Parallel and Distributed Algorithms

Paweł Czarnul

Dept. of Computer Architecture
Faculty of Electronics, Telecommunications and Informatics
Gdansk University of Technology

March 9, 2015

Plan of the Lecture

Focus on useful parallel and distributed algorithms applicable to various fields of study

- Passing criteria, grade criteria
  - minimum 50% out of lab
  - minimum 50% out of written exam
  - the exam and the lab each contribute 50% to the final grade
- outlinining algorithms covered in this part of the lecture
- understand and design of parallel algorithms – abstract from particular implementation technologies (MPI, CUDA, OpenCL etc.)

Algorithms covered in this part of the lecture

- bubble/mergesort/quicksort
- numerical algorithm(s)
- minimax/alphabeta search
- solving a system of linear equations using
  1. Gaussian elimination
  2. Jacobi (relation to differential equations)
- genetic algorithms
- interpolation algorithms – such as IDW (Inverse Distance Weighted Interpolation)

Fundamentals

Analysis of algorithms should consider:

- complexity of fundamental blocks vs input data size
    1. computational blocks
    2. communication blocks (point-to-point, collective)
    3. synchronization (usually global)
- coefficients which are very important in practice
    - for instance, GB Ethernet and Infiniband offer the same complexity vs input data size O(d) but various coefficients

Exemplary blocks and complexities

- computational ones (sequential):
    - sorting based on exchange – mergesort O(nlogn), quicksort (average) O(nlogn), bubblesort O(n^2)
    - simple image filters O(n)
- communication blocks:
    - point-to-point $t_{\text{startup time}} + \frac{d}{bandwidth}$
    - collective e.g. scatter – assuming pairs of processes can communicate at the same time O(dnlogn)
    - barrier – assuming pairs of processes can communicate at the same time O(nlogn)

Parallelization of algorithms

Parallelization of sequential algorithms involves identification and understanding of the following:

- key blocks in the algorithm – computations
- dependencies between the blocks
- necessary synchronization

In case of parallelization – decide about the following – what partitioning method is adopted – decide about the paradigm:

- master-slave
- SPMD (geometric)
- pipeline
- divide-and-conquer

Amdahl's law

- t(1) – the execution time of application A running on a parallel machine with one processor
- t(N) – the execution time of application A running on the same parallel machine with N processors.
- then Speed-up S can be defined as S=t(1)/t(N)
- Let us assume that:
    - s_i denotes the i-th part of the program's instructions to be executed sequentially
    - r_j denotes the j-th part of the program's instructions to be executed in parallel
    - Let us define: $s = s_0 + s_1 + \ldots + s_k$ and $r = r_0 + r_1 + \ldots + r_l$.
- If s+r=1 then S=(s+r)/(s+r/N)=1/(s+(1-s)/N).
- If $N \to \infty$ then $S(N) \to 1/s$ which limits the speed-up.

Speed-ups

Speed-ups for various values of s – see how it limits the speed-up

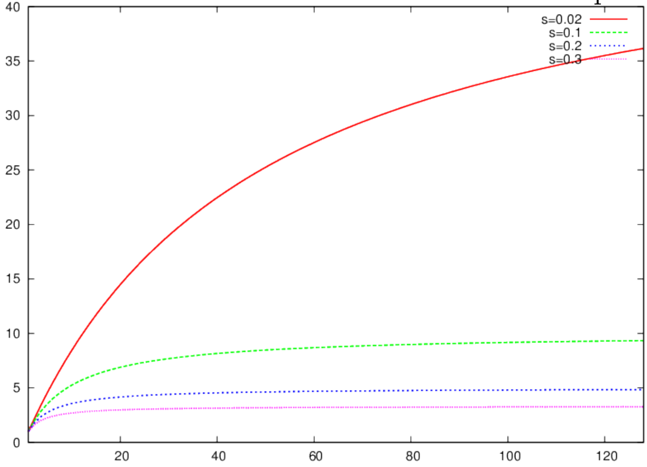


Figure: Speed-ups for various value of s

Speed-ups

Speed-ups for various values of s – including ideal

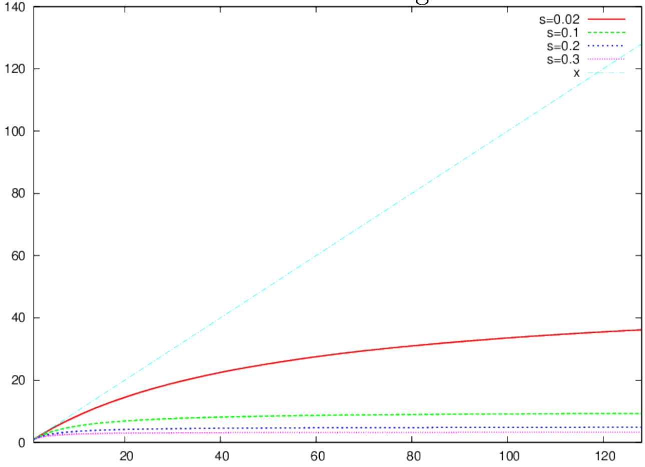
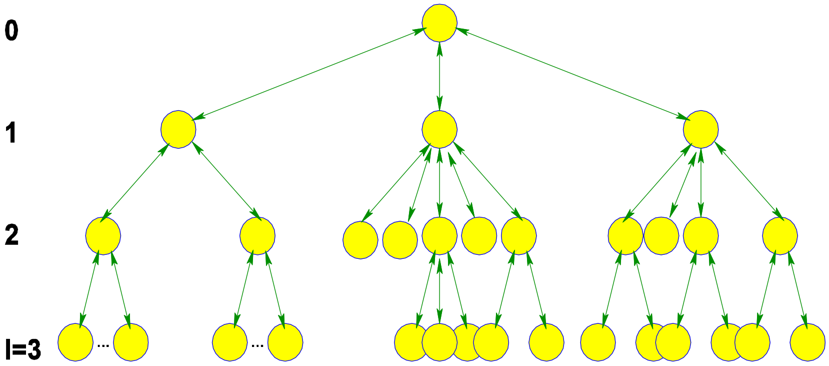


Figure: Speed-ups for various value of s

Other performance metrics

- Parallel efficiency (PE) can be defined as $PE = \frac{S}{N}$ where N is the number of CPUs.
- Cost of computations can be defined as $C = \text{execution time} \times N$
- Scaled speed-up is used in the context of solving a larger problem on a larger system i.e. instead of considering the problem of size d on N CPUs it is considered that the parallel part r can be scaled up and the scaled speed-up (SS) becomes: $SS = \frac{s+rN}{s+r}$

Structure of a divide-and-conquer algorithm



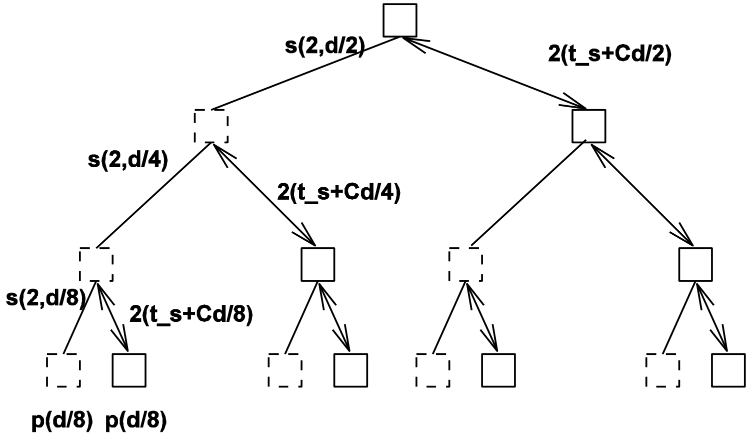
Typical algorithm analysis – based on divide-and-conquer

Costs that should be considered in the algorithm:

- p(d) – the execution time of algorithm A applied to data of size d
- c(d) – the memory read/write time of data of size d or communication time of sending data of size d between processes in a divide-and-conquer tree
- s(k,d)- the time spent on activities by a non-leaf process before/after it sends/receives data to/from its k child processes e.g. partitioning/merging k parts of size d/k
- overheads: process (thread) creation (if any), communication startup times etc.

Divide-and-conquer – complexity

Assume the following algorithm processing structure:



Divide-and-conquer – complexity

- Let us assume a binary tree allowing pairs of processes to communicate in parallel
- a vector is distributed down the tree towards the leaves that perform computations:
- communication can be expressed as $2\sum_{i=1}^{log_2(k)}(t_{startup}+C\frac{d}{2^i}) = 2(log_2(k)t_{startup}+Cd(1-\frac{1}{k}))$
- processes merge data in pairs which gives $\sum_{i=1}^{log_2(k)} s(2,\frac{d}{2^i})$
- finally the total execution time of a parallel algorithm will be: $p(\frac{d}{k})+2(log_2(k)t_{startup}+Cd(1-\frac{1}{k}))+\sum_{i=1}^{log_2(k)} s(2,\frac{d}{2^i})$

Problems and optimization

Notice the following:

- increasing the number of processes/threads (k in the previous slide) one can decrease the execution time but at the same time communication costs are increased – in particular the startup time
- the overall algorithm execution time $t_{total}(k)$ is a function that will have a local minimum
- it may not be beneficial to engage too many CPUs/cores for solving the problem
- $t^{'}_{total}(k) = 0$ will allow to find the configuration that minimizes the overall execution time

Sample embarrassingly parallel algorithms

1. image operations such as rotation, scaling, level adjustment
2. standard numerical integration in which the initial range is divided into a large number of subranges for which area is computed as an area of a rectangle – execution time is the same for every subrange
3. Monte Carlo methods
4. computing fractals such as the Mandelbrot set – each pixel can be computed independently

Usually embarrassingly parallel algorithms are straightforward and can be optimized or replaced by more refined algorithms. However, the former can be parallelized very easily

Sample embarrassingly parallel algorithm – image processing

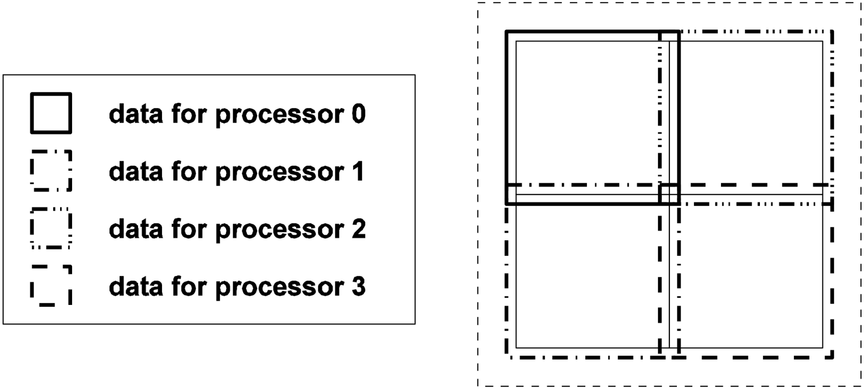


Figure: Parallel image processing

Sample embarrassingly parallel algorithm – image processing

There is a 2D $d^{0.5}$ x $d^{0.5}$ picture which needs to be adjusted using a certain filter so that each pixel requires data from neighboring ones. Suppose there are k processes. Each needs to process d/k data and requires $d/k + 4(d/k)^{0.5}$ pixels (neighboring pixels, assuming k is a power of 2) and will send d/k pixels back. The computational time is equal to Pd/k.

The total execution time of a parallel version can be (assuming communication is sent using point-to-point operations without overlapping):

$t_{total} = (k-1)(2t_{startup} + 2Cd/k + 4(d/k)^{0.5}) + Pd/k$

Sample divide-and-conquer algorithms

1. sorting such as:
    - mergesort
    - quicksort
2. certain numerical algorithms such as adaptive quadrature integration
3. search – such as minimax, $\alpha\beta search$

These are much more difficult to parallelize, especially if the problem is irregular – which ones out of these are irregular?

Parallelization of a divide-and-conquer algorithm – adaptive numerical integration

1. what is wrong with the naive approach which is embarrassingly parallel and is easy to parallelize?
    - subrange width the same in every part of the range – in some cases it should be smaller, in the others it could be larger without a sacrifice in accuracy
2. how to design an algorithm that would adapt to the function automatically?
3. how to parallelize it efficiently?

Parallelization of a divide-and-conquer algorithm – adaptive numerical integration

A divide-and-conquer approach! Idea:

1. check if a subrange can be computed in a simple way with a desired accuracy
2. if yes then do it
3. otherwise divide and apply the method recursively

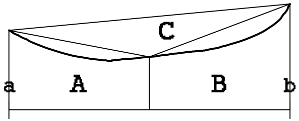
An alternative but a similar approach:

1. compute the area using a simple method with a desired accuracy – result A_1
2. compute the area using a simple method with a greater accuracy – result A_2
3. if $A_1 - A_2 \geq$ threshold then use A_2
4. otherwise divide and apply the method recursively

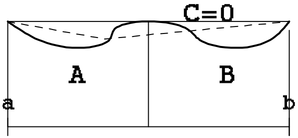
Parallelization of a divide-and-conquer algorithm – adaptive numerical integration

How to parallelize this application?

- a divide-and-conquer tree is generated
- it can be highly irregular
- how many pivot points should be checked? Why?



a.



b.

Parallelization of a divide-and-conquer algorithm – adaptive numerical integration

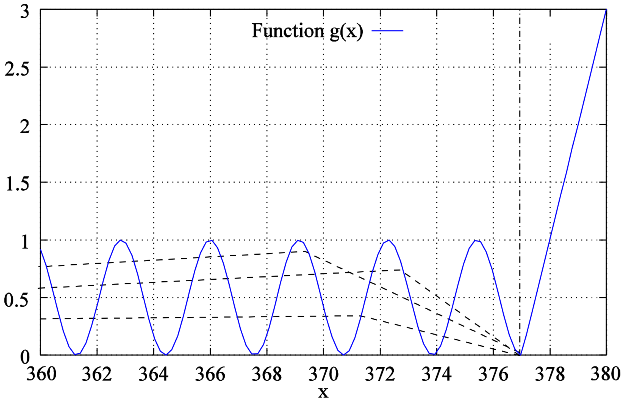
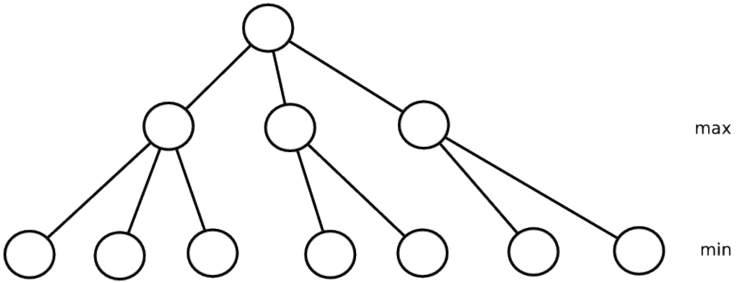


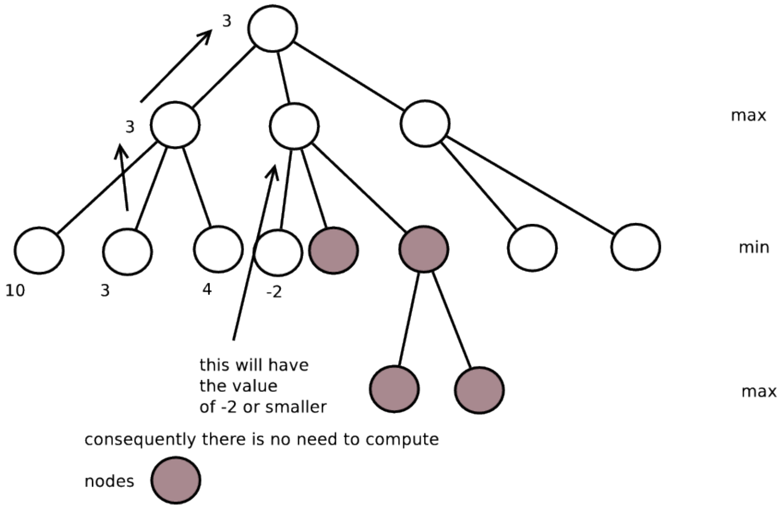
Figure: Sample function

Parallelization of a divide-and-conquer algorithm – minimax

- minimax is a search method that can be used in games such as chess, checkers etc.
- it unfolds a tree that represents successive moves taken by the players
- the tree is balanced
- the tree is very large and results in very long computations



Parallelization of a divide-and-conquer algorithm – $\alpha\beta$ search



Parallelization of a divide-and-conquer algorithm – $\alpha\beta$ search

- smarter than minimax – can cut off subtrees
- much more difficult to parallelize
- what parallelization strategy could be adopted?
- what about the following:
  1. how to rate moves?
  2. is the order of analysis important?
  3. what about transposition tables?
  4. are sequential and parallel analyzed positions the same?

Sorting – general notes

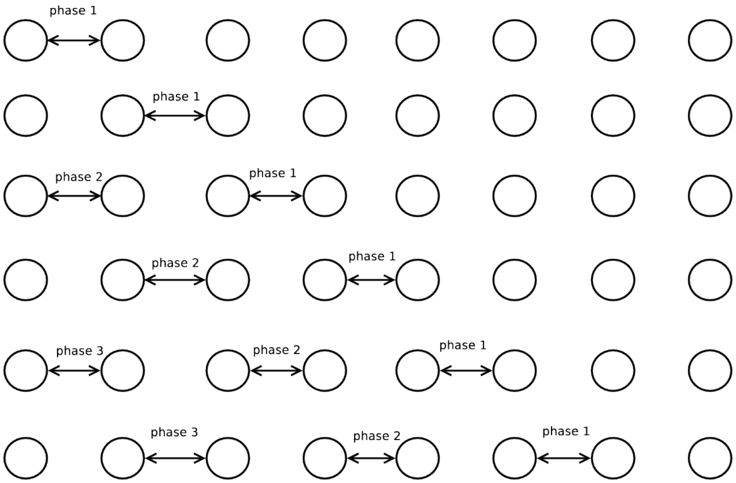
- the sequential bubble sort is simple and inefficient (complexity of $O(n^2)$). Can it be parallelized efficiently?
- note that partitioning of the initial vector and recursive application of the bubble sort on subvectors + O(n) merging does work faster than the initial bubble sort even in a sequential version – why?
- partitioning of the initial vector recursively until single elements are integrated leads to mergesort. What is the complexity of a parallel implementation?
- how parallelization of quick sort differs from parallelization of mergesort?

Sorting – parallelization of bubble sort

How can the bubble sort be parallelized efficiently?
Sequentual algorithm:

- in the sequential version each phase starts after the previous phase has completed
- the complexity of each phase is O(n)
- this results in complexity of the whole sequential algorithm of $O(n^2)$

Sorting – parallelization of bubble sort

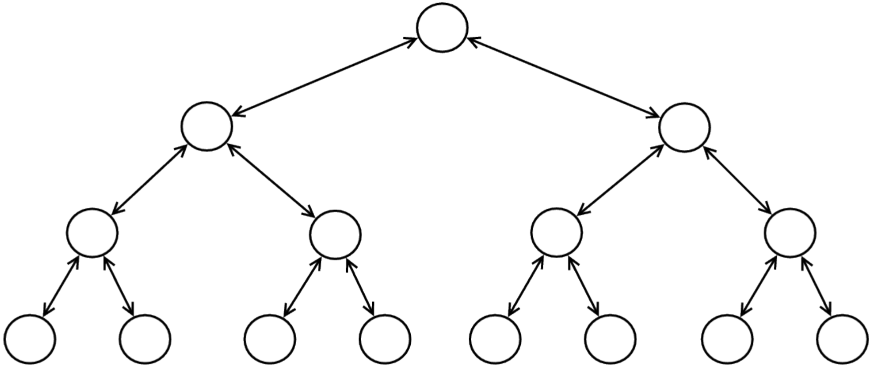


Sorting – parallelization of bubble sort

How can the bubble sort be parallelized efficiently?
Parallel algorithm:

- the idea of this parallel algorithm is the practical application of pipelining
  - comparison and substitution in the next phase at the beginning of the vector takes place in parallel with comparision and substitution of the further part of the vector from the previous phase
- the complexity of each phase is O(n) but the phases are parallelized
- consequently the start of each phase is delayed by 2 steps from the start of the previous phase
- additionally the last phase would need to execute some steps sequentially
- this results in complexity of the whole parallel algorithm of O(n)

Sorting – parallelization of merge sort



let us assume k processors in a parallel version of mergesort

Sorting – parallelization of merge sort

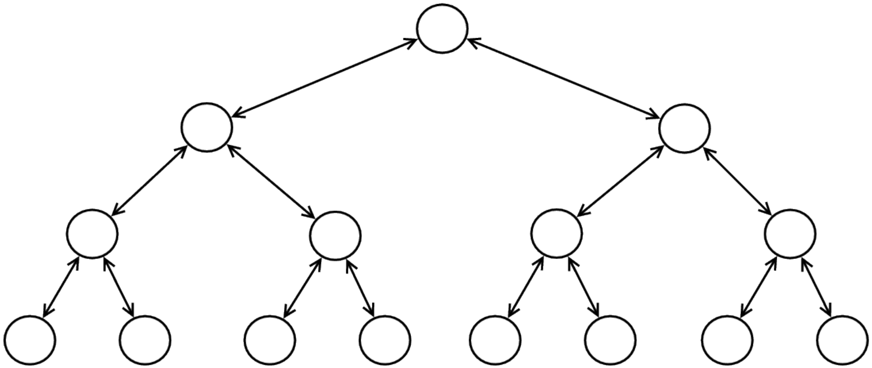
How can the merge sort be parallelized efficiently?
Parallel algorithm:

- the idea of this parallel algorithm is that there are k processes each of which can sort parts of the initial vector in parallel
- the tree would have the height of $log_2(k)$
- in each phase i each processor would need to merge 2^i elements that takes $O(2^i)$ steps
- consequently the merging phase would take $\sum_{i=1}^{log_2(k)} 2 * 2^i$ steps
- this results in complexity of the whole parallel algorithm for the computing part of O(k) assuming there are as many processors as the number of elements
- communication – there will be O(k) steps involving startup times as well as the total data size sent in all time steps would amount to the size which is O(k)
- finally the total complexity of the algorithm is O(k)

Sorting – parallelization of quick sort

What would be different in case of quick sort compared to merge sort?

- average case
- worst case



let us assume k processors in a parallel version of mergesort

Sorting – parallelization of quick sort

How can the quick sort be parallelized efficiently?
Parallel algorithm (average case):

- the idea of this parallel algorithm is that there are k processes each of which operates in parallel
- at first one process operates on k elements
- in the next phase two processes each operate on $\frac{k}{2}$ elements and so on
- this results in the total complexity of O(k) for the parallel version
- communication – this is very much similar to merge sort i.e.:
  - startup times – the complexity here will be O(log k) because of the height of the divide-and-conquer tree
  - total data transferred – similar to computing steps will amount to O(k)
  - finally the total complexity of the parallel version will be O(k)

How to handle irregular trees? How to balance load in this case?

N-body problem – problem statement

Problem:

- there are N bodies in a space (2D, 3D) that change speeds and locations over time due to gravitational forces
- the goal is to perform a simulation over time – in parallel
- basic equations include:
  1. $F = G\frac{m_i m_j}{R_{ij}^2}$ where m_i is the mass of body i and R_{ij} is the distance between body i and body j
  2. $F = ma$
  3. $F = m\frac{dv}{dt}$
  4. $v = \frac{dx}{dt}$

It is an example of a problem in which there are multiple particles that can be stored in an array/list. Consequently, locations can be stored at the same time.

N-body problem – problem statement

Problem:

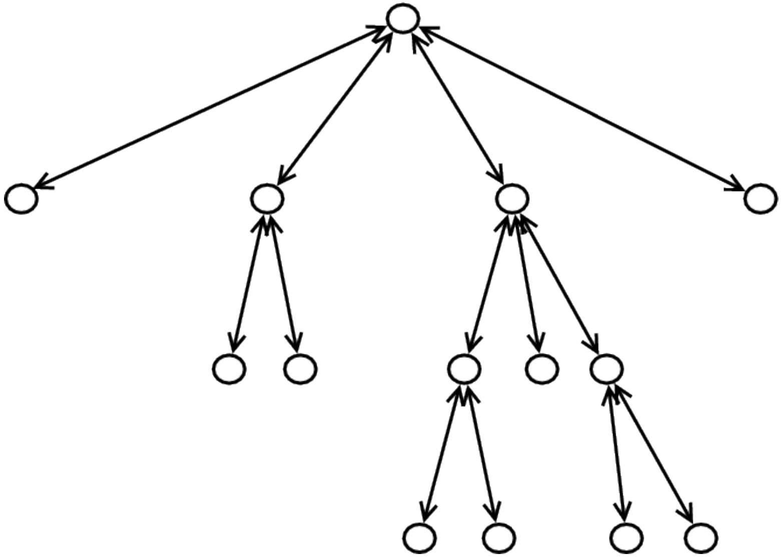
- after selection of a proper $\triangle t$ proper equations that couple values in successive time steps can be written
- $v_{t+1} = v_t + F\frac{\triangle t}{m}$ from $F = ma$ and substitution for a
- $v = \frac{x_{t+1} - x_t}{\triangle t}$ leading to $x_{t+1} = x_t + v\triangle t$
- in real simulations new locations and velocities can be computed in half iterations one after another
- the resulting complexity of the solution for one iteration of the simulation is $O(N^2)$ because each body influences each of the other bodies in space

N-body problem – solution

Solution:

- one optimization can be to compute center masses for groups of objects and then use such a center of mass when computing forces against a body that is far away
- the Barnes-Hut algorithm proposes to partition the domain in order to allow this:
    1. the initial space is divided into cubes (can be 4 or 8 depending on whether we consider 2D or 3D)
    2. each of the cubes will correspond to a node at a certain level of the divide-and-conquer tree
    3. such a node would hold a center mass for its own nodes (inside) that can be used for computing with distant objects
    4. if there are nodes inside then the cube is partitioned again, otherwise terminated
    5. the tree can be very imbalanced

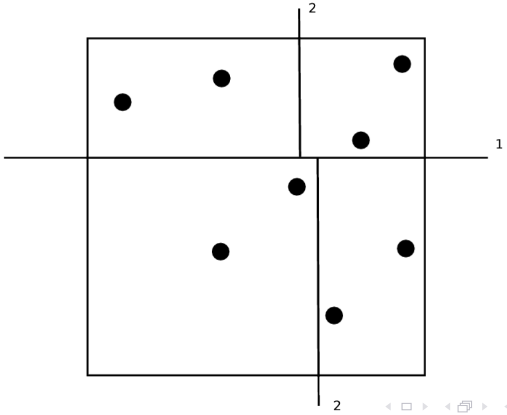
N-body problem – solution



N-body problem – parallel solution

Solution requires parallelization of bodies among available CPUs/cores. A reasonable approach is Recursive Coordinate Bisection – the algorithm operates recursively:

- cuts the space in successive dimensions (one by one)
- cuts in such a way that the number of bodies in each dimension is the same



Computing prime numbers within a particular range

Solution:

- naive:
  - check each number individually
  - high computational cost
  - time needed for numbers may be considerably different especially if there are large ranges
- sieve based:
  - start with small numbers – compute multiples and reject
  - how to parallelize such a solution? One approach would be to partition small numbers up to a certain threshold.

Computing prime numbers in parallel – other formulations

Other problems to consider:

- What about the case in which there are multiple subranges?
- Computing twin prime numbers within a particular range – does it change the problem?

Solving a system of linear equations – problem

Problem:

- The problem can be stated as solving **Ax**=**b**
- expanding the problem is as follows:

$$\begin{aligned} a_{0,0}x_0 + a_{0,1}x1 + ... + a_{0,n-1}x_{n-1} &= b_0 \\ &\dots \\ a_{n-1,0}x_0 + a_{n-1,1}x1 + ... + a_{n-1,n-1}x_{n-1} &= b_{n-1} \end{aligned} \tag{1}$$

There are several solutions to the problem including:

1. elimination that converts the initial problem into a triangular form that allows to find a solution
2. iterative

Solving a system of linear equations – Gauss method

Solution:

- in this solution in the sequential version in the n-1 iterations we modify coefficients in each row in order to eliminate one more row from updates in successive iterations and finally obtain the trinagular form
- in each iteration:
  1. it is necessary to go through each row
  2. update coefficients in each element of this row

This solution has the complexity of $O(n^3)$. In a parallel version it is possible to use n processors each of which can be responsible for updates of each row in each iteration. Finally this gives the complexity of $O(n^2)$. Note, though, that some of the processors will be idle for some time because of the triangular form

Solving a system of linear equations – Jacobi

Solution:

- this is an iterative solution to the problem of $\mathbf{Ax}=\mathbf{b}$ that can be stated as follows:
- for the k-th iteration each variable to be found x_i^k can be updated as follows:

$$x_i^k = \frac{1}{a_{i,i}}(b_i - \sum_{j \neq i} a_{i,j} x_j^{k-1})$$

This solution comes from rearranging of the equations. Depending on the problem, accuracy and convergence might be an issue. The algorithm may be solved in parallel since each processor may handle updates of individual variables. This requires synchronization between iterations.

Solving a system of linear equations – real life problems

The Jacobi solution may correspond to solutions to various real life problems [2] that are modeled by differential equations. For instance, the Laplace equation $\frac{\delta^2 f}{\delta x^2} + \frac{\delta^2 f}{\delta y^2} = 0$. Using a discrete formulation this leads to
$f(x,y) = \frac{1}{4}(f(x-\triangle, y) + f(x, y-\triangle) + f(x+\triangle, y) + f(x, y+\triangle))$.
Now if we arrange successive $f(x+\alpha\triangle, y+\beta)$ as x_i then the last equation can be rewritten as a linear equation. Finally, an equation for each i can be written. This results in a sparse matrix that can be solved using the Jacobi solution in parallel.

FDTD, MRTD – dynamic parallel solutions

Problem:

- Typical formulation of a problem such as a physical phenomenon – as a (set of) differential equations
- the formulation is then transformed into a solution in discrete time steps
- in each of the time step it is necessary to update values assigned to cells
- synchronization is necessary between time steps
- in the parallel version the same applies forcing the following two steps:
    1. computations
    2. communication – synchronization of so-called ghost cells

  Notice that contrary to the N-body problem here we need to update many cells aligned in 1, 2 or 3 dimensions

FDTD, MRTD – dynamic parallel solutions

Certain formulations such as FDTD correspond to regular computations (same across the domain) which results in partitioning that can be done before the main time loop:

- weights same
- different weights – how to partition then? how to find weights?
  1. partition the domain into blocks – a certain number of cuts in X, Y, Z planes – how to find this number? We need to consider not only execution times but also communication
  2. another reasonable solution can be Recursive Coordinate Bisection (RCB) – but contrary to the N-body problem in this case we consider many cells aligned in X, Y and Z dimensions with potentially various weights – we use sums of weights for partitioning, cuts in successive dimensions minimize communication

Note that some other formulations (such as MRTD) which are better in a sequential version (because give up computations in some parts of the domain) are much more difficult to parallelize and require dynamic repartitioning

Genetic algorithms – parallelization

Possible approaches:

1. partition multiple chromosomes within a population
2. use multiple populations and synchronize computations from time to time

What problems can be solved using this method? How to find e.g.:

- zeros of a complex function
- solution to a set of linear equations

Parallel interpolation – IDW

Problem – the problem is to interpolate certain problem-dependent values based on values available for certain known locations. The typicall solution would work as follows:

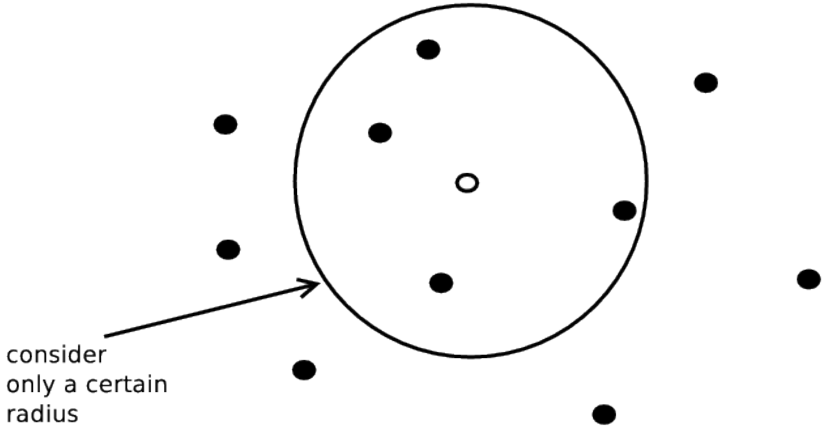
- for each point with an unknown value d_x and another point d_i a weight is computed $w_{xi} = \frac{1}{d(d_x,d_i)}$ where d() denotes the distance
- finally the unknown value is computed as follows: $v_x = \sum_{i=1}^{n} v_i \frac{w_{xi}}{\sum_{j=1}^{n} w_{xj}}$

The standard problem could be parallelized just by assignment of unknown points among available CPUs/cores. The following should be noted:

1. if there are obstacles in the space then some points with known data values then some points within the radius would not affect the given point
2. there can be various numbers of points with known data values within the radius – there is a need for efficient load balancing
3. how does this problem compare to the N-body problem?

Parallel interpolation – IDW

●	point in a space for which measurements are available
○	an exemplary point in a space for which a value must be interpolated



Bibliography I

[1] David B Kirk and W Hwu Wen-mei.
Programming massively parallel processors: a hands-on approach.
Morgan Kaufmann, 2010.

[2] Barry Wilkinson and Michael Allen.
Parallel programming. Techniques and Applications using Networked Workstations and Parallel Computers.
Prentice Hall, 2004.