

Lecture 13 – Data Structures for Decomposed Domains on Parallel Computers

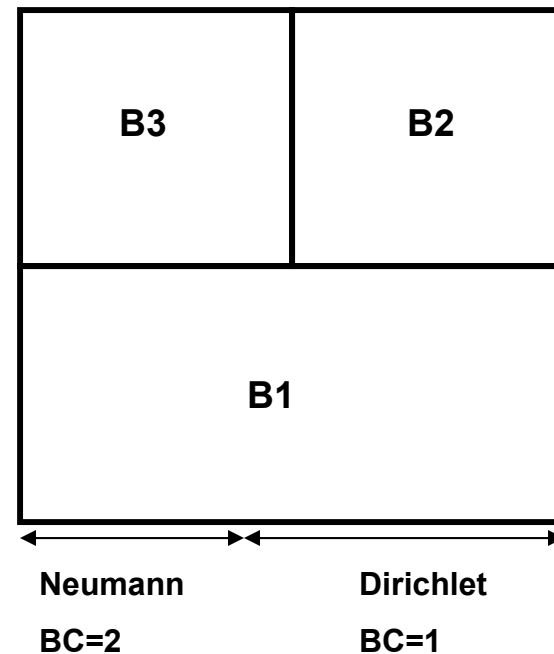
- **Now that we have discussed techniques for performing processor partitioning, we should also discuss the data structures that must be put in place to keep track of other block/processor boundaries**
- **This information can be kept as part of the “connectivity” file**

Additional Connectivity Information

- **In project 2, you developed a data structure to keep track of block neighbors for your multi-block solvers.**
- **This connectivity information kept track of**
 - Neighbor information on sides (and corners) of each block
 - Boundary condition options on physical domain boundaries
- **All blocks were kept in a single processor (project 3)**
- **All block numbers were considered to be in global space (ie they were all on the same processor)**
- **All blocks were assumed to have only a single neighbor or boundary condition to each side (or corner)**
- **All blocks had same orientation**
- **Now let's consider a more general capability**

Additional Connectivity Information

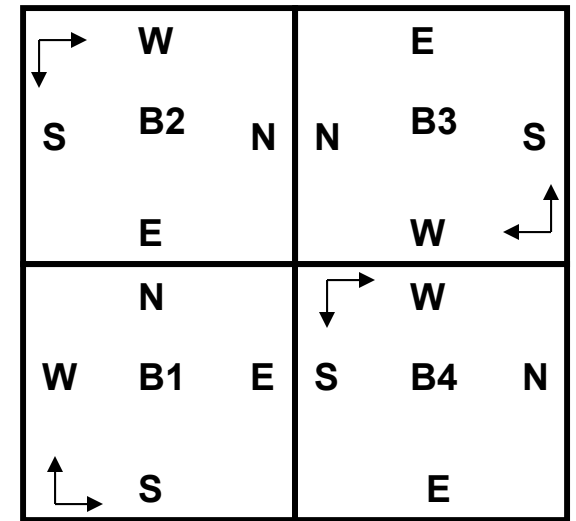
- Additional connectivity information would be required if blocks could have more than one neighbor or boundary condition type per edge/corner
- *Sub-faces* can be used to enable multiple boundary conditions per side/corner



The same idea also holds for processor boundaries

Additional Connectivity Information

- **Orientation of blocks can be non-uniform with any face matched to any other face of a neighbor**
- **Block neighbor orientations can also be kept as part of a connectivity file**
- **For instance:**
 - Let $+i\text{-direction}=1$, $+j\text{-direction}=2$
 - Block 1's orientation could be $1\ 2$
 - (ie its i -direction is in the $+i$ -direction and
 - its j -direction is in the $+j$ -direction
 - Block 2's orientation could be $-2\ 1$
 - (ie its i -direction is in the $-j$ -direction and
 - its j -direction is in the $+i$ -direction



Another technique would be to just describe the neighboring face number for each block face

Virtual (or Local) Space

- **When blocks are partitioned to processors, virtual (local) block numbers must be maintained in addition with global block numbers**
 - For any given processor, the blocks will be looped from 1 to the number of blocks on that processor (ie virtual (local) space)
- **A mapping scheme must be created that can be used to determine the global block number for each virtual (local) block on a given processor**

Example of Virtual Block Information

- Consider this example of an 8x7 blocks divided up into 6 processors. The PLB would be 56/6 or approximately 9 blocks per processor...a difficult decomposition.
- Multi-block grid, temperature, flow, etc files will exist for each processor
- A single global connectivity file could be created, or a virtual (local) connectivity file for each processor could be used
- Mappings between local and global block numbers should be created
- Each block must now keep track of its neighbor block numbers (local and global) and processor number

49 1							
25 1			28 1				
1 1			4 1			7 1	

Virtual block numbers

global block numbers

For each processor, blocks must be looped over local block numbers

Example of General Connectivity Files

- **For *general* configurations on parallel processors, a connectivity file could keep track of:**
 - Global block number, local block number, processor number
 - Orientation, OR
 - Number of south (face-1) sub-faces
 - Start index of south sub-face, end index of south sub-face, neighbor global block number, neighbor local block number, neighbor processor number, neighbor face number, neighbor sub-face start index, neighbor sub-face end index
 - Number of north (face-2) sub-faces
 - Start index of north sub-face, end index of north sub-face, neighbor global block number, neighbor local block number, neighbor processor number, neighbor face number, neighbor sub-face start index, neighbor sub-face end index
 - Etc.

Project 4

- **Take the decomposed set of blocks and connectivity file for the 10x10 and 5x4 block decompositions that you created under project-2, and write a code that will map them onto P processors.**
 - Use what information you know about the grid to maximize data-locality and minimize communication costs
 - You can write your own code or use existing libraries such as Metis
 - I want you to write your own code, however, for this project.
 - In your new connectivity file(s), somehow mark the neighbor information to keep track of neighbor processor numbers and a way to map local (virtual) block numbers to global block numbers

Project 4

- Write out either a single new connectivity file or P connectivity files, multi-block grid, and multi-block initial temperature files with names corresponding to rank number (eg conn.dat.p0, xyz.dat.p0, temp.dat.p0) so that each processor can “read-and-go” in project 5.
 - I have put an example how to write sequential files in “Codes” on smartsite
 - You can combine this code with your project-2 spatial decomposition code if you wish.
 - That way, you can decompose and map in one step.
 - Or you can create a new code just for the parallel/processor mapping.
-
- **The goal is for each processor in the project 5 code to read only it’s information and start iterating! No processor should contain the global information (except possibly upon convergence to agglomerate the blocks back to the global domain for plotting purposes)**

Project 4

- **Due Tuesday, November 17th**
 - Brief description of your project
 - How you decomposed the domain in project 2
 - Methods used to map the decomposed grid onto P processors
 - Listing of parallel/processor decomposition code
 - Demonstration that your code works for 4 and 6 processors on the 10x10 and 5x4 set of blocks
 - Write out a sample of the connectivity file(s) for the 4 and 6 processor parallel/processor decompositions on both set of blocks
 - Give a measure of the load balance of each processor taking into account the weight of the block given by the number of cells and the weight of the communication given by the number of face nodes

Parallel Performance Analysis

- **Objectives of Parallel Computing**

- Ultimately, in parallel computing, we intend to achieve:
 - Faster execution speed
 - Enable multiple analysis in a fixed amount of time
 - Decrease time necessary to complete one solution
 - Increase the level of modeling of our physical system
 - Enable paradigm shifts in the way that computational science is used.
 - Lower cost
 - Strictly speaking, it is nearly impossible to obtain lower cost when using a parallel computer (parallel processing overhead, additional expense of interconnection network, etc.)
 - Lower cost can be derived from additional benefits that result from the ability to execute a given program in a shorter amount of time (ie lower labor costs)
 - However, we must strive to maintain the cost of parallel computing from departing severely from single processor computations

Objectives of Parallel Computing

- **We would like to achieve these goals using *reasonable* resources:**
 - High parallel efficiency
 - Large computational speedups / scalability
 - Low development cost / investment
 - Efficient memory scalability

Objectives of Parallel Computing

- **In order to achieve these goals we need to understand:**
 - Single-processor performance models and efficient programming techniques
 - Parallel processing performance and bottlenecks
 - Mapping between a specific algorithm and the computational hardware available
- **We have discussed all of these somewhat already. Let's look at these further.**

Objectives of Parallel Computing

- In summary, our objective is to learn about the necessary models to:

⇒ *Guide algorithmic software development by using predictive models of performance*

⇒ *Determine, a priori, the best choice of algorithm to be executed on a parallel computer*

Example

- **Building of the Hadrian's wall in the border between England and Scotland. Commissioned by Emperor Hadrian of Rome in 122 A.D. (73.3 miles in length).**
- **Perfect example of a parallel computing problem.**
- **Large size, embarrassingly parallel, parallel I/O**



Example

- **A large number of legionnaires (processors) can be used at the same time with high efficiency.**
- **Global communication is not required: each legionnaire only needs to talk to the two legionnaires to his right and left.**
- **SPMD (Single Program Multiple Data) strategy is being used: legionnaires are replicated along the wall.**
- **Each legionnaire can retrieve his own mortar and stone (parallel I/O) assuming there is no bottleneck at the supply location.**

Example

- **Consider instead the building of the tower of Babel. Is there any parallelism to be found in this problem? Some, at a floor level, but this problem is inherently *sequential*. Similar observations can be made in the time-accurate integration of PDEs unless you can think cleverly about the problem and reduce it.**
- **Your job as a parallel computing scientist is to rediscover the way of building a tower such that additional areas of parallelism can be found and exploited.**

Single-Processor Performance Factors

- **Factors affecting single-processor performance include:**
 - Mflop rating
 - Vector vs. Cache architectures
 - Memory hierarchy
 - Microprocessor architectural issues (no. of ops/cycle, no. of load/store ops/cycle, pipelining ability, etc.)
 - Careful decomposition and programming issues

Multiple-Processor Performance Factors

- **Things affecting multiple-processor performance in parallel systems include:**
 - CPU scalability
 - Memory scalability
 - Interconnection network
 - Bandwidth and latency issues
 - Problem size and granularity
 - *How many processors can we use efficiently?*

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**

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

- **Execution time:**

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

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

$$V \gg S$$

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:

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

Amdahl's Law - Single Processor

- **Computational performance:**

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

- **Therefore:**

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

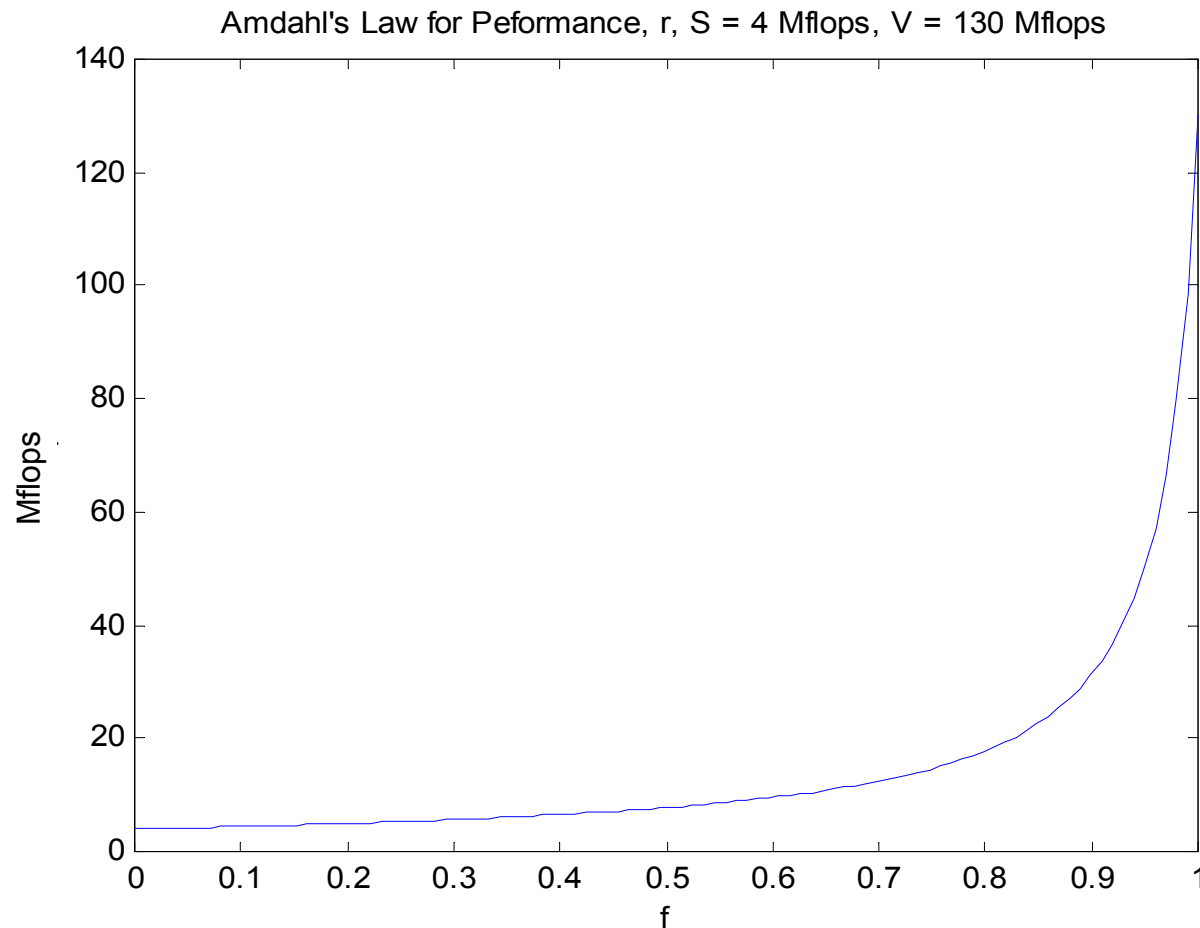
- **If $V \gg S$, then**

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

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!!!

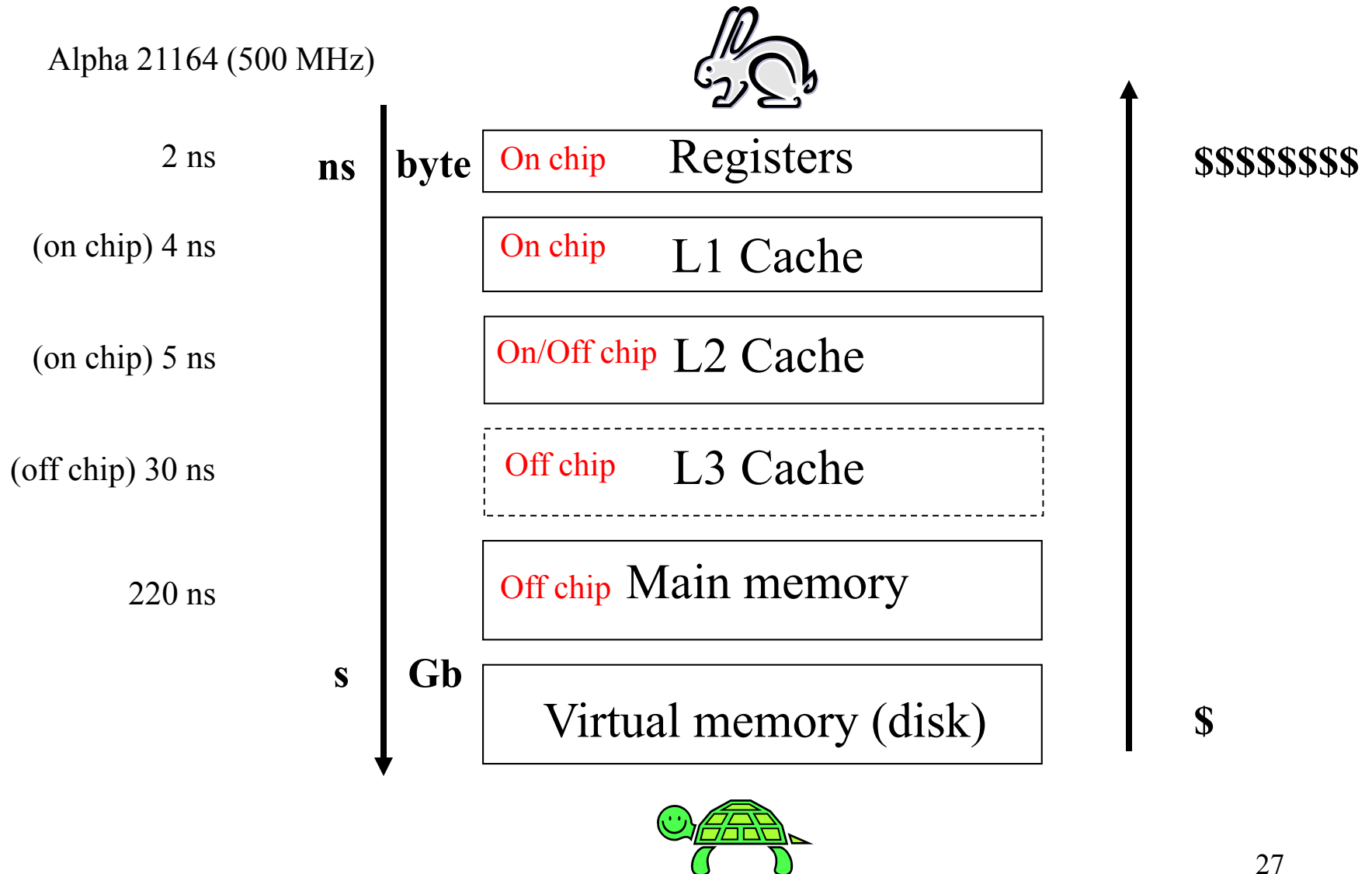
http://www.cecalc.ula.ve/documentacion/tutoriales/O2KPerfTuning/007-3430-003/sgi_html/pr01.html

<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

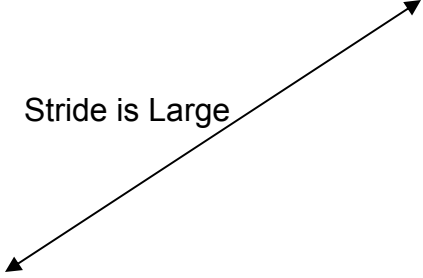
Remember that in Fortran
 $a(i,j)$ is stored as:

- **Stride-1 accesses:**

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
```

Stride is Large



$a(1,1)$

$a(2,1)$

...

$a(1,2)$

$a(2,2)$

...

$a(n,m)$

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

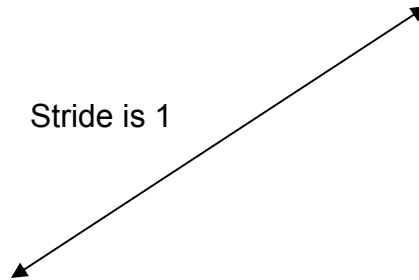
Single Processor Performance

- **Stride-1 accesses:**

Good programming.

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

Stride is 1



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

$a(1,1)$

$a(2,1)$

...

$a(1,2)$

$a(2,2)$

...

$a(n,m)$

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

Single Processor Performance

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

```
D = 0.0
```

Bad programming.

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

```
do i=1,n
```

```
  j = ind(i)
```

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

```
  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. Data locality is crucial.**
- **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

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

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

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

where f is now the fraction of the algorithm that can be carried out in parallel using p processors³⁷

Amdahl's Law - Parallel Processing

- The *speedup* on p processors is then

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

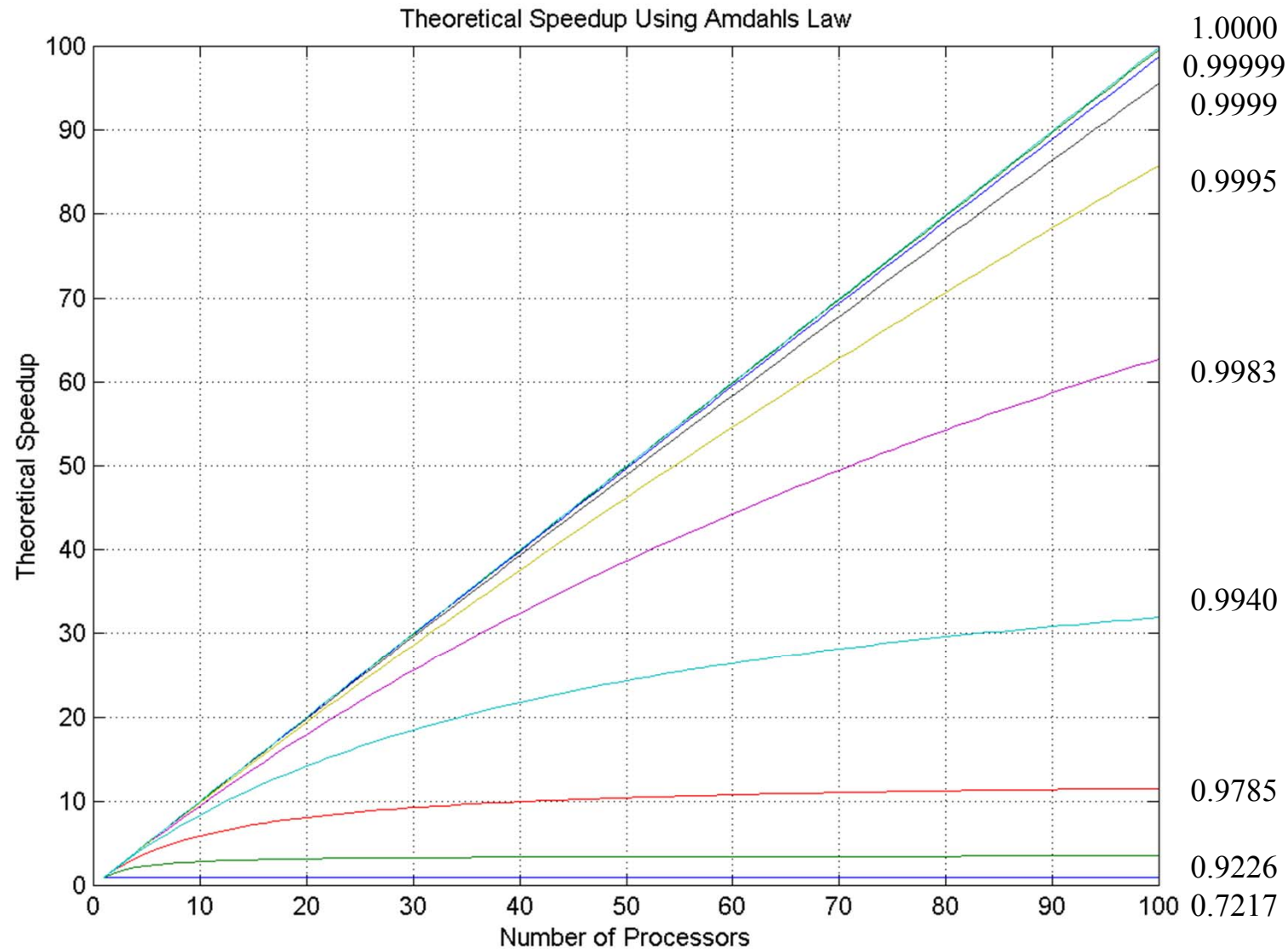
- This is often called **Ware's Law**
- This shows that the speedup is considerably reduced even for pretty large values of f (close to 95%)
- Example: If $f = 0.8$ and we used 5 processors, the theoretical speedup would be 2.8! This formula drives us to achieve a high level of parallelism in our algorithms

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

f values:



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