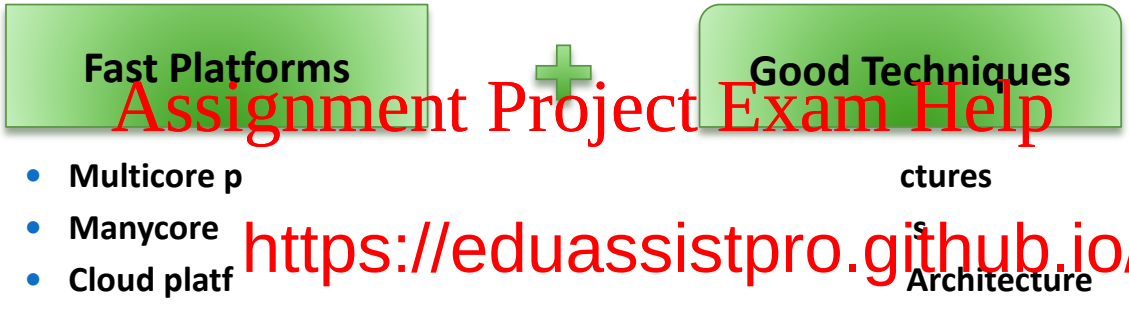
Carnegie Mellon University

Ian Lane

What we discussed last time:



- Highlighted the difference between mu core platforms
- Discussed the multicore and manycore platform intricacies

- **Exposing** concurrency in the k-means algorithm
- **Exploiting** parallelism by **exploring** mappings from application to platform

Example 7: Sequential Optimization

```
#include <omp.h>
#include <stdio.h>
static long num_steps = 100000000; double step;
#define NUM_THREADS 2
int main ()
{
     int i; double p
     step = 1.0/(dou
     #pragma omp par                                  ate(x, local_sum)
     {
          int i, id = omp_get_thread_num();
          int nthrds = omp_get_num_threads
          double hoop = nthrds*step;
          for (i=id, x=(i+0.5)*step;i< num_steps; i=i+nthrds) {
            x += hoop;
            local_sum += 4.0/(1.0+x*x);
          }
          #pragma omp critical
          pi += local_sum * step;
     }
     printf("pi = %f \n", pi);
     return 0;
}
```

Sequential: 2.12 seconds

0.46 seconds

Outline

- Multicores Shared-Memory Platforms
- OpenMP – API for Shared-Memory Multiprocessing
  - Overview
  - Example: numerical integration
  - Shared variable
  - **SPMD vs worksharing**
  - Data environment options
  - Advanced worksharing options
  - Synchronization options
- Sequential Optimizations
  - Cost Model
  - Cost Reduction

SPMD vs. Worksharing

- A parallel construct by itself creates an SPMD
  - "Single Program Multiple Data"
  - Programmer must explicitly specify what each thread must do differently
  - The division of work is hard-coded in the program
- Opportunity:
  - Many parallel regions are loops
- Question:
  - Can we make it easier to parallelize these loops?

OpenMP: Parallel For

- One of the worksharing constructs OpenMP provide is **"omp for"**

Sequential code

```
for(i=0;i<N;i++)   { a[i] = a[i] + b[i];}
```

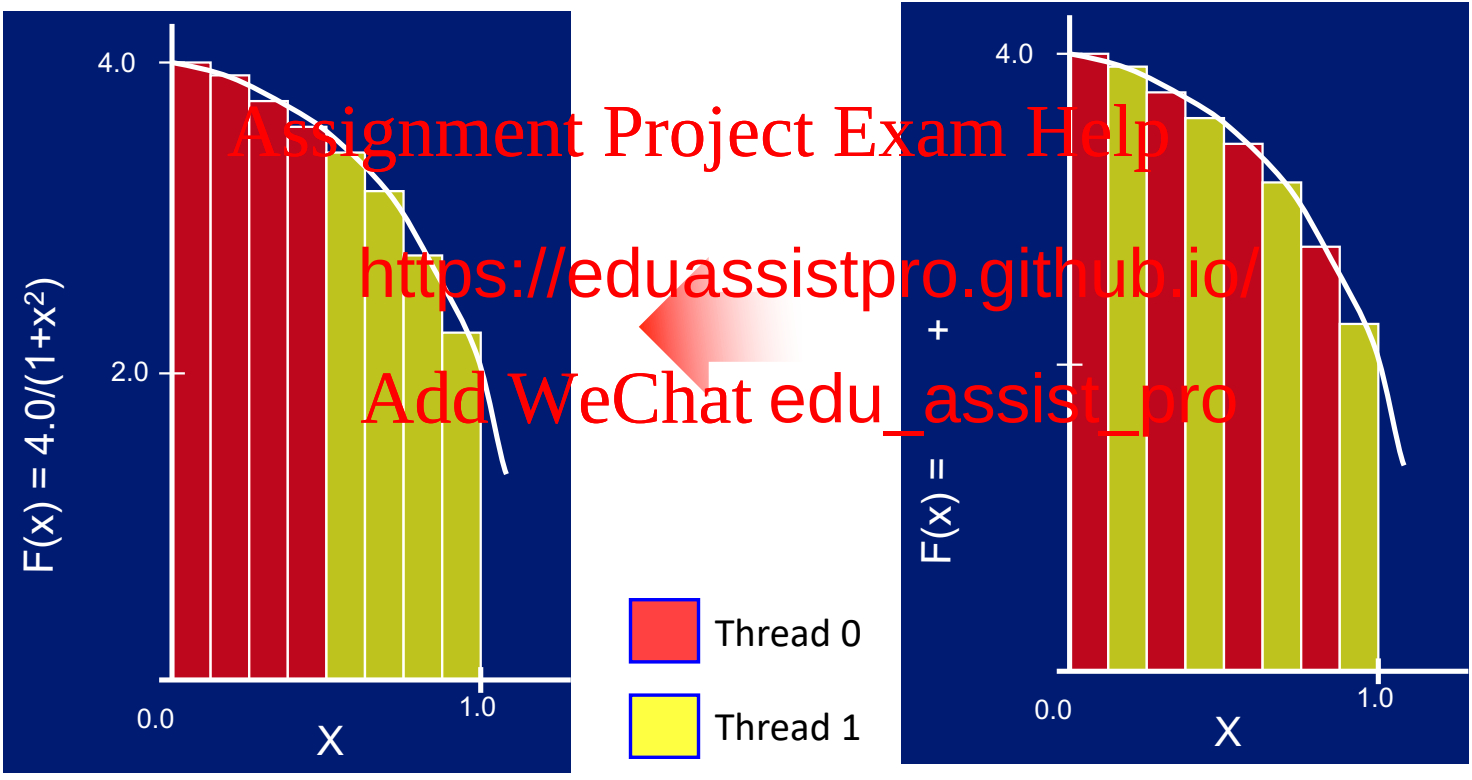

OpenMP **"parallel"** region

```
#pragma omp parallel
                                        ;
                                        ();
                Nthrds = o                              ds();
                istart = id *
                iend = (id+
                if (id == Nthrds-1)iend = N;
                for(i=istart;i<iend;i++)   { a[i] = a[i] + b[i];}
}
```

OpenMP **"parallel"** region and a worksharing **"for"** construct

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

Other Concurrency Opportunities



Example 8: Parallel For and Reduction

```
#include <omp.h>
#include <stdio.h>
static long num_steps = 100000000; double step;
#define NUM_THREADS 2
int main ()
{
     int i; double p
     step = 1.0/(dou
     #pragma omp par
     {
          #pragma omp for reduction(+:sum)
          for (i=0;i< num_steps; i++){
               x = (i+0.5)*step;
               sum = sum + 4.0/(1.0+x*x);
          }
     }
     Pi = sum * step;
     printf("pi = %f \n", pi);
     return 0;
}
```


Sequential: 2.12 seconds

1.22 seconds

Reduction is more scalable than critical sections.

"i" is private by default

Loops must not have loop carry dependency. **Limitation of compiler technology.**

Combined Parallel/Worksharing Construct

- OpenMP shortcut: Put the "parallel" and the worksharing directive on the same line


```
double  res[MAX];
#pragma omp pa
{
   #pragma omp for
   for (i=0;i< MAX; i++) {
       res[i] = huge();
   }
}
```



```
res[MAX]; int i;
a omp parallel for
    i< MAX; i++) {
    = huge();
}
```

OpenMP: Working With Loops

- Basic approach
  - Find compute intensive loops
  - Make the loop iterations independent . So they can safely execute in any order without loop-carried dependencies
  - Place the appro


```
double hoop
for (i=id, x=(i+0.5)*step;i< num                    s) {
  x += hoop;
  sum = sum + 4.0/(1.0+x*x);
}
```

x depend on i not in for

```
for (i=0;i< num_steps; i++){
     x = (i+0.5)*step;
     sum = sum + 4.0/(1.0+x*x);
}
```

OpenMP: Reductions

```
for (i=0;i< num_steps; i++){
    x = (i+0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
}
```

```
#pragma omp for r
for (i=0;i< num_s
    x = (i+0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
}
```

reduction (op : list)

1. A local copy of each list variable is made and initialized depending on g. 0 for "+")
2. Compute the local copy
3. Update the global copy are reduced into a combined with the alue

- Accumulating values into a single variable (sum) creates true dependence between loop iterations that can't be trivially removed
- This is a very common situation … it is called a "reduction".
- Support for reduction operations is included in most parallel programming environments.

OpenMP: Reduction Operators

- Many different associative operands can be used with reduction:
- Initial values are the ones that make sense mathematically.

Operator	Initial value
+	0
*	1
-	0

	Initial value
	~0
	0
^	0
&&	1
||	0

Example 8: Parallel For and Reduction

```
#include <omp.h>
#include <stdio.h>
static long num_steps = 100000000; double step;

int main ()
{
     int i; double p
     step = 1.0/(dou
     #pragma omp par                    m
     for (i=0;i< num_steps; i++){
          x = (i+0.5)*step;
          sum = sum + 4.0/(1.0+x*x);
     }
     Pi = sum * step;
     printf("pi = %f \n", pi);
     return 0;
}
```

Sequential: 2.12 seconds

1.22 seconds

we created a parallel
out changing any code
and by adding 2 simple lines of text!

Disadvantage:
1) Lot's of pitfalls if you don't understand system architecture
2) The basic result may not the fastest one can do

Outline

- Multicores Shared-Memory Platforms
- OpenMP – API for Shared-Memory Multiprocessing
  - Overview
  - Example: numerical integration
  - Shared variable
  - SPMD vs worksharing
  - **Data environment options**
  - Advanced worksharing options
  - Synchronization options
- Sequential Optimizations
  - Cost Model
  - Cost Reduction

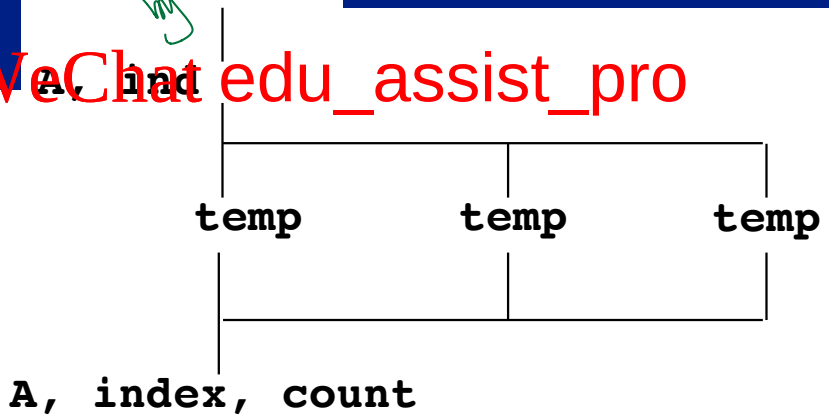
Data Environment in OpenMP

- Shared Memory programming model:
  - Most variables are shared by default
- Global variables are **SHARED** among threads
  - File scope v
  - Dynamically ew)
- But not everything is shared...
  - Functions called from parallel regions are **PRIVATE**
  - Automatic variables within a statement block are **PRIVATE**.

Example: Data Sharing

```
double A[10];
int main() {
    int index[10];
    #pragma omp parallel
        work(index)
    printf("%d\n", ind
}
```

```
extern double A[10];
void work(int *index) {
  double temp[10];
  static int count;
```

A, index and count are shared by all threads.

temp is local to each thread

Changing Storage Attributes

- One can selectively change storage attributes for constructs using the following clauses:
  - SHARED
  - PRIVATE
  - FIRSTPRIVA
- The final value of a private inside p can be transmitted to the shared variab loop with:
  - LASTPRIVATE

global before openMP, get value for each open

typically use critical / atom(...

Example 9: firstprivate example

```
#include <omp.h>
#include <stdio.h>
static long num_steps = 100000000; double step;
#define NUM_THREADS 2
int main ()
{
     int i; double pi, x, local_sum = 0.0; int nthreads;
     step = 1.0/(dou
     omp_set_num_thr
     #pragma omp par sum
     {
          int i, id = omp_get_thread_num();
          int nthrds = omp_get_num_threads();
          double hoop = nthrds*step;
          for (i=id, x=(i+0.5)*step;i< num_steps; i=i+nthrds) {
            x += hoop;
            local_sum += 4.0/(1.0+x*x);
          }
          #pragma omp critical
          pi += local_sum * step;
     }
     printf("pi = %f \n", pi);
     return 0;
}
```

Outline

- Multicores Shared-Memory Platforms
- OpenMP – API for Shared-Memory Multiprocessing
  - Overview
  - Example: numerical integration
  - Shared variable
  - SPMD vs worksharing
  - Data environment options
  - **Advanced worksharing options**
  - Synchronization options
- Sequential Optimizations
  - Cost Model
  - Cost Reduction

The Scheduling Cause

- The schedule clause affects how loop iterations are mapped onto threads
  - schedule(static [,chunk])
    - Deal-out blocks of iterations of size "chunk" to each thread.
  - schedule(dyna
    - Each thre til all iterations have been handled.
  - schedule(guided[,chunk])
    - Threads dynamically grab blocks of e of the block starts large and shrinks down to size "chunk" as the calculation proceeds.
  - schedule(runtime)
    - Schedule and chunk size taken from the OMP_SCHEDULE environment variable (or the runtime library)

The Scheduling Cause

Schedule Clause	When To Use
STATIC	programmer
DYNAMIC	Unpredictable, variable work per iteration
GUIDED	Special case of dynamic to reduce scheduling overhead

Least work at runtime : scheduling done at compile-time

Most work at runtime : complex scheduling logic used at run-time

Example 9: Scheduling

```
#include <omp.h>
#include <stdio.h>
static long num_steps = 100000000; double step;

int main ()
{
     int i; double p
     step = 1.0/(dou
     #pragma omp par                    ) schedule(static,
     10000000)
     for (i=0;i< num_steps; i++){
          x = (i+0.5)*step;
          sum = sum + 4.0/(1.0+x*x);
     }
     Pi = sum * step;
     printf("pi = %f \n", pi);
     return 0;
}
```

Sequential: 2.12 seconds

1.31 seconds

10 chunks.

Outline

- Multicores Shared-Memory Platforms
- OpenMP – API for Shared-Memory Multiprocessing
  - Overview
  - Example: numerical integration
  - Shared variable
  - SPMD vs worksharing
  - Data environment options
  - Advanced worksharing options
  - **Synchronization options**
- Sequential Optimizations
  - Cost Model
  - Cost Reduction

Synchronization: ordered

- The **ordered** region executes in the sequential order
- Important for some scientific code and optimization code
  - Order of reduction may cause rounding differences


```
#pragma om
#pragma omp for ordered reducti
      for (I=0;I<N;I++){
            tmp = NEAT_STUFF(I);
#pragma ordered
            res += consum(tmp);
      }
```

Synchronization Barrier

- **Barrier**: Each thread waits until all threads arrive


```
#pragma omp parallel shared (A, B, C) private(id)
{
      id=omp_get_
      A[id] = big_c
#pragma omp barrier
#pragma omp for
      for(i=0;i<N;i++){C[i]=big_calc3(i,A);}
#pragma omp for nowait
      for(i=0;i<N;i++){ B[i]=big_calc2(C,  i); }
      A[id] = big_calc4(id);
}
```

it barrier at the end of
orkSharing construct

it barrier due to
nowait

Implicit barrier at the end of
a parallel region

“Single” Worksharing Construct

- The **single** construct denotes a block of code that is executed by only one thread (not necessarily the master thread).
- A barrier is implied at the end of the single block
  - can remove the barrier with a nowait clause


```
#pragma omp parallel
{
      do_many_things();
#pragma omp single
      {    exchange_boundaries();   }
      do_many_other_things();

}
```

```
                arallel
      do_many_things();
#pragma omp single nowait
      {    exchange_boundaries();   }
      do_many_other_things();

}
```

Nested Parallelism

- **Q:** Is nested parallel possible with OpenMP?
- **A:** Yes. But be sure to understand why you want to use it.


```
#include <omp.h>
#include <stdio.h>
void report_num_thre
     #pragma omp sin
     { printf("L %d: num_threads = %d\n",                    ());}
}
int main() {
     omp_set_dynamic(0);
     #pragma omp parallel num_threads(2)
     {
          report_num_threads(1);
          #pragma omp parallel num_threads(2)
          {
               report_num_threads(2);
               #pragma omp parallel num_threads(2)
                    report_num_threads(3);
          }
     }
     return(0);
}
```

http://download.oracle.com/docs/cd/E19205-01/819-5270/aewbc/index.html

Nested Parallelism

- Compiling and running this program with nested parallelism enabled produces the following (sorted) output:


```
$ export OMP_NE                                  tN
$ ./experimentN                                  NESTED=FALSE
L 1: # threads                 s in team 2
L 2: # threads in team 2                          n team 1
L 2: # threads in team 2                          n team 1
L 3: # threads in team 2                          n team 1
L 3: # threads in team 2                          n team 1
L 3: # threads in team 2
L 3: # threads in team 2
```

OpenMP Environment Variables

- OMP_SCHEDULE=algorithm
  - dynamic[, n]
  - guided[, n]
  - Runtime
  - **static[, n]**
- OMP_NUM_THREADS=num
- OMP_NESTED=TRUE|FALSE
- OMP_DYNAMIC=TRUE|FALSE

Outline

- Multicores Shared-Memory Platforms
- OpenMP – API for Shared-Memory Multiprocessing
  - Overview
  - Example: numerical integration
  - Shared variable
  - SPMD vs worksharing
  - Data environment options
  - Advanced worksharing options
  - Synchronization options
- **Sequential Optimizations**
  - **Cost Model**
  - Cost Reduction

Optimizing the Base Case

- Base case is usually the guts of the algorithm that computes on data in caches
- Should fit in regis
- Must use SIMD

Cost Model

- **Algebraic model**

  `+, -, /, *`

  `sin, cos, etc.`

- **Realistic model**
  - `+, -, /, *`
  - std libs (sin, cos, etc)
  - `array[i]`
  - `A.sum, A.prod`
  - `(double) x`
  - `func(…)`
  - `i < n,` compute
  - `i < n,` test and branch
  - { } unconditional branch

```
int func(int i) {
  return i+2;
}
struct A {
                prod;
A a
                          array[]) {
A                        .0};
  for(int i=0; i<n; ++i) {
    A.sum = A.sum +
    (double)func(array[i]);
    A.prod = A.prod *
    (double)func(array[i]);
  }
  return res;
}
```

Cost Model (cont'd)

- **Algebraic model**

  cost = **3 n**

- **Realistic model**

```
+ and *
2 int -> doubl
4 a.x
++i
i<n conditional jump
2 func() calls

cost = 19 n
```

```
int func(int i) {
  return i+2;
}

struct A {

A acc(in
  A res
  for(in                {
    A.sum = A.sum + (double)func(i);
    A.prod = A.prod * (double)func(i);
  }
  return res;
}
```


6x discrepancy in cost

Using the Cost Model to Optimize

```
int func(int i) { return i+2; }

struct A { double sum, prod; };

A acc
  A r

  for(int i=0; i<n; ++i)
    A.sum = A.sum + (dou
    A.prod = A.prod * (d
  }
  return res;

}
```


Cost = 19n

Using the Cost Model to Optimize I

```
int func(int i) { return i+2; }

struct A { double sum, prod; };

A acc
  dou
  for(int i=0; i<n; ++i)
    f = (double)(i+2);
    sum = sum + f;
    prod = prod * f;
  }
  A res = {sum, prod}
  return res;
}
```


Cost = 6n

Cost Reduction = closing down the gap

Using the Cost Model to Optimize II

```
int func(int i) { return i+2; }

struct A { double sum, prod; };

A acc
  dou
  for(int i=2; i<n+2; ++
    f = (double)(i);
    sum = sum + f;
    prod = prod * f;
  }
  A res = {sum, prod};
  return res;
}
```


Cost = 5n

Optimizations enable new optimizations

Using the Cost Model to Optimize III

```
int func(int i) { return i+2; }

struct A { double sum, prod; };

A acc(int n, int array[]) {
  dou
  #pr
  for(int i=2; i<n+2; ++
    f = (double)(i);
    sum = sum + f;
    prod = prod * f;
  }
  A res = {sum, prod};
  return res;
}
```


Cost = 3.5n

Final result: 3.5n vs 19n

Outline

- Multicores Shared-Memory Platforms
- OpenMP – API for Shared-Memory Multiprocessing
  - Overview
  - Example: numerical integration
  - Shared variable
  - SPMD vs worksharing
  - Data environment options
  - Advanced worksharing options
  - Synchronization options
- Sequential Optimizations
  - Cost Model
  - **Cost Reduction**

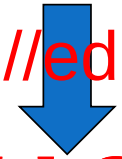
Cost Reduction: A Compiler Problem

- **Solution:** "Human Compiler"
- Compilers often fail on complex codes (many assumptions are violated)

- Optimizations
  - **Strength reduc**
  - **Function inlini**
  - **Loop unrolling**
  - **Common subexpression elimination**
  - **Load/store elimination**
  - **Table lookups**
  - **Branch elimination**

Strength Reduction

```
for i = 1..n
    sum = sum + a[i] / c;
```


```
double c_inv = 1/c;
for i = 1..n
    sum = sum + a[i] * c_inv;
```

Expensive operations:
/, %, sin, cos, log

Function Inlining

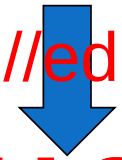
```
double f(a) { return a/c; }
.
.
for i = 1..n
```


```
for i = 1..n
    sum = sum + a[i] / c;
```

Now strength reduction can apply

Loop Unrolling

```
for i = 1..n
    sum = sum + a[i]
```


```
for i = 1:4:n
    sum = sum + a[i] + a[i+1] + a[i+2] + a[i+3]

// remaining iterations
for i = i:n
    sum = sum + a[i]
```

Often enables other optimizations

Common Sub-expression Elimination

```
for j = 1..m
  for i = 1..n
    sum = sum + cos(j*PI/180)*a[i];
```

```
for j = 1..m
  double c = cos(j*PI/180);
  for i = 1..n
    sum = sum + c*a[i];
```

Table Lookups

```
for j = 1..m
  for i = 1..n
    sum = sum + cos(j*PI/180)*a[i];
```

```
for j = 1..m
  double c = COS_TAB[j];
  for i = 1..n
    sum = sum + c*a[i];
```

Load/Store Elimination

Loop Merging


```
for i = 1..n
    sum = sum + f(a[i]);
for i
    p
```


```
for i = 1..n
    sum = sum + f(a[i]);
    prod = prod * f(a[i]);
```

One of the most important optimizations!
Almost always enables other optimizations.

Branch Elimination

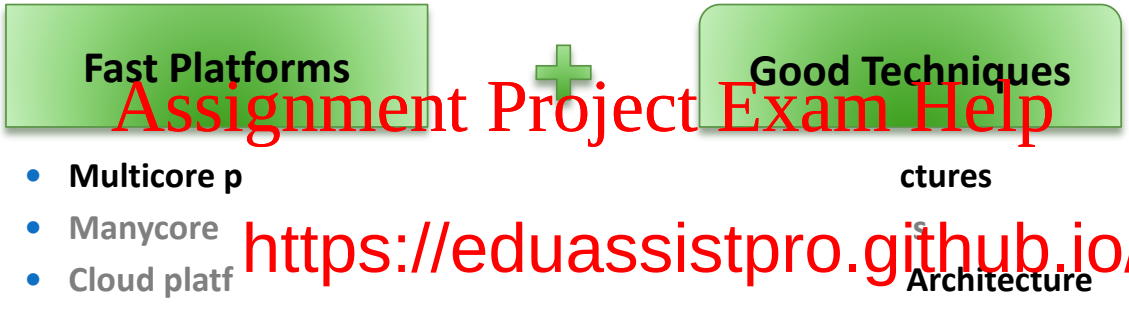
```
for i = 1..n
   if(i != except)
      sum = sum + a[i];
```



```
for i = 1..except-1
   sum = sum + a[i];
for i = except+1..n
   sum = sum + a[i];
```

```
= 0.0;
sum = sum + a[i];
```

What we have discussed in the last 2 lectures



- **OpenMP Abstractions:**
  - Help establish a mental model for the programmers
  - Make it easier to write parallel code
  - **Performance depend on deep understanding of the implementation platform**