

Introduction to Performance Analysis and Modeling

(Viewpoint from Computer Engineering)

薛巍 (Wei XUE)

清华大学计算机科学与技术系

xuwei@tsinghua.edu.cn



◆ About me

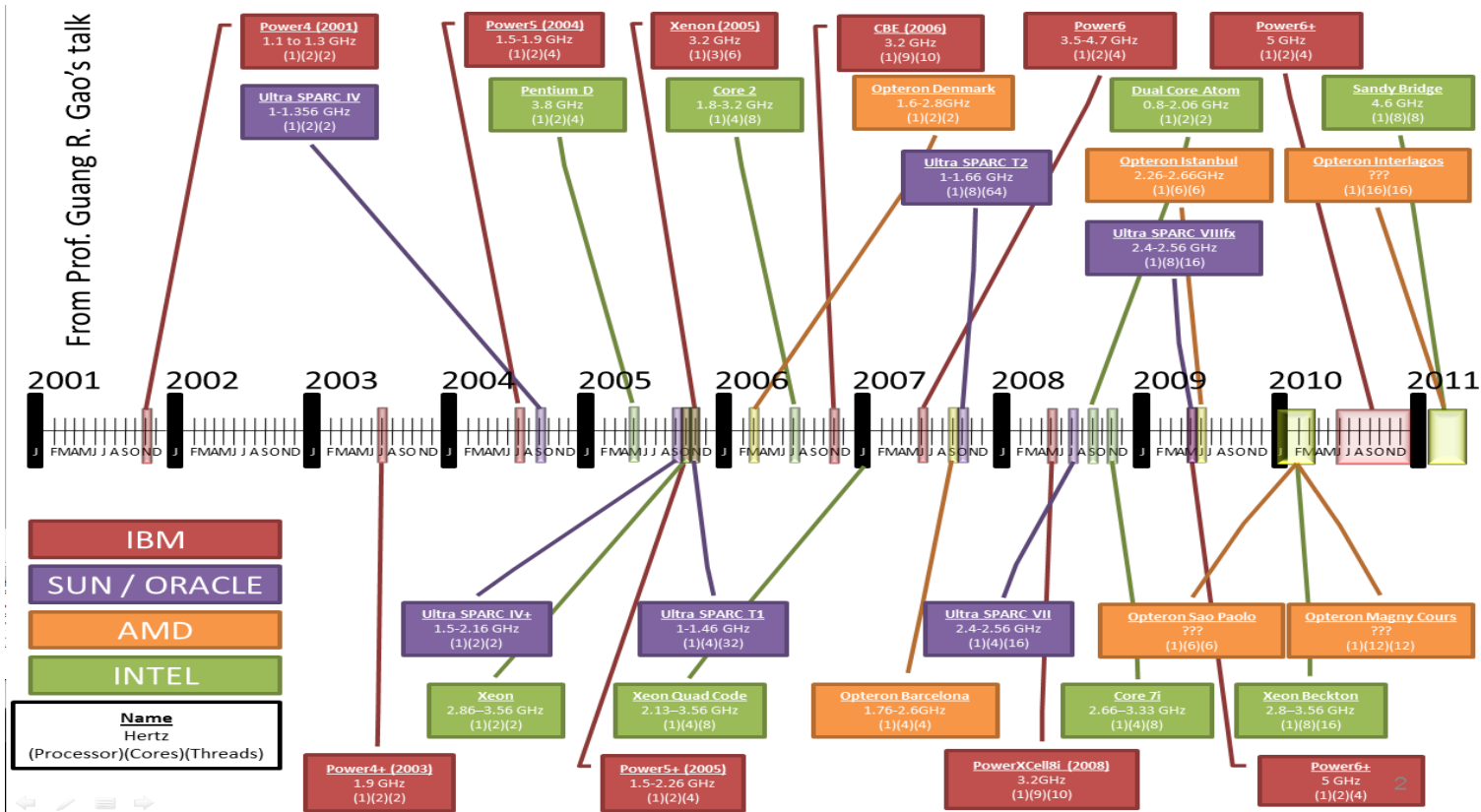
	薛 巍 (Wei XUE)
Employer	清华计算机系高性能计算研究所
Phone	13910010177
Email	xuewei@tsinghua.edu.cn
Research Interests	High Performance Computing, Scientific Computing, HPC+AI

◆ Outline

1. **Motivation**
 2. Performance Analysis of Parallel Programs
 3. Performance Modeling of Parallel Programs
-

Motivation

- Parallel computing is general purpose technology now



Question: how well do we use parallel computing systems and how can we do better?

◆ Motivation

- Processors become more and more complicated

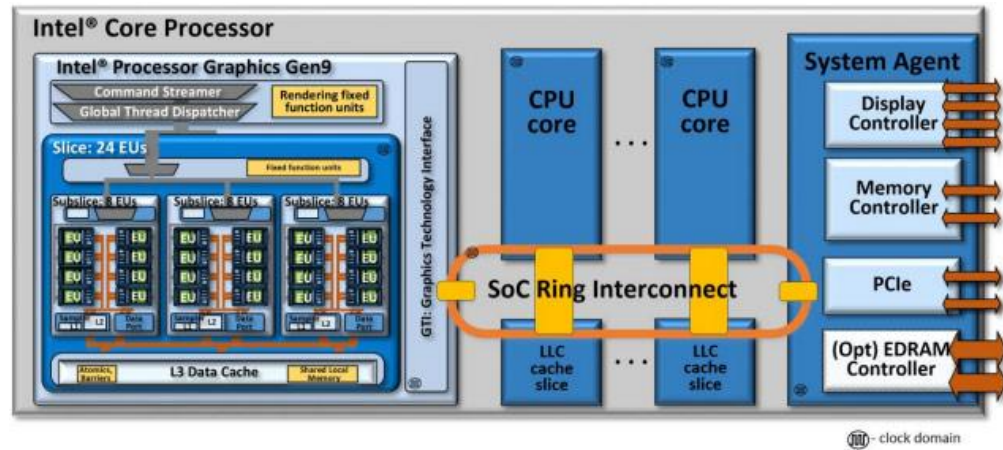
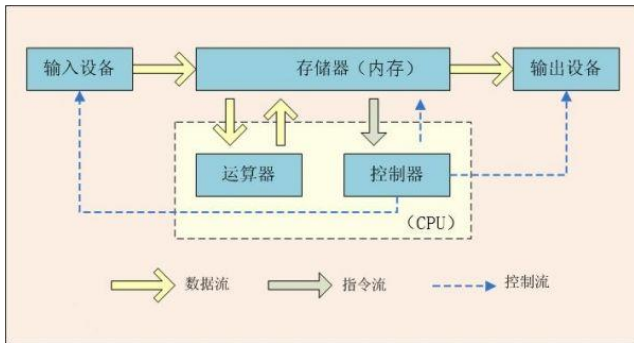
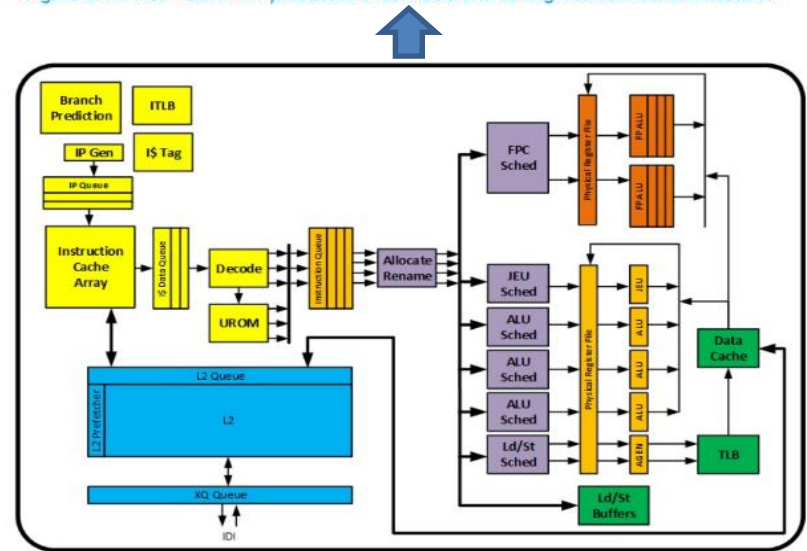
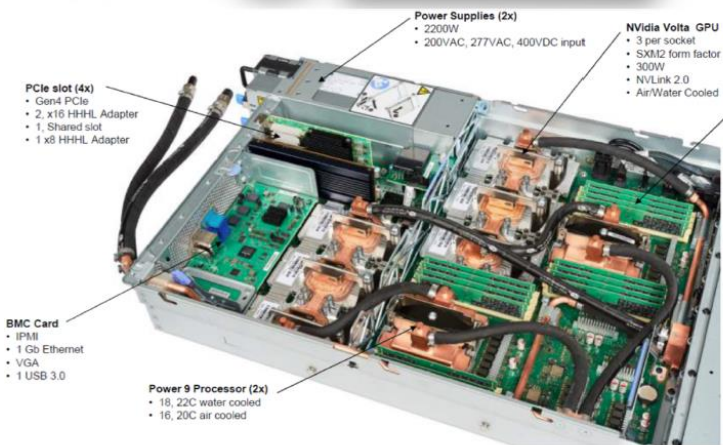


Figure 2: An Intel® Core™ i7 processor 6700K SoC and its ring interconnect architecture.

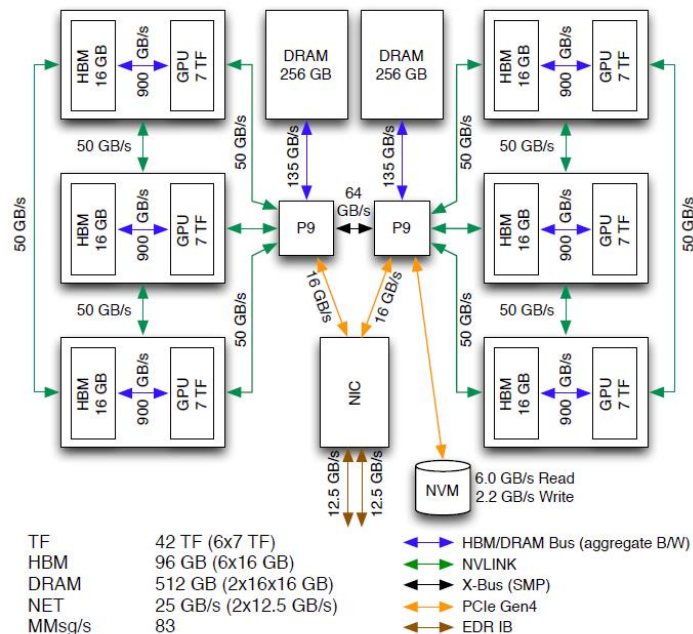


Motivation

Summit Node Overview



[无标题]

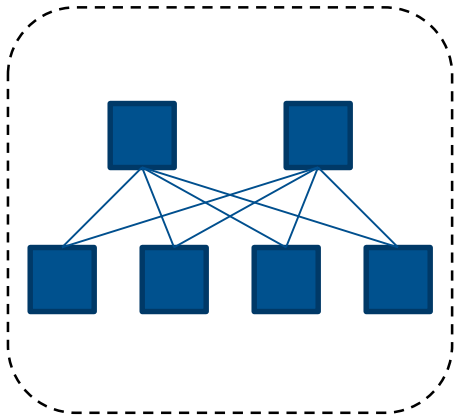


HBM & DRAM speeds are aggregate (Read+Write).
All other speeds (X-Bus, NVLink, PCIe, IB) are bi-directional.

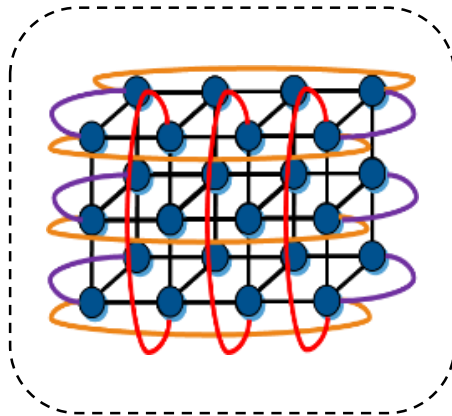
More about Node architecture: <https://queue.acm.org/detail.cfm?id=2513149&ref=qnh>

More about PCIE: <https://arstechnica.com/features/2004/07/pcie/>

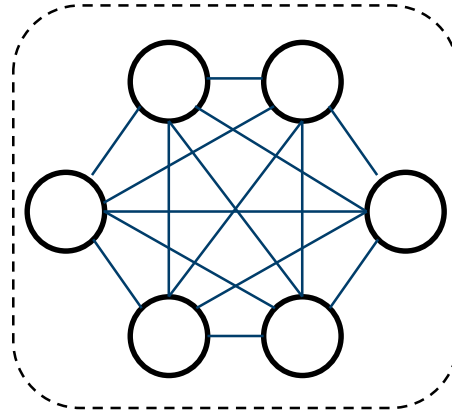
◆ Motivation



Fat Tree

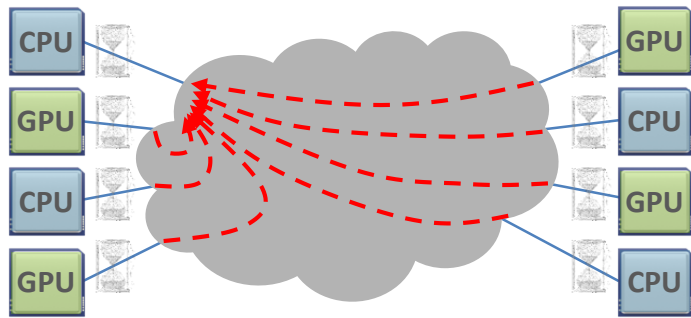


Torus

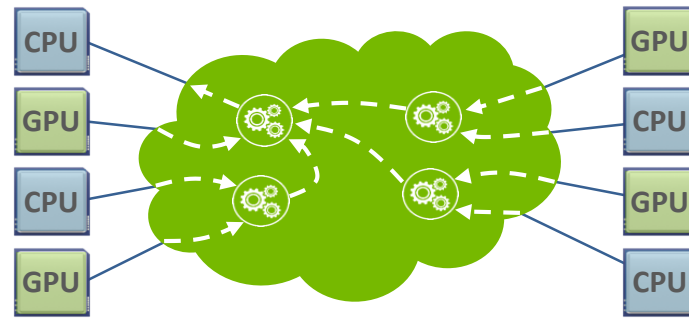


Dragonfly

bisection BW
latency
scalability
cost

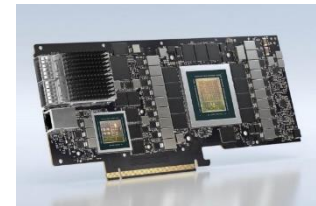


Communications Latencies
of 30-40us



Communications Latencies
of 3-4us

DPU



◆ Challenge of Performance Engineering

Performance

1 $\sim 10^3$

OH MY GOD

is tricky

`dgemm("N", "N", 50, 50, ...)`

Performance

Execution Time

Invocation

RSC_TIME

not modular



$\sim 4x$

Date



Motivation

The Top

Technology	<pre>01010011 01100011 01101001 01100101 01101110 01100011 01100101 00000000</pre> <p>Software</p>	 <p>Algorithms</p>	 <p>Hardware architecture</p>
Opportunity	Software performance engineering	New algorithms	Hardware streamlining
Examples	Removing software bloat Tailoring software to hardware features	New problem domains New machine models	Processor simplification Domain specialization

The Bottom

for example, semiconductor technology

There's plenty of room at the Top: What will drive computer performance after Moore's law?

Science, 2020.6

◆ Outline

1. Motivation
 2. **Performance Analysis of Parallel Programs**
 3. Performance Modeling of Parallel Programs
-

◆ Questions for Performance



- How can we tell if a program is performing well? Or isn't? What is "good"?
- If performance is not "good," can we identify the causes?
- What can we do for optimizations? (Not for Today)

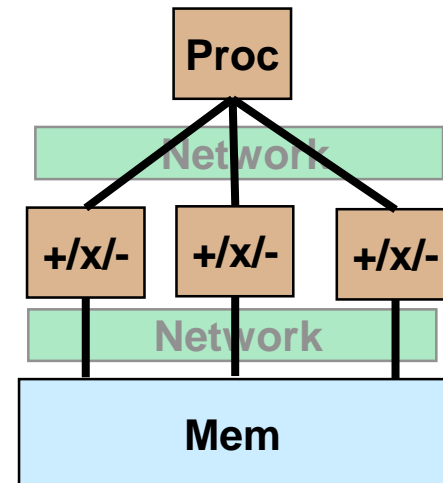
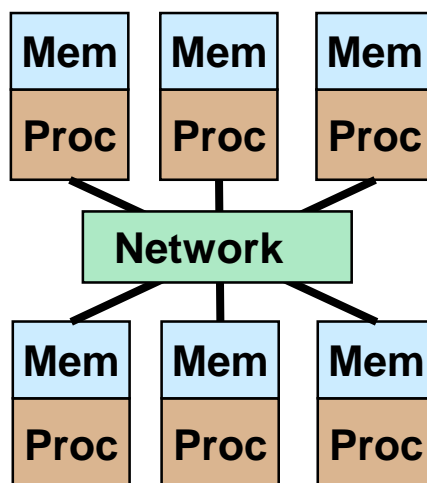
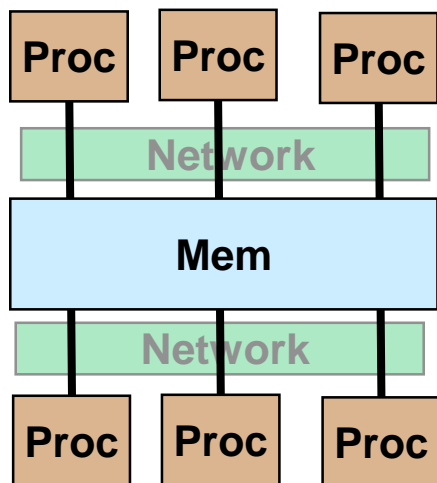
◆ Is Your Code Performing Well?

- **No single answer, but**
 - Does it scale well?
 - Is MPI time <20% of total run time?
 - Is I/O time <10% of total run time?
 - Is it load balanced?
 - If GPU code, does GPU+Processor perform better than 2 Processors?
 - **“Theoretical” CPU performance vs. “Real World” performance in a highly parallel environment**
 - Cache-based x86 processors: >10% of theoretical is pretty good
 - GPUs, Xeon Phi: >few% in today’s real full HPC applications pretty good?
 - Time-to-solution vs. sustained flops
-



Topic 1: Principles of Parallel Computing

Parallel Machines and Programming



Shared Memory

Processors execute own instruction stream

Communicate by reading/writing memory

Cost of a read/write is constant

Distributed Memory

Processors execute own instruction stream

Communicate by sending messages

Message time depends on size, but not location

Single Instruction Multiple Data (SIMD)

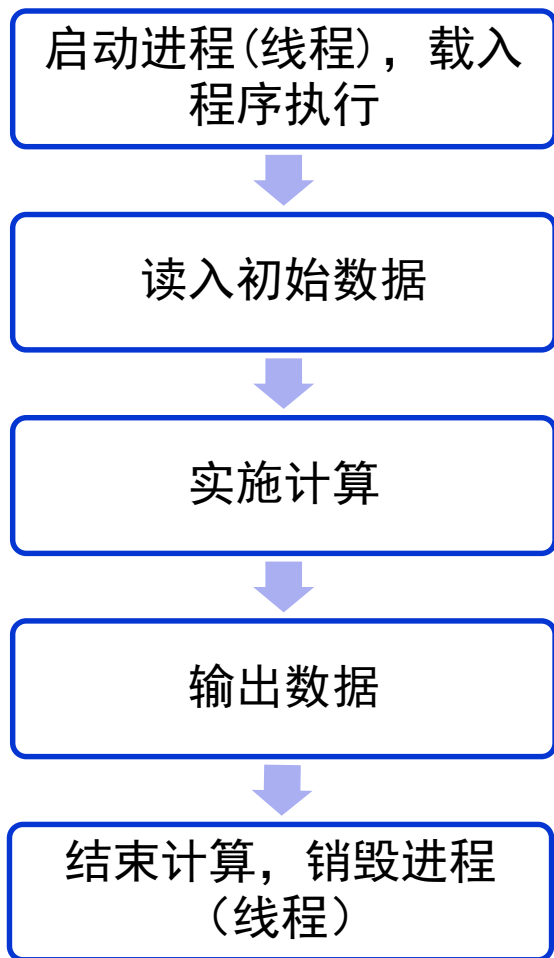
One instruction stream (all run same instruction)

Communicate through memory

Assume unbounded # of arithmetic units

- These are the natural “abstract” machine models

Abstraction of Parallel Programs



Principles of Parallel Computing

- All of these machines and implementations rely on dividing up work into parts that are:
 - Mostly independent (little synchronization)
 - About same size (load balanced)
 - Have good locality (little communication)

Writing (fast) parallel programs is not easy

- All about performance
(speed, efficiency, scale ...)
- Hardware and software co-design

Principles of Parallel Computing (details)

- Finding enough parallelism (Amdahl's Law)
- Locality – moving data costs more than arithmetic
- Load balance – don't want 1K processors to wait for one slow one
- Coordination and synchronization – sharing data safely
- Granularity – how big should each parallel task be

- Performance modeling/debugging/tuning

“Automatic” Parallelism in Modern Machines

- Bit level parallelism
 - within floating point operations, etc.
- Instruction level parallelism (ILP)
 - multiple instructions execute per clock cycle
- Memory system parallelism
 - overlap of memory operations with computation
- OS parallelism
 - multiple jobs run in parallel on commodity SMPs

Limits to all of these -- for very high performance, need user to identify, schedule and coordinate parallel tasks

Finding Enough Parallelism

- Suppose only part of an application seems parallel
- Amdahl's law
 - let s be the fraction of work done sequentially, so $(1-s)$ is fraction parallelizable
 - P = number of processors

$$\text{Speedup}(P) = \text{Time}(1)/\text{Time}(P)$$

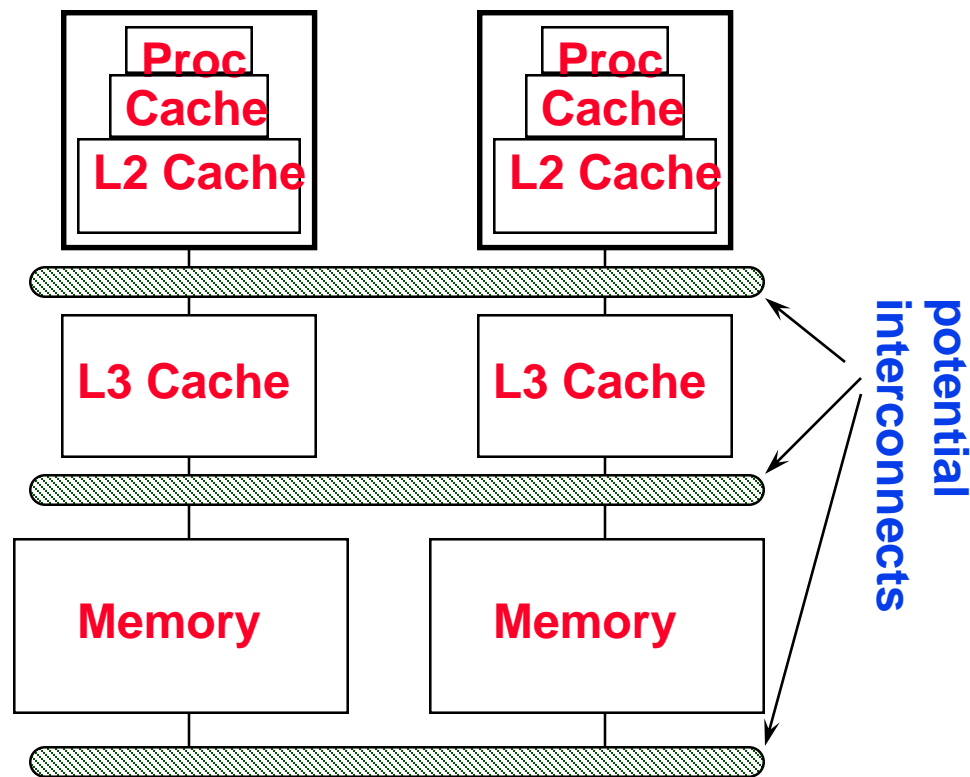
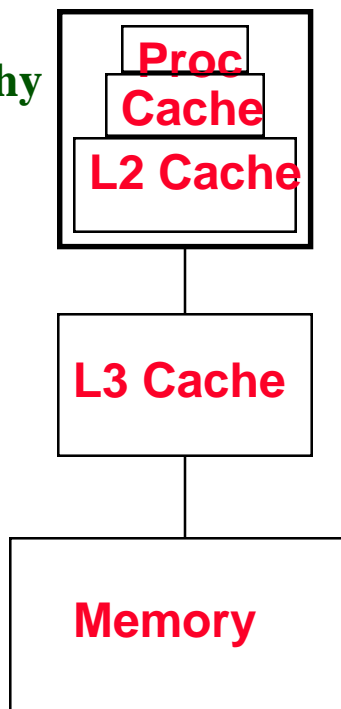
$$\leq 1/(s + (1-s)/P)$$

$$\leq 1/s$$

- **Even if the parallel part speeds up perfectly performance is limited by the sequential part**

Locality and Parallelism

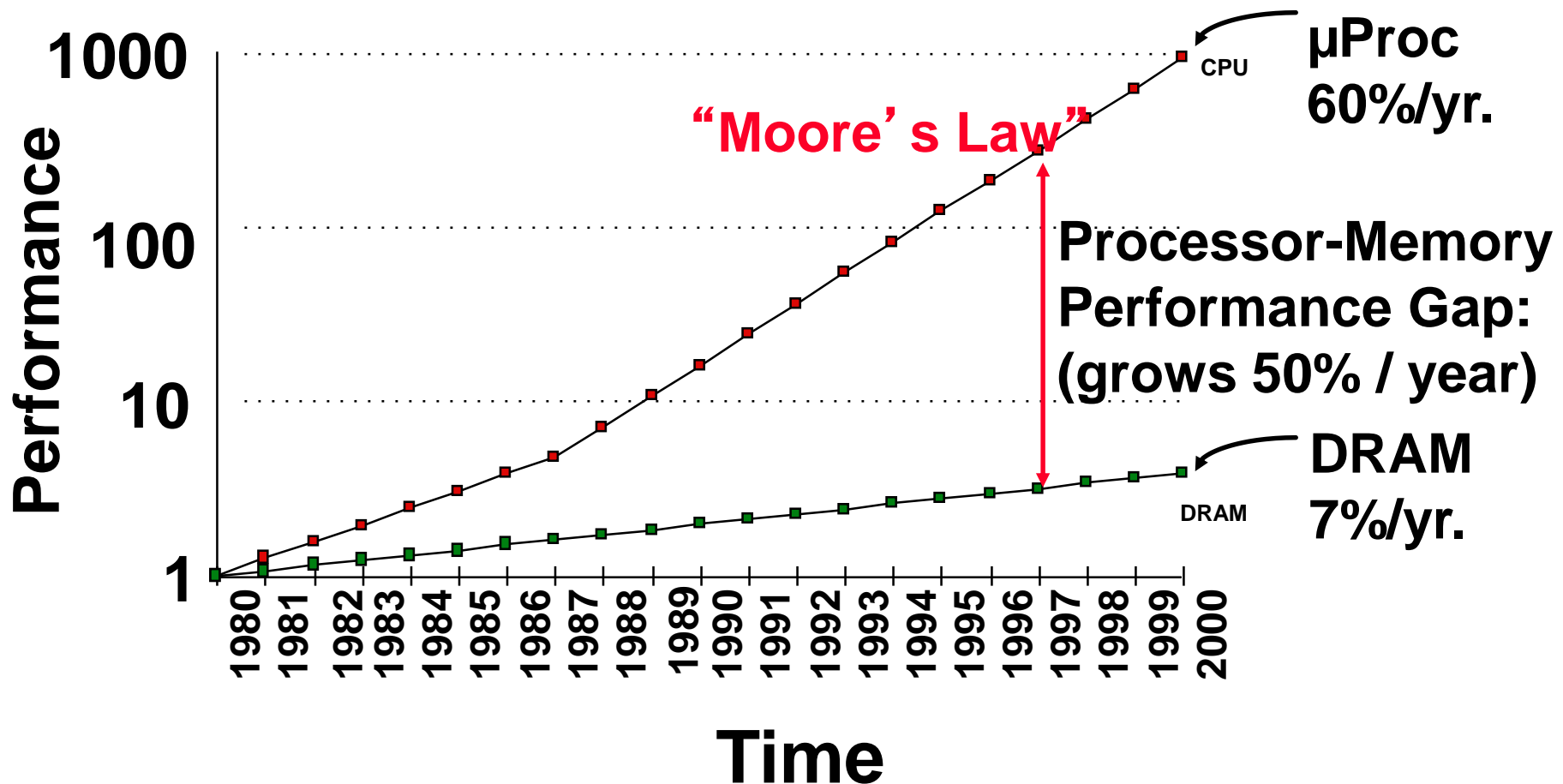
Conventional Storage Hierarchy



- Large memories are slow, fast memories are small
- Storage hierarchies are large and fast on average
- Parallel processors, collectively, have large, fast cache
- NUMA architecture
- **Algorithm should do most work on local data**

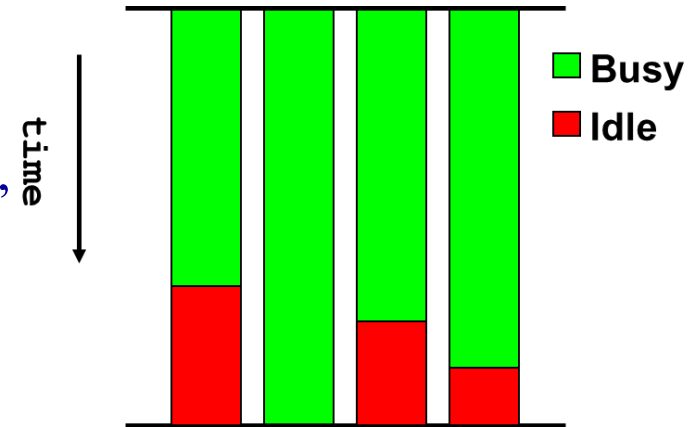
Processor-DRAM Gap (latency)

Goal: find algorithms that minimize communication, not necessarily arithmetic



Load Imbalance

- Load imbalance is the time that some processors in the system are idle due to
 - **insufficient parallelism (during that phase)**
 - **unequal size tasks**
- Examples of **the latter**
 - adapting to “interesting parts of a domain”
 - tree-structured computations
 - fundamentally unstructured problems
- Algorithm needs to balance load
 - Sometimes can determine work load, divide up evenly, before starting
 - “Static Load Balancing”
 - Sometimes work load changes dynamically, need to rebalance dynamically
 - “Dynamic Load Balancing”



Overhead of Parallelism

- Given enough parallel work, this is the biggest barrier to get desired performance
- Parallelism overheads include:
 - cost of starting threads or processes
 - cost of communicating shared data
 - cost of synchronization
 - extra (redundant) computation

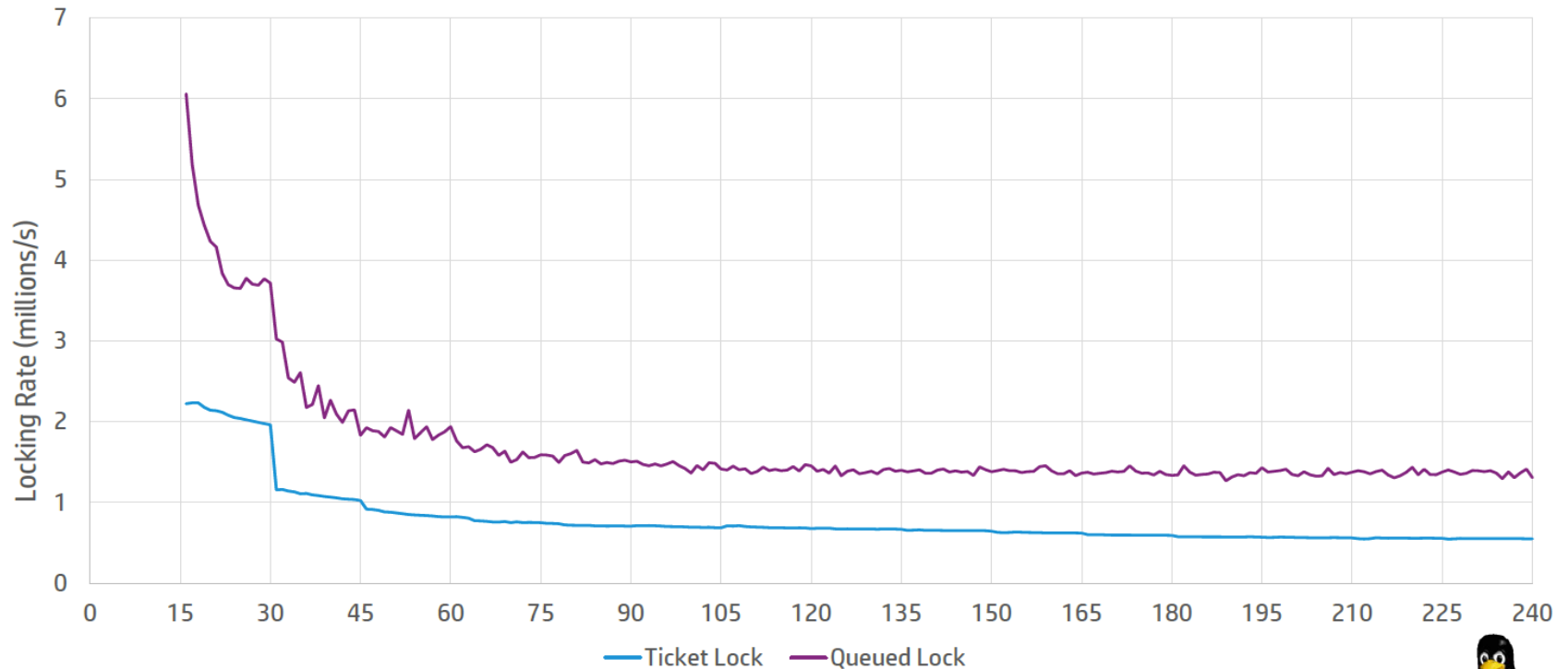
Overhead of Parallelism

A list of [numbers](#) every programmer should know (Jeff Dean, google)

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
<u>Mutex lock/unlock</u>	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
<u>Send 2K bytes over 1 Gbps network</u>	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
<u>Round trip within same datacenter</u>	500,000 ns
<u>Disk seek</u>	10,000,000 ns
<u>Read 1 MB sequentially from disk</u>	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

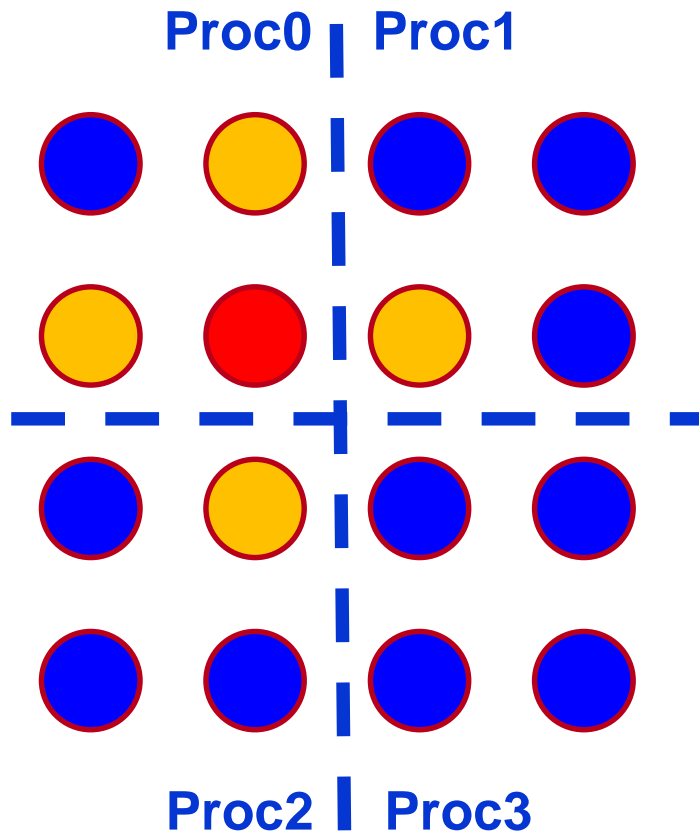
Lock overhead

Ticket Lock vs. Queued Lock (16-240 Threads, No Load)

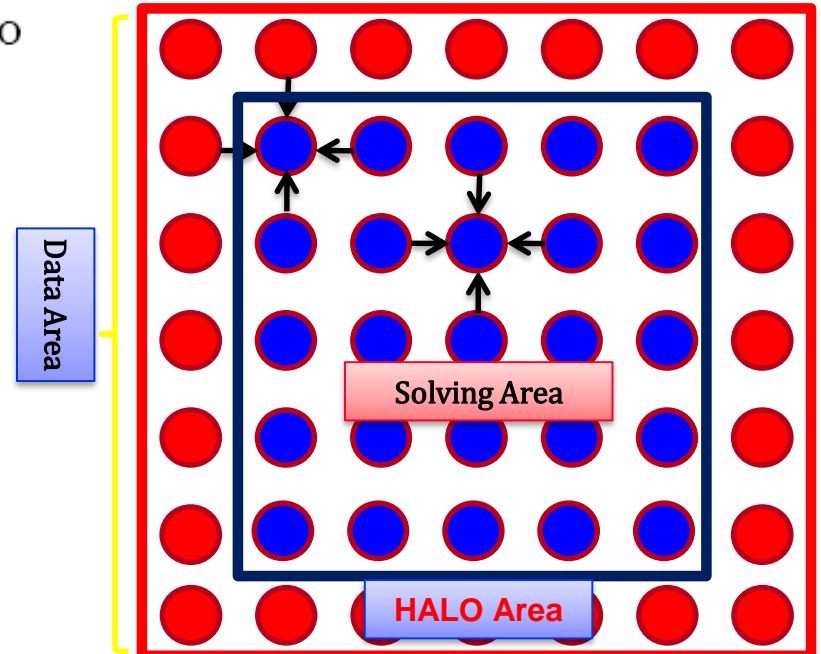


- Ticket spinlock is the spinlock implementation used in the Linux kernel prior to 4.2. A lock waiter gets a ticket number and spin on the lock cacheline until it sees its ticket number. By then, it becomes the lock owner and enters the critical section.
- Queued spinlock is the new spinlock implementation used in 4.2 Linux kernel and beyond. A lock waiter goes into a queue and spins in its own cacheline until it becomes the queue head. By then, it can spin on the lock cacheline and attempt to get the lock.

Redundant computation



```
DO J=1,N
  DO I=1,N
    B(I,J)=0.25*(A(I-1,J)+A(I+1,J)+A(I,J+1)+A(I,J-1))
  END DO
END DO
DO J=1,N
  DO I=1,N
    A(I,J)=B(I,J)
  END DO
END DO
```

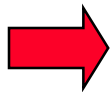


Granularity

- Each of these overhead mentioned can be in the range of milliseconds (=millions of flops) on some systems
- Tradeoff: Algorithm needs **sufficiently large units of work** to run fast in parallel (i.e. large granularity), but **not so large** that there is not enough parallel work

Principles of Parallel Computing

- Finding enough parallelism (Amdahl's Law)
- Locality – moving data costs more than arithmetic
- Load balance – don't want 1K processors to wait for one slow one
- Coordination and synchronization – sharing data safely
- Granularity – how big should each parallel task be
- Performance modeling/debugging/tuning



All of these things makes parallel programming even harder than sequential programming.



Topic 2: Method and Tools of performance analysis

how to benchmarking?

How to collect performance data?

How to analyze the data?

Performance Analysis of Parallel Programs

- **Correctly and comprehensively** diagnose performance behaviors of parallel programs, to support effective optimizations
 - **Experiment design and implementation**, have to care
 - Representative of samples
 - External constrains on the viewpoint of system level
 - Correctness check
 - Scientific benchmarking of Parallel Computing Systems
 - **Methods for Performance analysis**
 - Speedup, efficiency
 - Amdahl and Gustafson laws, critical path
 - **Performance data collection**
 - **Profiling**: statistics of program performance behaviors, such as time of functions
 - Subroutine profiling vs. PMU sampling
 - **Tracing**: records of most of the events happened, such as sequence of memory access and instructions
 - Performance tools

Mostly used profiling: Timing of Parallel Program

■ Timers

- Linux: rdtsc (cycle precision) / gettimeofday(us) / clock_getres, clock_gettime(ns)
- Windows: __rdtsc (cycle precision) / QueryPerformanceFrequency / QueryPerformanceCounter
- Intel Timer Utility: <https://software.intel.com/en-us/code-samples/intel-c-compiler/utilities/Timer-Utility>
- MPI_Wtime/MPI_Wtick

■ Timing of Parallel Programs

1. Barrier
2. Start Timer
3. Run Program
4. End Timer
5. $\text{Max}(\text{EndTime}[i] - \text{StartTime}[i])$

Hardware performance monitoring unit (PMU)

- Most processors nowadays have special, on-chip hardware that monitors micro architectural events
 - Core: instructions retired, elapsed core clock ticks, core frequency including Intel® Turbo boost technology, L2 cache hits and misses, L3 cache misses and hits (including or excluding snoops).
 - Uncore: read bytes from memory controller(s), bytes written to memory controller(s), data traffic transferred by the Intel® QuickPath Interconnect links.
- It is a subsystem on the processor which helps in analyzing how an application or operating systems are performing on the processor.
- The Performance Monitoring Events can be broadly categorized in two types
 - Hardware: CPU-Cycles, Instructions, SIMD Instructions, Cache References, Memory Access, Stalls, TLB miss
 - Software: Page Fault, Context Switch, etc

Hardware performance monitoring unit (PMU)

- PMU consists of two components:

Performance Event Select Registers

Configuration registers to control what events to be monitored and how to monitor.

Event Counters (both configurable and fix ones)

The registers which actually count the number of events based on the event select register's configuration.

- For monitoring an event a counter is paired with an event select register.

Sampling of PMU

- Periodically interrupt the processor to obtain execution status and context
 - Periodic sampling
 - OS Timer Services (RTC), Every N Processor Clockticks
 - Event Based Sampling (EBS).
 - Specific processor events, such as L2 Cache Misses, Branch Mispredictions, Floating-point instructions retired.
- Manual sampling

```
1 PCM * m = PCM::getInstance();
2
3 // program counters, and on a failure just exit
4
5 if (m->program() != PCM::Success) return;
6
7 SystemCounterState before_sstate = getSystemCounterState();
8
9     [run your code here]
10
11 SystemCounterState after_sstate = getSystemCounterState();
12
13 cout << "Instructions per clock:" << getIPC(before_sstate,after_sstate)
14 "L3 cache hit ratio:" << getL3CacheHitRatio(before_sstate,after_sstate)
15 "Bytes read:" << getBytesReadFromMC(before_sstate,after_sstate)
16 [and so on]...
```

Advantages of PMU sampling

- No code modification and no libraries to link
 - May need to add compiler option (-g) to get the program call-stack information
- System level Sampling
 - Not only your program is sampled, but all the programs are scheduled to run on the same processor/core are sampled
- Low overhead and high accuracy (when your program/function takes long time enough)

Hotspot Analysis (热点分析) with PMU

- Where in an application/system where there is a significant amount of activity
 - Where = Memory Address => OS Process => OS Thread => Executable or module => Function => Lines of code
 - Significant = 如果相关活动并不频繁发生，就可能对系统性能不会造成太大影响
 - Activity = 花费的时间或者其它处理器内部事件
 - Cache misses, Branch mispredictions, Floating-point instructions retired, Partial register stalls, etc.
- From hotspots to bottleneck

Performance Analysis of Parallel Programs

- Correctly and comprehensively diagnose performance behaviors of parallel programs, to support effective optimizations
 - Experiment design and implementation, have to care
 - Representative of samples
 - External constraints on the viewpoint of system level
 - Correctness check
 - Scientific benchmarking of Parallel Computing Systems
 - **Methods for Performance analysis**
 - Speedup, efficiency
 - Amdahl law, Gustafson laws, and critical path analysis
 - Performance data collection
 - Profiling: statistics of program performance behaviors, such as time of functions
 - Subroutine profiling vs. PMU sampling
 - Tracing: records of most of the events happened, such as sequence of memory access and instructions
 - Performance tools

Speedup

- The *speedup* of a parallel application is

$$\text{Speedup}(p) = \text{Time}(1)/\text{Time}(p)$$

- Where

- $\text{Time}(1)$ = execution time for a single processor and
- $\text{Time}(p)$ = execution time using p parallel processors

- If $\text{Speedup}(p) = p$ we have *perfect speedup* (also called *linear scaling*)

- As defined, speedup compares an application with itself on one and on p processors, but it is more useful to compare

- The execution time of the best serial application on 1 processor

versus

- The execution time of best parallel algorithm on p processors

Superlinear Speedup

Question: can we find “*superlinear*” speedup, that is

$$\text{Speedup}(p) > p \quad ?$$

- Choosing a bad “baseline” for $T(1)$
 - Old serial code has not been updated with optimizations
 - Avoid this, and always specify what your baseline is
- Shrinking the problem size per processor
 - May allow it to fit in small fast memory (cache)
- Application is not deterministic
 - Amount of work varies depending on execution order
 - Search algorithms have this characteristic

Efficiency

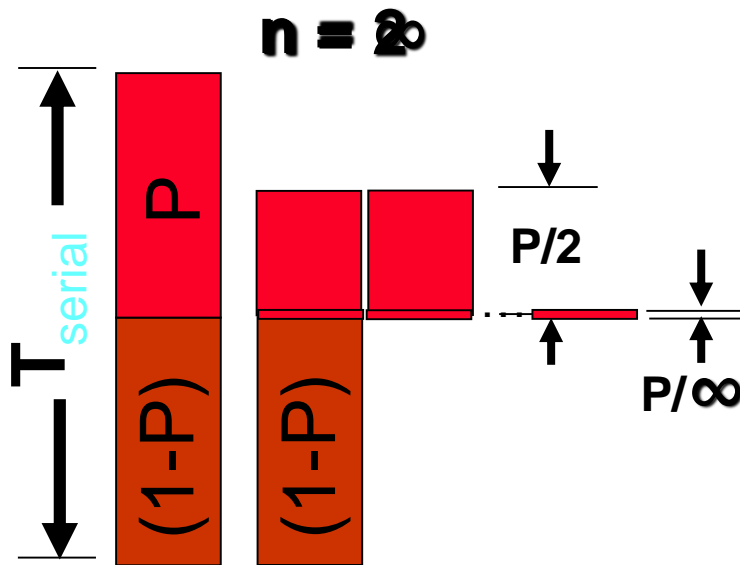
- The *parallel efficiency* of an application is defined as

$$\text{Efficiency}(p) = \text{Speedup}(p)/p$$

- $\text{Efficiency}(p) \leq 1$
- For perfect speedup $\text{Efficiency}(p) = 1$
- We will rarely have perfect speedup.
 - Lack of perfect parallelism in the application or algorithm
 - Imperfect load balancing (some processors have more work)
 - Cost of communication
 - Cost of contention for resources, e.g., memory bus, I/O
 - Synchronization time
- Understanding why an application is not scaling linearly will help finding ways improving the applications performance on parallel computers.

Amdahl law

- Describes the upper bound of parallel execution speedup



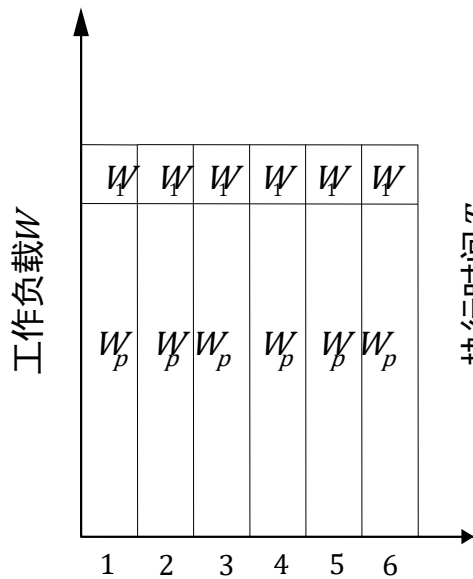
$$T_{\text{parallel}} = \{0.5 + \frac{0.5}{\infty}\} T_{\text{serial}}$$

$n = \text{number of processors}$

$$\text{Speedup} = T_{\text{serial}} / T_{\text{parallel}} = 1.0 / 0.55 = 2.033$$

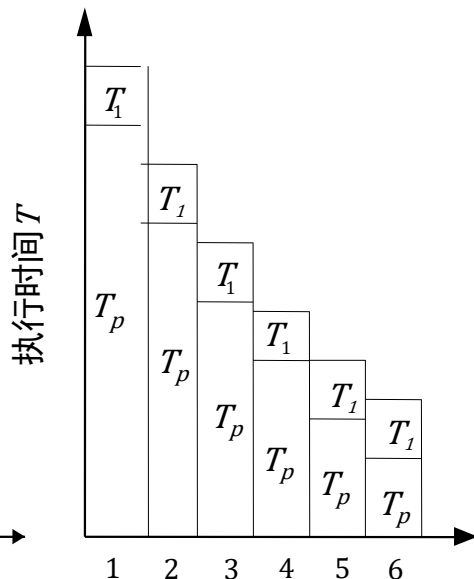
Serial code limits speedup

Amdahl law



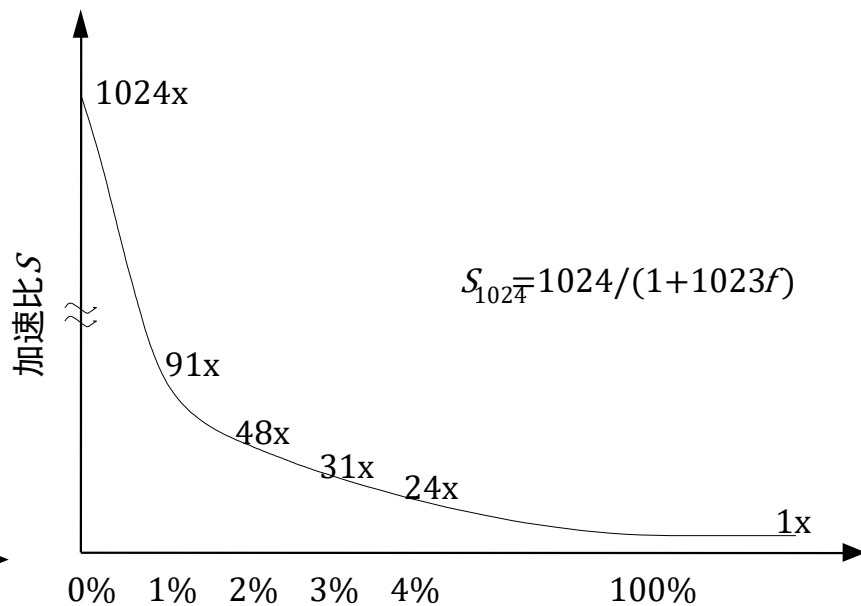
处理器数 P

(a)



处理器数 P

(b)



程序中顺序部分的百分比 f

(c)

Gustafson law

- Use more computing resources (processors/nodes) to solve larger problem

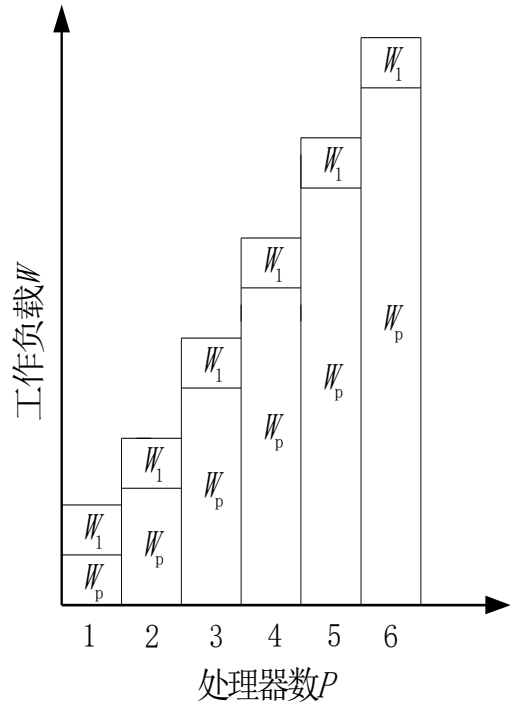
$$S' = \frac{W_S + p W_P}{W_S + p \cdot W_P / p} = \frac{W_S + p W_P}{W_S + W_P}$$

$$S' = f + p(1-f) = p + f(1-p) = p - f(p-1)$$

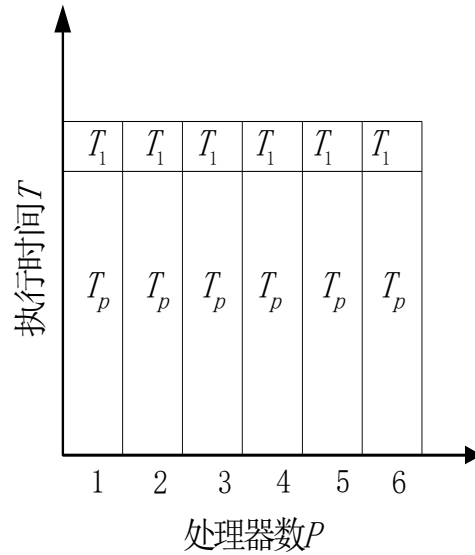
If parallel overhead W_o

$$S' = \frac{W_S + p W_P}{W_S + W_P + W_o} = \frac{f + p(1-f)}{1 + W_o / W}$$

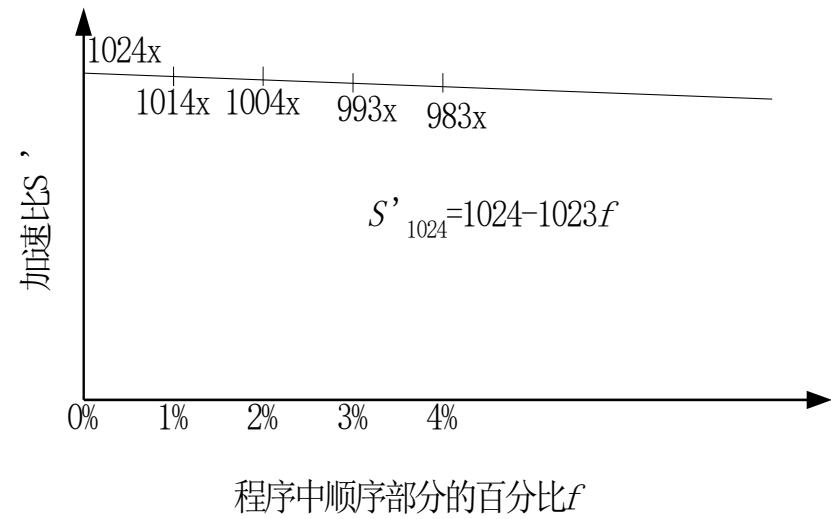
Gustafson Law



(a)



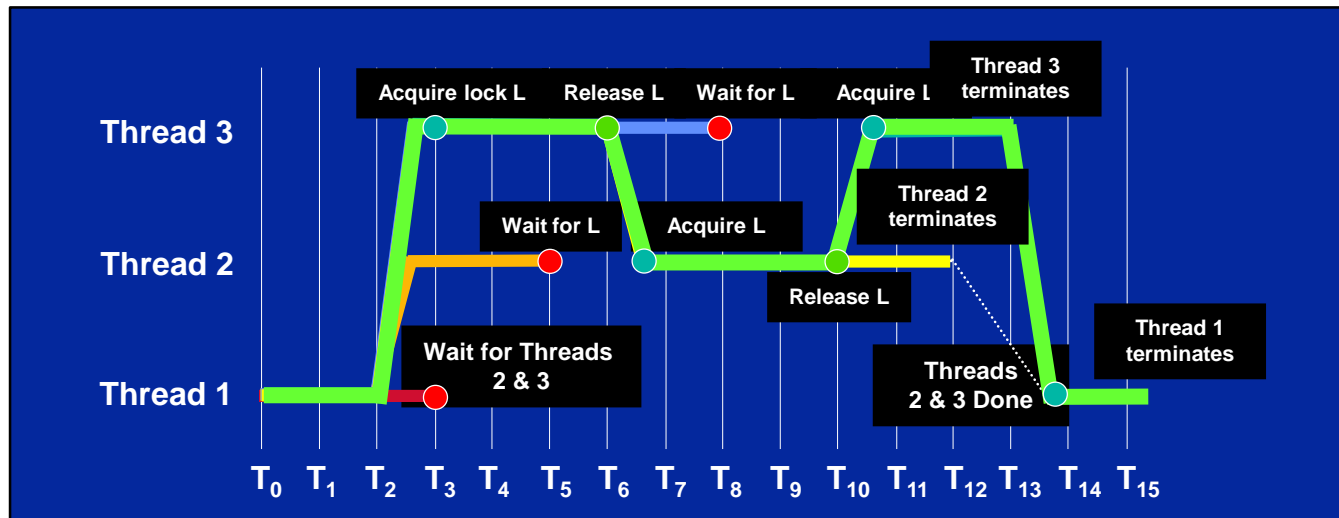
(b)



(c)

Critical Path

- Parallel applications contain multiple execution flows
- A new flow is created when a thread/process is created or resumes
- Flow ends when a thread/process terminates or blocks on a synchronization primitive



The **critical path** is the longest execution flow

Performance Analysis of Parallel Programs

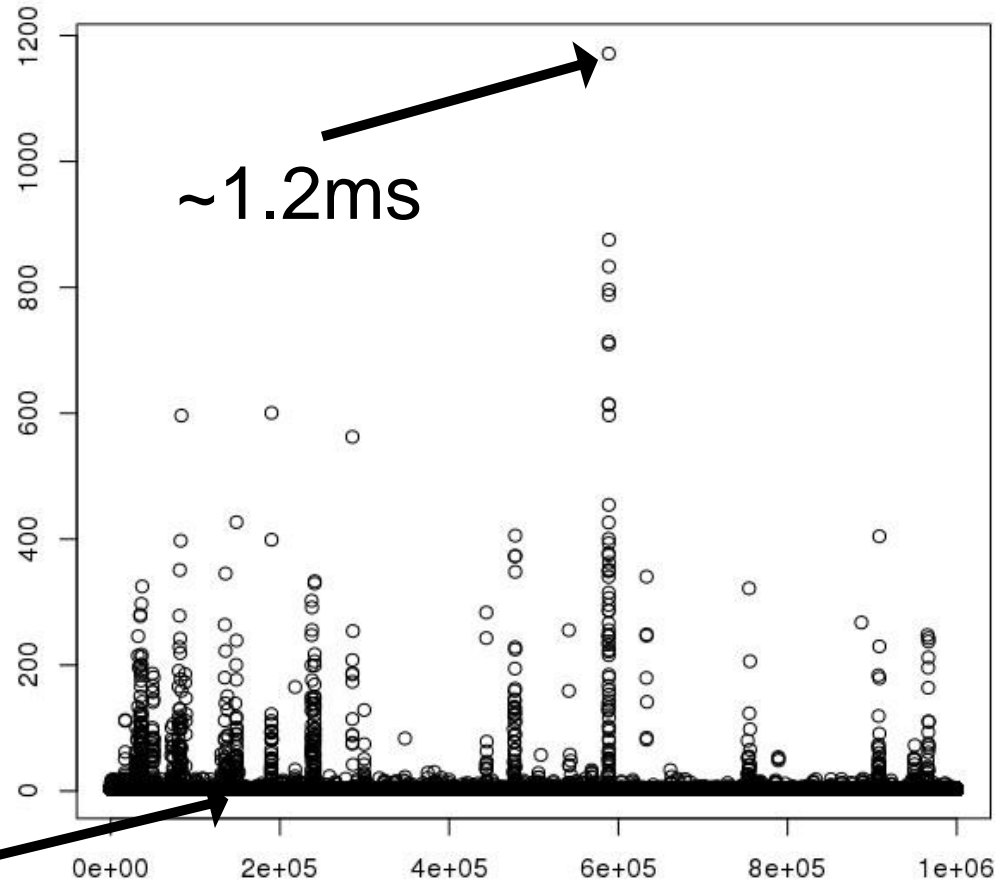
- Correctly and comprehensively diagnose performance behaviors of parallel programs, to support effective optimizations
 - Experiment design and implementation, have to care
 - Representative of samples
 - External constraints on the viewpoint of system level
 - Correctness check
 - **Scientific benchmarking of Parallel Computing Systems**
 - Methods for Performance analysis
 - Speedup, efficiency
 - Amdahl law, Gustafson laws, and critical path analysis
 - Performance data collection
 - Profiling: statistics of program performance behaviors, such as time of functions
 - Subroutine profiling vs. PMU sampling
 - Tracing: records of most of the events happened, such as sequence of memory access and instructions
 - Performance tools

Performance may be changing



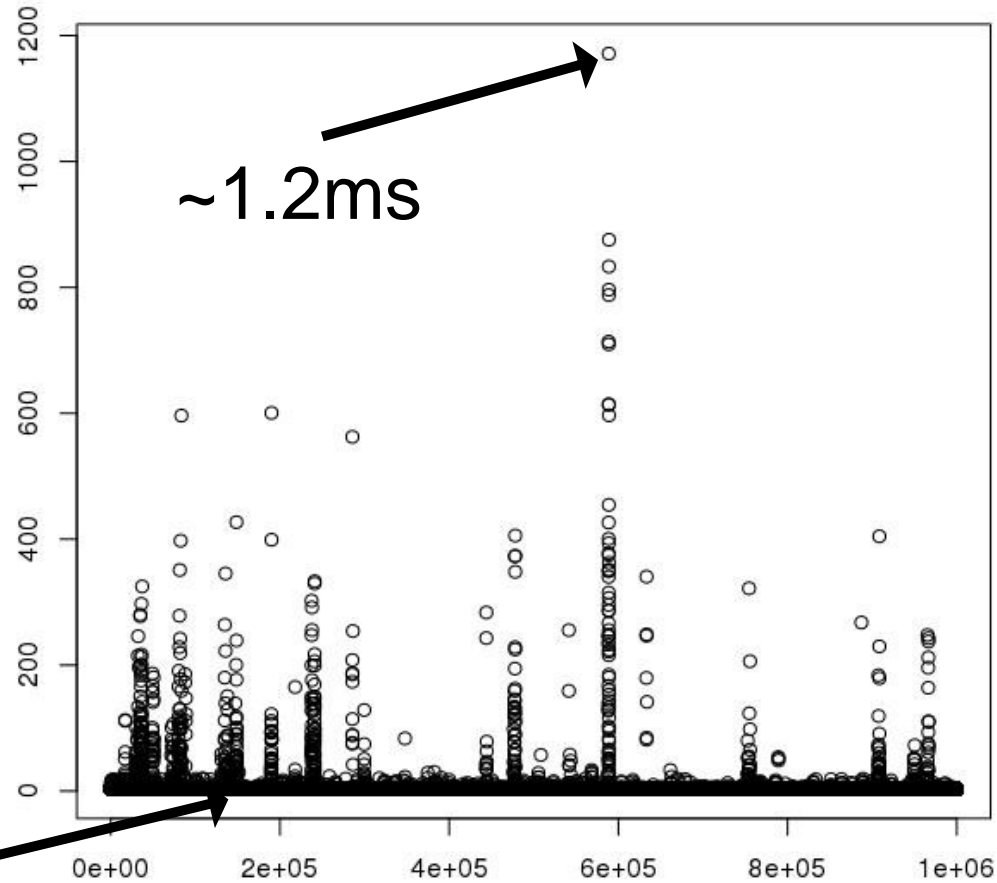
- What is the real performance of point-to-point communication on this supercomputer?

- 1.77us?



~1.77us

Performance may be changing



- How to report performance results of point-to-point communication
 - Normal distribution?
 - confidence interval?

~1.77us

Performance analysis in contended system

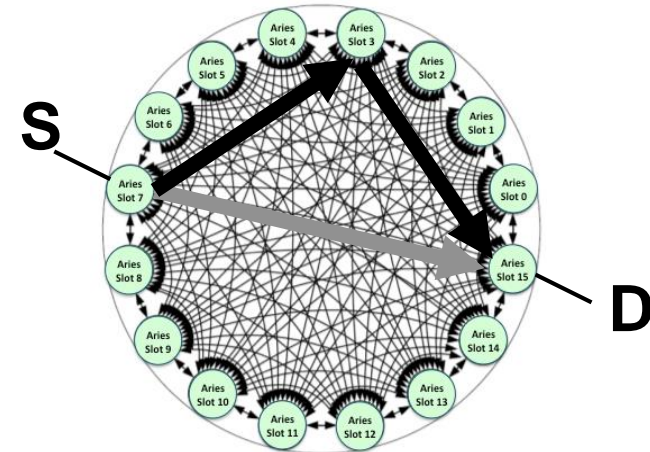
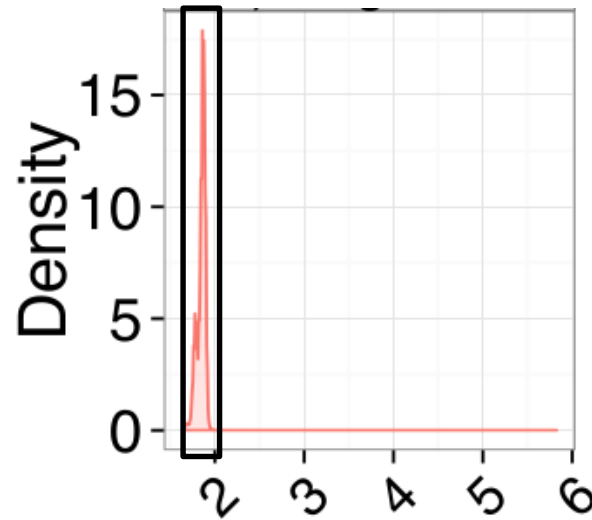
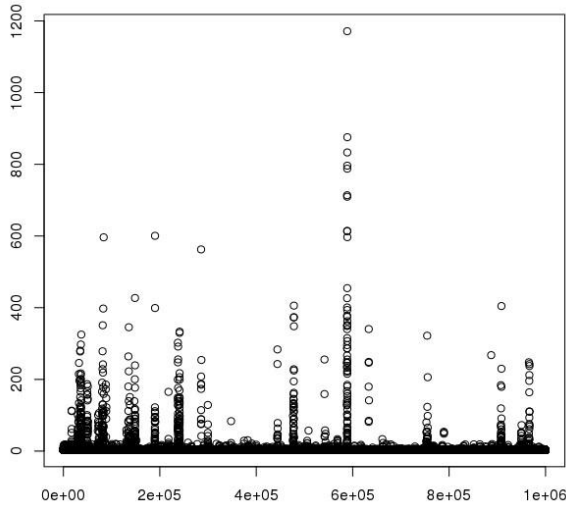
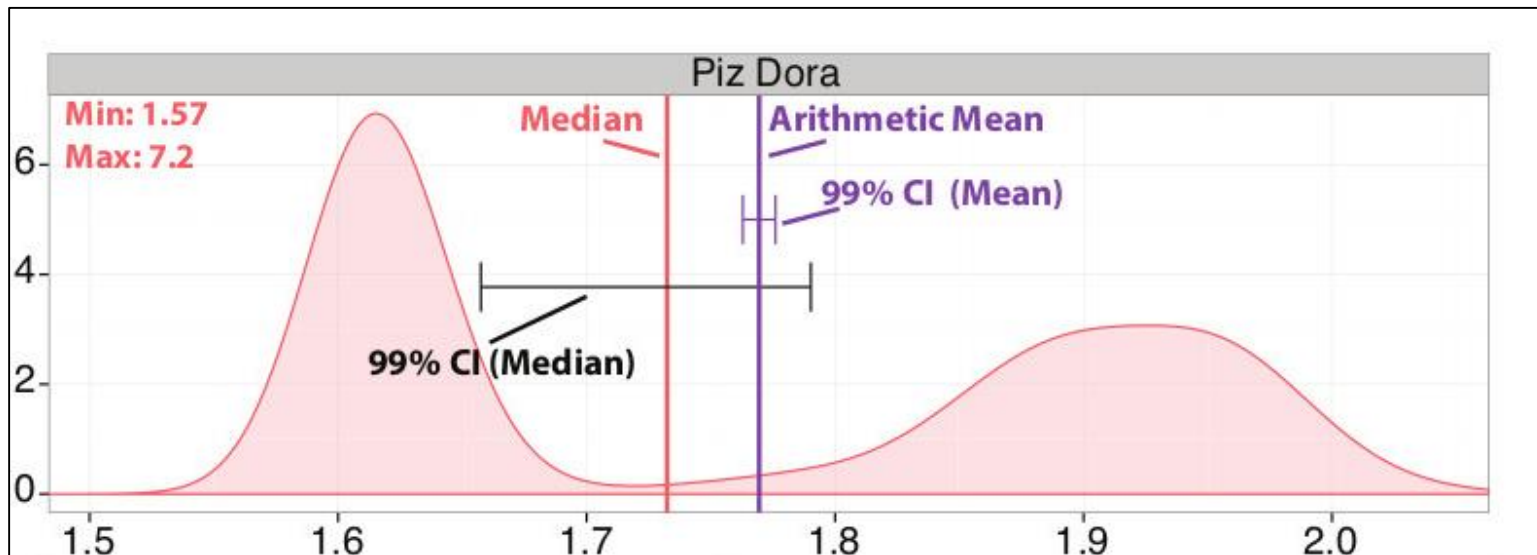


Image credit: nersc.gov



Scientific benchmarking of parallel computing systems

ACM/IEEE Supercomputing 2015 (SC15) + talk online on youtube!

Scientific Benchmarking of Parallel Computing Systems

Twelve ways to tell the masses when reporting performance results

Torsten Hoefler
Dept. of Computer Science
ETH Zurich
Zurich, Switzerland
htor@inf.ethz.ch

Roberto Belli
Dept. of Computer Science
ETH Zurich
Zurich, Switzerland
bellir@inf.ethz.ch

ABSTRACT

Measuring and reporting performance of parallel computers constitutes the basis for scientific advancement of high-performance computing (HPC). Most scientific reports show performance improvements of new techniques and are thus obliged to ensure reproducibility or at least interpretability. Our investigation of a stratified sample of 120 papers across three top conferences in the field shows that the state of the practice is lacking. For example, it is often unclear if reported improvements are deterministic or observed by chance. In addition to distilling best practices from existing work, we propose statistically sound analysis and reporting techniques and simple guidelines for experimental design in parallel computing and codify them in a portable benchmarking library. We aim to improve the standards of reporting research results and initiate a discussion in the HPC field. A wide adoption of our minimal set of rules will lead to better interpretability of performance results and improve the scientific culture in HPC.

Categories and Subject Descriptors

D.2.8 [Software Engineering]: Metrics—*complexity measures, performance measures*

Keywords

Benchmarking, parallel computing, statistics, data analysis

1. INTRODUCTION

Reproducing experiments is one of the main principles of the scientific method. It is well known that the performance of a computer program depends on the application, the input, the compiler, the runtime environment, the machine, and the measurement methodology [20, 43]. If a single one of these aspects of *experimental design* is not appropriately motivated and described, presented results can hardly be reproduced and may even be misleading or incorrect.

The complexity and uniqueness of many supercomputers makes reproducibility a hard task. For example, it is practically impossible to recreate most hero-runs that utilize the world's largest machines because these machines are often unique and their software configurations changes regularly. We introduce the notion of *interpretability*, which is weaker than reproducibility. We call an *experiment interpretable* if it provides enough information to allow scientists to understand the experiment, draw own conclusions, assess their certainty, and possibly generalize results. In other words, interpretable experiments support sound conclusions and convey precise information among scientists. Obviously, every scientific paper should be interpretable; unfortunately, many are not.

For example, reporting that an High-Performance Linpack (HPL) run on 64 nodes (N=314k) of the Piz Daint system during normal operation (cf. Section 4.1.2) achieved 77.38 Tflop/s is hard to interpret. If we add that the theoretical peak is 94.5 Tflop/s, it becomes clearer, the benchmark achieves 81.8% of peak performance. But is this true for every run or a typical run? Figure 1

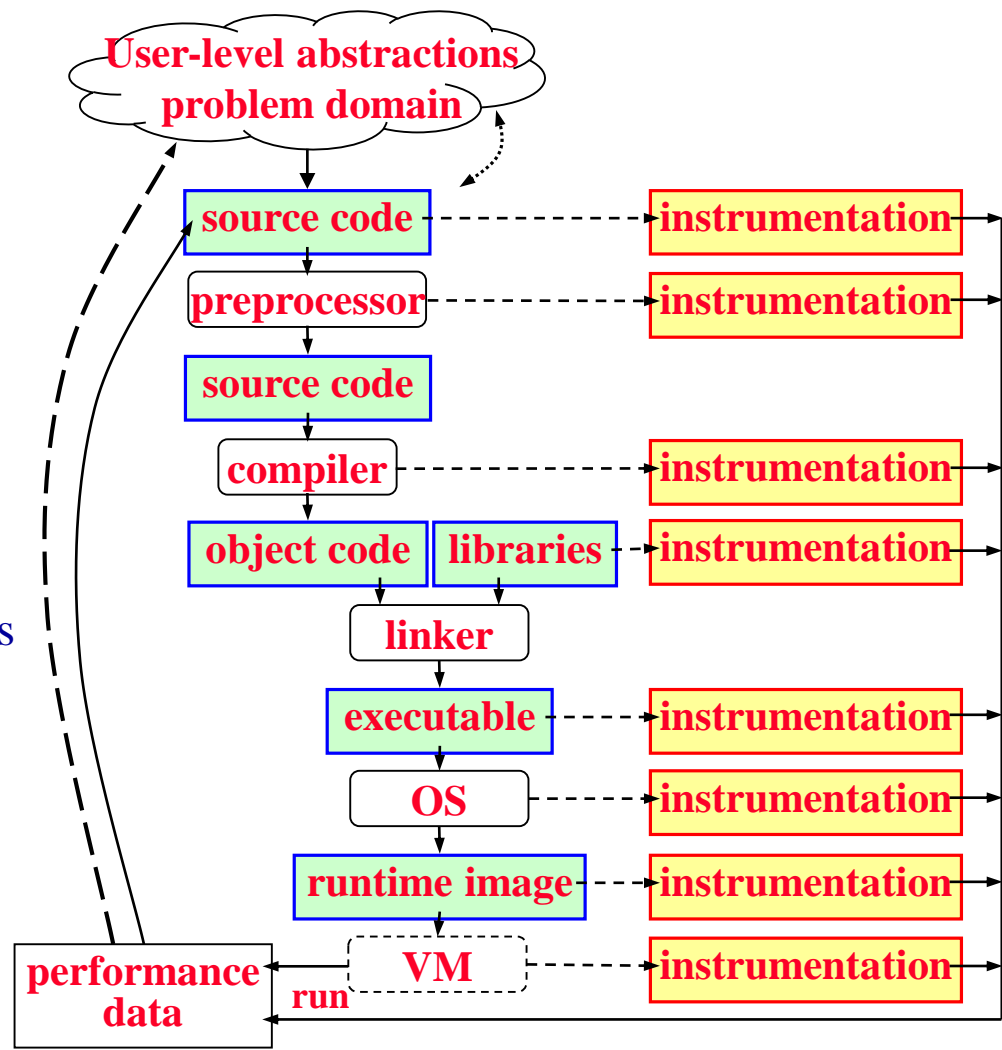


Rule 12
experim
the

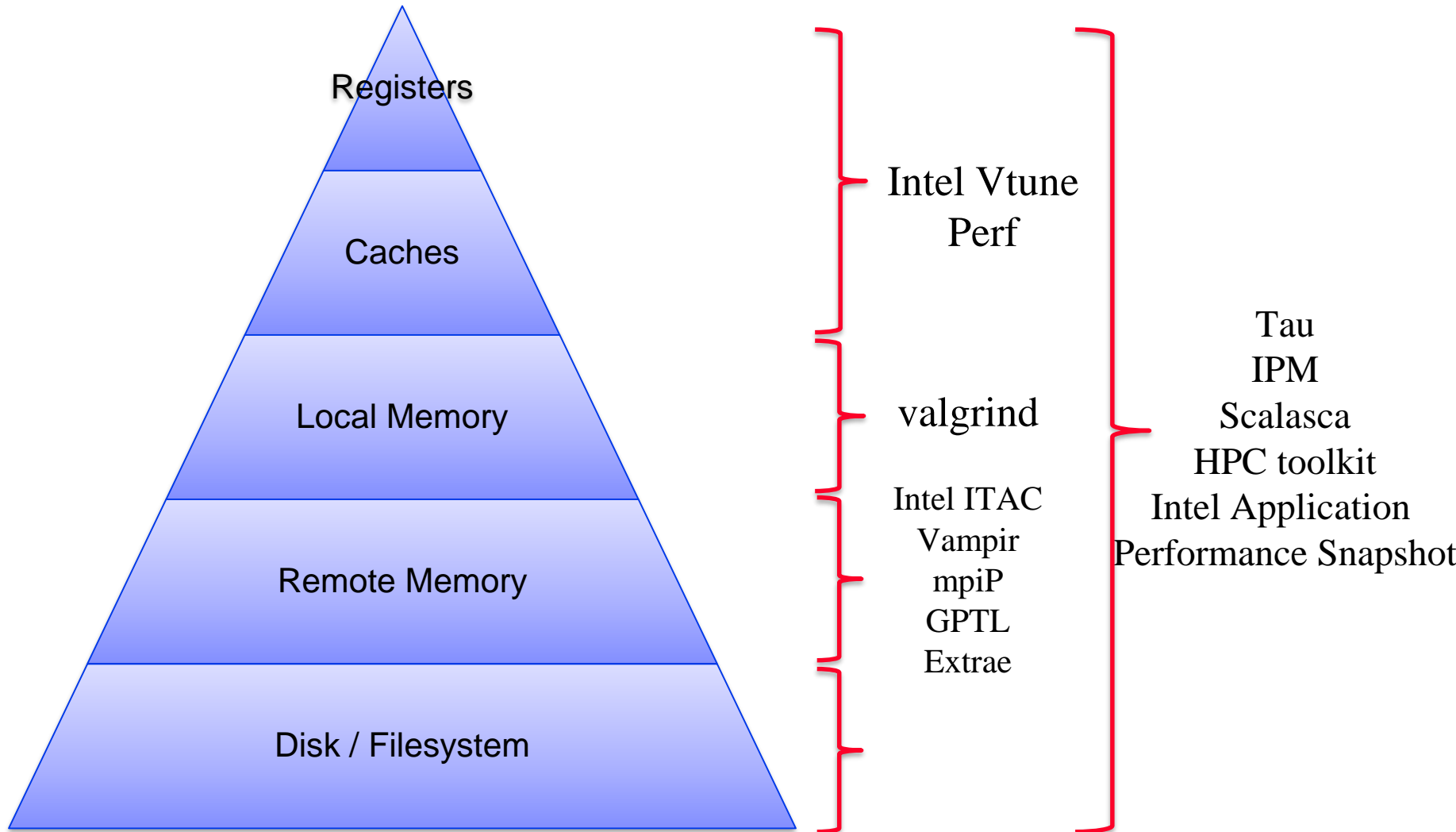
ll
zation
he
s if

HPC Performance Tools

- **Instrumenting**
 - Insert hooks into program to record and time events
- **Sampling**
 - Regularly interrupt the program and record where it is
 - Build up a statistical profile
 - Use Hardware Event Counters (PMUs)



Tools are Hierarchical



printf is the most used and easy-use tool



Intel® VTune™ Amplifier XE Performance Profiler

Where is my application...

Spending Time?

Function - Call Stack	CPU Time
algorithm_2	3.560s
do_xform ←	3.560s
algorithm_1	1.412s
BaseThreadInitTh	0.000s

- Focus tuning on functions taking time
- See call stacks
- See time on source

Wasting Time?

Line		MEM_LOAD... LLC_MISS
475	float rx, ry, rz =	
476	float param1 = (A2	30,000
477	float param2 = (A2	
478	bool neg = (rz < 0	

- See cache misses on your source
- See functions sorted by # of cache misses

Waiting Too Long?

	Wait Time	Wait Count
176.504s	Idle Poor Ok Ideal	18,277
84.681s	Idle Poor Ok Ideal	5,499
84.612s	Idle Poor Ok Ideal	5,489

- See locks by wait time
- Red/Green for CPU utilization during wait

- Windows & Linux
- Low overhead

- ✓ 热点分析
- ✓ 并行度分析

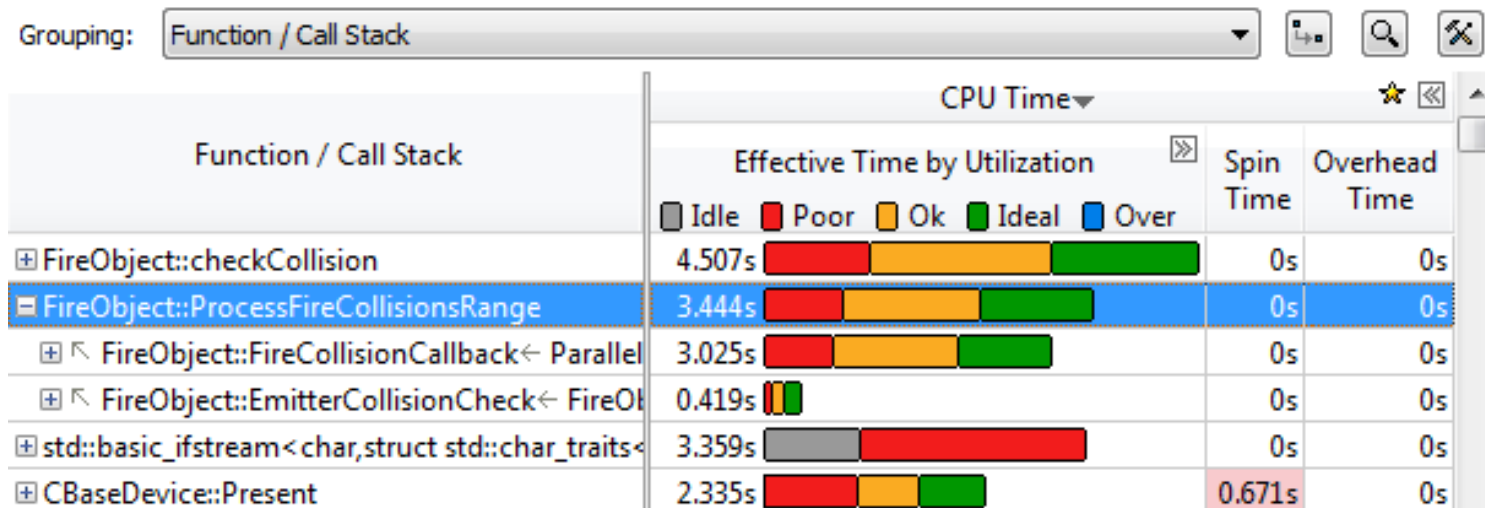
We improved the performance of the latest run 3 fold. We wouldn't have found the problem without something

<https://software.intel.com/en-us/vtune>

<https://software.intel.com/en-us/vtune/features/single-threaded>

Find Hotspots

- Identifying code that uses a lot of processor time is often the first step in single-threaded optimization. **Hotspot analysis** gives you a list of functions sorted by the amount of time they consume. Optimizing the longest running functions provide you with the biggest performance gain.



View Profiling Results on Your Source Code

- Once you find the time-consuming functions, the next step is to figure out what part of each function needs improvement. Double-clicking the hotspot list takes you directly to the source, showing the hottest spot in the function.
- Intel® VTune™ Amplifier supports most native compilers that follow industry standards, such as C, C++, and Fortran.

Source Line	Source	Effective Time by Utilization	Spin Time	Overhead Time
1,456	for(u32 j = rangeBegin; j < range	0.5%	0.0%	0.0%
1,457	{	0.0%	0.0%	0.0%
1,458	FireObject *pfo = m_pFireObj	0.4%	0.0%	0.0%
1,459	if(checkCollision(ttp, ttp,	5.4%	0.0%	0.0%
1,460	{	0.0%	0.0%	0.0%
1,461	// if it passes this test	0.0%	0.0%	0.0%

Analyze Faster with Highlighted Tuning Opportunities

- Event-based sampling uses the hardware performance monitoring unit (PMU) built into Intel® processors. PMU events can find specific tuning opportunities fast—like backend stalls or cache misses—highlighting them to facilitate easier analysis and optimization.

Function / Call Stack	Clockticks	Instructions Retired	CPI Rate	Unfilled Pipeline Slots (Stalls)	
				Back-end Bound	Front-end Bound
FireObject::checkCollision	16,639,342,359	9,528,342,771	1.746	Blue bar	Red bar
FireObject::ProcessFireCollisionsRange	7,789,603,182	6,154,123,581	1.266	Blue bar	Red bar
FireObject::FireCollisionCallback	5,696,103,361			Blue bar	Red bar
ParallelForBody::operator()← [TBB	5,692,521,630			Blue bar	Red bar
[TBB parallel_for on class ParallelF	3,581,731			Blue bar	Red bar
Selected 1 row(s):	16,639,342,359				

A significant proportion of pipeline slots are remaining empty, containing useful work to be retired per cycle than the machine operations can support. operations can cause this, as can too many operations being can support).

Threshold: (((1 - ((IDQ_UOPS_NOT_DELIVERED.CORE + L

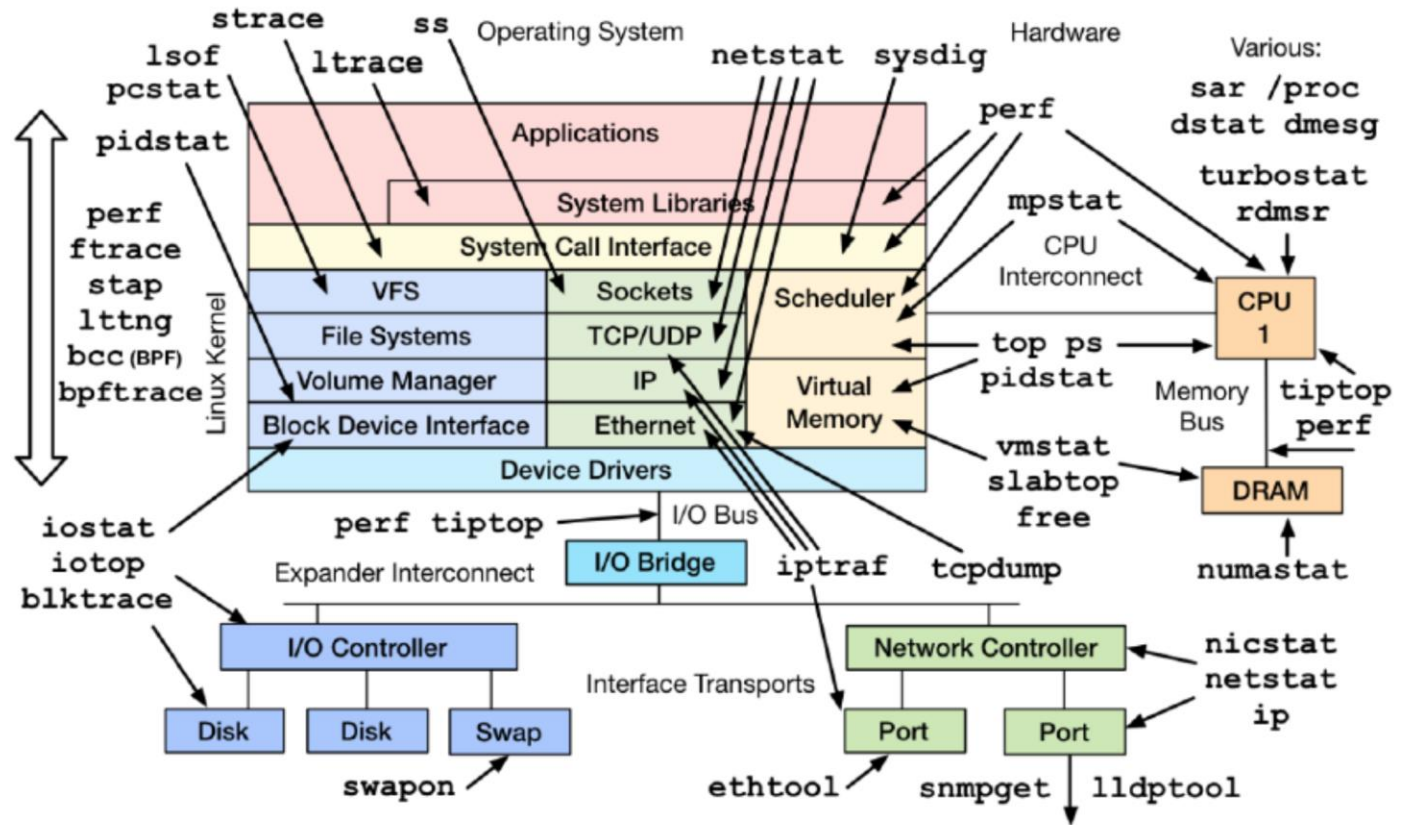
Linux Perf

Brendan D. Gregg

本页内容引自:

<http://www.brendangregg.com/>

Linux Performance Observability Tools



GPTL with PAPI

- 用户使用GPTL插数据

- GPTL负责调用PAPI
- GPTL负责调用PAPI
- 一次GPTLstart与GPTLend

- Usage of GPTL

- 手动插装代码段
 - 代码调用GPTL
- 编译器插装函数
 - GCC选项 `-finstrument-functions`
 - 函数头插装
 - `-g`与`-rdynamic`
 - 允许GPTL
 - 否则GPTL
 - `static inline`



```
PAPI event multiplexing was OFF
Description of printed events (PAPI and derived):
  GPTL_IPC: Instructions per cycle
  Instr completed
```

```
PAPI events enabled (including those required for derived events):
  PAPI_TOT_INS
  PAPI_TOT_CYC
```

```
Underlying timing routine was gettimeofday.
Per-call utr overhead est: 1.65e-06 sec.
Per-call PAPI overhead est: 3.4e-07 sec.
If overhead stats are printed, roughly half the estimated number is
embedded in the wallclock stats for each timer.
Print method was most_frequent.
If a '%_of' field is present, it is w.r.t. the first timer for thread 0.
If a 'e6_per_sec' field is present, it is in millions of PAPI counts per sec.
```

```
A '*' in column 1 below means the timer had multiple parents, though the
values printed are for all calls. Further down the listing is more detailed
information about multiple parents. Look for 'Multiple parent info'
```

```
Stats for thread 0:
```

	Called	Recurse	Wallclock	max	min	IPC	TOT_INS	e6/_sec
main	1	-	2.000	2.000	2.000	2.81e-01	18060	0.01
do_work	1	-	2.000	2.000	2.000	2.61e-01	12547	0.01
A	1	-	1.000	1.000	1.000	3.01e-01	4958	0.00
* B	2	-	2.000	1.000	1.000	1.09e-01	2812	0.00
AA	1	-	0.000	0.000	0.000	7.77e-01	488	244.00

```
Total calls = 6
Total recursive calls = 0
```

```
Multiple parent info (if any) for thread 0:
```

```
Columns are count and name for the listed child
```

```
Rows are each parent, with their common child being the last entry, which is indented
```

```
Count next to each parent is the number of times it called the child
```

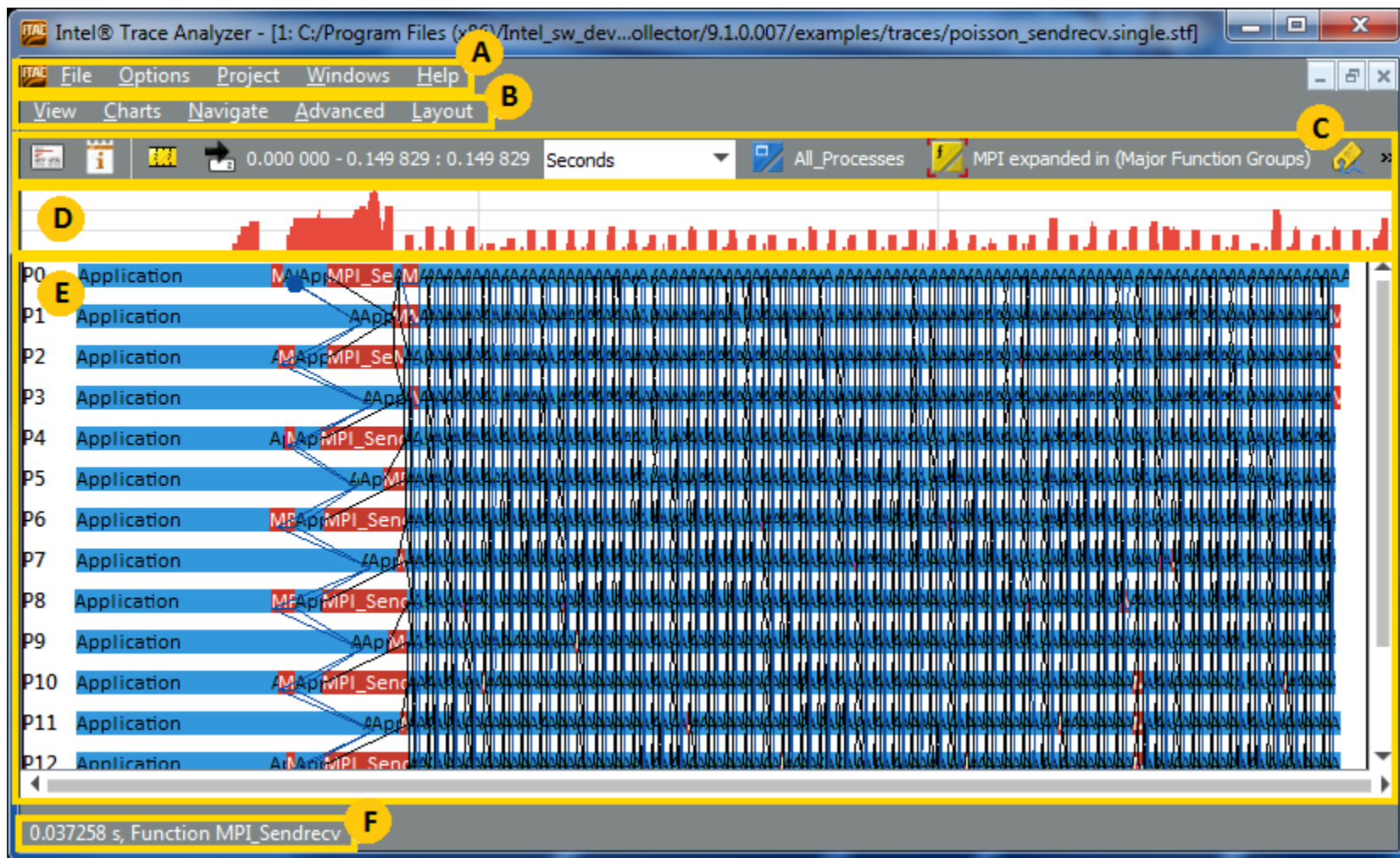
```
Count next to child is total number of times it was called by the listed parents
```

```
1 A
1 do_work
2 B
```

mpiP

- mpiP is a lightweight profiling library for MPI applications
 - **Low overhead**
 - Only collects statistical information about MPI functions
 - All the information captured by mpiP is task-local. It only uses communication during report generation, typically at the end of the experiment, to merge results from all of the tasks into one output file
 - **Scalability**
 - A variety of C/C++/Fortran applications from 2 to 262144 processes
 - **Easy to use**
 - Introduce mpiP and related libraries during Link stage
 - Work well with dispatcher

Intel Trace Analyzer and Collector



<https://software.intel.com/en-us/trace-analyzer>

<https://software.intel.com/en-us/articles/intel-parallel-studio-xe-analysis-tools-on-clusters-with-slurm-srun>

Intel Trace Analyzer and Collector

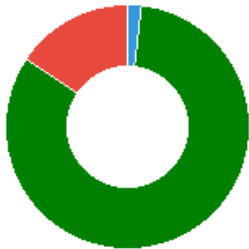
Summary: monte_carlo_openmp.single.stf

Total time: 1.41 sec. Resources: 4 processes, 1 node.

Continue >

Ratio

This section represents a ratio of all MPI calls to the rest of your code in the application.



Serial Code	0.0257 sec	1.8 %
OpenMP	1.16 sec	82.5 %
MPI calls	0.219 sec	15.6 %

Top MPI functions

This section lists the most active MPI functions from all MPI calls in the application.

MPI_Allreduce	0.218 sec (15.5 %)
MPI_Finalize	0.00138 sec (0.098 %)
MPI_Comm_size	3e-06 sec (0.000213 %)
MPI_Comm_rank	0 sec (0 %)

Where to start with analysis

For deep analysis of the MPI-bound application click "Continue >" to open the tracefile View and leverage the **Intel® Trace Analyzer** functionality:

- *Performance Assistant* - to identify possible performance problems
- *Imbalance Diagram* - for detailed imbalance overview
- *Tagging/Filtering* - for thorough customizable analysis

To optimize node-level performance use:

Intel® VTune™ Amplifier XE for:

- algorithmic level tuning with hotspots and threading efficiency analysis
- microarchitecture level tuning with general exploration and bandwidth analysis

Intel® Advisor XE for:

- vectorization optimization and thread prototyping.

Use the following command lines to run these tools for the most CPU-bound rank.

Intel® VTune™ Amplifier XE:

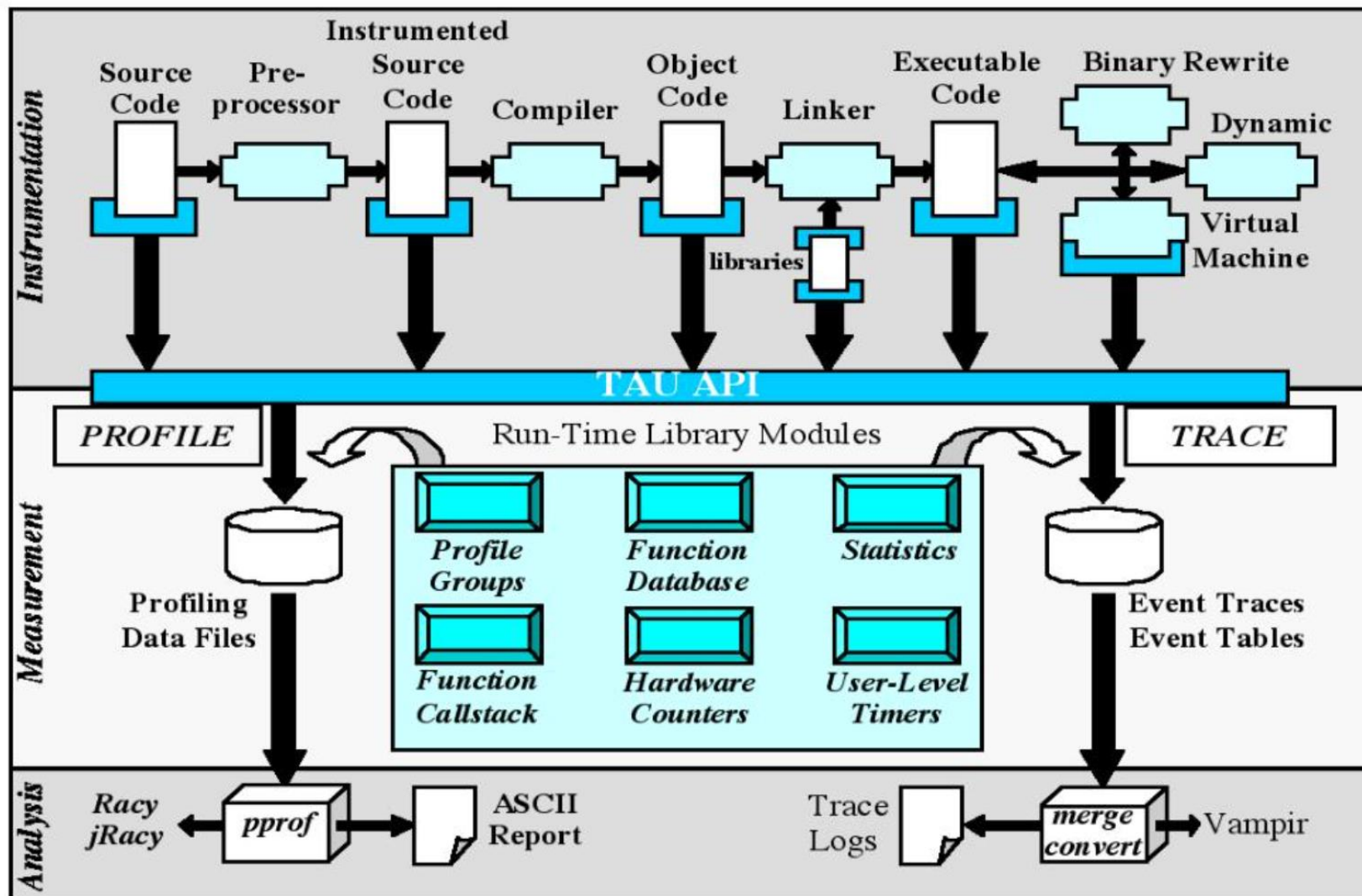
```
mpiexec.hydra -gtool "amplxe-cl -collect hotspots -r result:3" -n 4  
./monte_carlo_openmp
```

Intel® Advisor XE:

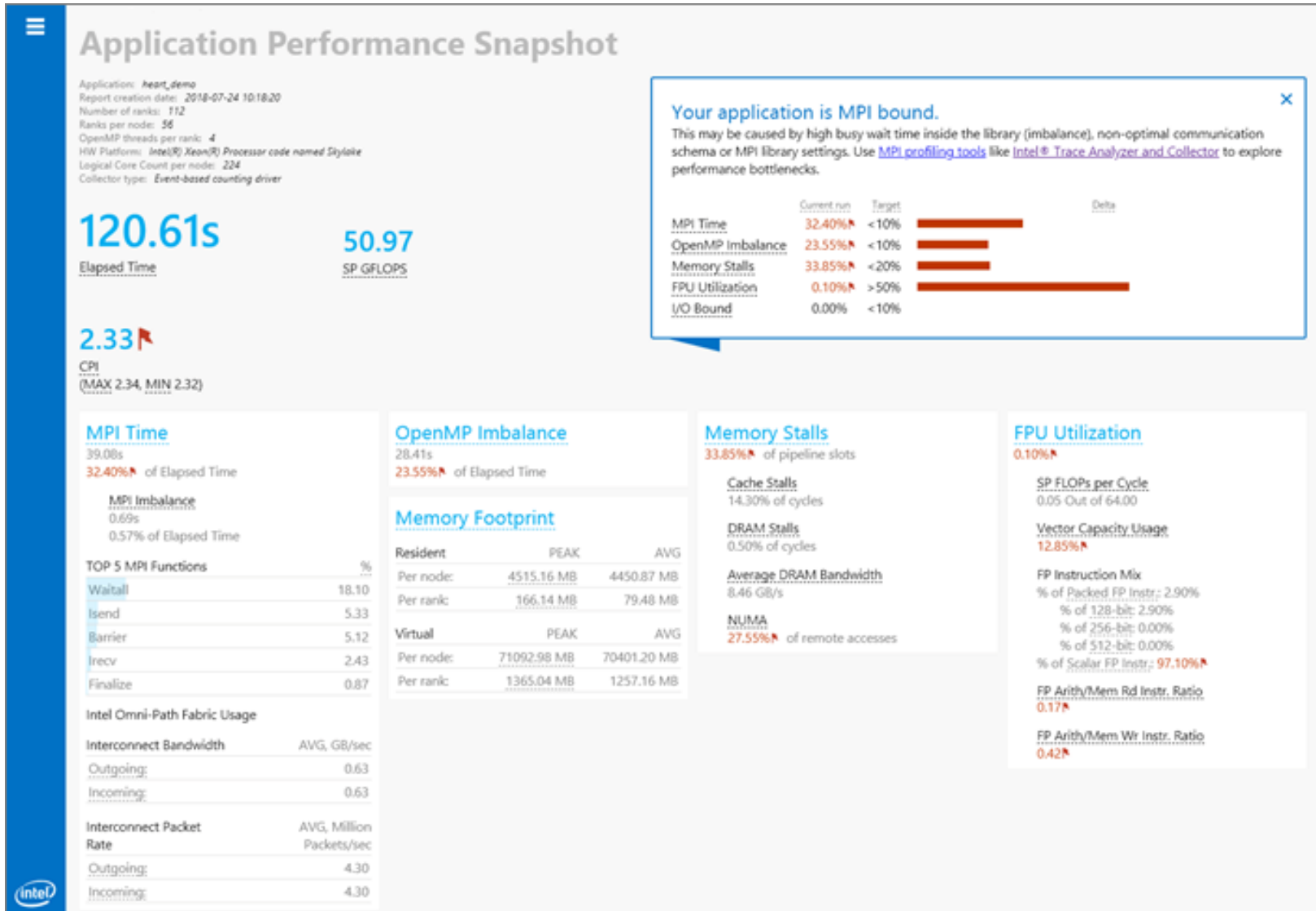
```
mpiexec.hydra -gtool "advixe-cl -collect survey:3" -n 4  
./monte_carlo_openmp
```

Show Summary Page when opening a tracefile

TAU Performance System Architecture



Intel Application Performance Snapshot



◆ Outline

1. Motivation
 2. Performance Analysis of Parallel Programs
 3. **Performance Modeling of Parallel Programs**
-

◆ Simple definition

- Performance modelling is the process of simulating **various user and system loads** against **varying system configurations** by using **a mathematical approximation** of how the model works. This is typically **much cheaper** than performance testing and can produce **very accurate results**.

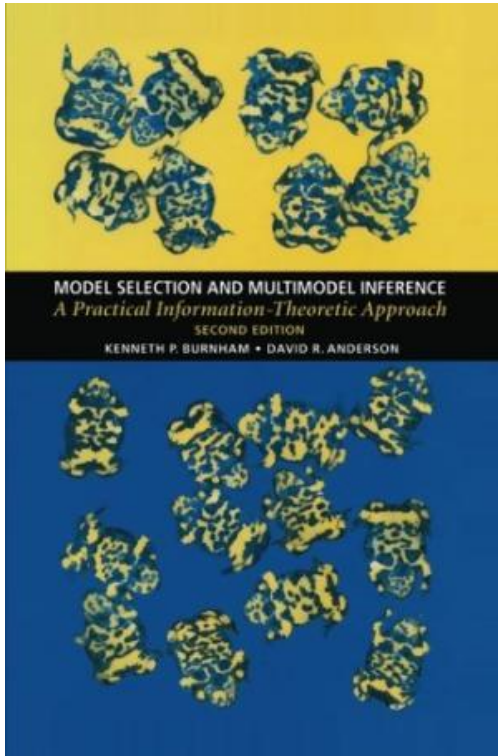
<http://www.testingperformance.org/definitions/what-is-performance-modelling>

◆ Why we need performance modeling?

- Evaluate effectiveness of a computing platform to solve a particular problem
 - Predict resources and costs to solve a particular problem instance
 - Runtime prediction for different input-size
 - Predict time when scaling a problem
 - Help identify bottlenecks for hardware
- Makes programmers **think** about the **structured performance profile** of an application or platform
 - Find performance bugs and assess upper and lower bounds of code optimizations
 - Scalability bug detection
- Build a **surrogate model** in **automated performance tuning**
 - Performance modeling as part of a software engineering discipline in HPC

Performance Modeling is Performance analysis v2.0

◆ Limitations



Burnham, Anderson: “A model is a **simplification or approximation of reality** and hence will not reflect all of reality. ... Box noted that “all models are wrong, but some are useful.” While a model can never be “truth,” a **model might be ranked from very useful, to useful, to somewhat useful to, finally, essentially useless.**”

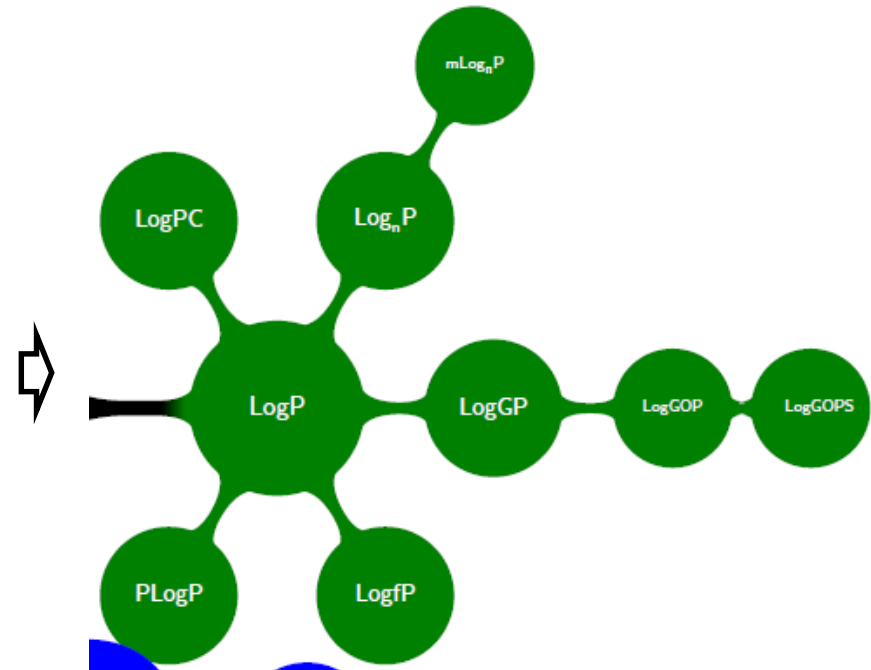
◆ Classification of performance model

- Capability Model
 - Roofline model (Prof. Zhang chensong)
 - Parallel performance model (communication side)
 - Architecture-oriented model
 - Application performance model
 - Analytical Model
 - Empirical model
 - Automated performance modelling
-

◆ Parallel Performance Model

Latency/Bandwidth model for Parallelism

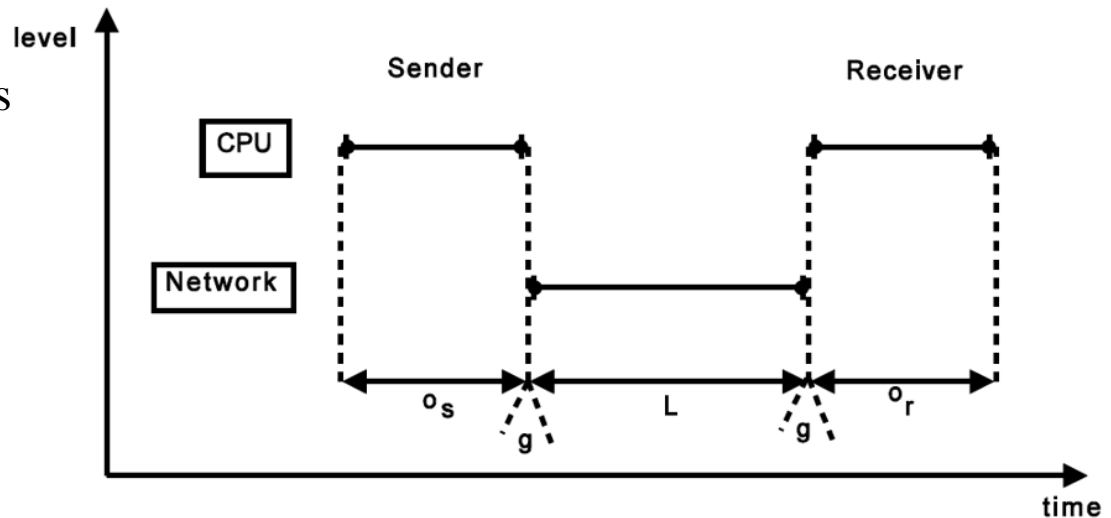
- Transfer time $T(s) = a + bs$
 - a = startup time (latency)
 - b = cost per byte
(bandwidth = $1/b$)
- As s increases, bandwidth approaches $1/b$
- Often assuming no pipeline (new messages can only be issued from a process after all arrived)



◆ Parallel Performance Model

LogP Model to model pipeline, computation/communication overlap and endpoint congestion/overhead

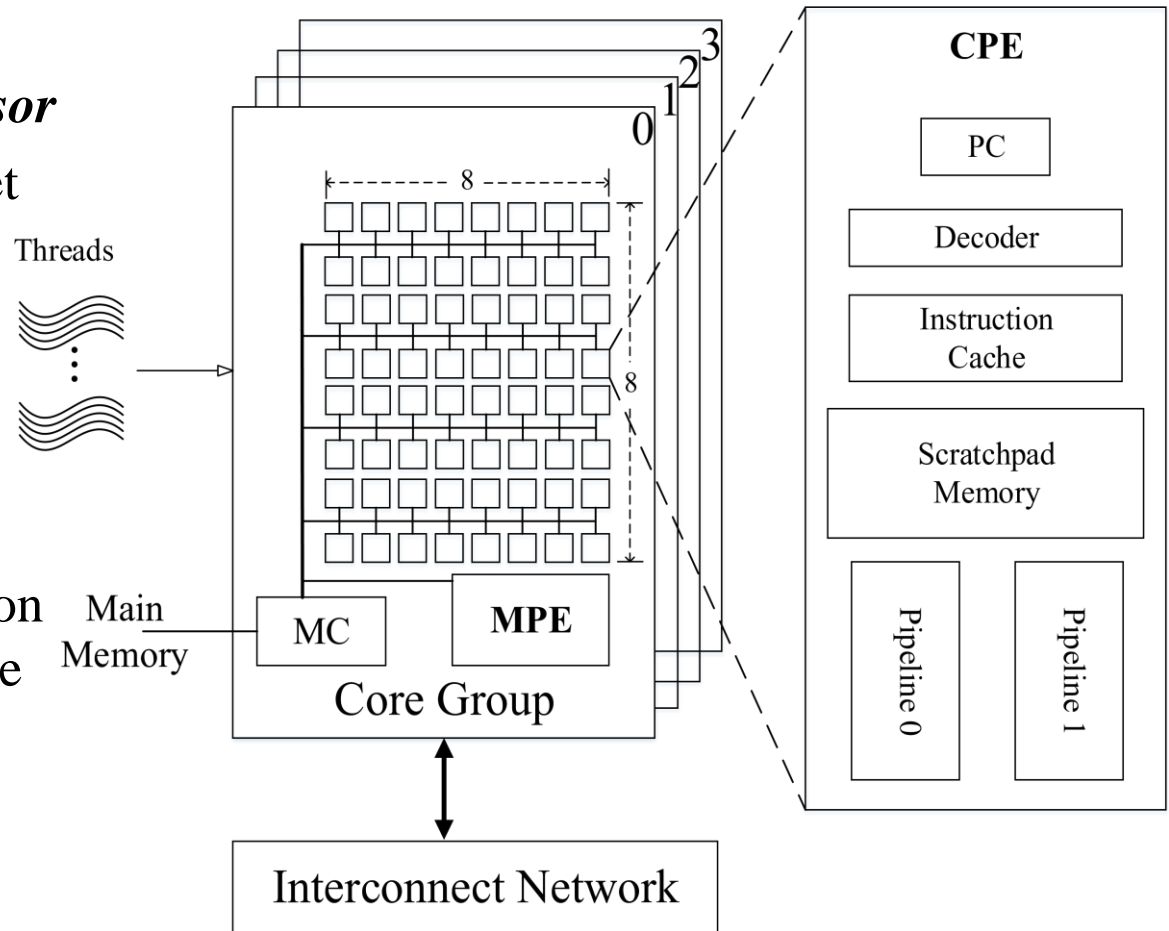
- Four parameters of model
 - L: an upper bound on the latency or delay
 - o: the overhead defined as the length of time that a processor is engaged in the transmission or reception of each message
 - g: the gap defined as the minimum time interval between consecutive message transmissions or consecutive message receptions at a processor ($1/b$ in L/B model)
 - P: the number of processors/memory modules
- Transmitting n messages
 $T(n) = L + (n-1) * \max(g, o) + 2o$
- Concerns
 - Bulk message? -> LogGP
 - Complex to use



◆ Architecture-oriented model

SW26010 Many-core Processor

- 4 core groups (CG) in socket and each CG consists of a MPE and 64 CPEs
- Each CPE has 64KB LDM (local data mem.)
- Shared memory for socket
- 128b Register communication among same column or same row



◆ Architecture-oriented model

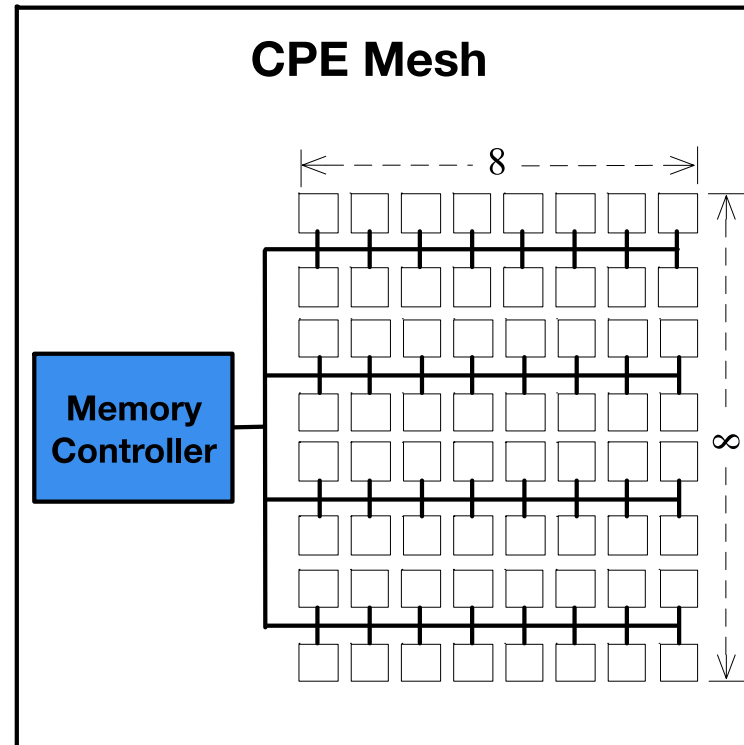
1. Memory access transaction analysis:



- Two Kinds Memory Requests: DMA, Gload
 - DMA Requests' MR from 1Byte to LDM
 - Gload Requests' MR from 1Byte to 32Bytes
 - Gload Requests and small granularity DMA access waste more Bandwidth

Architecture-oriented model

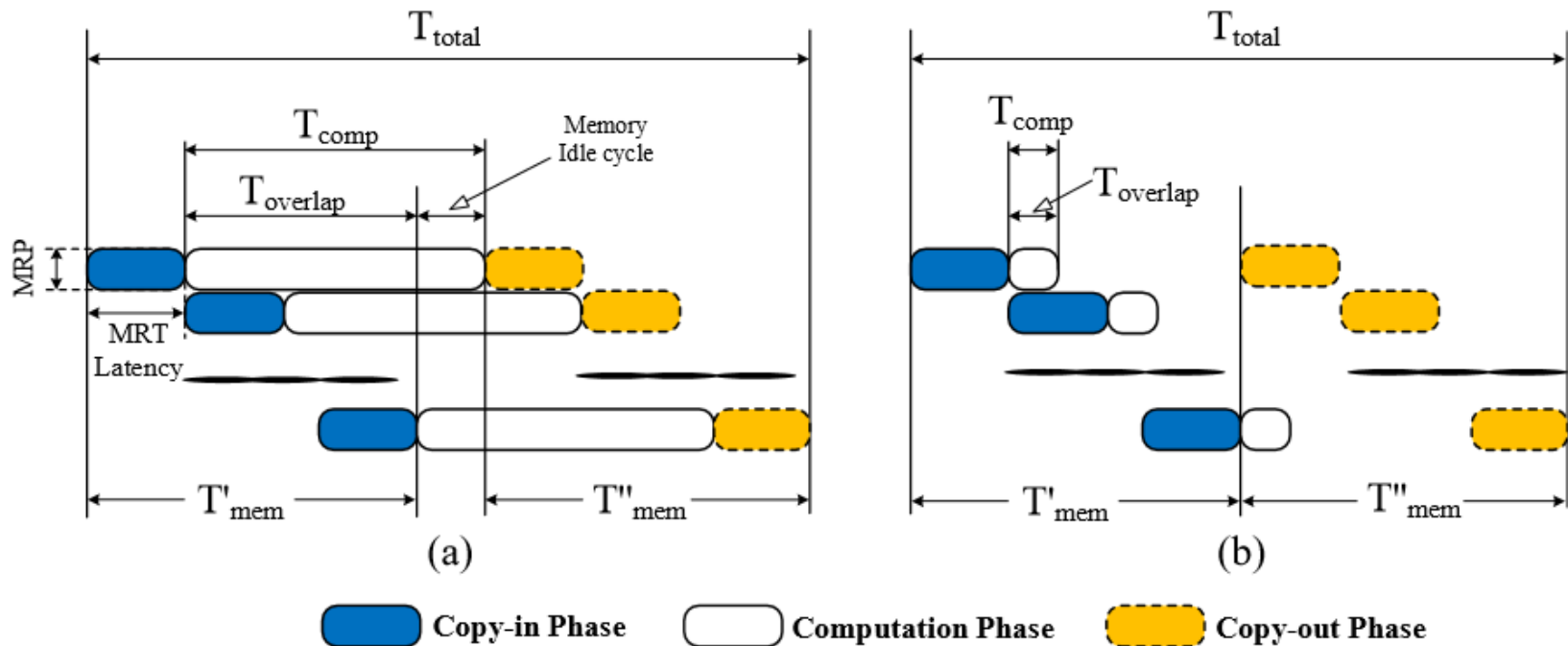
CPEs	256B	4096B
1	1.3	8.1
2	2.3	13.7
4	4.5	21.7
8	8.8	21.7
16	16.2	21.7
32	25.5	26.7
64	27.2	27.1



- 16 CPEs are connected to a memory bus
- Four buses are merged into one bus and to the MC
- Designed BW is 32GB/s, designed sub bus BW is 25.6GB/s
- Default layout use one sub bus until all the CPEs on this bus is active

◆ Architecture-oriented model

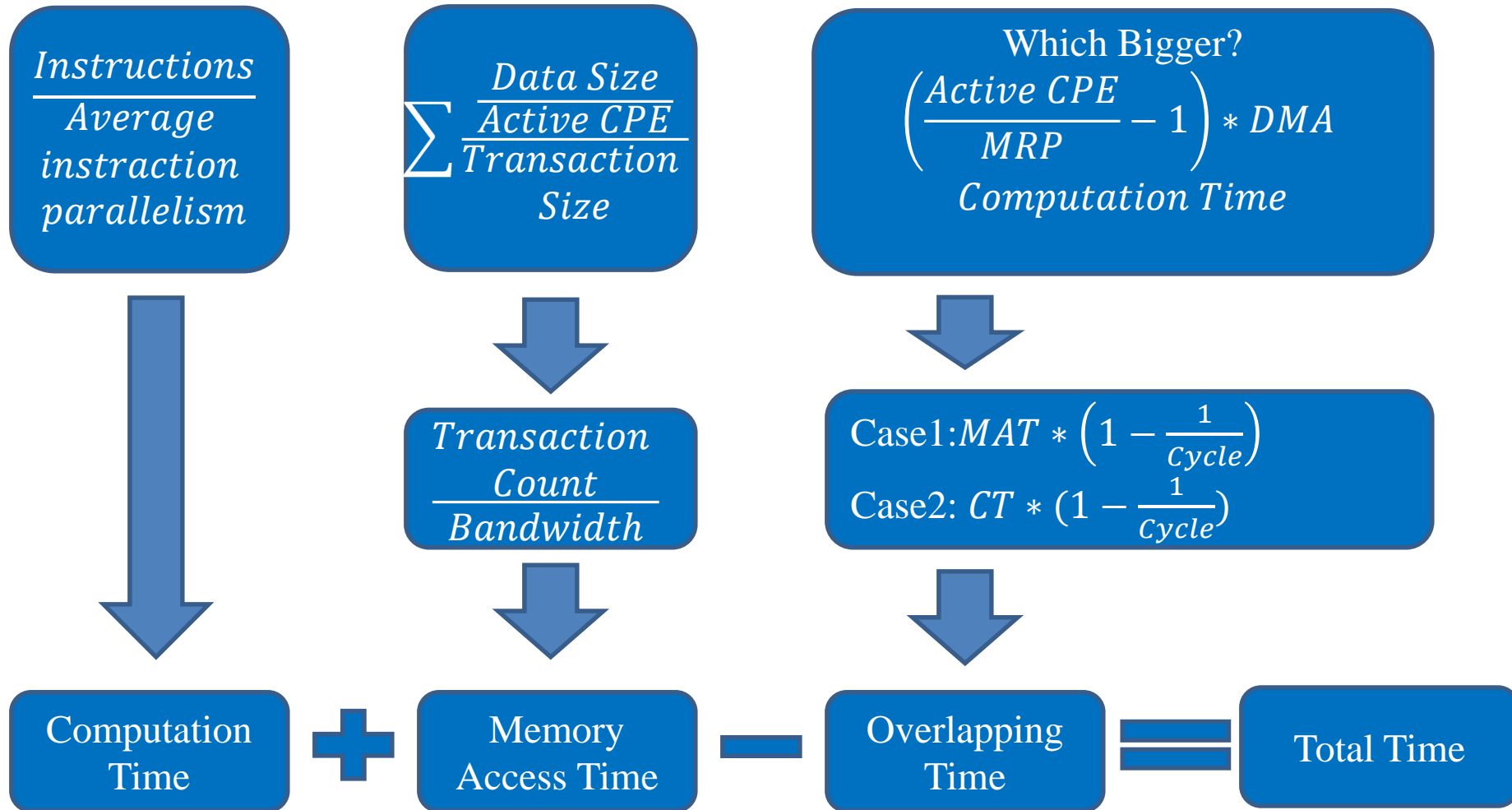
2. Overlap analysis



MRT Latency refers to memory access time for CPE

MRP is usually 4.

◆ Architecture-oriented model



◆ Architecture-oriented model

Intel Xeon Phi KNL

NVIDIA GPGPU

Intel x86-64

2015 IEEE International Symposium on Workload Characterization

Quantifying the Performance Impact of Memory Latency and Bandwidth for Big Data Workloads

Russell Clapp
Intel Corporation
Hillsboro, Oregon

Martin Dimitrov
Intel Corporation
Chandler, Arizona

Karthik Kumar
Intel Corporation
Chandler, Arizona

Vish Viswanathan
Intel Corporation
Hillsboro, Oregon

Thomas Willhalm
Intel GmbH
Walldorf, Germany

{Russell.M.Clapp, Martin.P.Dimitrov, Karthik.Kumar, Vish.Viswanathan, Thomas.Willhalm}@intel.com

Abstract— In recent years, DRAM technology improvements have scaled at a much slower pace than processors. While server processor core counts grow from 33% to 50% on a yearly cadence, DDR 3/4 memory channel bandwidth has grown at a slower rate, and memory latency has remained relatively flat for some time. Combined with new computing paradigms such as big data analytics, which involves analyzing massive volumes of data in real time, there is a trend of increasing pressure on the memory

can be determined using performance counter data from real systems. Further, we use these components and the measured data to classify about a dozen workloads based on their inherent bandwidth demand and latency sensitivity, including big data workloads that utilize both structured and unstructured data. This classification enables us to create synthetic component values for the performance equations for each workload cluster.

2017年12月12日
星期二

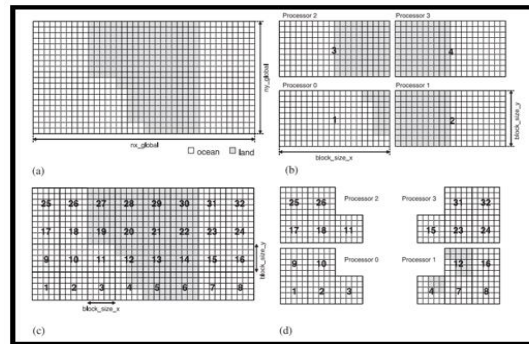
◆ Classification of performance model

- Capability Model
 - Roofline model (Prof. Zhang chensong)
 - Parallel performance model (communication side)
 - Architecture-oriented model
 - Application performance model
 - Analytical Model
 - Empirical model
 - Automated performance modelling
-

◆ Application Performance Model

分析模型

- 分析程序算法结构
- 分析程序实现方案
- HPCA'16 LRU cache
- COMMU'14 Cache-aware Roofline model
- CCGrid'12 su3_rmd
- SC'12 Aspen
- JHPCA'10 Sweep3D



$$T_{POP}(\mathbf{P}, \mathbf{N}, G) = T_{baroclinic}(\mathbf{P}, \mathbf{N}, G) + T_{barotropic}(\mathbf{P}, \mathbf{N}, G)$$

$$T_{barotropic}(\mathbf{P}, \mathbf{N}, G) = T_{barotropic_comp}(\mathbf{P}, \mathbf{N}, G) + N_{bound_tropic} \cdot T_{bound_ex}(\mathbf{P}, \mathbf{N}, G) + N_{global_sums} \cdot T_{global_red}(\mathbf{P})$$

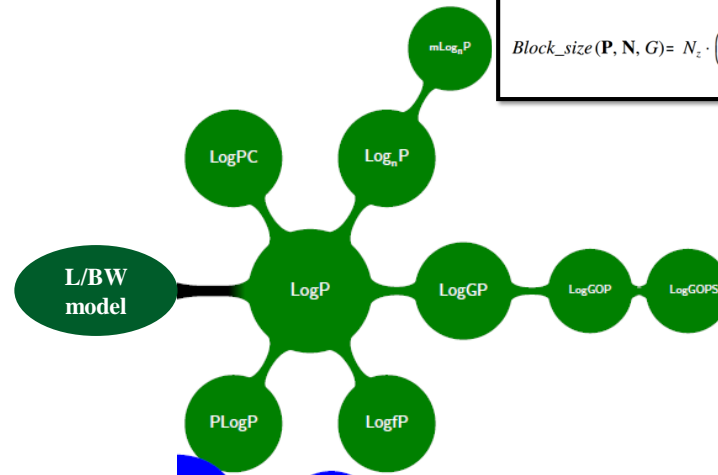
$$T_{bound_ex}(\mathbf{P}, \mathbf{N}, G) = T_{comm}(8 \cdot N_x \cdot G, P_x \cdot P_y, C_x) + T_{comm}(8 \cdot N_y \cdot (G + 1), P_x \cdot P_y, C_x)$$

$$N_{bound_clitic} = ((nsteps - 1) \times 2 \times N_x) \text{ and } N_{bound_tropic} = \left(nsteps \times \left(4 + Av_scans \times \left(1 + \frac{1}{ncheck} \right) \right) \right)$$

$$Block_size(\mathbf{P}, \mathbf{N}, G) = N_z \cdot \left(\frac{N_x + 2 \cdot G}{P_x} \right) \cdot \left(\frac{N_y + 2 \cdot G}{P_y} \right)$$

示例：并行海洋程序POP

- 第一步：计算单元到进程的映射关系
- 第二步：一个计算单元中，关键应用变量到时间的映射关系
- 第三步：设计实验、量测性能、拟合模型参数



◆ Application Performance Model

经验模型

- 回归拟合运行时间
- 机器学习
- SC'17 Transfer learning
- IPDPS'16 NPB
- SC'15 Extra-P
- PACT'14 NPB HPCCG
- SC'13 Sweep3D, HOMME
- ICS'08 NPB

可自动化

基函数的选取问题
可解释性问题

Ranking:

- Target scale p_t

1. foo
2. compute
3. main

$$c_1 \times \log(p) + c_2 \times p$$

$$c_1 \times \log(p) + c_2 \times p \times \log(p)$$

$$c_1 \times \log(p) + c_2 \times p^2$$

$$c_1 \times \log(p) + c_2 \times p^2 \times \log(p)$$

$$c_1 \times p + c_2 \times p \times \log(p)$$

$$c_1 \times p + c_2 \times p^2$$

$$c_1 + c_2 \times \log(p)$$

$$c_1 \times p + c_2 \times p^2 \times \log(p)$$

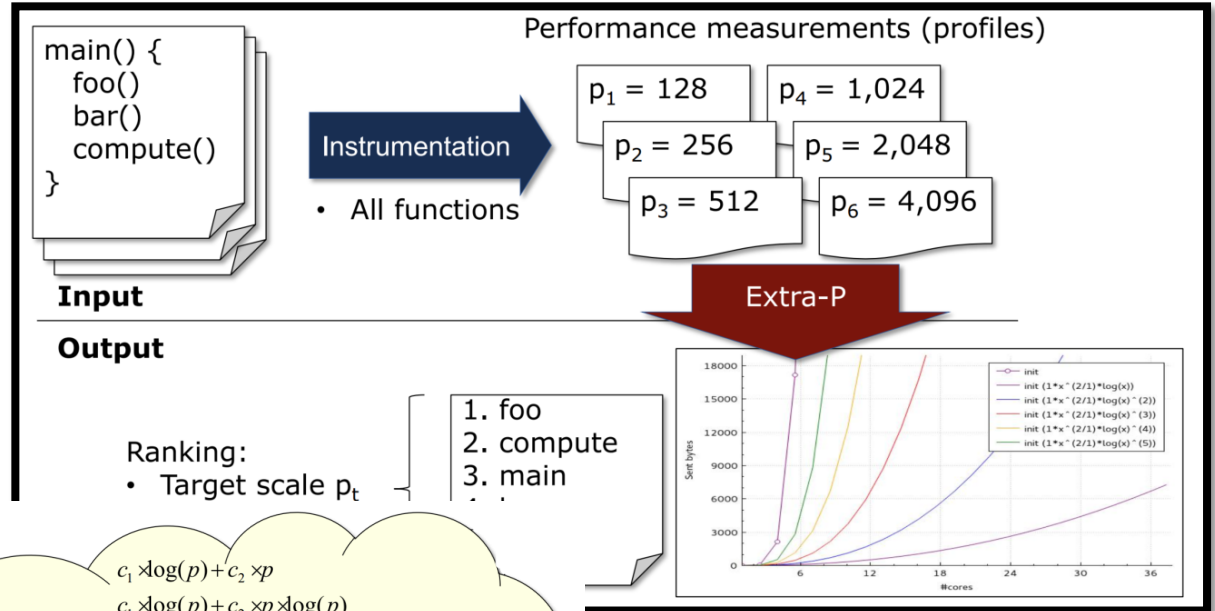
$$c_1 + c_2 \times p \times \log(p)$$

$$c_1 \times p \times \log(p) + c_2 \times p^2$$

$$c_1 \times p \times \log(p) + c_2 \times p^2 \times \log(p)$$

$$c_1 \times p^2 + c_2 \times p^2 \times \log(p)$$

Extra-P



$$f(p) = \sum_{k=1}^n c_k \times p^{i_k} \times \log_2^{j_k}(p)$$

Application Performance Model

可移植、通用化

指导原因分析

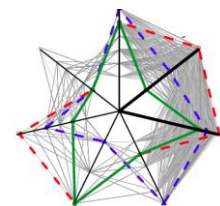
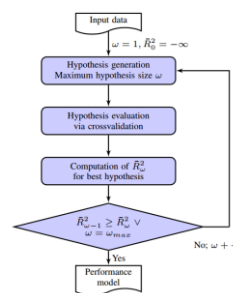
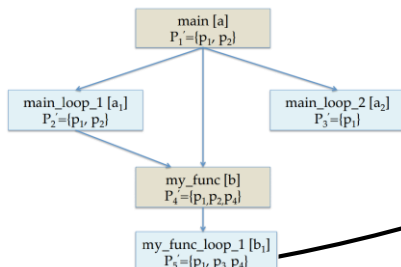
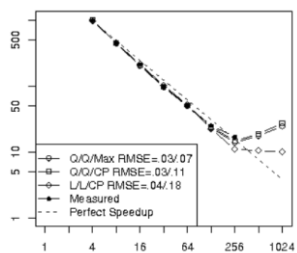
应用程序关键因素

与计算平台交互

分析模型

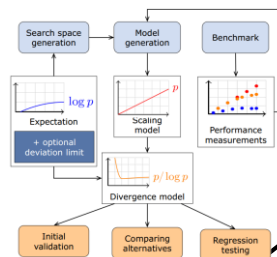


经验模型



SC'17 迁移学习法

CLUSTER'16 多参数快速建模



SC'15 自动化建模工具，在预定义基函数中回归拟合

PACT'14 借助编译器技术探测程序结构与基本程序块的输入+在线采集，细粒度的建模

资源导向性能建模

从top-down到bottom-up的自动性能建模

目标：寻找一种建模方式，可以同时提供应用的特征、应用与计算平台交互特征、最大化降低建模成本、可跨平台使用

硬件计数器技术

普遍支持



IBM Power 3 处理器配置了8个硬件计数器



英特尔P4系列处理器18个硬件计数器



神威架构的处理器提供了7个硬件计数器

广泛使用

- Intel Vtune
- HPCToolkit
- TAU
- LIKWID
- PAPI
- Scalasca
- Perf
- Score-P

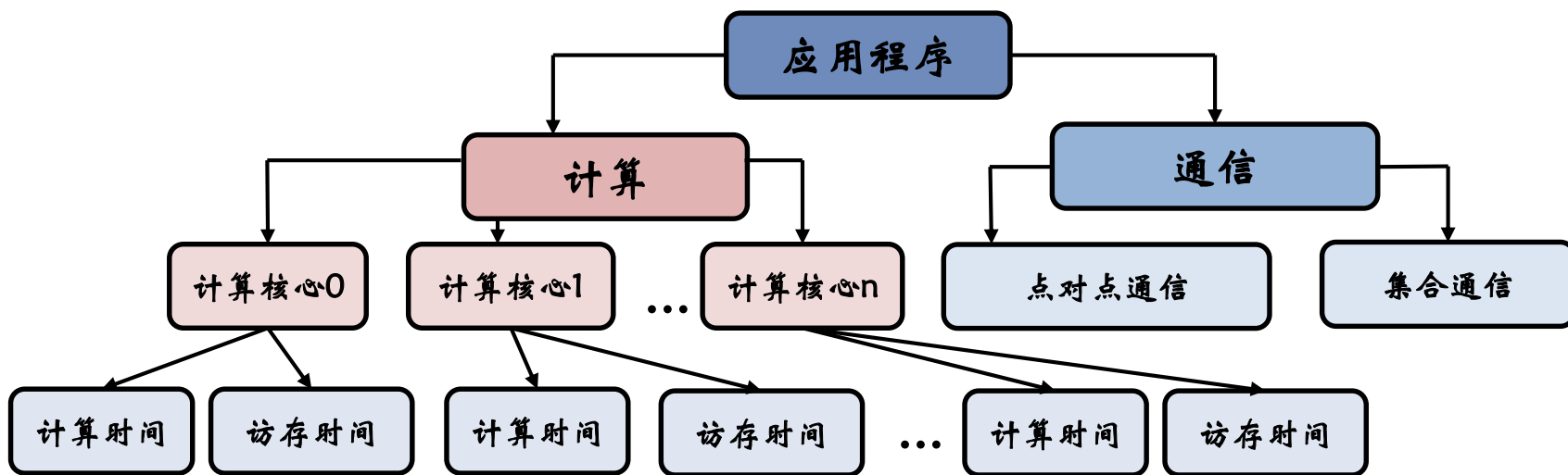
轻量采集

- 运行时间类：时钟周期数
- 程序指令类：执行指令数等
- 存储事件类：各级缓存的访问计数、不命中数目等
- 流水线类：指令流水线空周期等
- 分支预测类：错误分支预测数等

资源导向性能建模

自动化资源导向性能模型

$$T_{app} = T_{comp} + BF_{comm} * T_{comm} + T_{others}$$



$$T_{comp} = \sum_{k=1}^K (T_{comp-k} + T_{nonoverlap-k})$$
$$T_{comp} = \sum_{k=1}^K \left(\frac{\#inst_k * CPI_k}{F * P} + BF_{mem_k} * T_{mem_k} \right)$$



关键路径分析

https://safari.ethz.ch/architecture/fall2017/lib/exe/fetch.php?media=quantifying_the_performance_impact_of_memory_latency_and_bandwidth_for_big_data_workloads.pdf

资源导向性能建模

自动化资源导向性能模型

硬件计数器事件

Intel(R) Xeon(R) CPU E5-2680 v3	
#inst	INSTRETIRED.ANY_PS
#uops	MEM_LOAD_UOPS_RETIRED.L1_HIT_PS
	MEM_LOAD_UOPS_RETIRED.L2_HIT_PS
	MEM_LOAD_UOPS_RETIRED.L3_HIT_PS
	MEM_UOPS_RETIRED.ALL_LOADS_PS
	MEM_UOPS_RETIRED.ALL_STORES_PS
#stall	RESOURCE_STALLS.LB RESOURCE_STALLS.ST

SW26010		
CPE	#inst	penv_slave0_inst_count penv_slave_ipc
	#uops	penv_slave2_gld_count penv_slave2_dma_count
		#stall
	MPE	#inst
#uops		penv_memory_access_count penv_dcache_miss_ratio_count penv_scache_miss_count penv_scache_access_count

Type	Intel Skylakex(x86)	Huawei Kunpeng920(ARMv8.2)
运行时间类	CPU_CLK_UNHALTED.THREAD_P	CPU_CYCLES
程序指令类	INST_RETIRED.ANY_P	INST_RETIRED
流水线类	RESOURCE_STALLS.ANY	MEM_STALL_ANYLOAD
	RESOURCE_STALLS.SB	MEM_STALL_ANYSTORE
	RESOURCE_STALLS.SB	MEM_STALL_ANYSTORE
	CYCLE_ACTIVITY.STALLS_L1D_MISS	MEM_STALL_L1MISS
	CYCLE_ACTIVITY.STALLS_L2_MISS	MEM_STALL_L2MISS

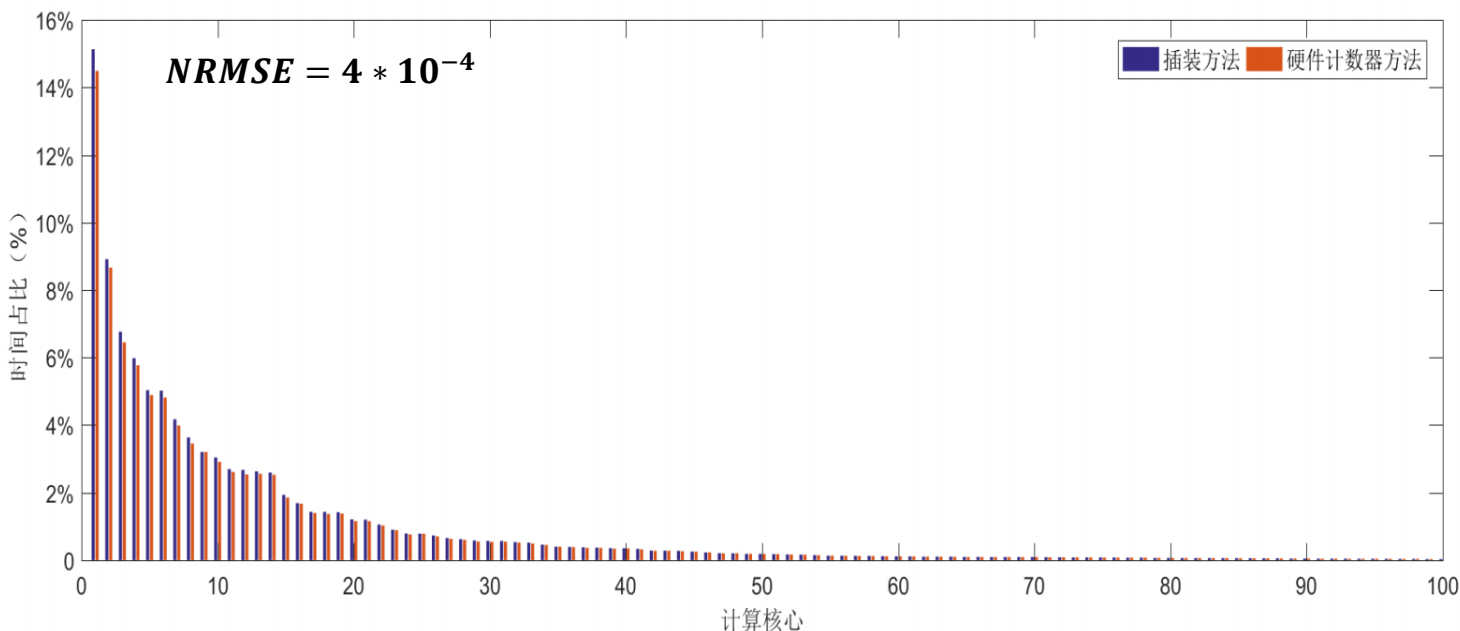
表 1: 计算核心的性能采集

资源导向性能建模

自动化资源导向性能模型

采样机制是否可以开展性能建模？

$T_{app} > 500ms$ 硬件计数器采样结果具有较高的准确度



x86平台，CICE为例

$$\text{归一化均方根误差: } NRMSE = \frac{\sqrt{\sum_{i=1}^K t_i \times (t_i - r_i)^2}}{\max_{1 \leq i < K} (t_i \cup r_i) - \min_{1 \leq i < K} (t_i \cup r_i)} < = 10^{-3}$$

资源导向性能建模

自动化资源导向性能模型

如何选取关键计算核心?

类1: 时间占比大的函数

类2: 时间占比不减小

类3: 类1U类2

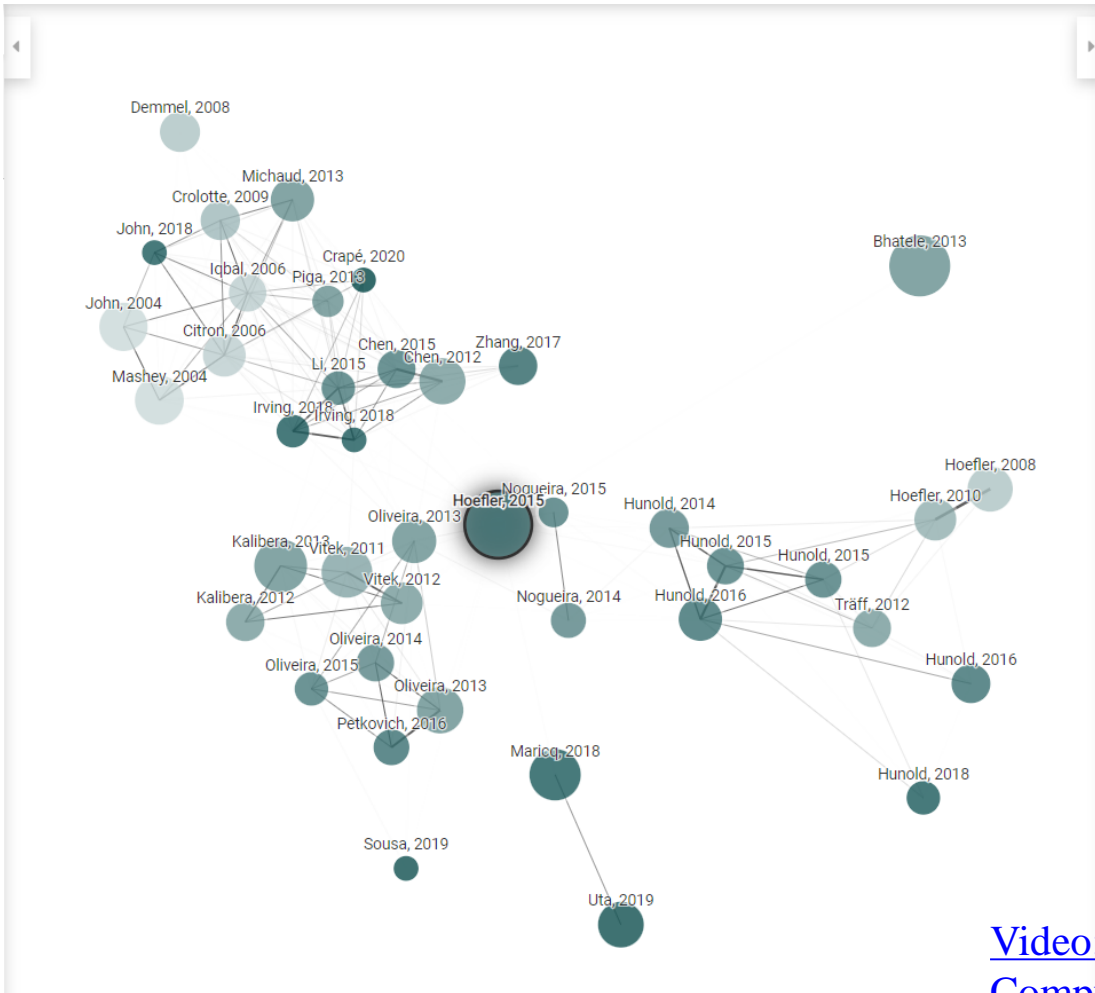
模型准确度一致情况下，资源导向模型与更细粒度（循环级）建模方法相比，可以降低建模开销。

	Func	ker	nonk	LOCs	Lker	size
CICE	109	7	2	75,000	-	116 · 100 384 · 320
HOMME	210	11	6	113,095	-	32 · 32 · 6 · 128 256 · 256 · 6 · 128
EP	5	3	0	359	12	2^{28} 2^{30}
MG	16	5	1	2,568	98	256 · 256 · 256 · 4 512 · 512 · 512 · 20
FT	10	5	2	2,034	39	256 · 256 · 128 · 6 512 · 256 · 256 · 20
SP	15	5	1	4,902	229	64 · 64 · 64 · 400 102 · 102 · 102 · 400
LU	11	6	2	5,957	165	64 · 64 · 64 · 250 102 · 102 · 102 · 250
BT	14	8	2	9,162	211	64 · 64 · 64 · 200 102 · 102 · 102 · 200
CG	3	6	1	1,901	30	14,000 · 15 75,000 · 75

◆ Summary

- Performance analysis is critical to efficient and effective computing (know knows and unknowns, unknown unknowns)
 - Complexities of computing systems and parallel programs
 - Performance interference makes things harder
 - Fundamental of performance engineering
 - Focus on experiment design, performance data collection, and analysis methods
 - Performance modeling is performance analysis v2.0
 - Increasingly important
 - Necessary to algorithm/program design, performance portability and hardware design
 - Hybrid analytical and empirical models for parallel applications based on capability models
-

◆ Recommended paper




Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results

T. Hoefler, Roberto Belli

2015, SC15: International Conference for High Performance Computing, Networking, Storage and Analysis

184 Citations Save

Open in: 

Measuring and reporting performance of parallel computers constitutes the basis for scientific advancement of high-performance computing (HPC). Most scientific reports show performance improvements of new techniques and are thus obliged to ensure reproducibility or at least interpretability. Our investigation of a stratified sample of 120 papers across three top conferences in the field shows that the state of the practice is lacking. For example, it is often unclear if reported improvements are deterministic or observed by chance. In addition to distilling best practices from existing work, we propose statistically sound analysis and reporting techniques and simple guidelines for experimental design in parallel computing and codify them in a portable benchmarking library. We aim to improve the standards of reporting research results and initiate a discussion in the HPC field. A wide adoption of our minimal set of rules will lead to better interpretability of performance results and improve the scientific culture in HPC.

[Video: Scientific Benchmarking of Parallel Computing Systems - insideHPC](#)

Thanks!

薛巍

清华大学计算机系

xuewei@tsinghua.edu.cn