High Performance Parallel Programming (CS61064)

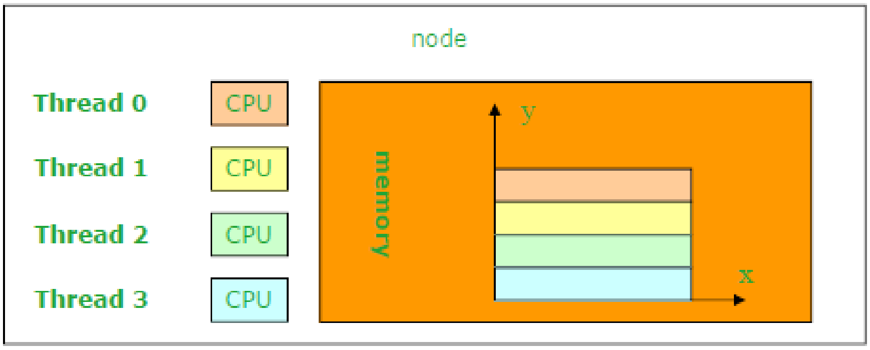
Week – 2
Part 1

Pralay Mitra

Parallel Software Models and Languages

- **Programming Models**
  - Shared Memory (OpenMP)
  - Message Passing (MPI)
  - Hardware Accelerators (CUDA, OpenCL)
  - Hybrid
- **Programming Language:**
  - C
  - C++
  - Fortran

Shared Memory



Shared Memory

OpenMP

– Main Characteristics

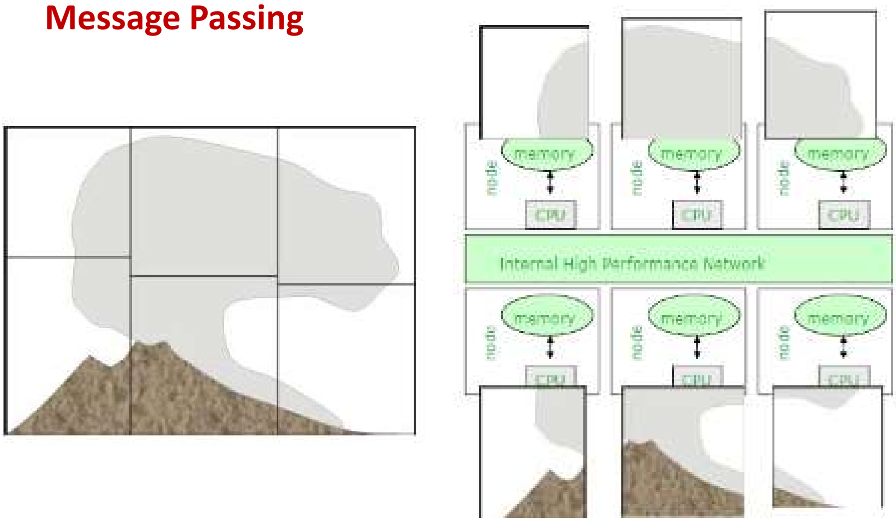
- Compiler directives
- Medium grain
- Intra node parallelization
- Loop or iteration partition
- Shared memory
- Many HPC App

– Open Issues

- Thread creation overhead
- Memory/core affinity
- Interface with MPI

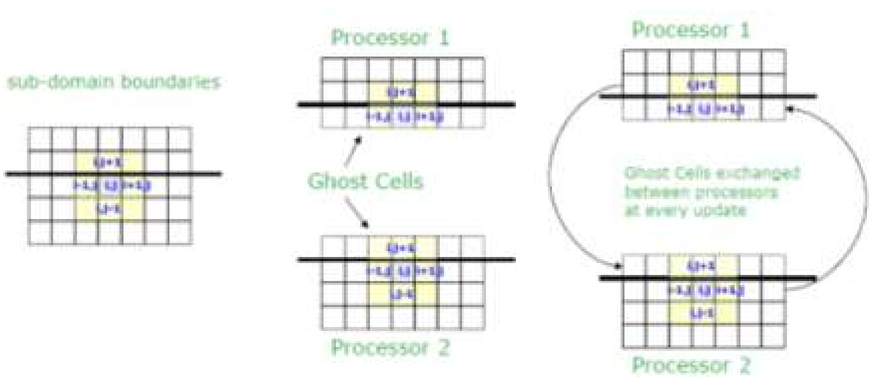
Private Memory

Message Passing



Private Memory

Message Passing



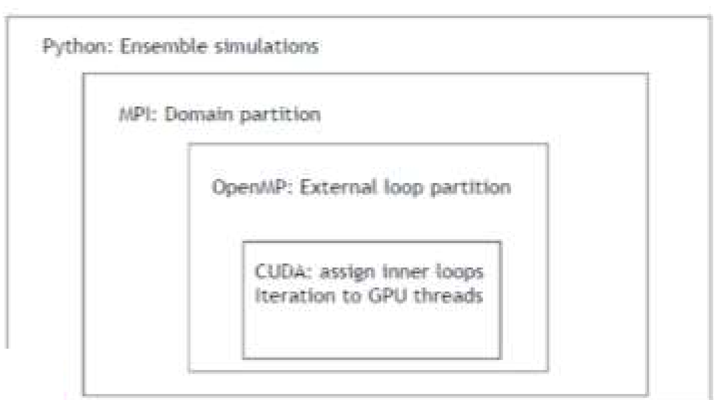
Private Memory

MPI

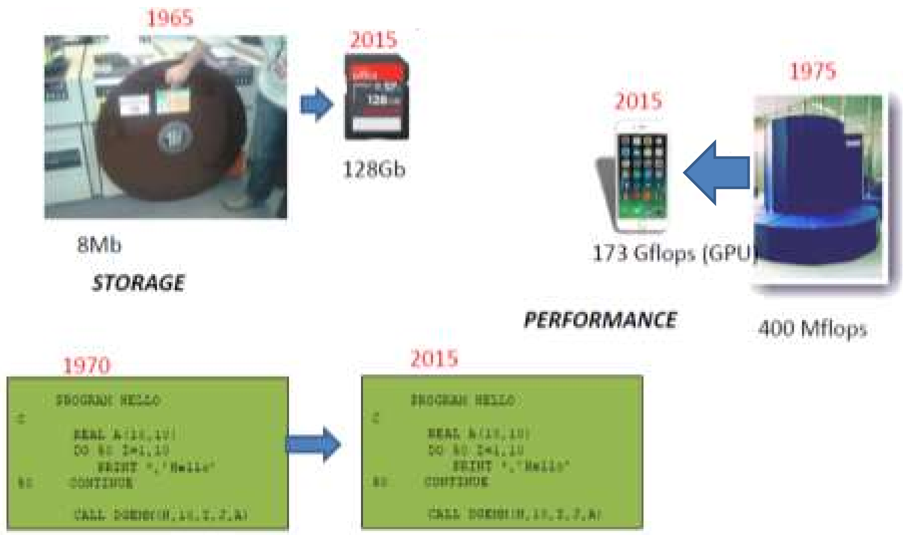
– **Main Characteristics**
  - Library
  - Coarse grain
  - Inter node parallelization
  - Domain partition
  - Distributed memory
  - Almost all HPC parallel App

– **Open Issues**
  - Latency
  - OS Jitter
  - Scalability

Hybrid Parallel Programming



Advances: Hardware vs Software



Real HPC Crisis: Software

A supercomputer application and software are usually much more long-lived than a hardware

- Hardware life typically four-five years at most.
- Fortran and C are still the main programming models

Programming is stuck

- Arguably hasn't changed so much since the 70's

Software is a major cost component of modern technologies.

- The tradition in HPC system procurement is to assume that the software is free.

It's time for a change

- Complexity is rising dramatically
- Challenges for the applications on Petaflop systems
- Improvement of existing codes will become complex and partly impossible.
- The use of O(100K) cores implies dramatic optimization effort.
- New paradigm as the support of a hundred threads in one node implies new parallelization strategies
- Implementation of new parallel programming methods in existing large applications can be painful

Software Difficulties

- Legacy applications (includes most scientific applications) not designed with good software engineering principles. Difficult to parallelise programs with many global variables, for example.
- Memory/core decreasing.
- I/O heavy impact on performance, esp. for BlueGene where I/O is handled by dedicated nodes.
- Checkpointing and resilience.
- Fault tolerance over potentially many thousands of threads.
  - In MPI, if one task fails all tasks are brought down.

Summary

- HPC is only possible via parallelism and this must increase to maintain performance gains.
- Parallelism can be achieved at many levels but because of limited code scalability with traditional cores increasing role for accelerators (e.g. GPUs, MICs). The Top500 is becoming now becoming dominated by hybrid systems.
- Hardware trends forcing code re-writes with OpenMP, OpenCL, CUDA, OpenACC, etc in order to exploit large numbers of threads.
- Unfortunately, for many applications the parallelism is determined by problem size and not application code.
- Energy efficiency (Flops/Watt) is a crucial issue. Some batch schedulers already report energy consumed and in the near future your job priority may depend on predicted energy consumption.

Your first openMP program

```
#include <stdio.h>
#include <omp.h>

int main()
{
        #pragma omp parallel
        {
                printf("Hello world!\n");
        }
        return 0;
}
```

Your first openMP program

Check your system support: locate omp.h
/usr/lib/gcc/x86_64-redhat-linux/4.8.2/include/omp.h

Compilation: g++ -fopenmp first_openMP.c

Conditional Compilation:

```
#ifdef _OPENMP
        printf("Compiled with OpenMP support:%d",_OPENMP);
#else
        printf("Compiled for serial execution.");
#endif
```

Execution: ./a.out

Flags:

GNU: **-fopenmp** for Linux, Solaris, AIX, MacOSX, Windows.
IBM: **-qsmp=omp** for Windows, AIX and Linux.
Sun: **-xopenmp** for Solaris and Linux.
Intel: **-openmp** on Linux or Mac, or **-Qopenmp** on Windows
PGI: **-mp**

Your first openMP program

```
$ ./a.out
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

Setting Environmental Variable

- **Know your shell**

    $ echo $SHELL

    $ /bin/bash

$ export OMP_NUM_THREADS=16

Your first openMP program

```
#include <stdio.h>
#include <omp.h>

int main()
{
        #pragma omp parallel
        {
                printf("Hello world!\n");
        }
        return 0;
}
```

Setting Environmental Variable and Executing

```
//Know your shell
$ echo $SHELL
$ /bin/bash
//Setting environmental variables
$ export OMP_NUM_THREADS=8
//Compilation
$ g++  -fopenmp  first_openMP.c
//Execution
$ ./a.out
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

Directives

- **Syntactically directives are just comments**
  - **#pragma omp** *directive-name [clause[[,] clause]...] new-line*

- **Examples**
  - #pragma omp parallel

- **Clause is one of the followings**
  - if(scalar-expression)
  - private(variable-list)
  - firstprivate(variable-list)
  - default(shared | none)
  - shared(variable-list)
  - copyin(variable-list)
  - reduction(operator: variable-list)
  - num_threads(integer-expression)

- **Multiple directive names are not allowed**
  - #pragma omp parallel barrier

parallel construct

#pragma omp parallel

- Forms a team of *N* threads before starting executing parallel region
- *N* is set by *OMP_NUM_THREADS* environment or using function *omp_set_num_threads()*
- Semantics is (almost) same as serial program

Comments on *parallel* construct

- At most one **if** clause can appear on the directive (serial/parallel)
- It is unspecified whether any side effects inside the **if** expression or **num_threads** expression occur.
- Only a single **num_threads** clause can appear on the directive. The **num_threads** expression is evaluated outside the context of the parallel region, and must evaluate to a positive integer value.
- The order of evaluation of the **if** and **num_threads** clauses is unspecified.
- A nested parallel region is executed by a team composed of one thread. The default behavior may be changed by using either the runtime library function **omp_set_nested.**
- If the **num_threads** clause is present then it supersedes the number of threads requested by the **omp_set_num_threads** library function or the **OMP_NUM_THREADS** environment variable only for the parallel region it is applied to. Subsequent parallel regions are not affected by it.

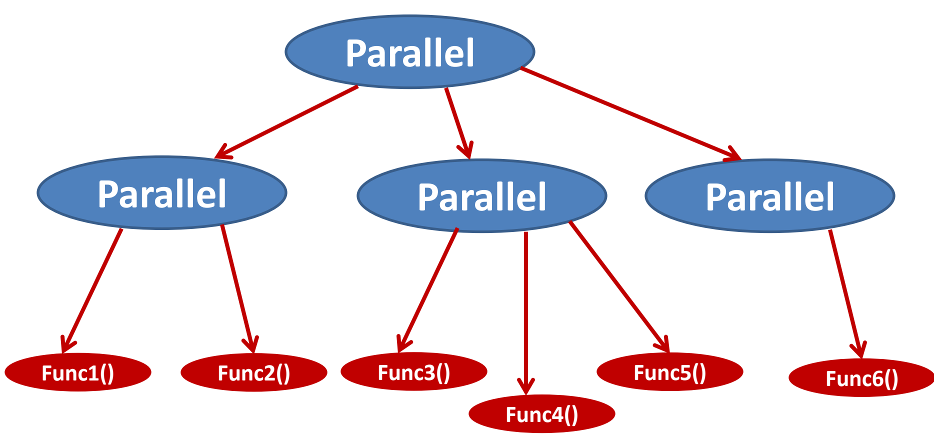
High Performance Parallel Programming (CS61064)

Week – 2
Part 2

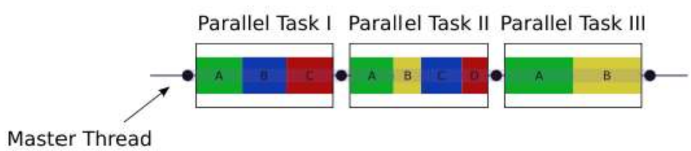
Pralay Mitra

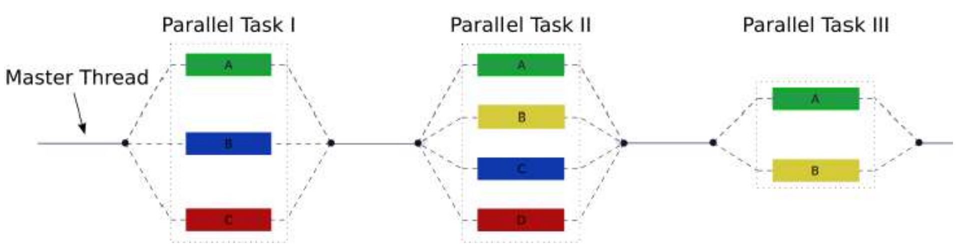
OpenMP

Overview



Overview





Amdahl's law

- If a proportion (in time) **P** of a code can get **S** speed up then total speedup will be (less than)

$$S_{latency}(S) = \frac{1}{(1-P)+\frac{P}{S}}$$

- Improving **P** is more important than improving **S** that is often more difficult too.

Parallelization

- A serial code has three parts (A, B, C).
  - A takes 5secs (wish to make it 2 times faster)
  - B takes 1secs (wish to make it 7 times faster)
  - C takes 2secs (wish to make it 3 times faster)
  - You are allowed to parallelize only one part.

    How many computing resources are required to achieve maximum speed up (which one will you choose)?

    What is the percentage of improvement?

Take home message

- Do you need infinite number of computing and storage power?

  ~~Possibly~~ NO!!

Run time library functions

#include <omp.h>

- omp_get_num_threads() returns number of threads
- omp_get_thread_num() returns the thread ID of the current thread
- omp_set_num_threads() sets number of threads
- omp_get_wtime() returns the wall clock time in sec
- omp_get_max_threads()
- omp_get_num_procs()
- omp_in_parallel()
- omp_set_dynamic()
- omp_get_dynamic()
- omp_set_nested()
- omp_get_nested()

Example -2

```
# include <stdio.h>
# include <omp.h>
int main ( int argc, char *argv[] ) {
 int id;
 double wtime;
 printf ( "Number of processors available = %d\n", omp_get_num_procs ( ) );
 printf ( "Number of threads =          %d\n",  omp_get_max_threads ( ) );
 wtime = omp_get_wtime ( );
 printf ( "OUTSIDE the parallel region.\n" );

 id = omp_get_thread_num ( );
 printf ( "HELLO from process %d\n Going INSIDE the parallel region:\n ", id ) ;

# pragma omp parallel \
 private ( id )  {
  id = omp_get_thread_num ( );
  printf ("  Hello from process %d\n", id );
 }
 wtime = omp_get_wtime ( ) - wtime;

 printf ( "Back OUTSIDE the parallel region.\nNormal end of execution.\nElapsed wall clock time = %f\n", wtime );
 return 0;
}
```

Example -2

Initialization:

export OMP_NUM_THREADS=16

Compilation:

g++ -fopenmp example.c

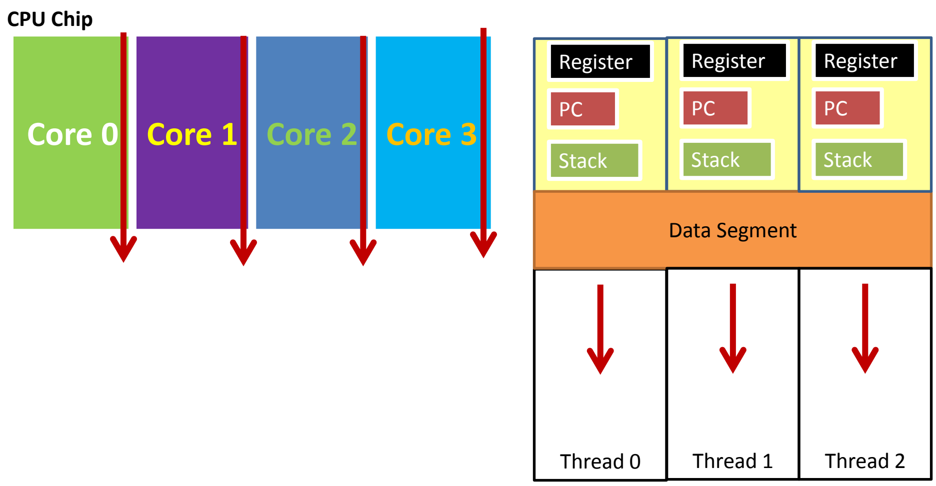
Execution:

./a.out

Example -2

```
Number of processors available = 8
Number of threads =           16
OUTSIDE the parallel region.
HELLO from process 0
Going INSIDE the parallel region:
  Hello from process 4
  Hello from process 0
  Hello from process 2
  Hello from process 14
  Hello from process 12
  Hello from process 13
  Hello from process 3
  Hello from process 7
  Hello from process 1
  Hello from process 8
  Hello from process 9
  Hello from process 10
  Hello from process 11
  Hello from process 5
  Hello from process 6
  Hello from process 15
Back OUTSIDE the parallel region.
Normal end of execution.
Elapsed wall clock time = 0.001034
```

Multi-core CPU



Example -3

```
#include <stdio.h>
#include <omp.h>

int main()
{
        int ii;
        #pragma omp parallel
        {
                for(ii = 0; ii < 10; ++ii)
                        printf("iteration %d\n", ii);
        }
        return 0;
}
```

What will be the output?

There is a loop. I wish to make it parallel.

WRONG

Example -4

```
#include <stdio.h>
#include <unistd.h>
#include <omp.h>

int main()
{
    int i,j,n,m,temp,a[100][100];
    n=m=7;
    for(i=0;i<=n*m-1;i++) {
        temp=i/m+1;
        j=i%m+1;
        sleep(1);
        a[temp][j]=temp+100*(j-1);
    }
    for(i=0;i<=n*m-1;i++) {
        temp=i/m+1;
        j=i%m+1;
        if(i%m==0) printf("\n");
        printf("%d\t",a[temp][j]);
    }
    printf("\n");
    return 0;
}
```

```
$ ./a.out

1    101  201  301  401  501  601
2    102  202  302  402  502  602
3    103  203  303  403  503  603
4    104  204  304  404  504  604
5    105  205  305  405  505  605
6    106  206  306  406  506  606
7    107  207  307  407  507  607
```

Example -4

```
#include <stdio.h>
#include <unistd.h>
#include <omp.h>
int main()
{
    int i,j,n,m,temp,a[100][100];
    n=m=7;
    #pragma omp parallel
    {
            for(i=0;i<=n*m-1;i++) {
                temp=i/m+1;
                j=i%m+1;
                sleep(1);
                a[temp][j]=temp+100*(j-1);
            }
    }
    for(i=0;i<=n*m-1;i++) {
        temp=i/m+1;
        j=i%m+1;
        if(i%m==0) printf("\n");
        printf("%d\t",a[temp][j]);
    }
    printf("\n");
    return 0;
}
```

```
$ ./a.out

1    0    201  0    401  0    601
0    102  0    302  0    502  0
3    0    203  0    403  0    603
0    104  0    304  0    504  0
5    0    205  0    405  0    605
0    106  0    0    0    506  606
7    0    207  0    407  0    607
```

Example -4

```
#include <stdio.h>
#include <unistd.h>
#include <omp.h>
int main()
{
    int i,j,n,m,temp,a[100][100];
    n=m=7;
    #pragma omp parallel private (temp, j)
    {
            for(i=0;i<=n*m-1;i++) {
                temp=i/m+1;
                j=i%m+1;
                sleep(1);
                a[temp][j]=temp+100*(j-1);
            }
    }
    for(i=0;i<=n*m-1;i++) {
        temp=i/m+1;
        j=i%m+1;
        if(i%m==0) printf("\n");
        printf("%d\t",a[temp][j]);
    }
    printf("\n");
    return 0;
}
```

```
$ ./a.out

1   101  201  301  401  501  601
2   102  202  302  402  502  602
3   103  203  303  403  503  603
4   104  204  304  404  504  604
5   105  205  305  405  505  605
6   106  206  306  406  506  606
7   107  207  307  407  507  607
```

Work-sharing Constructs

- **for Construct**
  - **#pragma omp for** *[clause[[,] clause] ...] new-line*
    for-loop
  - ***Clause***
    - **private(***variable-list***)**
    - **firstprivate(***variable-list***)**
    - **lastprivate(***variable-list***)**
    - **reduction(***operator***:** *variable-list***)**
    - **ordered**
    - **schedule(***kind[***,** *chunk_size]***)**
    - **nowait**

Work-sharing Constructs

– **for (***init-expr***;** *var logical-op b***;** *incr-expr***)**

- **init-expr/incr-expr: same as C**
- **var:** A signed integer variable. If this variable would otherwise be shared, it is implicitly made private for the duration of the for. This variable must not be modified within the body of the for statement. Unless the variable is specified lastprivate, its value after the loop is indeterminate.
- **logical-op:** >, <, >=, <=
- ***lb*, *b*, and *incr:*** Loop invariant integer expressions. There is no synchronization during the evaluation of these expressions. Thus, any evaluated side effects produce indeterminate results.

Loop Construct

```
void nowait_example(int n, int m, floa t *a, floa t *b, floa t *y, floa t *z)
{
        #pragma omp parallel
        {
                #pragma omp for nowait
                        for (int i=1; i<n; i++)
                                b[i] = (a[i] + a[i-1]) / 2.0;
                #pragma omp for nowait
                        for (int i=0; i<m; i++)
                                y[i] = sqrt(z[i]);
        }
}
```

Work out - 1

- Write a serial program to output the prime numbers occurring between 1 and 131072. Report the time required to compute the i^{th} prime (where i will be taking as input).

- Convert it to a OpenMP code. Report the percentage of improvement over serial program using 4 and 16 cores.