Solutions for the Exam in DAT400 (Chalmers) and DIT431 (GU) High Performance Parallel Programming, Monday, August 23rd, 2021, 8:30h - 12:30h

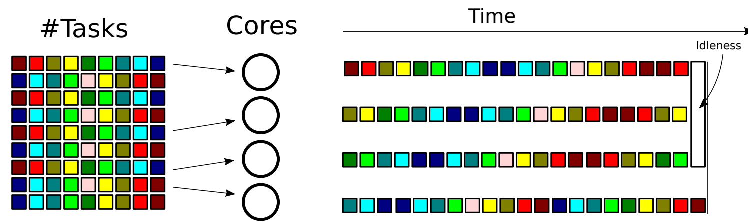


Figure 1: Fine-Grained schedule (FG)

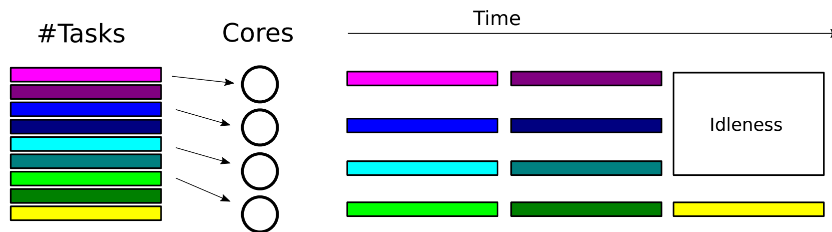


Figure 2: Coarse-Grained schedule (CG)

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 you schedule a task)

- Number of cores (threads): 4

Tasks:

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

(b) In 1 line, give an example of 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).

Answer to Problem 4:

Part (a) Lecture 3 - Slide 21 - Bullets 1 and 2

Part (b) Lecture 3 - Slide 21 - Bullet 3

Part (c) FG time - overhead = 21 x 1 = 21 sec
FG time + overhead = 21 x (1 + 2) = 63 sec

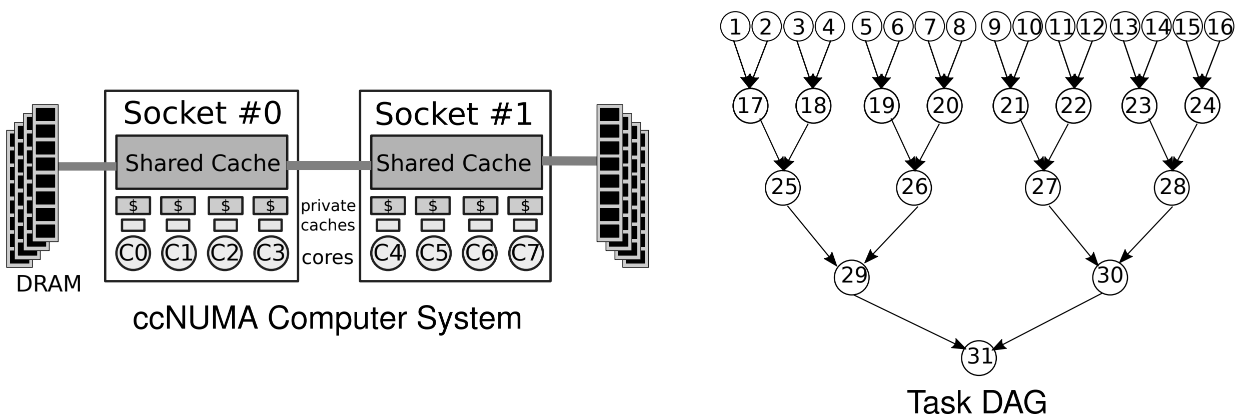
Part (d) CG time - overhead = 3 x 10 = 30 sec
CG time + overhead = 3 x (10 + 2) = 36 sec

Part (e) FG execution performs better than CG when there's no scheduling overhead, while it is worse when there's scheduling overhead.

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, and 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 a programmer would like to split the DAG's execution on two sockets. Which tasks will be executed by each core? What will be the new execution time of the DAG in this case? What conclusion do you get 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?

Answer to Problem 2:

Part (a) Consider the following schedule and mapping of tasks to cores:

```
C0 : Tasks | 1| 2| 3| 4|17|18|25|29|31|
C1 : Tasks | 5| 6| 7| 8|19|20|26|--|--|
C2 : Tasks | 9|10|11|12|21|22|27|30|--|
C3 : Tasks |13|14|15|16|23|24|28|--|--|
```

Total execution time = 9 * 20 = 180 cycles.

Part (b) One possible solution could be:

```
C0 : Tasks | 1| 2|17|25|29|31|
C1 : Tasks | 3| 4|18|26|30|--|
C2 : Tasks | 5| 6|19|27|--|--|
C3 : Tasks | 7| 8|20|28|--|--|
C4 : Tasks | 9|10|21|--|--|--|
C5 : Tasks |11|12|22|--|--|--|
C6 : Tasks |13|14|23|--|--|--|
C7 : Tasks |15|16|24|--|--|--|
```

Tasks 9, 10, 11, 12, 13, 14, 15, 16, 21, 22, 23, 24 require to be transferred from socket #0 to #1, which cost 12 * 10 = 120 cycles.
Total execution time = 6 * 20 + 120 = 240 cycles.

Part (c) Amdahl's law: $Speedup = \frac{1}{s+(1-s)/p}$
Improvement idea 1: all fp instructions sped up by a factor 1.5.
Sequential part (s): 1 - 0.6 = 0.4, p = 1.5.
The application would observe a total speedup of:
$Speedup = \frac{1}{s+(1-s)/p} = \frac{1}{0.4+(1-0.4)/1.5} = 1.25$

Part (d) Improvement idea 2: square root instructions sped up by a factor of 8.
Sequential part (s): 0.4 + (1 - 0.6 * 0.25) = 0.85. p = 8.
The application would observe a total speedup of:
$Speedup = \frac{1}{s+(1-s)/p} = \frac{1}{0.85+(1-0.85)/8} = 1.15$
Thus, the application would benefit the most from the first alternative.

Part (e) The speedup achieved on a 16-CPU system is:
$Speedup = \frac{1}{s+(1-s)/p} = \frac{1}{0.1+0.9/16} = 6.4$.
To attain a speedup of 10, a 96% of the code would need to be perfectly parallelizable. This value is obtained by solving the equation:
10 = $\frac{1}{s+(1-s)/16}$.

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
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?

Answer to Problem 3:

Part (a) The race condition is when the sum variable is updated each iteration. If several threads update this shared variable at once, some writes could be lost. Once way to solve it is using a critical section.

```
#pragma omp parallel for
for (i = 0; i < steps; i++) {
    double step = ((double)i) / ((double)steps)
        * (high_limit - low_limit);
    #pragma omp critical
    sum += (f(step) + f(step+h)) / 2 * h;
}
```

Another is to use atomics.

```
#pragma omp parallel for
for (i = 0; i < steps; i++) {
    double step = ((double)i) / ((double)steps)
        * (high_limit - low_limit);
    #pragma omp atomic
    sum += (f(step) + f(step+h)) / 2 * h;
}
```

A third is to use a reduction.

```
#pragma omp parallel for reduction(+:sum)
for (i = 0; i < steps; i++) {
    double step = ((double)i) / ((double)steps)
        * (high_limit - low_limit);
    sum += (f(step) + f(step+h)) / 2 * h;
}
```

This can also be done manually, by creating private variables for each thread, adding to those, and then adding everything at the end to the sum variable.
Reductions tend to be the best for performance.

Part (b) The main other thing to do is changing *omp parallel* into *omp parallel for*, or every thread will run every iteration instead of splitting them amongst the threads.

Part (c) In almost all cases a static scheduling would be best. It could be possible to have an $f(x)$ that varies widely in computation time based on x, such as

$$f(x) = \left\{ \begin{array}{ll} 0, & \text{for } x \leq 0 \\ x^{2x^{ln(x)}}, & \text{for } 0 \leq x \leq 1 \\ 1, & \text{for } x > 1 \end{array} \right\},$$

and then a case could be made for a dynamic or guided

Part (d)

```
int N = 10000;
    float b[N];
    initialize(b);
if(rank == 0){
    MPI_Isend(&b[0], N, MPI_FLOAT, 1, 0, comm, &request);
      //or MPI_Send(&b[0], N, MPI_FLOAT, 1, 0, comm);
    MPI_Wait(&request, MPI_STATUS_IGNORE);
    for(int i=0; i<N; i++)
      b[i] *=b[i];

}
if(rank == 1){
    float a[N];
    MPI_Recv(&a[0], N, MPI_FLOAT,0, 0, comm, &status);
}
```

The problem with the code is that it is using non-blocking point to point call which releases buffer immediately. Any changes made in the buffer will be reflected on the receiving side as well, this leads to non-deterministic behavior. To solve it, either use a blocking Send() or use MPI_Wait() to ensure that copy is done before changing the buffer.

Part (e) MPI_Ssend will take more time since it waits for the receiver to be ready. As Receiver is sleeping for 60 seconds this will cause a delay in process 0 to return from sending operation. MPI_Ssend imposes an implicit barrier prior to communication, therefore we do not need an explicit barrier.

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).

Answer to Problem 4:

Part (a)

OpCount = $N^2 \times (1 \times (1cycle)\text{subtract} + 1 \times (1cycle)\text{multiply} + 1 \times (1cycle)\text{addition} + 1 \times (5cycle)\text{division} + 1 \times (10cycle)\text{sqrt} + 1 \times (1cycle)\text{addition} + 1 \times (1cycle)\text{multiply}) + N \times (\times(2cycle)\text{add} + \text{multiply}) = 20N^2 + 2N$

The code is potentially data/loop parallel because there are no dependencies across the loop iterations as each "target" iteration outputs data to an independent element at index i.

```
#pragma omp parallel for
for (int i = 0; i < N; i++)
```

or

```
#pragma omp parallel for schedule (static)
for (int i = 0; i < N; i++)
```

For the case of `for (int j = rand(N); j < N; j++)`, dynamic scheduling should be used. Static scheduling will likely create work-imbalanced partitions due to the random sizes of the `source` iterations.

Part (c) Global data transfer size: 2 (considering the HOST to DEVICE and DEVICE to HOST trips) x 4 (float size) x 2 (the velocity/potential vector width) x 10000 (N)

```
int main() {
...
  const int nBodies = 32768$;
  int bytes = sizeOf(p)*4; //type=float
  cudaMalloc(p_device,bytes);
  cudaMemcpy(p_device,p,bytes,cudaMemcpyHostToDevice);
  int nBlocks = (nBodies + BLOCK_SIZE - 1) / BLOCK_SIZE;
  n_body<<<nBlocks, BLOCK_SIZE>>>(d_p.pos, d_p.vel, dt, nBodies);
  cudaMemcpy(p,p_device,bytes,cudaMemcpyDeviceToHost);
  ...
}
```

```
main(){
...
    MPI_Init(NULL, NULL);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    n_body(&p, dt, N/world_size, rank, N); //consider N%world_size =0
    MPI_Finalize();
...
}
```

```
void n_body(Body *p, float dt, int N, int rank, int N_max) {
  for (int i = rank*N; i < rank*N+1 ; i++) {   // target loop
    float Fx = 0.0f;
    #pragma omp paralle for private(Fx)
    for (int j = 0; j < N_max; j++) { // source loop
      float dx = p[j].x - p[i].x;
      float distSqr = dx*dx + SOFTENING;
      float invDist = 1.0f / sqrtf(distSqr);
      Fx += dx * invDist;
    }
    // update velocities
    p[i].vx += dt*Fx;
  }
}
```

Part (f) OMP: #pragma omp barrier, scope: All threads in the parallel region. CUDA: __syncthreads (), : Scope: All thread in the thread block. MPI: MPI_Barrier, Scope All processes in the communicator.

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:

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

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 Compute to Global Memory Access (CGMA) ratio) 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?

Answer to Problem 5:

Part (a) scalarProd<<<VECTOR_N, ELEMENT_N>>>(d_C, d_A, d_B, ELEMENT_N);

Part (b) Missing the synchronization. Fix: add __syncthreads(); before line 23.

Part (c) For each thread, 2 loads (A[] and B[]) + 1 store(d_C[]). Line 21: 1 floating point arithmetic for each thread. Reduction (line 24): 8 floating point arithmetic in thread 0, 7 in thread 1, 6 in thread 2, 5 in thread 3, etc. Threads 128-255 have no floating point arithmetic in reduction phase. Therefore, largest ratio: (1+8)/(1+2), smallest ratio: 1/(1+2).

Part (d) Line 26 stores an identical value to global memory 256 times to the same address, which causes massive performance losses in the memory system performance. The simple solution is to add a conditional so that only thread 0 will do so. We can eliminate 1024*255 stores to the global memory.