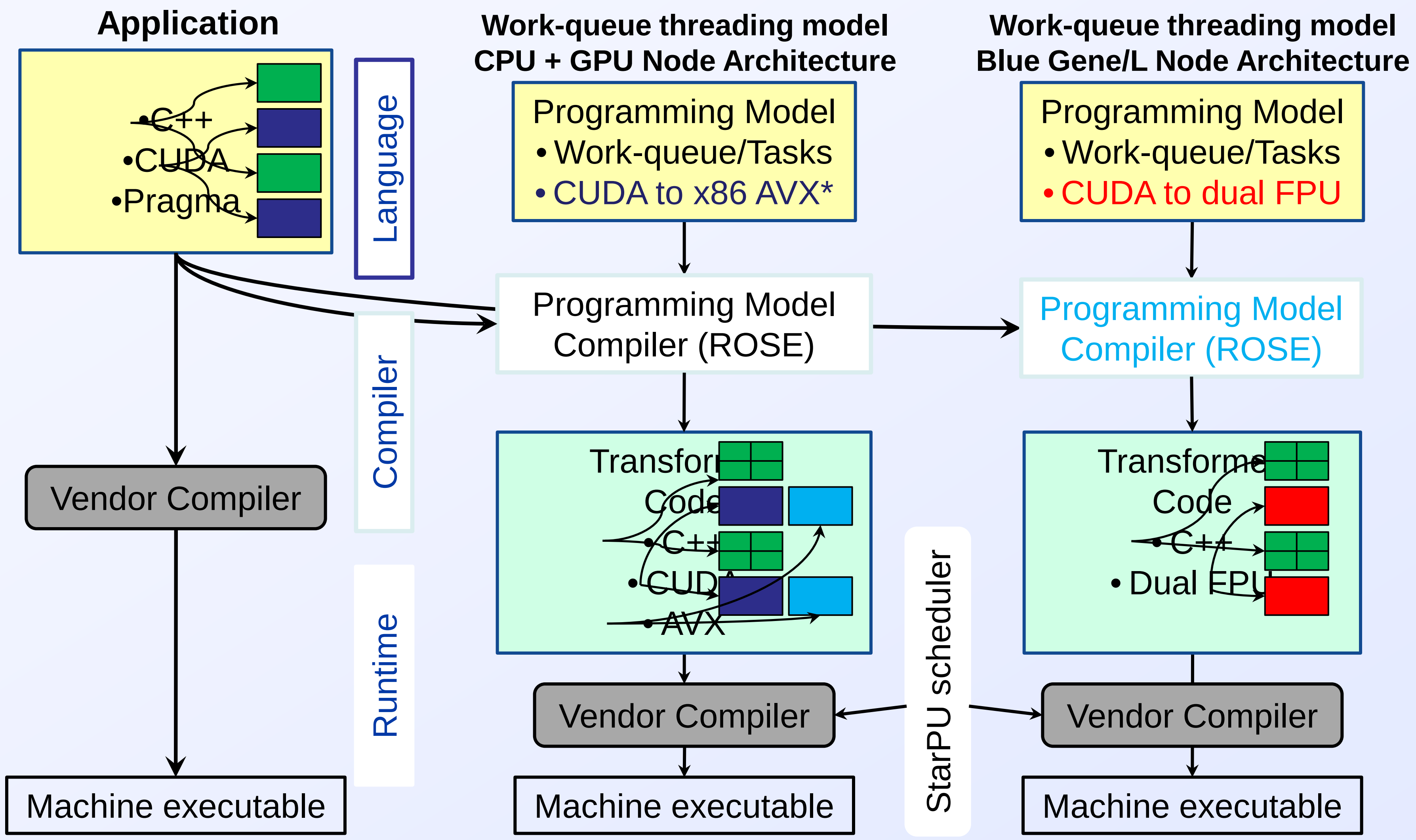
//The GPU is similarly scheduled

Compiler (level 2): Compiler transformations

1. parse pragma statements, 2. outline tasks, 3. translate CUDA to x86 AVX, and 4. push onto queue.

```
for (int idxPatch = 0; idxPatch < numPatch; ++idxPatch) {
   CPUQueue.add(outlined_task_1(globalData, idxPatch);
 GPUQueue.add(outlined_task_2(globalData, idxPatch);
 }
```

Our framework allows users to easily target new architectures (use case #3)

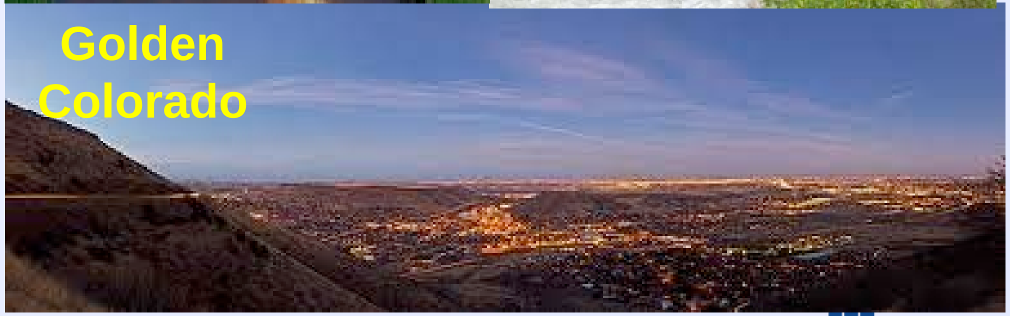


*AVX: Advanced Vector Extensions

Different types of driving change…

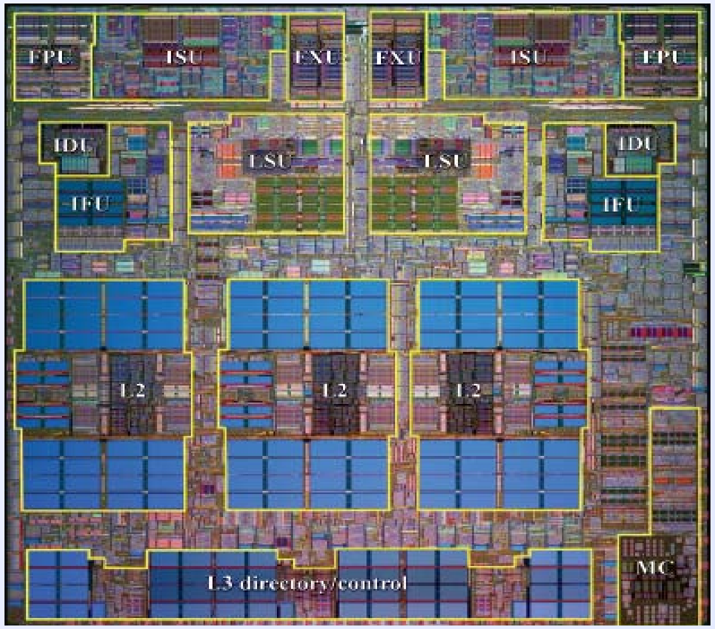
- Geologic Change
- Periodic Change
- Economic Change

HPC is driven by economics



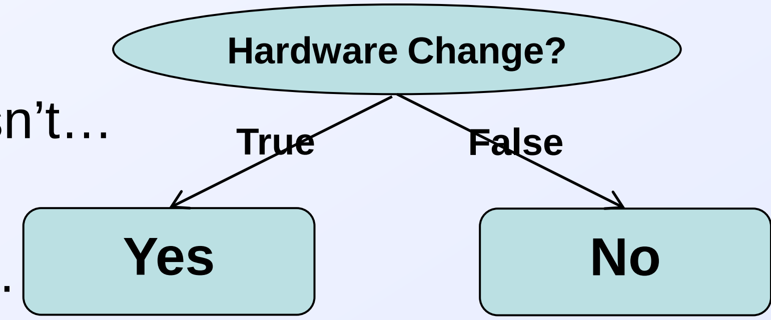
HPC is driven by Economics

- ***Hardware Rules***
  - How well SW runs on new hardware, drives a lot:
    - Applications code focus
    - Math algorithms selected
    - Computer Science research
- Problems generate opportunities
  - Performance
  - Architecture Design



- In the coming decade, will there be any fundamental shifts in how we do computational science?

In the coming decade, will there be any fundamental shifts in how we do computational science?



- Yes, if the hardware changes; No if it doesn't…

- Large changes in HPC hardware coming…

  So, let's focus on the **True** branch…

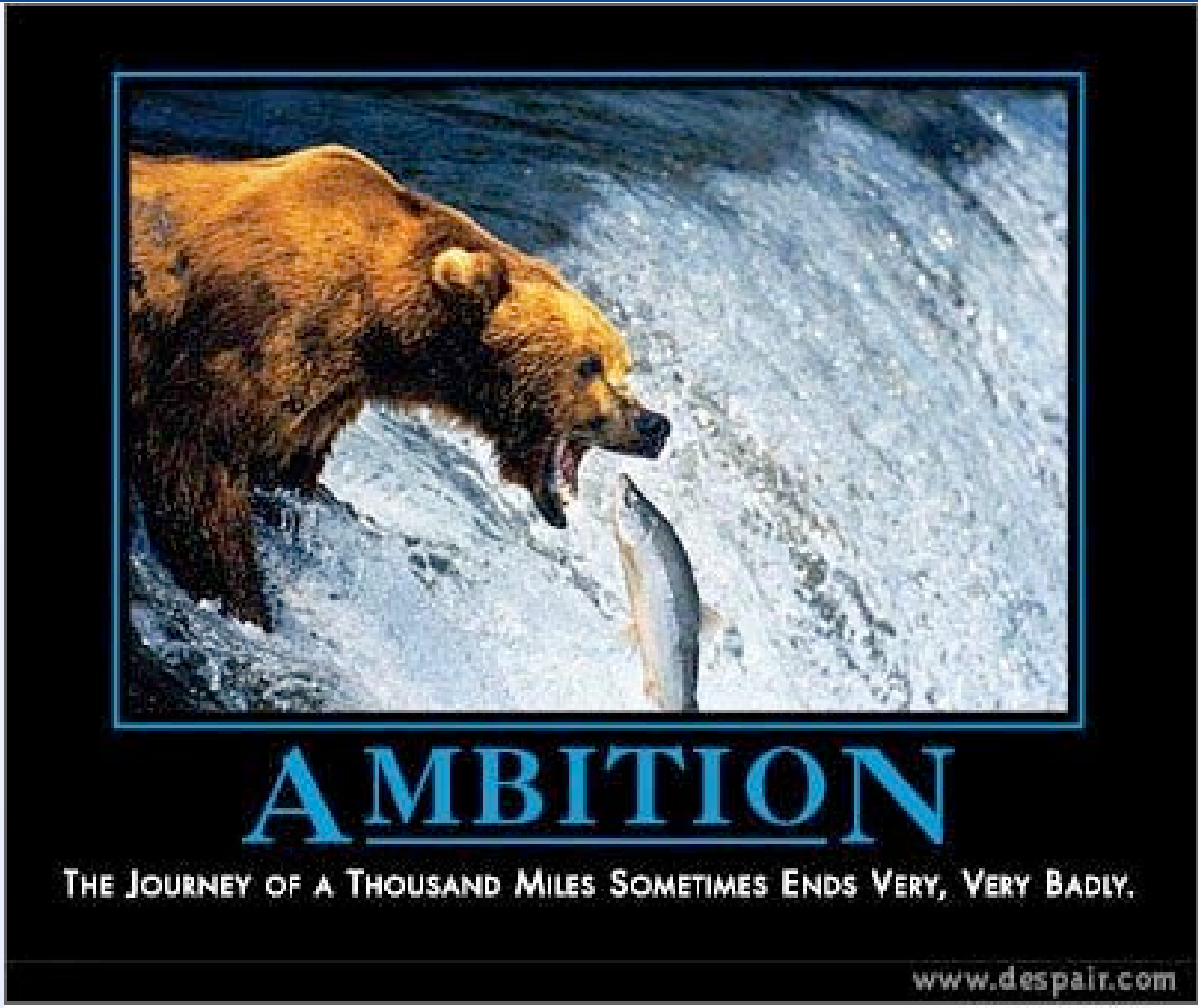
- *Algorithms will be more important as machines get more complex*
  - Performance differences may be dramatic
  - Winning and loosing algorithms (harsh reality)
  - Algorithm use will be machine dependent (SW complexity)
  - But change in hardware can make dramatic shifts in performance of different algorithms

In the coming decade, will there be any fundamental shifts in how we do computational science?

- *Software will be more expensive as machines get more complex*
  - Software will be more difficult to write
    - Software developers will bear the burden of addressing new hardware features
    - Performance problems will be more complex
  - Focus on community codes
  - Standards and libraries supporting standards
  - Programming Models will be an emphasis
    - Economics preclude new programming languages
    - MPI + X and other programming models
    - But programming models lag hardware (~5 years)
    - Not all programming models are focused on HPC
  - Requirements for tools will increase

In the coming decade, will there be any fundamental shifts in how we do computational science?

- YES
- *Algorithms*
- *Software*
- We are ambitious!
- *Let's not let this happened to us…*


AMBITION
The Journey of a Thousand Miles Sometimes Ends Very, Very Badly.
www.despair.com

Despair Inc.