MODULE 03 – HIGH PERFORMANCE COMPUTING

MULTITHREAD PROGRAMMING

HYUN MIN KANG @ UNIVERSITY OF MICHIGAN

A naïve matrix multiplication

- For a *m x n* matrix *A* and *n x k* matrix *B*, consider calculating a matrix *C* = *AB*.

- Then, C_{ij} is

$$C_{ij} = \sum_{s=1}^{n} A_{is} B_{sj}$$

- What is the time complexity to calculate *C*?

An example matrix multiplication

$$\begin{pmatrix} 1 & 2 & 3 & \dots & n \\ 1 & 2 & 3 & \dots & n \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & 2 & 3 & \dots & n \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1/2 & 1/2 & 1/2 & \dots & 1/2 \\ 1/3 & 1/3 & 1/3 & \dots & 1/3 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1/n & 1/n & 1/n & \dots & 1/n \end{pmatrix}$$

- What would be the resulting matrix?
- What would be the sum of elements of the resulting matrix?

Implementing the naïve multiplication

```
int main(int argc, char** argv) {
  int32_t n1, n2, n3;
  int32_t i, j, k;
  n1 = atoi(argv[1]);
  n2 = atoi(argv[2]);
  n3 = atoi(argv[3]);
  printTime("Initializing %d by %d matrix", n1, n2);
  vector< vector<double> > A;
  A.resize(n1);
  for(i=0; i < n1; ++i) {
    A[i].resize(n2);
    for(j=0; j < n2; ++j)
      A[i][j] = j+1.0;
  }
  ...
```

```
  printTime("Initializing %d by %d matrix", n2, n3);
  vector< vector<double> > B;
  B.resize(n2);
  for(i=0; i < n2; ++i) {
    B[i].resize(n3);
    for(j=0; j < n3; ++j) B[i][j] = 1.0/(i+1.);
  }
  printTime("Performing a naive matrix multiplication");
  vector< vector<double> > C;
  C.resize(n1);
  double sum = 0;
  for(i=0; i < n1; ++i) {
    C[i].resize(n3, 0);
    for(j=0; j < n3; ++j) {
      for(k=0; k < n2; ++k) C[i][j] += ( A[i][k] * B[k][j] );
      sum += C[i][j];
    }
  }
  printTime("Finished a naive matrix multiplication. Sum = %lg", sum);
  return 0;
}
```

Modified printTime() function

```
#include <ctime>
#include <sys/time.h>
#include <cstdarg>
void printTime(const char* msg, ...) { // advanced syntax to allow variable
  va_list ap;                          // number of function arguments
  va_start(ap, msg);
  char buff[255];
  struct timeval tv;
  gettimeofday(&tv, NULL);
  time_t current_time = tv.tv_sec;
  strftime(buff, 120, "%Y/%m/%d %H:%M:%S", localtime(&current_time));
  fprintf(stderr,"[%s.%06d]\t", buff, tv.tv_usec);
  vfprintf(stderr, msg, ap);
  fprintf(stderr,"\n");
  va_end(ap);
}
```

A running example : 131s

```
$ ./mat_mult 1000 10000 1000
[2016/11/09 01:12:32.751429]  Initializing 1000 by 10000 matrix
[2016/11/09 01:12:32.819379]  Initializing 10000 by 1000 matrix
[2016/11/09 01:12:32.868526]  Performing a naive single-threaded (1000 x 10000) by (10000 x 1000) matrix multiplication
[2016/11/09 01:14:43.347102]  Finished performing a naive single-threaded matrix multiplication. Sum = 1e+10
```

Parallelizing the computation

- The calculation of C_{ij}

$$C_{ij} = \sum_{s=1}^{n} A_{is} B_{sj}$$

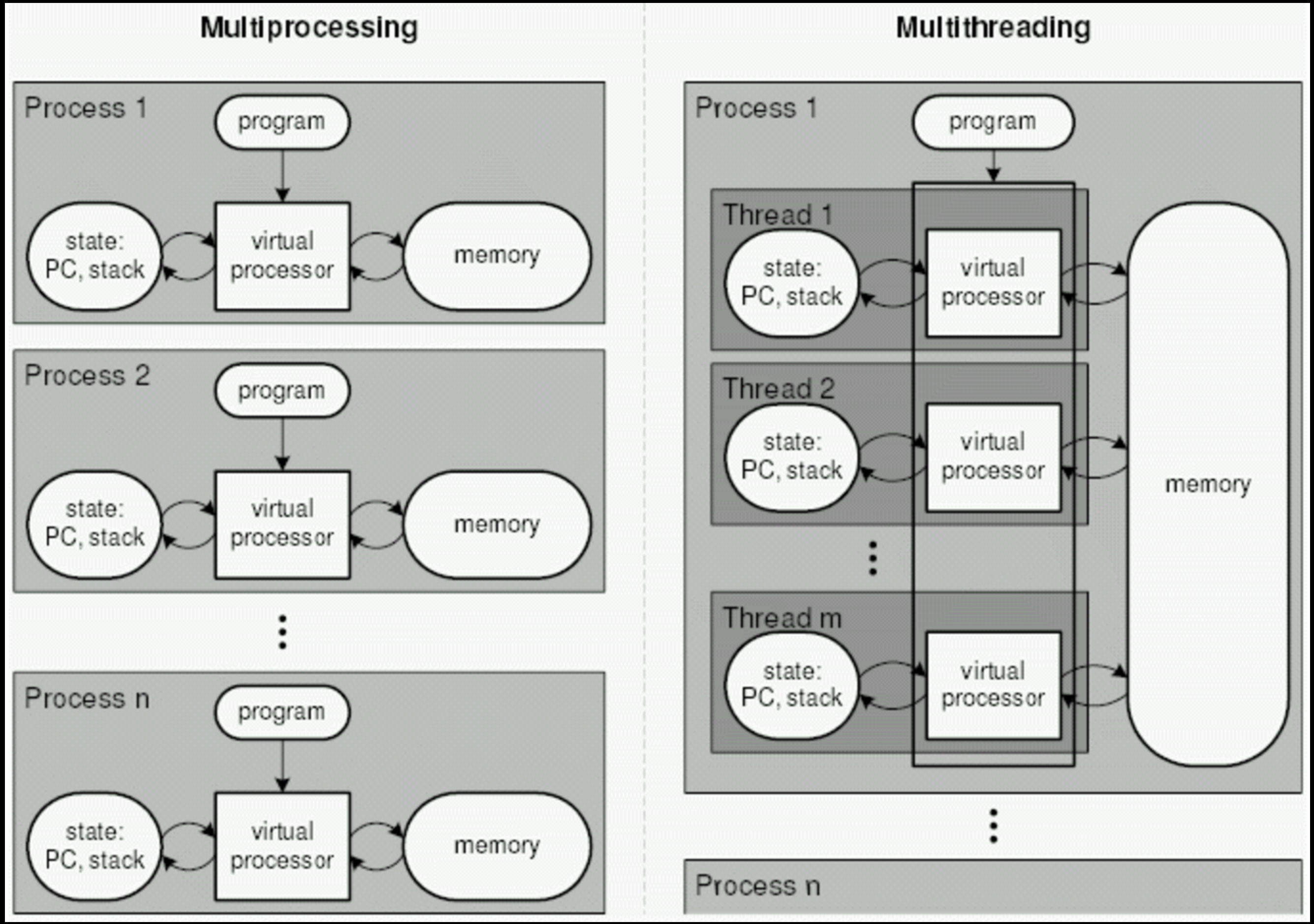
does not depend on other elements in C

- Can we use multiple CPUs to parallelize the process?

Multithreading

- A programming model that allows to use multiple CPUs in a single process.
- Threads run independently, but share the resources of the same process.
- Advantages: Efficiency
- Disadvantages: Programming Difficulty & Synchronization

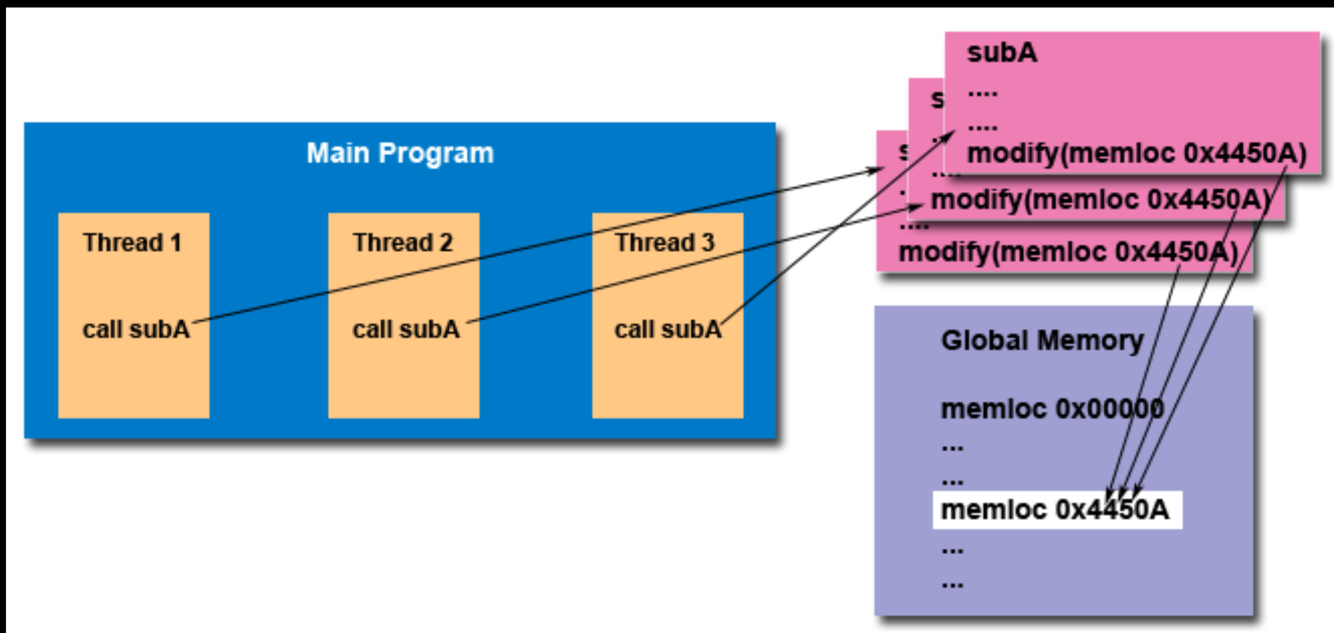
Multiprocessing vs. Multithreading



http://www.fmc-modeling.org/category/projects/apache/amp/A_1_Unix_Processes.html

POSIX Threads (pthreads)

- A language-independent model for multithreading
- A thread is implemented as a subroutine (function)



https://computing.llnl.gov/tutorials/pthreads/images/threadUnsafe.gif

Steps for using POSIX threads

1. Define a number of threads with type `pthread_t`

2. Create a number of threads using

```
pthread_create( pthread_t* thread, pthread_attr_t* attr,
                void *(*func) (void*), void* arg )
```

   - `thread` should be the pointer to each thread to create
   - `func` is the function to execute as the thread
   - arg is the pointer to the arguments of the function, but should not have a type

3. Wait for the threads to end using

```
pthread_join(pthread_t thread, void** value_ptr)
```

pthread matrix multiplication : arguments

```
class mult_thread_args {
public:
   vector< vector<double> >* pA;
   vector< vector<double> >* pB;
   vector< vector<double> >* pC;
   int32_t from_n1;
   int32_t to_n1;
   int32_t from_n3;
   int32_t to_n3;
   mult_thread_args(vector< vector<double> >* _pA,
        vector< vector<double> >* _pB,
        vector< vector<double> >* _pC,
        int32_t _from_n1, int32_t _to_n1, int32_t _from_n3,
       int32_t _to_n3) : pA(_pA), pB(_pB), pC(_pC),
   from_n1(_from_n1), to_n1(_to_n1), from_n3(_from_n3),
to_n3(_to_n3) {}
};
```

Define the function to invoke

```
void* mult_thread(void* args) {
  mult_thread_args* mt_args = (mult_thread_args*)args;
  const vector< vector<double> >& A = *mt_args->pA;
  const vector< vector<double> >& B = *mt_args->pB;
  vector< vector<double> >& C = *mt_args->pC;
  int32_t i, j, k;
  int32_t n2 = (int32_t)B.size();
  for(i=mt_args->from_n1; i < mt_args->to_n1; ++i) {
    for(j=mt_args->from_n3; j < mt_args->to_n3; ++j) {
      for(k=0; k < n2; ++k)
     C[i][j] += ( A[i][k] * B[k][j] );
    }
  }
  return NULL;
}
```

Parallelization by `pthread`

```
pthread_t thread[4];
mult_thread_args args1(&A, &B, &C, 0, n1/2, 0, n3/2);
if ( pthread_create(&thread[0], NULL, mult_thread, &args1) != 0 ) perror("Can't create");

mult_thread_args args2(&A, &B, &C, 0, n1/2, n3/2, n3);
if ( pthread_create(&thread[1], NULL, mult_thread, &args2) != 0 ) perror("Can't create");

mult_thread_args args3(&A, &B, &C, n1/2, n1, 0, n3/2);
if ( pthread_create(&thread[2], NULL, mult_thread, &args3) != 0 ) perror("Can't create");

mult_thread_args args4(&A, &B, &C, n1/2, n1, n3/2, n3);
if ( pthread_create(&thread[3], NULL, mult_thread, &args4) != 0 ) perror("Can't create");

for(i=0; i < 4; ++i) {
  pthread_join(thread[i], NULL);
}
```

A running example : 30s

```
$ ./mat_mult 1000 10000 1000
```

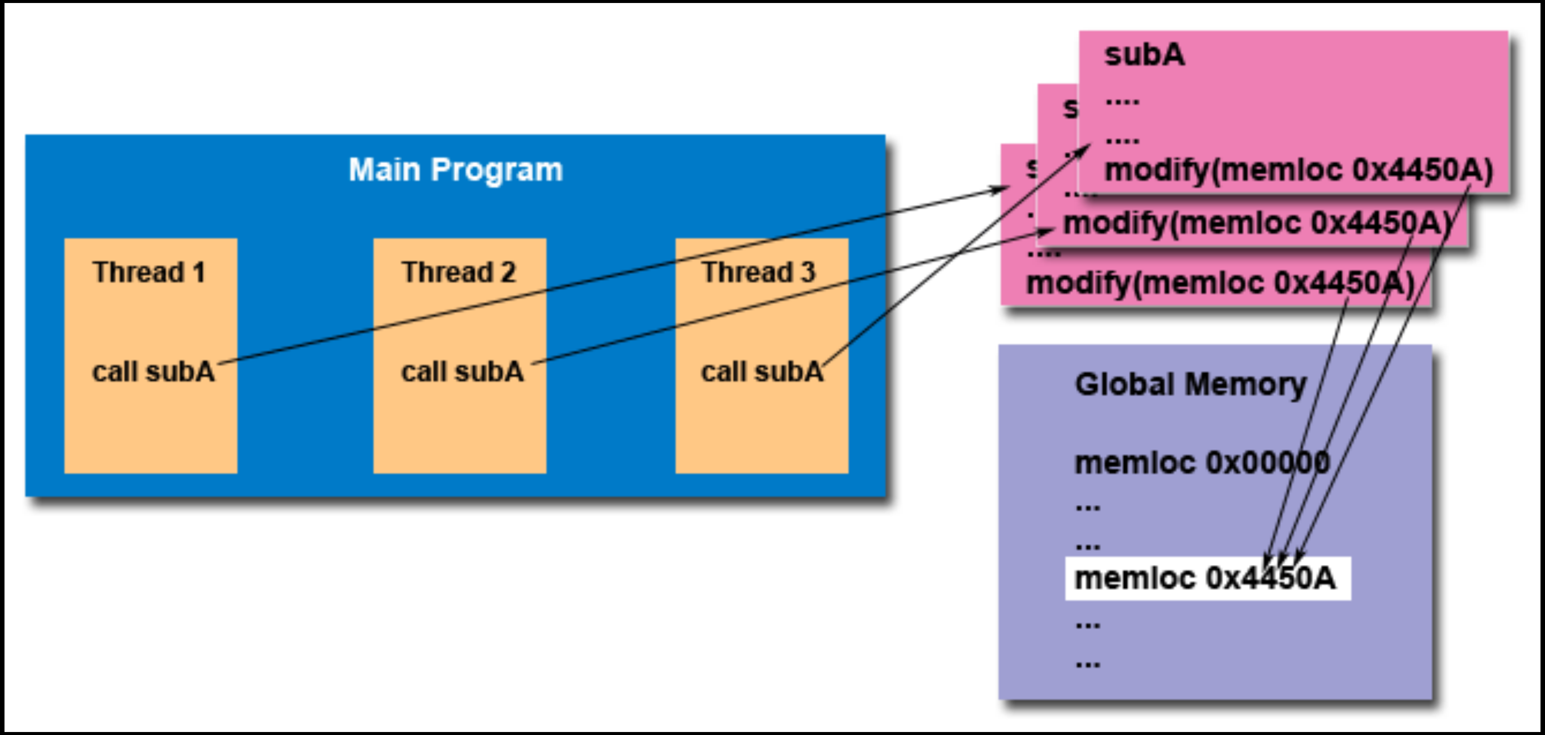
- [2016/11/09 02:49:44.367431] Performing a naive 4-pthreaded (1000 x 10000) by (10000 x 1000) matrix multiplication
- [2016/11/09 02:50:14.271771] Finished performing a naive 4-threaded matrix multiplication. Sum = 1e+10

Another possible way to parallelize

```
void* mult_thread(void* args) {
  mult_thread_args* mt_args = (mult_thread_args*)args;
  const vector< vector<double> >& A = *mt_args->pA;
  const vector< vector<double> >& B = *mt_args->pB;
  vector< vector<double> >& C = *mt_args->pC;
  int32_t i, j, k;
  int32_t n1 = (int32_t)A.size(), n3 = (int32_t)B[0].size(),
  for(i=0; i < n1; ++i) {
    for(j=0; j < n3; ++j) {
      for(k=mt_args->from_n2; k < mt_args->to_n2; ++k)
      C[i][j] += ( A[i][k] * B[k][j] );
    }
  }
  return NULL;
}
```

Do we see any problem?

Race condition : Challenges in multithreading



- To resolve race condition, “mutex” is used to lock variables.

OpenMP : An easier way of multithreading

```
#include <omp.h>
...
 printTime("Performing a naive 4 omp-threaded
    (%d x %d) by (%d x %d) matrix multiplication", n1, n2, n2, n3);
 omp_set_num_threads(4);
#pragma omp parallel for private(i,j,k) schedule(dynamic)
 for(i=0; i < n1; ++i) {
   for(j=0; j < n3; ++j) {
     for(k=0; k < n2; ++k)
       C[i][j] += ( A[i][k] * B[k][j] );
     sum += C[i][j];
   }
 }
printTime("Finished performing a naive omp-threaded
          matrix multiplication. Sum = %lg", sum);
```

Compiling with OpenMP & running : 34s

```
$ g++ -fopenmp -O2 -o mat_mult mat_mult.cpp -std=c++0x

$ ./mat_mult 1000 10000 1000
[2016/11/09 05:23:19.077628]  Performing a naive 4 omp-threaded
(1000 x 10000) by (10000 x 1000) matrix multiplication
[2016/11/09 05:23:53.417607]  Finished performing a naive omp-
threaded matrix multiplication. Sum = 9.3e+09
```

What was wrong with the code?

Resolving race condition with OpenMP

```
#include <omp.h>
...
 printTime("Performing a naive 4 omp-threaded
    (%d x %d) by (%d x %d) matrix multiplication", n1, n2, n2, n3);
 omp_set_num_threads(4);
#pragma omp parallel for private(i,j,k) schedule(dynamic)
 for(i=0; i < n1; ++i) {
   for(j=0; j < n3; ++j) {
     for(k=0; k < n2; ++k)
       C[i][j] += ( A[i][k] * B[k][j] );
#pragma omp critical
     sum += C[i][j];
   }
 }
printTime("Finished performing a naive omp-threaded
           matrix multiplication. Sum = %lg", sum);
```

Results with race condition resolution

```
$ ./mat_mult 1000 10000 1000
[2016/11/09 05:36:47.281005]  Performing a naive 4 omp-threaded (1000 x 10000) by (10000 x 1000) matrix multiplication
[2016/11/09 05:37:21.578820]  Finished performing a naive omp-threaded matrix multiplication. Sum = 1e+10
```

Summary : Multithread Programming

- **An efficient way to parallelize a program utilizing multiple CPUs.**
- **POSIX thread provides a standard method for multithread programming across many environments.**
- **OpenMP provides an easier way to parallelize a loop or a block.**
- **Multi-thread programming requires a very careful attention of race condition on each variable used within a thread.**
  - Simultaneous update may result in unexpected results, and much be locked properly to execute exclusively for each thread