Day 3: Parallel Computing

Thomas Hauser
Director of Research Computing
University of Colorado Boulder
thomas.hauser@colorado.edu

Outline

- Topics today
  - Computer Architecture 101
  - Single processor optimization
  - Shared memory parallel programming
  - Programming accelerators
  - Distributed memory parallel programming
- Provide you with
  - Overview and simple guidelines for code development
  - Simple examples
  - Pointers to other materials

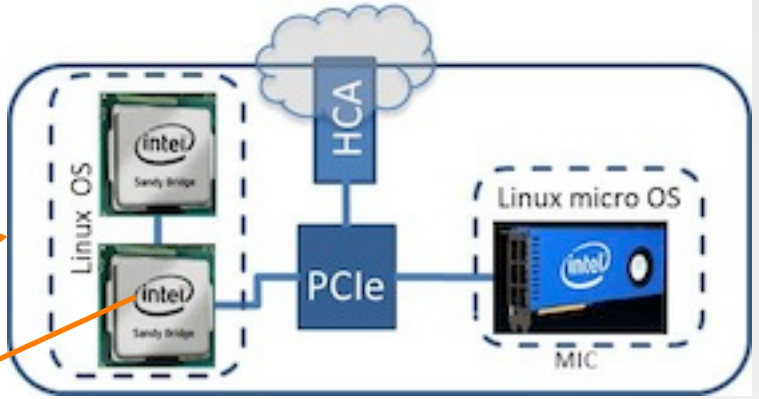
References

1. Hager, G, and G Wellein. 2010. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, Jul 2, 2010 - 356 pages.
2. Levesque, John, and Gene Wagenbreth. 2010. *High Performance Computing*. CRC Press.
3. Stampede User Guide: https://www.tacc.utexas.edu/user-services/user-guides/stampede-user-guide
4. Henry Neeman, Supercomputing in Plain English, A High Performance Computing Workshop Series, http://www.oscer.ou.edu/education.php

Computer Architecture 101

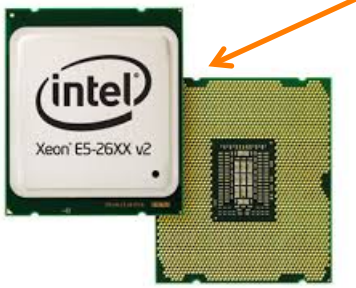
Stampede Supercomputer

Stampede has 6,400 nodes [3]
56 GB/s FDR Infiniband interconnect



Stampede compute node [3]:

- 2 Sockets per Node → 2 Xeon E5 processors
- 1 Xeon Phi coprocessor
- 32 GB Memory
- 250 GB Disk



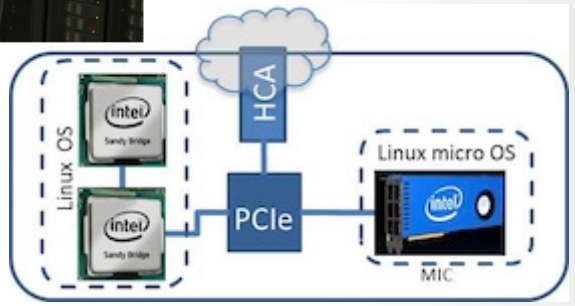
www.scan.co.uk

Socket:

- 2.7 GhZ
- 8 Cores
- 8 DP FP operations per clock cycle
- 64 GB L1 Cache/core
- Vector width: 4 double precision items

Parallelism at All Levels

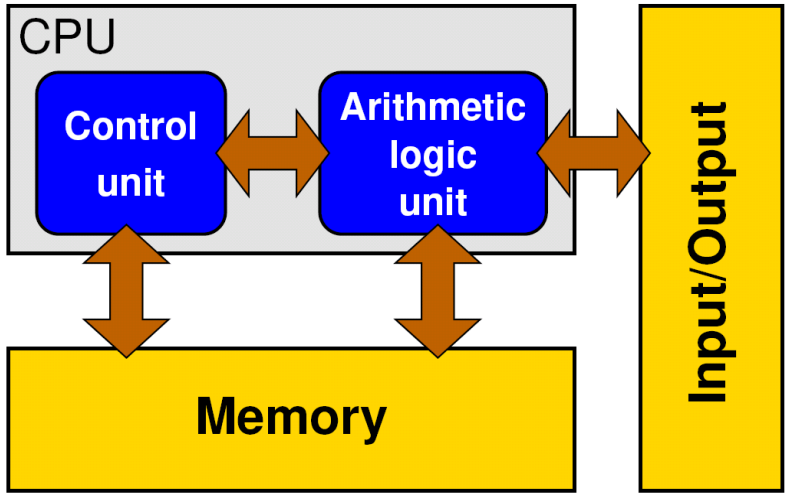
- Parallelism across multiple nodes or process
- Parallelism across threads
- Parallelism across instructions
- Parallelism on data – SIMD Single Instruction Multiple Data



www.scan.co.uk

Von Neumann architecture

- Instruction and data are stored in memory
- Instructions are read by a control unit
- Arithmetic/logic unit is responsible for computation
- CPU consists of control and arithmetic units with interfaces to memory and I/O
- Compiler translates high level language into instructions that can be stored and executed



Difficulties with the van Neumann architecture

- Instructions and data must be fed continuously to control and arithmetic units
  - Speed of memory is the limiting factor on compute performance
  - *Neumann bottleneck*
- Sequential architecture
  - SISD (Single Instruction Single Data)
- Modified and extended for modern processors
  - Parallelism on all levels of a machine

Performance Metrics

- Every component of a CPU can operate at some maximum speed
  - **Peak Performance**
- Applications generally cannot achieve peak performance
- Floating-point operations per second (FLOPS)
  - Counting only multiply add
  - Other operations are more expensive
- Modern processors can deliver at between 2 or 8 FLOPS per clock cycle
- Typical clock frequency between 2 and 3 GHz

Floating Point Performance

$$P = n_{\text{core}} * F * S * \nu$$

- Example: Intel Xeon E5 on Stamped
  - Number of cores: 8 n_{core}
  - FP instructions per cycle: 2 (1 Multiply and 1 add) F
  - FP operations / instruction (SIMD): 4 (dp) / 8 (sp) S
  - Clock speed: 2.7 GHZ ν

$$P = 173\ GF/s\ (dp) \quad \text{or} \quad 346\ GF/s\ (sp)$$

  - But: P= 5.4 GF/s (dp) for serial, non-SIMD code

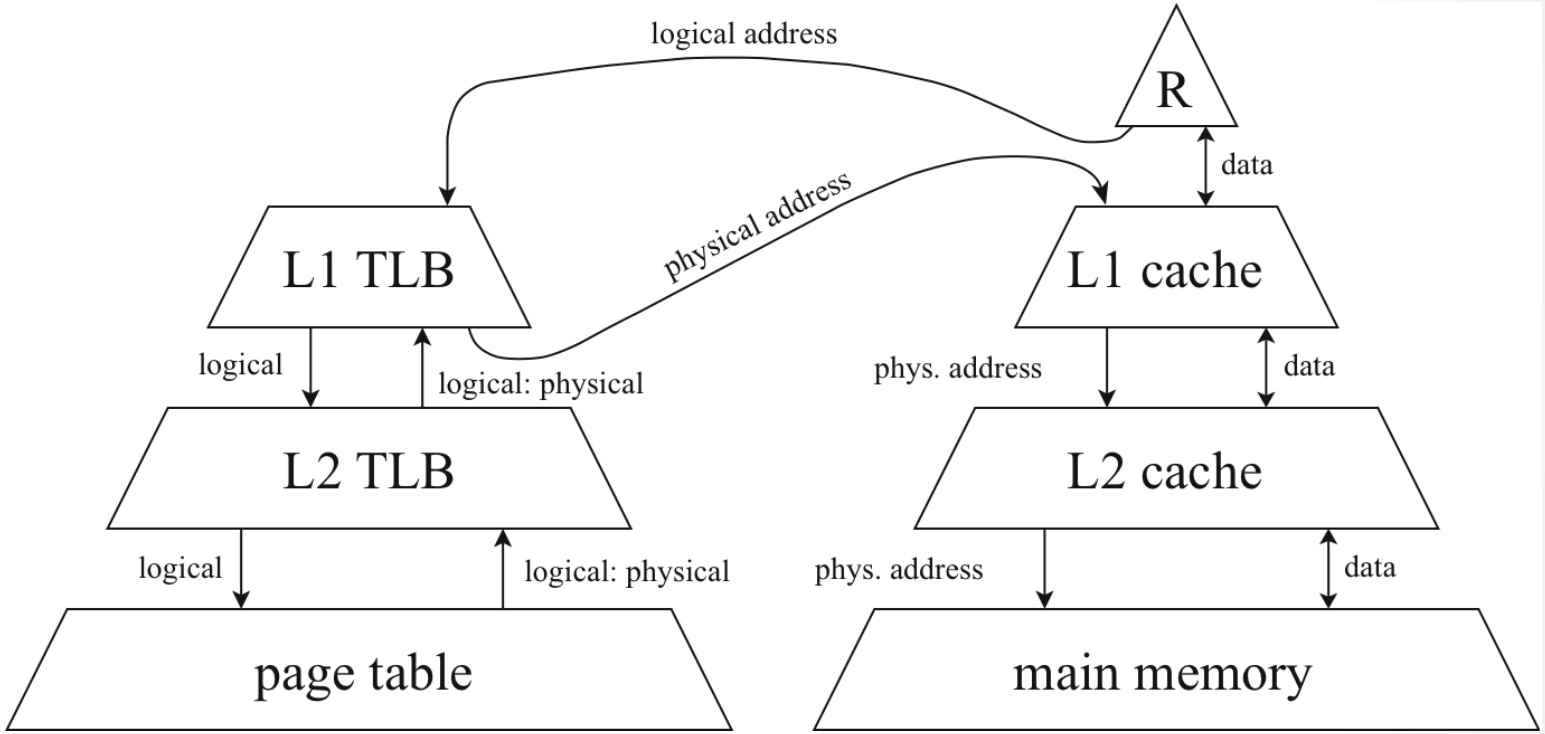
Memory Performance

- Path from and to caches and memory is generally the bottle neck for application performance.
- Bandwidth is the performance measure for memory
  - Gbytes/sec

Data-centric view of a microprocessor

- GFlops/sec
- Gbytes/sec
- Describe the most relevant performance features of microprocessors
- Memory access is generally the limiting factor for performance
  - Data centric view

Memory system

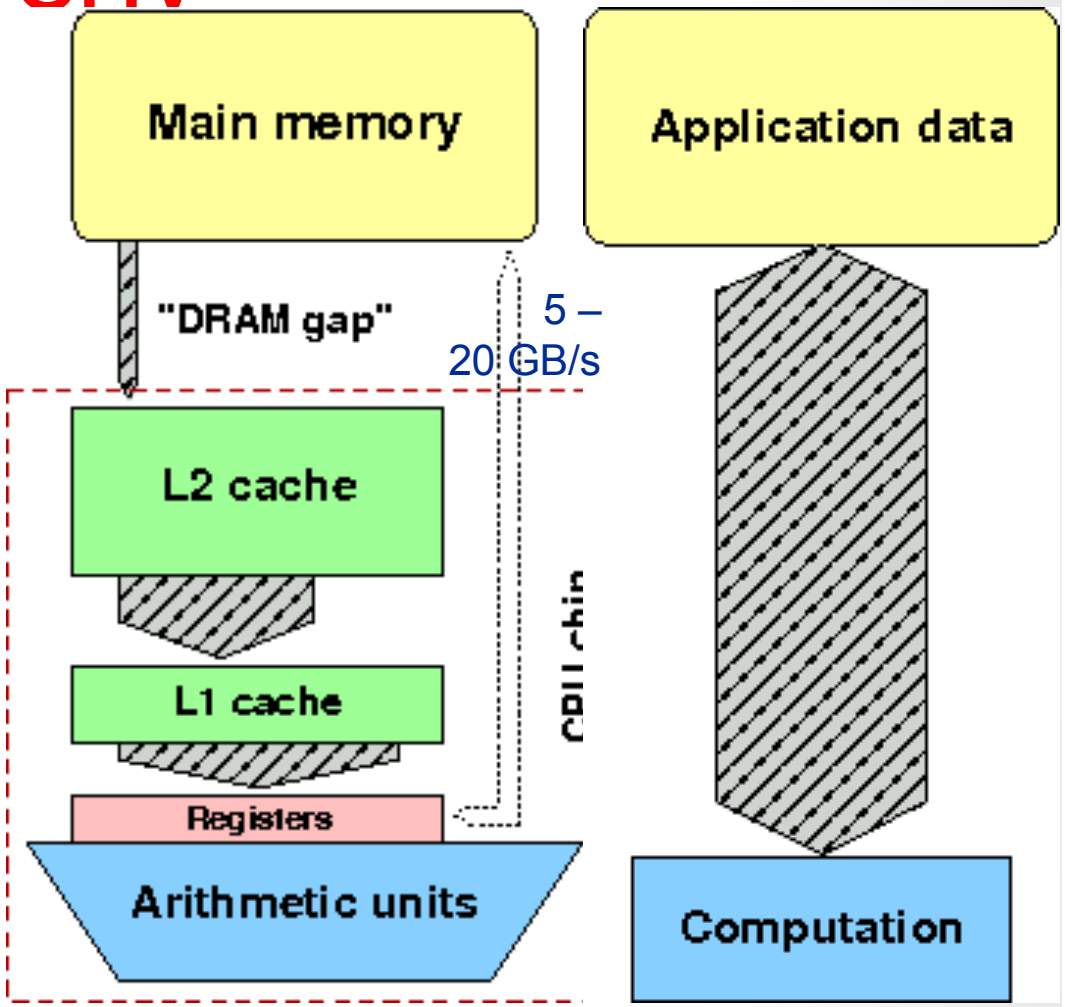


- R: register
- Right: Data and instruction caches
- Left: Translation caches

Memory hierarchy

1. CPU/Arithmetic unit issues a load request to transfer a data item to a register
2. Cache logic automatically checks all cache levels if data item is already in cache
3. If data item is in cache "cache hit" it is loaded to register.
4. If data item is in no cache level ("cache miss") data item is loaded from main memory and a copy is held in cache

If cache is already full another cache line must be invalidated or "evicted" in 4



Optimize for memory access

- Page Fault, file on IDE disk: 1.000.000.000 cycles
- Page Fault, file in buffer cache: 10.000 cycles
- Page Fault, file on ram disk: 5.000 cycles
- Page Fault, zero page: 3.000 cycles
- Main memory access: about 200 cycles
- L3 cache hit: abut 52 cycles
- L1 cache hit: 2 cycles

The Core i7 can issue 4 instructions per cycle. So a penalty of 2 cycles for L1 memory access means a missed opportunity for 7 instructions.

Peak vs. Realized Performance

Peak performance = guaranteed not to exceed

Realized performance = what you achieve

Scientific applications realize as low as
1-25% of peak on microprocessor-based systems

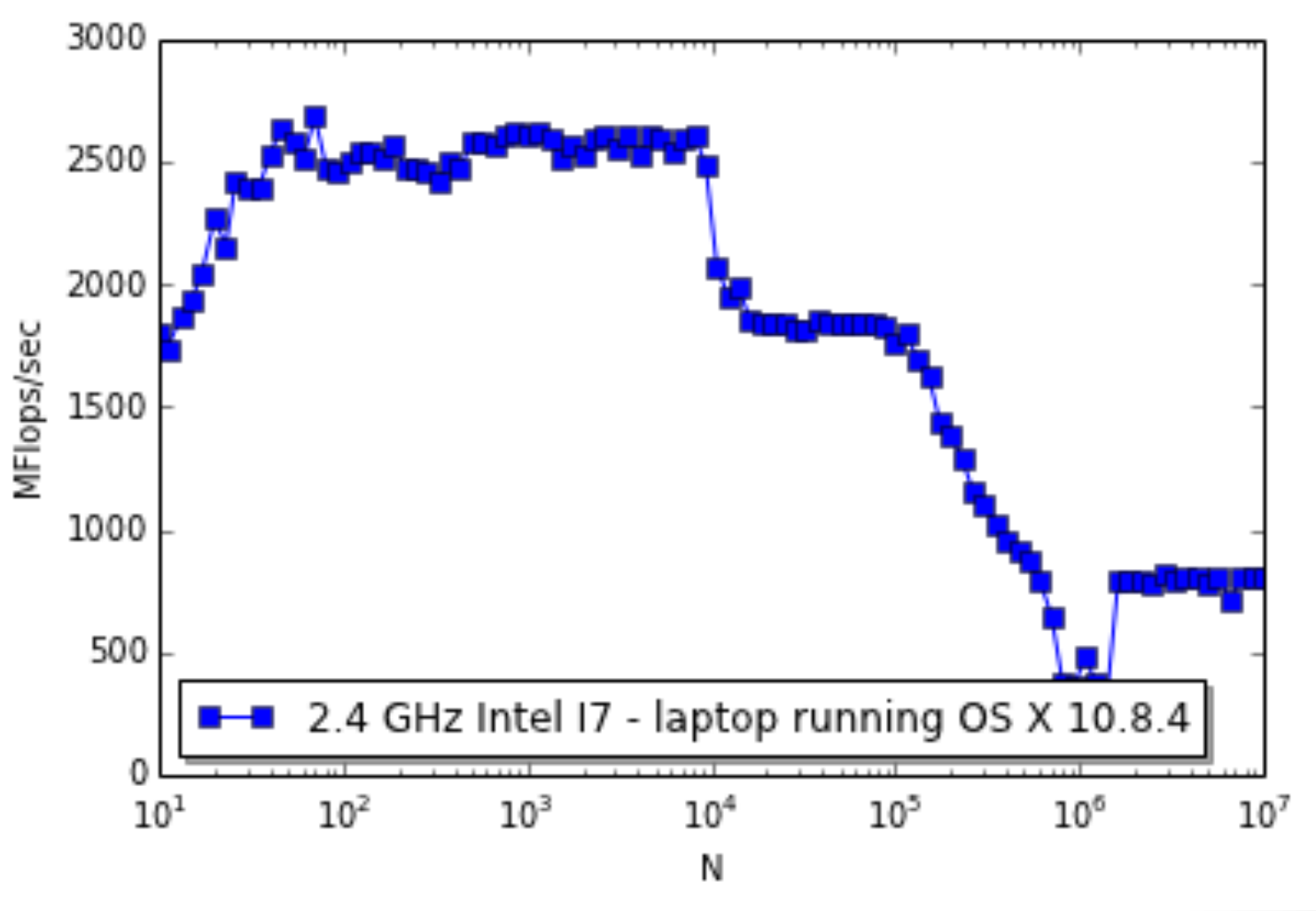
Reason: mismatch between application and architecture capabilities

- Architecture has insufficient bandwidth to main memory:
  - microprocessors often provide < 1 byte from memory per FLOP
    - scientific applications often need more
- Application has insufficient locality
  - irregular accesses can squander memory bandwidth
    - use only part of each data block fetched from memory
  - may not adequately reuse costly virtual memory address translations
- Exposed memory latency
  - architecture: inadequate memory parallelism to hide latency
  - application: not structured to exploit memory parallelism
- Instruction mix doesn't match available functional units

Low-level benchmarking

- Benchmark to test some specific feature of the architecture
  - Peak performance
  - Memory bandwidth
- Example of benchmark is *vector triad*
  - Measures performance of data transfers between memory and arithmetic units
  - A(:) = B(:) + C(:) * D(:)
  - 3 load streams
  - 1 store stream
  - 2 Flops

Results for Triad from my Laptop



Stream example

- See notebook

Performance Analysis and Tuning

- Increasingly necessary
  - Gap between realized and peak performance is growing
- Increasingly hard
  - Complex architectures are harder to program effectively
    - complex processors: pipelining, out-of-order execution, VLIW
    - complex memory hierarchy: multi-level non-blocking caches, TLB
  - Optimizing compilers are critical to achieving high performance
    - small program changes may have a large impact
  - Modern scientific applications pose challenges for tools
    - multi-lingual programs
    - many source files
    - complex build process
    - external libraries in binary-only form
  - Many tools don't help you identify or solve your problem

Tuning Applications

- Performance is an interaction between
  - Numerical model
  - Algorithms
  - Problem formulation (as a program)
  - Data structures
  - System software
  - Hardware
- Removing performance bottlenecks may require dependent adjustments to all

Having fun with compiler options

The Joy of Compiler Options

- Every compiler has a different set of options that you can set.
- Among these are options that control single processor optimization: superscalar, pipelining, vectorization, scalar optimizations, loop optimizations, inlining and so on.

GCC optimization options

-falign-functions[=*n*] -falign-jumps[=*n*] -falign-labels[=*n*] -falign-loops[=*n*] -fassociative-math -fauto-inc-dec -fbranch-probabilities -fbranch-target-load-optimize -fbranch-target-load-optimize2 -fbtr-bb-exclusive -fcaller-saves -fcheck-data-deps -fcombine-stack-adjustments -fconserve-stack -fcompare-elim -fcprop-registers -fcrossjumping -fcse-follow-jumps -fcse-skip-blocks -fcx-fortran-rules -fcx-limited-range -fdata-sections -fdce -fdce -fdelayed-branch -fdelete-null-pointer-checks -fdse -fdevirtualize -fdse -fearly-inlining -fipa-sra -fexpensive-optimizations -ffast-math -ffinite-math-only -ffloat-store -fexcess-precision=*style* -fforward-propagate -ffp-contract=*style* -ffunction-sections -fgcse -fgcse-after-reload -fgcse-las -fgcse-lm -fgraphite-identity -fgcse-sm -fif-conversion -fif-conversion2 -findirect-inlining -finline-functions -finline-functions-called-once -finline-limit=*n* -finline-small-functions -fipa-cp -fipa-cp-clone -fipa-matrix-reorg -fipa-pta -fipa-profile -fipa-pure-const -fipa-reference -fipa-struct-reorg -fira-algorithm=*algorithm* -fira-region=*region* -fira-loop-pressure -fno-ira-share-save-slots -fno-ira-share-spill-slots -fira-verbose=*n* -fivopts -fkeep-inline-functions -fkeep-static-consts -floop-block -floop-flatten -floop-interchange -floop-strip-mine -floop-parallelize-all -flto -flto-compression-level -flto-partition=*alg* -flto-report -fmerge-all-constants -fmerge-constants -fmodulo-sched -fmodulo-sched-allow-regmoves -fmove-loop-invariants fmudflap -fmudflapir -fmudflapth -fno-branch-count-reg -fno-default-inline -fno-defer-pop -fno-function-cse -fno-guess-branch-probability -fno-inline -fno-math-errno -fno-peephole -fno-peephole2 -fno-sched-interblock -fno-sched-spec -fno-signed-zeros -fno-toplevel-reorder -fno-trapping-math -fno-zero-initialized-in-bss -fomit-frame-pointer -foptimize-register-move -foptimize-sibling-calls -fpartial-inlining -fpeel-loops -fpredictive-commoning -fprefetch-loop-arrays -fprofile-correction -fprofile-dir=*path* -fprofile-generate -fprofile-generate=*path* -fprofile-use -fprofile-use=*path* -fprofile-values -freciprocal-math -fregmove -frename-registers -freorder-blocks -freorder-blocks-and-partition -freorder-functions -frerun-cse-after-loop -freschedule-modulo-scheduled-loops -frounding-math -fsched2-use-superblocks -fsched-pressure -fsched-spec-load -fsched-spec-load-dangerous -fsched-stalled-insns-dep[=*n*] -fsched-stalled-insns[=*n*] -fsched-group-heuristic -fsched-critical-path-heuristic -fsched-spec-insn-heuristic -fsched-rank-heuristic -fsched-last-insn-heuristic -fsched-dep-count-heuristic -fschedule-insns -fschedule-insns2 -fsection-anchors -fselective-scheduling -fselective-scheduling2 -fsel-sched-pipelining -fsel-sched-pipelining-outer-loops -fsignaling-nans -fsingle-precision-constant -fsplit-ivs-in-unroller -fsplit-wide-types -fstack-protector -fstack-protector-all -fstrict-aliasing -fstrict-overflow -fthread-jumps -ftracer -ftree-bit-ccp -ftree-builtin-call-dce -ftree-ccp -ftree-ch -ftree-copy-prop -ftree-copyrename -ftree-dce -ftree-dominator-opts -ftree-dse -ftree-forwprop -ftree-fre -ftree-loop-if-convert -ftree-loop-if-convert-stores -ftree-loop-im -ftree-phiprop -ftree-loop-distribution -ftree-loop-distribute-patterns -ftree-loop-ivcanon -ftree-loop-linear -ftree-loop-optimize -ftree-parallelize-loops=*n* -ftree-pre -ftree-pta -ftree-reassoc -ftree-sink -ftree-sra -ftree-switch-conversion -ftree-ter -ftree-vect-loop-version -ftree-vectorize -ftree-vrp -funit-at-a-time -funroll-all-loops -funroll-loops -funsafe-loop-optimizations -funsafe-math-optimizations -funswitch-loops -fvariable-expansion-in-unroller -fvect-cost-model -fvpt -fweb -fwhole-program -fwpa -fuse-linker-plugin --param *name*=*value* -O -O0 -O1 -O2 -O3 -Os -Ofast

Example Compile Lines

- GCC
  - gfortran -Ofast
- Intel

```
ifort -Ofast
```

- Portland Group f90

```
pgf90 -fastsse -Mipa=fast
```

Scalar Optimizations

- Copy Propagation
- Constant Folding
- Dead Code Removal
- Strength Reduction
- Common Subexpression Elimination
- Variable Renaming
- Vectorization of loops
- Not every compiler does all of these, so it sometimes can be worth doing these by hand.

What is Vector or SIMD Code

- Processor can execute one instruction on a few elements
  - SSE: 4 elements at a time
  - AVX: 8 elements at a time
  - MIC: 16 elements at a time

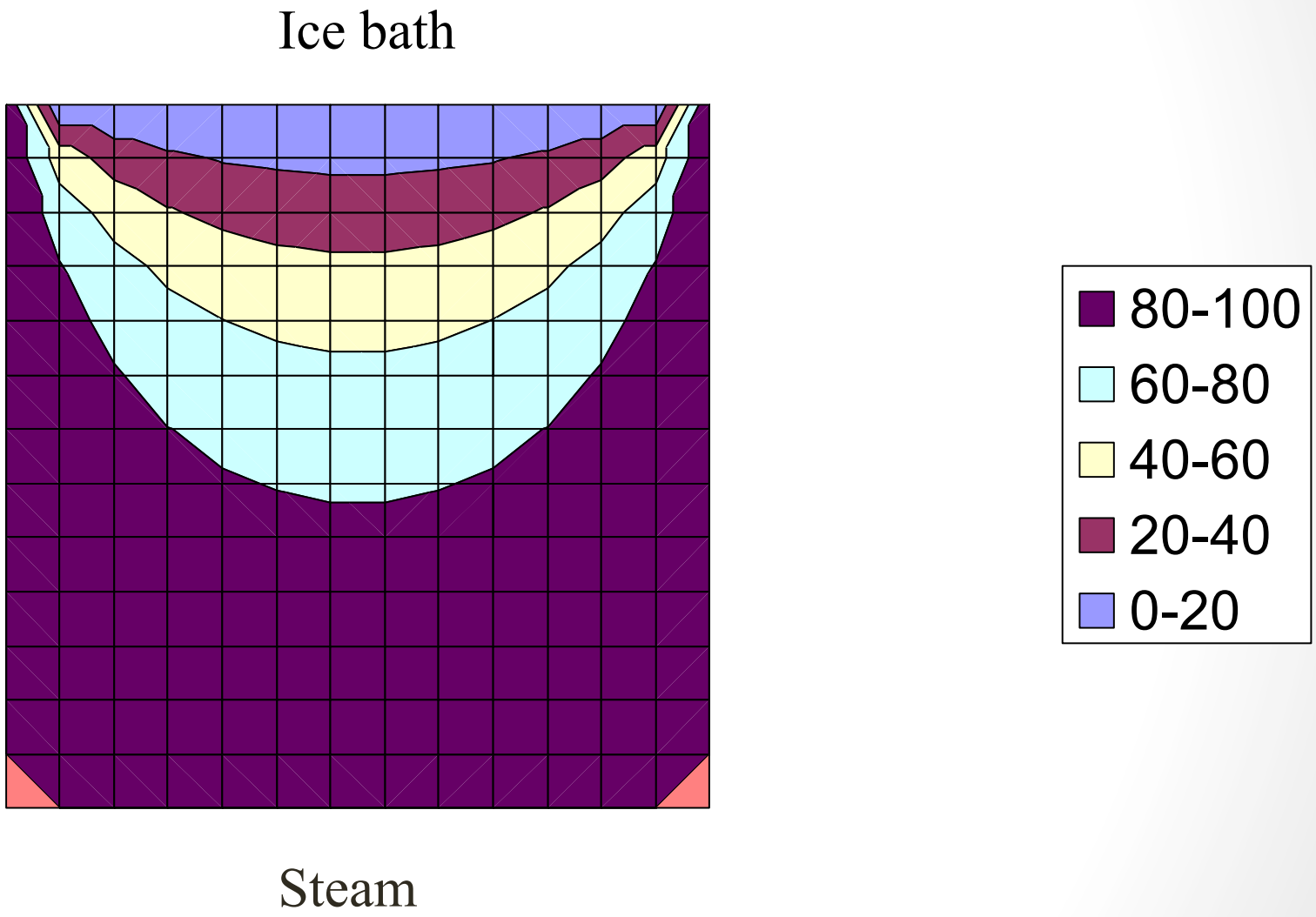
- Stream:
  - Scalar code computes result one element at a time

```
for (j=0; j<STREAM_ARRAY_SIZE; j++)
    a[j] = b[j]+scalar*c[j];
```

Programming Guidelines

- To vectorize a loop you may need to make simple changes
- Use
  - Simple for loops
  - No if statement in loops
  - No function calls
  - Fortran (or use restrict in C on pointers)

Steady State Heat Distribution Problem



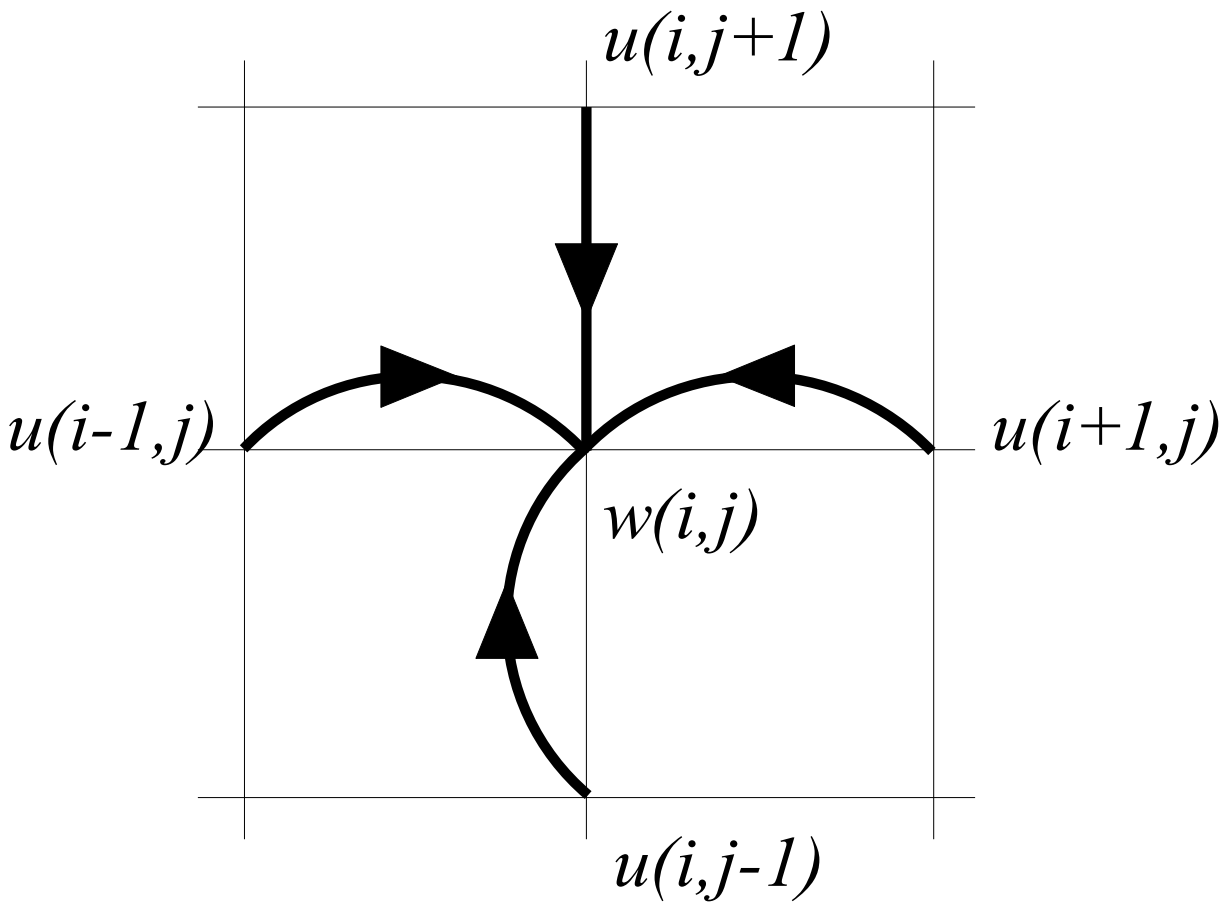
Solving the Problem

- Underlying PDE is the Poisson equation

$$u_{xx} + u_{yy} = f(x, y)$$

- This is an example of an elliptical PDE
- Will create a 2-D grid
- Each grid point represents value of state state solution at particular (x, y) location in plate

Heart of Sequential Program



Heat equation example

- gfortran 4.6.2
- 2.4. GHz Intel Core 2 Duo
- 1000x2000 problem size with 100 iterations
- -O
  - 2.443s
- -O2
  - 2.452s
- -O3
  - 1.163s
- -Ofast
  - 1.147s

Heat equation example

- -O
  - 2.443s
- -O2
  - 2.452s
- -O3
  - 1.163s
- -Ofast
  - 1.147s
- -Ofast –funroll-all-loops
  - 1.123

Loop Optimizations

- Hoisting Loop Invariant Code
- Unswitching
- Iteration Peeling
- Index Set Splitting
- Loop Interchange
- Unrolling
- Loop Fusion
- Loop Fission
- Not every compiler does all of these, so it sometimes can be worth doing some of these by hand.

Loop Carried Dependency

```
DO i = 2, length
   a(i) = a(i-1) + b(i)
END DO
```

Here, each iteration of the loop depends on the previous iteration. That is, the iteration `i=3` depends on iteration `i=2`, iteration `i=4` depends on `i=3`, iteration `i=5` depends on `i=4`, etc.

This is sometimes called a ***loop carried dependency***.

There is no way to execute iteration `i` until after iteration `i-1` has completed, so this loop can't be parallelized.

Reductions Aren't Dependencies

```
array_sum = 0
DO i = 1, length
  array_sum = array_sum + array(i)
END DO
```

- Other kinds of reductions:
  - product, .AND., .OR., minimum, maximum, index of minimum, index of maximum, number of occurrences of a particular value, etc.
- Reductions are so common that hardware and compilers are optimized to handle them.
- Also, they aren't really dependencies, because the order in which the individual operations are performed doesn't matter.

Why Do We Care?

Loops are the **favorite control structures** of High Performance Computing, because compilers know how to optimize their performance using instruction-level parallelism: superscalar, pipelining and vectorization can give excellent speedup***.***

Loop carried dependencies affect whether a loop can be parallelized, and how much.

Loop Dependency Example

```
do index = 2, length
  dst(index) = dst(index-1) + dst(index)
end do

do index = 2, length
  dst(index) = dst(index-1) + src2(index)
end do

do index = 2, length
  dst(index) = src1(index-1) + dst(index)
end do

do index = 2, length
  dst(index) = src1(index-1) + src1(index)
end do

do index = 2, length
  dst(index) = src1(index-1) + src2(index)
end do
```

Hoisting Loop Invariant Code

Code that doesn't change inside the loop is called **loop invariant** It doesn't need to be calculated over and over

Before

```
DO i = 1, n
  a(i) = b(i) + c * d
  e = g(n)
END DO
```

After

```
temp = c * d
DO i = 1, n
  a(i) = b(i) + temp
END DO
e = g(n)
```

Unswitching

```
DO i = 1, n
  DO j = 2, n
    IF (t(i) > 0) THEN
      a(i,j) = a(i,j) * t(i) + b(j)
    ELSE
      a(i,j) = 0.0
    END IF
  END DO
END DO
```

Before

The condition is j-independent.

```
DO i = 1, n
  IF (t(i) > 0) THEN
    DO j = 2, n
      a(i,j) = a(i,j) * t(i) + b(j)
    END DO
  ELSE
    DO j = 2, n
      a(i,j) = 0.0
    END DO
  END IF
END DO
```

After

So, it can migrate outside the j loop.

Loop Interchange

Before

```
DO i = 1, ni
  DO j = 1, nj
    a(i,j) = b(i,j)
  END DO
END DO
```

After

```
DO j = 1, nj
  DO i = 1, ni
    a(i,j) = b(i,j)
  END DO
END DO
```

Array elements **a(i,j)** and **a(i+1,j)** are near each other in memory, while **a(i,j+1)** may be far, so it makes sense to make the **i** loop be the inner loop. (This is reversed in C, C ++ and Java.)

Summery

- Find optimized libraries first
- Try the compiler next
  - Select options
  - Review compiler output, e.g a loop can’t be vectorized
- Profile
  - Rewrite only code which is relevant to your performance
- Rewrite code
  - Get the loop ordering right
  - Avoid if statement in a loop
  - Look at cache behavior too
    - Blocking/Tiling

Summary

- Readable and correct code is more important than fast and obfuscated code
- Optimize for memory hierarchy
- Profile to identify hotspots
- Applied to real CFD code