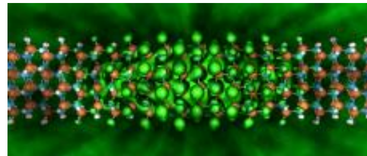
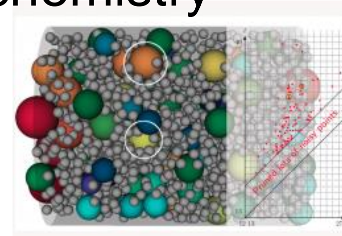
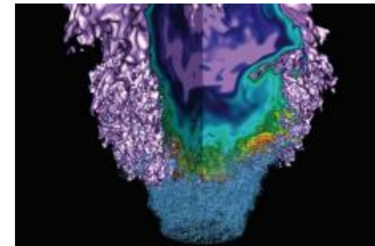
HIGH PERFORMANCE NUMERICAL LINEAR ALGEBRA

Chao Yang
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, CA, USA

Why do we care?

- Numerical linear algebra is the building block of many scientific computing codes and software tools
  - Computational mechanics (structure analysis, fluids)
  - Computational materials science and chemistry
  - Optimization
  - Data analysis
  - Etc.

- Modern computer architecture and high performance computers pose new challenges to numerical linear algebra algorithm
- Lessons learned from designing high performance numerical linear algebra algorithms can be leverage to develop efficient algorithms for other types of computation

Goals

- Brief introduction to problems and issues in high performance numerical linear algebra
- Current capability of numerical linear algebra and how these capabilities are achieved through algorithmic improvement and efficient implementation
- How algorithms are tied to machine architecture, programming tools and libraries
- Concepts and ideas, not so much details
- Good scientific computing practices

Topics not covered

- Specific architectures (e.g. GPUs, Xeon Phi)
- Implementation details
- Problems with special structures (e.g. Toeplitz matrices, Hamiltonian matrices, polynomial eigenvalue problems, tensors)
- Many more software packages and toolboxes
- Latest performance comparisons

OUTLINE

1. Modern computer architecture and high performance computing
2. BLAS and solving dense linear systems of equation
3. Solving dense linear least squares and eigenvalue problems
4. Solving sparse linear systems of equations
5. Solving sparse eigenvalue problems

OUTLINE for TODAY

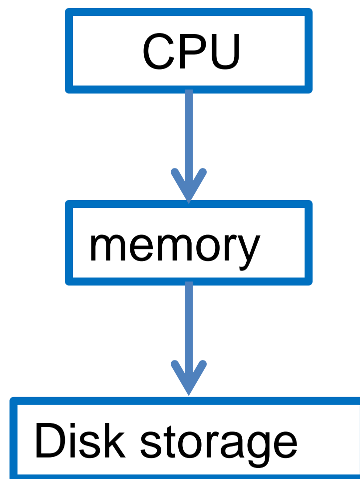
- Modern computer architecture and high performance computing
  - Instruction level parallelism
  - Vectorization
  - Memory hierarchy
  - Concurrency at thread level and shared memory parallelism
  - Inter-processor concurrency and message passing
  - Bandwidth, latency
  - Load balance
  - Scalability (Amdah’s Law, Gustfason models)
- General good programming practices for scientific computing
- Performance model, profiling and optimization

References

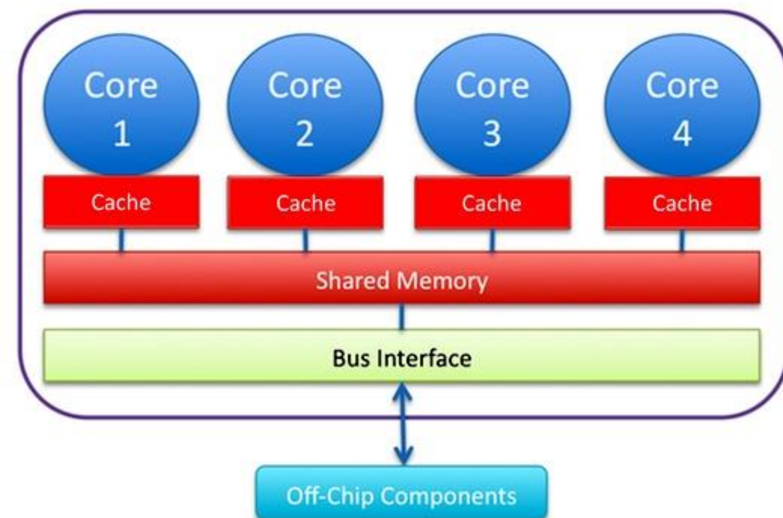
- Dongarra, Duff, Sorensen and Van der Vorst, Numerical Linear Algebra for High Performance Computers, SIAM 1998
- J. Demmel, Applied Numerical Linear Algebra, SIAM
- J. Hennessy and D. Patterson, Computer Architecture: A quantitative approach
- UC Berkeley CS267 web page and course materials
- V. Eijkhout, Introduction to High-Performance Scientific Computing (online) http://pages.tacc.utexas.edu/~eijkhout/istc/html/index.html
- NERSC web page http://www.nersc.gov
- Other latest research

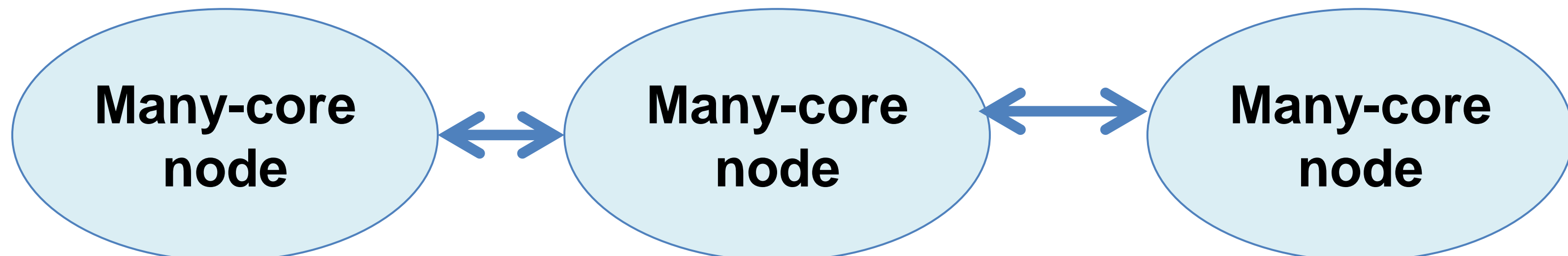
Modern Computer Architecture

- Single processor



- Parallel processors





Functional units in a single processor

- Program counter (PC)
- Load (LD)
- Store (ST)
- Add (ADD)
- Subtract (SUB)
- Multiplication (MUL)
- Division (DIV)

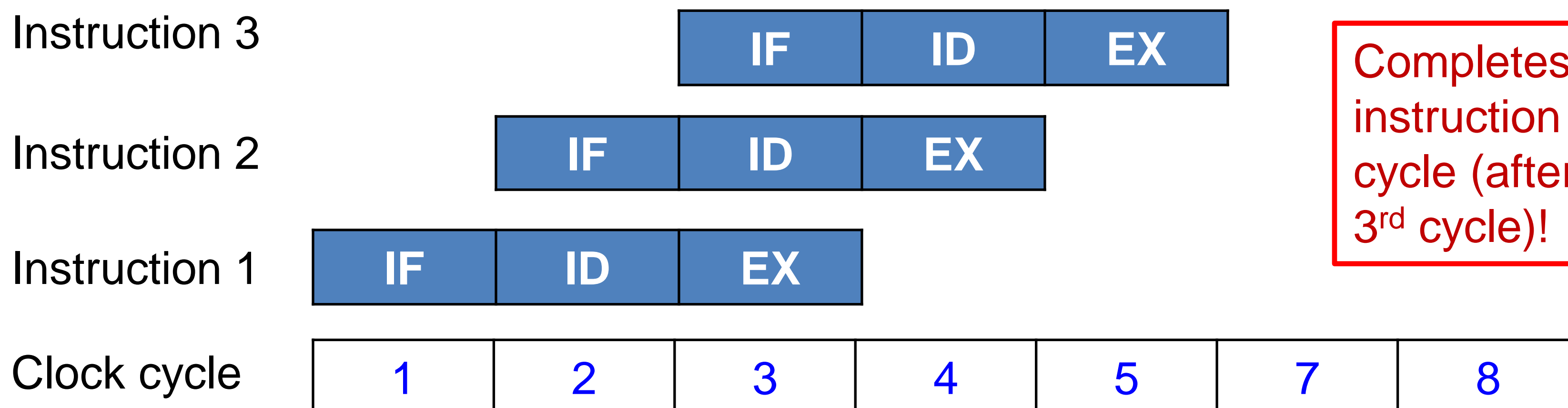
They operate on registers

Each function is further divided in subtasks

Data must be moved into the register before operations can be performed.

Pipelining

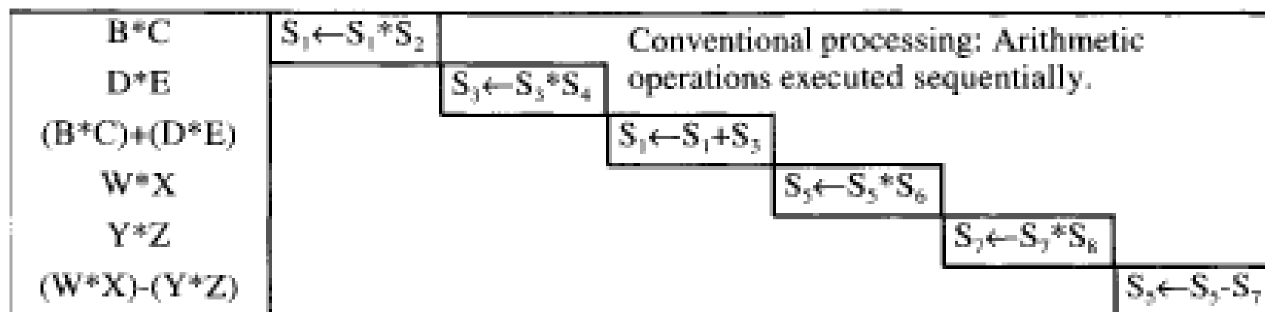
- Segmentation of a functional unit into different part (e.g., instruction fetch (IF), instruction decode(ID), execution (EX) etc.)
- Facilitate instruction-level parallelism and reduce the average number of cycles per instruction
- Successive tasks streamed into the pipe and get executed in an overloaded fashion

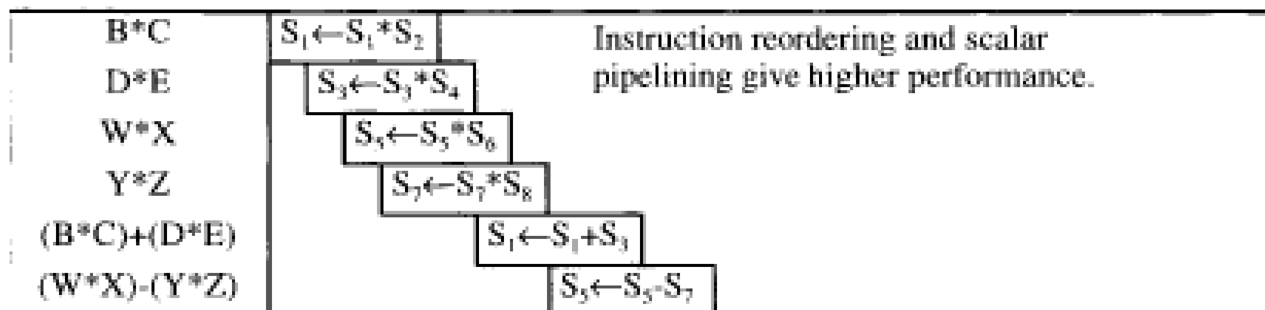


Completes one instruction per cycle (after the 3rd cycle)!

Instruction level of parallelism (ILP)

- Two instructions are parallel if they can execute simultaneously in a pipeline of arbitrary depth without causing any stalls
  - e.g. A=B*C+D*E; V=W*X-Y*Z
  - B*C and D*E can be pipelined, so are W*X and Y*Z





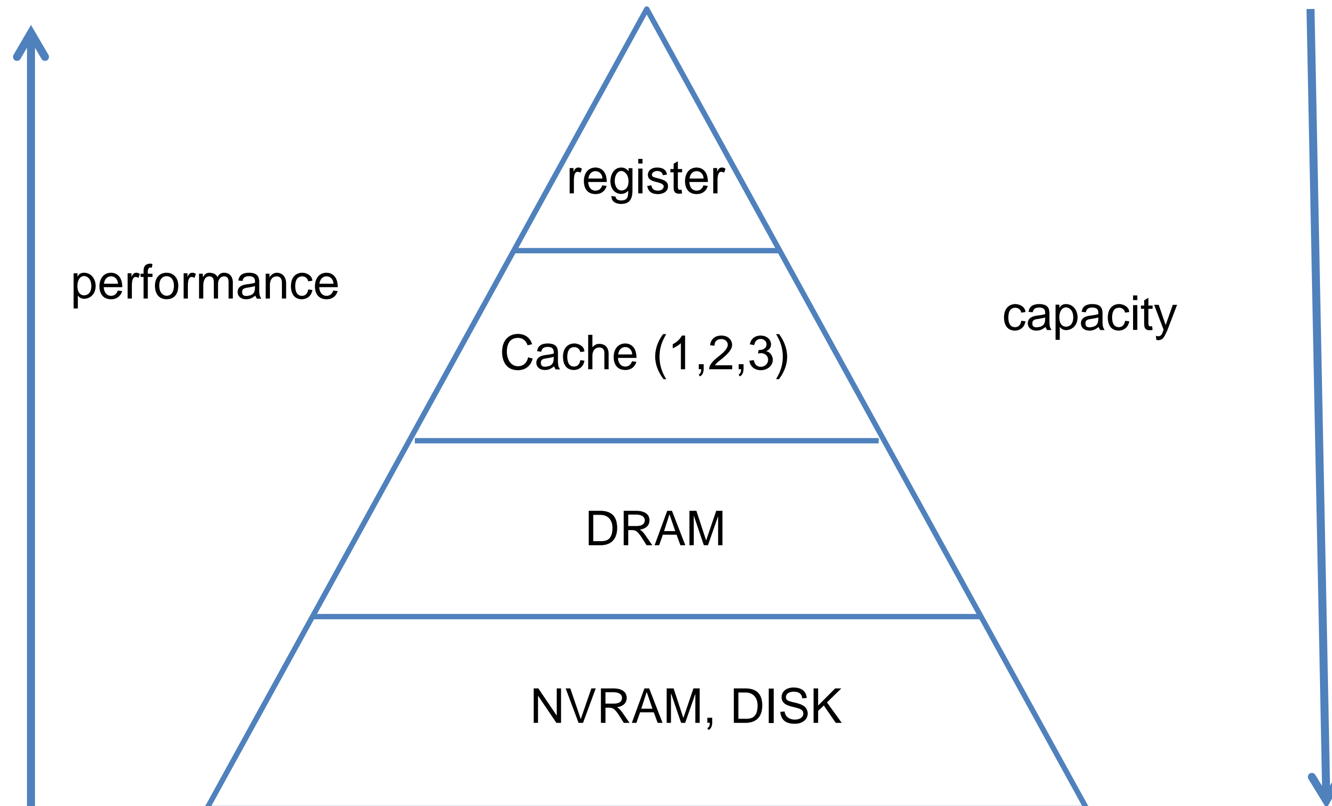
Improving ILP

- What prevents instruction level parallelism
  - Data dependency
  - Control dependency (branching)
- Techniques for improving ILP (often implemented in compilers)
  - Instruction reordering (scheduling, out-of-order execution) and loop unrolling
  - Branch prediction
  - prefetching
- Details (see Hennessy and Patternson, Computer Architecture: A Quantatative Approach, 4th edition, 2007

Vectorization

- Exploit data parallelism, e.g.,
  - for j = 1:n
    - Z(j) = X(j)+Y(j)
  - end
- Use vector instruction to execute several components of vectors (SIMD)
  - for j = 1:4:n
    - Z(j) = X(j)+Y(j)
    - Z(j+1)=X(j+1)+Y(j+1)
    - …
  - end
- Intel SSE(P3), SSE2 (P4), …
- Intel Advanced Vector Extensions (AVX) can simultaneously operate on 8 pairs of single precision (4 bytes) operands or 4 pairs of double precision (8 bytes) operands
- Intel compiler's –vec-report1 and –vec-report2 options

Memory hierarchy and organization



Bandwidth and Latency

- Memory bandwidth: volume of data moved per second
  - e.g. NERSC Edison:
    - L1/L2/L3 cache: 100/40/23GB/sec
- Latency: start up cost, typically small, but can accumulate rapidly with large number of memory access requests
- Bandwidth improvement:
  - Memory divided into banks, sequential memory access falling into different bankds can be completed in parallel. Need to avoid bank conflict when stride is larger than 1
  - Avoid cache misses
  - Avoid TLB (translational look aside buffer) misses (virtual memory organized into pages)
- Latency improvement: vector instructions, data blocking

Performance models

- Metric
  - FLOPS (r)=$\frac{\text{floating point operations}}{\text{time (seconds)}}$
  - $r_{\infty},\ n^{1/2}$: performance of a very long loop, and the length that achieves half of that performance
- LINPACK BENCHMARK and TOP 500 list

http://www.top500.org/

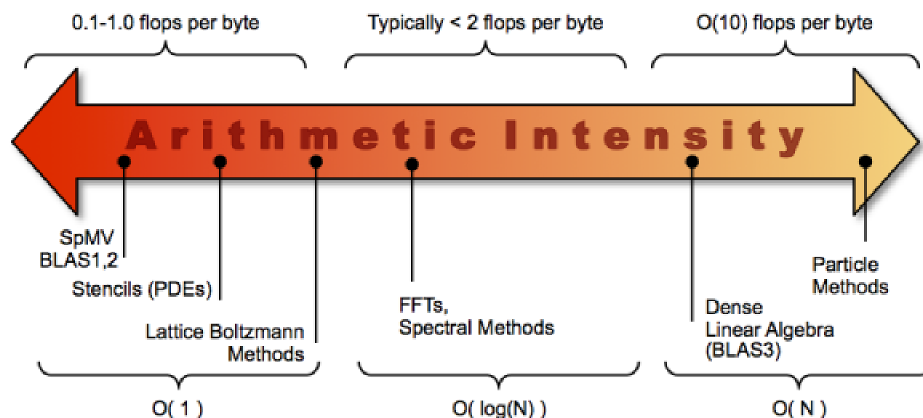
- Amdahl's Law: performance is limited by the slowest part of the program

$$t = f\frac{W}{FF} + (1-f)\frac{W}{SF}$$

Roofline model

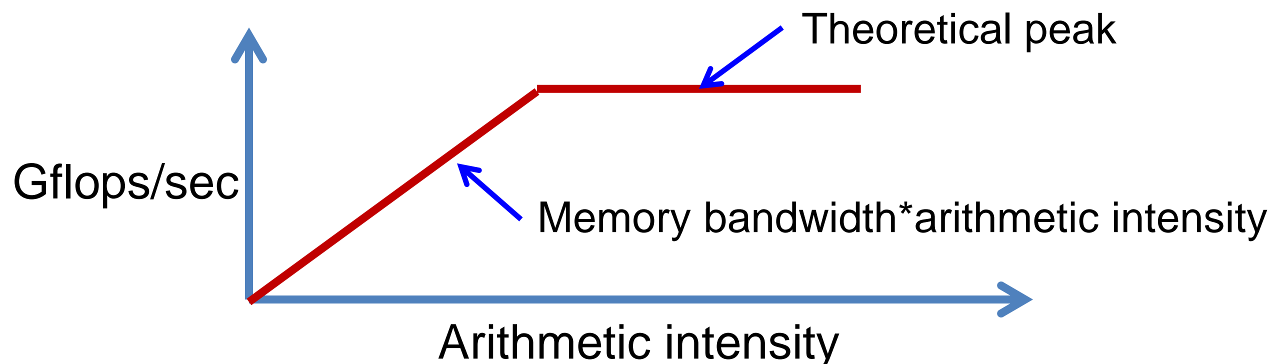
- Arithmetic intensity:

  floating point operations/data movement (in bytes)



picture from Sam Williams http://crd.lbl.gov/departments/computer-science/PAR/research/roofline/

- Roofline model



Techniques for improving single processor performance

- Many of these can be done by compilers these days
- Still good to understand these techniques
- May require code restructuring
- Pay one-time overhead to reorganize data for kernels that are used over and over again
- Optimization may produce (slightly) different results
- Useful tips and exercises can be found at

http://pages.tacc.utexas.edu/~eijkhout/istc/html/index.html

Improve pipeline by loop unrolling

- Inner products don't pipeline well (why?)

```
for i=1:n
    s=s+a(i)*b(i)
end
```

- Unrolled loop

```
for i = 1:n/2
    s1 = s1 + a(2*i-1)*b(2*i-1)
    s2 = s2 + a(2*i)*b(2*i)
end
s = s1+s2
```

Be aware of cache line size

- Underutilized cache line results in higher bandwidth penalty
- Loops with non unit stride:

```
for i=1:stride
    x[i] = 2.3*x[i]+1.2;
end
```

- Performance differs quite a bit for different stride

Minimize TLB misses

- Recall: TLB maintains a small list of frequently used memory pages and their locations
- accessing data on one of these pages is much faster than data on multiple pages that have to be swapped in and out;

```
/* traversal #1 */
for j=1:n
  for i=0:m
    a(i,j)=a(i,j)+1;
  end
end
```

```
/* traversal #2 */
for i=1:m
   for j=1:n
      a(i,j)=a(i,j)+1
   end
end
```

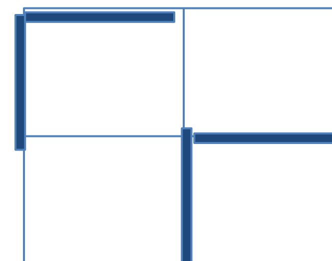
Loop tiling

- Sometimes performance can be improved by breaking a loop into two nested loops
- The goal is to fit the inner loop in the cache

```
for I = 1: n
   a[i] = …
end

for j = 1:nblocks
   for i=(j-1)*nblocks+1:j*nblocks
      a[i] = …
   end
end
```

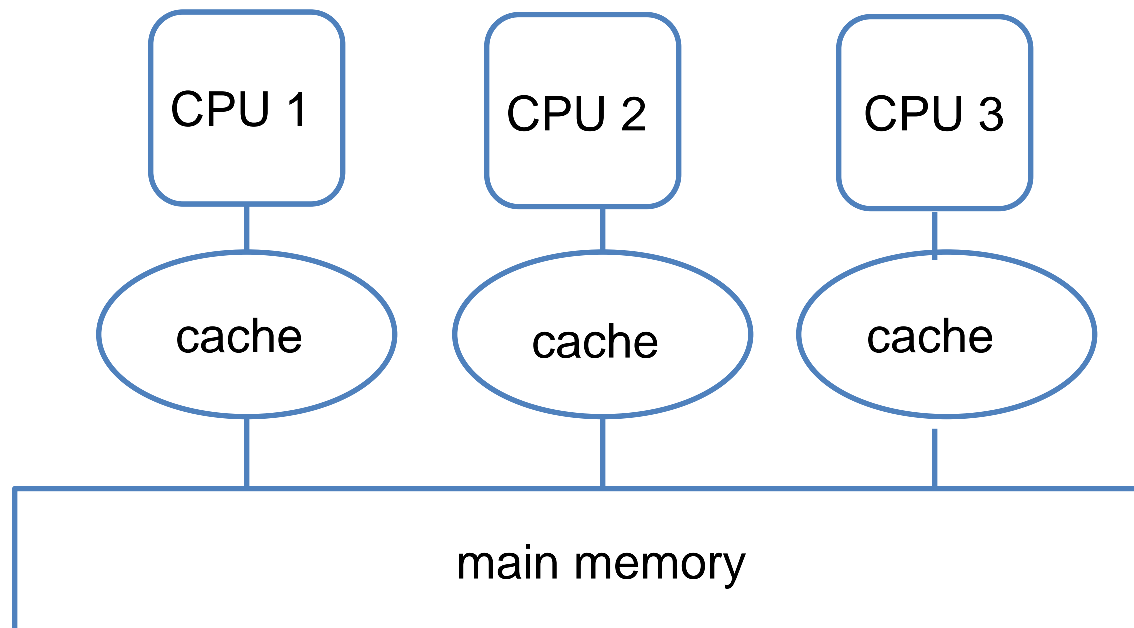
Cache aware and cache oblivious programming

- Cache aware: blocking according to L1, L2, L3 cache sizes
  - Different for different machines
  - Tedious
  - Autotuning by trying different sizes (FFTW, OSKI, ATLAS)
- Cache oblivious: recursive, divide and conquer algorithms
  - Not all algorithms can be organized in such a way
  - Low level tuning is still required
  - e.g. matrix transpose



Share memory parallelism and threads

- Shared memory machines NEC-SX series, Cray T90, SGI Origin
- Multicore (multiple processors on a single die)
- Thread is a light-weight process schedule to perform various tasks



Pthreads

- Unix POSIX threads: a set of primitives that can be used to perform concurrent tasks
  - Fork-join programming model

```
#define NTHREADS 50
int main() {
        int i;
        pthread_t threads[NTHREADS];
        printf("forking\n");
        for (i=0; i<NTHREADS; i++)
                If (pthread_create(threads+i,NULL,&adder,NULL)!=0) return
                i+1;
        printf("joining\n");
        for (i=0; i<NTHREADS; i++)
                if (pthread_join(threads[i],NULL)!=0) return NTHREADS+i+1;
        printf("Sum computed: %d\n",sum);
return 0; }
```

OpenMP

- Compiler directives
- Easy to use, but limited flexibility

```
!$OMP PARALLEL DO
!$OMP& SHARED(A,B,C,CHUNK)
PRIVATE(I)
!$OMP& SCHEDULE(STATIC,CHUNK)
DO I = 1, N
     C(I) = A(I) + B(I)
ENDDO
!$OMP END PARALLEL DO
```

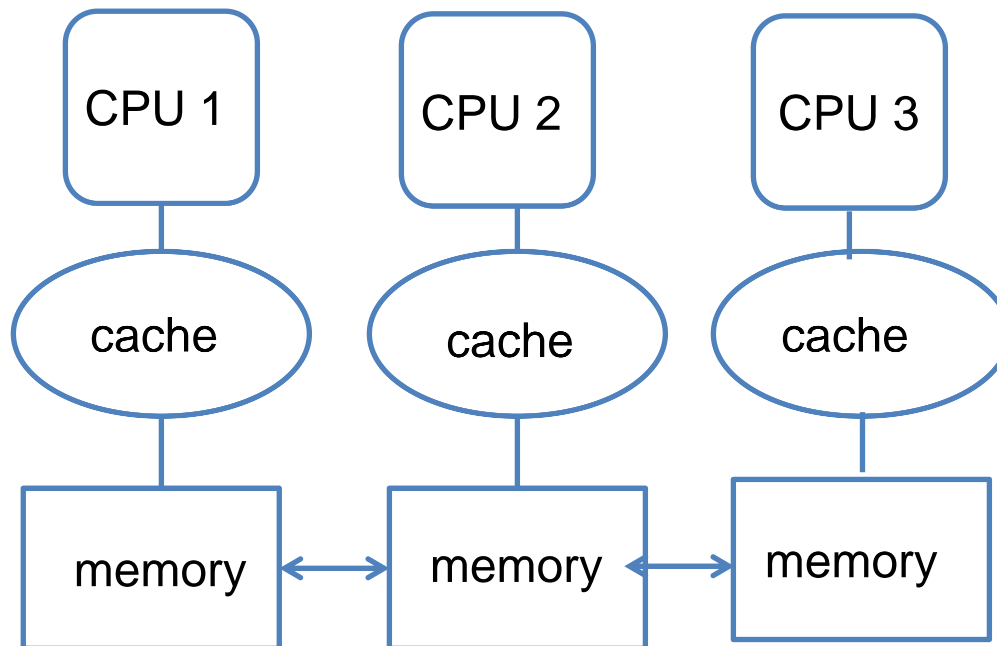
Cache coherence

- Problem: caching shared and private data accessed by each thread
- Since cache is typically local to a core, we need to make sure all cores have a consistent view of shared data after one core modifies the data
- Various strategies to enforce cache coherence (at the OS level) at some cost
- From a programmer's point of view, make sure shared data is not modified simultaneously by multiple cores at the same time. Private data needs to be properly initialized
  - False sharing

```
double x, y;
```

Distributed memory parallelism

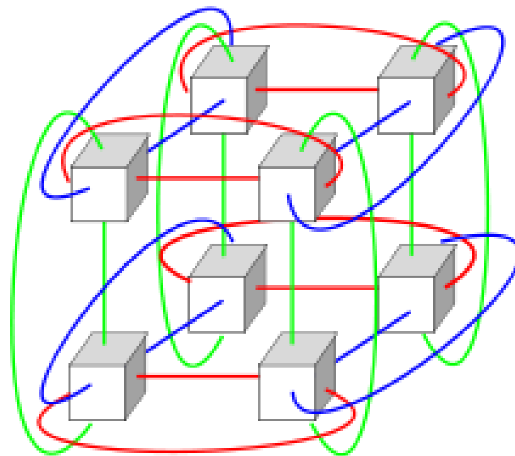
- Purely distributed-memory machine



- Distributed multi-core systems (more common these days); each CPU has multiple cores share local memory

Network topology

- Ring
- Mesh
- Hypercube
- Torus
- Dragonfly

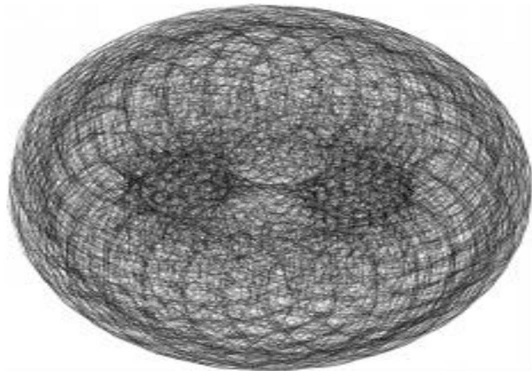


$\mathcal{G} = (\mathcal{V}_\mathcal{G}, \mathcal{E}_\mathcal{G})$: communication graph

$\mathcal{V}_\mathcal{G}$: the set of

$e = \{u, v\} \in \mathcal{E}_\mathcal{G}$: message from *u* to *v*

$\Gamma : \mathcal{V}_\mathcal{G} \rightarrow \mathcal{V}_\mathcal{H}$: defines a mapping



$\mathcal{H} = (\mathcal{V}_\mathcal{H}, \mathcal{E}_\mathcal{H}, c_\mathcal{H})$: interconnection network

$\mathcal{V}_\mathcal{H}$: the set of compute nodes

$\mathcal{E}_\mathcal{H}$: the set of physical links between nodes

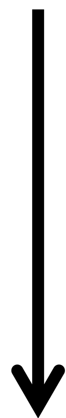
$c_\mathcal{H}(e)$: the capacity of link *e*

Effective bandwidth and latency

$\mathcal{G} = (\mathcal{V}_\mathcal{G}, \mathcal{E}_\mathcal{G})$: communication graph

$\mathcal{V}_\mathcal{G}$: the set of

$e = \{u, v\} \in \mathcal{E}_\mathcal{G}$: message from *u* to *v*



$\Gamma : \mathcal{V}_\mathcal{G} \to \mathcal{V}_\mathcal{H}$: defines a mapping

$\mathcal{H} = (\mathcal{V}_\mathcal{H}, \mathcal{E}_\mathcal{H}, c_\mathcal{H})$: interconnection network

$\mathcal{V}_\mathcal{H}$: the set of compute nodes

$\mathcal{E}_\mathcal{H}$: the set of physical links between nodes

$c_\mathcal{H}(e)$: the capacity of link *e*

Dilation:

$$\mathcal{D}(\Gamma) = \frac{\sum_{\{u,v\} \in \mathcal{E}_\mathcal{G}} |p(\Gamma(u), \Gamma(v))|}{|\mathcal{E}_\mathcal{G}|}$$

Network Traffic:

$$\mathcal{T}_\Gamma(e) = \sum_{\{u,v\} \in \mathcal{E}_\mathcal{G}} \frac{|\mathcal{S} = \{p : p \in \mathcal{P}(\Gamma(u), \Gamma(v)) \wedge e \in \mathcal{P}\}|}{|\mathcal{P}(\Gamma(u), \Gamma(v))|}$$

$$\mathcal{T}(\Gamma) = \frac{\sum_{e \in \mathcal{E}_\mathcal{H}} \mathcal{T}_\Gamma(e)}{|\mathcal{E}_\mathcal{H}|}$$

Congestion:

$$\mathcal{X}(e) = \mathcal{T}_\Gamma(e) / c_\mathcal{H}(e)$$

$$\mathcal{X}(\Gamma) = \max_{e \in \mathcal{E}_\mathcal{H}} \mathcal{X}(e)$$

where $p(u, v)$ is a shortest path *u*→*v*

$\mathcal{P}(u, v)$ is the set of all shortest paths *u*→*v*

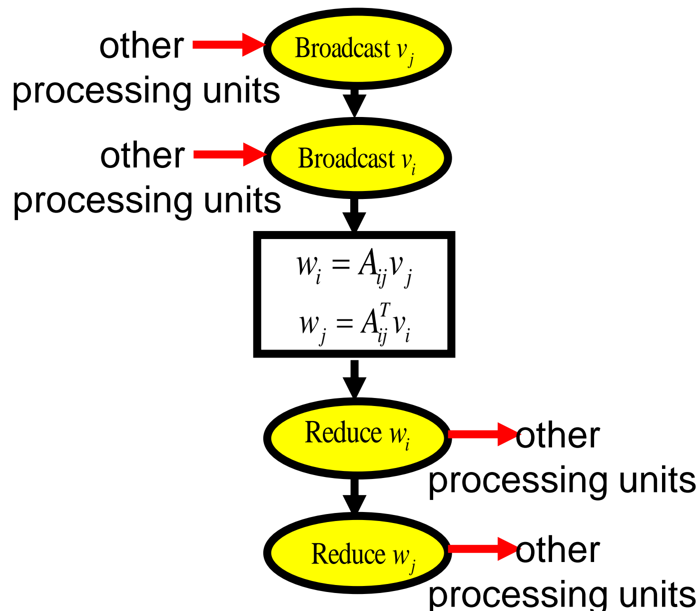
see T. Hoefler, M. Snir, ICS 2011 paper

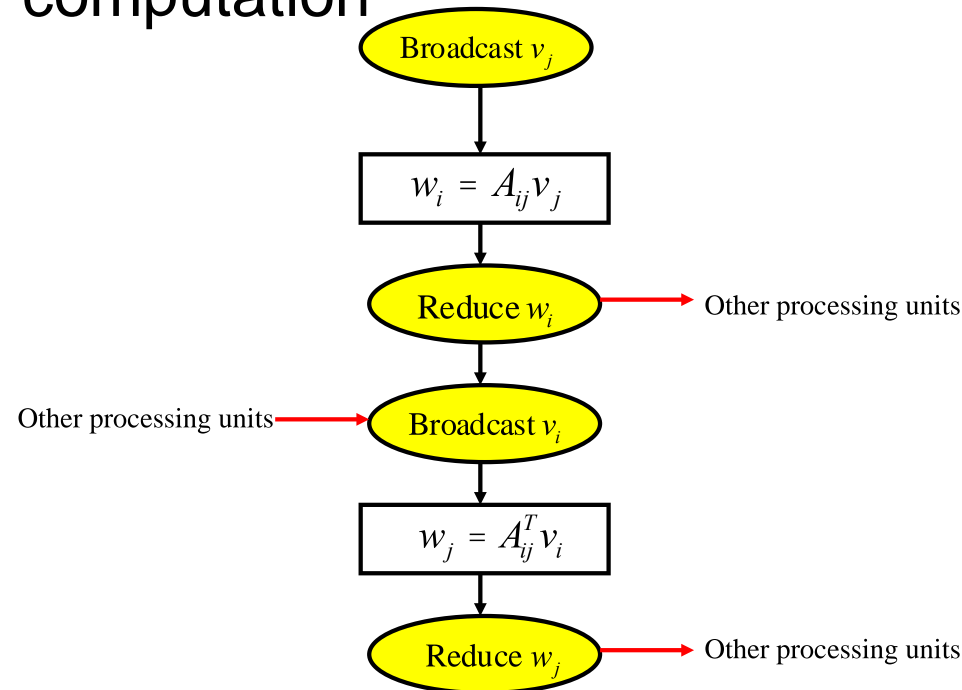
Programming model

- SOCKETS: low-level primitives tightly coupled to network operating system
- Message Passing Interface (MPI): high level communication APIs that serve the needs of general scientific computing
  - Communication group and communicator,
    MPI_COMM comm;
  - Point-point communication: MPI_SEND, MPI_RECV
  - Synchronous (blocking) and asynchronous communication: MPI_ISEND, MPI_IRECV, MPI_WAIT
  - Collective communication: MPI_REDUCE, MPI_ALLREDUCE, MPI_BCAST, MPI_ALLGATHERV, MPI_SCATTER

MPI+OpenMP

- Suitable for distributed multi/many-core systems
- MPI across the distributed memory nodes;OpenMP within each node
- Reduce memory requirement
- Overlap communication with computation





Other programming models

- CUDA and OpenACC for GPUs
- High Performance FORTRAN (HPF)
  - ✓ Data parallel model
  - ✓ New syntax for supporting parallel loops: e.g. FORALL
  - ✓ Compiler directives
  - ✓ Extrinsic procedure interface for interfacing with MPI
  - ✓ Additional library routines
- Partitioned Global Address Space (PGAS): program distributed memory machines as if they have a large shared memory address space
  - UPC++
  - Global Array
  - One side communication such as RMA and asynchronous remote function invocation
  - Developers need to be aware of latency issues
- Co-array FORTRAN, Chapel, Fortress, X10,

Scalability

- Speedup: $S_p = t_1/t_p$, where t_1 is single processor time, t_p is time required to execute in parallel on p processors
- Amdahl's law (strong scaling) If a fraction f of the program can achieve p-fold speedup, the overall speedup is

$$S_p = \frac{p}{f + (1-f)p}$$

  - $p = 100, f = 0.9$, $S_p \approx 9$, speed up limited by 1/(1-f)
  - $S_p = \frac{t_1}{t_1/p + t_c}$, where t_c is communication time
- Gustafson's model (weak scaling)
  - Large machines are used to solve large problems
  - f may be a function of p

$$S_p = p - (p-1)(1-f)$$

  - $S_p = p(1 - \frac{t_c}{t_1}p)$

Factors that impact the performance of a parallel program

- Hardware capacity in terms of communication bandwidth and latency
- The level of concurrency (Amdahl’s law and Gustafson model)
- The length of the critical path
- How well the program is load balanced
- The number of synchronization points

Granularity

- Coarse grained parallelism tends to have lower communication overhead or higher flops/communication ratio, but maybe difficult to load balance
- Fine grained parallelism is easy to load balance (dynamically), but may incur higher communication/thread overhead
- Multiple levels of parallelism suitable for distributed multicore machines

Techniques to improve parallel performance

- Identify the problem by using proper performance analysis tools
- Improve load balancing through proper data/task distribution
- Topology-aware parallelization (ordering of the MPI ranks may be important)
- Reduce communication as much as possible (both the message size and the number of messages)
- Reduce communication overhead by overlapping communication with computation (via asynchronous communication)

Performance analysis tools

- A number of tools are available: IPM, CrayPAT, Vtune, PAPI, TAU, PerfSuite, HPCToolkit
- Profile the program to identify the time-consuming part of the computation
- Probe various performance characteristics to identify the source of the problem
  - Flop rate
  - Load balancing
  - Communication volume
  - Message count
  - Cache misses
  - TLB misses

Sampling vs tracing

- Sampling
  - Use hardware counter to find out what is being executed and what and how frequent resources are used
  - Code instrumentation
  - Low overhead
  - Useful for identifying performance hotspots and bottlenecks
- Tracing
  - Focus on selected functions/subroutines to examine performance in detail
  - User specify which functions to trace
  - Large overhead

Integrated Performance Monitoring (IPM)

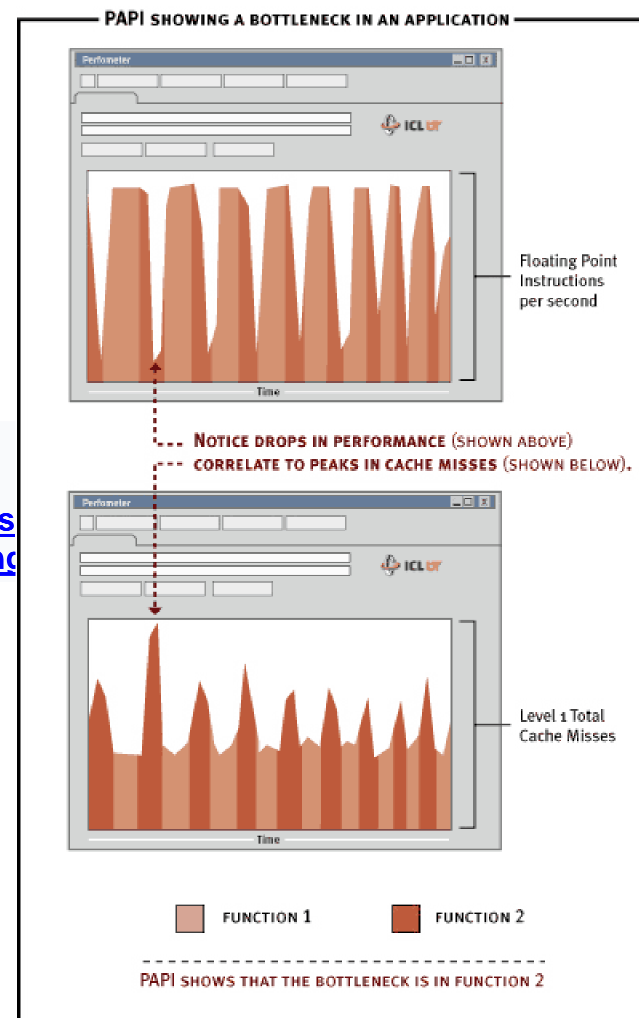
- **http://ipm-hpc.sourceforge.net/**
- portable profiling infrastructure, runtime library
- high level report
  - hardware counters data,
  - MPI function timings
  - memory usage.

```
##IPMv0.8#####################################################################
#
# code   : ./bin/cg.B.32 (completed)
# host   : s05601/006035314C00_AIX         mpi_tasks : 32 on 2 nodes
# start  : 11/30/04/14:35:34               wallclock : 29.975184 sec
# stop   : 11/30/04/14:36:00               %comm     : 27.72
# gbytes : 6.65863e-01 total               gflop/sec : 2.33478e+00 total
#
#
#                     [total]        <avg>          min          max
# wallclock           953.272      29.7897      29.6092      29.9752
# user                837.25       26.1641        25.71        26.92
# system               60.6        1.89375         1.52         2.59
# mpi                 264.267      8.25834      7.73025      8.70985
# %comm                            27.7234      25.8873      29.3705
# gflop/sec           2.33478    0.0729619     0.072204    0.0745817
# gbytes             0.665863    0.0208082    0.0195503    0.0237541
# PM_FPU0_CMPL    2.28827e+10  7.15084e+08  7.07373e+08  7.30171e+08
# PM_FPU1_CMPL    1.70657e+10  5.33304e+08  5.28487e+08  5.42882e+08
# PM_FPU_FMA      3.00371e+10   9.3866e+08  9.27762e+08  9.62547e+08
# PM_INST_CMPL    2.78819e+11  8.71309e+09  8.20981e+09  9.21761e+09
# PM_LD_CMPL      1.25478e+11  3.92118e+09  3.74541e+09  4.11658e+09
# PM_ST_CMPL      7.45961e+10  2.33113e+09  2.21164e+09  2.46327e+09
# PM_TLB_MISS     2.45894e+08  7.68418e+06  6.98733e+06  2.05724e+07
# PM_CYC           3.0575e+11  9.55467e+09  9.36585e+09  9.62227e+09
#
#                      [time]      [calls]       <%mpi>      <%wall>
# MPI_Send            188.386       639616        71.29        19.76
# MPI_Wait            69.5032       639616        26.30         7.29
# MPI_Irecv           6.34936       639616         2.40         0.67
# MPI_Barrier       0.0177442           32         0.01         0.00
# MPI_Reduce       0.00540609           32         0.00         0.00
# MPI_Comm_rank    0.00465156           32         0.00         0.00
# MPI_Comm_size   0.000145341           32         0.00         0.00
###############################################################################
```

Performance API (PAPI)

- http://icl.cs.utk.edu/papi/overview/
- Sample hardware counters
- Require instrumenting the code
- Open source performance analysis tools built on top of PAPI (PerfSuite, TAU, HPCToolkit)

i
n
t PAPI_flops
*ptime, long

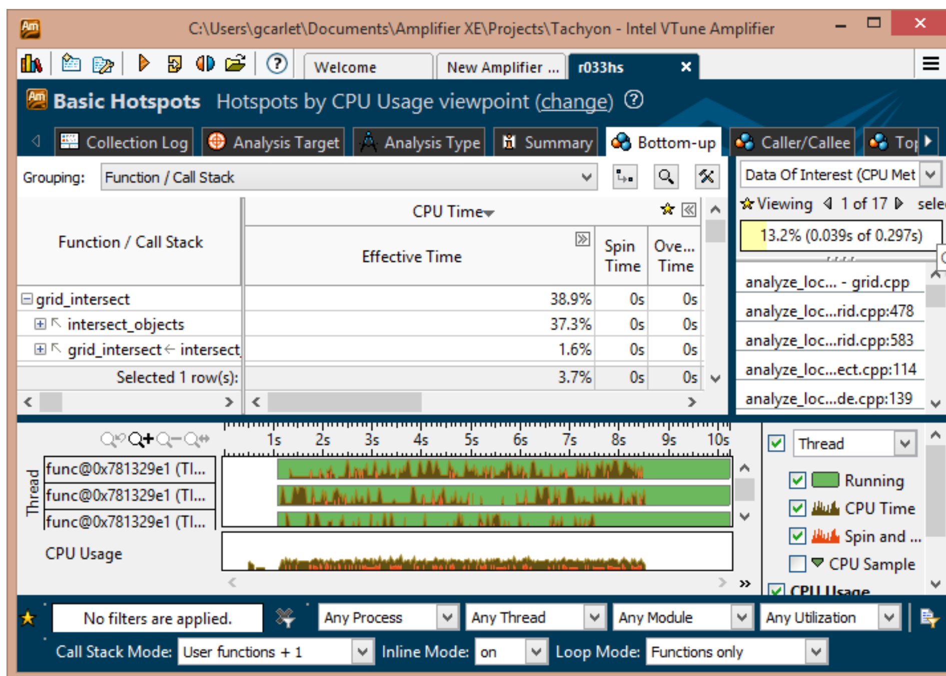


PAPI example

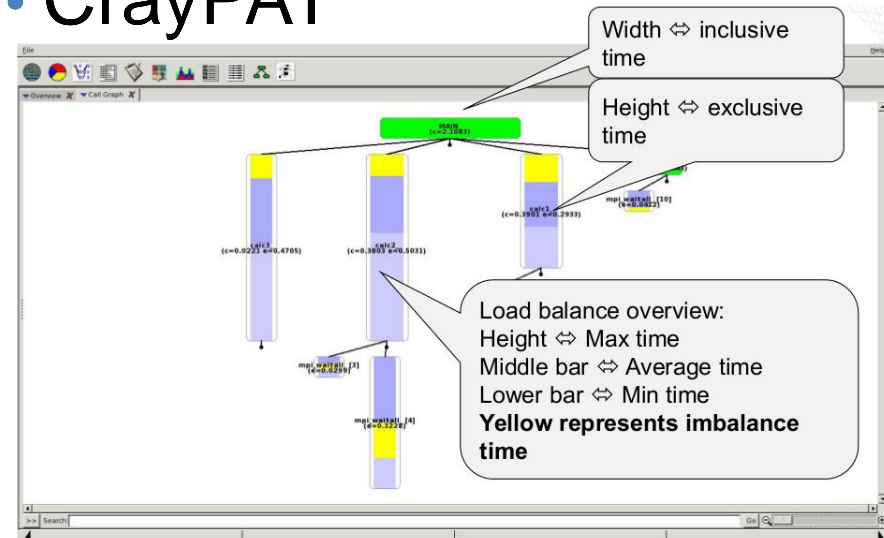
```
int events[2] = {PAPI_L1_DCM, PAPI_FP_OPS };

ret = PAPI_start_counters(events, 2);
t0 = gettime();
  /* codes to be profiled */
   ...
t1 = gettime();
ret = PAPI_start_counters(events, 2);

printf("Total hardware flops = %lld\n",(float)values[1]);
printf("MFlop/s = %f\n", (float)(TOT_FLOPS/MEGA)/(t1-t0));
printf("L2 data cache misses is %lld\n", values[0]);
```

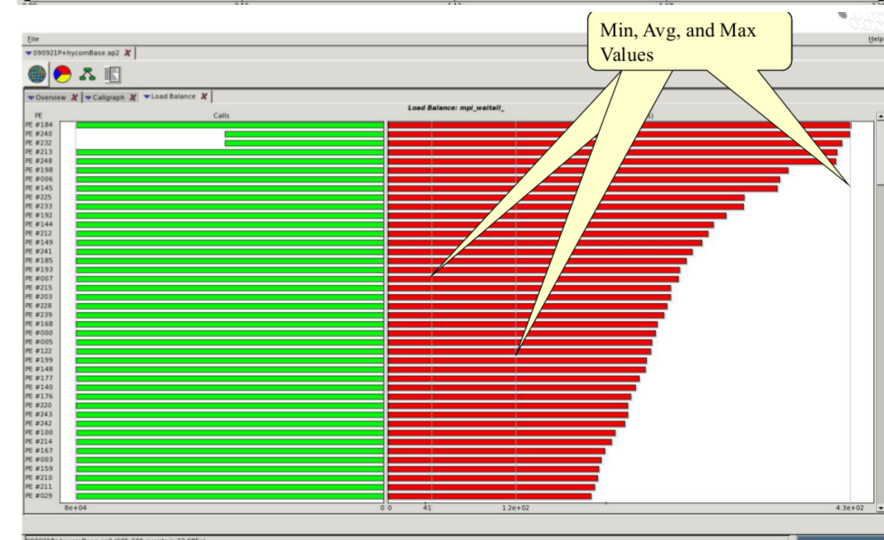
Vendor supplied performance tools

- Intel Vtune



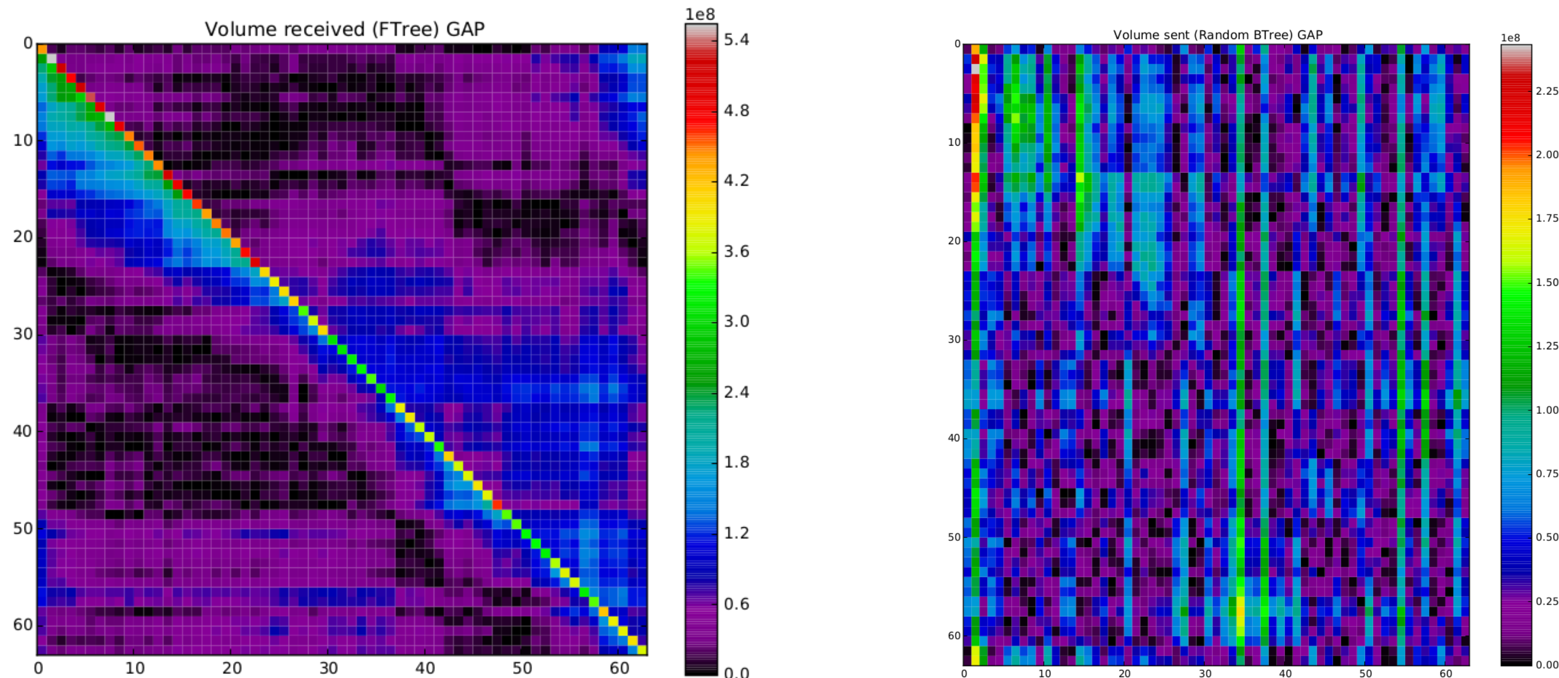
- CrayPAT





Manual profiling of target functions and events

- Using timing and simple counting (e.g., message size and number) to measure performance characteristics



Final word on performance optimization

- Optimizing a single component of the code can often be done with some effort
- Optimizing the overall performance of an application that consists of several computational components (e.g. FFT and matrix-matrix multiplication) is much harder