Parallel Programming and High-Performance Computing

Part 6: Dynamic Load Balancing

Dr. Ralf-Peter Mundani
CeSIM / IGSSE / CiE
Technische Universität München

Overview

- definitions
- examples of load balancing strategies
- space filling curves
- swarm intelligence

Computers make it easier to do a lot of things,
but most of the things they make it easier to do
don't need to be done.
—Andy Rooney

Definitions

- motivation
  - central issue: fairly distribution of computations across all processors / nodes in order to optimise
    - run time (user's point of view)
    - system load (computing centre's point of view)
  - so far, division of a problem into a fixed number of processes to be executed in parallel
  - problem
    - amount of work is often not known prior to execution
    - load situation changes permanently (adaptive mesh refinement within numerical simulations, I/O, searches, …)
    - different processor speeds (heterogeneous systems, e. g.)
    - different latencies for communication (grid computing, e. g.)
  - objective: load distribution or load balancing strategies

Definitions

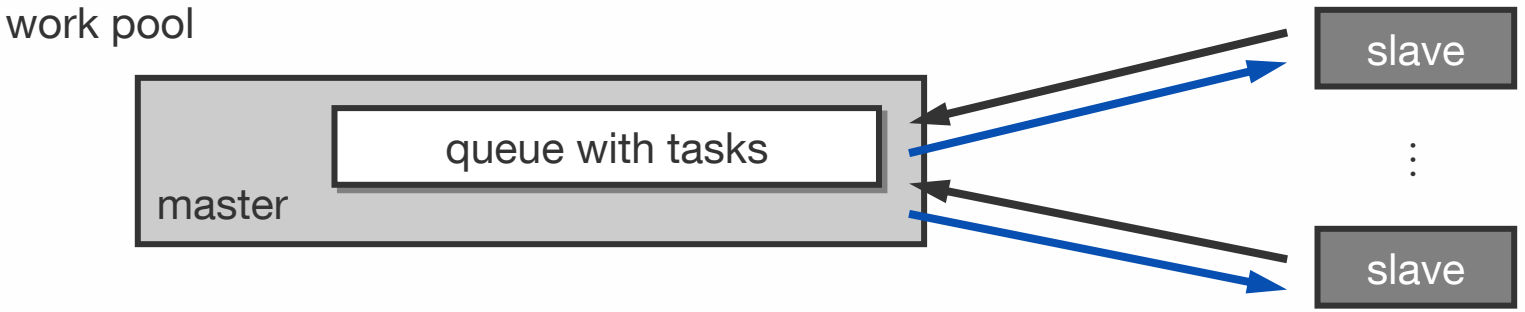
- static load balancing
  - to be applied before the execution of any process (in contrast to dynamic load balancing to be applied during the execution)
  - usually referred to as mapping problem or scheduling problem
  - potential static load-balancing techniques
    - *round robin*: assigning tasks (more general formulation than work to cover both data and function parallelism) in sequential order to processes, coming back to the first when all processes have been given a task
    - *randomised*: selecting processes at random to assign tasks
    - *recursive bisection*: recursive division into smaller subproblems of equal computational effort with less communication costs
    - *genetic algorithm*: finding an optimal distribution of tasks according to a given objective function

Definitions

- dynamic load balancing
  - division of tasks dependent upon the execution of parts of the program as they are being executed ➔ entails additional overhead (to be kept as small as possible, else bureaucracy wins)
  - assignment of tasks to processes can be classified as
    - *centralised*
      - tasks are handed out from a centralised location
      - within a master-slave structure one dedicated master process directly controls each of a set of slave processes
    - *decentralised*
      - tasks are passed between arbitrary processes
      - worker processes operate upon the problem and interact among themselves ➔ a worker process may receive tasks from other or may send tasks to other worker processes

Definitions

- centralised dynamic load balancing
  - example: work pool
    - master process holds a collection of tasks to be performed by the slave processes
    - tasks are sent (→) to slave processes
    - when a task is completed, a slave process requests (←) another task from the master process
    - all slaves are the same (*replicated worker*), but specialised slaves capable of performing certain tasks are also possible

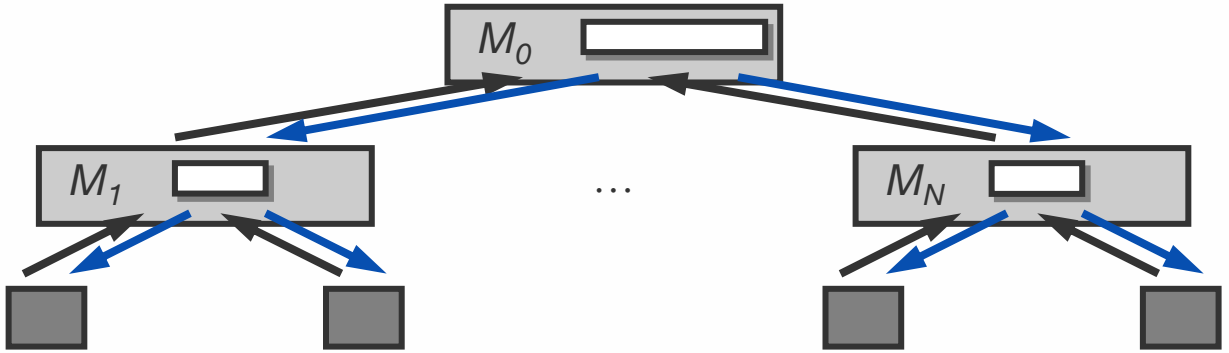


Definitions

- centralised dynamic load balancing (cont'd)
  - work pool techniques can also be readily applied when
    - tasks are quite different and of different size (in general, it is best to hand out larger tasks first to prevent idle waiting)
    - amount of tasks may change during execution, i. e. execution of one task might generate new tasks (to be submitted to the master)
  - computation terminates if both of the following are satisfied
    1) task queue is empty
    2) every process made a request for another task without any new tasks being generated (even if (1) is true a still running process may provide new tasks for the task queue)
  - a slave may detect the program termination condition by some local termination condition (searches, e. g.), hence it has to send a termination message to the master for closing down all others

Definitions

- decentralised dynamic load balancing
  - example: distributed work pool
    - drawback of centralised model: master might become bottleneck in case of too many slaves
    - hence, work pool is distributed among several masters
    - each master controls one group of slaves
    - several layers of decomposition possible ➔ building up a tree hierarchy with tasks being passed downwards and requests / messages being passed upwards



Definitions

- decentralised dynamic load balancing (cont'd)
  - example: fully distributed work pool
    - once tasks are (initially) distributed among processes (that moreover are able to generate new tasks), all processes can execute tasks from each other
    - tasks could be transferred by a
      - *receiver-initiated method*: a process that has only few or no tasks to perform requests tasks from other processes it selects (works well at high system loads)
      - *sender-initiated method*: a process with heavy load sends tasks to other processes it selects and that are willing to accept them (works well at light overall system loads)
    - in general, avoid passing on the same task that is received
    - which one to prefer, what kind of flaws do they have?

Overview

- definitions
- examples of load balancing strategies
- space filling curves
- swarm intelligence

Examples of Load Balancing Strategies

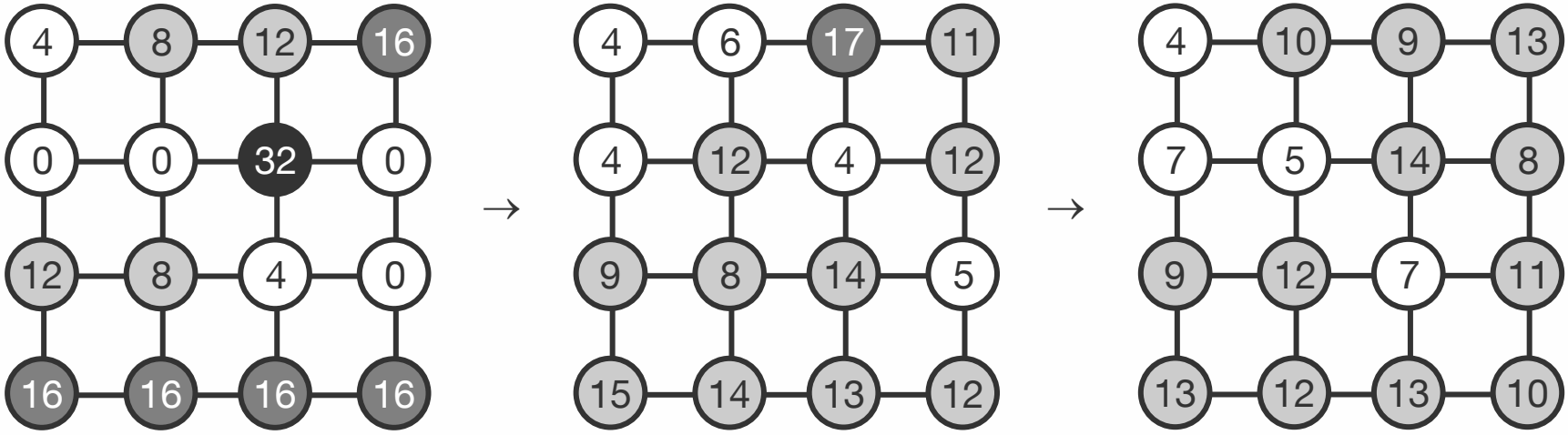
- diffusion model (a. k. a first order scheme)
  - analogy to physical processes in nature (salt or ink in water, e. g.)
  - original algorithm introduced by CYBENKO (1989) for static network topologies, meanwhile it has been often studied and derived (second order scheme, dynamic network topologies, e. g.)
  - *idea*: a process P_i balances its load simultaneously with all its neighbours $N(i)$ ➔ ratio α_{ij} of the load difference between process P_i and P_j is swapped between them according to

$$w_i^{(t+1)} = w_i^{(t)} - \sum_{j \in N(i)} \alpha_{ij} \cdot \left(w_i^{(t)} - w_j^{(t)} \right), \quad 1 \le i \le p, \quad -1 < \alpha_{ij} < 1$$

    where $w_i^{(t)}$ is the workload done by process P_i at time t
  - various methods to be found that determine parameter α_{ij} such as
    - optimal choice: needs global knowledge of the network
    - BOILLAT choice: needs only local knowledge of the neighbours

Examples of Load Balancing Strategies

- diffusion model (cont'd)
  - update of workload can be done
    a) after all balancing factors have been computed (JACOBI-like)
    b) during computation of balancing factors (GAUSS-SEIDEL-like)
  - example: first two iteration steps according to method a) for a 2D grid with a ratio of $\alpha = 0.25$ for workload swapping



initial setup ($t = 0$) first step ($t = 1$) second step ($t = 2$)

Examples of Load Balancing Strategies

- bidding (economic model)
  - analogy to mechanisms of price fixing in markets
  - idea
    - process (with high workload) advertises tasks to its neighbours
    - neighbours submit their free resources as bid
    - process with highest bid (i. e. largest free resources) wins
  - remarks
    - maybe several rounds of bidding necessary ➔ successively extending the range of bidders
    - in case of sudden workload peaks, a process might reject the purchased tasks
    - processes with free resources are still allowed to ask for tasks
  - drawback: quite complex analysis of this model

Examples of Load Balancing Strategies

- balanced allocation (balls into bins)
  - basic idea: placing *N* balls into *N* bins at random choice (extensively studied problem from probability and statistics)
  - variant of the above
    - each ball comes with *D* possible destinations (to be placed), chosen independently and uniformly at random
    - then the ball is placed in the least full bin among the *D* possible destinations
  - applied to load balancing: a process selects *D* processes at random and passes some of its workload to the least loaded one
  - for temporary tasks (i. e. tasks that are finished at unpredictable times) this strategy has a competitive ratio of $O(\sqrt{N})$ compared to the optimal off-line strategy (that has global knowledge)

Examples of Load Balancing Strategies

- precalculation of the load
  - all strategies so far are based on local information only
  - hence, load balancing is often quite expensive since (from a global point of view) balancing steps not always lead to a better load distribution among the processors
  - idea
    - global determination of the workload at program start or at certain points in time
    - global determination of an appropriate load distribution
    - workload transfer with less communication
  - developed and used for hierarchical network topologies ➔ workload recording and load balancing between child and parent nodes

Overview

- definitions
- examples of load balancing strategies
- space filling curves
- swarm intelligence

Space Filling Curves

- definition
  - origin of the idea: analysis and topology (“topological monsters”)
  - nice example of a construct from pure mathematics that gets practical relevance only decades later
  - definition of a space filling curve (SFC)
    - curve: image of a continuous mapping f: $[0,1] \rightarrow [0,1]^D$
    - SFC: continuous, surjective mapping f: $[0,1] \rightarrow [0,1]^D$ that covers an area (with a JORDAN content) greater than zero
  - prominent representatives
    - HILBERT’s SFC (1891): most famous SFC
    - PEANO’s SFC (1890): oldest SFC
    - LEBESGUE’s SFC: most important SFC for computer science
  - further reading: H. Sagan, *Space-Filling Curves*, Springer (1994)

Space Filling Curves

- HILBERT's space filling curve
  - for reasons of simplicity only in 2D ➔ f: $I = [0,1] \rightarrow [0,1]^2 = Q$
  - construction of SFC follows the geometric conception

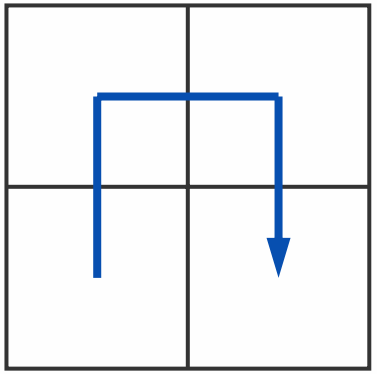
    If I can be mapped onto Q in the space filling sense, then each of the four congruent subintervals of I can be mapped to one of the four quadrants of Q in the space filling sense, too.

  - recursive application of above preserves
    - *neighbourhood relations*: neighbouring subintervals in I are mapped onto neighbouring subsquares of Q
    - *subset relations (inclusion)*: from $I_1 \subseteq I_2$ follows $f(I_1) \subseteq f(I_2)$
  - border case: HILBERT's SFC

Space Filling Curves

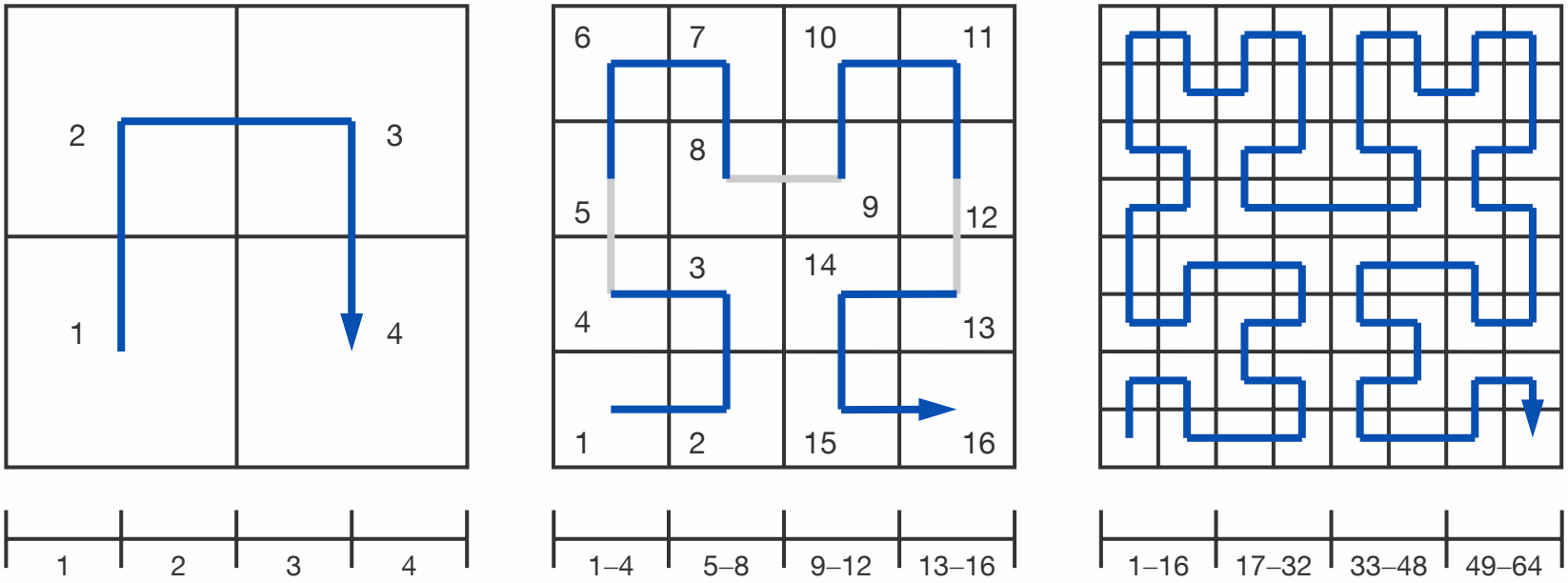
- HILBERT's space filling curve (cont'd)
  - correspondence of nested intervals in *I* and nested squares in *Q* provides pairs of points in *I* with corresponding image points in *Q*
  - of course, the iterative steps in this generation process are of practical relevance, not the border case itself
    1) starting with a generator or "Leitmotiv" that defines the order in which the subsquares are visited
    2) applying generator in each subsquare (with appropriate similarity transformations if necessary)
    3) connecting the open ends

generator for HILBERT's SFC



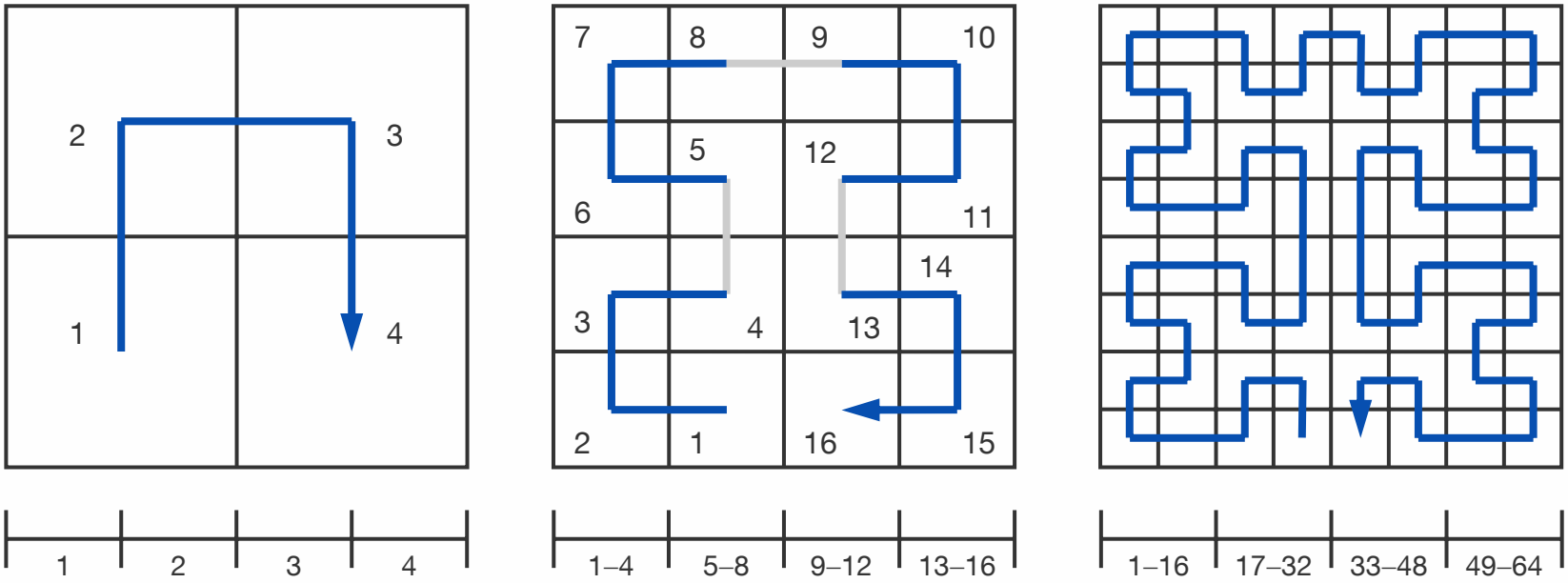
Space Filling Curves

- HILBERT's space filling curve (cont'd)
  - classical version of HILBERT



Space Filling Curves

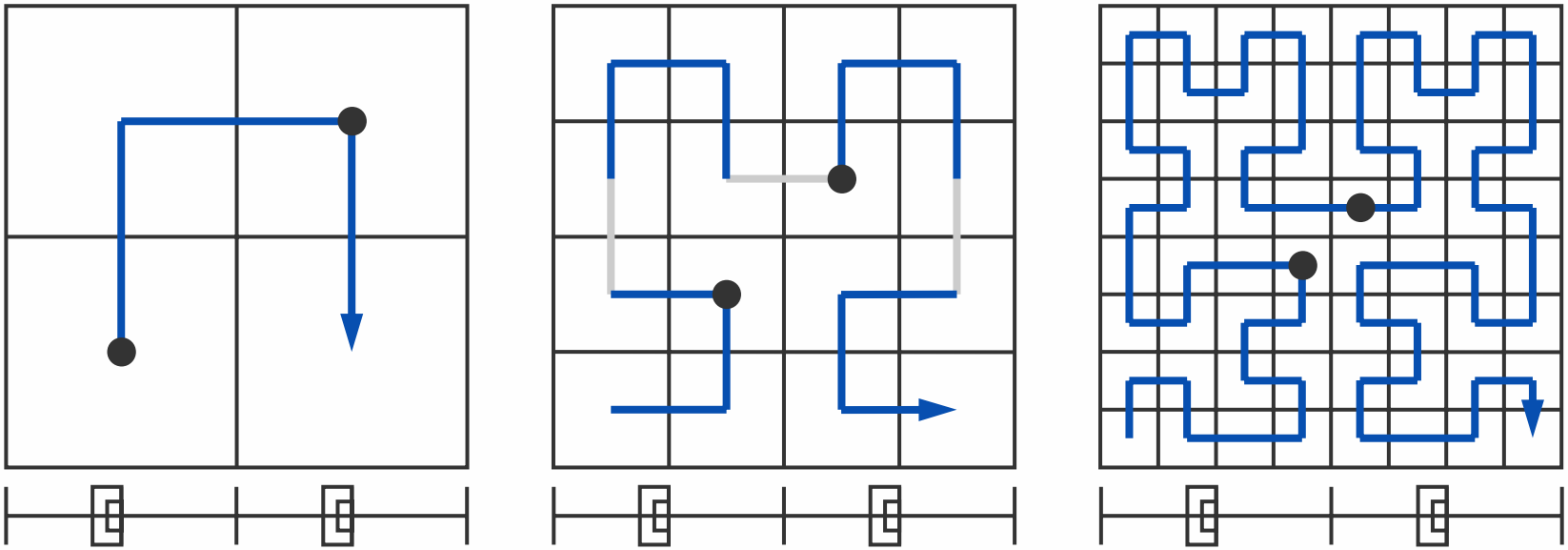
- HILBERT's space filling curve (cont'd)
  - variant of MOORE



  - modulo symmetry, these are the only two possibilities

Space Filling Curves

- HILBERT's space filling curve (cont'd)
  - all iterations are injective, but HILBERT's SFC itself is not injective (there are image points with more than one original point in *I*)

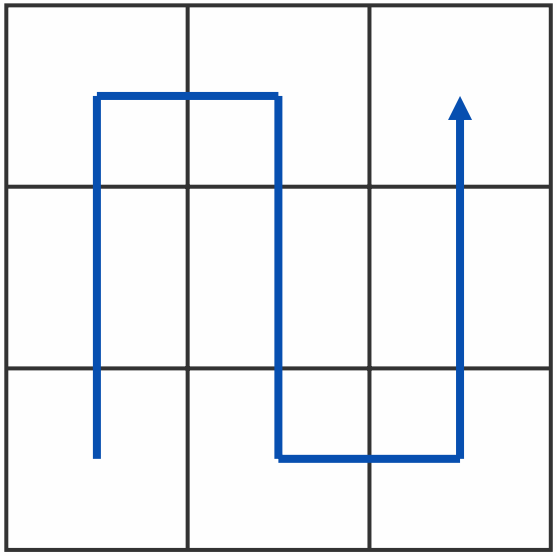


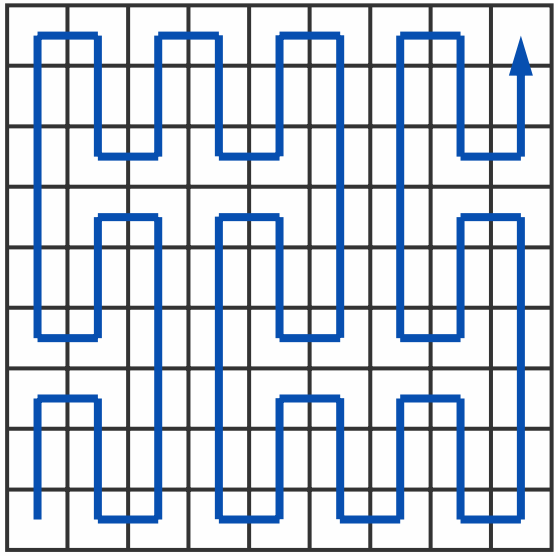
  - important precondition: there exists a bijective mapping between two finite-dimensional smooth manifolds (CANTOR, 1878), but it cannot be both bijective and continuous (NETTO, 1879)

Space Filling Curves

- PEANO's space filling curve
  - ancestor of all SFCs
  - subdivision of *I* and *Q* into nine congruent subdomains
  - definition of a generator, again, defines the order of visit

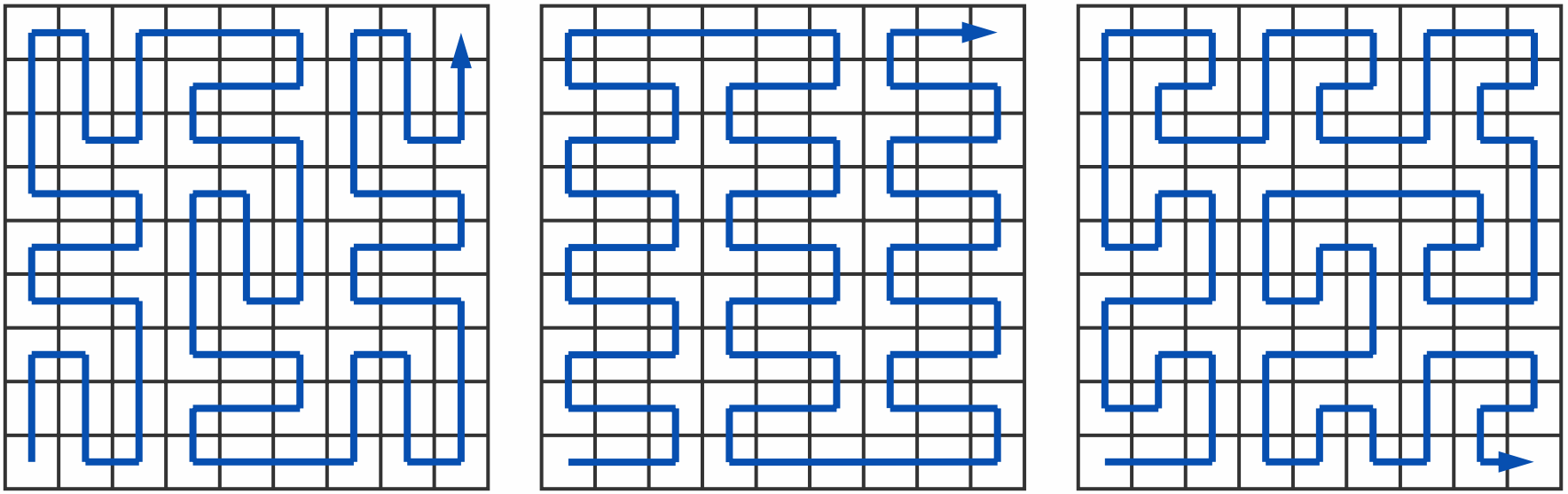
3	4	9
2	5	8
1	6	7





Space Filling Curves

- PEANO's space filling curve (cont'd)
  - now, there are (modulo symmetry) 273 different possibilities to recursively apply the generator preserving neighbourhood and inclusion



serpentine type (left and centre) and meander type (right)

Space Filling Curves

- LEBESGUE's space filling curve
  - definition of LEBESGUES's SFC by the CANTOR set
  - CANTOR set *C*: repeatedly deleting the open middle thirds of [0,1]

  - *C* is defined as set of points not excluded, hence the remaining interval can be computed by the total length removed

$$\sum_{N=0}^{\infty} \frac{2^N}{3^{N+1}} = \frac{1}{3} + \frac{2}{9} + \frac{4}{27} + \frac{8}{81} + \cdots = \frac{1}{3} \cdot \left(\frac{1}{1 - \frac{2}{3}} \right) = 1$$

  - the proportion of the remaining interval seems to be $1 - 1 = 0$, but in fact *C* has the same cardinality as the unit interval [0,1] (!)

Space Filling Curves

- LEBESGUE's space filling curve (cont'd)
  - nested intervals of C to be represented by ternary numbers of the form $0_3.w_1w_2w_3\ldots$ with $w_i \in \{0, 1, 2\}$

  - example: parameter $T = 2/9$ can be written as [0,1], $[0_3.0,0_3.1]$, $[0_3.02,0_3.10]$, $[0_3.020,0_3.021]$, $[0_3.0200,0_3.0201]$, …

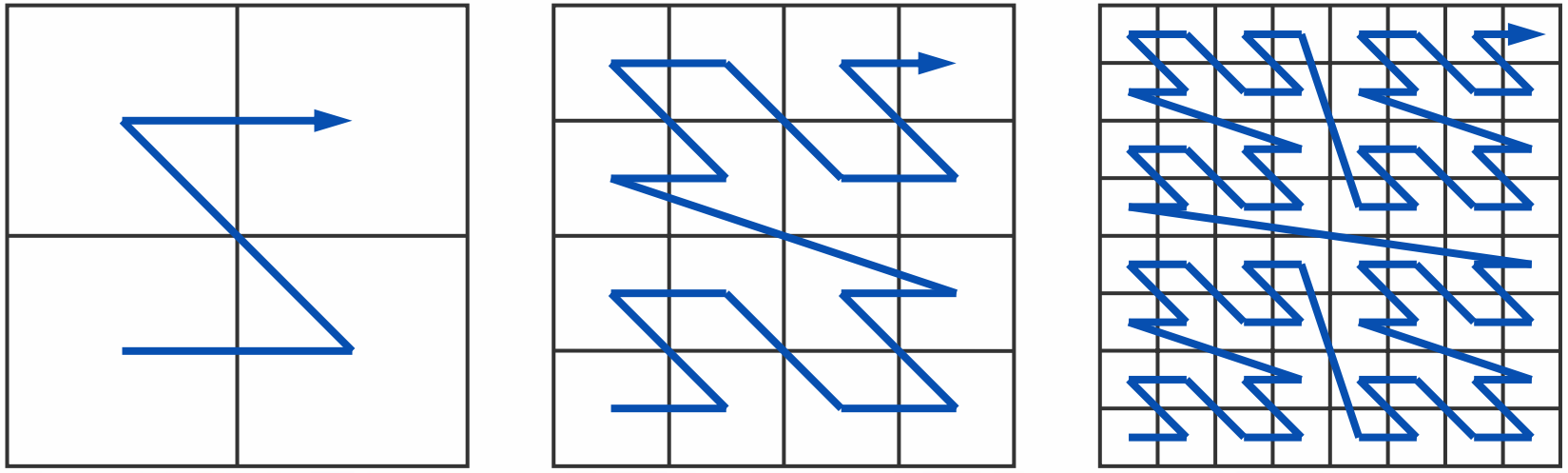
  - since all interval borders can be written in two different ways ($1_3.0$ or $0_3.222\ldots$, e. g.) and the middle third ($[0_3.1,0_3.2]$, e. g.) is repeatedly deleted, the CANTOR set only contains ternary numbers that consist of "0" and "2"

Space Filling Curves

- LEBESGUE's space filling curve (cont'd)
  - when mapping C to $[0,1]^2$ according to

$$f : \left(\frac{0_3.w_1w_2w_3w_4\ldots}{2}\right) = \begin{pmatrix} 0_2.x_1x_3\ldots \\ 0_2.y_2y_4\ldots \end{pmatrix}$$

and connecting the image points via linear interpolation, this results to LEBESGUE's SFC also referred to as "Z-order"



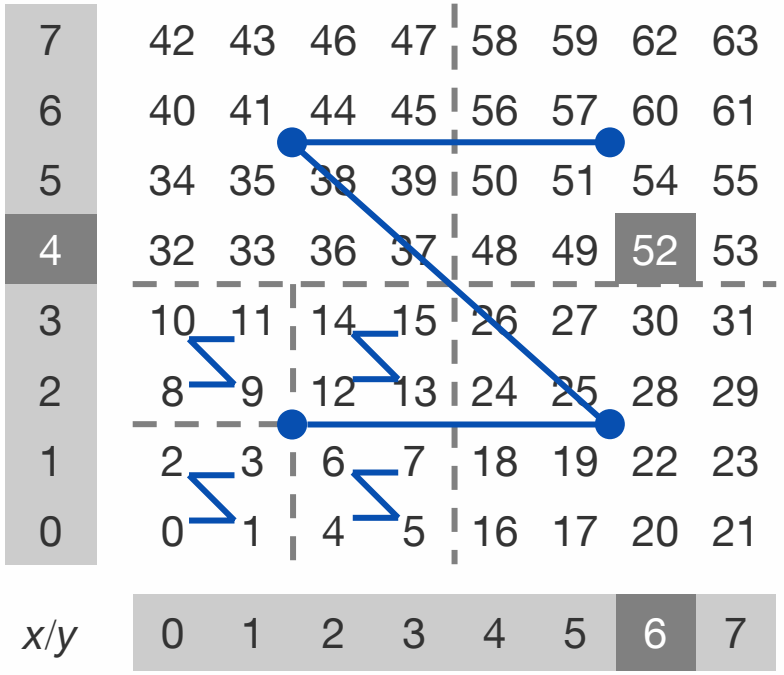
Space Filling Curves

- LEBESGUE's space filling curve (cont'd)
  - *Z*-ordering is well-known from quadtrees and octrees when lineariseing a tree by a depth-first traversal (provides a common naming scheme for cells ➔ lexicographic or MORTON index)
  - bitwise interleaving of coordinate values (*x*, *y*) leads to *Z*-value
  - useful for multidimensional range searches, e. g.

x: 02.100 → 4

y: 02.110 → 6

Z: 02.110100 → 52



Space Filling Curves

- LEBESGUE's space filling curve (cont'd)
  - compared to SFCs studied so far, there are several differences
    - both HILBERT's and PEANO's SFC can be nowhere differentiated, whereas LEBESGUE's SFC can be differentiated almost everywhere
    - both HILBERT's and PEANO's SFC are self-similar (close relation to fractals such as KOCH's snowflake or SIERPIŃSKI's triangle), but LEBESGUE's SFC is not self-similar
    - generation is less complicated as for HILBERT and PEANO
  - many applications of this SFC in computer science

Space Filling Curves

- applications
  - sequentialisation of multidimensional data to one dimension while preserving locality
    - data are stringed sequentially like pearls
    - neighbouring points in the unit interval [0,1] have neighbouring images in $[0,1]^D$
  - important applications such as
    - efficient multidimensional range searches in databases (Oracle, e. g.)
    - multi-particle or *N*-body problems
    - adaptive grid refinement for partial differential equations
    - dynamic load balancing

Space Filling Curves

- applications (cont'd)
  - example: range search in multidimensional data
    - query range $x = [1,6]$, $y = [1,3]$; highest Z-value MAX = 45
    - starting from $S = 22$, e. g., search goes between S and MAX

to speed up, BIGMIN = 33 (smallest Z-value in search range) is computed ➔ search goes only between BIGMIN and MAX

remark: for searches in the range lower than S, also a LITMAX (highest Z-value in search range) exists

7	42	43	46	47	58	59	62	63
6	40	41	44	45	56	57	60	61
5	34	35	38	39	50	51	54	55
4	32	33	36	37	48	49	52	53
3	10	11	14	15	26	27	30	31
2	8	9	12	13	24	25	28	29
1	2	3	6	7	18	19	22	23
0	0	1	4	5	16	17	20	21
x/*y*	0	1	2	3	4	5	6	7

Space Filling Curves

- load balancing
  - idea
    1) assign points q_i of some iteration of an SFC (i. e. $q_i \in Q$) to points s_i in the D-dimensional space
    2) linearly order points q_i according to SFC's original points $p_i \in I$
    3) simple partitioning based on this sequential order of points p_i (while preserving locality) to processors possible
  - two techniques for (1) using LEBESGUE's SFC (2D case)
    - compute binary numbers of length K from point s_i and retrieve corresponding point p_i in ternary representation

$$s_i := \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \rightarrow \begin{pmatrix} 0_2.x_1x_2x_3\ldots x_K \\ 0_2.y_1y_2y_3\ldots y_K \end{pmatrix} \rightarrow 0_3.w_1w_2w_3\ldots w_{2K} =: p_i$$

    - recursively "construct" K^{th} iteration of SFC for all points s_i and get all corresponding points p_i in ternary representation

Space Filling Curves

- load balancing (cont'd)
  - in both cases, keys (i. e. ternary numbers) already provide respective positions on *I*
  - steps left
    - sorting of the keys (step 2; main computational task of the load balancing algorithm)
      - may be costly at the beginning
      - nevertheless, new workload easily to be inserted into the sorted list afterwards
    - partitioning of the workload (step 3)
      - simple "cutting" of ordered list into equals parts for a fairly load distribution

Space Filling Curves

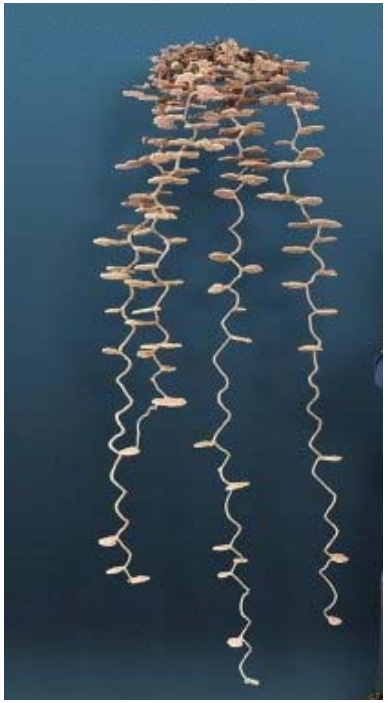
- quality considerations and costs
  - locality
    - continuity guarantees that originals that are close together will be mapped to closes image points
    - more important would be a “continuity of the inverse mapping”, but due to missing injectivity this is not possible
    - best properties for HILBERT’s SFC
  - load distribution
    - excellent parallelisation properties, almost perfect balancing
    - good efficiency already for small problem sizes
  - costs
    - communication along subdomain boundaries, but more complicated boundaries compared to recursive bisection
    - less overhead compared to other strategies

Overview

- definitions
- examples of load balancing strategies
- space filling curves
- swarm intelligence

Swarm Intelligence

- basics
  - origin of the idea: ant colonies in nature
  - ants and termites (as well as some bees and wasps) belong to the class of so called social insects
  - main characteristic of the above: building colonies
  - ants communicate indirectly via scent, also known as stigmergy (modification of local environment)
  - therefore, ants use pheromones
    - to be left along their paths for others to follow
    - to label bulk material inside the nest
  - if an ant finds a path to some food it will mark it, hence other can follow until they find a better path
  - nevertheless, ants may by chance decide not to follow ➔ chance of exploring alternatives



model of ant nest

Swarm Intelligence

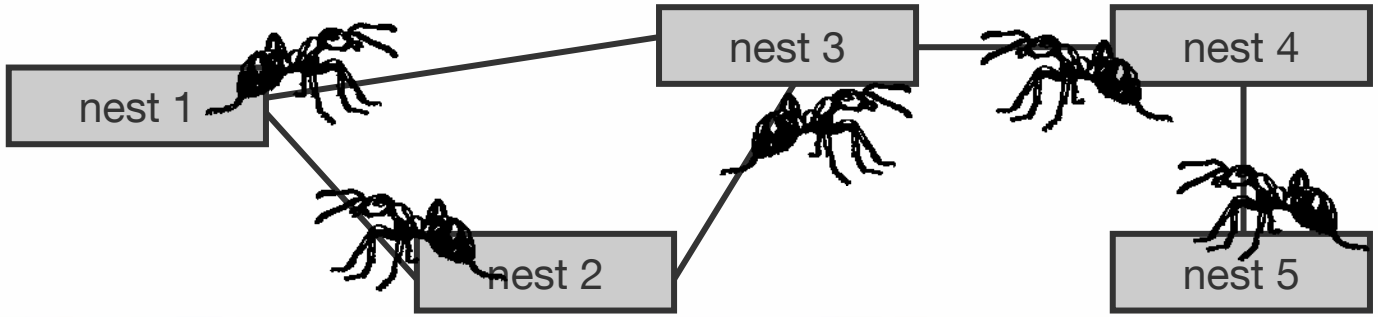
- basics (cont'd)
  - even a central decision maker is missing, ant colonies have a high grade of structure and organisation ➔ self-organisation
  - self-organisation is based on the following properties
    - *positive /negative feedback*: one ant follows a path or it does not follow a path (due to dissolution of pheromones, e. g.)
    - *amplification of deviation*: if one doesn't follow and finds some closer food, successively all others will follow the new path
    - *mutual communication*: key for spreading information among ants and exploiting advantages induced by negative feedback
  - hence, ant colonies are often referred to as collective or swarm intelligence
    - they can adopt to their environment and related problems
    - they can adopt their environment according to their demands

Swarm Intelligence

- complex adaptive systems
  - three simple rules for artificial ant colonies (RESNICK, 1994)
    1) ant wanders around randomly, until it encounters an object
    2) if it was carrying an object, it drops the object and continues to wander randomly
    3) if it was not carrying an object, it picks the object up and continues to wander
  - independent of the initial distribution of objects, a colony of those ants is able to group objects into large clusters
  - although there are no rules specific to initial conditions, unforeseen scenarios, variations in the environment, or presence of failures ➔ global behaviour of large enough colonies is adaptive and resilient
  - hence, CAS can achieve these properties without explicitly embedding them into the individual agents

Swarm Intelligence

- anthill system
  - proposed by MONTRESOR, MELING, and BABAOĞLU (2002)
  - composed of a self-organising overlay network of interconnected nests, where each nest is capable of hosting resources and performing computations
  - every node in the system is allowed to generate new tasks and to submit them to the network (tasks may remain in the originator or being transferred to other nests ➔ load balancing)
  - ants (autonomous agents) are generated by nests and travel across the nest network to detect unused computational power



Swarm Intelligence

- load balancing
  - variation of RESNICK's artificial ant algorithm
    1) when an ant is not carrying any object, it wanders about randomly until it encounters an object and picks it up
    2) when an ant is carrying an object, the ant drops it only after having wandered about randomly "for a while" without encountering other objects
  - colonies of such ants try to disperse objects (i. e. the actual tasks) uniformly over their environment rather than clustering them
  - ants may assume two different states
    - *searchMAX*: ant wanders about until it finds an "overloaded" nest; ant records nest's identifier and switches to searchMIN
    - *searchMIN*: ant wanders about looking for an "underloaded" nest; ant requests local task manager for task transfer from over- to underloaded nest and switches back to searchMAX

Swarm Intelligence

- load balancing (cont'd)
  - ants do not transport tasks ➔ to avoid carrying potentially large amount of data from one node to another while wandering about
  - concept of overloaded and underloaded nests are relative to the average load of the nests recently visited by an ant ➔ enables ants to make decisions about task transfers between nests with unbalanced loads without the necessity of a global knowledge
  - each nest stores collected information about an ant's last visited nests to be used by subsequent ants to drive their searchMAX or searchMIN phase ➔ at each step, the ant randomly selects the next node to visit among those that are believed to be more overloaded or underloaded, resp.
  - results: 100 idle nests and initially 10,000 tasks in one single node
    - only 15—20 iterations to transfer tasks to all other nodes
    - after 50 iterations, the workload is perfectly balanced