Parallel Algorithm Design

Departamento de Electrónica y Ciencias de la Computación

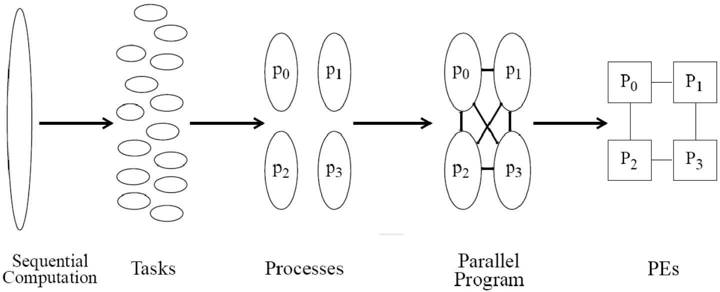
Facultad de Ingeniería y Ciencias

Pontificia Universidad Javeriana – Cali

Santiago de Cali, Mayo de 2022

Outline

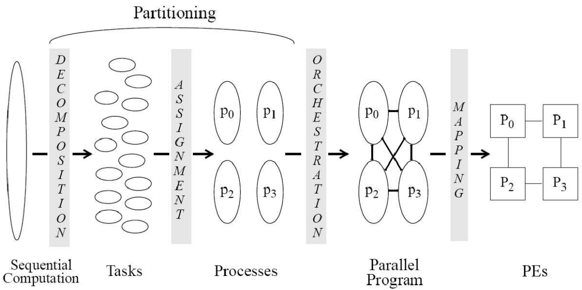
Some Definitions[1]



- A *(Primitive) Task* is an arbitrarily defined piece of the work done by the program. It is the smallest unit of concurrency that the parallel program can exploit.
- A *Process* is an abstract entity that performs tasks. A *Parallel Program* is composed of multiple cooperating processes, each of which performs a subset of the tasks in the program. Processes may need to communicate and synchronise with one another to perform their assigned tasks.
- The way processes perform their assigned tasks is by executing them on the physical *PEs* in the machine.

[1] [Culler+98]

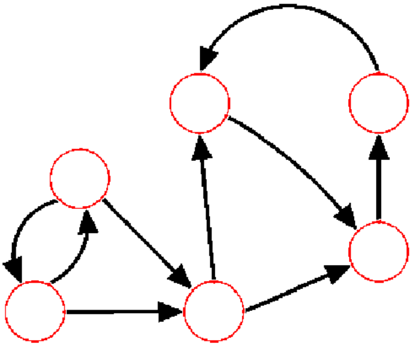
Steps in the Process[2]



1. *Decomposition* of the Sequential Computation into Tasks.
2. *Assignment* of Tasks to Processes.
3. *Orchestration* of the necessary data accesses, communication, and synchronisation among Processes.
4. *Mapping* or binding of Processes to PEs.

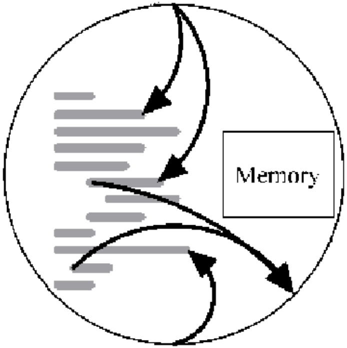
[2] [Culler+98]

Parallel Computation



Parallel Computation = set of *Processes*

Process

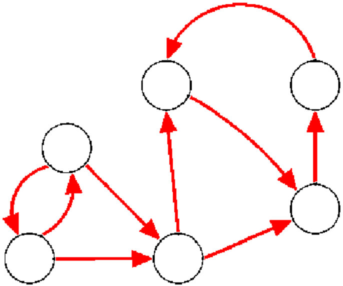


Process

- Program
- Local Memory
- Collection of I/O ports

Parallel Computation

Can be represented by a directed graph:



- Vertices represent Processes
- Directed Edges represent *Communication Channels*
- Processes interact by sending messages through Communication Channels

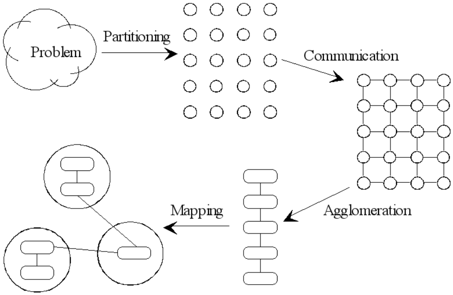
Communication Channel

- Message Queue that connects one process output port with another process input port
- Receiving is a *synchronous* (blocking) operation
- Sending is an *asynchronous* (non-blocking) operation

Execution Time

- The period of time during which any process is active
- The *starting time* is when the first process starts executing
- The *finishing time* is when the last process has stopped executing

Foster's Parallel Algorithm Design Methodology[3] - Overview



- Together, Decomposition and Assignment are called *Partitioning*, since they divide the work done by the program among the cooperating processes.
- Orchestration is decomposed into *Communication* and *Agglomeration*.

[3] [Quinn04]

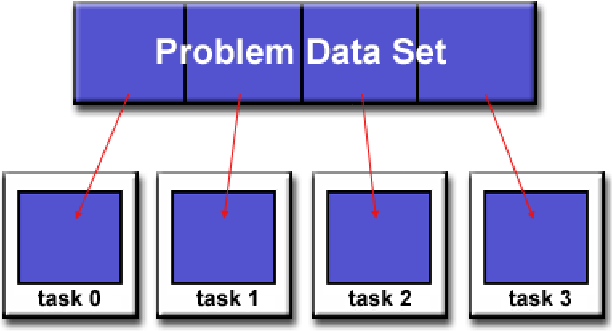
Outline

Foster's Parallel Algorithm Design Methodology (1)

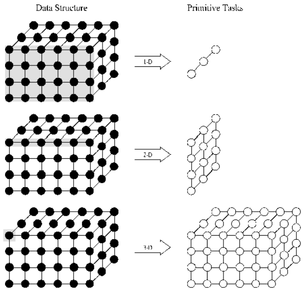
Since:
The number of primitive tasks is an upper bound on the maximum degree of parallelism one can exploit.
Goal:
Identify as many primitive tasks as possible.

1. Partitioning (dividing computation and data into pieces):
   - Domain decomposition (data-centric)
     - Divide data into pieces; e.g., an array into sub-arrays (reduction); a loop into sub-loops (matrix multiplication), a search space into sub-spaces (chess)
     - Determine how to associate computations with the data
   - Functional decomposition (computation-centric)
     - Divide computation into pieces; e.g., pipelines (floating point multiplication), workflows (payroll processing)
     - Determine how to associate data with the computations

Domain Decomposition

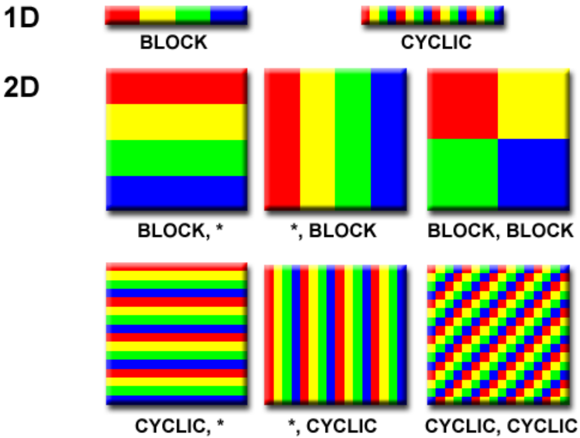


Domain Decomposition Example



- Typically the focus is to divide the largest and/or the most frequently accessed data structure in the program
- It is usually best to maximise the number of primitive tasks

Domain Decomposition Types



Domain Decomposition Types (2)

- Block
  - n is multiple of p: straightforward...
  - n is not multiple of p: Every process gets $\left\lfloor \frac{n}{p} \right\rfloor$; process 0 gets the remainder ($n \bmod p$)
- Cyclic (a.k.a. interleaved)
  - Process 0 is responsible for blocks $0, 0+p, 0+2p, \ldots$
  - Process 1 is responsible for blocks $1, 1+p, 1+2p, \ldots$
  - and so on...

Domain Decomposition Types (3)

Block (n is not multiple of p): Calculate $r = n \bmod p > 0$

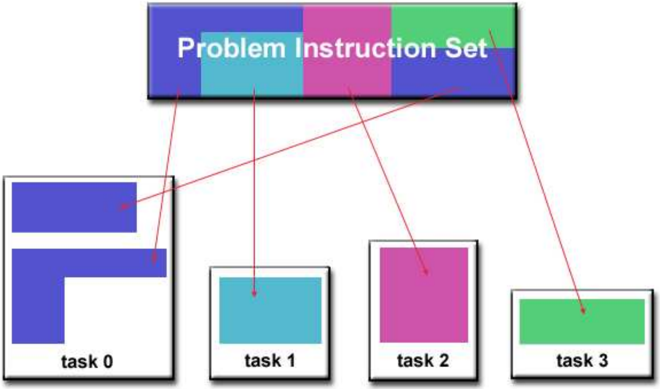
1. Concentrate all of the larger blocks among the smaller-numbered processes:
   - First r processes get a block of size $\left\lceil \frac{n}{p} \right\rceil$
   - The remaining $p - r$ processes get a block of size $\left\lfloor \frac{n}{p} \right\rfloor$
   - First element controlled by process p_id: $p\_id \left\lfloor \frac{n}{p} \right\rfloor + \min(p\_id, r)$
   - Last element controlled by process p_id: $(p\_id + 1) \left\lfloor \frac{n}{p} \right\rfloor + \min(p\_id + 1, r) - 1$

Domain Decomposition Types (4)

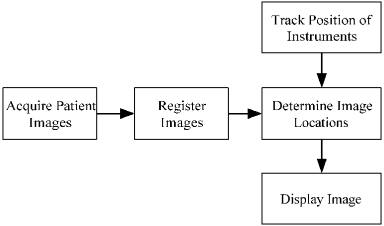
Block (n is not multiple of p): Calculate $r = n \bmod p > 0$

2. Distribute the larger blocks among the processes:
   - First element controlled by process p_id: $\left\lfloor p\_id \, \frac{n}{p} \right\rfloor$
   - Last element controlled by process p_id: $\left\lfloor (p\_id + 1) \, \frac{n}{p} \right\rfloor - 1$

Functional Decomposition

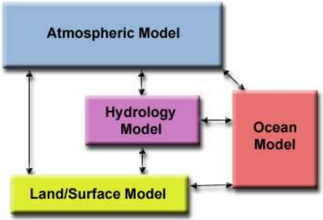


Functional Decomposition Example



- Task1: Track instrument positions for the next image
- Task2: Convert an image from physical coordinates to image coordinates
- Task3: Display the previous image

Functional Decomposition Example 2



- 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

Foster's Parallel Algorithm Design Methodology (2)

Since:
The number of primitive tasks is an upper bound on the maximum degree of parallelism one can exploit.
Goal:
Identify as many primitive tasks as possible.
Step 1:
Partitioning (dividing computation and data into pieces) Checklist:

- Are redundant computations/data storage minimised?
- Are primitive tasks roughly the same size?
- Is the number of tasks an increasing function of the problem size?

Outline

Foster's Parallel Algorithm Design Methodology (3)

Since:
Communication between tasks is part of the overhead of a parallel algorithm.
Goal:
Minimise parallel overhead.

2. Communication (establish communication pattern):
   - Local communication
     - Task needs values from a small number of other tasks
     - Create channels illustrating data flow
   - Global communication
     - Significant number of tasks contribute data to perform a computation
     - Do not create channels for them early in design

Foster's Parallel Algorithm Design Methodology (4)

Since:
Communication between tasks is part of the overhead of a parallel algorithm.
Goal:
Minimise parallel overhead.
Step 2:
Communication (establish communication pattern)
Checklist:

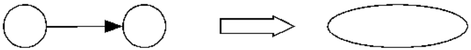
- Are communication operations balanced among tasks?
- Does each task communicates with only small group of neighbours?
- Can tasks perform communications concurrently?
- Can tasks perform computations concurrently?

Outline

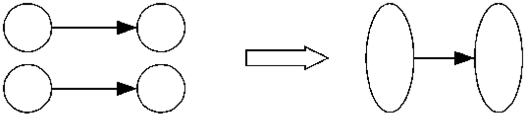
Foster's Parallel Algorithm Design Methodology (5)

3. Agglomeration (reducing the amount of parallel overhead):
    - Grouping tasks into larger tasks
    - Goals
        - Lower communication overhead (increase locality)
        - Maintain scalability (for porting purposes)
        - Reduce software engineering costs (by using existing sequential code)
    - In MPI programming, goal often to create one agglomerated task (process) per PE

Agglomeration Examples



Combining tasks into processes eliminates communication and increases locality



Combining groups of sending and receiving tasks into processes reduces the number of messages being sent (fewer, longer messages)

Foster's Parallel Algorithm Design Methodology (6)

Step 3:
Agglomeration (reducing the amount of parallel overhead)
Checklist:

- Does the locality increased?
- Do replicated computations take less time than the communication they replaced?
- Is the amount of replicated data small enough to allow the algorithm to scale?
- Do agglomerated tasks have similar computational and communications costs?

Foster's Parallel Algorithm Design Methodology (7)

Step 3:
Agglomeration (reducing the amount of parallel overhead)
Checklist (cont.):

- Is the number of tasks an increasing function of the problem size?
- Is the number of tasks as small as possible, yet at least as great as the number of PEs in the likely target computers?
- Is the trade-off between the chosen agglomeration and the cost of modifications to existing sequential code reasonable?

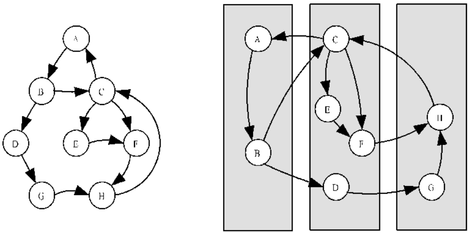
Outline

Foster's Parallel Algorithm Design Methodology (8)

4. Mapping (assigning tasks/processes to PEs):
    - Goals
        - Maximise PE utilisation
        - Minimise interPE communication
    - Centralised multiprocessor: mapping done by operating system
    - Distributed memory system: mapping done by user

   Finding an optimal solution is NP-hard...

Mapping Example



- What if every PE has the same speed and that every task/process requires the same amount of execution time?
- What if every channel communicates the same amount of data?

Mapping Decision Tree

- Static number of processes
  - Structured communication
    - Constant computation time per task
      Agglomerate tasks into processes to minimise communication
      Create one process per PE
    - Variable computation time per task
      Cyclically map tasks/processes to PEs
  - Unstructured communication
    - Use a static load-balancing algorithm
- Dynamic number of tasks/processes

Mapping Decision Tree

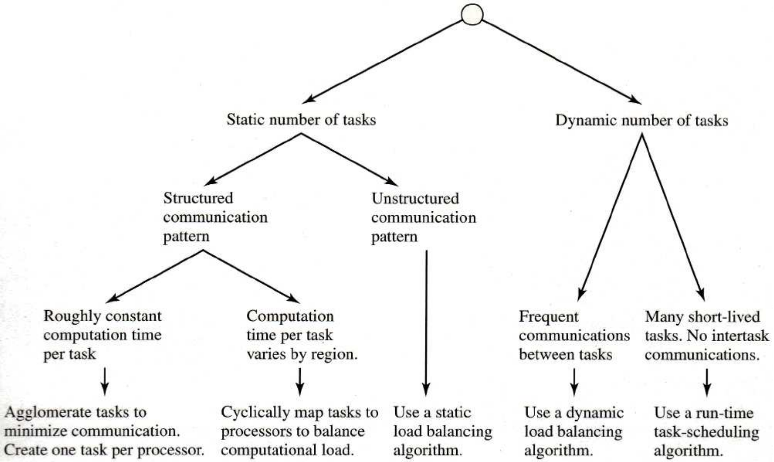
- Static number of processes
- Dynamic number of tasks/processes
  - Use a run-time task-scheduling algorithm
    - centralised:
      e.g., a master-slave strategy
    - distributed:
      push strategy: PEs with too many processes send some of them to neighbouring PEs
      pull strategy: PEs with no work to do ask neighbouring PEs for work
  - Use a dynamic load-balancing algorithm
    e.g., share load among neighbouring PEs; remapping periodically

Foster's Parallel Algorithm Design Methodology (9)

Step 4:
Mapping (assigning processes to PEs)
Checklist:

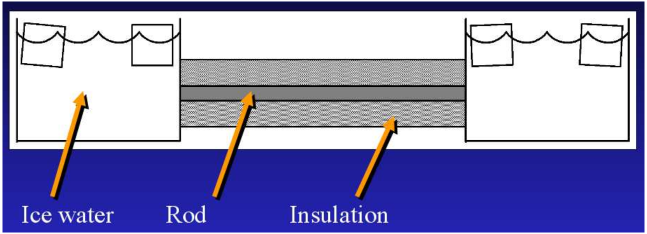
- Considered designs based on one process per PE and multiple processes per PE
  - If multiple processes per PE chosen, ratio of processes to PEs is at least 10:1
- Evaluated static and dynamic process allocation
- If dynamic process allocation chosen, process allocator is not a bottleneck to performance

Mapping Decision Tree



Outline

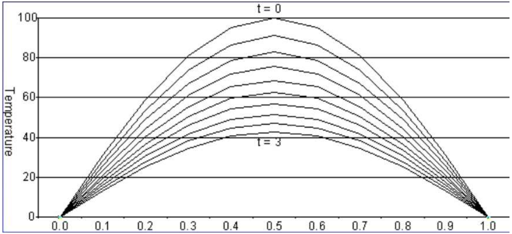
Introduction



- Rod length = 1
- Ice bath temperature = 0 ℃
- $temp_0(x) = 100 \sin(\pi x)$

Rod Cools as Time Progresses

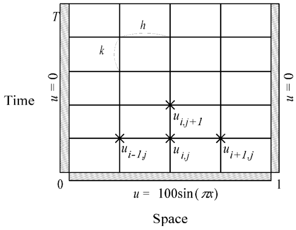
- A partial differential equation models the temperature at any point of the rod at any point in time



A finite difference approximation to the rod-cooling problem

- Each curve represents the temperature distribution of the rod at some point in time

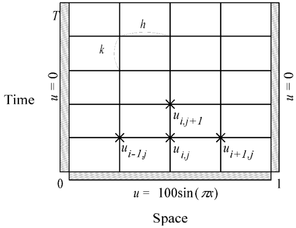
Finite Difference Approximation – Data Structure (1)



Every point $u_{i,j}$ represents a matrix element containing the temperature at position i on the rod at time j

- The rod is divided into n sections of length h, so each row has $n + 1$ elements. Increasing n reduces the error in the approximation
- Time from 0 to T is divided into m discrete entities of length k, so the matrix contains $m + 1$ rows

Finite Difference Approximation – Data Structure (2)



- The finite difference algorithm steps forward in time, using values from time j to compute the value for time $j+1$ using the formula

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

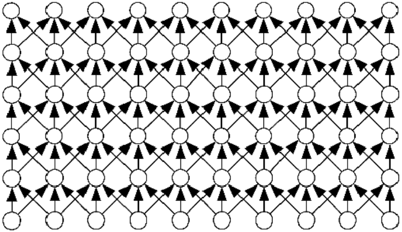
where $r = k/h^2$

Partitioning

- One data item per grid point
- Associate one primitive task with each grid point
- Two-dimensional domain decomposition

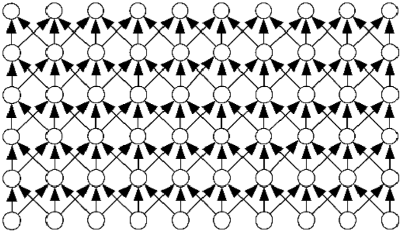
Communication

Identify communication pattern between primitive tasks:



- Each primitive task computing $u_{i,j+1}$ requires the values of $u_{i-1,j}$, $u_{i,j}$, and $u_{i+1,j}$
- Each interior primitive task has three incoming and three outgoing channels
- Primitive tasks on the edges have fewer channels

Agglomeration



Tasks computing rod temperatures later in time depend upon the results produced by tasks computing rod temperatures earlier in time

- Agglomerate all the tasks associated with each point in the rod into a process

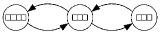
A linear array of processes

Mapping

A linear array of processes

In a real problem the number of rod segments would be large

- Agglomerate processes so that computational workloads are balanced and communication is minimised



A process is responsible for computing, over all time steps, the temperatures for a contiguous group of rod locations

Sequential Execution Time

- $t_{comp} = \tau$ – time to compute $u_{i,j+1}$
- n – number of pieces of size h
- m – number of time steps
- $t_s = m(n-1)t_{comp}$

Parallel Execution Time

- p – number of PEs
- γ – time to communicate a value to another task
- $t_p = t_{comp} + t_{comm}$
    - $t_{comp} = \left\lceil \frac{n-1}{p} \right\rceil \tau$ – computation time for each iteration
    - $t_{comm} = 2\gamma + 2\gamma$ – time to send/receive to/from left and right neighbours for each iteration
- $t_p = m \times \left(\left\lceil \frac{n-1}{p} \right\rceil \tau + 4\gamma \right)$

Outline

Introduction (1)

Boundary value problems arising from the real world are too complicated to solve analitically

- x – computed solution
- c – correct solution
- $error = \left| \frac{x - c}{c} \right|$

Computed	0.15	0.16	0.16	0.19
Correct	0.15	0.16	0.17	0.18
Error (%)	0.00%	0.00%	6.25%	5.26%

Maximum error = 6.25 %

Introduction (2)

Given

- a set of values $a_0, a_1, a_2, \ldots, a_{n-1}$
- associative operator $\oplus$

reduction is the process of computing

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

Examples:

- Add
- Multiply
- And, Or
- Maximum, Minimum

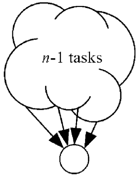
Partitioning

Since reduction requires exactly $n - 1$ operations, it has $\Theta(n)$ time complexity on a sequential computer. How quickly can we perform a reduction on a parallel computer?

- Divide the list into n pieces
- Associate one primitive task with each piece

Communication (1)

Brute force approach:

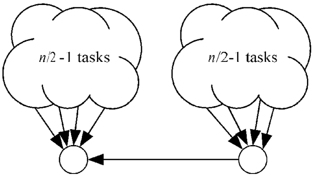


One task (the root task) receives a list element from each of the other $n-1$ tasks and performs all the additions

- γ – time to communicate a value to another task
- τ – time to perform an addition
- $t_p = (n-1)(\gamma + \tau)$ – slower than the sequential algorithm...

Communication (2)

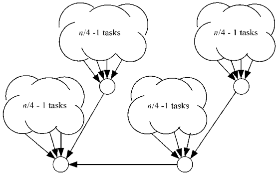
What if two tasks cooperate to perform the reduction?



- In time $\left(\frac{n}{2} - 1\right)(\gamma + \tau)$ each semiroot task has a subtotal for its half of the elements
- In one additional communication/computation step a single task has the grand total
- $t_p = \frac{n}{2}(\gamma + \tau)$

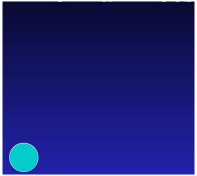
Communication (3)

Why not continue the process?



- Each task is responsible for $n/4$ of the list elements
- After the four subtotals have been computed, two remaining communication/computation steps yield the grand total

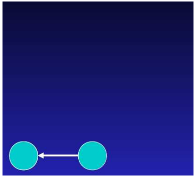
Communication – Binomial Trees



$k = 0$

- In a tree with $p = 2^k$ nodes, the maximum distance from any node to the root in the lower left corner is $k = \log_2 p$
- The binomial tree (a subgraph of a hypercube) is one of the most common communication patterns in parallel algorithm design

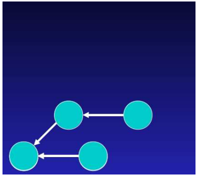
Communication – Binomial Trees



$k = 1$

- In a tree with $p = 2^k$ nodes, the maximum distance from any node to the root in the lower left corner is $k = \log_2 p$
- The binomial tree (a subgraph of a hypercube) is one of the most common communication patterns in parallel algorithm design

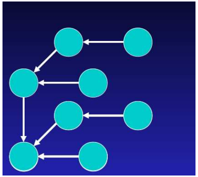
Communication – Binomial Trees



$k = 2$

- In a tree with $p = 2^k$ nodes, the maximum distance from any node to the root in the lower left corner is $k = \log_2 p$
- The binomial tree (a subgraph of a hypercube) is one of the most common communication patterns in parallel algorithm design

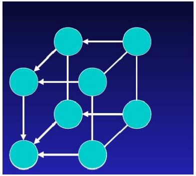
Communication – Binomial Trees



$k = 3$

- In a tree with $p = 2^k$ nodes, the maximum distance from any node to the root in the lower left corner is $k = \log_2 p$
- The binomial tree (a subgraph of a hypercube) is one of the most common communication patterns in parallel algorithm design

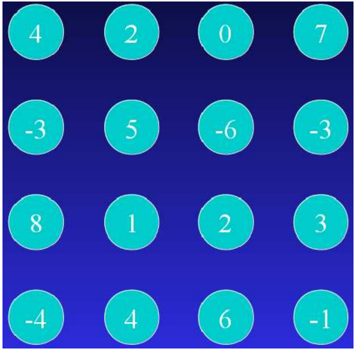
Communication – Binomial Trees



$k = 3$

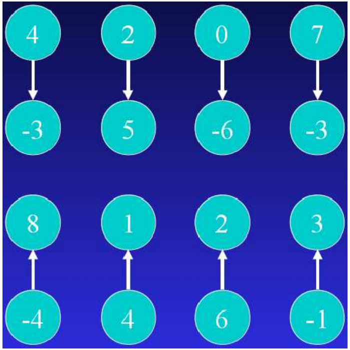
- In a tree with $p = 2^k$ nodes, the maximum distance from any node to the root in the lower left corner is $k = \log_2 p$
- The binomial tree (a subgraph of a hypercube) is one of the most common communication patterns in parallel algorithm design

Communication – Using Binomial Trees



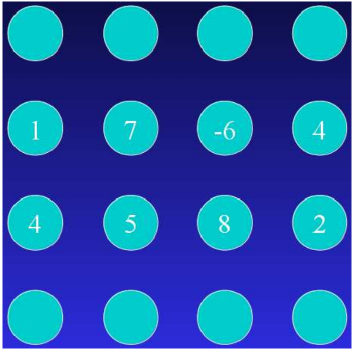
16 tasks cooperate to find the summation value

Communication – Using Binomial Trees



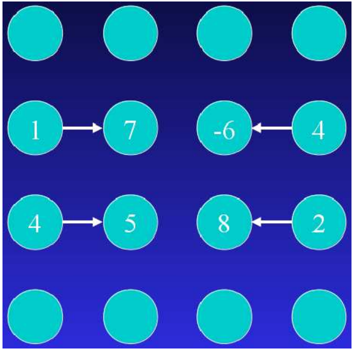
Half of the tasks send values, and half of the tasks receive values

Communication – Using Binomial Trees



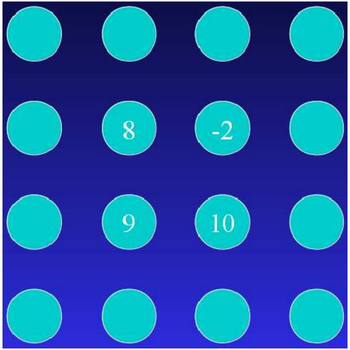
Half of the tasks add values, and half of the tasks become inactive

Communication – Using Binomial Trees



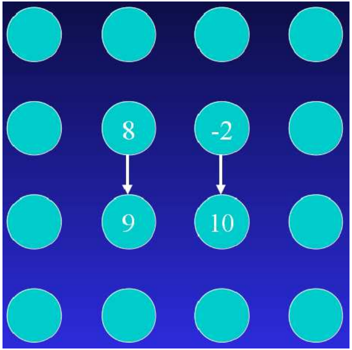
A quarter of the tasks send values, and a quarter of the tasks receive values

Communication – Using Binomial Trees



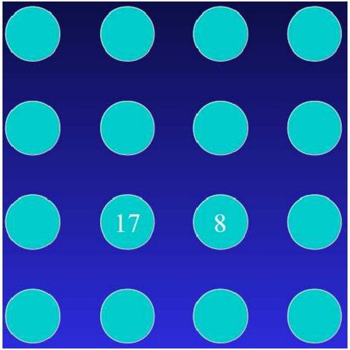
A quarter of the tasks add values, and three quarters of the tasks are inactive

Communication – Using Binomial Trees



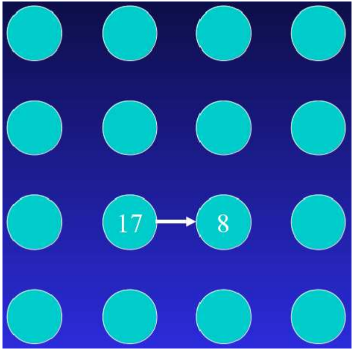
An eighth of the tasks send values, and an eighth of the tasks receive values

Communication – Using Binomial Trees



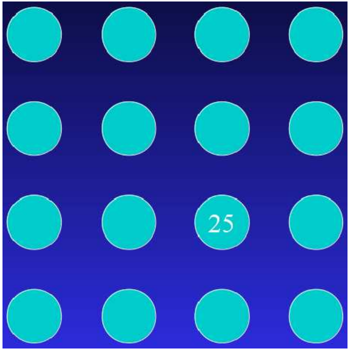
An eighth of the tasks add values, and seven eighths of the tasks are inactive

Communication – Using Binomial Trees



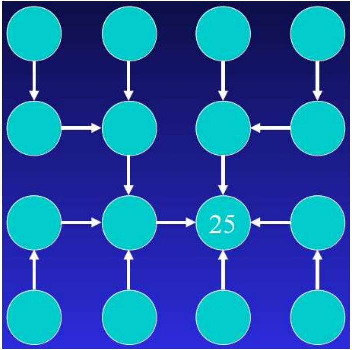
One task send its value, and one task receive a value

Communication – Using Binomial Trees



One task add two values, and the remaining 15 tasks are inactive

Communication – Using Binomial Trees

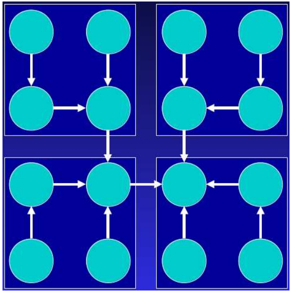


The task/channel graph forms a binomial tree

Agglomeration and Mapping

The number of tasks is static, computations per task are trivial, and the communication pattern is regular (agglomerate tasks to minimise communication by following the previous graph)

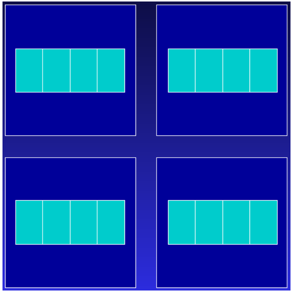
- 16 tasks are mapped to 4 PEs



Agglomeration and Mapping

The number of tasks is static, computations per task are trivial, and the communication pattern is regular (agglomerate tasks to minimise communication by following the previous graph)

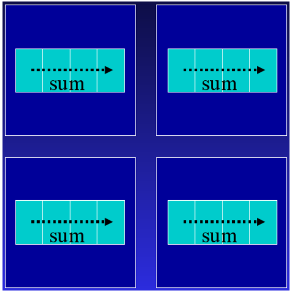
- 4 tasks on each PE are agglomerated into a single process



Agglomeration and Mapping

The number of tasks is static, computations per task are trivial, and the communication pattern is regular (agglomerate tasks to minimise communication by following the previous graph)

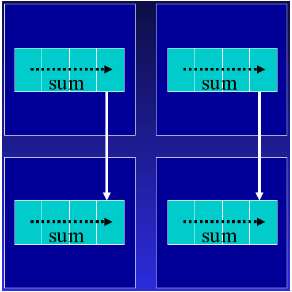
- Each PE adds n/p values



Agglomeration and Mapping

The number of tasks is static, computations per task are trivial, and the communication pattern is regular (agglomerate tasks to minimise communication by following the previous graph)

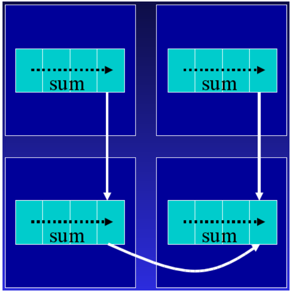
- Two processes send values, and two processes receive values and add



Agglomeration and Mapping

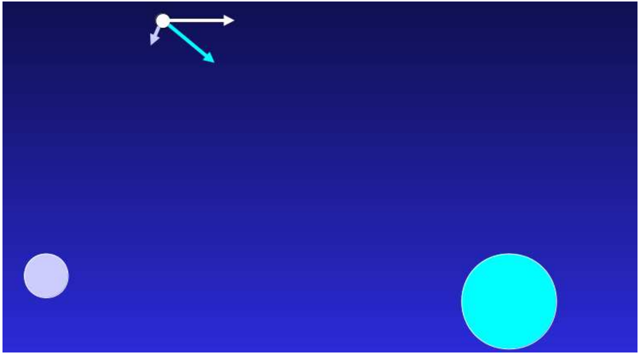
The number of tasks is static, computations per task are trivial, and the communication pattern is regular (agglomerate tasks to minimise communication by following the previous graph)

- One process send its value, and one process receives a value and add. The latter has the grand total



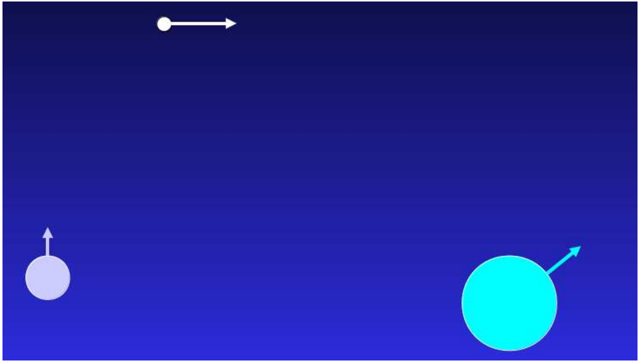
Outline

Introduction (1)

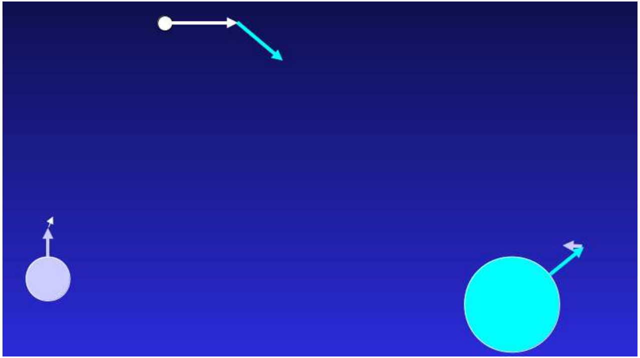


- Every particle exerts a gravitational pull on every other particle
- The white particle has a particular position and velocity vector (indicated by the white arrow)
- Its future position is influenced by the gravitational forces exerted by the other two particles

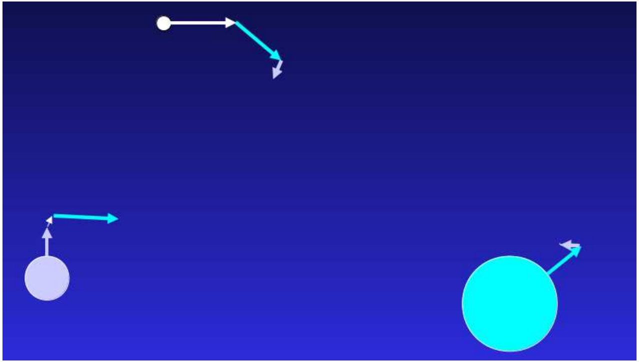
Introduction (2)



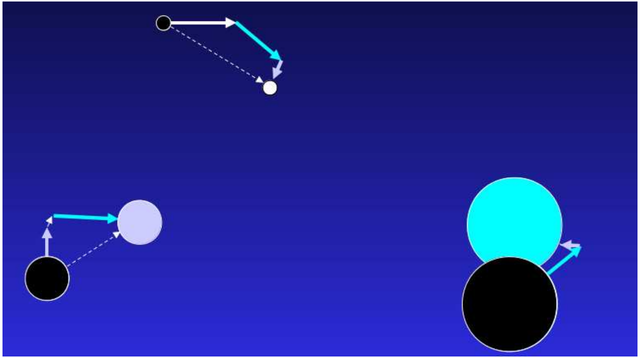
Introduction (2)



Introduction (2)



Introduction (3)

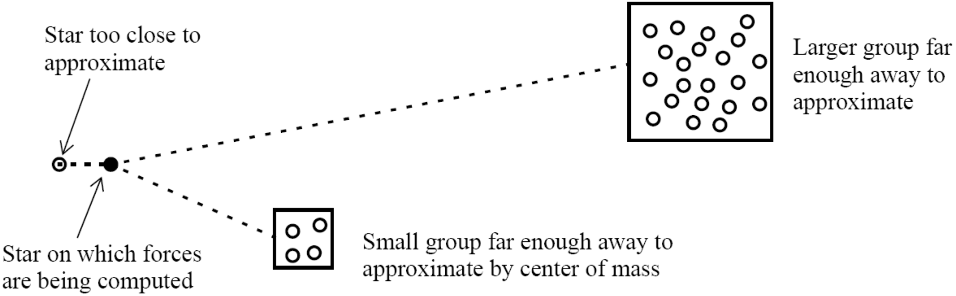


Solved by performing computations on all pairs of bodies

- gravitational interaction = $G\frac{m_1 m_2}{d^2}$
- typically, time complexity = $\Theta(n^2)$, where n is the number of bodies

Introduction (4)

A group of bodies that is far enough away from a given body may be approximated by the centre of mass of the group. The farther apart the bodies, the larger the group that may be thus approximated:

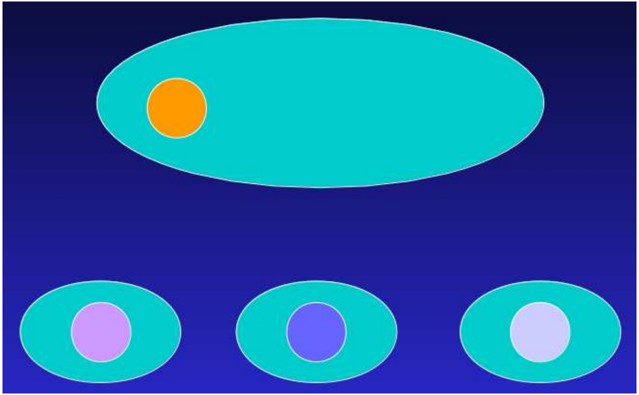


Partitioning

- Domain partitioning
- Assume one task per particle
- Task has particle's position, velocity vector
- Iteration
  - Get positions of all other particles
  - Compute new position, velocity

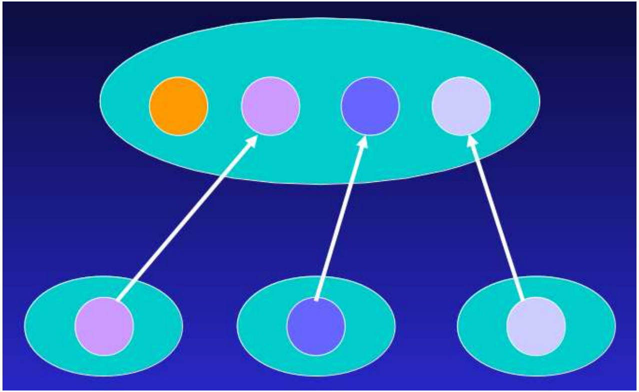
Communication (1)

A gather operation is a global communication that takes a dataset distributed among a group of processes and collects the items on a single process:



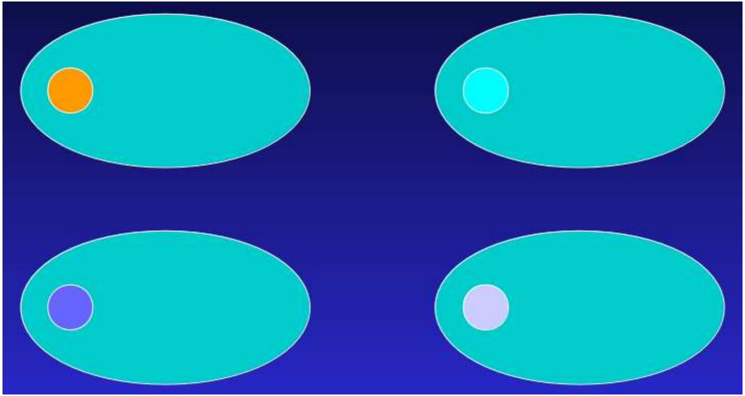
Communication (1)

A gather operation is a global communication that takes a dataset distributed among a group of processes and collects the items on a single process:



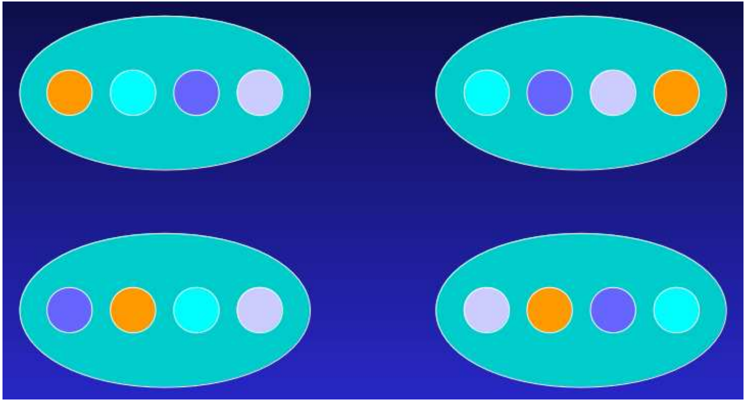
Communication (2)

An all-gather operation is similar to gather, except at the end of the communication every process has a copy of the entire dataset:



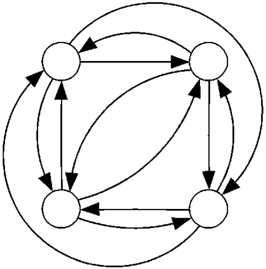
Communication (2)

An all-gather operation is similar to gather, except at the end of the communication every process has a copy of the entire dataset:



Communication (3)

Since it is necessary to update the location of every particle, an all-gather operation is called for:

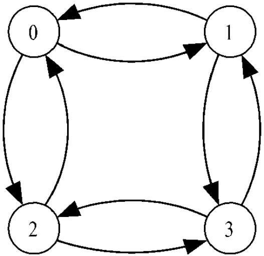


Set up a channel between every pair of tasks:

- $p - 1$ communication steps
- message length: 2 coordinates

Communication (3)

Since it is necessary to update the location of every particle, an all-gather operation is called for:



Use a bottom-up approach:

- $\log_2 p$ communication steps
- message length: $2 \times 2^{i-1}$, where i is the exchange step

Agglomeration and Mapping

In general, there are far more particles n than PEs p.

- Assume n is an multiple of p.
- Associate one process per PE and agglomerate n/p particles into each process.
- The all-gather communication operation requires $\log_2 p$ communication steps.
- In the first step the messages have length n/p, in the second step the messages have length $2n/p$, etc.

Communication Time

- Setting up a channel between every pair of processes:

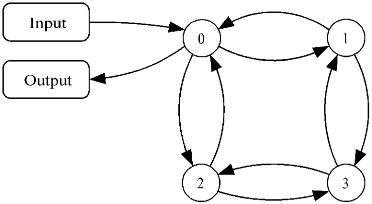
$$\begin{aligned} t_{comm} &= (p-1) \times \left(\lambda + \frac{n/p}{\beta}\right) \\ &= \lambda(p-1) + \frac{n(p-1)}{\beta p} \end{aligned}$$

- Using all-gather (for each iteration):

$$\begin{aligned} t_{comm} &= \sum_{i=1}^{\log_2 p} \left(\lambda + \frac{2^{i-1} n}{\beta p}\right) \\ &= \lambda \log_2 p + \frac{n(p-1)}{\beta p} \end{aligned}$$

Introduction

- Assume a single process (0) is responsible for performing file I/O operations
- By adding new channels for file I/O the resulting process/channel graph is:

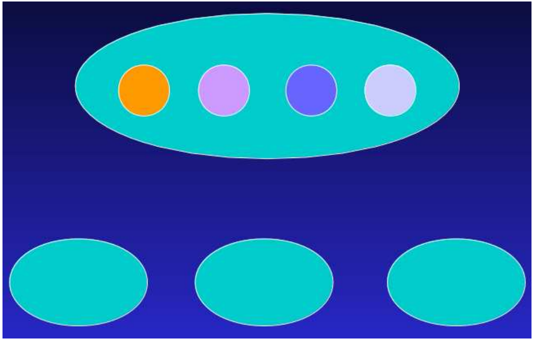


- A pair of coordinates identify a particle's location
- Velocity can be represented by another pair of values
- Reading the position and velocities of *n* particles requires time

$$t_{read} = \lambda_{IO} + \frac{4n}{\beta_{IO}}$$

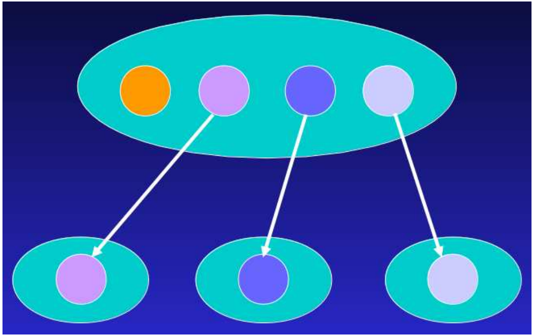
Communication (1)

After the I/O process inputs the particles, it is necessary to break up the input data into pieces so that each process has its assigned subsection containing n/p elements. This global operation is called scatter:



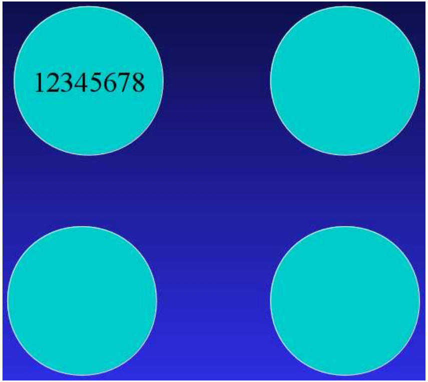
Communication (1)

After the I/O process inputs the particles, it is necessary to break up the input data into pieces so that each process has its assigned subsection containing n/p elements. This global operation is called scatter:



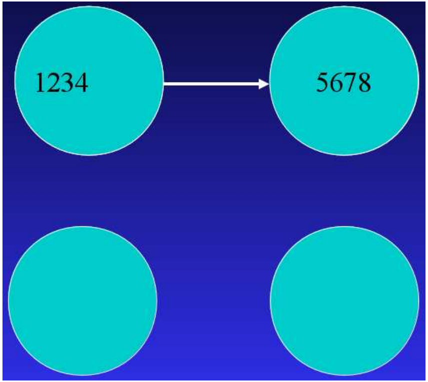
Communication (2)

One way to scatter the particles is for the I/O process to simply send the corresponding n/p particles to each of the other tasks. Another is to derive a scatter operation requiring $\log_2 p$ steps:



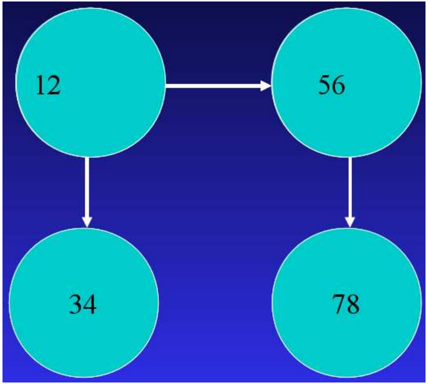
Communication (2)

One way to scatter the particles is for the I/O process to simply send the corresponding n/p particles to each of the other tasks. Another is to derive a scatter operation requiring $\log_2 p$ steps:



Communication (2)

One way to scatter the particles is for the I/O process to simply send the corresponding n/p particles to each of the other tasks. Another is to derive a scatter operation requiring $\log_2 p$ steps:



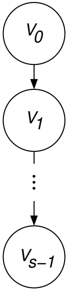
Scattering Time

- The I/O process sends the corresponding n/p particles to each of the other tasks:

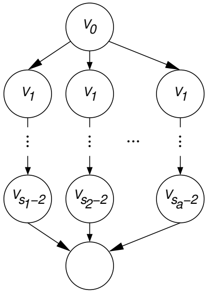
$$\begin{aligned} t_{scattering} &= (p-1) \times \left(\lambda + \frac{4n}{\beta p}\right) \\ &= \lambda(p-1) + \frac{4n}{\beta p} \times (p-1) \end{aligned}$$

- Using a scatter operation requiring $\log_2 p$ steps:

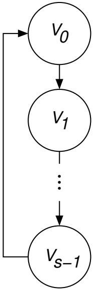
$$\begin{aligned} t_{scattering} &= \sum_{i=1}^{\log_2 p} \left(\lambda + \frac{4n}{2^i \beta p}\right) \\ &= \lambda \log_2 p + \frac{4n}{\beta p} \times \left(1 - \frac{1}{p}\right) \end{aligned}$$



$$\sum_{i=0}^{s-1} WCET(V_i)$$



$$\max\{WCET(path_j)\}$$
$$\forall\, j = 1, \ldots, a$$



$$I \times \sum_{i=0}^{s-1} WCET(V_i)$$

- Cormen T. H., Leiserson C. E., Rivest R. L., Stein C. Introduction to Algorithms. Second Edition. MIT Press, 2002 (Appendix A – Summations)
- Culler D., Singh J. P., Gupta, A. Parallel Computer Architecture – A Hardware-Software Approach. Morgan Kaufmann, 1998
- Quinn M. J. Parallel Programming in C with MPI and OpenMP. McGraw Hill, 2004