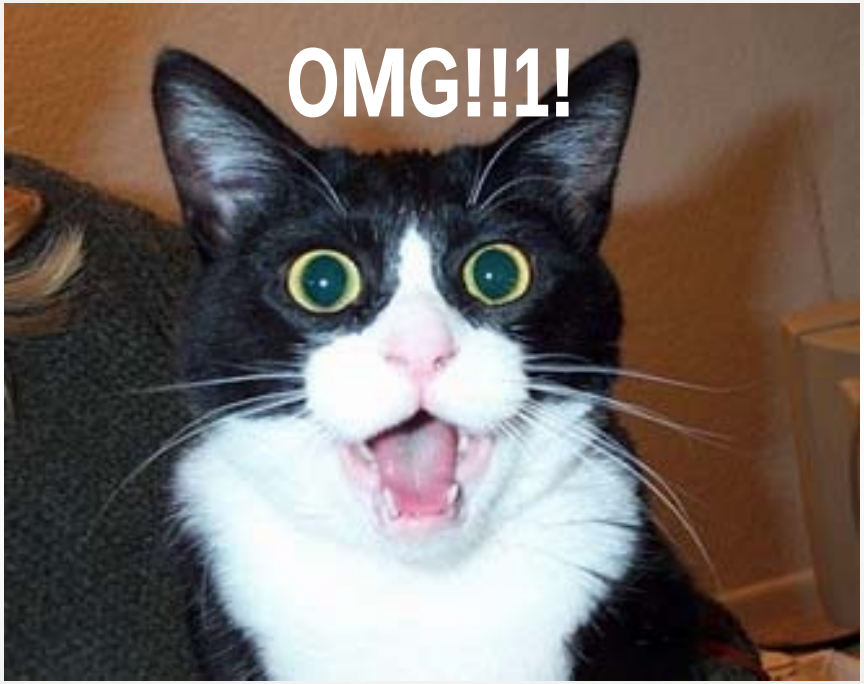
Five Things About HPC Programming Models I Can Live Without

Sung-Eun Choi, Cray Inc.

DOE Workshop on Exascale Programming Challenges

July 27th, 2011

1. Exposing platform details in algorithms



1. Exposing platform details in algorithms

Just because we don't (for the most part) use inline assembly anymore, doesn't mean we're not exposing platform-specific details into our codes

For example:

- Specifying communication styles
- Using hybrid schemes for parallelism

1. Exposing: Specifying communication styles

Communication is required when there are data dependences in computation

- The style used (1-sided, 2-sided, asynchronous, synchronous, shared memory, etc.) is often dictated by machine-specific details such as hardware primitives, bandwidth, latency, memory hierarchy, etc.
- Embedding the communication style will all but guarantee a non-portable program

1. Exposing: A good start

Chapel's data-centric programming model:

- Rich data types (e.g., arrays: multidimensional, sparse, associative, etc.)
  - Support iteration and other "collective" operations
- Global-view programming
  - Index-free
  - Rank-independent

1. Exposing: Hybrid schemes

Hybrid schemes are no fun to use

- Does it seem reasonable to anyone to have as many parallel programming notations as there are levels of parallelism in your hardware (machine, node, core, thread, vector, instruction, …) and software (executable, function, loop, iteration, thread, …)?

1. Exposing: A good start

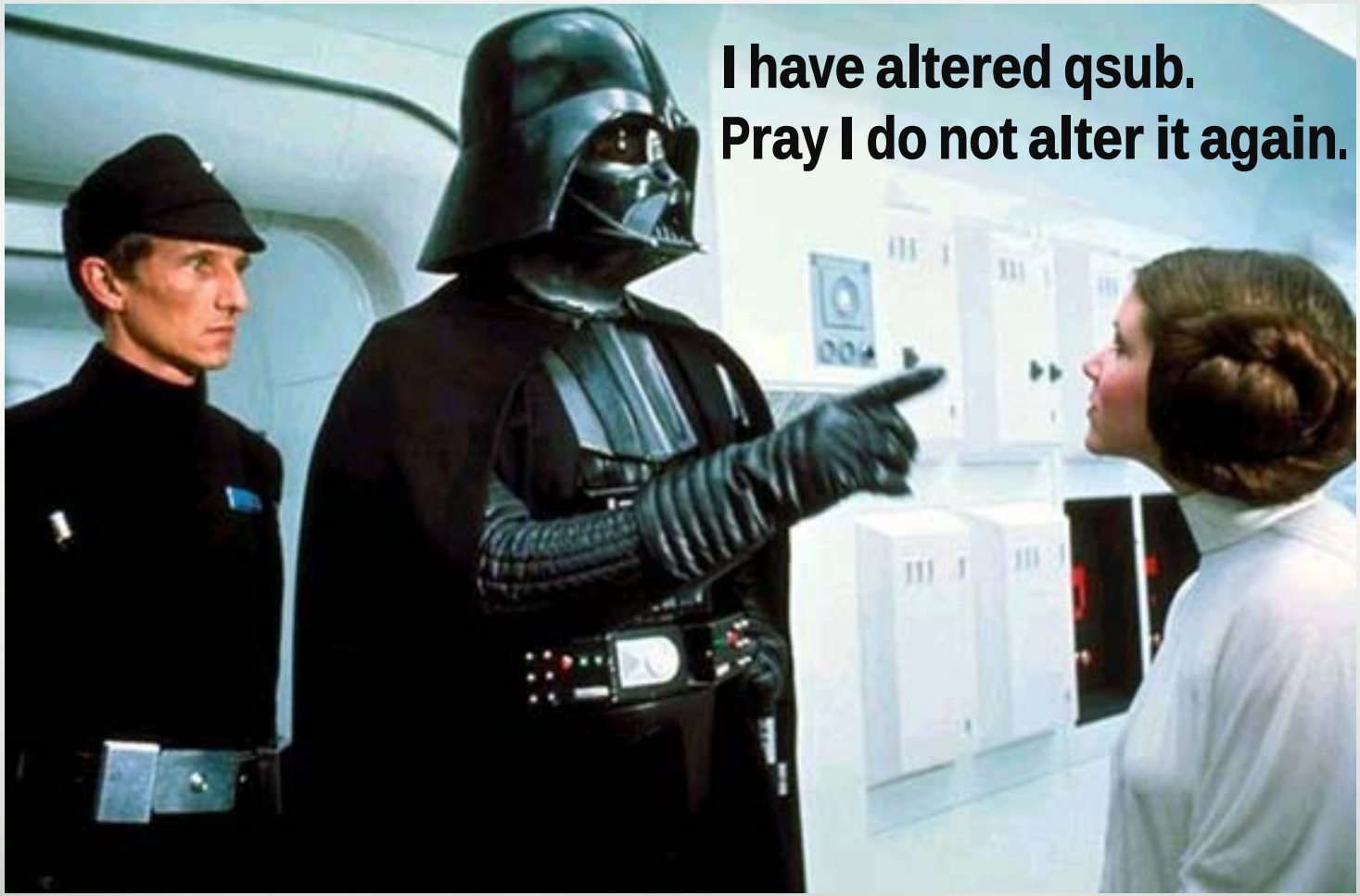
Chapel's multi-resolution design:

- Enables programming at multiple levels of abstraction
  - For the domain expert: high level concepts for specification of an algorithm and parallelism
  - For the HPC experts: lower levels for details like data distribution and task scheduling
- With the correct abstractions, the compiler works in concert with the HPC experts (programming in Chapel) to generate optimized code
  - Several key optimizations have been implemented using this model including some that are traditionally performed by a compiler

1. Exposing: Challenges

- Correctly defining the levels of abstraction
  - Defining roles for a hierarchy of programmers
- It's very difficult for hard-core (HPC) programmers to let go of details
  - It's macho
  - We want to understand the details
- It will probably be necessary to be able to perform cross-level optimizations
  - Can we generalize auto-tuning?

2. Unportable, unperformant programs



2. Unportable, unperformant programs

My dream:

```
% cc –fast my_working_program.c
% ./a.out
```

- .. to do this on any machine and on the first try get within 90% of the performance I could acheive under the best of circumstances
- No qsub/bsub/*sub or mpirun/aprun/*run commands

2. Unportable: My reality

My reality:

```
% cc –fast my_working_program.c
```

(Let's assume this always works)

1. Copy a qsub/bsub/*sub script
   - Plug in my account name, appropriate queue name, site specific parameters, etc.
2. Figure out the correct flags for mpirun/aprun/*run
   - Everyone has their own
3. Run my program

2. Unportable: My reality continued

4. Curse the results
   - It's not unheard of to get a 10x performance degradation using similar but new/faster hardware on the first try
5. Try to get help
   a) Read mailing list archives or search the web
   b) Email friends/colleagues who might have had similar experiences
   c) Email HPC consultants
6. Fix the proposed issues and go to step 3

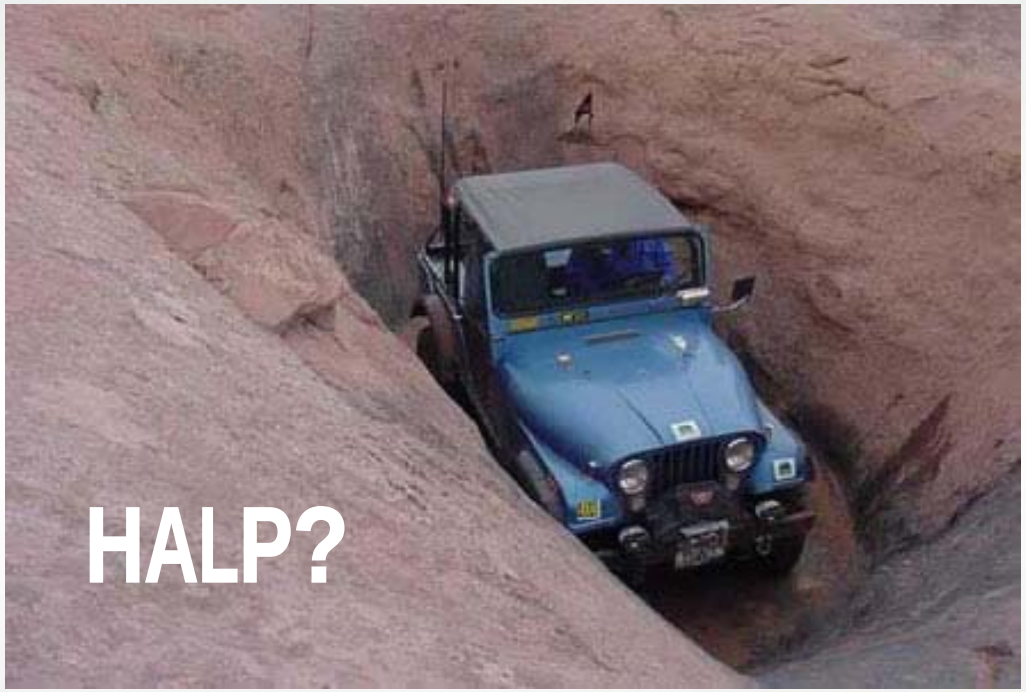
2. Unportable: A good start

- Compiling
  - Multi-platform compilers
  - Cray's PrgEnv modules
- Running
  - Chapel's launching model
    - User specifies the launching mechanism, and the generated code uses the most sensible arguments for that mechanism
- Performance
  - Chapel's multi-resolution design
  - Profile-guided compilation/optimization
  - Auto-tuning

2. Unportable: Challenges

- Compiling
  - Using mainstream libraries/packages
- Running
  - Sites often wrap qsub/bsub/*sub with their own scripts that customize flags according to their scheduling policies and restrictions
  - Launcher flags can affect performance
- Performance
  - Is auto-tuning generalizable?

3. Not providing a path for migration



3. Not providing a path for migration

It seems unrealistic to invest in any effort in a new programming model without a solid path for migrating existing large scale applications

Some may view this as a *lateral* port (migrating functionality), but it's not

- It requires a holistic approach
- But in reality, it probably needs to be done in phases

3. Migration: A start

- Chapel's extern facility
  - Call your routines from a Chapel program
- Interoperability at the language level
  - Call new kernels written in Chapel from your program
- Data format standards
  - Implement phases of your entire system in Chapel

3. Migration: Challenges

It's difficult to convince people to make what they perceive to be a lateral move

- Maintaining performance during the transition
  - Separate compilation issues
- Insuring the a path will eventually lead to good performance
  - Holistic approach
  - Could we be hampered by the initial design?
- Ease of use
  - Could we have language support for interoperability?

4. Forgetting about the general case

4. Forgetting about the general case

Just because we are targeting Exascale doesn't mean we should forget about the *lesser* scales

- We should optimize the cases we care about, but implement a general language

Two examples of how this hurts us:

- Limited applicability
- Reduced supply of "new blood"

4. Forgetting: Limited applicability

Without the inclusion of general *and modern* concepts, any Exascale programming model will have a difficult time being successful

- Why? Because early adopters are often those don't need much performance, and in the early stages, you don't have performance, so you'd better have all the *right* features

4. Forgetting: New blood

Without general applicability, the HPC workforce is going to find it even more difficult to attract fresh talent from universities

- Why? HPC is less approachable if it uses esoteric and seemingly outdated tools

4. Forgetting: A good start

Chapel includes a rich base language with modern features such as high-level data types (tuples, ranges, etc.), object-oriented support, iterators, function overloading, and generic programming capabilities

- Consequently, it's also natural to use it for rapid prototyping
- Chapel users include programming language geeks, scientists, engineers, students, educators, and HPC experts

4. Forgetting: Challenges

- Engaging the broader academic community in our efforts
  - No disrespect to those from the academic community here, but we need to get more of you involved
- Salesforce.com, Zillow, Facebook, Google, etc.
  - The field needs renewed sex appeal (hopefully some of it will come from the above)

5. Ignoring the elephants in the room

5. Ignoring the elephants in the room

Any new programming model cannot ignore:

- Debuggers
- Performance tools
- I/O
- What else? I can't see the other elephants because these take up too much room

and we can't easily re-use these from another model (trust me, I tried)

5. Ignoring: A good start

The *Baby Steps Model* (BSM)

What I tell new Chapel programmers:

- Write and debug your Chapel program on your desktop as a single *locale* (node), multi-*task* (threaded) program
- When it's working, rebuild it for *multi-locale* execution on your desktop
- When that's working, take it to the Big Iron

This seems obvious. Apparently it is not.

5. Ignoring: A good start

If you have a truly portable program, then you should be able to implement the BSM

- Then we can focus on the smaller scale elephants

5. Ignoring: Challenges

- Engaging the folks working on the entire tool chain
- The BSM buys us some time, but we still have to be able to address the hardest problems at the Exascale
  - Need to design for the Exascale, even if implementing at the small scale
- Can we even apply the small scale model to
  - Performance debugging?
  - I/O?
  - Resiliency?

Summary of Challenges

- Getting the programming model right
  - Correctly defining abstractions so as to clearly define the roles of the hierarchy of programmers
  - Correctly defining a path for migration that can take full advantage of the programming model
- Engaging a broader community
  - For acceptance
  - For tool development
  - For the future of the field


CRAY
THE SUPERCOMPUTER COMPANY