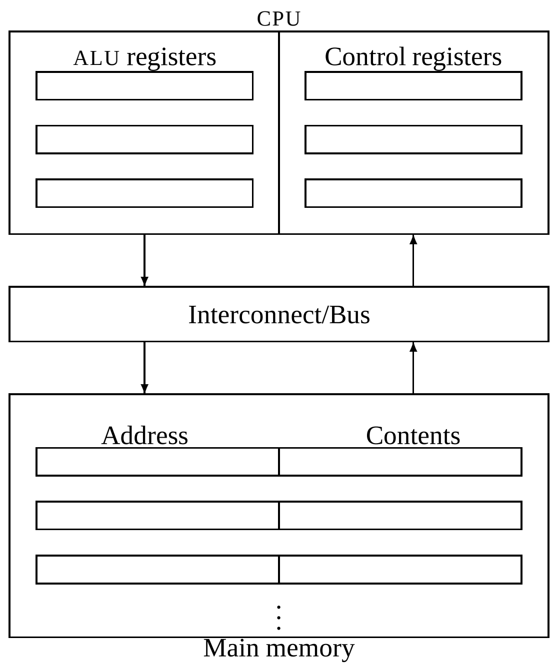
Parallel Hardware and Parallel Software

von Neumann Architecture

- Describes a computer system as a CPU (or core) connected to the main memory through an interconnection network
- Executes only one instruction at a time, with each instruction operating on only a few pieces of data
- Main memory has a set of addresses where you can store both instructions and data
- CPU is divided into a control unit and an ALU
  - Control unit decides which instructions in a program need to be executed
  - ALU executes the instructions selected by control unit
  - CPU stores temporary data and some other information in registers
  - Special register PC in the control unit



- Interconnect/bus used to transfer instructions and data between CPU and memory
  - Data/instructions *fetched/read* from memory to CPU
  - Data/results *stored/written* from CPU to memory
- Separation of memory and CPU known as von Neumann bottleneck
  - Problem because CPUs can execute instructions more than a hundred time faster than they can fetch items from main memory

Modifications to the von Neumann Model

- Achieved by caching, virtual memory, and low-level parallelism

Processes, multitasking, and threads

- OS manages hardware and software resources; selects the processes to run and time to run them; allocates memory and other resources
- Process as an abstraction of a running program; process control block
- Multitasking – Concurrent execution of multiple processes; possibly on a single core using time slices or quanta
- Threads – Lightweight processes
  - Process may block on a multitasking OS
  - Threading allows a programmer to divide the processes into parts that execute concurrently so that the blockage of one part does not impede other parts
  - Faster to switch between threads than processes
    - They have their own activation record (program counter and call stack) to allow independent execution, but share most of the other resources with other threads in the process
  - Threads are forked off a process and join the process upon termination

Parallel Algorithm Design

- Parallelizing a serial program
  - Divide the work among processes/threads
  - Ensure load balancing
  - Minimize communications and synchronization steps
- Importance of abstraction and modularity
- Task/channel model
  - A simple model for parallel programming
  - Facilitates the development of efficient parallel programs for distributed memory parallel computers
  - Defines a computation as a set of tasks connected by channels

Task/channel model

- Represents parallel computation as a set of tasks that may interact with each other by sending messages through channels
- Parallel computation
  - Two or more tasks executing concurrently
  - Number of tasks may vary during program execution
- Task
  - Sequential program and its local storage, along with a collection of I/O ports
    - Effectively a virtual von Neumann machine
    - A set of in-ports and out-ports define its interface to the environment
  - Local storage contains instructions and data for the program
  - Sends local values to other tasks via output ports
  - May receive data values from other tasks via input ports
  - A task can perform four basic operations in addition to reading/writing local memory

    1. Send a message
    2. Receive a message
    3. Create tasks
    4. Terminate a task
  - Task may be mapped to physical PE; mapping does not affect the program semantics
    * Multiple tasks may be mapped to a single PE

- Channel
  - Link between two tasks over which messages can be sent/received
  - Connects the in-port of one task to the out-port of another
  - May be created or deleted dynamically; references to channels (ports) can be included in messages to allow dynamic variation in connectivity
  - Implemented as a message queue
    * Queue connects one task's output port to the other task's input port
    * Queue preserves the order in which messages are sent/received
    * A sender can place messages on the queue and a receiver can remove messages
    * The queue is said to be *blocking* if there are no messages available for removal
  - Blocked task
    * If a task tries to receive a value and none is available, the receiving task is blocked (synchronous task)
    * A sending task is never blocked (asynchronous task), even if the previous message sent by the same task has not yet been received
      · Send operation completes immediately
- Local access of private data are easily distinguished from nonlocal data access that occurs over channel
  - Data in a task's local memory are *close*; other data are *remote*
  - Local data access is much faster than nonlocal data access
  - Channel abstraction provides a mechanism to indicate that computation in one task requires data in another task to proceed; termed data dependency
- Execution time of parallel algorithm
  - Period during which any task is active
  - Starting time is when all tasks simultaneously begin executing
  - End time is when the last task has stopped executing

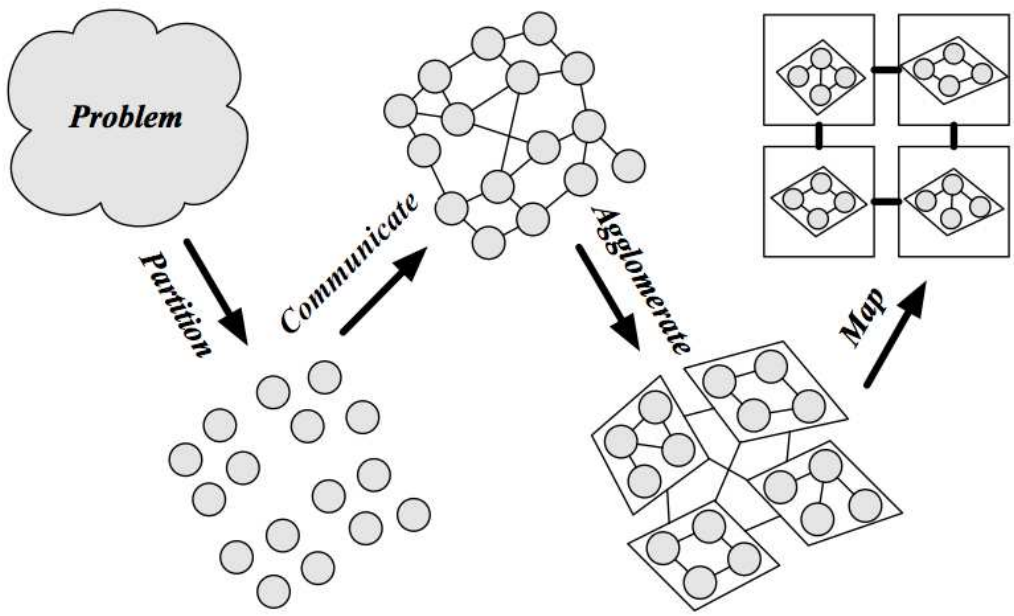
Foster's design methodology

- Four step process for designing parallel algorithms
- Encourages development of scalable parallel algorithms by delaying machine-dependent considerations to later steps
- Example: Laplace equation in 1D [Michael Heath]
  - Integral transform to represent and analyze linear systems using algebraic methods
  - Resolves a function or signal into its moments
  - Used for the analysis of linear time-invariant systems such as electrical circuits, harmonic oscillators, and optical devices
  - Often interpreted as a function from time domain into frequency domain

- Given Laplace equation in 1D
$$u''(t) = 0$$
on interval $a < t < b$ with boundary conditions
$$u(a) = \alpha, \;\; u(b) = \beta$$
- Seek approximate solution values $u_i \approx u(t_i)$ at mesh points $t_i = a + ih$, $i = 0, \ldots, n+1$, where $h = \frac{b-a}{n+1}$
- Finite difference approximation
$$u''(t_i) \approx \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2}$$
yields tridiagonal system of algebraic equations
$$\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} = 0, \;\; i = 1, \ldots, n$$
for $u_i, i = 1, \ldots, n$, where $u_0 = \alpha$ and $u_{n+1} = \beta$
- Starting from initial guess $u^{(0)}$, compute Jacobi iterates
$$u_i^{(k+1)} = \frac{u_{i-1}^{(k)} + u_{i+1}^{(k)}}{2}, \;\; i = 1, \ldots, n$$
for $k = 1, \ldots$ until convergence
- Define n tasks, one for each u_i, $i = 1, \ldots, n$
- Task i stores initial value of u_i and updates it at each iteration until convergence
- To update u_i, necessary values of u_{i-1} and u_{i+1} are obtained from neghboring tasks $i-1$ and $i+1$
$$u_1 \;\underset{\leftarrow}{\rightarrow}\; u_2 \;\underset{\leftarrow}{\rightarrow}\; u_3 \;\underset{\leftarrow}{\rightarrow}\; \cdots \;\underset{\leftarrow}{\rightarrow}\; u_n$$
- Tasks 1 and n determine u_0 and u_{n+1} from boundary conditions
- Program

```
initialize u[i]
for k = 1, ...
    if ( i > 1 ) send u[i] to task i-1     // Send to left neighbor
    if ( i < n ) send u[i] to task i+1     // Send to right neighbor
    if ( i < n ) recv u[i+1] from task i+1 // Receive from right neighbor
    if ( i > 1 ) recv u[i-1] from task i-1 // Receive from left neighbor
    u[i] = ( u[i-1] + u[i+1] ) / 2         // Update my value
end
```

- Mapping tasks to processors
  - Tasks must be assigned to physical processors for execution
  - Tasks can be mapped to processors in various ways, including multiple tasks per processor
  - Semantics of program should not depend on number of processors or particular mapping of tasks to processors
  - Performance usually sensitive to assignment of tasks to processors due to concurrency, workload balance, and communication patterns
  - Computational model maps naturally onto distributed memory multicomputer using message passing
- Four-step design methodology: partition, communicate, agglomerate, map

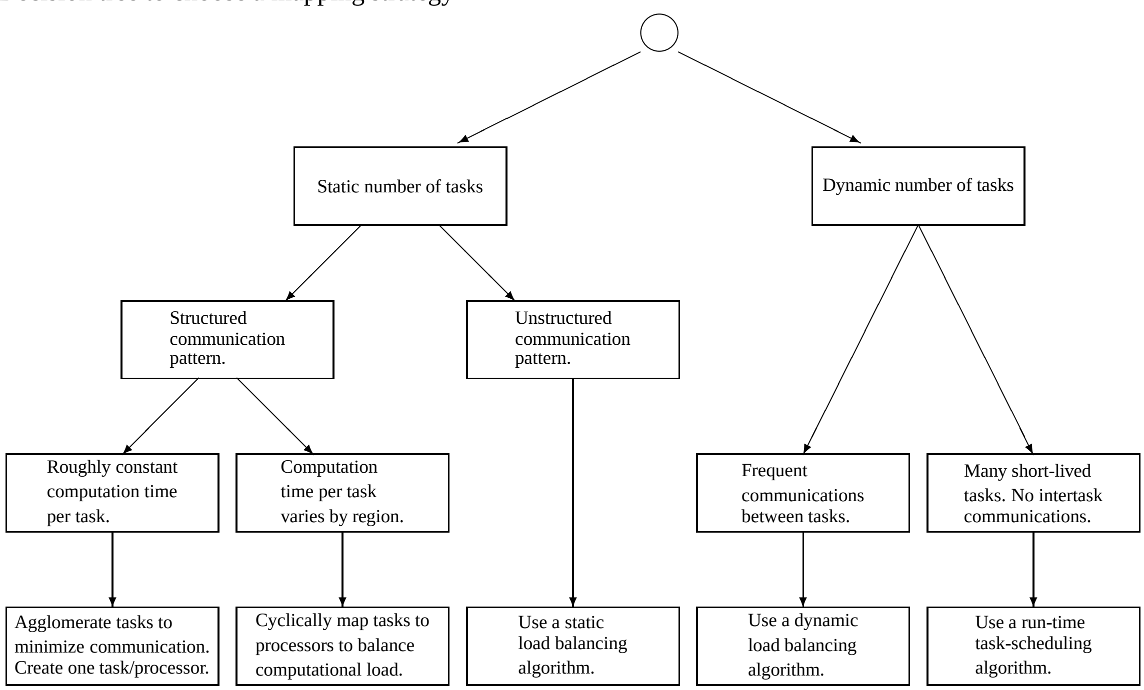


1. Partition
   - Decompose problem into primitive tasks, maximizing number of tasks that can execute concurrently
     * Use a data-centric approach or a computation-centric approach to decompose the problem
   - Data-centric approach/Domain decompositon
     * Divide data into pieces and then, determine how to associate communications with the data
     * Focus on largest/most frequently accessed data structure in program
     * Example: Consider a 3D matrix as the data structure targeted for decomposition
       · Partition matrix into a collection of 2D slices, giving a 1D collection of primitive tasks
       · Partition matrix into a collection of 1D slices, giving a 2D collection of primitive tasks
       · Consider each matrix element individually, giving a 3D collection of primitive tasks
   - Functional decomposition
     * Divide the computation into pieces and associate data items with individual computations
     * Image processing through pipelining
   - Each decomposition piece is called a primitive task
     * At least an order of magnitude more primitive tasks than number of processors in target parallel computer
     * Redundant computations and redundant data structure storage are minimized
     * Primitive tasks are roughly the same size
     * The number of tasks is an increasing function of problem size
2. Communication
   - Determine communication pattern among primitive tasks, yielding task graph with primitive tasks as nodes and communication channels as edges
   - Overhead in a parallel algorithm (not required in sequential task)
   - Local communication
     * Task needs values from a small number of other tasks
   - Global communication
     * Significant number of primitive tasks must contribute data
     * Compute the sum of values held by primitive processes
   - Foster's checklist – minimizing overhead

    - Communication operations are balanced among tasks
    - Each task communicates with only a small number of neighbors
    - Tasks can perform communications concurrently
    - Tasks can perform their computations concurrently

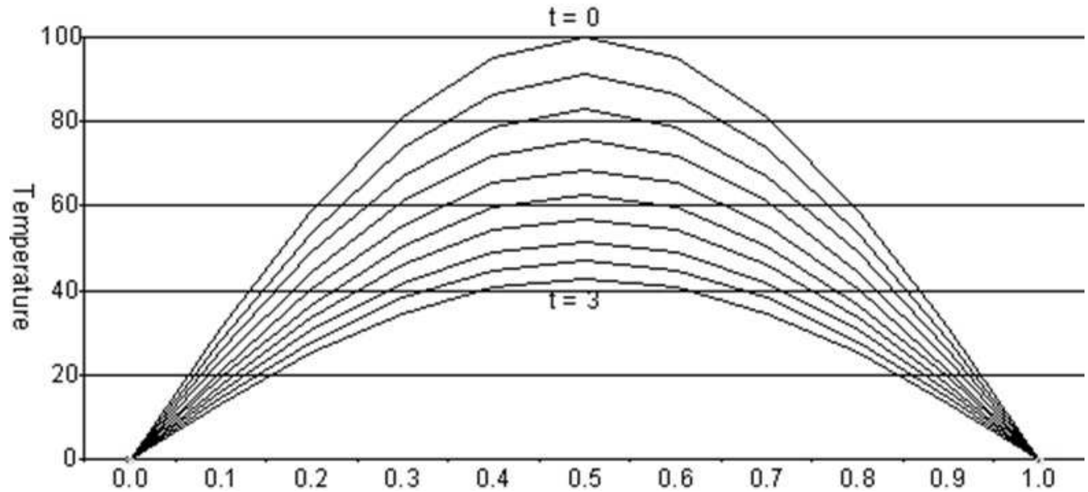
3. Agglomeration
   - Combine groups of primitive tasks to form fewer but larger tasks, thereby reducing communication requirements (lower communication overhead)
     * If number of tasks is several order of magnitudes larger than the number of PEs, the creation of those tasks will create a significant overhead
     * Last two steps depend on target architecture (centralized multiprocessor or multicomputer)
   - Increasing the locality of parallel algorithm; lower communications overhead
     * Agglomerate primitive tasks that communicate with each other
     * Eliminate communication because data values for primitive tasks are in memory of consolidated task
   - Combine groups of sending and receiving tasks
     * Reduce the number of messages being sent
     * Send fewer longer messages to reduce per message overhead (message latency)
   - Maintain the scalability of parallel design
   - Reduce the software engineering costs
   - Foster's checklist
     * Increase in locality of parallel algorithm
     * Replicated computations take less time than the communications they replace
     * Amount of replicated data is small enough to allow the algorithm to scale
     * Agglomerated tasks have similar computational and communications costs
     * Number of tasks is an increasing function of problem size
     * Number of tasks is as small as possible, yet at least as great as the number of processors in likely target computers
     * Trade-off between the chosen agglomeration and the cost of modifications to existing sequential programs is reasonable
4. Mapping
   - Assign consolidated tasks to processors, subject to tradeoffs between communication costs (minimize) and concurrency (maximize)
   - Processor utilization
     * Average percentage of time the processors are actively executing tasks necessary for solving the problem
     * Maximized when computation is balanced evenly
   - Interprocess communication
     * Increases [Decreases] when two processes connected by a channel are mapped to different [same] processors
   - Maximization of processor utilization and minimization of interprocess communication are conflicting goals
     * Finding an optimal solution is NP-hard problem
   - Static load-balancing algorithm
     * Executed before the program begins running to determine mapping strategy
   - Dynamic load-balancing algorithm
     * Used when tasks are created and destroyed at run-time
     * Communication or computational requirements vary widely between processes
     * Algorithm invoked occassionally *during* the execution of parallel programs
   - Centralized task scheduling algorithms
     * Processors divide into one manager and many workers

    - Workers request tasks from manager
    - Single manager becomes the bottleneck
  - Distributed task scheduling algorithms
    - Each processor maintains its own list of available tasks
    - Push strategy – Processors with many available tasks send some to neighboring processors
    - Pull strategy – Processors with no work ask neighboring processors for work
    - Difficult to know when all sub-tasks have completed
  - Foster's checklist
    - Designs based on one task per processor and multiple tasks per processor have been considered
    - Both static and dynamic allocation of tasks to processors have been evaluated
    - If a dynamic allocation of tasks to processors has been chosen, the task allocator is not a bottleneck to performance
    - If a static allocation of tasks to processors has been chosen, the ratio of tasks to processors is at least 10:1
  - Decision tree to choose a mapping strategy



Boundary Value Problem

- Thin rod of length 1 unit made of uniform material surrounded by a blanket of insulation
- Temperature changes along the length of rod are result of heat transfer at the ends of rod and heat conduction along the length of rod
- Both ends of rod are exposed to an ice bath at temperature $0°$C
- Initial temperature at distance x from end of rod is $100 \sin(\pi x)$
  - The rod gradually cools over time
- Temperature at any point of rod at any point in time modeled by a differential equation
- Differential equation solved on computer by finite difference method to get an approximate solution as shown below



- Finite difference method
  - Stores temperatures in a 2D matrix
  - Each row contains temperature distribution of the rod at some point
  - Rod divided into n sections of length h, $n+1$ elements in each row
  - Time from 0 to T divided into m discrete entities of length k; $m+1$ rows in the matrix
  - Initial temperature distribution along the length of rod represented by points in bottom row (known values)
  - Temperature at the ends of rod represented by left and right edges of grid (known values)
  - $u_{i,j}$ represents the temperature of rod at point i at time j
  - $u_{i,j+1}$ is computed by

    $$u_{i,j+1} = ru_{i-1,j} + (1-2r)u_{i,j} + ru_{i+1,j}$$

    where $r = k/h^2$
- Partitioning
  - One data item per grid point
  - Associate one primitive task with each grid point, leading to 2D domain decomposition
- Communication
  - Draw channels between tasks to show the dependence
  - Task $u_{i,j+1}$ requires values of $u_{i-1,j}$, $u_{i,j}$, and $u_{i+1,j}$
  - Each task has three incoming channels and three outgoing channels
- Agglomeration and mapping
  - Later tasks depend on earlier tasks; vertical paths from bottom to top
  - Agglomerate all tasks associated with each point in the rod
  - Task/channel graph reduced to a single row; much simpler
    * Linear array of tasks, each communicating solely with its neighbors
  - The number of tasks is static and the communication pattern between them is regular; each task performs the same computation
    * Create one task per processor
    * Agglomerate primitive tasks to balance computational workload and minimize communication

- Analysis
  - Rod divided into n pieces of size h
  - Let χ represent the time needed to compute $u_{i,j+1}$, given $u_{i-1,j}$, $u_{i,j}$, and $u_{i+1,j}$
  - Using single processor to update $n-1$ interior values requires time $(n-1)\chi$
  - m time steps in the algorithm give the total execution time of sequential algorithm as $m(n-1)\chi$
  - Computation of parallel algorithm time
    - p processors; each processor responsible for equal size portion of rod's elements
    - Computation time for each iteration: $\chi\lceil(n-1)/p\rceil$
    - Account for communication time as well
    - Each processor sends two values and receives two values from neighbors
    - Let λ be the time required to send/receive one value, giving communication time as 2λ
    - Send and receive may overlap in time (proceed concurrently)
    - Overall parallel algorithm execution time: $\chi\lceil(n-1)/p\rceil + 2\lambda$
    - For m iterations, the time is: $m(\chi\lceil(n-1)/p\rceil + 2\lambda)$

Finding the maximum

- The above solution to compute the temperature distribution is approximate
- For each of the m points in the rod, difference between computed solution x and correct solution c is given by $|(x-c)/c|$
- Modify the parallel algorithm to find the maximum error
- Given a set of n values $a_0, a_1, \ldots, a_{n-1}$ and an associative binary operator $\oplus$, reduction is defined as

$$a_0 \oplus a_1 \oplus a_2 \oplus \cdots \oplus a_{n-1}$$

  - Addition is an example of an associative binary operator
  - Finding the sum $a_0 + a_1 + a_2 + \cdots + a_{n-1}$ is an example of a reduction
  - Maximum and minimum of two numbers are also associative binary operators
- Reduction requires $n-1$ operations giving a time complexity of $\Theta(n)$ on a sequential computer
  - How quickly can we do it on a parallel machine?
- Partitioning
  - n values in the list, divide into n pieces
  - Associate one task per piece
  - Goal is to find the sum of all n values
- Communication
  - Set up communication channels between tasks
  - Channel from task A to task B allows B to compute the maximum of values held by two tasks
  - In one communication step, a task may send/receive one message
  - The task holding the maximum at the end of communication is called root task
  - Time λ to communicate a value to another task and time χ to find maximum of the two
  - Overall time: $(n-1)(\lambda + \chi)$ – worse than sequential
    - Communication time is $(n-1)\lambda$ because root task must receive $n-1$ messages

  - Create a tree-like topology; binary tree with 1, 2, 4, 8 nodes
  - Depth of tree given by $k = \log n$
  - Overall time reduces to $\log n(\lambda + \chi)$

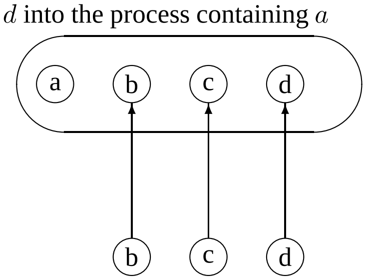
- Agglomeration and mapping
  - Number of tasks is static, computation per task is trivial, communication pattern is regular
  - Agglomerate tasks to minimize communication
    * Assign n/p leaf tasks to each of the p processors

- Analysis
  - χ: time needed to perform binary operation
  - λ: time needed to communicate a value from one task to another via channel
  - Divide n values evenly among p tasks; each task has at most $\lceil n/p \rceil$ values
  - All tasks perform concurrently, time needed to compute subtotals is $(\lceil n/p \rceil - 1)\chi$
  - Reduction of p values distributed among p tasks performed in $\lceil \log p \rceil$ communication steps
  - Receiving process waits and performs reduction requiring time $\lambda + \chi$
  - $\lceil \log p \rceil$ communication steps yield overall time for parallel program as

$$(\lceil n/p \rceil - 1)\chi + \lceil \log p \rceil (\lambda + \chi)$$

The n-body problem

- Parallelize a sequential algorithm in which computation is performed on every pair of objects
- Simulate the motion of n particles of varying mass in two dimensions due to gravitational pull
- During each iteration, compute new position and velocity vector of each particle, given the position of all other particles
  - Complexity of $\Theta(n^2)$ for every iteration for n objects
- Partitioning
  - One task per particle
  - To compute the location of the particle, the task must know the location of all other particles
- Communication
  - `gather` operation
    * Global communication that takes a dataset distributed among a group of tasks and collects the items on a single task
    * Concatenation of data items b, c, and d into the process containing a



  - `all-gather` operation
    * Similar to gather, except that at the end of communication, every task has a copy of the entire dataset

    * Useful in current context to update the location of every particle
  - Can be accomplished by putting a channel between every pair of tasks
    * During each communication step, each task sends its vector element to one other task
    * After $n - 1$ communication steps, each task has the positions of all other particles
  - Possible to improve communication performance to achieve above in logarithmic number of steps
    * Exchange one particle between every pair of processors
    * Exchange two particles between odd numbered processors and two between even numbered processors
    * Continue till all processors have all particles, with increasing number of particles at every step
    * Achieved by hypercube topology

- Agglomeration and mapping
  - Generally, $n \gg p$
  - Assume that n is a multiple of p
  - Agglomerate n/p particles per task
  - `all-gather` communication requires $\log p$ steps
    * In the first step, length of messages is n/p
    * In the second step, length of messages is $2n/p$

- Analysis
  - Derive an expression for execution time of the algorithm
  - λ is the latency to initiate communication
  - Bandwidth β represents the number of data items sent over a channel in one unit of time
  - Sending a message with n items now requires $\lambda + n/\beta$ units of time
  - Communication time for each iteration

$$\sum_{i=1}^{\log p} \left(\lambda + \frac{2^{i-1}n}{\beta p} \right) = \lambda \log p + \frac{n(p-1)}{\beta p}$$

  - Each task responsible for performing gravitational force computation for n/p list elements
    * Time needed for computation denoted by χ
    * Computation time for parallel algorithm is $\chi(n/p)$
  - From above, expected parallel execution time per iteration is

$$\lambda \log p + \frac{n(p-1)}{\beta p} + \chi \frac{n}{p}$$

Adding data input

- Parallel program inputs the original positions and velocity vectors for n particles
  - Assume that a single task responsible for all I/O (I/O task)
  - Open data file and read the position and velocities of n particles
  - Time needed to input or output n data elements: $\lambda_{io} + n/\beta_{io}$
  - Time to read the position (2 data items as x and y coordinates) and velocities of all n particles: $\lambda_{io} + 4n/\beta_{io}$

- Communication
  - Break up input data into pieces to assign n/p elements to each task

- `scatter` operation – reverse of `gather`
- Send the correct n/p particles to each task in turn
  - $p-1$ messages, each of length $4n/p$
  - Time used: $(p-1)(\lambda + 4n/(p\beta))$
  - Not efficient because communication is not balanced among processors
- Derive a scatter operation requiring $\log p$ communication steps
  - Send half the list to another task
  - Next, each process sends quarter list to previously inactive tasks
  - And keep on going by sending half of previous step
  - Time required for this is
$$\sum_{i=1}^{\log p} \left(\lambda + \frac{4n}{2^i p\beta} \right) = \lambda \log p + \frac{4n(p-1)}{\beta p}$$
- Data transmission time is identical for both algorithms
  - Task/channel model supports the concurrent transmission of messages from multiple tasks, as long as they use different channels, and no two active channels have the same source or destination task

• Analysis

- Derive an expression for the total expected execution time of the parallel n-body algorithm
- I/O of positions and velocities of n particles is a completely sequential operation requiring time
$$2(\lambda_{io} + 4n/\beta_{io})$$
- Scattering at the beginning and gathering particles at the end of the computation requires time
$$2\left(\lambda \log p + \frac{4n(p-1)}{\beta p}\right)$$
- Each iteration of parallel algorithm requires an all-gather communication of particles' position, requiring time
$$\lambda \log p + \frac{2n(p-1)}{\beta p}$$
- Each processor performs its share of computation, requiring time
$$\chi \left\lceil \frac{n}{p} \right\rceil (n-1)$$
- If algorithm executes for m iterations, overall execution time of parallel computation is about
$$2\left(\lambda_{io} + \frac{4n}{\beta_{io}}\right) + 2\left(\lambda \log p + \frac{4n(p-1)}{\beta p}\right) + m\left(\lambda \log p + \frac{2n(p-1)}{\beta p} + \chi \left\lceil \frac{n}{p} \right\rceil (n-1)\right)$$

<u>Sieve of Eratosthenes</u>

Sequential algorithm

```
Create a Boolean array from 1 to n
Mark all values as true
k = 2
while k^2 < n
    Change all multiples of k between k^2 and n to false
    Find smallest index p > k that contains true
    k = p
The indices that are true represent prime numbers
```

- Not practical to find large primes
- Complexity of algorithm is $\Theta(n \ln \ln n)$; n is exponential in number of digits

Source of parallelism

- Domain decomposition
  - Algorithm involves marking the elements of the array representing integers
  - Break the array into $n - 1$ elements
  - Associate a primitive task with each of these elements
- Key parallel task
  - Change all multiples of k between k^2 and n to `false`

```
for ( j = k * k; j <= n; j += k )
    p[j] = ( j % k ) != 0;
```

- Two communications needed to change the value of k in the main loop
  - Reduction to find the value of k (smallest k that is `true`)
  - Broadcast to convey new k to all tasks
  - Problem: Too many reduction/broadcast operations
- Agglomeration goals
  - Consolidate tasks to utilize reasonable number of processors
  - Reduce communication costs
  - Balance computations among processes

Data decomposition options

- Final grouping of data elements – the result of partitioning, agglomeration, and mapping
- Interleaved data decomposition
  - Process i responsible for indices $i, i + p, i + 2p, \ldots$
  - Given an index i, it is easy to determine the *owner* of that index (process $i\%p$)
  - May lead to significant load imbalance among processes
    - Two processes marking multiples of 2
    - Process 0 marks $\lceil (n - 1)/2 \rceil$ elements; process 1 marks none
  - Finding the next prime number may still require some sort of reduction/broadcast
- Block data decomposition
  - Balanced loads
  - More complicated to determine owner if n is not a multiple of p
  - Divide the array into p contiguous blocks of roughly equal size
    - Problem if n is not a multiple of p
    - Let $n = 1024$ and $p = 10$; $n/p = 102$.
      - If we give every process 102 elements, there will be 4 left over

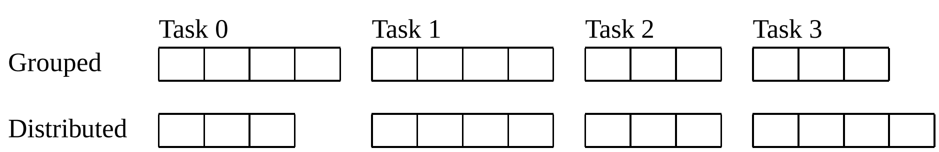
        - We cannot give every process 103 elements because the array is not that large
    - Give first $p-1$ processes $\lceil n/p \rceil$ processes and give the leftover to process p
        - There may be no elements left for process p
        - Complicates logic of programs if processes exchange values
        - Leads to less efficient utilization of communication network
- Balance workload by assigning either $\lceil n/p \rceil$ or $\lfloor n/p \rfloor$ elements to each process
- Questions
    1. What is the range of elements controlled by a given process?
    2. Which process controls a given element?
- Method 1
    - Compute $r = n\%p$
    - If $r == 0$ every process gets a block of size n/p
    - Otherwise
        - First r blocks have size $\lceil n/p \rceil$
        - Remaining $p-r$ blocks have size $\lfloor n/p \rfloor$
    - First element controlled by process i: $i\lfloor n/p \rfloor + \min(i, r)$
    - Last element controlled by process i: $(i+1)\lfloor n/p \rfloor + \min(i, r) - 1$
    - Process controlling element j:

$$\max\left(\left\lfloor \frac{j}{\lfloor n/p \rfloor + 1} \right\rfloor, \left\lfloor \frac{j-r}{\lfloor n/p \rfloor} \right\rfloor\right)$$

    - The expressions for the first and last element are easy to compute and can be saved for each process at the beginning of algorithm
    - The expression to find the controlling process for element j is more complex and needs to be computed on the fly (not good)
- Method 2
    - Scatter larger blocks among processes
    - First element controlled by process i: $\lfloor in/p \rfloor$
    - Last element controlled by process i: $\lfloor (i+1)n/p \rfloor - 1$
    - Process controlling an element j

$$\left\lfloor \frac{p(j+1)-1}{n} \right\rfloor$$

- Distributing 14 elements among four tasks



- Method 2 is superior because it requires fewer operations to perform the three common block management operations
    - Even better as integer division automatically gives floor

• Block decomposition macros

- Applicable to any parallel program
- Define three C macros to be used for block limits and ownership

```
#define BLOCK_LOW(id,p,n) ((id)*(n)/(p))
#define BLOCK_HIGH(id,p,n) (BLOCK_LOW((id)+1,(p),(n)) - 1 )
#define BLOCK_SIZE(id,p,n) (BLOCK_HIGH((id),(p),(n)) - BLOCK_LOW((id),(p),(n)) + 1)
#define BLOCK_OWNER(index,p,n) (((p)*((index)+1)-1)/(n))
```

– `BLOCK_LOW` gives the fi rst index controlled by the process

  – `BLOCK_HIGH` gives the last index controlled by the process
  – `BLOCK_SIZE` gives the number of elements controlled by the process
  – `BLOCK_OWNER` evaluates to the rank of the process controlling the element of the array

- Local index vs global index
  - Limit (localize) the indices within overall (global) array

	Task 0			Task 1				Task 2			
Global index	0	1	2	3	4	5	6	7	8	9	10
Local index	0	1	2	0	1	2	3	0	1	2	3

    * Local index varies from 0 to 2 or 3, depending on process
    * Global index varies from 0 to 10
    * Sequential code uses global index; we need to substitute to local index when working with parallel code
  - Sequential program

```
for ( i = 0; i < n; i++ )
{
    ...
}
```

  - Parallel program

```
size = BLOCK_SIZE(id,p,n);
for ( i = 0; i < size; i++ )
{
    gi = i + BLOCK_LOW(id,p,n);      // Global index; replaces i
    ...
}
```

- Ramifications of block decomposition
  - Largest prime used to sieve integers up to n is $\sqrt{n}$
  - First process has $\lfloor n/p \rfloor$ elements
    * It has all sieving primes if $p < \sqrt{n}$
    * Reasonable assumption since n is expected to be in millions
  - Fast marking
    * Block decomposition allows same marking as sequential algorithm

$$j, j + k, j + 2k, j + 3k, \ldots$$

      instead of

```
for all j in block
    p[j] = ( j % k ) != 0;
```

    * This gives about $(n/p)/k$ assignment statements
  - Effectively, block decomposition results in fewer computational steps *and* fewer communications steps

Developing the parallel algorithm

- Translate each step in sequential algorithm to its equivalent in parallel

  1. `Create a Boolean array from 1 to n`
     `Mark all values as true`
     - Each process can create and initialize its own share of the array
     - The size of the array is either $\lceil n/p \rceil$ or $\lfloor n/p \rfloor$
  2. `k = 2`
     - Each process does this as a trivial assignment

3. In the `while` loop, each process marks its share of the array
   (a) `Change all multiples of k between k^2 and n to false`
      – Each process marks the multiples of k within its block between k^2 and n
      – We need to determine the location of first multiple of k within the block
   (b) `Find smallest index p > k that contains true`
      – Always done by process 0
   (c) `Broadcast the value of k to all processes`

- Function `MPI_Bcast`
  - Broadcast a message from the process with rank `root` to all other processes of the communicator

```
int MPI_Bcast ( void * buffer, int count, MPI_Datatype datatype, int root,
    MPI_Comm comm );
```

    `buffer` Starting address of the array of data items to broadcast
    `count` Number of data items in the array
    `datatype` Data type of each item (uniform since it is an array); defined by an MPI constant
    `root` Rank of broadcast root – the process that initiates the broadcast
    `comm` Communicator; group of processes participating in this communication function

  - In parallel sieve, process 0 needs to broadcast a single integer k to all other processes

```
MPI_Bcast ( &k, 1, MPI_INT, 0, MPI_COMM_WORLD );
```

  - Processes can trivially determine the number of prime numbers found within their own arrays at the end of the `while` loop
    * The values can be accumulated into a grand total by using `MPI_Reduce`

Analysis of parallel sieve algorithm

- Time to mark each cell as multiple of prime is given by χ
  - Includes the time to
    * Change the value to `false`
    * Increment loop index
    * Testing for termination
- Sequential algorithm execution time: $\Theta(n \ln \ln n)$ or with known χ, $\chi n \ln \ln n$
- Cost of each broadcast: $\lambda \lceil \log p \rceil$
  - λ is message latency
  - Only a single value is broadcast per iteration
- Number of broadcasts: $\frac{\sqrt{n}}{\ln \sqrt{n}}$
  - Based on number of primes between 2 and n given by $\frac{n}{\ln \sqrt{n}}$
- Expected execution time

$$\frac{\chi(n \ln \ln n)}{p} + \frac{\sqrt{n}}{\ln \sqrt{n}} \lambda \lceil \log p \rceil$$

The code

Benchmarking

- Determine the value of χ by running a sequential implementation on a single processor of the cluster
- Determine λ by performing a series of broadcasts on 2, . . . , 10 processors
- Plug in the values and find performance gain

Improvements

- Delete even integers
  - Change sieve algorithm to represent only odd integers
    - Half the storage
    - Double the speed
- Eliminate broadcast
  - Broadcast step to give starting value of k is repeated $\frac{\sqrt{n}}{\ln\sqrt{n}}$ times
  - Replicate computation of primes up to $\sqrt{n}$
  - Useful if

$$\begin{array}{rccc} & \frac{\sqrt{n}}{\ln\sqrt{n}}\lambda\lceil\log p\rceil & > & \chi\sqrt{n}\ln\ln\sqrt{n} \\ \Rightarrow & \frac{\lambda\lceil\log p\rceil}{\ln\sqrt{n}} & > & \chi\ln\ln\sqrt{n} \\ \Rightarrow & \lambda & > & \frac{\chi\ln(\ln\sqrt{n}+\sqrt{n})}{\lceil\log p\rceil} \end{array}$$

  - Expected time complexity now is

$$\chi\left(\frac{n\ln\ln n}{2p}+\sqrt{n}\ln\ln\sqrt{n}\right)+\lambda\lceil\log p\rceil$$

- Reorganize loops
  - Each process marking widely dispersed elements of a very large array leads to poor cache hit rate
  - Improve cache hit rate by exchanging inner and outer loops
- Benchmarking