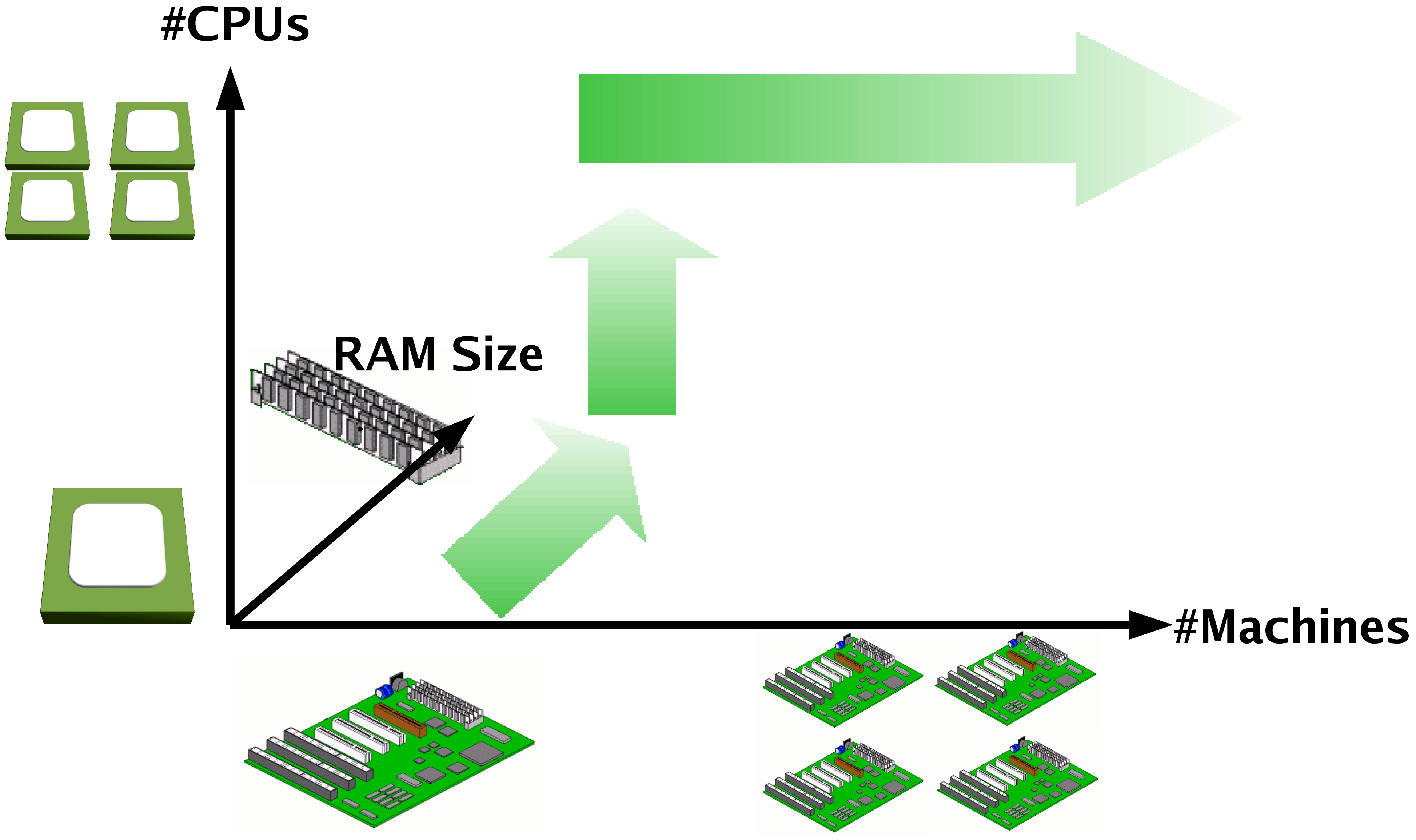
Adapt Programming for More Complex Machines

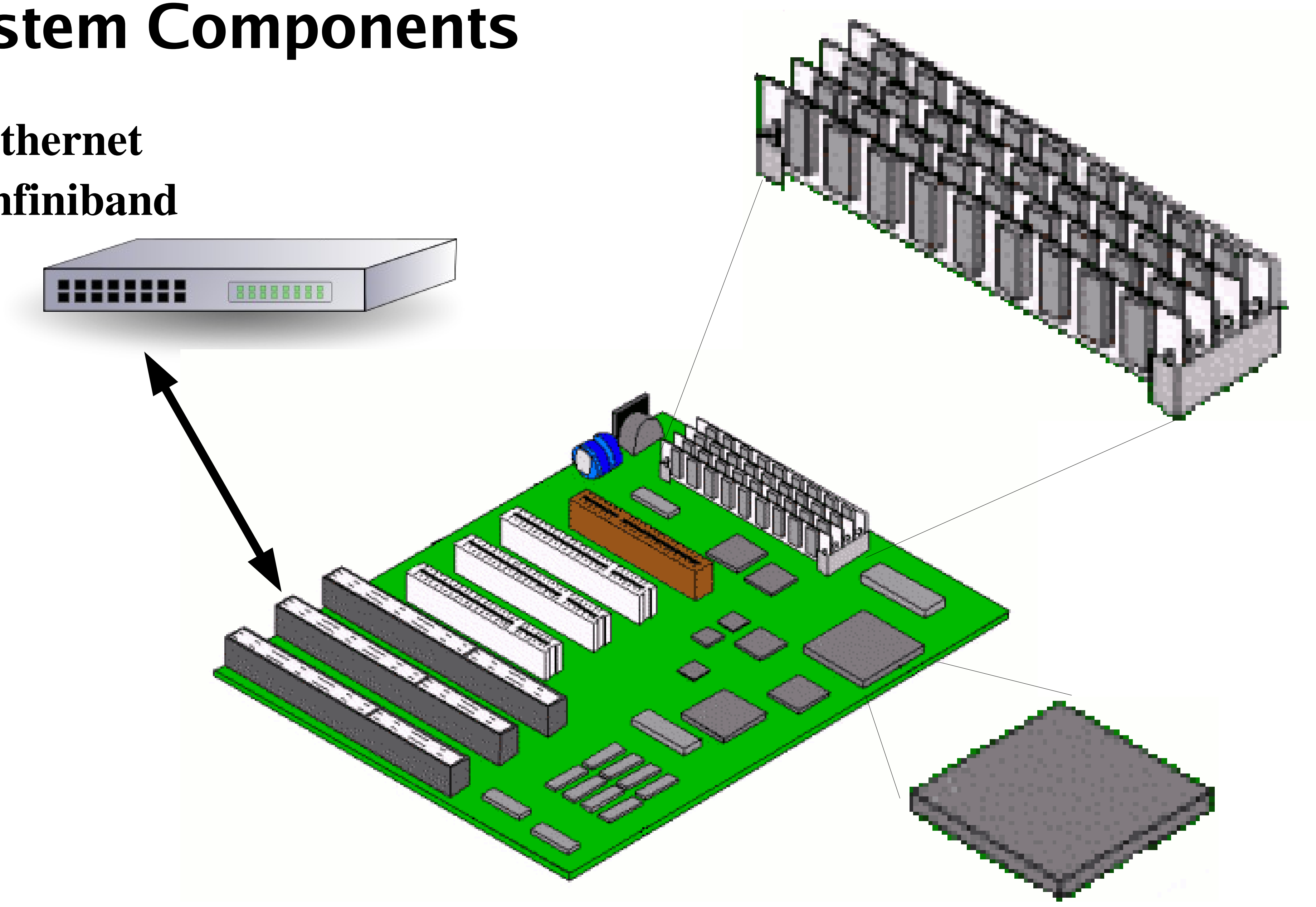
Ulrich Drepper

Performance and Scalability



System Components

Ethernet
Infiniband



Processor Changes

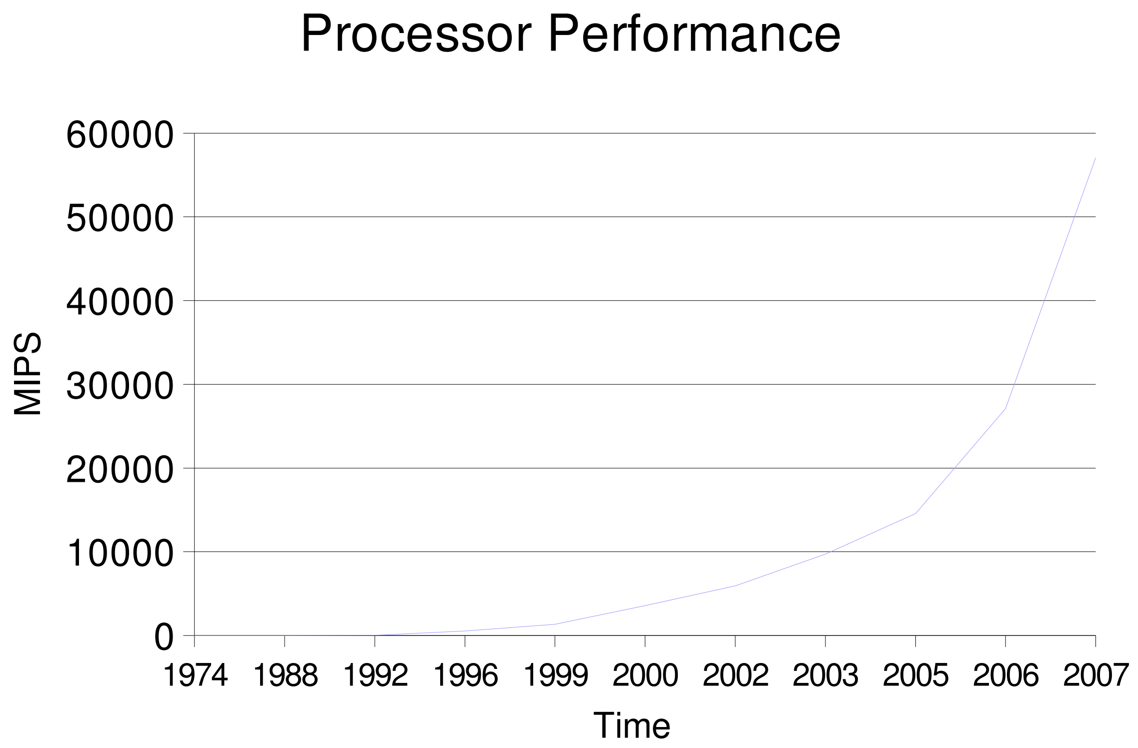
- Processor Features
  - Process Reduction
  - Pipelining
  - Out-Of-Order execution
  - Code and Data Caches

Good News: CPU handles it by itself

Although Cache-Aware Programming Can Help

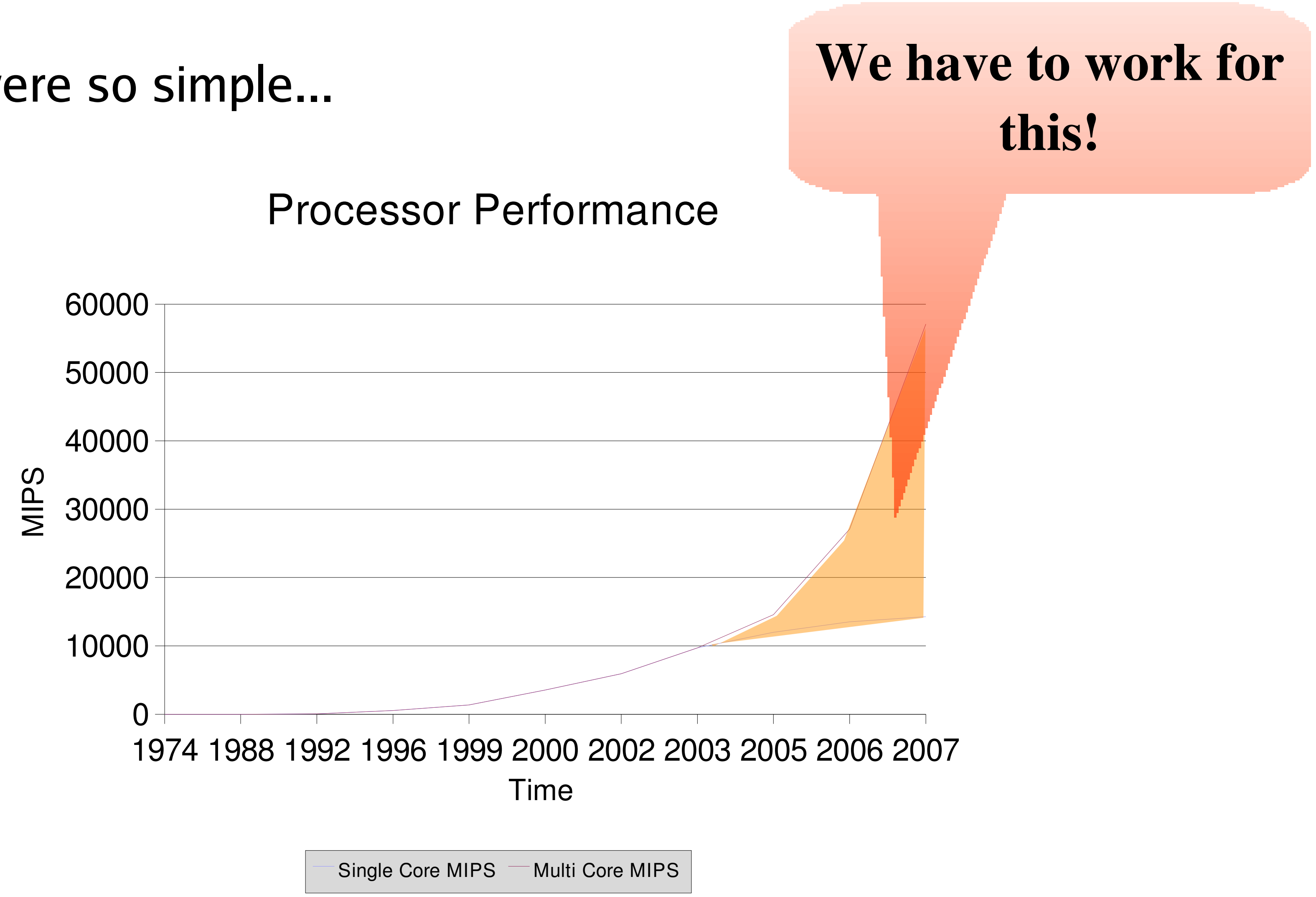
Easy Timers for Programmers Are Over

- Moore's Law seems still be in full swing



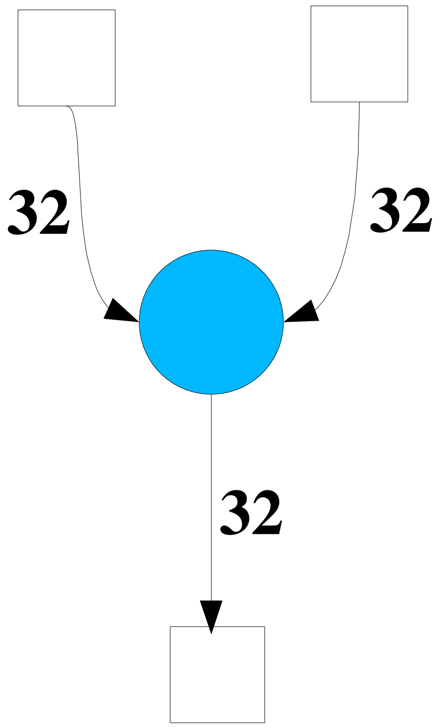
Easy Timers for Programmers Are Over

- If it only were so simple...



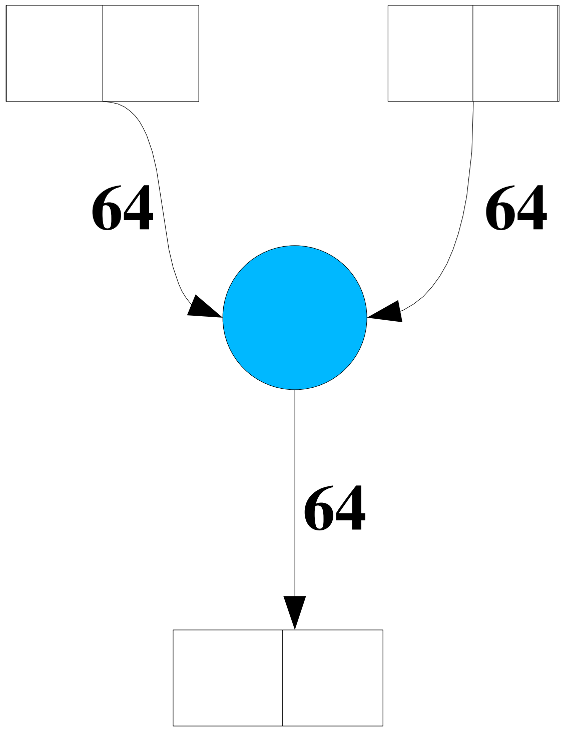
SIMD

- Single-Instruction/Multiple Data
  - Normal Arithmetic:



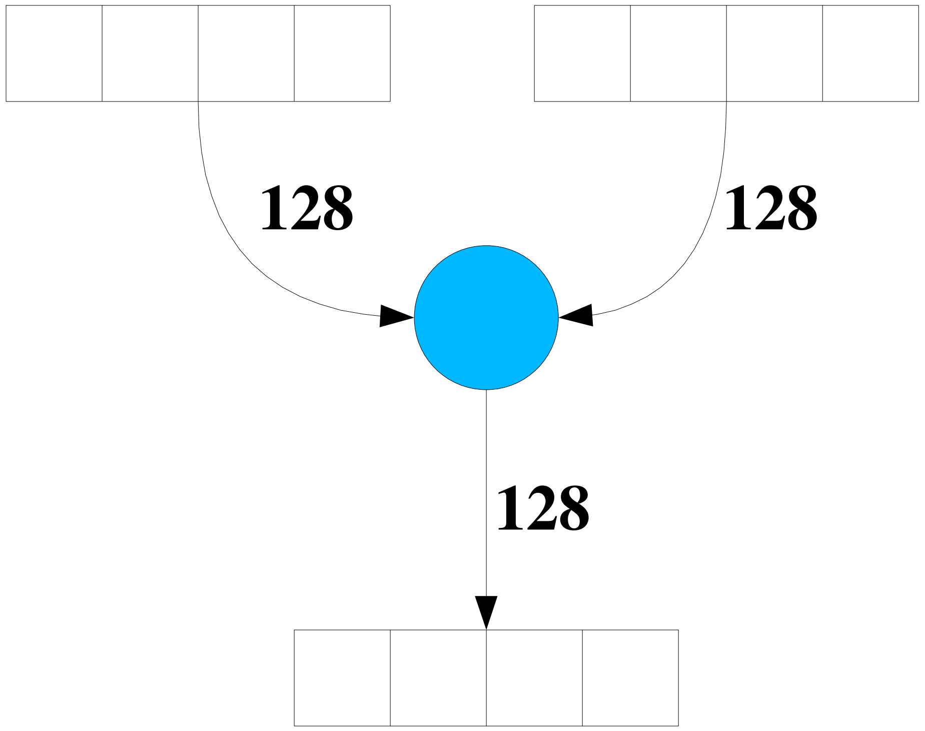
SIMD

- Single-Instruction/Multiple Data
  - Intel: MMX



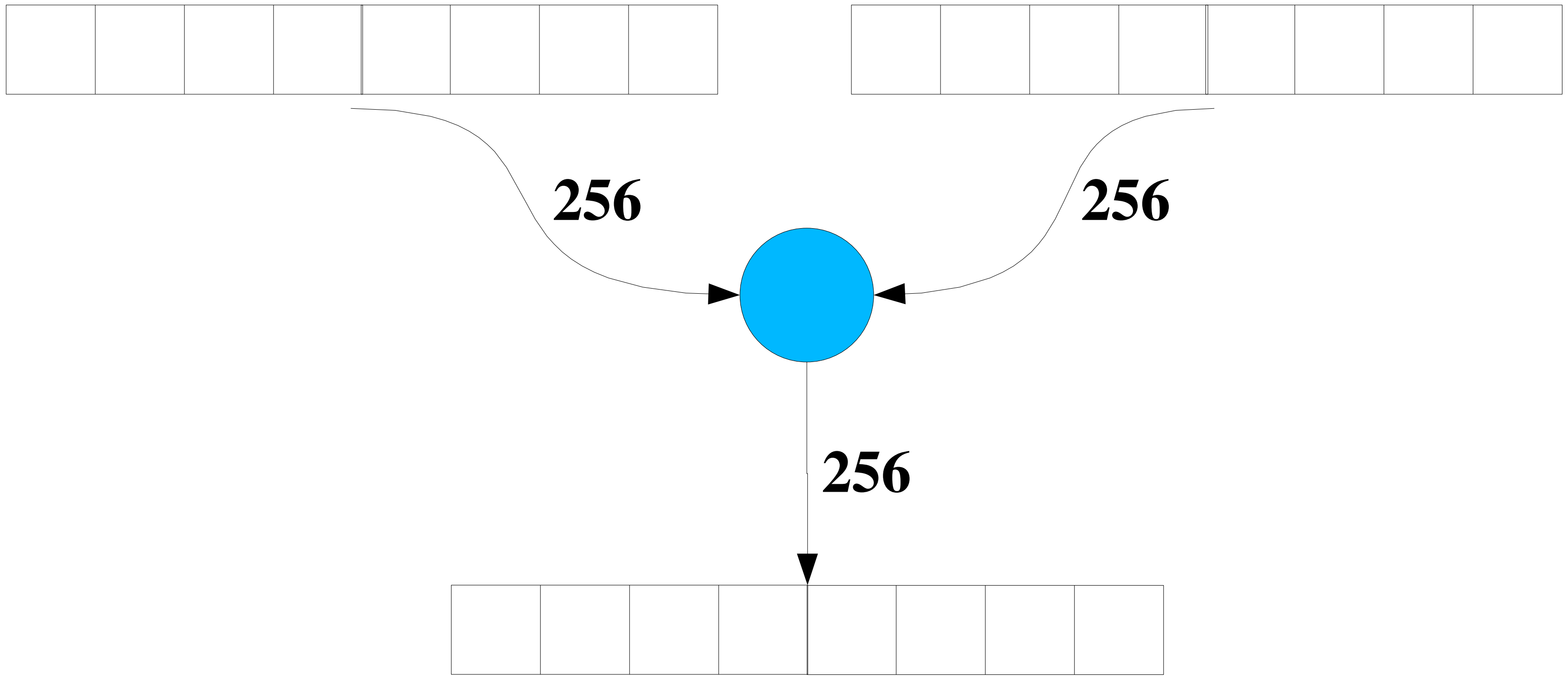
SIMD

- Single-Instruction/Multiple Data
  - Intel: SSE



SIMD

- Single-Instruction/Multiple Data
  - Intel: AVX



SIMD

- Auto-vectorization:
  - Research for 30 years
  - Somewhat working
  - Pattern matching in compiler
    - Slight changes might mean miss
- Better approach:
  - Compiler intrinsics
  - Special, hand-coded assembler

Processor Changes

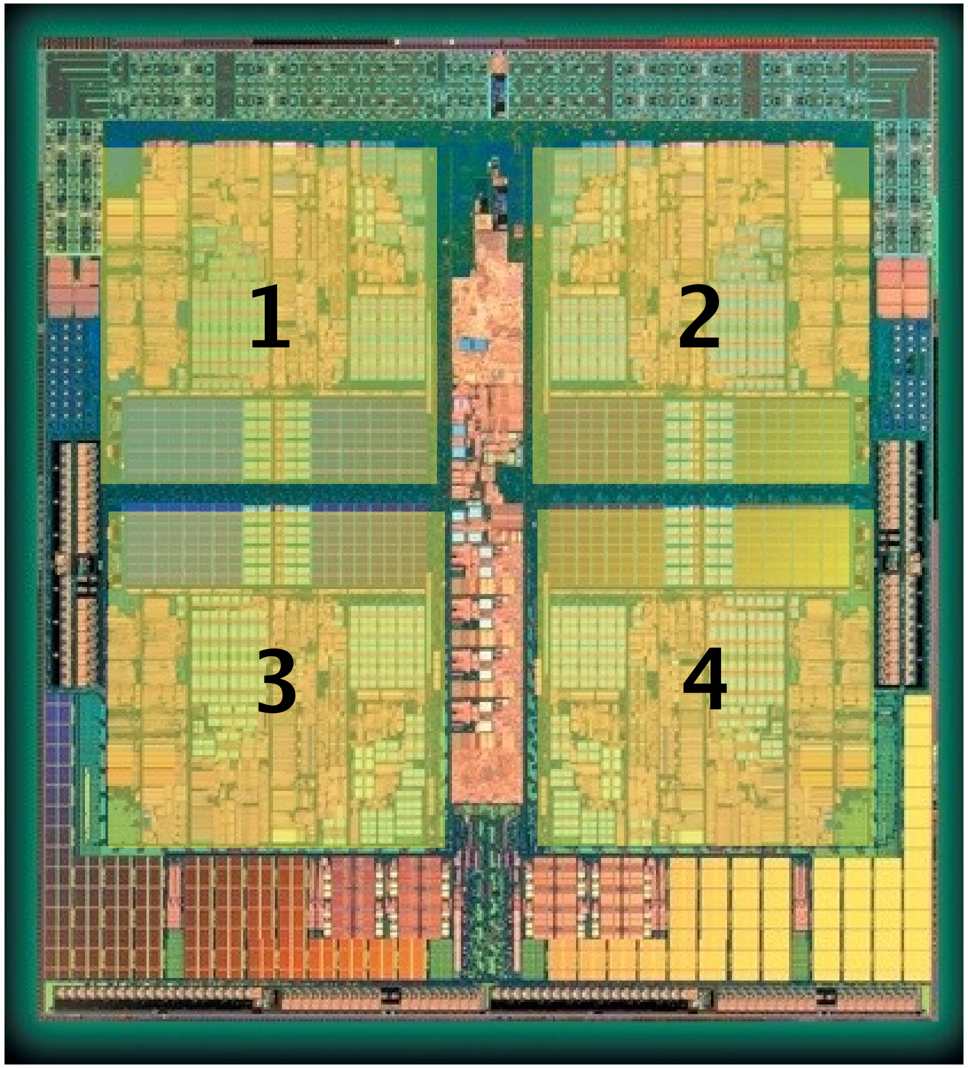
- Symmetric Multi-Processor (SMP)
- Expensive cache domain transfer

- Multiple of single-socket memory bandwidth

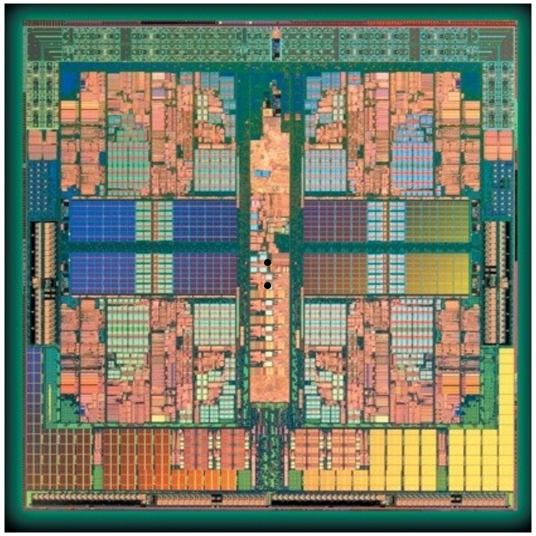
SMP Usage

- Ideal usage:
  - Multiple processes
    - Unix model: many small programs
    - Multi-process application
  - Use IPC or explicit shared memory
- Instead:
  - Multi-threaded, share-everything model
  - Problems:
    - False sharing
    - More synchronization requirements
    - Inadvertent changes
    - Not robust (one thread dies → entire process dies)

Multi-Core

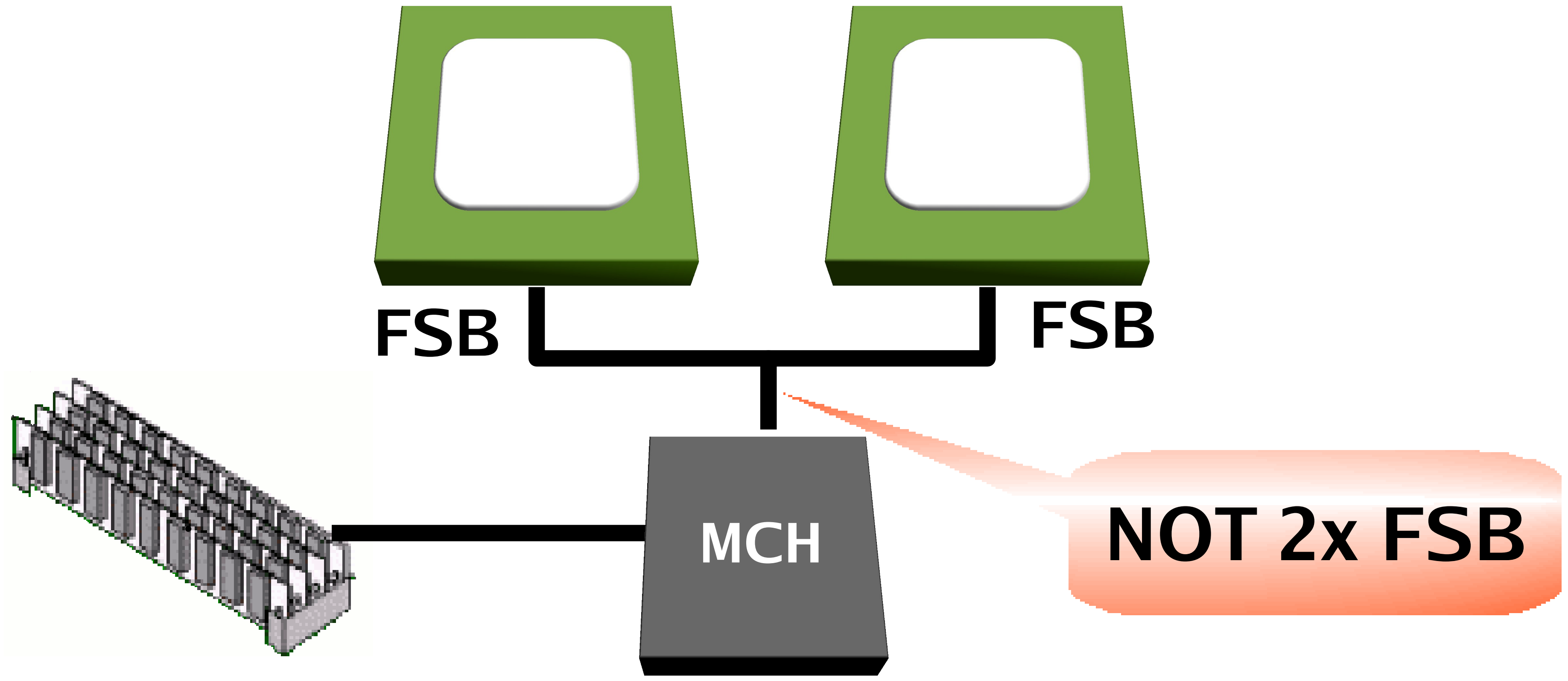


Multi-Core Programming



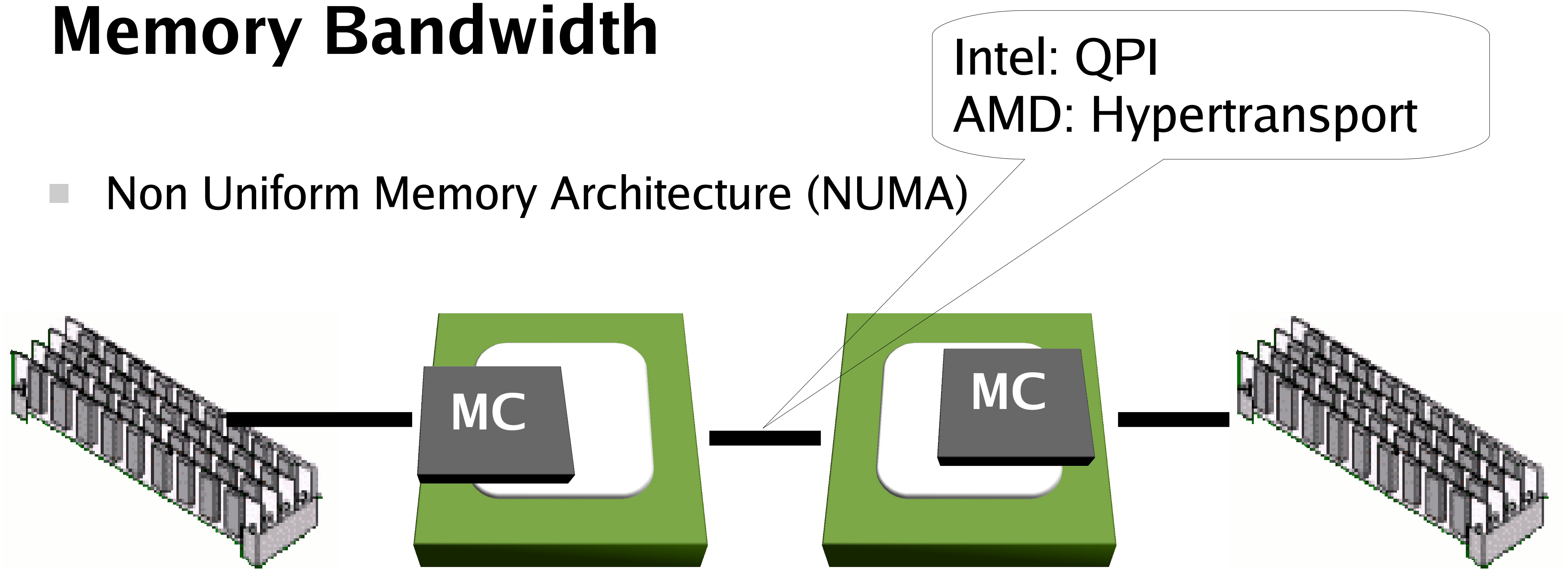
- Shared caches
- Faster cache line transfer between domains
- Deeper cache hierarchy
- Advantages:
  - Faster sharing of cache lines
  - Can be of advantage in closely collaborating code
- Disadvantages:
  - Cache size split between processes and threads with non-overlapping working set
  - Shared bandwidth to RAM

Memory Bandwidth



Memory Bandwidth

- Non Uniform Memory Architecture (NUMA)

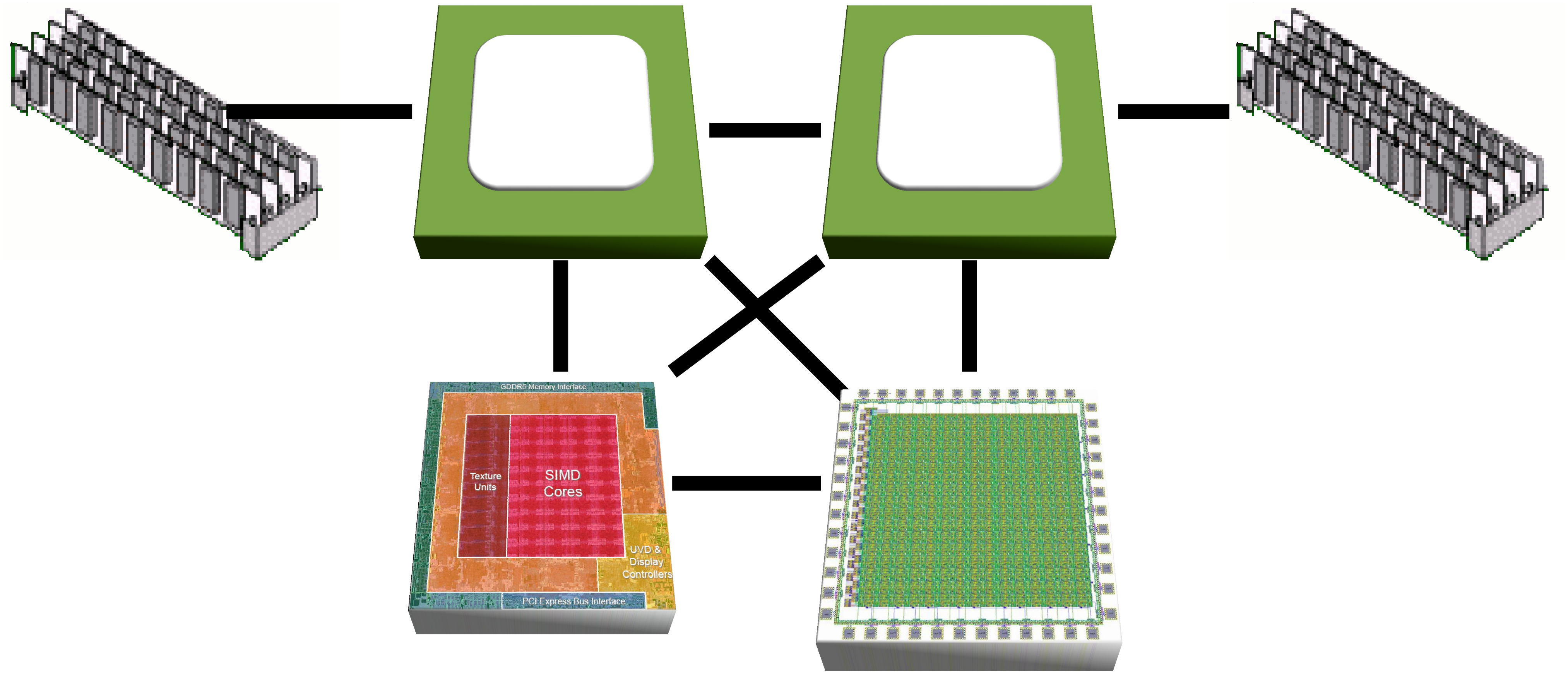


- Memory Controller in CPU
- High-Speed Interconnect between CPUs
- Increased total capacity

Aside from General Purpose CPUs

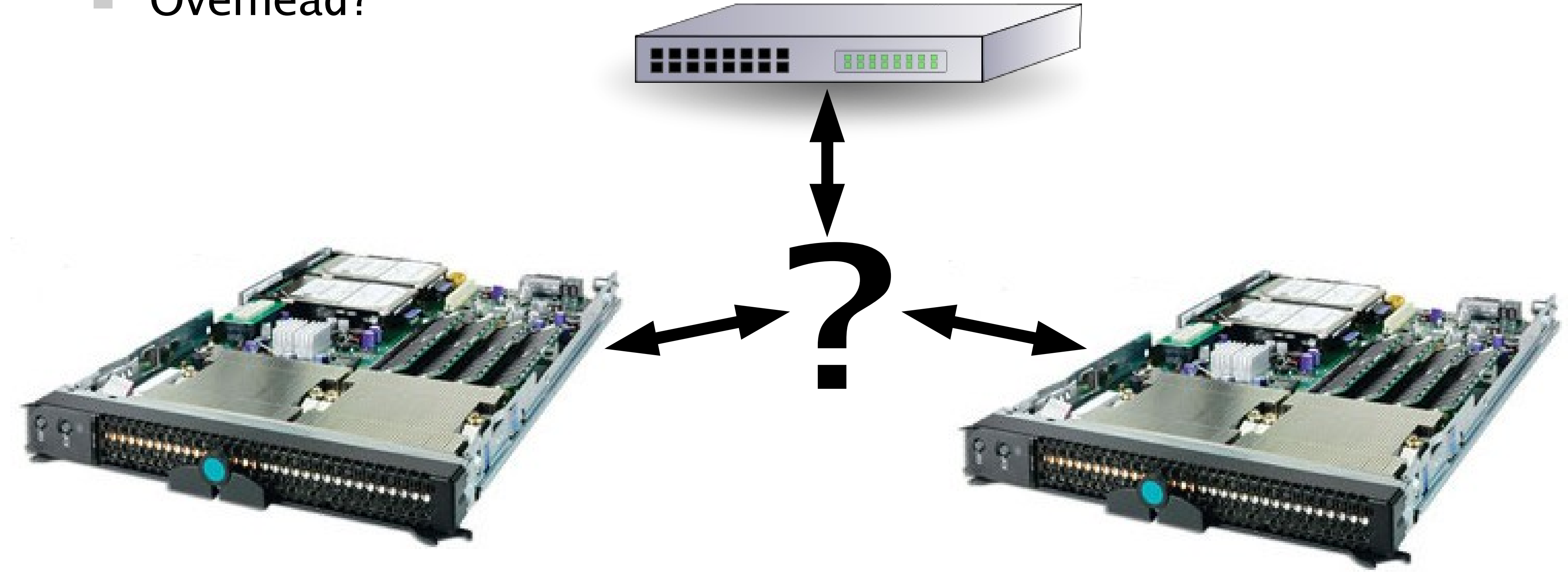
- Co-processors are coming back
  - Intel Geneseo, AMD Torrenza
  - IBM Cell, Intel Larrabee, AMD Fusion
- Huge performance advantage through specialization:
  - All purpose CPU: 50-60 GFLOPS
  - Cell: 210 GFLOPS
  - NVidia GPU: 500 GFLOPS
- Better energy efficiency:
  - FPGA: 1/10th of the energy, potentially 100x faster
  - With appropriate power control:
    - GPGPUs: 100% to 300% of energy
    - Translates to 10% to 50% of cost per GFLOPS

Connecting Co-Processors



Machine Interconnects

- How to connect?
- Direct connections?
- Overhead?



Machine Interconnects

- Requirement:
  - High bandwidth and low latency
- Uses:
  - Traditional network (socket, network filesystem)
  - Message passing (MPI, AMQP)
- Ideal: zero-copy
- Solutions:
  - RDMA over Infiniband
  - Soon: RDMA over Ethernet
- Red Hat solutions using RDMA for
  - MPI, AMQP
  - Future: NFS

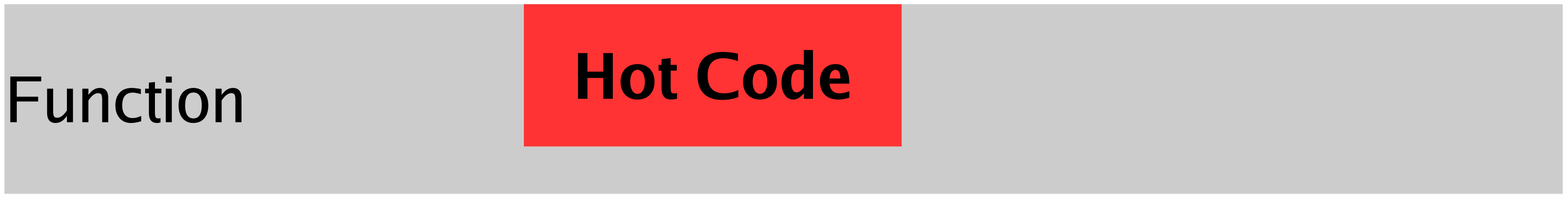
Problems of HPC Programming

- Wide range of hardware features to exploit
  - Not likely to be smaller in future
- Different hardware in same environment
- Working sets growing
- Per-core performance not growing (as much)
- Cache and memory hierarchy getting deeper

Expertise needed for high performance

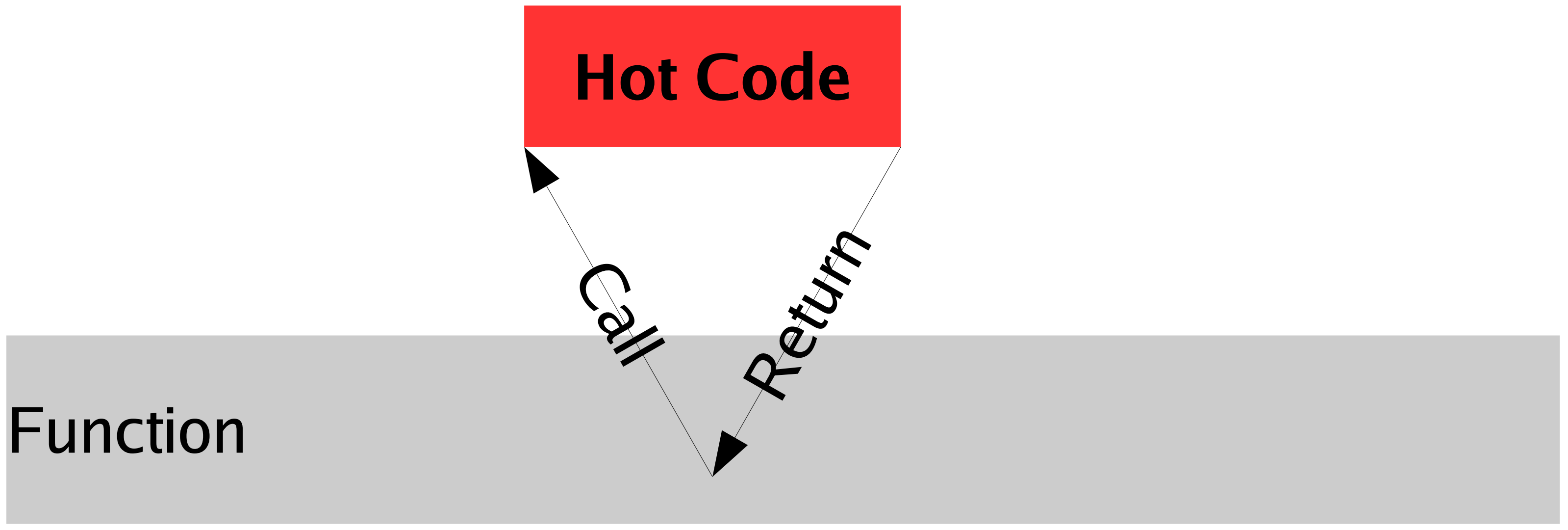
More than ever

- Structuring program important
  - Recognize building blocks
  - Implement in library functions
  - Optimize, if necessary, by experts



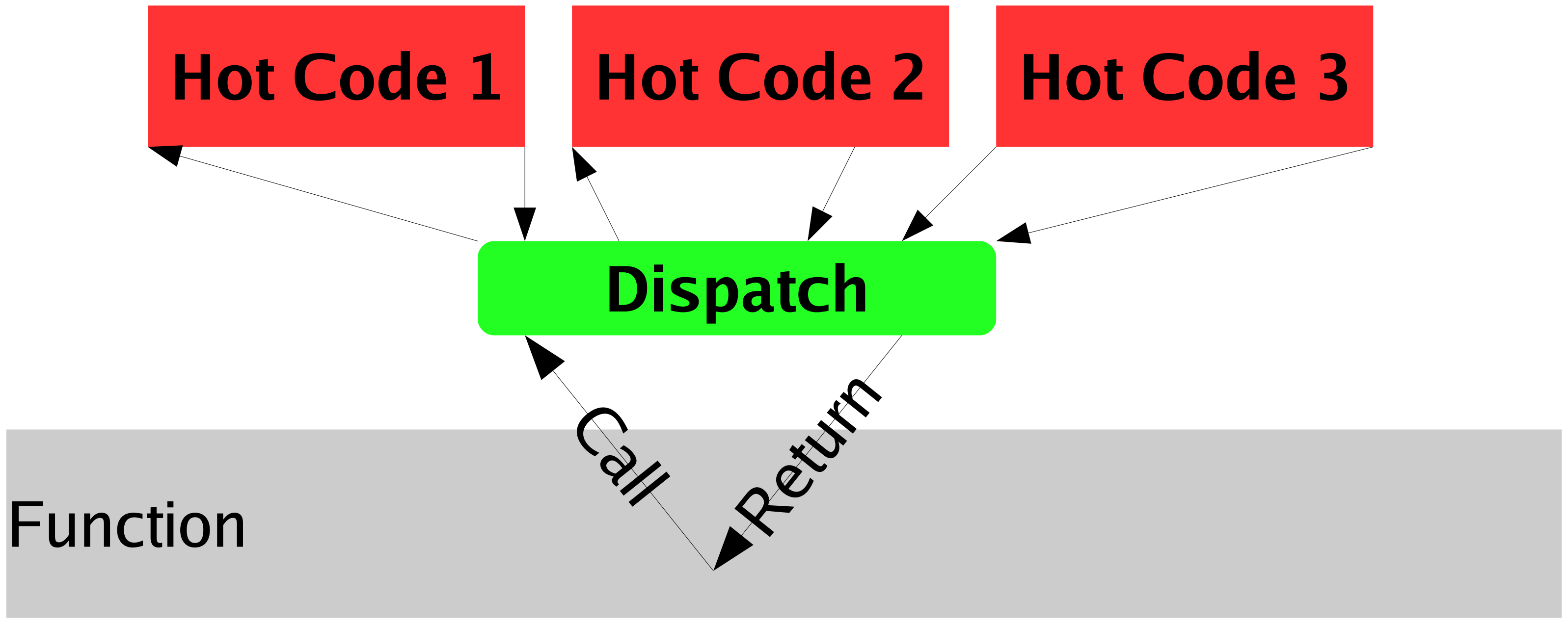
More than ever

- Structuring program important
  - Recognize building blocks
  - Implement in library functions
  - Optimize, if necessary, by experts



More than ever

- Structuring program important
  - Recognize building blocks
  - Implement in library functions
  - Optimize, if necessary, by experts



Dispatching

- Explicit:

Dispatch:

```
  if (feature1) hotcode1()
  else if (feature2) hotcode2()
  else hotcode3()
```

  - e.g., using `cpuid` instruction
  - efficient using STT_IFUNC (indirection function)
    - Overhead only during first call
- Implicit
  - Separate symbol table for hardware configurations
  - Different (sub-)shared objects

Problems with Hot Code Isolation

- Code has to be kept minimal
  - Isolate from program logic
  - Increase reusability
- But not too small

What is “too small”?

- Result:
  - Loss of abstraction
  - Additional overhead

Unavoidable?

- Ideal
  - Algorithms and data structures defined independently

Example: Too Small

- Matrix: represented as array of arrays
- Multiplication: L x M ⊙ M x N
  - Represent transposed M x N matrix
  - Compute L x N vector product
  - Optimize vector products
- Problem:
  - Horrible cache locality
  - Factor 10 slower than necessary
- Correct:
  - Matrix data structure continuous 2-dimensional array
  - Optimize matrix multiplication as a whole
  - Exception: sparse matrix

Loss of Abstraction?

- Necessary to think memory representation all the time?
  - Allocation and deallocation of temporaries sloooow
  - Possible vector arithmetic:

```
vec1 = allocfillvec(n, file1);
vec2 = allocfillvec(n, file2);
multscalar(vec1, sc);
addvec(vec1, vec2);
output(vec1)
```

  - Not as readable as

```
cout << vec1 * sc + vec2
```

Loss of Abstraction?

- ISO C++0x (ehm, ISO C++1x) has solution:
  - rvalue references

```
vec &&operator+(vec &&l, vec &r) {
   for (size_t i=0; i<N; ++i) l.e[i] += r.e[i];
   return l;
}
vec &&operator*(vec &&l, float f) {
   for (size_t i=0; i<N; ++i) l.e[i] *= f;
   return l;
}
```

Reuse of temporary

Loss of Abstraction?

- Data structures design
  - Most cache efficient or
  - Most compact
- Algorithm design
  - Take execution unit operations into account
  - “Complicate” algorithm if of advantage for hardware
  - API should remain unchanged

Algorithm Design

- Example: use fused multiply-add
- Solution: delay operation

```
scaledvec &&operator*(vec &v, float f) {
   return new scaledvec(vm f);
}
vec &&operator+(scaledvec &&sc, vec &&r) {
   for (size_t i=0; i<N; ++i)
      r.e[i] = fma(sc.v.e[i], sc.f, r.e[i]);
   return r;
}
vec &&operator+(vec &&l, vec &r) { ... }
```

Additional Overhead?

- Can be avoided for memory handling
- How about introduction of parallelism?
  - Before:
    - ① create N threads
    - ② multiply matrix A and B, producing C
      split work in N pieces, each executed in one thread
    - ③ multiply matix C and D, producing E
      split work in N pieces, each executed in one thread
    - ④ dispose of threads
  - When matrix multiplication algorithm used, the implementation must create and dispose threads
  - Result: overhead

Additional Overhead?

- Not necessarily: use OpenMP
  - After:
    - ① start parallel region
    - ② C = A * B
    - ③ E = C * D
    - ④ end parallel region
  - `operator*()` uses OpenMP parallel construct

OpenMP Parallelism

- Parallel region
  - Introduces pool of parallelism
  - By default automatically throttled
- Parallel operation
  - **`for`** loops (integers or random-access iterators)
  - Parallel sections
- Parallel operations use up parallelism from ***dynamically*** enclosing parallel region
- Parallelism of parallel region kept around in thread pool

Advantages of OpenMP Parallelism

- Amount of parallelism throttled process-wide
  - As opposed to explicit threads (nested thread creation)
- Use of parallelism independent of creation
  - Parallel matrix multiplication used in non-OpenMP code (without enclosing parallel region) causes one thread to be used
- Fine grained parallelism realistic
  - Amortization of thread creation cost
  - On Linux: very low synchronization costs

- Nice use: automatic parallelization of ISO C++ library (gcc 4.3)
  - Even better with C++0x concepts

Pervasive Parallelism

- Multi-core and/or SMP require parallelism everywhere
- OpenMP drastically reduces complexity
  - Programmer focus is expressing parallelism, not implementing
  - Compiler can help locating problems
- Problem with parallel programming (Amdahl's Law):

$$T=\frac{1}{(1-P)+\frac{P}{S}}$$

P: parallelizable fraction, S: number execution units

- P is reduced due to synchronization requirements

The Synchronization Problem

- Foremost:: needed for correctness
  - Often insurmountable problem for novices
  - Hard to debug
- Two extreme approaches:
  - Coarse grained: easy to use, little overhead, potentially large reduction of P
  - Fine grained: hard to get right, high(er) overhead
- And anything in between

Different approach needed

Transactional Memory

- Inspiration: database programming
  - Read and write access to multiple tables atomic
- Transferred to C/C++ programming:
  - Read and assignments of memory locations atomic
- Language extension:

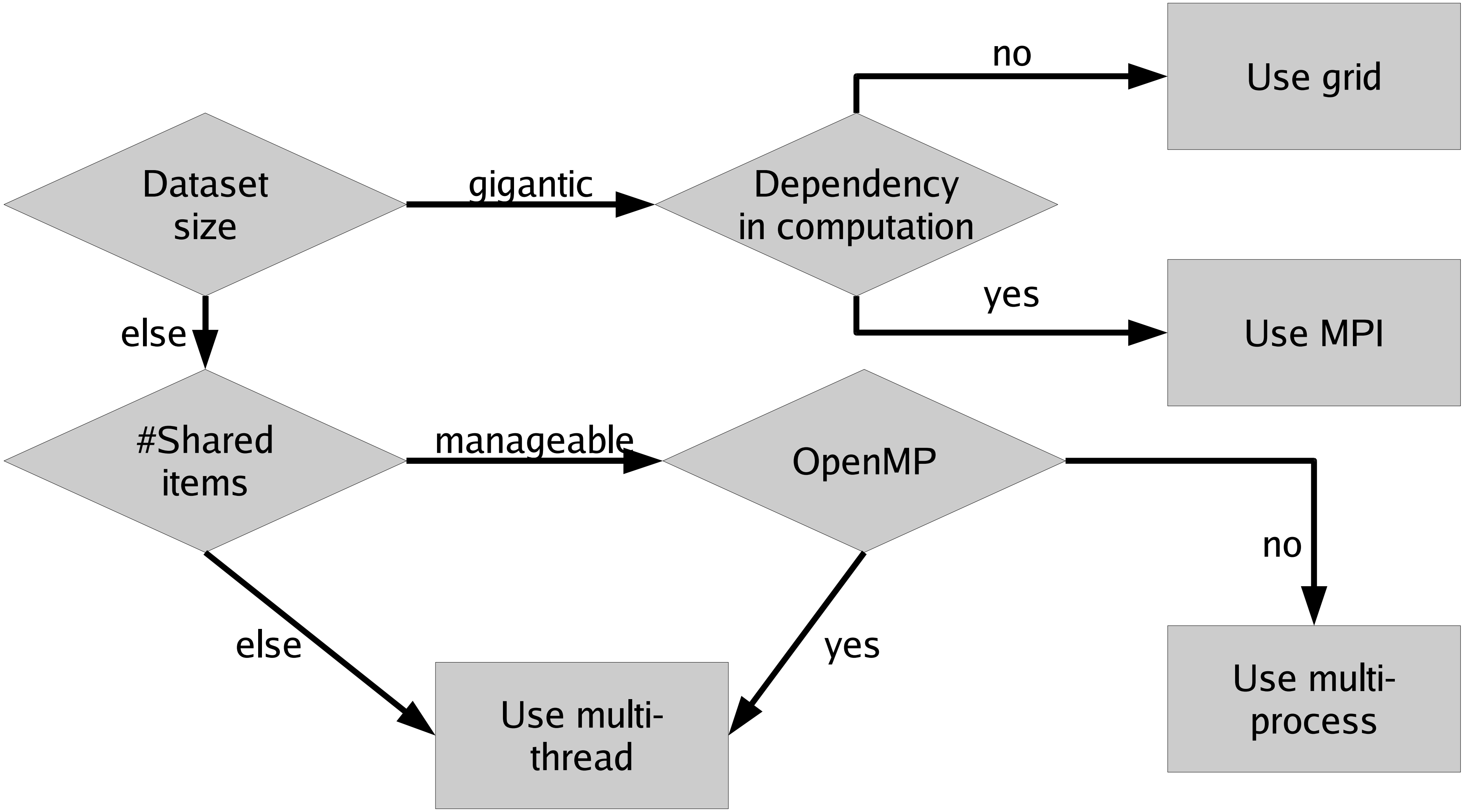
```
__tm_atomic {
    if (a > b) { a -= b; ++c; }
}
```

  - Compiler recognizes memory accesses
  - Optimistic execution, rollback on failure

Transactional Memory

- Significantly simplifies synchronization
- Can utilize upcoming hardware support
- Slowdown manageable in many/most cases
  - Still research topic (Red Hat actively participating)
- Ideally increases parallelism (dramatically)
  - Example: hash table
  - Coarse grained severe limitation (hashes meant to spread accesses)
  - Fine grained locking: high overhead (time and space)
  - TM: optimistic execution mostly succeeds
    - No significant overhead, no limit in parallelism

Parallelization Models



Multi-Process

- Similar to multi-thread
- No automatically shared address space
- POSIX shared memory to explicitly share data
- Advantages:
  - No inadvertent sharing, corruption
  - No accidental false sharing
  - Better suited for execution on different sockets
  - Fault isolation (only one process dies)
    - Robust mutexes can help recovering

MPI

- De-facto standard
- Library support for synchronization and data exchange
- Highly optimized implementations available
  - Optimized intra-socket communication/syncronization
  - Use of advanced network technology
    - Special interconnects
      - Infiniband, Quadrics, ...
    - RDMA
      - In future over Ethernet as well
- Given good network hardware
  - High bandwidth, low latency communication/synchronization

Grid

- Schedule execution for potentially huge number of execution units
- Tasks require not much communication for run
  - Input, process, output
- Can use unreliable resources
  - Just restart task
- Different programs can be executed and controlled in parallel
- Grid scheduler responsible for operation of entire grid
  - Highly customizable (Condor part of Red Hat's MRG product)

Summary

1. Select best suited parallelization model
2. Design data structures
   1. Ideally reusable
3. Design operators for data structures
   1. Learn functional programming
   2. Use Haskell, Ocaml, or…
   3. Program functional in C++
      1. ISO C++ library a good start
      2. C++0x introduces lambda etc
   4. Meta programming for reusability
      1. Concepts allow expressing optimization possibilities
4. Profile
   1. Call in expert to write optimized version of algorithm

Questions?

drepperl@redhat.com | people.redhat.com/drepper