Lecture 19a – Single and Parallel Processor Performance

- **Mflop Rating of a Single Processor**
  - 1 Mflop = 10^6 Floating Point Operations/Sec
- **Speed of computation: *N* operations in *t* microseconds**

$$r = \frac{N}{t} Mflops$$

- **Execution time:**

$$t = \frac{N}{r} \mu \sec$$

Amdahl's Law – Single Processor

- **Algorithm execution requires, *N* flops**
- **Fraction, *f*, can be executed at speed *V* Mflops; the rest executes at *S* Mflops, and**

$$V >> S$$

Say V is like a Vector speed and S is like a Scalar speed

V is not only relevant for vector machines but is also relevant for GPUs

- **Total execution time:**

$$t = \frac{fN}{V} + \frac{(1-f)N}{S} = N\left(\frac{f}{V} + \frac{1-f}{S}\right)\mu \sec$$

Amdahl's Law - Single Processor

- **Computational performance:**

$$r = \frac{N}{t} = \frac{1}{f/V + (1-f)/S} Mflops$$

- **Therefore:**

$$t > \frac{(1-f)N}{S} \mu \sec$$

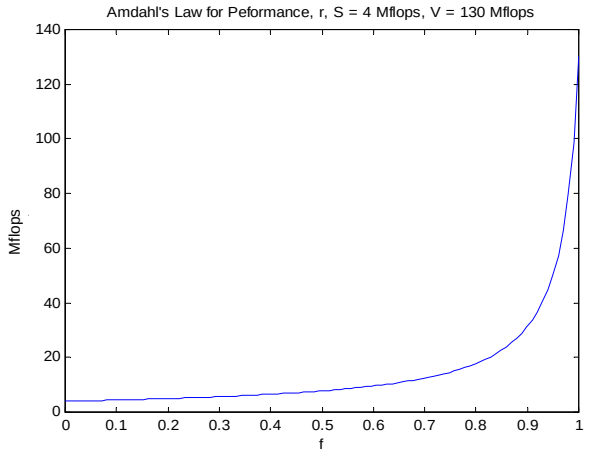
- **If *V* >> *S*, then**

$$r = \frac{1}{f/V + (1-f)/S} \approx \frac{1}{(1-f)} S$$

Amdahl's Law - Single Processor

- **Relative gain in CPU time bounded by** $\frac{1}{1-f}$
- **f needs to be quite large to obtain reasonable performance**

Remember that f is the fraction of algorithm that works at V speed



Amdahl's Law - Single Processor

- **In serial programs, if even a small portion of your algorithm executes at a very slow speed, *S*, the performance of the complete program may be limited by these effects.**
- **Usual approach is to identify the areas of the program that consume the largest percentages of the execution time (main contribution to *f*), and make sure that they are coded in such a way that they can achieve execution speeds closer to *V* than to *S*.**

Single Processor Performance

- **The issues involved in optimizing single processor performance are many, and they are usually dependent on the architecture of the processor at hand.**
- **Consult computer manufacturer's performance tuning guides (usually online) to discover the issues of highest relevance.**
- **For modern RISC, cache-based microprocessors a lot of these most important issues are common.**
- **Link from web site to SGI tuning guide. Very useful!!!**

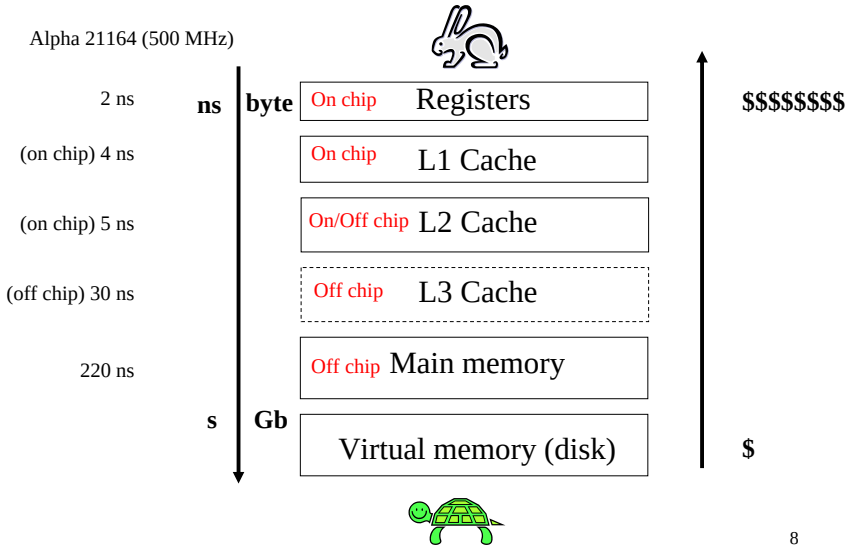
http://www.cecalc.ula.ve/documentacion/tutoriales/O2KPerfTuning/007-3430-003/sgi_html/pr01.html

http://techpubs.sgi.com/library/tpl/cgi-bin/init.cgi

Single Processor Performance

- **Things to worry about:**
  - Use existing optimized code when possible
  - Find out where to tune: profiling tools, performance analysis tools, etc. (prof, hardware counters, etc.)
    - time each subroutine to determine it's overall contribution to total time
  - Loop ordering: Fortran vs. C, column- and row-major order
    - In Fortran, always put loop over the right-most index of arrays on outside
  - Understand cache hierarchy:
    - 2-level cache hierarchies. Size? 32Kb for primary cache, 1-4 Mb for secondary cache
    - cache-block or cache line concept. Size? 32 bytes for primary cache, 128 bytes for secondary cache

Memory Hierarchy



Single Processor Performance

– Understand cache hierarchy:
  - Remember that cache- and page-misses hurt performance a great deal
  - Cache hit and cache miss (hiding memory latency)
  - Access times:
    – registers: 0 cycles
    – 1st level cache: 2-3 clock cycles
    – 2nd level cache: 8-10 clock cycles
    – main memory: 200-1100 nanosec (60-200 cycles!)

Single Processor Performance

Remember that in Fortran a(i,j) is stored as:

- **Stride-1 accesses:**

Stride is Large

Bad programming. Put j-loop on outside, instead.

```
do i=1,n
  do j=1,n
    a(i,j) = b(i,j)
  end do
end do
```

```
a(1,1)
a(2,1)
….
a(1,2)
a(2,2)
…
a(n,m)
```

- **If matrices are large, every element in the loop will have to be loaded multiple times from main memory!**

Single Processor Performance

Remember that in Fortran a(i,j) is stored as:

- **Stride-1 accesses:**

Stride is 1

Good programming.

```
do j=1,n
  do i=1,n
    a(i,j) = b(i,j)
  end do
end do
```

```
a(1,1)
a(2,1)
….
a(1,2)
a(2,2)
…
a(n,m)
```

- **Operations are done one row at a time. Cache is reused efficiently.**

Single Processor Performance

- **Group together data used at the same time:**

Bad programming. Put x,y,z in single array, instead.

Making j-index a function of an array is indirect-addressing…more expensive due to lack of ensured data locality

```
D = 0.0
do i=1,n
  j = ind(i)
  d = d + sqrt(x(j)*x(j) +y(j)*y(j) +z(j)*z(j))
end do
```

- **3 cache lines need to be loaded every time, since x, y, and z, are likely to be stored in different areas of memory.**

Single Processor Performance

- **Instead:**

```
D = 0.0
do i=1,n
  j = ind(i)
  d = d+sqrt(r(1,j)*r(1,j)+r(2,j)*r(2,j)+r(3,j)*r(3,j))
end do
```

Good programming. r-array contains x,y,z even though indirect addressing issues remain

- **Data for x, y, z are now stored in a contiguous array. When loading the cache line for r(1,j), it is likely that the other elements will be loaded at the same time.**

Single Processor Performance

- **Cache thrashing (cache associativeness):**

```
parameter (max=1024*1024)
dimension a(max), b(max), c(max), d(max)
do i=1,max
  a(i) = b(i) + c(i)*d(i)
end do
```

- **Data is stored in cache according to the lower n bits of the memory address (2-way, 4-way, etc)**
- **Arrays are allocated sequentially, therefore, they will be contiguous in memory**

Single Processor Performance

- **Since their total size is 4Mb (single precision), they will share the lowest 22 bits of the memory addresses and the 4 vectors will map to the same cache location**
- **Cache is constantly loading and unloading because of this reason, even though accesses are stride-1, and the performance degrades significantly.**
- **You can re-dimension the arrays so that they do not all map to the same cache location. Good rule of thumb, *do not use powers of 2 in dimension statements.***
- **You can also introduce padding variables in the declaration.**
- **Refer to "Computer Architecture – A Quantitative Approach," by J. L. Hennessy and D. A. Patterson for more information on efficient uses of cache**

Single Processor Performance

- **Pipelining: modern processors can decompose typical operations (multiplication, division, etc) into a series of shorter operations that can be pipelined.**
- **Once the pipeline is filled, results are produced at a much faster rate.**
- **Remember, however, that cache must be used efficiently to pipeline operations, since the CPU must be fed variables to operate on.**
- **Effect of loop length on pipeline performance is significant.**

Amdahl's Law - Parallel Processing

- **In an ideal world, if computation can be carried out in *p* equal parts, the total execution time will be nearly *1/p* of the time required by a single processor**
- **Suppose t_j denotes the wall clock time required to execute a task with j processors**
- ***Speedup*, S_p , for p processors is defined as**

$$S_p = \frac{t_1}{t_p}$$

- **Where t_1 is the time required for the most-efficient sequential algorithm to complete the calculation, and t_p is the time required for the most efficient parallel implementation of the same algorithm, from beginning to end, using p processors.**

Amdahl's Law - Parallel Processing

- **The *computational efficiency* using *p* processors is defined as**

$$E_p = \frac{S_p}{p} \quad , \quad 0 \le E_p \le 1$$

- **Then, the *total execution time* using *p* processors is given by**

$$t_p = \frac{ft_1}{p} + (1-f)t_1 = \frac{t_1(f + (1-f)p)}{p} \ge (1-f)t_1$$

where f is now the fraction of the algorithm that can be carried out in parallel using p processors

Amdahl's Law - Parallel Processing

- **The *speedup* on p processors is then**

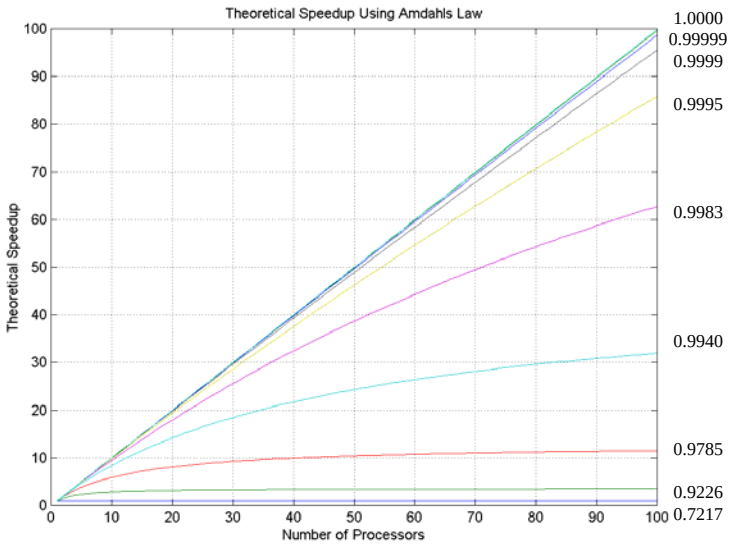
$$S_p = \frac{p}{(f + (1-f)p)} \le \frac{1}{1-f}$$

- **This is often called Ware's Law**
- **This shows that the speedup is considerably reduced even for pretty large values of f (close to 95%)**
- **Example: If f = 0.8 and we used 5 processors, the theoretical speedup would be 2.8! This formula drives us to achieve a high level of parallelism in our algorithms**

Amdahl's Law - Parallel Processing

- ***Parallel overhead* is the additional amount of work that is required on the parallel implementation of a sequential algorithm arising from the use of a parallel computer:**
  - Inter-processor communication
  - Load imbalance
  - Additional computation resulting from the algorithm that is being parallelized not being as efficient as the most efficient serial algorithm.

Amdahl's Law - Parallel Processing



Amdahl's Law - Parallel Processing

- **Amdahl's law is a simplistic, yet a powerful way of looking at the problem of scalability.**
- **In a naïve way, it points out that a large number of processors cannot be used on any computational task, since f needs to be very close to 1.**
- **For example, f=0.999 would allow the use of a maximum number of processors equal to 1000. Does your problem have a 0.999 portion of parallelism?**

Amdahl's Law - Parallel Processing

- **In the next lecture we will investigate the following problems:**
  - Alternative views of Amdahl's law
  - Effect of problem size on parallel efficiency
  - Coarse vs. fine grain parallelism
  - Bandwidth and latency issues
  - Load balancing issues
  - I/O scalability
  - Performance analysis tools