2013.1 Multicore Computing
Final Exam (June 17th 9am-10am)

supervisor signature	

StudentID# : ( ) , Name : ( )

* You may answer in either Korean or English.

1. (28points) Fill out the blanks (a)~(n) with the most appropriate English words.

- (a.) is the overhead in traffic to and from memory as a result of multiple threads concurrently attempting to access the same locations in memory.
- (b.) is an atomic instruction used in multithreading to achieve synchronization. It compares the contents of a memory location to a given value and, only if they are the same, modifies the contents of that memory location to a given new value. This is done as a single atomic operation. The atomicity guarantees that the new value is calculated based on up-to-date information; if the value had been updated by another thread in the meantime, the write would (c.).
- The divide-and-conquer paradigm improves program modularity, and often leads to simple and efficient algorithms. Since the subproblems created in the divide step are often (d.), they can be solved in parallel. If the subproblems are solved recursively, each recursive divide step generates even more (d.) subproblems to be solved in parallel. In order to obtain a highly parallel algorithm, it is often necessary to parallelize the (e.) and (f.) steps, too
- GPU is specialized for highly (g.) computation, where the same program is executed on many data elements in parallel.
- In pthread programming, (h.) subroutine blocks the calling thread until the specified thread terminates. The programmer is able to obtain the target thread's termination return status if it was specified in the target thread's call to (i.).

  ※ You should fill out the blank (h) and the blank (i) with appropriate pthread library function names.

2. (12points) In pthread programming, there are several ways in which a pthread (thread) can be terminated. List at least four ways.

(i) : ( )
(ii) : ( )
(iii) : ( )
(iv) : ( )

3. (10points) Fill out the box below with description of the phase 2 in PSRS algorithm.

Suppose we are sorting n keys on p processes and around n/p elements are assigned to each process. The PSRS (Parallel Sorting by Regular Sampling) algorithm has four phases.

In phase 1, each process uses the sequential quicksort algorithm to sort its share of elements. Each process selects regular sample of the sorted list.

In phase 2,

In phase 3, each process partitions its list into p pieces using pivot values and sends partitions to other processes.
In phase 4, each process merges its partitions into a single sorted partition.

4.(16points) Insert appropriate JAVA code in the box below for implementing push/tryPush operation of lock free concurrent stack.

```
public class LockFreeStack {// assume pop is correctly written
  private AtomicReference top = new AtomicReference(null);
  public boolean tryPush(Node node){
  }
  public void push(T value) {
    Node node = new Node(value);
    while (true) {
      if (tryPush(node)) {
        return;
      } else backoff.backoff(); // wait a little (random) time.
  }}
```

```
// insert your code here
```

5. (16points) Consider following C and OpenMP code that computes the approximate value of π(pi). In the program, ten threads simultaneously execute for-loop (line 9~line 12). Fill out the box in the code with appropriate C and OpenMP code.

```
1: #include <omp.h>
2: #include <stdio.h>

3: static long num_steps = 10000000; double step;
4: #define NUM_THREADS 10

5: void main ()
6: {
7:        long i; double x, pi, sum = 0.0;
8:        step = 1.0/(double) num_steps;
```

```
9:        for (i=0;i< num_steps; i++){     //  (*)
10:                x = (i+0.5)*step;
11:                sum = sum + 4.0/(1.0+x*x);
12:       }
13:       pi = step * sum;
14:       printf("pi=%.24lf\n",pi);
15: }
```

Execution Output result:

```
pi=3.1415926536
```

6.(18points) Consider following C and CUDA code that adds two vectors using many-core GPU. Assume vector size is 12. Write a CUDA kernel function **add** in the box (a). Insert appropriate code into the box (b) and (c) that are necessary for the management of device and host memory. Note that kernel function call (line 15) generates 3 blocks and each block generates 4 threads.

```
1: #include <stdio.h>
2: #include <stdlib.h>

3: #define N 12
```

```
(a)
```

```
4: int main(void)
5: {
6:        int *a, *b, *c; // host copies of a, b, c
7:        int *d_a, *d_b, *d_c; // device copies of a, b, c
8:        int size = N * sizeof(int);

9:        cudaMalloc((void **)&d_a, size);
10:       cudaMalloc((void **)&d_b, size);
11:       cudaMalloc((void **)&d_c, size);

12:       a = (int *)malloc(size); random_ints(a, N);
13:       b = (int *)malloc(size); random_ints(b, N);
14:       c = (int *)malloc(size);
```

```
(b)
```

```
15:       add<<<3,4>>>(d_a, d_b, d_c);
```

```
(c)
```

```
16:       for (int i=0;i<N;i++) printf("a[%d]=%d , b[%d]=%d, c[%d]=%d\n"
                                                ,i,a[i],i,b[i],i,c[i]);

17:       free(a); free(b); free(c);
18:       cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
19:       return 0;
20:}
```

```
void random_ints(int* x, int size)
{
          int i;
          for (i=0;i<size;i++) {
                    x[i]=rand()%10;
          }
}
```

Execution Output result:

```
a[0]=1 , b[0]=1, c[0]=2
a[1]=7 , b[1]=7, c[1]=14
a[2]=4 , b[2]=1, c[2]=5
a[3]=0 , b[3]=1, c[3]=1
a[4]=9 , b[4]=5, c[4]=14
a[5]=4 , b[5]=2, c[5]=6
a[6]=8 , b[6]=7, c[6]=15
a[7]=8 , b[7]=6, c[7]=14
a[8]=2 , b[8]=1, c[8]=3
a[9]=4 , b[9]=4, c[9]=8
a[10]=5 , b[10]=2, c[10]=7
a[11]=5 , b[11]=3, c[11]=8
```