Mastermath midterm examination
Parallel Algorithms. Solution

Teacher: Rob H. Bisseling, Utrecht University

October 20, 2021

1. (a) The algorithm given as Algorithm 1 is based on the inner-product algorithm presented in the book PSC2 (Algorithm 1.1 in section 1.3), but adapted to a norm computation. The supersteps for both norms are merged to save 3 synchronisations.

   (b) The cost of superstep (0) is $\frac{4n}{p}$ because 1 absolute-value operation, 2 additions, and 1 multiplication are performed for every local element. The cost of superstep (1) is $2(p-1)g$ because 2 local partial sums are sent to all other processors. The cost of superstep (2) is $2p+2$ because of $2p$ additions, a square-root operation, and a division. The total cost is thus

$$T = \frac{4n}{p} + 2p + 2 + 2(p-1)g + 3l.$$

2. (a) An efficient parallel Modified Gram-Schmidt algorithm is given as Algorithm 2. It parallelises the steps of the MGS algorithm, making use of the inner-product algorithm presented in the book PSC2 (Algorithm 1.1 in section 1.3), but adapted to a block distribution. This algorithm is first used to compute the norm r_{kk}.

   After that the inner-product algorithm is used to compute a set of $m-k-1$ inner products r_{kj}. This can be done efficiently by first computing all the partial sums for all the inner products, and only then computing all the inner products, and finally using all of them. This saves many synchronisations.

   A subtle point is that each processor is made responsible for computing about $(m-k-1)/p$ inner products, instead of computing all inner products redundantly. This is worthwhile since $m \geq p$, so that the work can be spread. This is different than in the case

Algorithm 1 Norms-ratio algorithm for processor $P(s)$, with $0 \leq s < p$.

input: $\mathbf{x}$: vector of length n, $\text{distr}(\mathbf{x}) = \phi$, with $\phi(i) = i \bmod p$, for $0 \leq i < n$.
output: $\gamma = \frac{||\mathbf{x}||_2}{||\mathbf{x}||_1}$, $\text{repl}(\gamma) = P(*)$.

$\alpha_s := 0;$
$\beta_s := 0;$ ▷ Superstep (0)
for $i := s$ **to** $n-1$ **step** p **do**
    $\alpha_s := \alpha_s + |x_i|;$
    $\beta_s := \beta_s + x_i^2;$

for $t := 0$ **to** $p-1$ **do** ▷ Superstep (1)
    put α_s in $P(t)$;
    put β_s in $P(t)$;

$\alpha := 0;$ ▷ Superstep (2)
$\beta := 0;$
for $t := 0$ **to** $p-1$ **do**
    $\alpha := \alpha + \alpha_t;$
    $\beta := \beta + \beta_t;$

$\beta := \sqrt{\beta};$
$\gamma := \frac{\beta}{\alpha};$

of a single inner product. (Still, it would be OK to choose for the simpler option of computing all inner products redundantly. This would save $2m$ synchronisations.)

(b) The BSP cost of the algorithm can be obtained by analysing the separate supersteps in stage k: superstep (0) costs $2b$; (1) costs $(p-1)g$; (2) costs $p+1+b+2b(m-k-1)$.
In superstep (3), each processor sends at most $r-\lfloor \frac{r}{p} \rfloor$ values where $r = m-k-1$ and it receives at most $\lceil \frac{r}{p} \rceil (p-1)$ values. (We write r for brevity.) It can be shown that the the latter is the maximum (by making a distinction between the cases $r \bmod p = 0$ and $r \bmod \ > 0$). The cost is thus $\lceil \frac{r}{p} \rceil (p-1)g$. Superstep (4) costs $\lceil \frac{r}{p} \rceil p$. Superstep (5) costs $\lceil \frac{r}{p} \rceil (p-1)g$, the same as (3). Superstep (0') costs $2rb$. The total synchronisation cost of one stage is $6l$.
The total cost of all stages is then obtained by adding the costs above, where we make use of Lemma 2.10 in section 2.4 of PSC2, with the convenient assumption that $m \bmod p = 1$ (so that we can apply the lemma directly):

$$\sum_{k=0}^{m-1} \left\lceil \frac{m-k-1}{p} \right\rceil = \sum_{k=0}^{m-1} \left\lceil \frac{k}{p} \right\rceil = \frac{(m-1)(m-1+p)}{2p}.$$

The total computation cost then becomes

$$\begin{aligned} T_{\text{Comp}} &= \sum_{k=0}^{m-1} (2b+p+1+b+4b(m-k-1)) + \frac{(m-1)(m-1+p)}{2p} p \\ &= \sum_{k=0}^{m-1} (p+1-b+4b(m-k)) + \frac{(m-1)(m-1+p)}{2} \\ &= (p+1-b)m + 2bm(m+1) + \frac{(m-1)(m-1+p)}{2} \\ &\approx \frac{2nm^2}{p}. \end{aligned}$$

Similarly, the total communication time becomes

$$\begin{aligned} T_{\text{Comm}} &= \left(m(p-1) + 2\frac{(m-1)(m-1+p)}{2p}(p-1) \right) g \\ &\approx m^2 g. \end{aligned}$$

The total synchronisation time is $6ml$. Here, we obtained a linear number of synchronisations, not a quadratic one, by repeating the for-loop over j.

Algorithm 2 Modified Gram-Schmidt (MGS) for processor $P(s)$.

input: $\mathbf{a}_j, 0 \leq j < m$, set of m independent vectors, distr($\mathbf{a}_j$) = block.
output: $\mathbf{q}_j, 0 \leq j < m$, set of m orthonormal vectors, distr($\mathbf{q}_j$) = block.

$b = n/p$;
for $k := 0$ **to** $m-1$ **do**
  $\alpha_s := 0$; ▷ Superstep (0)
  for $i := sb$ **to** $(s+1)b-1$ **do**
    $\alpha_s := \alpha_s + a_{ik}^2$;

  for $t := 0$ **to** $p-1$ **do** ▷ Superstep (1)
    put α_s in $P(t)$;

  $r_{kk} := 0$; ▷ Superstep (2)
  for $t := 0$ **to** $p-1$ **do**
    $r_{kk} := r_{kk} + \alpha_t$;
  $r_{kk} := \sqrt{r_{kk}}$;
  for $i := sb$ **to** $(s+1)b-1$ **do**
    $q_{ik} := \frac{a_{ik}}{r_{kk}}$;
  for $j := k+1$ **to** $m-1$ **do**
    $\alpha_{sj} := 0$;
    for $i := sb$ **to** $(s+1)b-1$ **do**
      $\alpha_{sj} := \alpha_{sj} + q_{ik}a_{ij}$;

  for $j := k+1$ **to** $m-1$ **do** ▷ Superstep (3)
    put α_{sj} in $P(j \bmod p)$;

  for $j := k+1$ **to** $m-1$ **do** ▷ Superstep (4)
    if $j \bmod p = s$ **then**
      $r_{kj} := 0$;
      for $t := 0$ **to** $p-1$ **do**
        $r_{kj} := r_{kj} + \alpha_{tj}$;

  for $j := k+1$ **to** $m-1$ **do** ▷ Superstep (5)
    if $j \bmod p = s$ **then**
      for $t := 0$ **to** $p-1$ **do**
        put r_{kj} in $P(t)$;

  for $j := k+1$ **to** $m-1$ **do** ▷ Superstep (0')
    for $i := sb$ **to** $(s+1)b-1$ **do**
      $a_{ij} := a_{ij} - r_{kj}q_{ik}$;

Note that we can view the set of column vectors $\mathbf{a}_j$ as an $n \times m$ matrix A, distributed in the 1D block row distribution. To reduce the communication further, we have to view the computation as a matrix computation and use a 2D distribution, preferably cyclic since the set of active columns becomes smaller during the computations. This also helps reduce the load balance.

3. (a) The computation starts with $c_{0j} = j$ for all j and $c_{i0} = i$ for all i. This is the trivial case where one sequence is empty and the number of gaps equals the length of the other sequence. The cost c_{ij} for $i, j > 0$ can be expressed in terms of previous costs by

$$c_{ij} = \min(c_{i,j-1} + 1, c_{i-1,j} + 1, c_{i-1,j-1} + \gamma_{ij}),$$

where γ_{ij} is defined by

$$\gamma_{ij} = \begin{cases} 0 & \text{if } x_i = y_j, \\ 2 & \text{if } x_i \neq y_j. \end{cases} \tag{1}$$

This formula expresses the fact that we can extend an alignment of i nucleotides from $\mathbf{x}$ and $j-1$ nucleotides from $\mathbf{y}$ by appending a gap to the $j-1$ nucleotides from $\mathbf{y}$, or extend an alignment with the roles reversed, or extend an alignment of $i-1$ nucleotides from $\mathbf{x}$ and $j-1$ nucleotides from $\mathbf{y}$ at no extra cost provided $x_i = y_j$, or at cost 2 otherwise. The resulting sequential algorithm is given as Algorithm 3.

Note that we have some freedom in the order of computing the costs c_{ij}. The sequential algorithm does this by row, but it is also possible by column, or by *wavefront* (a set of matrix elements with $i + j = k$ for a fixed k). The sequential algorithm takes about $4n^2$ flops, counting 3 additions and one comparison for determining γ.

(b) In the computation by row, we do not have any parallelism: only one cost c_{ij} can be computed at the same time. In a computation by wavefront we do have parallelism: all $k-1$ unknown values can be computed at the same time, because the values they need lie in the previous two wavefronts.. (The two values c_{0k} and c_{k0} of wavefront k are already known by their initialisation.)

A possible choice of distribution is the 1D block column distribution: c_{ij} is stored at processor $P(j \text{ div } b)$, where $b = n/p$ is the block size, assuming $n \bmod p = 0$. This means that values only have to be sent by $P(s)$ to $P(s+1)$. Only the last column of each block has to be communicated, one element at a time.

Algorithm 3 Sequential sequence alignment

input: $\mathbf{x}, \mathbf{y}$: vector of length n, representing DNA.
output: Cost c_{nn} of an optimal alignment.

```
for i := 0 to n do
    c_i0 := i;
    c_0i := i;
for i := 1 to n do
    for j := 1 to n do
        if x_i = y_j then
            γ := 0;
        else
            γ := 2;
        c_ij := min(c_{i,j-1} + 1, c_{i-1,j} + 1, c_{i-1,j-1} + γ);
return c_nn
```

The computation then has k stages. In each stage, processor $P(s)$ first computes its part of the wavefront and then communicates 1 value to its neighbour $P(s+1)$.

(c) The computation has stages $k = 0, 1, \ldots, 2n-2$, each with computation cost at most $4n/p$, communication cost g, and synchronisation cost $2l$, for a total of

$$T_{\text{DNA}} = \left(\frac{4n}{p} + g + 2l\right)(2n-1) \approx \frac{8n^2}{p} + 2ng + 2nl.$$

Note that we can only achieve a maximum speedup of $p/2$, where we lost a factor of two because at the start and end of the algorithm fewer processors are working. Halfway the algorithm, all are working. The load balance can be improved by using a 1D block-cyclic column distribution at the expense of an increase in communication.

The amount of synchronisation can be reduced by viewing a square $\beta \times \beta$ block of matrix elements as a single element and running the algorithm on the blocks. This algorithmic blocking reduces the synchronisation time to n/β and it does not affect the total communication time or the upper bound on the computation time. A good choice would then be $\beta = n/p$, bringing the synchronisation time down to $2pl$.

(d) You need to store at most three wavefronts, so the others can be discarded. The amount of memory needed per processor is then n for $\mathbf{x}$, n/p for $\mathbf{y}$, and $3n/p$ for the wavefronts, together $n + 4n/p$, which is a lot less than the total amount of memory $\mathcal{O}(n^2)$, or the local amount $\mathcal{O}(n^2/p)$.