Cloud Innovator
MEGAZONE

Megazone is delivering values through the business expertise and experiences built from the Platform Service, Digital Marketing and Digital Service areas

OpenACC
More Science, Less Programming

Computational Fluid Dynamics with Parallel programming

Matilda Lab / Hybrid AI Center / Megazone Cloud Corp.
Sungho Kim, Ph.D.

2023. 12.

Contents

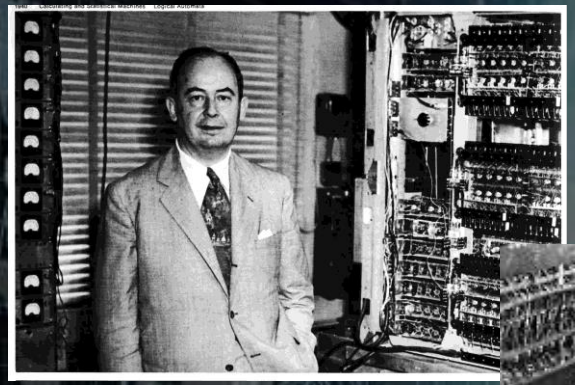
- First Day on OpenACC
  - Computer Architectures
    - NVIDIA HGX Server Architecture
    - NVIDIA H100 Architecture
  - NVIDIA HPC SDK
  - CUDA Platform
  - GPU Programming for HPC
  - OpenACC
  - Lab Exercise
- Second Day on Parallel CFD
  - Parallel Architecture review
  - Parallel Programming
  - Parallel Computational Science
  - Domain Decomposition Method
  - Parallel Matrix Solver
  - N-Body Problems

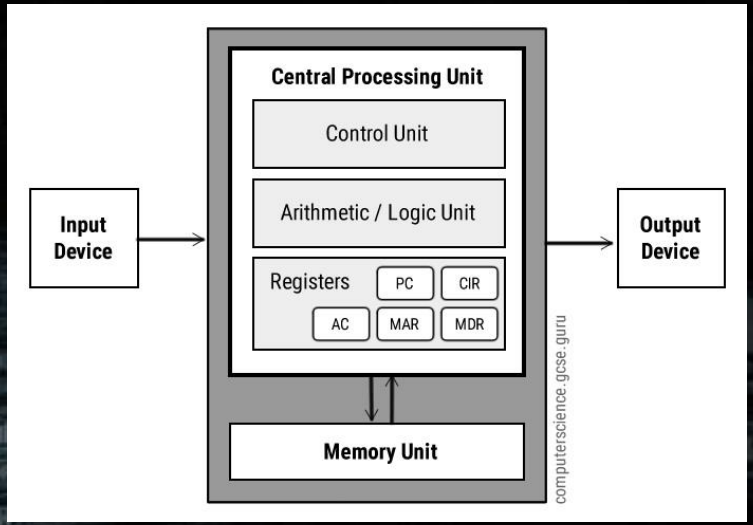
1. Computational Methods of Computational Science
 - Structured Grid
 - Unstructured Grid
 - Molecular Dynamics
2. Fundamentals of Parallel CFD
 - Data Parallel for Structured Grid
 - Domain Decomposition for Unstructured Grid
 - Essentially Parallel for MD
 - Vectorization for Acceleration
3. Data Parallel Programming
 - OpenMP, OpenACC
 - Vectorization to prevent memory overwriting
4. Domain Decomposition Method
 - Domain Decomposition Method
  - OpenACC or MPI
5. Algorithmic Parallel Programming
 - Parallel Linear Algebra
6. Hybrid Programming Method
 - OpenACC with MPI
7. Overcome Parallel Overhead
 - Profiling
  - Vectorization
  - Development Environments

Parallel Architecture

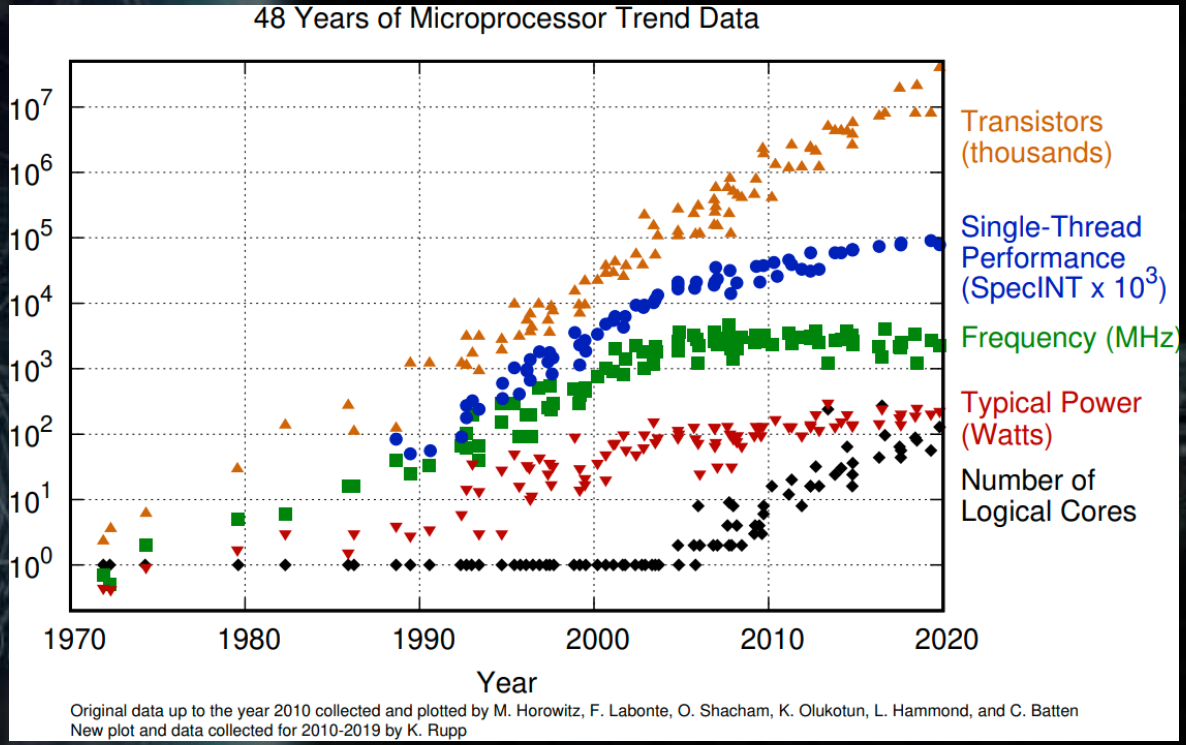
Review

von Neuman Architecture

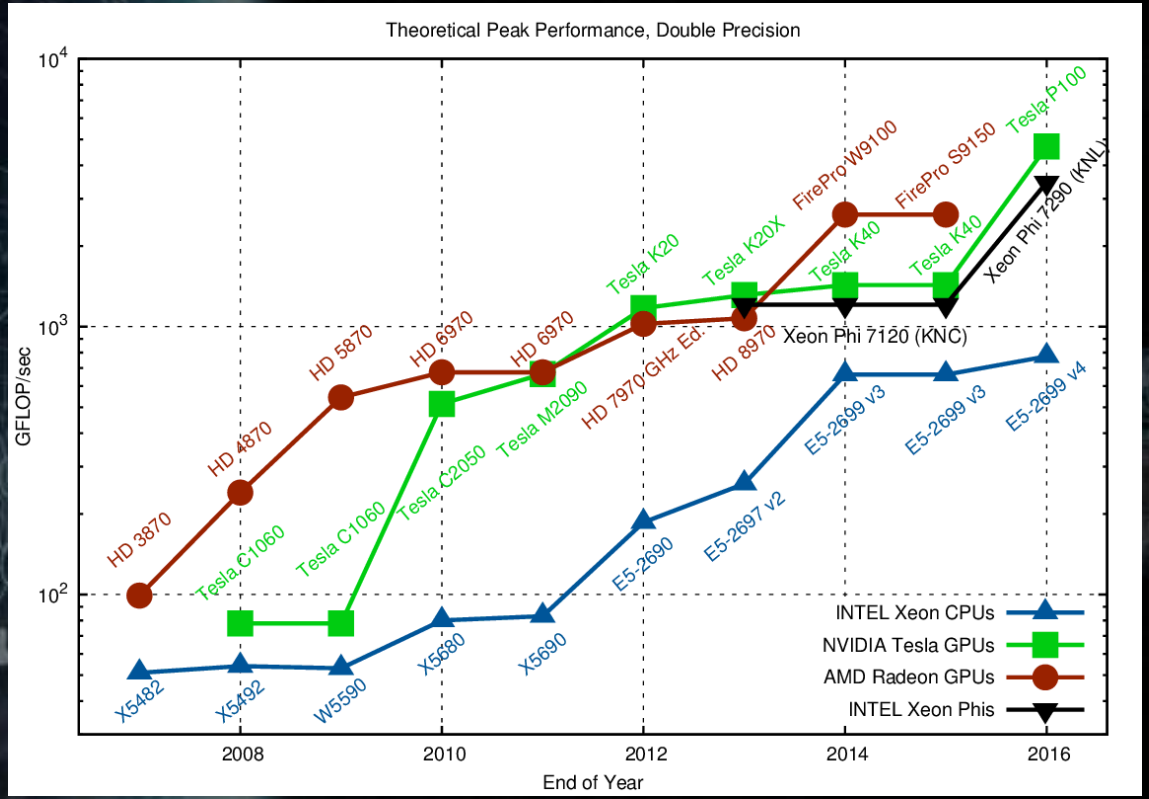




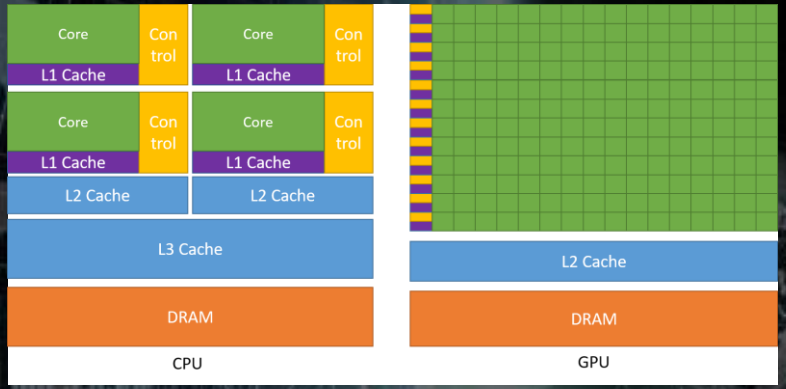
Moore's Law


48 Years of Microprocessor Trend Data
10^7
10^6
10^5
10^4
10^3
10^2
10^1
10^0
1970
1980
1990
2000
2010
2020
Year
Transistors (thousands)
Single-Thread Performance (SpecINT x 10^3)
Frequency (MHz)
Typical Power (Watts)
Number of Logical Cores
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2019 by K. Rupp

Accelerator performance growth



GPU Tensor Core

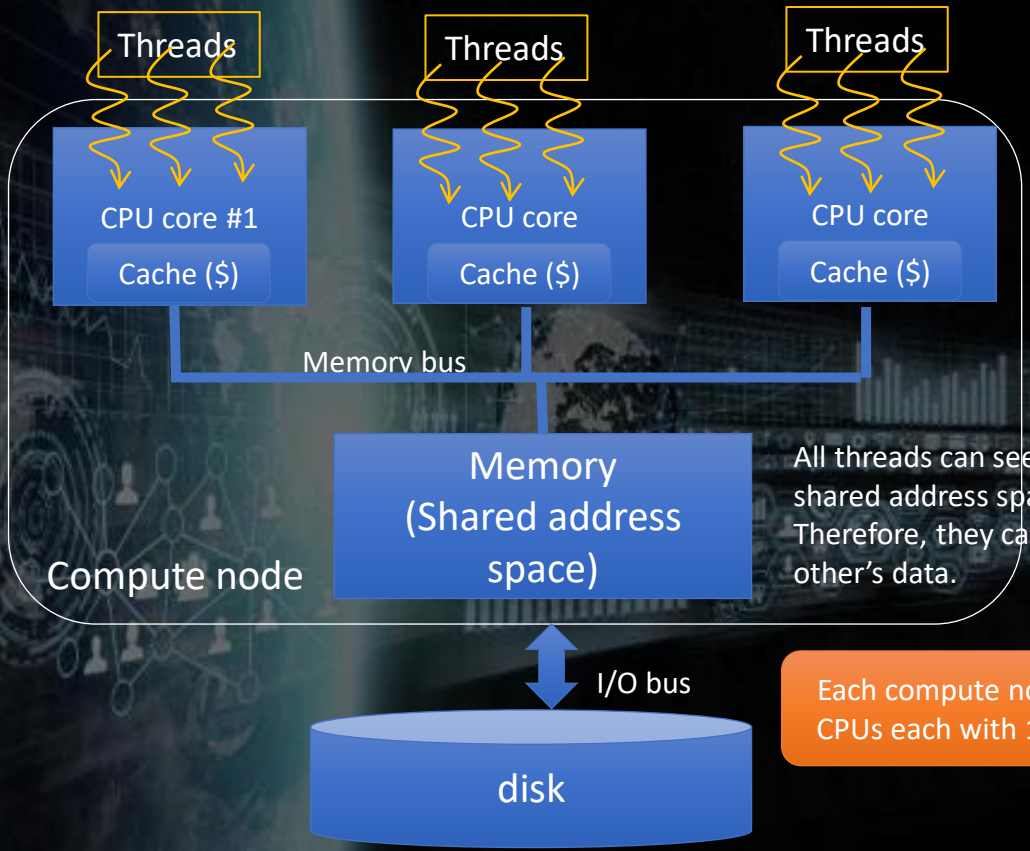


NVIDIA V100

Parallel Architecture by Memory Type

- Two major classes of parallel programming models:
  - Shared Memory
    - Multi-Thread
  - Distributed Memory

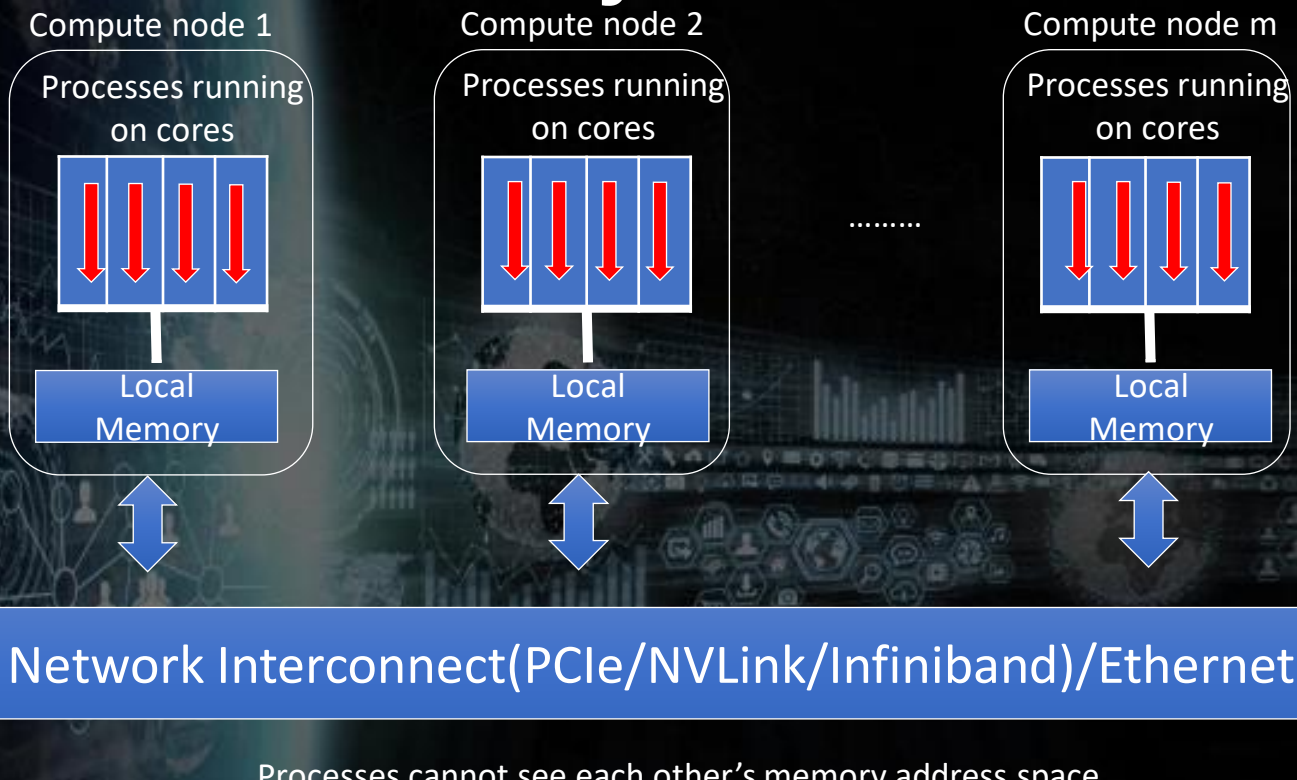
Shared Memory Architecture



All threads can see a single shared address space. Therefore, they can see each other's data.

Each compute node has 1-2 CPUs each with 10-14 cores

Distributed Memory Architecture



Processes cannot see each other's memory address space.
They have to send inter-process messages (using MPI).

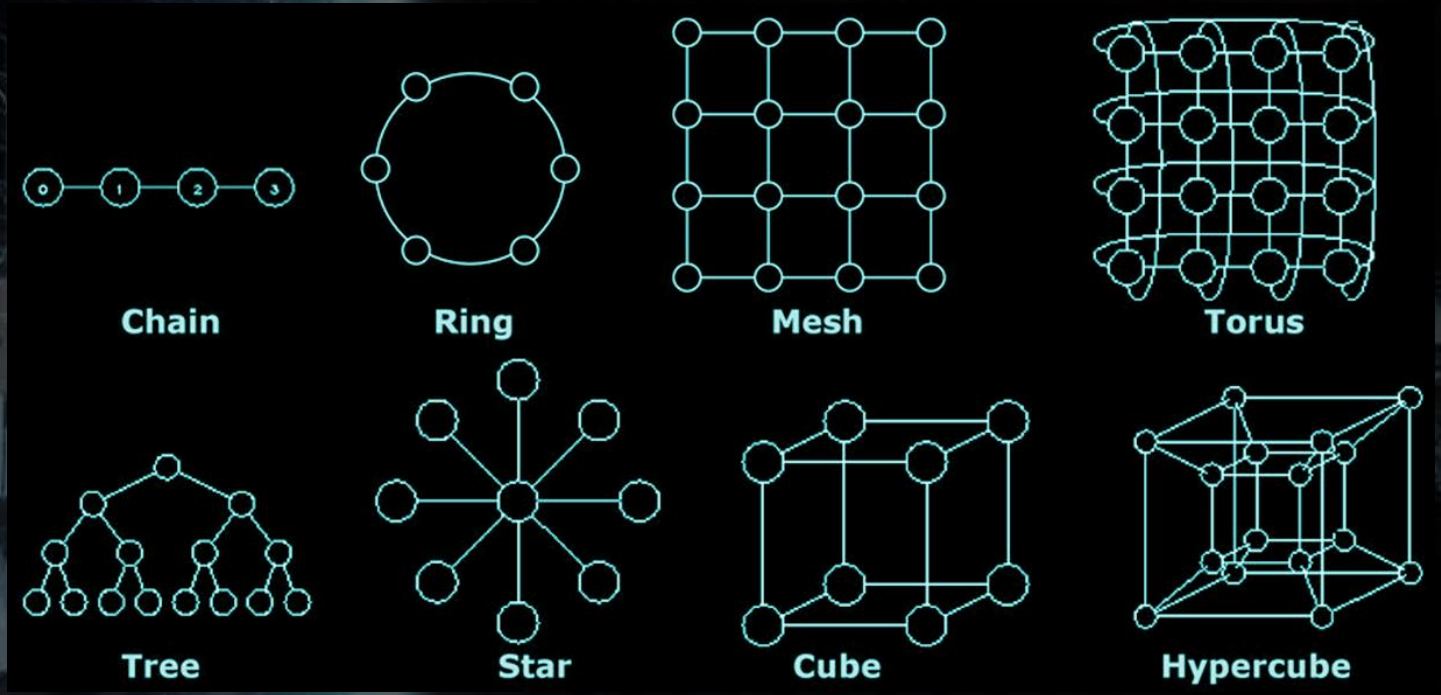
Distributed Memory System

- Clusters
  - A collection of commodity systems.
  - Connected by a commodity interconnection network.
- Nodes of a cluster are individual computers joined by a communication network.

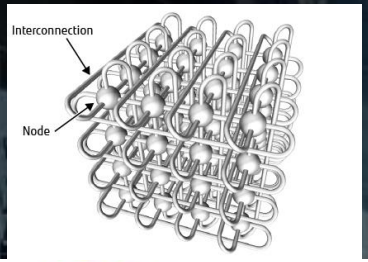
Cluster provides an Infiniband interconnect between all compute nodes

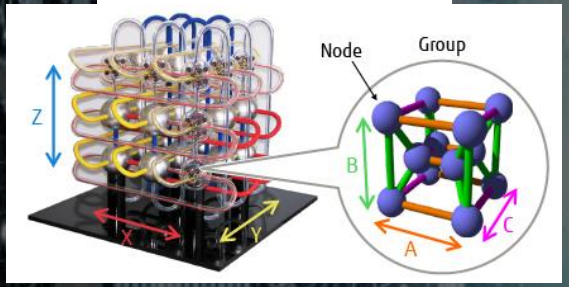
a.k.a. hybrid systems

Network Topology in Parallel Computer

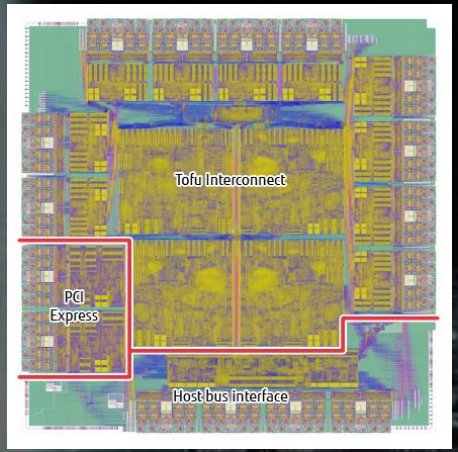


Fujitsu KEI Supercomputer Architecture

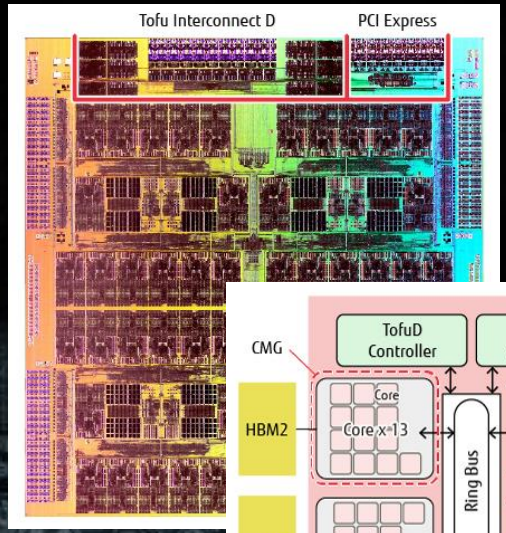


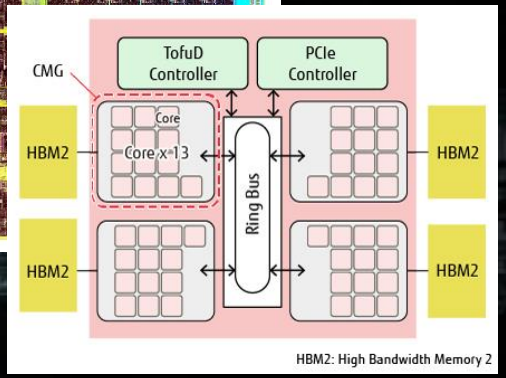


3D Torus Topology
6 dimensional mesh/torus



ICC Tofu Interconnect





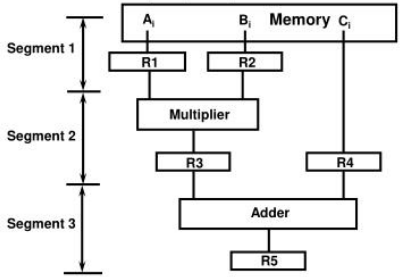
- Kei : 88,192 interconnected nodes in 2012
- Fugaku : 158,976 nodes in 2019

Pipelining for arithmetic operations

PIPELINING

A technique of decomposing a sequential process into suboperations, with each subprocess being executed in a partial dedicated segment that operates concurrently with all other segments.

$A_i * B_i + C_i$ for i = 1, 2, 3, ... , 7

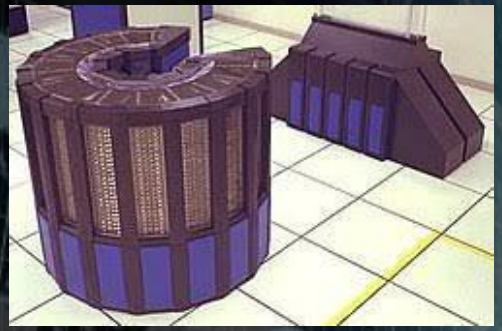


$R1 \leftarrow A_i$, $R2 \leftarrow B_i$	Load A_i and B_i
$R3 \leftarrow R1 * R2$, $R4 \leftarrow C_i$	Multiply and load C_i
$R5 \leftarrow R3 + R4$	Add

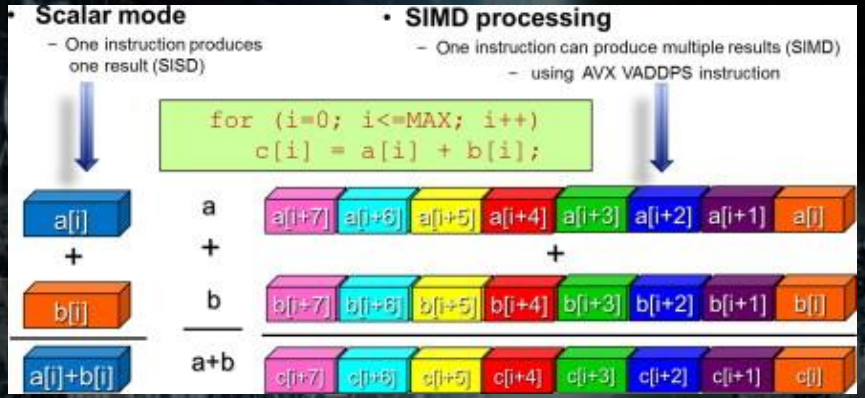
OPERATIONS IN EACH PIPELINE STAGE

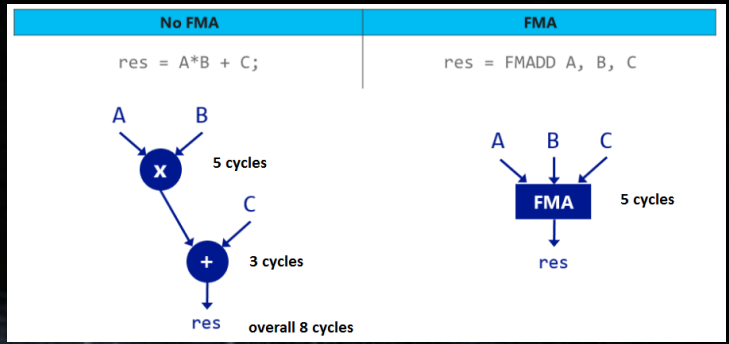
Clock Pulse Number	Segment 1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	A1	B1			
2	A2	B2	A1 * B1	C1	
3	A3	B3	A2 * B2	C2	A1 * B1 + C1
4	A4	B4	A3 * B3	C3	A2 * B2 + C2
5	A5	B5	A4 * B4	C4	A3 * B3 + C3
6	A6	B6	A5 * B5	C5	A4 * B4 + C4
7	A7	B7	A6 * B6	C6	A5 * B5 + C5
8			A7 * B7	C7	A6 * B6 + C6
9					A7 * B7 + C7

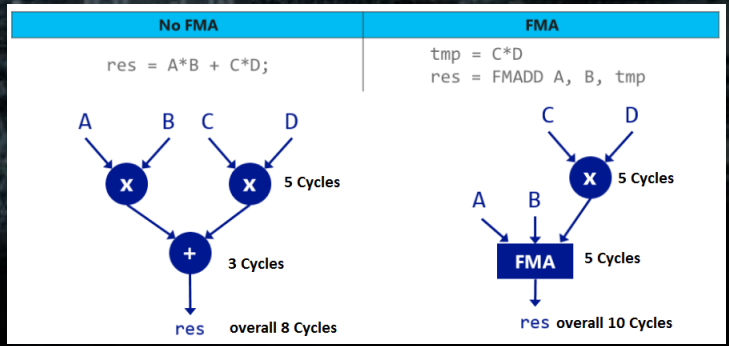
Cray Supercomputers



Vector ADD, MUL and FMA



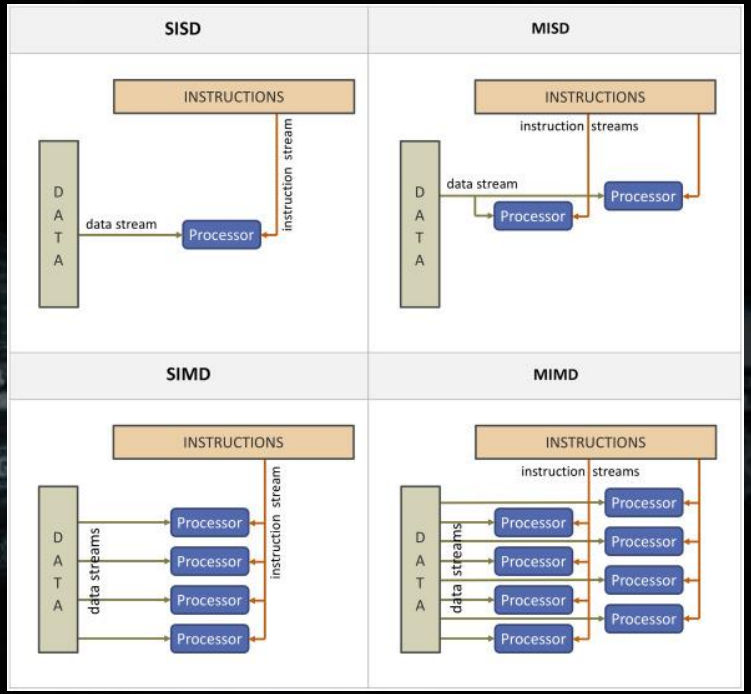




SSE2 : compute 2 floating point single precision computation at the same time
SSE4 : compute 4 floating point single precision computation at the same time
AVX : compute 8 floating point single precision computation at the same time
AVX 512 : compute 16 floating point single precision computation at the same time

Classification of Parallel Architecture

- Computer Architecture Classification depends on Instruction and Memory/Data Streams
  - SISD : Single Instruction and Single Data
  - MISD : Multiple Instruction and Single Data
  - SIMD : Single Instruction and Multiple Data
  - MIMD : Multiple Instruction and Multiple Data
- Thread Programming
  - SIMT : Single Instruction and Multiple Threads



Single Program Models: SIMD vs. MIMD

- **SP**: Single Program
  - Your parallel program is a single program that you execute on all threads (or processes)
- **SI**: Single Instruction
  - Each thread should be executing the same line of code at any given clock cycle.
- **MI**: Multiple Instruction
  - Each thread (or process) could be independently running a different line of your code (instruction) concurrently
- **MD**: Multiple Data
  - Each thread (or process) could be operating/accessing a different piece of the data from the memory concurrently

Single Program Models: SIMD vs. MIMD

SIMD

```
// Begin: parallel region of the code
…
..
..
..
..
..
..
..
..
..
..
// End: parallel region of the code
```

All threads executing the same line of code.
They may be accessing different pieces of data.

MIMD

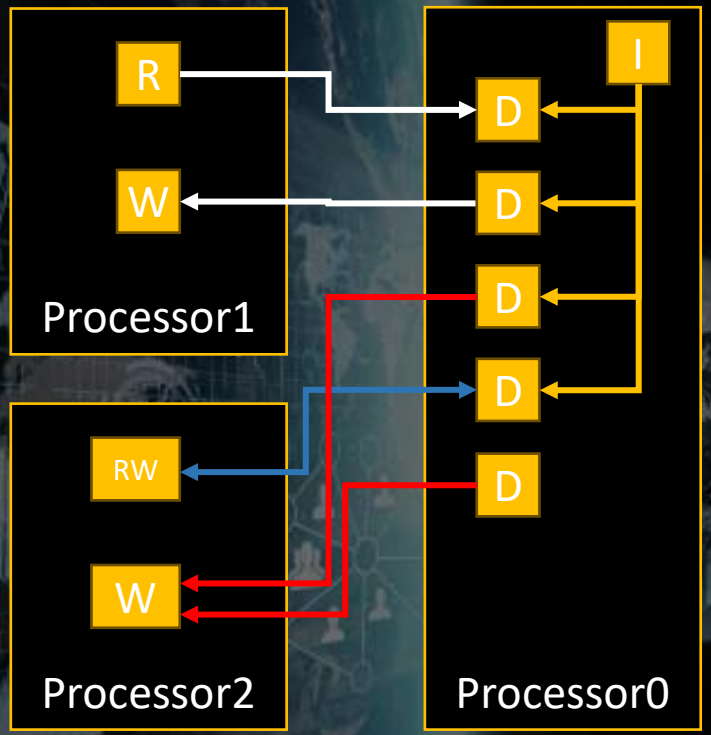
```
// Begin: parallel region of the code
…
..
..
..
..
..
..
..
..
..
..
// End: parallel region of the code
```

Thread 1

Thread 2

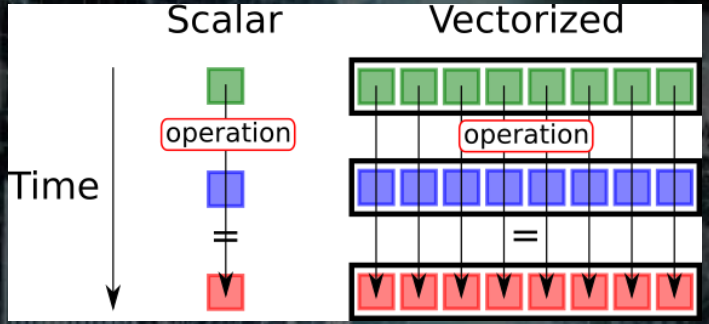
Thread 3

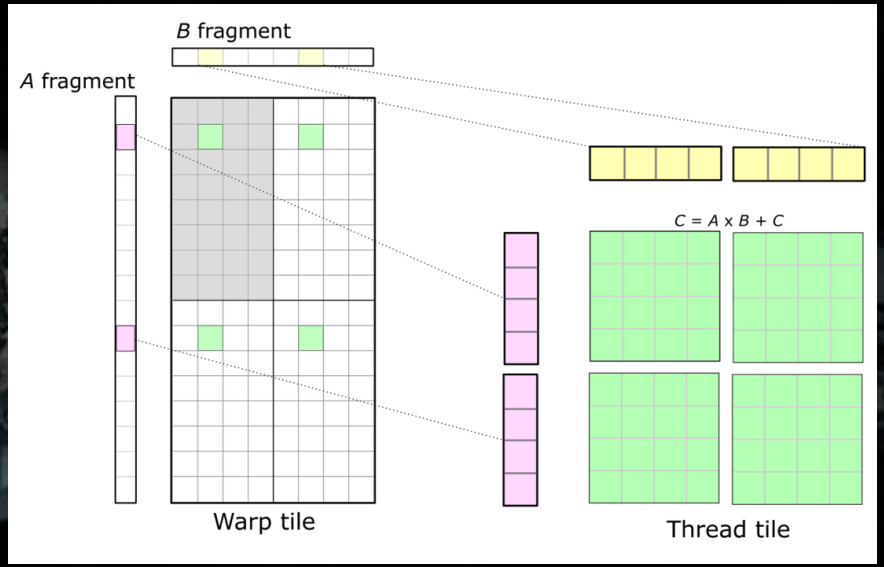
Memory Dependency of Parallel Architecture



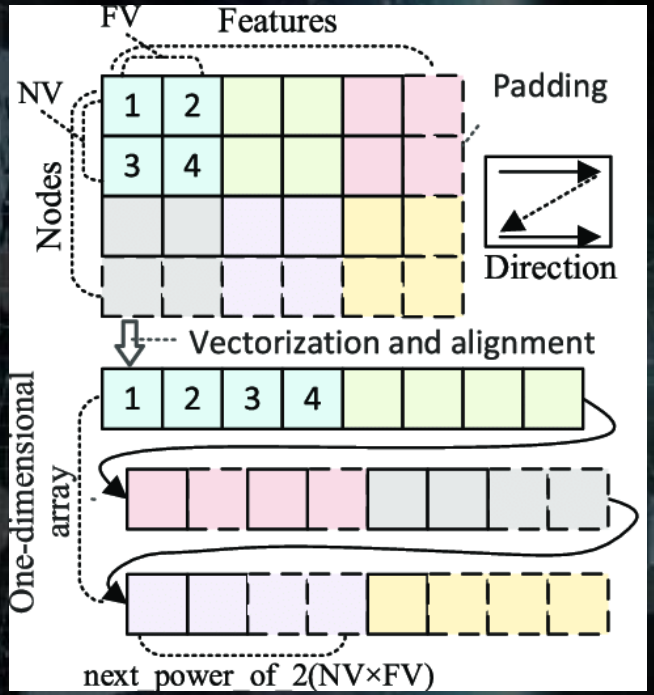
- Read from Processor
  - Anytime Good
- Write to Processor
  - On vacancy
- Read-Write from-to Processor
  - Sometimes data overwritten
- Write from Processors
  - Data conflict : have to schedule write sequence, but overwritten

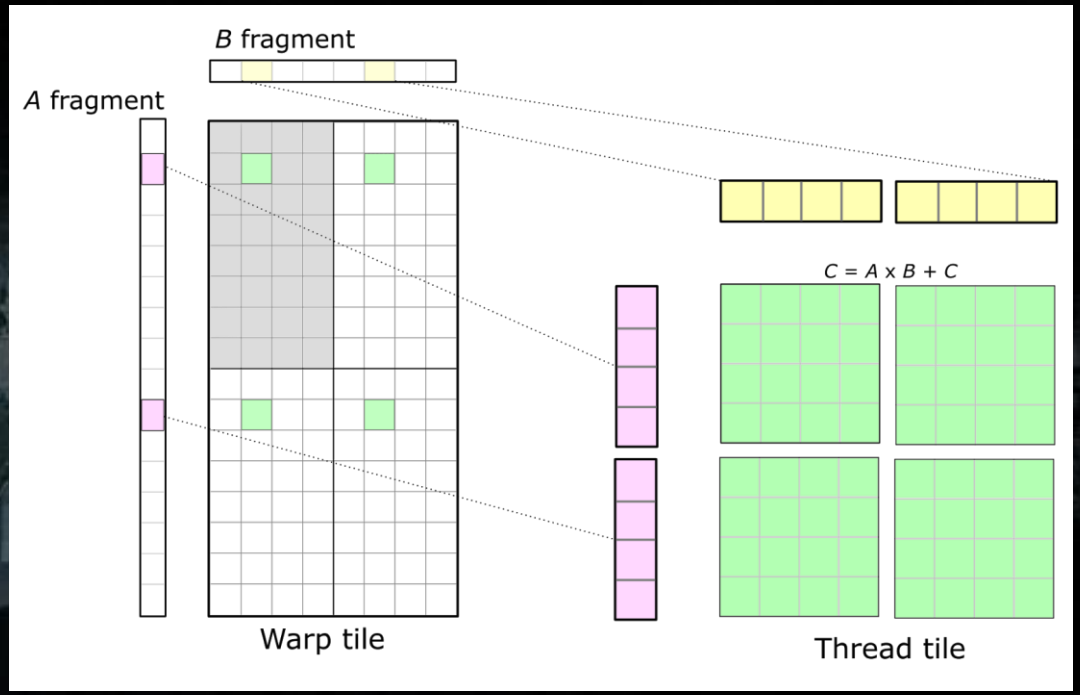
Vector vs. Tensor Core





Vector and Tensor Core

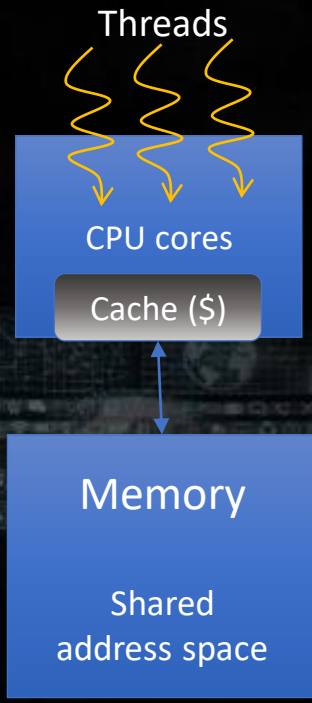




Multi-Threads Architecture

for shared memory architectures

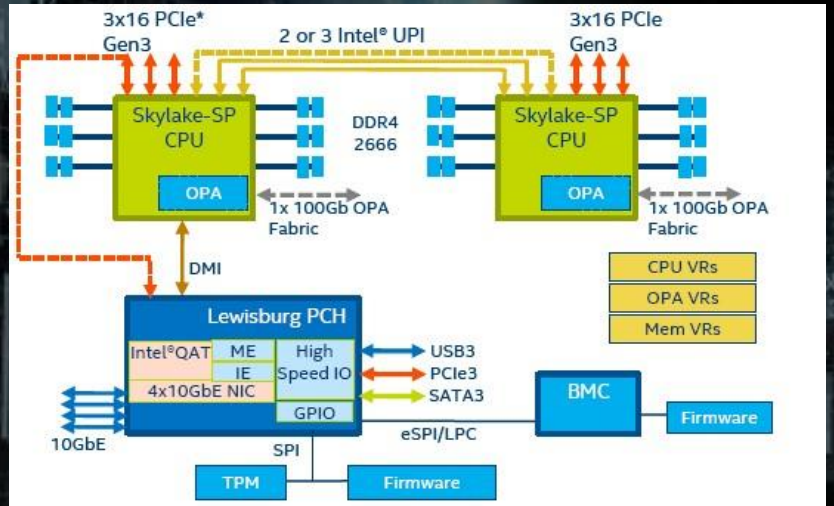
- Threads are contained within processes
  - One process => multiple threads
- All threads of a process share the same address space (in memory).
- Threads have the capability to run concurrently
  - executing different instructions and accessing different pieces of data at the same time
- But if the resource is occupied by another thread, they form a queue and wait.
  - For maximum throughput, it is ideal to map each thread to a unique/distinct core



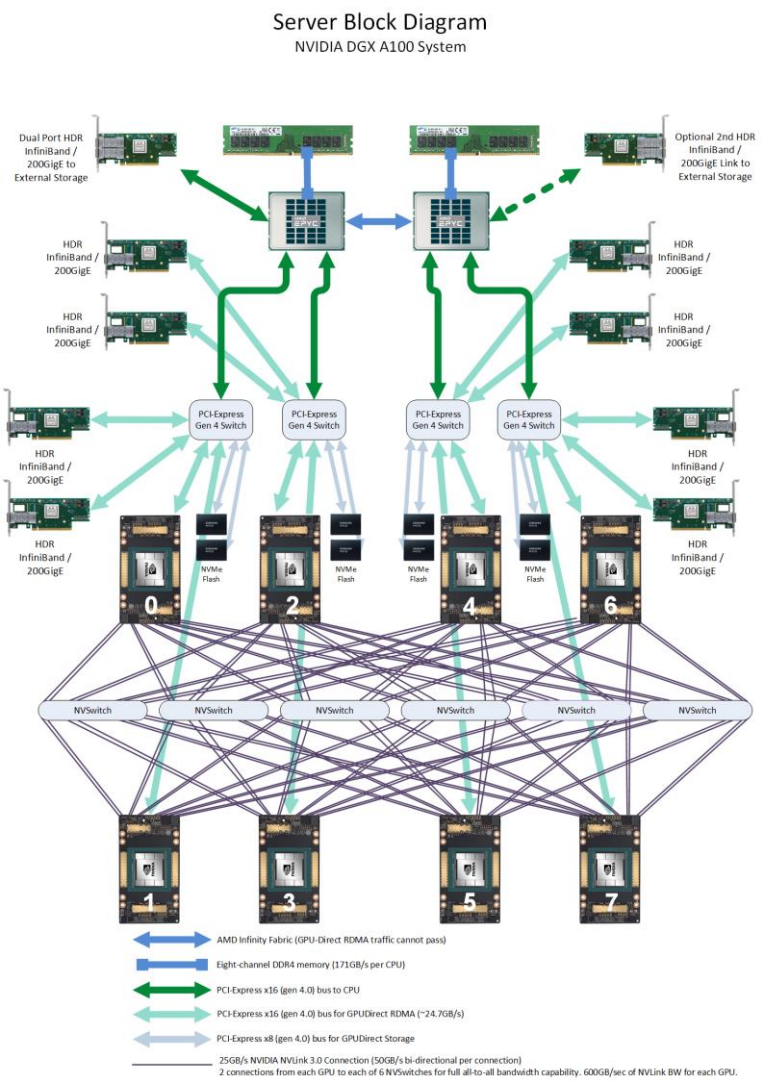
Cloud Innovator MEGAZONE

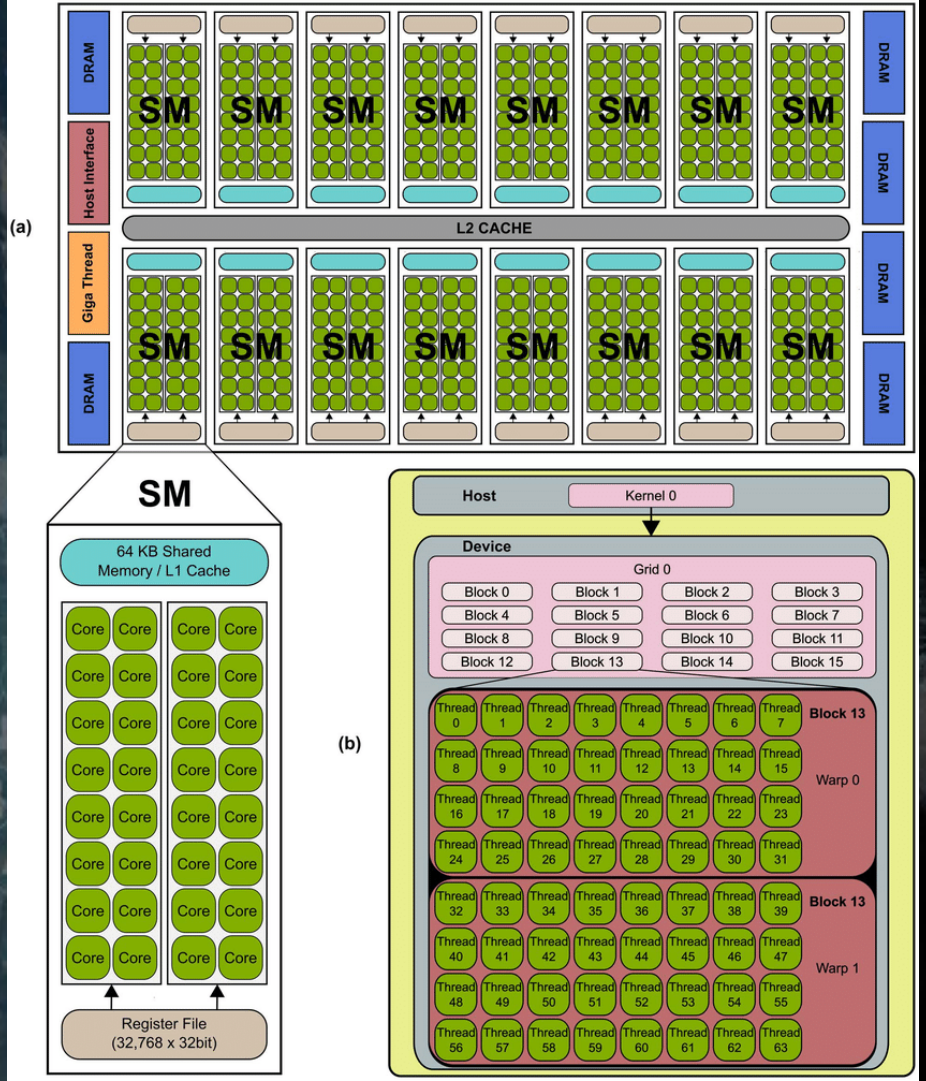
Hardware Diagram

- Dual Socket CPU and IOHUB

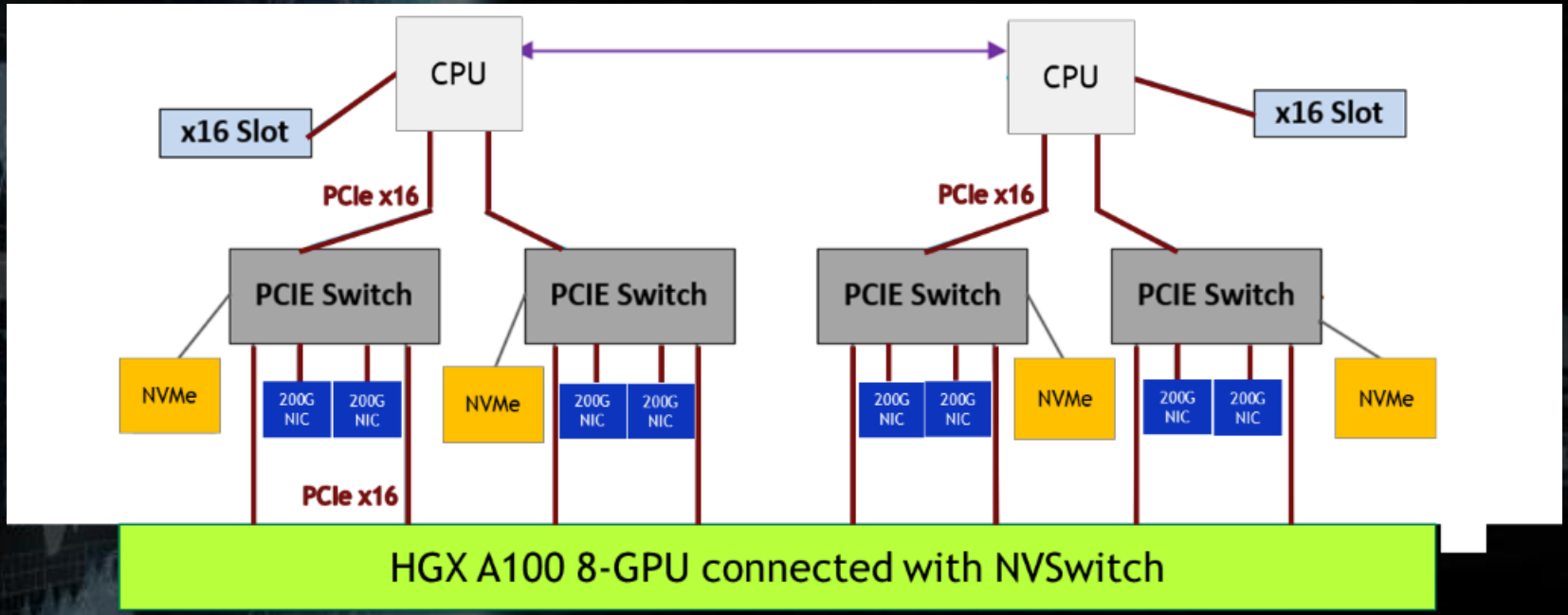


BMC: Baseboard Management Controller	PCH: Intel® Platform Controller Hub	IE: Innovation Engine
Intel® OPA: Intel® Omni-Path Architecture	Intel QAT: Intel® QuickAssist Technology	ME: Manageability Engine
NIC: Network Interface Controller	VMD: Volume Management Device	NTB: Non-Transparent Bridge





CPU PCIE, NVMe and NVSwitch



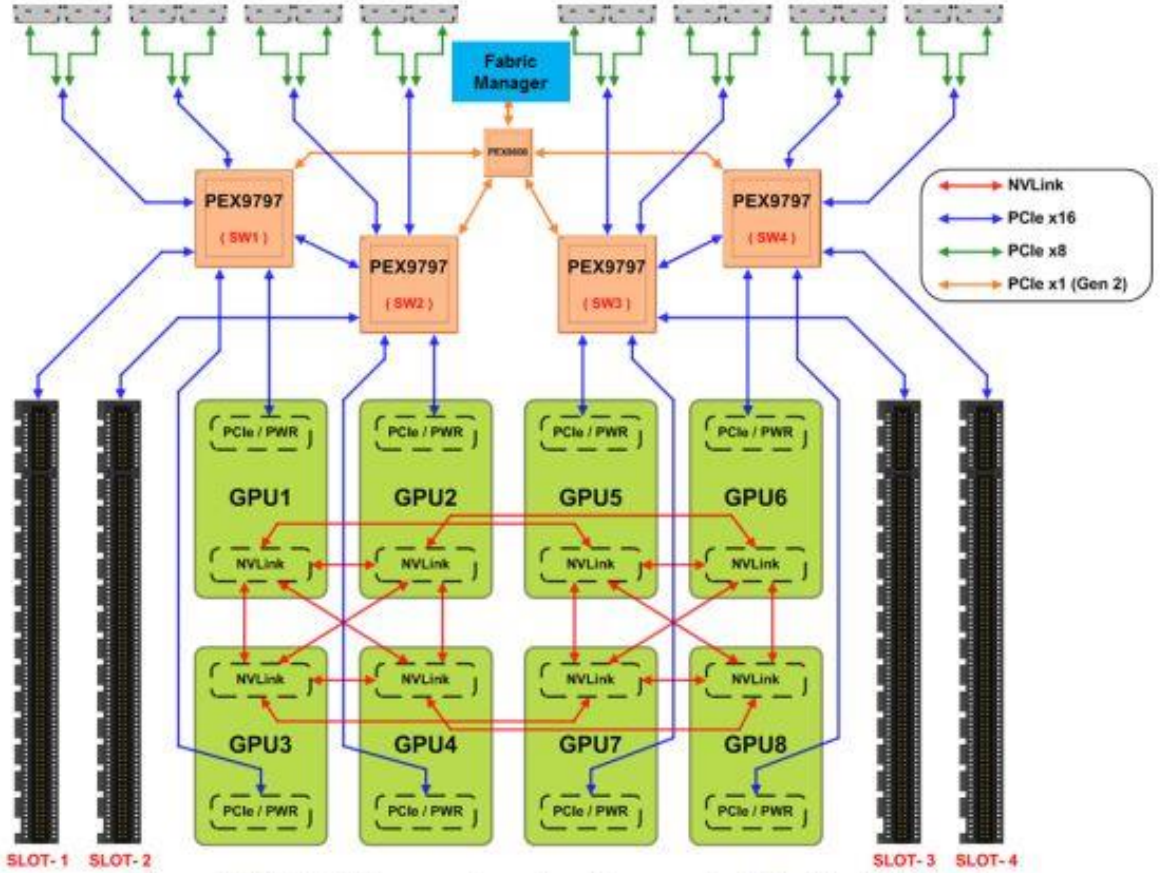


Figure 5 NVLink in Hyper-cube and cable-connected, flexible PCIe Topology

Programming Model

Parallel Architecture

Data Parallel Programming

- OpenMP (Open Multi-Processing)
  - Similar with OpenACC, OpenMP is an application programming interface (API) that supports multi-platform shared memory multi-processing programming in C, C++, and Fortran, on many platforms, instruction-set architectures and operating systems, including Solaris, AIX, FreeBSD, HP-UX, Linux, macOS, and Windows. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.
- OpenACC
  - Previous session

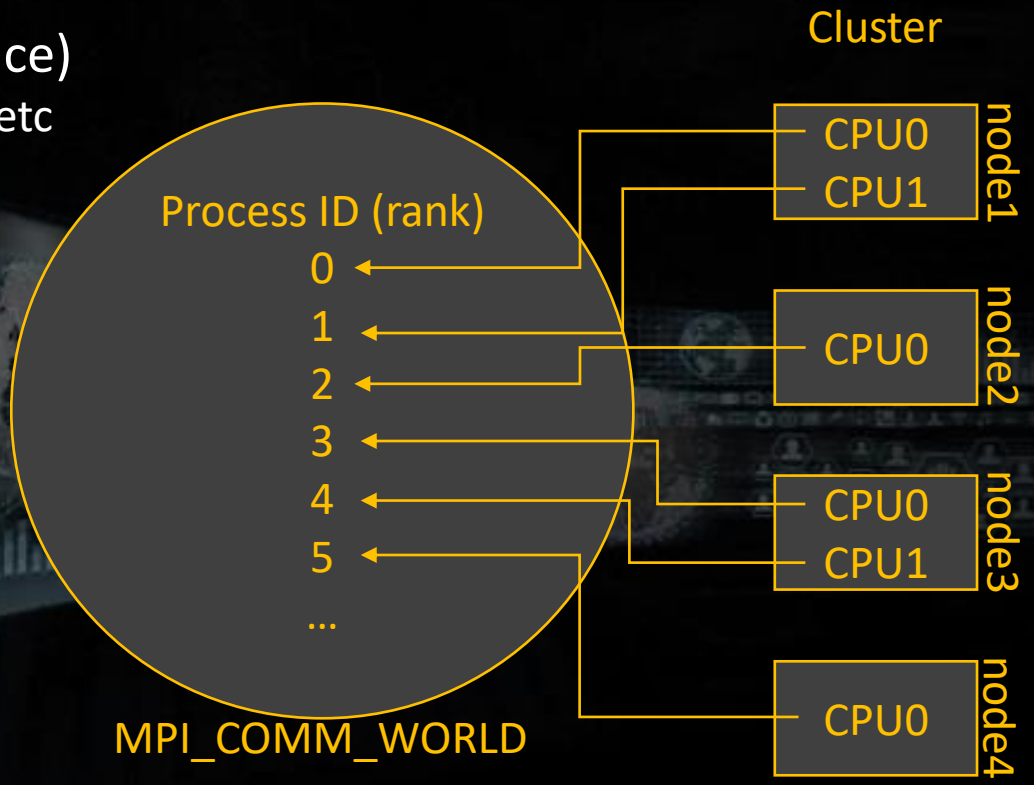
Message Passing Programming

- MPI(Message Passing Interface)
  - The message passing interface (MPI) is a standardized means of exchanging messages between multiple computers running a parallel program across distributed memory.
  - In parallel computing, multiple computers – or even multiple processor cores within the same computer – are called nodes. Each node in the parallel arrangement typically works on a portion of the overall computing problem.
  - The challenge then is to synchronize the actions of each parallel node, exchange data between nodes, and provide command and control over the entire parallel cluster.
  - The message passing interface defines a standard suite of functions for these tasks. The term message passing itself typically refers to the sending of a message to an object, parallel process, subroutine, function or thread, which is then used to start another process.

- CUDA
  - a proprietary and closed source parallel computing platform and application programming interface (API) that allows software to use certain types of graphics processing units (GPUs) for general purpose processing, an approach called general-purpose computing on GPUs (GPGPU).
  - CUDA is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels.

MPI: Message Passing Interface

- MPI(Message Passing Interface)
  - OpenMPI, MPICH, MVAPICH, etc
- APIs
  - MPI_SEND
  - MPI_RECEIVE
  - MPI_BROADCAST
  - MPI_BARRIER
  - MPI_WAIT
  - MPI_INIT
  - MPI_FINALIZE
  - etc



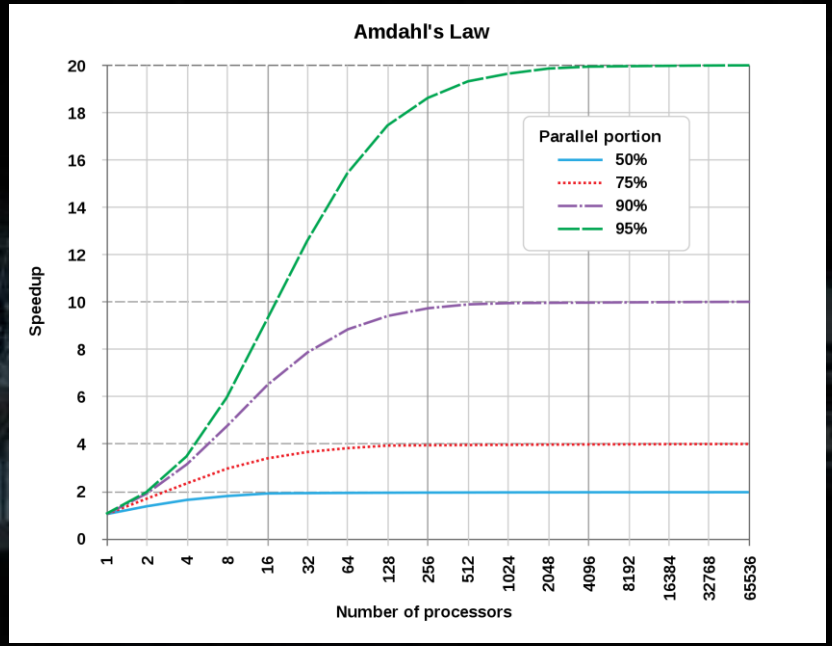
Scalability and Performance

- Amdahl's Law

$$S_{latency}(n) = \frac{1}{(1-p) + \frac{p}{n}}$$

- Theoretical ***speedup*** in latency o f the execution of a task at fixed workload that can be expected of a system whose resources are improved.

$$\lim_{n \to \infty} S_{latency}(n) = \frac{1}{(1-p)}$$

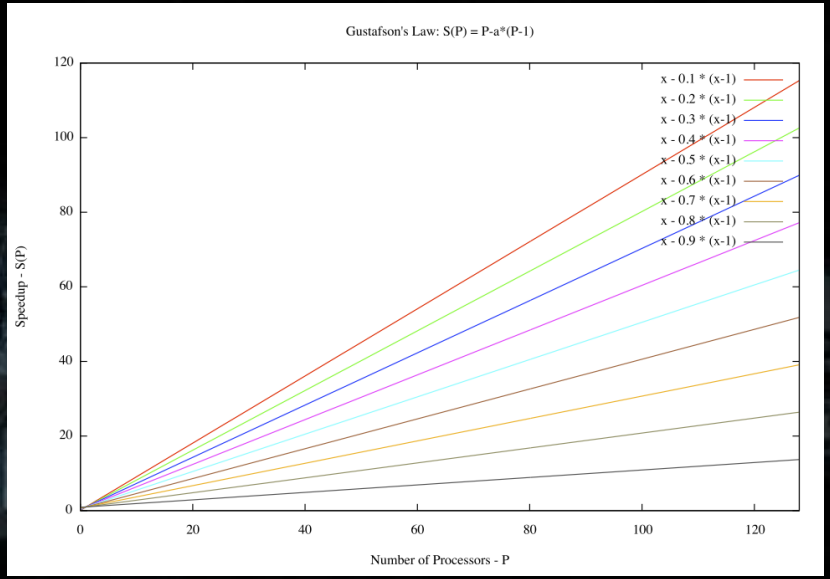


Scalability and Performance

- Gustafson's Law : Theoretical ***speedup***

$$S = s + p \times N$$
$$= s + (1 - s) \times N$$
$$= N + (1 - N) \times s$$
$$S = (1 - p) + p \times N$$
$$= 1 + (N - 1) \times p$$

- S and p are the fractions of time spent executing the serial parts and the parallel parts of the program on the parallel system, where $s+p=1$.
- Gustafson proposes that programmers tend to increase the size of problems to fully exploit the computing power that becomes available as the resources improve.



Computational Science

- FDM: Finite Difference Method
  - Implicit method and tridiagonal matrix solver

$$\frac{u_j^{n+1} - u_j^n}{k} = \frac{u_{j+1}^{n+1} - 2u_j^{n+1} + u_{j-1}^{n+1}}{h^2}$$

$$-ru_{j-1}^{n+1} + (1+2r)u_j^{n+1} - ru_{j+1}^{n+1} = u_j^n$$

- FVM: Finite Volume Method

$$\frac{dU_i}{dt} + \frac{1}{V_i}\oint_{\partial V_i} F(U_i)\cdot \hat{n}\, dS = 0$$

$$U_i^{n+1} - U_i^n = -\frac{\Delta t}{V_i}\sum_{j=1}^{N}\left[F\left(U_i^{n+1}\right) - F\left(U_j^{n+1}\right)\right]\Delta S_j$$

- FEM: Finite Element Method
  - Using weak formulation, the governing equation will be converted into Matrix form
- MC: Monte Carlo Methods
  - a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results.

Seven Dwarfs

of Scientific Computing Applications

1. Dense Linear Algebra
   (e.g., BLAS [Blackford et al 2002], ScaLAPACK [Blackford et al 1996], …)
2. Sparse Linear Algebra
   (e.g., SpMV, OSKI [OSKI 2006], or SuperLU [Demmel et al 1999])
3. Spectral Methods
   (e.g., FFT [Cooley and Tukey 1965])
4. N-Body Methods
   (e.g., Barnes-Hut [Barnes and Hut 1986], Fast Multipole Method [Greengard and Rokhlin 1987])
5. Structured Grids
   (e.g., Cactus [Goodale et al 2003] or LatticeBoltzmann Magnetohydrodynamics [LBMHD 2005])
6. Unstructured Grids
   (e.g., ABAQUS [ABAQUS 2006] or FIDAP [FLUENT 2006])
7. Monte Carlo
   (e.g., Quantum Monte Carlo [Aspuru-Guzik et al 2005])

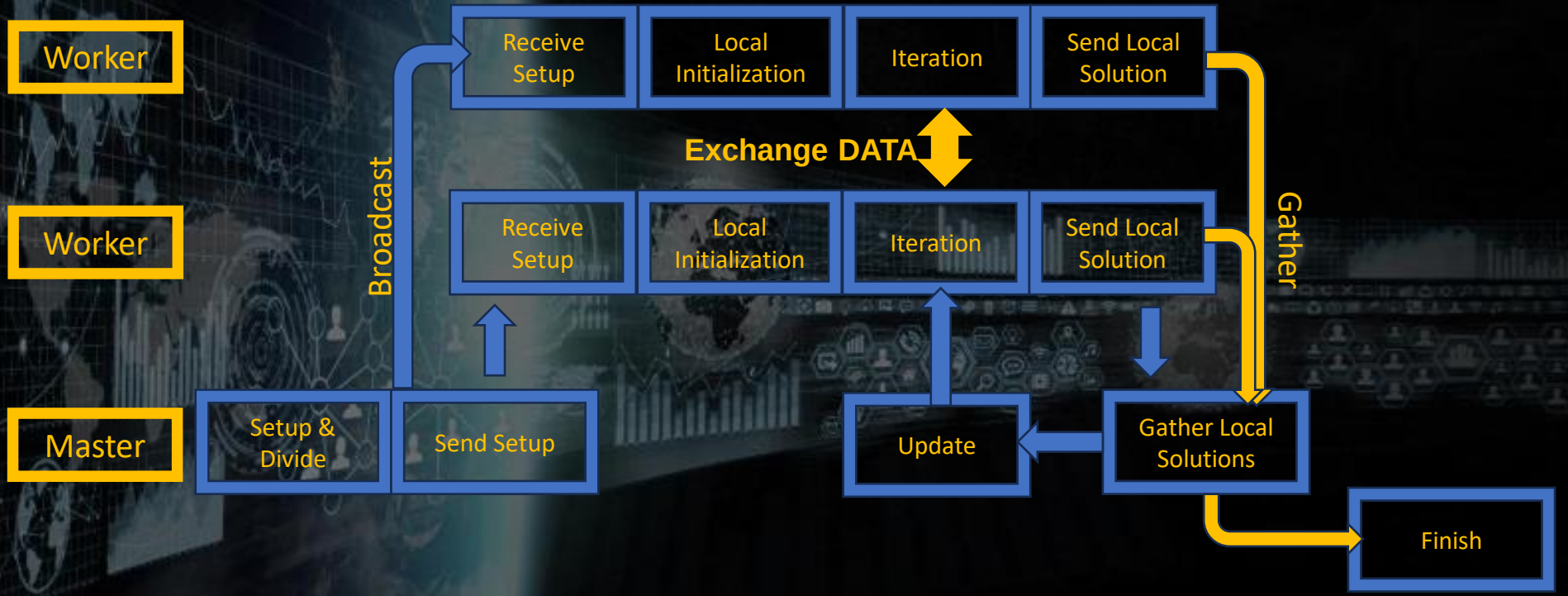
https://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf

Seven Dwarfs and More

- SPEC2006
  - **Combinational Logic:** generally involves performing simple operations on very large amounts of data often exploiting bit-level parallelism. For example, computing Cyclic Redundancy Codes (CRC) is critical to ensure integrity and RSA encryption for data security.
  - **Graph Traversal** applications must traverse a number of objects and examine characteristics of those objects such as would be used for search. It typically involves indirect table lookups and little computation.
  - **Graphical Models** applications involve graphs that represent random variables as nodes and conditional dependencies as edges. Examples include Bayesian networks and Hidden Markov Models.
  - **Finite State Machines** represent an interconnected set of states, such as would be used for parsing. Some state machines can decompose into multiple simultaneously active state machines that can act in parallel.
- Machine Learning
  - **Dynamic programming** is an algorithmic technique that computes solutions by solving simpler overlapping subproblems. It is particularly applicable for optimization problems where the optimal result for a problem is built up from the optimal result for the subproblems.
  - **Backtrack and Branch-and-Bound:** These involve solving various search and global optimization problems for intractably large spaces. Some implicit method is required in order to rule out regions of the search space that contain no interesting solutions. Branch and bound algorithms work by the divide and conquer principle: the search space is subdivided into smaller subregions ("branching"), and bounds are found on all the solutions contained in each subregion under consideration.
- Traditional ML
  - Support Vector Machines [Cristianini and Shawe-Taylor 2000]: Dense linear algebra.
  - Principal Component Analysis [Duda and Hart 1973]: Dense or sparse linear algebra, depending on the details of implementation.
  - Decision Trees [Poole et al 1998]: Graph traversal.
  - Hashing: Combinational logic.

Parallel Computational Science

Workflow



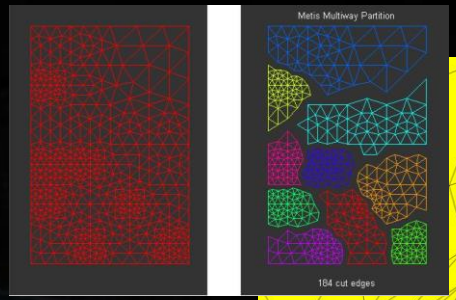
Cost of Parallel Computing

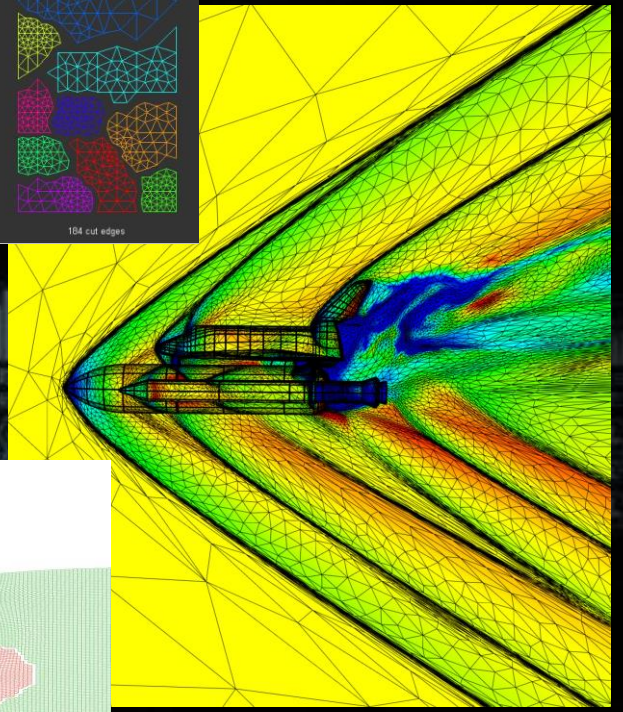
$$Cost = Cost(Computation + Communications)$$

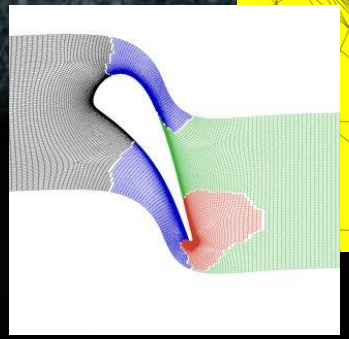
- Goal: Maximize Core usage & Minimize Cost of communication
  - If computation load is imbalanced, some processor have to be wait to exchange data to go next steps.
    1. Computation Loads have to balanced
    2. Communication must be minimized in view of data size and frequency of exchange
       - Communications = overhead due to parallel processing
  - For overcome the communication issues,
    - Use the High bandwidth network
    - Use the low latency network
    - Reduce the communication data size with good partitioner
- In other words, divide work evenly and minimize communication

Parallel Computational Science

- Mesh/Graph Partitioning
  - Computational mesh/domain decomposition
  - e.g. Metis, hMetis, ParMetis, Chaco,
- Parallel Linear Solvers
  - Many publicly available packages available nowadays
  - PETSc, Diffpack, FEMLib, AZTEC, PSPARSKIT, …
- re-Partitioning or re-balancing
  - Automatic Mesh Refinement
  - Adaptive Mesh Refinement

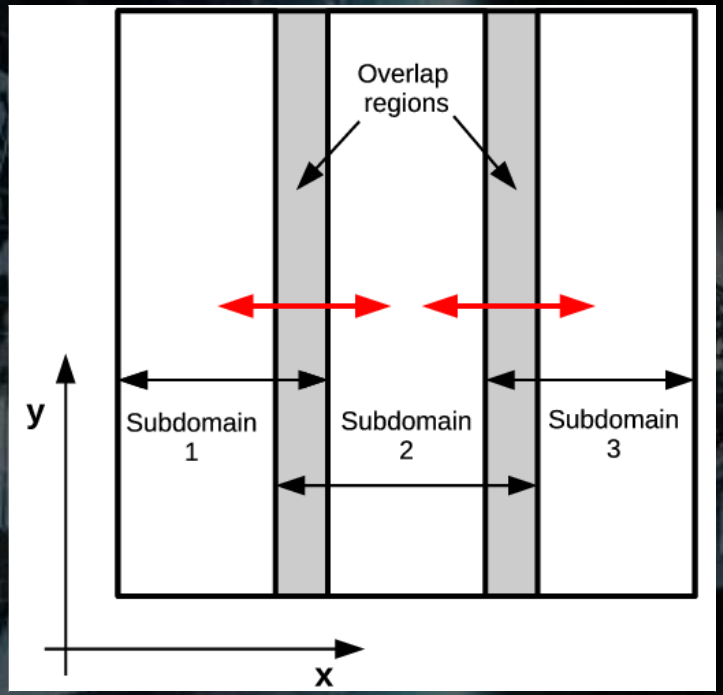


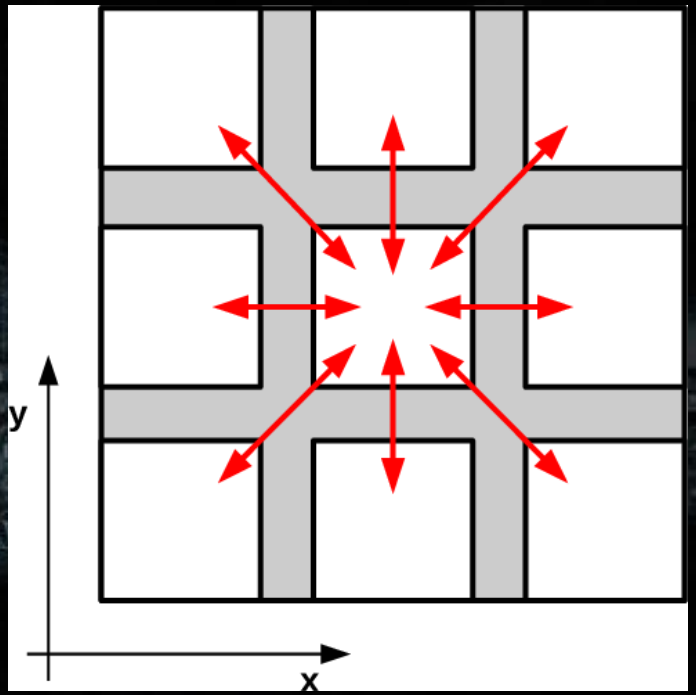




Domain Decomposition

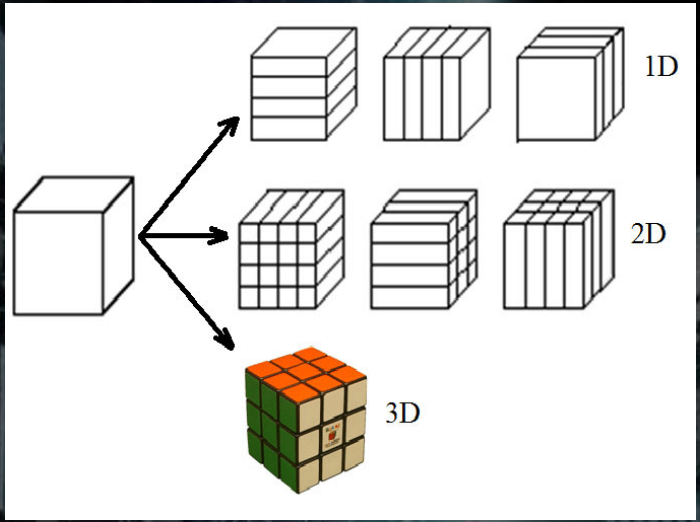
1D, 2D: Algorithm Dependent Decomposition for Structured Grids

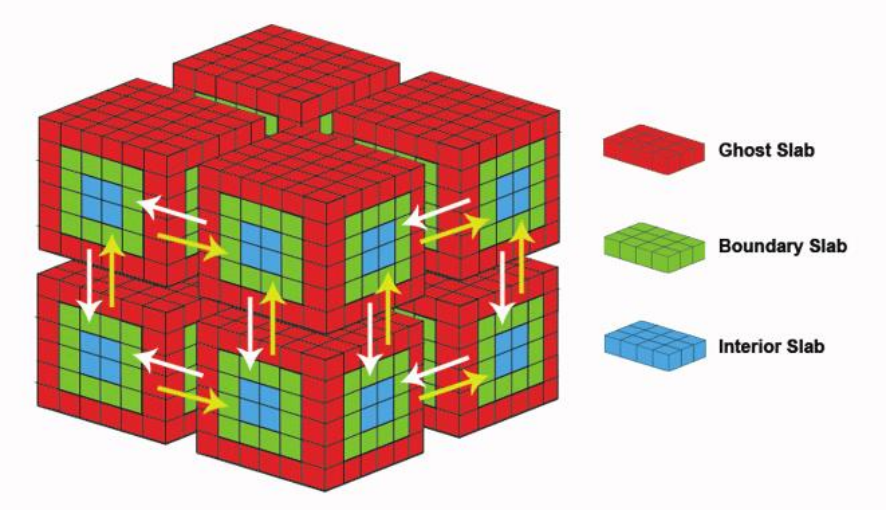




Domain Decomposition

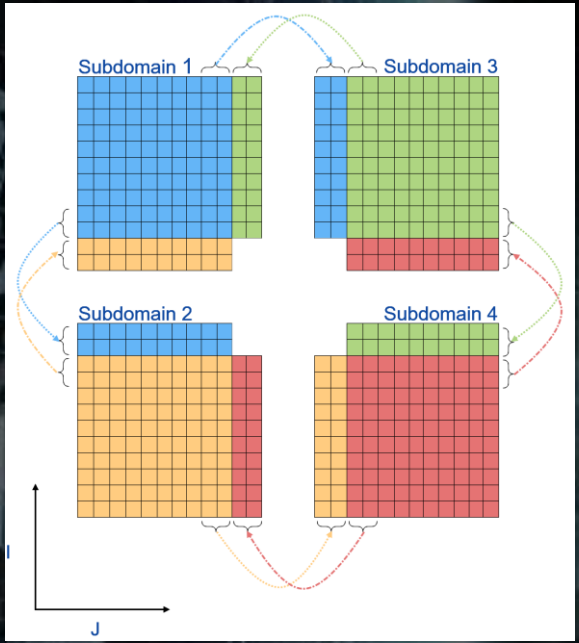
3D for Structured Grids





Domain Decomposition

2D: Higher Order Algorithm Dependent Decomposition for Structured Grids



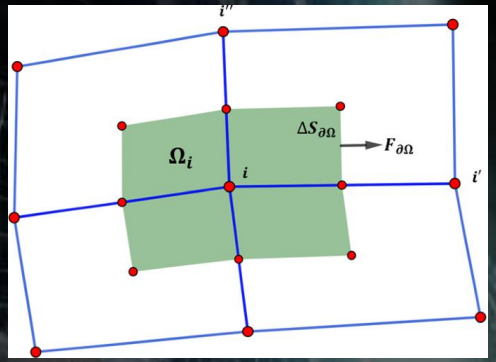
- 2nd Order Accurate Numerical Schemes

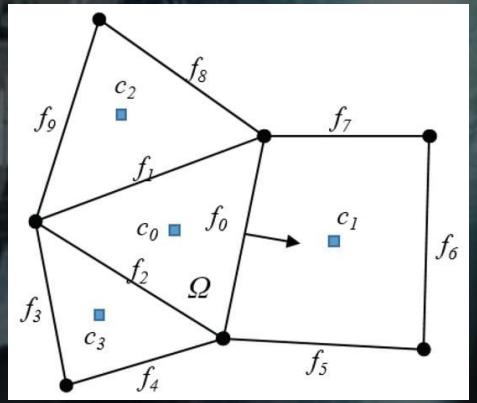
$$A_{ij} = \sum f\left(A_{i-2,j}, A_{i-1,j}, A_{i+1,j} A_{i+2,j}\right)$$

$$+ \sum f\left(A_{i,j-2}, A_{i,j-1}, A_{i,j+1} A_{i,j+2}\right)$$

Governing Equations of Fluid Dynamics

2D Finite Volume Methods





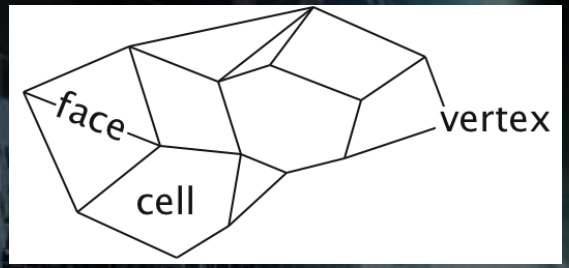
- The typical physical governing equations for Euler, Navier-Stokes, etc. is:

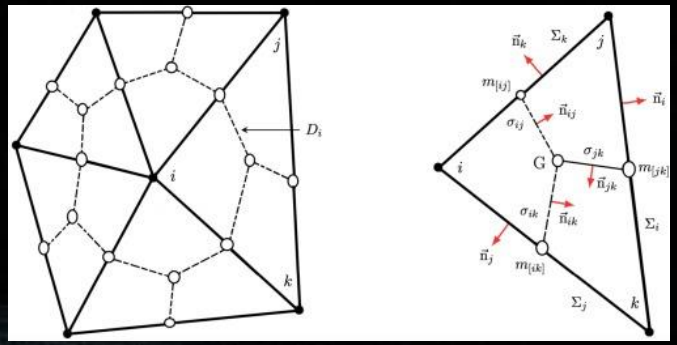
$$\frac{\partial}{\partial t}\int_{\Omega} U dV + \oint_{\partial\Omega} \boldsymbol{F}\cdot d\boldsymbol{S} = 0$$

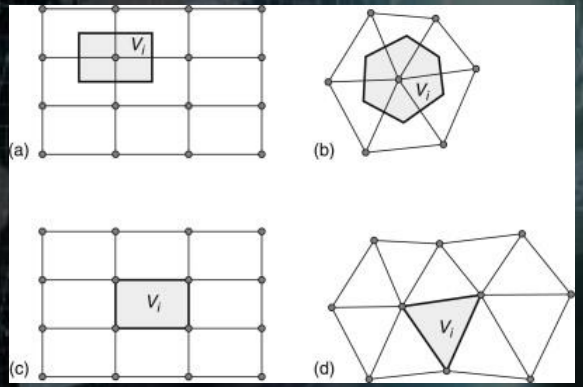
$$U = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho E \end{bmatrix} \quad F_x = \begin{bmatrix} \rho u \\ \rho u u^2 + p \\ \rho v u \\ \rho u H \end{bmatrix} \quad F_x = \begin{bmatrix} \rho v \\ \rho u v \\ \rho v^2 + p \\ \rho u H \end{bmatrix}$$

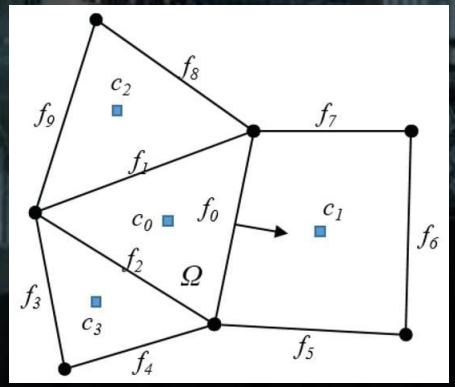
$$\frac{\Omega_i}{\Delta t}\left(U_i^{n+1} - U_i^n\right) = -\sum_{\partial\Omega} \boldsymbol{F}_{i,\partial\Omega}^{n+1}\cdot\Delta S_{i,\partial\Omega}^{n+1}$$

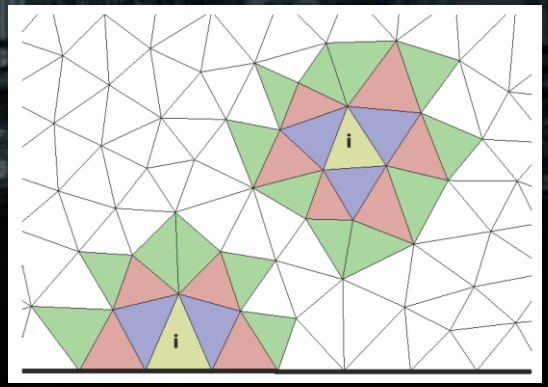
FVM for Unstructured Grids





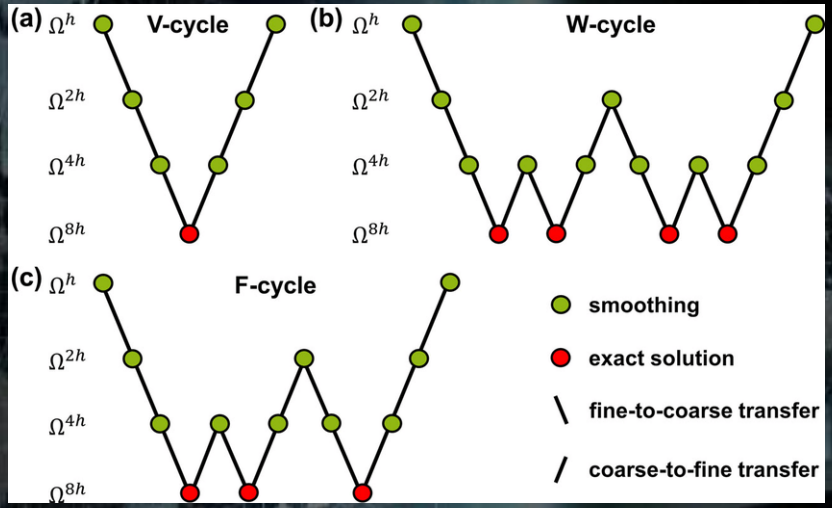


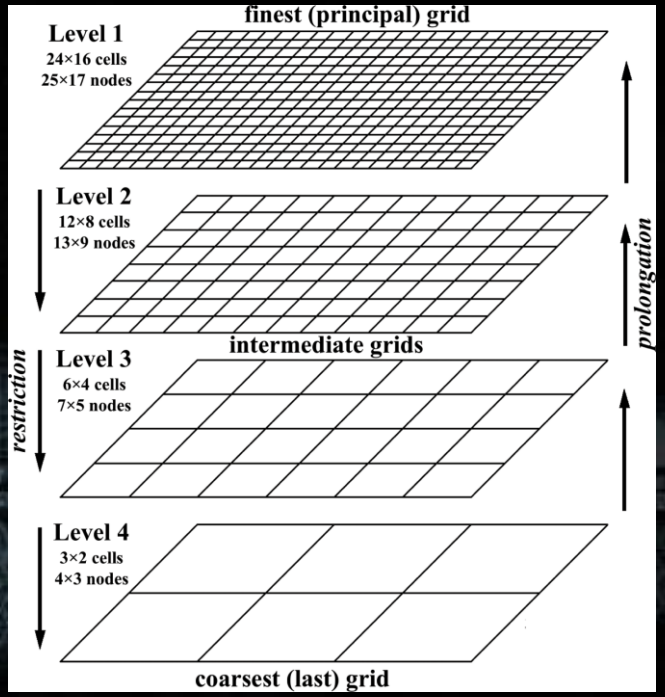




Multigrid Methods

for accelerating computation





https://developer.nvidia.com/blog/high-performance-geometric-multi-grid-gpu-acceleration/

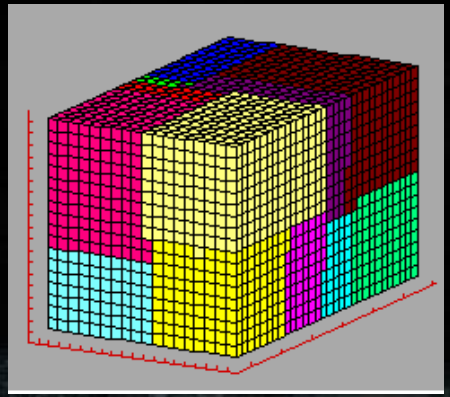
Graph Partitioning

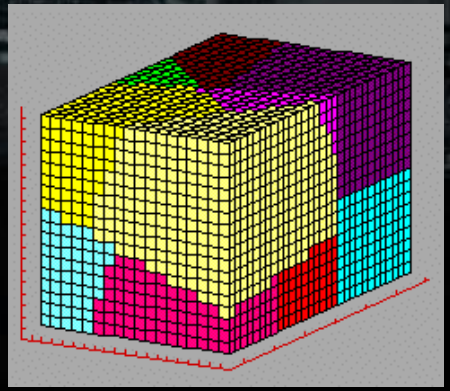
- Given a graph G = (N, E, WN, WE)
  - N = nodes (or vertices)
  - E = edges
  - W_N = node weights,
  - W_E = edge weights 3

- N can be thought of as tasks, W_N are the task costs, edge (j, k) in E means task j sends $W_E(j, k)$ words to task k
- Choose a partition $N = U_1 \cup U_2 \cup U_3 \cup \cdots \cup U_P$ such that
  - The sum of the node weights in each N_j is distributed evenly (load balance)
  - The sum of all edge weights of edges connecting all different partitions is minimized (decrease parallel overhead)
- In other words, divide work evenly and minimize communication

Domain Decomposition Methods

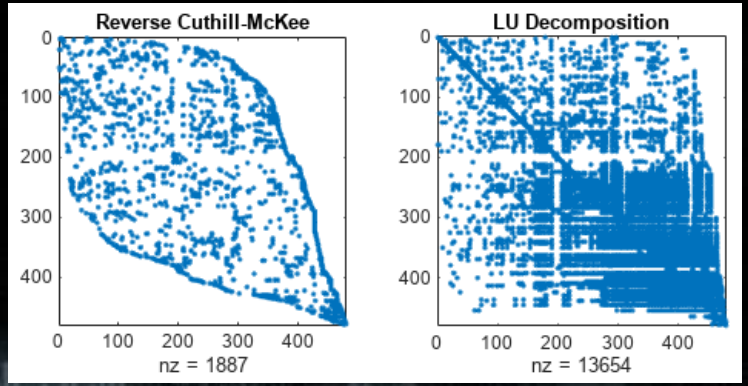
- Recursive Coordinate Bisection
  - Divide work into two equal parts using cutting plane orthogonal to coordinate axis For good aspect ratios cut in longest dimension.
- Recursive Inertial Bisection
  - For domains not oriented along coordinate axes can do better if account for the angle of orientation of the mesh.
  - Use bisection line orthogonal to principal inertial axis (treat mesh elements as point masses).
  - Project centers-of-mass onto this axis; bisect this ordered list. Typically gives smaller subdomain boundary.





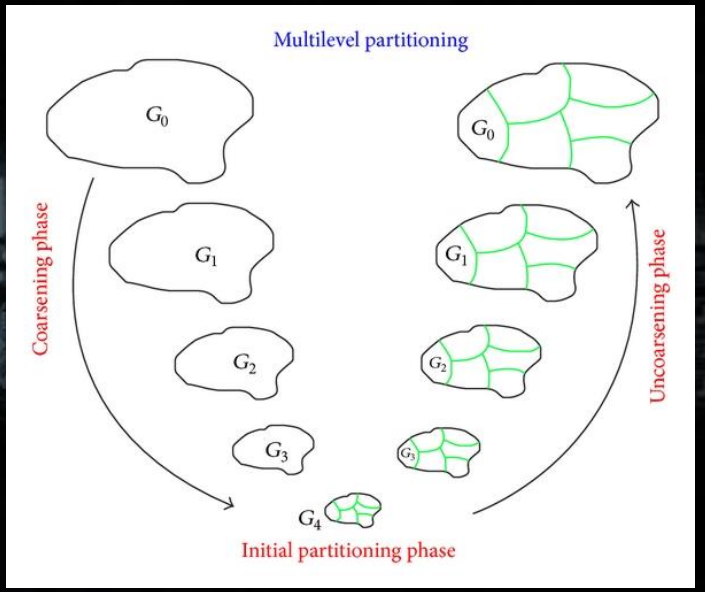
Domain Decomposition Methods

- Reverse Cuthill-McKee
  - The Cuthill-McKee algorithm is used for reordering of a symmetric square matrix. It is based on Breadth First Search algorithm of a graph, whose adjacency matrix is the sparsified version of the input square matrix.
  - The ordering is frequently used when a matrix is to be generated whose rows and columns are numbered according to the numbering of the nodes. By an appropriate renumbering of the nodes, it is often possible to produce a matrix with a much smaller bandwidth.
- Graph Partitioning for Sparse Matrix Factorization
  - Nested dissection for fill-reducing orderings for sparse matrix factorizations.
  - Recursively repeat:
    - Compute vertex separator, bisect graph,
    - edge separator = smallest subset of edges such that removing them divided graph into 2 disconnected subgraphs)
    - vertex separator = can extend edge separator by connecting each edge to one vertex, or compute directly.
    - Split a graph into roughly equal halves using the vertex separator
  - At each level of recursion number the vertices of the partitions, number the separator vertices last. Unknowns ordered from n to 1.

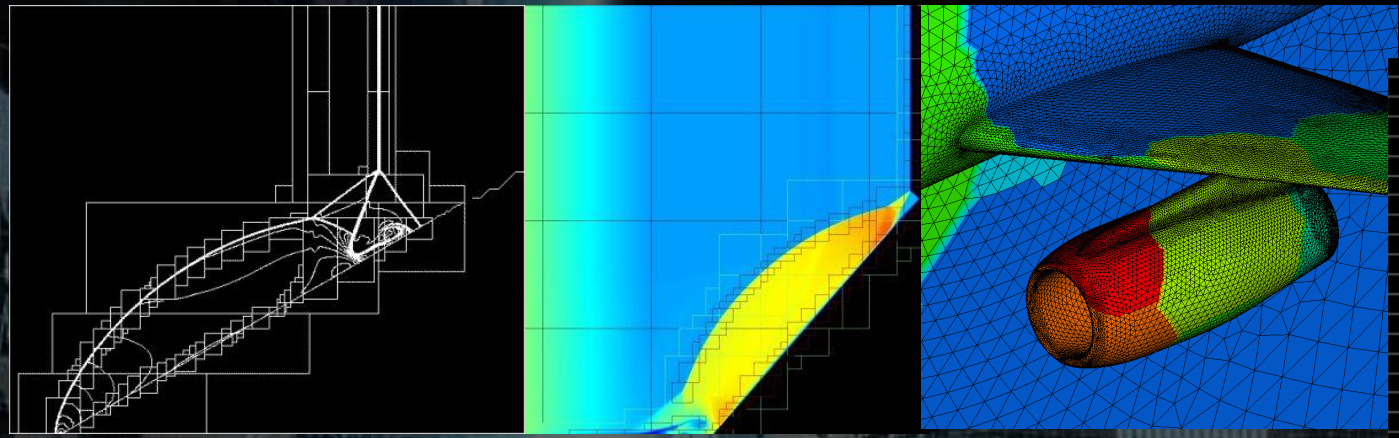


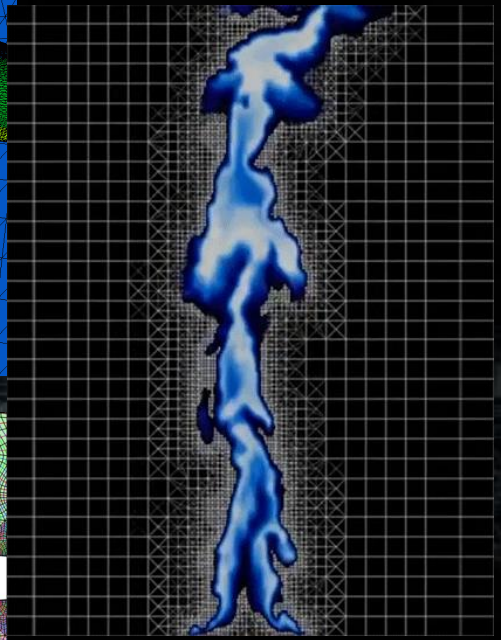
Domain Decomposition Methods

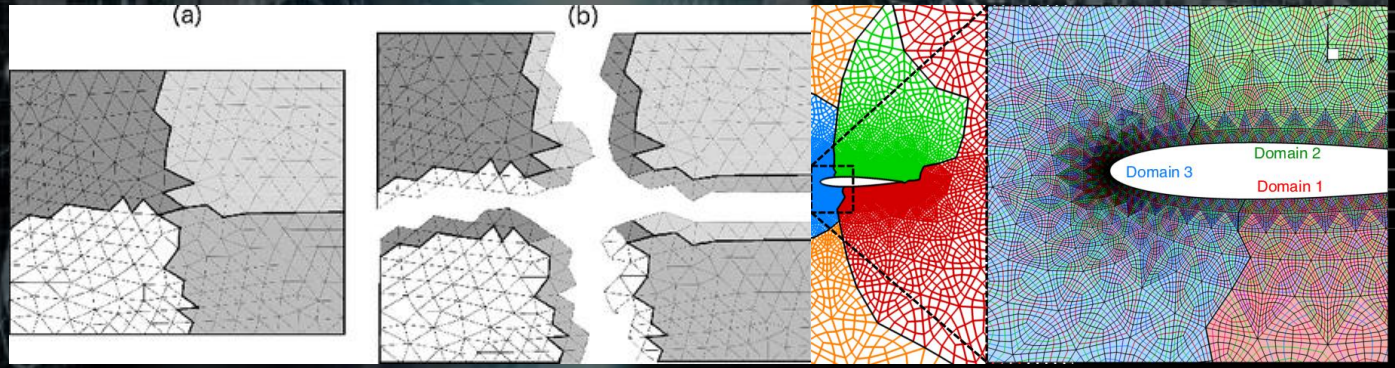
- Recursive Spectral Bisection
  - The recursive spectral bisection (RSB) algorithm was proposed by Pothen and Simon as the basis for computing small vertex separators for sparse matrices.
  - Simon applied this algorithm to mesh decomposition and showed that spectral bisection compared favorably with other decomposition techniques.
  - Since then, the RSB algorithm has been widely accepted in the scientific community because of its robustness and its consistency in the high-quality partitionings it generates.
- Multilevel Graph Partitioning
  - Coarsen graph → Partition the coarse graph → Refine graph, using local refinement algorithm
    - vertices in larger graph assigned to same set as coarser graph's vertex.
    - since vertex weight conserved, balance preserved
    - similarly for edge weights
  - METIS is a family of programs for partitioning unstructured graphs and hypergraphs and computing fill-reducing orderings of sparse matrices.



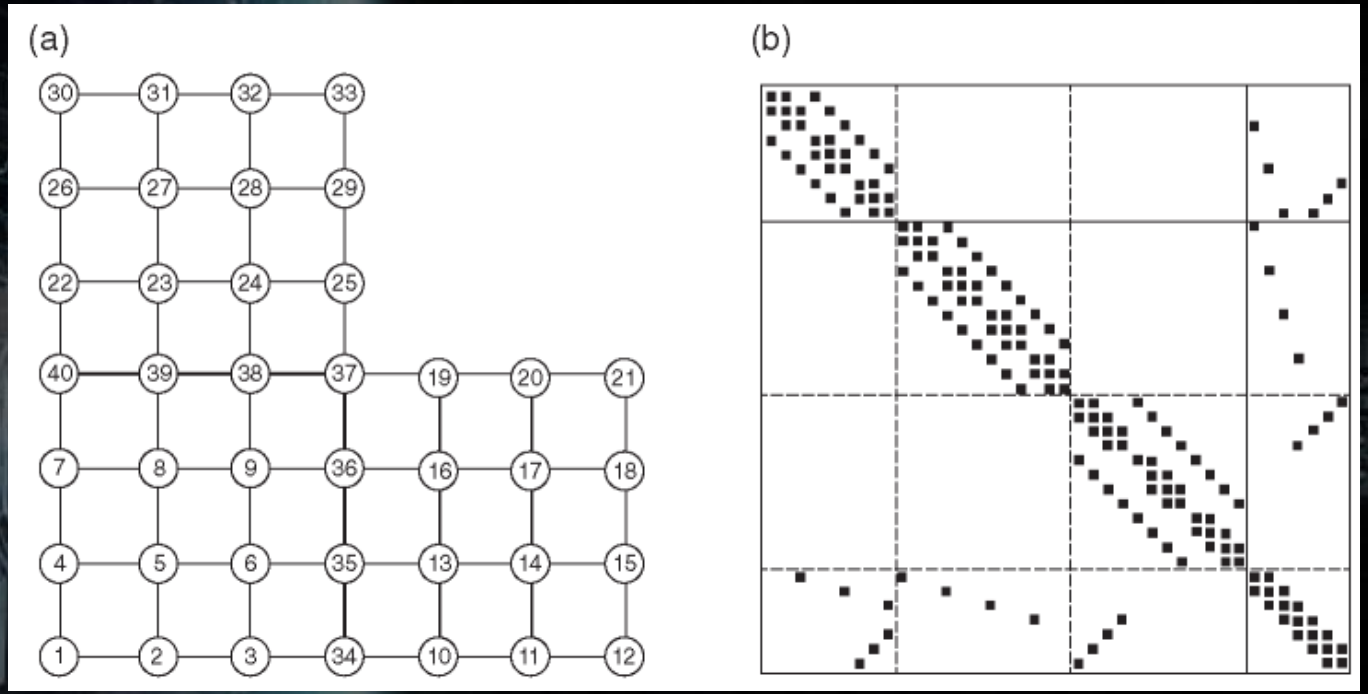
Domain Decomposition of Unstructured Grids







Domain Decomposition for Finite Element Methods



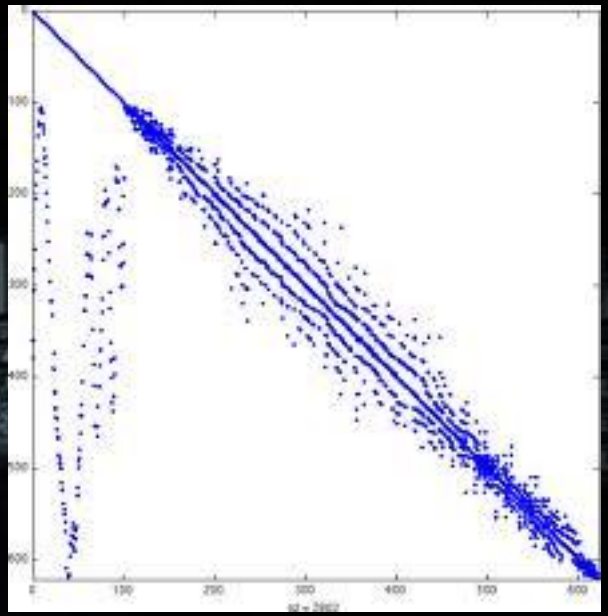
Parallel Matrix Solver

Solving Ax = b in parallel

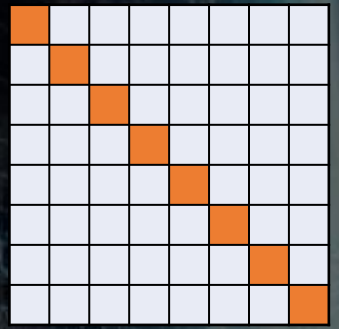
- Dense A: Gaussian elimination with partial pivoting (LU)
  - Same flavor as matrix x matrix, but more complicated
- Sparse A: Gaussian elimination – Cholesky, LU, etc.
  - Graph algorithms
- Sparse A: Iterative methods – Conjugate gradient, etc.
  - Sparse matrix times dense vector
- Sparse A: Preconditioned iterative methods and multigrid
  - Mixture of lots of things

What is a sparse matrix?

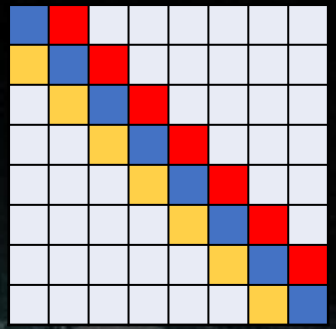
- A matrix with primarily zeros (>> 90%)
- Representing them using dense data structures wastes
  - Memory
  - Computation
- Sparse matrices arise in many applications
  - Simulating climate
  - Analyzing images (photos, MRIs,…)
  - Web page ranking for search
  - Graphs, including Graph Neural Nets



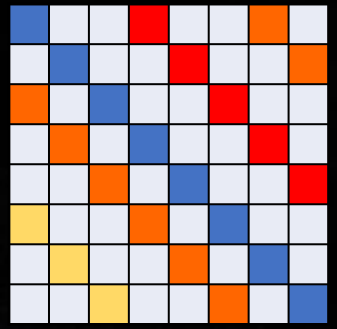
Examples of sparse matrices



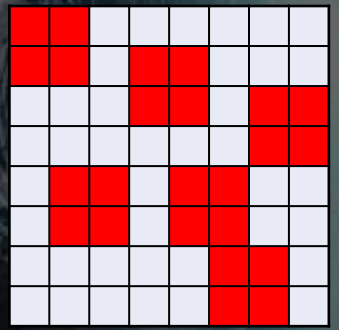
Diagonal



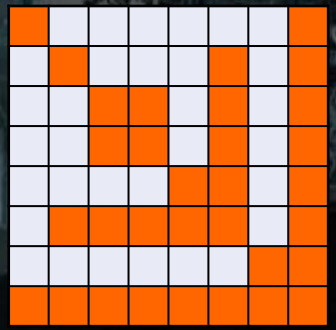
Tridiagonal



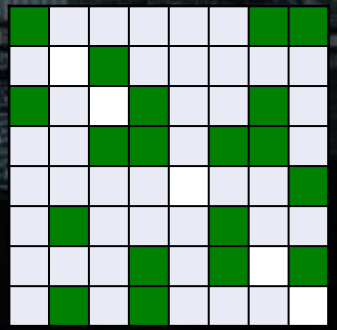
Generalized diagonal



Block



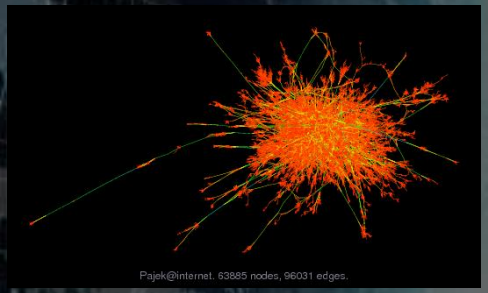
Symmetric

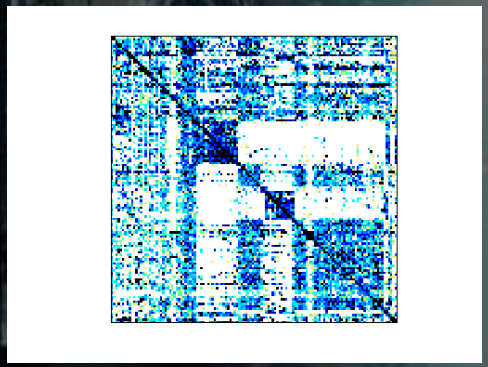


irregular

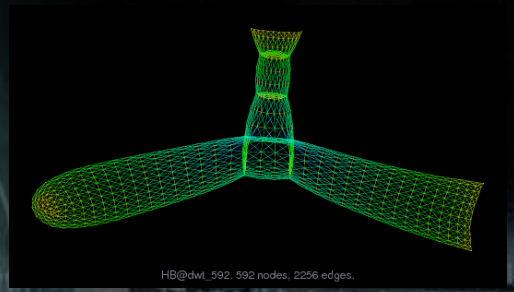
Sparse matrices are everywhere

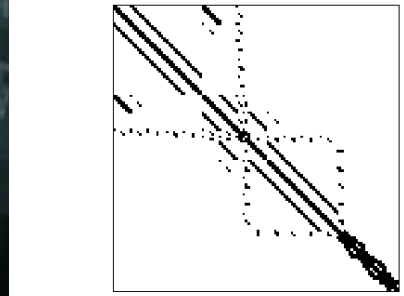
Internet connectivity



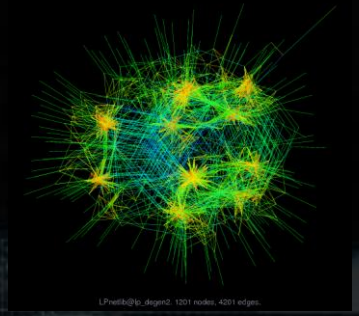


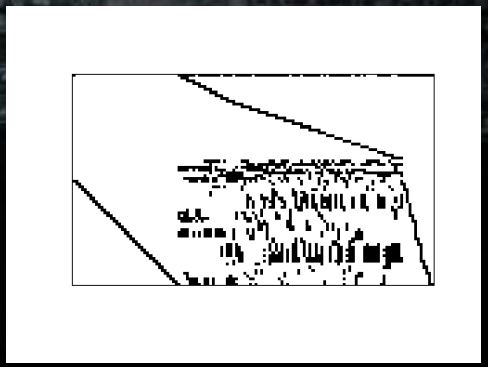
Structural design





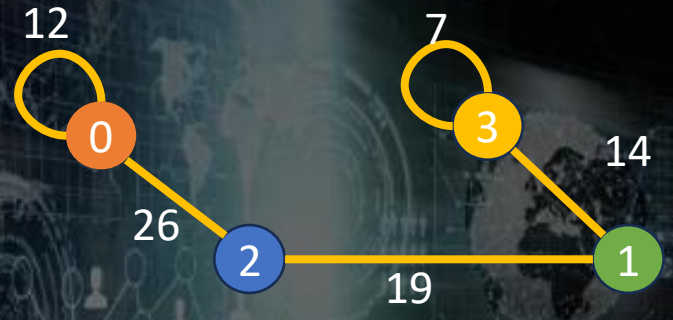
Linear Programming





Graph representations

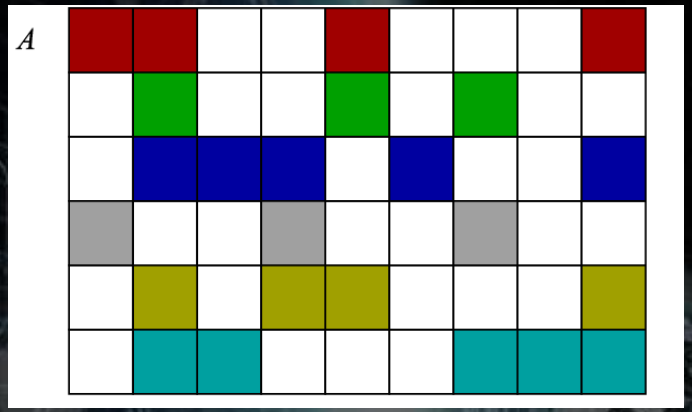
- Compressed sparse row (CSR) = compressed version of dense adjacency matrix

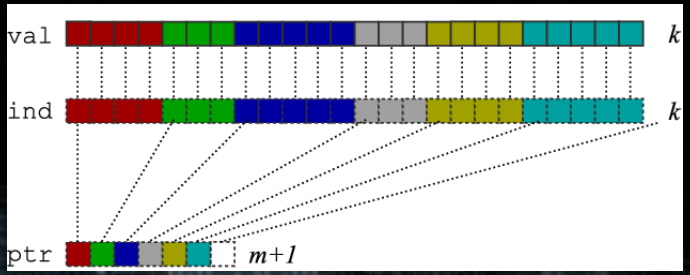


	0	1	2	3
0	12	0	26	0
1	0	0	19	14
2	26	19	0	0
3	0	14	0	7

Index into adj array	0	2	2	3	5	(row starts in CSR)
adjacencies	0	2	1	1	3	(column ids in CSR)
weights	12	26	19	14	7	(values in CSR)

Compressed Sparse Row (CSR) Storage





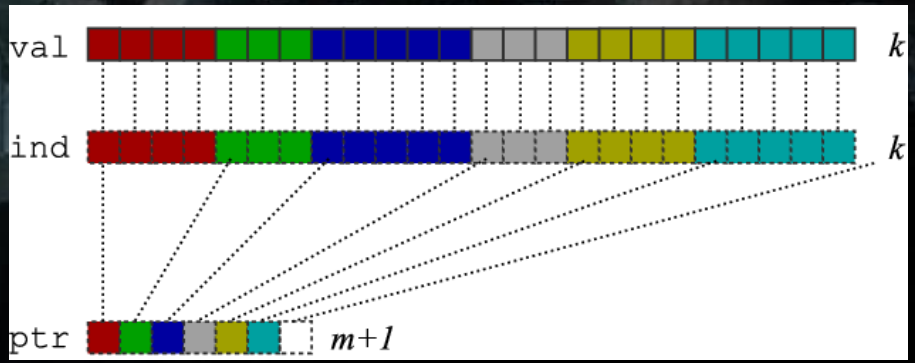
CSR has:

- Size nnz = number of nonzeros
- Array of the nonzero values (val) of size nnz
- Array of the column indices for each value of size nnz
- Array of row start pointers of size n = number of row

- CSC, COO, DIAG, etc.

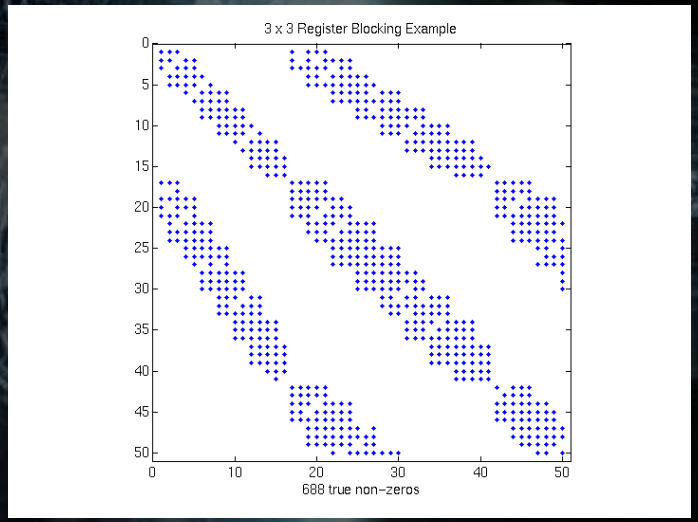
SpMV in CSR: OpenMP Parallel

```
#pragma omp parallel num_threads(thread_num)
{
#pragma omp for private(j, i, tmp) schedule(static)
  for (int i=0; i<m; i++)  {
     for (j = ptr[i]; j < ptr[i+1]; j++)  {
        tmp = ind[j];
        y[i] += val[j] * x[tmp];
     }
  }
}
```

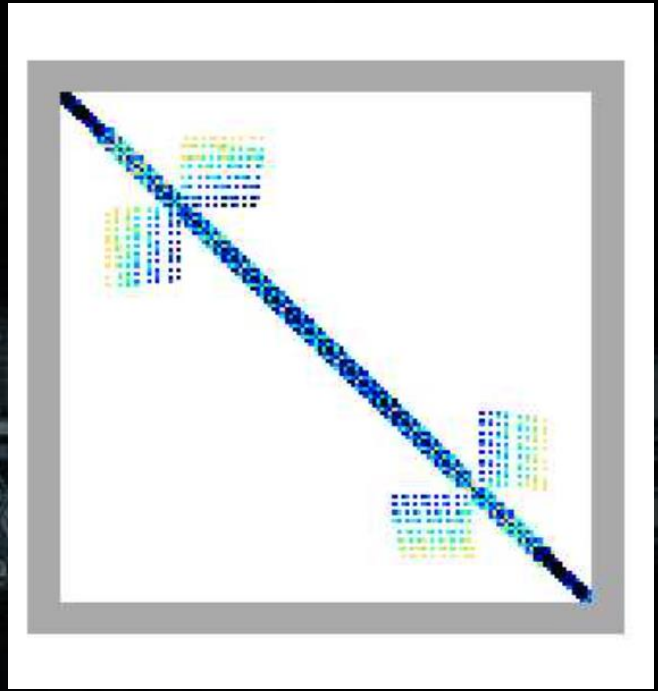


But most matrices don't block so easily

- FEM Fluid dynamics problems
- More complicated non-zero structure in general

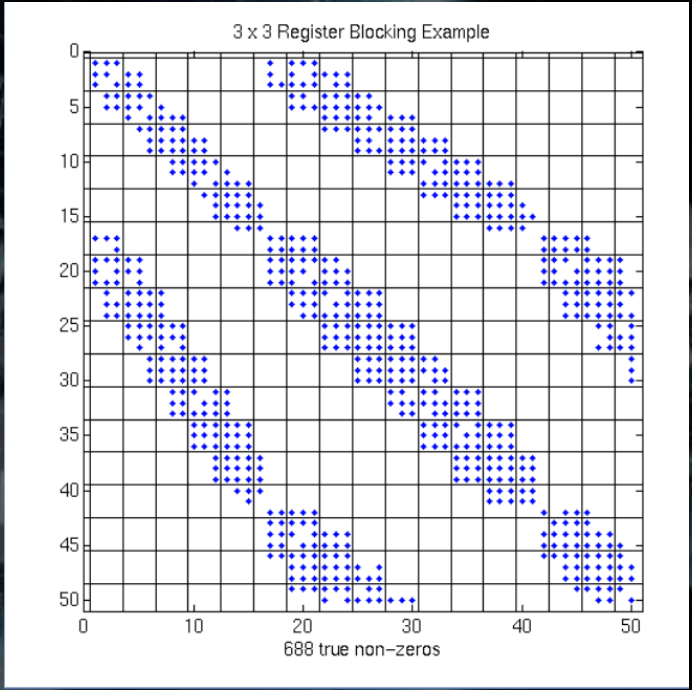


Zoom in to top corner



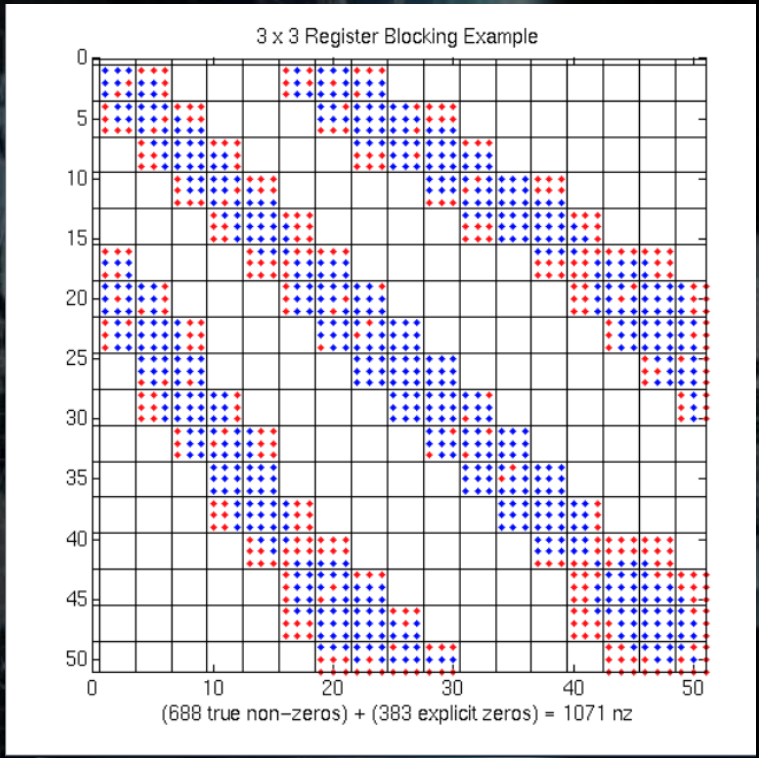
- N = 16614
- NNZ = 1.1M

3x3 blocks look natural, but…



- More complicated non-zero structure
- Example: 3x3 blocks
  - Grid of 3x3 cells
  - Many cell are not full
- N = 16614
- NNZ = 1.1M

3x3 blocks look natural, but…



- More complicated non-zero structure
- Example: 3x3 blocks
  - Grid of 3x3 cells
  - Add explicit zeros: 1.5x "fill overhead"
  - Unroll loops
- More work but faster

- N = 16614
- NNZ = 1.1M

Sparse linear Solver

Solving Ax = b

- Direct methods of factorization
  - For solving Ax = b, least squares problems
    - Cholesky, LU, QR, LDLT factorizations
  - Limited by fill-in/memory consumption and scalability
- Iterative solvers
  - For solving Ax = b, least squares, Ax = λx, SVD
  - When only multiplying A by a vector is possible
  - Limited by accuracy/convergence
- Hybrid methods
  - As domain decomposition methods

Krylov subspace methods

- Solve $Ax = b$ by finding a sequence $x_1, x_2, \ldots, x_k$ that minimizes some measure of error over the corresponding spaces

$$x_0 = \mathcal{K}_i(A, r_0), \qquad i = 1, \ldots, k$$

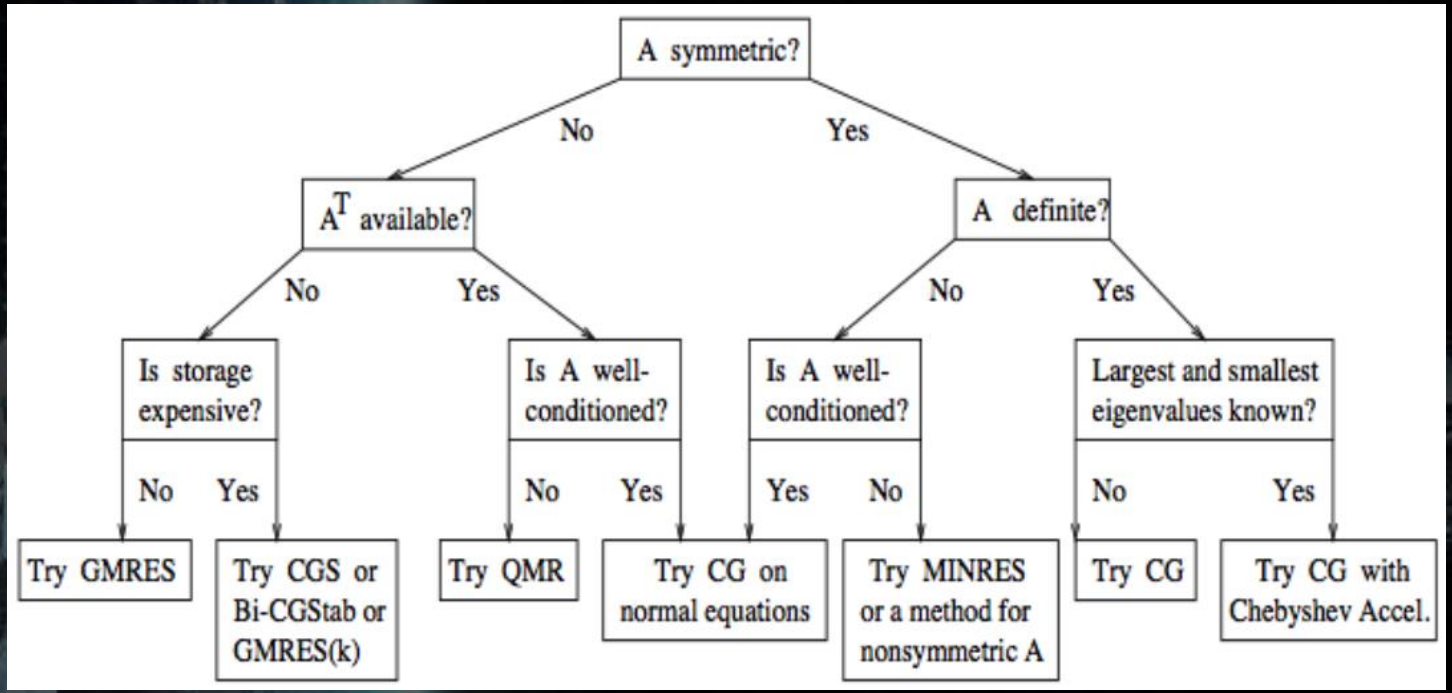
- They are defined by two conditions:
  1. Subspace condition: $x_0 \in x_0 + \mathcal{K}_k(A, r_0)$
  2. Petrov-Galerkin condition: $r_k \perp \mathcal{L}_k$

$$\Leftrightarrow (r_k)^t y = 0, \forall y \in \mathcal{L}_k$$

where

- x_0 is the initial iterate, r_0 is the initial residual,
- $\mathcal{K}_k(A, r_0) = span\{r_0, Ar_0, A^2 r_0, \ldots, A^{k-1} r_0\}$ is the Krylov subspace of dimension k
- $\mathcal{L}_k$ is a well-defined subspace of dimension k.

Choosing a Krylov method



Ref. www.netlib.org

Preconditioning – Basic principles

- Basic idea is to use the Krylov subspace method on a modified system such as

$$M^{-1}Ax = M^{-1}b$$

The matrix $M^{-1}A$ need not be formed explicitly; only need to solve whenever needed.

$$M\,w = v$$

- Consequence:
  - Fundamental requirement is that it should be easy to compute $M^{-1}b$ for an arbitrary vector v.

Left, Right, and Split preconditioning

- Left preconditioning

$$M^{-1}Ax = M^{-1}b$$

- Right preconditioning

$$AM^{-1}u = b \quad with \quad x = M^{-1}u$$

- Split preconditioning .
  - Assume M is factored: $M = M_L M_R$

$$M_L^{-1}AM_R^{-1}u = M^{-1}b \qquad with\, x = M_R^{-1}u$$

e.g. Preconditioned CG (PCG)

1. *Compute* $r_0 := b - Ax_0$, . $z_0 = M^{-1} r_0$, *and* $p_0 := z_0$
2. *For* $j = 0, 1, \ldots$, *until convergence Do:*
3. $\alpha_j := (r_j, z_j)/(Ap_j, p_j)$
4. $x_{j+1} := x_j + \alpha_j p_j$
5. $r_{j+1} := r_j - \alpha_j Ap_j$
6. $z_{j+1} := M^{-1} r_{j+1}$
7. $\beta_j := (r_{j+1}, z_{j+1})/(r_j, z_j)$
8. $p_{j+1} := z_{j+1} + \beta_j p_j$
9. *EndDo*

Note $M^{-1} A$ is also self-adjoint with respect to $(.,.)_A$:

$$(M^{-1}Ax, y)_A = (AM^{-1}Ax, y) = (x, AM^{-1}Ay) = (x, M^{-1}Ay)_A$$

GMRES with (right) Preconditioning

1. *Start*: Choose x_0 and a dimension m of the Krylov subspaces.

2. Arnoldi process:

Compute $r_0 = b - Ax_0$, $\beta = \|r_0\|$, *and* $v_1 = r_0/\beta$

For $j = 0, 1, \ldots, m$ *Do:*

Compute $z_j := M^{-1} v_j$

Compute $w := Az_j$ x_j

for $i=1, \ldots, j$ *do :*

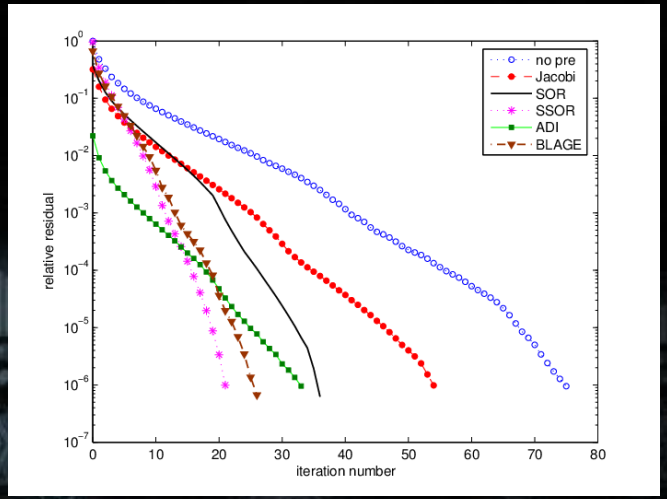
$h_{i,j} := (w, v_j)$

$w := w - h_{i,j}$

$h_{j+1,1} = \|w\|_2$

$w = w - h_{i,j} v_j$

Define $V_m [v_1, \ldots, v_m]$ and $:= \bar{H}_m = \{h_{i,j}\}$

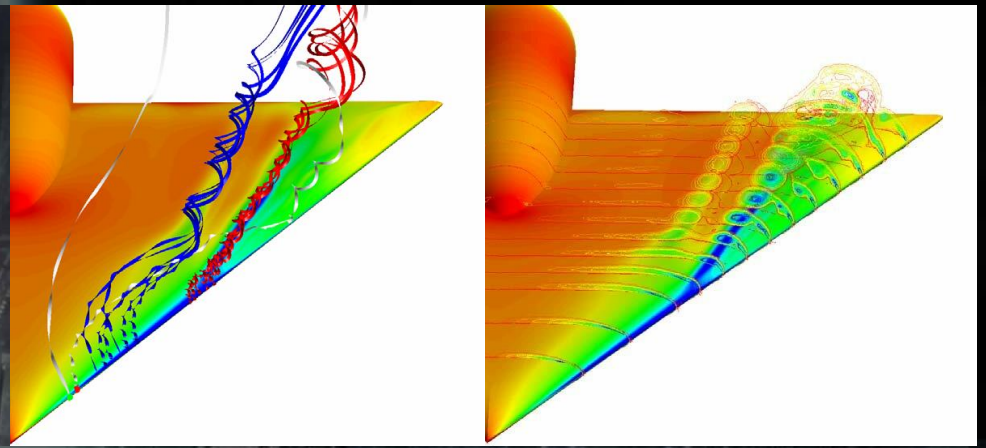


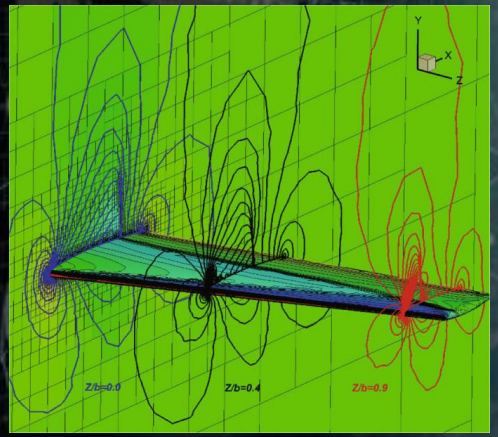
3. Form the approximate solution: Compute $x_m = x_0 + M^{-1} V_m y_m$

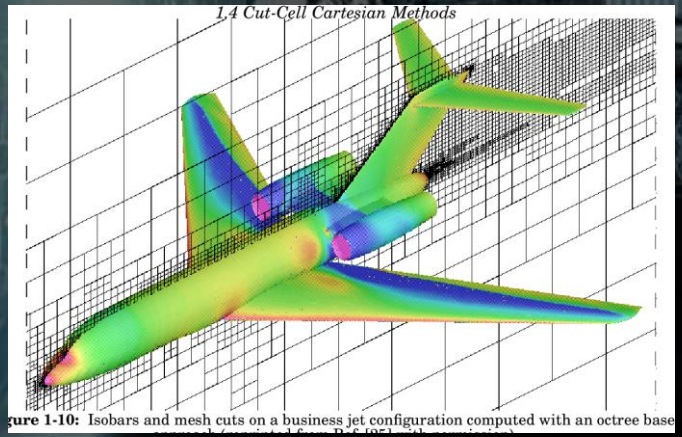
where $y_m = argmin \|\beta e_1 - \bar{H}_m y\|_2$ *and* $e_1 = [1,0, \ldots \ldots, 0]^T$

4. Restart: If satisfied stop, else set $x_0 \leftarrow x_m$ *and goto 2.*

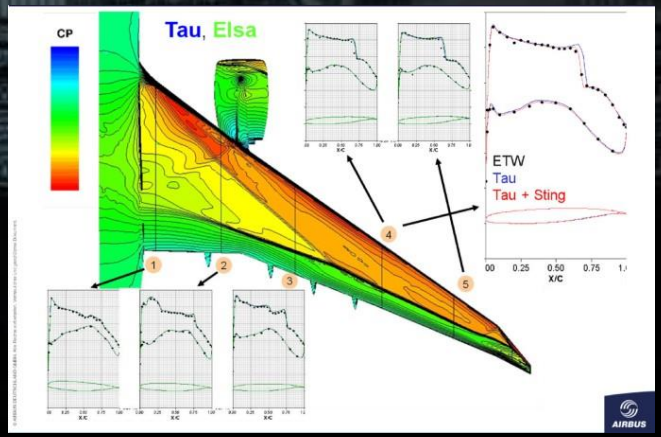
Numerical Solutions for the VFE-2 Configuration on Unstructured Grids at USAFA, United States (2009)







ure 1-10: Isobars and mesh cuts on a business jet configuration computed with an octree base



N-Body Problems

- Suppose the answer at each point depends on data at all the other points
  - Electrostatic, gravitational force
  - Solution of elliptic PDEs
  - Graph partitioning
- Seems to require at least O(n*n) work, communication
- If the dependence on “distant” data can be compressed
  - Because it gets smaller, smoother, simpler…
- Then by compressing data of groups of nearby points, can cut cost (work, communication) at distant points
  - Apply idea recursively: cost drops to O(n log n) or even O(n)
- Examples:
  - Barnes-Hut or Fast Multipole Method (FMM) for electrostatics/gravity/…
  - Multigrid for elliptic PDE
  - Multilevel graph partitioning (METIS, Chaco,…)

Outline of N-Body problem

- Motivation
  - Obvious algorithm for computing gravitational or electrostatic force on N bodies takes O(N*N) work
- How to reduce the number of particles in the force sum
  - We must settle for an approximate answer (say 2 decimal digits, or perhaps 16 …)
- Basic Data Structures: Quad Trees and Oct Trees
- The Barnes-Hut Algorithm (BH)
  - An O(N log N) approximate algorithm for the N-Body problem
- The Fast Multipole Method (FMM)
  - An O(N) approximate algorithm for the N-Body problem
- Parallelizing BH, FMM and related algorithms

Particle Simulation

F(i) = external force + nearest neighbor force + N-Body force

- External force is usually embarrassingly parallel and costs O(N) for all particles
- Nearest neighbor force requires interacting with a few neighbors, so still O(N)
  - Van der Waals, bouncing balls
- N-Body force (gravity or electrostatics) requires all-to-all interactions
  - f(i) = Σ f(i,k) …
  - f(i,k) = c*v/||v||^(2 or 3)
    - v = vector from particle i to particle k , c = product of masses or charges
      - ||v|| = length of v
  - Obvious algorithm costs O(n2), but we can do better...

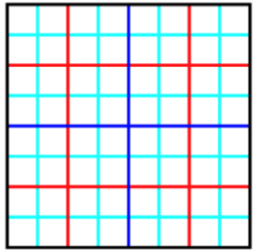
```
t = 0
while t < t_final
  for i = 1 to n … n = number of particles
    compute f(i) = force on particle i
  for i = 1 to n
    move particle i under force f(i) for dt
    compute interesting properties of particles
  t = t + dt
end while
```

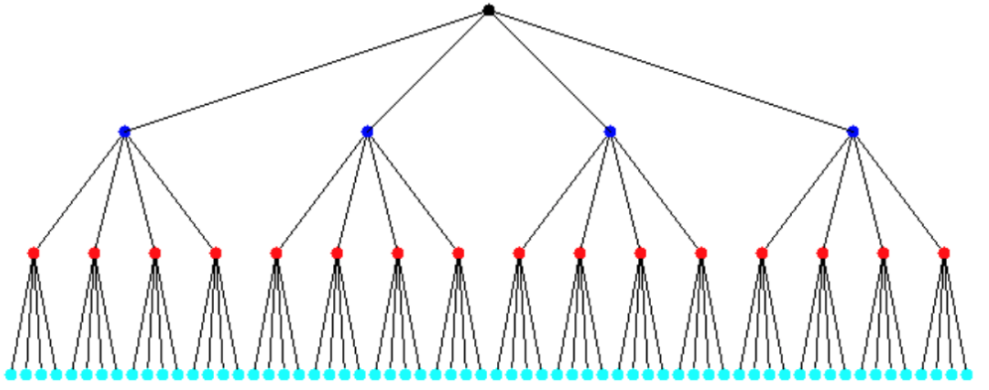
Applications

- Astrophysics and Celestial Mechanics - 1992
  - Intel Delta = 1992 supercomputer, 512 Intel i860s
  - 17 million particles, 600 time steps, 24 hours elapsed time
    - M. Warren and J. Salmon
    - Gordon Bell Prize at Supercomputing 1992
  - Sustained 5.2 Gigaflops = 44K Flops/particle/time step
  - 1% accuracy
  - Direct method (17 Flops/particle/time step) at 5.2 Gflops would have taken 18 years, 6570 times longer
- Vortex particle simulation of turbulence – 2009
  - Cluster of 256 NVIDIA GeForce 8800 GPUs
  - 16.8 million particles
    - T. Hamada, R. Yokota, K. Nitadori. T. Narumi, K. Yasoki et al
    - Gordon Bell Prize for Price/Performance at Supercomputing 2009
  - Sustained 20 Teraflops, or $8/Gigaflop

Basic Data Structures: Quad Trees and Oct Trees

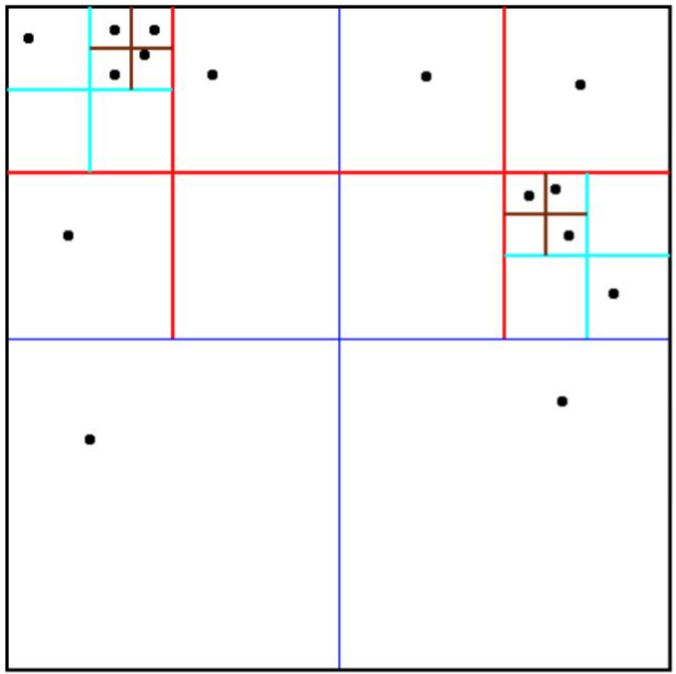
- Data structure to subdivide the plane
  - Nodes can contain coordinates of center of box, side length
  - Eventually also coordinates of CM, total mass, etc.
- In a complete quad tree, each nonleaf node has 4 children
- All our algorithms begin by constructing a tree to hold all the particles
- Interesting cases have nonuniformly distributed particles
  - In a complete tree most nodes would be empty, a waste of space and time
- Adaptive Quad (Oct) Tree only subdivides space where particles are located

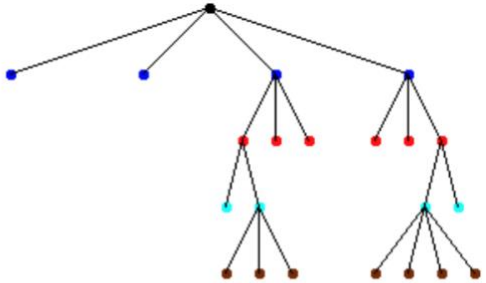




Example of an Adaptive Quad Tree

- Adaptive quadtree where no square contains more than 1 particle





Child nodes enumerated counterclockwise from SW corner, empty ones excluded

Parallelizing Hierachical N-Body codes

- Barnes-Hut, FMM and related algorithm have similar computational structure:
  - 1) Build the QuadTree
  - 2) Traverse QuadTree from leaves to root and build outer expansions for Barnes-Hut)
  - 3) Traverse QuadTree from root to leaves and build any inner expansions for FMM only
  - 4) Traverse QuadTree to accumulate forces for each particle
- One parallelization scheme will work for them all
  - Assign regions of space to each processor
  - Regions may have different shapes, to get load balance
    - Each region will have about N/p particles
  - Each processor will store part of Quadtree containing all particles (=leaves) in its region, and their ancestors in Quadtree
    - Top of tree stored by all processors, lower nodes may also be shared
  - Each processor will also store adjoining parts of Quadtree needed to compute forces for particles it owns
    - Subset of Quadtree needed by a processor called the Locally Essential Tree (LET)
  - Given the LET, all force accumulations (step 4) are done in parallel, without communication

Programming Model - BSP

- BSP Model = Bulk Synchronous Programming Model
  - All processors compute; barrier; all processors communicate;barrier; repeat
- Advantages
  - easy to program (parallel code looks like serial code)
  - easy to port (MPI, shared memory, TCP network)
- Possible disadvantage
  - Rigidly synchronous style might mean inefficiency?
- OK with few processors; communication costs low
  - FMM 80% efficient on 32 processor Cray T3E
  - FMM 90% efficient on 4 PCs on slow network
  - FMM 85% efficient on 16 processor SGI SMP (Power Challenge)
  - Better efficiencies for Barnes-Hut, other algorithms

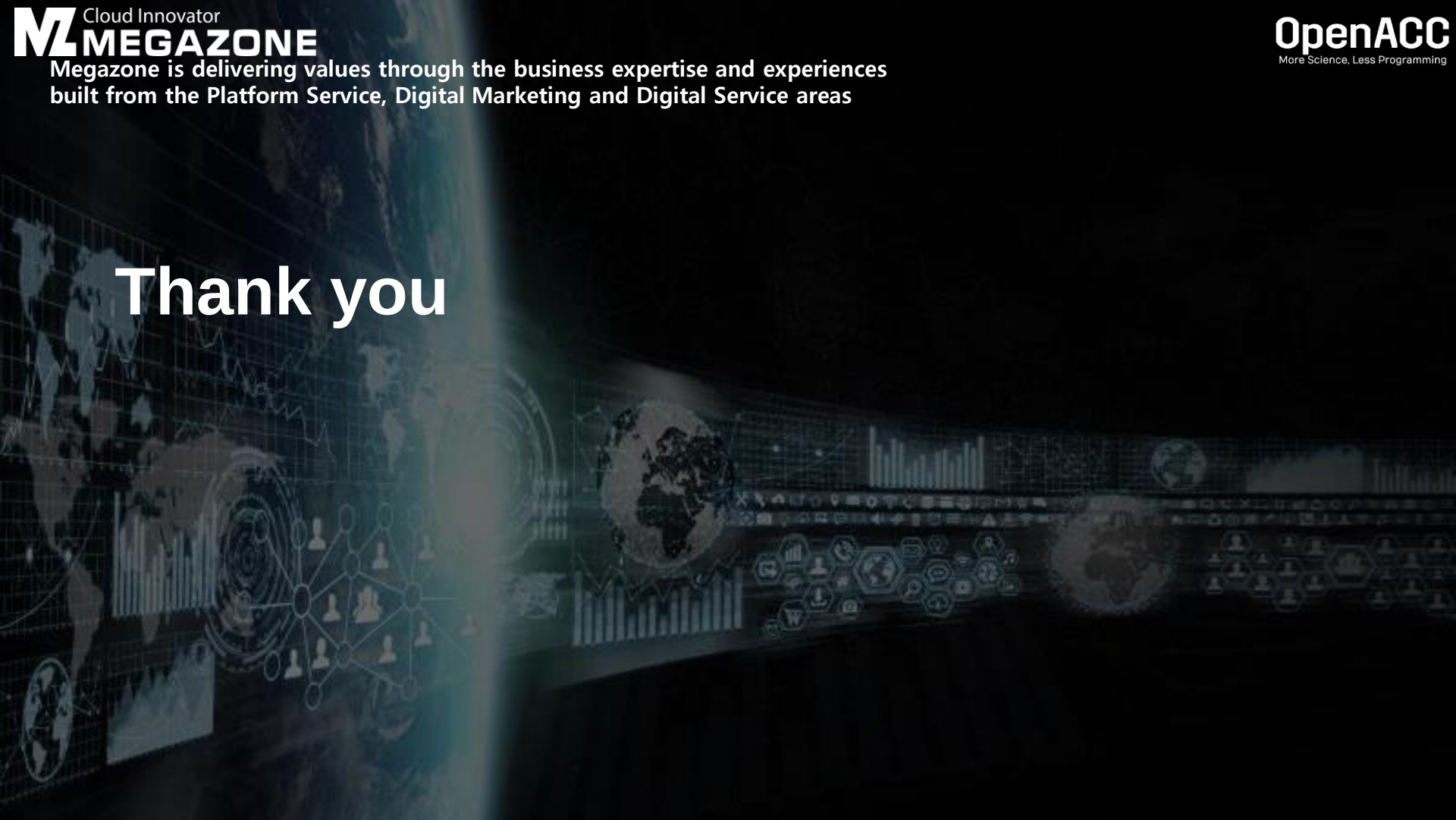
Load Balancing

- And Other load balancing schemes
  - Using Recursive split region with Orthogonal Recursive Bisection
  - Costzone method for shared memory
  - Hashed Oct Tree for distributed memory
  - Partitioning Quad Tree instead of space

Cloud Innovator
MEGAZONE

Megazone is delivering values through the business expertise and experiences built from the Platform Service, Digital Marketing and Digital Service areas

OpenACC
More Science, Less Programming



Thank you