NTNU
Norwegian University of
Science and Technology

Department of Mathematical Sciences

Examination paper for
TMA4280 Supercomputers, introduction to

Academic contact during examination: Eivind Fonn

Phone: 41449889

Examination date: 7 June 2016

Examination time (from–to): 09:00–13:00

Permitted examination support material: B: All printed and handwritten aids permitted. Specific, simple calculator permitted.

Language: English

Number of pages: 8

Number of pages enclosed: 0

Checked by:

Dato Signature

Note! Students find their grading in Studentweb. If you have questions about the grading, contact the department. The Examination office will not be able to answer such questions.

Throughout, you may assume that the time it takes to send a message of k bytes can be expressed as

$$\tau_c(k) = \tau_s + \gamma k,$$

where τ_s is the startup time (latency) and γ is the inverse bandwidth.

Problem 1

We wish to solve the cumulative sum problem: given an input vector $\boldsymbol{u} = (u_1, u_2, \ldots, u_N)$ we wish to form the result $\boldsymbol{v} = (v_1, v_2, \ldots, v_N)$, where

$$v_i = u_1 + u_2 + \ldots + u_i$$

for all i.

a) A simple serial implementation written in C follows. The programming language used is not significant.

```
void cumsum_serial(double *in, double *out, int N)
{
  out[0] = in[0];
  for (int i = 1; i < N; i++)
    out[i] = out[i-1] + in[i];
}
```

Exactly how many floating point operations are required, as a function of N?

Solution. As written, this function needs $N - 1$ floating point operations.

b) We first attempt to parallelize using OpenMP, but it turns out that the following implementation doesn't run as quickly as expected. Why? How would you fix it?

```
void cumsum_openmp_1(double *in, double *out, int N)
{
  #pragma omp parallel for
  for (int i = 0; i < N; i++) {
    out[i] = 0.0;
    for (int j = 0; j <= i; j++)
      out[i] += in[j];
  }
}
```

Solution. The loops don't have the same runtime, so the default scheduler (static) is not appropriate. Dynamic would be better, but since the shortest iterations are very short, probably guided would be best.

c) We attempt fixing the previous implementation by moving the parallel section to the inner loop. Again we find that the performance is lacking. What could be the problem this time?

```
void cumsum_openmp_2(double *in, double *out, int N)
{
  for (int i = 0; i < N; i++) {
    double temp = 0.0;
    #pragma omp parallel for reduction(+:temp)
    for (int j = 0; j <= i; j++)
      temp += in[j];
    out[i] = temp;
  }
}
```

Solution. This time the iterations are equally long, so the scheduler is not a problem. The problem is more likely that the runtime is dominated by forking and joining threads, which happens N times in this implementation, as opposed to only once in the previous one.

In a parallel implementation using MPI, it is assumed that the input vector $\boldsymbol{u}$ is split equally over P processors, where N is a perfect multiple of P. We consider the problem solved if the output vector is split in the same manner. Such an implementation follows (here, `Np = N/P`).

```
void cumsum_parallel(double *in, double *out,
                     int Np, int P, int rank)
{
  cumsum_serial(in, out, Np);
  double *partsums = (double *)malloc(P * sizeof(double));
  MPI_Allgather(&out[Np-1], 1, MPI_DOUBLE,
                partsums, 1, MPI_DOUBLE, MPI_COMM_WORLD);
  double sum = 0.0;
  for (int p = 0; p < rank; p++)
    sum += partsums[p];
  for (int i = 0; i < Np; i++)
    out[i] += sum;
  free(partsums);
}
```

The MPI library function `MPI_Allgather` combines data from all processes into one array and distributes it to all processes. If the total datasize is m bytes, it can run in time $\tau_c(m) \cdot \log_2 P$. It has the arguments

```
MPI_Allgather(sendbuf, sendcount, sendtype,
              recvbuf, recvcount, recvtype,
              comm)
```

d) Describe in your own words how this algorithm works. What is being communicated in the call to `MPI_Allgather`, and what is the meaning of the variables `partsums` and `sum`?

Solution. Each processor first computes the cumulative sum of its own part. It remains to find the sum of all the elements to the "left", namely those belonging to processes with lower rank. Since each process has already computed the sum of its subvector (in the last element), the data are already available, and must just be communicated. Since each process needs the sum of a different subset of the global vector, we find it more convenient to gather all the sub-sums to all the processes, and for this we use `MPI_Allgather`. The data sent is the sum of the local subvector (the last element in `out`), and it's gathered into a temporary array called `partsums`. Each process can then compute the sum of all elements belonging to lower-rank processes and add them to the result.

e) Give a precise upper bound on the number of floating point operations required by any single process. Is it always true that at least one process needs this maximal number of floating point operations?

Solution. The serial cumulative sum requires (as we've seen) $N/P - 1$ operations. The computation of `sum` requires p operations, where p is the rank of a process, and the final computation requires an additional N/P operations: in total $2N/P+p-1$.

Since $p < P$, no process will use more than $2N/P + P - 2$ operations (obtained by setting $p = P - 1$). The highest ranked process will need all of these operations, so the bound is tight.

f) For a given N, what is the optimal number of processes P that minimizes the maximal number of floating point operations? For this problem, you may ignore the assumption that N must be a multiple of P and that P is an integer.

Solution. Let $f(P; N)$ be the number of operations computed in the previous problem. It is minimized as a function of P when $f'(P; N) = 0$. The derivative with respect to P is

$$f'(P; N) = -\frac{2N}{P^2} + 1$$

which gives

$$P = \sqrt{2N}.$$

g) Assume that a single floating point operation runs in time τ_a. What is the estimated speedup for P processes, as a function of $N, P, \tau_a, \tau_s, \gamma$?

Given the optimal number of processes $P = P(N)$ found in the previous task, what is the asymptotic speedup and efficiency as a function of N, as $N \to \infty$? Comment on the result.

Note: As a simplification, you may assume that N, P and N/P are all $\gg 1$.

Solution. The time it takes for one process to complete the serial program is

$$T_1 = \tau_{\mathrm{a}} N.$$

The time it takes the parallel version to run is

$$T_P = \tau_{\mathrm{a}} \left(\frac{2N}{P} + P \right) + T_{\mathrm{comm}},$$

where T_{comm} is the communication time, which is entirely caused by the call to `MPI_Allgather`. By assumption, it runs in time

$$T_{\mathrm{comm}} = \tau_{\mathrm{c}}(m) \log_2 P = (\tau_{\mathrm{s}} + \gamma m) \log_2 P$$

where m is the number of bytes in the total dataset. Since each process contributes one floating point number (let us say double precision), $m = 8P$. Therefore,

$$T_{\mathrm{comm}} = (\tau_{\mathrm{s}} + 8P\gamma) \log_2 P$$

so

$$T_P = \tau_{\mathrm{a}} \left(\frac{2N}{P} + P \right) + (\tau_{\mathrm{s}} + 8P\gamma) \log_2 P.$$

We now set $P = \sqrt{2N}$. The dominant term in T_P will then be the term $P \log P \sim \sqrt{N} \log N$, and we will get an asymptotic speedup rate of

$$S_P = \frac{T_1}{T_P} \sim \frac{\sqrt{N}}{\log N},$$

and an efficiency of

$$\eta_P = \frac{S_P}{P} \sim \frac{1}{\log N}.$$

If we choose P optimally (from a flops perspective), the communication cost is the dominant term, which leads to an efficiency that does not $\to 1$. However, this is not unusual: it's a common problem with parallel algorithms. In spite of this, the speedup is always increasing, so it's always better to add more processes.

Problem 2

a) An N-vector $\boldsymbol{x}$ is split over P processes, so that process p owns all elements x_i such that $i \in \mathcal{I}_p$. Here, $\mathcal{I}_p$ are pairwise disjoint index sets satisfying

$$\bigcup_{p=1}^{P} \mathcal{I}_p = \{1, 2, \ldots, N\}.$$

A sparse matrix $\boldsymbol{A}$ is similarly split by columns, so that process p owns all elements a_{ij} where $j \in \mathcal{I}_p$.

When forming the matrix-vector product $\boldsymbol{Ax}$, what is the minimal amount of communication necessary *from* a given process p *to* a given process q?

Hint: The answer should depend on the sparsity pattern of $\boldsymbol{A}$.

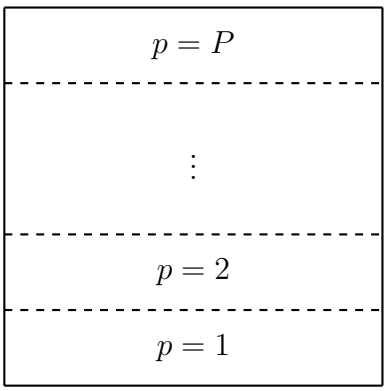
Solution. After multiplying the local matrix with the local vector, we obtain a global vector, where the elements owned by process q must be communicated to process q. However, if $\boldsymbol{A}$ is sparse, there's a good chance that many of these elements are zero. In particular, we can say that if $i \in \mathcal{I}_q$, then the resulting element i must be communicated if there is at least one nonzero A_{ij} with $j \in \mathcal{I}_p$.

The minimal amount of communication necessary is then the total number of q-owned indices i for which there is at least one p-owned index j for which A_{ij} is nonzero.

Note that this is an asymmetrical condition!

b) Assume that $\boldsymbol{A}$ is the matrix obtained from applying the five-point stencil on a two-dimensional grid with $n = \sqrt{N}$ points in each direction. Assume also that the index sets $\mathcal{I}_p$ are chosen so that each process owns one complete "slice", i.e. the split only runs in the x-direction.

What is *now* the minimal amount of communication from process p to process q required to form the matrix-vector product?



Solution. In this matrix, the only nonzero elements are those A_{ij} for which i and j correspond to the same node, or neighboring nodes on the grid. The only elements that need to be communicated are thus the boundary elements between the processes, for which the matrix has nonzero entries in both parts.

Therefore, process p must communicate n elements to and from processes $p+1$ and $p-1$ (unless one of them is on the boundary).

Note that this is a symmetrical condition because the matrix $\boldsymbol{A}$ is symmetrical.

c) Propose a splitting strategy that improves this method by reducing the communication required for a matrix-vector product. Does your strategy impose any restrictions on the number of processes P?

Solution. If we split the domain in both directions instead of just one direction, the boundary length per process is a lot smaller: namely $n/\sqrt{P}$ instead of n. This should reduce communication.

Since the grid is square, this is easiest to do if there's an equal number of processes in each direction: in other words, if P is a square. However it's certainly possible to do it in an approximate sense even without this condition.

d) One iteration with the conjugate gradient method requires one matrix-vector product and two inner products (the other operations do not require communication). Give an expression for the total communication cost for one such iteration both for the "slice" strategy and your improved one.

Solution. The inner products require $\tau_\mathrm{s} \log_2 P$ time each, due to an MPI reduction.

In the first case there are two communication stages ("up" and "down"), each with a cost of $\tau_\mathrm{c}(8n)$, while in the second case there are four communication stages ("up", "down", "left" and "right"), each with a cost of $\tau_c(8n/\sqrt{P})$.

Therefore,

$$T^{(1)}_\mathrm{comm} = 2\tau_\mathrm{s} \log_2 P + 2\left(\tau_\mathrm{s} + 8n\gamma\right),$$
$$T^{(2)}_\mathrm{comm} = 2\tau_\mathrm{s} \log_2 P + 4\left(\tau_\mathrm{s} + \frac{8n\gamma}{\sqrt{P}}\right).$$

Problem 3

Please choose the correct alternative for each task. Unless explicitly noted, you do not have to explain your answer.

a) A good strategy for reducing communication overhead is to increase the number of processes. Answer: true or false.

Solution. False. Communication overhead usually increases with the number of processes (see problem 1).

b) Since threads do not need to communicate (like processes do), there is no penalty incurred by using more of them. Answer: true or false.

Solution. False. Forks and joins are costly, as are locks in cases where threads must synchronize.

c) It is always OK to call MPI library functions from different threads. Answer: true or false.

Solution. False. Some MPI libraries may not support this.

d) It is never OK to call MPI library functions from different threads. Answer: true or false.

Solution. False. Some MPI libraries support this.

e) It is usually possible to use more processors with pure MPI than with pure OpenMP. Answer: true or false.

Solution. True. It is considerably easier to connect a large number of processors if you don't require that they must each connect to the same memory. All large supercomputers use distributed memory.

f) The following loops are compiled with an optimizing compiler. Which of them will likely have the highest performance (more flops)? Answer: A, B or none. Briefly explain why.

```
for (int i = 0; i < N; i++)
  a[i] = a[i] + b[i];            // A
for (int i = 0; i < N; i++)
  a[i] = a[i] + c * b[i];        // B
```

Solution. B. A uses fewer operations per memory access, and B can exploit pipelining.

g) The following loops are compiled with an optimizing compiler. Which of them will likely have the highest performance (more flops)? Answer: A, B or none. Briefly explain why.

```
for (int i = 0; i < N; i++)
  a[i] = a[i] + c[i] * b[i];     // A
for (int i = 0; i < N; i++)
  a[i] = a[i] + c * b[i];        // B
```

Solution. B. In this case pipelining is available to both but A needs one more memory access operation per iteration.

h) BLAS is a high performance library for solving linear systems of equations. Answer: true or false.

Solution. False. BLAS does not solve linear systems. (LAPACK does that.)