Requirements for Parallel Programs

- Real problem(observation, experiments) $\Rightarrow$
  Parallel algorithm(descriptive pseudo code) $\Rightarrow$
  Parallel computer program (programming language code).
- To devise an efficient parallel program we have to identify:
  - concurrency,
  - scalability,
  - locality, and
  - modularity.

Concurrency

- Ability of performing calculation at the same time
  - independent calculation
  - accessible data in the time of calculation
- Sum of integers 1-10 (instead of 10 steps we use 4 steps)
  1. a=1+2, b=3+4, c=5+6, d=7+8, e=9+10
  2. a=a+b, b=c+d, e,
  3. a=a+b
  4. a=a+e
- Can not start step 3 before step 2 is finished..
- But the price is: a need for more adders able to compute concurrently and more memory locations.

Scalability

- Resilience to increasing processor counts.
- Ability to execute a parallel program with a similar efficiency on different number of processors.
- Usually, efficiency decreases with the increased number of processors.
- Program able to use only a fixed number of processors is a bad parallel program.
- Scalability enables portability for protecting software investments.

Locality

- A possibility of a program for exploiting accesses to local memory (same-node).
- Local accesses are less expensive (faster) than accesses to remote memory (different-node). i.e.: read and write are less costly than send and receive.
- The ratio remote/local cost can vary from 10:1 to 1000:1 or greater, depending on the relative performance of the local computer, the network, and the mechanisms used to move data to and from the network.
- Efficiency of a parallel algorithm depends on the ratio of the number of remote to local access.

Modularity

- In modular program design its components are developed separately, as independent modules.
- These modules can be combined or reuse to obtain a new complete program.
- Interactions between modules should be restricted to well-defined interfaces.
- A module implementation can be changed without modifying other program modules.
- The properties of a program can be determined from the specifications of program modules and the main program code.
- Modular design reduces program complexity and facilitates code reuse.

Amdhal's Law

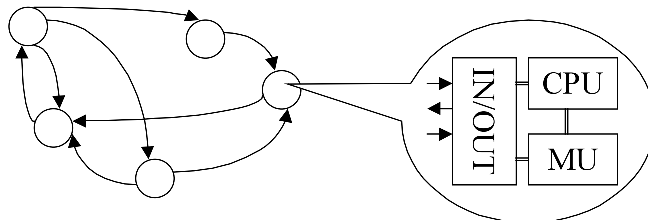
- Let a program Pr be composed of sequential part Sq (e.g.: reading data from disk), and a parallel part Pp that can be ideally parallelized: Pr = Sq + Pp.
- On a single processor Sq takes 10% of the total CPU time, Pp can be implemented in 90% of the total execution time TPr.
- What is the maximal speedup that can be reached with an arbitrary large number of processors?

  $S = T1/Tp = (TSq+TPp) / (TSq + (TPp/p)] =$

  $= 1 + (TPp/TSq) = 1 + (0.9/0.1) = 10$
- The parallel program has to be probably redesigned !

Parallel Programming Model

- Parallel programs can be represented by a set of tasks and channels, and modelled by a directed graph, where vertices represent tasks and edges represent channels.
- A task incorporates sequential program and in/out ports connected to channels.



Task Actions

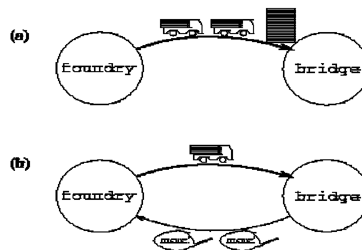
- Read or write from/to local memory,
- Send or receive messages to/from other tasks,
- Create new tasks,
- Terminate task execution,
- Create new channels,
- Change connectivity.

Task Mapping

- If one task requires data from another task, in order to proceed, we talk about **data dependency**.
- If these two tasks are mapped on different computers, send/receive communication is needed, else only read/write to local memory is used.
- Tasks can be mapped on physical processors with no effect on the program semantic. For example, single task or multiple tasks can be mapped on a single processor.
- But after mapping, all processors should be loaded equally because the slowest processor dictates the execution time of the parallel program.

Illustration-Bridge Construction

- Case (a)-two tasks: foundry (girders production) and bridge (bridge construction) and a single channel: trucks transporting girders (overflowing?)



- Case (b): one additional channel for requesting girders and for finishing the job when the bridge is complete.

Other Programming Models

- Message Passing
  - instead of “on channel xx” use “to task yy”
- Single Program Multiple Data (SPMD)
  - fixed number of identical tasks
- Data Parallelism
  - many identical tasks for different pieces of data
- Shared Memory
  - communication by the asynchronous read and write,
  - no explicit communication code,
  - data locks, semaphores

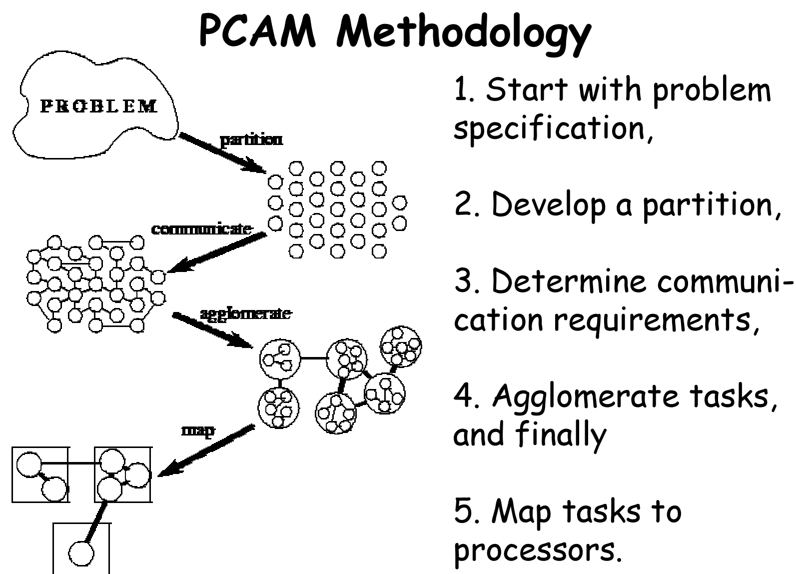
Designing Parallel Algorithms

- Programming problems have several parallel solutions.
- The best parallel solution may differ from an existing sequential algorithm.
- Machine-independent issues (concurrency) should be considered in an **early stage** of design
- Machine-specific aspects of design are **delayed** and are considered in a final stage of design process.

Methodology for Designing Parallel Algorithms

- Composed of four stages:
  - **P**artitioning
  - **C**ommunication
  - **A**gglomeration
  - **M**apping
- Acronym PCAM
- First two stages are focused on concurrency and scalability
- Third and fourth stages search for locality and other performance-related issues.

PCAM Methodology



1. Start with problem specification,

2. Develop a partition,

3. Determine communication requirements,

4. Agglomerate tasks, and finally

5. Map tasks to processors.

Partitioning

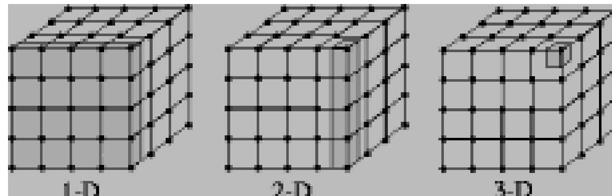
- Partitioning - opportunities for parallel execution.
- Defining a large number of small tasks able to being executed in parallel: fine-grained decomposition.
- Domain decomposition - data first then computation
- Functional decomposition - computation first then data.
- In later design stages the original partition is revisited and
- agglomerate tasks to increase their size, or granularity.

Domain decomposition

- In the domain decomposition approach to problem partitioning, we seek first to decompose the data associated with a problem.
- If possible, we divide these data into small pieces of approximately equal size.
- Next, we partition the computation that is to be performed, typically by associating each operation with the data on which it operates.
- This partitioning yields a number of tasks, each comprising some data and a set of operations on that data. An operation may require data from several tasks.
- In this case, communication is required to move data between tasks.

Domain decomposition (exm.)

- Example of a domain decompositions for a problem involving a three-dimensional grid. One-, two-, and three-dimensional decompositions are possible; in each case, data associated with a single task are shaded.
- A three-dimensional decomposition offers the greatest flexibility and should be used in the early stages of a design.

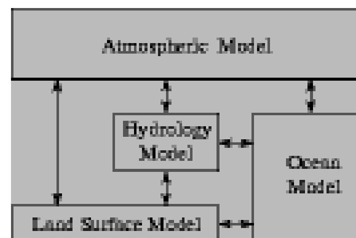


Functional decomposition

- Functional decomposition represents a complementary way of thinking about problems. The initial focus is on the computation that is to be performed rather than on the data manipulated by the computation.
- If we are successful in dividing this computation into disjoint tasks, we proceed to examine the data requirements of these tasks.
- These data requirements may be disjoint, in which case the partition is complete. Alternatively, they may overlap significantly, in which case considerable communication will be required to avoid replication of data.
- This is often a sign that a domain decomposition approach should be considered instead.

Functional decomposition (exm.)

- Functional decomposition in a computer model of climate. Each model component can be thought of as a separate task, to be parallelized by domain decomposition.
- Arrows represent exchanges of data between components during computation: the atmosphere model generates wind velocity data that are used by the ocean model, the ocean model generates sea surface temperature data that are used by the atmosphere model, and so on.



Communication

- The tasks generated by a partition are intended to execute concurrently but cannot, in general, execute independently.
- The computation to be performed in one task will typically require data associated with another task.
- Data must then be transferred between tasks so as to allow computation to proceed.
- This information flow is specified in the communication phase of a design.

Communication (cont.)

- First, we define a channel structure that links, either directly or indirectly, tasks that require data (consumers) with tasks that possess those data (producers).
- Second, we specify the messages that are to be sent and received on these channels.
- In domain decomposition problems, communication requirements can be difficult to determine.
- In contrast, communication requirements in parallel algorithms obtained by functional decomposition are often straightforward: they correspond to the data flow between tasks.

Communication (cont.)

- In *local* communication, each task communicates with a small set of other tasks (its ``neighbors'');
- *global* communication requires each task to communicate with many tasks.

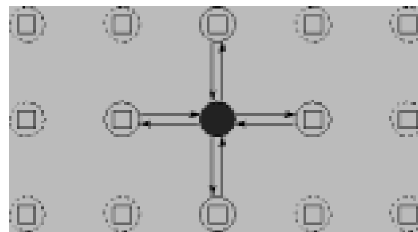
- In *structured* communication, a task and its neighbors form a regular structure, such as a tree or grid;
- *unstructured* communication networks may be arbitrary graphs.

Communication (cont.)

- In *synchronous* communication, producers and consumers execute in a coordinated fashion, with producer/consumer pairs cooperating in data transfer operations;
- in contrast, *asynchronous* communication may require that a consumer obtain data without the cooperation of the producer.

Local Communication

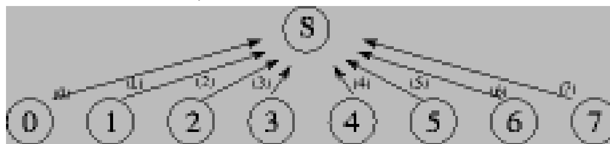
- A local communication structure is obtained when an operation requires data from a small number of other tasks.
- For illustrative purposes, we consider the communication requirements associated with a Jacobi finite difference method, using a five-point stencil to update each element of a 2-D grid.



$$X_{i,j}^{(t+1)} = \frac{4X_{i,j}^{(t)} + X_{i-1,j}^{(t)} + X_{i+1,j}^{(t)} + X_{i,j-1}^{(t)} + X_{i,j+1}^{(t)}}{8}.$$

Global Communication

- Many tasks must participate. When such operations are implemented, it may not be sufficient simply to identify individual producer/consumer pairs. Such an approach may result in too many communications or may restrict opportunities for concurrent execution.
- For example, consider the problem of performing a parallel reduction operation, a centralized sum in S.



- We have N=8 tasks, and each of the 8 channels are connected to S (labels denotes the number of step). This approach takes O(N) steps, not good!

Global Communication (cont.)

- This example illustrates two general problems that can hinder efficient parallel execution in algorithms based on a purely local view of communication:

1. The algorithm is *centralised* : it does not distribute computation and communication. A single task (in this case, the manager S) must participate in every operation.
2. The algorithm is *sequential* : it does not allow multiple computation and communication operations to proceed concurrently.

Global Communication (cont.)

- We first consider the problem of distributing the computation and communication associated with the summation. Let Si=Xi+Si-1.

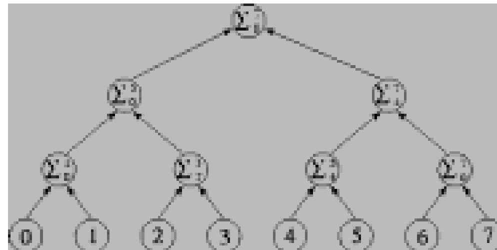
- This algorithm distributes the N-1 communications and additions, but permits concurrent execution only if multiple summation operations are to be performed. (The array of tasks can then be used as a pipeline,through which flow partial sums.)
- A single summation still takes N-1 steps.

Global Communication (cont.)

- Opportunities for concurrent computation and communication can often be uncovered by applying a problem-solving strategy called *divide and conquer*. To solve a complex problem (such as summing N numbers), we seek to partition it into two or more simpler problems of roughly equivalent size (e.g., summing N/2 numbers).
- This process can be applied recursively to produce a set of subproblems that cannot be subdivided further (e.g., summing two numbers).

Global Communication (cont.)

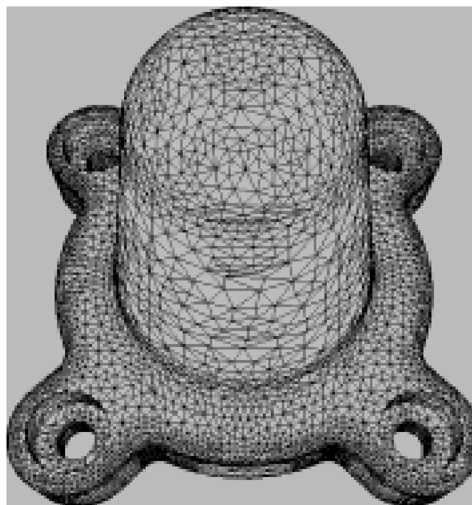
- Tree structure for divide-and-conquer summation algorithm with N=8 . The complete sum is available at the root of the tree after log(N) steps.



- We have distributed the N-1 communication and computation operations and have modified the order in which these operations are performed so that they can proceed concurrently with a regular local communication structure (a small set of neighbors).

Unstructured and Dynamic Communication

- Example of a problem requiring unstructured communication.
- In this finite element mesh generated for an assembly part, each vertex is a grid point.
- Notice that different vertices have varying numbers of neighbours.



Unstructured and Dynamic Communication

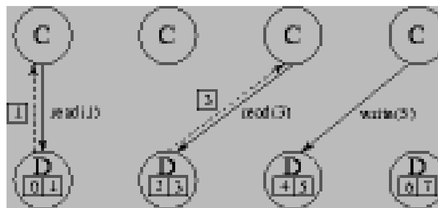
- Here, the channel structure representing the communication partners of each grid point is quite irregular and data-dependent and, furthermore, may change over time if the grid is refined as a simulation evolves.
- Unstructured communication patterns do not generally cause conceptual difficulties in the early stages of a design. (it is simple to define a single task for each vertex).
- But the tasks of agglomeration and mapping becomes more complicated. We have to create tasks of approximately equal size and minimises communication requirements by the least number of intertask edges.

Asynchronous Communication

- The examples considered in the preceding section have all featured synchronous communication, in which both producers and consumers are aware when communication operations are required, and producers explicitly send data to consumers.
- In asynchronous communication, tasks that possess data (producers) are not able to determine when other tasks (consumers) may require data; hence, consumers must explicitly request data from producers.
- This situation commonly occurs when a computation is structured as a set of tasks that must periodically read and/or write elements of a shared data structure (too large or too frequently accessed to be encapsulated in a single task).

Asynchronous Communication (exm.)

- Four computation tasks (C) generate read and write requests to eight data items distributed among four data tasks (D). Solid lines represent requests; dashed lines represent replies.



- One compute task and one data task could be placed on each of four processors so as to distribute computation and data equitably.

Asynchronous Communication (cont.)

A mechanism is needed that allows this data structure to be distributed while supporting asynchronous read/write operations on its components.

1. The data structure is distributed among the computational tasks. Each task both performs computation and generates requests for data located in other tasks (periodical polling for pending requests).
2. The distributed data structure is encapsulated in a second set of tasks responsible only for responding to read/write requests (previous slide) (switching between tasks is needed, no local data for computational tasks.)
3. On a computer that supports a shared-memory, computational tasks can access shared data without any special arrangements (data integrity, read/write order)

Agglomeration

- In the first two stages of the design process, we partitioned the computation to be performed into a set of tasks and introduced communication to provide data required by these tasks.
- The resulting algorithm is still abstract it is not specialized for efficient execution on any particular parallel computer. It may creates many more tasks than there are processors, or the computer is not designed for efficient execution of small tasks.
- In the third stage, *agglomeration*, we move to the concrete. We consider whether it is useful to combine, or agglomerate, tasks (smaller number).
- We also determine whether it is worthwhile to replicate data and/or computation.

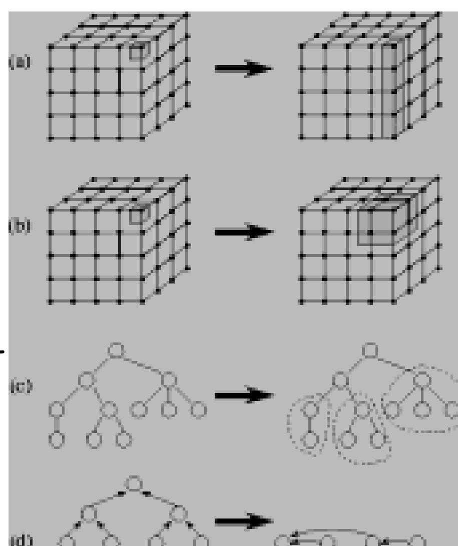
Agglomeration (exm.)

(a) the size of tasks is increased by reducing the dimension of the decomposition from 3->2.

(b) adjacent tasks are combined to yield a decomposition of higher granularity.

(c) subtrees in a divide-and-conquer structure are merged.

(d) nodes in a tree algorithm are combined.



Agglomeration Goals

- Three sometimes-conflicting goals guide decisions concerning agglomeration and replication:
- *reducing communication* costs by increasing computation and communication granularity (communication costs, task creation costs), by replicated computation (trade off), or by agglomeration of tasks thatcannot execute concurrently.
- *preserving flexibility* with larger number of tasks than the number of processors (scalability and mapping) to preserve portability on computers with different number of processors.
- and *reducing software engineering costs*, small changes when parallelizing existing sequential codes.

Mapping

- In the fourth and final stage of the parallel algorithm design process, we specify where each task is to execute.
- The mapping problem does not arise on uniprocessors or on shared-memory computers that provide automatic task scheduling by operating system and associated communication by shard memory.
- Unfortunately, general-purpose mapping mechanisms is not developed yet for scalable parallel computers. In general, mapping remains a difficult problem that must be explicitly addressed when designing parallel algorithms.

Mapping

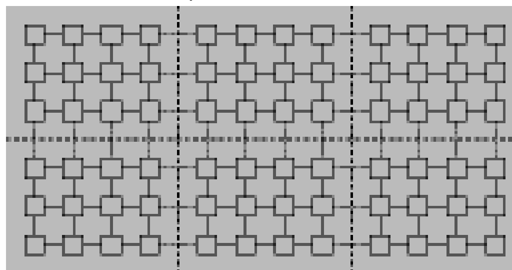
- Our goal in developing mapping algorithms is to minimize total execution time. We use two strategies:

1. We place tasks that are able to execute concurrently, on different processors, so as to enhance concurrency.
2. We place tasks that communicate frequently, on the same processor, so as to increase locality.

- Clearly, these two strategies will sometimes conflict, in addition, resource limitations may restrict the number of tasks that can be placed on a single processor (trade off).
- The mapping problem is known to be NP-complete, meaning that no computationally tractable (polynomial-time) algorithm can exist for evaluating these tradeoffs in the general case.

Mapping (exm. 1)

- Mapping in a grid problem in which each task performs the same amount of computation and communicates only with its four neighbors is straightforward. The heavy dashed lines delineate processor boundaries. The grid and associated computation is partitioned to give each processor the same amount of computation and to minimize off-processor communication.

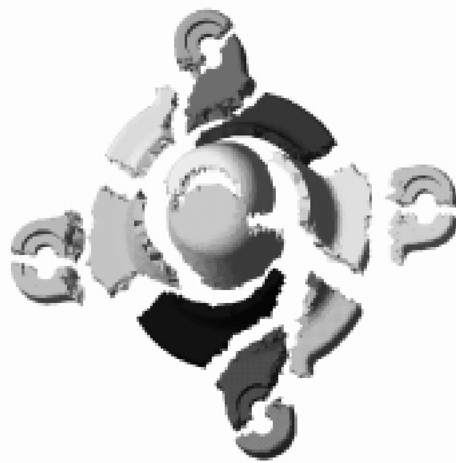


Load Balancing Algorithms

- More complex domain decomposition-based algorithms with variable amounts of work per task and/or unstructured communication patterns. We have to use load balancing algorithms that seek to identify efficient agglomeration and mapping strategies, typically by using heuristic techniques.
- The time required to execute these algorithms must be weighed against the benefits of reduced execution time.
- Probabilistic load-balancing methods have sometimes lower overhead.
- The most complex problems needs dynamic load-balancing strategy in which a load-balancing algorithm is executed periodically.

Load Balancing (exm.)

- A final result of a load balancing algorithm of the object analysed by finite elements (PP cover).
- Recursive bisection techniques are used to partition a domain into subdomains of approximately equal computational cost while attempting to minimize communication costs (number of channels crossing task boundaries).
- Many other techniques.



Still to Do

- We have now completed the design of parallel application with several algorithms developed. However, we are not quite ready to start writing code:
- First, we need to conduct some simple *performance analyses* in order to choose between alternative algorithms and to verify that our design meets performance goals.
- We should also think about the implementation costs of our designs (reusing existing code, how algorithms fit into larger systems of which they may form a part).
- The program complexity can be controlled by the application of modular design techniques.

Modularity

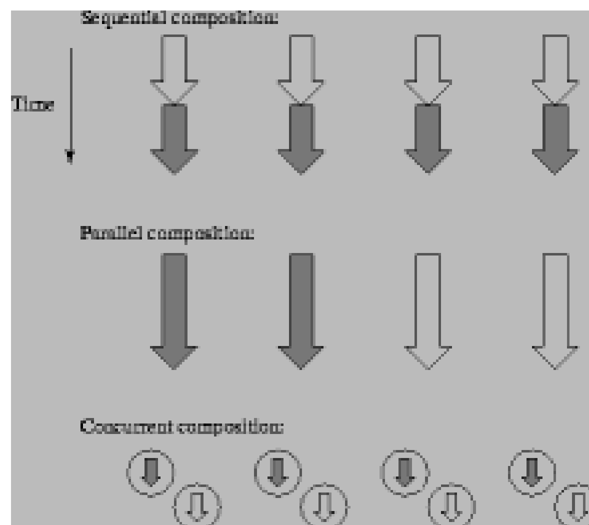
- Real application programs may need to incorporate multiple parallel algorithms, each operating on different data structures and requiring different partitioning, communication, and mapping strategies for its efficient execution.
- The key idea is to encapsulate complex or changeable aspects of a design inside separate program components, or modules, with well-defined interfaces indicating how each module interacts with its environment.
- Complete programs are developed by putting together or *composing*, these modules. (increased reliability and reduced costs, easier to build, change and reuse).

Composition

- We distinguished three general forms of composition that can be used for the modular construction of parallel programs:
- *sequential composition* - two program components execute in sequence on the same set of processors
- *parallel composition* - two program components execute concurrently on disjoint sets of processors
- *concurrent composition* - two program components execute on potentially non disjoint sets of processors.
- MPI's MPMD programming model supports well first two compositions, the full generality of concurrent composition is not generally available.

Composition - Figure

- A program composed of two different components (grey, dark grey).
- It is executing on four processors, with each arrow representing a separate thread of control.



Parallel Algorithm Examples

- In order to introduce Task-Channel model and all other issues mentioned, we present four typical parallel algorithms:

- Finite differences (fixed number of tasks with the same function - SPMD structure),
- Pairwise Interactions (SPMD structure),
- Parallel Search (dynamic creation of tasks), and
- Parameter Study (fixed number of tasks with different functions).

Finite Differences

- Finite differences are a fundamental tool for the solution of partial differential equations.
- A one-dimensional finite difference example, in which we have an initial state represents by vector X^0 of size N and must compute the solution X^T with step-by-step updating of X^t by:

t	X_0	X_1	X_2	X_3	X_4	.....	X_{N-2}	X_{N-1}
t+1	X_0	X_1	X_2	X_3	X_4	.....	X_{N-2}	X_{N-1}

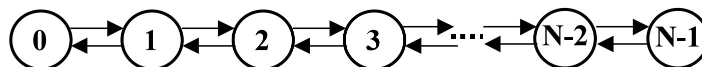
$$0 < i < N-1,\ 0 \le t < T\ :\ X_i^{(t+1)} = \frac{X_{i-1}^{(t)} + 2X_i^{(t)} + X_{i+1}^{(t)}}{4}.$$

Finite Differences (cont.)

- The new value of X^{t+1} can be calculated after all tasks calculate their own value for X^t, and receive values for X^t from the left and the right neighbour.
- Values X_0 and X_{N-1} are boundary values. Boundary conditions have to be devised for the boundary values.
- Let say that the boundary values has fixed values that are equal to the initial values X^0_0 and X^0_{N-1} .
- Communication between nearest neighbours is necessary.

Finite Differences-Parallel Model

- We have N tasks, each responsible for computing $X_0, X_1, ..., X_{N-1}$, in each of T calculation steps.
- Tasks 0 and N-1 are boundary tasks (with i.e.: fixed values), and have a slightly different calculation than other tasks.
- Data transfer is implemented with two bi-directional channels to the right and to the left neighbours.



Finite Differences-Performance Analysis

- Assume that we distribute N tasks (points) among P processors, each responsible for equal number N/P points. Each processor performs the same computation on each grid point and at each time step. Because the parallel algorithm does not replicate computation, we can model computation time in a single time step as: Tcomp = tc*N
- Each processor must exchange a single data point on each boundary (two messages received two sent). The number of all messages is: Tcomm = 2*P*(ts + tw*1)
- The total execution time Tp = tc*N/P + 2*ts + tw

Finite Differences - Algorithm

- Each task i performs the following step algorithm:

```
if (taskID not 0 or N-1) {
   send local data X^t to left and right outports;
   receives X^t from its left and right inports;
   use these values to compute X^(t+1);}
else X^(t+1) = X^t ;
```

- Single Program Multiple Data algorithm.
- All tasks can execute independently between steps and within a single step.
- Execution order is synchronised by the receive operations (no data value is updated at step t+1 until the data values in neighbouring tasks have been updated at step t).

Pairwise Interactions

- Many problems require the computation of all N(N-1) pairwise interactions $I_{i,j}$, $i \neq j$, $i,j=0,\ldots,N-1$. between N data. Interactions may be symmetric, $I_{i,j} = -I_{j,i}$, in which case only N(N-1)/2 interactions need to be computed.
- For N particles:
  - A simple parallel algorithm for the general pairwise interactions problem might create N tasks, one for each particle.
  - N(N-1)/2 independent interactions must be computed. Each particle has to communicate by (N-1) neighbours. The problem calculation complexity is of $O(N^2)$.

Pairwise Interactions - Molecular Dynamics

- In the simulation of Molecular Dynamics the total force f_i has to be calculated in each step for each molecule X_i:

$$f_i = \sum_{j=0}^{N-1} F(X_i, X_j).$$

where $F(X_i,X_j) = -F(X_j,X_i)$, what means that we have symmetric interactions.

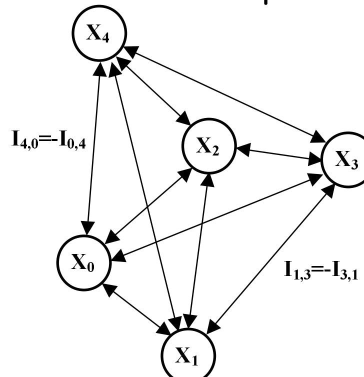
- The local force in x direction $F_x(X_i,X_j)$ is calculated by the Lenard-Jones potential as:

$$F_x(X_i,X_j)=c*(x_i-x_j)*((r_{i,j})^{-14} - 0.5(r_{i,j})^{-8})$$

where $r_{i,j}$ is the distance between particles i and j, x_i the x coordinate of a particle I, and c a constant.

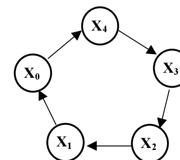
Pairwise Interactions-Parallel Model1

- N=5 particles (tasks)
- N(N-1)/2=10 independent interactions (channels)
- N=5 tasks, N-1 bi-directional channels per task
- Expensive design.



Pairwise Interactions-Parallel Model2

- Instead of totally N(N-1) channels, we can use only N channels: one inport and one outport channel per particle.
- We can create N same tasks, one for each particle, each with its local particle data (particle position, accumulated force).
- Tasks are connected by a directed ring communication structure.



- Each calculation step consist of N-1 send-receive-calculate sub-steps (passes), causing N data to flow around the ring.
- All passes can be executed independently.

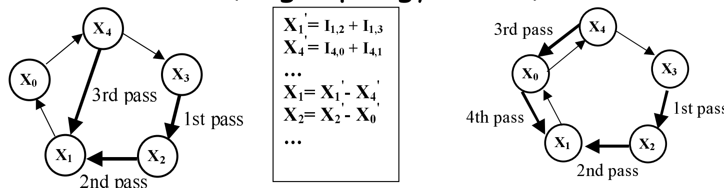
Pairwise Interactions - Algorithm

- Execution order is synchronised in each pass by the receive operations (new interaction is calculated after the reception of new particle data).
- New particle step position is calculated after N-1 passes when the final accumulation of all interactions can be performed in each task.
- Each task perform the same step algorithm:

```
Repeat N-1 times  {
  send particle position, received in the
  previous pass (own position in 1st step),
  on the outport;
  receive next particle position from the inport;
  compute interaction I_i,j and accumulate force;}
calculate new local particle position.
```


Pairwise Interactions - Improvements

- Because of symmetry, we can improve the algorithm, that will need only (N-1)/2 passes per step, instead of previous (N-1) passes.
- 1st approach: to add one additional channel to each task in order to get partially accumulated interactions (ring topology is lost).



- 2nd approach: to add one additional channel in ring and send partially accumulated interactions in the reverse direction (ring topology is saved).

Pairwise Interaction-Performance Analysis

- Assume that we distribute N tasks among P processors, each responsible for equal number of N/P local particles. We have N(N-1)/2 interactions for each task and in each step, so the total computation time is: Tcomp = tc*N(N-1)/2
- Each processor must exchange the actual position and velocities of N/P local particles for each of (P-1) passes. Two messages are needed in the ring (concurrently: one send, one receive):

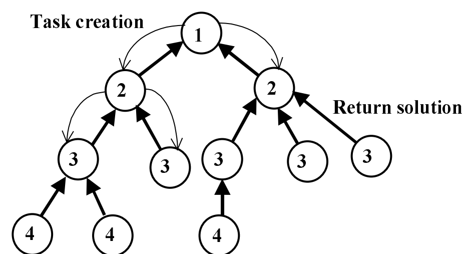
  Tcomm = P*(P-1)*(ts + tw*(2*N/P))
- Ideally, messages on all nodes from the ring will be send in parallel: The parallel execution step time will be: Tp = tc*N(N-1)/2P + (P-1)ts + tw*(2*N).

Search

- The program should explore a search tree looking for nodes that correspond to "solutions".
- This example illustrates the dynamic creation of tasks and channels during program execution.
- A parallel algorithm for this problem can be structured in the following way:
  - A single task is created for the root of the search tree. The task evaluates its node and then, if that node is not a solution, it creates a new task for each `search` call in its child nodes.
  - A channel created for each new task in a tree node is used to return any solutions eventually located in its subtree.

Search - Parallel Model

- New tasks and channels are created in a wavefront starting at the root and progressing down the leafs of the search tree.
- Each node represents a task that call procedure `Search( );`



Search - Algorithm

- Procedure `Search()` checks whether the actual node represents a solution. If not, the algorithm makes recursive calls to expand each of the child nodes.

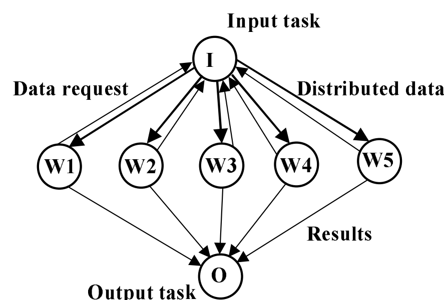
```
Procedure search(A)
    begin
        if (solution(A)) then
           score = eval(A)
           report solution and score
        else
           foreach child A(i) of A
                search(A(i))
            endfor
        endif
    end
```

Parameter Study

- Some computations can be modelled with several tasks that can execute independently, without communication.
- Easy to adapt for parallel execution.
- An example is a parameter study, in which the same computation must be performed using a range of different input parameters.
- The parameter values are read from an input file, and the results are written to an output file.

Parameter Study - Parallel Model

- Workers (W) request parameters from the input task (I) that sends portion of parameters to the selected W. After finishing the calculation Wi send results to the output task (O).



Parameter Study-Parallel Model (cont.)

- Besides to the already known one-to-one communication (parameters from I to W), many-to-one connections are needed to store results and send requests from Wi to O and Wi to I, respectively.
- The I and O tasks receive data from Wi in an arbitrary order, therefor the program is non-deterministic.
- Wi tasks can be of the same or different complexity, computers used can also be of the same (homogenous network) or different type (heterogeneous network).

Parameter Study-Algorithm

- We have a fixed number of Wi tasks with different calculation.
- Tasks I and O have also different algorithms.

Worker task Wi:

```
Do until Data available
    Request new data
    Receive new data
    Perform calculation
    Send results
Enddo
Send Termination
    message to O
Terminate task
```

Input task I:

```
Do until Data available
    Receive Wi request
    Send new data
Enddo
Du until No_more_data
    sent to all Wi
Terminate task
```

Output task O:

```
Receive new results
Save results
Do until Termination
    message received
    from all Wi
Enddo
Terminate task
Terminate Execution
```