Parallel Programming Concepts

Jon Johansson
AICT, University of Alberta

Help!!!

• the people who can help port, compile, debug, and run your parallel programs:

support@westgrid.ca
Serial Programming

- von Neumann architecture executes a program serially (one instruction at a time)
- to speed up we can:
  - optimize the program
  - move to the fastest hardware
  - change the algorithms
- it’s difficult to speed up a well-written program
- the program is generally straightforward to write

It’s Not Easy to Make it Fast

"Sequential programming is really hard, and parallel programming is a step beyond that."

– Andrew Tanenbaum, quoted at the June 2008 Usenix conference

MINIX 3:

http://www.minix3.org/
What and How to Program??

parallel programming environments
OpenMP, MPI, HPF, TBB, Erlang, Shmem, Ct, CUDA, DirectCompute, OpenCL, Portals, ZPL, BSP, CHARM++, Cilk, Co-array Fortran, PVM, Pthreads, Windows threads, Tstreams, GA, Java, UPC, Titanium, Parlog, NESL, Split-C, etc.

parallel processing hardware

x86 Cluster

GPGPU

FPGA

Cell processor

A Big Job in the Kitchen

• what does the chef do when faced with a big job?
• too many meals for one cook to prepare in the given time
• split the work over a team of cooks
A Big Job in the Kitchen

• try to split the work into pieces that can be done by different people
• distribute the work to the team members
• we can organize the workers and resources in a couple of ways

A Big Job in the Kitchen

• put all the chefs in a common environment
• all the resources are available to all the chefs
• large pile of food and utensils
• some resources are limited
• they need to organize sharing
  – take turns with the only knife
A Big Job in the Kitchen

- chefs can communicate with messages on space they can all see

A Big Job in the Kitchen

- give each chef access to his own resources
- everyone has his own set of food, utensils and ovens
- each chef has only enough resources to do his task
A Big Job in the Kitchen

- to communicate use a phone to coordinate finishing the meal at the same time
- appetizer needs to be done and served before main course, before dessert

Dividing the Tasks

- some of the work can be done in parallel
- some work can only be done serially
- some work needs to be completed before other work can continue
- consider a team of chefs with one person to clear and serve
Why Parallel Programming?

- would like to use a number of processors at the same time to speed up calculations
- the problem must be broken into parts that can be solved concurrently
- each part of the problem becomes a program to run on its own processor
- programming becomes more complex

Why Parallel Programming?

- if we can use many CPUs efficiently, we can
  - run simulations faster
  - increase problem sizes
  - run simulations at greater accuracy
- run a program on a cpu that can provide 1 gigaflop/s ($10^9$ flop/s)
- if you need 1 teraflop/s ($10^{12}$ flop/s) to finish the calculation in a reasonable time you can use 1000 cpus
  - you need to use them efficiently!
Terminology

• **parallel computing** – harnessing a bunch of processors on the same machine to run your computer program
• note that this is one machine
• generally a homogeneous architecture
  – same processors, memory, operating system
• all the machines in the Top 500 are in this category

Terminology

• cluster:
  – a set of generally homogeneous machines
  – originally built using low-cost commodity hardware
    • to increase density, clusters are now commonly built with 1-u rack servers or blades
  – can use standard network interconnect or high performance interconnect such as Infiniband or Myrinet
    • cluster hardware is becoming quite specialized
  – thought of as a single machine with a name, e.g. “checkers” – checkers.westgrid.ca
Types of Parallelism

- **Task parallelism**
  - Different programs operate on copies of the same data
  - *functional decomposition*

- **Data parallelism**
  - Copies of the same program operate on different data
  - *domain decomposition*

**Different programs operate on copies of the same data**

- Consider 3 tasks that take equal times to complete: T₁, T₂, T₃
- And a bunch of data to run the tasks on
- We break the data into 9 equal sized pieces
  - Each task operating on a chunk of data takes 1 unit of time

- There are 27 time units of work to be done in this situation
- For example:
  - T₁ → multiply by 5
  - T₂ → apply sin(x)
  - T₃ → take square root
Types of Parallelism

Task parallelism

Data parallelism

- either tasks are concurrent or data is crunched concurrently
- processing time is now 9 time units

Pipelining

- we can mix the parallelism
- a task processes some data and then passes the data on to the next task
- total time to process all the data is 11 time units
Domain Decomposition

- data can be decomposed in different ways
- depends on how you want to schedule the work
- finer grain may make load balancing easier
- depends on the problem

Processes and threads

- A process is created by the operating system, and requires a fair amount of "overhead".
- Processes contain information about program resources and program execution state, including:
  - Process ID, process group ID, user ID, and group ID
  - Environment
  - Program instructions
  - Registers, Stack, Heap
  - Inter-process communication tools (such as message queues, pipes, semaphores, or shared memory).
- a thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system
A serial program runs a single thread

- a computer program exists as an executable file on your computer
- a **process** is an instance of a computer program that is being sequentially executed by a computer system that has the ability to run several computer programs concurrently.

Parallel Processing Categories

- we can broadly classify parallel processing into these categories:
  - **Thread Parallelism** – use a single process with multiple threads to perform a set of tasks
    - OpenMP, Pthreads (*shared memory for communication*)
  - **Process Parallelism** – use more than one process to perform a set of tasks
    - MPI (*message passing for communication*)
  - **Hybrid Parallelism** – use multiple processes, some of which can have multiple threads
    - MPI + OpenMP
Operating systems

- an Operating System (OS) is an interface between the computer hardware and the user
- the operating system manages and coordinates the sharing of the computer’s resources

<table>
<thead>
<tr>
<th>Tools and Applications (ls, grep, env, pine)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shell (cd, setenv, history)</td>
</tr>
<tr>
<td>Kernel</td>
</tr>
<tr>
<td>Hardware</td>
</tr>
</tbody>
</table>

Architectures

- Symmetric Multiprocessing (SMP)
  - Uniform Memory Access (UMA)
  - multiple CPUs, residing in one cabinet, share the same memory
  - processors and memory are tightly coupled
  - the processors share memory and the I/O bus or data path

Memory

Memory Bus

CPU CPU CPU
Architectures

• SMP
  – a single copy of the operating system is in charge of all the processors
  – SMP systems range from two to as many as 32 or more processors
Architectures

• UMA-SMP negatives
  – as the number of CPUs get large the buses become saturated
  – long wires cause latency problems

Architectures

• Non-Uniform Memory Access (NUMA)
  – NUMA is similar to SMP - multiple CPUs share a single memory space
    • hardware support for shared memory
  – memory is separated into close and distant banks
    • basically a cluster of SMPS
  – memory on the same processor board as the CPU (local memory) is accessed faster than memory on other processor boards (shared memory)
    • hence "non-uniform"
  – NUMA architecture scales much better to higher numbers of CPUs than SMP
Architectures

- Cache Coherent NUMA (ccNUMA)
- each CPU has an associated cache
- ccNUMA machines use special-purpose hardware to maintain cache coherence
  - typically done by using inter-processor communication between cache controllers to keep a consistent memory image when the same memory location is stored in more than one cache
  - ccNUMA performs poorly when multiple processors attempt to access the same memory area in rapid succession

Distributed Memory Multiprocessor (DMMP)
- each computer has its own memory address space
- looks like NUMA but there is no hardware support for remote memory access
  - the special purpose switched network is replaced by a general purpose network such as Ethernet or more specialized interconnects:
    - Infiniband
    - Myrinet
- all computer clusters look like this now
  - Lattice was decommissioned on Nov. 3, 2009

Lattice: Calgary’s HP ES40 and ES45 cluster – each node has 4 processors
Architectures

- Massively Parallel Processing (MPP) Cluster of commodity PCs
  - processors and memory are loosely coupled
  - "capacity computing"
  - each CPU contains its own memory and copy of the operating system and application.
  - each subsystem communicates with the others via a high-speed interconnect.
  - in order to use MPP effectively, a problem must be breakable into pieces that can all be solved simultaneously
Today’s Common System Architecture

Architectures

• lots of “how to build a cluster” tutorials on the web – just Google:
  • http://www.beowulf.org/
  • http://www.cacr.caltech.edu/beowulf/tutorial/building.html
Parallel Programming

• need to do something to your program to use multiple processors
• need to incorporate commands into your program which allow multiple threads to run
• one thread per processor
• each thread gets a piece of the work
• several ways (APIs) to do this ...

OpenMP

• Open Multi-Processing (OpenMP)
• introduce statements into your code that look like comments to a serial compiler
  – in C:          #pragma
  – in FORTRAN:   C$OMP or !$OMP
• can compile serial and parallel executables from the same source code
• restricted to shared memory machines
  – not clusters!
• www.openmp.org
Threads

• a thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system
  – this means that executing threads can be started and stopped by the OS
  – threads can move from one CPU to another while executing
• a process can contain many threads

How to See The Threads?

• Windows Task Manager on a 4-core system
• “spinparallel.exe”
• OMP_NUM_THREADS=2
• OMP_NUM_THREADS=3
• OMP_NUM_THREADS=4
How to See The Threads?

- use “Top” on the cluster: “spinparallel”
- toggle views with “shift-h” i.e. “H”

To see threads on the Cluster

- use the command:

  ```bash
  ps -elf | grep <your_program_name>
  ```
OpenMP Threads

- OpenMP uses the “fork-join” model of parallel execution
- all OpenMP programs begin as a single process: the master thread
- when the master thread encounters a parallel region it creates a team of parallel threads

OpenMP Threads

- to create a parallel region in OpenMP use the pragma (in C):
  ```c
  #pragma omp parallel
  {
    // code here
  }
  ```
- the block of code in the parallel region is run by all threads
Compile and Run

- to compile your program with OpenMP using gfortran
  
gfortran -fopenmp -o prog prog.f90

- to run with N threads set the environment variable OMP_NUM_THREADS
  
export OMP_NUM_THREADS=2

- you might need to set the following
  
export OMP_DYNAMIC=false

---

Activity 2 - A Parallel Program

- add OMP directives to the serial program
- they look like comments to the compiler unless compiling for OpenMP
- compile with:
  
gfortran -fopenmp -o spinparallel spinparallel.f90

Program SpinParallel
Implicit None
!$OMP Parallel
Do While (.true. )
  ! Spinning my wheels
End Do
!$OMP End Parallel
End Program SpinParallel
Activity 2 - A Parallel Program

- run the program with 1, 2 and 4 threads

`export OMP_NUM_THREADS=4`

- observe the threads with both top and the ps command

```
Program SpinParallel
Implicit None
!$OMP Parallel
Do While(.true.)
  ! Spinning my wheels
End Do
!$OMP End Parallel
End Program SpinParallel
```

Activity 3 - A Parallel Program

- change the infinite loop to a print statement
- compile and run serial and parallel versions of the program with 1, 2 and 4 threads

`export OMP_NUM_THREADS=4`

```
Program HelloWorld
Implicit None
!$OMP Parallel
Print*, "Hello World"
!$OMP End Parallel
End Program HelloWorld
```
Posix Threads (pthreads)

- Posix threads provide a standardized programming interface for the creation of threads on a Unix system
- creating and managing a process requires much more operating system overhead than thread creation and management
- the work is packaged in a function which is passed to the thread when it is created

Thread Creation is Relatively Fast

<table>
<thead>
<tr>
<th>Platform</th>
<th>fork() Real Time (= Wall time??)</th>
<th>pthread_create() Real Time (= Wall time??)</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD 2.4 GHz Opteron (8cpus/node)</td>
<td>41.07</td>
<td>0.66</td>
<td>62</td>
</tr>
<tr>
<td>IBM 1.9 GHz POWER5 p5-575 (8cpus/node)</td>
<td>64.24</td>
<td>1.75</td>
<td>37</td>
</tr>
<tr>
<td>IBM 1.5 GHz POWER4 (8cpus/node)</td>
<td>104.05</td>
<td>2.01</td>
<td>52</td>
</tr>
<tr>
<td>INTEL 2.4 GHz Xeon (2 cpus/node)</td>
<td>54.95</td>
<td>1.64</td>
<td>34</td>
</tr>
<tr>
<td>INTEL 1.4 GHz Itanium2 (4 cpus/node)</td>
<td>54.54</td>
<td>2.03</td>
<td>27</td>
</tr>
</tbody>
</table>

Timings reflect 50,000 process/thread creations, were performed with the time utility, and units are in seconds, no optimization flags
Thread Creation

- **pthread_create()** arguments:
  - **thread**: a unique identifier for the new thread returned by the subroutine
  - **attr**: an attribute object that may be used to set thread attributes
    - you can specify a thread attributes object, or NULL for the default values
  - **start_routine**: the C routine that the thread will execute once it is created.
  - **arg**: a single argument that may be passed to `start_routine()`
    - It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.

Posix Threads

- your **main()** program is the single, default first thread
- all other threads must be explicitly created by the programmer
- after a thread is created, the OS schedules when it will execute
Threads

• we have a bunch of threads doing the same thing
• we need to split the work between threads
• each thread in a team has a unique identifier
• get the thread id with the OMP runtime function

\[Omp\_Get\_Thread\_Num()\]
• the function returns an integer

---

Activity 4 – Thread IDs

• use the runtime function

\[omp\_get\_thread\_num()\]

• change the print statement to include the output of the function
• run the program with 1, 2, 4 and 16 threads

export OMP_NUM_THREADS=4
Sharing the Work

• we now have enough to split the work between threads
• use a “Select-Case” structure to control the work that a thread does
• this construct allows both task and data parallelism

\[
\text{thrld} = \text{omp_get_thread_num()}
\]

\[
\text{Select Case}(\text{thrd})
\]

\[
\text{Case(0)} \quad ! \text{The master thread}
\text{Put work code here}
\]

\[
\text{Case(1)} \quad ! \text{Team member}
\text{Put work code here}
\]

\[
\text{Case Default}
\text{Default work}
\]

\[
\text{End Select}
\]

Activity 5(a) - Sharing the Work

• write serial program that does the following tasks and prints the results:
  - \(6 + 5 = ?\)
  - \(16 / 4 = ?\)
  - Print “I am a computer”
  - Print “Boo!”
• use a “Select-Case” structure to control the work that a thread does
Activity 5(b) - Sharing the Work

• modify your serial program by placing the tasks in a parallel section and run the program with 4 threads
  – how many results are printed?
• use a “Select-Case” structure to control the work that a thread does and split the work between 4 threads
• run the program with 4 threads
• run the program with 5 threads, does it behave sensibly? (Use the “Case Default”)

OpenMP

• compiler directive based; can use serial code
• jointly defined and endorsed by a group of major computer hardware and software vendors
  – OpenMP Fortran API was released October 28, 1997
  – C/C++ API was released in late 1998
• portable/multi-platform, including Unix and Windows NT platforms
• can be very easy and simple to use – provides for "incremental parallelism"
• www.openmp.org
OpenMP Directives

- free format Fortran files can contain only one OMP sentinel (the text string that begins a directive)
  
  !$OMP
- the directive has the form
  
  !$OMP directive-name optional_clauses...
- a directive may appear anywhere on a line
  - preceded only by whitespace
  - an ampersand (&) at the end of the line identifies a continued line.
- Fortran comments may appear on the directive line, initiated by an exclamation mark (!)
  - the rest of the line is ignored

Conditional Compilation

- the OpenMP Fortran API accepts a conditional compilation sentinel
- for free format source files this is
  
  !$
- this sentinel can appear in any column, preceded only by white space
  - must appear as a single word
  - Fortran free format conventions apply to the rest of the line

  !$ iam = OMP_GET_THREAD_NUM() + &

  !$ index
- with OpenMP compilation enabled, the sentinel is replaced by two blanks
Conditional Compilation

• the following shows the most straightforward and common use of the conditional compilation sentinel in F95

\[
\text{integer :: Thrd\_id}
\]

\[
\ldots
\]

\[
\text{Thrd\_id} = 0
\]

\[
!$ \text{Thrd\_id} = \text{omp\_get\_thread\_num()}
\]

• the OMP runtime function is only executed if the file is compiled with OpenMP enabled

Message Passing Interface (MPI)

• a standard set of communication subroutine libraries
  – works for SMPs and clusters
• programs written with MPI are highly portable
• information and downloads
MPI

• supports the SPMD, single program multiple data model
  – all processors use the same program
  – each processor has its own data
• think of a cluster – each node is getting a copy of the program but running a specific portion of it with its own data

Parallel Programming

• starting mpi jobs is not standard
  – for mpich2 use “mpiexec”
• start a job with 6 processes
• 6 copies of the program run in the default Communicator Group “MPI_COMM_WORLD”
• each process has an ID – its “rank”
Parallel Programming

- example: start N processes to calculate N-1 factorial
  - 0! = 1
  - 1! = 1
  - 2! = 2 x 1 = 2
  - 3! = 3 x 2 x 1 = 6
  - ...
  - n! = n x (n-1) x...x 2 x 1

Parallel Programming

- generally the master process will:
  - send work to other processes
  - receive results from processes that complete
  - send more work to those processes
  - do final calculations
  - output results
- designing an efficient algorithm for all this is up to you
Parallel Programming

- note that getting data through messages can be 10 to 1000 times slower than accessing local main memory
- try to keep the amount of data that you need to send between processes down

the basic structure of an MPI program is what we’ve seen in exercise 5(b)
get the rank of the process
do work allocation based on the rank

Select Case(rank)
Case(0) ! The master process
! Put work code here
Case(1) ! Process 1
! Put work code here
Case Default
! Default work
End Select
Parallel Programming

• it’s possible to combine OpenMP and MPI for running on clusters of SMP machines
• the trick in parallel programming is to keep all the processors
  – working ("load balancing")
  – working on data that no other processor needs to touch (there aren’t any cache conflicts)
• parallel programming is generally harder than serial programming

Loops

• the bulk of parallelizable work in a program typically resides in repetitive sections
• how to split the work in a loop between threads?
• split range of the index between threads

Do i = 1, 1000
R(i) = A(i) + B(i)*C(i)
End Do

Thread 1:
Do i = 1, 500
R(i) = A(i) + B(i)*C(i)
End Do

Thread 2:
Do i = 501, 1000
R(i) = A(i) + B(i)*C(i)
End Do
Activity 6(a) – Sharing Loops

- write a serial program which initializes an array of length 1000
- all array elements have value 1.0
- sum the array elements and print out the result
- use a Select-Case to split the work over 2 threads
- what happens?

```
Real :: a(1000)
a = 1.0
sum = 0.0
Do i = 1, 1000
  sum = sum + a(i)
End Do
Print*, "The sum is: ", sum
```

Activity 6(b) – Sharing Loops

- use a Select-Case to split the work over 2 threads
- what happens?
- any ideas what might be going on?

```
!$OMP Parallel
thrdId = omp_get_thread_num()
Select Case(thrdId)
  Case(0) ! The master thread
    Do i = 1, 500
      sum = sum + a(i)
    End Do
    Write(*, "(F12.5)") sum
  Case(1) ! Team member
    Do i = 501, 1000
      Case Default
      Write(*, "(I2)") thrdId
    End Select
  End Select
End Select!
$OMP End Parallel
```
Shared Memory

- a process has access to memory
- the threads that run as part of that process have access to that memory

Speedup

- how can we measure how much faster our program runs when using more than one processor?
- define Speedup $S$ as:
  - the ratio of 2 program execution times
  - constant problem size
- $T_1$ is the execution time for the problem on a single processor (use the “best” serial time)
- $T_P$ is the execution time for the problem on $P$ processors

$$S = \frac{T_1}{T_P}$$
Speedup

- **Linear speedup**
  - the time to execute the problem decreases by the number of processors
  - if a job requires 1 week with 1 processor it will take less than 10 minutes with 1024 processors

- **Sublinear speedup**
  - the usual case
  - there are generally some limitations to the amount of speedup that you get
  - communication
Speedup

• **Superlinear speedup**
  – very rare
  – memory access patterns may allow this for some algorithms

---

Speedup

• why do a speedup test?
• it’s hard to tell how a program will behave
• e.g.
  – “Strange” is actually fairly common behaviour for untuned code
  – in this case:
    • linear speedup to ~10 cpus
    • after 24 cpus speedup is starting to decrease
Speedup

• to use more processors efficiently change this behaviour
  – change loop structure
  – adjust algorithms
  – ??
• run jobs with 10-20 processors so the machines are used efficiently

![Graph showing Speedup vs Number of CPUs]

Copyright 2009, University of Alberta Parallel Programming, Nov. 4, 2009

Speedup

• one class of jobs that have linear speed up are called “embarrassingly parallel”
  – a better name might be “perfectly” parallel
• doesn’t take much effort to turn the problem into a bunch of parts that can be run in parallel:
  – parameter searches
  – rendering the frames in a computer animation
  – brute force searches in cryptography
Speedup

we have been discussing Strong Scaling
  - the problem size is fixed and we increase the number of processors
    • decrease computational time (Amdahl Scaling)
  - the amount of work available to each processor decreases as the number of processors increases
  - eventually, the processors are doing more communication than number crunching and the speedup curve flattens
  - difficult to have high efficiency for large numbers of processors

we are often interested in Weak Scaling
  - double the problem size when we double the number of processors
    • constant computational time (Gustafson scaling)
  - the amount of work for each processor has stays roughly constant
  - parallel overhead is (hopefully) small compared to the real work the processor does

• e.g. Weather prediction
Amdahl’s Law

- Gene Amdahl: 1967
- parallelize some of the program – some must remain serial
- \( f \) is the fraction of the calculation that is serial
- \( 1-f \) is the fraction of the calculation that is parallel
- the maximum speedup that can be obtained by using \( P \) processors is:

\[
S_{\text{max}} = \frac{1}{f + \frac{(1-f)}{P}}
\]

Amdahl’s Law

- if 25% of the calculation must remain serial
  the best speedup you can obtain is 4
- need to parallelize as much of the program as possible to get the best advantage from multiple processors
Granularity

- a qualitative measure of the ratio between computation and communication or synchronization
- fine-grain: a small amount of work is done before communication is required
- coarse-grain: a large amount of work is done before communication is required

Parallel Errors

- there are two types of errors that occur only in a parallel program:
- **Race Conditions**
  - a result depends on which thread executes a section of code first
  - this leads to unpredictable results
- **Deadlocks**
  - two threads are each waiting for a result from the other
  - no work gets done
Example: Convolution - discrete

- In 2 dimensions the convolution is:
  \[(f * g)_{m,n} = \sum_{i,j=-\infty}^{\infty} f_{i,j} g_{m-i,n-j}\]

- Apply a 3x3 filter to the image

\[g \rightarrow \quad f \rightarrow\]

Convolution - discrete

- For each image point:
  - Multiply the corresponding filter and image values
  - Sum the result
  - Multiply by a normalizing factor if necessary

- For a 3x3 filter each new image point requires 9 multiplies and 8 adds
Convolution - discrete

\[(f * g)_{2,2} = \sum_{i,j=1}^{3} f_{i,j} g_{2-i,2-j}\]

\[= f_{1,1} g_{1,1} + f_{1,2} g_{1,0} + f_{1,3} g_{1,-1}\]
\[+ f_{2,1} g_{0,1} + f_{2,2} g_{0,0} + f_{2,3} g_{0,-1}\]
\[+ f_{3,1} g_{-1,1} + f_{3,2} g_{-1,0} + f_{3,3} g_{-1,-1}\]

Image Padding

• pixels at the edge of the image aren’t surrounded
• convolution operations can’t be done
• need some special handling
• add pixels around the image: “padding”
  – pad with zeros
  – copy the edge row or column as many times as necessary
  – mirror the image
Convolution Calculation – Serial

- the calculation involves 4 nested loops
- two outside loops move over the image
- two inside loops do multiplication and sum for new image point
- image size: 7000x7000
- filter size: 7x7
- Serial times:
  - program: 34.13 sec
  - loops: 29.7 sec
  - serial section: 4.42 sec
  - max speedup = 34.13/4.42 = 7.7

```c
for(i = offset; i < nx + offset; i++) {
    for(j = offset; j < ny + offset; j++) {
        // Operate in each pixel in the image with the filter
        sum = 0.0;
        for(m = 0; m < nf; m++) {
            for(n = 0; n < nf; n++) {
                sum = sum + filter[m][n] * paddedimage[i-offset+m][j-offset+n];
            }
        }
        newimage[i][j] = sum;
    }
}
```

Convolution Calculation - OpenMP

- to create OpenMP threads and tell OpenMP that we are parallelizing a loop we can combine two directives:
  - `#pragma omp parallel`
  - and
  - `#pragma omp for`
- into the directive
  - `#pragma omp parallel for`

```c
#pragma omp parallel for private (i,j,m,n,sum) schedule(dynamic,1) {
    for(i = offset; i < nx + offset; i++) {
        for(j = offset; j < ny + offset; j++) {
            // Operate in each pixel in the image with the filter
            sum = 0.0;
            for(m = 0; m < nf; m++) {
                for(n = 0; n < nf; n++) {
                    sum = sum + filter[m][n] * paddedimage[i-offset+m][j-offset+n];
                }
            }
            newimage[i][j] = sum;
        }
    }
}
```
Convolution Calculation - OpenMP

- **clauses** modify the behaviour of the loop
  ```cpp
  #pragma omp parallel for private (i,j,m,n,sum)
  schedule(dynamic,1)
  {
    for(i = offset; i < nx + offset; i++) {
      for(j = offset; j < ny + offset; j++) {
        // Operate in each pixel in the image with the filter
        sum = 0.0;
        for(m = 0; m < nf; m++) {
          for(n = 0; n < nf; n++) {
            sum = sum + filter[m][n] * padded_image[i-offset+m][j-offset+n];
          }
        }
        newimage[i][j] = sum;
      }
    }
  }
  ```
- each thread has its own copy of these variables
- `schedule(dynamic,1)`
- each thread gets a "chunk" of work of size 1
- when the work is done the thread requests another chunk

---

OpenMP Speedup

Loop parallelizes nicely

Amdahl limit $S = 7.7$

Serial work prevents the program from speeding up well
Gustafson’s Law

- keep the total time of execution fixed
- the serial part of the program is fixed
- increase the parallel work as the number of processors N increases – increase grid size
- work done by each thread is constant

\[ S = N - f(N - 1) \]

f is the serial fraction of the program
S is the speedup

OpenMP - Gustafson's Law, 7000x7000 image

- 16 threads
- 9 threads
- 4 threads
- 1 thread

- 28x28 filter
- 21x21 filter
- 14x14 filter
- 7x7 filter
Convolution Calculation – MPI

- rewrite the program to include the MPI functions
- make decisions about what work to do based on the rank of the process
- the master process gathers the results
- MPI_GATHER is a serial process

```c
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
if (rank == 0) {
    // do something
}
// Get the results
MPI_Gather(...);
MPI_Finalize();
```
MPI_Gather()

- Gathers together values from a group of processes
- This subroutine collects individual messages from each task in comm at the root task and stores them in rank order
- image: 7000x7000
- filter: 7x7
- time for the MPI_Gather()
  - 4 sec for 2 processes
  - 15 sec for 14 processes

Implicit Parallelism in Matlab

- Matlab can use multiple threads running in a single Matlab instance
- speed up element wise computations, or those that use the BLAS library
- set/get the number of threads using maxNumCompThreads
- this is enabled by default as of Matlab 2009
- disable multithreaded computation by starting MATLAB with the option matlab -singleCompThread

Element wise:
>> x = rand(15000, 15000);
>> z = sin(x);

Using BLAS:
>> x = rand(3000, 3000);
>> y = x;
>> x * y;
Explicit Parallelism in Matlab

• Matlab can use multiple processes running on one machine, or a cluster of machines
• requires the Parallel Computing Toolbox
  – one license gives up to 8 worker processes
• use “parfor” to parallelize a loop
• work in the loop must be independent for each iteration

Start 3 worker processes:
```
matlabpool 3
```

Serial work:
```
x = rand(15000, 15000);
for k = 1:15000;
  vec = x(:, k);
  y(:, k) = log(vec + 1.0);
end
```

Parallel work:
```
parfor k = 1:15000;
  ...
end
```
```
matlabpool close
```

Summary

• parallel programming is a bit harder than serial
• threads (OpenMP) vs processes (MPI)
  – each thread/process has a unique identity that it can use to control the work that it does
• you don’t know how efficiently your program uses multiple processors until you do a speedup test
• do a speedup test
• DO a speedup test!
• MPI session in 2 weeks with Masao ...
Resources

- OpenMP docs: http://openmp.org/wp/
- MPICH2 docs: http://www.mcs.anl.gov/research/projects/mpich2/