

COMP 364 / 464 High Performance Computing

Introduction to High Performance and Parallel Computing (cont'd)

COMP 364/464: High Performance Computing

This topic is a first introduction to parallel computing. It addresses basic terminology and concepts appropriate for research scientists and programmers transitioning to parallel machines for the first time.

Certainly much of this introduction is terminology. But we do try to emphasize a few key ideas:

The difference between shared and distributed memory

An intuitive feel for the big picture concepts behind Amdahl's (and Gustafson's) Law

How parallel programming depends on identifying independent sub-tasks

Outline

- Parallelism
- Theoretical background
- Parallel computing systems
- Parallel programming models
- MPI/OpenMP examples

Parallelism: The Basic Idea

- Spread operations over many processors or instruction streams.
- If n operations take time t on 1 processor,
- Does this become t/p on p processors ($p \leq n$)?

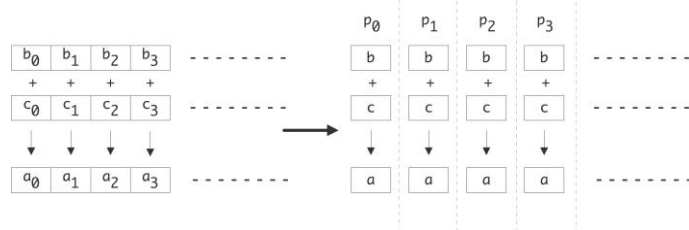
```
for (i = 0; i < n; ++i)
    a[i] = b[i] + c[i];
```

Idealized version:
every core has one
array element

In this (unrealistic) idealized scenario, every processor does the calculation associated with a single array index: a canonical "single instruction multiple data (SIMD)" calculation. Of course there's probably not enough work here to justify splitting the work across multiple processors at all. This kind of loop is instead a prime candidate for vectorization.

Use `++i` ... not `i++` ... it can make a difference in reality.

The Basic Idea (Idealized Version)



The Basic Idea

- Spread operations over many processors
- If n operations take time t on 1 processor...
- ...does this become t/p on p processors ($p \leq n$)?

```
for (i = 0; i < n; ++i)
    a[i] = b[i] + c[i];
```

Idealized version:
every process has one
array element

Slightly less ideal: each core has part of the array

```
for (i = my_begin; i < my_end; ++i)
    a[i] = b[i] + c[i];
```

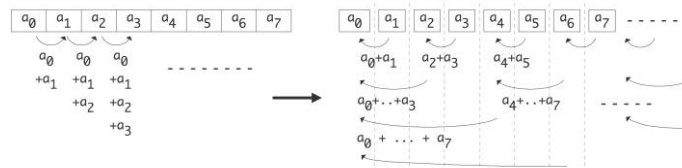
The Basic Idea (...)

- Spread operations over many processors
- If n operations take time t on 1 processor...
- ...does this become t/p on p processors ($p \leq n$)?

```
sum = 0.0;  
for (i = 0; i < n; ++i)  
    sum = sum + a[i];
```

Can we do this in parallel?

The Basic Idea (Continued)



Conclusion: N operations can be done with $N/2$ processors, in total time $\log_2(N)$.

Theoretical question: can addition be done faster?

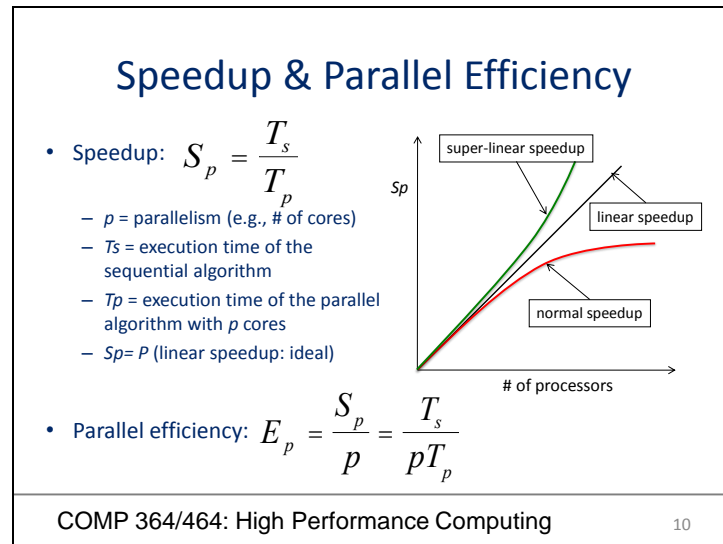
Practical question: can we even do this?

The Basic Idea (...)

- Spread operations over many processors
- If n operations take time t on 1 processor...
- ...does this become t/p on p processors ($p \leq n$)?

```
for (k = 0; k < log2(n); ++k) {  
    s = pow(2,k); /* stride = 2^k */  
    p = 2*s; /* increment = 2*s */  
    for (i = 0; i < n; i += p)  
        a[i] = a[i] + a[i+s];  
}
```


THEORETICAL BACKGROUND



Let's do this with words: if we recruit 4 workers to mow the lawn, we'd hope that working together they could do the job 4 times as fast as 1 worker could do it. It's unlikely that we'd achieve super-linear speedup (e.g. they are able to do it 6 times as fast), and realistically we'd expect something short of linear speedup. See comments on the next few slides.

See first slide with Tractors ...

Limits of Parallel Computing

- Theoretical Upper Limits
 - Amdahl's Law
- Practical Limits
 - Load balancing
 - Non-computational sections
- Other Considerations
 - Time to re-write code

The coding time should be a serious question. 10x speed-up for a problem that you only run once is not worth the effort unless the solution takes >> than 10x as long as the coding.

Amdahl's Law

- All parallel programs contain parallel sections and serial sections
- Serial sections limit the parallel effectiveness (efficiency)
- Amdahl's Law states this formally
 - Effect of parallelism on speed up

$$S_p \leq \frac{T_s}{T_p} = \frac{1}{f_s + \frac{f_p}{P}} \rightarrow \frac{1}{f_s}, P \rightarrow \infty$$

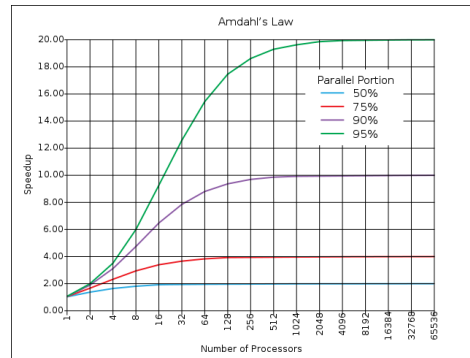
where

- f_s = serial fraction of code
- f_p = parallel fraction of code
- P = number of processors

Suppose you gather a collection of cooks (processors) to bake a cake. Suppose it takes 60 minutes to prepare the ingredients and 45 minutes to bake the cake. Then you can never do better than 45 minutes, no matter how many cooks you have. The computation time is always bound by the serial portion of the work -- the part that cannot be done in parallel. You would achieve a time of 45 minutes if you were able to accomplish the parallel portion of the work (preparing the ingredients) infinitely fast.

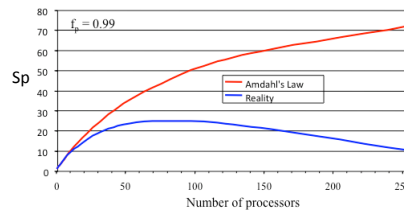
But you could bake 2 cakes with 2 cooks and 2 ovens in the same amount of time. This is known as ‘weak’ scaling.

Amdahl's Law



Practical Limits: Amdahl's Law vs. Reality

- In reality, the situation is even worse than predicted by Amdahl's Law due to:
 - Load balancing (waiting)
 - Scheduling (shared processors or memory)
 - Cost of Communication
 - I/O



If there are 10 or 15 workers trying to work together to mow a lawn, we'd expect inefficiencies as they get in each other's way. We'd also expect limits to how many workers we could employ on a given lawn; eventually we'd get to the point where more workers (processors) would actually make things worse.

Gustafson's Law

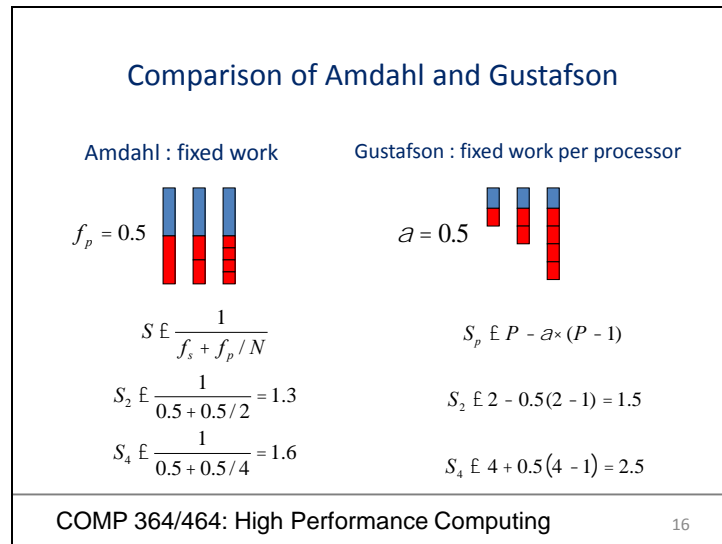
- Effect of multiple processors on run time of a problem with a *fixed amount of parallel work per processor*.

$$S_p \propto P - \alpha \times (P - 1)$$

- α is the fraction of non-parallelized code where the parallel work per processor is fixed (not the same as f_p from Amdahl's)
- P is the number of processors

Amdahl's Law tells us what to expect we try to solve a given problem with an increasing number of processors (strong scaling). Gustafson's Law tells us what to expect as we try to solve bigger problems with more processors.

Conceptually, Gustafson's law says that if the workload can be increased in conjunction with the resources, we can solve a larger problem in equal time.



We rarely speak of ‘speed-up’ when we don’t hold the problem size fixed. We think of speed and time as correlated. Gustafson’s law holds time (approximately) fixed so speed-up is a stretched.

Scaling: Strong vs. Weak

- We want to know how quickly we can complete analysis on a particular problem size by increasing the core (processing element – PE) count
 - Amdahl's Law
 - Known as "strong scaling"
- We want to know if we can analyze a larger problem in approximately the same amount of time by increasing the PE count
 - Gustafson's Law
 - Known as "weak scaling"

Think of the cake example. Strong scaling would increase the # of chefs preparing one cake ... and always limited by the 45 baking time. But you could bake 10 cakes with 10 chefs in 10 ovens in the same amount of time it took to do just one ... assuming you have the resources. This is weak scaling.

PARALLEL SYSTEMS

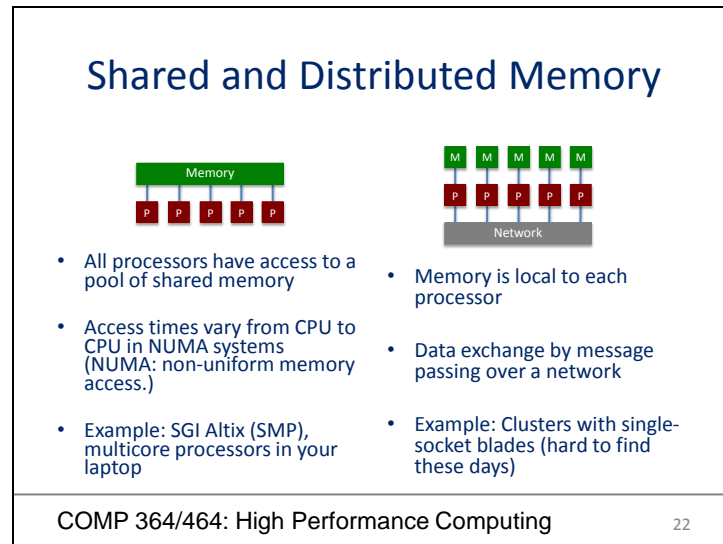
Classification #1: Instruction Streams

Hardware Classification		
SISD	Single Instruction / Single Data	von Neumann 1-instruction-at-a-time
SIMD	Single Instruction / Multiple Data	Array processors, vector pipelines, SSE/AVX instructions
MIMD	Multiple Instruction / Multiple Data	Every processor has its own data and instruction stream
SPMD	Single Program / Multiple Data	Like MIMD, but all the same executable
SIMT	Single Instruction / Multiple Thread	Like SIMD, but not entirely synchronized: GPUs

SIMD: Every processor does the same work (runs the same program, completes the same operations, and/or applies the same workflow) against its own data. Where's the line that defines where SIMD ends and other models begin? Answer: it's just a model. It's not a certification that gets stamped on your hardware or software. Use the model to the extent that it helps explain or understand what's going on!

SIMT -- the "T" stands for "Threads". In fact, SIMD, SPMD, SIMT illustrate the understandable attempt to distinguish among different ways of running the same instructions against different data. Some would say this is unnecessary; perhaps it's enough to interpret the "I" in SIMD more broadly.

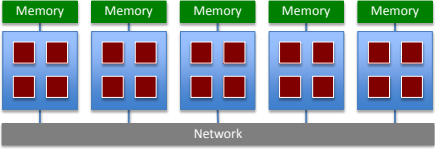
Classification #2: Memory Model



This is arguably the most important concept in this lesson. Eventually students must understand clearly that OpenMP is a mechanism for programming shared memory architectures, and MPI is a mechanism for programming distributed memory architectures.

A metaphor: working on a jigsaw puzzle. Imagine six people sitting around a table, with the puzzle pieces in a bowl in the center of the table. Everybody has access to all the pieces (information, memory). This is shared memory. The challenge here is making sure people don't get in each other's way. Imagine instead that everybody has a paper cup with his or her own pieces to the puzzle. This is distributed memory -- I can only see the information that is present in my own memory (cup). The challenge here is communication -- sending pieces (or information about the pieces) to the people who need them (to say nothing of the problem of figuring out who needs what!).

Hybrid systems



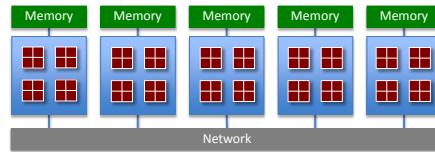
The diagram illustrates a hybrid system architecture. It consists of five nodes, each represented by a blue square containing four red squares (processors). Above each node is a green rectangle labeled 'Memory'. All five nodes are connected to a common horizontal grey bar at the bottom labeled 'Network'.

- A limited number, N , of processors have access to a common pool of shared memory
- To use more than N processors requires data exchange over a network
- Example: Cluster with multi-socket blades

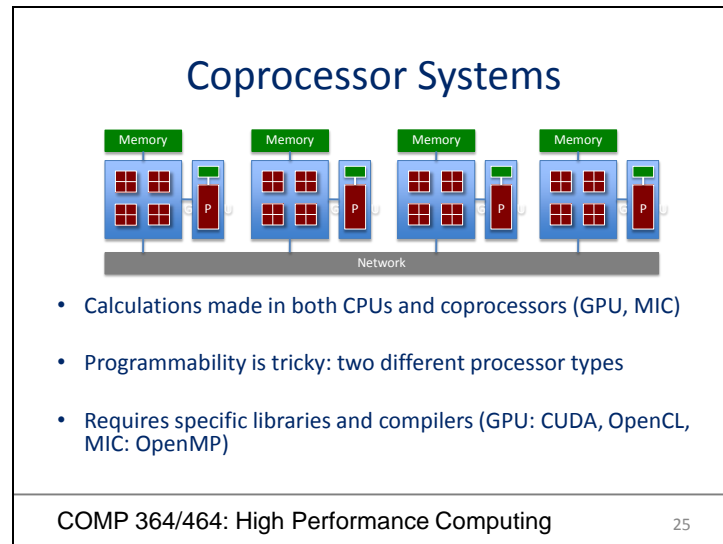
COMP 364/464: High Performance Computing 23

The typical situation: each node of a large cluster like Stampede is a shared memory computer. But the memory on one node is unavailable to other nodes. We will sometimes program as if the entire cluster were a distributed memory computer (we do this when we assign one MPI task to each core). There are those who work on programming models that allow us to think of the entire supercomputer as a shared memory machine. But one cannot do this with OpenMP, and students have a hard time seeing this: one cannot run an OpenMP program on more than one shared memory node!

Multi-core / Multi-processor Systems



- Extension of hybrid model
- Communication details increasingly complex
 - Cache access
 - Main memory access
 - Quick Path / Hyper Transport socket connections
 - Node to node connection via network



GPU = Graphics Processing Unit, used for computation rather than display.

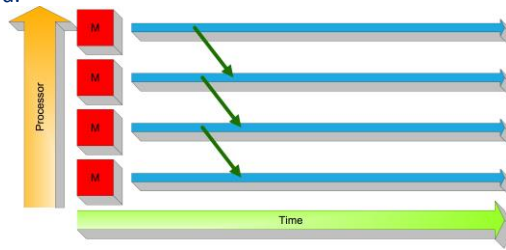
MIC (pronounced "Mike") = Many Integrated Core, Intel's term for the architecture behind the Xeon Phi coprocessor that is the innovative component in TACC's Stampede supercomputer. See for example https://en.wikipedia.org/wiki/Intel_MIC.

GPU vs. MIC -- to a large extent these are two competing coprocessor technologies.

Classification #3: Process Dynamism

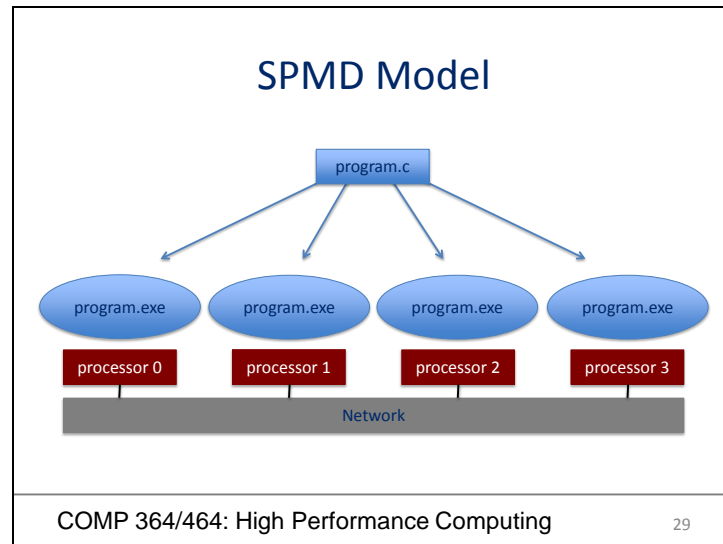
“Process-based” Parallelism

- MIMD & SPMD: one process per processor/core, lives for the life of the run
- Great for distributed memory: task creation and migration is hard.



Single Program Multiple Data

- SPMD: dominant programming model for shared and distributed memory machines.
 - One source code is written
 - Code can have conditional execution based on which processor is executing the copy
 - All copies of code start simultaneously and communicate and sync with each other periodically via messages
- MPMD: not often used (climate models, multi-physics engineering models)
 - Mostly loosely coupled programs: e.g., climate models combine concurrent ocean, atmospheric, and land models and only communication occasionally.

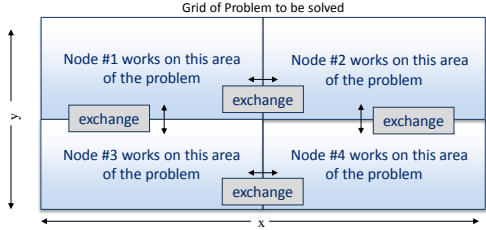


This is the essence of the model underlying MPI programming. The runtime environment is sending a copy of the binary to each processor, not a copy of the source code.

Help students to understand that, under the MPI model, every participating processor or cores runs its own copy of the same binary. That doesn't mean the binary behaves the same on every process: instead, the program engages in what might be called "rank-aware activity". The simplest example: we often code MPI with this type of logic: "if rank is 0 do such and such and send a signal to everyone when finished; else wait for a signal from rank 0 that it's time to move on."

Data Decomposition

- For distributed memory systems, we often decompose a problem (data domain) into smaller pieces and each PE is responsible for one subdomain.
- In physics codes, we usually decomposed spatially
 - Each node works on its section of the problem.
 - Nodes can exchange information, the less the better.



Grid of Problem to be solved

Node #1 works on this area of the problem Node #2 works on this area of the problem

Node #3 works on this area of the problem Node #4 works on this area of the problem

exchange exchange exchange exchange

x y

COMP 364/464: High Performance Computing 30

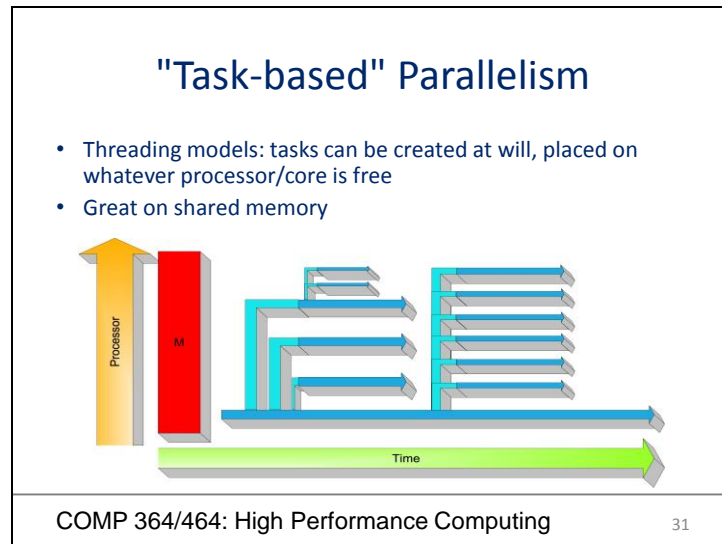
Example is N-body particle dynamics like astrophysics or molecular dynamics. This requires a lot of communication due to All-to-All computation.

Engineering computations for fluid mechanics use grids to model fluid. Partition the grid across processors.

Image processing is similar. Partition a sub-region set of pixel to each processor and apply analysis locally.

Often need overlapping halo. Image filtering good example.

Hadoop (et al.) work in similar fashion. Carve up the data space into independent pieces and apply the same analysis on each (Map). At the end, gather the results (Reduce).



When we speak of threads (= lightweight processes, generally but not always spawned by a single executable), we're normally talking shared memory.

Great model for thing the workload can vary in time and simple decomposition isn't plausible.

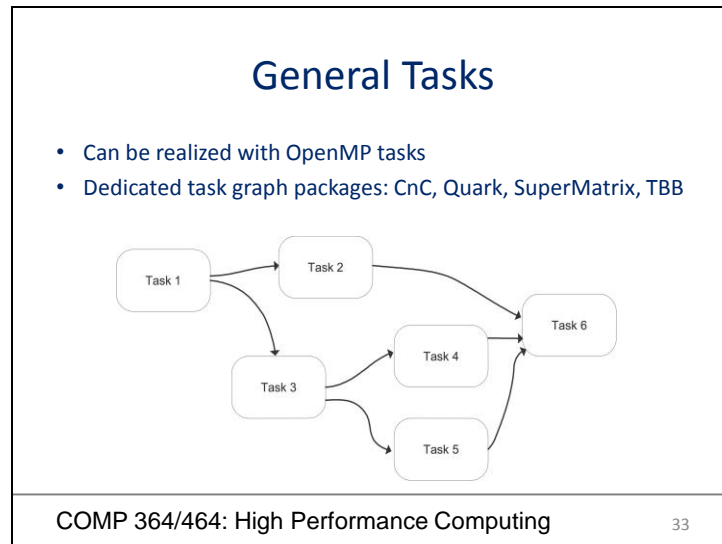
Dynamic Thread Creation

- Old: pthreads
- Newer: Cilk+ (Intel), OpenMP (open standard), Java Threads

```
int sum = 0; /* Global!!! */
void adder(){ sum = sum+1; }

int main() {
    int i;
    pthread_t threads[NTHREADS];
    for (i=0; i<NTHREADS; i++)
        pthread_create
            (threads+i, NULL, &adder, NULL);
    for (i=0; i<NTHREADS; i++)
        pthread_join(threads[i], NULL);
}
```

```
cilk int fibonacci(int n){
    if (n<2) return 1;
    else {
        int rst=0;
        rst += spawn fibonacci(n-1);
        rst += spawn fibonacci(n-2);
        sync;
        return rst;
    }
}
```

Tasks are self-contained units of work that often pair an operation on a data object. Often associated with OO programming. Tasks can have input dependencies which form a graph: nodes are tasks, edges are dependencies.

OpenMP Example: Parallel Loop

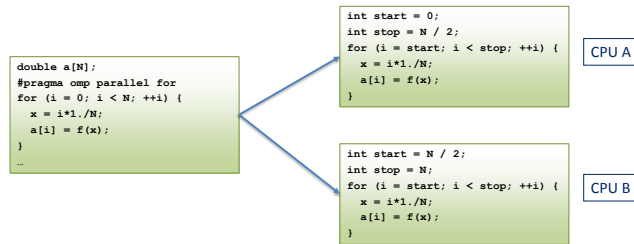
```
#pragma omp parallel for  
for (i = 0; i < N; ++i)  
    b[i] = a[i] + c[i];
```

- Easy parallelism: tasks correspond to loop iterations
- Actually, tasks are *groups* of iterations!
- The directive specifies that (1) a parallel region with K threads is about to be entered and (2) the loop immediately following should be executed in parallel.
- By default, the iteration space is divided into K contiguous chunks of approximately equal size ... N / K .
- For codes that spend the majority of their time executing the content of simple loops, the PARALLEL FOR directive can result in significant parallel performance.
- OpenMP also has a general task mechanism though it's not this easy.

Different World Views

Shared Memory Data Access

- One code will run on 2 CPUs
- Program has array of data to be operated on by 2 CPUs so array is split into two parts.



Distributed Memory Data Access

- Since each CPU has local address space: local indexing only
- Must translate between local and global index space.

CPU A

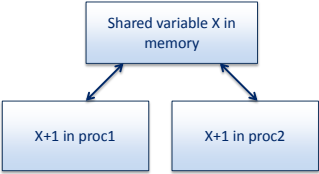
```
int start = 0;
int stop = N / 2;
double a[N/2];
for (i = start; i < stop; ++i) {
    x = i*1.0/N;
    a[i-start] = f(x);
}
```

CPU B

```
int start = N / 2;
int stop = N;
double a[N/2];
for (i = start; i < stop; ++i) {
    x = i*1.0/N;
    a[i-start] = f(x);
}
```

Accessing Shared Variables

- If multiple processors want to write to a shared variable at the same time, there could be conflicts:
 - Process 1 and 2
 - read X
 - compute X+1
 - write X
 - Is the answer X, X+1, X+2?
 - Known as race condition.
- Programmer, language, and/or architecture must provide ways of resolving conflicts



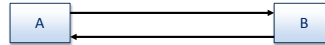
COMP 364/464: High Performance Computing 38

This slide illustrates one of the fundamental issues with shared memory programming: two processes fighting for access to the same memory location. This effects both processes and threads. Processes could come in with cache-coherence and threads are more straightforward race conditions and mutex.

Synchronization is critical to shmem.

Message Passing Communication

- Processes in message passing programs communicate by passing messages



- Basic message passing primitives
 - Send (parameters list)
 - Receive (parameters list)
 - Parameters depend on the library used

MPI: Sends and Receives

- Message Passing Interface (MPI) programs must send and receive data between the processors (communication)
- The most basic calls in MPI (besides the three initialization and one finalization calls) are:
 - MPI_Send
 - MPI_Recv
- These calls are blocking: the source processor issuing the send/receive cannot move to the next statement until the target processor issues the matching receive/send.

Introduce the concept of synchronization across processes. Message based. Blocking?

Final Thoughts

- Systems with multiple shared memory processors are very common for reasons of economics and engineering.
- Going forward, this means that the most practical programming paradigms to learn are
 - Pure MPI
 - OpenMP + MPI (or Threads + MPI) ... MPI+X

On a typical modern cluster with multiple shared memory nodes, there are two dominant programming models: pure MPI (ignore the fact that cores on a single node share memory) or OpenMP+MPI (typically one MPI task per node, and that task does OpenMP/threading on that node. Again, OpenMP is not possible across multiple nodes!

Further reading



- General page:
<http://www.tacc.utexas.edu/~eijkhout/istc/istc.html>
- Direct download:
<http://tinyurl.com/EijkhoutHPC>
- Content presented here is based on content from "Parallel Computing for Science and Engineering" course materials by The Texas Advanced Computing Center, 2013. Available under a Creative Commons Attribution Non-Commercial 3.0 Unported License"

We gratefully acknowledge the sponsorship of Chevron Corporation, whose generous support of TACC has made possible this Scientific Computing Curriculum and other student-focused initiatives.

License

© The University of Texas at Austin, 2013

This work is licensed under the Creative Commons Attribution Non-Commercial 3.0 Unported License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/3.0/>

When attributing this work, please use the following text:
"Parallel Computing for Science and Engineering course materials by The Texas Advanced Computing Center, 2013. Available under a Creative Commons Attribution Non-Commercial 3.0 Unported License"

COMP 364/464: High Performance Computing