MHPC
Master in High Performance Computing

Lecture 4: optimization techniques

Stefano Cozzini

CNR/IOM and eXact-lab srl

SISSA
Scuola Internazionale Superiore di Studi Avanzati

ICTP The Abdus Salam International Centre for Theoretical Physics

MHPC
Master in High Performance Computing

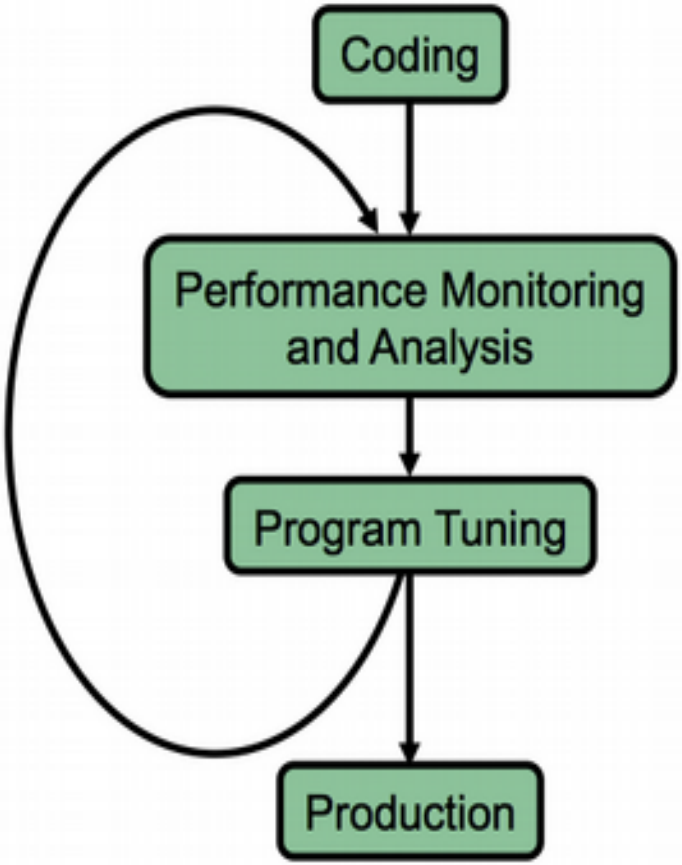
Introduction

- Discuss how to measure performance of your cpu
- Discuss performance tuning techniques common to most modern CPU architecture
- Present some optimization techniques users have control over
  - Code modification
  - Compiler options
- Optimization is a dirty work (and dangerous one for your code...)
- Compiler is your best friend..

SISSA ICTP

How to optimize...

- Iterative optimization
  - 1. Check for correct answers (program must be correct!)
  - 2. Profile to find the hotspots, e.g.
  - 3. Optimize these routines
  - Repeat 1-3
- When satisfied go to production



MHPC
Master in High Performance Computing

Finding hotspot

- Profiler help find “hotspots”
  - to find hotspots, run program and profiler for small, medium, and large data sets
  - Hotspots show a large percentage of a particular profiling statistic such as time or cache misses
- Hotspots for time
  - Parts of the program that are frequently executed
  - Parts of the program that are slow
- Hotspots for data cache misses
  - Parts of the program that are data intensive

SISSA ICTP

Let compiler do the work...

- Help the compiler understand your code
- Look at what the compiler says it can and cannot do
- Make changes when appropriate

MHPC
Master in High Performance Computing

Compiler Optimizations

- Most compilers provides different levels of code optimizations. The compiler option is usually in -O[N] format.
- N goes from 0 to 4/5 levels. Each level introduces new levels of optimization
- "man [compiler command]" provides you a detailed report on all possible options. You can always add/ reduce optimization options.

SISSA ICTP

MHPC
Master in High Performance Computing

But first: which compiler ?

- We focus on two compiler suites:
  - Intel (not free software but quite powerful)
  - GNU (free but not so performant..)
- Which should you use?
  - The one that works better for your application.
  - Both of them to test correctness and quality of your code.
- What about other compiler suites?
  - There are several other: PGI for instance

SISSA ICTP

MHPC
Master in High Performance Computing

Performance Evaluation process

Monitoring your System:

- Use monitoring tools to better understand your machine's limits and usage
- is the system limit well suited to run my application ?
- Observe both overall system performance and single-program execution characteristics. Monitoring your own code
- Is the system doing well ? Is my program running in a pathological situation ?

Monitoring your own code:

- Timing the code:
- timing a whole program (time command :/usr/bin/time)
- timing a portion (all portions) of the program
- Profiling the program

SISSA ICTP

MHPC
Master in High Performance Computing

Useful Monitoring Commands (Linux)

Uptime(1) returns information about system usage and user load

ps(1) lets you see a "snapshot" of the process table

top /atop process table dynamic display

free memory usage

vmstat memory usage monitor

```
top - 08:48:53 up 6 days, 18:35,  5 users,  load average: 0.98, 0.55, 0.22
Tasks:  91 total,   2 running,  89 sleeping,   0 stopped,   0 zombie
Cpu(s): 99.3% us,  0.7% sy,  0.0% ni,  0.0% id,  0.0% wa,  0.0% hi,  0.0% si
Mem:    507492k total,   440336k used,    67156k free,    57928k buffers
Swap:  2048248k total,   108456k used,  1939792k free,    99908k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
18806 cozzini   25   0  2164  464  396 R 90.8  0.1   3:52.85 a.out
 9874 moses     15   0  211m  94m  18m S  7.6 19.0  54:25.92 firefox-bin
 9578 root      15   0  194m  38m 4836 S  1.0  7.7  16:00.59 X
18807 cozzini   16   0  2320  956  756 R  0.3  0.2   0:00.53 top
```

SISSA ICTP

MHPC
Master in High Performance Computing

Monitoring your own code (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 output giving timing statistics about
       this program ..


---------------->time ./a.out
 [program output]

real    0m1.361s
user    0m0.770s
sys     0m0.590s
```

user time: Cpu-time dedicated to your program

sys time: time used by your program to execute system calls

real time: total time aka walltime

SISSA ICTP

User/System/Walltime

- Real time (or wall clock time) is the total elapsed time from start to end of a timed task
- CPU user time is the time spent executing in user space
  - Does not include time spent in system (OS calls) and time spent executing other processes
- CPU system time is the time spent executing system calls (kernel code)
  - System calls for I/O, devices, synchronization and locking, threading, memory allocation
    - Typically does not include process waiting time for non-ready devices such as disks
- CPU user time + CPU system time < real time
- CPU percentage spent on process = 100% * (user+system) / real

Optimization Techniques

There are basically three different categories:

- Improve CPU performance
  - Create more opportunities to go superscalar (high level)
  - Better instruction scheduling (low level)
- Use already highly optimized libraries/subroutines
- Improve memory performance (The most important)

  Better memory access pattern

  Optimal usage of cache lines (improve spatial locality)

  Re-usage of cached data (improve temporal locality)

MHPC
Master in High Performance Computing

Basic rules for optimizing codes

- Do less work!!
  - Elimination of common sub-expressions
- Avoid expensive operations
  - Reduce your math to cheap operations
  - Avoid branches
- Think as a the compiler works
  - Reduce data dependencies (if possible)
  - Facilitate the work of the compiler

SISSA ICTP

MHPC
Master in High Performance Computing

Optimization on Microprocessor

- Pipelined Functional Units + Superscalar processors
  - Unrolling loops can help
- Instruction set extension
  - Newer processors have additional instructions beyond the usual floating point add and multiply instructions:
    - SSE2/SSE3/SSE4.2/AVX/AVX2
- Compilers have a set of special flag to help on this
- Programmer have some ad hoc tools/routines to enhance this

SISSA ICTP

Recall on memory hierarchy

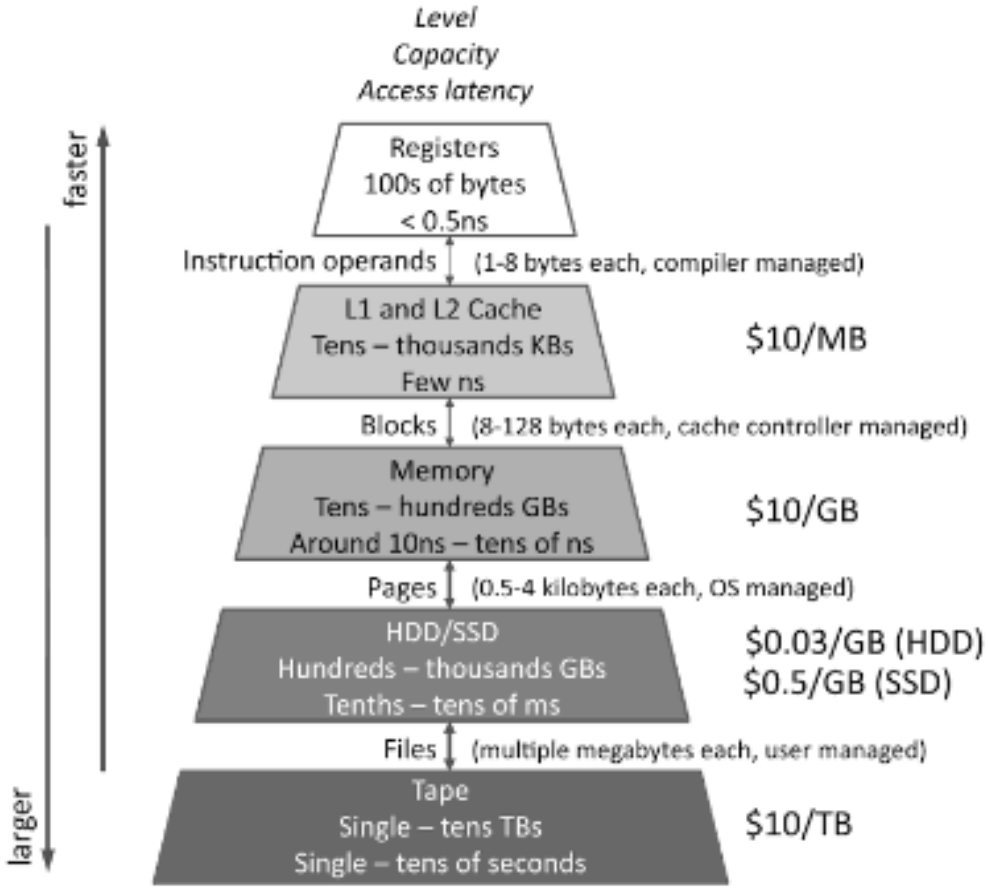


FIGURE 6.7

MHPC
Master in High Performance Computing

Intel Xeon E5 v2 2680 caches

- L1
  - 32 KB
  - 8-way set associative
  - 64 byte line size
- L2
  - 4 MB
  - 8-way set associative
  - 64 byte line size

TASK find out all details on your cache hierarchy

SISSA ICTP

MHPC
Master in High Performance Computing

a top disaster: swapping..

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 done by the system not by your program !

sometimes the system assumes a killer to kill your program.. (oom killer)

```
top - 08:57:02 up 6 days, 19:35,  7 users,  load average: 2.77, 0.73, 0.25
Tasks:  86 total,   2 running,  84 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.3% us,  4.8% sy,  0.0% ni,  0.0% id, 94.2% wa,  0.6% hi,  0.0% si
Mem:    507492k total,   506572k used,      920k free,      196k buffers
Swap:  2048248k total,   941984k used,  1106264k free,     4740k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
11656 cozzini   18   0 2172m 408m  260 D  4.3 82.4   0:03.75 a.out
   33 root      15   0     0    0    0 D  0.7  0.0   0:00.54 kswapd0
 3195 root      15   0 20696 1432 1140 D  0.3  0.3   0:06.81 clock-applet
```

SISSA

top disaster example (1)

```
[cozzini@stroligo optimization]$ /usr/bin/time ./a.out
 provide an integer (suggested range 100-250)
 larger values can be very memory and time-consuming
300
 inizialisation time= 11.787208
10.86user 0.98system 0:14.22elapsed 83%CPU (0avgtext+0avgdata
0maxresident)k
0inputs+0outputs (5major+106090minor)pagefaults 0swaps

[cozzini@stroligo optimization]$ /usr/bin/time ./a.out
 provide an integer (suggested range 100-250)
 larger values can be very memory and time-consuming
320
Command terminated by signal 2
0.18user 1.81system 0:29.27elapsed 6%CPU (0avgtext+0avgdata
0maxresident)k
0inputs+0outputs (5846major+170788minor)pagefaults 0swaps
```

top disaster example (2)

```
[cozzini@stroligo optimization]$ /usr/bin/time ./a.out <300 &
[cozzini@stroligo optimization]$ free
             total       used       free     shared    buffers
cached
Mem:        507492     484916      22576          0       1156
10172
-/+ buffers/cache:     473588      33904
Swap:      2048248      78108    1970140


[cozzini@stroligo optimization]$ /usr/bin/time ./a.out <320 &
[cozzini@stroligo optimization]$ free
             total       used       free     shared    buffers
cached
Mem:        507492     506412       1080          0        252
3936
-/+ buffers/cache:     502224       5268
Swap:      2048248     546348    1501900
```

Locality of Reference

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

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

MHPC
Master in High Performance Computing

Optimization Techniques for memory

- Loop Interchanges
- Effective Reuse of Data Cache
- Loop Unrolling
- Loop Fusion/Fission
- Prefetching

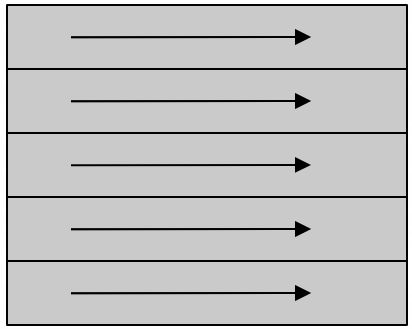
SISSA ICTP

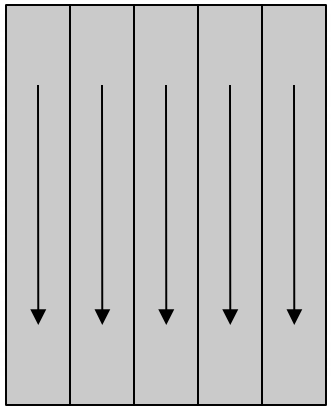
Storage in Memory

The storage order is language dependent:

Fortran stores “column-wise”

C stores “row-wise”





Accessing elements in storage order greatly enhances the performance for problem sizes that do not fit in the cache(s)

(spatial locality: stride 1 access)

MHPC
Master in High Performance Computing

Array Indexing

There are several ways to index arrays:

```
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

The addressing scheme can (and will) have an impact on the performance

SISSA ICTP

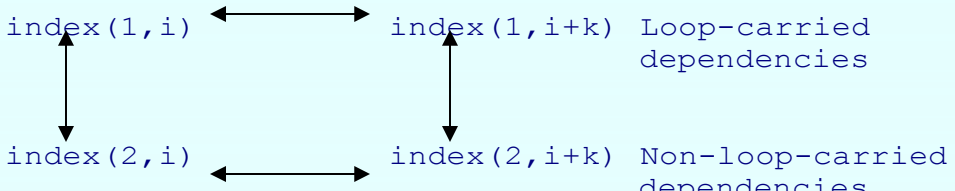
Data Dependencies

Independent instructions can be scheduled at the same time on the multiple execution units in superscalar CPU.

Independent operations can be (software) pipelined on the CPU

Loop-carried dependencies

```
do i=1,n
     a (index (1,i)) = b(i)
     a (index (2,i)) = c(i)
end do
```



Standard prog language (F77/F90/C/C$_{++}$) do not provide explicit information on data dependencies.

Compilers assume worse case for the data dependencies:

-problem for indirectly addressed arrays in Fortran

-problem for all pointers C

MHPC
Master in High Performance Computing

Loop Interchange

Basic idea: In a nested loop, examine and possibly change the order of the loop

Advantages:

Better memory access patterns (leading to improved cache and memory usage)

Elimination of data dependencies (to increase the opportunities for CPU optimization and parallelization)

Disadvantage:

May make a short loop innermost (which is not good for optimal performances)

Exercise: try this on your matrix-matrix multiplication program

SISSA ICTP

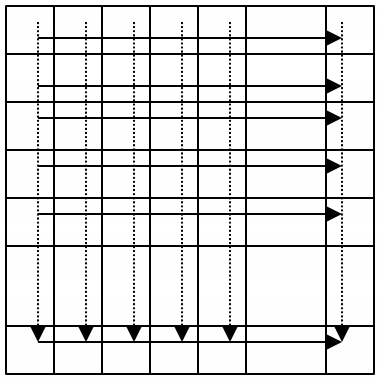
Loop Interchange - Example 1

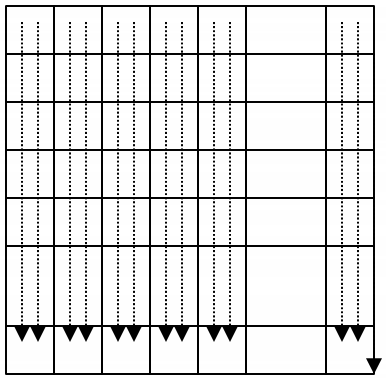
Original

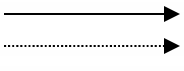
```
DO i=1,N
   DO j=1,M
     C(i,j)=A(i,j)+B(i,j)
   END DO
END O
```

Interchanged loops

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







Loop Interchange in C

In C, the situation is exactly the opposite

```
for (j=0; j<M; j++)
    for (i=0; i<N; i++)
        C[i][j] = A[i][j] +B[i][j];
```

interchange

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        C[i][j] = A[i][j] +B[i][j];
```

index reversal

```
for (j=0; j<M; j++)
    for (i=0; i<N; i++)
        C[j][i] = A[j][i] +B[j][i];
```

The performance benefit is the same in this case

In many practical situations, loop interchange is much easier to achieve than index reversal

Loop Interchange – Mnemonic rule

With row-major, the column or "rightmost" index varies most quickly (C/C+)

With column-major, the row of "leftmost" index varies most quickly.(Fortran/F90)

Loop Interchange - Example 2

```
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
```

orderLoop	2.4GHz(IBM)x335	1.4GHz(IBM)x330
kji	78	69
jki	67	26
kij	2	62
ikj	50	21
jik	90	51
ijk	40	21

Timings are in seconds

MHPC
Master in High Performance Computing

Loop Interchange Compiler Options

- Compilers accept option to automatically play this trick
- Gnu compiler
  - -floop-interchange

TEST IF WHAT THEY CLAIM TO DO IS WHAT THEY ACTUALLY DO

SISSA ICTP

MHPC
Master in High Performance Computing

Cache Thrashing

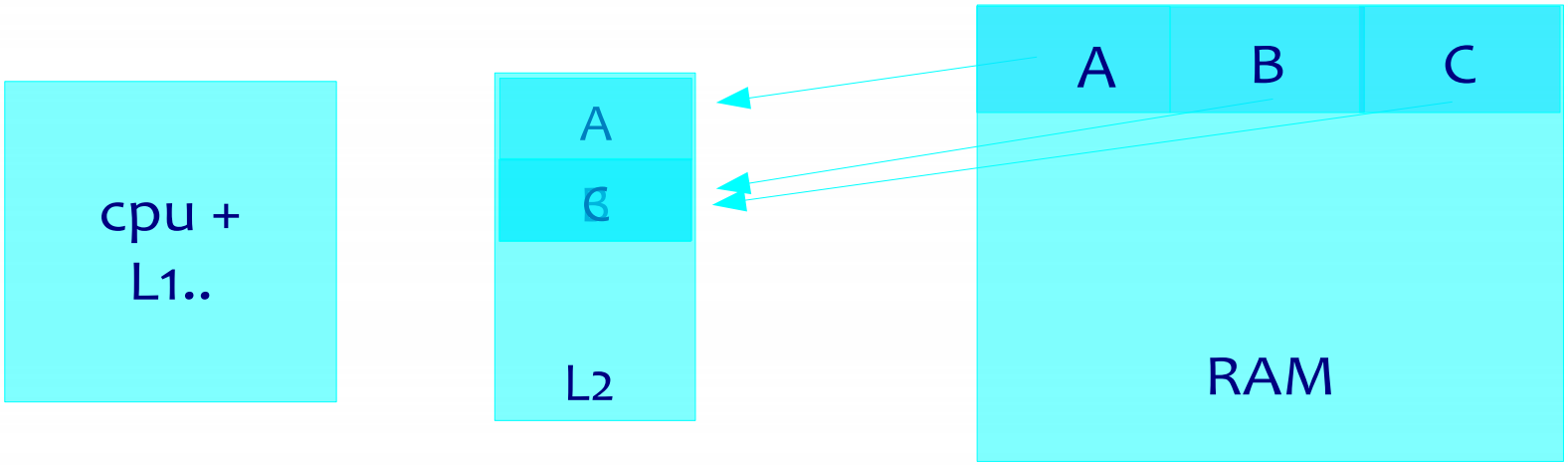
Typical problem in code performance is *cache thrashing*.

Cache trashing happens when data in cache are rewritten and fully reused.

In the previous case, the cache thrashing was minimized by loop interchange.

Another optimization technique aimed at minimizing cache thrashing is *COMMON block padding*.

consider: a(i) +b(i)=c(i)



SISSA ICTP

MHPC
Master in High Performance Computing

Prefetching

Prefetching is the retrieval of data from memory to cache before it is needed in an upcoming calculation. This is an example of general optimization technique called latency hiding in which communications and calculations are overlapped and occur simultaneously.

The actual mechanism for prefetching varies from one machine to another.

When using GNU:

-fprefetch-loop-arrays

SISSA ICTP

MHPC
Master in High Performance Computing

Loop Unrolling

Loop unrolling is an optimization technique which can be applied to loops which perform calculations on array elements.

Consists of replicating the body of the loop so that calculations are performed on several array elements during each iteration.

Reason for unrolling is to take advantage of pipelined functional units. Consecutive elements of the arrays can be in the functional unit simultaneously.

Programmer usually does not have to unroll loops "by hand" -- compiler options and directives are usually available. Performing unrolling via directives and/or options is preferable

- Code is more portable to other systems
- Code is self-documenting and easier to read

Loops with small trip counts or data dependencies should not be unrolled!

SISSA ICTP

MHPC
Master in High Performance Computing

Loop Unrolling Example

- Normal loop

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

- Manually unrolled loop

```
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
```

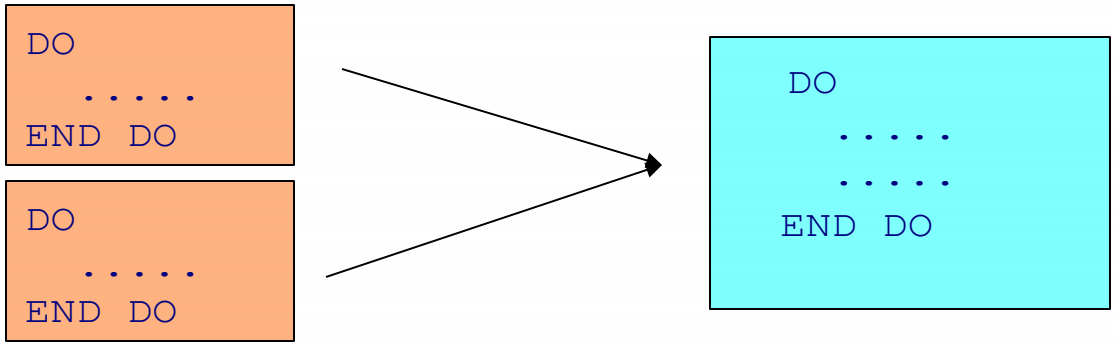
SISSA ICTP

Loop Unrolling Compiler Options

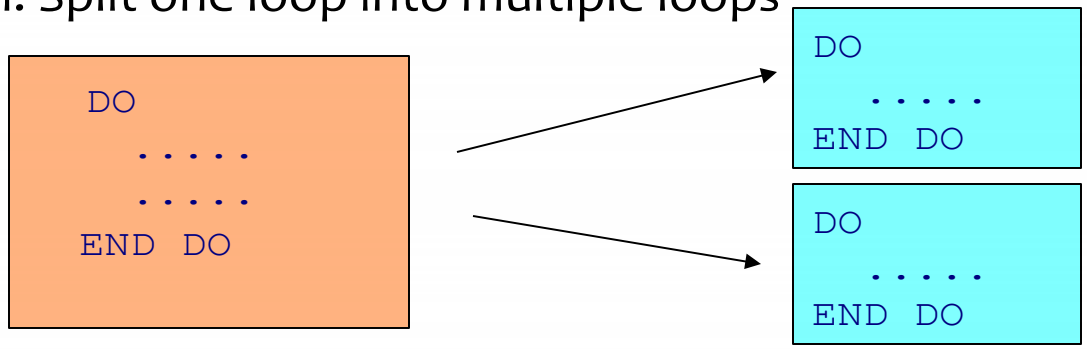
- Gnu
  - -funroll-loops
    - Enable loop unrolling
  - - funroll-all-loops
    - Unroll all loops; not recommended

Loop Fusion and Fission

Fusion: Merge multiple loops into one



Fission: Split one loop into multiple loops



Example of Loop Fusion

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

```
DO k=1,N
   C(k)=B(k)+D(k)
END DO
```

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

Potential for Fusion: dependent operations in separate loops

Advantage:

Re-usage of array B()

Disadvantages:

In total 4 arrays now contend for cache space

More registers needed

Example of 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
```

Potential for Fission: independent operations in a single loop

Advantage:

First loop can be scheduled more efficiently and be parallelised as well

Disadvantages:

Less opportunity for out-of-order superscalar execution
Additional loop created (a minor disadvantage)

MHPC
Master in High Performance Computing

Data reuse

- Performance is limited by data transfer rate
- High performance if data items are used multiple times
- Example: vector addition

  $x_i=x_i+y_i$: 1 op, 3 mem accesses

- Example: inner product
  - $s=s+x_i*y_i$:
  - 2 op, 2 mem access (s in register; also no writes)

SISSA ICTP

Data reuse: matrix-matrix product

- Matrix-matrix product: $2n^3$ ops, $2n^2$ data

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    s = 0;
    for (k=0; k<n; k++) {
      s = s+a[i][k]*b[k][j];
    }
    c[i][j] = s;
  }
}
```

Is there any data reuse in this algorithm?

MHPC
Master in High Performance Computing

Data reuse: matrix-matrix product

- Matrix-matrix product: $2n^3$ ops, $2n^2$ data
- If it can be programmed right, this can overcome the bandwidth/cpu speed gap
- Again only theoretically: naïve implementation inefficient
- *Do not code this yourself..*

SISSA ICTP

Blocking for cache (tiling)

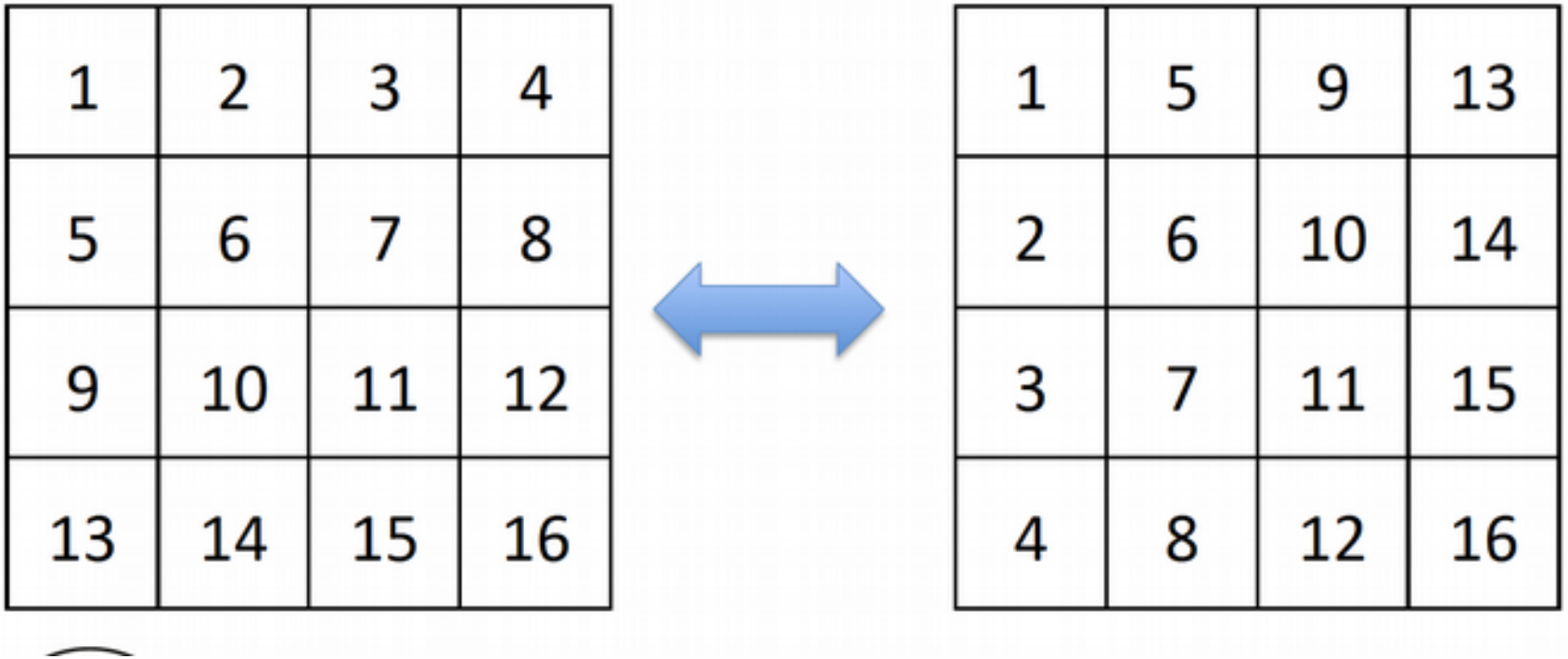
Blocking for cache is:

- *An optimization that applies for datasets that do not entirely fit in the (2nd level) data 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 re-usage*

By way of example, let discuss the transpose of a matrix...

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

Transpose



1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

1	2
5	6

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

MHPC
Master in High Performance Computing

Block algorithm for transposing a matrix:

block data size= bsize

mb=n/bsize

nb=n/bsize

Code is a little bit more complicated if

MOD(n,bize) is not zero

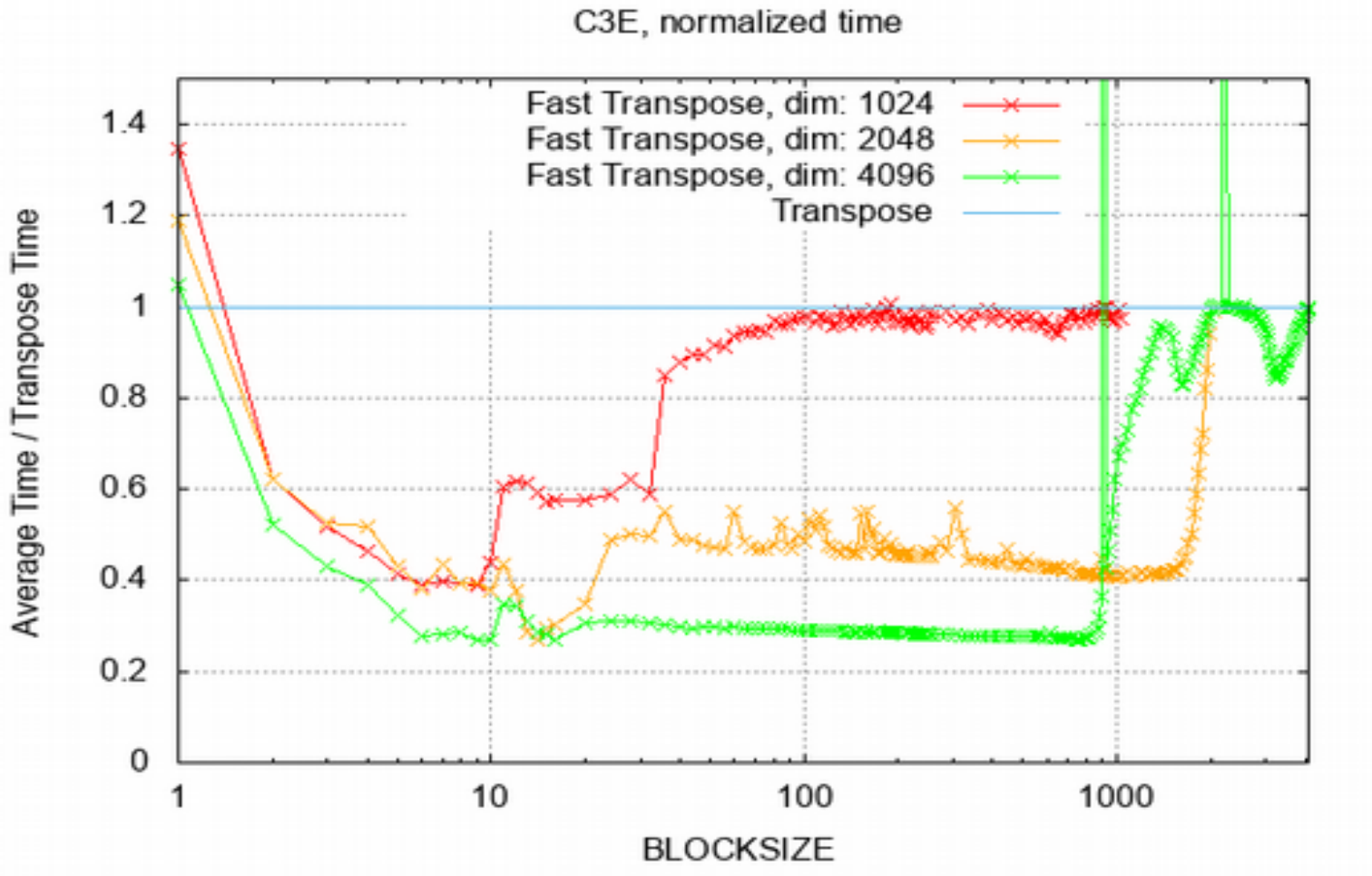
MOD(m,bize) is not zero

```
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
```

SISSA ICTP

MHPC
Master in High Performance Computing

Results... (Piero Coronica's data)



SISSA ICTP

Optimizing Matrix Multiply for Caches

- Several techniques for making this faster on modern processors
  - heavily studied
- Some optimizations done automatically by compiler, but can do much better
- In general, you should use optimized libraries (often supplied by vendor) for this and other very common linear algebra operations
  - BLAS = Basic Linear Algebra Subroutines
- Other algorithms you may want are not going to be supplied by vendor, so need to know these techniques

MHPC
Master in High Performance Computing

Optimization (on intel compiler)

- Start with: -O1, -O2, -O3
  - -O1 Limit code size (no loop unrolling, …)
  - -O2 Intra-file interprocedural optimizations (inlining …), loop unrolling, …
  - -O3 High level optimizations (loop reordering, …), array padding
- Some defaults:
  - vectorization at O2 and higher (-vec- to disable)

SISSA ICTP

MHPC
Master in High Performance Computing

Optimization (on intel compiler)

- What about -fast?
- -fast = “enable -xT -O3 -ipo -no-prec-div -static”
- -static can conflict with bss/zero initialized data and loading of libraries:
- use -shared-intel
  - -static inhibits use of tools that use pre-loading from working

SISSA ICTP

Source code change

- Hand loop unrolling and loop distribution can inhibit compiler optimizations.
- The optimization and vectorization reports can tell you what is preventing optimization or vectorization.
- With this information, making changes to the source may make optimization or vectorization possible and beneficial.

Optimization

- Enable aggressive FP optimizations (which may not be safe)
  - fp-model fast=2 “aggressive optimizations on floating-point data “
  - fno-exceptions “disables exception handling”
  - fno-math-errno “no test of errno from math calls”

Correctness

- Optimizations often affect results of computation.
- HLO: try lower optimization level
- Vectorization: disable with -vec-
- FP: control assumptions but still try high level of optimization

Correctness

-fp_model strict

value-safe optimizations for floating-point calculations, floating-point exception semantics.

-fp-speculation strict

disable speculation on floating-point operations.

-fltconsistency | -mieee-fp

improved floating-point consistency

-prec-div | -prec-sqrt

fully precise division/sqrt implementations