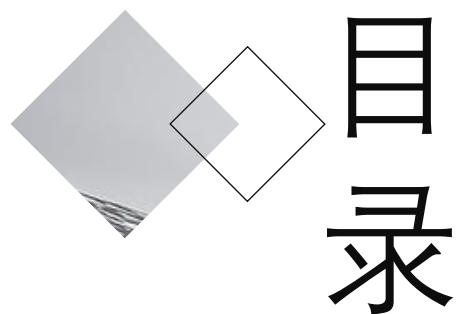


IPCC

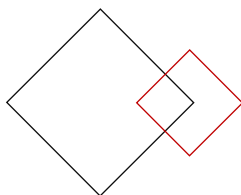
第二讲 - 性能优化方法论

时间：2021年5月

主讲人：赵雄君



CONTENTS



PART 1

性能的度量指标

- | Amdahl定律
- | time, cpi, Floas
- | CPU利用率, Cache
- | 文件IO, network

PART 2

性能分析实用工具

- | Linux工具
- | perf, gprof, valgrind
- | Paramon和Paratune

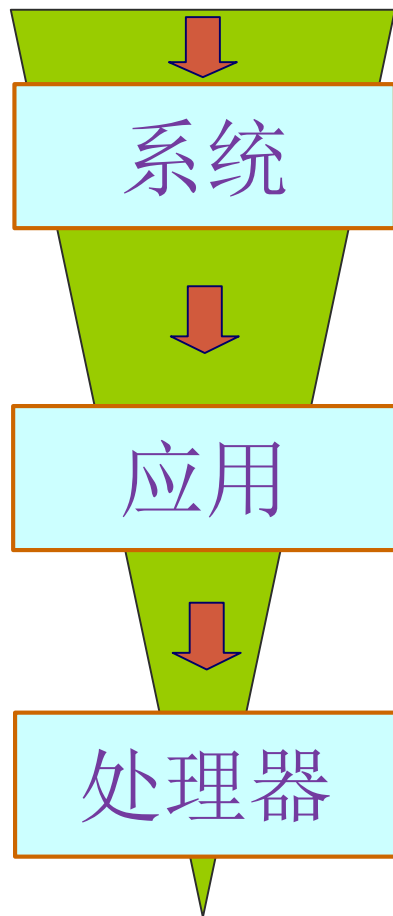
PART 3

案例分析

- | 矩阵乘法
- | perf分析

性能优化-自上而下的方法论

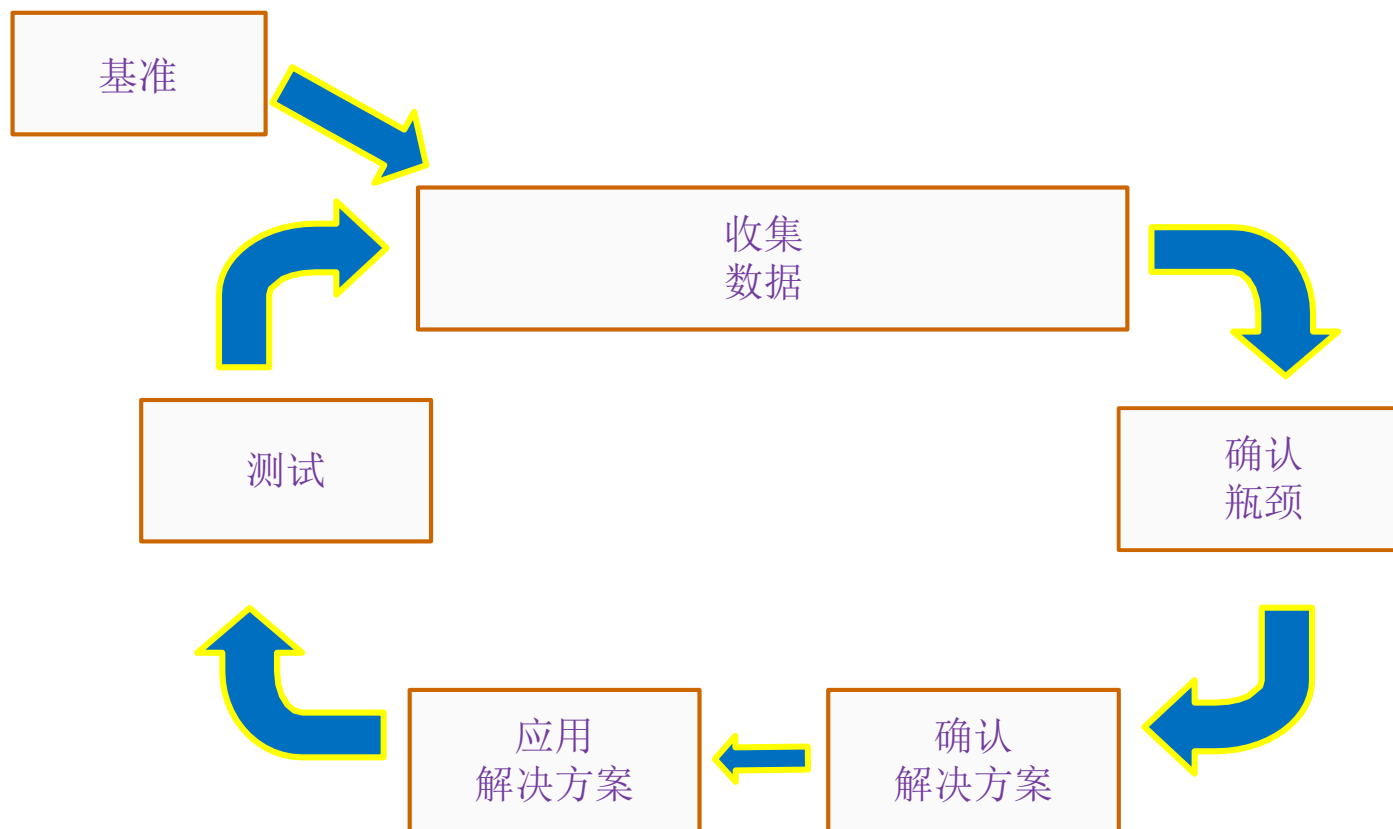
- 系统配置
- 网络 I/O
- 磁盘 I/O
- 数据库调试
- 操作系统



- 基于高速缓存的优化
- 使用指令集进行底层优化

- 应用设计
- 应用服务器优化
- 驱动程序优化
- 并行化
- 隐藏数据传输

使用可重复且具有代表性的基准测试时的问题假设



01

PART

性能的度量指标

➤ 系统性能指标

- ✓ Time
- ✓ CPU利用率
- ✓ 内存使用率及Swap分区
- ✓ 文件系统
- ✓ 网络带宽

➤ 处理器性能指标

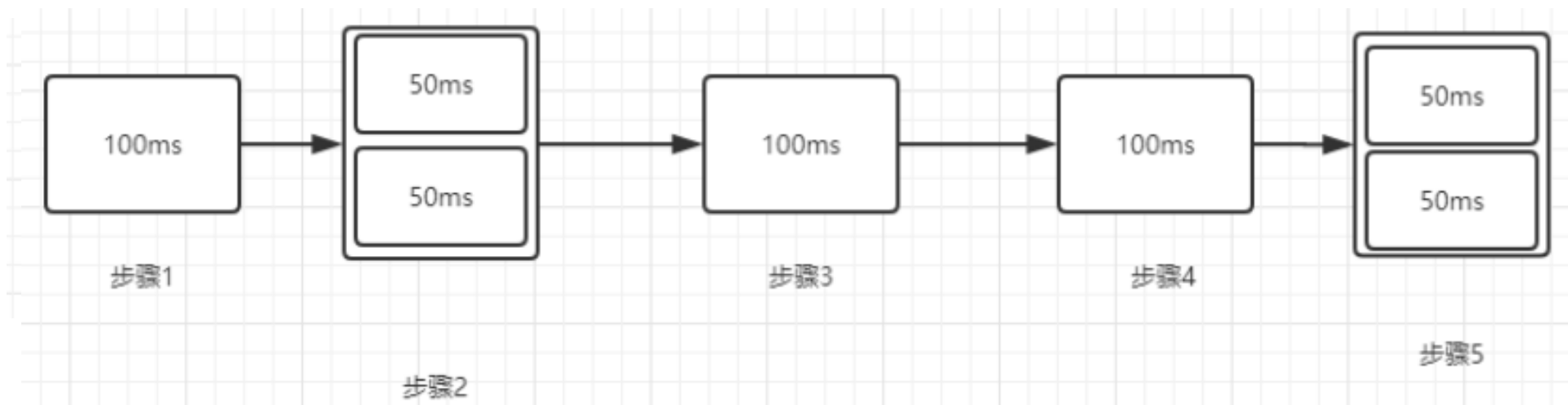
- ✓ CPI
- ✓ 浮点运算峰值
- ✓ Cache miss
- ✓ 向量化比率

Amdahl定律：定义了串行程序并行化后的加速比的计算公式和理论上限。

加速比 = 优化前的程序耗时 / 优化后的程序耗时

$$T_n = T_1(F + \frac{1}{n}(1 - F))$$

$$\text{加速比} \stackrel{\text{def}}{=} \frac{T_1}{T_n} = \frac{T_1}{T_1(F + \frac{1}{n}(1 - F))} = \frac{1}{F + \frac{1}{n}(1 - F)}$$



判断程序优劣最常用，最简单得方法就说计算程序得运行时间

计时的方法有：标准C库的clock系列函数，Windows下的GetTickCount，Linux下的time等命令

```
[root@localhost ~]# time ls
anaconda-ks.cfg  install.log  install.log.syslog  satools  text

real    0m0.009s
user    0m0.002s
sys     0m0.007s
```

real: 命令开始执行到结束的时间。包括其他进程所占用的时间片，和进程被阻塞时所花费的时间

user和 **sys** 时间是 CPU 真正花费在此程序上的时间

user: 进程花费在用户态中的CPU时间

sys: 进程花费在内核态中的CPU时间

当 **user + sys** \geq **real** 时，说明存在计算密集型任务； 当 **user + sys** 远小于 **real** 时，说明存在较多的 IO 等待。

- 运行程序占用的CPU资源的比率。
- 直观反映系统CPU的繁忙程度。

```
top - 08:11:49 up 54 days, 21:14, 1 user, load average: 0.27, 0.31, 0.24
Tasks: 1486 total, 1 running, 1485 sleeping, 0 stopped, 0 zombie
Cpu(s): 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 65973360k total, 3602372k used, 62370988k free, 209800k buffers
Swap: 32989176k total, 0k used, 32989176k free, 1817832k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
48830		20	0	16108	2224	836	R	10.8	0.0	0:00.10	top
514	root	20	0	0	0	0	S	1.8	0.0	42:04.55	kacpid
1	root	20	0	19364	1540	1228	S	0.0	0.0	0:04.74	init
2	root	20	0	0	0	0	S	0.0	0.0	0:00.02	kthreadd
3	root	RT	0	0	0	0	S	0.0	0.0	0:10.25	migration/0
4	root	20	0	0	0	0	S	0.0	0.0	0:00.49	ksoftirqd/0

```
[root@localhost ~]# top
top - 22:47:07 up 2 min, 1 user, load average: 1.00, 0.79, 0.32
Tasks: 176 total, 1 running, 175 sleeping, 0 stopped, 0 zombie
%Cpu0 : 0.3 us, 1.0 sy, 0.0 ni, 98.7 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu1 : 0.3 us, 1.7 sy, 0.0 ni, 97.7 id, 0.0 wa, 0.0 hi, 0.3 si, 0.0 st
KiB Mem : 995924 total, 283744 free, 348004 used, 364176 buff/cache
KiB Swap: 2097148 total, 2097148 free, 0 used. 441036 avail Mem
```

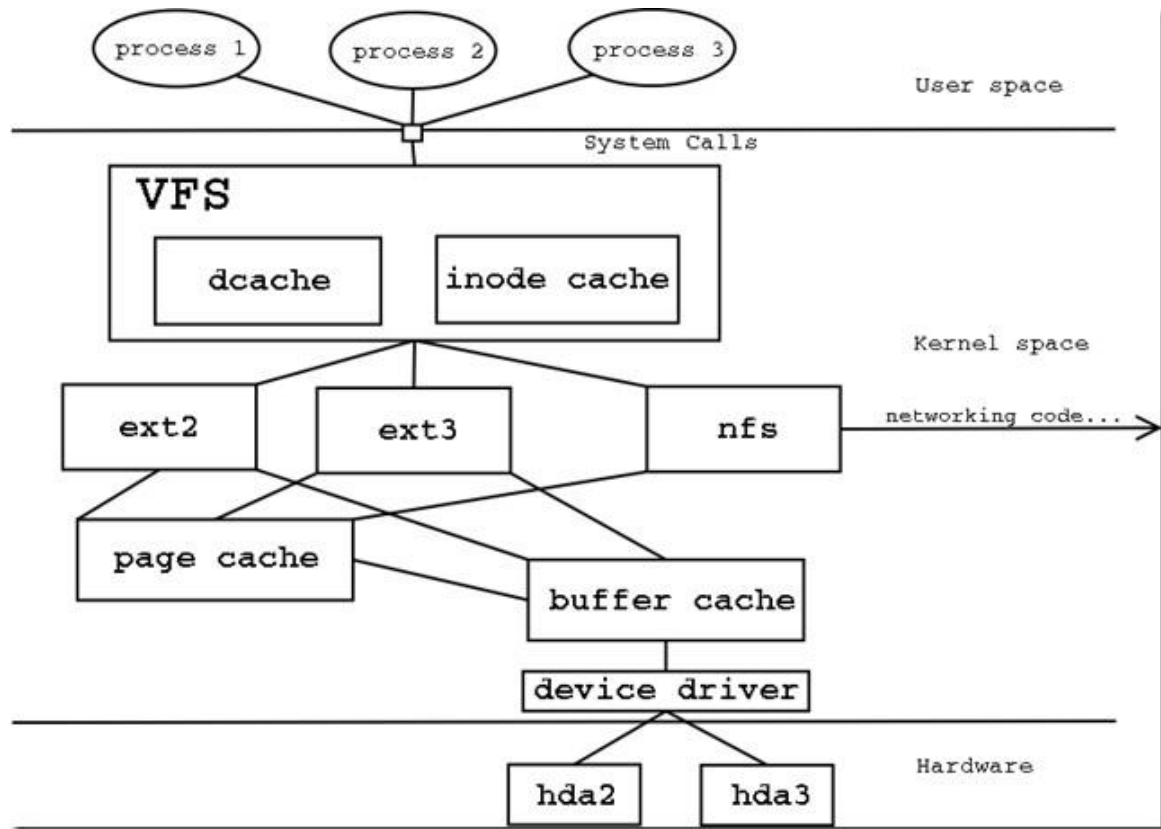
PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
7529	root	20	0	113180	1592	1332	S	0.7	0.2	0:00.25	bash

- Swap分区是将硬盘作为内存扩展。系统在物理内存不够时，与Swap进行交换。
- Swap可以扩展内存使系统功能正常运行，但对性能影响极大。
- 原则上避免使用Swap分区，当应用程序使用Swap分区时，及时分析其原因。

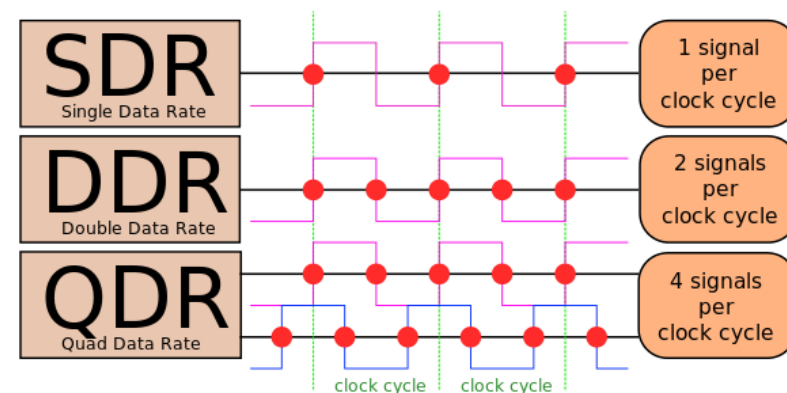
```
top - 08:11:49 up 54 days, 21:14, 1 user, load average: 0.27, 0.31, 0.24
Tasks: 1486 total, 1 running, 1485 sleeping, 0 stopped, 0 zombie
Cpu(s):  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  65973360k total,  3602372k used, 62370988k free,  209800k buffers
Swap: 32989176k total,      0k used, 32989176k free, 1817832k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
48830		20	0	16108	2224	836	R	10.8	0.0	0:00.10	top
514	root	20	0	0	0	0	S	1.8	0.0	42:04.55	kacpid
1	root	20	0	19364	1540	1228	S	0.0	0.0	0:04.74	init
2	root	20	0	0	0	0	S	0.0	0.0	0:00.02	kthreadd
3	root	RT	0	0	0	0	S	0.0	0.0	0:10.25	migration/0
4	root	20	0	0	0	0	S	0.0	0.0	0:00.49	ksoftirqd/0

- 文件系统对外提供统一接口
- 文件系统缓存磁盘内容：
 - ✓ 逻辑I/O V.S. 物理I/O
 - ✓ 文件I/O未必引发真正磁盘操作
- 硬盘分类
 - ✓ HDD 111.63 MB/sec
 - ✓ SSD 370.89 MB/sec
- 网络文件系统NFS
 - ✓ 在磁盘文件系统的基础上增加网络通讯



- 网络
 - ✓ 以太网（千兆、万兆）
 - ✓ Infiniband网络
- 当机器配置多种网络时，需要查看是否使用 高速的网络。



	SDR	DDR	QDR	FDR-10	FDR	EDR	HDR	NDR
Signaling rate (Gb/s)	2.5	5	10	10	14.0625 ^[4]	25	50	
Theoretical effective throughput, Gbs, per 1x ^[5]	2	4	8	9.70	13.64	24.24		
Speeds for 4x links (Gbit/s)	8	16	32	38.79	54.54	96.97		
Speeds for 12x links (Gbit/s)	24	48	96	116.36	163.64	290.91		
Encoding (bits)	8/10	8/10	8/10	64/66	64/66	64/66		
Latency (microseconds) ^[6]	5	2.5	1.3	0.7	0.7	0.5		
Year ^[7]	2001, 2003	2005	2007		2011	2014 ^[5]	~2017 ^[5]	after 2020

程序执行时间 = 程序总指令数 * 每 CPU 时钟周期时间 * 每指令执行所需平均时钟周期数 (CPI)

- CPI表示每条指令完成所需要的周期数。
- CPU使用多发射技术，每个周期最多4条指令
 - ✓ 理想状态 $CPI = 0.25$
- CPI过高：
 - ✓ 所选代码进一步优化的可能性高。

CPI 小于 1，程序通常是 CPU Bound； CPI 大于 1，程序通常是 I/O Bound；

CPU浮点计算理论峰值 = CPU核数*CPU主频*CPU每个时钟周期浮点运算次数。

例如：2x xxx processor E5-2670 2.6GHz

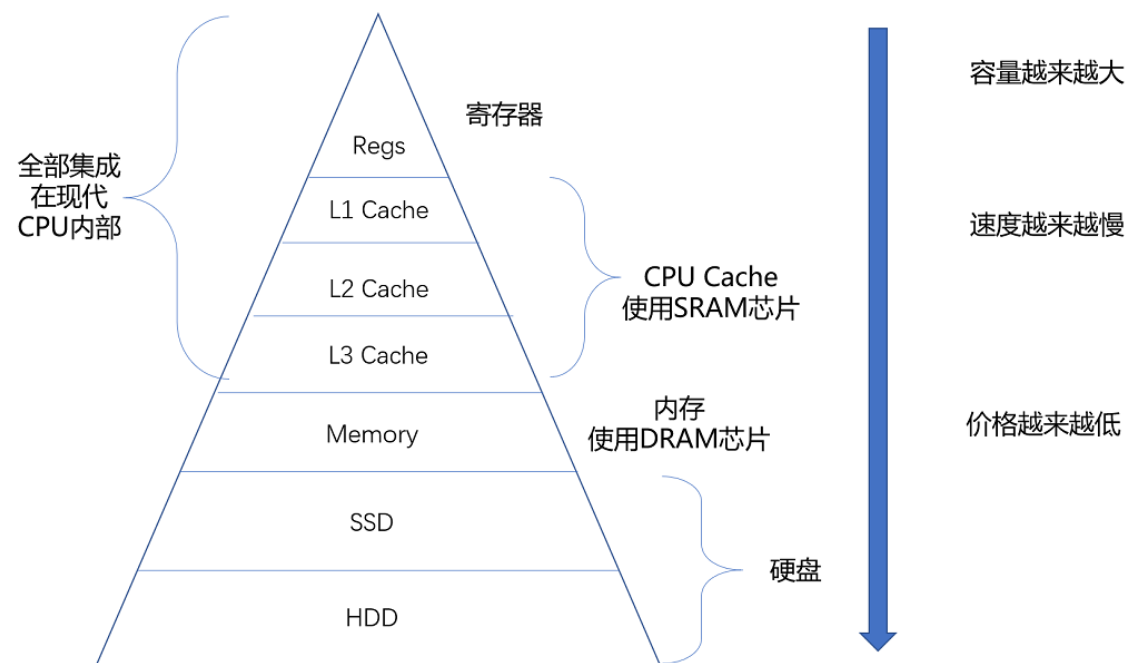
✓ $8 \text{ cores/socket} * 2 \text{ sockets} * 2.6\text{GHz} * 8 \text{ Flops/clock} = 332.8 \text{ GF/s}$

- 科学计算度量以浮点数为主
- DGEMM ~93
- LINPACK ~91

应用名称	最大Gflops	平均Gflops
vasp	52.725	3.642
Gaussian	17.581	0.843
lammps	0.541	0.221
nand	31.408	4.376
gromacs	23.417	0.986

- 数据的存储层次（访问延迟）
 - ✓ 寄存器 (1cycle)
 - ✓ cache: L1 (~4cycles), L2(~10cycles), LLC(~50cycles)
 - ✓ 内存(~300 cycles)
 - ✓ 本地硬盘（固态 50~150us，旋转 1~10ms）
 - ✓ 网络传输
- cache 利用数据访问局部性原理，缓存最近常访问的数据。
- L1-D cache miss range**

Good	Poor
0.005	0.10



- 向量化是CPU峰值计算的倍数因子。
- 对应用程序性能影响很大。
- 取值范围为0~100 。
- 向量化指令需要根据应用的逻辑。

02

PART

性能分析工具

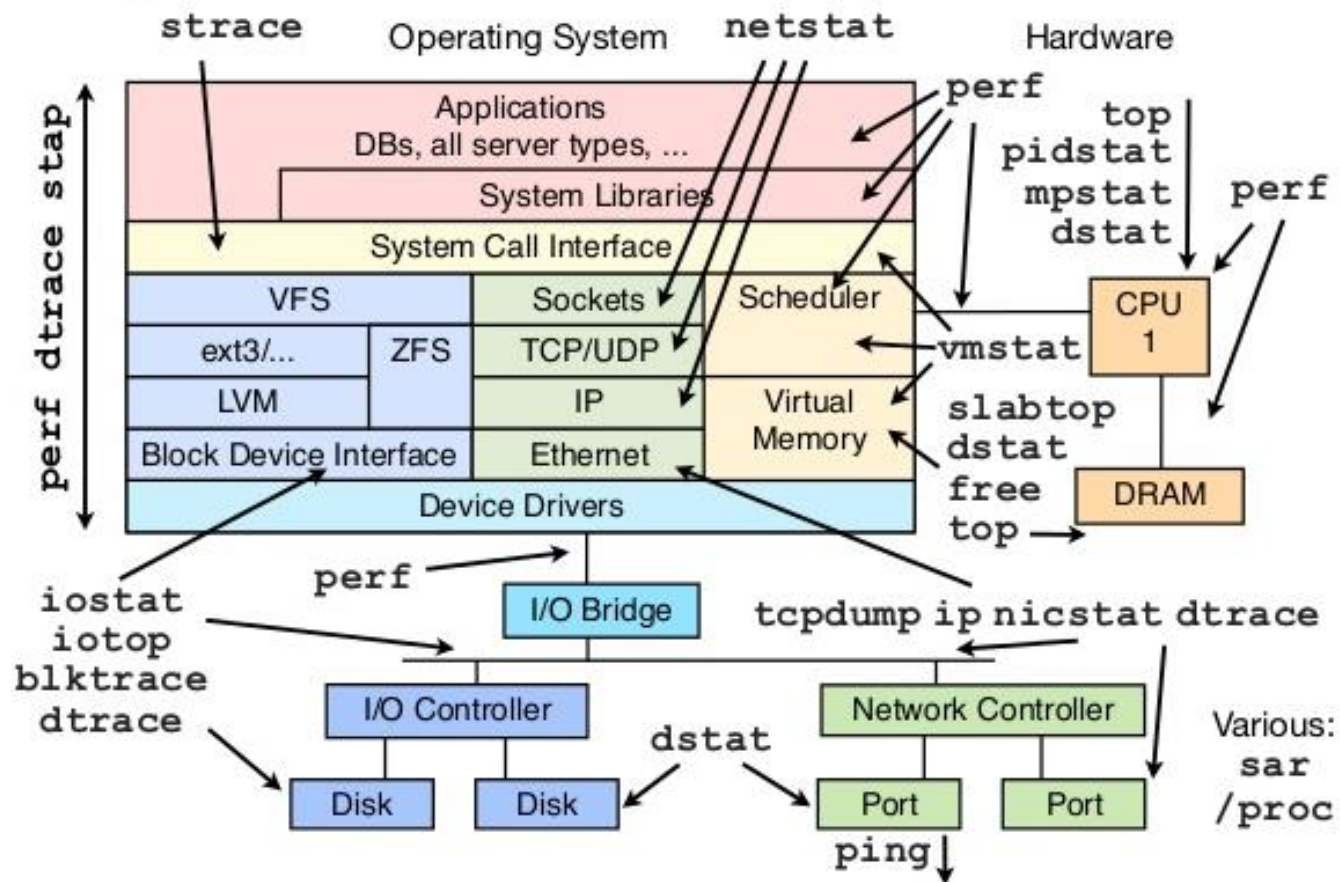
- 分析程序运行过程对系统资源（CPU、内存，网络带宽，文件系统）的使用情况
- 找出使用饱和的系统资源。

Linux 性能调优工具—Brendan Gregg


Analysis and Tools

- vmstat--虚拟内存统计
- iostat--用于报告中央处理器统计信息
- top、htop、free、netstat
-

都可以通过**man**命令来获得它的使用文档



- Perf
- Gprof
- Valgrind

```
test.cpp >  accu(int *, long &)  
1  #include <stdio.h>  
2  #include <unistd.h>  
3  using namespace std;  
4  #define NUM 500000  
5  void init(int* int_array){  
6      for(int i=0; i<NUM;i++){  
7          int_array[i]=i;  
8      }  
9  }  
10 void accu(int* int_array, long& sum ){  
11     for(int i=0; i<NUM; i++){  
12         sum+=int_array[i];  
13         usleep(3);  
14     }  
15 }  
16 int main()  
17 {  
18     int int_array[NUM];  
19     init(int_array);  
20  
21     long sum=0;  
22     accu(int_array, sum);  
23     return 0;  
24 }
```

Perf是内置于Linux内核源码中的性能剖析(profiling)工具。其基于事件采样原理，以性能事件为基础，常用于性能瓶颈的查找与热点代码的定位，查看**cashe miss**的比率。

- ✓ perf的使用可以分为两种方式:
 - 直接使用perf启动服务
 - 挂接到已启动的进程

```
perf record -e cpu-clock -g ./run
或者
perf record -e cpu-clock -g -p 4522
```

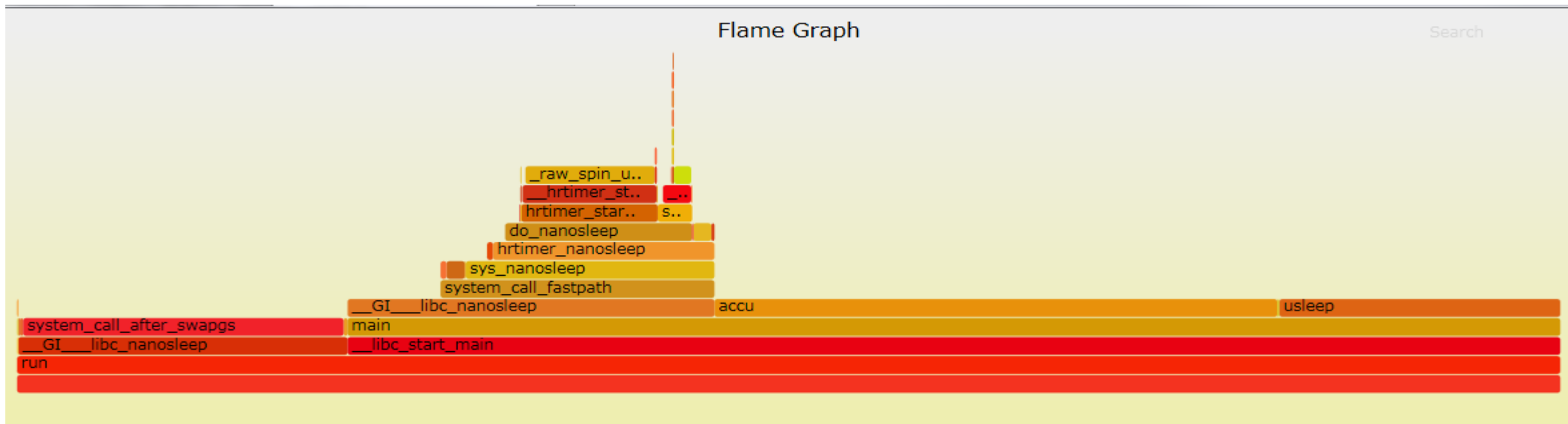
- ✓ 生成perf.data的文件，使用perf report查看

Events: 1K cpu-clock				
+	36.48%	run	run	[.] accu(int*, long&)
+	20.76%	run	[kernel.kallsyms]	[k] system_call_after_swaps
+	18.21%	run	libc-2.12.so	[.] usleep
+	8.45%	run	[kernel.kallsyms]	[k] _raw_spin_unlock_irqrestore
+	6.02%	run	libc-2.12.so	[.] __GI___libc_nanosleep
+	1.57%	run	[kernel.kallsyms]	[k] sys_nanosleep
+	1.31%	run	[kernel.kallsyms]	[k] do_nanosleep
+	1.18%	run	[kernel.kallsyms]	[k] hrtimer_init
+	1.18%	run	[kernel.kallsyms]	[k] copy_user_enhanced_fast_string
+	1.11%	run	[kernel.kallsyms]	[k] finish_task_switch
+	0.85%	run	[kernel.kallsyms]	[k] hrtimer_nanosleep
+	0.59%	run	[kernel.kallsyms]	[k] __schedule
+	0.39%	run	[kernel.kallsyms]	[k] __copy_from_user
+	0.33%	run	[kernel.kallsyms]	[k] schedule
+	0.26%	run	[kernel.kallsyms]	[k] hrtimer_start_range_ns
+	0.26%	run	[kernel.kallsyms]	[k] system_call_fastpath
+	0.20%	run	[kernel.kallsyms]	[k] __hrtimer_start_range_ns
+	0.13%	run	run	[.] init(int*)
+	0.13%	run	[kernel.kallsyms]	[k] _raw_spin_lock_irqsave
+	0.13%	run	[kernel.kallsyms]	[k] _raw_spin_lock_irq
+	0.13%	run	[kernel.kallsyms]	[k] ret_from_sys_call
+	0.07%	run	ld-2.12.so	[.] __GI___dl_allocate_tls_init
+	0.07%	run	[kernel.kallsyms]	[k] __do_softirq
+	0.07%	run	[kernel.kallsyms]	[k] run_timer_softirq
+	0.07%	run	[kernel.kallsyms]	[k] lock_hrtimer_base
+	0.07%	run	[kernel.kallsyms]	[k] hrtimer_cancel

perf的结果可以生成火焰图（.svg文件）。生成火焰图需要借助Flame Graph

Flame Graph项目位于GitHub: <https://github.com/brendangregg>

- 程序火焰图



gprof用于监控程序中每个方法的执行时间和被调用次数，方便找出程序中最耗时的函数。在程序正常退出后，会生成gmon.out文件，解析这个文件，可以生成一个可视化的报告

1. 程序在编译时，加入-pg选项
2. 运行程序生成gmon.out文件
3. 生成报告文件：

gprof -b run gmon.out >>report.txt

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
100.87	0.04	0.04	1	40.35	40.35	accu(int*, long&)
0.00	0.04	0.00	1	0.00	0.00	init(int*)

FF

Call graph

granularity: each sample hit covers 2 byte(s) for 24.78% of 0.04 seconds

index	% time	self	children	called	name
					<spontaneous>
[1]	100.0	0.00	0.04		main [1]
		0.04	0.00	1/1	accu(int*, long&) [2]
		0.00	0.00	1/1	init(int*) [9]

		0.04	0.00	1/1	main [1]
[2]	100.0	0.04	0.00	1	accu(int*, long&) [2]

		0.00	0.00	1/1	main [1]
[9]	0.0	0.00	0.00	1	init(int*) [9]

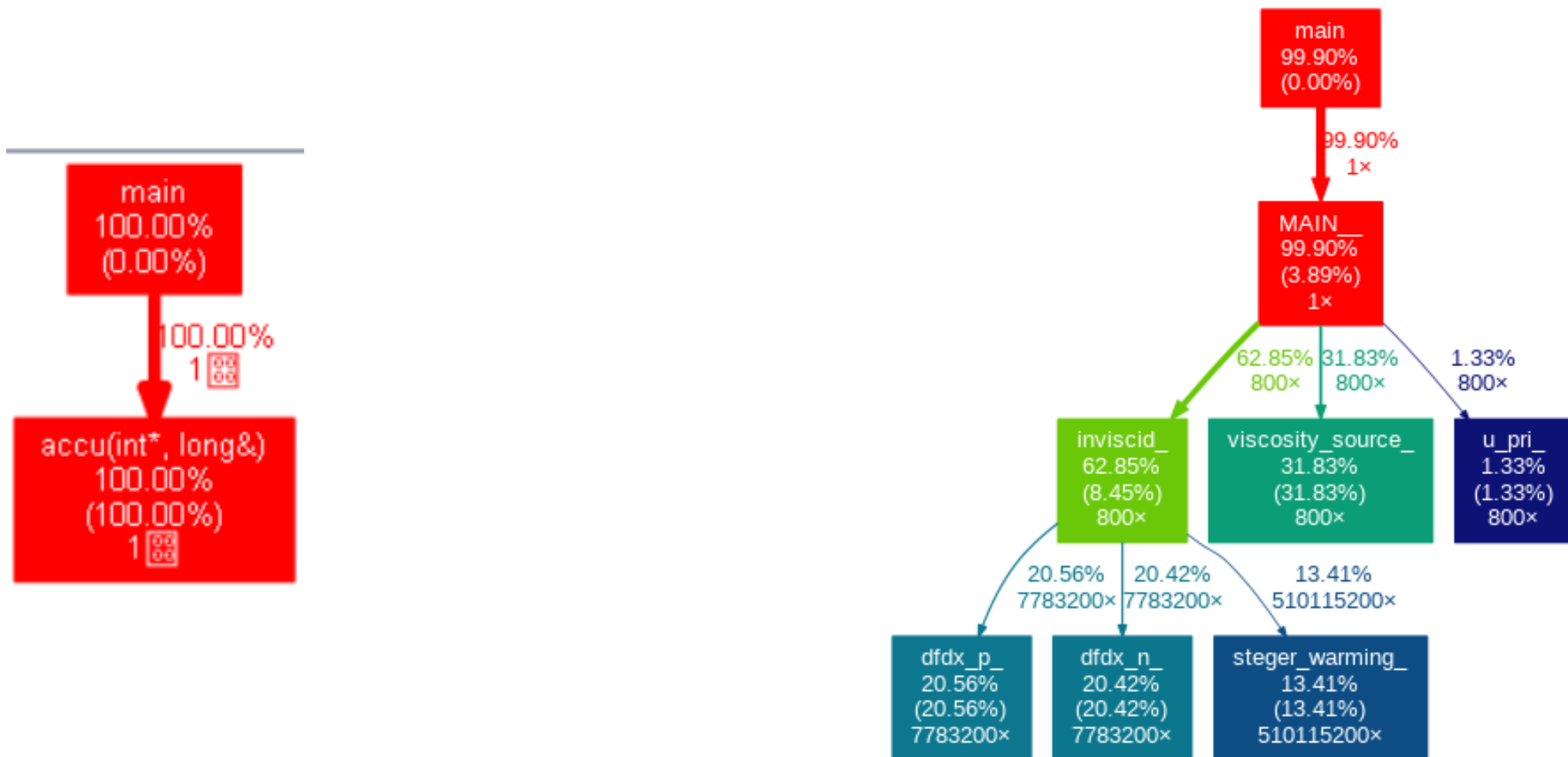
FF

Index by function name

[2] accu(int*, long&)

[9] init(int*)

gprof的结果文件可使用gprof2dot.py和graphviz，生成png文件（需安装Python）



valgrind不是linux的原生工具，需要自行安装。valgrind自身包含了多个工具：

- Memcheck: 用于内存泄漏检查
- Callgrind: 用于性能分析，会收集程序运行时间和调用关系
- 以及Cachegrind、Helgrind等

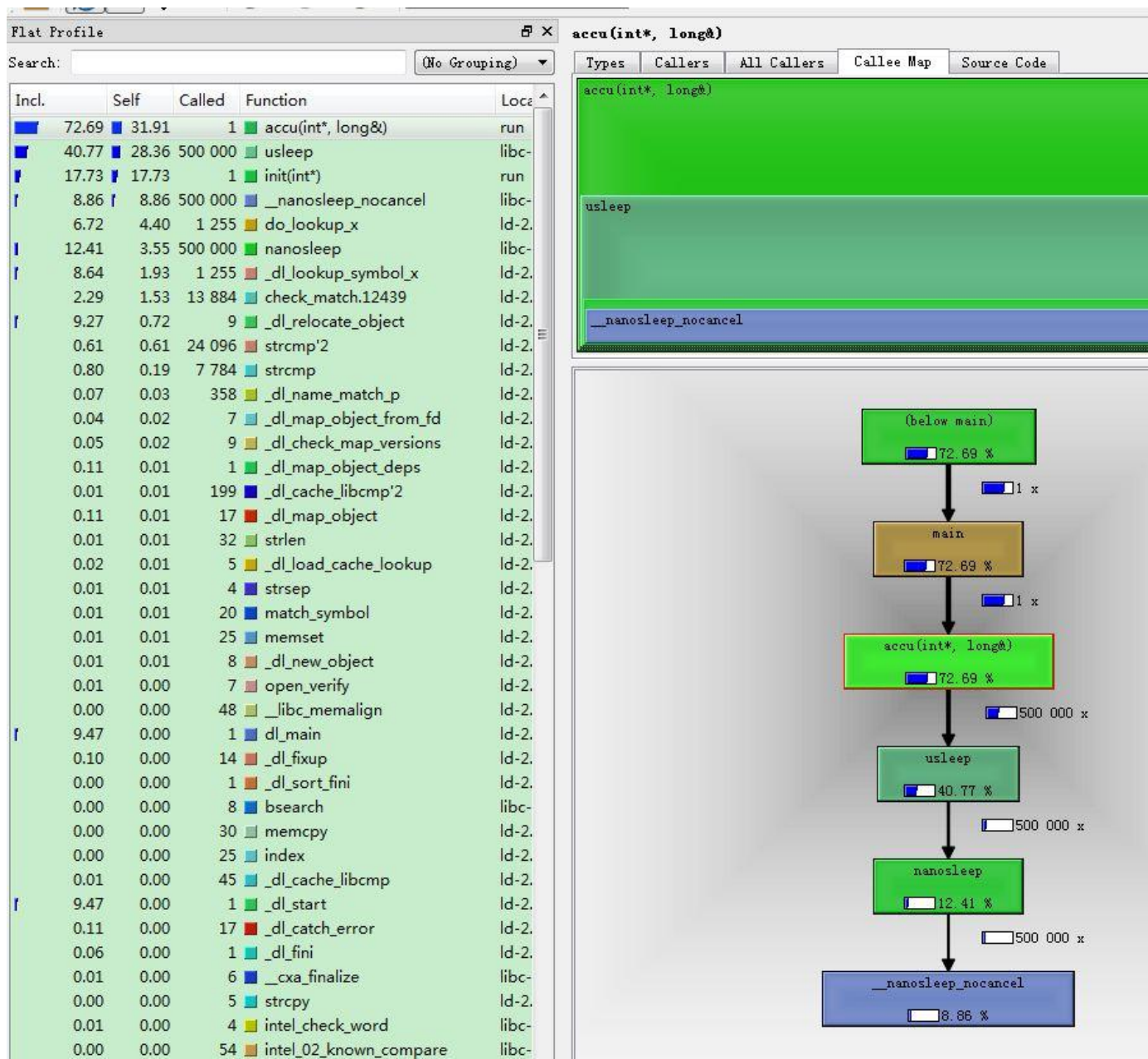
主要使用的Callgrind工具

用valgrind启动程序：

```
valgrind --tool=callgrind --separate-threads=yes ./run
```

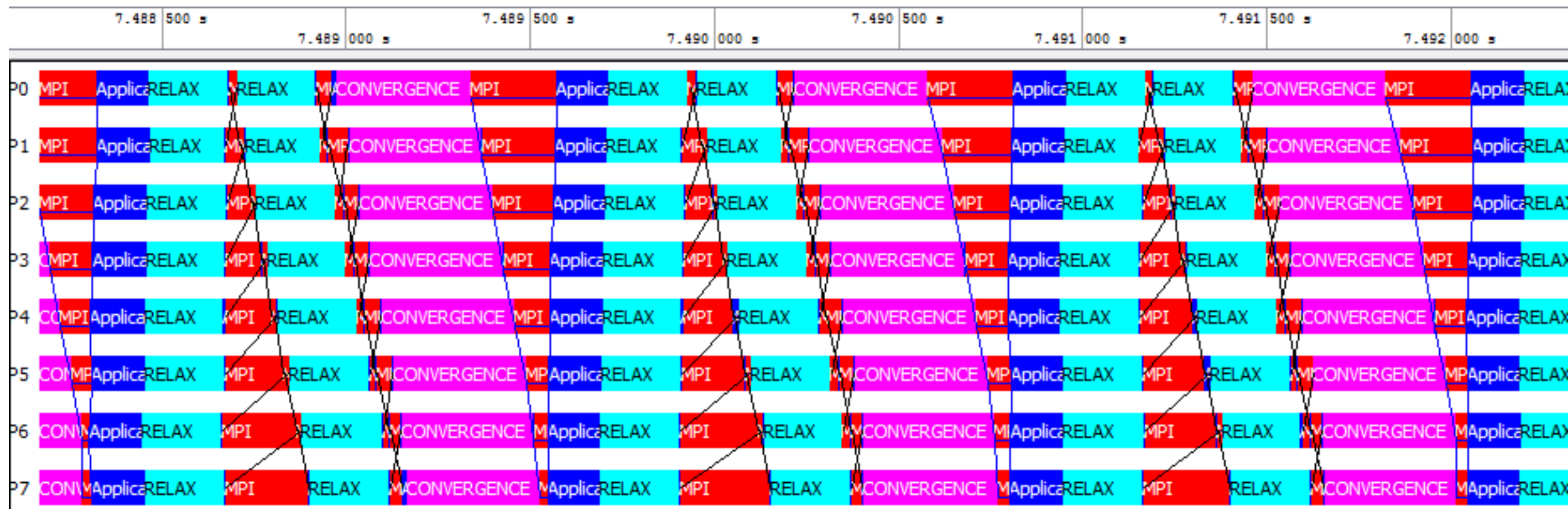

Valgrind 的图形化需要借助 kcachegrind.exe

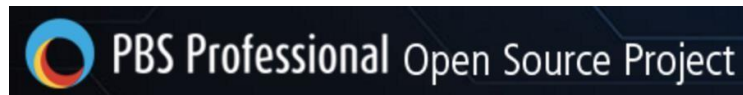
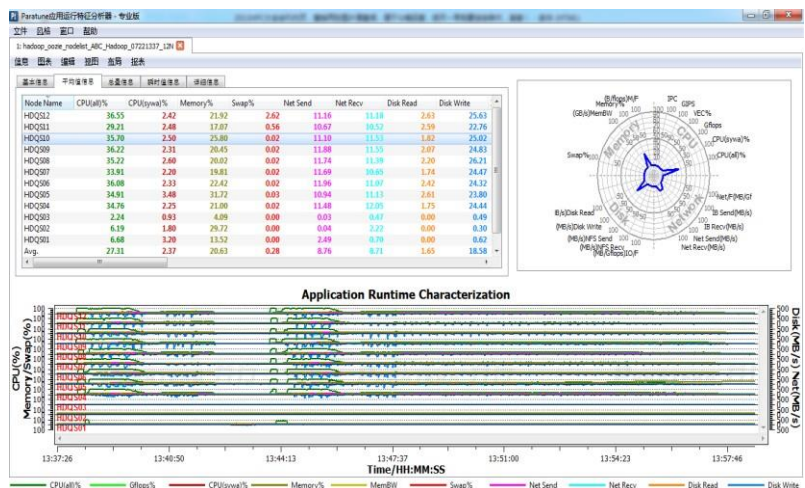
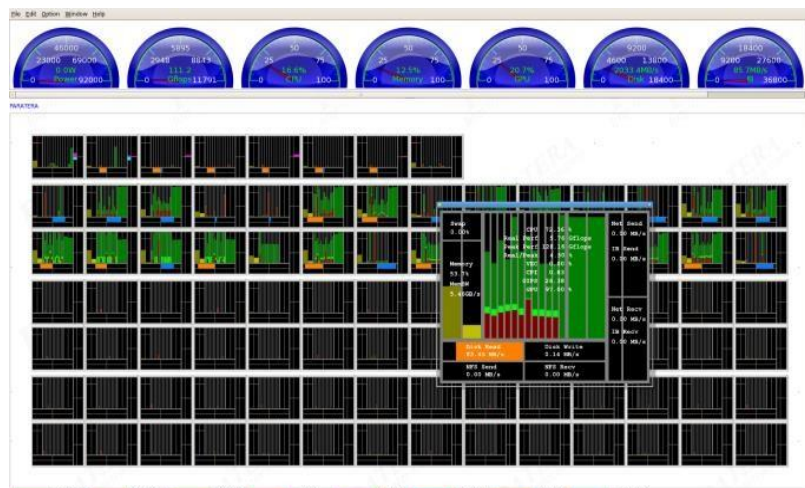
调用关系比gprof更细节



MPI运行时性能分析工具ITAC

复现整个MPI程序过程中，具体通讯函数操作历程

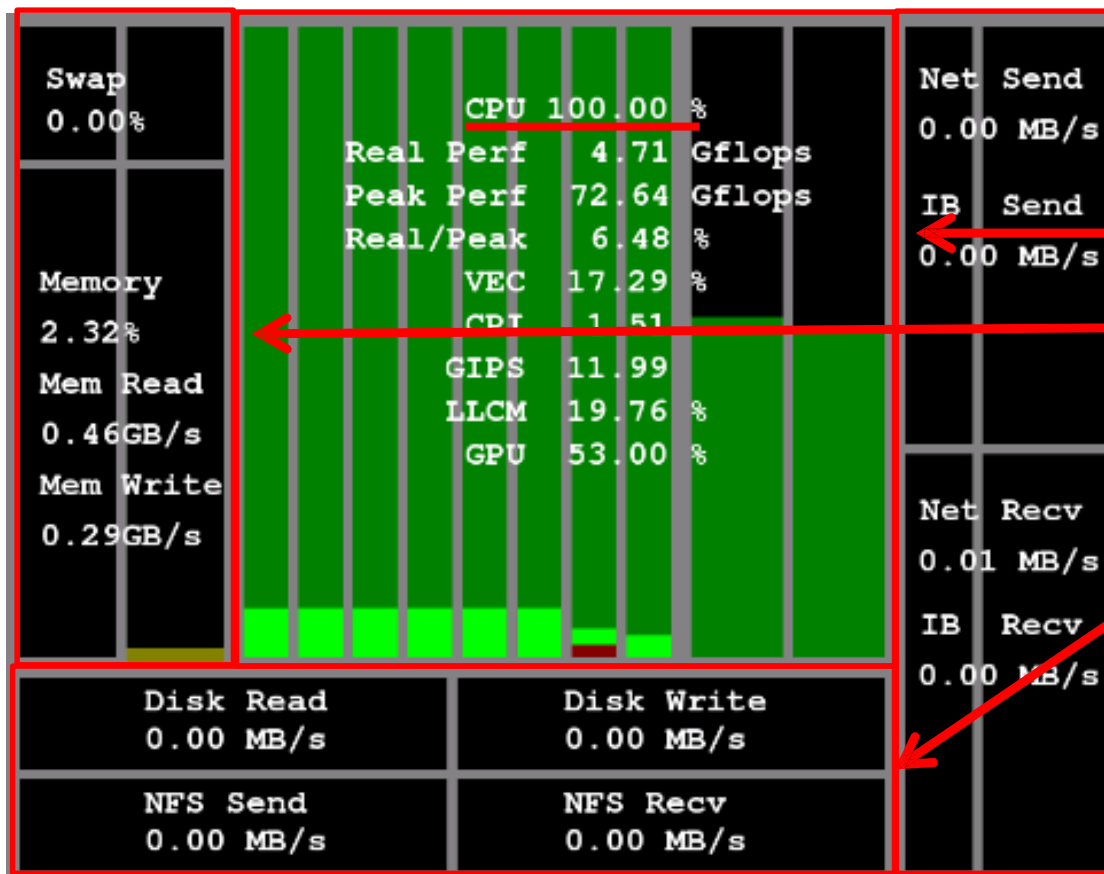




Paramon直观可视化程序运行过程中的系统级、微架构级和函数级等性能指标，为程序开发者清晰指明程序优化方向。

Paratune用于分析优化程序性能，尤其针对大规模并行计算程序，通过多节点间系统级和微架构级等性能指标定性与定量对比分析，帮助程序开发者快速定位程序性能瓶颈点，优化编程提高程序性能。

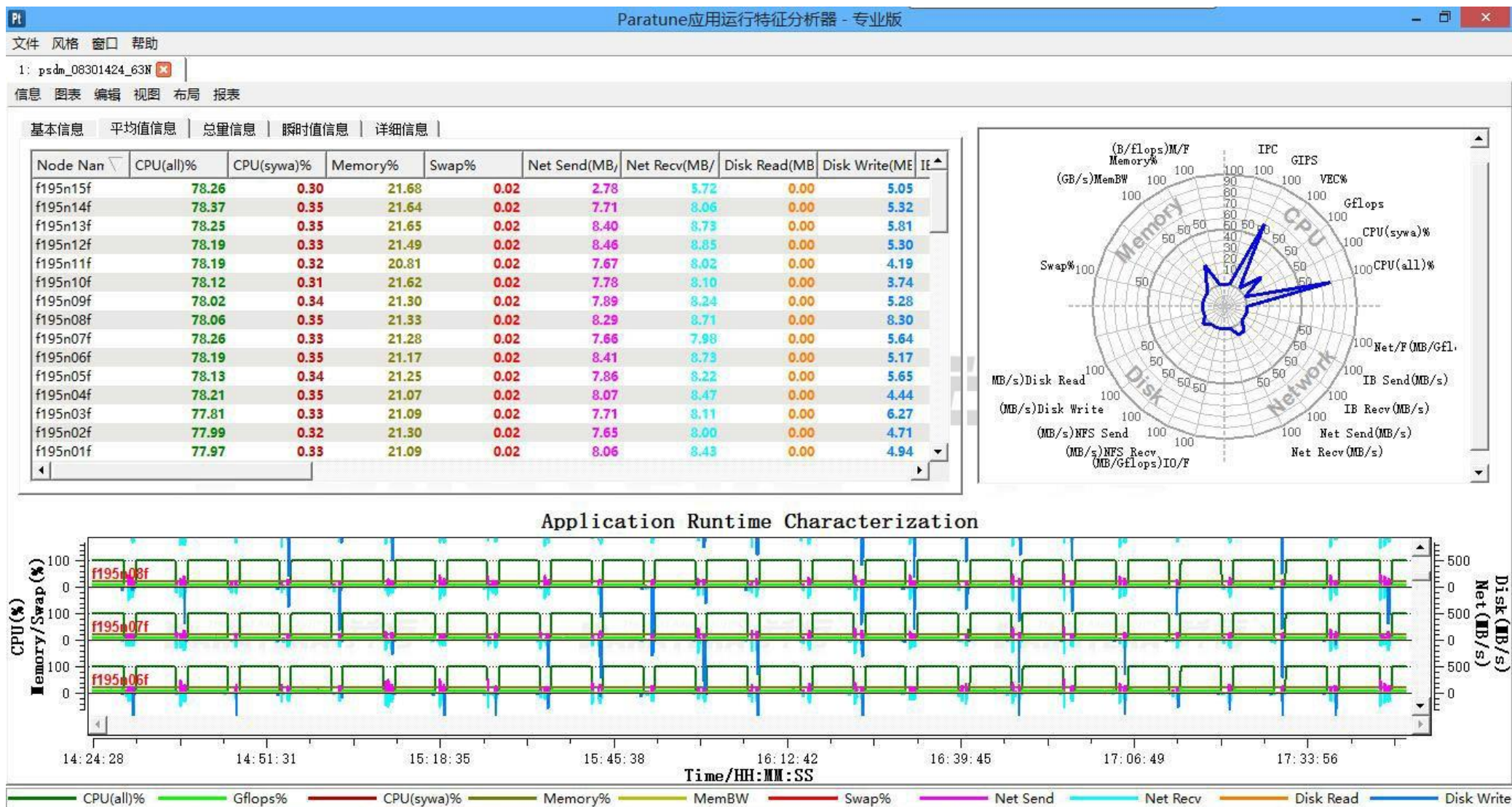
- Paramon 单体窗口以四个维度进行划分



维度三：方向

CPU USED%/GPU%	UP
CPU efficiency calculation %	UP
CPU SYS%	UP
Memory%	UP
Memory BandWidth (GB/s)	UP
SWAP%	UP
Net Send(MB/s)	UP
Net Recv(MB/s)	down
Disk Read(MB/s)	left
Disk Write(MB/s)	right

维度四：数值



03

PART

案例分析

—矩阵乘法, Perf工具

```
double A[NUM][NUM], B[NUM][NUM], C[NUM][NUM];
void Muti(){
    int i,j,k;
    double sum=0;
    for (i=0;i<n;i++){
        for(j=0;j<n;j++){
            sum=0.0;
            for(k=0;k<n;k++){
                sum+=A[i][k]*B[k][j];
            }
            C[i][j]+=sum;
        }
    }
```

```
double A[NUM][NUM], B[NUM][NUM], C[NUM][NUM];
void Muti(){
    int i,j,k;
    double sum=0;
    for (j=0;j<n;j++){
        for(k=0;k<n;k++){
            sum=B[k][j];
            for(i=0;i<n;i++){
                C[i][j]+=A[i][k]*sum;
            }
        }
    }
```

```
double A[NUM][NUM], B[NUM][NUM], C[NUM][NUM];
void Muti(){
    int i,j,k;
    double sum=0;
    for (k=0;k<n;k++){
        for(i=0;i<n;i++){
            sum=A[i][k];
            for(j=0;j<n;j++){
                C[i][j]+=B[k][j]*sum;
            }
        }
    }
```

j k i

Performance counter stats for './test' (5 runs):

344,469,856	cache-references			(+- 1.06%)	(33.33%)
133,361	cache-misses	#	0.039 % of all cache refs	(+- 29.06%)	(50.01%)
21,521,231,104	instructions	#	2.05 insn per cycle	(+- 0.01%)	(66.68%)
10,515,757,098	cycles			(+- 1.69%)	(83.34%)
6,157,849,725	L1-dcache-loads			(+- 0.01%)	(83.30%)
1,026,484,801	L1-dcache-load-misses	#	16.67% of all L1-dcache hits	(+- 0.01%)	(33.32%)

3.5206 +- 0.0168 seconds time elapsed (+- 0.48%)

i j k

Performance counter stats for './test' (5 runs):

65,839,775	cache-references			(+- 0.09%)	(33.32%)
116,318	cache-misses	#	0.177 % of all cache refs	(+- 46.06%)	(50.03%)
16,404,068,739	instructions	#	2.86 insn per cycle	(+- 0.03%)	(66.70%)
5,726,997,438	cycles			(+- 0.44%)	(83.35%)
5,134,882,294	L1-dcache-loads			(+- 0.01%)	(83.28%)
531,383,748	L1-dcache-load-misses	#	10.35% of all L1-dcache hits	(+- 0.03%)	(33.30%)

1.9985 +- 0.0814 seconds time elapsed (+- 4.07%)

[' ']@localhost ~]\$ perf stat -r 5 -e cache-references,cache-misses,instructions,cycles,L1-dcache-loads,L1-dcache-load-misses ./test

Performance counter stats for './test' (5 runs):

3,263,657	cache-references			(+- 1.53%)	(33.33%)
56,386	cache-misses	#	1.728 % of all cache refs	(+- 11.98%)	(50.02%)
21,510,781,056	instructions	#	3.41 insn per cycle	(+- 0.01%)	(66.69%)
6,311,460,315	cycles			(+- 0.22%)	(83.35%)
6,156,628,522	L1-dcache-loads			(+- 0.00%)	(83.29%)
65,660,314	L1-dcache-load-misses	#	1.07% of all L1-dcache hits	(+- 0.05%)	(33.31%)

2.3022 +- 0.0404 seconds time elapsed (+- 1.75%)

k i j

优化方法

◆ 缓存优化

分析stencil.cc中的代码容易发现，两层for的计算，内层循环是按列循环，根据局部性原理，这种写法的容易造成缓存不命中。

因此，把代码的内外层循环进行调换，优先按行进行计算。提高cache命中率！

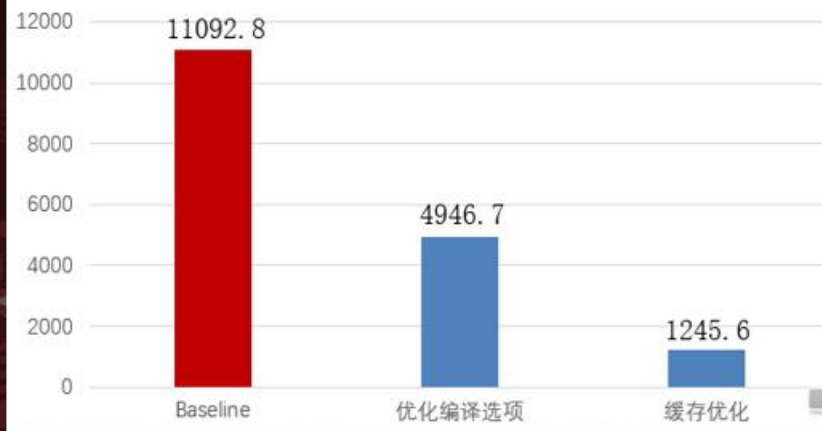
程序平均运行时间为1245.6ms，**加速比为：8.90**

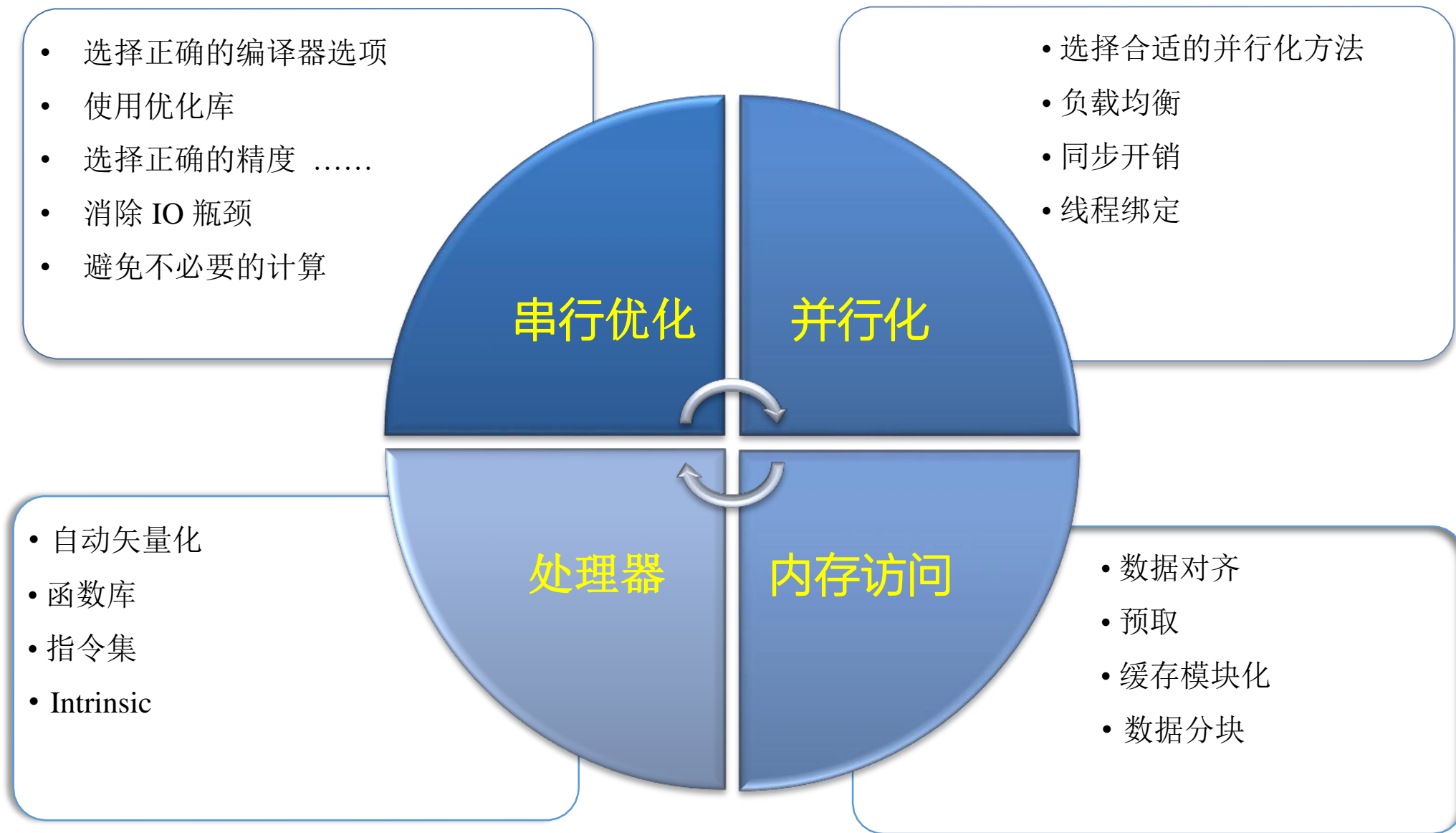
```
for (int j = 1; j < width-1; j++) {  
    for (int i = 1; i < height-1; i++) {  
  
        int im1jm1 =(i-1)*width + j-1;  
        int im1j   =(i-1)*width + j;
```



```
for (int i = 1; i < height-1; i++) {  
    for (int j = 1; j < width-1; j++) {  
  
        int im1jm1 =(i-1)*width + j-1;  
        int im1j   =(i-1)*width + j;
```

优化编译选项+缓存优化







↑ ↑ 扫码参加IPCC

↓↓ 学习交流群



利用并行应用性能监测与分析工具，快速直观定位性能瓶颈，进而优化程序性能

- 正确提交并行规模，充分发挥计算性能
- 选择合适的调度规模，避免频繁调度，系统开销过大
- 在新平台编译运行，充分利用新处理器的高效先进指令集
- 选择合适的并行规模，避免耗光内存导致程序严重低效运行
- 为内存带宽需求较敏感的应用，匹配合适的计算平台
- 针对磁盘交互敏感的应用，配置SSD
- 合理设置文件系统参数
- 正确配置使用高速网络

2021

谢谢观看

时间：2021年5月

主讲人：赵雄君