MPI commands

```
MPI_Isend(sendbuf, scnt, dtype, dest, tag, comm, *request)
MPI_Irecv(recvbuf, rcnt, dtype, source, comm, *request)
MPI_Wait(request, status)
MPI_Waitall(count, array_of_requests, array_of_statuses)
```

```
MPI_Scatter(sendbuf, scnt, dtype, rbuf, rcnt, dtype, root, comm)
MPI_Scan(sendbuf, recvbuf, count, dtype, op, comm)
```

```
MPI_Sendrecv(sbuf, scnt, dtype, dest, stag, rbuf, rcnt, dtype, source, rtag, comm, *stat)
```

OpenMP

`#pragma omp parallel [clause list]`
Clauses commonly used:
`default(shared)` or `default(none)`
`if (scalar expression)`
`num_threads(integer expression)`
`private(variable list)`
`firstprivate(variable list)`
`shared(variable list)`
`reduction (operator: variable list)`
where operators are $+$, $-$, $-$, $*$, &, |, ˆ , &&, ||.

`#pragma omp for [clause list]`
Clauses are here `private`, `firstprivate`, `lastprivate`, `reduction`, `schedule`, `nowait`, `ordered`.

A loop ends with an implicit barrier, the nowait clause removes this. For an explicit barrier, use `#pragma omp barrier`.

We also have `#pragma omp master` and `pragma omp single`.

Analytical Modeling

Parallel overhead: Interprocess interaction, idling and excess computation.

Serial runtime T_S
Parallel runtime T_P
Total parallel overhead is given by

$$T_o = T_{all} - T_S = pT_P - T_S.$$

Speedup and parallel efficieny

$$S = T_S/T_p, \qquad E = S(p)/p$$

It is ideally bounded by T_S/p (not in practice, superlinear speedup).

Point-to-point communication cost $t_{\text{comm}} = t_s + wt_w$
Startup time t_s
Numer of words w
Per word transfer time t_w

Total cost of a parallel system is pT_P, and a parallel system is said to be cost-optimal if pT_P is asymptotically identical with T_s. Parallel efficiency is $E = \mathcal{O}(1)$ for a cost-optimal system.

The efficiency of any parallel system decreases with p, this is apparent, because efficiency is given as

$$E = \frac{T_S}{pT_P} = \frac{1}{1 + T_O/T_S},$$

and T_O is at least $(p-1)T_{\text{serial}}$, which grows linearly.

A parallel system is called scalable if we can keep a constant level of E by increasing the problem size and the number of processes p at the same time.

The parallel runtime is

$$T_P = (W + T_O(W, p))/p$$

Efficiency is then

$$E = W/(W + T_O) = 1/(1 + T_O/W)$$

For a given efficiency, we have

$$W = KT_O,$$

where $K = E/(1-E)$.

Dense matrix algorithms

Matrix-vector multiplication

$$\mathbf{A}\boldsymbol{x} = \boldsymbol{y}, \quad \mathbf{A} \in \mathbb{R}^{n \times n}.$$

Rowwise 1D-partitioning

Each row A_i must be multiplied with $\boldsymbol{x}$, resulting in y_i. In 1D row-partitioning, each process is responsible for a set of rows, and gets a set of $\boldsymbol{x}$, as each process must know the entire $\boldsymbol{x}$-vector, we start by doing an all-to-all broadcast. The process responsible for row A_i then computes y_i.

We have

$$T_P = \frac{n^2}{p} + t_s \log p + t_w n.$$

So $pT_P = n^2 + t_s p \log p + t_w np$. Cost-optimal as long as $p = \mathcal{O}(n)$. For isoefficiency we have $W = KT_O$, where

$$T_O = pT_p - T_S = t_s p \log p + t_w np.$$

We see that the first term in the overhead gives

$$W = \mathcal{O}(p \log p).$$

While the second term gives

$$W = K^2 t_w^2 p^2 = \mathcal{O}(p^2).$$

2D partitioning

A 2D mesh is introduced, so each process gets a block of the original matrix. The $\boldsymbol{x}$-vector is split among the processes in the right-most column in the mesh. First the parts of $\boldsymbol{x}$ must be aligned along the main diagonal of the mesh, and then column-wise one-to-all broadcasts are done. Each process then calculates it's local partial-sum. Finally rowwise reductions take place to find the final $\boldsymbol{y}$.

Matrix-matrix multiplication

$$\mathbf{A} \times \mathbf{B} = \mathbf{C}, \qquad \mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathbb{R}.$$

The components of C are given by

$$c_{ij} = \sum_k a_{ik} b_{kj}.$$

But note that we can also write it as block matrix multiplication

$$\mathbf{C}_{ij} = \sum_k \mathbf{A}_{ik} \times \mathbf{B}_{kj}.$$

Simple algorithm

A 2D block partitioned is used. Process $P_{i,j}$ gets sublocks $\mathbf{A}_{ij}$ and $\mathbf{B}_{ij}$. Each process has the job to calculate the block $\mathbf{C}_{ij}$, to do this, it needs to know all the subblocks $\mathbf{A}_{ik}$ and $\mathbf{B}_{kj}$, so an all-to-all broadcast of $\mathbf{A}$'s blocks is performed in each row, and an all-to-all broadcast of $\mathbf{B}$'s blocks is performed in each column. This algorithm is cost-optimal for $p = \mathcal{O}(n^2)$

Cannon's Algorithm

A more memory-efficient version of the simple algorithm. Instead of all-to-all broadcasting sublocks of $\mathbf{A}$ and $\mathbf{B}$. First, process $P_{i,j}$ gets $\mathbf{A}_{i,j}$ and $\mathbf{B}_{i,j}$. We then align the subblocks by each process shifting their A subblocks i steps to the left, and their B subblocks up j steps- Then each process calculate their partial sum of the subblock C_{ij}, each A and subblock is shifted one step to the left, and each B subblock is shifted one step up. This is repeated p times in total. Each process now have their final $\mathbf{C}_{ij}$ subblock.

DNS algorithm

The simple algo and Cannon's both have a maximum of n^2 processes. The DNS algo has a max of n^3. Arrange the processes in a three-dimensional grid. Process P_{ijk} is responsible for multiplying $\mathbf{A}_{ik} \times \mathbf{B}_{kj}$. The results the processes are then gathered to obtain $\mathbf{C}_{ij}$. It is cost optimal for $p = \mathcal{O}(n^3/\log n)$.

Gaussian Elimination

$$\mathbf{A}\boldsymbol{x} = \boldsymbol{b}, \qquad \mathbf{A} \in \mathbb{R}^{n \times n}.$$

Here, $\mathbf{A}$ and $\boldsymbol{b}$ are known, and we want to find $\boldsymbol{x}$. This is done in two steps. First we reduce the extended matrix to upper triangular form, so we have $\mathbf{U}\boldsymbol{x} = \boldsymbol{y}$. Where $\mathbf{U}$ has 1 on it's main diagonal, and only zeroes under the diagonal. We then do back-substitution to find $\boldsymbol{x}$ in reverse order.

Serial Gaussian Elimination

```
for k=0:n
    // Division step
    for j=k:n
        A[k,j] /= A[k,k]
    y[k] = b[k]/A[k,k]
    // Elimination step
    for i=k+1:n
        for j=k:n
            A[i,j] -= A[i,k]*A[k,j]
        b[i] -= A[i,k]*y[k]
```

Note that after iteration k has finished, only $A_{i,j}$ for $i, j > k$ is active.

Rowwise 1D-partitioning

Process P_i is responsible for row i of A. At the $k'th$ iteration, process P_k performs the divison step and broadcasts the result to the rows below it, the elimination step is then performed. The algorithm is *not* cost optimal.

Pipelined rowwise partitioning

We can easily pipeline the division/broadcast/elimination stages. The priorities is then

1. Send data destined for other processes.
2. Do computations with data already recieved.

This version *is* cost-optimal

Less processes than rows If we have $p < n$ then we can either partition using either block 1D or cyclic 1D. The block partitioning leads to uneven load distribution and so a cyclic distribution should be used.

2D block partitioning Process P_{ij} is responsible for A_{ij}. At the beginning of iteration k, P_{kk} must do a one-to-all broadcast to the active processes in row k. After each process in row k recieve P_{kk}, they compute the division, and perform a one-to-all broadcast to the active processes in their column. As in the 1D case, this is *not* cost-optimal unless pipelined. Also, again a cyclic 2D partitioning should be used.

Sorting

In the most basic case, we can assign a process to each number, the list can then be sorted through a series of *compare-exchange* operations. This has poor performance, as $t_s \gg t_w$, and thus the exchange time will dominate the compare time. If each process has more elements, they can sort their lists locally, and then combine the results through a *compare-split* operation. For a compare-split between two processes, each send their sublist to the other, they independantly merge their lists, and keep only the lower or higher half of the combined list.

Bubble sort and Odd-Even transposition

```
# Sequential bubble sort
for (i=n-2; i>=0; i--)
    for (j=0; j<=i; j++)
        compare_exchange(a[j],a[j+1]);
```

The basic bubble sort is inherently sequential, and thus ill-suited for parallelization. Instead we use a Odd-Even transposition.

```
#ODD-Even
for (i=1; i<=n; i++)
    if i is odd
        for (j=0; j<n/2; j++)
            compare_exchange(a[2*j+1], a[2*j+2]);
    else
        for (j=1; j<n/2; j++)
            compare_exchange(a[2*j],[2*j+1]);
#ODD-Even parallel
for (i=1; i<= n; i++)
    if i is odd
        if (i and id are both odd or even)
            compare_exchange_min(id+1)
        else
            compare_exchange_max(id-1)
```

The odd-even transposition is very similar to bubble sort (the same number of operations), but is easier to parallelize. Odd-even transposition is cost-optimal when $p = \mathcal{O}(\log n)$, meaning isoefficiency function is $\Theta(p2^p)$, exponential scaling means it is poorly scalable and it's suited to only a small number of processes.

Shellsort

Shellsort is performed in two stages, where the last stage is an odd-even transposition. The first stage consists of $\log_2 p$ iterations, where first process p_i does a compare split with p_{p-i-1} for $i = 0, \ldots, p/2$. Then the list is split in two at the middle, and half the processes get assigned to each half. Then process p_i does a compare-split with $p_{p/2-i-1}$ for $0 \leq i < p/4$. And so on. The idea is that the first stage makes the second stage converge much faster. Analysing this is thus hard.

```
#1. Split list of length n to p processes
#2. Internally sort each sublist
#3. Execute p phases
    # 4. If (p and id are both even or odd)
        compare-split-min(id+1)
    # 5. else
        compare-split-max(id-1)
```

Quicksort

The sequential quicksort algorithm is

```
function quicksort(A, q, r):
    if q<r:
        x = A[q]
        s = q
        for i in range(q+1, r):
            if A[i] <= x:
                s = s+1
                swap(A, s, i)
        swap(A, q, s)
        quicksort(A, q, s)
        quicksort(A, s+1, r)
```

Here, the pivot selection used is just picking the first element in the list, which can be very bad. More advanced pivot selection rules can be used (such as random pivot etc, outside the scope of our curriculum).

Naive parallel quicksort

```
# 1. Process 0 partitions the original array
# 2. Sends recursive calls to new processes
```

This is inefficient. To be efficient, the partitioning must be done in parallel!

Shared memory

```
# 1. Each process is responsible for a subblock of A
# 2. A pivot is chosen and broadcast, each process
     splits their subblock according to global pivot
# 3. Global array is updated from local subblocks
# 4. All processes are now split in two
     one for left global, one for right
```

After each process has sorted their local list into S_i and L_i, a non-inclusive prefix-sum must be calculated, so that S_i can be placed first in the global list, and then the L_i's .

Message passing This method is similar to the shared memory method, but the rearranging is now somewhat more complicated. The first stage is similar, process P_i rearranges it's local sublist according to some broadcasted pivot. After each process is done, it figures out which processes will be responsible for the S list and which will be responsible for the L list, and send their local subarrays to the right process. Once a process is responsible for an entire subarray (S or L) it locally sorts it.

Both the shared memory and message passing algorithms have isoefficiency functions of $\mathcal{O}(p \log^2 p)$, which results from the pivot-broadcast and prefix sum operations.

Pivot selection

Pivot selection is very important for the performance on quicksort, even more so in the parallel case. If a bad pivot is selected, a process can become idle, lowering the effectivity for the rest of the computation. In the sequential quicksort, a random pivot is a decent choice, as a bad pivot only effects that subsequence, but in the parallel case - it puts a process out of work. A better pivot selection, if we assume the elements to be uniformly distributed, is the local median. This is because (at least for decently sized subarrays) the local median of a subarray will be close to the global median. This pivot will then (hopefully) split each subarray in almost two equally sized lists, giving an even load distribution - and maintaing the uniform distribution property.

Graph Algorithms

An unweighted graph has the adjacency matrix

$$a_{i,j} = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases}$$

A weighted graph has adjacency matrix

$$a_{i,j} = \begin{cases} w(v_i, v_j) & \text{if } (v_i, v_j) \in E \\ 0 & \text{if } i = j \\ \infty & \text{otherwise} \end{cases}$$

Prim's algorithm

For finding a minimum spanning tree. It is a greedy algorithm. First a vertex is chosen arbitrarily, then a new vertex is chosen and included per iteration.

```
tree = [root_vertex]
d[r] = 0
for vertices not in tree:
    if edge(r,v) exists:
        d[v] = w(r,v)
    else:
        d[v] = INF
while tree != V:
    find u so that d[u] = min{d[v]|v not in tree}
    tree.append(u)
    for all v not in tree:
        d[v] = min{d[v], w(u,v)}
```

Parallel Prim's Algorithm
We divide the adjacency matrix among the processes using columnwise 1D block mapping. The process that contains the i'th column, will also have the job of calculating $d_i[v]$ for it's vertices $v \in V_i$. Each iteration begins by process 0 selecting the vertex u that will be added to the MST, it then broadcasts this selection. Each process calculates their new $d_i[u]$ values, and an all-to-one reduction using `MPI_MIN` is used

Dijkstra's

A single-source shortest path algorithm. It is very similar to Prim's algorithm.

```
tree = [source_vertex]
l[s] = 0
for vertices not in tree:
    if edge(s,v) exists:
        l[v] = w(s,v)
    else:
        l[v] = INF
while tree != V:
    find u so that l[u] = min{l[v]|v not in tree}
    tree.append(u)
    for all v not in tree:
        l[v] = min{l[v], l[u]+w(u,v)}
```

The parallel version of Dijkstra's is similar to Prim's. The adjancy matrix is split using columnwise 1D block mapping, so each process get a number of columns and is responsible for computing $l_i(v)$ for $v \in V_i$.

Dijkstra's All-pairs shortest path

To find all-pairs shortest paths with Dijkstra's, we simply run the single-source algorithm n times, using each vertex as the source. This can be parallelized in to ways.

First, in the source-partitioned formulation, we can let each process run the single-source Dijkstra's sequentially on each source. This method has no inter-process communication, so it is obviously cost-optimal. However, as $W = n^3$ and $p \leq n$ we get an isoefficiency function $W = \mathcal{O}(p^3)$, and so this problem is poorly scalable.

The second version, source-parallel formulation, we let p/n processes work each source-partitioned sequence in parallel. This formulation needs some communication, and is cost-optimal if $p \log p / n^2 = \mathcal{O}(1)$, i.e., the algo can use up to $\mathcal{O}(n^2 / \log n)$ processes efficiently.

Floyd's Algorithm

Floyd's is an all-pairs shortest path algorithm. The sequential version is as follows

```
for (k=0;k<n;k++)
    for (i=0;i<n;i++)
        for(j=0;j<n;j++)
            a[i][j] = min(a[i][j], a[i][k]+a[k][j])
```

As $\min(a[i][k], a[i][k] + a[k][j]) = a[i][k]$, and $\min(a[k][j], a[k][k] + a[j][k]) = a[k][j]$ we can run each iteration of k concurrently.

Rowwise 1D block mapping
Each process is responsible for a set of rows of the a-matrix. To update $a[i][j]$, row's a_i and a_k must be known, the process responsible for this entry already knows row a_i but must recieve row a_k. At the begining of the k'th iteration, the owner of the k'th row should broadcast it to all processes.

2D block mapping
The matrix is now divided using a 2D block mapping. At the begging of the k'th iteration, any process that owns part of the k'th row broadcasts it to all processes in it's column, and any process that owns part of the k'th column broadcasts it processes in ist row.

Pipelining Floyd's algorithm Just like for Gausian elimination, we can speed up the 2D block mapping version of Floyd's algorithm by pipelining it. Instead of using broadcast operations, the owner of the $k'th$ row and column send their to their closest neighbors,