Parallel Programming Concepts

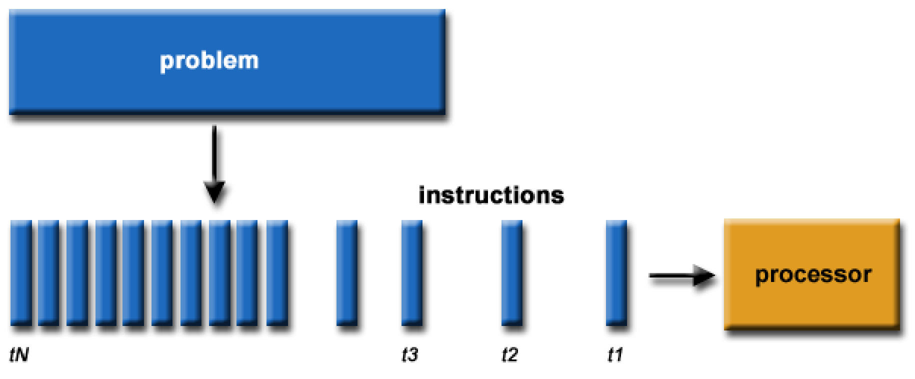
Alexander B. Pacheco
LTS Research Computing
December 8, 2016

Outline

Introduction

What is Serial Computing?

- Traditionally, software has been written for serial computation:
  - A problem is broken into a discrete series of instructions
  - Instructions are executed sequentially one after another
  - Executed on a single processor
  - Only one instruction may execute at any moment in time

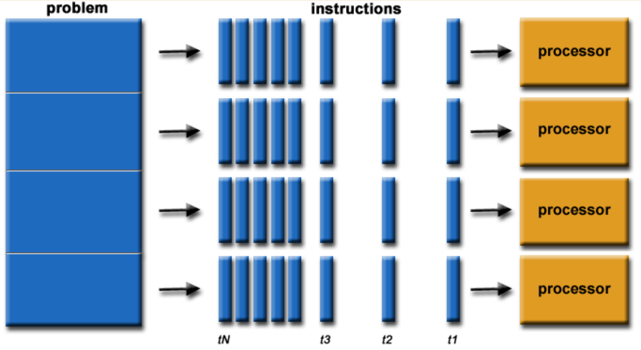


What is Parallel Computing?

- In the simplest sense, parallel computing is the simultaneous use of multiple compute resources to solve a computational problem:
    - A problem is broken into discrete parts that can be solved concurrently
    - Each part is further broken down to a series of instructions
    - Instructions from each part execute simultaneously on different processors
    - An overall control/coordination mechanism is employed
    - The computational problem should be able to:
        - Be broken apart into discrete pieces of work that can be solved simultaneously;
        - Execute multiple program instructions at any moment in time;
        - Be solved in less time with multiple compute resources than with a single compute resource.
    - The compute resources are typically:
        - A single computer with multiple processors/cores
        - An arbitrary number of such computers connected by a network

Why Parallel Computing?

- Parallel computing might be the only way to achieve certain goals
  - Problem size (memory, disk etc.)
  - Time needed to solve problems
- Parallel computing allows us to take advantage of ever-growing parallelism at all levels
  - Multi-core, many-core, cluster, grid, cloud, $\cdots$

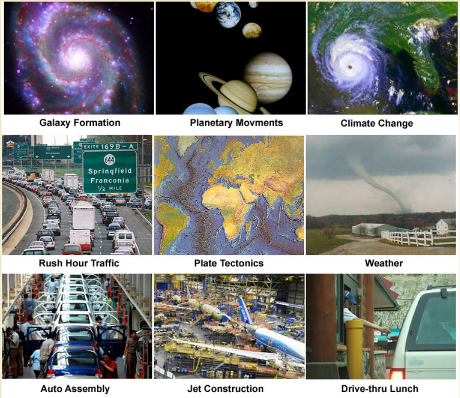


What are Parallel Computers?

- Virtually all stand-alone computers today are parallel from a hardware perspective:
  - Multiple functional units (L1 cache, L2 cache, branch, prefetch, decode, floating-point, graphics processing (GPU), integer, etc.)
  - Multiple execution units/cores
  - Multiple hardware threads
  - Networks connect multiple stand-alone computers (nodes) to make larger parallel computer clusters.

Why Use Parallel Computing? I

- The Real World is Massively Parallel:
    - In the natural world, many complex, interrelated events are happening at the same time, yet within a temporal sequence.
    - Compared to serial computing, parallel computing is much better suited for modeling, simulating and understanding complex, real world phenomena.



Why Use Parallel Computing? II

- SAVE TIME AND/OR MONEY:
  - In theory, throwing more resources at a task will shorten its time to completion, with potential cost savings.
  - Parallel computers can be built from cheap, commodity components.
- SOLVE LARGER / MORE COMPLEX PROBLEMS:
  - Many problems are so large and/or complex that it is impractical or impossible to solve them on a single computer, especially given limited computer memory.
  - Example: "Grand Challenge Problems" (en.wikipedia.org/wiki/Grand_Challenge) requiring PetaFLOPS and PetaBytes of computing resources.
  - Example: Web search engines/databases processing millions of transactions every second
- PROVIDE CONCURRENCY:
  - A single compute resource can only do one thing at a time. Multiple compute resources can do many things simultaneously.
  - Example: Collaborative Networks provide a global venue where people from around the world can meet and conduct work "virtually".

Why Use Parallel Computing? III

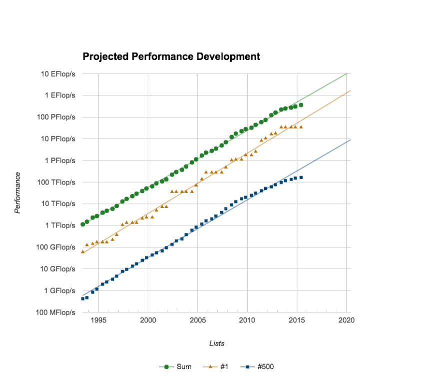
- TAKE ADVANTAGE OF NON-LOCAL RESOURCES:
  - Using compute resources on a wide area network, or even the Internet when local compute resources are scarce or insufficient.
  - Example: SETI@home (setiathome.berkeley.edu) over 1.5 million users in nearly every country in the world. Source: www.boincsynergy.com/stats/ (June, 2015).
  - Example: Folding@home (folding.stanford.edu) uses over 160,000 computers globally (June, 2015)
- MAKE BETTER USE OF UNDERLYING PARALLEL HARDWARE:
  - Modern computers, even laptops, are parallel in architecture with multiple processors/cores.
  - Parallel software is specifically intended for parallel hardware with multiple cores, threads, etc.
  - In most cases, serial programs run on modern computers ”waste” potential computing power.

Why Use Parallel Computing? IV

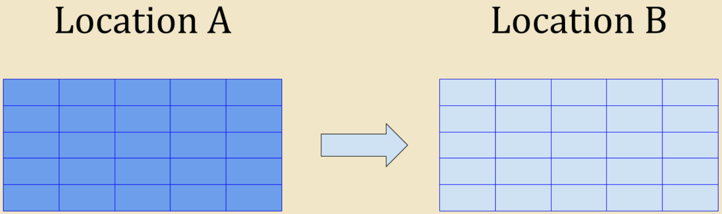
- The Future:

- During the past 20+ years, the trends indicated by ever faster networks, distributed systems, and multi-processor computer architectures (even at the desktop level) clearly show that parallelism is the future of computing.
- In this same time period, there has been a greater than 500,000x increase in supercomputer performance, with no end currently in sight.
- The race is already on for Exascale Computing!

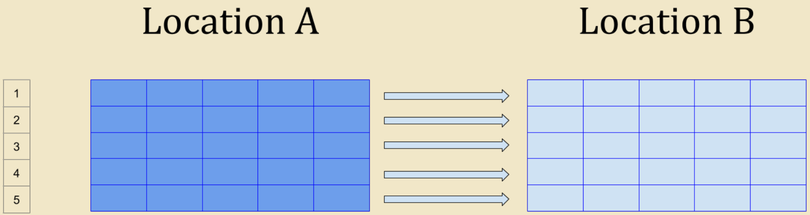
  Exaflop $= 10^{18}$ calculations per second



- Consider an example of moving a pile of boxes from location A to location B
- Lets say, it takes x mins per box. Total time required to move the boxes is 25x.
- How do you speed up moving 25 boxes from Location A to Location B?



- You enlist more people to move the boxes.
- If 5 people move the boxes simultaneously, it should theoretically take 5x mins to move 25 boxes.

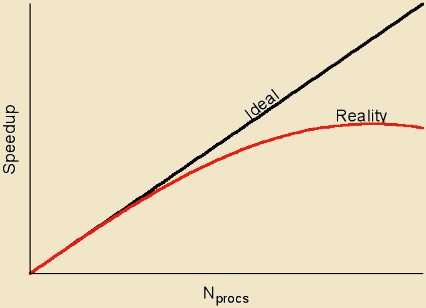


Evaluating Parallel Programs

- Speedup
  - Let N_{Proc} be the number of parallel processes
  - $\text{Speedup}(N_{\text{Proc}}) = \dfrac{\text{Time used by best serial program}}{\text{Time used by parallel program}}$
  - Speedup is usually between 0 and N_{Proc}
- Efficiency
  - $\text{Efficiency}(N_{\text{Proc}}) = \dfrac{\text{Speedup}(N_{\text{Proc}})}{N_{\text{Proc}}}$
  - Efficiency is usually between 0 and 1

Speedup as a function of N_{Proc}

- Ideally
  - The speedup will be linear
- Even better
  - (in very rare cases) we can have superlinear speedup
- But in reality
  - Efficiency decreases with increasing number of processes



Amdahl's Law

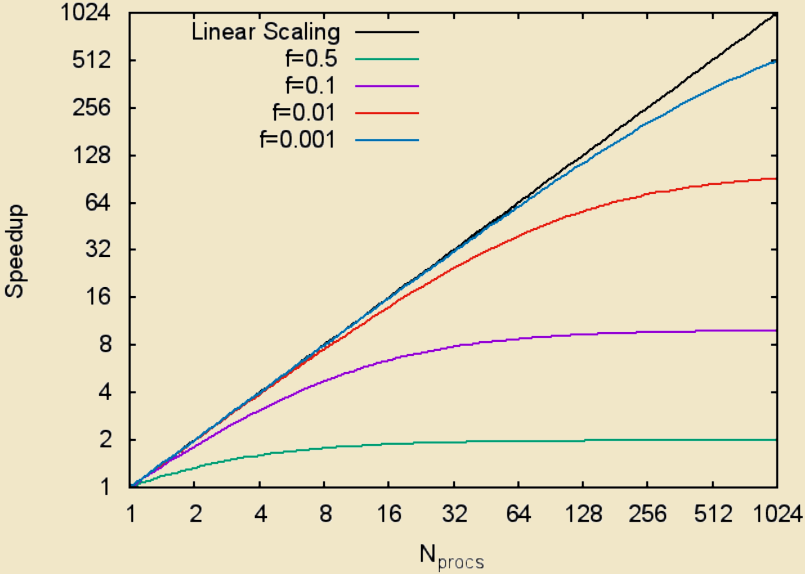
- Let f be the fraction of the serial program that cannot be parallelized
- Assume that the rest of the serial program can be perfectly parallelized (linear speedup)

$$\text{Time}_{\text{parallel}} = \text{Time}_{\text{serial}} \cdot \left(f + \frac{1-f}{N_{\text{proc}}} \right)$$

- Or

$$\text{Speedup} = \frac{1}{f + \dfrac{1-f}{N_{\text{proc}}}} \leq \frac{1}{f}$$

Maximal Possible Speedup



Amdahl's Law

- What Amdahl's law says
  - It puts an upper bound on speedup (for a given f), no matter how many processes are thrown at it
- Beyond Amdahl's law
  - Parallelization adds overhead (communication)
  - f could be a variable too
    - It may drop when problem size and N_{proc} increase
  - Parallel algorithm is different from the serial one

Writing a parallel program step by step

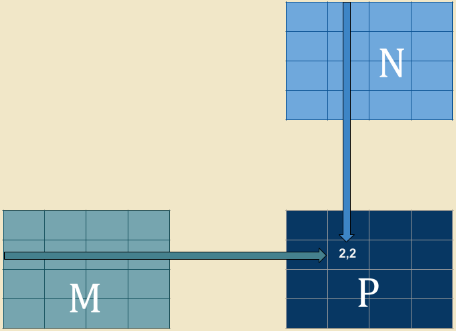
1. Start from serial programs as a baseline
   - Something to check correctness and efficiency against
2. Analyze and profile the serial program
   - Identify the "hotspot"
   - Identify the parts that can be parallelized
3. Parallelize code incrementally
4. Check correctness of the parallel code
5. Iterate step 3 and 4

A REAL example of parallel computing

- Dense matrix multiplication $M_{md} \times N_{dn} = P_{mn}$

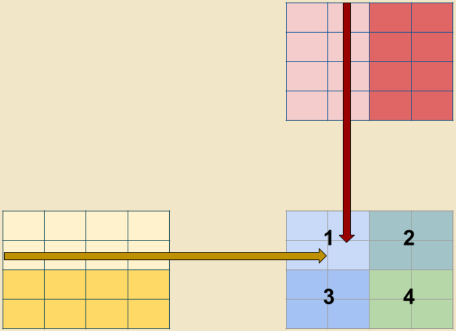
$$P_{m,n} = \sum_{k=1}^{d} M_{m,k} \times N_{k,n}$$

$$P_{2,2} = M_{2,1} * N_{1,2} + M_{2,2} * N_{2,2} + M_{2,3} * N_{3,2} + M_{2,4} * N_{4,2}$$



Parallelizing matrix multiplication

- Divide work among processors
- In our 4x4 example
    - Assuming 4 processors
    - Each responsible for a 2x2 tile (submatrix)
    - Can we do 4x1 or 1x4?



Pseudo Code

Serial

```
for i = 1, 4
  for j = 1, 4
    for k = 1, 4
       P(i,j) += M(i,k)*N(k,j);
```

Parallel

```
for i = istart, iend
  for j = jstart, jend
    for k = 1, 4
       P(i,j) += M(i,k)*N(k,j);
```

```python
m __future__ import division

import numpy as np
from mpi4py import MPI
from time import time

#=============================================================================#

my_N = 3000
my_M = 3000

#=============================================================================#

NORTH = 0
SOUTH = 1
EAST = 2
WEST = 3



def pprint(string, comm=MPI.COMM_WORLD):
    if comm.rank == 0:
        print(string)


if __name__ == "__main__":
    comm = MPI.COMM_WORLD

    mpi_rows = int(np.floor(np.sqrt(comm.size)))
    mpi_cols = comm.size // mpi_rows
    if mpi_rows*mpi_cols > comm.size:
        mpi_cols -= 1
    if mpi_rows*mpi_cols > comm.size:
        mpi_rows -= 1

    pprint("Creating a %d x %d processor grid..." % (mpi_rows, mpi_cols) )

    ccomm = comm.Create_cart( (mpi_rows, mpi_cols), periods=(True, True), reorder=True)
```

```python
my_mpi_row, my_mpi_col = ccomm.Get_coords( ccomm.rank )
neigh = [0,0,0,0]

neigh[NORTH], neigh[SOUTH] = ccomm.Shift(0, 1)
neigh[EAST],  neigh[WEST]  = ccomm.Shift(1, 1)


# Create matrices
my_A = np.random.normal(size=(my_N, my_M)).astype(np.float32)
my_B = np.random.normal(size=(my_N, my_M)).astype(np.float32)
my_C = np.zeros_like(my_A)

tile_A = my_A
tile_B = my_B
tile_A_ = np.empty_like(my_A)
tile_B_ = np.empty_like(my_A)
req = [None, None, None, None]

t0 = time()
for r in xrange(mpi_rows):
    req[EAST]  = ccomm.Isend(tile_A , neigh[EAST])
    req[WEST]  = ccomm.Irecv(tile_A_, neigh[WEST])
    req[SOUTH] = ccomm.Isend(tile_B , neigh[SOUTH])
    req[NORTH] = ccomm.Irecv(tile_B_, neigh[NORTH])

    #t0 = time()
    my_C += np.dot(tile_A, tile_B)
    #t1 = time()

    req[0].Waitall(req)
    #t2 = time()
    #print("Time computing %6.2f  %6.2f" % (t1-t0, t2-t1))
comm.barrier()
t_total = time()-t0

t0 = time()
np.dot(tile_A, tile_B)
t_serial = time()-t0
```

```python
pprint(78*"=")
pprint("Computed (serial) %d x %d x %d in  %6.2f seconds" % (my_M, my_M, my_N, t_serial))
pprint(" ... expecting parallel computation to take %6.2f seconds" % (mpi_rows*mpi_rows*mpi_cols*
      t_serial / comm.size))
pprint("Computed (parallel) %d x %d x %d in        %6.2f seconds" % (mpi_rows*my_M, mpi_rows*my_M,
      mpi_cols*my_N, t_total))


#print "[%d] (%d,%d): %s" % (comm.rank, my_mpi_row, my_mpi_col, neigh)

comm.barrier()
```

Parallel programming models

Single Program Multiple Data (SPMD)

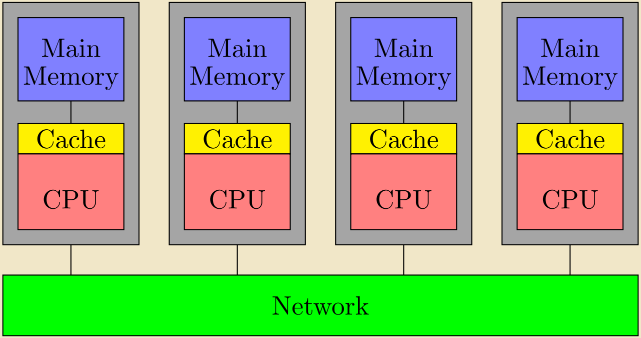
- All program instances execute same program
- Data parallel - Each instance works on different part of the data
- The majority of parallel programs are of this type
- Can also have
  - SPSD: serial program
  - MPSD: rare
  - MPMD

Memory system models

- Different ways of sharing data among processors
  - Distributed Memory
  - Shared Memory
  - Other memory models
    - Hybrid model
    - PGAS (Partitioned Global Address Space)

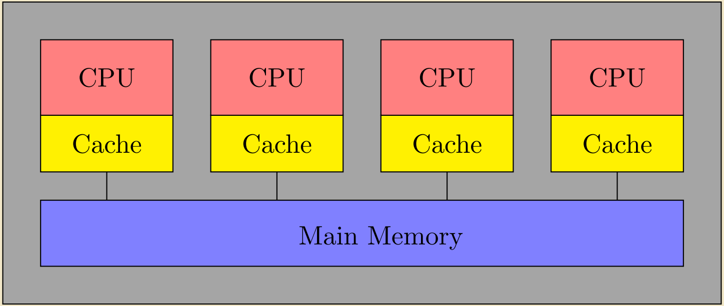
Distributed Memory Model

- Each process has its own address space
  - Data is local to each process
- Data sharing is achieved via explicit message passing
- Example
  - MPI



Shared Memory Model

- All threads can access the global memory space.
- Data sharing achieved via writing to/reading from the same memory location
- Example
  - OpenMP
  - Pthreads



Shared vs Distributed

Shared

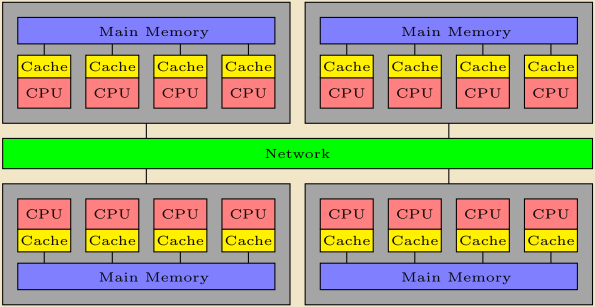
- Pros
  - Global address space is user friendly
  - Data sharing is fast
- Cons
  - Lack of scalability
  - Data conflict issues

Distributed

- Pros
  - Memory scalable with number of processors
  - Easier and cheaper to build
- Cons
  - Difficult load balancing
  - Data sharing is slow

Hybrid model

- Clusters of SMP (symmetric multi-processing) nodes dominate nowadays
- Hybrid model matches the physical structure of SMP clusters
  - OpenMP within nodes
  - MPI between nodes



Potential benefits of hybrid model

- Message-passing within nodes (loopback) is eliminated
- Number of MPI processes is reduced, which means
  - Message size increases
  - Message number decreases
- Memory usage could be reduced
  - Eliminate replicated data
- Those are good, but in reality, (most) pure MPI programs run as fast (sometimes faster than) as hybrid ones $\cdots$

Reasons why NOT to use hybrid model

- Some (most?) MPI libraries already use internally different protocols
  - Shared memory data exchange within SMP nodes
  - Network communication between SMP nodes
- Overhead associated with thread management
  - Thread fork/join
  - Additional synchronization with hybrid programs

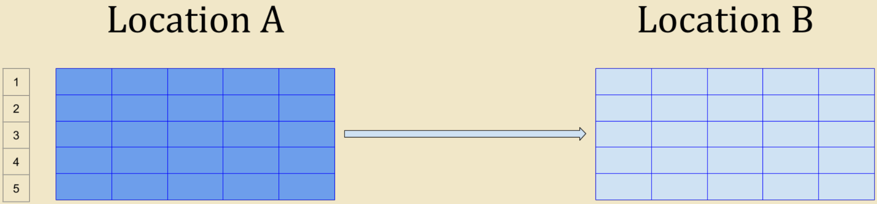
Parallel programming hurdles

Parallel Programming Hurdles

- Hidden serializations
- Overhead caused by parallelization
- Load balancing
- Synchronization issues

Hidden Serialization

- Back to our box moving example
- What if there is a very long corridor that allows only one work to pass at a time between Location A and B?



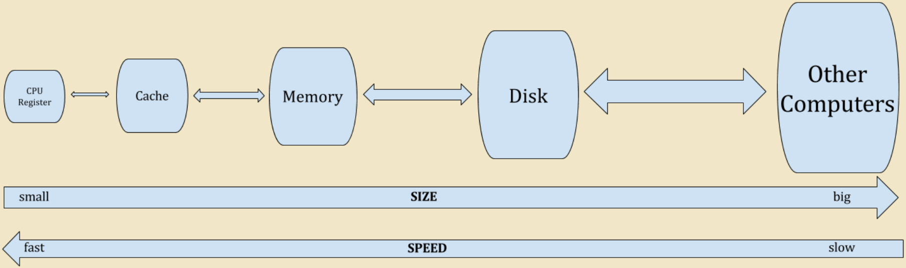
Hidden Serialization

- It is the part in serial programs that is hard or impossible to parallelize
    - Intrinsic serialization (the f in Amdahl's law)
- Examples of hidden serialization:
    - System resources contention, e.g. I/O hotspot
    - Internal serialization, e.g. library functions that cannot be executed in parallel for correctness

Communication overhead

- Sharing data across network is slow
  - Mainly a problem for distributed memory systems
- There are two parts of it
  - Latency: startup cost for each transfer
  - Bandwidth: extra cost for each byte
- Reduce communication overhead
  - Avoid unnecessary message passing
  - Reduce number of messages by combining them

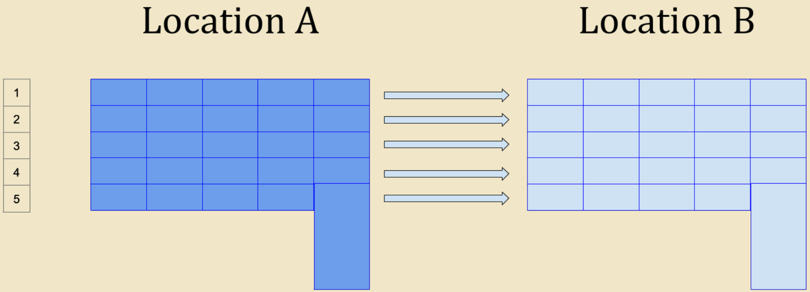
Memory Heirarchy



- Avoid unnecessary data transfer
- Load data in blocks (spatial locality)
- Reuse loaded data (temporal locality)
- All these apply to serial programs as well

Load balancing

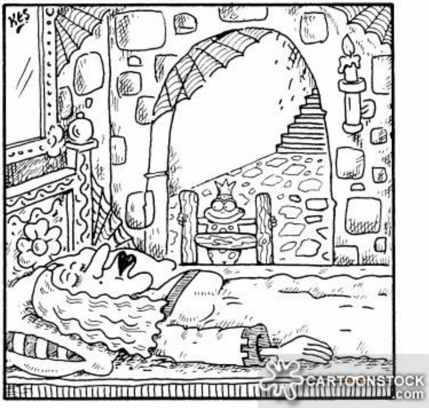
- Back to our box moving example, again
- Anyone see a problem?



Load balancing

- Work load not evenly distributed
  - Some are working while others are idle
  - The slowest worker dominates in extreme cases
- Solutions
  - Explore various decomposition techniques
  - Dynamic load balancing
- Hard for distributed memory
- Adds overhead

Synchronization issues - deadlock



The frog prince figured that as Sleeping Beauty needed a kiss of a handsome prince and he, the kiss of a princess. Why not kill two birds with one stone?

CARTOONSTOCK .com
Search ID: ksm0321

Deadlock

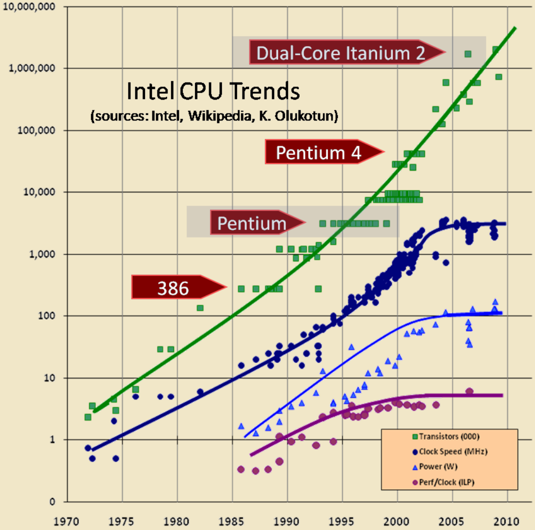
- Often caused by "blocking" communication operations
  - "Blocking" means "I will not proceed until the current operation is over"
- Solution
  - Use "non-blocking" operations
  - Caution: trade-off between data safety and performance

Heterogeneous computing

Heterogeneous computing

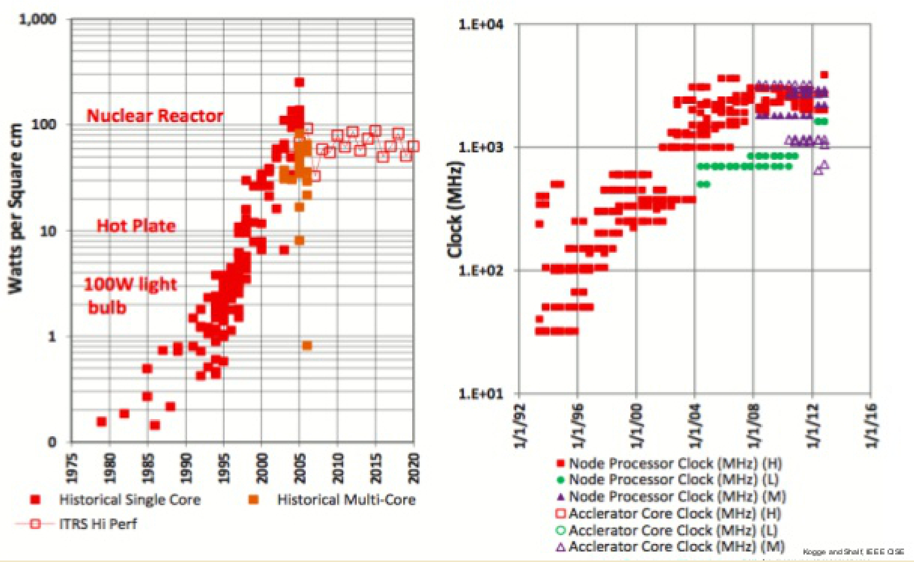
- A heterogeneous system solves tasks using different types of processing units
    - CPUs
    - GPUs
    - DSPs
    - Co-processors
    - ...
- As opposed to homogeneous systems, e.g. SMP nodes with CPUs only

The Free Lunch Is Over



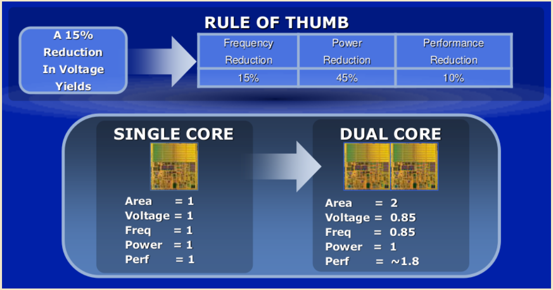
Source: Herb Sutter
http://www.gotw.ca/publications/concurrency-ddj.htm

Power and Clock Speed



Power efficiency is the key

- We have been able to make computer run faster by adding more transistors
  - Moore's law
- Unfortunately, not any more
  - Power consumption/heat generation limits packing density
  - Power $\sim$ speed^2
- Solution
  - Reduce each core's speed and use more cores - increased parallelism



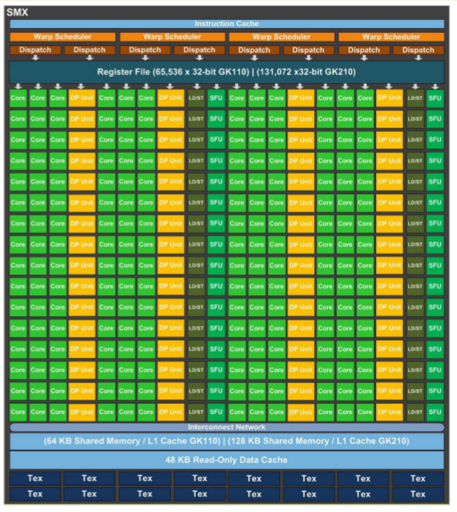
Source: John Urbanic, PSC

Graphic Processing Units (GPUs)

- Massively parallel many-core architecture
  - Thousands of cores capable of running millions of threads
  - Data parallelism
- GPUs are traditionally dedicated for graphic rendering, but become more versatile thanks to
  - Hardware: faster data transfer and more on-board memory
  - Software: libraries that provide more general purposed functions
- GPU vs CPU
  - GPUs are very effectively for certain type of tasks, but we still need the general purpose CPUs

nVIDIA Kepler K80

- Performance:
  - 1.87 TFlops (DP)
  - 5.6 TFlops (SP)
- GPU: 2x GK210
- CUDA Cores: 4992
- Memory (GDDR5): 24GB
- Memory (Bandwidth): 480GBs
- Features
  - 192 SP CUDA Cores
  - 64 DP units
  - 32 Special function units (SFU)
  - 32 load/store units (LD/ST)



GPU programming strategies

- GPUs need to copy data from main memory to its onboard memory and copy them back
  - Data transfer over PCIe is the bottleneck, so one needs to
- Avoid data transfer and reuse data
- Overlap data transfer and computation
- Massively parallel, so it is a crime to do anything antiparallel
  - Need to launch enough threads in parallel to keep the device busy
  - Threads need to access contiguous data
  - Thread divergence needs to be eliminated

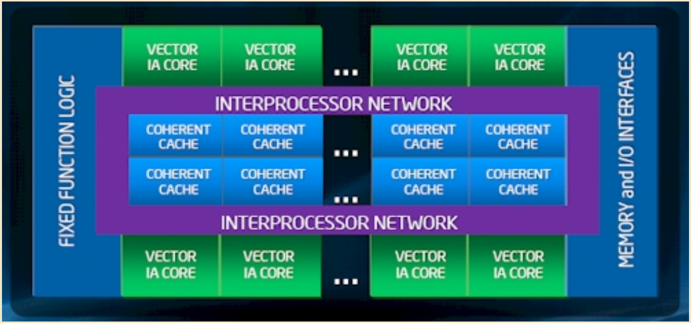
Intel Many Integrated Core Architecture

- Leverage x86 architecture (CPU with many cores)

  X86 cores are simpler, but allow for more compute throughput
- Leverage existing x86 programming models
- Dedicate much of the silicon to floating point ops
- Cache coherent
- Increase floating-point throughput
- Implement as a separate device
- Strip expensive features (out-of-order execution, branch prediction, etc.)
- Widen SIMD registers for more throughput
- Fast (GDDR5) memory on card
- Runs a full Linux operating system (BusyBox)

Intel Xeon Phi 7120P

- Add-on to CPU-based system
- 16 GB memory
- 61 x86 64-bit cores (244 threads)
- single-core 1.2 GHz
- 512-bit vector registers
- 1.208 TFLOPS = 61 cores * 1.238 GHz * 16 DP FLOPs/cycle/core



MICs comparison to GPUs

- Disadvantages
  - Less acceleration
  - In terms of computing power, one GPU beats one Xeon Phi for most cases currently.
- Advantages
  - X86 architecture
  - IP-addressable
  - Traditional parallelization (OpenMP, MPI)
  - Easy programming, minor changes from CPU codes
  - Offload: minor change of source code.
  - New. Still a lot of room for improvement.