NCI Serial Programming

National Computational Infrastructure

Outline

Introduction

Course is:

- motivated by common user problems and misconceptions
- not specific to any processor or operating system but YMMV
- aimed at understanding the basic optimisation techniques
- assuming predominantly floating point scientific computing

Exercise material

```
> tar xf /short/c23/SerialOpt.tar
> cd SerialOpt
> ls
```

Outline

IO - the good, the bad and the ugly ...

Most of it is UGLY!

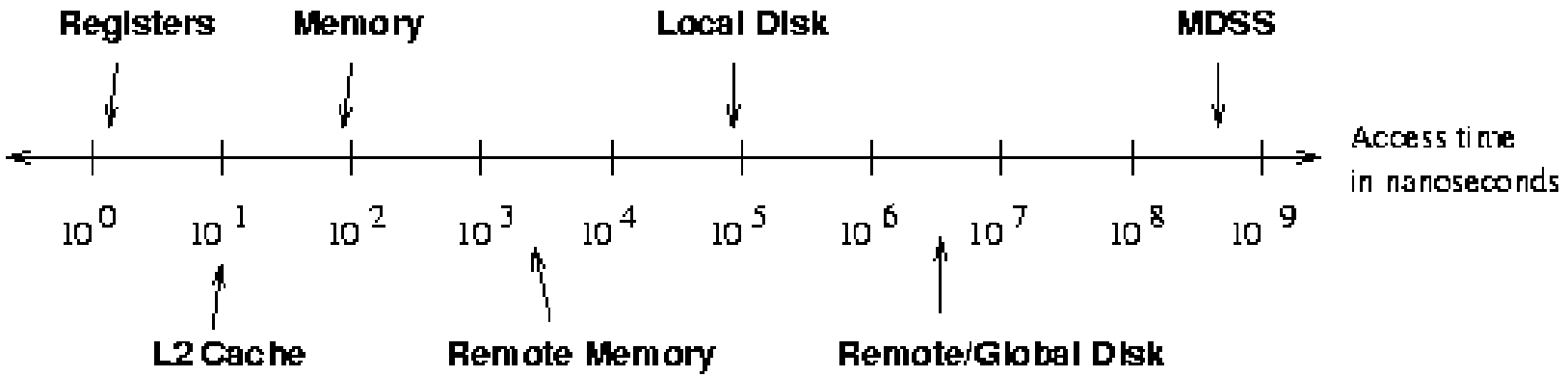
- Easily the most serious performance problem on the AC
- Filesystems on large parallel systems geared to BIG IO - cannot support 500 "general purpose" users.
- What works OK on your PC may be disastrous on the AC
- Do IO in big chunks as infrequently as possible
- Filesystems for all occasions. Different file systems can give vastly different performance.

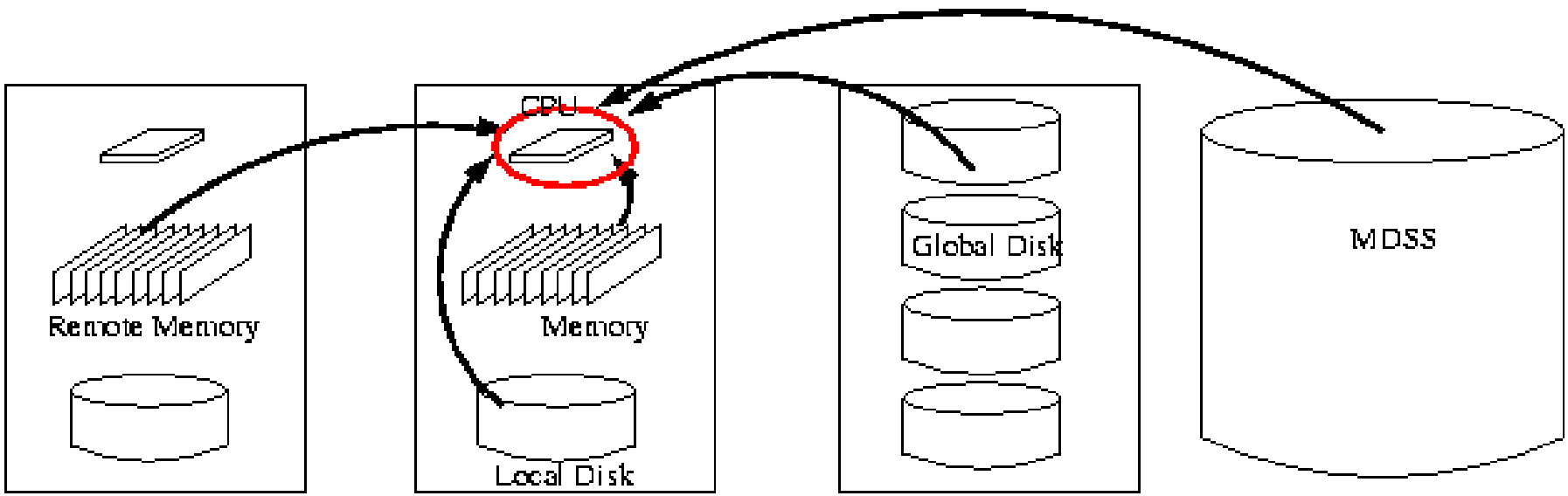
Exercise: IO

Compile and run each of `goodio1.f90`, `goodio2.f90`, `goodio3.f90` and `bestio.f90` and compare the different IO methods. Beware, the array sizes are different and the timings include the computation (which should be negligible).

The demonstrator will run `poorio.f90` to show you how bad it can be.

Disk IO Time Scales and Locality





Timers

- measuring cputime (shared system) and real time (dedicated system)
- granularity of timers - might need to time multiple runs
- cputime is only accurate to 1/100th or 1/1024th sec on Linux
- overhead of timer call may affect timings
- watch out for wraparound of timer counters
- for accurate walltime, use:
  - `gettimeofday()` on Linux
  - `MPI_Wtime()` on some systems
- simple `time` command, eg. `> time a.out`

```
48.231u 0.303s 0:48.67 99.7% 0+0k 0+0io 0pf+0w
```

Exercise: Timers

Compile and run `timers.(c|f90)` and compare the difference between the timers. For Fortran, we have provided a C wrapper to `gettimeofday()` - compile using:

```
> icc -c fgettimeofday.c
> ifort timers.f90 fgettimeofday.o -o timers
```

- Read the code to find what timer calls are used.
- Note how many iterations of a loop can fit within ticks of the clock.

Outline

Compilers

- Usually large wins from using compiler optimization flags
- Default often uses some optimization
- Beware of slowdowns or wrong answers(!) at highest optimization (eg. `-O5`)
- Look for good generic optimization flags like `-fast` or `-Ofast`
  For Intel: `-fast = "-ipo -O3 -static"` where `ipo` means interprocedural optimization
- Recognizing basic assembler (produced by `-S` option) might be useful

Compiler option gotchas

- Beware of Intel `-ipo`: object files (.o files) are not really object files, they are translated source files. Optimized compilations happens at "link" time when all .o's are visible. Cant manage .o's with link tools like `ld` or `ar`.
- Beware of result changes due to `-fast_maths` (`-IPF-fp-relaxed` for Intel compilers) options.
- Watch out for `-fpe` or `-speculate all` type flags (eg. `-IPF-fp-speculation fast` for Intel). May cause real FPE problems to be hidden.

Optimization levels

- -O0 disable optimization
- -O1 optimizations without code bloat (no SWP or loop unrolling)
- -O2 (default) intra-file IPO, SWP, speculation
- -O3 aggressive loop transforms, data prefetch, ...

IPO = Inter-procedural optimization
SWP = Software pipelining

```
-opt-report -opt-report-level ....  :
```

- the gory details of what the compiler did
- terse and not self-explanatory but can pick up some info

Profile-based compilation

- Build code with `-prof_gen` (optimization restricted and not important)
- Run code on typical dataset. Produces statistics (coverage, branching ratios, ...) in a file like `4444e620_02211.dyn`
- Rebuild with `-prof_use` and high optimization (eg. `-fast`)
- Best suited to "branchy code" - lots of conditionals, line execution counts not obvious.
- Always used in SPEC benchmarking

Parallel options

- `-parallel -par-report [0|1|2|3]`
  autoparallelism and a report on why not!
- `-openmp -openmp-report [0|1|2]`
  detailed info on OpenMP parallelism

Exercise: compiler options

Play :-)

If you have your own code, try compiler options on that.
If not, try them on `oceancode.f`

Profilers

- counter based versus sampling based profiling
- `gprof` and `histx`: statistical sampling
- `pfmon`: instruction counting, procedure profile
- Often require `-g` option for source line-based profile.
- Optimization will "smear out" source line-based profiles. Think of code blocks, not lines.
- IPO will muddy profiles further.
- hardware monitoring (`profile.pl`): uses hardware counters to analyse a program

gprof

Need to build your binary with instrumentation (`-p` option):

```
> icc -O3 -g -p program.c -o prog
> ./prog
> gprof ./prog
> gprof -l ./prog  # source line-based profile
```

`-fast` may obfuscate source line-based profiles

histx

Compile your binary as normal (no instrumentation) and run histx with the resultant binary as an argument:

```
> icc -O3 program.c -o prog
> module load histx
> histx ./prog      # creates hist.prog.{pid}
> iprep histx.prog.{pid}
> icc -O3 -g program.c -o prog
> histx -l ./prog   # will generate source line
                    # profile information
```

Basic documentation online in
`/opt/histx/histx-1.3b/doc/histx.txt`
No profiling output using `-fast`

Hardware monitoring

Provide cycle counting, cache misses, stalls, memory access and a whole heap of very detailed information. A course in its own right. Most performance gains can be made without hardware monitors.

Exercise: Profiling

Build and profile (using both `gprof` and `histx`) your own code or `laplace.(c|f)` or `oceancode.f`. Try using the different profiling features eg. determine which line in the code is the most often executed.

Outline

Architecture

Common features:

- multiple floating point units
- pipelined
- out-of-order execution
- cycle counts for operations
- caches and cache lines
- memory access times

Arithmetic Units

- Loads one or two operands (from registers) and writes the result back to a register
- Processor may or may not support the following FP operations in hardware:

Operation	Stages	Pipelined
Add	5	YES
Multiply	5	YES
FMA	5	YES
Divide	20	NO
Sqrt	20	NO

Pipelining

Hardware execution of arithmetic operation:

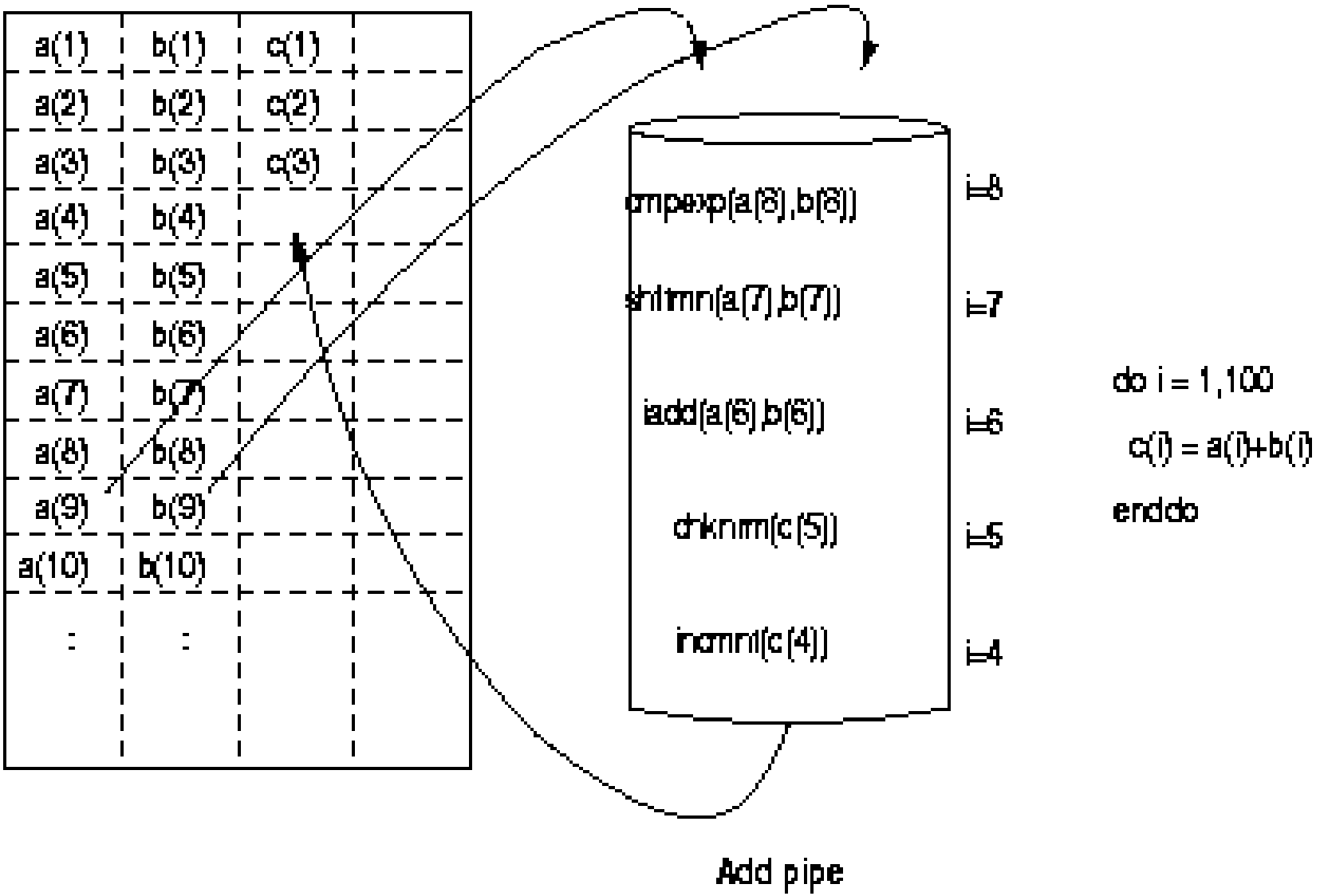
- 5 to 10 “single-cycle” sub-operations (stages)
- scalar execution:
  - operands pass through all stages and result produced before next pair of operands start
  - 1 results every 5 clock cycles
- pipelined execution:
  - new operation starts every cycle
  - operands of different iteration at every stage simultaneously
  - results every clock cycle

Pipelining 2

FP Add operation stages:

1. compare exponents
2. shift mantissa bits in one operand
3. integer add of the mantissas (XOR of bits)
4. check for normalization
5. increment or decrement the results mantissa if necessary

Example Pipeline



Pipelining 3

- compiler generally manages pipelining
- only way to get anywhere near peak flops
- often limited by memory bottleneck (see next)
- requires lots of **independent** operations
- recursion kills pipelining:

```
do i = 1,n
    a(i) = b(i) + a(i-1)
enddo
```

- see later for other problems

Outline

Memory access

- Processors are getting faster much faster than memory!
- Systems use hierarchical memory to try to overcome this
- Small amounts of "fast memory" and large amounts of "slow memory"
- All sorts of sophisticated mechanisms to integrate these:
  - caching
  - cacheline loading
  - prefetching
  - ...
- Lots of pitfalls to kill performance! (strided access, cache misses, TLB misses, ...)

Memory levels

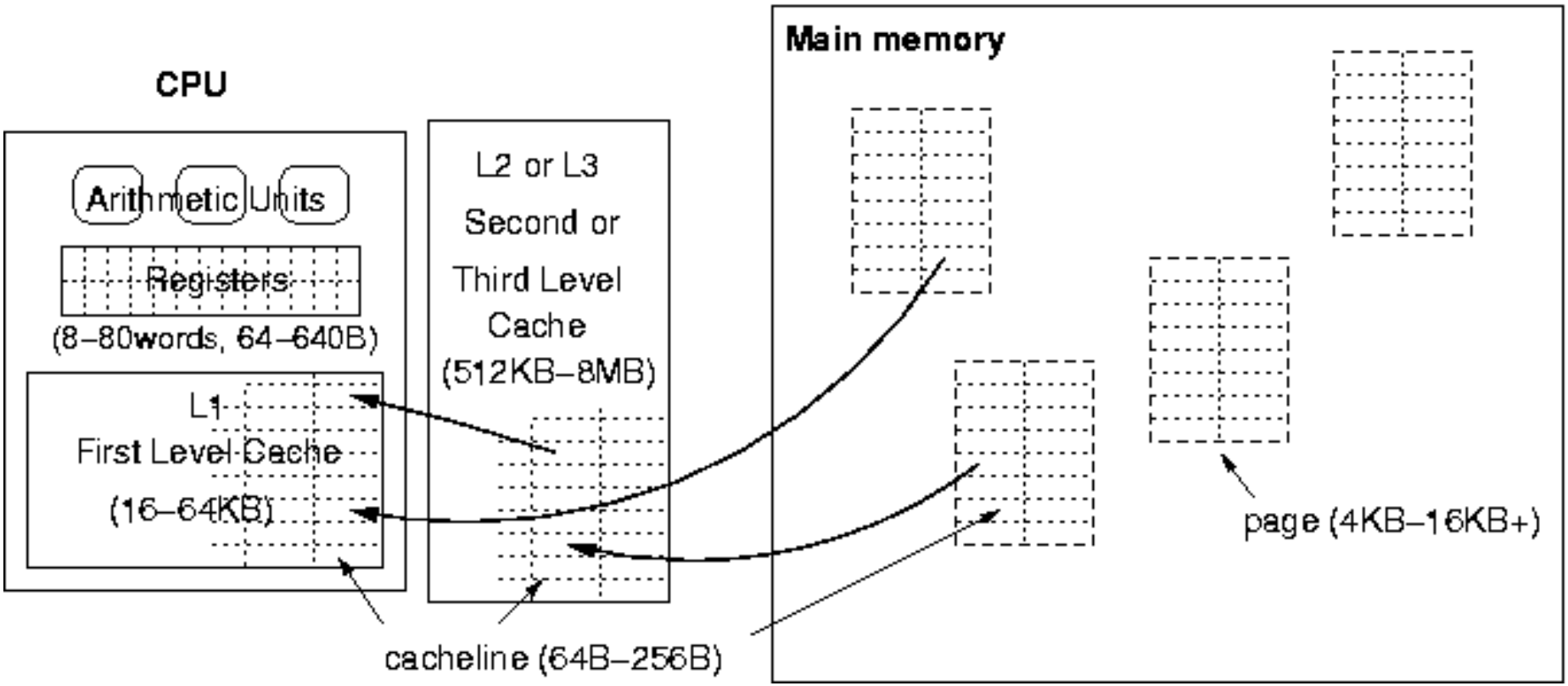
Memory level	Size	Access time
Registers	64-640B	1ns
L1 cache	16-64KB	2-5ns
L2 cache	512KB-8MB	5-10ns
Main memory	128MB-...	70-400ns

1ns = 1-3 clock cycles.

Cache

- Cache is fast memory close to the CPU.
- A load of a word from a memory address actually loads a *cacheline* (typically 32-256 contiguous bytes or 4-32 words).
- So a long memory access time (*high latency*) returns a lot of data (*reasonable bandwidth*)
- If you only use 1 word from the cacheline, you waste the bandwidth!
- Using all the contiguous words amelorates the load cost

Typical memory system



Exercise: Strided access

Take a look at `memory.(f90|c)` and notice how it is doing a strided access through memory.

- Compile the code with
  `(ifort|icc) memory.(f90|c) -o memory`
- Run it with
  `./memory`

Run it with different values for the stride (make sure you do 1) and notice how as you increase the stride, the time to execute increases. What happens if you use `-fast`?

Memory access patterns

Worst-to-Best List:

- strided or random access over a large range (˜1GB) with gap > pagesize (4-16KB)
- random or strided access with a gap >= cacheline size (64B-256B)
- small stride with gaps between 1 and cacheline size (2-8)
- contiguous access
- repeat use of data:
  - from a small set of pages
  - from a small set of cachelines (512KB-8MB) 2nd or 3rd level cache
  - from an even smaller set of cachelines (16KB-64KB) 1st level cache
  - from a small set of words (8-80) in registers

Multidimensional arrays

- Contiguous indices may not mean contiguous memory
- Multidimensional arrays are "linearized" in memory
- Fortran has first index contiguous, C has last

```
parameter :: N = 128
real*8 a(N,N,N)

do i = 1,N
   do j = 1,N
      do k = 1,N
       ....  a(i,j,k) ....
      enddo
   enddo
enddo
       BAD!
```

```
#define N 128
double a[N][N][N];

for(i=0;i<N;i++)
  for(j=0;j<N;j++)
    for(k=0;k<N;k++)
          .... a[i][j][k] .....

       GOOD!
```

Cache Reuse

- Streaming contiguously through lots of memory is still slow
- Getting data directly from cache on every load is up to an order of magnitude faster
- Reuse data already in cache as many times as possible before it gets expelled
- Blocked algorithms for dense linear algebra
- Reordering of loop operations in other algorithms

Exercise: Cache reuse

The simplest way to make better use of cache it to continually use the same memory addresses. By interleaving iterations, modify the code `fd.(f90|c)` to make better use of cache.

Outline

Code Optimization

- Strength reduction
- Conditionals
- Procedure calls and inlining
- Unrolling
- Aliasing
- ...

Strength reduction

- Should know which operations are expensive
- Example

```
divides
a**2.0d0   vs a*a
a**0.5d0   vs sqrt(a)
a**2.5d0   vs a*a*sqrt(a)
pow()
```

Exercise ####

Conditionals

- try to avoid conditional in inner loops
- breaks pipelining
- branch mispredictions cause major disruption to op flow
- unusable memory loads, registers and caches polluted . . .
- Use of:
  - `c=merge(a,b,logical)` in Fortran
  - `c=logical?a:b;` in C

  can generate single instruction (`cmov`) and no branch

Exercise: Conditionals

Compile and time the code `conditional.(c|f90)`

```
> ifort -fast conditional.f90 -o conditional
> time ./conditional
```

and in C

```
> icc -fast condition.c -o conditional
> time ./conditional
```

Now removing as many conditionals as possible and see what performance increase is observed.

Procedure calls and inlining

- Procedure calls are a GOOD THING^{TM} (for code structure and management)
- But they do cost cycles to push registers on the stack etc
- Much worse - procedure calls break pipelining in "tight" loops

Procedure calls and inlining 2

- Inlining involves inserting the procedure code in place of the call:
  - by hand
  - by preprocessors and macros
  - by the compiler
    - often requires more ops etc
    - compiler has to "see" the procedure source when compiling the call
- Intel compiler option `-ipo` (or `-fast`) attempts massive inlining
- C++ has an `inline` directive for procedures (but does it always work?)

Exercise: Procedure calls

Compile the code subroutine code and time how long it takes

```
> ifort -O3 -c disp.f90
> ifort -O3 -c procedure.f90
> ifort disp.o procedure.o -o procedure
> time ./procedure
```

and in C

```
> icc -O3 -c disp.c
> icc -O3 -c procedure.c
> icc disp.o procedure.o -o procedure
> time ./procedure
```

Rewrite `procedure.(c|f90)` and `disp.(c|f90)` removing the procedure calls. (Or cheat and use `-fast`!)

Loop Unrolling

Pros and Cons

Exercise?

Language Issues

- Aliasing
  - causes load/store ordering restrictions
  - limits instruction reordering
  - Fortran rules to disallow hidden aliasing
- Loop counters
  - is loop counter modifiable?
  - is loop iteration count fixed?

Other Tools

Show other code problems

truss on Tru64 and Solaris, strace on Linux

Lotsa system level info iostat, vmstat, etc

Floating Point

IEEE FP
accuracy and optimization
fast math libraries

FPE and overhead

Exercise

Summary

- Get your IO right!
- Use intelligent algorithms
- Get your memory access as contiguous and as infrequent as possible
- Keep the arithmetic pipes hot! Avoid
  - conditionals and branchs
  - procedure calls

  in tight inner loops
- PROFILE, PROFILE, PROFILE