HPC using OpenMP

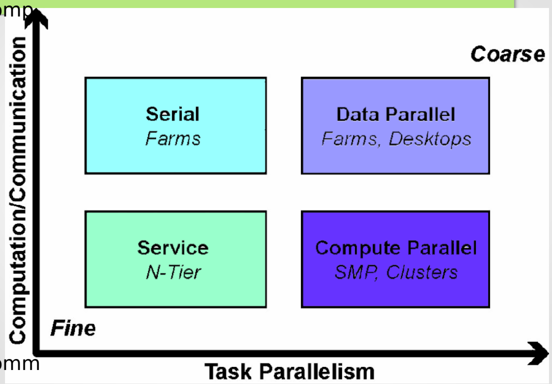
Dr. Alejandro Zunino
CONICET / ISISTAN–UNICEN
azunino@exa.unicen.edu.ar

Granularity

- how often their subtasks need to synchronize or communicate:
  - **fine-grained parallelism**: subtasks must communicate many times per second
  - **coarse-grained parallelism**: they do not communicate many times per second,
  - **embarrassing parallelism**: they rarely or never have to communicate

Applications and Architectures

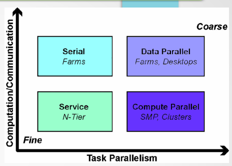
+ Comp



+ Comm

Applications and Architectures

- vertical axis:
  - identifies a specific ratio of computation (increasing from bottom to top) to communication (increasing from top to bottom)
- horizontal axis:
  - **degree of parallelism present in the application** (increasing from left to right)



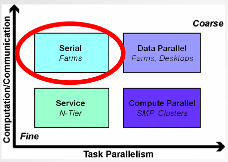
Types of paralellism: Fine grained

- a program is broken down to a large number of small tasks
- the amount of work associated with a parallel task is **low and the work is evenly distributed** among the processors.
- facilitates load balancing
- best exploited in architectures which support **fast communication**
- it is usually the compilers' responsibility to detect fine-grained parallelism

Types of paralellism: Coarse grained

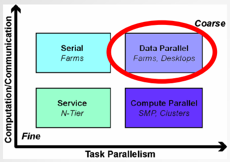
- a program is split into large tasks.
- a large amount of computation takes place in processors
- this might result in load imbalance, wherein certain tasks process bulk of the data while others might be idle.
- coarse-grained parallelism **fails to exploit the parallelism in the program** as most of the computation is performed sequentially on a processor
- **low communication and synchronization overhead.**
- Message-passing architecture takes a long time to communicate data among processes which makes it suitable for coarse-grained parallelism

Serial Applications

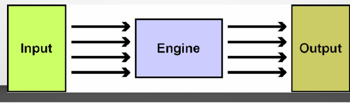


- Each step of the scientific calculation be performed in sequence.
- Can be executed on isolated desktops, servers, or supercomputers to farms.
- Compute farms are loosely coupled compute architectures in which system software is used to virtualize compute servers into a single system environment (SSE).

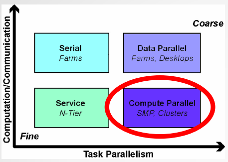
Data-driven Parametric Processing



- Focus on data processing
- Subdividing input data into multiple segments
- Processing each data segment independently via the same executable
- Reassembling the individual results to produce the output data.
- Not suitable for general applications...

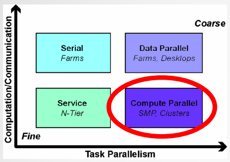


Compute Parallel Applications



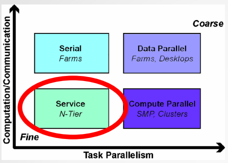
- In some problems parallelism can be exploited at the source-code level:
  - by taking advantage of loop constructs in which each calculation is independent of others in the same loop.
  - Parallelism in data is absent or of minor consequence.

Compute Parallel Applications (2)



- shared memory (threads):
  - OpenMP: minimal language extensions
  - UPC (unified parallel C)
  - SMP & threads
- distributed memory (processes):
  - PVM, MPI: extensive source level modifications
- Require tightly coupled architectures

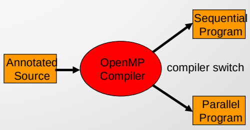
Service Applications



- Communicate extensively while carrying out minimal processing
- Focus is networking itself
- Can be applied in a variety of contexts:
  - loosely coupled architectures

OpenMP

- Application program interface (API) for shared memory parallel programming
- A specification for a set of compiler directives, library routines, and environment variables
- Make it easy to create multi-threaded (MT) programs in Fortran, C and C++
- Portable / multi-platform
- Jointly defined and endorsed by a group of major computer hardware and software vendors



OpenMP is Not

- Not Automatic parallelization
  - User explicitly specifies parallel execution
  - Compiler does not ignore user directives even if wrong
- Not just loop level parallelism
  - Functionality to enable coarse grained parallelism
- Not meant for distributed memory parallel systems
- Not necessarily implemented identically by all vendors
- Not Guaranteed to make the most efficient use of shared memory

Why?

- Parallel programming before OpenMP
  - Standard way to program distributed memory computers (MPI and PVM)
  - No standard API for shared memory
- Several vendors had directive based API for shared memory programming
  - All different, vendor proprietary
- Commercial users, high end software vendors have big investment in existing code
- Portability possible only through MPI
  - Library based, good performance and scalability
  - But sacrifice the built in shared memory advantage of the hardware

Goals

- Standardization :
  - Provide a standard among a variety of shared memory architectures/platforms
- Lean and mean :
  - Establish a simple and limited set of directives for programming shared memory machines.
- Ease of Use :
  - Provide capability to incrementally parallelize a serial program
  - Provide the capability to implement both coarse-grain and fine-grain parallelism
- Portability :
  - Support Fortran, C, and C++. Java (JOMP)

HelloWorld With Pthread

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


void* thrfunc(void* arg)
{
   printf("hello from thread %d\n", *(int*)arg);
}
```

HelloWorld With Pthread

```
int main(void)
{
   pthread_t thread[4];
   pthread_attr_t attr;
   int arg[4] = {0,1,2,3};
   int i;

   // setup joinable threads with system scope
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

   // create N threads
   for(i=0; i<4; i++)
      pthread_create(&thread[i], &attr, thrfunc, (void*)&arg[i]);
   // wait for the N threads to finish
   for(i=0; i<4; i++)
      pthread_join(thread[i], NULL);
}
```

Motivation

- Thread libraries are hard to use:
  - Pthreads threads have many library calls for initialization, synchronization, thread creation, condition variables, etc.
  - Programmer must code with multiple threads in mind
- Synchronization between threads introduces a new dimension of program correctness

Motivation

Wouldn't it be nice to write serial programs and somehow parallelize them "automatically"?

OpenMP can parallelize many serial programs with relatively few annotations that specify parallelism and independence

OpenMP is a small API that hides cumbersome threading calls with simpler *directives*

OpenMP: Methodology

- Start with a *parallelizable* algorithm
  - Embarrassing parallelism is good, loop-level parallelism is necessary
- Implement serially, mostly ignoring:
  - Data Races
  - Synchronization
  - Threading Syntax
- Test and Debug
- Annotate the code with parallelization (and synchronization) directives
  - Hope for linear speedup
- Test and Debug

HelloWorld With OpenMP

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

int main(void)
{
    #pragma omp parallel
    printf("hello from thread %d\n", omp_get_thread_num());
}
```

How to compile and run OpenMP programs?

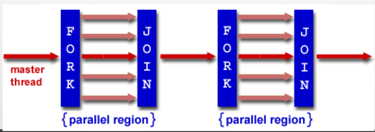
- GCC >= 4.2 supports OpenMP 3.0, VS2010 supports OpenMP 2.0
- To compile:
  - gcc –fopenmp helloOpenMP.c
- To execute:
  - export OMP_NUM_THREADS=4
  - ./a.out

Programming Model

- Thread Based Parallelism
  - A shared memory process with multiple threads
  - Based upon multiple threads in the shared memory programming paradigm
- Explicit Parallelism
  - Explicit (not automatic) programming model
  - Offer the programmer full control over parallelization

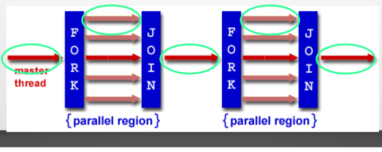
Programming Model

- Fork - Join Model
  - All OpenMP programs begin as a single sequential process: the **master thread**
  - Fork at the beginning of parallel constructs
    - The master thread creates a *team* of parallel threads
    - The statements enclosed by the parallel region construct are executed in parallel
  - Join at the end of parallel constructs
    - The threads synchronize and terminate after completing the statements in the parallel construct
    - Only the master thread exists



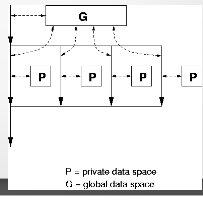
Programming Model

- Master Thread
  - Thread with ID=0
  - Only thread that exists in sequential regions
  - Depending on implementation, may have special purpose inside parallel regions
  - Some special directives affect only the master thread (like master)



Data Model

- Private and shared variables:
  - Shared: accessed by all parallel threads
  - Private: in a thread's private space. Can only be accessed by the thread



General Code Structure

```
#include <omp.h>
main ()
{
   int var1, var2, var3;

   Serial code
   ...
   /* Beginning of parallel section. Fork a team of threads.
      Specify variable scoping */
   #pragma omp parallel private(var1, var2) shared(var3)
   {
      Parallel section executed by all threads
      ...
      All threads join master thread and disband
   }
   Resume serial code
}
```

Matrix Multiplication: Sequential Version

```
for (i=0; i<N; i++) {
   for (j=0; j<N; j++) {
      temp = 0;
      for (k=0; k<N; k++)
         temp += a[i][k] * b[k][j];
      c[i][j] = temp;
   }
}
```

Matrix Multiplication: OpenMP Version

```
#pragma omp parallel for private(temp), schedule(static)
for (i=0; i<N; i++) {
   for (j=0; j<N; j++) {
      temp = 0;
      for (k=0; k<N; k++)
         temp += a[i][k] * b[k][j];
      c[i][j] = temp;
   }
}
```

Parallel for loop index is private by default

static: all the threads are allocated iterations before they execute the loop iterations. The iterations are divided among threads equally by default.

Loop Scheduling

- schedule clause determines **how loop iterations are divided among the thread team**
  - static([chunk]) divides iterations statically between threads
    - Each thread receives [chunk] iterations, rounding as necessary to account for all iterations
    - Default [chunk] is ceil(#iterations/#threads)

Loop Scheduling

- dynamic([chunk]) allocates [chunk] iterations per thread, allocating an additional [chunk] iterations when a thread finishes
  - Forms a logical work queue, consisting of all loop iterations
  - Default [chunk] is 1
- guided([chunk]) allocates dynamically, but [chunk] is exponentially reduced with each allocation

Loop Scheduling: Static

```
#pragma omp parallel for
schedule(static)
for( i=0; i<16; i++ )
{
  doIteration(i);
}
```

```
int chunk = 16/T;
int base = tid * chunk;
int bound = (tid+1)*chunk;

for( i=base; i<bound; i++ )
{
  doIteration(i);
}

Barrier();
```

Loop Scheduling: Dynamic

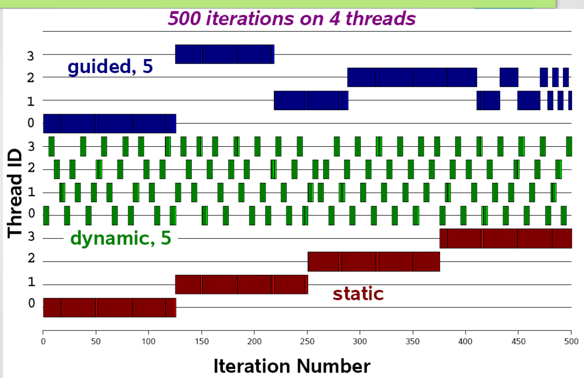
```
#pragma omp parallel for
schedule(dynamic)
for( i=0; i<16; i++ )
{
  doIteration(i);
}
```

```
int current_i;

while( workLeftToDo() )
{
  current_i = getNextIter();
  doIteration(current_i);
}

Barrier();
```

Loop Scheduling

- Static:
  - each thread gets approximately the same number of iterations as any other thread
  - each thread can independently determine the iterations assigned to it
- Dynamic:
  - threads receive varying or unpredictable amounts of work for each iteration
  - useful if the threads arrive at the for construct at varying times (nested omp)
  - with synchronization for each assignment
- Guided:
  - When threads may arrive at varying times at a for construct with each iteration requiring about the same amount of work.

Loop Scheduling



Nested Control Structures

```
#pragma omp parallel default(none)\
       shared(n,a,b,c,d) private(i)
 {
   #pragma omp for nowait
     for (i=0; i<n-1; i++)
         b[i] = (a[i] + a[i+1])/2;
   #pragma omp for nowait
     for (i=0; i<n; i++)
         d[i] = 1.0/c[i];

 } /*-- End of parallel region --*/
```

(implied barrier)

Manual Parallelism

```
#pragma omp parallel default(none)\
            shared(n,a,b,c,d) private(i)
  {
    #pragma omp sections nowait
    {
      #pragma omp section
        for (i=0; i<n-1; i++)
            b[i] = (a[i] + a[i+1])/2;
      #pragma omp section
        for (i=0; i<n; i++)
            d[i] = 1.0/c[i];

    } /*-- End of sections --*/

  } /*-- End of parallel region --*/
```

Synchronization: barrier

```
for(I=0; I<N; I++)
   a[I] = b[I] + c[I];

for(I=0; I<N; I++)
   d[I] = a[I] + b[I]
```

Both loops are in parallel region
With no synchronization in between.
What is the problem?

Fix:

```
for(I=0; I<N; I++)
   a[I] = b[I] + c[I];

#pragma omp barrier

for(I=0; I<N; I++)
   d[I] = a[I] + b[I]
```

Synchronization: critical section

```
for(I=0; I<N; I++) {
   ......
   sum += A[I];
   ......
}
```

Cannot be parallelized if sum
is shared (speedup is reduced)

Fix:

```
for(I=0; I<N; I++) {
   ......
   #pragma omp critical
  {
     sum += A[I];
   }
   ......
}
```

An Example: Vector Dot Product

- Iterations of the parallel loop will be distributed in **equal sized blocks to each thread in the team (schedule static)**
- OpenMP reduction:
  - performs a reduction on the variables that appear in its list.
  - A private copy for each list variable is created for each thread.
  - at the end of the reduction, the reduction variable is applied to all private copies of the shared variable, and the final result is written to the global shared variable.

An Example: Vector Dot Product

```
int   i, n, chunk;
float a[100], b[100], result;

/* Some initializations */
n = 100;  chunk = 10; result = 0.0;
for (i=0; i < n; i++) {    a[i] = i * 1.0;   b[i] = i * 2.0; }

#pragma omp parallel for default(shared) private(i)  schedule(static,chunk)
reduction(+:result)
  for (i=0; i < n; i++)
    result = result + (a[i] * b[i]);
printf("Final result= %f\n",result);
```

An Example: Traveling Salesman Problem

- The map is represented as a graph with nodes representing cities and edges representing the distances between cities.
- A special node (cities) is the starting point of the tour.
- Travelling salesman problem is to find the circle (starting point) that covers all nodes with the smallest distance.
- This is a well known NP-complete problem.

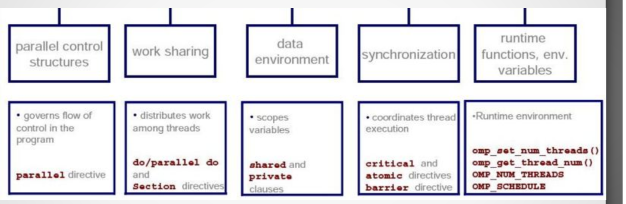
TSP: Sequential

```
init_q(); init_best();
while ((p = dequeue()) != NULL) {
   for each expansion by one city {
      q = addcity (p);
      if (complete(q)) {update_best(q);}
      else enqueue(q);
   }
}
```

TSP: OpenMP

```
do_work() {
  while ((p = dequeue()) != NULL) {
    for each expansion by one city {
        q = addcity (p);
        if (complete(q)) {update_best(q);}
        else enqueue(q);
    }
  }
}

main() {
   init_q(); init_best();
   #pragma omp parallel for
   for (i=0; I < NPROCS; i++)
      do_work();
}
```

OpenMP Constructs



Summary

- OpenMP provides a compact, yet powerful programming model for shared memory programming
  - It is easy to use OpenMP to create parallel programs.
- OpenMP preserves the sequential version of the program
- Developing an OpenMP program:
  - Start from a sequential program
  - Identify the code segment that takes most of the time.
  - Determine whether the important loops can be parallelized
    - The loops may have critical sections, reduction variables, etc
  - Determine the shared and private variables.
  - Add directives

Summary

- The performance of an OpenMP program is somewhat hard to understand.
- Is the performance issue more related to the fundamental way that we write parallel program?
  - OpenMP programs begin with sequential programs.
  - May need to find a new way to write efficient parallel programs in order to really solve the problem.

Summary

- OpenMP does not parallelize dependencies
  - Nasty race conditions still exist!
- OpenMP is not guaranteed to divide work optimally among threads
  - Programmer-tweakable with scheduling clauses
  - Rewrite...