

---

# ***Profiling and Optimizing Parallel Applications***

**May 28, 2019**

**Parallel Computational Science and Engineering  
PHYS 244**

**Robert Sinkovits  
San Diego Supercomputer Center**



**SAN DIEGO SUPERCOMPUTER CENTER**

---

*at the* UNIVERSITY OF CALIFORNIA; SAN DIEGO



---

Disclaimer – this lecture is really focuses on optimizing the serial code that underlies the parallel application. Of course, this can lead to much faster parallel code regardless of the parallelization method.

This presentation was adapted from a tutorial at SDSC's 2018 Summer Institute for High Performance Computing and Data Science. The full version can be found at

[https://github.com/sdsc/sdsc-summer-institute-2018/tree/master/hpc1\\_performance\\_optimization](https://github.com/sdsc/sdsc-summer-institute-2018/tree/master/hpc1_performance_optimization)

Additional material on gprof, top, timers and getting machine information

[https://github.com/sdsc/sdsc-summer-institute-2018/tree/master/6\\_software\\_performance](https://github.com/sdsc/sdsc-summer-institute-2018/tree/master/6_software_performance)

# Why optimize your code

- **Computer time is a limited resource.** Time on XSEDE systems is free\*\*, but awarded on a competitive basis – very few big users get everything they want. Time on Amazon Web Services or other cloud providers costs real dollars. Maintaining your own cluster/workstation requires both time and money.
- **Optimizing your code will reduce the time to solution.** Challenging problems become doable. Routine calculations can be done quickly enough to allow time for exploration and experimentation. In short, you can get more science done in the same amount of time.
- **Even if computer time was free, running a computation still consumes energy.** There's a lot of controversy over how much energy is used by computers and data centers, but estimates are that they account for 2-10% of total national energy usage.

\*\* XSEDE resources are not really free since someone has to pay. The NSF directly, tax payers indirectly. Average US citizen paid about \$0.07 to deploy and operate Comet over it's lifetime

---

... but I have a parallel code and processors are getting faster, cheaper and more energy efficient

- There will always be a more challenging problem that you want to solve in a timely manner
  - Higher resolution (finer grid size, shorter time step)
  - Larger systems (more atoms, molecules, particles ...)
  - More accurate physics
  - Longer simulations
  - More replicates, bigger ensembles, better statistics
- Most parallel applications have a limited scalability
- For the foreseeable future, there will always be limitations on availability of computation and energy consumption will be an important consideration

# Guidelines for software optimization

The prime directive of software optimization: **Don't break anything!**

Getting correct results slowly is much better than getting wrong results quickly

- **Don't obfuscate your code** unless you have a really good reason (e.g. kernel in a heavily used code accounts for a lot of time)
- **Clearly document your work**, especially if new code looks significantly different
- **Optimize for the common case**
- **Know when to start/stop**
- **Maintain portability**. If you need to include modifications that are architecture or environment specific, use preprocessor directives to isolate key code
- **Profile, optimize, repeat** – new hotspots may emerge
- **Make use of optimized libraries**. Unless you are a world-class expert, you are not going to write a faster matrix multiply, FFT, eigenvalue solver, etc.
- **Understand capabilities and limitations of your compiler**. Use compiler options (e.g. -O3, -xHost) for best performance

# Know when to start / stop

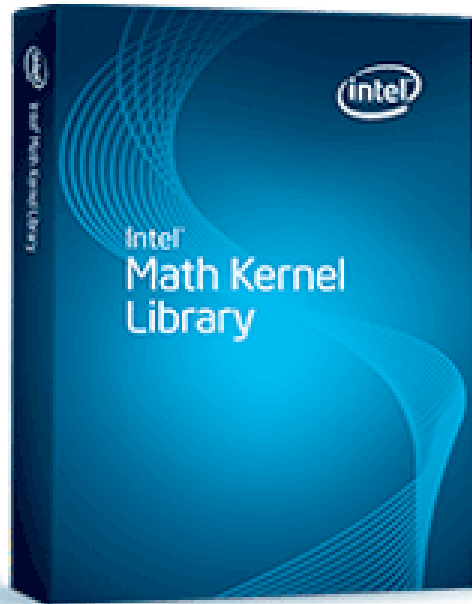
## Knowing when to start

- Is the code used frequently/widely enough to justify the effort?
- Does the code consume a considerable amount of computer time?
- Is time to solution important?
- Will optimizing your code help you solve new sets of problems?

## Knowing when to stop

- Have you reached the point of diminishing returns?
- Is most of the remaining time spent in routines beyond your control?
- Will your limited amount of brain power and/or waking hours be better spent doing your research than optimizing the code?

# Intel's Math Kernel Library (MKL)



Highly optimized mathematical library. Tuned to take maximum advantage of Intel processors. This is my first choice when running on Intel hardware.

Linear algebra (including implementations of BLAS and LAPACK), eigenvalue solvers, sparse system solvers, statistical and math functions, FFTs, Poisson solvers, non-linear optimization

Many of the routines are threaded. Easy way to get shared memory parallelism for running on a single node.

Easy to use. Just build executable with -mkl flag and add the appropriate include statement to your source (e.g. mkl.h)

[https://software.intel.com/en-us/mkl\\_11.1\\_ref](https://software.intel.com/en-us/mkl_11.1_ref)

# Profiling your code with gprof

gprof is a profiling tool for UNIX/Linux applications. First developed in 1982, it is still extremely popular and very widely used. It is always the first tool that I use for my work.

***Universally supported by all major C/C++ and Fortran compilers***

***Extremely easy to use***

1. Compile code with -pg option: adds instrumentation to executable
2. Run application: file named gmon.out will be created.
3. Run gprof to generate profile: gprof a.out gmon.out

***Introduces virtually no overhead***

***Output is easy to interpret***

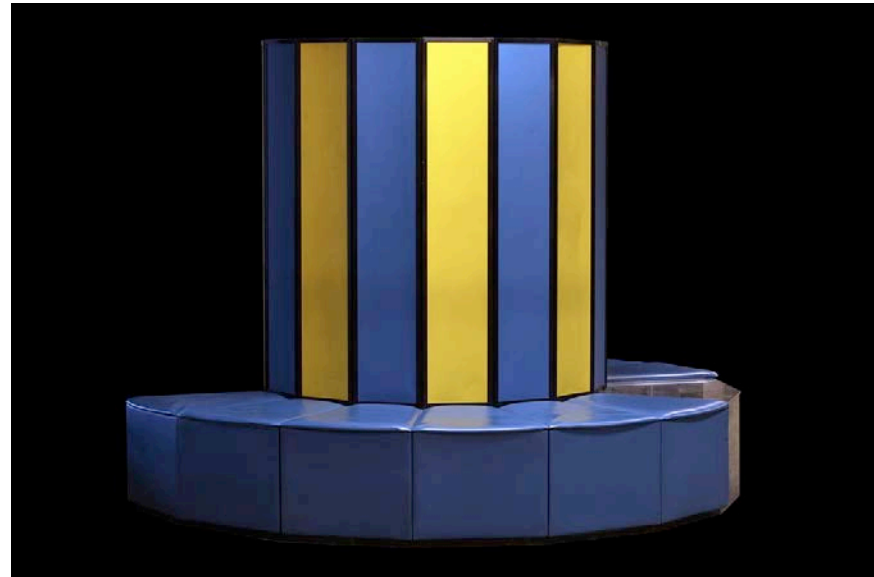


# 1982!

Worth reflecting on the fact that gprof goes back to 1982. Amazing when considered in context of the leading technology of the day



Michael Douglas as Gordon Gecko in Wall Street, modeling early 1980s cell phone. List price ~ \$3000



Cray X-MP with 105 MHz processor. High end configuration (four CPUs, 64 MB memory) has 800 MFLOP theoretical peak. Cost ~ \$15M

# gprof flat profile

The gprof flat profile is a simple listing of functions/subroutines ordered by their relative usage. Often a small number of routines will account for a large majority of the run time. Useful for identifying hot spots in your code.

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
68.60	574.72	574.72	399587	1.44	1.44	get_number_packed_data
13.48	687.62	112.90				main
11.60	784.81	97.19	182889	0.53	0.53	quickSort_double
2.15	802.85	18.04	182889	0.10	0.63	get_nearest_events
1.52	815.56	12.71				__c_mcopy8
1.28	826.29	10.73				_mcount2
0.96	834.30	8.02	22183	0.36	0.36	pack_arrays
0.12	835.27	0.97				__rouexit
0.08	835.94	0.66				__rouinit
0.06	836.45	0.51	22183	0.02	5.58	Is_Hump
0.05	836.88	0.44	1	436.25	436.25	quickSort

# gprof call graph

The gprof call graph provides additional levels of detail such as the exclusive time spent in a function, the time spent in all children (functions that are called) and statistics on calls from the parent(s)

```
index % time    self  children    called    name
[1]      96.9  112.90  699.04                main [1]
      574.72    0.00  399587/399587    get_number_packed_data [2]
      0.51   123.25   22183/22183      Is_Hump [3]
      0.44    0.00      1/1        quickSort [11]
      0.04    0.00      1/1      radixsort_flock [18]
      0.02    0.00      2/2      ID2Center_all [19]
-----
      574.72    0.00  399587/399587      main [1]
[2]      68.6  574.72    0.00   399587    get_number_packed_data [2]
-----
      0.51   123.25   22183/22183      main [1]
[3]      14.8   0.51   123.25   22183      Is_Hump [3]
      18.04   97.19  182889/182889    get_nearest_events [4]
      8.02    0.00   22183/22183    pack_arrays [8]
      0.00    0.00   22183/22183    pack_points [24]
```

# The value of re-profiling

After optimizing the code, we find that the function main() now accounts for 40% of the run time and would be a likely target for further performance improvements.

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
41.58	36.95	36.95				main
26.41	60.42	23.47	22183	1.06	1.06	get_number_packed_data
11.58	70.71	10.29				__c_mcopy8
10.98	80.47	9.76	182889	0.05	0.05	get_nearest_events
8.43	87.96	7.49	22183	0.34	0.34	pack_arrays
0.57	88.47	0.51	22183	0.02	0.80	Is_Hump
0.20	88.65	0.18	1	180.00	180.00	quickSort
0.08	88.72	0.07				_init
0.05	88.76	0.04	1	40.00	40.00	radixsort_flock
0.02	88.78	0.02	1	20.00	20.00	compute_position
0.02	88.80	0.02	1	20.00	20.00	readsource

# Limitations of gprof

- gprof only measures time spent in user-space code and does not account for system calls or time waiting for CPU or I/O
- gprof can be used for MPI applications and will generate a gmon.out.id file for each MPI process. But for reasons mentioned above, it will not give an accurate picture of the time spent waiting for communications
- gprof will not report usage for un-instrumented library routines
- In the default mode, gprof only gives function level rather than statement level profile information. Although it can provide the latter by compiling in debug mode (-g) and using the gprof -l option, this introduces a lot of overhead and disables many compiler optimizations.

In my opinion, I don't think this is such a bad thing. Once a function has been identified as a hotspot, it's usually obvious where the time is being spent (e.g. statements in innermost loop nesting)

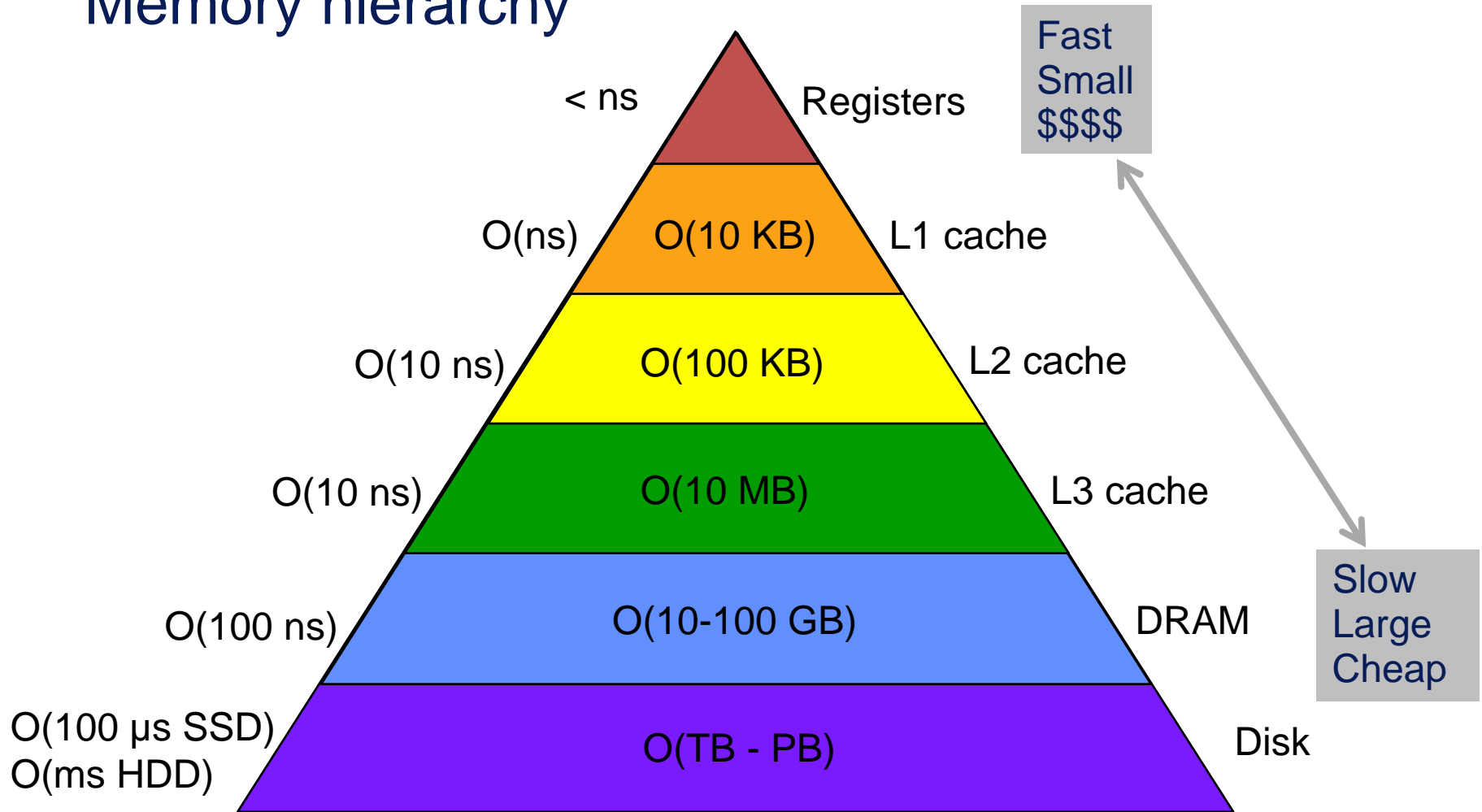
# Optimizing for cache

In order to obtain the best performance, it's essential to optimize your code to make best use of cache

In the next few slides, we'll cover

- Computer memory hierarchy
- The fundamental principles of cache management
- Layout of one-dimensional and multi-dimensional arrays
- Proper loop nesting
- Working with your compiler

# Memory hierarchy



# Cache essentials

**Temporal locality:** Data that was recently accessed is likely to be used again in the near future. To take advantage of temporal locality, once data is loaded into cache, it will generally remain there until it has to be purged to make room for new data. Cache is typically managed using a variation of the Least Recently Used (LRU) algorithm.

**Spatial locality:** If a piece of data is accessed, it's likely that neighboring data elements in memory will be needed. To take advantage of spatial locality, cache is organized into lines (typically 64 B) and an entire line is loaded at once.

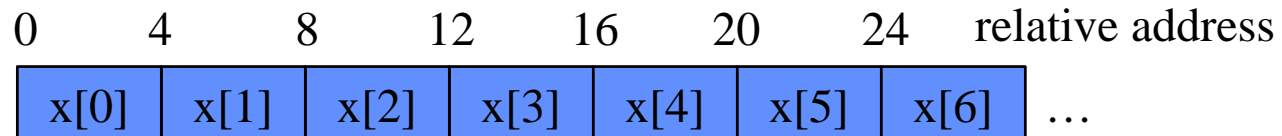
Our goal in cache level optimization is very simple – exploit the principles of temporal and spatial locality to minimize data access times



# One-dimensional arrays

One-dimensional arrays are stored as blocks of contiguous data in memory.

```
int *x, n=100;  
x = (int *) malloc(n * sizeof(int))
```



Cache optimization for 1D arrays is pretty straightforward and you'll probably write optimal code without even trying. Whenever possible, just access the elements in order.

```
for (int i=0; i<n; i++) {  
    x[i] += 100;  
}
```

# One-dimensional arrays

What is our block of code doing with regards to cache?

```
for (int i=0; i<n; i++) {  
    x[i] += 100;  
}
```

Assuming a 64-byte cache line and 4-byte integers:

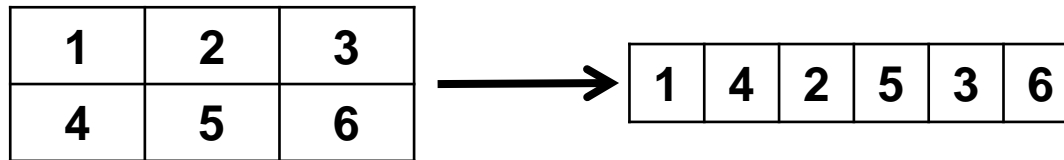
1. Load elements x[0] through x[15] into cache
2. Increment x[0] through x[15]
3. Load elements x[16] through x[31] into cache
4. Increment elements x[16] through x[31]
5. ...

In reality, the processor will do really clever things like recognizing the pattern of data access and **prefetching** the next cache line before it is needed.

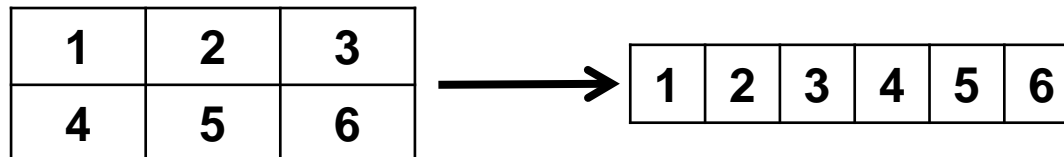
# Multidimensional arrays

From the computer's point of view, there is no such thing as a two-dimensional array. This is just syntactic sugar provided as a convenience to the programmer. Under the hood, array is stored as linear block of data

**Column-major order:** First or leftmost index varies the fastest. Used in Fortran, R, MATLAB and Octave (open-source MATLAB clone)



**Row-major order:** Last or rightmost index varies the fastest. Used in Python, Mathematica and C/C++



# Multidimensional arrays

## Properly written Fortran code

```
do j=1,n      ! Note loop nesting
  do i=1,n
    z(i,j) = x(i,j) + y(i,j)
  enddo
enddo
```

## Properly written C code

```
for (i=0; i<n; i++) { // Note loop nesting
  for (j=0; j<n; j++) {
    z[i][j] = x[i][j] + y[i][j]
  }
}
```

# Matrix addition exercise

- The dmadd\_good.f and dmadd\_bad.f files programs perform 2D matrix addition using optimal/non-optimal loop nesting. Arrays filled with random numbers ( $0 \leq x \leq 1$ ). Inspect the code and make sure you understand logic.
- Compile programs using the ifort compiler with default optimization level and explicitly stating -O0, -O1, -O2 and -O3
- On a Comet compute node, run with matrix ranks 30,000 (Programs accept single command line argument specifying the matrix rank)
- Examples: (feel free to use your own naming conventions)
  - ifort -xHost -o dmadd\_good\_df dmadd\_good.f
  - ifort -xHost -O3 -o dmadd\_bad\_O3 dmadd\_bad.f
  - ./dmadd\_good\_df 30000
  - ./dmadd\_bad\_O3 30000
- Keep track of the run times (reported by code)

# Matrix addition N = 30000

Optimization	dmadd_good	dmadd_bad
default	2.74	2.75
-O0	9.64	26.75
-O1	2.75	11.48
-O2	2.75	2.75
-O3	2.75	2.75

## Matrix addition exercise

- Try to explain the timings. Under what optimization levels was there a big difference between the optimal/non-optimal versions of the codes?
- What do you think the compiler is doing?
- Why do we have that mysterious block of code after the matrix addition? What possible purpose could it serve? \*
- Can you make an educated guess about the default optimization level?

```
if (z(1,1) == -1.0) then
  open (3,file="z.out",form="unformatted",access="sequential")
  write(3) z
endif
```

## Other loop-level optimizations

- Loop fusion
- Loop splitting
- Loop invariant optimizations
- Loop peeling
- Loop unrolling
- Loop blocking
- Breaking out of loops early
- Short loop optimizations



# Loop fusion

One of the most basic loop-level optimizations is loop fusion. Two or more loops with the same range of iterations are combined into a single loop

```
for (int i=0; i<n; i++) {  
    z[i] = x[i] + y[i]  
}  
for (int i=0; i<n; i++) {  
    w[i] = x[i] * y[i]  
}
```



```
for (int i=0; i<n; i++) {  
    z[i] = x[i] + y[i]  
    w[i] = x[i] * y[i]  
}
```

Sometimes the compiler can decide which is optimal (fused or not fused), but I've seen really obvious cases where it doesn't. My suggestion is to just try both and see which one is faster. Note that intervening code between the loops may prevent automatic fusion.

# Loop splitting

Loop splitting is the opposite of loop fusion. A single loop is split into two or more loops

```
for (int i=0; i<n; i++) {  
    z[i] = x[i] + y[i]  
    w[i] = x[i] * y[i]  
}
```



```
for (int i=0; i<n; i++) {  
    z[i] = x[i] + y[i]  
}  
for (int i=0; i<n; i++) {  
    w[i] = x[i] * y[i]  
}
```

Sometimes the compiler can decide which is optimal (fused or not fused), but I've seen really obvious cases where it doesn't. My suggestion is to just try both and see which one is faster.

# Loop fusion exercise

- Compile and run loop fusion example  
ifort -xHost -O3 -o fusion fusion.f  
./fusion 40000
- Look for obvious opportunity for loop fusion.
  - Can  $w(i)$  be computed at same time as  $z(i)$
  - Is the array  $z(i)$  even needed?
- Compare run times for original and modified codes
- If you have time, try compiling code with fused and unfused loops using -O1 and -O2 level of optimization

```
call system_clock(clock1)
do i=1,n*n
    z(i) = x(i) * y(i)
enddo

do i=1,n*n
    w(i) = z(i) + x(i) + y(i)
enddo
call system_clock(clock2)
```

## Loop fusion (N=40000)

optimization	original	Fused	Fused and eliminate array
-O1	11.40	8.68	4.88
-O2	11.29	8.42	4.90
-O3	8.47	8.48	4.88

# Loop invariant optimization

Pull repeated calculation out of loop and use the pre-calculated result in its place.

```
for (int i=0; i<n; i++) {  
    z[i] = x[i] + sqrt(c);  
}
```



```
sqrtc = sqrt(c);  
for (int i=0; i<n; i++) {  
    z[i] = x[i] + sqrtc;  
}
```

Compilers can often do this for you, particularly if the loops are simple. Still suggest that you do this yourself and be guaranteed that the optimization will be done. No real downsides.

# Loop invariant optimization

When working with nested loops, the invariants will sometime be less obvious and may even be a vector of results.

```
for (i=0; i<nx; i++) {  
  for (j=0; j<ny; j++) {  
    for (k=0; k<nz; k++) {  
      x2y2 = x[i]*x[i] + y[j]*y[j];  
      z2    = z[k] * z[k];  
      res[i][j][k] = exp(-a*z2) * sqrt(b*x2y2);  
    }  
  }  
}
```

x2y2 does not  
depend on index k

sqrt(b\*x2y2) does  
not depend on  
index k

# Loop invariant optimization

1. Moved calculation of  $x^2$  to outermost loop nesting  
Evaluated  $n_x$  times instead of  $n_x \cdot n_y \cdot n_z$
2. Moved calculation of  $\sqrt{x^2+y^2}$  out one level of nesting  
Evaluated  $n_x \cdot n_y$  times rather than  $n_x \cdot n_y \cdot n_z$

```
for (i=0; i<nx; i++) {  
    x2 = x[i]*x[i];  
    for (j=0; j<ny; j++) {  
        x2y2 = x2 + y[j]*y[j];  
        sqrtx2y2 = sqrt(b*x2y2);  
        for (k=0; k<nz; k++) {  
            z2 = z[k] * z[k];  
            res[i][j][k] = exp(-a*z2) * sqrtx2y2;  
        }  
    }  
}
```

z2 does not  
depend on indices  
i or j

$\exp(-a \cdot z2)$  does  
not depend on  
indices i or j

# Loop invariant optimization

Pre-calculate vector of  $\exp(-a \cdot z^2)$  results and reuse for every set of (i,j).  
Reduces number of exponential evaluations to nz from  $n_x \cdot n_y \cdot n_z$

```
for (k=0; k<nz; k++) {  
    zterm[k] = exp(-a*z[k]*z[k]);  
}  
  
for (i=0; i<nx; i++) {  
    x2 = x[i]*x[i];  
    for (j=0; j<ny; j++) {  
        x2y2 = x2 + y[j]*y[j];  
        sqrtx2y2 = sqrt(b*x2y2);  
        for (k=0; k<nz; k++) {  
            res[i][j][k] = zterm[k] * sqrtx2y2;  
        }  
    }  
}
```



# Loop invariant optimization

If our result involved  $\exp(-b * (x^2+y^2))$  instead of  $\sqrt{b*(x^2+y^2)}$ , we could have taken things one step further by pre-calculating three vectors. Innermost loop would only involve multiplications

```
for (i=0; i<nx; i++) { xterm[i] = exp(-b * x[i]*x[i]); }
for (j=0; j<ny; j++) { yterm[j] = exp(-b * y[j]*y[j]); }
for (k=0; k<nz; k++) { zterm[k] = exp(-a * z[k]*z[k]); }

for (i=0; i<nx; i++) {
    for (j=0; j<ny; j++) {
        for (k=0; k<nz; k++) {
            res[i][j][k] = xterm[i] * yterm[j] * zterm[k];
        }
    }
}
```

# Loop peeling

In a loop peeling optimization, one or more iterations are pulled out of the loop. Avoids unnecessary calculations associated with special iterations; also allows fusion of loops with slightly different iteration ranges

```
for (int i=0; i<n; i++) {  
    if (i == 0) {  
        z[i] = x[i] / y[i];  
    }  
    z[i] = x[i] + y[i]  
}  
for (int i=1; i<n; i++) {  
    w[i] = x[i] * y[i]  
}
```



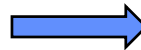
```
z[0] = x[0] / y[0];  
for (int i=1; i<n; i++) {  
    z[i] = x[i] + y[i]  
    w[i] = x[i] * y[i]  
}
```

The example above illustrates how peeling off the first iteration of the first loop (i=0) both avoids special case (product instead of sum) and allows fusion with the following loop

# Loop unrolling

Loop body is replicated and the stride is modified accordingly. This optimization can help the processor make better use of arithmetic functional units.

```
for (int i=0; i<1024; i++) {  
    z[i] = x[i] + y[i]  
}
```



```
for (int i=0; i<1024; i+=4) {  
    z[i]    = x[i]    + y[i]  
    z[i+1] = x[i+1] + y[i+1]  
    z[i+2] = x[i+2] + y[i+2]  
    z[i+3] = x[i+3] + y[i+3]  
}
```

Note that this example is particularly simple since the loop count is divisible by the unrolling depth. In general, you'll need to write cleanup code to handle the leftover iterations (remainder of  $n/\text{depth}$ ).

You will rarely beat the compiler and manual loop unrolling will make your code ugly and difficult to maintain. Best choice for unrolling depth may be processor architecture dependent.

# Loop unrolling

Although you'll rarely beat the compiler, sometimes you'll encounter a loop that is too complex for it to accurately analyze. Below is an example where manual loop unrolling by 4x did better than the compiler (original loop shown)


```
do i=0,4319,2 ! Unrolled loop → i=0,4319,8
  j0=mg63_miijj(0,i)
  j1=mg63_miijj(1,i)
  j2=mg63_miijj(2,i)
  i0=mg63_miijj(3,i)
  i1=mg63_miijj(4,i)
  i2=mg63_miijj(5,i)
  i3=mg63_miijj(6,i)
  pvi3jj(1 ) = pvi3jj(1 ) + d(i0,i1)*d(i0,i2)*d(i0,i3)*d(i0,j0)*d(i0,j1)
  pvi3jj(2 ) = pvi3jj(2 ) + d(i0,i1)*d(i1,i2)*d(i0,i3)*d(i0,j0)*d(i0,j1)
  pvi3jj(3 ) = pvi3jj(3 ) + d(i0,i1)*d(i1,i2)*d(i1,i3)*d(i0,j0)*d(i0,j1)
  ...
  pvi3jj(22) = pvi3jj(22) + d(i0,i2)*d(i0,i3)*d(i0,j0)*d(i1,j0)*d(j0,j1)
  pvi3jj(23) = pvi3jj(23) + d(i1,i2)*d(i0,i3)*d(i0,j0)*d(i1,j0)*d(j0,j1)
  pvi3jj(24) = pvi3jj(24) + d(i0,i2)*d(i2,i3)*d(i0,j0)*d(i1,j0)*d(j0,j1)
enddo
```

## Loop blocking (tiling, strip mining)


This technique is used when the arrays within a loop are accessed in different orders – one cache friendly and the other cache unfriendly

```
do j=1,n
  do i=1,n
    z(i,j) = x(i,j) + y(j,i)
  enddo
enddo
```

x <sub>11</sub>	x <sub>12</sub>	x <sub>13</sub>	x <sub>14</sub>	...
x <sub>21</sub>	x <sub>22</sub>	x <sub>23</sub>	x <sub>24</sub>	...
x <sub>31</sub>	x <sub>32</sub>	x <sub>33</sub>	x <sub>34</sub>	...
x <sub>41</sub>	x <sub>42</sub>	x <sub>43</sub>	x <sub>44</sub>	...
...	...	...	...	...



y <sub>11</sub>	y <sub>12</sub>	y <sub>13</sub>	y <sub>14</sub>	...
y <sub>21</sub>	y <sub>22</sub>	y <sub>23</sub>	y <sub>24</sub>	...
y <sub>31</sub>	y <sub>32</sub>	y <sub>33</sub>	y <sub>34</sub>	...
y <sub>41</sub>	y <sub>42</sub>	y <sub>43</sub>	y <sub>44</sub>	...
...	...	...	...	...



# Loop blocking (tiling, strip mining)

Block/tile one of the loops and interchange loop nesting so that we operate on blocks of code while they're in cache. This is a tricky optimization to get right since the best choice for tile size depends on the cache size, cache replacement strategies, problem size and other variables

```
do jj=1,n,tile
  do i=1,n
    do j=jj,min(n,jj+tile-1)
      z(i,j) = x(i,j) + y(j,i)
    enddo
  enddo
enddo
```

# Loop blocking exercise

- Compile and run the blocking example as follows  
`ifort -xHost -O3 -o blocking blocking.f`  
`./blocking 40000`
- Comment out the original code for the matrix addition and uncomment blocking example. Set different tile sizes (4-1024), recompile and run problem. Try to determine optimal tile size
- Note that this is not the most dramatic example of the impact of blocking. Is much more important for operations such as matrix multiplication where there is a lot more data reuse. Case chosen for simplicity.

## Blocking (N=40000)

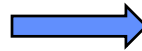
Tile size	t
No tiling	12.30
4	12.08
8	15.42
16	11.48
32	10.22
64	12.78
128	11.93
256	11.99



# Breaking out of loop early

Look for opportunities to break out of a loop early. This will generally require that you understand the semantics of your code

```
for (int i=0; i<n; i++) {  
    if (y[i] < const) {  
        // Do stuff  
    }  
}
```



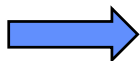
```
for (int i=0; i<n; i++) {  
    if (y[i] >= const) {  
        break;  
    } else {  
        // Do stuff  
    }  
}
```

In this simple example (taken from real-life application), I used my knowledge that the elements of array  $y$  are monotonically increasing ( $y[0] \leq y[1] \leq y[2] \leq y[3] \dots$ ). The compiler only understands the syntax of your code and cannot safely do this optimization for you.

## Short loop optimizations / special cases

If you expect that a loop body may be executed for a small number of iterations, can generate code for special cases.

```
for (dm=0; dm<num_dm; dm++) {  
    df = ngc[p][dm] - nd[i][dm];  
    dist = dist + df*df;  
}
```

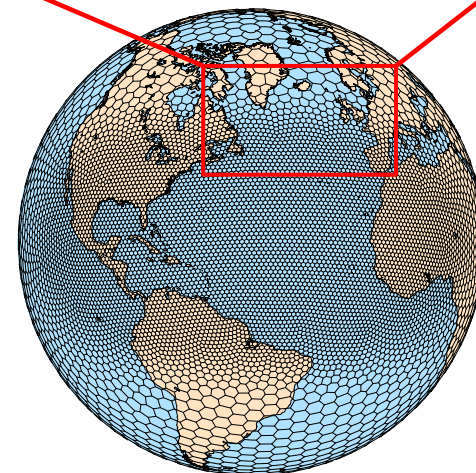
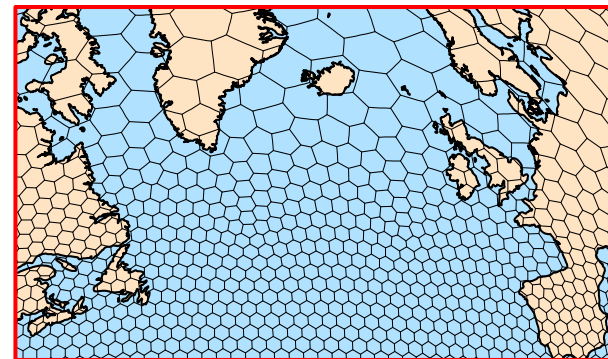
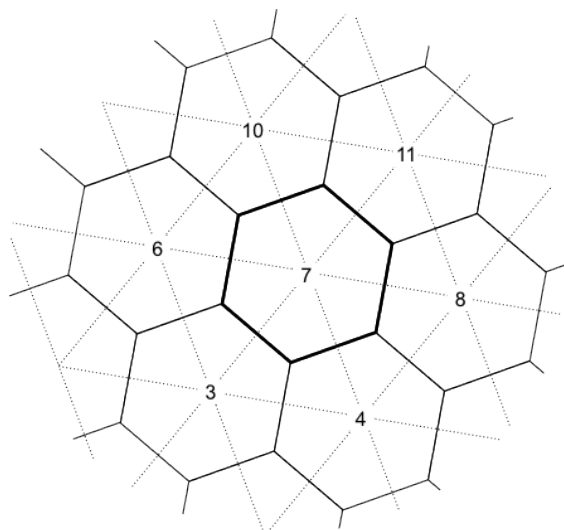


```
switch(num_dm)  
  
case 1:  
    df0 = ngc[p][0] - nd[i][0];  
    dist = df0*df0;  
    break;  
  
case 2:  
    df0 = ngc[p][0] - nd[i][0];  
    df1 = ngc[p][1] - nd[i][1];  
    dist = df0*df0 + df1*df1;  
    break;  
  
[Additional cases]  
  
default:  
    for (dm=0; dm<num_dm; dm++) {  
        df = ngc[p][dm] - nd[i][dm];  
        dist = dist + df*df;  
    }  
    break;
```

*Note - stripped down example taken from flow cytometry clustering code. In full application, loop is nested within two additional loops. May not have obtained much speedup as shown*

# MPAS-Atmosphere code

*The Model for Prediction Across Scales (MPAS) is a collaborative project between NCAR and LANL for developing atmosphere, ocean and other earth-system simulation components for use in climate, regional climate and weather studies.*



*Above: Centroidal Voronoi tessellations provide conformal meshes with smooth transitions between regions of differing resolution.*

# Short loop optimizations / special cases

Example taken from optimization of MPAS scalar advection routine. Optimized for the special (and overwhelmingly most common case) where cell is hexagon. Retained the original code as the default case

```
select case(nEdgesOnCell(iCell))
case(6)
  do k=1, nVertLevels
    s_max(k,iCell) = max(s_max(k,iCell), &
      scalar_old(k, cellsOnCell(1,iCell)), &
      scalar_old(k, cellsOnCell(2,iCell)), &
      scalar_old(k, cellsOnCell(3,iCell)), &
      scalar_old(k, cellsOnCell(4,iCell)), &
      scalar_old(k, cellsOnCell(5,iCell)), &
      scalar_old(k, cellsOnCell(6,iCell)))
  end do
case default
  do i=1, nEdgesOnCell(iCell)
    do k=1, nVertLevels
      s_max(k,iCell) = max(s_max(k,iCell), scalar_old(k, cellsOnCell(i,iCell)))
    end do
  end do
end select
```

## Select case gotcha

- When applying this type of optimization, may want to only do for a limited number of special/common cases. If the number of cases becomes too large, compiler may choke due to complexity of code.
- Limiting cases – especially those corresponding to rare conditions – keeps the code cleaner and easier to read
- This caught me by surprise since I expected that each case would have been optimized separately by the compiler
- Note that chained if ... else if statements are not a workaround. Appear to be translated in the same way as select case statement. Ditto for C/C++ switch case syntax

## Non cache/loop optimizations

- Although there is some overlap between loop and non-loop optimizations, these techniques are generally applied at the statement or whole program level.
- The compiler can do some of these for you (e.g. constant propagation). Others, such as force reduction and (especially) inter-procedural optimization will usually require that the programmer modify the code.
- Unless it makes your code excessively difficult to read or maintain, suggest that you do these yourself. Will avoid surprises when using different versions of compilers or optimization levels.

# Constant folding

Compiler recognizes constant expressions and evaluates at compile time rather than performing operations at run time. These are trivial for the compiler and there's no reason to do them yourself, especially if the original code is easier to understand

```
double precision pi = 2.0d0 * acos(0.0d0)
double precision e = exp(1.0d0)
integer gib = 2**30
```



```
double precision pi = 3.14159265358979
double precision e = 2.71828182845904
integer gib = 1073741824
```

# Constant propagation

Constants are propagated at compile time into other expressions that depend on them. Trivial for the compiler to do these, no reason to implement manually if it makes code more difficult to read.

```
int x = 17;  
int y = x + 12 - 3;  
int z = x + y;
```



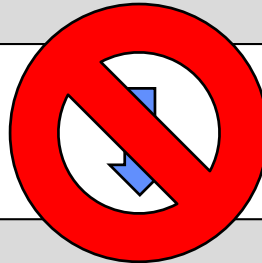
```
int x = 17;  
int y = 26;  
int z = 43;
```



# If the compiler does this, why are we even talking about it here?

Constant folding and propagation depend on the compiler having enough information to do the optimizations. If your program uses constants, be sure to code them explicitly.

```
myfunc(17, a);  
...  
void myfunc (int q, int a) {  
    int x = q;  
    int y = q + 12 - 3;  
    int z = q + y;
```



```
int x = 17;  
int y = 26;  
int z = 43;
```

# Force reduction

A force reduction optimization involves the replacement of an expensive operation with a less expensive one.

Exponentiation operations, especially floating point base raised to a floating point power, are particularly expensive. Look for opportunities to replace with multiplications, particularly if the exponent is known at compile time

<code>pow(x,8.0)</code>	→	<code>x2 = x*x; x4 = x2*x2; x8 = x4*x4</code>
<code>pow(x,1.5)</code>	→	<code>y = x * sqrt(x)</code>

Fortran and newer versions of C++ overload the `pow()` function. Use integer exponent whenever possible

double precision x,y	→	double precision x
<code>x**y</code>		integer n
		<code>x**n</code>

# Force reduction

Operations involving trig functions can often be simplified by using the trig identifies that you learned in high school. Just make sure that your transformation apply to all quadrants if applicable

$$\begin{aligned}\sin(x)*\cos(x) &\rightarrow 0.5 * \sin(2*x) \\ \sin(x)*\cos(y)+\cos(x)*\sin(y) &\rightarrow \sin(x+y)\end{aligned}$$

If a and b are fixed and sum needs to be calculated repeatedly for many values of x, can pre-calculate the constants c and phi.

$$\begin{aligned}a*\sin(x) + b*\cos(x) &\rightarrow \\ &c = \text{sqrt}(a*a + b*b) \\ &\text{phi} = \text{atan2}(b,a) \\ &c*\sin(x+\text{phi})\end{aligned}$$

# Force reduction

There are often hidden opportunities for force reductions. Look at logical tests that can be written in a more efficient way. Think about what results are really needed.

```
count = 0;
for (i=0; i<n; i++) {
    if (log(x[i]) < c) {
        count++;
    }
}
```



```
count = 0;
expc = exp(c)
for (i=0; i<n; i++) {
    if (x[i] < expc) {
        count++;
    }
}
```

In this example, we didn't really need to know the logarithm of  $x[i]$  and we could recast using a simple comparison to a pre-computed value.

# Force reduction exercise

- The disttest.c program generates a set of randomly distributed points in the unit square and calculates number that are separated by less than a specified tolerance. Inspect the code and make sure you understand logic.
- Compile program using the following options

```
icc -xHost -O3 -o disttest disttest.c
```

- Run executables on Comet compute node and keep track of run times as reported by code. Try  $n=10000$ ,  $50000$ ,  $100000$  and various values for tolerance

```
./disttest 50000 0.01
```

- Apply force reduction optimization to reduce the run time

```
for (i=0; i<n-1; i++) {  
    for (j=i+1; j<n; j++) {  
        r2 = (x[i]-x[j])*(x[i]-x[j]) + (y[i]-y[j])*(y[i]-y[j]);  
        if (sqrt(r2) < del) { count++; }  
    }  
}
```

## Force reduction

N	tol	original	Force reduction
50,000	0.01	3.15	0.64
100,000	0.01	12.60	2.65
50,000	0.1	3.15	0.64
100,000	0.1	12.60	2.65

# Short circuiting

Most imperative programming languages (MATLAB, Perl, Python, Java, Fortran, C/C++) use short circuit evaluation for compound logical tests

**Disjunctions** ('OR' tests) evaluate to TRUE once the first argument that evaluates to TRUE is encountered

**Conjunctions** ('AND' tests) evaluate to FALSE once the first argument that evaluates to FALSE is encountered

As a consequence, subsequent arguments are not evaluated once the final result is known. We can take advantage of this to write more efficient code.



# Short circuiting (disjunction)

Some good rules of thumb for ordering arguments ( $P \parallel Q$ )

1. If  $P$  and  $Q$  take roughly the same amount of time to evaluate, put the argument that is more commonly TRUE first

if (usually\_true  $\parallel$  usually\_false)

if ( $\sin(x) > 0.01 \parallel \cos(y) < 0.01$ ) //  $0 \leq x, y \leq \pi/2$  uniformly distributed

2. If  $P$  and  $Q$  are vastly different in the time required for evaluation, put the faster test first

if (fast\_test  $\parallel$  slow\_test)

if ( $x > y \parallel \text{pow}(x,y)/\text{atan2}(w,z) > \log(\text{sqrt}(x/y))$ )

# Short circuiting (conjunction)

Some good rules of thumb for ordering arguments ( $P \ \&\& \ Q$ )

1. If  $P$  and  $Q$  take roughly the same amount of time to evaluate, put the argument that is more commonly FALSE first

if (usually\_false && usually\_true)

if ( $\cos(y) < 0.01 \ \&\& \ \sin(x) > 0.01$ ) //  $0 \leq x, y \leq \pi/2$  uniformly distributed

2. If  $P$  and  $Q$  are vastly different in the time required for evaluation, put the faster test first

if (fast\_test && slow\_test)

if ( $x > y \ \&\& \ \text{pow}(x,y)/\text{atan2}(w,z) > \log(\text{sqrt}(x/y))$ )

# Avoid recalculating results

One of the easiest ways to reduce runtime is to avoid calculating a result multiple times. Sometimes the compiler can recognize this and pre-calculate the result

## Original

```
a = w + x*x + sqrt(y)
b = z + x*x + sqrt(y)
```

## Compiler will probably do this

```
temp = x*x + sqrt(y)
a = w + temp
b = z + temp
```

For user defined function, compiler needs to be careful of side effects and may not be able to safely perform the optimization

```
a = w + x*x + func(y)    // Did func change a global variable?
b = z + x*x + func(y)    // Will the 2nd call return the same result?
```

## Avoid recalculating results (cont.)

Pre-calculating results can have an even bigger impact when the result is calculated repeatedly in a loop body. This is known as a loop invariant optimization

```
for (i=0; i<n; i++) {  
    a[i] = b[i] + sqrt(c);  
}
```

The compiler will generally recognize simple invariants and pull outside of the loop. For the above example, the compiler will generate code like the following

```
sqrtc = sqrt(c);  
for (i=0; i<n; i++) {  
    a[i] = b[i] + sqrtc;  
}
```

# Summary

- There are many good reasons to optimize your code
  - Reduce time to solution, do more science, save energy
- But before you get started, make sure it's worth your effort
- Optimization can change results, ask yourself 'how critical is reproducibility?'
- Profile, optimize, repeat ...
- Take advantage of optimized libraries (especially MKL) and work of others
  - Good programmers write good code
  - Great programmers steal great code
- Know the capabilities and limitations of your compiler, but don't rely on the compiler to fix all your bad programming practices
- Optimizing for cache is critical – exploit spatial & temporal locality
- The biggest payoffs often come from a deep understanding of the semantics and structure of your code

---

# Additional materials

## Inter-procedural optimizations

# Inter-procedural optimizations

- Until now, we've mostly been focusing on optimizations at the loop or function level. Inter-procedural optimizations, which require considering the application as a whole
- Compilers are great at optimizing loops (inversion, unrolling, fusion, splitting, peeling ,etc.) and statements, but can rarely recognize opportunities for inter-procedural optimizations.
- These generally require an intimate understanding of your code.
- Very often, this optimization requires that you recognize operations that are repeated on the same set of data from one invocation of a function to the next.

# Inter-procedural optimization (example 1)

In a flow-cytometry code, noticed that a function was called five times in a row with slightly different sets of arguments (difs highlighted in red)

```
Ei=get_avg_dist(rpc[temp_i], temp_i, temp_j, population_ID,  
num_real_pop, file_Len, num_dm, norm_data, d1, d2, d3, size_c);  
  
Ej=get_avg_dist(rpc[temp_j], temp_i, temp_j, population_ID,  
num_real_pop, file_Len, num_dm, norm_data, d1, d2, d3, size_c);  
  
E1=get_avg_dist(center_1, temp_i, temp_j, population_ID,  
num_real_pop, file_Len, num_dm, norm_data, d1, d2, d3, size_c);  
  
E2=get_avg_dist(center_2, temp_i, temp_j, population_ID,  
num_real_pop, file_Len, num_dm, norm_data, d1, d2, d3, size_c);  
  
E3=get_avg_dist(center_3, temp_i, temp_j, population_ID,  
num_real_pop, file_Len, num_dm, norm_data, d1, d2, d3, size_c);
```



# Inter-procedural optimization (example 1)

Within the `get_avg_dist` function, the key loops involve a comparison between elements of `population_ID` and the scalars (`temp_i`, `temp_j`) to decide which elements of `norm_data` are used for the calculations. Recall that `center` is the only argument to change between calls and the same elements of `norm_data` are used all five times

```
get_avg_dist(center, temp_i, temp_j, population_ID,  
num_real_pop, file_Len, num_dm, norm_data, d1, d2, d3, size_c);
```

```
for (i=0; i<file_Len; i++) {  
    if (population_ID[i]==temp_i || population_ID[i]==temp_j) {  
        dist1 = center[d1] - norm_data[i][d1];  
        dist2 = center[d2] - norm_data[i][d2];  
        dist3 = center[d3] - norm_data[i][d3];  
        d = dist1*dist1 + dist2*dist2 + dist3*dist3;  
        if (d < radius) num_neighbors++  
    }  
}
```

# Inter-procedural optimization (example 1)

To avoid having to do the same tests five times in a row, do a “gather” operation to collect elements of packed data into an array and pass as argument to a modified `get_avg_dist`. Led to ~ 3x speedup of program.

```
npacked = 0;
for (i=0;i<file_Len;i++) {
    if (population_ID[i]==temp_i || population_ID[i]==temp_j){
        packed1[npacked] = norm_data[i][d1];
        packed2[npacked] = norm_data[i][d2];
        packed3[npacked] = norm_data[i][d3];
        npacked++;
    }
}
```

```
for (i=0; i<npacked; i++) {
    dist1 = center[d1] - packed1[i];
    dist2 = center[d2] - packed2[i];
    dist3 = center[d3] - packed3[i];
    d = dist1*dist1 + dist2*dist2 + dist3*dist3;
    if (d < radius) num_neighbors++
}
```

## Inter-procedural optimization (example 2)

In Latent Dirichlet Allocation code (identifies topics in free text), profiling shows that nearly all time spent in a single method

```
int model::sampling(int m, int n) {  
    int topic = z[m][n];  
    int w = ptrndata->docs[m]->words[n];  
    nw[w][topic] -= 1;  
    nd[m][topic] -= 1;  
    nwsum[topic] -= 1;  
    ndsum[m] -= 1;  
  
    for (int k = 0; k < K; k++) {  
        p[k] = (nw[w][k] + b) / (nwsum[k] + Vb) *  
              (nd[m][k] + a) / (ndsum[m] + Ka);  
    }  
  
    nw[w][topic] += 1;  
    nd[m][topic] += 1;  
    nwsum[topic] += 1;  
    ndsum[m] += 1;  
    return topic;  
}
```

```
for (int m=0; m<M; m++) {  
    for (int n=0; n<N; n++) {  
        int topic = sampling(m,n);  
        z[m][n] = topic;  
    }  
}
```

## Inter-procedural optimization (example 2)

Note that `sampling(m,n)` called repeatedly with same value of `m` and that only a few elements of `nd`, `nwsum` and `ndsum` are (temporarily) updated

```
int model::sampling(int m, int n) {  
    int topic = z[m][n];  
    int w = ptrndata->docs[m]->words[n];  
    nw[w][topic] -= 1;  
    nd[m][topic] -= 1;  
    nwsum[topic] -= 1;  
    ndsum[m] -= 1;  
  
    for (int k = 0; k < K; k++) {  
        p[k] = (nw[w][k] + b) / (nwsum[k] + Vb) *  
               (nd[m][k] + a) / (ndsum[m] + Ka);  
    }  
  
    nw[w][topic] += 1;  
    nd[m][topic] += 1;  
    nwsum[topic] += 1;  
    ndsum[m] += 1;  
    return topic;  
}
```

```
for (int m=0; m<M; m++) {  
    for (int n=0; n<N; n++) {  
        int topic = sampling(m,n);  
        z[m][n] = topic;  
    }  
}
```

Potential  
invariants

## Inter-procedural optimization (example 2)

Pre-calculate array of values that do not change (much) across successive calls to sampling and update only necessary elements

```
int model::sampling(int m, int n) {
    int topic = z[m][n];
    nd[m][topic] -= 1;
    nwsum[topic] -= 1;
    f1[topic] = (nd[m][topic] + a) /
                ((nwsum[topic] + Vb)*
                 (ndsum[m] - 1.0 + Ka));

    for (int k = 0; k < K; k++) {
        p[k] = (nw[w][k] + b) * f1[k];
    }

    nd[m][topic] += 1;
    nwsum[topic] += 1;
    f1[topic] = ...;
}
```

```
for (int m=0; m<M; m++) {
    for (int k = 0; k < K; k++) {
        f1[k] = (nd[m][k] + a) /
                ((nwsum[k] + Vb)*
                 (ndsum[m] - 1.0 + Ka));
    }
    for (int n=0; n<N; n++) {
        int topic = sampling(m,n);
        z[m][n] = topic;
    }
}
```

Entire application  
is now 1.5-2.2x  
faster, depending  
on number of topics

---

# Additional materials

## Unique / one-off optimizations



SAN DIEGO SUPERCOMPUTER CENTER

*at the* UNIVERSITY OF CALIFORNIA; SAN DIEGO



# Unique optimizations

- Standard techniques will take you a long way, but sometimes you get the biggest payoffs from novel, one-off optimizations
- These are also the most fun optimizations. To me, it's like getting paid to do brain ticklers
- Often requires a more intimate understanding of your application
- Once you've identified your hotspot, single-mindedly focus your efforts on a better, faster solution
- Hard to provide concrete advice since the optimizations tends to be very problem specific. In many cases though, they require little advanced knowledge beyond high school algebra, trig and geometry

# Approximate expensive function and redo accurate calculation only when necessary

Fortran application was spending most of its time calculating inverse cosine (acos) function. Result used in test that is rarely satisfied

```
if (acos(xprod) < abs(xi-xj)) then
  -- do some calculations --
endif
```

Google search found an inexpensive approximation (20x faster) to inverse cosine ( $\pi/2 - Ax^5 - Bx$ ) with a known maximum error. Use for initial test and recalculate acos only when needed. Note that this has no impact on final results

```
if(acos_approx(xprod) < abs(xi-xj) + max_err) then
  if(acos(xprod) < abs(xi-xj) then
    -- do some calculations --
  endif
endif
```



# Fast calculation of sum over logs

Many problems in bioinformatics, statistical physics and other fields require the calculation of log probabilities. The direct product over small probabilities results in underflow, so we need to calculate sum over logs instead

*Want  $\log(\prod_{i=1}^n p_i)$ , but  $\prod_{i=1}^n p_i$  underflows*

*Instead, calculate  $\sum_{i=1}^n \log(p_i)$*

The downside is that the latter is  $n$  times more expensive. This can have a big impact on performance if  $n$  is large and/or log probabilities are frequently calculated

## Fast calculation of sum over logs (cont.)

To avoid the expensive logarithm calculations, first split arguments into normalized fractions ( $1/2 \leq x < 1$ ) and powers of two. Can do this with the very fast C frexp function. Then accumulate product over fractions, sum over powers of two and do a little algebra at the end

$$\begin{aligned} 5 \times 17 \times 37 &= (0.625 \times 2^3) \times (0.53125 \times 2^5) \times (0.578125 \times 2^6) \\ &= (0.625 \times 0.53125 \times 0.578125) \times 2^{14} \\ &= 0.1919556 \times 2^{14} \end{aligned}$$

$$\log(5 \times 17 \times 37) = \log(0.1919556) + \log(2.0) \times 14 = 3.49762 \checkmark$$

## Fast calculation of sum over logs (cont.)

What if the product over fractions underflows? Multiply by constant to keep product close to one and correct for this later. Assumes that the fractional parts of the arguments are uniformly distributed between  $\frac{1}{2}$  and 1. Here's the final solution

```
c=1.358858;
sprod = 1.0;
xsum = 0.0;

for (i=0; i<n; i++) {
    s = frexp(p[i], &x); // Split into fraction and power of 2
    sprod *= (s * c);    // Product over fractions, with correction
    xsum += x;           // Sum over powers of two
}
logsum = log(sprod) + log(2.0)*xsum - n*log(c);
```

# log sum exercise

- The logsum.c program performs log-sum using both standard and optimized method. Inspect the code and make sure you understand logic.
- Compile program using the following options

```
gcc -O3 -o logsum_gcc logsum.c -lm  
icc -O3 -o logsum_icc logsum.c
```

- Run executables on Comet compute node and note run times for the original and modified log-sum calculations.

## log sum exercise

- Are there any noticeable differences between the timings for the Intel (icc) and GNU (gcc) compilers?
- Did you notice any trends in the ratio of run times as a function of problem size?
- If you noticed a difference between the compilers why do you think this happened?

# Linear algebra - DSYRK

After initial rounds of optimization, determined that application was spending most of its time in the DSYRK linear algebra routine

$$D \leftarrow \alpha AA^T + \beta C \quad (\text{A and C matrices, } \alpha \text{ and } \beta \text{ constants})$$

Figured that there was no room left for improvement, then took a more careful look at the way the matrix A is constructed from the concatenation of two matrices

*Rewrite A as  
concatenation of  
two matrices B  
and C*

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} b_{11} & b_{12} & c_{11} & c_{12} \\ b_{21} & b_{22} & c_{21} & c_{22} \\ b_{31} & b_{32} & c_{31} & c_{32} \\ b_{41} & b_{42} & c_{41} & c_{42} \end{bmatrix} = \begin{bmatrix} B & C \end{bmatrix}$$

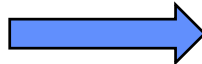
*$AA^T$  can be  
expressed as  
 $BB^T + CC^T$*

$$AA^T = \begin{bmatrix} B & C \end{bmatrix} \begin{bmatrix} B & C \end{bmatrix}^T = \begin{bmatrix} B & C \end{bmatrix} \begin{bmatrix} B^T \\ C^T \end{bmatrix} = BB^T + CC^T$$

## Linear algebra - DSYRK (cont.)

Further inspection revealed that the nested loops over indices provides opportunities for pre-calculating the partial results ( $BB^T$ ,  $CC^T$ ) and replacing the DSYRK call with a much faster matrix addition.

```
Loop over i (1:n)
  Loop over j (1:n)
     $A \leftarrow [B_i \ C_j]$ 
     $R \leftarrow \text{DSYRK}(A)$ 
    ...
```



```
Loop over i (i:n)
   $X_i \leftarrow B_i B_i^T$ 
   $Y_i \leftarrow C_i C_i^T$ 
  Loop over j (i:n)
    Loop over j (i:n)
       $R \leftarrow X_i + Y_j$ 
    ...
```

---

# Additional materials Reproducibility



# Reproducibility of optimized codes

- Bit-wise reproducibility means obtaining exactly the same binary (internal machine representation) results
- An ASCII dump (formatted output) may look the same but can hide differences in the floating point representation if too few significant digits are printed
- Think about whether or not you really need bit-wise reproducibility and under which conditions (compiler, hardware, processor count). Can be done, but not without tradeoffs.
- Decide how much accuracy is needed. Is it acceptable to get a result that is correct to within a specified tolerance? Consider constructing a test suite that can be used to test reproducibility.

# Confirming bit-wise reproducibility

md5sum can confirm that results are exactly the same. Uses a 128-bit cryptographic hash function to generate digital fingerprint of file. Hash collisions are possible, but probability is astronomically low.

```
$ ls -lh *
```

```
-rw-r--r-- 1 sinkovit use300 6.1M May  2 08:31 fleeting_ref_AAPL_050610.csv  
-rw-r--r-- 1 sinkovit use300 101M May  2 08:31 msg_AAPL_050610.csv  
-rw-r--r-- 1 sinkovit use300  83M May  2 08:31 settled_AAPL_050610.csv
```

```
$ md5sum *
```

```
d7dcee609d3536d072875856d1a0c253 fleeting_ref_AAPL_050610.csv  
f8655644fb37eedd1c30b8e58fe79d50 msg_AAPL_050610.csv  
b77eaed47a1dc94eb75efb1d2a32432d settled_AAPL_050610.csv
```

# Looking for differences in ASCII files

md5sum is a great way to tell if files are identical, but is useless if there are even small differences between runs. The Linux diff utility can be used instead, provides details of the differences.

```
$ wc -l file1 file2
```

```
464691 file1
```

Files are exactly the same length (good sign)

```
464691 file2
```

```
$ md5sum file1 file2
```

```
13b71bb7b8274c1657b815735046e411 file1
```

Ugh, different md5sums

```
0234c9a3dbc4b94ade7822edc3ae2f61 file2
```

```
$ diff file1 file2
```

```
1c1
```

```
< Fri Jun 27 12:36:02 PDT 2014
```

Different time stamps

```
> Fri Jun 27 12:35:46 PDT 2014
```

```
464691c464691
```

```
< Run time: 1236.78 seconds
```

Different run times

```
> Run time: 1234.56 seconds
```

# Looking for differences in ASCII files

If the **acceptable changes** (e.g. time stamps, run times, node names) between output files occur in predictable formats and or locations, we can make clever use of sed, grep, head, tail and other utilities to build more complex tests

```
$ diff file1 file2
```

```
1c1
```

```
< Fri Jun 27 12:36:02 PDT 2014
```

First line of output

```
> Fri Jun 27 12:35:46 PDT 2014
```

```
464691c464691
```

```
< Run time: 1236.78 seconds
```

Last line of output / only line containing string 'Run time'

```
> Run time: 1234.56 seconds
```

```
$ sed -n '1!p' file1 | grep -v 'Run time' | md5sum
```

```
3169c7872c74b2e1593dcde1f4d7f2be -
```

```
$ sed -n '1!p' file2 | grep -v 'Run time' | md5sum
```

```
3169c7872c74b2e1593dcde1f4d7f2be -
```

# Using diff on directories

We can use diff to recursively compare the contents of entire directories as long as the files are named identically.

-r = recursive --brief = only report whether files differ

This will work for both plain text and binary files

```
$ diff -r --brief DIR1/ DIR2/
```

```
$
```

no output – directory contents identical

```
$ diff -r --brief DIR1 DIR2
```

```
Only in DIR2: file1
```

Unique to DIR2

```
Only in DIR1: file2
```

Unique to DIR1

```
Files DIR1/fleeting_ref_AAPL_050610.csv
```

Files differ – no details provided

```
and DIR2/fleeting_ref_AAPL_050610.csv differ
```

# A very brief intro to sed, grep, head, tail

**grep** prints the lines in a file that match (or don't match) a particular pattern

```
$ grep bird file1 # Has bird
line2 bird dog cat
line3 fish cat bird
$ grep -v dog file1 # Not has dog
line3 fish cat bird
```

**sed** is a powerful stream editor that (among many other capabilities) selects lines by record number

```
$ sed -n '2p' file1 # Line 2
line2 bird dog cat
$ sed -n '2!p' file1 # All but line 2
line1 dog fish cat
line3 fish cat bird
```

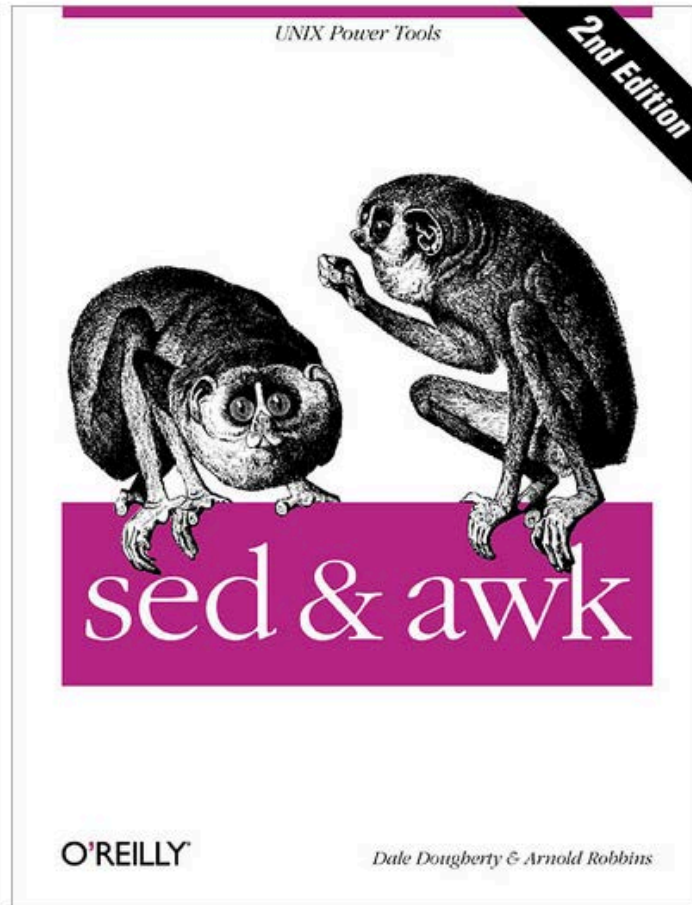
**head** prints the top of a file

```
$ head -n1 file1 # First line
line1 dog fish cat
$ head -n2 file1 # First two lines
line1 dog fish cat
line2 bird dog cat
```

**tail** prints the bottom of a file

```
$ tail -n1 file1 # Last line
line3 fish cat bird
$ tail -n2 file1 # Last two lines
line2 bird dog cat
line3 fish cat bird
```

## Some other resources on ask, sed and grep



<http://www.thegeekstuff.com/2009/03/15-practical-unix-grep-command-examples/>

<http://sed.sourceforge.net/sed1line.txt>

<http://www.hcs.harvard.edu/~dholland/computers/awk.html>

# Reproducibility in integer/string codes

Integers and characters are represented exactly. The same program should give the same results on any system using any compiler. When working with integers, just need to be aware of a few potential gotchas when modifying your software

(1) Division results are truncated. As a consequence, some basic arithmetic identities are not integer math identities

$$(a/b) + (c/d) \neq (ad + bc)/bd$$

$$(2/3) + (5/2) = 0 + 2 = 2$$

$$(2 \bullet 2 + 5 \bullet 3) / (2 \bullet 3) = 19/6 = 3$$

(2) Avoid modifications to order of operations that might result in overflows

$$\Sigma(\text{neg terms}) + \Sigma(\text{pos terms}) = (?) \Sigma(\text{all terms})$$



# Reproducibility in floating point codes

FP operations are subject to round-off error and seemingly trivial code modifications or changes to run conditions can change the answers. If any of the following lead to **significantly different** results, you should re-examine your algorithms

(1) Arithmetic identities that are not necessarily floating point identities.

$$\begin{aligned}(a + b) + c &\neq a + (b + c) \\ (a/b) + (c/d) &\neq (ad + bc) / bd \\ \text{sqrt}(\text{sqrt}(a)) &\neq a^{0.25}\end{aligned}$$

(2) Software parallelization, particularly involving global reduction operations (e.g. summing over elements of an array). The exact answers may depend on the number of threads and/or processes.

(3) Aggressive compiler optimization (typically -O3 and higher) may lead to code modifications that do not preserve bit-wise reproducibility

(4) Running on different processor architectures or linking different library versions

# Random number generation

Many applications rely on random number generators to set the initial conditions or perform Monte Carlo simulations. If developing your own software, do yourself a big favor and provide the capability to **set the seed**.

Can make this an optional argument that overrides the default behavior. Otherwise, you'll never be sure that the modified version of the software is correct

```
$ ./a.out -i infile -o outfile ... [-seed 1234]
```

Seems obvious, but surprising how many codes implement something like the following without documenting behavior

```
srandom(time(0))
```