CHALMERS TEKNISKA HÖGSKOLA
Institutionen för data- och informationsteknik
Avdelningen för datorteknik

Exam in DAT400 (Chalmers) and DIT431 (GU) High Performance Parallel Programming, Friday, October 29th, 2021, 14:00h - 18:00h

---

Teacher/Lärare: Mustafa Abduljabbar, tel 7721083. Exam hall visits by teaching staff are planned approx. 1h and 3h after examination start.

Language/Språk: Answers shall be given in English.

Solutions/Lösningar: Solutions will be posted on Thursday, November 4th, 2021 on the Canvas page.

Exam review/Granskning: The review date will be posted on the course Canvas page by the time you receive the email from LADOK.

Aids/Hjälpmedel: Only Chalmers-authorised aids are allowed during examination. This includes pencils, erasers, rulers and dictionaries. Electronic dictionaries or graphing calculators are not allowed.

Grades:

Chalmers				
Points	0-23	24-35	36-47	48-60
Grade	Failed	3	4	5

GU				
Points	0-23	24-41	42-60	
Grade	Failed	G	VG	

Good Luck!

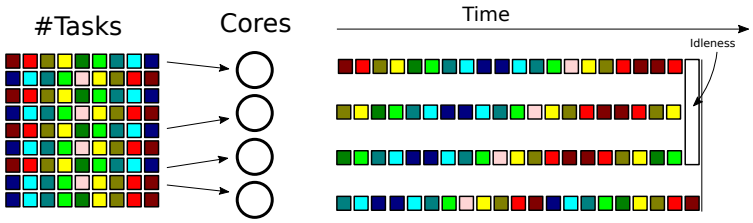


Figure 1: Fine-Grained schedule (FG)

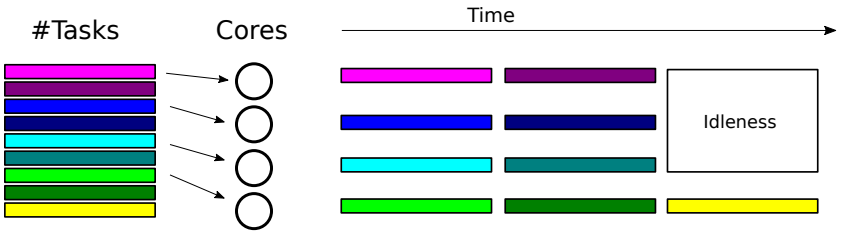


Figure 2: Coarse-Grained schedule (CG)

Problem 1 (12p)

Suppose we have a 9x9 matrix multiplication that is to be computed in parallel. To do that, the computation is decomposed into tasks with the two granularities shown in Figure 1 (Fine-Grained (FG)) and Figure 2 (Coarse-Grained (CG)).
Note that in the fine-grain case, each task computes 1 cell in the output matrix C, whereas in the coarse-grain case, each task computes 1 row of the output matrix C.
Assumptions:
- Task execution times:

- $T_{\text{exec\_fine\_grained}} = 1$ seconds
- $T_{\text{exec\_coarse\_grained}} = 10$ seconds

- Overheads:

- $T_{\text{scheduling}} = 2$ seconds (an overhead introduced every time a task is scheduled for execution)

- Number of cores (threads): 4

Tasks:

(a) In 1-2 lines, explain what makes optimal scheduling so computationally intensive?

(b) In 1 line, give an example of a low-overhead scheduling algorithm.

(c) For Figure 1 and the matrix multiplication problem described above, what is the expected execution time with and without scheduling overhead?

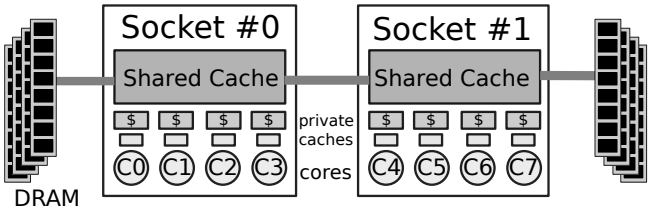
(d) For Figure 2 and the matrix multiplication problem described above, what is the expected execution time with and without scheduling overhead?

(e) Reason about which granularity is better to use based on your answers in (c) and (d).

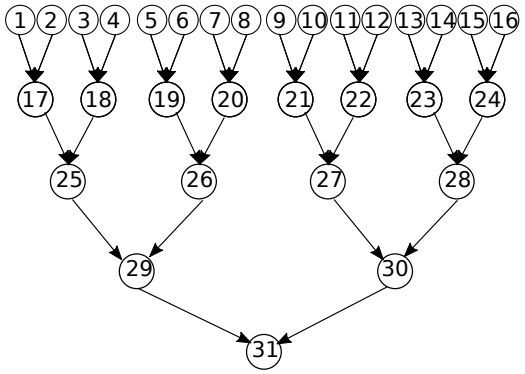
Problem 2 (12p)

Consider the (ccNUMA) computer system shown below consisting of two sockets, each with four cores per socket. Consider also the below DAG (Directed Acyclic Graph) which describes the tasks of the application and shows how tasks communicate data. Consider the following assumptions:

- The initial location of the data is in the DRAM of Socket #0.
- All tasks perform the same operation on the same amount of data and each task consumes 20 cycles for execution on a single core.
- Communication between sockets costs 10 cycles for each task. Assume the computation cannot begin until all tasks' required data is fully transferred.



ccNUMA Computer System



Task DAG

Tasks:

(a) Come up with a good schedule and a good mapping of this DAG on the ccNUMA computer system, when only using socket #0. Which tasks will be executed by each core? What will be the total execution time (cycles) of the DAG?

(b) Assume now the programmer wants to split the DAG's execution on the two sockets. Which tasks will be executed by each core? What will be the new execution time of the DAG in this case? What do you conclude from comparing parts (a) and (b)?

Now we focus on a single task, i.e., a single node in the DAG. In the task, the execution of floating-point instructions on a single core consumes 60% of the total runtime, which can be parallelized. Moreover, 25% of the floating-point execution time is specifically spent in square root calculations. Based on some initial research, the design team would like to improve the performance of the task by using a next-generation processor.

(c) They believe that the new processor could improve the performance of all floating point instructions by a factor of 1.5. According to the Amdahl's law, calculate the speedup they can get from this idea.

(d) The alternative idea is to use the new processor to only speed up the square root operation by a factor of 8. Calculate the speedup of this idea using Amdahl's law. By comparing (c) and (d), which design would benefit the most for the task?

(e) Instead of waiting for the next processor generation, the design team decides to parallelize the code of the task. What speedup can be achieved if running the single task on a 16-CPU system, if 90% of the code can be perfectly parallelized? What fraction of the code has to be parallelized to get a speedup of 10?

Problem 3 (12p)

Consider the following function, which integrates a general function f, which takes a double and returns a double. An attempt has been made to make it parallel using OpenMP, but it does not seem to work.

```
1  double integrate(double low_limit, double high_limit, double h) {
2      int i;
3      double sum = 0.0;
4      int steps = (int)((high_limit - low_limit) / h);
5      // help, the following doesn't work!!
6      #pragma omp parallel for
7      for (i = 0; i < steps; i++) {
8          double step = ((double)i) / ((double)steps)
9              * (high_limit - low_limit);
10         sum += (f(step) + f(step+h)) / 2 * h;
11     }
12     return sum;
13 }
```

Listing 1: Code for tasks (a) to (c)

```
1  int N = 10000;
2      float b[N];
3      initialize(b);
4  if(rank == 0){
5      MPI_Isend(&b[0], N, MPI_FLOAT, 1, 0, comm, &request);
6      for(int i=0; i<N; i++)
7        b[i] *=b[i];
8  }
9  if(rank == 1){
10     float a[N];
11     MPI_Recv(&a[0], N, MPI_FLOAT,0, 0, comm, &status);
12 }
```

Listing 2: Code for task (d)

```
1  int N = 10000;
2      float b[N];
3      initialize(b);
4  if(rank == 0)
5      MPI_XXXXX(&b[0], N, MPI_FLOAT, 1, 0, comm, &request);
6  if(rank == 1){
7      sleep(60); //process sleeps for 60 seconds
8      float a[N];
9      MPI_Recv(&a[0], N, MPI_FLOAT,0, 0, comm, &status);
10 }
```

Listing 3: Code for task (e)

Tasks:

(a) Locate the race condition in Listing 1. Explain why it is a problem, and show **three** approaches of dealing with it. Which approach is the best?

(b) Fix the rest of the parallelization.

(c) Which scheduling policy would you use for this function, and why? Can the function f have an effect on the answer?

(d) Now consider an MPI implementation in Listing 2. What is the problem with this code? Suggest an implementation that works correctly.

(e) In Listing 3, process 0 sends array b to process 0. You have the choice to use 1). **MPI_Ssend** or 2). **MPI_Issend**, which option will take more time to finish in process 0. Also, if it is required to enforce barrier before MPI communication between process 0 and 1, which option (MPI_Ssend or MPI_Issend) will not require an explicit barrier and why?

Problem 4 (12p)

Listing 4 below is a minimal 1D N-Body code that calculates the force and potentional in free space due to all particles using the equation (for potential) $P_i = \sum_{j=1}^{N} \frac{m_i m_j \hat{r_{ij}}}{\sqrt{\Delta r^2}}$ and assumes a unit mass. The outer loop is called the "target" loop, whereas the inner loop is called the "source" loop.

```
1  typedef struct {
2    float x;  // position
3    float vx; // velocity
4    } Body;
5
6  void n_body(Body *p, float dt, int N) {
7    for (int i = 0; i < N; i++) {   // target loop
8      float Fx = 0.0f;
9      for (int j = 0; j < N; j++) { // source loop
10       float dx = p[j].x - p[i].x;
11       float distSqr = dx*dx + SOFTENING;
12       float invDist = 1.0f / sqrtf(distSqr);
13       Fx += dx * invDist;
14     }
15     p[i].vx += dt*Fx;  // update velocities
16   }
17 }
```

Listing 4: Code for problem 4

```
1  // CUDA malloc API
2  // devPtr: pointer to the memory address
3  // size: size of data in bytes
4  cudaError_t cudaMalloc  (void** devPtr, size_t size)
5  // CUDA memory transfer API
6  // dst: destination address
7  // srd: source address
8  // count: size of data in bytes
9  // cudaMemcpyKind transfer direction such as cudaMemcpyHostToHost
10 cudaError_t cudaMemcpy(void* dst, const void* src, size_t count,
11                         enum cudaMemcpyKind kind)
12 // CUDA kernel launch
13 Kernel_Name<<< GridSize,  BlockSize >>>(arguments ...)
```

Listing 5: CUDA library functions

```
1  //Input Parameters comm: communicator (handle)
2  //Output Parameters rank: rank of the calling process
3  //in the group of comm (integer)
4  int MPI_Comm_rank(MPI_Comm comm, int *rank)
5
6  //Input Parameters comm: communicator (handle)
7  //Output Parameters size: number of processes
8  //in the group of comm (integer)
9  int MPI_Comm_size(MPI_Comm comm, int *size)
```

Listing 6: MPI library functions

Tasks:

(a) Give a formula of the total number of cycles required to execute the function `n_body()` as a factor of N given that,

- Addition, subtraction and multiplication takes 1 cycle.
- Division takes 5 cycles.
- Multiply and add takes 2 cycles.
- Square root operation takes 10 cycles.

(b) Using OpenMP, which scheduling policy should be used to parallelize the target loop if we modify the code, such that the inner loop is replaced by `for (int j = rand(N); j < N; j++)` where `rand(N)` is a random number generator such that $0 \leq j \leq N$, what do you think should be the best scheduling choice in this scenario? Justify your answer.

(c) In a CUDA implementation of the code given, how many bytes will be transferred from host to device and back, given that $N = 2^{15} = 32768$. Just write the formula and plug in the numbers. **You do not need to provide the final result.**

(d) For $N = 2^{15} = 32768$, write down the memory transfer CUDA calls required to move data between the host and device before and after the kernel execution. Considering $1D$ CUDA implementation of the code above, write down the kernel launch call with your choice of thread block distribution. Note that for this GPU model, the max number of threads per thread block is $2^{10} = 1024$. The function calls in Listing 5 could be useful).

(e) Now consider an implementation of the code above with hybrid programming model (MPI and OpenMP). How can the target (outer) loop be divided among MPI processes? To answer this part, write a pseudo code to explain the steps for partitioning the outer loop (Mention all MPI calls. The function calls in Listing 6 could be useful).

Problem 5 (12p)

The following code computes 1024 dot products, each of which is calculated from a pair of 256-element sub-vectors. The dot product of two vectors $a = [a_1, a_2, \ldots, a_{256}]$ and $b = [b_1, b_2, \ldots, b_{256}]$ is defined as:

$$a \cdot b = \sum_{i=1}^{256} a_i b_i = a_1 b_1 + a_2 b_2 + ... + a_{256} b_{256} \quad (1)$$

Use the code to answer the following questions. **Note that memory copy from host(device) to device(host) has been omitted for simplicity, thus it is not a problem for the code execution.**

```
1  #define VECTOR_N 1024
2  #define ELEMENT_N 256
3  const int DATA_N = VECTOR_N * ELEMENT_N;
4  const int DATA_SZ = DATA_N * sizeof(float);
5  const int RESULT_SZ = VECTOR_N * sizeof(float);
6  ...
7  float *d_A, *d_B, *d_C; ...
8  cudaMalloc((void **)&d_A, DATA_SZ);
9  cudaMalloc((void **)&d_B, DATA_SZ);
10 cudaMalloc((void **)&d_C, RESULT_SZ);
11 ...
12 dotProduct<<<NUM_THREAD_BLOCKS, NUM_THREADS>>>(d_C, d_A, d_B, ELEMENT_N);
13
14 __global__ void dotProduct(float *d_C, float *d_A, float *d_B, int ElementN){
15     __shared__ float accumResult[ELEMENT_N];
16     //Current vectors bases
17     float *A = d_A + ElementN * blockIdx.x;
18     float *B = d_B + ElementN * blockIdx.x;
19     int tx = threadIdx.x;
20
21     accumResult[tx] = A[tx] * B[tx];
22     for(int stride = ElementN / 2; stride > 0; stride >>= 1){
23         if(tx < stride)
24             accumResult[tx] += accumResult[stride + tx];
25     }
26     d_C[blockIdx.x] = accumResult[0];
27 }
```

(a) Considering the code above, give your choice of number of thread blocks and number of threads per block (line 12) for the dotProduct kernel and explain shortly how you structure each thread for the calculation with your thread organization.

(b) You may have noticed that from line 22 to line 25, it is performing a reduction as a tree. Do you think it can give correct result? If not, identify the problem and fix it.

(c) How many global memory loads and stores are done for each thread? What is the largest and smallest ratios of floating point arithmetic to global memory access (i.e., the Arithmatic Intensity, AI) in each thread?

(d) Identify an opportunity to significantly reduce the bandwidth requirement on the global memory. How would you achieve this? How many accesses can you eliminate?