Shared Memory Programming with OpenMP (1)

2014 Spring

Jinkyu Jeong
(jinkyu@skku.edu)
OpenMP

- An API for shared-memory parallel programming.
- MP = multiprocessing
- Designed for systems in which each thread or process can potentially have access to all available memory.
- System is viewed as a collection of cores or CPU’s, all of which have access to main memory.
Parallel Programming Practice

- **Current**
  - Start with a parallel algorithm
  - Implement, keeping in mind
    - Data races
    - Synchronization
    - Threading syntax
  - Test & Debug
  - Debug ....

- **Ideal way**
  - Start with some algorithm
  - Implement serially, ignoring
    - Data races
    - Synchronization
    - Threading syntax
  - Test & Debug
  - *Auto-magically* parallelize
Implementation on Shared Memory

- **Thread Library**
  - Library calls
  - Low level programming
    - Explicit thread creation & work assignment
    - Explicit handling of synchronization
  - Parallelism expression
    - Task: create/join thread
    - Data: detailed programming
  - Design concurrent version from the start

- **OpenMP**
  - Compiler directives
  - Higher abstraction
    - Compilers convert code to use OpenMP library, which is actually implemented with thread APIs
  - Parallelism expression
    - Task: task/taskwait, parallel sections
    - Data: parallel for
  - Incremental development
    - Start with sequential version
    - Insert necessary directives
Implementation Examples

- **Threaded functions**
  - Exploit data parallelism

```c
node A[N], B[N];
main() {
    for (i=0; i<nproc; i++)
        thread_create(par_distance);
    for (i=0; i<nproc; i++)
        thread_join();
}
void par_distance() {
    tid = thread_id();
    n = ceiling(N/nproc);
    s = tid * n;
    e = MIN((tid+1)*n, N);
    for (i=s; i<e; i++)
        for (j=0; j<N; j++)
            C[i][j] = distance(A[i], B[j]);
}
```

- **Parallel loops**
  - Exploit data parallelism

```c
node A[N], B[N];
#pragma omp parallel for
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        C[i][j] = distance(A[i], B[j]);
```
Implementation on Distributed Memory

- **MPI (message passing interface)**
  - Language independent communication library
  - Freely available implementation
    - MPICH (Argonne Lab), Open MPI

```c
/* processor 0 */
node A[N], B[N];

Dist_calc0() {
    Send (A[N/2 .. N-1], B[0 .. N-1]);

    for (i=0; i<N/2; i++)
        for (j=0; j<N; j++)
            C[i][j] = distance(A[i], B[j]);

    Recv (C[N/2 .. N-1][0 .. N-1]);
}

/* processor 1 */
node A[N], B[N]; /* duplicate copies */

Dist_calc1() {
    Recv (A[N/2 .. N-1], B[0 .. N-1]);

    for (i=N/2; i<N; i++)
        for (j=0; j<N; j++)
            C[i][j] = distance(A[i], B[j]);

    Send (C[N/2 .. N-1][0 .. N-1]);
}
```
OpenMP Solution Stack

User
Application
Layer

OpenMP
Program
Layer

System
Layer

User
Application

Directives, Compiler
OpenMP Library
Environment Variables

Runtime Library
OS/System support with Shared memory
Pragmas

- **Special preprocessor instructions**
  - `#pragma`
  - Provides extension to the basic C
  - Compilers that don’t support the pragmas ignore them

- **OpenMP pragmas**
  - `#pragma omp parallel`
  - Most basic parallel directive in openMP
  - Additional clauses follow the directive
Hello World in OpenMP

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

void Hello(void); /* Thread function */

int main(int argc, char* argv[]) {
  /* Get number of threads from command line */
  int thread_count = strtol(argv[1], NULL, 10);

  #pragma omp parallel num_threads(thread_count)
  Hello();

  return 0;
} /* main */

void Hello(void) {
  int my_rank = omp_get_thread_num();
  int thread_count = omp_get_num_threads();

  printf("Hello from thread %d of %d\n", my_rank, thread_count);
} /* Hello */
Hello World in OpenMP – con’t

- Compile
  ```
  gcc -g -Wall -fopenmp -o omp_hello omp_hello.c
  ```

- Run
  ```
  ./omp_hello 4
  ```

Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4

possible outcomes

Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4

Hello from thread 3 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4

compiling

running with 4 threads
Clause

- Text that modifies a directive.
- num_threads clause
  - Specifies the number of threads to run in parallel
  
  ```c
  # pragma omp parallel num_threads(thread_count)
  ```
  - Threads execute the following structured block of code

```c
int main(int argc, char* argv[]) {
    /* Get number of threads from command line */
    int thread_count = strtol(argv[1], NULL, 10);

    # pragma omp parallel num_threads(thread_count)
    Hello();
}
```
OpenMP Programming Model

- Fork-join model
  - `#pragma omp parallel num_threads(n)`
  - One master thread
  - Forks \( n-1 \) slave threads
  - \( n \) threads comprise a team

- Implicit barrier
  - When parallel execution completes, threads in a team have an implicit barrier
Error Checking

- When a compiler does not support openMP
  - #pragma clauses
    - No problem because compiler ignores them
  - OpenMP-related functions
    - Generates compilation errors

```c
void Hello(void) {
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();
    printf("Hello from thread %d of %d\n", my_rank, thread_count);
} /* Hello */
```

- “omp.h” does not exist in include directories
Error Checking – con’t

- When a compiler supports openMP
- And, a program is compiled with -fopenmp option
  - _OPENMP is defined automatically
  - Build a safe code using #ifdef _OPENMP

```c
#include <omp.h>

#define _OPENMP
#include <omp.h>
#endif

#ifndef _OPENMP
int my_rank = 0;
int thread_count = 1;
#else
int my_rank = omp_get_thread_num(

int thread_count = omp_get_num_threads(
#endif
```
MUTUAL EXCLUSION
(THE TRAPEZOIDAL RULE)
The Trapezoidal Rule

- Serial algorithm

```c
/* Input:  a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++) {
    x_i = a + i*h;
    approx += f(x_i);
}
approx = h*approx;
```
Using Foster’s methodology

• Task partitioning: two types of tasks
  – Computing the area of individual trapezoids
  – Adding the areas of trapezoids

• Communication:
  – No communication for the first task
  – Adding values (like reduction) for the second task

• Aggregation
  – Each thread calculates the area of contiguous trapezoids

• Mapping
  – Single thread to each core
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void Trap(double a, double b, int n, double* global_result_p);

int main(int argc, char* argv[]) {
    double global_result = 0.0;  /* Store result in global_result */
    double a, b;  /* Left and right endpoints */
    int n;  /* Total number of trapezoids */
    int thread_count;

    thread_count = strtol(argv[1], NULL, 10);
    printf("Enter a, b, and n\n\n");
    scanf("%lf %lf %d", &a, &b, &n);
    # pragma omp parallel num_threads(thread_count)
    Trap(a, b, n, &global_result);

    printf("With n = %d trapezoids, our estimate\n", n);
    printf("of the integral from %f to %f = %.14e\n", 
            a, b, global_result);
    return 0;
}  /* main */
void Trap(double a, double b, int n, double* global_result_p) {
    double h, x, my_result;
    double local_a, local_b;
    int i, local_n;
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();

    h = (b-a)/n;
    local_n = n/thread_count;
    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    my_result = (f(local_a) + f(local_b))/2.0;
    for (i = 1; i <= local_n-1; i++) {
        x = local_a + i*h;
        my_result += f(x);
    }
    my_result = my_result*h;

    *global_result_p += my_result;
}

race condition
Mutual Exclusion

- **Thread API**

```c
main() {
    for (i=0; i<nproc; i++)
        thread_create(par_dequeue);
    for (i=0; i<nproc; i++)
        thread_join();
}

void par_dequeue() {
    tid = thread_id();
    n = ceiling(N/nproc);
    s = tid * n; e = MIN((tid+1)*n, N);
    for (i=s; i<e; i++) {
        if (x[i] == cond) {
            thread_mutex_lock(queue_lock);
            dequeue(x[i]);
            thread_mutex_unlock(queue_lock);
        }
    }
}
```

- **OpenMP**

- Exclusive accesses among the team of threads
- `#pragma omp` structured block
- `#pragma omp atomic` statement

```c
main() {
    #pragma omp parallel for
doctor (i=0; i<N; i++) {
    for (i=0; i<nproc; i++)
        thread_create(par_dequeue);
    for (i=0; i<nproc; i++)
        thread_join();
}
```

```c
main() {
    #pragma omp parallel for
doctor (i=0; i<N; i++) {
        if (x[i] == cond)
            #pragma omp critical
            dequeue(x[i]);
    }
```
Trapezoidal Rule using OpenMP

```c
void Trap(double a, double b, int n, double* global_result_p) {
    double h, x, my_result;
    double local_a, local_b;
    int i, local_n;
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();

    h = (b-a)/n;
    local_n = n/thread_count;
    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    my_result = (f(local_a) + f(local_b))/2.0;
    for (i = 1; i <= local_n-1; i++) {
        x = local_a + i*h;
        my_result += f(x);
    }
    my_result = my_result*h;

    # pragma omp critical
    *global_result_p += my_result;
}
```
SCOPE OF VARIABLES
Scope of Variables

- In serial programming
  - Scope of variables is part of a language
    - Ex. Variables declared in a structured block is only visible inside the block

- In OpenMP
  - The scope of a variable refers to the set of threads that can access the variable in a parallel block
    - **Shared** scope
      - All threads in a team can access
    - **Private** scope
      - A variable is only accessible to each thread

```c
main() {
    int a, b;
    a = b;
    {
        int x;
    }
}
```
**Scope in OpenMP**

- **Default scope for variables declared before a parallel block is shared**
  
  ```c
  main() {
    int a, b; \Rightarrow \text{shared in threads}
    a = b;
    # \text{pragma omp parallel num_threads}(n)
    {
      int x; \Rightarrow \text{private in threads}
    }
  }
  ```

- **Data scoping semantics are somewhat complicated**
  - private, shared, copyin, firstprivate, lastprivate, copyprivate, threadprivate, ...
  - Implicit rules,...
THE REDUCTION CLAUSE
Global Sum in Trapezoidal Rule

- Task partitioning to threads
- Global sum of each thread’s result

Master thread

fork

• Each thread calculate the area of trapezoids assigned

join

• Global sum of each thread’s result
Global Sum in Trapezoidal Rule

- Critical section-based global sum

```c
void Trap(double a, double b, int n, double* global_result_p);

#pragma omp critical
    *global_result_p += my_result;
/* Trap */
```

- More intuitive approach
  - Race condition should be considered

```c
global_result = 0.0;
#pragma omp parallel num_threads(thread_count)
{
    global_result += Local_trap(double a, double b, int n);
}
```

→ Serialization of Local_trap calls
Avoiding serialization of function calls

- Declaring private variable inside the parallel block
- Global sum of each thread’s results within critical section

```c
double my_result = 0.0; /* private */
my_result += Local_trap(double a, double b, int n);

#pragma omp critical
global_result += my_result;
```
Reduction in OpenMP

- **A reduction**
  - Applies the same reduction operator to a sequence of operands in order to get a single result
  - All of the intermediate results of the operation should be stored in the same variable: the reduction variable

- **Reduction clause in OpenMP**

```
reduction(<operator>: <variable list>)
```

- Reduction operators → `+`, `*`, `-`, `&`, `|`, `^`, `&&`, `||`
- Commutative and associative operators can provide correct results
Reduction in OpenMP

- Reduction variable is initialized to the identity value
  - 0 for + operators
  - 1 for * operators

- Global sum using reduction clause

```c
global_result = 0.0;
#pragma omp parallel num_threads(thread_count) 
 reduction(+: global_result)
global_result += Local_trap(double a, double b, int n);
```

- Global_result is shared. But, due to the reduction clause, the result of `local_trap()` is stored into a private variable and finally added into the `global_result` variable