Intro to
Parallel Programming Models

http://www.oit.duke.edu/scsc
http://wiki.duke.edu/display/scsc
scsc@duke.edu

John Pormann, Ph.D.
jp1@duke.edu

Outline

❖ Types of computational parallelism
  ◆ Job parallelism
  ◆ Iterative parallelism or Domain Decomposition
  ◆ Recursive parallelism or Divide-and-Conquer
  ◆ Pipeline or Producer-Consumer parallelism
  ◆ Client-Server
  ◆ Interacting peers

❖ Programming Languages/Libraries
  ◆ Sequential “Parallel” Programming
  ◆ Shared Memory Programming
  ◆ Distributed Memory Programming
Job Parallelism

- You have a large number of unconnected jobs
  - “Pool of Work” or “Bag of Tasks” model
  - Parameter space studies
  - Lots of (experimental?) data sets, all of which needs to be processed by the same algorithm
  - No communication or interaction is needed between Job-#i and Job-#j

- This is often the most efficient form of parallelism possible!
  - *IF* you can fit the jobs onto individual machines
    - Memory could be a limiting factor
  - *AND* it will still take X hours for one job, it’s just that you will get back 10 or 100 results every X hours
    - Cutting a 15 hour run to down to 15 minutes really can ENABLE new science

Job Parallelism, cont’d
Smaller-scale Job Parallelism

"Fork-Join" Model / Threads

```c
void compute_sum( array ) {
    /* loop/ + values */
}

void compute_sum2( array ) {
    /* loop/ + values^2 */
}

main() {
    /* I/O usually has to be done separately */
    load_data( array );
    /* create two "children" to do the work */
    pid1 = fork_process( compute_sum, array );
    pid2 = fork_process( compute_sum2, array );
    /* wait for children to finish */
    join_process( pid1 );
    join_process( pid2 );
    /* we now have the computed sum/sum^2 */
}
```

Master Process

Child #1
compute_sum

Child #2
compute_sum2

Job Parallelism, Examples

- BLAST, PAUP, Mr.Bayes
  - Same code is run on 100’s or 1000’s of different DNA segments
  - Initial exploration of a function space, Mandelbrot set
- Successive refinement, minimization problems
  - Same function is evaluated at x=0,1,2,3,… and y=0,1,2,3,…
- Data mining
  - Each of 100 jobs looks at a different pair of factors from a common database
- Monte Carlo simulations (there may be better options)
  - Each of 100 jobs uses a different random number seed to explore the input space
- Initial analysis may lead to a more targeted analysis later
  - May combine the results of the 100 independent jobs
Iterative Parallelism / Domain Decomposition

- Iterative parallelism means breaking up a problem where there are large iterative processes or loops
  - Eg. “for” loops in C, “do” loops in Fortran
  - Domain decomposition
  - Large matrix-vector problems

Poisson’s Equation:

```
For I=1 to N
    For J=1 to N
        v_new(I,J) = 0.25*( v(I+1,J)+v(I-1,J)+v(I,J+1)+v(I,J-1) )
    Next J
Next I
```

- If \( N \) is large, then there is a lot of parallel work that can be done
- Note that \( v_{\text{new}}(I,J) \) requires only information at \( v(I\pm1,J\pm1) \)
  - So work on row \( I=1 \) is independent of rows \( I=\{3,4,5,\ldots\} \)

Iterative Parallelism, cont’d

Parallel "splitting" of the domain

Interprocessor communication is required to exchange neighbor information

Global Domain
A different view of Domain Decomposition

Domain is split into many pieces (more than #PEs)
Each PE takes some sub-set of the pieces
Almost a pool-of-tasks kind of approach
- Can help with load-balancing
- Can help with odd numbers of PEs

Iterative Parallelism, More Examples

- Linear algebra (ScaLAPACK)
  ◆ Split matrix and vectors across all machines
- Molecular Dynamics, Atomic Fracture
- Heart and neural modeling
- Ocean modeling, Atmospheric modeling
- Heat Equation, FEM – crash or fracture analysis
  ◆ Break large domain into smaller domains
- Genetic Algorithms
  ◆ Spread a large genetic population across all machines
- Monte Carlo methods
  ◆ Help ensure that random seeds don’t converge across parallel machines

For most real-world problems:
computational work ~ volume
communication ~ surface area
Recursive Parallelism

- “Divide and Conquer” approach
  - Instead of trying to directly compute something over the full range, say, 0-1, we compute 0-0.5 and 0.5-1 in parallel
    - Then split 0-0.5 into 0-0.25 and 0.25-0.5, etc.
  - Then combine the results
  - Of course, there need to be some “stopping rules” for when to subdivide or directly compute something

- Recursive Parallelism can be tricky
  - If subdivision is known a priori, then just use Iterative Parallelism
  - If subdivision is irregular, or computed on-the-fly, then it may be difficult to keep the set of CPUs load-balanced
  - Stopping rules may also place an upper-limit on available parallel speed-up

Recursive Parallelism, cont’d

![Diagram of recursive parallelism showing the integration of a function with four processors (PE1, PE2, PE3, PE4) splitting the task into smaller subtasks.](image)
Recursive Parallelism, Examples

- Adaptive Quadrature
  - Potentially could be used for “smart” parameter space studies
- Wave propagation in 2D/Multi-grid methods
  - Not all processors join in at all levels of the multi-grid
- Some linear systems
  - Tridiagonal and ADI methods lead to elegant recursive solutions
- Function minimization across a “tree” of choices
- Traveling Salesman Problem
  - “Best-First” search
- Molecular Dynamics, RADAR/MLFMA
  - Uses recursion to split up the problem
  - Then uses iterative parallelism to handle the resulting system

Pipeline Parallelism

- “Producer-Consumer” model
  - Output produced by previous stage is the input to (consumed by) the next stage
  - Like plugging in your stereo system
    - Output from CD player goes into the amp
    - Output from the amp goes into the speakers
  - Pre-process data, compute, post-process data
  - Can be fairly easy to program

- Pipeline depth is an upper limit to parallel speed-up
  - 3 stages in pipeline . . . 3x maximum speed-up
  - Pretty severe limitation under many circumstances
  - Many resources may not be well utilized
    - Pre-processing may take much less time to do than the main computations
Pipeline Parallelism, cont’d

Sequence Comparison: A vs B
Split A into 4 parts

Pipeline Parallelism, Examples

- Signal and Image processing
- Real-time signals
- Visualization and Post-Processing (AVS)
  - Each CPU performs one filtering operation in the chain
- Insertion sort
- Genetics searching (Smith-Waterman)
  - Push scoring results down the pipeline
- Iterative solution of some ODEs can benefit from pipelining
  - Each CPU handles a different time-step in the solution
    - As iterations are completed on T1, updated data is fed forward to T2
  - Not the most efficient means of using your CPUs
- Matrix multiply can be done in a ring/pipeline
  - But not usually the best way
Client-Server Parallelism

- Multiple independent clients that ONLY interact with the server
  - Client-to-Client data exchange may not be needed at all, or else it is handled through the server as 2 separate exchanges
    - Client #1 talks to Server
    - Server talks to Client #2
  - Server coordinates any data distribution that is needed

- Sometimes referred to as “Remote Procedure Calls”
  - Clients interact with server as though it was a local subroutine:
    ```c
    my_data = get_job_data( server );
    value = compute( my_data );
    err = post_job_results( server, my_data, value );
    
    Where the get_job_data and post_job_results functions just happen to execute on another machine
  - What if server has too many clients and slows down?

Client-Server Parallelism, cont’d
Client-Server Parallelism, Examples

- File serving, Web serving
- Database query
  - Server hands out the data requested by each client
  - Presumably, each client knows to request different pieces
- Data-mining
  - Multiple clients scan database for different parameters
- Monte Carlo methods
  - Server ensures unique random number seeds for clients
- Function mapping, minimization, successive refinement
  - If x=0 to x=0.3 have “high” values, then the server can focus future computational efforts elsewhere
- Web services
  - A new/emerging model for distributed client-server applications

Interacting Peers Parallelism

- Programs coordinate the work among themselves
  - No one CPU takes over as “master” or “server”
  - Different CPUs may be running different algorithms
- Programming issues
  - Requires that the programmer protect against “race conditions”
    - If 2 CPUs need to write data, who gets to write their data first?
  - May require some redundant computations and memory storage to keep track of what peers have what data to work on
    - Splitting of data onto machines may not be regular or fixed
  - Allows for a great deal of peer autonomy which can help with parallel performance
    - If data from CPU #5 is not available, then we’ll process data from CPU #6 instead
    - Or skip that phase of the algorithm altogether, and run another “inner” iteration
Interacting Peers Parallelism, cont’d

Atmospheric Model

Ocean Model

Error Checker

Each model operates asynchronously
Convergence is detected by a separate process

Event-Driven Interactions

Model_A

/* do primary work */
if( msg_type1 ) {
  /* do work */
} else if( msg_type2 ) {
  /* do other work */
}

Model_B

/* do primary work */
if( msg_type1 ) {
  /* do work */
} else if( msg_type2 ) {
  /* do other work */
}

Model_C

/* do primary work */
if( msg_type1 ) {
  /* do work */
} else if( msg_type2 ) {
  /* do other work */
}

msg1

msg1

msg1

msg2

msg2

msg1
Interacting Peers Parallelism, Examples

“Multi-Physics” models
- Different peers handle different aspects of micro- and macroscopic physics in the simulation

Genetic algorithms
- “Migration” of populations to other peers

Often, Peer-To-Peer and Iterative parallelism “look” the same
- Iterative Parallelism often treats each worker as an equal peer in the system … just with a different subdomain of the global domain
- A lot of times, Peer-To-Peer and Iterative Parallelism may be used for “parallel memory” and not necessarily for parallel speed-up
  - Ie. Most modern workstations can only hold 2GB of memory
  - If you need more … too bad!
  - With parallelism, you should be able to increase your available memory to (2GB x #CPUs)

Combinations of Parallel Methods

Ocean-Atmospheric modeling
- Ocean model uses 100+ CPUs
- Atmospheric model uses a different 100+ CPUs
- Only data at the air-water interface is exchanged between the two subsets of CPUs
- Iterative Parallelism within each model, Peer-To-Peer between

Data mining
- Parallel clients (multiple CPUs working on one data path)
- Parallel server (multiple CPUs working to serve out data)

Parallel Pipeline techniques for ODE systems
- Multiple CPUs (Domain Decomp) within each pipeline stage
  - Can also put more CPUs into compute stage, fewer into pre-processing
  - Keep each stage load-balanced
Grid Computing

The “Information Power Grid” is a new concept popularized by the SETI@Home project

- Your client downloads data from a radio telescope
- While your machine is in “screensaver” mode, the SETI software runs and processes the data
- “Cycle stealing” - there are lots of idle CPU cycles, how can we access them

“The Grid” can be viewed as a large queuing system

- Submit your job and it runs ... somewhere ... sometime (soon?) ...
  - Maybe it runs on your desk ... maybe in Japan!
- You get the same results back as if you had run it locally
- In return, you share your “spare cycles” with other users

TRUST is a big issue!

Grid Computing, cont’d

“The Grid” is definitely coming ... maybe not this year, but soon

- What can you do to get ready for it?

The grid will schedule WHEN and WHERE your job gets to run

- Don’t expect keyboard or monitor access
- Your program should be able to run “unattended”
- Your program should do something “intelligent” when errors occur
  - sounds familiar?

- You may also want to make your data-files accessible over the web
  - Web-servers can hand out binary and text data files as well as web pages
  - You may be able to do something clever with PHP/Perl scripts on the server too
Is Data Separable??

Is Data Sync??

More Than 1 Run ??

Data Changes to Future Runs ??

Job Parallelism

Client-Server

Pipeline

Inter. Peers

Dom. Decomp.

Job Parallelism, cont'd

Client-Server Parallelism, cont'd

Pipeline Parallelism, cont'd

Matching Parallel Models to Programming Models

Scripting

RPC

Pthreads

OpenMP

MPI
Suggestions for New Projects

If Job Parallelism or Client-Server will work for you, you’re in luck!
- Job Parallelism - - - Scripting (with an eye to Grid Computing)
- Client-Server - - - Web-based RPC (with an eye to Web Services)
If you own a LARGE SMP system, then use Pthreads or OpenMP (but consider MPI)

Otherwise, use MPI

Example: Lab with 5 dual-Intel workstations
- Job Parallelism: 10 single-CPU jobs
- Job Par/Threading: 5 dual-CPU SMP jobs
- Iter Par/MPI: single 10-CPU job

Outline

Types of computational parallelism
- Job parallelism
- Iterative parallelism or Domain Decomposition
- Recursive parallelism or Divide-and-Conquer
- Pipeline or Producer-Consumer parallelism
- Client-Server
- Interacting peers

Programming Languages/Libraries
- Sequential “Parallel” Programming
- Shared Memory Programming
- Distributed Memory Programming
Sequential “Parallel” Programming

✔️ For Job Parallelism (pool of work model), you may not have to write “real” parallel code at all
  ♦ E.g. you may have (or can generate) 1000 input files, each of which has to be processed on a single CPU

✔️ Really, this is handling the administrative overheads
  ♦ What input files do you need?
  ♦ What output files are produced?
    ● make sure to name files appropriately to avoid over-writing them
    ● no keyboard input, no screen output (or use redirection)
  ♦ How to get the files from your desktop/laptop over to the compute machine?
    ● ssh? scp? sftp?
    ● web? wget? lwp-request? curl?

SGE Scripting

✔️ SGE is the queuing system used on the DSCR (large Linux cluster)

✔️ “qsub script.q”
  ♦ SGE will start the script when resources are available
  ♦ You can request specific machine properties

✔️ E.g. you can write a program that writes 100 script.###.q files, and submits them to SGE
  ♦ Each *.q file is customized to use a different input file/subdir
  ♦ Perl/Python/shell scripts are good for this

```bash
#!/bin/tcsh
# $ -m b,e
# $ -l mem_free=1.5G
cd data.1
my_program > stdout
```

Have input data separated into different sub-directories
SGE “Array” Jobs

- For groups of identical jobs, you can submit one “Array” job
  - use `-t` argument to `qsub`
    - `-t min-max`  
    - `-t min-max:step`
  - Inside the script, you have to figure out which task is which
    - `JOB_ID`
    - `SGE_TASK_ID`

```bash
#!/bin/tcsh
# $ -m b,e
# $ -l mem_free=1.5G
# $ -t 1-100

cd data.$SGE_TASK_ID
my_program > stdout
```

Now, you `qsub` only one script, it figures out how to split the data

Scripting

- Learn a scripting language!!
  - Perl and Python are the obvious choices
  - These tend to be “interpreted” languages, not compiled
    - can make development and debugging a bit easier

- Scripting languages come with some “nice-ities”
  - Your script can launch a program, send it input, collect its output
    - system( "curl $url -o $file" )
    - open( PH, "ls -a $dir |" )
  - Easier text processing
    - Wildcard matches
      - $file =~ s/filename\s*=\s*(.*)/$1/
    - Line-oriented input (not byte-oriented)
  - Easy tests for file existence, write-ability
Scripting, csh Example

```csh
#!/bin/csh
foreach alpha ( 1.0 1.25 1.333 1.5 )
    if (-e infile_$alpha ) then
        echo "#!/bin/tcsh" > script.$alpha.q
        echo "# $ -m b,e" >> script.$alpha.q
        echo "# $ -l mem_free=1.5G" >> script.$alpha.q
        echo "run_program infile_$alpha" >> script.$alpha.q
    endif
    qsub script.$alpha.q
end
```

Alpha substitutes the value of the variable (as a string)

---

Scripting, Perl Example

```perl
#!/usr/bin/perl
for($alpha=1.0;$alpha<5.0;$alpha+=0.25) {
    open( FP, ">infile_$alpha" );
    print FP "alpha=$alpha\n"
    print FP "global_param=1.0\n"
    close( FP );
    open( QP, ">script.$alpha.q" );
    print QP "#!/bin/tcsh\n"
    print QP "# $ -m b,e\n"
    print QP "# $ -l mem_free=1.5G\n"
    print QP "run_program infile_$alpha\n"
    close( QP );
    system( "qsub script.$alpha.q" );
}
```

Creates input file and script file, then submits the script file
Grid Computing

- At some level, any scripted batch job should be easily adaptable to grid computing
  - Don’t use keyboard input or screen output
    - Everything should go through files
    - Think about using command line arguments (argc, argv or @ARGV)
    - Try to use program-friendly filenames
  - Try to assess what files are really needed for EVERY simulation in the job-pool
    - And what files are needed for a SPECIFIC simulation
  - Try to assess memory and CPU-run-time needs for your program

Client-Server

- The clients can be fairly easy to write
  - You may even be able to script the client-to-server interactions
    - E.g. client pulls down data and job-description files from server - the job-description file explains what needs to be done

- The server needs to run on a machine, 24x7, until the total job is done
  - This will make your local sys admin cringe!
  - Scripting languages will make the server programming easier, otherwise you might need to learn Unix sockets
    - Sometimes (often?) the server is not a performance-bottleneck, so the fact that it may be written in Perl/Python is not that bad

- A high-performance server can be difficult to write
  - Multi-threaded vs. forking vs. pre-forking servers
    - Perl and Python do allow forking/threading/etc.
Client-Server/RPC Programming

Remote Procedure Call (RPC) is a Sun protocol that is now open
- Fundamentally, it is just “y=function(x)”
- But the function executes on a remote machine

Create an RPC description file
```
program ALPHA_PROG {
    version ALPHA_VERS {
        double newalpha(void) = 1;
        int donealpha(double) = 2;
    } = 1;
} = 1234567;
```

Run ‘rpcgen’ on that file
```
rpcgen rpcALPHA.x
rpcgen -Ss rpcALPHA.x > server.c
rpcgen -Sc rpcALPHA.x > client.c
```

Add in your own code for the details of the functions
- The ‘rpcgen’ program creates the links necessary to transfer the data

Client-Server/RPC, example client
```
#include "rpcALPHA.h"
CLIENT* clnt;
double newalpha( void ) {
    double* rp;
    rp = newalpha_1(NULL, clnt);
    if( rp == NULL ) {
        clnt_perror(clnt, "call failed:");
    }
    return( *rp );
}
/* similar function for post_done_alpha */
int main( int argc, char* argv[] ) {
    double alpha;
    clnt = clnt_create( "localhost", ALPHA_PROG, ALPHA_VERS, "udp" );
    if( clnt != NULL ) {
        clnt_pcreateerror("localhost");
        exit(1);
    }
    alpha = newalpha();
    /* do work with alpha */
    donealpha( alpha );
    clnt_destroy( clnt );
}
```
Client-Server/RPC, example server

```c
#include "rpcALPHA.h"

double* newalpha_1_svc( void* argp, struct svc_req* rqstp) {
    static double result;
    int i,c = 0;
    FILE* fp;
    fp = fopen( "alphalist.db", "w" );
    while( !feof( fp ) ) {
        fseek( fp, SEEK_SET, 100*c );
        fscanf( fp, "%lf %i", &result, &i );
        if( i == 0 ) {
            fseek( fp, SEEK_SET, 100*c );
            fprintf( fp, "%lf %i", result, 1 );
            break;
        }
        c++;
    }
    fclose( fp );
    return(&result);
}
```

Client-Server/Web-Based

- Very similar to the RPC concept, but we rely on the HTTP (Web) protocol to transfer data
  - Very good with scripting
    - Use `wget`, `lwp-request`, or `curl` to download web pages (which are really your data files)
    - Then run the program on those downloaded files
  - Many languages contain primitive HTTP-GET/POST functions
    - HTTP is actually a very simple protocol, you can mimic it with about 10 lines of Perl code

- You can use HTTP-Post (fill-out forms) to send data back to the web server
  - Not so bad if your output is small
  - Be careful if you are trying to send back 100’s of MBs
  - Talk with your local sys admin first!!
Web, example client script

```perl
#!/usr/bin/perl

# create a unique temp directory ($$ is the PID)
mkdir("tmp.$$");
chdir("tmp.$$");

# get the input file
system("curl -o input.txt http://server.duke.edu/newjob.php");

# run the program, catch the error code in r
$r = system("run_program input.txt > output.txt");

# post the results
# -d and --data send HTTP-POST data back to server
system("curl -d err=$r --data output.txt http://server.duke.edu/jobdone.php");

# clean up
unlink("input.txt", "output.txt");
rmdir("tmp.$$");
```

Web, example server script

```php
<?php

// URL: http://server.duke.edu/newjob.php
// get next file in inputs directory
$dp = opendir("unassigned");
$file = readdir($dp);
closedir($dp);

// send the file's data to the client
$fp = fopen("unassigned/$file", "r");
$text = fread($fp, filesize("unassigned/$file");
fclose($fp);
print $text; // 'print' sends the data to the client

// now move that file to the "assigned" dir
rename("unassigned/$file", "assigned/$file");
?>
```

- lets you track the progress of your simulation
Client-Server, caveats

- The “server system” may fork off more than one copy of your server, especially if there is more than 1 CPU on the machine
  - Generally, this is a good thing . . . better performance!

- What happens when two programs write to the same file?
  - Take a look at the ‘flock’ function to lock files so only one server-process can read/write at a time
  - But this only works for local disks (not NFS-mounted disks)

- Any networked system (RPC or Web) increases the possibility of hackers damaging your system
  - “Damage” could mean taking down the server
  - Or maybe just tricking your server into thinking that jobs 10,11,12,13,14 were all completed (even though the data is bad)
  - Talk with your local sys admin about .htaccess files

Client-Server, caveats, cont’d

- Check ALL function calls for error values
  - With any networked system come network problems
  - Servers, routers, switches can be busy or even go down
  - Make sure your program does something intelligent when these errors happen (and they will!)
    - Perl/Python/shell scripts: some system commands will return errors to stderr, make sure you are capturing/piping stderr to your script

- Be aware of network “time-out” values
  - If a network link is busy, it may take 5-30 seconds to transmit the whole message and receive a reply
  - Many systems/programs are overly generous
    - Some time-outs may be 5 minutes or more!
    - Your program may have to wait 5 minutes just to find out that the server is down
Outline

- Types of computational parallelism
  - Job parallelism
  - Iterative parallelism or Domain Decomposition
  - Recursive parallelism or Divide-and-Conquer
  - Pipeline or Producer-Consumer parallelism
  - Client-Server
  - Interacting peers

- Programming Languages/Libraries
  - Sequential “Parallel” Programming
  - Shared Memory Programming
  - Distributed Memory Programming

Unix IPC

- Unix has many “parallel” features built in to it
  - “fork-exec” to start a new parallel process
  - Shared memory and semaphores allows programs on the same machine to communicate and share data
    - Not so easy to understand, and authentication/security may not be so great
  - Sockets allow programs to communicate with other machines
    - This is at the byte-level, you have to make sure that the remote machine is the same type
      - otherwise bytes 0x1234 may show up as 0x3412
  - That’s basically all you need for parallel programming
    - but it’s awfully ugly
      - “what is the bare minimum we need for parallelism?”
Unix IPC, example 1

```c
key_t ShmKey;
int   ShmID;
char* data;

/* get a shared memory key */
ShmKey = ftok("/tmp/some/path", 'm');
/* now try to get some memory - create if necessary */
ShmID = shmget( ShmKey, 1024, IPC_CREAT|0666 );
if( ShmID == -1 ) {
    printf("Error in shmget(): errno = %i:%s\n",
            errno,sys_errlist[errno]);
    exit(-1);
}
/* attach to the memory segment */
data = shmat( ShmID, 0, 0 );
if( data == (char*)(-1) ) {
    printf("Error in shmat(): errno = %i:%s\n",
            errno,sys_errlist[errno]);
    exit(-2);
}
```

Unix IPC, example 2

```c
i = socketpair( AF_UNIX, SOCK_STREAM, 0, sockets );
child = fork();

if( child != 0 ) {
    /* this is the parent */
    close( sockets[0] );
    i = read( sockets[1], buf, sizeof(buf) );
    i = write( sockets[1], "test string 111111", 15 );
    close( sockets[1] );
} else {
    /* child process */
    close( sockets[1] );
    sleep( 1 );
    i = write( sockets[0], "test string 222222", 15 );
    i = read( sockets[0], buf, sizeof(buf) );
    close( sockets[0] );
}
```
Shared Memory Programming

Recall the main caveats with shared memory programming:

- Limited to the number of CPUs on a single machine
  - Just 2-8 CPUs on each DSCR cluster machine
  - More than 16 or 32 CPUs gets VERY expensive
- Limited to the memory space of a single machine
  - And this memory is SHARED by all other users on the machine
- Note that a distributed memory program CAN BE RUN on a shared memory system (but not vice-versa)

It is often easier to get a SMP program running correctly
- But often it is hard to get SMP running fast
- By the time you tweak the SMP code, you may have exerted as much effort as writing an distributed memory MPI code

OpenMP

Many scientific applications use Domain Decomposition
- E.g. the same mathematical operations are being done on every piece of data in a large array
- So we want to split the array in pieces and have each CPU operate on its own “local” piece of the “global” array

```
for(i=0; i<3000000; i++) {
    a[i] = b[i] * c[i] + d[i];
}
```

```
“Parallel” Code
for(i=0; i<1000000; i++) {
    a[i] = b[i] * c[i] + d[i];
}
CPU0
for(i=1000000; i<2000000; i++) {
    a[i] = b[i] * c[i] + d[i];
}
CPU1
for(i=2000000; i<3000000; i++) {
    a[i] = b[i] * c[i] + d[i];
}
CPU2
```
OpenMP, example

Original Code

```c
for(i=0;i<3000000;i++) {
    a[i] = b[i]*c[i] + d[i];
}
```

OpenMP Code

```c
#pragma omp PARALLEL FOR
for(i=0;i<3000000;i++) {
    a[i] = b[i]*c[i] + d[i];
}
```

Not too much effort for a “real” parallel program!!

unfortunately...

performance is not so good

OpenMP, cont’d

◆ You must have a OpenMP-compliant C or Fortran compiler

```
#icc -openmp mycode.c
```

◆ The compiler “sees” those ‘#pragma omp’ lines and it ALTERS your code for parallel operation

- Some of those alterations may be based on guesses as to how best to split the problem ... and some of those guesses may be wrong, or there may be no efficient way for the compiler to parallel-ize the program

◆ You’re relying on the compiler to handle a lot of the performance optimization for you

- Vendor ($$) compilers tend to be much better than gcc

◆ But a few ‘#pragma omp …’ lines and you could see 2-4x performance
Pthreads

“Threads” are sometimes called “light-weight processes”
- Multiple threads share the same memory space (same variables)
  - But each thread has its own program counter and stack
  - ... each thread is its own virtual CPU
- You create threads and tell them to execute specific functions
  - Domain Decomposition/Iterative Parallelism: threads call the same “work” function with different start/stop points in the domain

Synchronization functions:
- Lock, Unlock - regions of memory or code
- Barrier - wait for all threads to reach a certain point in the code
- Conditional Signals - you can wake other threads that may be waiting for a lock that another threads owns
  - You can also use Unix signal handlers to allow certain threads to respond to external signals

Pthreads, example

```c
#include <pthreads.h>

#include <pthreads.h>

pthread_t PTlist[64];
int ThrSplit[65];

void work( void* ptr ) {
    int ThrNum;
    ThrNum = (int)(ptr);
    for( i=ThrSplit[ThrNum]; i<ThrSplit[ThrNum+1]; i++ ) {
        a[i] = b[i]*c[i] + d[i];
    }
}

int main( int argc, char** argv ) {
    ThrStart[0] = 0;
    for( i=1; i<=NumThreads; i++ ) {
        ThrSplit[i] = ThrSplit[i-1] + 10000;
    }
    for( i=1; i<NumThreads; i++ ) {
        pthread_create( &PTlist[i], NULL, work, (void*)(i) );
    }
    work(0);  
}
```

Now “master” thread joins workers

Compute parallel splitting of data

Could check pthread_self() against PTlist[]
Outline

- Types of computational parallelism
  - Job parallelism
  - Iterative parallelism or Domain Decomposition
  - Recursive parallelism or Divide-and-Conquer
  - Pipeline or Producer-Consumer parallelism
  - Client-Server
  - Interacting peers

- Programming Languages/Libraries
  - Sequential “Parallel” Programming
  - Shared Memory Programming
  - Distributed Memory Programming

Distributed Memory Programming

- Distributed memory programming generally refers to “message passing programming”

- Programmer must take care of all message-sends, message-receives, and synchronization
  - Note that while all machines may be identical and have excellent network connectivity . . . they are NOT necessarily synchronized
  - And if they are in-sync now, they won’t be in a few seconds

- MPI and PVM were the two big “open” standards
  - MPI eventually “won” over PVM
  - Vendors still provide their own proprietary libraries for their proprietary hardware
  - SHMEM is making some moves lately
    - Requires better network cards ($)
MPI

MPI is an industry standard that is supported by all major vendors
- Linux, Windows, Sun, SGI, HP, IBM, Cray, Fujitsu, NEC
- Assuming you don’t do anything strange with I/O or graphics, an MPI program should be able to run on anything
- Version 1.2 is the current standard
  - No “growth-on-demand” features, if you request 10 CPUs, you get exactly 10 CPUs and cannot expand later, even if more CPUs become available
- Version 2.0 is slowly gaining support
  - Does include dynamic growth/shrinking of the CPU set

MPI, in its entirety, encompasses over 120 functions
- But real-world MPI programs often need only 6-10 of them
  - This helps reduce the learning curve quite a bit!

“SPMD” Programming Style

MPI is refered to as “Single-Program, Multiple Data (SPMD)”
- Exact same program is running on each “Processing Element”
- But each PE can have different data
  - And thus, each PE can take different paths through the program
  - E.g. “if (SelfPE == 0 ) then . . else if (SelfPE == 1 ) then . .”
- Often, it is easiest to think of all PEs being in sync

You, the programmer, must explicitly call functions to send and receive data to/from other PEs
- If PE0 issues a SEND to PE1
- Then PE1 had better issue a RECV from PE0
  - or else the program may lock up!
MPI, example

```c
MPI_Init( &argc,&argv );
MPI_Comm_Size( MPI_COMM_WORLD, &NumPEs );
MPI_Comm_Rank( MPI_COMM_WORLD, &SelfPE );
if( SelfPE == 0 ) {
    MPI_Send( &data0, 5, MPI_INT, 1, 55, MPI_COMM_WORLD );
    MPI_Recv( &data1, 10, MPI_FLOAT, 1, 66, MPI_COMM_WORLD, &stat );
} else if( SelfPE == 1 ) {
    MPI_Recv( &data0, 5, MPI_INT, 0, 55, MPI_COMM_WORLD, &stat );
    MPI_Send( &data1, 10, MPI_FLOAT, 0, 66, MPI_COMM_WORLD );
}
MPI_Finalize();
```

Notice that the number of items in the array, the datatype, the message tag, and the remote PEs must match between the send and recv calls

MPI, example 2

```c
/* communicate "interface" data */
MPI_Irecv( &data1, 10, MPI_FLOAT, PE, 66, MPI_COMM_WORLD, &recvreq );
MPI_Isend( &data0, 10, MPI_FLOAT, PE, 55, MPI_COMM_WORLD, &sendreq );

/* work on "local" data */
/* wait for "interface" data to arrive */
MPI_Wait( &recvreq, &recvstat );

/* work on "interface" data */
MPI_Request_free( &sendreq );
```

This "local" work can hide the latency/delay associated with the network/comm
Parallel Programming, caveats

- Check return values of ALL function calls

- In addition to “regular” program bugs, you can now get “parallel” bugs
  - Deadlock conditions (waiting for a message that was never sent)
  - Race conditions (making sure that msg#1 arrives before msg#2)
    - Program runs fine 20 times in a row, then crashes
    - Non-repeatable crash behavior

- Parallel debugging is still in its infancy!!
  - Lots of “print” statements
  - But those statements alter the timing of the program, and hence can eliminate race conditions that were part of the original bug
    - Or they can create new race conditions that were not there before
  - Program-tracing is a bit lighter weight

Parallel Programming, caveats, cont’d

- Don’t do parallel file-write operations!! (at least not on NFS)
  - It won’t always work as you expect
  - Even flock() won’t help you in all cases
    - Push all output through a single PE

- Random numbers may not be so random, especially if you are running 1000 jobs which need 1000 “random” numbers (use SPRNG)

- Parallel (in)efficiency can also creep in
  - Correct results may be produced, but while using 100 CPUs, the simulation is only sped up by 10-20x
    - 99 CPUs waiting while 1 CPU finishes its work
    - Latency hiding is THE KEY to good parallel programming