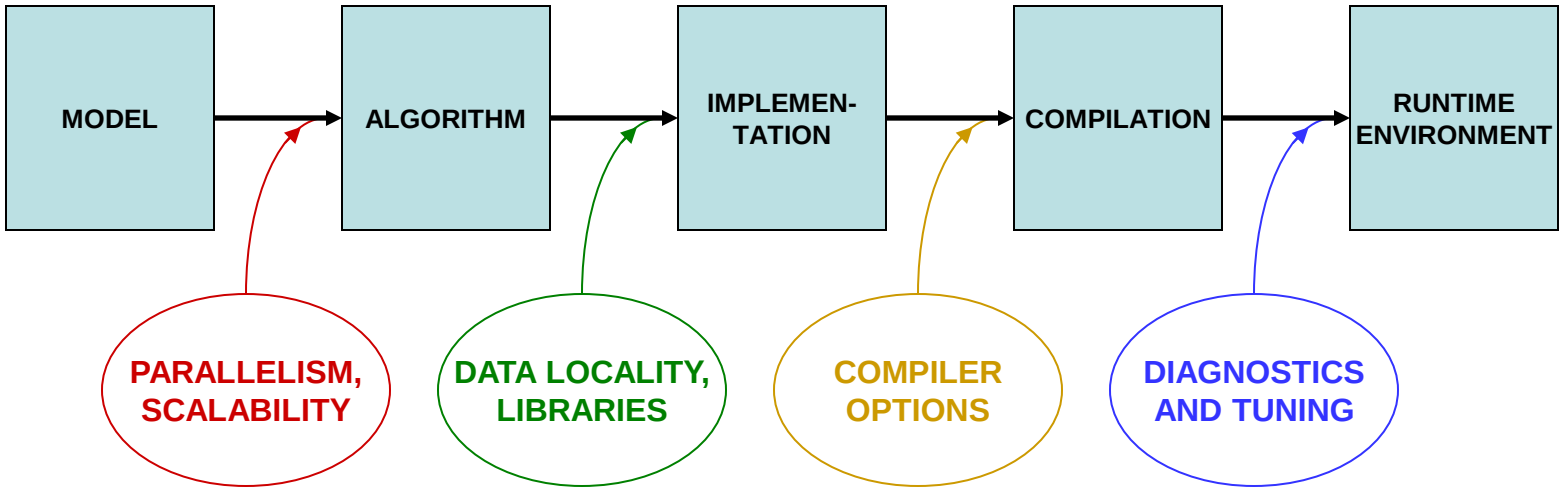
Scalability

Steve Lantz
Senior Research Associate
Cornell University Center for Advanced Computing (CAC)
slantz@cac.cornell.edu

Workshop: High Performance Computing on Stampede, Jan. 14-15, 2015

Putting Performance into Design and Development



We'll start with how to *design* for parallelism and scalability…

…later we'll talk about principles and practices during various stages of code *development* that lead to better performance on a per-core basis

Planning for Parallel

- Consider how your model might be expressed as an algorithm that naturally splits into many concurrent tasks
- Consider alternative algorithms that, even though less efficient for small numbers of processors, scale better so that they become more efficient for large numbers of processors
- Start asking these kinds of questions during the first stages of design, before the top level of the code is constructed
- Reserve matters of technique, such as whether to use OpenMP or MPI, for the implementation phase

Scalable Algorithms

- Generally the *choice of algorithm* is what has the biggest impact on parallel scalability

- An efficient and scalable algorithm typically has the following characteristics:
  - The work can be separated into numerous tasks that proceed almost totally independently of one another
  - Communication between the tasks is infrequent or unnecessary
  - Lots of computation takes place before messaging or I/O occurs
  - There is little or no need for tasks to communicate globally
  - There are good reasons to initiate as many tasks as possible
  - *Tasks retain all the above properties as their numbers grow*

What *Is* Scalability?

- Ideal is to get *N* times more work done on *N* processors

- Strong scaling: compute a fixed-size problem *N* times faster
  - Speedup $S = T_1 / T_N$; linear speedup occurs when $S = N$
  - Can't achieve it due to Amdahl's Law (no speedup for serial parts)

- Weak scaling: compute a problem *N* times bigger in the same amount of time
  - Speedup depends on the amount of serial work remaining constant or increasing slowly as the size of the problem grows
  - Assumes amount of communication among processors also remains constant or grows slowly

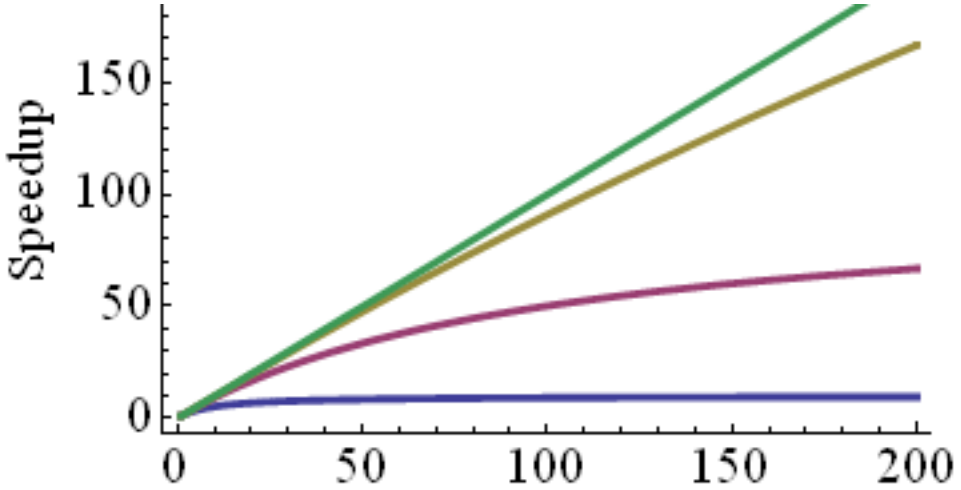
How Amdahl's Law Defeats Strong Scaling

- For large *N*, the parallel speedup doesn't asymptote to *N*, but to a constant 1/*a*, where *a* is the serial fraction of the work
- The graph below compares perfect speedup (green) with maximum speedup of code that is 99.9%, 99% and 90% parallelizable

```
T(N) = total time = p/N + s
     p = parallel workload
     s = serial time

S(N) = speedup = T(1)/T(N)
     = (p + s)/(p/N + s)

If a = s/(p + s), then
S(N) = N/[1+(N-1)a]
     -> 1/a for large N
```



Why Weak Scaling Tends to Work Better

- Let's relax the assumption that the parallel workload *p* is fixed; instead, assume *p*(*N*) = *Nt*, so that *p* grows with *N* (weak scaling)
- Again, the idea is to do *more* tasks of fixed size *t* in the *same* length of "wall" time, rather than a fixed workload in less time
- Gustafson's Law: the "scaled speedup" is linear in *N*
  - But slope is less than 1, unless the code is "embarrassingly parallel"

```
T(N) = total time = p/N + s
     p = N*t, grows with N

S(N) = speedup = T(1)/T(N)
     = (t + s)/(N*t/N + s)
     = 1, no speedup...

But more WORK gets done!...
```

```
U(N) = total WORK = p + s
     again, p = N*t, grows with N

W(N) = "scaled speedup" = U(N)/U(1)
     = (N*t + s)/(t + s)

If f = t/(t + s), then
W(N) = N*f + (1-f), scales with N
```

Is My Application Scalable?

If you're using Stampede, you're probably looking for *weak scaling*…

1. Need to run a *much larger* case using more resources
   - Example: run a fluid model at extremely high resolution
2. Need to run *many more* cases using more resources
   - Example: run a larger number of simulations to generate statistics
3. Commonly 1 and 2 are needed together
   - Local cluster has insufficient memory or takes unacceptably long

Getting *N* times the work done on *N* cores is feasible when…

- Small problem sizes keep every node of a local cluster busy
- Your code has the scalability properties mentioned earlier
- Easiest scenario: all cases are totally independent of each other
  - Yes, this is still parallel; it's called "embarrassingly parallel"

Capability vs. Capacity

- HPC jobs can be divided into two categories, capability runs and capacity runs
  - A capability run occupies nearly all the resources of the machine for a single job
  - Capacity runs occur when many smaller jobs fill up the machine simultaneously

- The big capability runs are typically achieved via weak scaling
  - Strong scaling usually applies only over some finite range of *N* and breaks down when *N* becomes huge because of Amdahl's Law, parallel overhead, etc.
  - A trivially parallelizable code is an extreme case of weak scaling; however, replicating such a code really just fills up the machine with a bunch of capacity runs instead of one big capability run

The Role of Benchmarks

- More sophisticated prediction of your code's scalability requires knowing details about hardware and software performance
- This is the purpose of running benchmarks
- Different types of benchmarks have different measurement goals:
  - *Hardware* or *micro-benchmarks* gauge low-level things like processor floating point speed, point-to-point bandwidth, and write speed to disk
  - *Synthetic benchmarks* focus on individual algorithms; for example, the NAS Parallel Benchmarks include separate tests oflinear algebra functions like pentadiagonal solvers and block tridiagonal solvers
  - *Application benchmarks* try to measure (in wall time) how much useful work is done by a system for a typical end-user code; in effect, it's a series of synthetic algorithms, with data movement and I/O in between
- Often these are run for various core counts, on multiple platforms

Predicting Actual Scalability

- Consider the time to compute a fixed workload due to N workers:

```
total time = computation + message initiation + message bulk
computation = parallel workload/N + serial time    (Amdahl's Law)
message initiation = number of messages * latency
message bulk = size of all messages / bandwidth
```

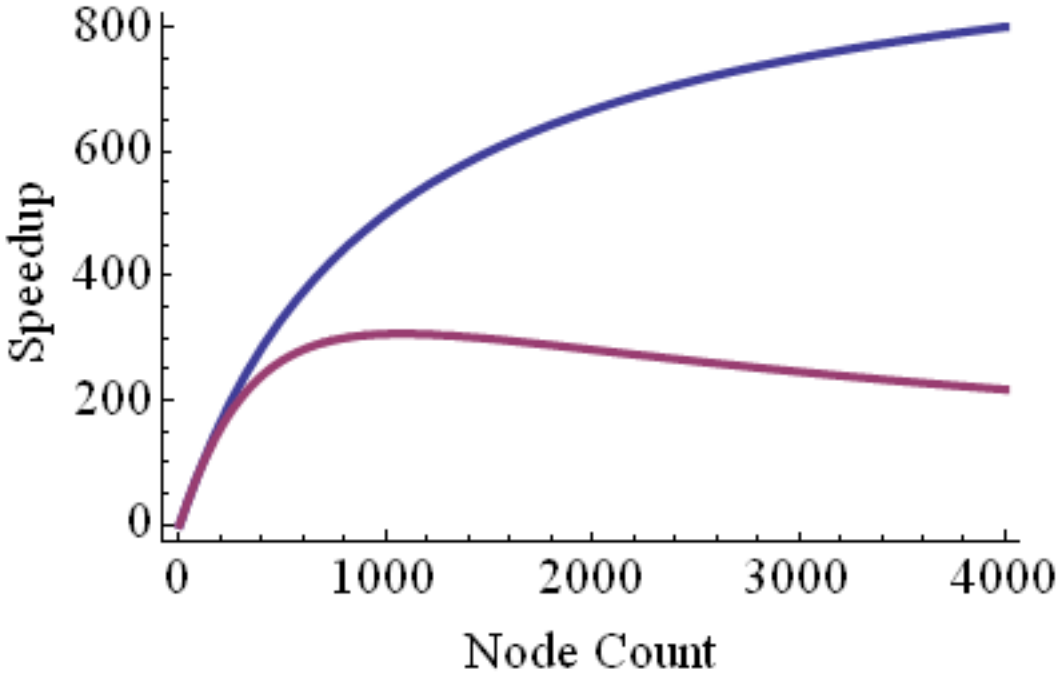
- The number and size of messages might themselves depend on N (unless all travel in parallel!), suggesting a model of the form:

```
total time = parallel workload/N + serial time
           + k0 * N^a * latency + k1 * N^b / bandwidth
```

- Latency and bandwidth depend on hardware and are measured via benchmarks; other constants depend partly on the application

The Shape of Speedup

Modeled speedup (purple) could be worse than Amdahl's Law (blue) due to the overhead of message passing. Look for better strategies.



Example of Performance Modeling

- Imagine a parallel code that simulates heat flow in a flat metal plate
  - Tasks are assigned different subdomains (domain decomposition)
  - Each task needs to communicate only with its nearest neighbors
- As *N* increases:
  - The number of messages per worker is *unchanged*
  - Message size per worker (edge data) actually *decreases* as $N^{-1/2}$
- Apply the previous model for strong scaling –

```
total time = parallel workload/N + serial time
           + k0 * N^a * latency + k1 * N^b / bandwidth
```

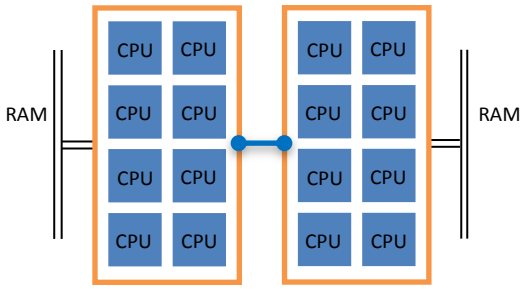
- Assuming our "non-blocking" network allows all workers' messages to travel in parallel (Stampede comes close!), we find a = 0, b = -1/2
- Our formula does not account for synchronization overhead

How Do You Get to Petascale with MPI?

- Favor local communications over global
  - Nearest-neighbor is fine; all-to-all is trouble
- Avoid frequent synchronization
  - Any *load imbalances* are paid for through waiting at sync points
  - Thus, MPI collective calls may become surprisingly long (if blocking)
  - Even random, brief OS interruptions (“jitter” or “noise”) can effectively cause load imbalances
  - Balancing must become ever more precise as the number of processes increases...
- But you don’t have to program with MPI alone
  - There are additional ways to use all the resources of an HPC system…

Non-Uniform RAM Arrangement on Stampede

- *Many nodes → distributed memory*
  - each node has its own local memory
  - not directly addressable from other nodes
- *Multiple sockets per node*
  - each node has 2 sockets (chips)
- *Multiple cores per socket*
  - each socket (chip) has 8 cores
- *Memory spans all 16 cores → shared memory*
  - node's full local memory is addressable from any core in any socket
- *Memory is attached to sockets*
  - 8 cores sharing the socket have fastest access to attached memory
  - we are ignoring any attached MIC coprocessors for the moment…



Dealing with NUMA

How do we deal with NUMA (Non-Uniform Memory Access)?

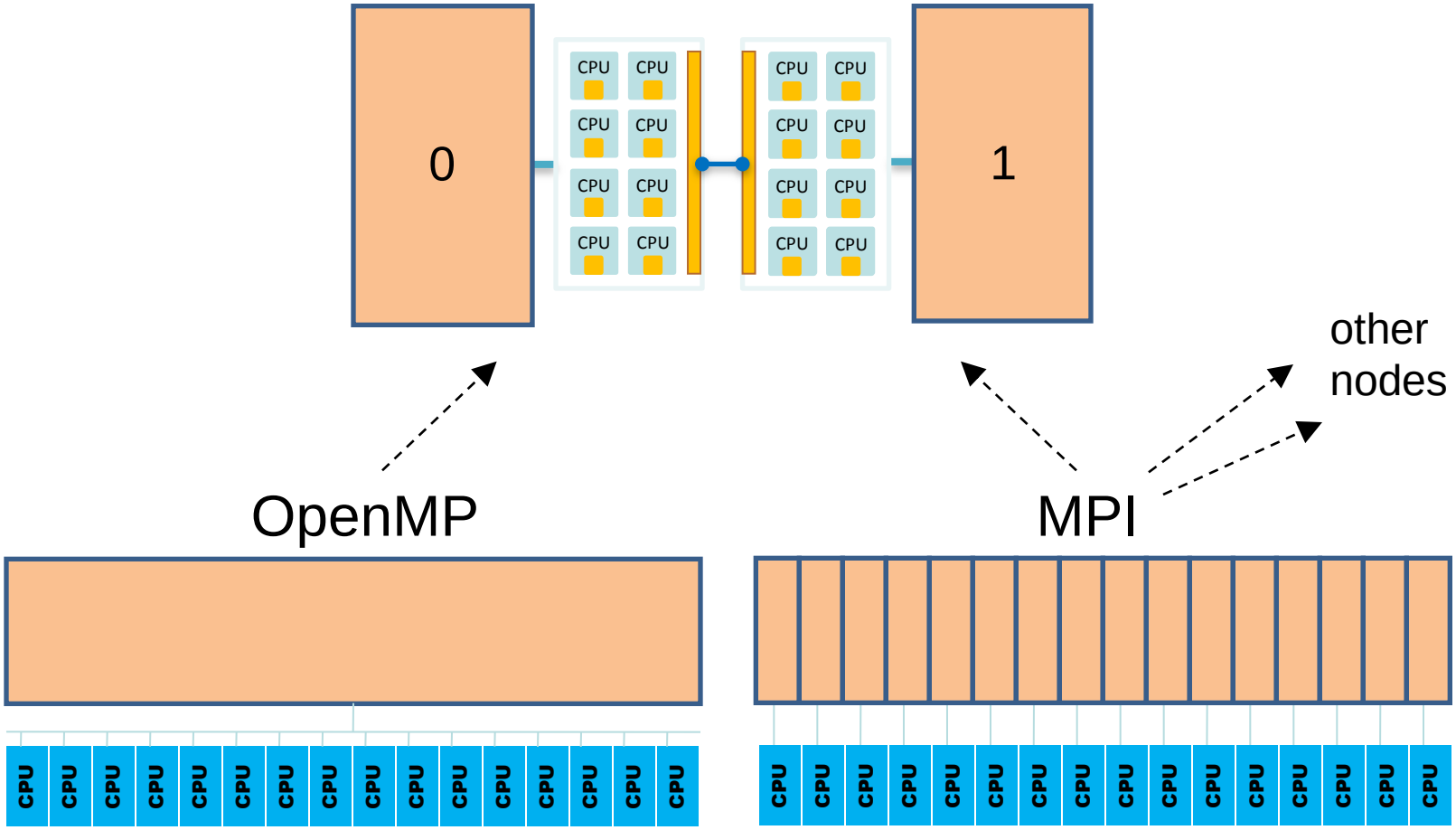
Parallel programs usually assume one of two uniform architectures

- Threads for ***shared memory***
  - parent process uses OpenMP or pthreads to fork multiple threads
  - threads share the same virtual address space
  - also known as SMP = Symmetric MultiProcessing
- Message passing for ***distributed memory***
  - processes use MPI to pass messages (data) between each other
  - each process has its own virtual address space

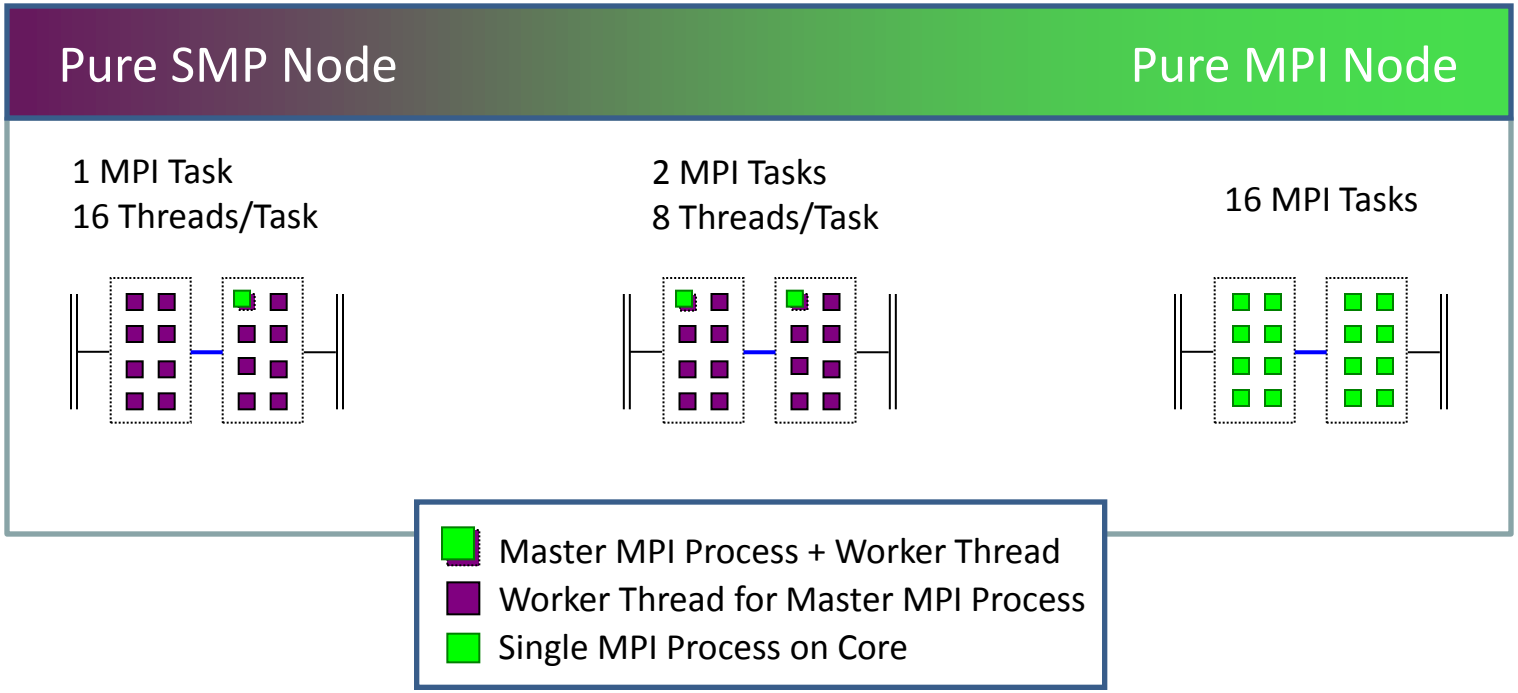
If we attempt to combine both types of models –

- ***Hybrid programming***
  - try to exploit the whole shared/distributed memory hierarchy

Two Views of a Stampede Node



Creating Hybrid Configurations



To achieve configurations like these, we must be able to:

- Assign to each process/thread an *affinity* for some set of cores
- Make sure the *allocation* of memory is appropriately matched

Threading Example: One MPI, Many OpenMP

Fortran

```
include 'mpif.h'
program hybsimp


call MPI_Init(ie)
call MPI_Comm_rank(...irk,ie)
call MPI_Comm_size(...isz,ie)
!Setup shared mem, comp/comm

!$OMP parallel do
  do i=1,n
    <work>
  enddo

!Compute & communicate
call MPI_Finalize(ierr)
end
```

C

```
#include <mpi.h>
int main(int argc,
  char **argv) {
int rank, size, ie, i;
ie= MPI_Init(&argc,&argv[]);
ie= MPI_Comm_rank(...&rank);
ie= MPI_Comm_size(...&size);
//Setup shared mem, comp/comm

#pragma omp parallel for
  for(i=0; i<n; i++){
    <work>
  }

// compute & communicate
ie= MPI_Finalize();
}
```

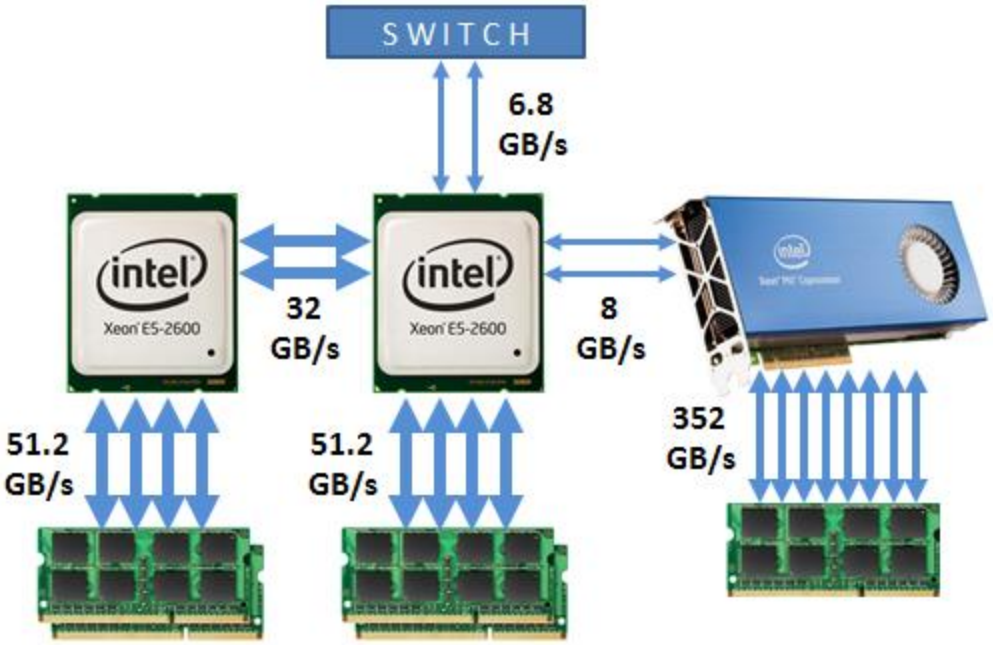
Programming for MIC: Hybrid *and* Heterogeneous

- Each Stampede node currently has 2 processors + *1 MIC card*
- MIC = Many Integrated Cores = a "coprocessor" on a PCIe card that features >60 cores; released as Xeon Phi™
  - Represents Intel's response to GPGPU, especially NVIDIA's CUDA
  - Answers the question: if 8 modern Xeon cores fit on a die, how many early Pentiums would fit?
- MIC answers CUDA's API problem: just compile like any normal code
  - Instruction set is x86 with support for 64-bit addressing
  - Recent x86 extensions may not be available
  - Developers use familiar Intel compilers, libraries, and tools
- However, MIC adds yet another level of programming complexity
  - Stampede is a multi-core machine where not all the cores are the same

Levels of Communication

Links to and within a node

- Least speedy: PCIe2 to the external InfiniBand
- Comparable speed: PCIe2 to the MIC!
- Fastest: channels to RAM (about 6GB/s/core on host and MIC alike)
- Comparable speed: dual QPI link between the two sockets on the host, for uniform memory sharing between the processors

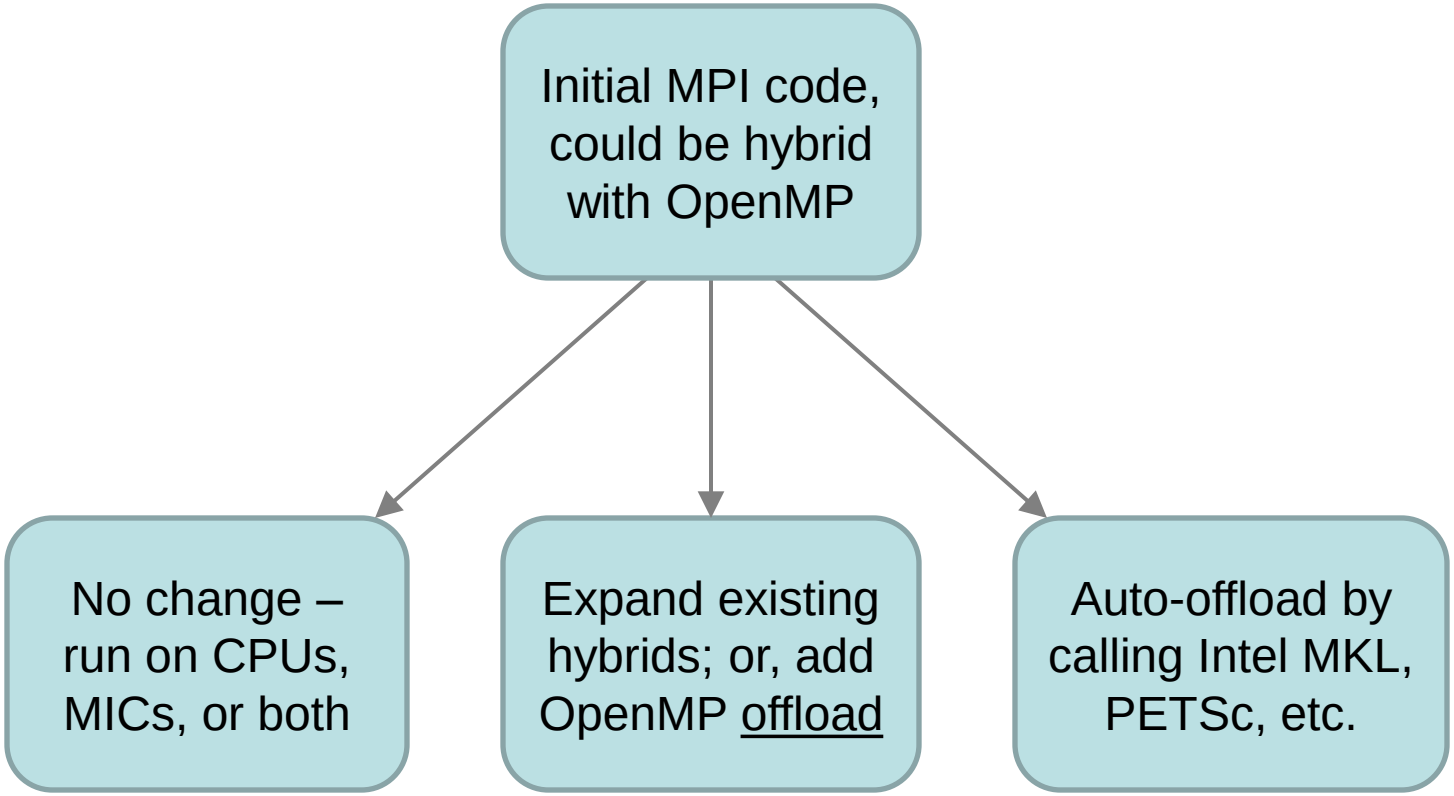


The MIC is like another node on the IB network, with its own OS, own internal memory, and own external IP address

Implications for Hybrid Programming

- Within a node, there ought to be loose coupling between the Sandy Bridge cores on the one hand, and the Xeon Phi cores on the other
- Precisely the same loose coupling ought to carry over to these hardware groups on other nodes…
- Conceptually, it's as if we have a double-size cluster consisting of two very different types of nodes (host and MIC)
- How does a hybrid code achieve the needed loose coupling?
  - Run several MPI processes on the MIC as well as on the host ("symmetric"); have each process fork enough OpenMP threads to keep all the cores busy
  - Run MPI processes only on host; use the offload capability to launch OpenMP threads on MIC

MIC Strategies for HPC Codes



Conclusions

- Scalability is *the* issue in large-scale computing
- Scalability is dominantly affected by the choice of algorithm
- A scalable algorithm has the following characteristics:
  - Natural *high-level* separation into many independent parallel tasks
  - Infrequent, asynchronous communication between tasks
  - Rare synchronization of tasks (even tasks that are load balanced)
- If the above isn't true of your parallel algorithm, look for another
- Weak scaling is sufficient: do *N* times the work in the same time
- Performance models and benchmarks help in understanding limits
  - Can account for particular software and hardware features
- Forward-looking architectures like Stampede require hybrid coding
  - Work must be split into processes and threads on heterogeneous cores