Unit 1, Part 2

Introduction to High Performance Computing

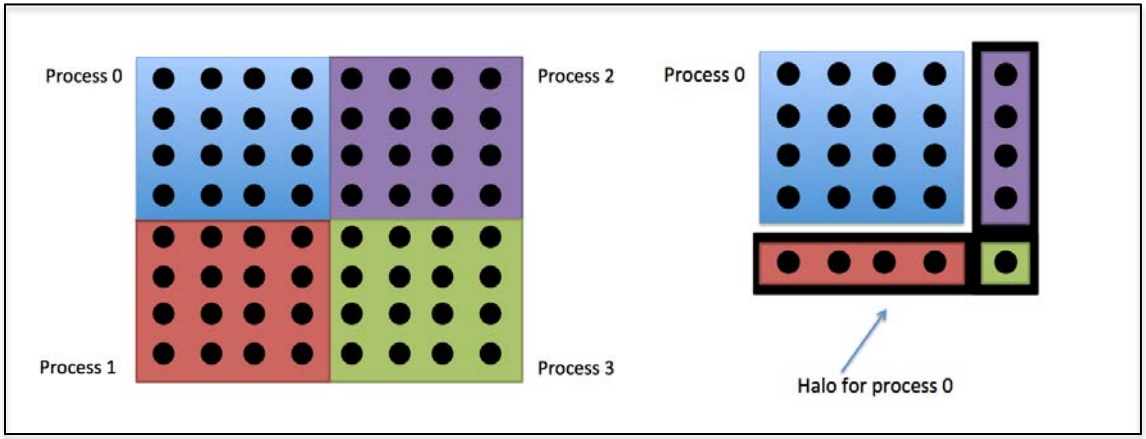
A Blue Waters Online Course
Fall 2016

David Keyes, Instructor

Professor of Applied Mathematics and Computational Science
Director, Extreme Computing Research Center

King Abdullah University of Science and Technology

Parallel Algorithms



David E. Keyes

King Abdullah University of Science and Technology

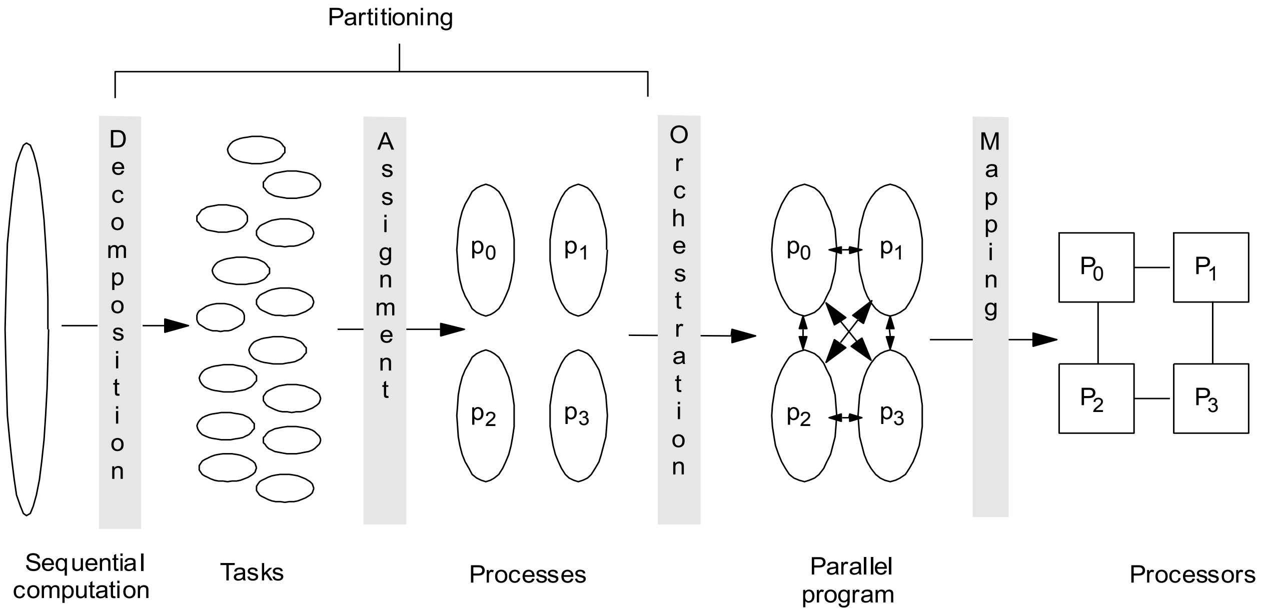
with acknowledgments to

Thomas Sterling

University of Indiana, Bloomington

Four steps to create a parallel program

- Decomposition of computation in tasks
- Assignment of tasks to processes
- Orchestration of data access, communication & synchronization
- Mapping processes to processors

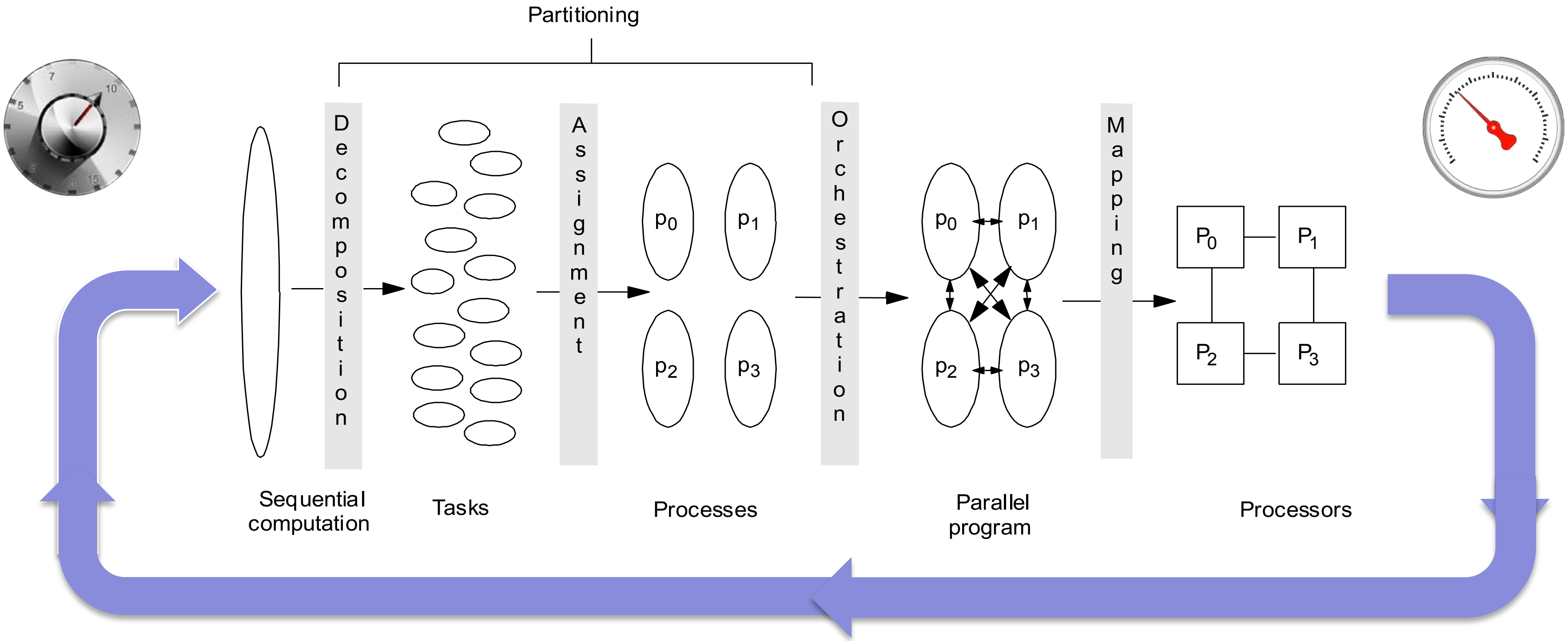


- First *three* involve the algorithm
- The *last* involves the architecture

c/o D. E. Culler, UC Berkeley, *et al.*

Co-design

- The one-way flow of the previous diagram is necessary, but usually *insufficient* if the goal is high performance
- Normally, we must *tune* the algorithm to the architecture
- This requires *measuring* the performance of the result and making changes to one or more stages of the algorithm
- A performance *model* should allow fewer iterations around the loop



Aside

- Occasionally, I will insert a stop sign to suggest that a student in self-learning mode – or an instructor in class mode – pause the video playback and brainstorm on what comes next, before proceeding

- Like having a performance model before collecting the data, anticipating what is coming next will help learn it – whether the anticipation proves met *or not*
- The video will not stop automatically and these signs can be ignored for fast review
- Probably, I will pause too little… so pause it, yourself!

What might you tune?

- Type of decomposition
- Number of tasks
- Clustering of tasks into processors
- Amount of information exchanged
- Frequency of exchanges
- Aggregation of exchanges
- Order of computation and communication
- Recomputation
- Mapping of tasks to processors
- Type of solution method

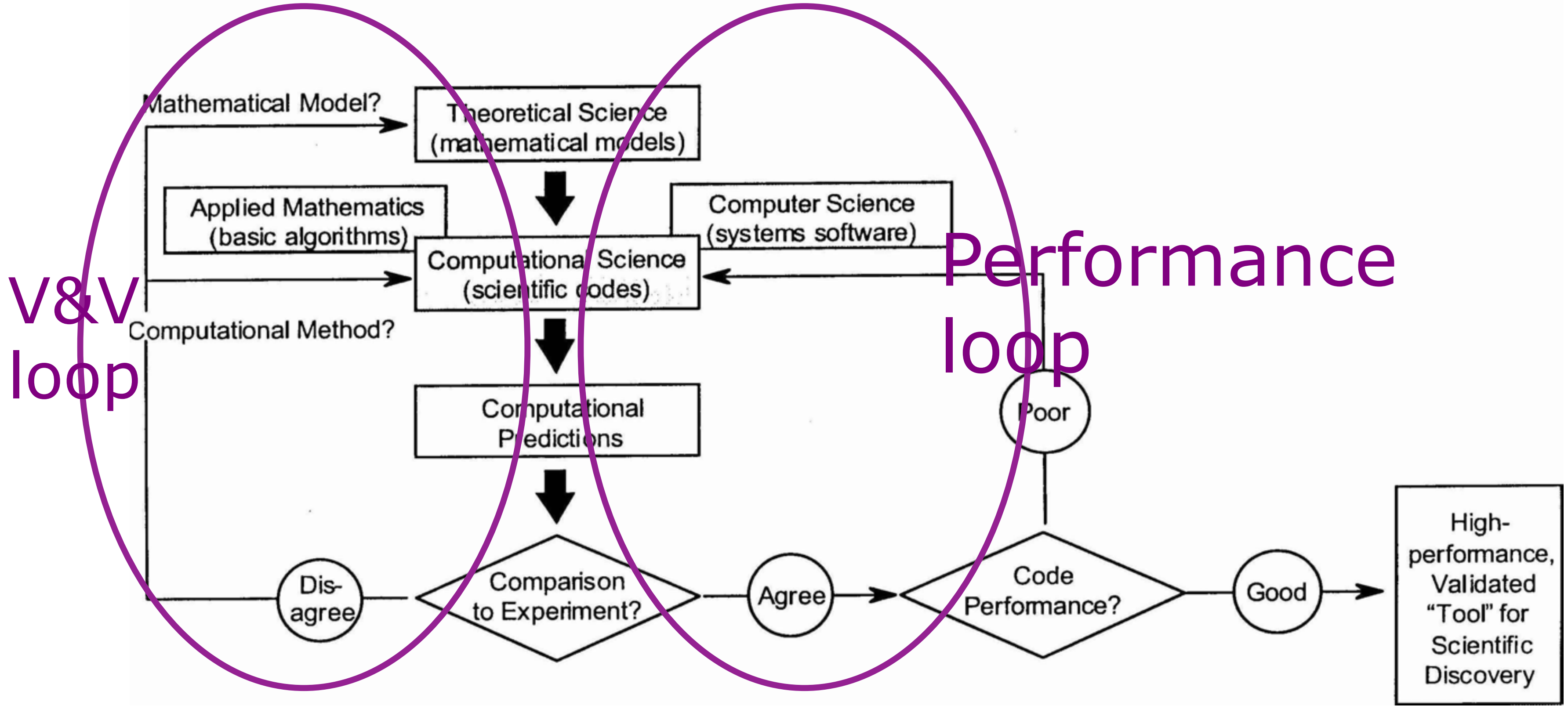
None of these change the problem being solved, just how to solve it

---

- Type of representation or basis
- Type of formulation (!)
- Goal of computation (!!)

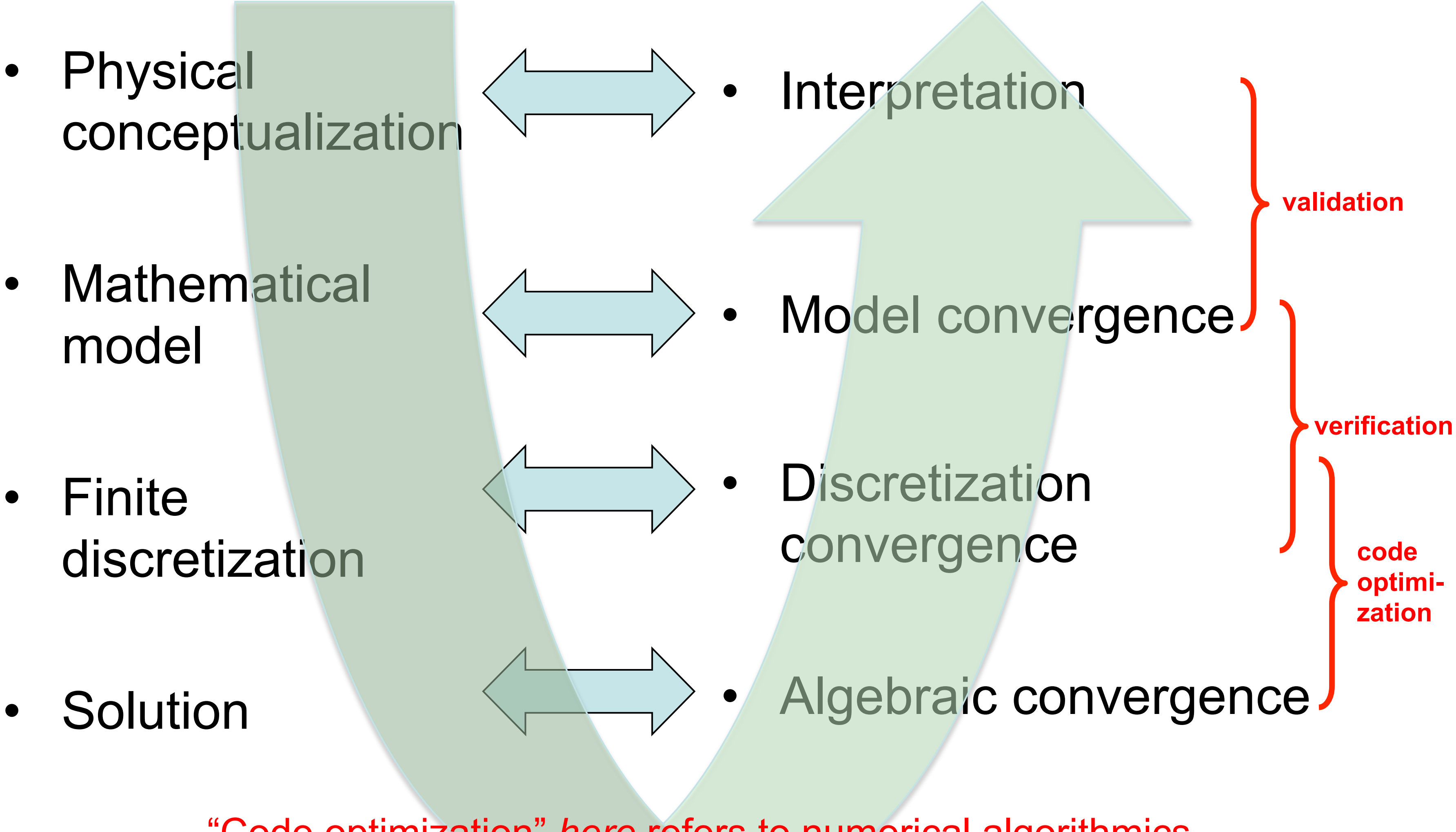
We may also be led to formulate a different problem that can be solved more efficiently!

Designing an HPC code – the diagram that launched SciDAC


Mathematical Model?
Theoretical Science (mathematical models)
Applied Mathematics (basic algorithms)
Computer Science (systems software)
Computational Science (scientific codes)
Computational Method?
Computational Predictions
Dis-agree
Comparison to Experiment?
Agree
Code Performance?
Poor
Good
High-performance, Validated "Tool" for Scientific Discovery
V&V loop
Performance loop

c/o T. Dunning, *SciDAC Report*, 2000

Modeling hierarchy



- Physical conceptualization
- Mathematical model
- Finite discretization
- Solution

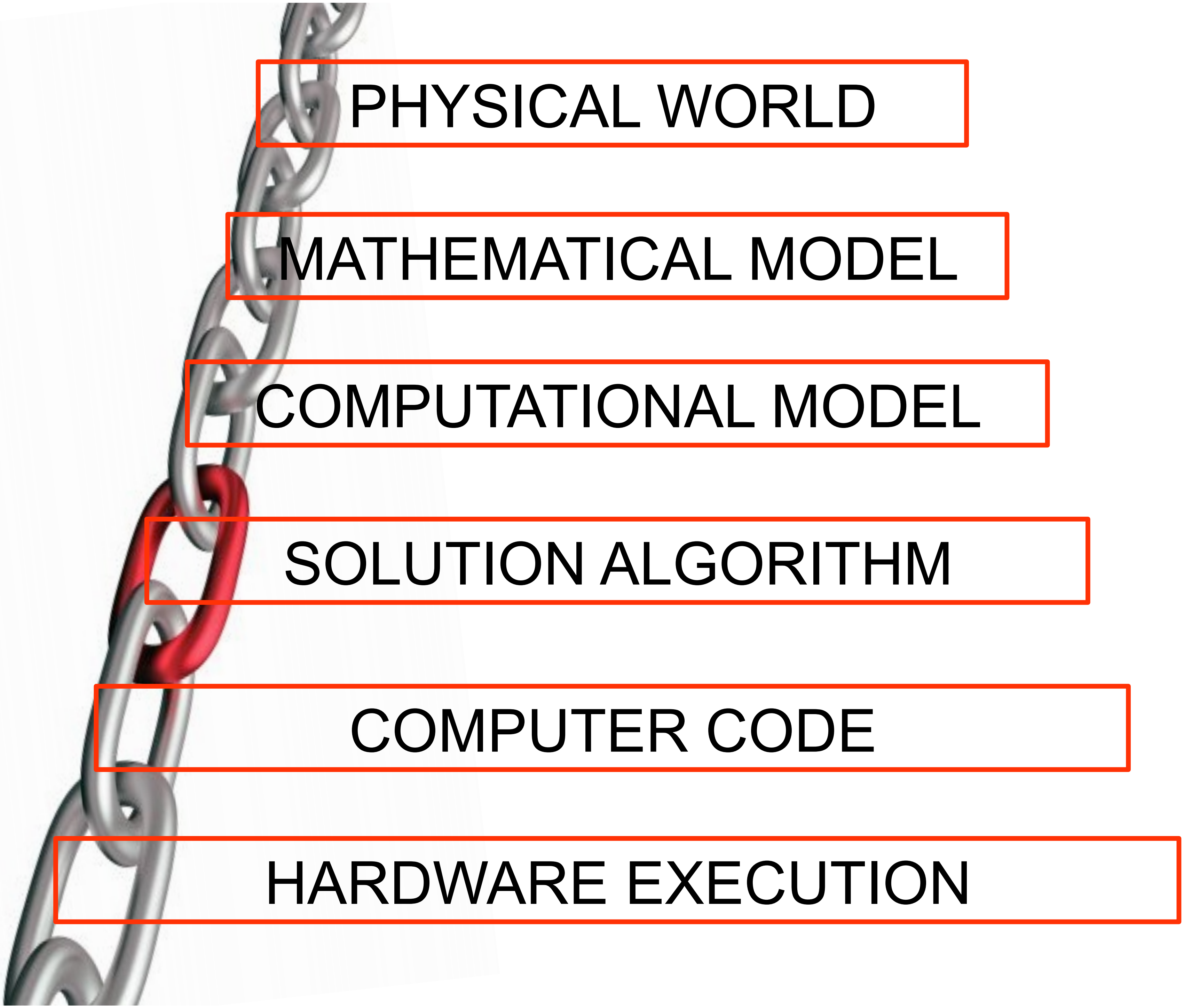
- Interpretation
- Model convergence
- Discretization convergence
- Algebraic convergence

validation

verification

code optimi-zation

“Code optimization” *here* refers to numerical algorithmics
Performance optimization is a *different* specialization



PHYSICAL WORLD

MATHEMATICAL MODEL

COMPUTATIONAL MODEL

SOLUTION ALGORITHM

COMPUTER CODE

HARDWARE EXECUTION

Some important algorithms

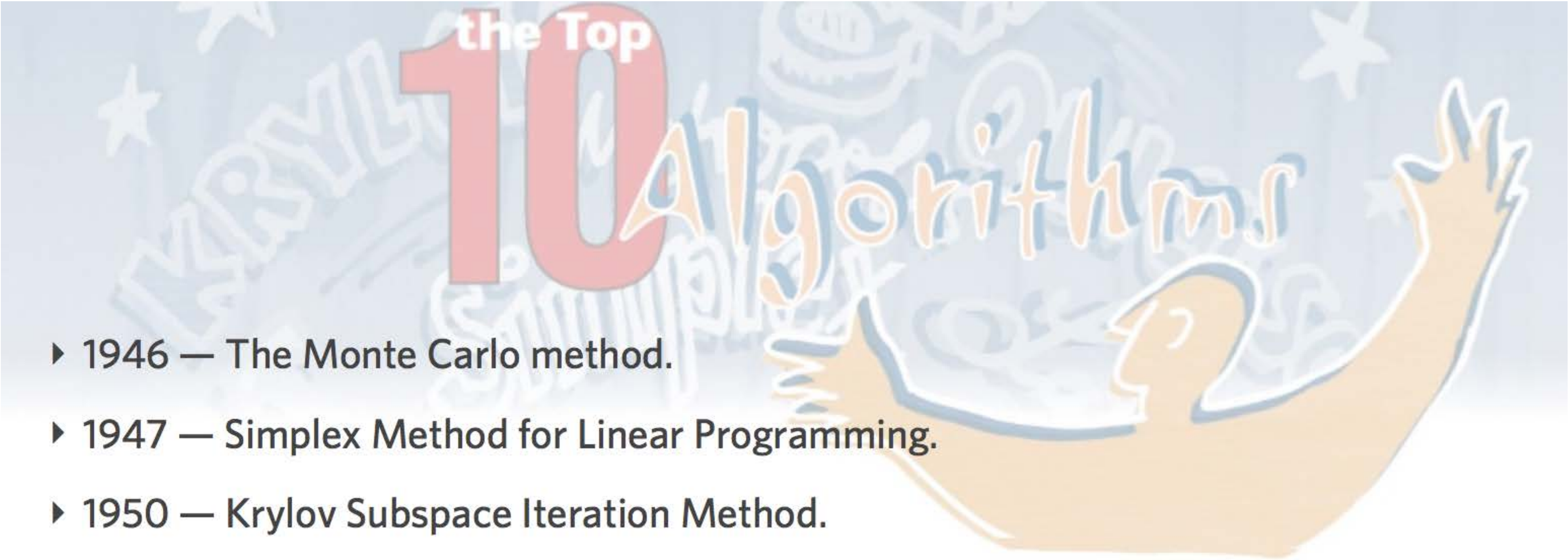
- In 2000, Jack Dongarra & Francis Sullivan whimsically introduced in *IEEE Computational Science and Engineering* a set of the “Top 10” algorithms of the previous century
  - actually, from the 1940s through the 1980s
  - some numerical, some non-numerical, and some “mixed”
  - typically simulation or optimization algorithms, or algorithms that support these tasks
- In 2008, Xindong Wu & Vipin Kumar systematically performed a similar selection focusing on data mining, for *Knowledge and Information Systems*
  - half from the 1990’s and half earlier
  - typically statistical and inferential tools

More important algorithms

- In 2004, Phil Colella, in a talk entitled *Defining Software Requirements for Scientific Computing,* named a set of seven key algorithms the “Seven Dwarves”
  - giant in importance (!)
- In 2006, a large team including David Patterson, John Shalf, and Kathy Yelick wrote a report entitled *Parallel Computing Research: A View from Berkeley*, in which the original 7 dwarves were complemented by 6 more from discrete mathematics – 13 in all
- In 2009, a large team led by Vladimir Safanov at St. Petersburg, published open source implementations of all 13 dwarves under the *Parallel Dwarfs project*
  - `https://paralleldwarfs.codeplex.com` ← Could be a good course project to push one of these along

STOP

What would you put on these lists?

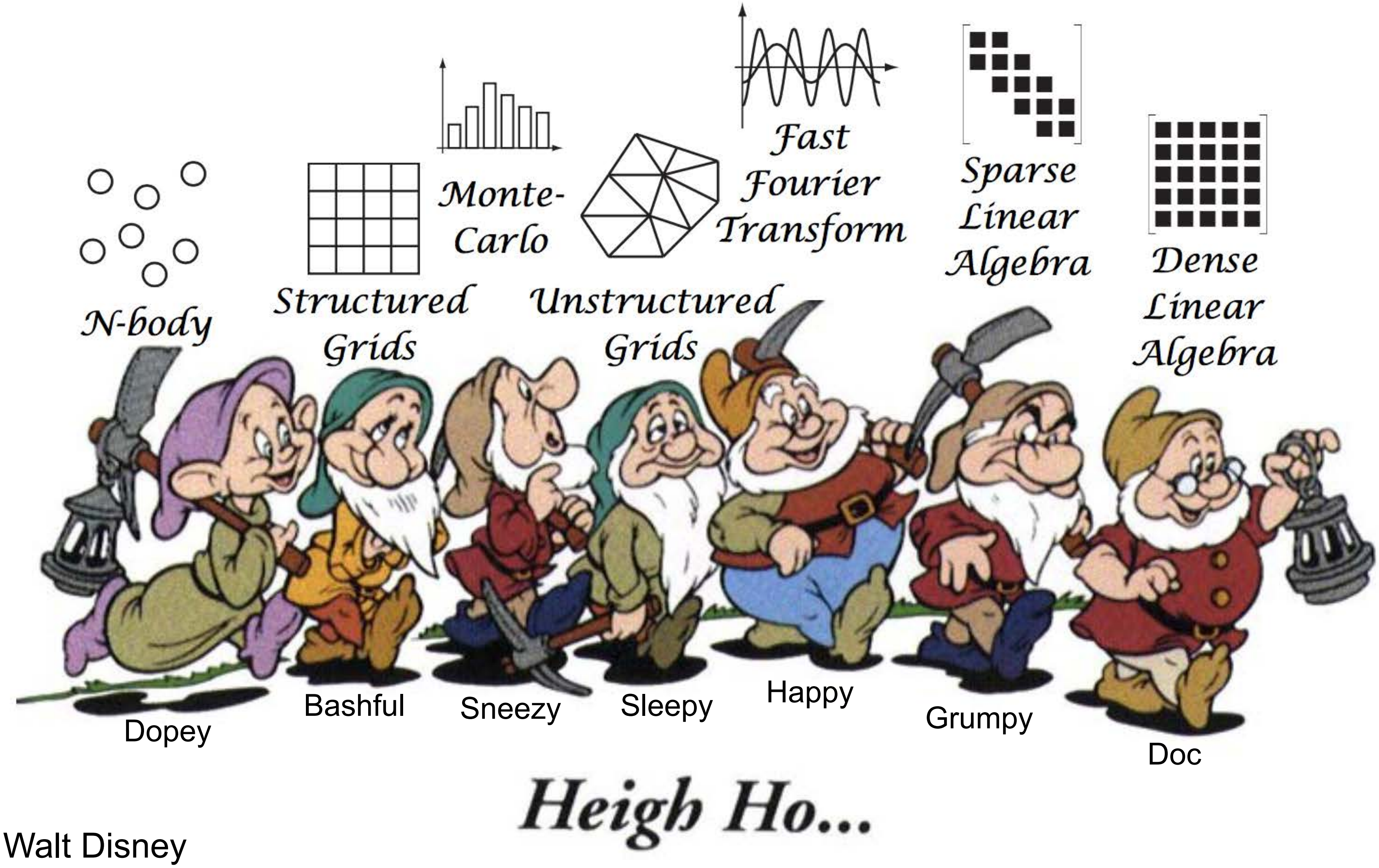


- 1946 — The Monte Carlo method.
- 1947 — Simplex Method for Linear Programming.
- 1950 — Krylov Subspace Iteration Method.
- 1951 — The Decompositional Approach to Matrix Computations.
- 1957 — The Fortran Compiler.
- 1959 — QR Algorithm for Computing Eigenvalues.
- 1962 — Quicksort Algorithms for Sorting.
- 1965 — Fast Fourier Transform.
- 1977 — Integer Relation Detection.
- 1987 — Fast Multipole Method.

The DATA MINING Top 10

- 1951 – *k* Nearest Neighbor Classification (*k*NN)
- 1957 – *K*-means Clustering
- 1961 – Decision Trees
- 1977 – Expectation Maximization
- 1984 – Classification and Regression Trees (CART)
- 1994 – Apriori Algorithm
- 1995 – Support Vector Machines
- 1997 – Naïve Bayes
- 1997 – AdaBoost
- 1998 – PageRank

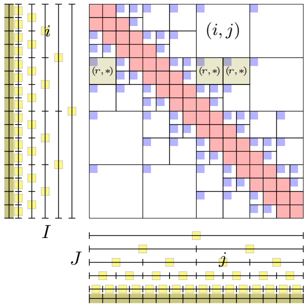
Rio Yokota's interpretation of the seven dwarves



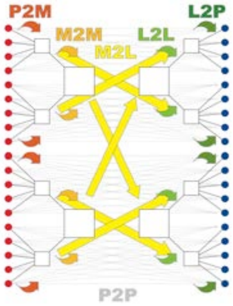
c/o Walt Disney

The Scalable Hierarchical 5

- 1960's – Fast Fourier Transform
- 1970's – Multigrid Methods
- 1980's – Fast Multipole Methods
- 1990's – Sparse Grid Methods
- 2000's – Hierarchically Low-Rank Matrix Methods



2016 KAUST Workshop
on Scalable Hierarchical Algorithms
for Extreme Computing (SHAXC'16)

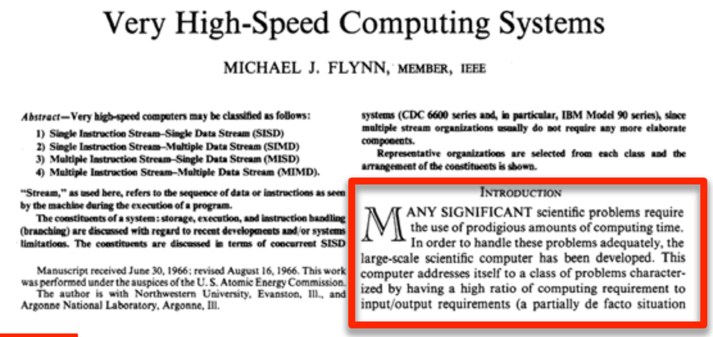


High performance implementations

- All high performance implementations of these algorithms will exploit parallelism
- A goal of the course will be to classify parallel approaches and associate them with
  - Hardware that can support them
  - Algorithms that can benefit from them
  - Software that can implement them
- A successful student
  - builds categories
  - populates them with examples
  - connects new material to an expanding reference set
- To formulate parallel programming categories, we look back 50 years to …

Flynn’s taxonomy (1966) of parallel architectures

Many significant scientific problems require the use of prodigious amounts of computing time. In order to handle these problems adequately, the large-scale scientific computer has been developed. This computer addresses itself to a class of problems characterized by having a high ratio of computing requirement to input/output requirements (a partially *de facto* situation caused by the unavailability of matching input/output equipment). The complexity of these processors, coupled with the advancement of the state of the computing art they represent has focused attention on scientific computers. Insight thus gained is frequently a predictor of computer developments on a more universal basis. This paper [reviews] possible organizations starting with “concurrent” organizations presently in operation and examining other theoretical possibilities.



Very High-Speed Computing Systems

MICHAEL J. FLYNN, MEMBER, IEEE

Abstract—Very high-speed computers may be classified as follows:

1) Single Instruction Stream–Single Data Stream (SISD)
2) Single Instruction Stream–Multiple Data Stream (SIMD)
3) Multiple Instruction Stream–Single Data Stream (MISD)
4) Multiple Instruction Stream–Multiple Data Stream (MIMD).

“Stream,” as used here, refers to the sequence of data or instructions as seen by the machine during the execution of a program.

The constituents of a system: storage, execution, and instruction handling (branching) are discussed with regard to recent developments and/or systems limitations. The constituents are discussed in terms of concurrent SISD systems (CDC 6600 series and, in particular, IBM Model 90 series), since multiple stream organizations usually do not require any more elaborate components.

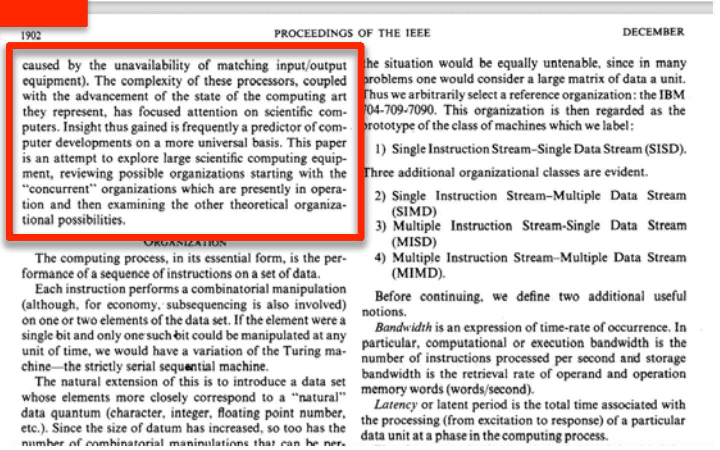
Representative organizations are selected from each class and the arrangement of the constituents is shown.

Manuscript received June 30, 1966; revised August 16, 1966. This work was performed under the auspices of the U. S. Atomic Energy Commission.

The author is with Northwestern University, Evanston, Ill., and Argonne National Laboratory, Argonne, Ill.

INTRODUCTION

MANY SIGNIFICANT scientific problems require the use of prodigious amounts of computing time. In order to handle these problems adequately, the large-scale scientific computer has been developed. This computer addresses itself to a class of problems characterized by having a high ratio of computing requirement to input/output requirements (a partially de facto situation



1902 PROCEEDINGS OF THE IEEE DECEMBER

caused by the unavailability of matching input/output equipment). The complexity of these processors, coupled with the advancement of the state of the computing art they represent, has focused attention on scientific computers. Insight thus gained is frequently a predictor of computer developments on a more universal basis. This paper is an attempt to explore large scientific computing equipment, reviewing possible organizations starting with the “concurrent” organizations which are presently in operation and then examining the other theoretical organizational possibilities.

ORGANIZATION

The computing process, in its essential form, is the performance of a sequence of instructions on a set of data.

Each instruction performs a combinatorial manipulation (although, for economy, subsequencing is also involved) on one or two elements of the data set. If the element were a single bit and only one such bit could be manipulated at any unit of time, we would have a variation of the Turing machine—the strictly serial sequential machine.

The natural extension of this is to introduce a data set whose elements more closely correspond to a “natural” data quantum (character, integer, floating point number, etc.). Since the size of datum has increased, so too has the number of combinatorial manipulations that can be per-

the situation would be equally untenable, since in many problems one would consider a large matrix of data a unit. Thus we arbitrarily select a reference organization: the IBM 704-709-7090. This organization is then regarded as the prototype of the class of machines which we label:

1) Single Instruction Stream–Single Data Stream (SISD).

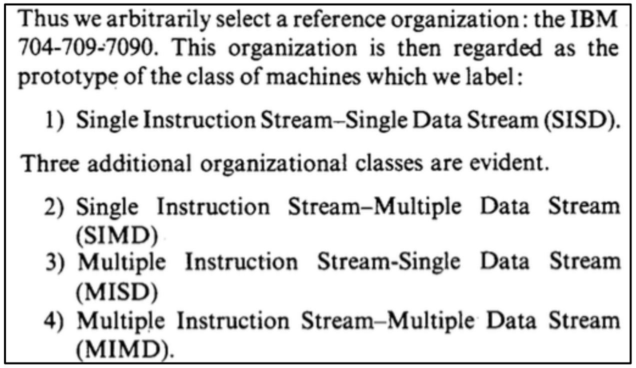
Three additional organizational classes are evident.

2) Single Instruction Stream–Multiple Data Stream (SIMD)
3) Multiple Instruction Stream-Single Data Stream (MISD)
4) Multiple Instruction Stream–Multiple Data Stream (MIMD).

Before continuing, we define two additional useful notions.

Bandwidth is an expression of time-rate of occurrence. In particular, computational or execution bandwidth is the number of instructions processed per second and storage bandwidth is the retrieval rate of operand and operation memory words (words/second).

Latency or latent period is the total time associated with the processing (from excitation to response) of a particular data unit at a phase in the computing process.

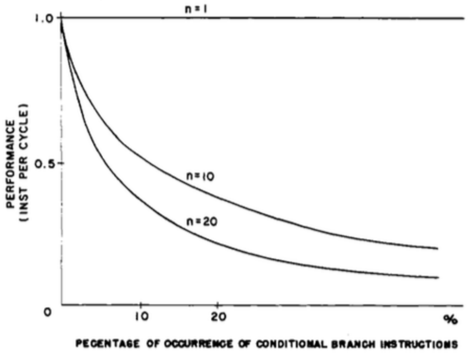


Thus we arbitrarily select a reference organization: the IBM 704-709-7090. This organization is then regarded as the prototype of the class of machines which we label:

1) Single Instruction Stream–Single Data Stream (SISD).

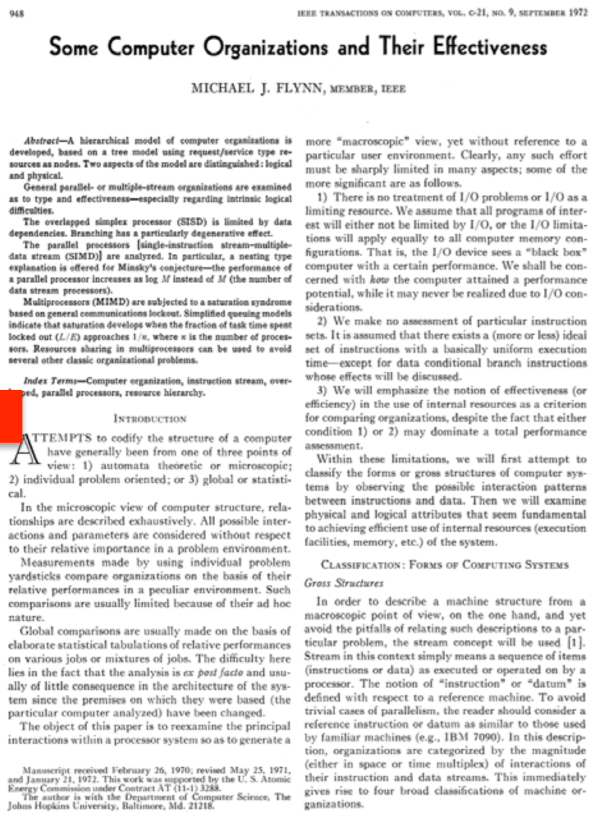
Three additional organizational classes are evident.

2) Single Instruction Stream–Multiple Data Stream (SIMD)
3) Multiple Instruction Stream-Single Data Stream (MISD)
4) Multiple Instruction Stream–Multiple Data Stream (MIMD).



Flynn’s analysis (1972) of architectural effectiveness

The historic view of parallelism […] is probably best represented by Amdahl […] that certain operations […] must be done in an absolutely sequential basis. These operations include, for example, the ordinary housekeeping operations in a program. In order to achieve any effectiveness at all […] parallel organizations processing *N* streams must have substantially less than 1 / *N* x 100 percent of absolutely sequential instruction segments. One can then proceed to show that typically for large *N* this does not exist in conventional programs. A major difficulty with this analysis lies in the [implication…] that what exists today in the way of programming procedures and algorithms must also exist in the future.



948 IEEE TRANSACTIONS ON COMPUTERS, VOL. C-21, NO. 9, SEPTEMBER 1972

Some Computer Organizations and Their Effectiveness

MICHAEL J. FLYNN, MEMBER, IEEE

Abstract—A hierarchical model of computer organizations is developed, based on a tree model using request/service type resources as nodes. Two aspects of the model are distinguished: logical and physical.

General parallel- or multiple-stream organizations are examined as to type and effectiveness—especially regarding intrinsic logical difficulties.

The overlapped simplex processor (SISD) is limited by data dependencies. Branching has a particularly degenerative effect.

The parallel processors [single-instruction stream–multiple-data stream (SIMD)] are analyzed. In particular, a nesting type explanation is offered for Minsky's conjecture—the performance of a parallel processor increases as log *M* instead of *M* (the number of data stream processors).

Multiprocessors (MIMD) are subjected to a saturation syndrome based on general communications lockout. Simplified queuing models indicate that saturation develops when the fraction of task time spent locked out (L/E) approaches $1/n$, where n is the number of processors. Resources sharing in multiprocessors can be used to avoid several other classic organizational problems.

Index Terms—Computer organization, instruction stream, overlapped, parallel processors, resource hierarchy.

Introduction

ATTEMPTS to codify the structure of a computer have generally been from one of three points of view: 1) automata theoretic or microscopic; 2) individual problem oriented; or 3) global or statistical.

In the microscopic view of computer structure, relationships are described exhaustively. All possible interactions and parameters are considered without respect to their relative importance in a problem environment.

Measurements made by using individual problem yardsticks compare organizations on the basis of their relative performances in a peculiar environment. Such comparisons are usually limited because of their ad hoc nature.

Global comparisons are usually made on the basis of elaborate statistical tabulations of relative performances on various jobs or mixtures of jobs. The difficulty here lies in the fact that the analysis is *ex post facto* and usually of little consequence in the architecture of the system since the premises on which they were based (the particular computer analyzed) have been changed.

The object of this paper is to reexamine the principal interactions within a processor system so as to generate a more "macroscopic" view, yet without reference to a particular user environment. Clearly, any such effort must be sharply limited in many aspects; some of the more significant are as follows.

1) There is no treatment of I/O problems or I/O as a limiting resource. We assume that all programs of interest will either not be limited by I/O, or the I/O limitations will apply equally to all computer memory configurations. That is, the I/O device sees a "black box" computer with a certain performance. We shall be concerned with *how* the computer attained a performance potential, while it may never be realized due to I/O considerations.

2) We make no assessment of particular instruction sets. It is assumed that there exists a (more or less) ideal set of instructions with a basically uniform execution time—except for data conditional branch instructions whose effects will be discussed.

3) We will emphasize the notion of effectiveness (or efficiency) in the use of internal resources as a criterion for comparing organizations, despite the fact that either condition 1) or 2) may dominate a total performance assessment.

Within these limitations, we will first attempt to classify the forms or gross structures of computer systems by observing the possible interaction patterns between instructions and data. Then we will examine physical and logical attributes that seem fundamental to achieving efficient use of internal resources (execution facilities, memory, etc.) of the system.

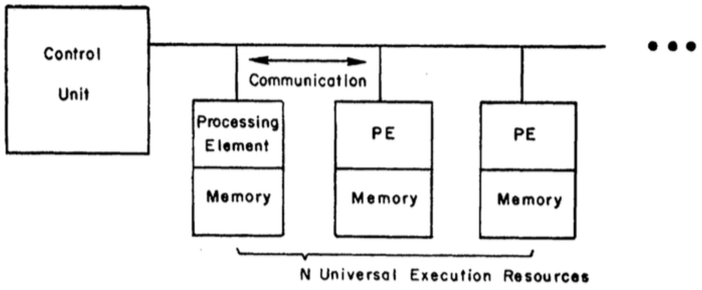
Classification: Forms of Computing Systems

Gross Structures

In order to describe a machine structure from a macroscopic point of view, on the one hand, and yet avoid the pitfalls of relating such descriptions to a particular problem, the stream concept will be used [1]. Stream in this context simply means a sequence of items (instructions or data) as executed or operated on by a processor. The notion of "instruction" or "datum" is defined with respect to a reference machine. To avoid trivial cases of parallelism, the reader should consider a reference instruction or datum as similar to those used by familiar machines (e.g., IBM 7090). In this description, organizations are categorized by the magnitude (either in space or time multiplex) of interactions of their instruction and data streams. This immediately gives rise to four broad classifications of machine organizations.

Manuscript received February 26, 1970; revised May 25, 1971, and January 21, 1972. This work was supported by the U. S. Atomic Energy Commission under Contract AT (11-1) 3288.

The author is with the Department of Computer Science, The Johns Hopkins University, Baltimore, Md. 21218.



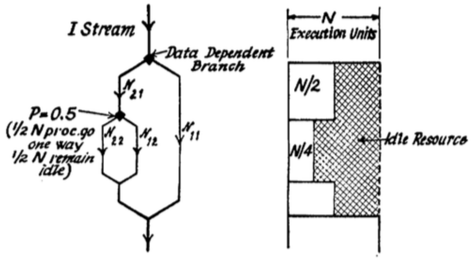
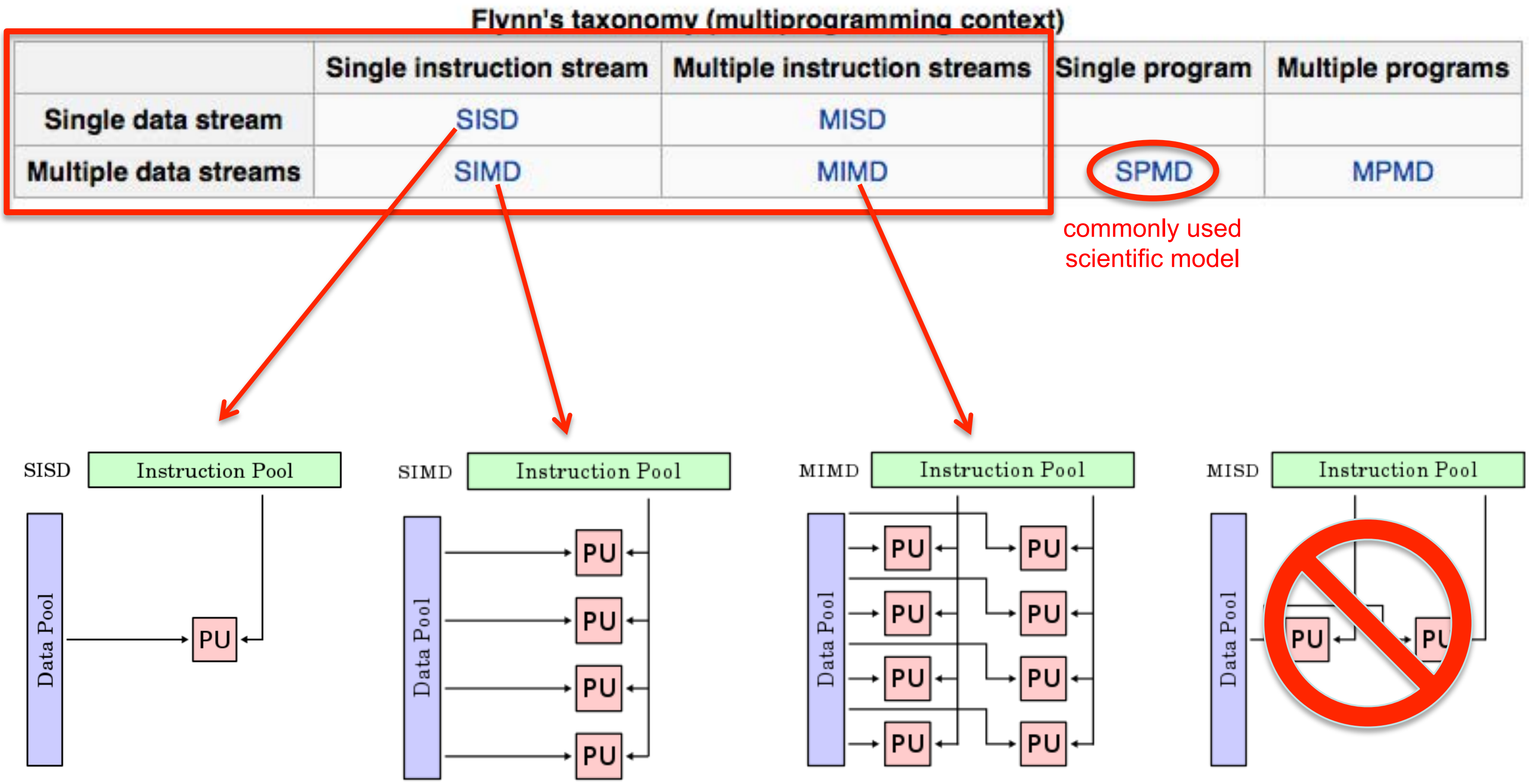


Fig. 7. SIMD branching.

Flynn’s hardware taxonomy



Flynn's taxonomy (multiprogramming context)

	Single instruction stream	Multiple instruction streams	Single program	Multiple programs
Single data stream	SISD	MISD		
Multiple data streams	SIMD	MIMD	SPMD	MPMD

commonly used scientific model

Single Program, Multiple Data (SPMD) is a natural generalization of Single Instruction, Multiple Data (SIMD) when each processing unit executes its own local copy of the instruction stream. These copies can branch differently depending upon the data.

c/o The Wikipedia

7 parallel algorithm paradigms

Paradigm	Examples
Embarrassingly Parallel	Monte Carlo
Fork-Join	OpenMP parallel for-loop
Manager-Worker	Genetic Algorithm
Divide-Conquer-Combine	Parallel Sort
Halo exchange	Stencil-based PDE solvers (FD, FV, FE)
Permutation	Matrix-matrix multiplication (Cannon’s algorithm)
Task Dataflow	Breadth-first Search

c/o Thomas Sterling

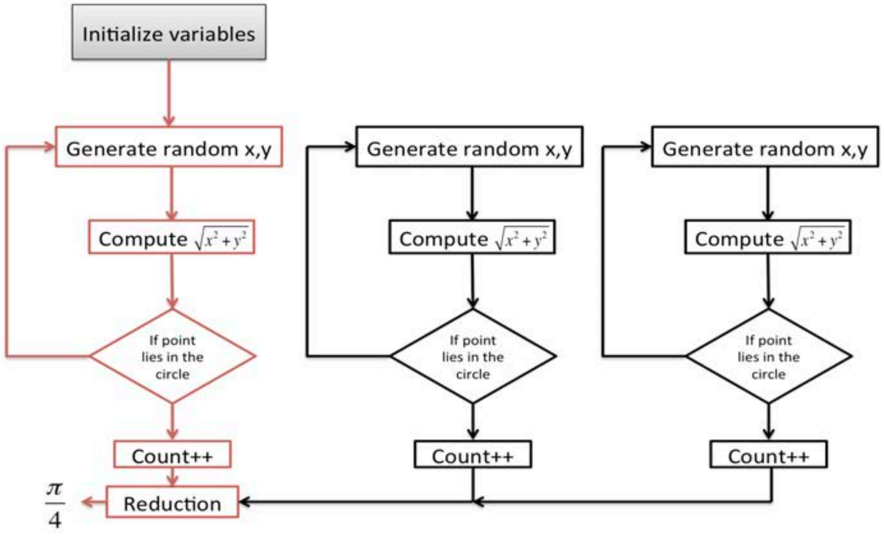
Embarrassingly Parallel

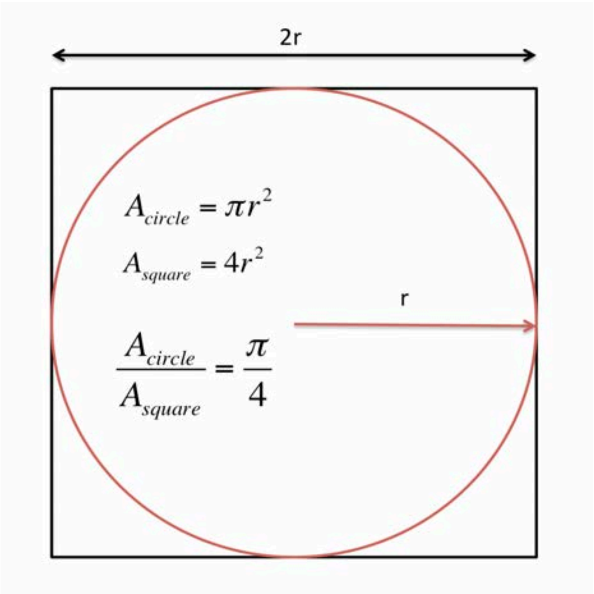
- A problem that requires little or no effort to identify and launch many concurrent tasks, and in which there is no internal synchronization or other communication or load balancing concern, is called "embarrassingly parallel"
  - The term dates to 1986, in an article by Cleve Moler (founder of MATLAB)
  - "Pleasingly parallel" is a popular alternative
- The classic such algorithm is Monte Carlo (1946), which is the first of the "Top 10" algorithms of the last century
  - Monte Carlo also has a popular alternative: the Metropolis Algorithm, named for Nicholas Metropolis who used it in early computations of the Ising Model of phase change in 1952

Embarrassingly Parallel example:

Monte Carlo

- Monte Carlo is a statistical approach for studying systems with a large number of degrees of freedom, such as neutron transport (for which it was first described by Von Neumann in 1947)
- It is the algorithm of choice for integration in high dimensions but it can be demonstrated in low, here two





Embarrassingly Parallel example:

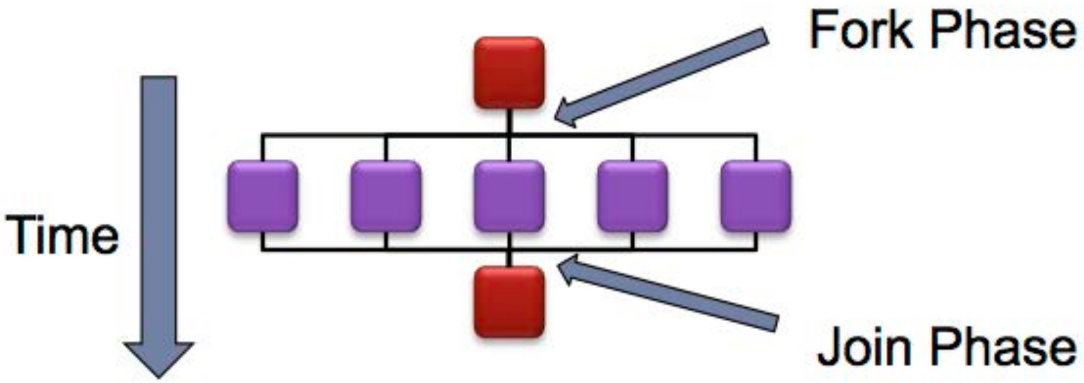
Monte Carlo

- Monte Carlo is arbitrarily parallelizable, but agonizingly slow
  - convergence rate is proportional to $N^{-1/2}$
    - increasing work (here the number of samples) from 10 to 1,000 gives only a reduction of error of about a factor of 10
  - ideally, we would like convergence rate like N^{-2}
    - increasing work from 10 to 1,000 would reduce error by a factor of 10,000
    - we are accustomed much better rates in the quadrature of continuous functions
- In high dimensions, when measured in terms of function evaluations, Monte Carlo can be the method of choice, because of the curse of dimensionality, but beware of an indictment attributed to Von Neumann: *“Anyone using Monte Carlo is in a state of sin.”*

Monte Carlo advances

- Markov Chain Monte Carlo (MCMC) is a form of Monte Carlo that chooses smartly conditioned random samples
  - not statistically independent, but correlated for efficient “mixing”
  - includes: Metropolis-Hastings (1970), Gibbs sampling (1984), and many others that restore some respectability to MC ☺
- Multi-level Monte Carlo (MLMC) is a form of Monte Carlo that evaluates its samples on a hierarchy of models, coarsened from the model for which the solution is sought
  - most of the work is done on the coarser models, which are often exponentially cheaper, while obtaining the accuracy of the finest
  - invented by Mike Giles (2008) and very hot presently
- Of course, there is also MLMCMC, the methods MCMC and MLMC being composable
- Quantum Monte Carlo (QMC) is an application to quantum many-body systems – many computer codes

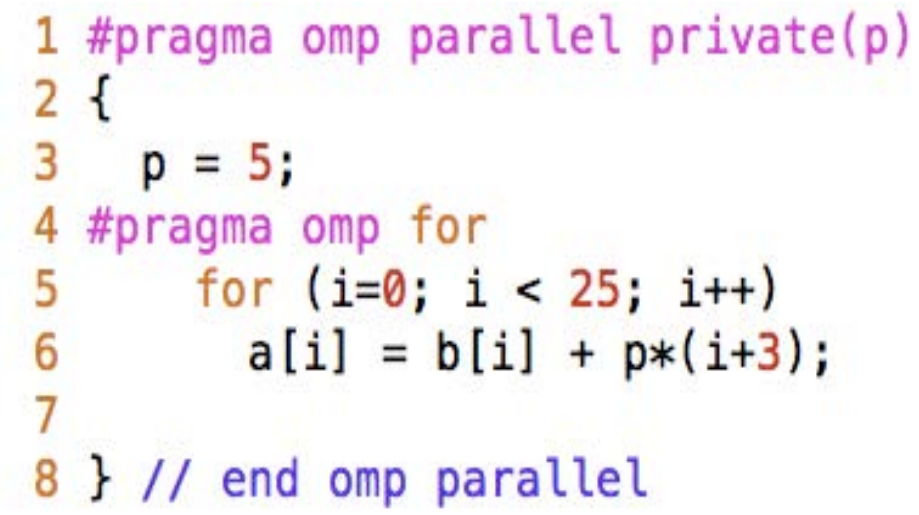
Fork-Join



- Launches parallel threads within an overall serial context
- Exploits concurrency independent tasks that become available in batches
- May incur overhead of creation and termination of the threads
- Presumes that the concurrent tasks require similar time to completion

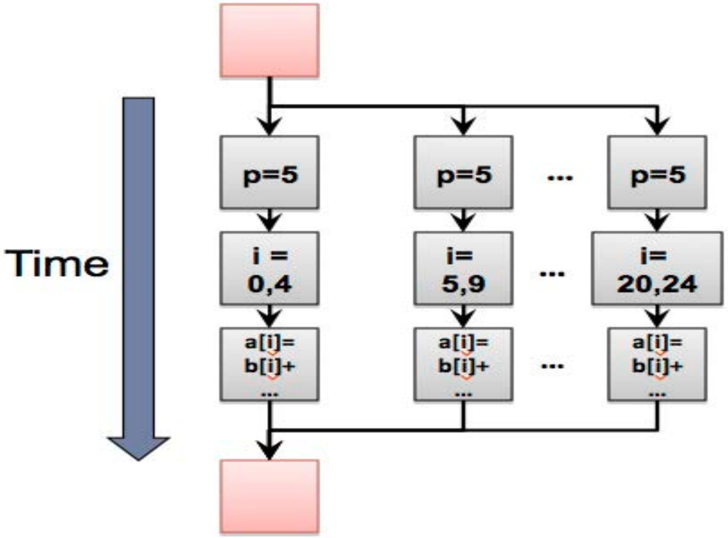
Fork-Join example:

OpenMP loop



```
#pragma omp parallel private(p)
{
  p = 5;
#pragma omp for
    for (i=0; i < 25; i++)
      a[i] = b[i] + p*(i+3);

} // end omp parallel
```

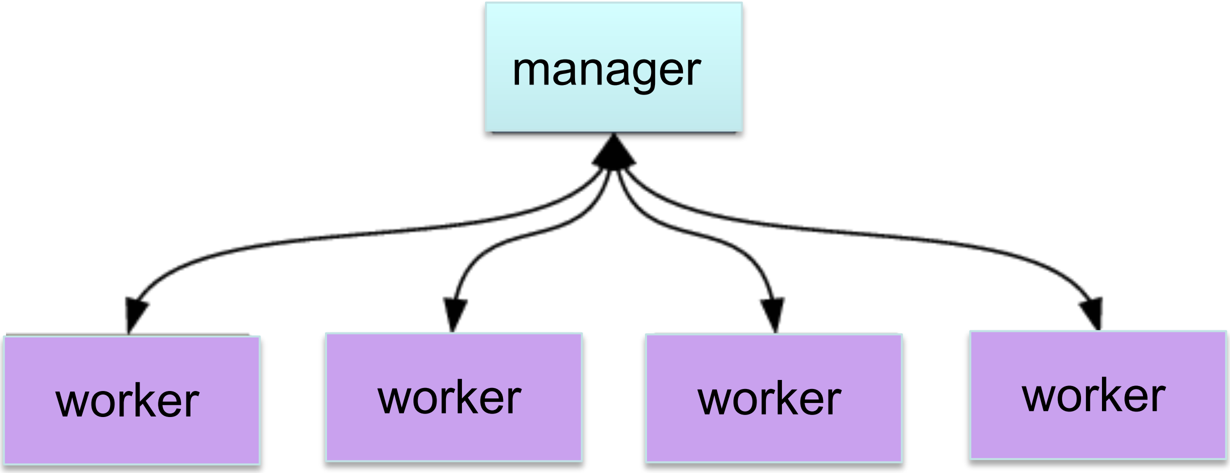


- Code expresses 25 independent ops in for-loop
- OpenMP pragma directs the compiler to batch them concurrently
- Storage of a[ ] and b[ ] is shared – visible and writeable to all threads of execution
- For OpenMP standard (and training) see `openmp.org`

Aside

- This is not a course in parallel *programming*
- Acquisition of skill in parallel programming (using MPI, OpenMP, CUDA, etc.) is encouraged
  - short courses are available at many conferences, at many universities, and online
  - rich literature, excellent open source demos
- Programming produces insight and intuition hard to obtain from other investments of time
- Fortunately, the maturity of parallel computing today allows many computational scientists to be productive at a higher level
  - e.g., working within an API like PETSc, Trilinos, …

Manager-Worker



- This paradigm is classically known as “master-slave”
- Manager assigns individual tasks to workers, receives the responses and dynamically constructs and assigns new tasks until work is complete
- Unlike Fork-Join, it does not presume that the tasks are well balanced, and it is not pre-scheduled
- Workers do not synchronize or exchange among selves

Manager-Worker example:

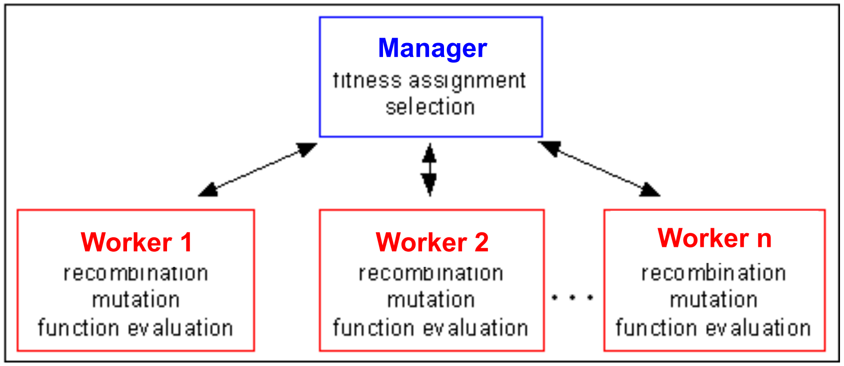
Global parallel genetic algorithm

- Genetic algorithms are efficient stochastic search methods based on principles of natural selection and genetics
- They find an optimum by manipulating a population of candidate solutions against a fitness function
- Good candidates are preferentially selected to mate (combine their traits by crossover) and reproduce
- Many variations exist, including mutations outside of the parents
- The population can allow global exchanges or can be confined to different islands

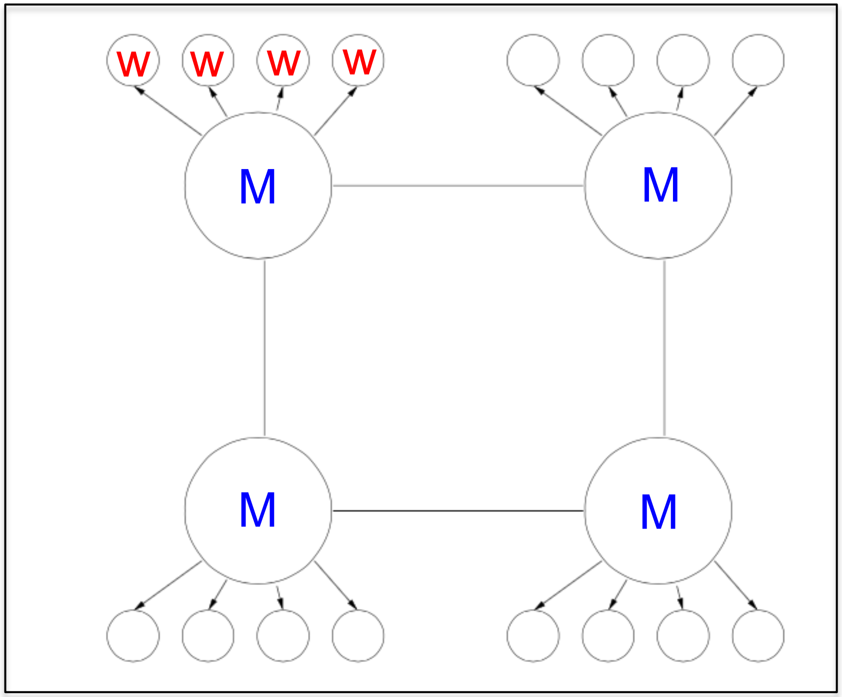
Manager-Worker example:

Global parallel genetic algorithm

- Simple manager-worker

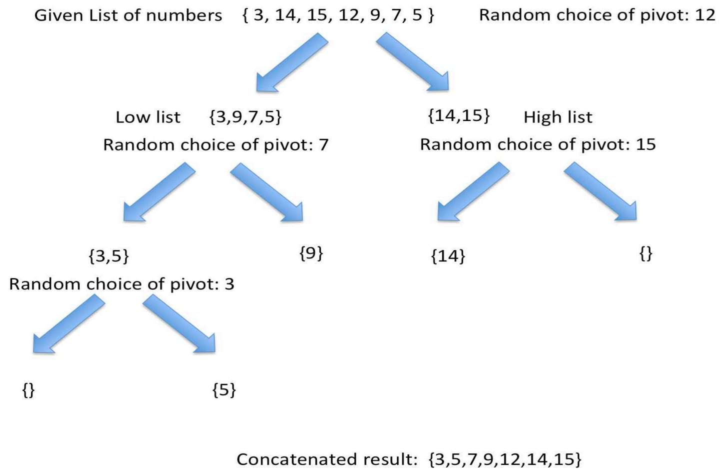


- Hierarchical model



c/o Eric Cantu-Paz

Divide-Conquer-Combine


Given List of numbers { 3, 14, 15, 12, 9, 7, 5 }
Random choice of pivot: 12
Low list {3,9,7,5}
Random choice of pivot: 7
{14,15} High list
Random choice of pivot: 15
{3,5}
Random choice of pivot: 3
{9}
{14}
{}
{}
{5}
Concatenated result: {3,5,7,9,12,14,15}

1. Given a "large" instance of a problem, partition it into two or more (independent) subproblems
2. Solve the subproblems (concurrently)
3. Merge the subsolutions into a solution of the original

Recur if step 2 is still "large"

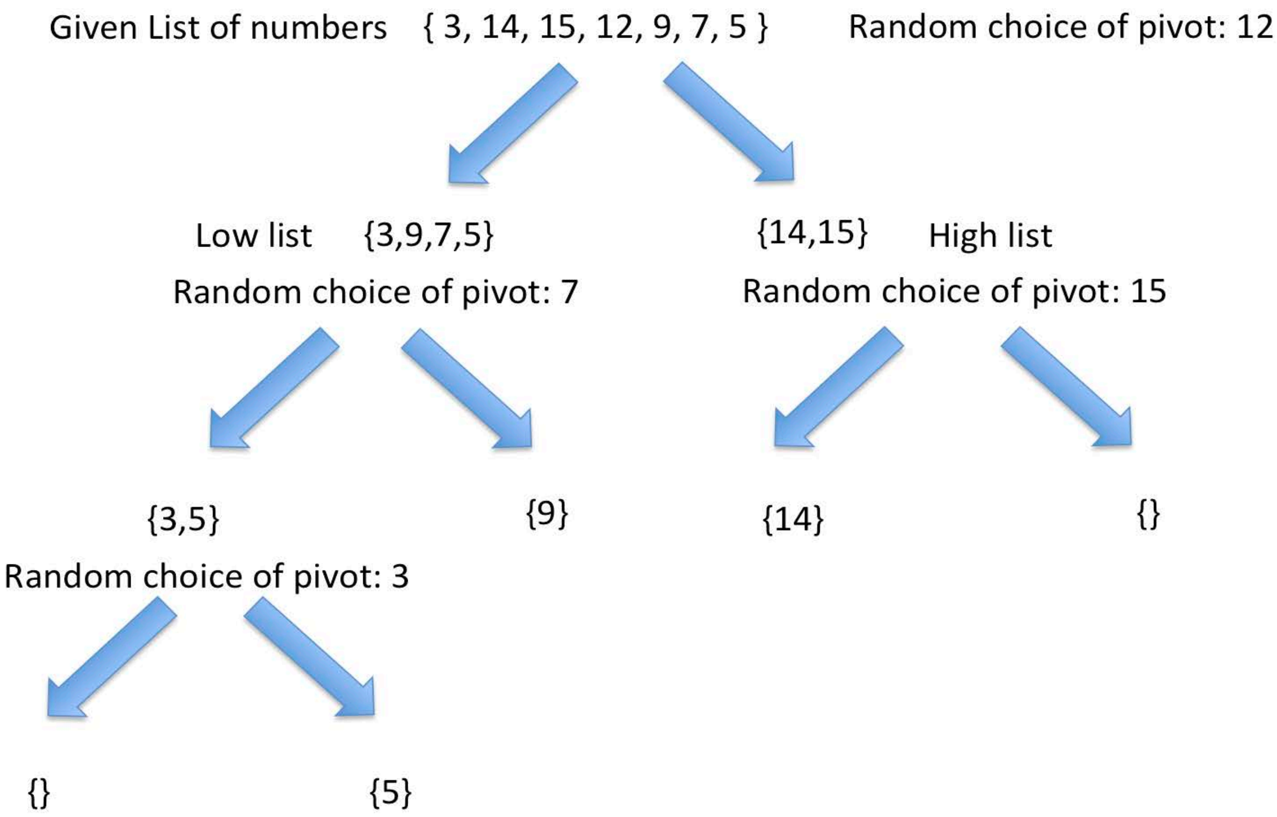
Divide-Conquer-Combine example:

Quicksort

- Named one of the “Top 10” algorithms of the last century
  - See introductory article by Joseph Jaja
- Invented in 1962 by Tony Hoare (born 1934, Turing Prize winner for definition and design of programming languages, long list of professional honors)
- Important not just as an early paradigm for parallel “divide-and-conquer”, but also as an example of a randomized algorithm
  - randomized algorithms are extremely hot today, in both discrete and continuous forms
- Long history of complexity analysis for different cases of algorithm choices and input conditions
- Worst case running time is $O(N^2)$ for input size N, but with probability $1\text{-}N^{-c}$, it is $O(N \log N)$ – an “optimal” algorithm

Divide-Conquer-Combine example:

Quicksort



Given List of numbers { 3, 14, 15, 12, 9, 7, 5 } Random choice of pivot: 12

Low list {3,9,7,5}

Random choice of pivot: 7

{14,15} High list

Random choice of pivot: 15

{3,5}

Random choice of pivot: 3

{9}

{14}

{}

{}

{5}

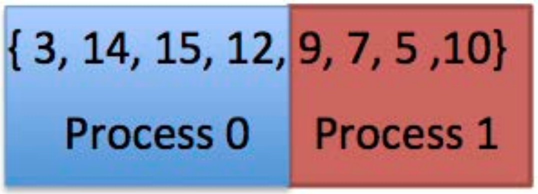
Concatenated result: {3,5,7,9,12,14,15}

The numbers represent a orderable set of keys in records to be sorted

Divide-Conquer-Combine example:

Parallel sample quicksort

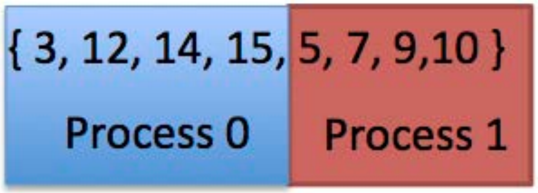
As evenly as possible distribute N keys among P processors (here $N=8$ and $P=2$)



{ 3, 14, 15, 12, 9, 7, 5 ,10}

Process 0 Process 1

Run sequential quicksort on local data

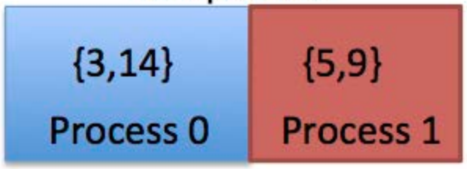


{ 3, 12, 14, 15, 5, 7, 9,10 }

Process 0 Process 1

Sample the sorted sublists at every N/P^2 locations

Sample Set



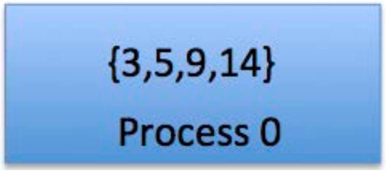
{3,14} {5,9}

Process 0 Process 1

Divide-Conquer-Combine example:

Parallel sample quicksort

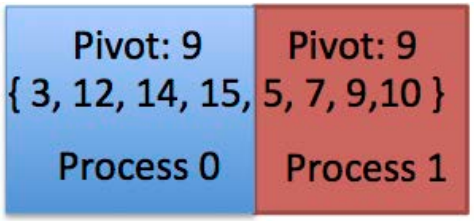
Gather samples to the root (P=0) and run sequential quicksort



{3,5,9,14}

Process 0

Broadcast P-1 pivot values to all processes



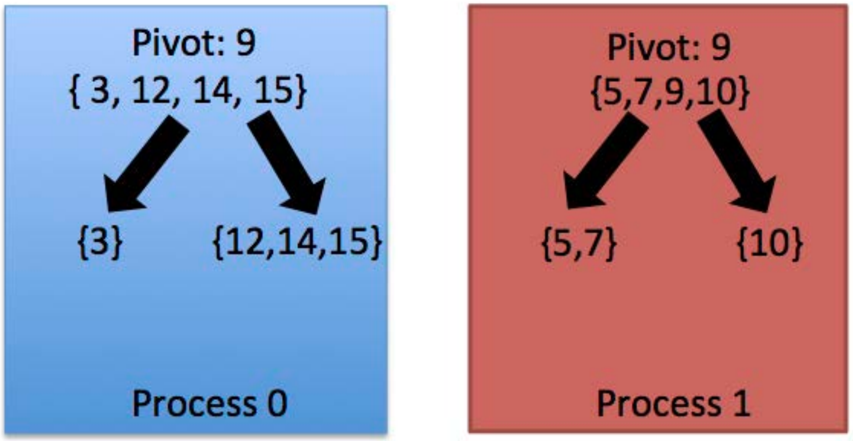
Pivot: 9 { 3, 12, 14, 15,

Process 0

Pivot: 9 5, 7, 9,10 }

Process 1

Divide local sorted segments on each process into P segments based on pivots



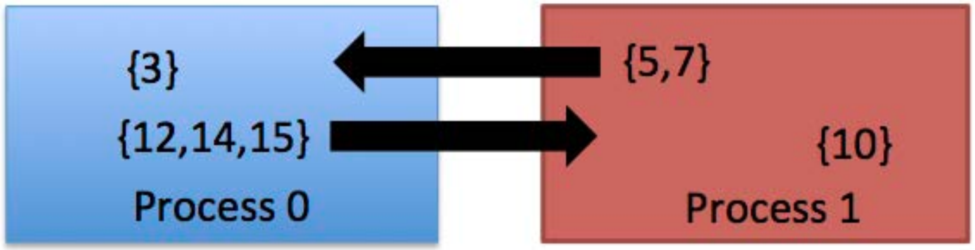
Pivot: 9
{ 3, 12, 14, 15}
{3} {12,14,15}
Process 0

Pivot: 9
{5,7,9,10}
{5,7} {10}
Process 1

Divide-Conquer-Combine example:

Parallel sample quicksort

Perform an all-to-all communication on the P segments, with P_i keeping the i^{th} segment and sending the j^{th} segment to P_j



{3}

{12,14,15}

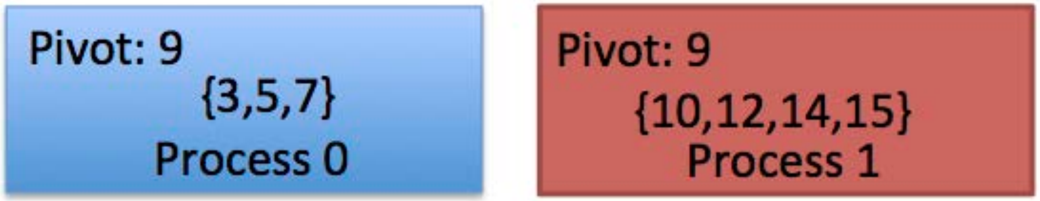
Process 0

{5,7}

{10}

Process 1

Merge incoming (already sorted) sub-segments into local lists



Pivot: 9

{3,5,7}

Process 0

Pivot: 9

{10,12,14,15}

Process 1

Final result: {3,5,7,9,10,12,14,15}

The final list of keys is distributed, which is probably what is desired if N is large compared to what a single processor can store in the first place

The rest of the record volume associated with each sorted key can be moved into place

Divide-Conquer-Combine example:

Quicksort

- After 54 years, quicksort remains the sorting routine of choice for large N unless there exists additional detailed information about the input
- Other means of selecting the pivot exist
  - Clearly selecting the median is best
  - Normally, there is no fast way to find the median
  - Fixed position works poorly unless the input is random
  - Finding the median of a randomly chosen subset may be superior to a single random choice
- Complexity can be measured in individual comparison operations, or can incorporate memory and communication costs

Halo Exchange

- Many computations, especially partial differential equations, are executed by SPMD programs in which each process advances at least one subdomain of the overall PDE domain, and every subdomain is assigned to at least one process (usually a 1-to-1 mapping)
- By their mathematical nature, PDE codes discretize derivatives, which are approximated with local “stencils”
- Partitioning into subdomains requires replication of the points of a stencil that belong to a neighboring subdomains, the replicated region called a “halo”
- Parallel implementation requires (usually symmetrical) nearest-neighbor communication

Halo Exchange example:
2nd-order Laplacian in 2D

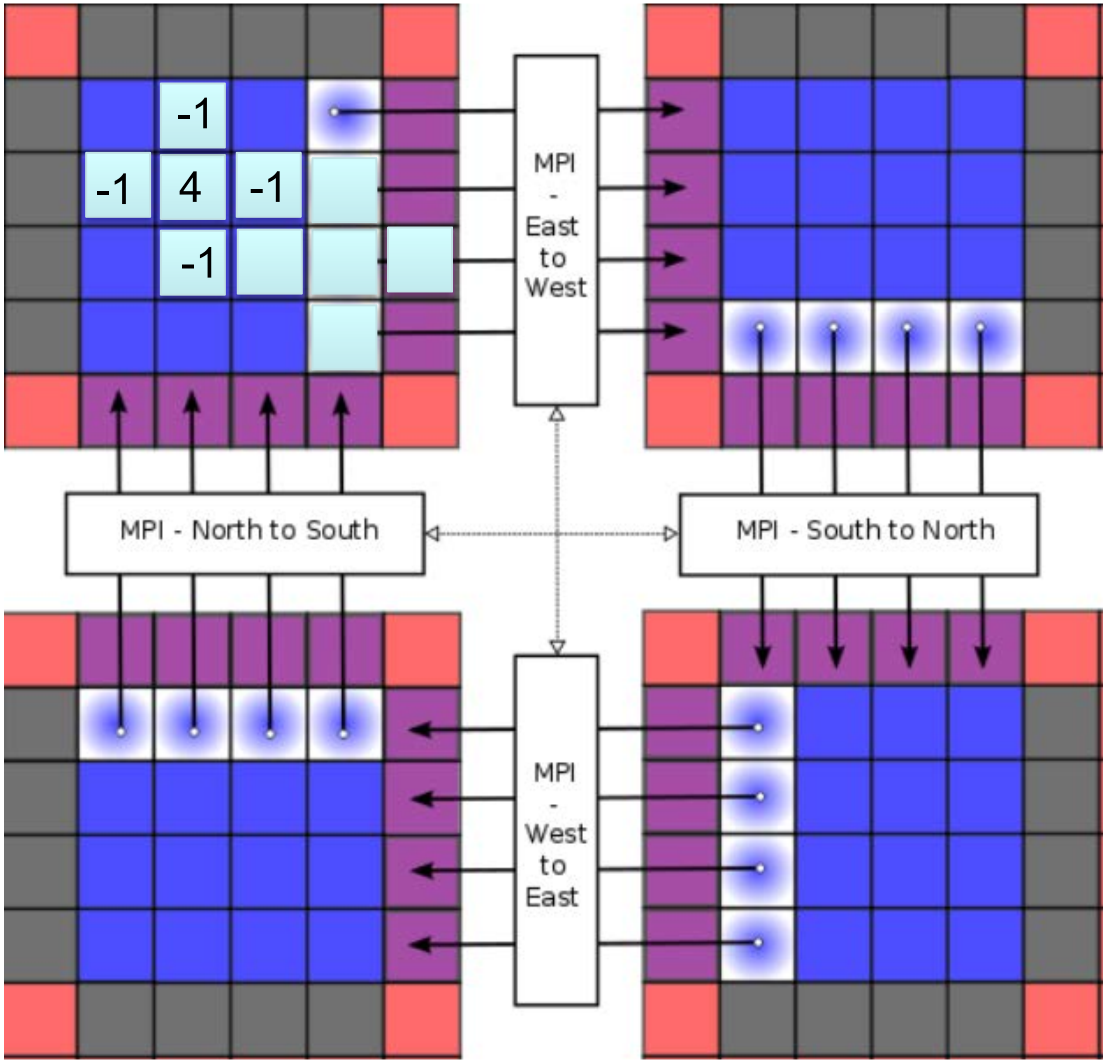
- Halo exchange is regular and frequent and is built into communication subroutines, e.g., in the standard Message-Passing Interface, MPI

🟦 owned cell

🟪 halo exchange cell

⬛ ghost boundary cell

🟥 unused for 5-pt star



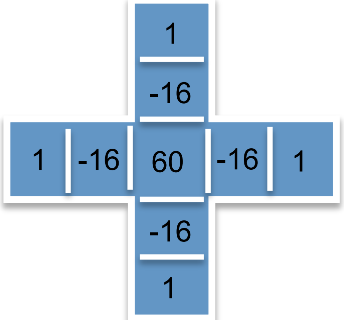
8x8 domain in four 4x4 subdomains, with halos and ghosts

c/o Fabian Dournac

Halo Exchange example:

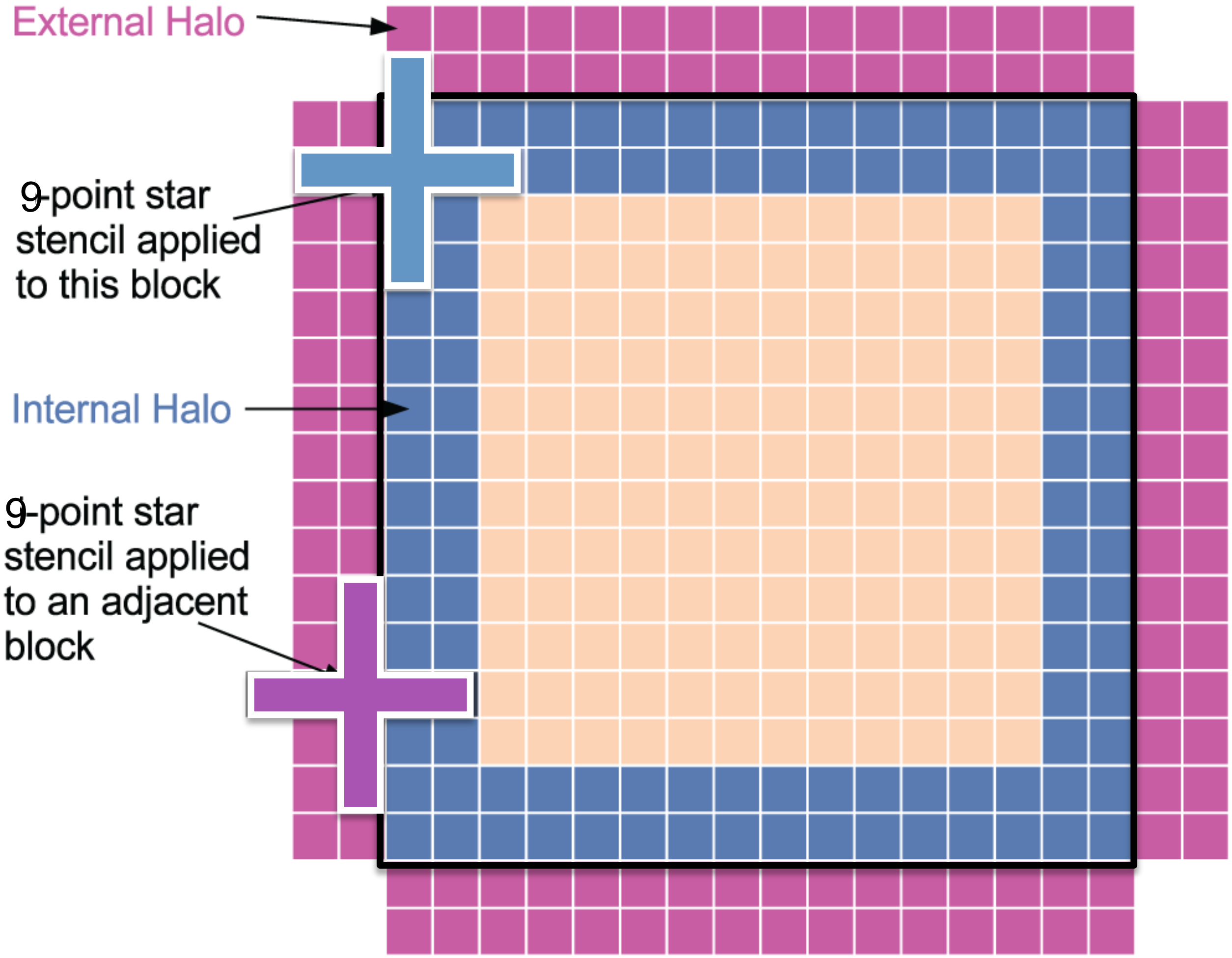
4th-order Laplacian in 2D

- For higher-order discretizations, stencils and halos need to be wider



- Everything in the black square is owned by one interior subdomain (with no boundary points)
- The magenta is a double-wide halo

c/o Michael Flynn


External Halo

9-point star stencil applied to this block

Internal Halo

9-point star stencil applied to an adjacent block

Halo Exchange example:
Sparse matrix-vector multiplication

- Stencil evaluation is a special, structured form of a more general operation, sparse matrix-vector multiplication
- For a matrix of size N x N and vector of size N, matrix-vector multiplication is given by

$$x_i = \sum_{j=0}^{N-1} A_{ij} b_j$$

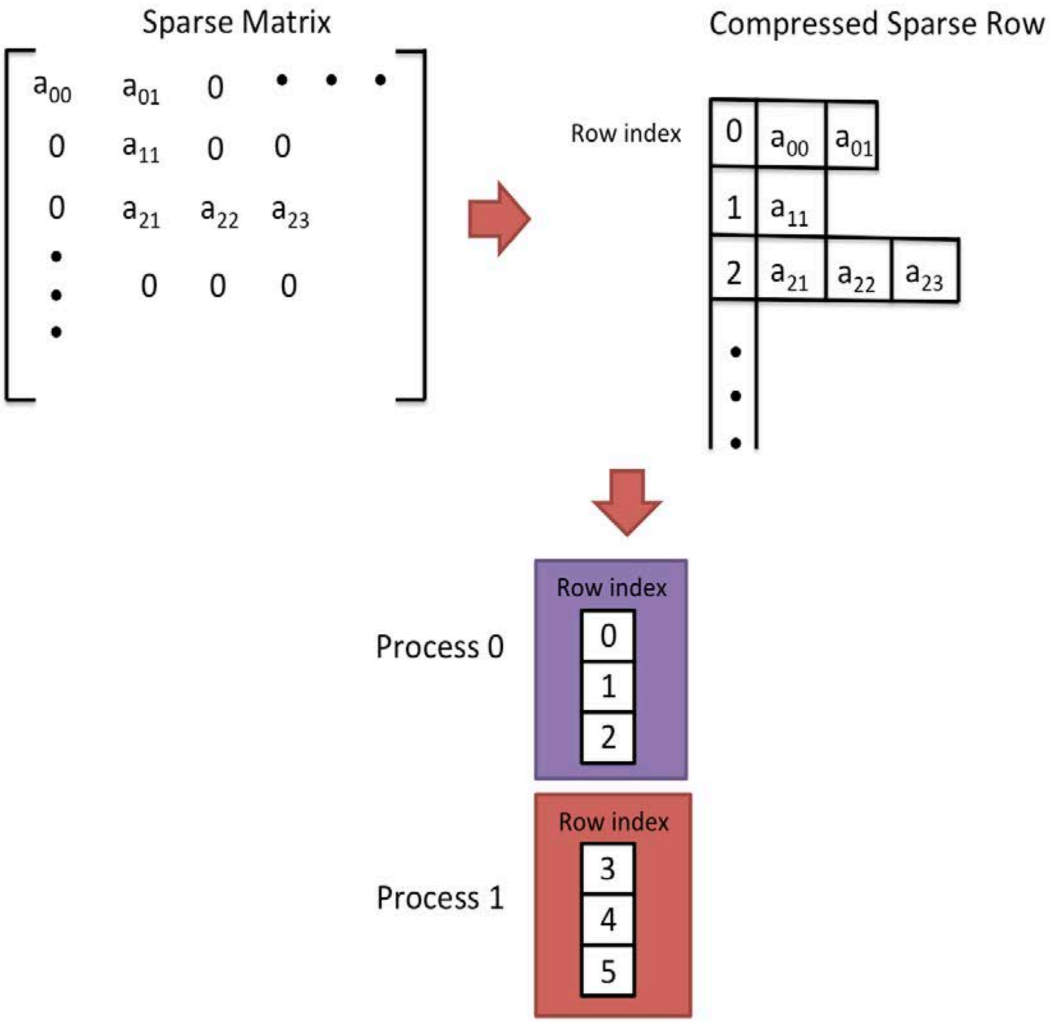
where A_{ij} is the $(i,j)^{th}$ element of the matrix and b_j is the j^{th} element of the vector

- For a sparse matrix, however, most of the A_{ij} values are zero, suggesting that memory should be allocated for every element
  - for 2nd-order Laplacian, $A_{ii} = 4$ and just four off-diagonals are nonzero

Halo Exchange example:

Sparse matrix-vector multiplication

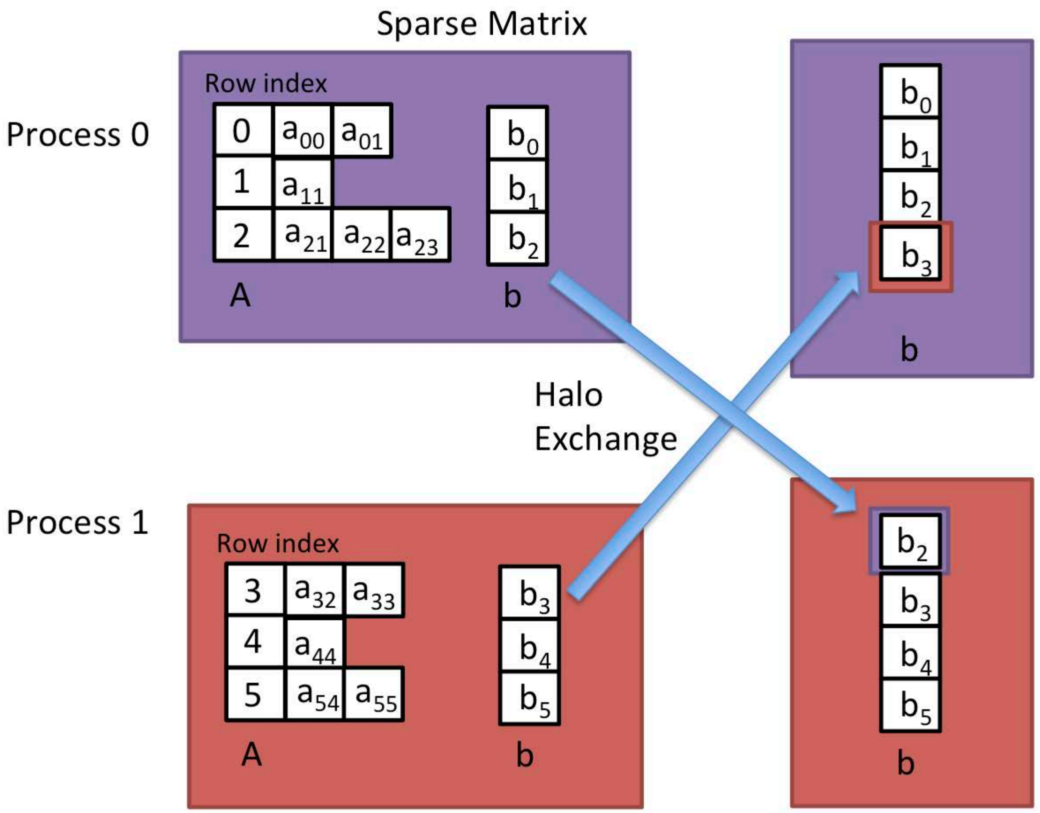
- Sparse matrix A_{ij} can use compressed sparse row (CSR) format
- Then the matrix, the input vector b, and the output vector x can all be partitioned by rows with a block of rows assigned to each processor
- Information from vector b is then potentially nonlocal to the processors computing a local pieces of x

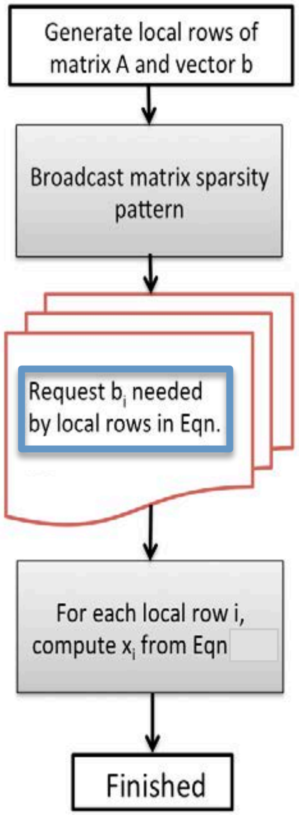


Halo Exchange example:

Sparse matrix-vector multiplication

$$x_i = \sum_{j=0}^{n_i} A_{ij}b_j = \sum_{local} A_{ij}b_j + \overset{\text{halo exchange terms}}{\boxed{\sum_{nonlocal} A_{ij}b_j}}$$




Generate local rows of matrix A and vector b
Broadcast matrix sparsity pattern
Request bi needed by local rows in Eqn.
For each local row i, compute xi from Eqn
Finished

Permutation

- Permutation is a class of data-parallel SPMD algorithms, in which the same local program is simultaneously applied to different data in a series of stages
- Between each stage is an all-to-all permutation of the data
- The Fast Fourier Transform is a famous example, considered later
- Continuing upon matrix-vector multiplication, we consider matrix-matrix multiplication, $C = A$ x B
- Cannon's algorithm (1999) is a special form of dense matrix-matrix multiplication in which two N x N matrices are multiplied by a $\sqrt{P} \times \sqrt{P}$ array of processors, each of which owns a $N/\sqrt{P} \times N/\sqrt{P}$ block of A, B , and C

Permutation example:

Dense matrix-matrix multiplication

- Matrix-matrix multiplication is an $O(N^3)$ operation on $O(N^2)$ data

$$C_{ij} = \sum_{k=0}^{N-1} A_{ik} B_{kj}$$

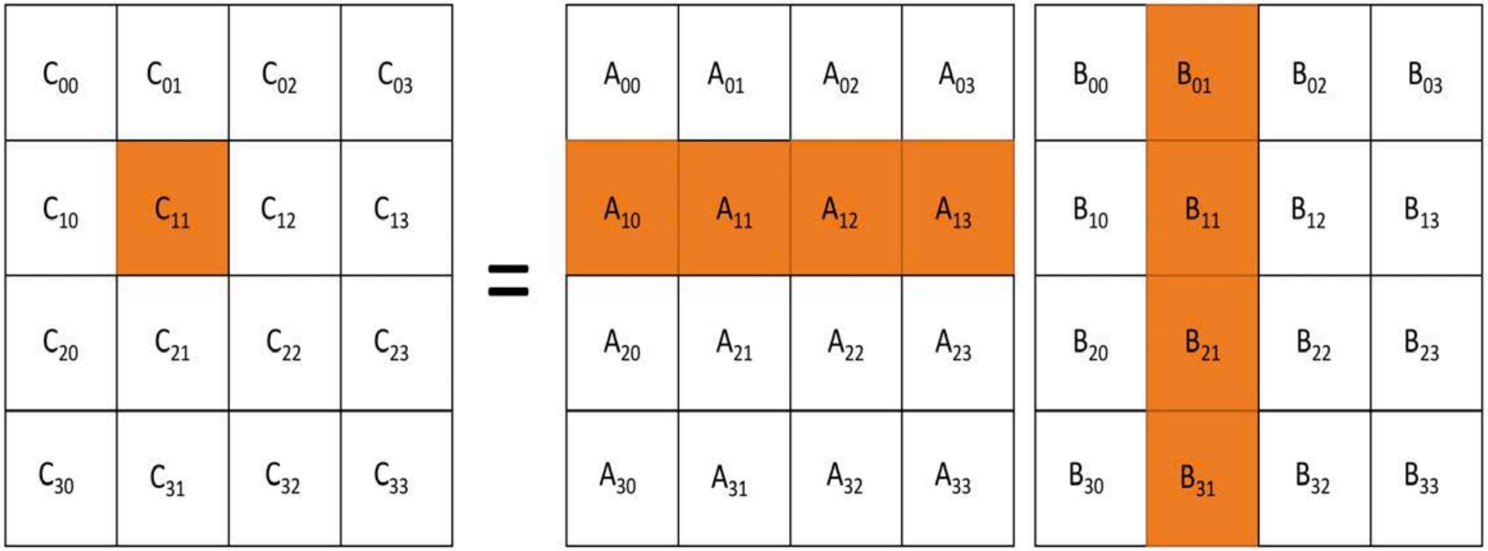
- Its high arithmetic intensity (ratio of flops to bytes) allows it to “cover” data motion well

$$\begin{bmatrix} C_{00} & C_{01} & C_{02} & C_{03} \\ C_{10} & C_{11} & C_{12} & C_{13} \\ C_{20} & C_{21} & C_{22} & C_{23} \\ C_{30} & C_{31} & C_{32} & C_{33} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} B_{00} & B_{01} & B_{02} & B_{03} \\ B_{10} & B_{11} & B_{12} & B_{13} \\ B_{20} & B_{21} & B_{22} & B_{23} \\ B_{30} & B_{31} & B_{32} & B_{33} \end{bmatrix}$$

Permutation example:

Dense matrix-matrix multiplication

- Each block of the product on each processor is the result of $\sqrt{P}$ blocks of each of the factors



$$\begin{bmatrix} C_{00} & C_{01} & C_{02} & C_{03} \\ C_{10} & C_{11} & C_{12} & C_{13} \\ C_{20} & C_{21} & C_{22} & C_{23} \\ C_{30} & C_{31} & C_{32} & C_{33} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} B_{00} & B_{01} & B_{02} & B_{03} \\ B_{10} & B_{11} & B_{12} & B_{13} \\ B_{20} & B_{21} & B_{22} & B_{23} \\ B_{30} & B_{31} & B_{32} & B_{33} \end{bmatrix}$$

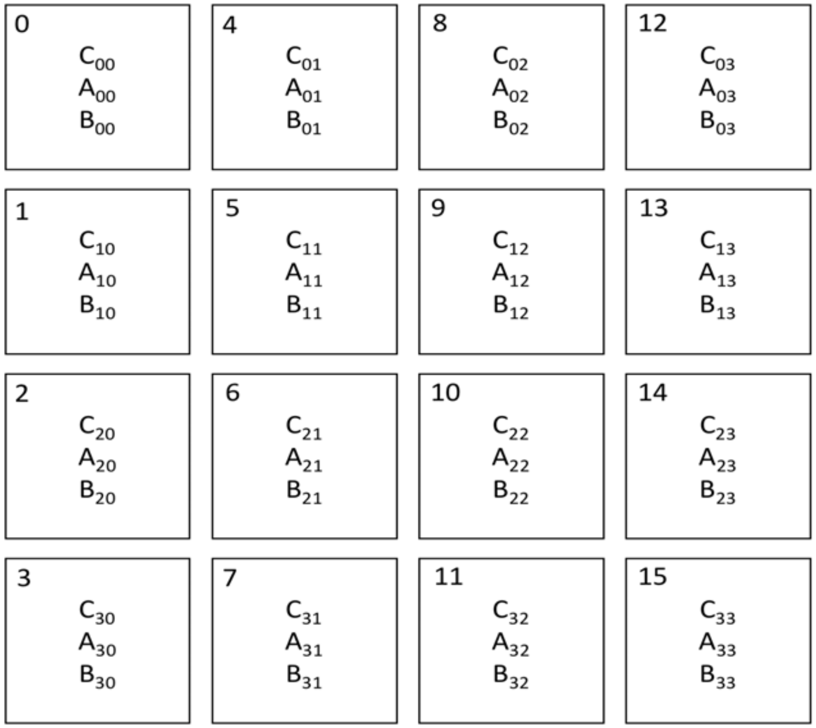
$$C_{11} = A_{10}B_{01} + A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31}$$

- All but one of these pairs are initially nonlocal

Permutation example:

Dense matrix-matrix multiplication

- From the initial state shown



0 C_{00} A_{00} B_{00}	4 C_{01} A_{01} B_{01}	8 C_{02} A_{02} B_{02}	12 C_{03} A_{03} B_{03}
1 C_{10} A_{10} B_{10}	5 C_{11} A_{11} B_{11}	9 C_{12} A_{12} B_{12}	13 C_{13} A_{13} B_{13}
2 C_{20} A_{20} B_{20}	6 C_{21} A_{21} B_{21}	10 C_{22} A_{22} B_{22}	14 C_{23} A_{23} B_{23}
3 C_{30} A_{30} B_{30}	7 C_{31} A_{31} B_{31}	11 C_{32} A_{32} B_{32}	15 C_{33} A_{33} B_{33}

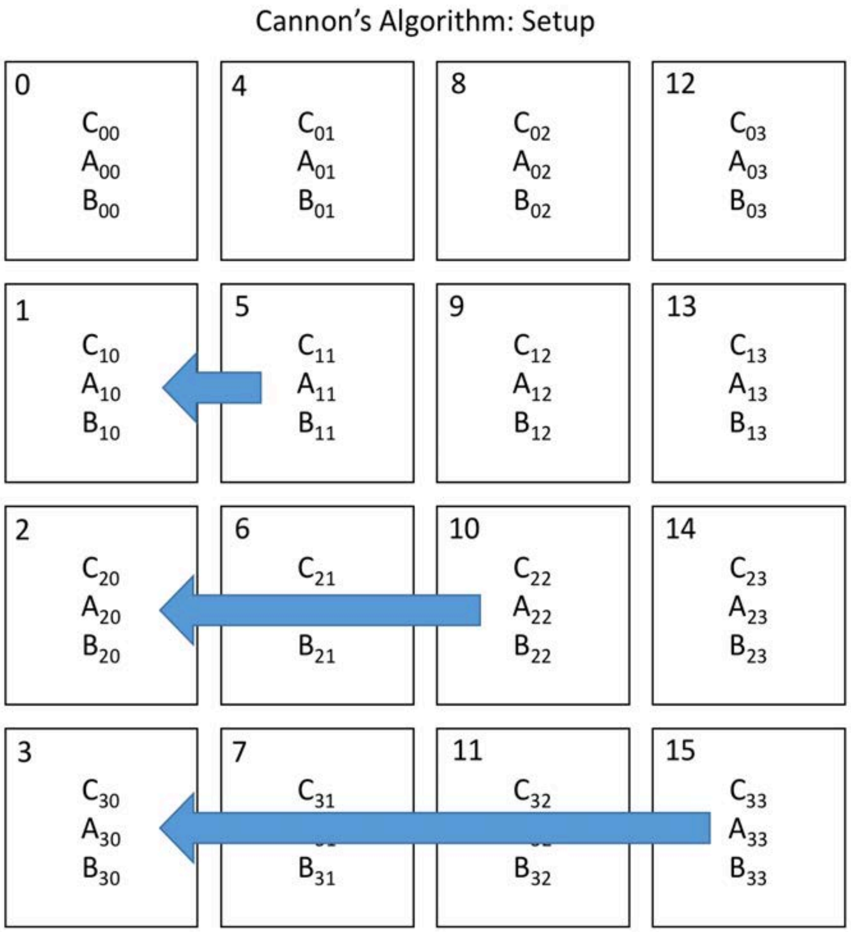
- Cannon's algorithm is

```
row $i$ of matrix $A$ is circularly shifted by $i$ blocks left
col $j$ of matrix $B$ is circularly shifted by $j$ blocks up
For $k = 0$ to $\sqrt{P}$-1:
    $P_{ij}$ multiplies its two entries and accumulates
    each row of $A$ is circularly shifted $1$ element left
    each col of $B$ is circularly shifted $1$ element up
```

Permutation example:

Dense matrix-matrix multiplication

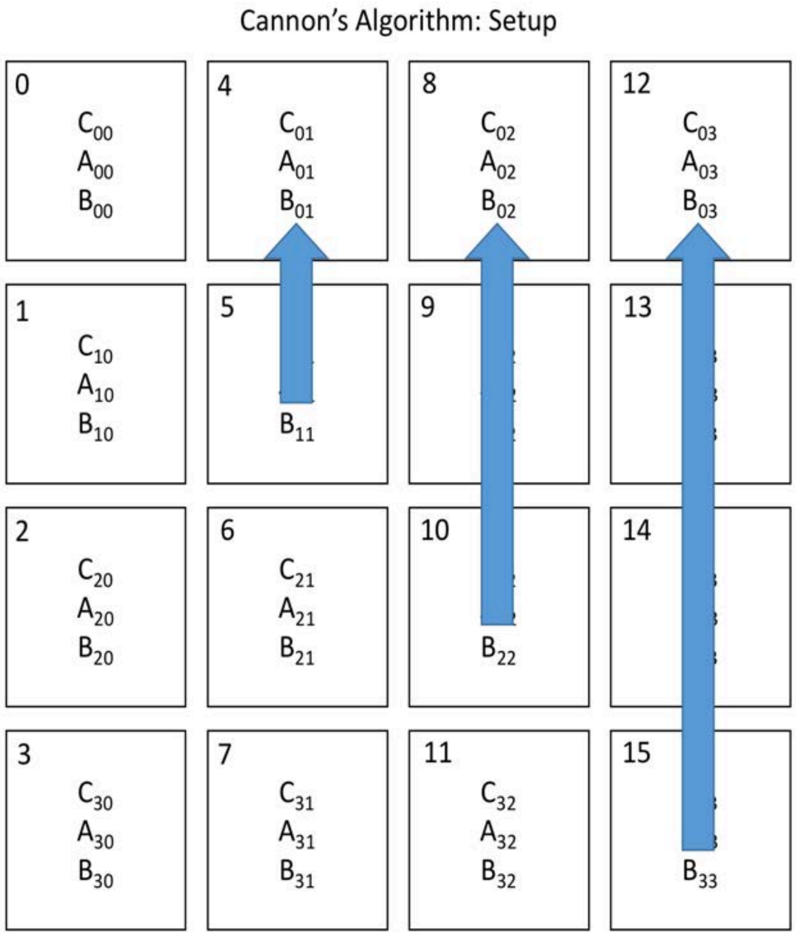
row i of matrix A is circularly shifted by i blocks left



Cannon's Algorithm: Setup

0 C_{00} A_{00} B_{00}	4 C_{01} A_{01} B_{01}	8 C_{02} A_{02} B_{02}	12 C_{03} A_{03} B_{03}
1 C_{10} A_{10} B_{10}	5 C_{11} A_{11} B_{11}	9 C_{12} A_{12} B_{12}	13 C_{13} A_{13} B_{13}
2 C_{20} A_{20} B_{20}	6 C_{21} B_{21}	10 C_{22} A_{22} B_{22}	14 C_{23} A_{23} B_{23}
3 C_{30} A_{30} B_{30}	7 C_{31} B_{31}	11 C_{32} B_{32}	15 C_{33} A_{33} B_{33}

col j of matrix B is circularly shifted by j blocks up



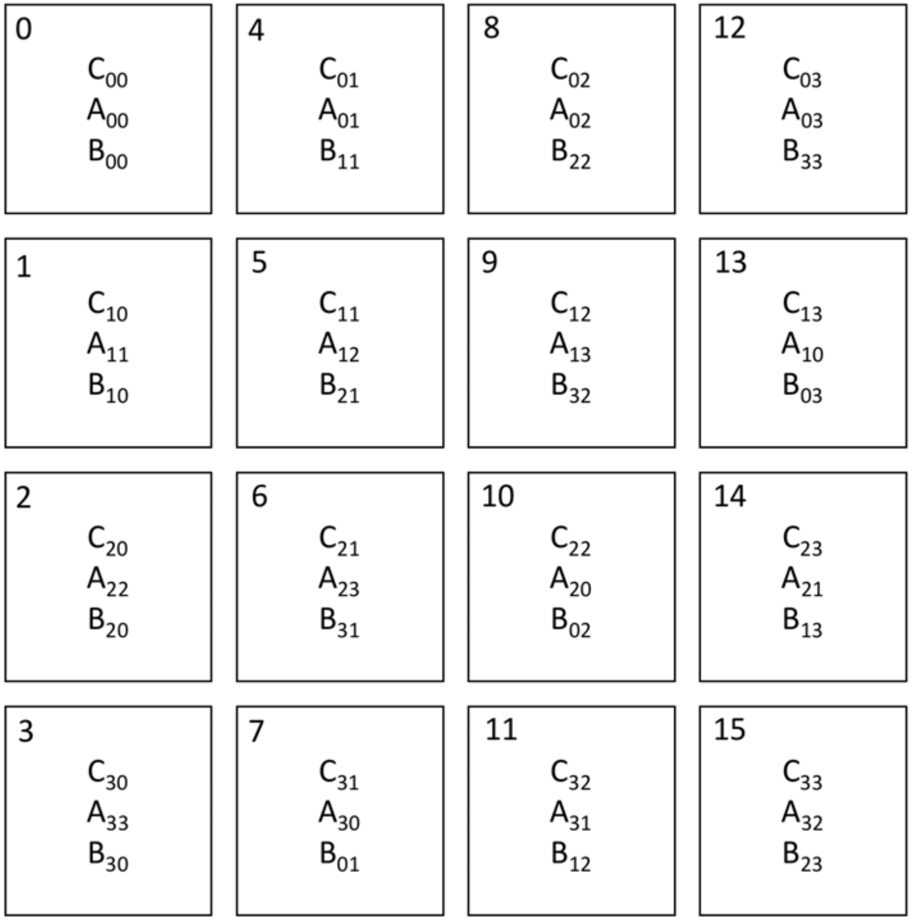
Cannon's Algorithm: Setup

0 C_{00} A_{00} B_{00}	4 C_{01} A_{01} B_{01}	8 C_{02} A_{02} B_{02}	12 C_{03} A_{03} B_{03}
1 C_{10} A_{10} B_{10}	5 B_{11}	9	13
2 C_{20} A_{20} B_{20}	6 C_{21} A_{21} B_{21}	10 B_{22}	14
3 C_{30} A_{30} B_{30}	7 C_{31} A_{31} B_{31}	11 C_{32} A_{32} B_{32}	15 B_{33}

Permutation example:

Dense matrix-matrix multiplication

Layout after initialization

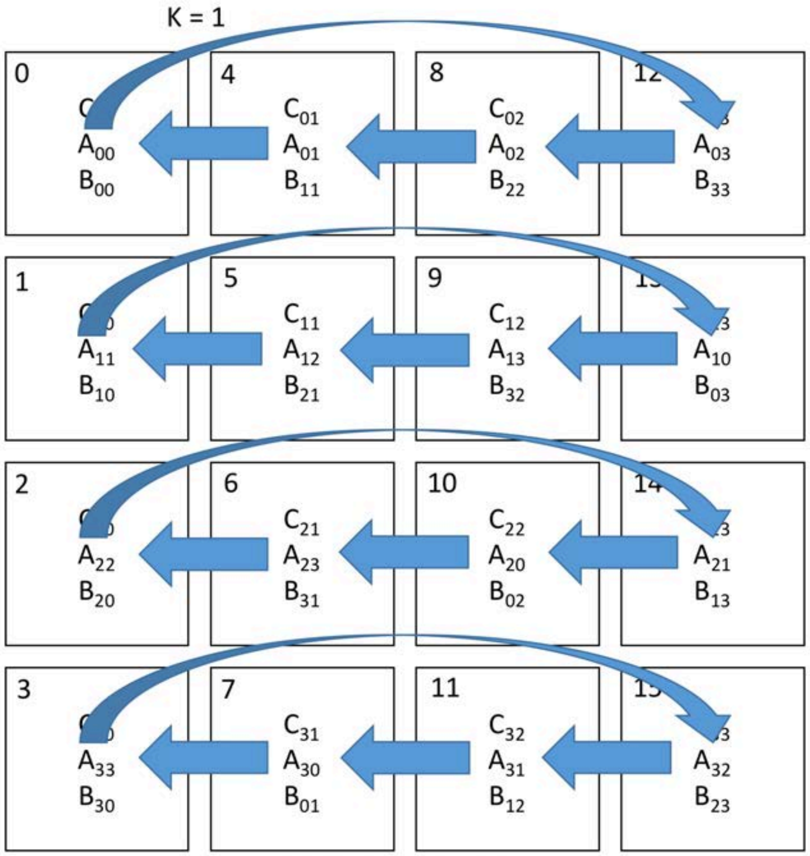


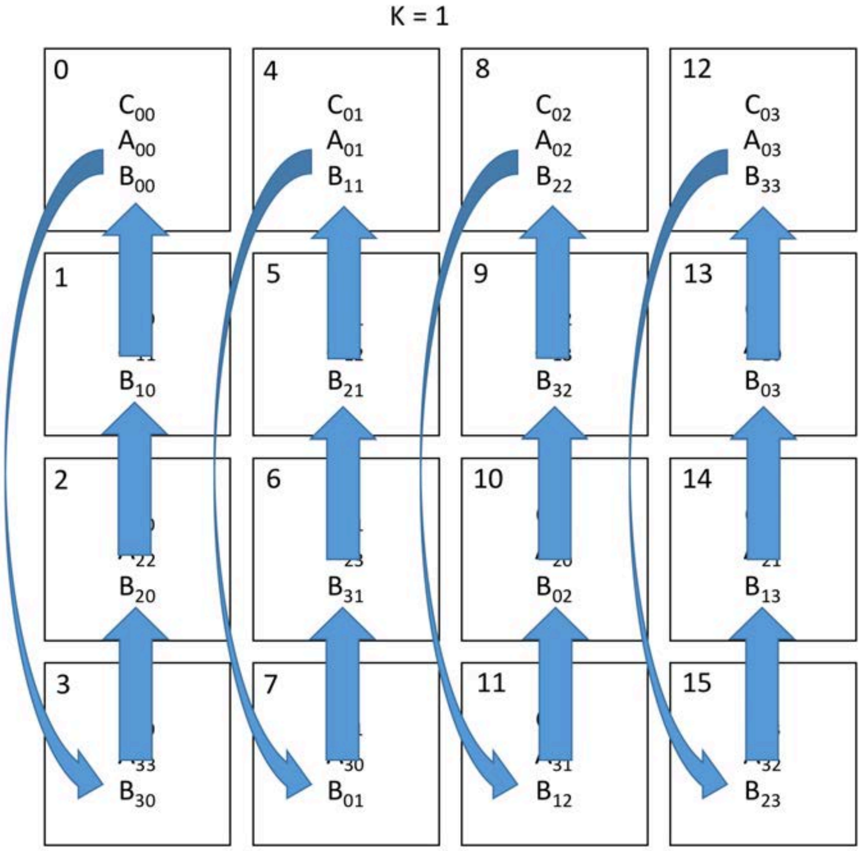
0 C_{00} A_{00} B_{00}	4 C_{01} A_{01} B_{11}	8 C_{02} A_{02} B_{22}	12 C_{03} A_{03} B_{33}
1 C_{10} A_{11} B_{10}	5 C_{11} A_{12} B_{21}	9 C_{12} A_{13} B_{32}	13 C_{13} A_{10} B_{03}
2 C_{20} A_{22} B_{20}	6 C_{21} A_{23} B_{31}	10 C_{22} A_{20} B_{02}	14 C_{23} A_{21} B_{13}
3 C_{30} A_{33} B_{30}	7 C_{31} A_{30} B_{01}	11 C_{32} A_{31} B_{12}	15 C_{33} A_{32} B_{23}

Permutation example:

Dense matrix-matrix multiplication

Accumulate* and shift $C_{ij} \mathrel{+}= A_{i,[i+j+k]} B_{[i+j+k],j}$


K = 1
0 A_{00} B_{00}
4 C_{01} A_{01} B_{11}
8 C_{02} A_{02} B_{22}
12 A_{03} B_{33}
1 A_{11} B_{10}
5 C_{11} A_{12} B_{21}
9 C_{12} A_{13} B_{32}
A_{10} B_{03}
2 A_{22} B_{20}
6 C_{21} A_{23} B_{31}
10 C_{22} A_{20} B_{02}
14 A_{21} B_{13}
3 A_{33} B_{30}
7 C_{31} A_{30} B_{01}
11 C_{32} A_{31} B_{12}
A_{32} B_{23}

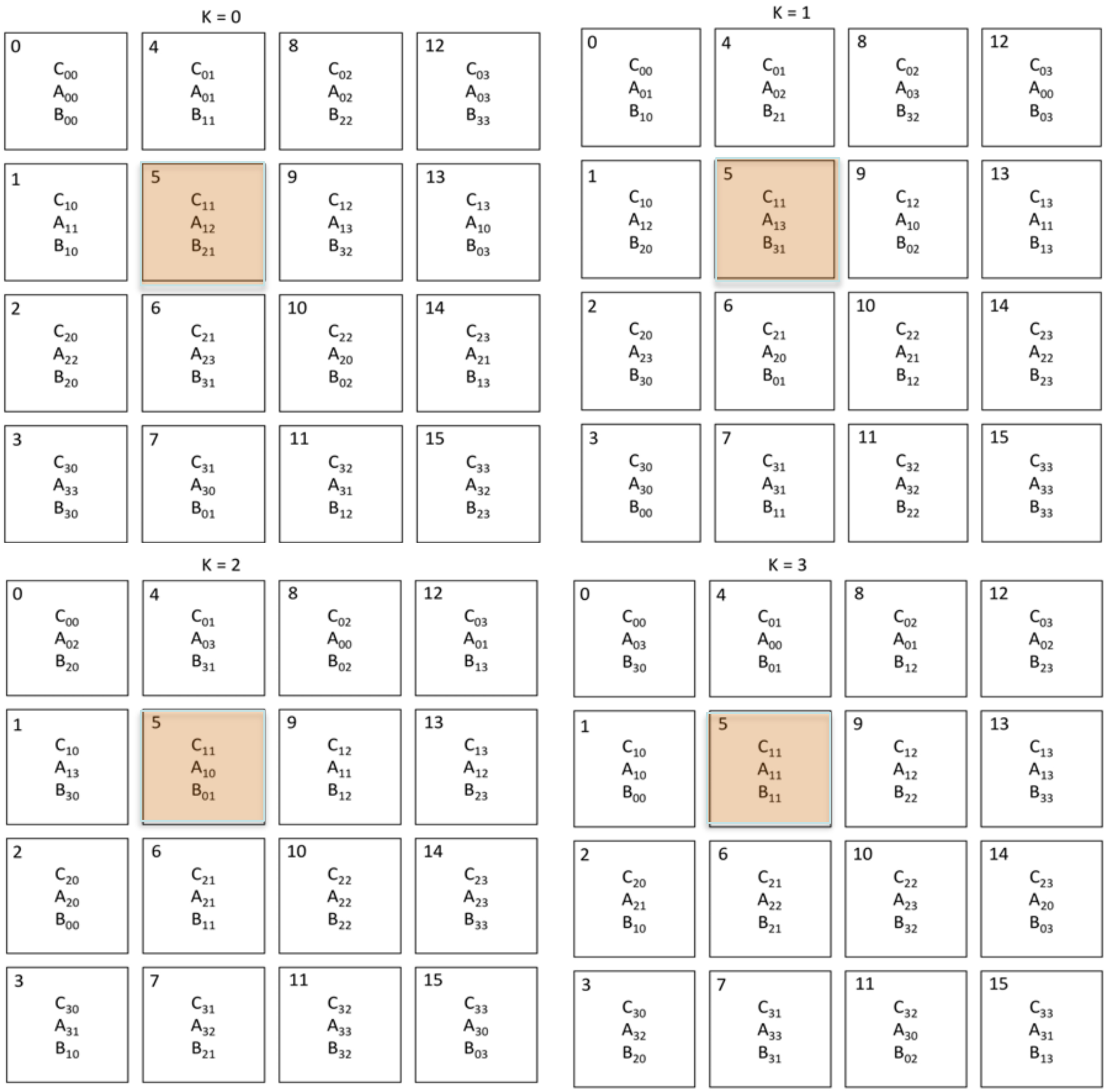


*The bracket $[i+j+k]$ is interpreted in the sense of modulus $\sqrt{P}$ (4 in this example)

Permutation example:

Dense matrix-matrix multiplication

- The distribution after each phase is depicted here
- It is readily verified that the correct accumulations have occurred



K = 0

0 C_{00} A_{00} B_{00}	4 C_{01} A_{01} B_{11}	8 C_{02} A_{02} B_{22}	12 C_{03} A_{03} B_{33}
1 C_{10} A_{11} B_{10}	5 C_{11} A_{12} B_{21}	9 C_{12} A_{13} B_{32}	13 C_{13} A_{10} B_{03}
2 C_{20} A_{22} B_{20}	6 C_{21} A_{23} B_{31}	10 C_{22} A_{20} B_{02}	14 C_{23} A_{21} B_{13}
3 C_{30} A_{33} B_{30}	7 C_{31} A_{30} B_{01}	11 C_{32} A_{31} B_{12}	15 C_{33} A_{32} B_{23}

K = 1

0 C_{00} A_{01} B_{10}	4 C_{01} A_{02} B_{21}	8 C_{02} A_{03} B_{32}	12 C_{03} A_{00} B_{03}
1 C_{10} A_{12} B_{20}	5 C_{11} A_{13} B_{31}	9 C_{12} A_{10} B_{02}	13 C_{13} A_{11} B_{13}
2 C_{20} A_{23} B_{30}	6 C_{21} A_{20} B_{01}	10 C_{22} A_{21} B_{12}	14 C_{23} A_{22} B_{23}
3 C_{30} A_{30} B_{00}	7 C_{31} A_{31} B_{11}	11 C_{32} A_{32} B_{22}	15 C_{33} A_{33} B_{33}

K = 2

0 C_{00} A_{02} B_{20}	4 C_{01} A_{03} B_{31}	8 C_{02} A_{00} B_{02}	12 C_{03} A_{01} B_{13}
1 C_{10} A_{13} B_{30}	5 C_{11} A_{10} B_{01}	9 C_{12} A_{11} B_{12}	13 C_{13} A_{12} B_{23}
2 C_{20} A_{20} B_{00}	6 C_{21} A_{21} B_{11}	10 C_{22} A_{22} B_{22}	14 C_{23} A_{23} B_{33}
3 C_{30} A_{31} B_{10}	7 C_{31} A_{32} B_{21}	11 C_{32} A_{33} B_{32}	15 C_{33} A_{30} B_{03}

K = 3

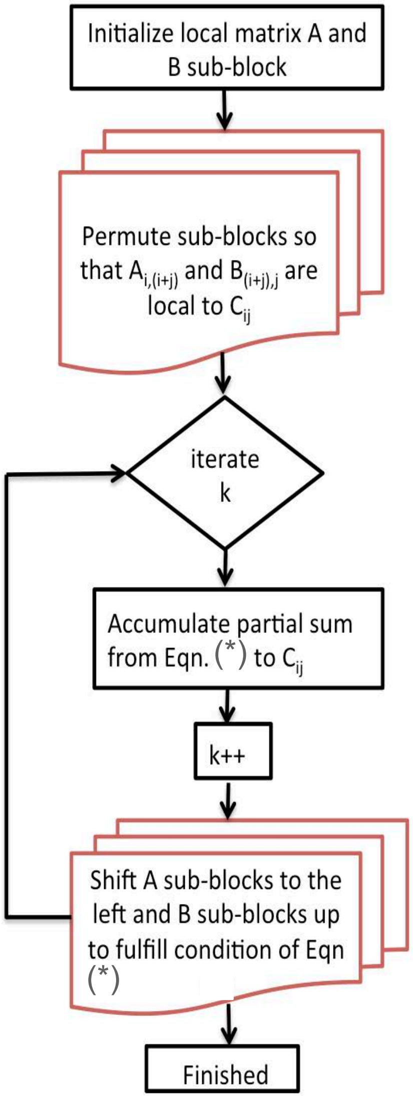
0 C_{00} A_{03} B_{30}	4 C_{01} A_{00} B_{01}	8 C_{02} A_{01} B_{12}	12 C_{03} A_{02} B_{23}
1 C_{10} A_{10} B_{00}	5 C_{11} A_{11} B_{11}	9 C_{12} A_{12} B_{22}	13 C_{13} A_{13} B_{33}
2 C_{20} A_{21} B_{10}	6 C_{21} A_{22} B_{21}	10 C_{22} A_{23} B_{32}	14 C_{23} A_{20} B_{03}
3 C_{30} A_{32} B_{20}	7 C_{31} A_{33} B_{31}	11 C_{32} A_{30} B_{02}	15 C_{33} A_{31} B_{13}

Permutation example:

Dense matrix-matrix multiplication

Flow chart and pseudocode

$$(*) \quad C_{ij} \mathrel{+}= A_{i,[i+j+k]} B_{[i+j+k],j}$$



Initialize local matrix A and B sub-block

Permute sub-blocks so that $A_{i,(i+j)}$ and $B_{(i+j),j}$ are local to C_{ij}

iterate k

Accumulate partial sum from Eqn. (*) to C_{ij}

k++

Shift A sub-blocks to the left and B sub-blocks up to fulfill condition of Eqn (*)

Finished

Task Dataflow

- While any parallel algorithm can be expressed as a directed graph where the nodes are tasks and the edges are data dependencies, many algorithms that explore graphs are themselves naturally expressed as task dataflow to maximize concurrency
- A goal is to maximize the number of tasks that can be executed concurrently (breadth) and to minimize the critical path of dependences (depth)

Task Dataflow example:

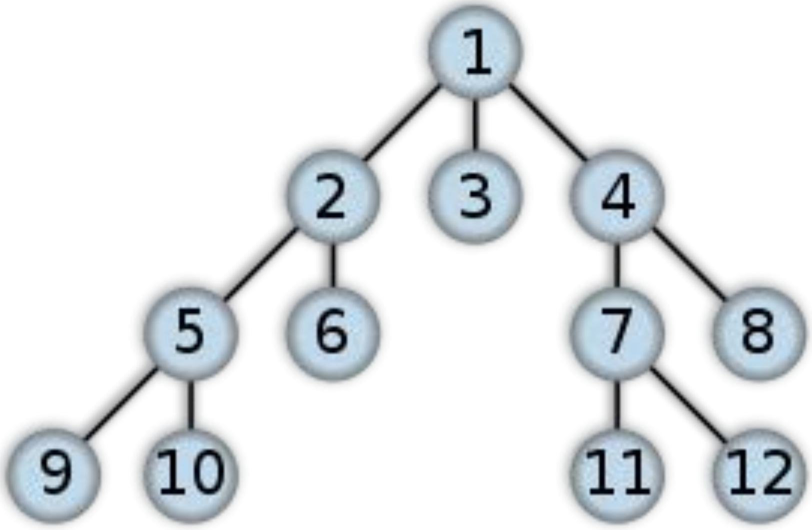
Breadth-first search

- Breadth-first search is a key component of numerous larger programs and is tested in the Graph500 benchmark
- A particular vertex is named as root
- Each adjacent vertex to the root is then traversed first
- When no more immediate root neighbors exist, previously labeled neighbors traverse their neighbors, thereby establishing the level (or distance) of every vertex from the root
- It should be compared to a depth-first search, which explores as far as possible before backtracking

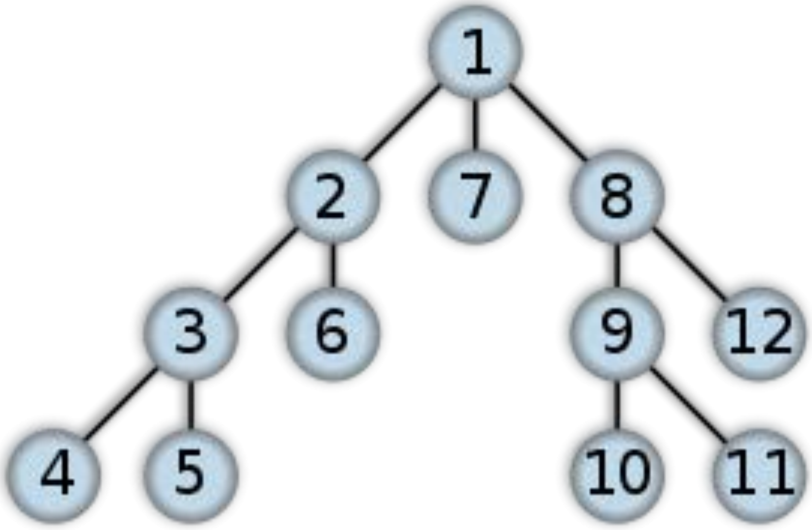
Task Dataflow example:

Breadth-first search

Vertex labeling from a breadth-first search



Vertex labeling from a depth-first search

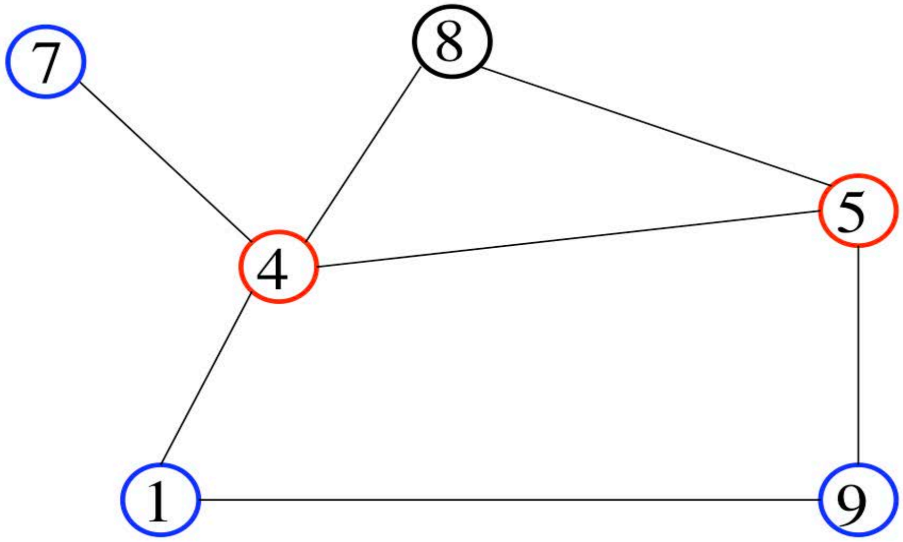


c/o Wikipedia

Task Dataflow example:

Breadth-first search

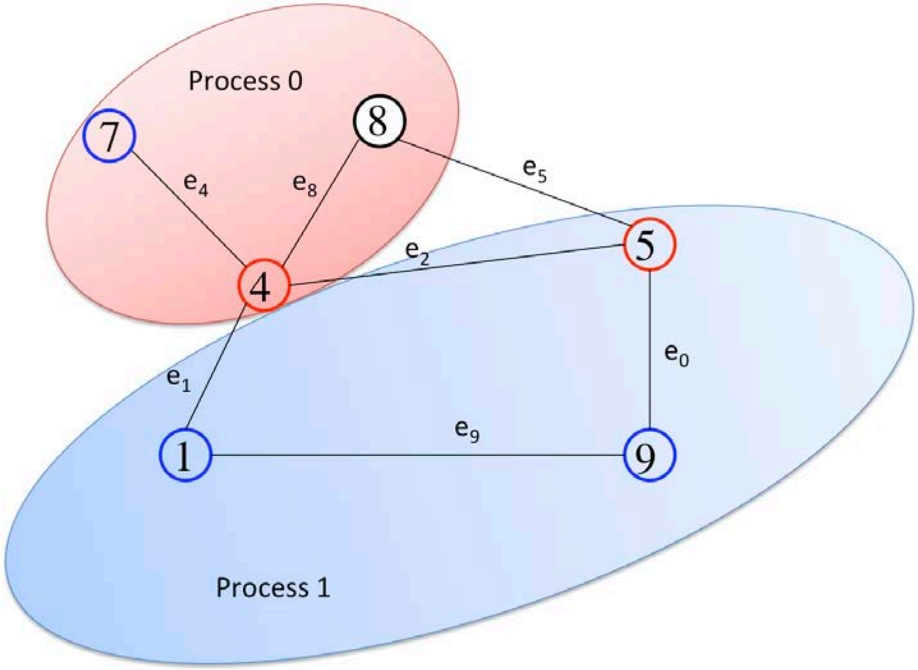
- Example for breadth-first search showing final level 0, level 1, and level 2
- Seven edges to traverse, on two of which the vertex has been previously visited, in sequential mode



Starting at root 8: 8, 4, 5, 1, 7, 9

Task Dataflow example:
Breadth-first search

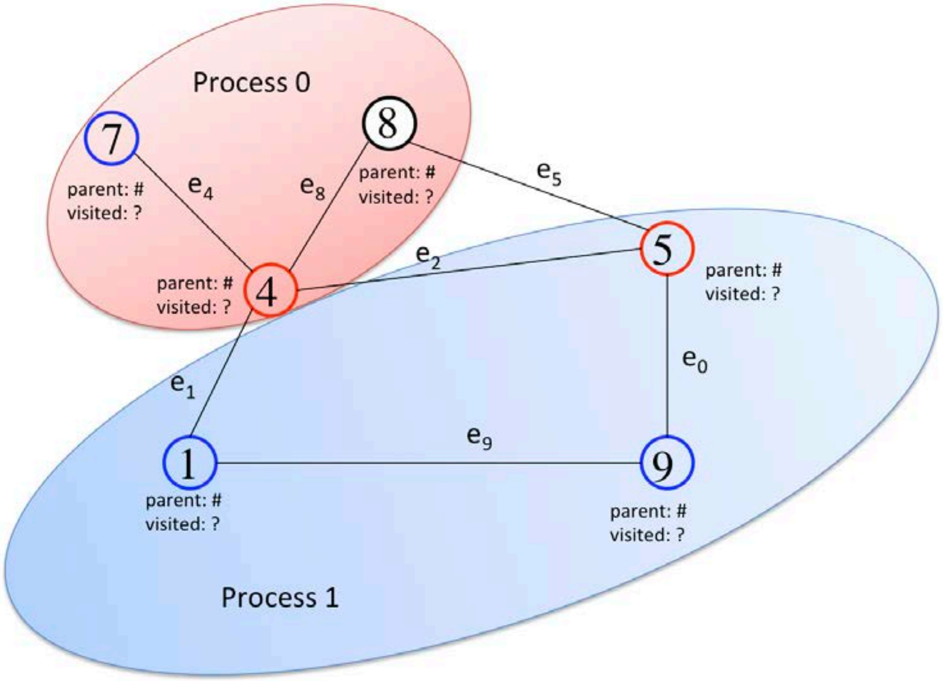
- Vertices are partitioned by process each with its own edge list, including the process number of the adjacent vertex



Task Dataflow example:

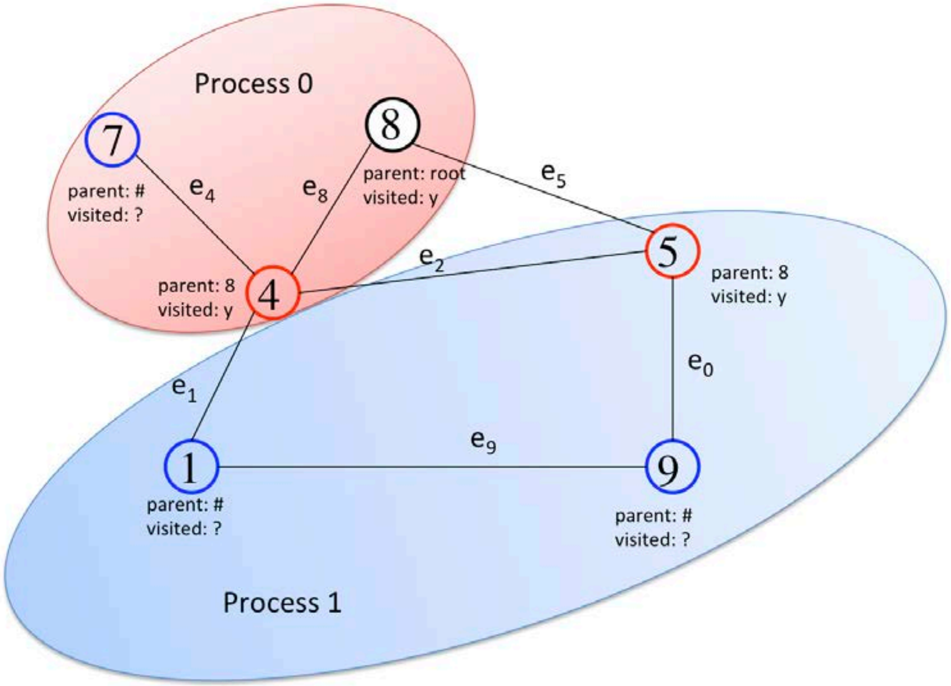
Breadth-first search

- To each vertex is associated a parent vertex label and a binary flag indicating if the vertex has been visited
- Parallel BFS is initialized



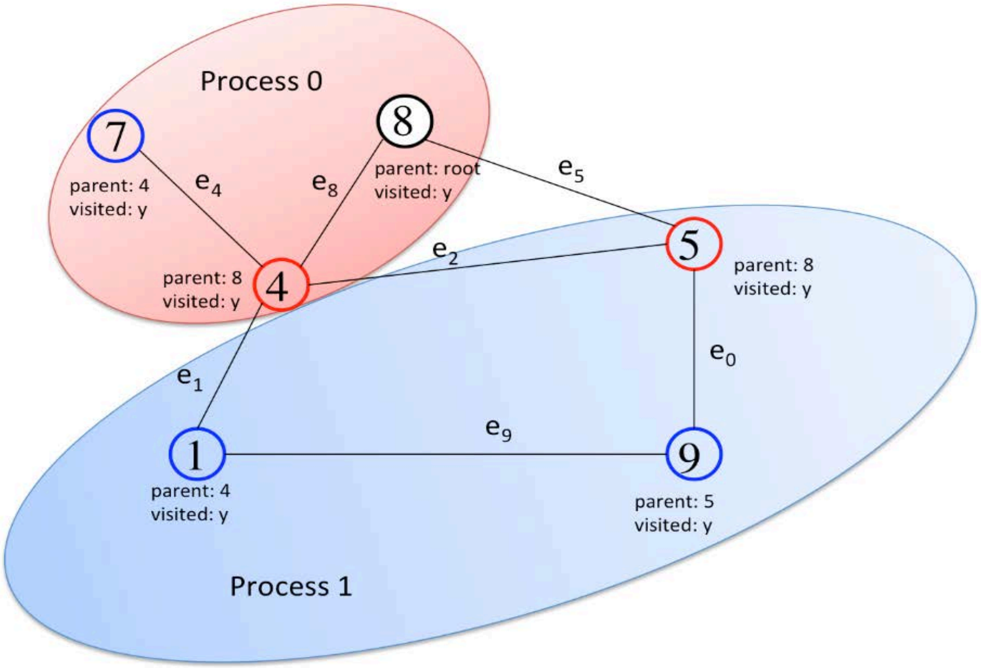
Task Dataflow example:
Breadth-first search

- At the first stage, a level 1 vertex is found on process 1, which then begins to search concurrently



Task Dataflow example:
Breadth-first search

- Two stages are completed in the time of five edge traversals rather than seven for sequential



Summary

- Four steps in creating a parallel program
- Co-design of solution algorithm and problem formulation with architecture, to tune for performance
- Top 10 algorithms for simulation
- Top 10 algorithms for data mining
- 7 floating point “dwarves” and 6 discrete “dwarves”
- 5 optimal scalable, hierarchical algorithms
- Flynn’s classification for programming paradigms supported in the hardware

Summary

- Embarrassing Parallelism
  - Monte Carlo
- Fork-Join
  - OpenMP “for” loop
- Manager-Worker
  - Global Parallel Genetic Algorithm
- Divide-Conquer-Combine
  - Quicksort

Summary

- Halo Exchange
  - Stencil Evaluation / Sparse Matrix-Vector Multiplication
- Permutation
  - Dense Matrix-Matrix Multiplication
- Task Dataflow
  - Breadth-first Search

Principal slide credits

- Thomas Sterling (U Indiana)
- Matthew Anderson (U Indiana)
- Maciej Brodowicz (U Indiana)
- plus individual slides as marked