Solutions for the Re-Exam in DAT400 (Chalmers) and DIT430 (GU) High Performance Parallel Programming, Wednesday, January 7th, 2020, 8:30h - 12:30h

Problem 1 (10 points)

In this problem we ask you to describe, and to reason about parallelization techniques.

Tasks:

(a) Describe in 3-5 sentences the differences between task and data parallelism. Provide an example (in pseudo-code) for each case

(b) SPMD and SIMD are two related concepts: In 3-5 sentences, what are the similarities and what are the differences between them?

Now consider the (ccNUMA) computer system shown below consisting of two sockets, each with four cores. Each core has a single SIMD unit where each input register can hold up to four double-precision floating-point values. Consider the following program, in which `x`, `y`, `z` are vectors of type `double`.

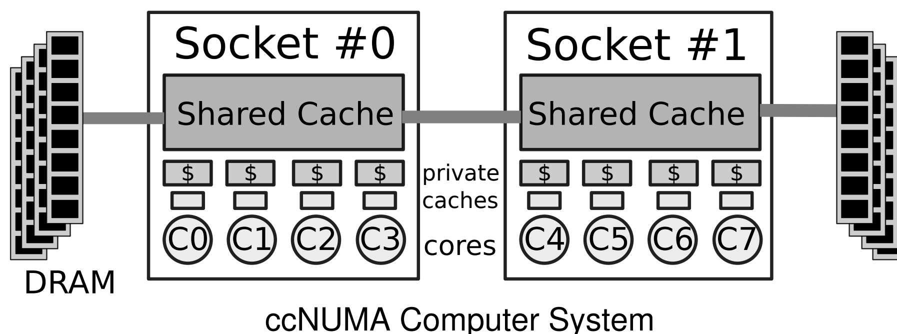
```
for(int i = 0; i < 1024; i++)
    z[i] = x[i] + y[i];
```

The following parameters describe the scenario to be considered:

- Each SIMD operation takes 10 cycles when operating on elements of type `double` (8 bytes). This applies both to loads/stores and arithmetic operations.
- The cost (latency) of communicating data elements within a socket is negligible.
- The cost of communicating data across sockets is 10 cycles per cache line. Cache lines have a size of 64 bytes and cannot be overlapped during communication.
- The data is initially in the DRAM of Socket #0, and should be stored to the same DRAM at the end of the execution.

(c) The first task is to devise a parallelization of the program (i.e. partitioning, scheduling and mapping) that minimizes execution time on a single socket. Describe the partitioning and compute the execution time.

(d) The second task is to devise a parallelization of the program (i.e. partitioning, scheduling and mapping) that minimizes execution time on both sockets. How can we find the optimal number of iterations to be executed on each socket? To simplify, assume that computation on Socket #1 can not start until all data has been transferred.



ccNUMA Computer System

Answer to Problem 1:

Part (a) See Lecture 3, slides 25 and 26

Part (b) See Lecture 3, slide 37

Part (c) Each socket has four cores. There are no dependencies so each core takes care of one fourth of the computation (256 iterations) so that it processes a consecutive chunk of data.
Because of SIMD with of 4, each SIMD-ized loop can compute 4 iterations, hence each core processes 256/4 = 64 loop iterations. Each loop consists of two vector loads (x[] and y[]), one arithmetic operation (+), and one store operation. Hence, a loop takes 40 cycles, and the total execution time would be $64 \times 40 = 2560$ cycles.
In practice, modern processors would apply pipelining and speculation to overlap computations with independent memory accesses of other iterations. This is not in the scope of this course.

Part (d) The optimal partitioning in part (d) is a bit more complex because one needs to consider the amount of data to be moved to the second socket. If socket 2 were to compute half of the iterations (512 iterations, the trivial partitioning), then $512 \times 3/8 = 192$ cache lines need to be transferred, incurring 1920 cycles of latency, which is almost as high as the execution time! Hence this problem requires a bit of mathematical modeling. To simplify we will assume that computations on socket #2 do not start until all data has been transferred. Considering that each SIMDized iteration performs four of the original iterations, we introduce x as the percentage of iterations to be executed on socket 1. We obtain:

$$\text{Texec} = \max\left(\frac{256\times x\times 40}{4}, \frac{256\times(1-x)\times 40}{4} + \frac{256\times(1-x)\times 30}{2}\right)$$

The minimum execution time will be achieved for a balanced execution, i.e. when both sides are equal. Solving this for x we obtain that $\frac{6400}{8960}$ iterations should be computed on the first socket. This is approximately 71% of all iterations.

Problem 2 (6p)

In this problem, we consider various types of loop schedulers on a system with P processors: static scheduler, dynamic scheduler (self-scheduling), and dynamic scheduler (chunk-scheduling) with chunk size = 16. Assume a loop of N iterations performing `add` operations such that each iteration executes in 10ns.

Tasks:

(a) Assume N = 1024 and P = 8, how many tasks will the three scheduling methods generate? What is the number of iterations performed by each task?

(b) What are the execution times on the three schedulers? Assume that for static scheduling there are no per-iteration overheads, and that for dynamic scheduling there is a fixed overhead each time a chunk is assigned, this overhead being 200ns.

(c) Now assume that the generation of all the threads for the parallel loop is needed and it costs an extra overhead T = 50ns + P $\times$ 10ns, which depends on the number of processors. What is the new execution time of static schedulers considering P = 8? After what value of P does adding new processors result in a slowdown instead of a speed-up?

Answer to Problem 2:

Part (a)

1. Static scheduler: 8 tasks, each task has 1024/8 = 128 iterations
2. Dynamic scheduler (self-scheduling): 1024 tasks, each task has 1 iteration
3. Dynamic scheduler (chunk-scheduling): 1024/16 = 64 tasks,each task has 16 iterations

Part (b)

1. Static scheduler: $T_{\text{exec}} = 128 \times 10ns = 1.28us$
2. Dynamic scheduler (self-scheduling): $T_{\text{exec}} = 1024/8 \times (10ns + 200ns) = 26.88us$
3. Dynamic scheduler (chunk-scheduling): $T_{\text{exec}} = 1024/8 \times 10ns + 1024/(8 \times 16) \times 200ns = 2.88us$

Part (c)

1. Static scheduler: $T_{\text{exec}} = 128 \times 10ns + 50ns + 8 \times 10ns = 1.41us$
2. The execution time as a function of P is: $T_{\text{exec}} = 1024/P \times 10ns + 50ns + P \times 10ns$. We take the derivative with respect to P and obtain: $T'_{\text{exec}} = -10240 \times P^{-2} + 10$. By setting $T'_{\text{exec}} = 0$, we get P = 32, therefore once more than 32 processors are employed in this problem, the execution time will start grow because of the thread generation overhead.

Problem 3 (6p)

An important concept in parallel programming is the way in which parallelism is represented in the source code. At the system level, libraries that support parallel programming implement a particular execution model with support from the operating system. In this problem, we ask you to describe different options.

Tasks:

(a) Describe in 3-5 sentences the difference between implicit and explicit parallelism. For each of these two types, provide one example of a technology that follows this model.

(b) Both implicit and explicit representations can generate parallelism on top of processes and threads. What are the main differences between these two execution abstractions (processes and threads)? Please provide an example of a technology that relies on each of these abstractions.

(c) Even if an operating system does not natively support threads, it is possible to use a thread programming style by following a mapping known as N:1. Describe in 3-5 sentences what this means how this can be implemented.

Answer to Problem 3:

Part (a) See Lecture 3, slides 33 and 34

Part (b) See Lecture 4, slides 13 and 14

Part (c) See Lecture 4, slide 15 and 17

Problem 4 (8p)

You are part of a team developing an MPI_Chalmers library and you have been assigned a task to provide an implementation of the `MPI_Ibcast` function, which "broadcasts a message from the process with rank "root" to all other processes of the communicator in a nonblocking way" according to the MPI standard. Below is a list of function APIs that may be useful.

```
int MPI_Isend(const void *buf, int count, MPI_Datatype datatype,
    int dest, int tag, MPI_Comm comm, MPI_Request * request)
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
    int source, int tag, MPI_Comm comm, MPI_Request * request)
int MPI_Send (const void *buf, int count, MPI_Datatype datatype,
    int dest, int tag, MPI_Comm comm)
int MPI_Recv (void *buf, int count, MPI_Datatype datatype,
    int dest, int tag, MPI_Comm comm, MPI_Status *status)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
int MPI_Comm_size(MPI_Comm comm, int *size)

int MPI_Ibcast(void *buffer, int count, MPI_Datatype datatype,
                int root, MPI_Comm comm, MPI_Request * request) {
    // Your implementation
}
```

Tasks:

(a) Write an implementation of the `MPI_Ibcast` that would work as per the standard's description (ignore the `request` parameter in your implementation).

(b) Now you are writing a parallel application that uses `MPI_Ibcast`. You run out of heap memory, and you would like to reuse the `buffer`. Can you reuse the buffer immediately after calling `MPI_Ibcast`? If not, what should be done.

(c) Your library users use `MPI_Bcast` (the blocking version) to occasionally synchronize/checkpoint in their code. They report that the calls from different participants do not really synchronize. Should you accept this as an issue? Why or why not? If not, what would you suggest them to use?

(d) In your application, you would like to use `MPI_Ibcast` to broadcast the message from 0 to processors whose ranks are multiples of 10 only (i.e., 10, 20, 30, etc.). Describe in 2-3 lines what you should do to achieve this **without changing the implementation of** `MPI_Ibcast`.

Answer to Problem 4:

Part (a)

```
int MPI_Ibcast(void *buffer, int count, MPI_Datatype datatype,
                int root, MPI_Comm comm, MPI_Request * request) {
    int myrank, mpisize;
    MPI_Comm_rank(comm, &myrank);
    MPI_Comm_size(comm, &mpisize);
    if(myrank == root) {
        // the root sending to itself is
        // an acceptable simplification
        for(int irank = 0; irank < mpisize; ++irank)
            MPI_Isend(buffer, count, datatype, irank, 0);
    }
    MPI_Irecv(buffer, count, datatype, root, 0, comm);
}
```

Part (b) No, the buffer cannot be reused immediately after non-blocking calls. It can be reused after returning from `MPI_Wait` on the corresponding request handle.

Part (c) This should not be accepted as an issue. Blocking calls do not necessarily synchronize participants as per the standard. `MPI_Barrier` should be used instead.

Part (d)

- Create MPI group.
- Include the ranks that are multiples of 10 in a group and get a handle to the new communicator
- Use `MPI_Ibcast` with this communicator while passing 0 as root.

Splitting the communicator in divisible and non-divisible by 10 is also an option.

Problem 5 (6p)

The execution time of a serial program is 500 seconds for problem size N = 95. Only 5% of this program cannot be parallelized. Assume a machine with P processors could be used to parallelize this program.

Tasks:

(a) What is the upper limit speedup for this program? What is the maximum speedup if a user tries to parallelize the program on the machine?

(b) Assume that thread creation introduces overheads, which adds an extra time of $\mathrm{T} = 10s + 0.2s \times \mathrm{P}$, where P is the number of processors. Give the new speedup formula the program could get on the machine.

(c) The serial execution time as a function of N is $\mathrm{T} = 25s + 5s \times \mathrm{N}$. For the serial case described above (N = 95 and T = 500s), what problem size could be solved without increasing the execution time when running on P processors? Assume that the overhead of thread creation is similar to part (b).

Answer to Problem 5:

Part (a) The serial fraction is 0.05, assuming we have infinite processors, the upper limit speedup is 1/0.05 = 20. The speedup with respect to P is: $S = \frac{1}{0.05+\frac{0.95}{P}}$.

Part (b) With the new data structure, the speedup becomes: $S = \frac{500}{10+0.2\times P+25+\frac{475}{P}}$.

Part (c) The execution time when using P processors: $T = 25 + \frac{5\times N}{P} + 10 + 0.2 \times P = 500$. Hence, $N = 93 \times P - 0.04 \times P^2$.

Problem 6 (8p)

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.

```
typedef struct {
  float x;  // position
  float vx; // velocity
  } Body;

void n_body(Body *p, float dt, int N) {
  for (int i = 0; i < N; i++) {   // target loop
    float Fx = 0.0f;
    for (int j = 0; j < N; 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;
  }
}
```

Tasks:

(a) Give a formula of the total floating point operations (additions, subtractions, division, multiplication, etc.) as a factor of N? Why is this code potentially data parallel?

(b) Using OpenMP, parallelize the target loop with static scheduling enabled. If the inner loop is replaced by `for (int j = 0; j < rand[i]; j++)` where `rand` is an array of random integers $< N$, what do you think should be the best scheduling choice? Justify your answer.

(c) List at most 2 implications (advantages or disadvantages) of parallelizing the inner-loop of the code above.

(d) Assume you run on a dual-socket, 16 core/socket, 4-way multithreaded (SMT). What is the available number of threads on such hardware with multi-threading enabled? Modify your code in (b) such that it binds an OpenMP thread to a hardware thread.

Answer to Problem 6:

Part (a)

OpCount = $N^2 \times (1\text{subtract} + 1\text{multiply} + 1\text{addition} + 1\text{division} + 1\text{sqrt} + 1\text{addition} + 1\text{multiply}) + N \times (\text{add} + \text{multiply}) = 7N^2 + 2N$

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.

Part (b)

```
#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 = 0; j < rand[i]; 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)

- Overhead of synchronization or reduction on the global sum `Fx`.
- Creating and joining the OpenMP threads N times.
- Less cache locality in a thread: (`p[i].x`) is only reused as opposed to (`p[i:k].x`) in the case of target loop parallelism.

Part (d)

The available number of threads on the described platform = 2 sockets x 16 cores/socket x 4 threads/core = 128 threads.

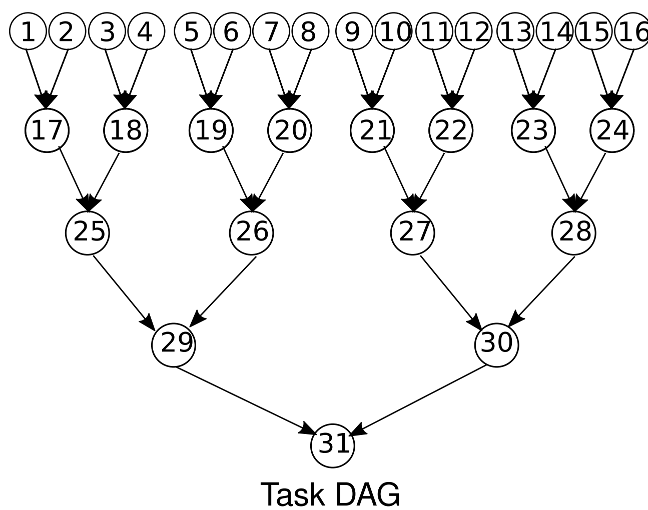
To bind an OpenMP thread to a hardware thread, set the OMP places and number of threads as follows:

```
OMP_PLACES = threads
OMP_NUM_THREADS = 128
```

Problem 7 (8p)

You have written a program that includes the following reduction pattern described as a task DAG, where each task is a single-threaded computation executing on one core and taking the same time (1 time-unit). Consider the following assumptions:

- The DAG is executed on a machine with a number of cores that are all contained in a single chip (i.e. single socket).
- Communication costs are negligible.
- The best sequential algorithm has an execution time of 24 time-units.



Task DAG

Tasks:

(a) For machines with 2, 4 and 8 cores, come up with a schedule and mapping that minimizes execution time.

(b) For each machine and optimal schedule, compute the Cost and the Efficiency.

(c) What is the largest number of cores that can accelerate the computation? Why?

(d) If the achieved level of performance with this number of cores is still unsatisfactory, how can the execution be further improved? Propose at least one parallel programming technique that can potentially improve the execution time of this reduction.

Answer to Problem 7:

Part (a) Case 2 cores: 16 timesteps

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

Case 4 cores: 9 timesteps

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

Case 2 cores: 6 timesteps

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

Part (b)

Case 2 cores: Cost = $16 \times 2 = 32$. Efficiency = $24/32 = 0.75$
Case 4 cores: Cost = $9 \times 4 = 36$. Efficiency = $24/36 = 0.666$
Case 8 cores: Cost = $6 \times 8 = 48$. Efficiency = $24/48 = 0.5$

Part (c)

Because the maximum parallelism in the first step of the computation is 16, the maximum number of cores that can be expected to provide speed-up is 16. At this time the execution time would be 5 steps, which is the same as the critical path.

Part (d)

If higher performance needs to be achieved, there are several potential approaches:

- One can try to speed-up each individual task, for example by making use of SIMD or optimizing the task for better cache reuse
- Alternatively one can try to exploit finer-grained parallelism and split each task into multiple tasks.

Problem 8 (8p)

The code below is an incomplete (GPU) device function that implements a 1D N-Body kernel.

```
__global__
void n_body(float *p, float *v, float dt, int N) {
  int i = /* MISSING 1 */
  /* MISSING 2 */
  {
    float Fx = 0.0f;
    for (int j = 0; j < /* MISSING 3 */; j++) {
      float dx = p[j].x - p[i].x;
      float distSqr = dx*dx + SOFTENING;
      float invDist = 1.0f / sqrtf(distSqr);
      Fx += dx * invDist;
    }
    v[i].x += dt*Fx;
  }
}
int main() {
  const int nBodies = 10000;
  ...
  int nBlocks = (nBodies + BLOCK_SIZE - 1) / BLOCK_SIZE;
  ...
  n_body<<<nBlocks, BLOCK_SIZE>>>(d_p.pos, d_p.vel, dt, nBodies);
  ...
}
```

Tasks:

(a) Fill in the 3 missing code snippets which do the following, respectively 1) Get the index of the element to be calculated. 2) Boundary check. 3) End of "source" array.

(b) List at most 2 conceptual differences between the CPU (OpenMP) and GPU versions of the kernel above.

(c) The code above is unoptimized. Describe and justify an approach to optimize it further in light of the GPU architecture.

(d) If the problem size is N, the PCIe bus speed = S GB/s, and the GPU single precision performance is P GFLOPS. Write a formula in terms of N, S and P that calculates the total execution time of the `n_body` code (including the transfer times). Reuse the total floating point operations from Problem 6(a).

Answer to Problem 8:

Part (a)

```
blockDim.x * blockIdx.x + threadIdx.x; // MISSING 1
if( i < N )                            // MISSING 2
N                                      // MISSING 3
```

Part (b)

- A kernel function in CUDA does not have a loop that corresponds to the target loop in the CPU kernel.
- The loop is replaced with the grid of threads.
- Each thread in the grid corresponds to a single iteration of the original loop $\rightarrow$ very fine grain parallelism enabled by hardware thread scheduling!

Part (c) The high bandwidth GPU shared memory is not used. The code should be tiled such that each thread block pre-loads the tile data in shared memory.

Part (d)
Data size in bytes = $4N$ bytes
Roundtrip transfer time = $2 \times \frac{4N}{SG} = \frac{8N}{SG}$ sec
Kernel execution time = $\frac{7N^2+2N}{PG}$ sec
Total time = $\frac{8N}{SG} + \frac{7N^2+2N}{PG}$ sec