Dr. Pekka Manninen
CSC - IT Center for Science
Finland

Performance Engineering

Overview

Wednesday

9.15-9.30	Introduction to application performance
9.30-10.00	Application performance analysis
10.00-10.15	Coffee break
10.15-11.00	Single-core performance considerations
11.00-11.15	Break
11.15-12.00	Improving parallel scalability
12.00-13.00	Lunch break
13.00-17.00	Lab: Performance engineering

A short summary on Thursday morning

INTRODUCTION TO APPLICATION PERFORMANCE

Improving application performance

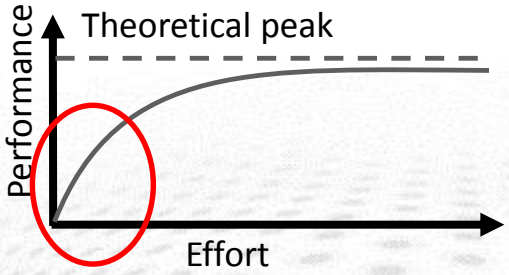
- Obvious benefits
  - Better throughput => more science
  - Cheaper than new hardware
  - Save energy, compute quota etc.
- ..and some non-obvious ones
  - Potential cross-disciplinary research
  - Deeper understanding of application
- Several trends making optimization even more important

Four easy steps to better application performance

- Find best-performing *compilers* and *compiler flags*
- Employ *tuned libraries* wherever possible
- Find suitable settings for *environment parameters*
- Mind the *I/O*
  - Do not checkpoint too often
  - Do not ask for the output you do not need

Optimal porting

- ”Improving application performance without touching the source code”
- Potential to get significant performance improvements with little effort
- Should be revisited routinely
  - Hardware, OS, compiler and library upgrades
  - Can be automated



Compilers
Compiler flags
Numerical libraries
Intranode placement
Internode placement
Filesystem parameters

Choosing a compiler

- Many different choices
  - GNU, PGI, Intel, Pathscale, IBM, Cray etc.
- Compatibility
  - Different proprietary intrinsics
  - Different rounding rules
- Performance
  - There is no universially fast compiler
  - Depends on the application or even input

The ”O” flags

- Standard flags for enabling typical optimizations
  - ’-O[0-4]’, sometimes also ’fast’
    - For example `gcc -O3` or `icc -fast`
  - The higher the level, the more aggressive optimization
    - Compilers default to some ”safe” level (typically ’-O2’)
    - ’-O0’ disables optimizations completely
- Typically improves performance but not always
- No standardized definition what the flags actually mean!

Compiler optimization techniques

- Architecture-specific tuning
  - Tunes all applicable parameters to the defined architecure
- Vectorization
  - Exploiting the vector units of the CPU (SSE, AVX etc.)
  - Improves performance in most cases
- Loop transformations
  - Fusing, splitting, interchanging, unrolling etc.
  - Effectiveness varies

Compiler flag examples

Feature	Cray	Intel	GNU

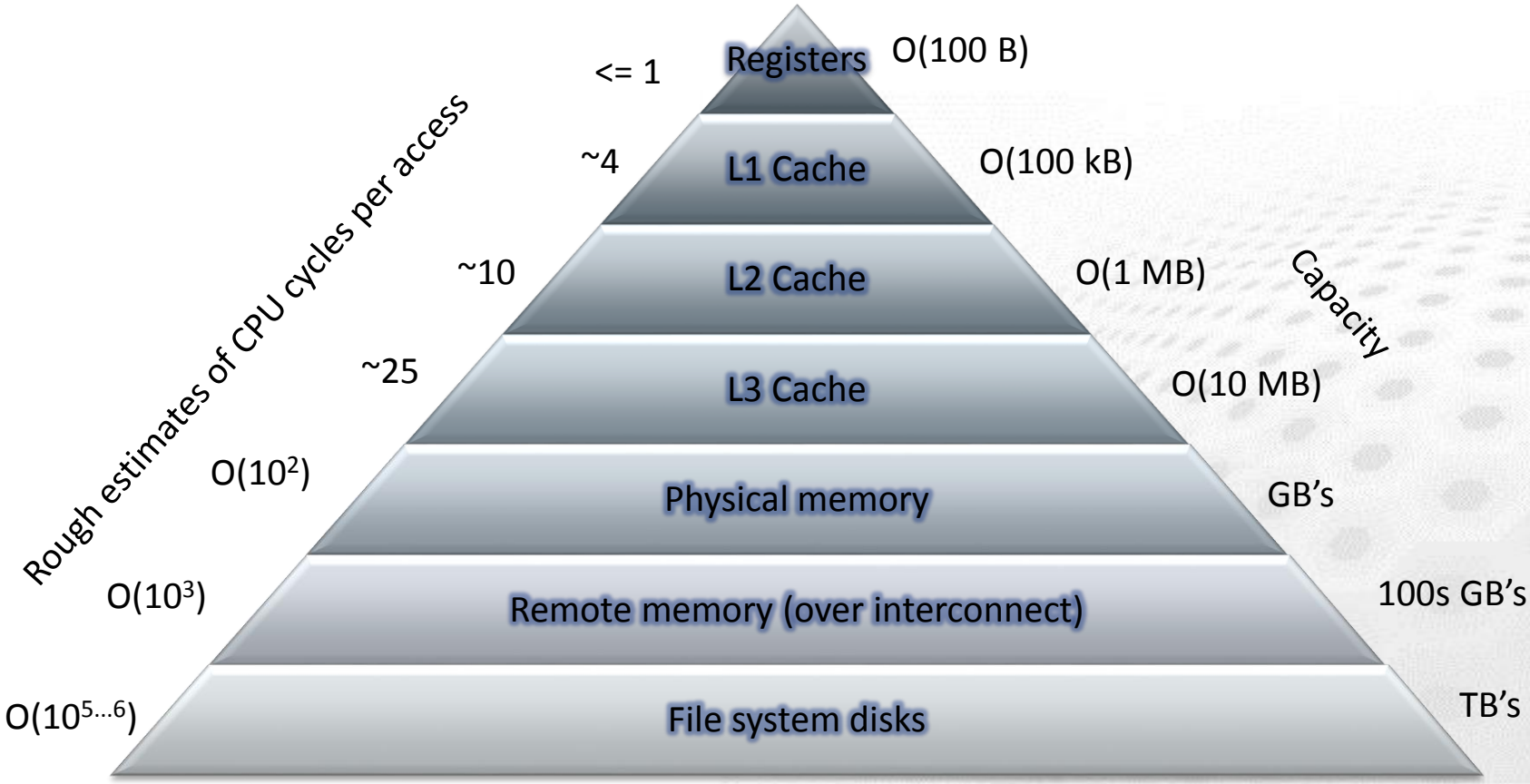
Listing	`-ra`	`-qopt-report=3`
		`-qopt-report-phase=vec`
		`-qopt-report-phase=par`
			`-ftree-vectorizer-verbose=9`

Diagnostic	`(produced by -ra)`	`-help diagnostic`	
Balanced Optimization	`(default)`	`-O2`	`-O3`
Aggressive Optimization	`-O3,fp3`	`-Ofast`	`-Ofast –funroll-loops`
Architecture specific tuning	`-h cpu=<target>`	`-xHost`	`-march=native`
Fast math	`-h fp3 and -h fp4`	`-fp-model fast=2`	`-ffast-math`

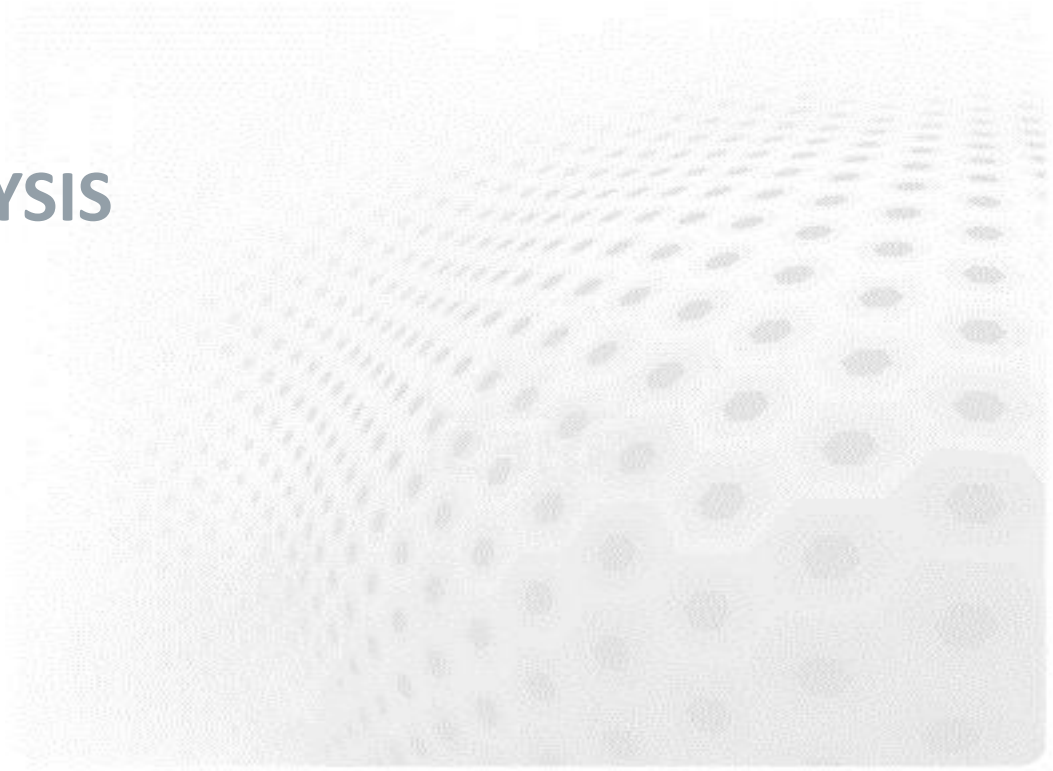
Code optimization

- Adapting the problem to the underlying hardware
- Combination of many aspects
  - Effective algorithms, doing things in a more clever way
  - High processor utilization
  - Efficient memory use
  - Parallel scalability
- Important to understand interactions
  - Algorithm – code – compiler – libraries – hardware
- Performance is not portable

Memory hierarchy



PERFORMANCE ANALYSIS



Application timing

- Most basic information: total wall clock time
  - Built-in timers in the program (e.g. MPI_Wtime)
  - System commands (e.g. time) or batch system statistics
- Built-in timers can provide also more fine-grained information
  - Have to be inserted by hand
  - Typically no information about hardware related issues
  - Information about load imbalance and communication statistics of parallel program is difficult to obtain

Performance analysis tools

- *Instrumentation* of code
  - Adding special measurement code to binary
  - Normally all routines do not need to be measured
- *Measurement*: running the instrumented binary
  - Profile: sum of events over time
  - Trace: sequence of events over time
- *Analysis*
  - Text based analysis reports
  - Visualization

Profiling

- Purpose of the profiling is to find the "hot spots" of the program
  - Usually execution time, also memory
- Usually the code has to be recompiled or relinked, sometimes also small code changes are needed
- Often several profiling runs with different techiques is needed
  - Identify the hot spots with one approach, identify the reason for poor performance

Profiling: sampling

The application execution is interrupted at constant intervals and the program counter and call stack is examined

Pros

- Lightweight
- does not interfere the code execution too much

Cons

- Not always accurate
- Difficult to catch small functions
- Results may vary between runs

Profiling: tracing

Hooks are added to function calls (or user-defined points in program) and the required metric is recorder

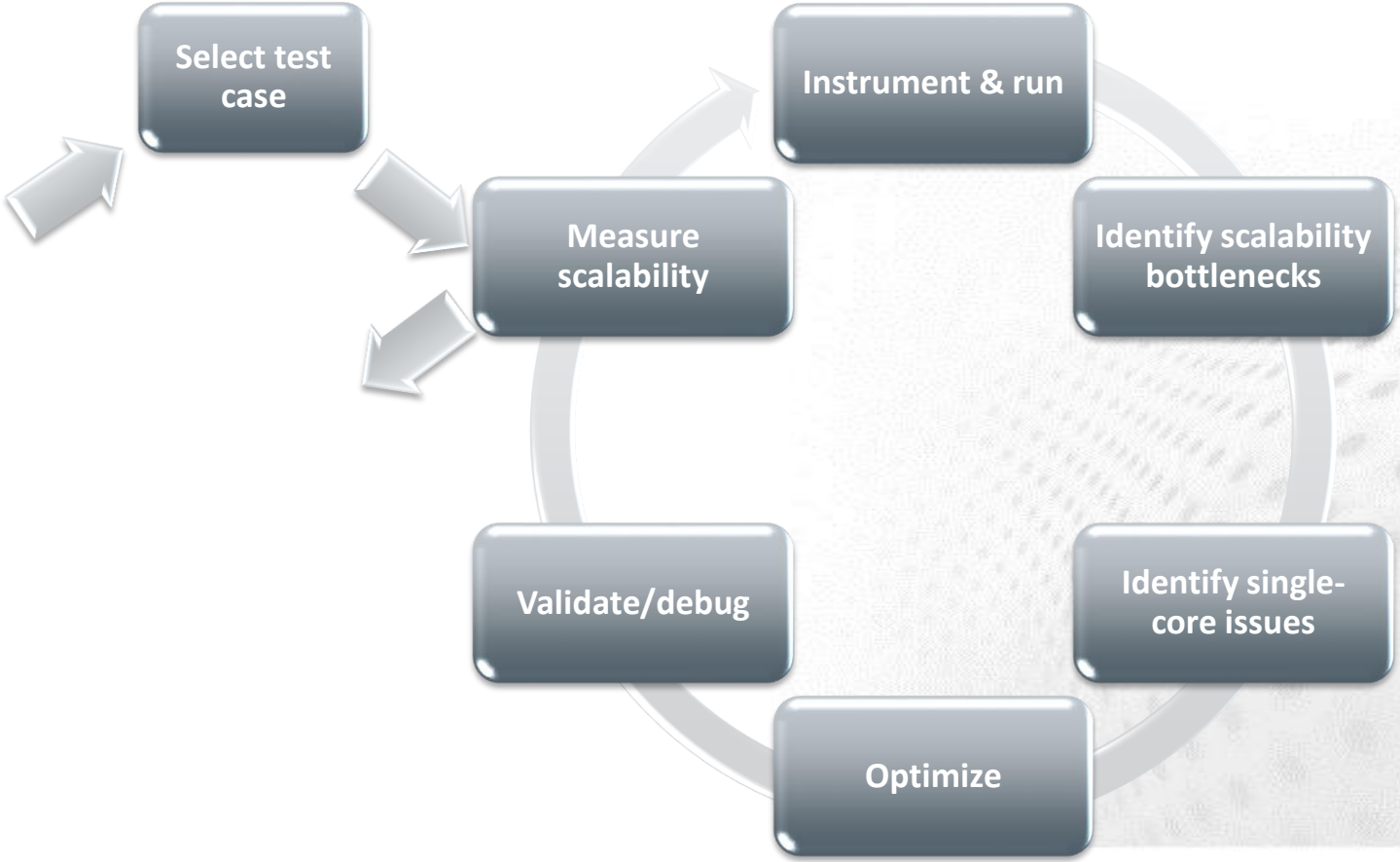
Pros

- Can record the program execution accurately and repeatably

Cons

- More intrusive
- Can produce prohibitely large log files
- May change the performance behaviour of the program

Code optimization cycle

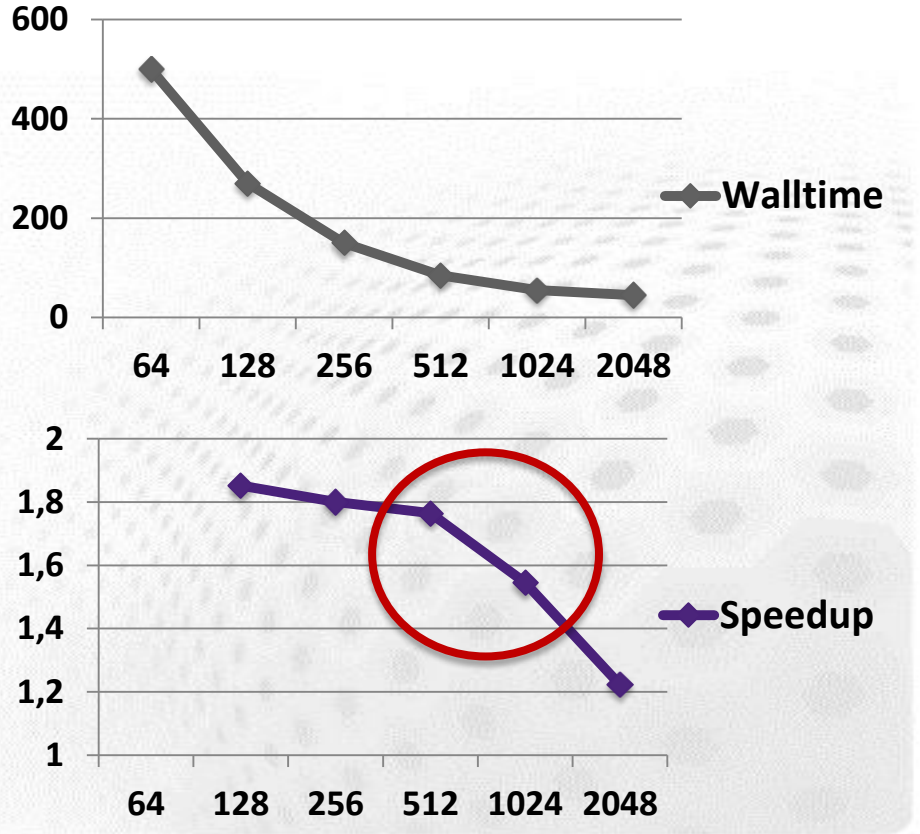


Step 1: Choose a test problem

- The dataset used in the analysis should
  - Make sense, i.e. resemble the intended use of the code
  - Be large enough for getting a good view on scalability
  - Complete in a reasonable time
  - For instance, with simulation codes almost a full-blown model but run only for a few time steps
- Remember that initialization/finalization stages are usually exaggerated and exclude them in the analysis

Step 2: Measure scalability

- Run the uninstrumented code with different core counts and see where the parallel scaling stops
- Often we look at strong scaling
  - Also weak scaling is definitely of interest



Step 3: Instrument & run

- Obtain first a sampling profile to find which user functions should be traced
  - With a large/complex software, one should not trace them all: it causes excessive overhead
  - Tracing also e.g. MPI, I/O and library (BLAS, FFT,...) calls
- Execute and record the first analysis with
  - The core count where the scalability is still ok
  - The core count where the scalability has ended

  and identify the largest differences between these profiles

Example with CrayPAT (1/2)

- Load performance tools software
  `module load perftools`
- Re-build application (keep .o files)
  `make clean && make`
- Instrument application for automatic profiling analysis
  - You should get an instrumented program `a.out+pat`
    `pat_build a.out`
- Run the instrumented application (...+pat) to get a sampling profile
  - You should get a performance file ("`<sdatafile>.xf`") or multiple files in a directory `<sdatadir>`

Example with CrayPAT (2/2)

- Generate text report and an `.apa` instrumentation file
  `pat_report <sdatafile>.xf`
  - Inspect the `.apa` file and sampling report whether additional instrumentation is needed
- Instrument application for further analysis (a.out+apa)
  `pat_build -O <apafile>.apa`
- Re-run application (...+apa)
- Generate text report and visualization file (.ap2)
  `pat_report -o my_text_report.txt <data>`
- View report in text and/or with Cray Apprentice2
  `app2 <datafile>.ap2`

Step 4: Identify scalability bottlenecks

- What are the most intensive MPI operations?
- Does the MPI_Sync time increase when going to the larger core count?
  - Note that the analysis tools may report load imbalances as ”real” communication
    - Put an MPI_Barrier before the suspicious routine - load imbalance will aggregate into it
- Are messages mostly small or large?

Step 4: Identify scalability bottlenecks

- Signature: User routines scaling but MPI time blowing up
  - Issue: Not enough to compute in a domain
    - Weak scaling could still continue
  - Issue: Expensive collectives
  - Issue: Communication increasing as a function of tasks
- Signature: MPI_Sync times increasing
  - Issue: Load imbalance
    - Tasks not having a balanced role in communication?
    - Tasks not having a balanced role in computation?
    - Synchronous (single-writer) I/O or stderr I/O?

Step 5: Find single-core hotspots

- Identify user routines that consume significant portion of the total time
- Collect the key hardware counters, for example
  - Cache & TLB metrics (PAT_RT_PERFCTR=1, default)
  - L1, L2, L3 cache metrics (PAT_RT_PERFCTR=2)
  - Instruction count (PAT_RT_PERFCTR=0)
- Trace the “math” group to see if expensive operations (exp, log, sin, cos,...) have a significant role

Step 5: Find single-core hotspots

- CrayPAT has mechanisms for finding “the” hotspot in one routine (e.g. in case the routine contains several and/or long loops)
  - CrayPAT API
    - Possibility to give labels to “PAT regions”
  - Loop statistics (works only with Cray compiler)
    - Compile & link with CCE using -h profile_generate
    - pat_report will generate loop statistics if the flag is being enabled

Step 5: Find single-core hotspots

- Signature: Low L1 and/or L2 cache hit ratios
  - <96% for L1, <99% for L1+L2
  - Issue: Bad cache alignment
- Signature: Low vector instruction usage
  - Issue: Non-vectorizable (hotspot) loops
- Signature: Traced ”math” group featuring a significant portion in the profile
  - Issue: Expensive math operations

Profiling: do’s and don’ts

- Profile your code
- Do the profiling yourself
- Profile the code on the hardware you are going to run it
- Profile with a representative test case
- Reprofile the code after optimizations

Web resources

- CrayPAT documentation
  http://docs.cray.com
- Scalasca
  http://www.scalasca.org/
- Paraver
  http://www.bsc.es/computer-sciences/performance-tools/paraver
- Tau performance analysis utility
  http://www.cs.uoregon.edu/Research/tau
- Intel VTune Amplifier
  https://software.intel.com/en-us/intel-vtune-amplifier-xe

SINGLE-CORE PERFORMANCE CONSIDERATIONS

Doesn't the compiler do everything?

- You can make a big difference to code performance
  - Helping the compiler spot optimisation opportunities
  - Using the insight of your application
  - Removing obscure (and obsolescent) “optimizations” in older code
    - Simple code is the best, until otherwise proven
- This is a dark art: optimize on case-by-case basis
  - But first, check what the compiler is already doing

Compiler feedback/output

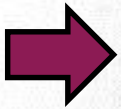
- Cray compiler: `ftn -rm …` or `cc/CC -hlist=m …`
  - Compiler generates an <source file name>.lst file that contains annotated listing of your source code
- Intel compiler 14: `ftn/cc -opt-report=3 -vec-report=6`
  - See `ifort --help reports`
  - 15 has different options, see manual page
- GNU compiler 4.9: `ftn/cc: -fopt-info-vec`
  - gcc 4.8: `-ftree-vectorizer-verbose=6`

Issue: Bad cache alignment

- If multi-dimensional arrays are addressed in a wrong order, it causes a lot of cache misses = bad performance
  - C is row-major, Fortran column-major
  - A compiler may re-order loops automatically (see output)

```
real a(N,M)
real sum = 0;

do i=1,N
  do j=1,M
    sum = sum + a(i,j)
  end do
end do
```



```
real a(N,M)
real sum = 0

do j=1,M
  do i=1,N
    sum = sum + a(i,j)
  end do
end do
```

Issue: Bad cache alignment

- *Loop blocking* = Large loops are partitioned by hand such that the data in inner loops stays in caches
  - A prime example is matrix-matrix multiply coding
- Complicated optimization: optimal block size is a machine dependent factor as there is a strong connection to L1 and L2 cache sizes
- Some compilers do loop blocking automatically
  - See the compiler output
  - You can assist it using compiler pragmas/directives

Issue: Bad cache alignment

```
double a[n][n], b[n][n], c[n][n];
for (i=0; i<n; ++i)
  for (j=0; j<n; ++j)
    for (k=0; k<n; ++k)
      C[i][j] += A[i][k] * B[k][j];
```

Loop blocking example

```
int BS = 8; // An example blocksize
for (i=0; i<n; i+=BS) {
  int iimax = i + MIN(BS, n-i);
  for (j=0; j<n; j+=BS) {
    int jjmax = j + MIN(BS, n-j);
    for (k=0; k<n; k+=BS) {
      int kkmax = k + MIN(BS, n-k);
      for (ii=i; ii<iimax; ii++)
        for (jj=j; jj<jjmax; jj++)
          for (kk=k; kk<kkmax; kk++)
            C[ii][jj] += A[ii][kk] * B[kk][jj];
} } }
```

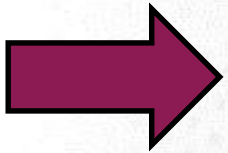
Issue: Bad cache alignment

- *Loop fusion*: Useful when the same data is used e.g. in two separate loops: cache-line re-use

```
int a[N], b[N];

for (i=0; i<N; ++i) {
  a[i] = i;
}

for (i=0; i<N; ++i) {
  b[i] = a[i] * a[i];
}
```



```
int a[N], b[N];

for (i=0; i<N; ++i) {
  a[i] = i;
  b[i] = a[i] * a[i];
}
```

Issue: Non-vectorizable loops

- The compiler will only vectorize loops
- Constant (unit) strides are best
- Indirect addressing will not vectorize (efficiently)
- Can vectorize across inlined functions but not if a procedure call is not inlined
- Needs to know loop tripcount (but only at runtime)
  - i.e. DO WHILE style loops will not vectorize
- No recursion allowed

Issue: Non-vectorizable loops

- Does the non-vectorized loop have true dependencies?
  - No: add the pragma/directive `ivdep` on top of the loop
  - Yes: Rewrite the loop
    - Convert loop scalars to vectors
    - Move if statements out of the loop
- If you cannot vectorize the entire loop, consider splitting it - so as much of the loop is vectorized as possible

Issue: Non-vectorizable loops

- See compiler feedback on why some loops were not vectorized
  - CCE: `-hlist=a`
  - Intel: `-vec-report=6`
  - GNU: `-fopt-info-vec`

```
CC-6290 CC: VECTOR File = ex7_heat.c, Line = 127
  A loop was not vectorized because a recurrence was
  found between "old" and "new" at line 129.
CC-6308 CC: VECTOR File = ex7_heat.c, Line = 128
  A loop was not vectorized because the loop
  initialization would be too costly.
CC-6005 CC: SCALAR File = ex7_heat.c, Line = 128
  A loop was unrolled 2 times.
```

```
127.  + 1------< for (i = 1; i < nx + 1; i++)
128.  + 1 r2---<   for (j = 1; j < ny + 1; j++) {
129.  + 1 r2         new[i][j] = old[i][j] + a * dt *
130.    1 r2           ((old[i+1][j] - 2.0 * old[i][j] + old[i-1][j]) / dx2 +
131.    1 r2            (old[i][j+1] - 2.0 * old[i][j] + old[i][j-1]) / dy2);
132.    1 r2-->>   }
```

Runtime: 8.55 s

Issue: Non-vectorized loops

```
127.  + 1-------< for (i = 1; i < nx + 1; i++)
128.    1           #pragma ivdep
129.    1 Vr2---<   for (j = 1; j < ny + 1; j++) {
130.  + 1 Vr2         new[i][j] = old[i][j] + a * dt *
131.    1 Vr2            ((old[i+1][j] - 2.0 * old[i][j] + old[i-1][j]) / dx2 +
132.    1 Vr2             (old[i][j+1] - 2.0 * old[i][j] + old[i][j-1]) / dy2);
133.    1 Vr2-->>   }
```

Tell the compiler that old and new do not overlap

CC-6294 CC: VECTOR File = ex7_heat.c, Line = 127

A loop **was not vectorized** because a better candidate was found at line 129.

CC-6005 CC: SCALAR File = ex7_heat.c, Line = 129

A loop was **unrolled 2 times**.

CC-6204 CC: VECTOR File = ex7_heat.c, Line = 129

A loop **was vectorized**.

Runtime: 6.55 s

Issue: Expensive operations

- Cost of different scalar FP operations is roughly as follows:

  ~1 cycle: +, *

  ~20 cycles: /, sqrt()

  ~100-300 cycles: sin, cos, exp, log, ...

- Note that there is also instruction latency and issues related to the pipelining

Issue: Expensive operations

- Loop hoisting: try to get the expensive operations out of innermost loops
- Minimize the use of sin, cos, exp, log, pow, ...
  - Consider precomputing values to lookup table
  - Use identities, e.g.
    - pow(x,2.5) = x*x*sqrt(x)
    - sin(x)*cos(x) = 0.5*sin(2*x)
  - Or use vectorized versions (through library calls)
- Consider replacing division (a/b) with multiplication by reciprocal (a*(1/b))

Numerical libraries

- Some key numerical routines have de-facto standardized interfaces
  - BLAS, LAPACK, ScaLAPACK
  - FFT (nearly)
- There are multiple implementations of interfaces
  - Both commercial and open-source
  - The so-called ”reference” implementations are useful for checking correctness but have poor performance

Optimized BLAS

- Cornerstone of performance for many upper-level libraries and applications
- Many optimized implementations
  - OpenBLAS, MKL, LibSci, ACML, ATLAS etc.
- Some compilers support translating intrinsic operations (matmul etc.) into calls to a BLAS library
  - Cray: -O pattern, enabled by -O3
  - Intel Fortran: -opt-matmul
  - GNU Fortran: -fexternal-blas

Summary

- Do the performance analysis!
  - Then you know what to look for
- Utilize the compiler diagnostics
  - Vectorization
- Try to utilize the caches efficiently

IMPROVING PARALLEL SCALABILITY

Scalability bottlenecks

- Signature: user routines scaling but MPI time blowing up
  - Issue: Not enough to compute in a domain
    - Weak scaling could still continue
  - Issue: Expensive (all-to-all) collectives
  - Issue: Communication increasing as a function of tasks
- Signature: MPI_Sync times increasing
  - Issue: Load imbalance
    - Tasks not having a balanced role in communication?
    - Tasks not having a balanced role in computation?
    - Synchronous (single-writer) I/O or stderr I/O?

Issue: Load imbalances

- Identify the cause
  - How to fix I/O related imbalance will be addressed later
- Unfortunately algorithmic, decomposition and data structure revisions are needed to fix load balance issues
  - Dynamic load balancing schemas
  - MPMD style programming
  - There may be still something we can try without code re-design

Issue: Load imbalances

- Consider hybridization (mixing OpenMP with MPI)
  - Reduces the number of MPI tasks - less pressure for load balance
  - May be doable with very little effort
    - Just plug omp parallel do's/for's to the most intensive loops
  - However, in many cases large portions of the code has to be hybridized to outperform flat MPI

Issue: Load imbalances

- Changing rank placement
  - So easy to experiment with that it should be tested with every application!
  - CrayPAT is able to make suggestions for optimal rank placement: `pat_report -O mpi_rank_order datafile.xf`
    - This output can then be copied or written into a file named MPICH_RANK_ORDER and used with `MPICH_RANK_REORDER_METHOD=3`

Issue: Point-to-point communication consuming time

- Use non-blocking operations and try to overlap communication with other work
- Bandwidth and latency depend on the used protocol
  - *Eager* or *rendezvous*
    - Latency *and* bandwidth higher in rendezvous
  - Rendezvous messages usually do not allow for overlap of computation and communication, even when using non-blocking communication routines
  - The platform will select the protocol basing on the message size, these limits can be adjusted

Issue: Point-to-point communication consuming time

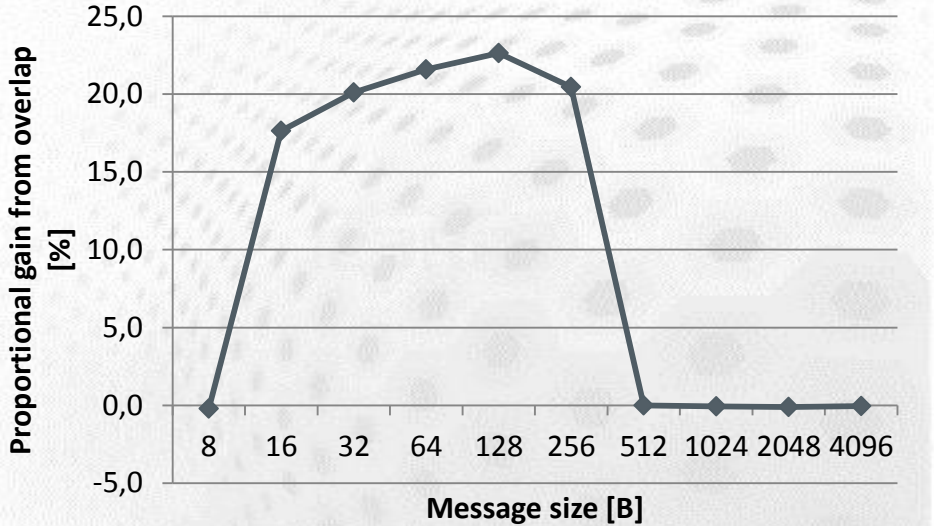
- One way to improve performance is to send more messages using the eager protocol
  - This can be done by raising the value of the eager threshold, consult the MPI man pages
  - E.g. on Cray by setting environment variable `export MPICH_GNI_MAX_EAGER_MSG_SIZE=X`
- Post MPI_Irecv calls before the MPI_Isend calls to avoid unnecessary buffer copies and buffer overflows

Issue: Expensive collectives

- Reducing MPI tasks by hybridizing with OpenMP is likely to help
- See if you can live with the basic version of a routine instead of a vector version (`MPI_Alltoallv` etc)
  - May be faster even if some tasks would be receiving dummy data
- In case of sparse Alltoallv's, point-to-point or one-sided communication may outperform the collective operation

Issue: Expensive collectives

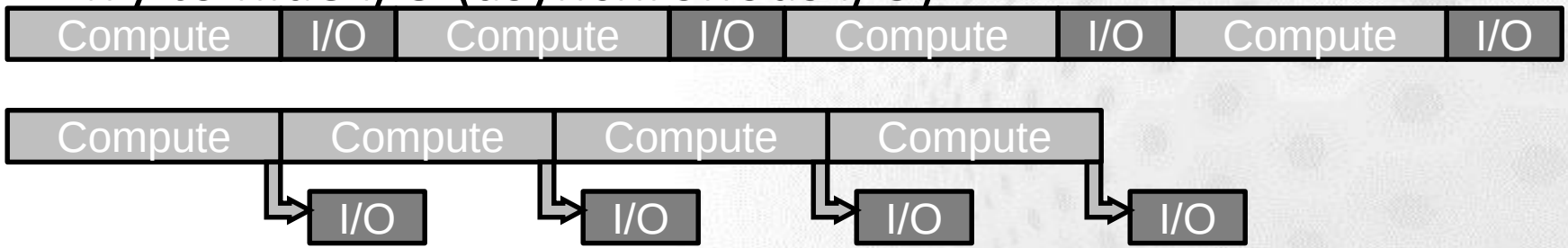
- Use non-blocking collectives (MPI_Ialltoall,...)
  - Allow for overlapping collectives with other operations, e.g. computation, I/O or other communication
  - May be faster than the blocking corresponds even without the overlap
  - Replacement is trivial



MPI_Ialltoall, 1024 cores Cray XC30

Issue: Performance bottlenecks due to I/O

- Parallelize your I/O !
  - MPI I/O, I/O libraries (HDF5, NetCDF), hand-written schemas,...
  - Without parallelization, I/O will be a scalability bottleneck in every application
- Try to hide I/O (asynchronous I/O)

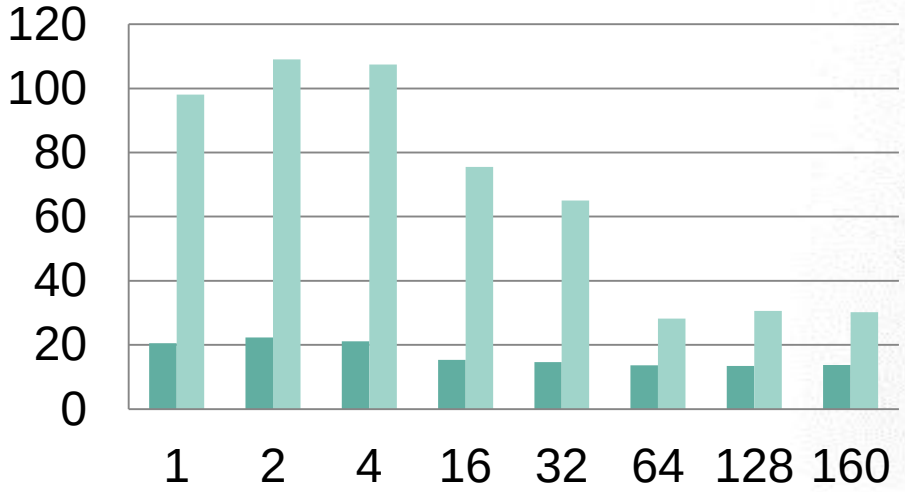


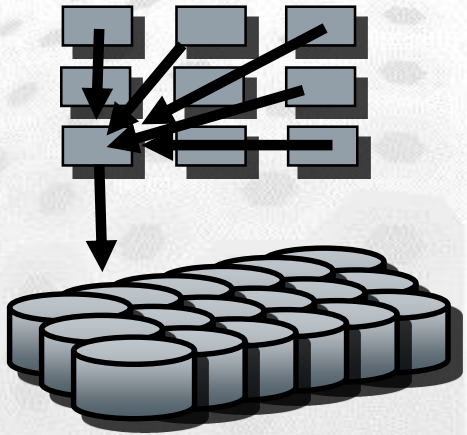
Issue: Performance bottlenecks due to I/O

- Tune filesystem (Lustre) parameters
  - Lustre stripe counts & sizes, see ”man lfs”
  - Rule of thumb:
    - # files > # OSTs => Set stripe_count=1
      You will reduce the lustre contention and OST file locking this way and gain performance
    - #files==1 => Set stripe_count=#OSTs
      Assuming you have more than 1 I/O client
    - #files<#OSTs => Select stripe_count so that you use all OSTs

Case study: Single-writer I/O

- 32 MB per OST (32 MB – 5 GB) and 32 MB Transfer Size
  - Unable to take advantage of file system parallelism
  - Access to multiple disks adds overhead which hurts performance





Case study: Parallel I/O into a single file

- A particular code both reads and writes a 377 GB HDF5 file, runs on 6000 cores on Cray XE6
  - Total I/O volume (reads and writes) is 850 GB
- Original stripe settings: count =4, size=1M
  - 1800 s run time (~ 30 minutes)
- New stripe settings: count=-1, size=1M
  - 625 s run time (~ 10 minutes)

Summary

- Possible approaches for alleviating typical scalability bottlenecks
  - Find the optimal decomposition & rank placement
  - Use non-blocking communication operations for p2p and collective communication both
  - Hybridize (=mix MPI+OpenMP) the code to improve load balance and alleviate bottleneck collectives
  - All file I/O needs to be parallel, I/O performance is sensitive to the platform setup