Optimizing scientific codes: profiling and optimizing

Antun Balaž
Scientific Computing Laboratory
Institute of Physics Belgrade
http://www.scl.rs/

27 Feb 2012

Advanced School on Scientific Software Development, ICTP, Trieste, 20 Feb – 2 Mar 2012

Overview

- Performance
  - Evaluation
  - Timing and profiling
  - gprof
- Optimization techniques

Motivation (1)

- Real processors have registers, cache, parallelism - they are complicated!
- Why is this your problem?
  - In theory, compilers understand all of this and can optimize your code
  - Generally optimizing algorithms across all computational architectures is an impossible task, hand optimization will always be needed.
- We need to learn how…
  - to measure performance of codes on modern architectures
  - to tune performance of the codes by hand (32/64 bit commodity processors)

Motivation (2)

- When you are charged with optimizing an app
  - Don't optimize the whole code
    - Profile the code, find the bottlenecks
    - They may not always be where you thought they were
  - Break the problem down
    - Try to run the shortest possible test you can to get meaningful results
    - Isolate serial kernels
  - Keep a working version of the code
    - Getting the wrong answer faster is not the goal.
  - Optimize on the target architecture
    - Optimizations for one architecture will not necessarily translate
  - The compiler is your friend!
    - If you find yourself coding in machine language, you are doing too much

Performance

- The peak performance of a chip
  - The number of theoretical floating point operations per second
  - Example: 2.4 Ghz Opteron can theoretically do 2 floating point operations per cycle, for a peak performance of 4.8 Gflops
- Real performance
  - Algorithm dependent, the actually number of floating point operations per second
    - Generally, most programs get about 10% or lower of peak performance
    - 40% of peak, and you can go on holiday
- Parallel performance
  - The scaling of an algorithm relative to its speed on one CPU (core)

Performance evaluation

- Monitoring System
  - Observe both overall system performance and single-program execution characteristics
    - Look to see if the system is doing well and what percentage of the resources your program is using.
    - Pro: easy
    - Con: not very detailed
- Profiling and Timing the code
  - Timing a whole programs (/usr/bin/time)
  - Timing portions of the program (code modification)
  - Profiling

SCIENTIFIC COMPUTING LABORATORY

Useful monitoring commands on Linux

- **uptime** - information about system usage and user load
- **ps** - lets you see a “ snapshot” of the process table
- **top** - process table dynamic display
- **free** - memory usage
- **vmstat** - memory usage monitor

Swapping: your worst nightmare

- Virtual or swap memory
  - This memory is actually space on the hard drive
  - The operating system reserves a space on the hard drive for “ swap space”
- Time to access virtual memory VERY large
- This time is attributed to the system, not to your program, and you can observe it e.g. as a difference in wall clock time and CPU time

Monitoring your code

- `man time`

NAME

time - time a simple command or give resource usage

SYNOPSIS

time [options] command [arguments...]

DESCRIPTION

The **time** command runs the specified program command with the given arguments. When command finishes, **time** writes a message to standard error giving timing statistics about this program run. These statistics consist of (i) the elapsed real time between invocation and termination, (ii) the user CPU time (the sum of the tms_utime and tms_cutime values in a struct tms as returned by **times**(2)), and (iii) the system CPU time (the sum of the tms_stime and tms_cstime values in a struct tms as returned by **times**(2)).

Note: some shells (e.g., **bash**(1)) have a built-in **time** command that provides less functionality than the command described here. To access the real command, you may need to specify its pathname (something like /usr/bin/time).

```
[antun@n01 th]$ time ./imagtime1d > out.txt

real    0m36.523s
user    0m36.514s
sys     0m0.001s
[antun@n01 th]$
```

- user time: CPU time dedicated to your program
- sys time: time used by your program to execute system calls
- real time: total time - walltime

Timing a portion of the code

- Most programming languages provide a means to access the systems own timing functions
- C function: clock

```
clock_t c0, c1;
c0 = clock();
section of the code…
c1= clock();
cputime = (c1 - c0)/(CLOCKS_PER_SEC );
```

- Fortran subroutine: cpu_time

```
call cpu_time(t0)
section of the code…
call cpu_time(t1)
cputime = t1 – t0
```

SCIENTIFIC COMPUTING LABORATORY

Timing is a good programming practice!

- Good application writers will take full advantage of these to give users insight into code performance
- Therefore, when writing an application, you should consider timing of critical parts of the program and printing timing info as a part of its output

Profiling (1)

- Profiling is an approach to performance analysis in which the amount of time spent in sections of code is measured (using either a sampling technique or on entry/exit of a code block) and presented as a histogram
- Allows a developer to target key time consuming portions of codes
- Profiling can be done at varied levels of granularity
  - Function/subroutine, code block, loop and source code line

Profiling (2)

- Profiling: investigation of program behavior using run- time information
- Profiler: conceptual module that collects/ analyzes run- time data
- Profile: a set of frequencies associated with run-time events

Hardware Performance Counters

- Most modern processors have one or more registers dedicated to count low level hardware information
  - e.g. floating point operations, L1 cache misses, etc.
- This information is really useful to understand at a very fine grain of detail what a program is doing on the architecture
- PAPI (Performance API)
  - The API provides function handles for setting and accessing these counters
  - http://icl.cs.utk.edu/papi/

Profiler implementations

- Hybrid (HW-assisted)
  - Hardware Performance Monitors (HPMs)2.
  - Dedicated HW collectors that deliver data to SW module
  - Fixed, low-overhead
- Software – Pure software implementations
  - Portable, flexible, high-overhead

GCC profiling and gprof

- Simple gcc compiler flags can be used to get profiling information
  - Great place to start
- GNU:
  - -p Generate extra code to write profile information suitable for analysis program prof
  - -pg Generate extra code to write profile information suitable for analysis by program gprof.
- Procedure
  - gcc -pg prog.c -o prog
  - ./prog
  - gprof prog gmon.out

Example for gprof

```
[antun@n01 th]$ gprof imagtime1d gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 74.91     26.87    26.87                             main
 16.29     32.71     5.84   100001    58.42    58.42  normalize
  3.26     33.88     1.17 999920000     0.00     0.00  V
  0.94     34.22     0.34                             frame_dummy
  0.00     34.22     0.00        2     0.00    11.69  muE
```

SCIENTIFIC COMPUTING LABORATORY

Many advanced tools available

- TAU is a portable profiling and tracing toolkit for performance analysis of parallel programs
  - http://www.cs.uoregon.edu/research/tau/home.php
- Intel VTune
  - http://en.wikipedia.org/wiki/VTune
- TotalView
  - http://en.wikipedia.org/wiki/TotalView
- Profiling and optimization tuning tools intermingle!

SCIENTIFIC COMPUTING LABORATORY

Optimization flow-chart

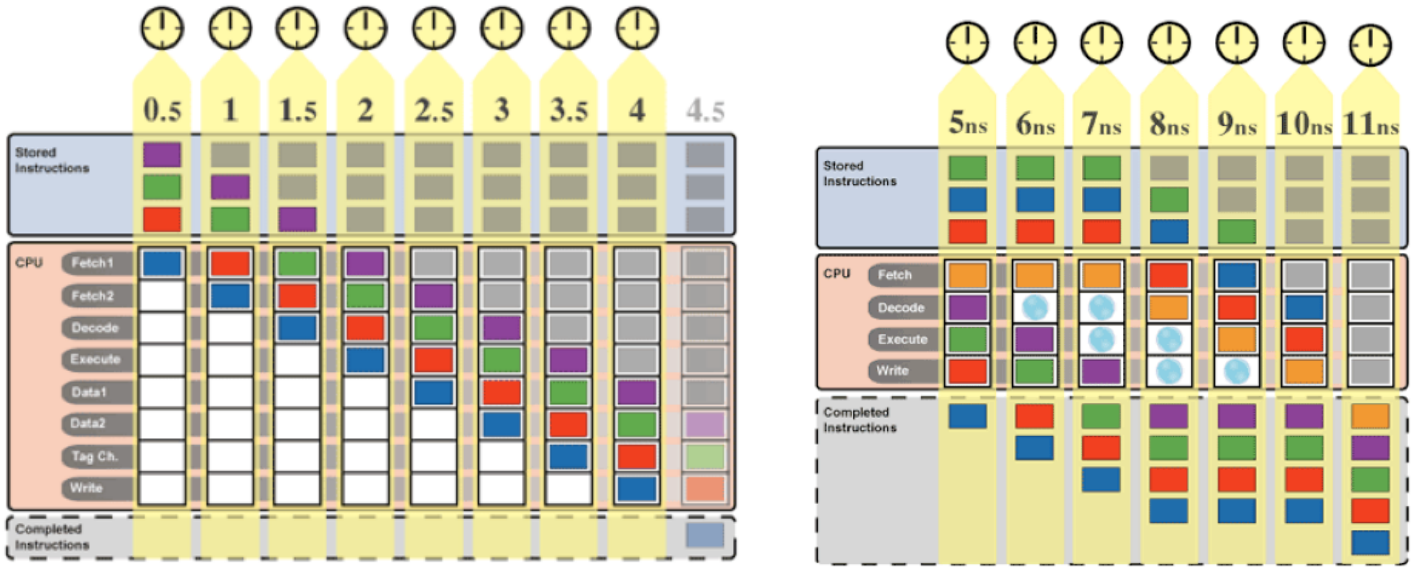
- Profiling of the code
- Identification of bottlenecks
- Optimizing of one loop/function at a time
  - Starting with the most time consuming functions (that is why we profile)
  - Then the second and the third one
- Parallelizing of the program
  - Then we can work on improving the parallel performance (communication, load balancing, etc..)

Optimization techniques

- Improve memory performance (taking advantage of locality)
  - Better memory access patterns
  - Optimal usage of cache lines
  - Re-use of cached data
- Improve CPU performance
  - Reduce flop count
  - Better instruction scheduling
  - Use optimal instruction set
- Use of highly optimized numerical libraries

Pipelining

- Pipelining allows for a smooth progression of instructions and data to flow through the processor
- Any optimization that facilitate pipelining will speed the serial performance of your code
- As chips support more SSE like character, filling the pipeline is more difficult.
- Stalling the pipeline slows codes down
  - Out of cache reads and writes; Conditional statements

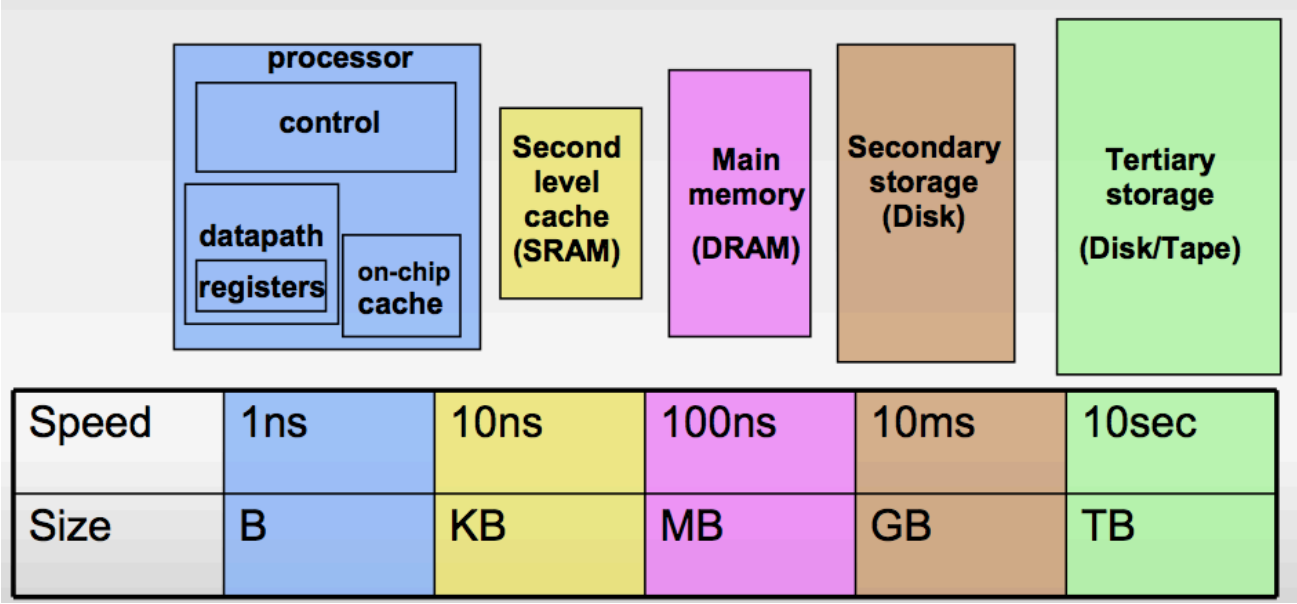


SCIENTIFIC COMPUTING LABORATORY

27 Feb 2012

Memory locality (1)

- Effective use of the memory hierarchy can facilitate good pipelining
- Temporal locality:
  - Recently referenced items (instr or data) are likely to be referenced again in the near future
  - iterative loops, subroutines, local variables
  - working set concept

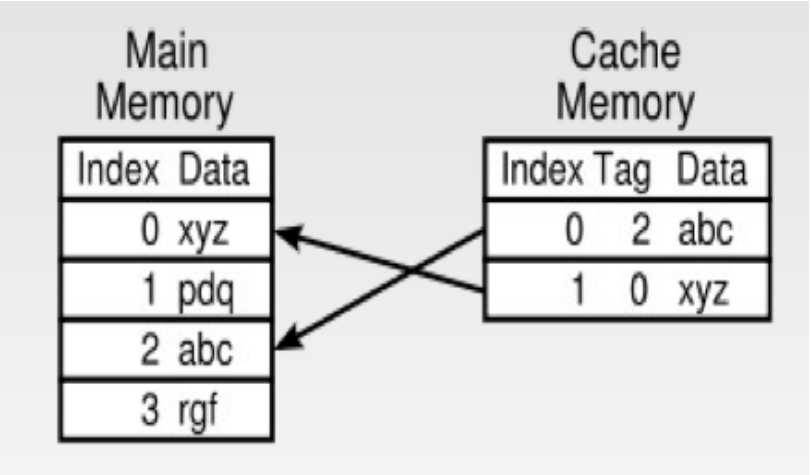


Memory locality (2)

- Spatial locality:
  - programs access data which is near to each other
  - operations on tables/arrays
  - cache line size is determined by spatial locality
- Sequential locality:
  - processor executes instructions in program order
  - branches/in-sequence ratio is typically 1 to 5

Caching

- CPU cache is generally set up as a series of lines that can pull in a specified amount of data a given time
- Accessing cache is infinitely faster than the main memory
- Get as much data in at a time
- Use that data to its fullest



Optimization Techniques for Memory

- Strides - contiguous blocks of memory
- Accessing memory in stride greatly enhances the performance
- Array indexing

```
Do j=1,M
    Do i=1,N
      ..A(i, j)
    END DO
END DO
```
Direct

```
Do j=1,M
    Do i=1,N
      ..A(i+(j-1)*N)
    END DO
END DO
```
Explicit

```
Do j=1,M
    Do i=1,N
       k=k+1
      ..A(k)
    END DO
END DO
```
Loop carried

```
Do j=1,M
    Do i=1,N
      ..A(index(i,j))..
    END DO
END DO
```
Indirect

Loop interchange (1)

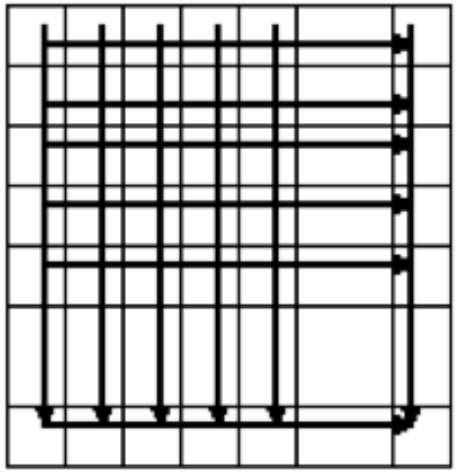
- Basic idea: change the order of data independent nested loops.
  - Advantages:
    - Better memory access patterns (leading to improved cache and memory usage)
    - Elimination of data dependencies (to increase opportunity for CPU optimization and parallelization
  - Disadvantage:
    - May make a short loop innermost
- Usually, compilers cannot do this

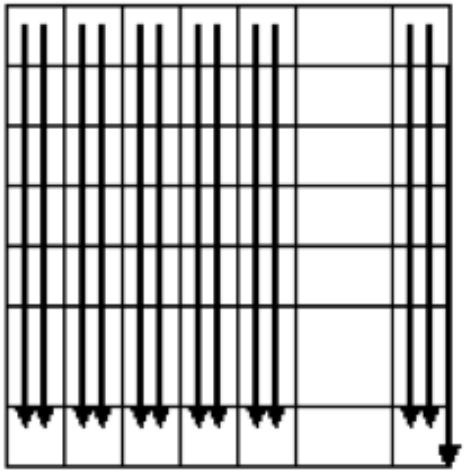
Loop interchange (2)

- Fortran:

```
DO i=1,N
   DO j=1,M
     C(i,j)=A(i,j)+B(i,j)
   END DO
END O
```

```
DO j=1,M
   DO i=1,N
     C(i,j)=A(i,j)+B(i,j)
   END DO
END DO
```





Loop interchange (3)

```
DO i=1,300
   DO j=1,300
      DO k=1,300
         A (i,j,k) = A (i,j,k)+ B (i,j,k)* C (i,j,k)
      END DO
   END DO
END DO
```

Loop order	Execution time (Intel, 2.4 GHz)
i, j, k	8.77
i, k, j	7.61
j, i, k	2.00
j, k, i	0.57
k, i, j	0.90
k, j, i	0.44

Loop unrolling (1)

- Computation is cheap, while branching is very expensive
- Loops, conditionals, etc. cause branching instructions to be performed:

```
for(i = 0; i < N; i++) {
   do something useful(i);
}
```

- Each time `for` statement is hit, a branching instruction is called
- Therefore, (partially or fully) unrolling a loop may be (highly) beneficial

Loop unrolling (2)

```
do i=1,N
   a(i)=b(i)+x*c(i)
enddo
```

```
do i=1,N,4
   a(i)=b(i)+x*c(i)
   a(i+1)=b(i+1)+x*c(i+1)
   a(i+2)=b(i+2)+x*c(i+2)
   a(i+3)=b(i+3)+x*c(i+3)
enddo
```

- Compilers can do unrolling, but may make them where it is not sensible
- This is not helpful when the work inside the loop is not mostly number crunching
- GNU: **-funrollloops, -funrollallloops**
- PGI: **-Munroll, -Munroll=n:M**
- Intel: **-unroll, -unrollM**

Blocking for cache (tiling)

- Blocking for cache is
  - An optimization that applies for datasets that do not fit entirely into cache
  - A way to increase spatial locality of reference i.e. exploit full cache lines
  - A way to increase temporal locality of reference i.e. improves data reuse
- Example: transposing a matrix

```
do i=1,n
  do j=1,n
    a(i,j)=b(j,i)
  end do
end do
```

Block algorithm for transposing a matrix

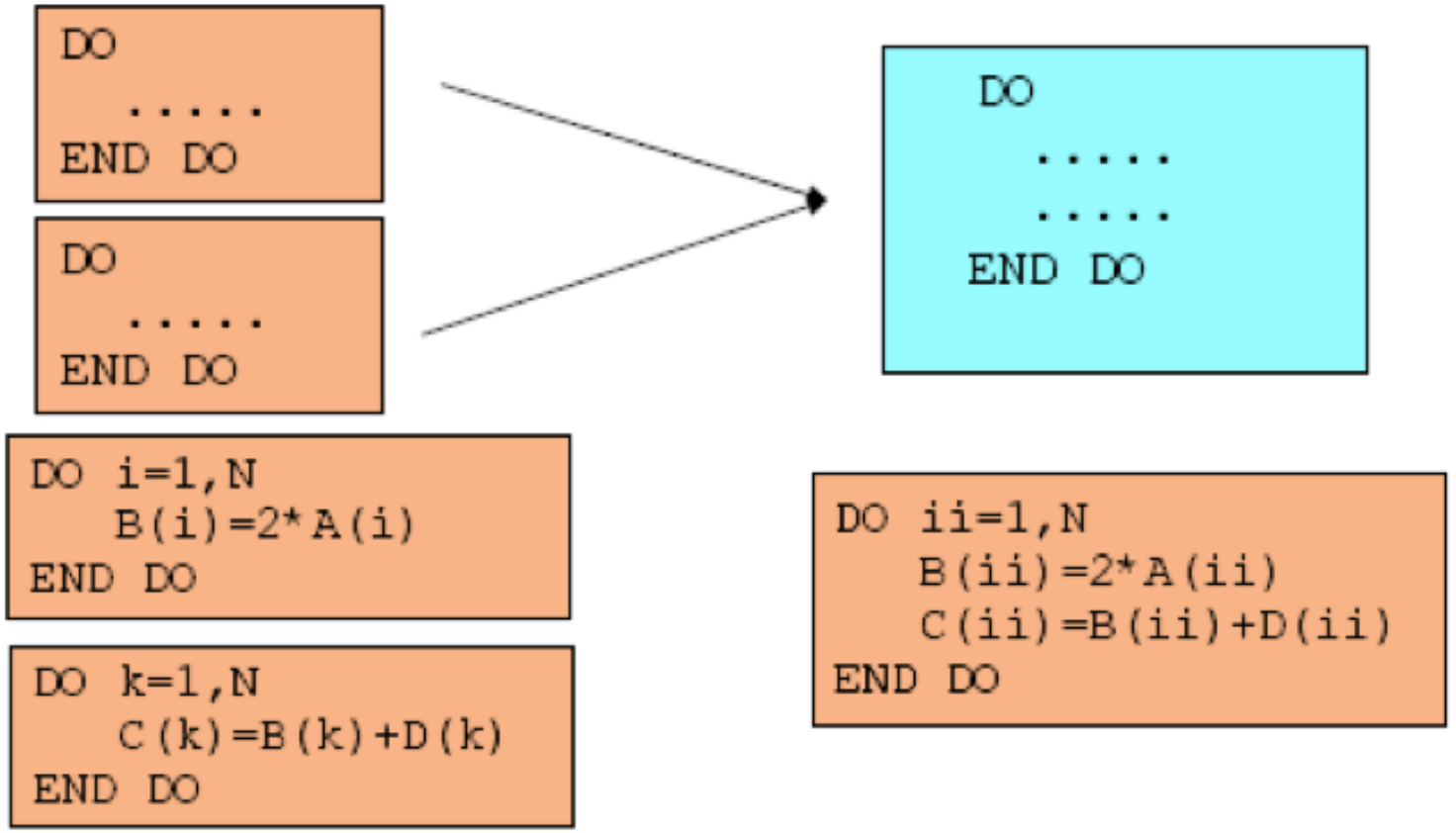
- Block data size = bsize
  - mb = m/bsize
  - nb = n/bsize
- These sizes can be manipulated to coincide with actual cache sizes on individual architectures

```
do ib = 1, nb
 ioff = (ib-1) * bsiz
 do jb = 1, mb
   joff = (jb-1) * bsiz
    do j = 1, bsiz
      do i = 1, bsiz
        buf(i,j) = x(i+ioff, j+joff)
     enddo
    enddo
    do j = 1, bsiz
      do i = 1, j-1
       bswp = buf(i,j)
       buf(i,j) = buf(j,i)
       buf(j,i) = bswp
      enddo
     enddo
    do i=1,bsiz
      do j=1,bsiz
       y(j+joff, i+ioff) = buf(j,i)
      enddo
    enddo
  enddo
enddo
```

SCIENTIFIC COMPUTING LABORATORY

27 Feb 2012

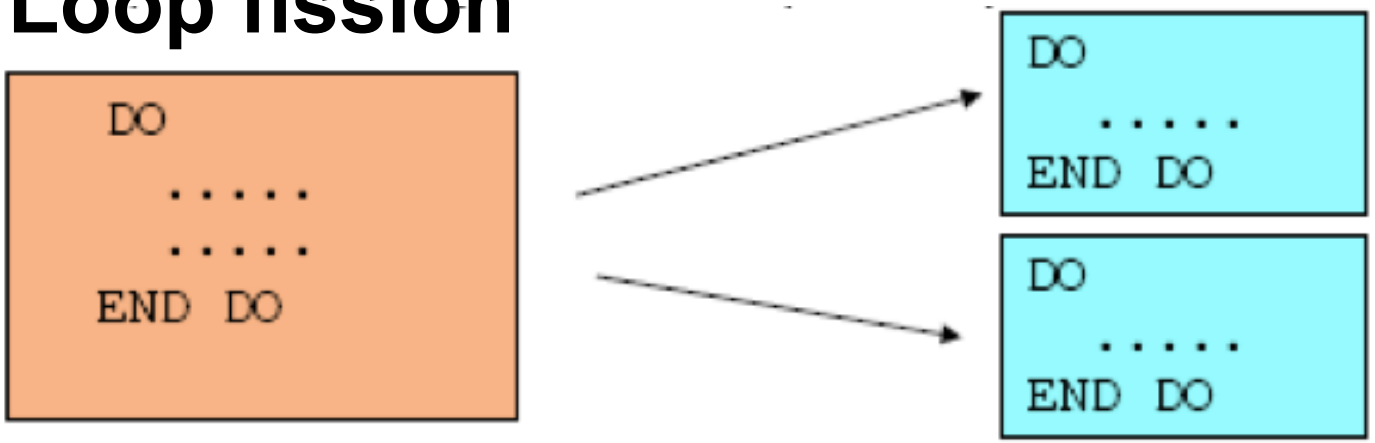
Loop fusion



- Pro: Re-use of the array B
- Cons: Four arrays now fight for cache; more registers needed

SCIENTIFIC COMPUTING LABORATORY

Loop fission




```
DO ii=1,N
   B(i)=2*A(i)
   D(i)=D(i-1)+C(i)
END DO
```

```
DO ii=1,N
   B(ii)=2*A(ii)
END DO
```

```
DO ii=1,N
   D(ii)=D(ii-1)+C(ii)
END DO
```

- Pro: First loop may be scheduled more efficiently and parallelized
- Cons: Less opportunity for out-of-order superscalar execution; Additional loop created (minor)

Prefetching

- Modern CPU's can perform anticipated memory lookups ahead of their use for computation
  - Hides memory latency and overlaps computation
  - Minimizes memory lookup times
- This is a very architecture specific item
- Very helpful for regular, in-stride memory patterns
- GNU: **-fprefetch-loop-arrays**
- PGI: **-Mprefetch[=option:n]**
- Intel: **-O3**

Optimizing Floating Point

- Operation replacement
  - Replacing individual time consuming operations with faster ones
  - Floating point division
    - Notoriously slow, implemented with a series of instructions
    - So does that mean we cannot do any division if we want performance?
  - IEEE standard dictates that the division must be carried out
    - We can relax this and replace the division with multiplication by a reciprocal
    - Compiler level optimization, rarely helps doing this by hand
    - Much more efficient in machine language than straight division, because it can be done with approximates
    - GNU: -funsafe-math-optimizations

Function inlining

- Calling functions and subroutines requires overhead by the CPU to perform
  - The instructions need to be looked up in memory, the arguments translated, etc...
- Inlining is the process by which the compiler can replace a function call in the object with the source code
  - It would be like creating your application in one big function-less format
- Advantage:
  - Increase optimization opportunities
  - Particularly advantegeous (necessary) when a function is called a lot, and does very little work (e.g. max and min functions)
- GNU: -finline-functions, Intel: -ip, -ipo

There is more…

- Elimination of redundant work
- Making use of superscalar features of CPUs (instruction level parallelism)
- Special instructions (SSE - Streaming SIMD Extensions)
- Multi-core CPU's
  - The key issue is memory bandwidth, and good caching performance will be key
    - This problem is worsened as more cores are added.
  - Caching and memory performance vary greatly
    - Some share L2 cache between all cores, some have their own
    - Varying number of pipelines to memory
- Increasing SIMD operations

Conclusions

- Performance programming on single processors requires
  - Detailed profiling
  - Understanding memory
    - levels, costs, sizes
  - Understand SSE and how to get it to work
    - In the future this will one of the most important aspects of processor performance.
  - Understand your program
    - No substitute for spending quality time with your code.
- Do not spend a lot of time doing what compiler can and will do automatically
    - Start with compiler optimizations!
- Code optimization is hard work!
  - And here we did not even consider parallelization!