INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI

Department of Computer Science and Engineering

CS528 (High Performance Computing) Mid Semester Examination

Date: 20th Sept 2016 **Timing: 2 Hrs** **Full Marks: 60** **Answer all questions**

Q1 [20 Marks] [Topic: Processor Architecture and Code Optimization]

```
int A[N], B[N][128],Period=128;//Assume N=1200
for(i=0;i<N; i++)A[i]=0;
for(i=0;i<N; i++)
for(j=0;j<N; j++){
      d=(i*N+j)%Period; A[i]=A[i]+B[j][d];
}
```

A. [5 marks] Assuming a 32KB direct map L1 data cache with cache block size of 64B, calculate the number of L1 data cache miss occurs if we execute above code. (Assume no other program is executing)

Ans: Assume int data takes 4Bytes, a cache block can hold 16 data

In first for loop Accessing A[i]; 1200 access, 1 miss per 16 access, so 1200/16=75 miss, even if all element of not fitting in to L1
*In 2nd and 3rd for loop for A[i], in general it get allocate 1 block and rest 32*1024/64-1=512-1=511 block for B*
*Every time you access B you will get a miss for B, as B get access column wise and B don't fit to L1. So the number of miss for both A and B in 2nd and 3rd loop will be 1200*1200 (for B) and 75 (for A)*

*So total number of miss for whole code = 75+1200*1200+75= 1,440,150*
Run this code on Valgrind, you will get 1,430,800, which is almost similar as calculated.

B. [5 marks] Calculate the branch probability of the same code, assuming each basic operations (load, store, add, cmp, mod and mul) require one instruction (but not for the array operation).

Ans: Inside every for loop, loop structure have least these three instruction {INCrement, Compare and JUMP to level on CompareCondition}.

*Assigning a value to array at index i require these three instruction {loading address to BaseReg, one multiplication (i*4), one addition, one store at base+i*4},*
*Accessing 2D array have 5 instructions and these are: one load base address, one multiplication (d*128), one addition (d*128+j) and one another multiplication (4*(d*128+j) and load of data from memory.*
Similarly Modifiying an 1D array A[i]=A[i]+x require, 4 instructions
`d=(i*N+j)%Period; require 3 instructions`

*So 1st loop have 1200*7 instructions and have 1200 branch instructions*
*2nd loop have 1200*3 instructions and have 1200 branch instructions*
3rd loop have 1200(3+4+5) instruction and have 1200 branch instruction*

Total instruction 1200(7+3+12) and branch instructions 1200*3, so branch probability = 3/(7+3+12)=3/22= 0.1363.*

C. [2+2+2+2+2 marks] Optimize the code using any 5 techniques for possible reduction of execution time.

```
int A[N], B[N][128],Period=128;
for(i=0;i<N; i++){ //Loop Fusion
  A[i]=0;
  for(j=0;j<N; j++){
      d=(i*N+j)%Period;
      A[i]=A[i]+B[j][d];
  }
}
```

```
int A[N], B[N][128],Period=128;
for(i=0;i<N; i++){
  int S=0;
  for(j=0;j<N; j++){
      d=(i*N+j)%Period;
      S=S+B[j][d];//S access
  }
  A[i]=S; //No Write Alloc Opt
}
```

```
int A[N], B[N][128],Period=128,IN =0;
for(i=0;i<N; i++){
  int S=0;
  unsigned char Dp=IN%Period;
  for(j=0;j<N; j++){
      S=S+B[j][Dp]; Dp++;
  }
  A[i]=S; IN=IN+N;
}
```

```
int A[N], B[N][128],Period=128,IN =0;
for(i=0;i<N; i++){
int S=0;
unsigned char Dp=IN%Period;
  for(j=0;j<N; j++){
      S=S+B[j][Dp]; Dp++;
  }
A[i]=S; IN=IN+N;
}
```

```
int A[N], B[N][128],Period=128;//Assume N=1200
for(i=0;i<N; i++)A[i]=0;
for(j=0;j<N; j++)  // i and j interchange
  for(i=0;i<N; i++){
      d=(i*N+j)%Period; A[i]=A[i]+B[j][d];   //A[i] is row access
      //for j=0, d=[0,48,96,16,64,112,32] repeats  only 7 miss in i-loop per j
      //for j=1 d=[1,49,97,17,65,113,33] repeats
}
```

Q2 [20 Marks] [Topic: Explicit Threading and Synchronization]

A. [7 marks] Suppose ***N (>> 16)*** threads running on 16-core SMP and a database is available on that SMP and the same is shared among the threads. Each thread modifies the database in every ***m*** seconds and modification need to be done in safe manner. Every modification takes ***k*** seconds if database is not locked. Based on values of ***N***, ***k*** and ***m***, design a locking protocol for the same for better performance. Assume any kind of suitable data structure for the database.

Ans: As N >> 16, many threads are running and if accessing the shared DB takes longer times, it is advisable to go for exponential backup locks and should yield to other threads if locking is unsuccessful instead of wasting CPU time in spinning around the lock.
Accessing the shared DB is longer if DB modification performance is slower than the rate of modification request from all the threads.
DB modification performance is 1/k updates per second, rate of update requests from all the threads is N (1/m) per second. If **1/k < N/m**, we should go for exponential backup locks.*

*In case of Distributed **data bases** or **DB implement using a HASH data structure, data base can be protected by many locks**. Suppose L locks are there for the FB, so the performance of this Distributed DB will be L times better than the DB with one lock. Suppose updates from threads distributed equally to L baskets then Distributed DB modification performance is **L/k** updates per second.*

B. [7 marks] Suppose there are ***N*** memories each of latency ***L*** and bandwidth ***B*** are connected to ***M***-core CPU. We want to optimize the running time of ***W*** independent work using ***T*** threads. Suppose each work access ***A*** amount of data/second from randomly chosen memories at finer granularity. Calculate the value of optimal number of threads T to be run in term of ***L***, ***B***, ***M***, ***N***, ***W*** and ***A***.

*Ans: In general memory sub systems are slower. It is desirable that amount of data required by compute sub system should be same as bandwidth provided by memory sub system. So **N.B ≈ T.A** => **T= N.B/A** and if **T≥M** there is no harm in running more threads on one core. If **N>>T**, latency have a small impact as there are N number of memory and all accesses are randomly goes to different memories.*

C. [6 marks] RAII lock (`lock_guard<std::mutex> lock(m);`) use for locking the shared variable till the execution scope of the **lock** variable exist. Comment on the performance of RAII lock as compared to timed lock and exponential backup lock.
Ans: Performance `lock_guard<std::mutex> lock(m)` is same as simple Mutex variable. It simply wait till lock is occupied by other, which means it spin around lock for lock to be free and it don't do any useful work for own or don't give CPU to other thread. So performance of this lock is not good as compared to exponential back up lock. Timed lock spin around the busy lock upto specified time, so performance of this lock is better as compared to simple lock_guard.

Q3 [20 Marks] [Topic: Implicit Threading and Accelerator Programming]

A. [7 marks] ***HOTPO (Half Or Triple Plus One)*** function defined as **`int HOTPO(n){if(n%2==0) return n/2; return 3*n+1;}`**. ***HOTPO*** distance of a number defined as the number of iteration required to reach 1. HOTPO sequence of 6 is 6, 3, 10, 5, 16, 8, 4, 2, 1 and HOTPO distance is 9. ***HOTPO*** distance doesn't follow any distribution. We want to compute ***HOTPO*** distance of first ***N*** natural numbers using ***m***(<<***N***) OpenMP threads. Design an efficient approach to implement this using OpenMP. If you want, you can infer any properties of HOTPO and use them.

Ans: Basic code for HOTPO distance can be written as

```
int HOTPODist(int n){
    if (n==1)  return 0;
    if(n%2==0)return 1+ HOTPODist(n/2); else return 1+HOTPODist(3*n+1);
  }
void HOTPODist1toN(){
 #pragma omp parallel
 {
   for(i=0;i<N;i++) HDist[i]= HOTPODist(i);
}
}
```

As HOTPO distance execution time is not same for all value of i, so it better to use dynamic scheduling of HOTPODist to thread.

```
void HOTPODist1toN(){
#pragma omp parallel schedule (dynamic, SMALLNUMBER)
{
  for(i=0;i<N;i++) HDist[i]= HOTPODist(i);   //Small number can be 10 or 5
}
}
```

*As the calculation of HOTPO distance of a number requires HOTPO distance of other numbers and it depends on the HOTPO sequence of that number. If we have already calculated HOTPO distance of a number, we should reuse that instead of going for a lengthy recursive function call. (Somewhat similar to creation/striking out of Prime array in **sieve of Eratosthenes**).*

```
int Dist[N];
int HOTPODist1(int n){
    int x;
    if (n==1) { Dist[1]=0; return 0;}
    if(n%2==0) x=n/2; else x=3*n+1;
    if(Dist[x]==0) Dist[x]=HOTPODist(x);
    return 1+Dist[x];
}
void HOTPODist1toN(){
int i;
#pragma omp parallel
{
  for(i=0;i<N;i++) Dist[i]= 0
}
#pragma omp parallel schedule (dynamic, SMALLNUMBER)
{
  for(i=0;i<N;i++) Dist[i]= HOTPODist(i);
}
}
```

Important read-write properties of the Dist array is that "many threads can read from a location of Dist" and also many thread tries to write same value to a particular location of Dist. So it do not violate the memory consistency /sequential property, and this will give consistent result.

B. [7 marks] Calculate Work (or T_1), Span (T_∞) and Parallelism (P) available in matrix multiplication application using order notation. You may assume **Cilk implementation** of matrix multiplication, where ***N X N*** matrix can be

multiplied by 8 ***N/2*** X ***N/2*** size matrix multiplications and one ***N X N*** matrix addition, and again the ***N X N*** addition can be done recursively by 4 ***N/2*** X ***N/2*** size matrix additions.

Ans: Recursive addition of NxN matrix $T_1add(n) = 4.T1add(n/2)+1=O(n^2)$, Span for matrix addition $T_\infty add(n)= T_\infty add(n/2)+1=O(\lg n)$
Based on parallel recursive definition $T_1(n)= 8\ T_1(n/2)+O(n^2) = O(n^3)$
$T_\infty(n) = T_\infty(n/2) + T_\infty add(n)= T_\infty(n/2) +O(\lg n) = O(\lg^2 n)$
Parallelism $P = T_1(n)/ T_\infty(n) = O(n^3)/O(\lg^2 n)= O(n^3/\lg^2 n)$

If you are not considering recursive definition, $T_1(n)= O(n^3)$ and $T_\infty(n)= O(\lg n)$, every Cij can be computed parallelly but the dot product of row i and column j takes at least O(lgn) time in N parallel machines.
so $P=O(n^3/\lg n)$

C. [6 Marks] In HPC accelerators, we have thousands of tiny processors and most of the time the processors are clustered in many clusters. All the clusters shares a common global memory and each cluster have own local memory, local memory of a cluster generally shared among all the processors of cluster.

Design an efficient approach of utilizing the shared memory of cluster among all the cores of cluster to improve the performance for Martix-Vector-Product (***N X N*** matrix, ***N*** size vector) application, where every elements of vector access ***N*** times. Assume ***N*** >> size of the share local memory. If one core of cluster brings some elements of the vector to local shared memory then other cores of the cluster can simply use the data instead of accessing from global memory.

You need to write Pseudo code (need not to be syntactically correct) for the same Martix-Vector-Product application. Assume global memory can hold all the element of matrix and vector. Local shared memory can hold ***S*** number of data and number of cores in a cluster is ***m***, total number of cores in system is ***M***.

Ans: Suppose you are performing A X=Y, where A is a matrix of size NxN and X is a vector of size N.

```
for(i=0;i<N;i++) {
    Sum=0;
    for(j=0;j<N;j++)  Sum=Sum+A[i][j]*X[j];
    Y[i]=Sum;
}
```

In this above code, every element of A get accesses once but every element of X get access N times. As the shared memory can be used as user defined explicit cache. We should use this vector X for explicit caching but the size of the X cannot fit to one shared memory.

Assume N>M, each Y[i] can be computed in phased by M cores. We can compute a Y[i] on a core in ceil[N/S] phases.
In each phase partial values of Y[i] will be computed using S elements of X vector, this S element of vector S can be brought in to shared memory by a core of cluster and others simply use.

```
Parallel for(i=0;i<N;i++) {//for each thread on GPU/accelerator
    Sumi=0;
    for(j=0;j<N;j=j+S)  {
        if(core_id=1st_core_of_cluster) Copy(LocalX[1:S]=X[j:J+S]);
        sync;//Need to wait till al threads reach this point
        Sumi=Sumi+A[i][j]*localX[j];
        }
    Y[i]=Sumi;
}
```