Performance Evaluation

Master 2 Research Tutorial: High-Performance Architectures

Arnaud Legrand et Jean-François Méhaut

ID laboratory, arnaud.legrand@imag.fr

November 29, 2006

Code Performance

- We will mostly talk about how to make code go fast, hence the "High Performance".
- Performance conflicts with other concerns:

  Correctness. You will see that when trying to make code go fast one often breaks it

  Readability. Fast code typically requires more lines! Modularity can hurt performance (e.g., Too many classes)

  Portability.
  - Code that is fast on machine A can be slow on machine B
  - At the extreme, highly optimized code is not portable at all, and in fact is done in hardware!

Why Performance?

- To do a time-consuming operation in less time
  - I am an aircraft engineer
  - I need to run a simulation to test the stability of the wings at high speed
  - I'd rather have the result in 5 minutes than in 5 hours so that I can complete the aircraft final design sooner.
- To do an operation before a tighter deadline
  - I am a weather prediction agency
  - I am getting input from weather stations/sensors
  - I'd like to make the forecast for tomorrow before tomorrow
- To do a high number of operations per seconds
  - I am the CTO of Amazon.com
  - My Web server gets $1,000$ hits per seconds
  - I'd like my Web server and my databases to handle $1,000$ transactions per seconds so that customers do not experience bad delays (also called scalability)
  - Amazon does "process" several GBytes of data per seconds

Outline

1. Performance: Definition?
   - Time?
   - Rate?
   - Peak performance
   - Benchmarks
2. Speedup and Efficiency
   - Speedup
   - Amdahl's Law
3. Performance Measures
   - Measuring Time
4. Performance Improvement
   - Finding Bottlenecks
   - Profiling Sequential Programs
   - Profiling Parallel Programs
   - The Memory Bottleneck

Outline

1 Performance: Definition?
- Time?
- Rate?
- Peak performance
- Benchmarks

2 Speedup and Efficiency
- Speedup
- Amdahl's Law

3 Performance Measures
- Measuring Time

4 Performance Improvement
- Finding Bottlenecks
- Profiling Sequential Programs
- Profiling Parallel Programs
- The Memory Bottleneck

Performance as Time

- Time between the start and the end of an operation
  - Also called running time, elapsed time, wall-clock time, response time, latency, execution time, ...
  - Most straightforward measure: "my program takes 12.5s on a Pentium 3.5GHz"
  - Can be normalized to some reference time
- Must be measured on a "dedicated" machine

Performance as Rate

Used often so that performance can be independent on the "size" of the application (e.g., compressing a 1MB file takes 1 minute. compressing a 2MB file takes 2 minutes $\rightsquigarrow$ the performance is the same).

MIPS Millions of instructions / sec $= \dfrac{\text{instruction count}}{\text{execution time} \times 10^6} = \dfrac{\text{clock rate}}{\text{CPI} \times 10^6}$.

But Instructions Set Architectures are not equivalent

- 1 CISC instruction = many RISC instructions
- Programs use different instruction mixes
- May be ok for same program on same architectures

MFlops Millions of floating point operations /sec

- Very popular, but often misleading
- e.g., A high MFlops rate in a stupid algorithm could have poor application performance

Application-specific

- Millions of frames rendered per second
- Millions of amino-acid compared per second
- Millions of HTTP requests served per seconds

Application-specific metrics are often preferable and others may be misleading

"Peak" Performance?

Resource vendors always talk about peak performance rate

- computed based on specifications of the machine
- For instance:
  - I build a machine with 2 floating point units
  - Each unit can do an operation in 2 cycles
  - My CPU is at 1GHz
  - Therefore I have a 1*2/2 =1GFlops Machine
- Problem:
  - In real code you will never be able to use the two floating point units constantly
  - Data needs to come from memory and cause the floating point units to be idle

Typically, real code achieves only an (often small) fraction of the peak performance

Benchmarks

- Since many performance metrics turn out to be misleading, people have designed benchmarks
- Example: SPEC Benchmark
  - Integer benchmark
  - Floating point benchmark
- These benchmarks are typically a collection of several codes that come from "real-world software"
- The question "what is a good benchmark" is difficult
  - If the benchmarks do not correspond to what you'll do with the computer, then the benchmark results are not relevant to you

How About GHz?

- This is often the way in which people say that a computer is better than another
  - More instruction per seconds for higher clock rate
- Faces the same problems as MIPS

Processor	Clock Rate	SPEC FP2000 Benchmark
IBM Power3	450 MHz	434
Intel PIII	1.4 GHz	456
Intel P4	2.4GHz	833
Itanium-2	1.0GHz	1356

- But usable within a specific architecture

Program Performance

- In this class we're not really concerned with determining the performance of a compute platform (whichever way it is defined)
- Instead we're concerned with improving a program's performance
  - For a given platform, take a given program
  - Run it an measure its wall-clock time
  - Enhance it, run it an quantify the performance improvement (i.e., the reduction in wall-clock time)
  - For each version compute its performance
    - preferably as a relevant performance rate
    - so that you can say: the best implementation we have so far goes "this fast" (perhaps a % of the peak performance)

Outline

1. Performance: Definition?
   - Time?
   - Rate?
   - Peak performance
   - Benchmarks
2. Speedup and Efficiency
   - Speedup
   - Amdahl's Law
3. Performance Measures
   - Measuring Time
4. Performance Improvement
   - Finding Bottlenecks
   - Profiling Sequential Programs
   - Profiling Parallel Programs
   - The Memory Bottleneck

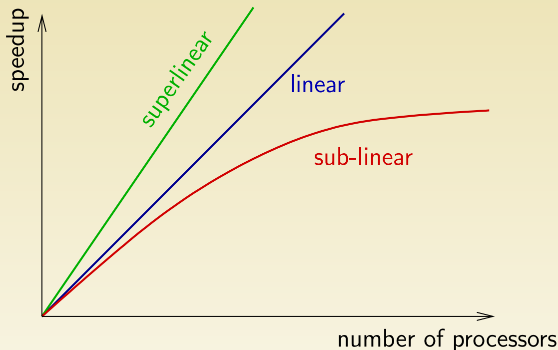
Speedup

- We need a metric to quantify the impact of your performance enhancement
- Speedup: ratio of "old" time to "new" time
  - new time = 1h
  - speedup = 2h / 1h = 2
- Sometimes one talks about a "slowdown" in case the "enhancement" is not beneficial
  - Happens more often than one thinks

Parallel Performance

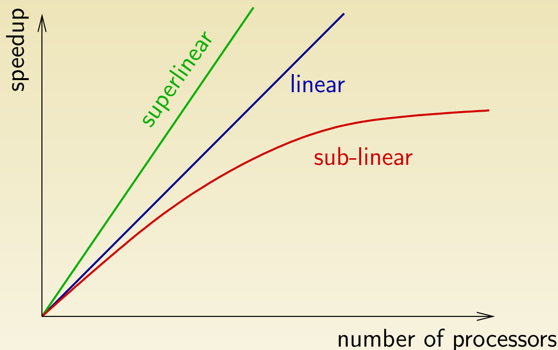
- The notion of speedup is completely generic
  - By using a rice cooker I've achieved a 1.20 speedup for rice cooking
- For parallel programs one defines the Parallel Speedup (we'll just say "speedup"):
  - Parallel program takes time T_1 on 1 processor
  - Parallel program takes time T_p on p processors
  - Parallel Speedup: $S(p) = \frac{T_1}{T_p}$
- In the ideal case, if my sequential program takes 2 hours on 1 processor, it takes 1 hour on 2 processors: called linear speedup

Speedup



Superlinear Speedup? There are several possible causes

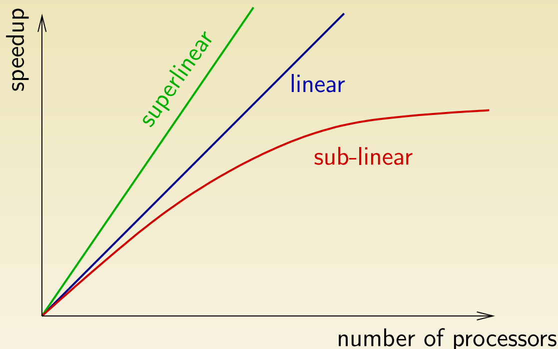
Speedup



Superlinear Speedup? There are several possible causes

Algorithm with optimization problems, throwing many processors at it increases the chances that one will "get lucky" and find the optimum fast

Speedup



Superlinear Speedup? There are several possible causes

Algorithm with optimization problems, throwing many processors at it increases the chances that one will "get lucky" and find the optimum fast

Hardware with many processors, it is possible that the entire application data resides in cache (vs. RAM) or in RAM (vs. Disk)

Bad News: Amdahl's Law

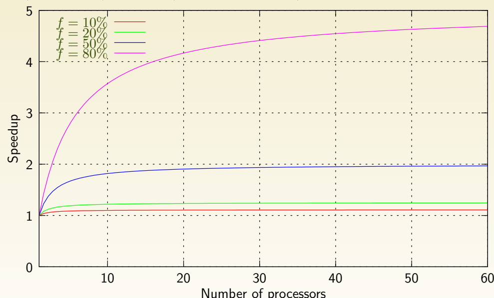
Consider a program whose execution consists of two phases

1. One sequential phase : $T_{seq} = (1 - f)T_1$
2. One phase that can be perfectly parallelized (linear speedup) $T_{par} = fT_1$

Therefore: $T_p = T_{seq} + T_{par}/p = (1 - f)T_1 + fT_1/p$.

Amdahl's Law:

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



Lessons from Amdahl's Law

- It's a law of diminishing return
- If a significant fraction of the code (in terms of time spent in it) is not parallelizable, then parallelization is not going to be good
- It sounds obvious, but people new to high performance computing often forget how bad Amdahl's law can be
- Luckily, many applications can be almost entirely parallelized and f is small

Parallel Efficiency

- Efficiency is defined as $\mathit{Eff}(p) = S(p)/p$
- Typically < 1, unless linear or superlinear speedup
- Used to measure how well the processors are utilized
  - If increasing the number of processors by a factor 10 increases the speedup by a factor 2, perhaps it's not worth it: efficiency drops by a factor 5
  - Important when purchasing a parallel machine for instance: if due to the application's behavior efficiency is low, forget buying a large cluster

Scalability

- Measure of the "effort" needed to maintain efficiency while adding processors
- Efficiency also depends on the problem size: $\mathit{Eff}(n, p)$
- Isoefficiency: At which rate does the problem size need to be increase to maintain efficiency
  - $n_c(p)$ such that $\mathit{Eff}(n_c(p), p) = c$
  - By making a problem ridiculously large, on can typically achieve good efficiency
  - Problem: is it how the machine/code will be used?

Outline

1. Performance: Definition?
   - Time?
   - Rate?
   - Peak performance
   - Benchmarks
2. Speedup and Efficiency
   - Speedup
   - Amdahl's Law
3. Performance Measures
   - Measuring Time
4. Performance Improvement
   - Finding Bottlenecks
   - Profiling Sequential Programs
   - Profiling Parallel Programs
   - The Memory Bottleneck

Performance Measures

This is all well and good, but how does one measure the performance of a program in practice?

Two issues:

1. Measuring wall-clock times (We'll see how it can be done shortly)
2. Measuring performance rates
    - Measure wall clock time (see above)
    - "Count" number of "operations" (frames, flops, amino-acids: whatever makes sense for the application)
        - Either by actively counting (count++)
        - Or by looking at the code and figure out how many operations are performed
    - Divide the count by the wall-clock time

Measuring time by hand?

- One possibility would be to do this by just "looking" at a clock, launching the program, "looking" at the clock again when the program terminates
- This of course has some drawbacks
  - Poor resolution
  - Requires the user's attention
- Therefore operating systems provide ways to time programs automatically
- UNIX provide the `time` command

The UNIX time Command

- You can put `time` in front of any UNIX command you invoke
- When the invoked command completes, time prints out timing (and other) information

```
surf:~$ /usr/bin/X11/time ls -la -R ~/ > /dev/null
4.17user 4.34system 2:55.83elapsed 4%CPU
(0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (0major+1344minor)pagefaults 0swaps
```

  - 4.17 seconds of user time
  - 4.34 seconds of system time
  - 2 minutes and 55.85 seconds of wall-clock time
  - 4% of CPU was used
  - 0+0k memory used (text + data)
  - 0 input, 0 output output (file system I/O)
  - 1344 minor pagefaults and 0 swaps

User, System, Wall-Clock?

- User Time: time that the code spends executing user code (i.e., non system calls)

User, System, Wall-Clock?

- User Time: time that the code spends executing user code (i.e., non system calls)
- System Time: time that the code spends executing system calls

User, System, Wall-Clock?

- User Time: time that the code spends executing user code (i.e., non system calls)
- System Time: time that the code spends executing system calls
- Wall-Clock Time: time from start to end

User, System, Wall-Clock?

- User Time: time that the code spends executing user code (i.e., non system calls)
- System Time: time that the code spends executing system calls
- Wall-Clock Time: time from start to end
- Wall-Clock $\geq$ User $+$ System. Why?

User, System, Wall-Clock?

- User Time: time that the code spends executing user code (i.e., non system calls)
- System Time: time that the code spends executing system calls
- Wall-Clock Time: time from start to end
- Wall-Clock $\geq$ User + System. Why?
  - because the process can be suspended by the O/S due to contention for the CPU by other processes

User, System, Wall-Clock?

- User Time: time that the code spends executing user code (i.e., non system calls)
- System Time: time that the code spends executing system calls
- Wall-Clock Time: time from start to end
- Wall-Clock $\geq$ User + System. Why?
  - because the process can be suspended by the O/S due to contention for the CPU by other processes
  - because the process can be blocked waiting for I/O

Using time

- It's interesting to know what the user time and the system time are
  - for instance, if the system time is really high, it may be that the code does to many calls to malloc(), for instance
  - But one would really need more information to fix the code (not always clear which system calls may be responsible for the high system time)
- Wall-clock - system - user $\simeq$ I/O $+$ suspended
  - If the system is dedicated, suspended $\simeq 0$
  - Therefore one can estimate the ecost of I/O
  - If I/O is really high, one may want to look at reducing I/O or doing I/O better
- Therefore, time can give us insight into bottlenecks and gives us wall-clock time
- Measurements should be done on dedicated systems

Dedicated Systems

- Measuring the performance of a code must be done on a "quiescent", "unloaded" machine (the machine only runs the standard O/S processes)
- The machine must be dedicated
  - No other user can start a process
  - The user measuring the performance only runs the minimum amount of processes (basically, a shell)
- Nevertheless, one should always present measurement results as averages over several experiments (because the (small) load imposed by the O/S is not deterministic)

Drawbacks of UNIX `time`

- The `time` command has poor resolution
  - "Only" milliseconds
  - Sometimes we want a higher precision, especially if our performance improvements are in the 1-2% range
- `time` times the whole code
  - Sometimes we're only interested in timing some part of the code, for instance the one that we are trying to optimize
  - Sometimes we want to compare the execution time of different sections of the code

Timing with `gettimeofday`

- `gettimeofday` from the standard C library
- Measures the number of microseconds since midnight, Jan 1st 1970, expressed in seconds and microseconds

```
struct timeval start;
...
gettimeofday(&tv,NULL);
printf("%ld,%ld\n",start.tv_sec, start.tv_usec);
```

- Can be used to time sections of code
  - Call gettimeofday at beginning of section
  - Call gettimeofday at end of section
  - Compute the time elapsed in microseconds: `(end.tv_sec*1000000.0 + end.tv_usec - start.tv_sec*1000000.0 - start.tv_usec) / 1000000.0`)

Other Ways to Time Code

- `ntp_gettime()` (Internet RFC 1589)
  - Sort of like `gettimeofday`, but reports estimated error on time measurement
  - Not available for all systems
  - Part of the GNU C Library
- Java: `System.currentTimeMillis()`
  - Known to have resolution problems, with resolution higher than 1 millisecond!
  - Solution: use a native interface to a better timer
- Java: `System.nanoTime()`
  - Added in J2SE 5.0
  - Probably not accurate at the nanosecond level
- Tons of "high precision timing in Java" on the Web

Outline

1. Performance: Definition?
   - Time?
   - Rate?
   - Peak performance
   - Benchmarks
2. Speedup and Efficiency
   - Speedup
   - Amdahl's Law
3. Performance Measures
   - Measuring Time
4. Performance Improvement
   - Finding Bottlenecks
   - Profiling Sequential Programs
   - Profiling Parallel Programs
   - The Memory Bottleneck

Why is Performance Poor?

Performance is poor because the code suffers from a performance bottleneck

Definition:

- An application runs on a platform that has many components (CPU, Memory, Operating System, Network, Hard Drive, Video Card, etc.)
- Pick a component and make it faster
- If the application performance increases, that component was the bottleneck!

Removing a Bottleneck

There are two may approaches to remove a bottleneck:

Brute force Hardware Upgrade

- Is sometimes necessary
- But can only get you so far and may be very costly (e.g., memory technology)

Modify the code

- The bottleneck is there because the code uses a “resource” heavily or in non-intelligent manner
- We will learn techniques to alleviate bottlenecks at the software level

Identifying a Bottleneck

- It can be difficult
  - You're not going to change the memory bus just to see what happens to the application
  - But you can run the code on a different machine and see what happens
- One Approach
  - Know/discover the characteristics of the machine
  - Instrument the code with gettimeofdays everywhere
  - Observe the application execution on the machine
  - Tinker with the code
  - Run the application again
  - Repeat
  - Reason about what the bottleneck is

A better approach: profiling

- A profiler is a tool that monitors the execution of a program and that reports the amount of time spent in different functions
- Useful to identify the expensive functions
- Profiling cycle
  - Compile the code with the profiler
  - Run the code
  - Identify the most expensive function
  - Optimize that function (i.e. call it less often if possible *or* make it faster)
  - Repeat until you can't think of any ways to further optimize the most expensive function

Using gprof

- Compile your code using gcc with the -pg option
- Run your code until completion
- Then run gprof with your program's name as single command-line argument
- Example: `gcc -pg prog.c -o prog; ./prog gprof prog > profile_file`
- The output file contains all profiling information (which fraction of the code is spent in which function)

kprof

File Help

Flat Profile | Hierarchical Profile | Object Profile

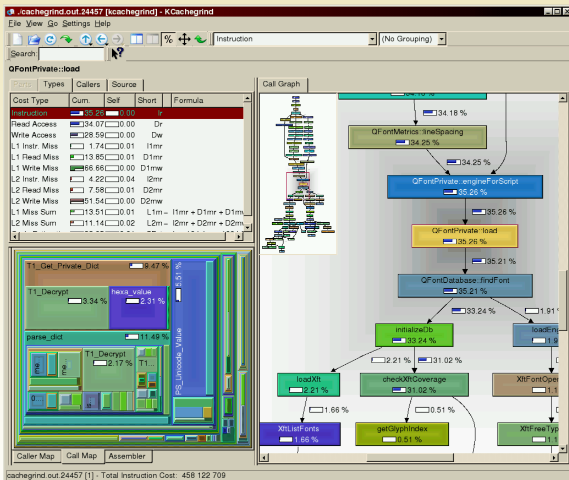
Function/Method	Count	Total (s)	%	Self (s)	Total ms/call	Self ms/call
CProfileInfo::CProfileInfo(void)	69	0.02	0	0	0	0
CProfileViewItem::CProfileViewItem(QListView *, CProfileInfo *)	157	0.02	0	0	0.03	0
CProfileViewItem::CProfileViewItem(QListViewItem *, CProfileInfo *)	224	0.02	0	0	0	0
KAboutData::~KAboutData(void)	1	0.02	0	0	0	0
KProfTopLevel::KProfTopLevel(int, QWidget *, char const *)	1	0.02	0	0	58.87	0
KProfTopLevel::setupActions(void)	1	0.02	0	0	0	0
KProfWidget::KProfWidget(QWidget *, char const *)	1	0.02	0	0	45.94	0
KProfWidget::applySettings(void)	2	0.02	0	0	7.18	0
KProfWidget::loadSettings(void)	1	0.02	0	0	4.31	0
KProfWidget::prepareProfileView(KListView *, bool)	3	0.02	0	0	10.05	0
QString::~QString(void)	6965	0.01	50	0.01	1.44	1.44
QShared::deref(void)	28621	0.02	0	0	0	0
QVector<CProfileInfo>::QVector(void)	1	0.02	0	0	0	0
KProfTopLevel::setupActions(void)	1	0.02	0	0	0	0
KProfTopLevel::staticMetaObject(void)	107	0.02	0	0	0	0
KProfWidget::KProfWidget(QWidget *, char const *)	1	0.02	0	0	45.94	0
KProfWidget::applySettings(void)	2	0.02	0	0	7.18	0
KProfWidget::fillFlatProfileList(void)	1	0.02	0	0	1.89	0
KProfWidget::fillHierProfileList(void)	1	0.02	0	0	1.89	0
KProfWidget::fillHierarchy(CProfileViewItem *, CProfileInfo *, QAr...	69	0.02	0	0	0	0

Callgrind

- Callgrind is a tool that uses runtime code instrumentation framework of Valgrind for call-graph generation
- Valgrind is a kind of emulator or virtual machine.
  - It uses JIT (just-in-time) compilation techniques to translate x86 instructions to simpler form called ucode on which various tools can be executed.
  - The ucode processed by the tools is then translated back to the x86 instructions and executed on the host CPU.
- This way even shared libraries and dynamically loaded plugins can be analyzed but this kind of approach results with huge slow down (about 50 times for callgrind tool) of analyzed application and big memory consumption.

Callgrind/Kcachegrind

Data produced by callgrind can be loaded into KCacheGrind tool for browsing the performance results.



mpiP

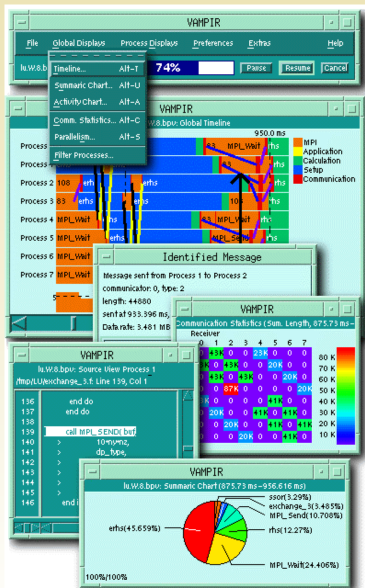
- mpiP is a link-time library (it gathers MPI information through the MPI profiling layer)
- It only collects statistical information about MPI functions
- All the information captured by mpiP is task-local

```
sleeptime = 10;
MPI_Init (&argc, &argv);
MPI_Comm_size (comm, &nprocs);
MPI_Comm_rank (comm, &rank);
MPI_Barrier (comm);
if (rank == 0) sleep (sleeptime);
MPI_Barrier (comm);
MPI_Finalize ();
```

Task	AppTime	MPITime	MPI%
0	10	0.000243	0.00
1	10	10	99.92
2	10	10	99.92
3	10	10	99.92
*	40	30	74.94

vaMPIr

- generate traces (i.e. not just collect statistics) of MPI calls
- These traces can then be visualized and used in different ways.



Removing bottlenecks

- Now we know how to
  - identify expensive sections of the code
  - measure their performance
  - compare to some notion of peak performance
  - decide whether performance is unacceptably poor
  - figure out what the physical bottleneck is
- A very common bottleneck: memory

The Memory Bottleneck

The memory is a very common bottleneck that programmers often don't think about

- When you look at code, you often pay more attention to computation
- `a[i] = b[j] + c[k]`
  - The access to the 3 arrays take more time than doing an addition
  - For the code above, the memory is the bottleneck for most machines!
- In the 70's, everything was balanced. The memory kept pace with the CPU (n cycles to execute an instruction, n cycles to bring in a word from memory)
- No longer true
  - CPUs have gotten 1,000x faster
  - Memory have gotten 10x faster and 1,000,000x larger
- Flops are free and bandwidth is expensive and processors are STARVED for data

Memory Latency and Bandwidth

- The performance of memory is typically defined by Latency and Bandwidth (or Rate)
- Latency: time to read one byte from memory (measured in nanoseconds these days)
- Bandwidth: how many bytes can be read per seconds (measured in GB/sec)
- Note that you don't have bandwidth $= 1$ / latency!
- There is pipelining: Reading 2 bytes in sequence is much cheaper than twice the time reading one byte only

Current Memory Technology

Memory	Latency	Peak Bandwidth
DDR400 SDRAM	10 ns	6.4 GB/sec
DDR533 SDRAM	9.4 ns	8.5 GB/sec
DDR2-533 SDRAM	11.2 ns	8.5 GB/sec
DDR2-800 SDRAM	???	12.8 GB/sec
DDR2-667 SDRAM	???	10.6 GB/sec
DDR2-600 SDRAM	13.3 ns	9.6 GB/sec

Memory Bottleneck: Example

- Fragment of code: `a[i] = b[j] + c[k]`
  - Three memory references: 2 reads, 1 write
  - One addition: can be done in one cycle
- If the memory bandwidth is 12.8GB/sec, then the rate at which the processor can access integers (4 bytes) is: $12.8 \times 1024 \times 1024 \times 1024/4 = 3.4GHz$
- The above code needs to access 3 integers
- Therefore, the rate at which the code gets its data is $\simeq 1.1GHz$
- But the CPU could perform additions at $4GHz$!
- Therefore: The memory is the bottleneck
  - And we assumed memory worked at the peak!!!
  - We ignored other possible overheads on the bus
  - In practice the gap can be around a factor 15 or higher

Dealing with memory

- How have people been dealing with the memory bottleneck?
- Computers are built with a memory hierarchy
  - Registers, Multiple Levels of Cache, Main memory
  - Data is brought in in bulk (cache line) from a lower level (slow, cheap, big) to a higher level (fast, expensive, small)
  - Hopefully brought in in a cache line will be (re)used soon
    - temporal locality
    - spatial locality
- Programs must be aware of the memory hierarchy (at least to some extent)
  - Makes life difficult when writing for performance
  - But is necessary on most systems

Memory and parallel programs

- Rule of thumb: make sure that concurrent processes spend most of their time working on their own data in their own memory (principle of locality)
  - Place data near computation
  - Avoid modifying shared data
  - Access data in order and reuse
  - Avoid indirection and linked data-structures
  - Partition program into independent, balanced computations
  - Avoid adaptive and dynamic computations
  - Avoid synchronization and minimize inter-process communications
- The perfect parallel program: no communication between processors
- Locality is what makes (efficient) parallel programming painful in many cases.
  - As a programmer you must constantly have a mental picture of where all the data is with respect to where the computation is taking place