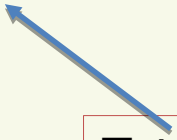
PHASM501 – Techniques of High-Performance Computing

Formalities

- 30 hours lectures
- 10 hours tutorials

Times:

Monday 2 to 4pm
Thursday 4 to 6pm



Tutorials scheduled as needed in these hours.

Assessment:

2 Courseworks (20% each)
One course project (60%)

Assignments: Hand in Week 6 and 10 (Friday 12 lunch time)
Project: Three phases with submission in weeks 5, 8 and 13 (also Friday 12 lunch time)

Syllabus

Part 1: HPC Technologies (6 hours)

- Design of HPC Systems
- Shared vs distributed memory parallelization
- Overview of HPC Programming Languages
- Python for High-Performance Computing
- An overview of PETSc and its Python interface

Part 2: Large-scale computational grids (6 hours)

- Types of grids
- Setting up a grid, data structures, connectivity, etc.
- Grid Partitioning, balancing
- Reference elements
- Evaluating integrals over grids
- Example: Setting up a Laplace problem
- Software for working with grids

Syllabus...

Part 3: Sparse matrices (6 hours)

- From grids to sparse matrices
- Data storage schemes for sparse matrices
- Sparse Matrix Vector Products
- Sparse LU Decomposition
- Parallel sparse matrix operations

Part 4: Iterative Linear Algebra (6 hours)

- Parallel Iterative Solvers
- Overview of Preconditioning techniques
- Algebraic multigrid preconditioning
- Iterative solvers and preconditioners in PETSc

Part 5: Full worked out parallel implementation of the following problem types (6 hours)

- Solving a transport equation with finite volume methods
- Finite element solution of a boundary value problem
- Finite difference methods for acoustic wave propagation

A big disclaimer

- This is only the second time this module is being run.
- We have a lot of freedom to shape the module not only for this term but also for the future.
- If you **do not like** something or have problems tell me straight away.
- Also tell me straight away if you **do like** something.

Performance of modern hardware

Samsung Galaxy S6 Edge
Linpack Benchmark: 0.8 GFlop/s

1GFlop/s = 10^9 floating point operations per second

4Ghz Intel Core I7-6700K
Linpack Benchmark: 250 GFlop/s

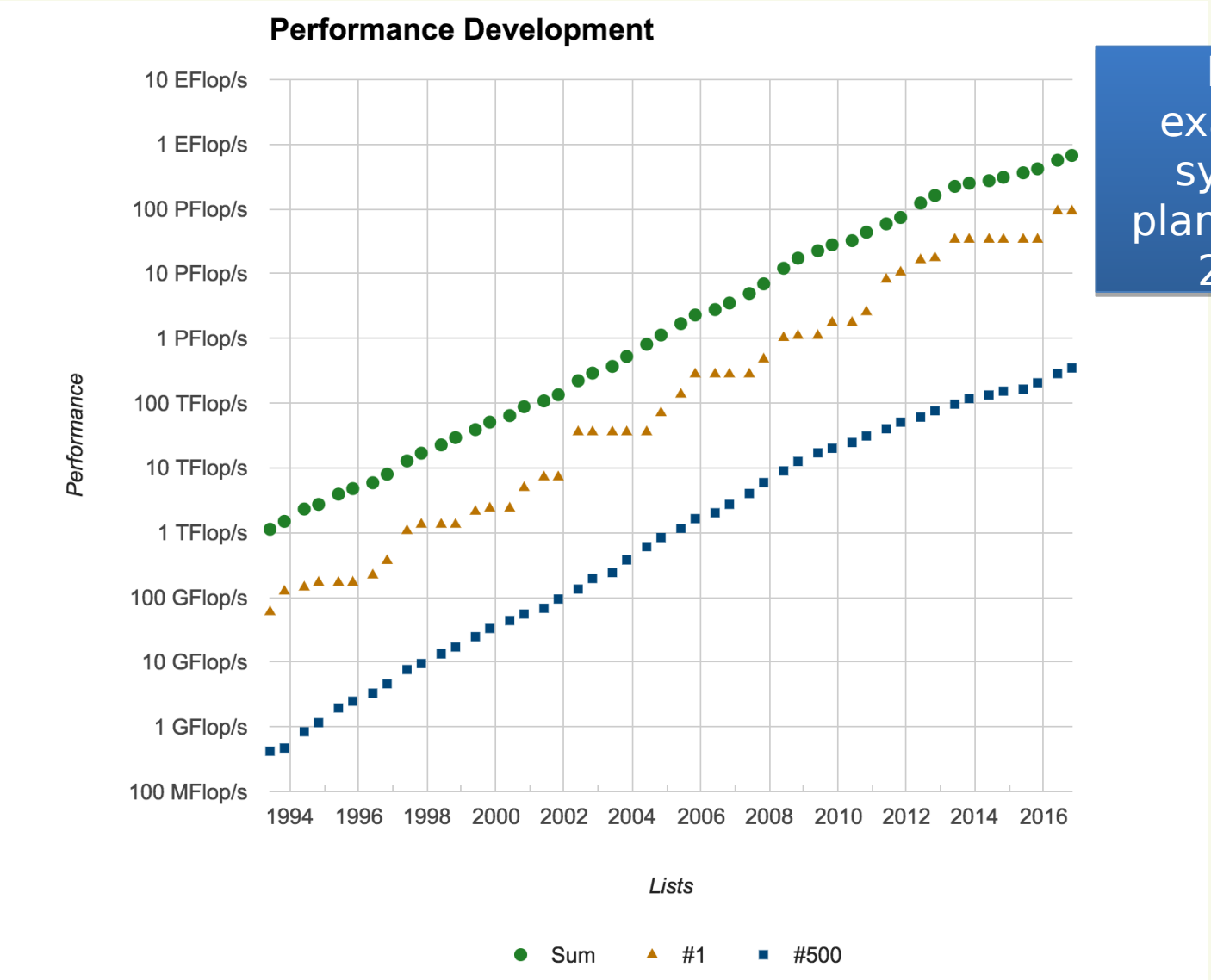
Intel Xeon E5-2699v3 (2.3Ghz, 18 Cores)
Linpack Benchmark: 10^3 Gflop/s

Sunway TaihuLight (Fastest computer in the world)
Linpack Benchmark: 93 x 10^6 GFlop/s

The Top500 List

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National Supercomputing Center in Wuxi China	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRCPC	10,649,600	93,014.6	125,435.9	15,371
2	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
3	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
4	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
5	DOE/SC/LBNL/NERSC United States	**Cori** - Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect Cray Inc.	622,336	14,014.7	27,880.7	3,939
6	Joint Center for Advanced High Performance Computing Japan	**Oakforest-PACS** - PRIMERGY CX1640 M1, Intel Xeon Phi 7250 68C 1.4GHz, Intel Omni-Path Fujitsu	556,104	13,554.6	24,913.5	2,719
7	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,660

Performance over time



First exascale system planned for 2021

Source: www.top500.org

Amdahl's Law

$$T = T_{parallel} + T_{serial}$$
$$= pT + (1-p)T$$

$$T(s) = \frac{p}{s}T + (1-p)T$$

T : Overall time

$T_{parallel}$: Execution time of parallelizable code

T_{serial} : Execution time of serial code

p : Fraction of parallelizable code

s : Speedup of parallel execution

Maximum possible speedup:

$$\lim_{s\to\infty} \frac{T}{T(s)} = \frac{1}{1-p}$$

Amdahl's law is too pessimistic as it assumes speedup for a constant program size. In practice larger parallel systems are used for larger problems.

Gustafson's Law

$$W = (1-p)W_{serial} + pW_{parallel}$$

T : Overall time

$W_{parallel}$: Parallelizable workload

W_{serial} : Serial workload

p : Fraction of parallelizable workload

s : Increase in parallel workload at constant execution time

$$W(s) = (1-p)W + spW$$

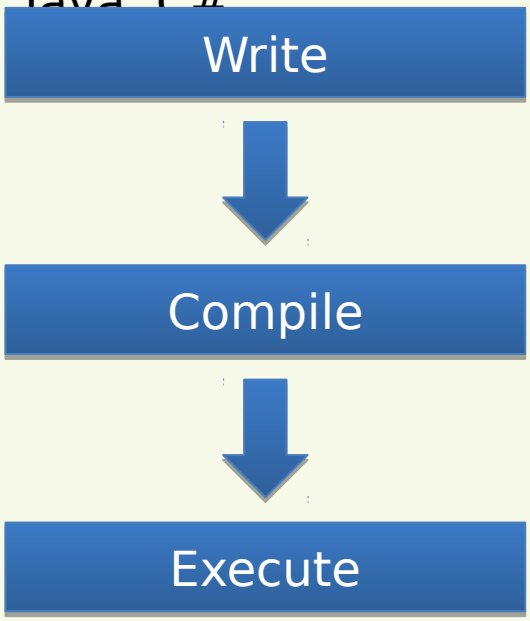
$$\frac{W(s)}{W} = 1 - p + sp$$

A parallel program my read parameters for a computation from disk (serial execution) and then distribute the workload onto a number of fully parallel threads.

Programming Languages

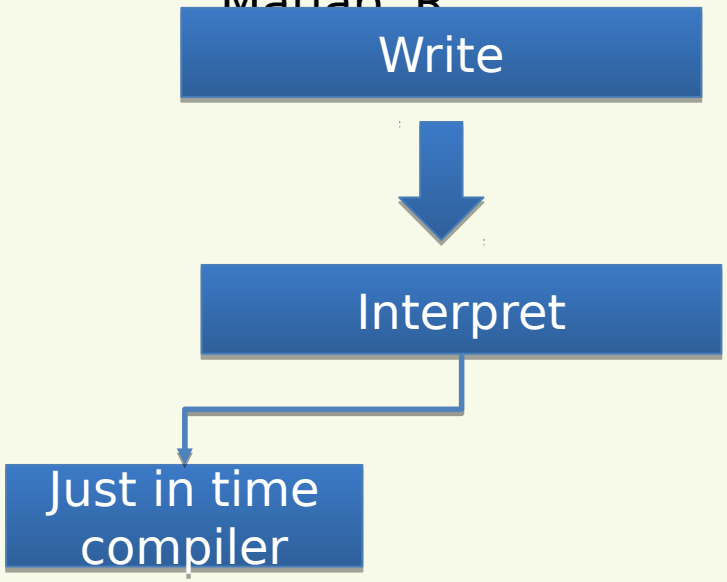
Compiled Languages

Fortran, C/C++, Java, C#



Interpreted Languages

Python, Julia, Matlab, R



Fortran

- Oldest language still used for HPC
- Many traditional HPC libraries written in Fortran or have Fortran interfaces
- Big difference between Fortran 77 and newer Fortran variants
- Fortran is legacy. Do not use for new projects.
- Compilers: gfortran, ifort, pgfortran, ...

```
program hello
    implicit none
    write (*,*) "Hello, world."
end program hello
```

C

- Mostly used for low-level libraries and embedded sector (with some notable exceptions such as PETSc or OpenCL)
- C programs compile fast and are easy to get to compile on virtually any C compiler
- Compiler optimizations harder than Fortran (though modern compilers do a great job)
- Compilers: gcc, clang, icc, pgcc, etc.

```
#include <stdio.h>

int main(void) {
      printf("Hello
World\n");   return 0;
}
```

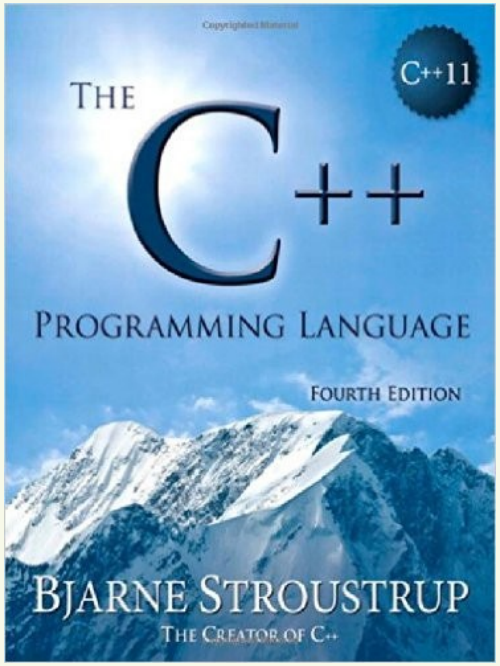
“C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off.”

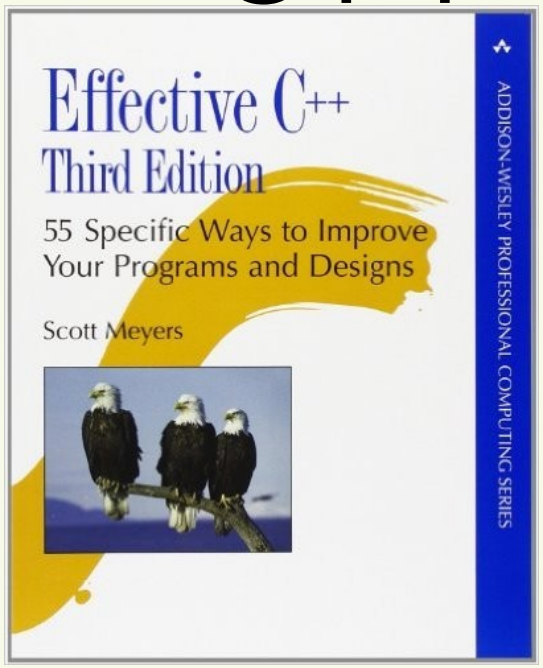
C++

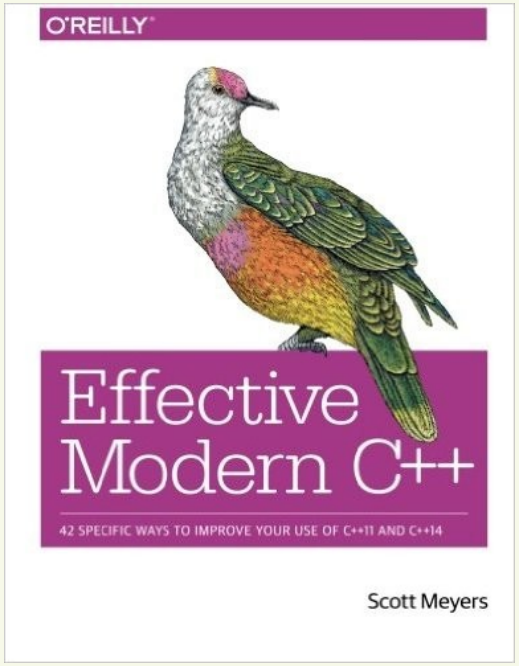
- Almost fully backward compatible to C.
- Implements modern object and templated design principles.
- Most new HPC projects use C++.
- Major redesign in C++11 standard
- Compilers: g++, clang, icpc, pgc++,...

```
#include <iostream>

int main(void) {
      std::cout << “Hello Wo

return 0;
```

Books for safe and modern C++







Java and C#

- Rarely used for HPC applications
- Java and C# compile into an intermediate virtual machine language and are typically just-in-time compiled during execution
- Virtual machine environment encapsulates from the hardware but allows much fewer optimizations
- Widely used for business applications

New compiled languages

- Google Go is a systems programming language. Designed by Google it has fast become popular
- Rust is designed as a much less error prone replacement for C++. Used by Mozilla for their new browser engine
- Swift is the new language by Apple for iOS development
- None of these languages have any traction in the HPC sector (maybe with the exception of Go for some cloud computing tools, but not for fast algorithms)

Python

- High-Level dynamic programming language
- Supports procedural, object and functional programming
- Very easy to interface with C/C++ and Fortran
- Python itself is slow. But it’s extension modules make it extremely powerful for HPC

```
print(“Hello world”)
```

Julia

- Modern dynamic high-performance programming language
- Still very young, but fast getting a lot of traction, especially in big data
- Julia is designed as fast, just-in-time compiled language.
- Claims to achieve near C-level performance

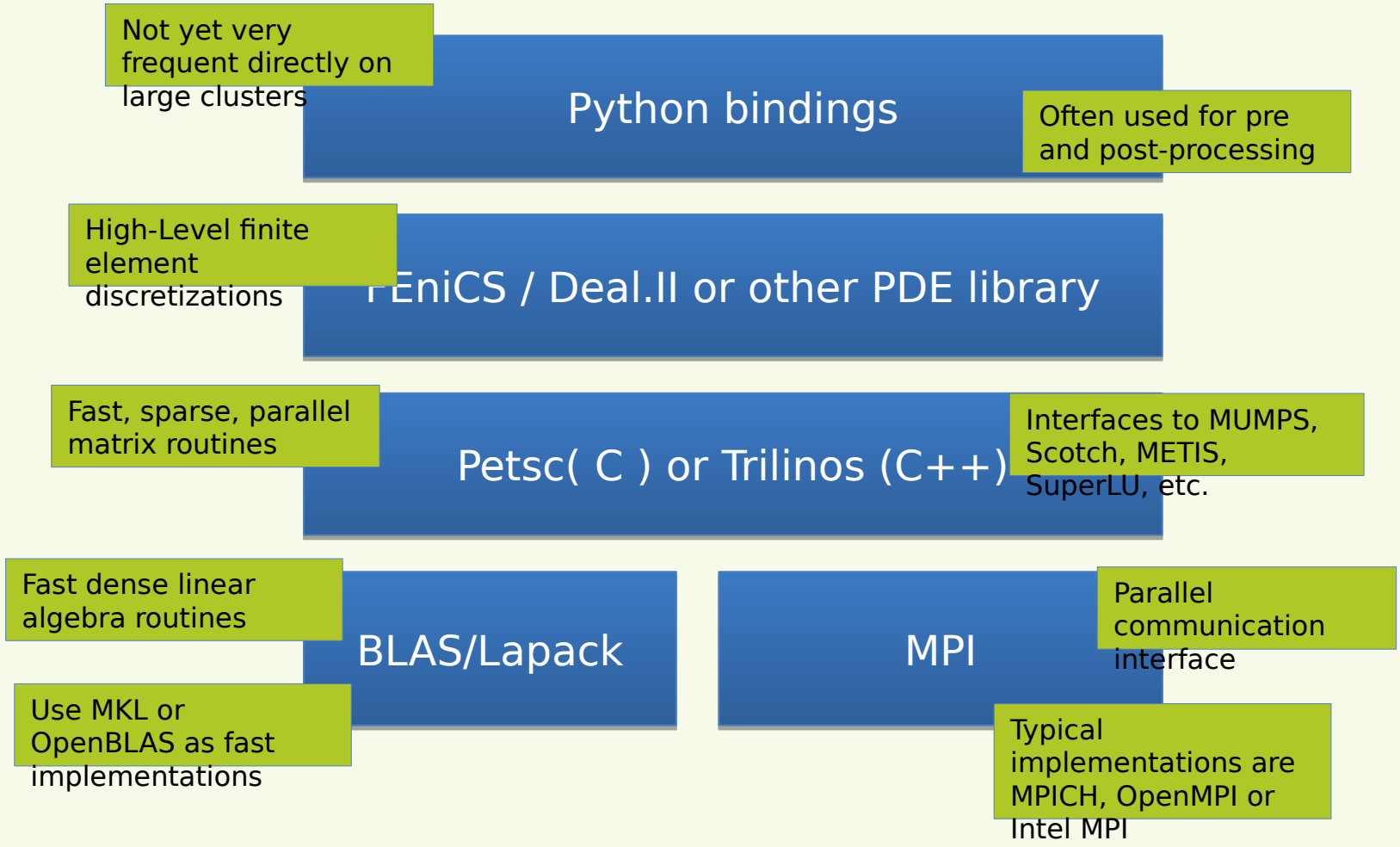
```
println(“Hello world”)
```

Matlab

- One of the first dynamic numerical computing language
- Originally designed as convenient interface to Fortran libraries
- Matlab extremely wide spread and suitable for a range of HPC applications
- Huge amount of toolboxes for various industry sectors
- Not open-source

disp(“Hello world”)

A typical HPC Stack

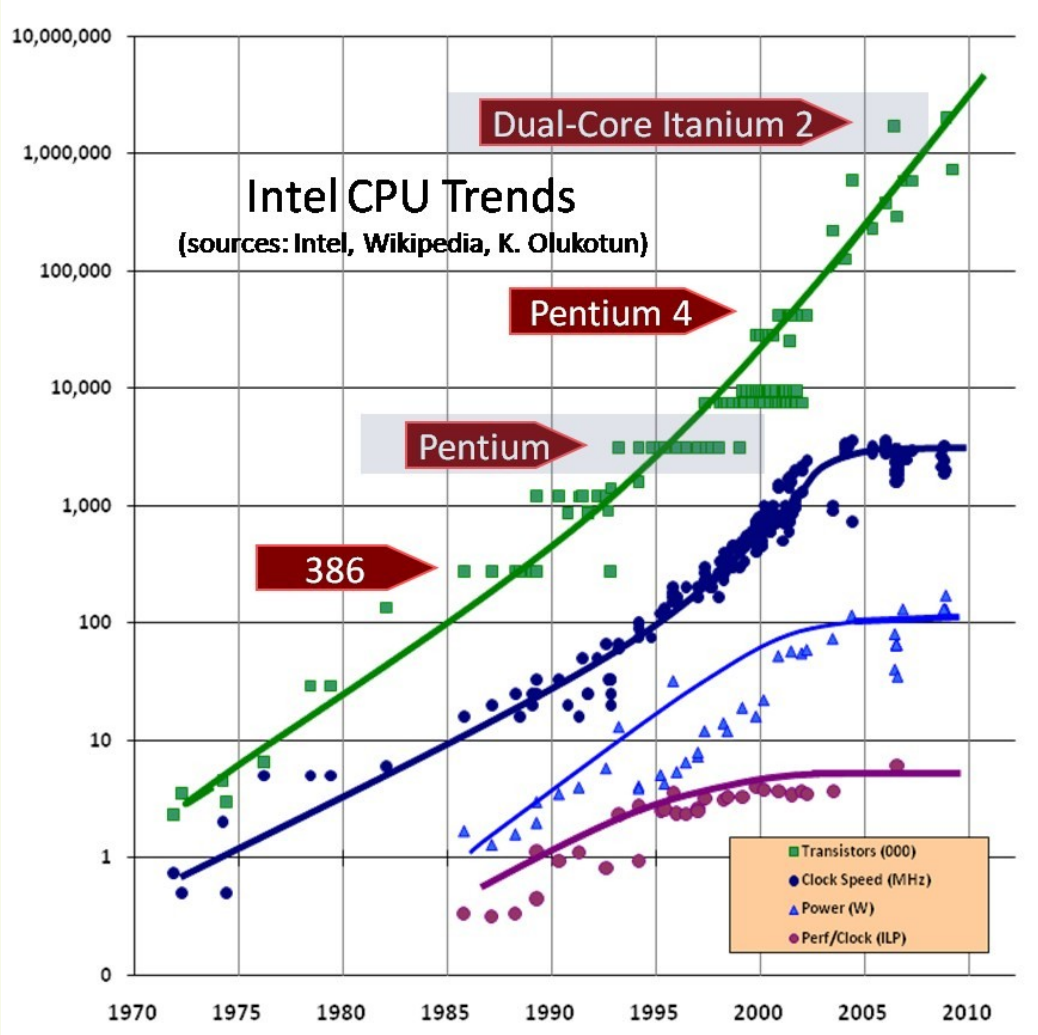


How do I get it?

- Easiest way: Download Ubuntu Linux. All relevant software available in repositories
- Hard way: Compile everything yourself together with performance libraries (e.g. Intel MKL) of your choice
- Unfortunately, we cannot give you a Petascale machine for playing around…
- For Lecture: Download Anaconda or Intel Python (more in the first tutorial)

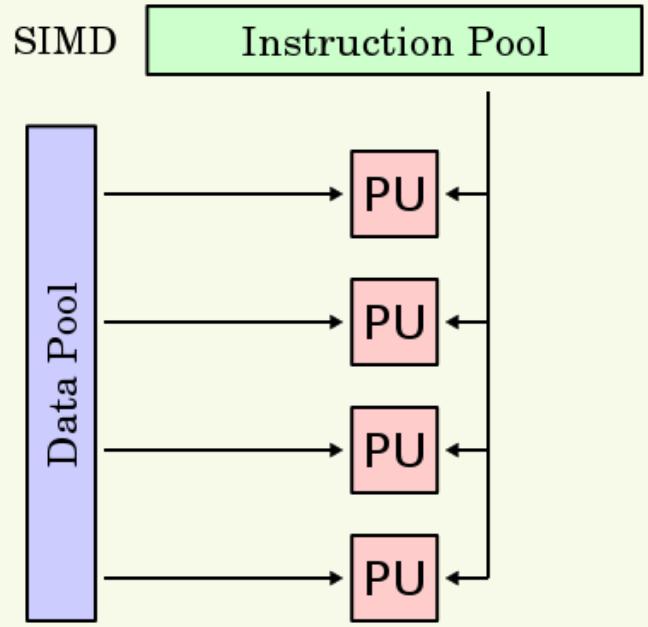
Why parallel processing?

- Number of instructions per second on CPUs cannot scale arbitrarily (energy consumption, heat dissipation, material design limits, etc.)
- Most CPUs run at around 2 to 3GHz. This has not changed for years
- Further performance increase only through massive parallelism



https://www.extremetech.com/computing/116561-the-death-of-cpu-scaling-from-one-core-to-many-and-why-were-still-stuck

Parallel Concepts...



Wikipedia: Flynn's Taxonomy

SIMD: Single Instruction – Multiple Data

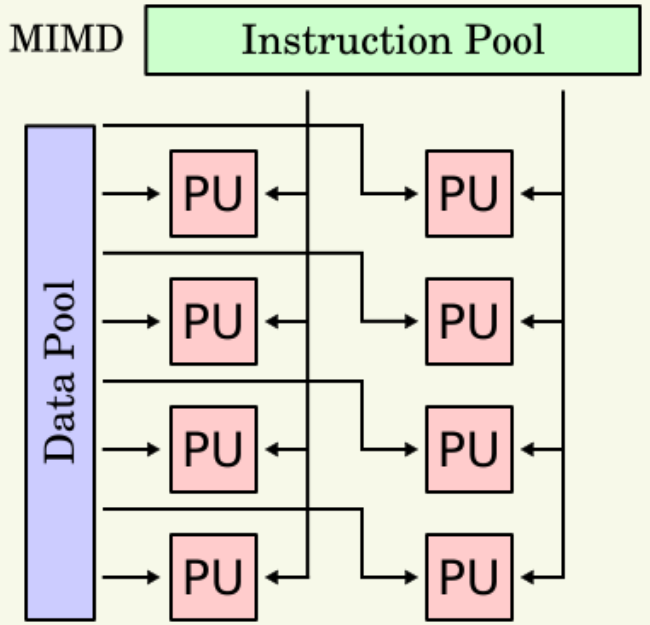
One set of instructions is executed in parallel on a vector of data units (e.g. numbers or matrix rows).

- Popular concept until the 1990s in HPC Architectures (Vector processors)
- Came out of fashion due to cluster computing
- Recent revival with streaming instruction sets in modern CPUs (Intel SSE, etc.)

Parallel Concepts...

MIMD: Multiple Instructions – Multiple Data

Each CPU unit can asynchronously execute different types of commands on different data sets.



Wikipedia: Flynn's Taxonomy

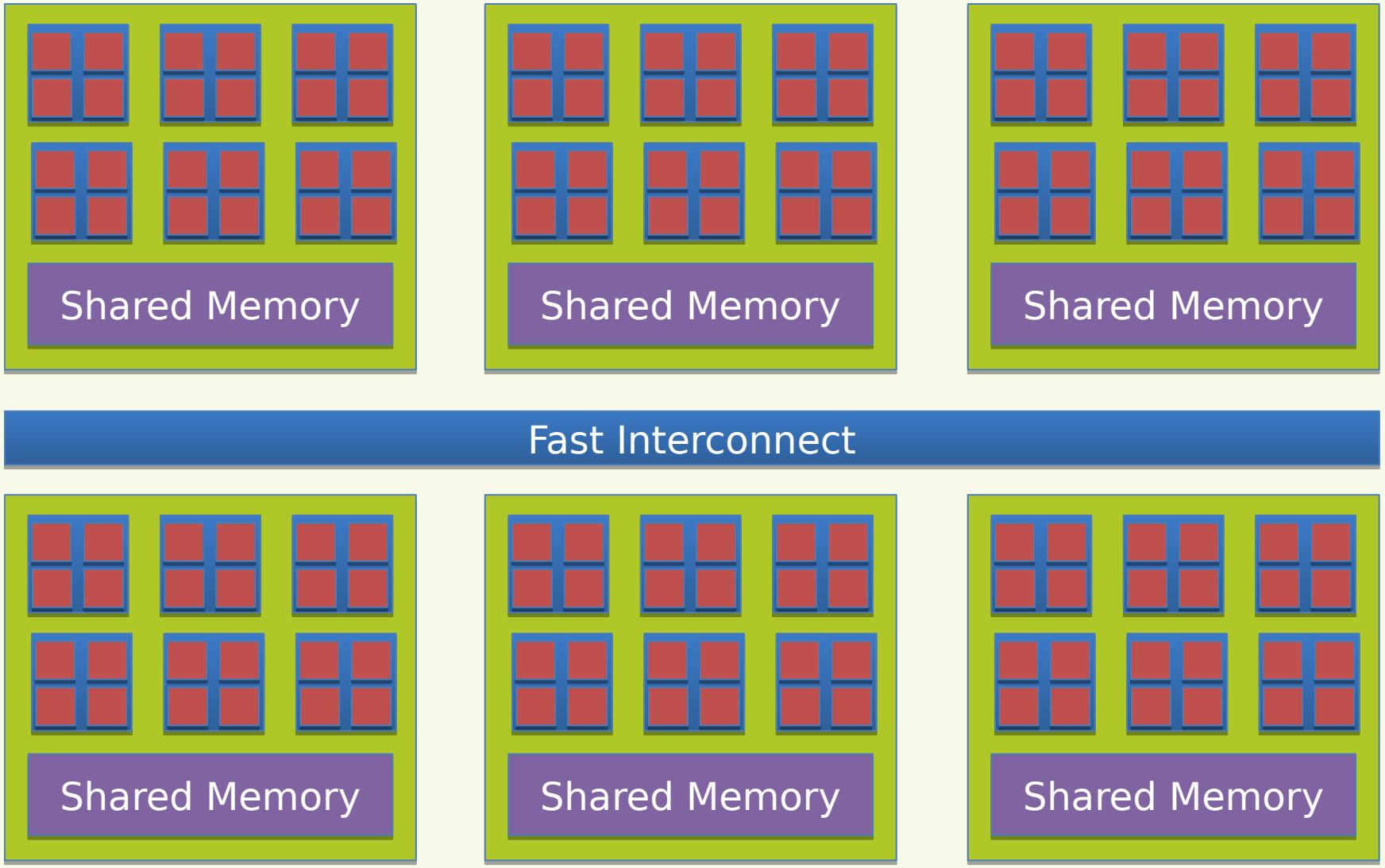
- All modern HPC systems are based on the MIMD design
- All modern multi-core CPUs use MIMD

Cluster Node

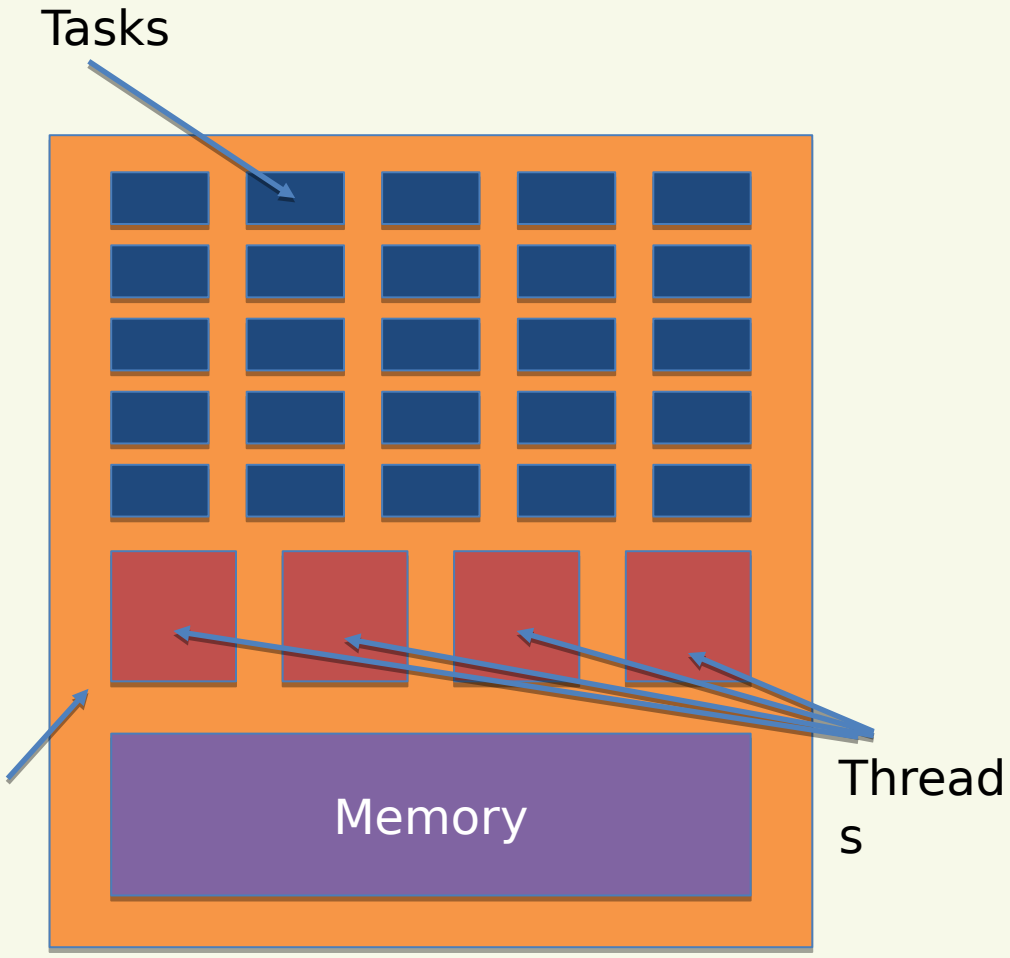
CPU

CPU Core with vector unit

A typical HPC system



Shared Memory parallelization (multithreading)



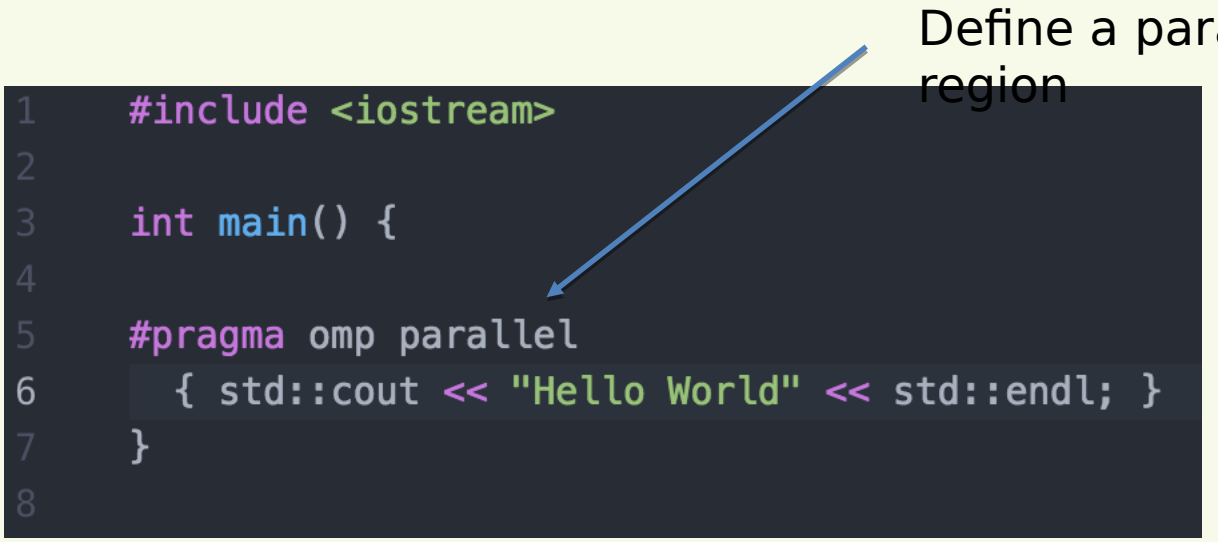
- Process: Independent program with its own allocated memory running on a physical cluster node.
- Thread: Independent unit of execution running within a thread.
- Task: Logical unit of execution.

- Different processes can not see each others memory.
- All threads within a process see the same memory.

Shared Memory Frameworks

- OpenMP: Industry standard. Supported by most compilers directly. Works with Fortran, C/C++. Cumbersome for complex task dependencies.
- Intel Threading Building Blocks (TBB): Versatile C++ library for task based threading. Parallel Container constructs. Needs to be linked to explicitly.
- Intel Cilk: Simple tasking framework for C/C++. Supported by more and more compilers.

OpenMP



- Everything inside the #pragma omp parallel block is executed in parallel.
- By default as many parallel threads are created as there are virtual CPU cores.
- The number of threads can be controlled by the environment variable OMP_NUM_THREADS.

Parallel for loops

```cpp
#include <iostream>
#include <vector>

int main() {

  int count = 100;
  std::vector<int> vec(100);

#pragma omp parallel for
  for (std::size_t i = 0; i < count; i++) {
    vec[i] = i * i;
  }
  return 0;
}

```

- The statements within the for loop are executed in parallel.
- Most frequent use form for OpenMP.

Race conditions

```cpp
#include <iostream>
#include <vector>

int main() {

  int count = 100;
  std::vector<int> vec(100);
  int sum = 0;

#pragma omp parallel for
  for (std::size_t i = 0; i < count; i++) {
    vec[i] = i * i;
    // The following code is wrong.
    // The result of the operation is completely random.
    sum += vec[i];
  }
  std::cout << sum << std::endl;
  return 0;
}
```

sum is shared among all the threads. The result of the addition becomes random if more than one thread is running.

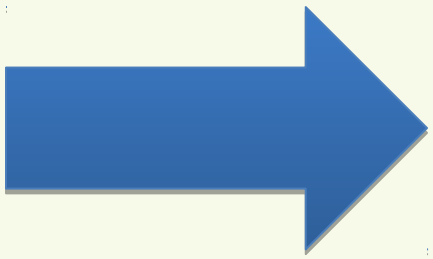
What can happen?

Example:

```
int i = 0;

#pragma omp paral
{
    i += 1;
}
```

Thread 1	Thread 2	Value of i
Read i into register		0
Add 1 to register	Read i into register	
Write result back into i	Add 1 to register	1
	Write result back into i	1

The result becomes indetermined.
We need to ensure that only one thread at a time writes into a shared variable.



Fix with omp critical region

Critical Regions

```cpp
#include <iostream>
#include <vector>

int main() {

  int count = 100;
  std::vector<int> vec(100);
  int sum = 0;

#pragma omp parallel for
  for (std::size_t i = 0; i < count; i++) {
    vec[i] = i * i;
// Only one thread at a time enters the critical region
#pragma omp critical
    { sum += vec[i]; }
  }
  std::cout << sum << std::endl;
  return 0;
}

```

- Only one thread at a time is allowed to enter the critical region.
- The race condition is resolved.
- Slow down as threads have to wait.

General Concept: Mutex

- A Mutex is the general concept underlying the critical region. The following abstract code shows its use.

```
Mutex mutex;

#Enter parallel region
      lock(mutex)
      i += 1
      release(mutex)
#Exit parallel region
```

If mutex is locked a thread has to wait. It can only enter if it is not locked already.

Release the mutex again when thread is finished.

Private and shared variables

- By default variables declared outside a parallel region are shared.
- Variables declared inside a parallel region are private copies for each thread.
- Can control the beahavior with the statements: private, shared, firstprivate, lastprivate

Nested parallelism

```cpp
#include <cstdio>
#include <iostream>
#include <omp.h>
#include <vector>

int main() {

  const int count = 10;
  double a[count][count];

#pragma omp parallel
  {
    if (omp_get_thread_num() == 0)
      printf("Outer number of threads: %i\n", omp_get_num_threads());
#pragma omp parallel
    {
      if (omp_get_thread_num() == 0)
        printf("Inner number of threads: %i\n", omp_get_num_threads());
    }
  }
  return 0;
}
```

- The outer region runs with 8 threads
- The inner region runs with 1 thread
- Prevents over-utilisation
- Can be controlled with ‘schedule’ directive

Output:

```
Inner number of threads: 1
Inner number of threads: 1
Outer number of threads: 8
Inner number of threads: 1
Inner number of threads: 1
Inner number of threads: 1
Inner number of threads: 1
Inner number of threads: 1
Inner number of threads: 1
Inner number of threads: 1
```

Tasking

- Direct mapping of loop iterations onto threads is cumbersome for complex scenarios
- Nested parallelisation is difficult to control
- Better idea: Separate threads and tasks
- OpenMP supports tasking since OpenMP 3

Fibonacci Sequence

- Fib(n) = Fib(n-1) + Fib(n-2)
- Fib(0) = Fib(1) = 1
- 1, 1, 2, 3, 5, 8, 13, 21, ...
- Standard Loop based threading is not suitable to represent recursive computations
- However, with tasks the problem becomes easy...

Fibonacci with OpenMP tasks

```cpp
#include <iostream>

int fib(int n) {

  if (n == 0 || n == 1)
    return 1;
  else {
    int f1, f2;
#pragma omp task shared(f1) firstprivate(n)
    f1 = fib(n - 1);
#pragma omp task shared(f2) firstprivate(n)
    f2 = fib(n - 2);
#pragma omp taskwait
    return f1 + f2;
  }
}

int main() {

  int n = 5;
#pragma omp parallel shared(n)
  {
#pragma omp single
    { std::cout << fib(n) << std::endl; }
  }
  return 0;
}
```

Start the thread pool

Enter the function with one thread

- #omp task: Schedule a new task
- shared: The variable is shared with the task
- firstprivate: A local copy is created from the previous value
- taskwait: Wait until all tasks complete

Warning: This code is for demonstration only. The tasks have far too little to do to be efficient!

Summary of OpenMP

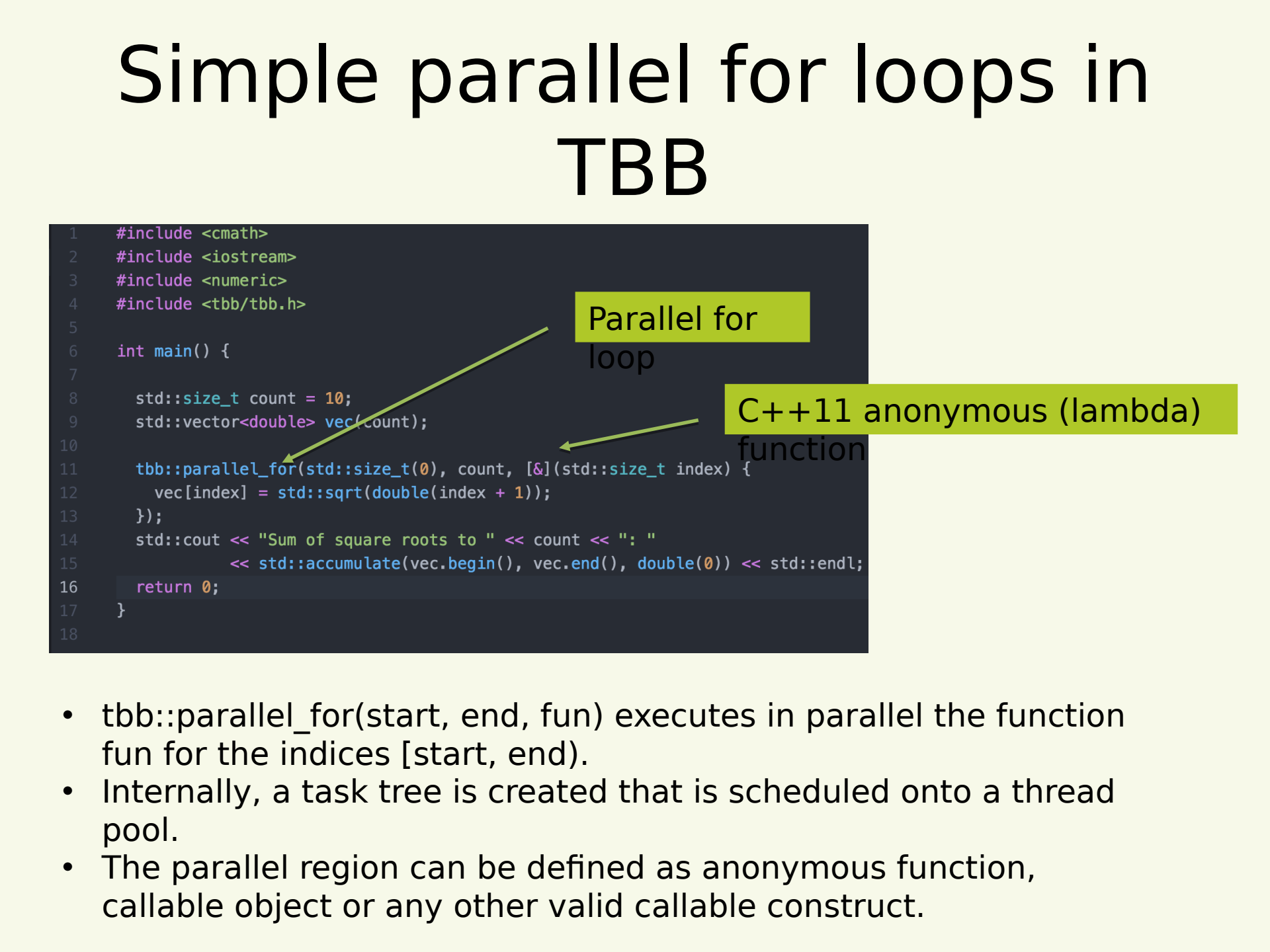
- Simple loop parallelism is trivial to do in OpenMP and can often give significant speed-ups
- OpenMP supports tasking for complicated parallelization strategies
- Almost every compiler supports it
- OpenMP is not a native language construct. Visibility of variables and other controls can be cumbersome and debugging difficult

Intel TBB: Advanced threading in C++

- Advanced threading library written in C++
- Available as open-source and commercially supported version
- Fully task-based design
- Support for advanced C++11 features
- Threadsafe container types
- Straight forward syntax
- Explicit linking of TBB libraries in projects necessary

Simple parallel for loops in TBB

```cpp
#include <cmath>
#include <iostream>
#include <numeric>
#include <tbb/tbb.h>

int main() {

  std::size_t count = 10;
  std::vector<double> vec(count);

  tbb::parallel_for(std::size_t(0), count, [&](std::size_t index) {
    vec[index] = std::sqrt(double(index + 1));
  });
  std::cout << "Sum of square roots to " << count << ": "
            << std::accumulate(vec.begin(), vec.end(), double(0)) << std::endl;
  return 0;
}
```



- tbb::parallel_for(start, end, fun) executes in parallel the function fun for the indices [start, end).
- Internally, a task tree is created that is scheduled onto a thread pool.
- The parallel region can be defined as anonymous function, callable object or any other valid callable construct.

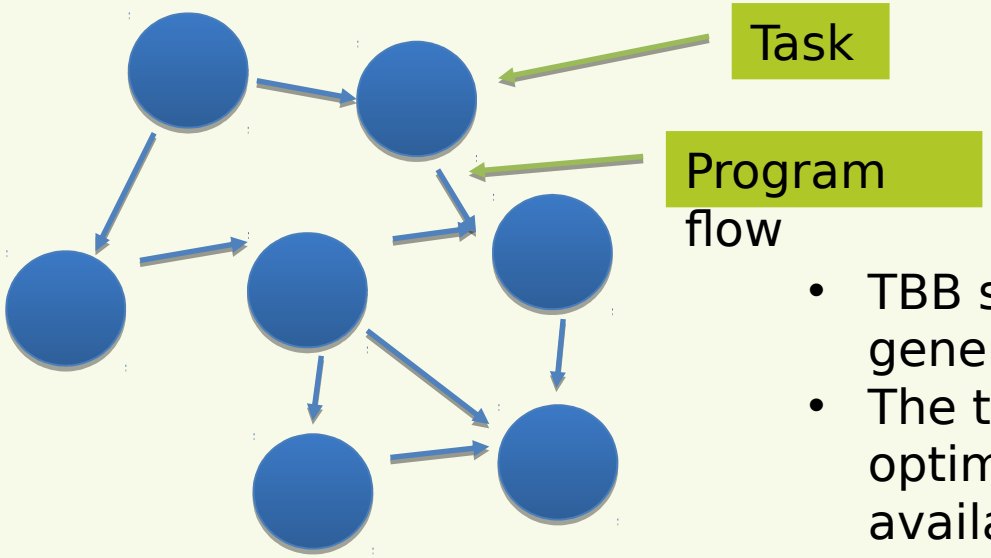
Fibonacci series in TBB

```cpp
#include <iostream>
#include <tbb/tbb.h>

int fib(int n) {
  if (n == 0 || n == 1)
    return 1;
  else {
    int f1, f2;
    tbb::task_group group;
    group.run([&] { f1 = fib(n - 1); });
    group.run_and_wait([&] { f2 = fib(n - 2); });
    return f1 + f2;
  }
}

int main() {

  int n = 5;
  std::cout << "Fib(" << n << ") = " << fib(n) << "." << std::endl;
  return 0;
}

```

- Create a task group with two tasks and submit the tasks
- Task scheduler collects the tasks in a tree and schedules them optimally on the threads

Task graphs

- Complex task dependencies can be represented as a graph

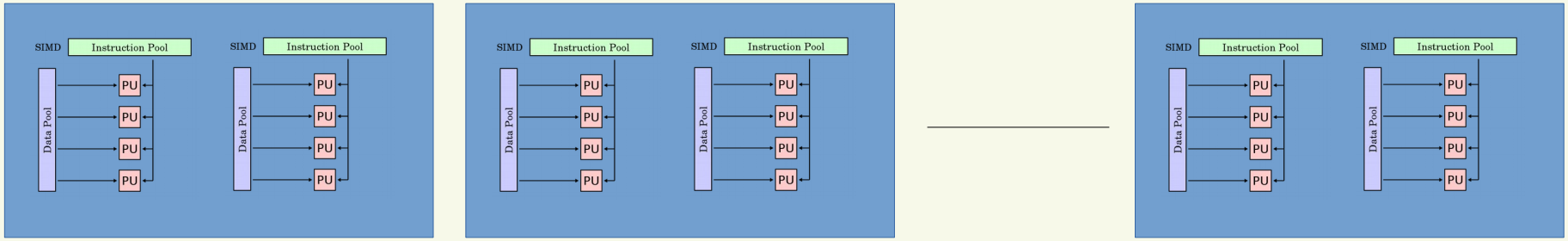


- TBB supports the explicit generation of task trees
- The tree is analyzed and then optimally scheduled on the available threads
- Allows complex execution scenarios

GPU Computing with OpenCL

28 Cores
3.2 Ghz
2 AVX 512 FMA Units
Peak (Single): 2.86 TFlops/s
Peak (Double): 1.43 TFlops/s

How to compute the peak performance?



- Each core has two SIMD Units (AVX 512) With Fused Multiply Add (FMA) each SIMD unit can perform up to 32 single precision or 16 double precision operations simultaneously.

GPU Peak Performance

NVIDIA Tesla V100

5120 CUDA Cores
80 Streaming Multiprocessors
Peak (Single): 14 TFLOPS
Peak (Double): 7 TFLOPS

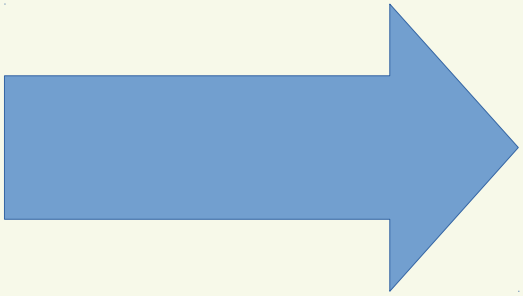
AMD Radeon Pro WX 9100

4096 Stream Processors
64 Compute Units
Peak (Single): 12.29 TFLOPS
Peak (Double): 0.77 TFLOPS

- CPU Core roughly similar to Streaming Multiprocessor (NVIDIA) or Compute Unit (AMD/OpenCL). CUDA Core, Stream Processors correspond to one execution in a SIMD Unit.
- Xeon Platinum has 28 * 2 * 16 = 896 Compute Units in single precision without FMA (twice that with FMA)
- Inside a Compute Unit (Streaming Multiprocessor) all execution paths must be identical.

CPU / GPU Differences

- CPUs optimised for general purpose computing tasks. Great for complex, irregular data structures, codes with a lot of branching.
- CPU have complex caching architecture and branch prediction.
- GPUs much more simple designed. Optimised for parallel processing of large blocks of structured data with the same operations (SIMD Execution).
- Data transfer to GPU is expensive via PCI bus. Algorithms need to be adapted for it.



AMD APU: Combine CPU and GPU unit on a single die. Already used in Xbox One and PS4.

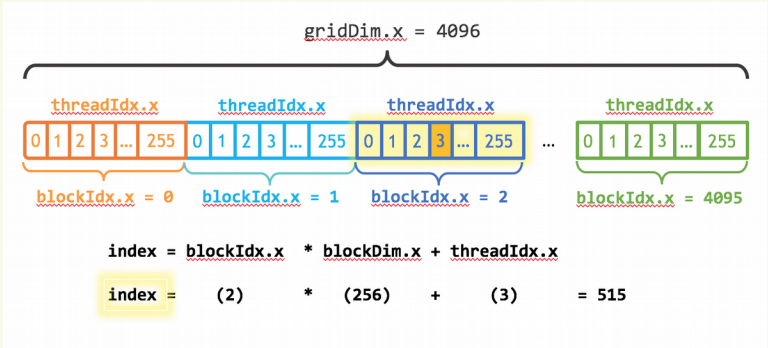
Don’t believe all marketing hype. The choice between CPU and GPU is not clear cut. In the medium term we will see a range of hybrid solutions.

GPU Programming Models

- NVIDIA Cuda: Started in 2007, based on C++ with proprietary extensions. NVIDIA CUDA SDK required. Only runs on NVIDIA GPUs.
- OpenCL: Started in 2009, C based open standards. Open-Source implementations exist. Runs on a range of hardware devices.
- DirectCompute: Windows only library as part of Microsoft DirectX.

A simple Cuda C example

```
#include <iostream>
#include <math.h>
// Kernel function to add the elements of two arrays
__global__
void add(int n, float *x, float *y)
{
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[i];
}
int main(void)
{
  int N = 1<<20;
  float *x, *y;
  // Allocate Unified Memory – accessible from CPU or GPU
  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));
  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }
  // Run kernel on 1M elements on the GPU
  int blockSize = 256;
  int numBlocks = (N + blockSize - 1) / blockSize;
  add<<<numBlocks, blockSize>>>(N, x, y);  // Wait for GPU to finish before accessing on host
  cudaDeviceSynchronize();
  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;
  // Free memory
  cudaFree(x);
  cudaFree(y);
  return 0;
```



https://devblogs.nvidia.com/parallelforall/even-easier-introduction-cuda/

An OpenCL example

https://www.olcf.ornl.gov/tutorials/opencl-vector-addition/

(Code too big to show on slide)

OpenCL remarks

- OpenCL is very verbose. It can be painful to setup a simple kernel in low-level OpenCL.
- Reason: OpenCL needs to work on a range of hardware devices. Cuda just needs to run on NVIDIA.
- In this module we will bypass all the pain by running everything through PyOpenCL, which does all the book keeping automatically.

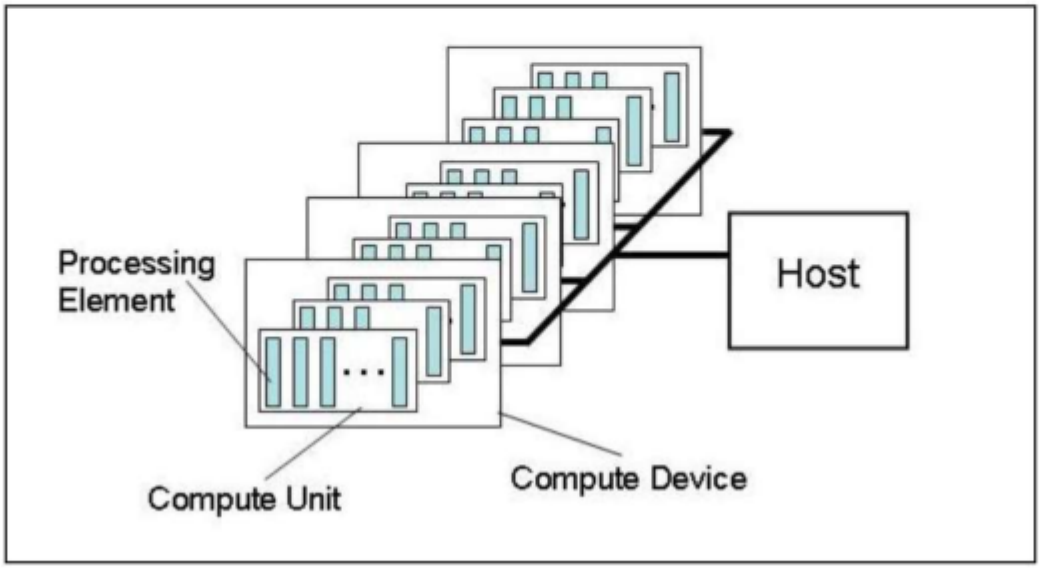
A PyOpenCL Example

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from __future__ import absolute_import, print_function
import numpy as np
import pyopencl as cl
a_np = np.random.rand(50000).astype(np.float32)
b_np = np.random.rand(50000).astype(np.float32)
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)
mf = cl.mem_flags
a_g = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a_np)
b_g = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b_np)
prg = cl.Program(ctx, """
__kernel void sum(
    __global const float *a_g, __global const float *b_g, __global float *res_g)
{
  int gid = get_global_id(0);
  res_g[gid] = a_g[gid] + b_g[gid];
}
""").build()
res_g = cl.Buffer(ctx, mf.WRITE_ONLY, a_np.nbytes)
prg.sum(queue, a_np.shape, None, a_g, b_g, res_g)
res_np = np.empty_like(a_np)
cl.enqueue_copy(queue, res_np, res_g)
# Check on CPU with Numpy:
print(res_np - (a_np + b_np))
print(np.linalg.norm(res_np - (a_np + b_np)))
```

https://documen.tician.de/pyopencl/

OpenCL Platform Model

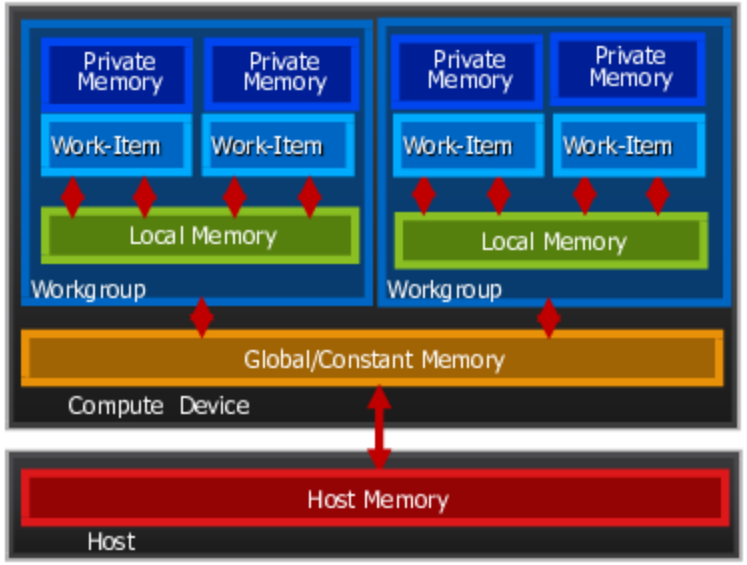
- **One Host + one or more Compute Devices**
  - Each Compute Device is composed of one or more Compute Units
    - Each Compute Unit is further divided into one or more Processing Elements



https://www.khronos.org/assets/uploads/developers/library/2012-pan-pacific-road-show-June/OpenCL-Details-Taiwan_June-2012.pdf

OpenCL Memory Model

- **Private Memory**
  - Per work-item
- **Local Memory**
  - Shared within a workgroup
- **Global/Constant Memory**
  - Visible to all workgroups
- **Host Memory**
  - On the CPU

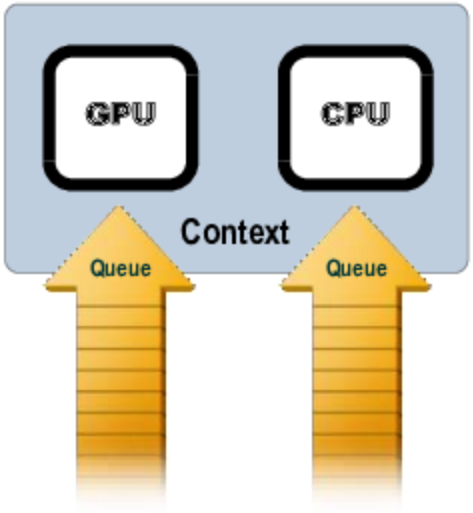


Memory management is Explicit
You must move data from host -> global -> local ... *and* back

https://www.khronos.org/assets/uploads/developers/library/2012-pan-pacific-road-show-June/OpenCL-Details-Taiwan_June-2012.pdf

OpenCL Execution Model

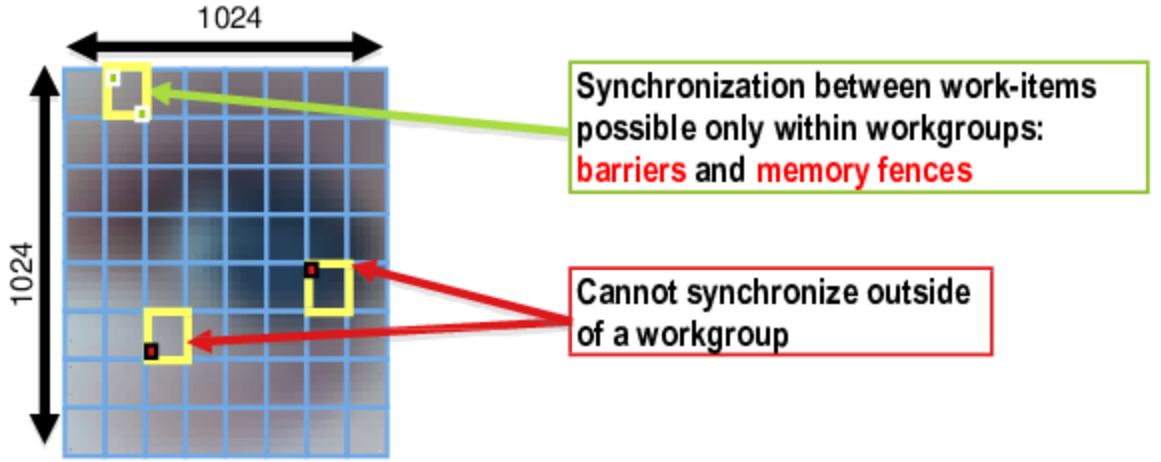
- **OpenCL application runs on a host which submits work to the compute devices**
  - **Context**: The environment within which work-items executes ... includes devices and their memories and command queues
  - **Program**: Collection of kernels and other functions (Analogous to a dynamic library)
  - **Kernel**: the code for a work item. Basically a C function
  - **Work item**: the basic unit of work on an OpenCL device
- **Applications queue kernel execution**
  - Executed in-order or out-of-order



https://www.khronos.org/assets/uploads/developers/library/2012-pan-pacific-road-show-June/OpenCL-Details-Taiwan_June-2012.pdf

An N-dimension domain of work-items

- **Kernels executed across a global domain of *work-items***
- **Work-items grouped into local *workgroups***
- **Define the "best" N-dimensioned index space for your algorithm**
  - Global Dimensions: 1024 x 1024 (whole problem space)
  - Local Dimensions: 128 x 128 (work group … executes together)



https://www.khronos.org/assets/uploads/developers/library/2012-pan-pacific-road-show-June/OpenCL-Details-Taiwan_June-2012.pdf

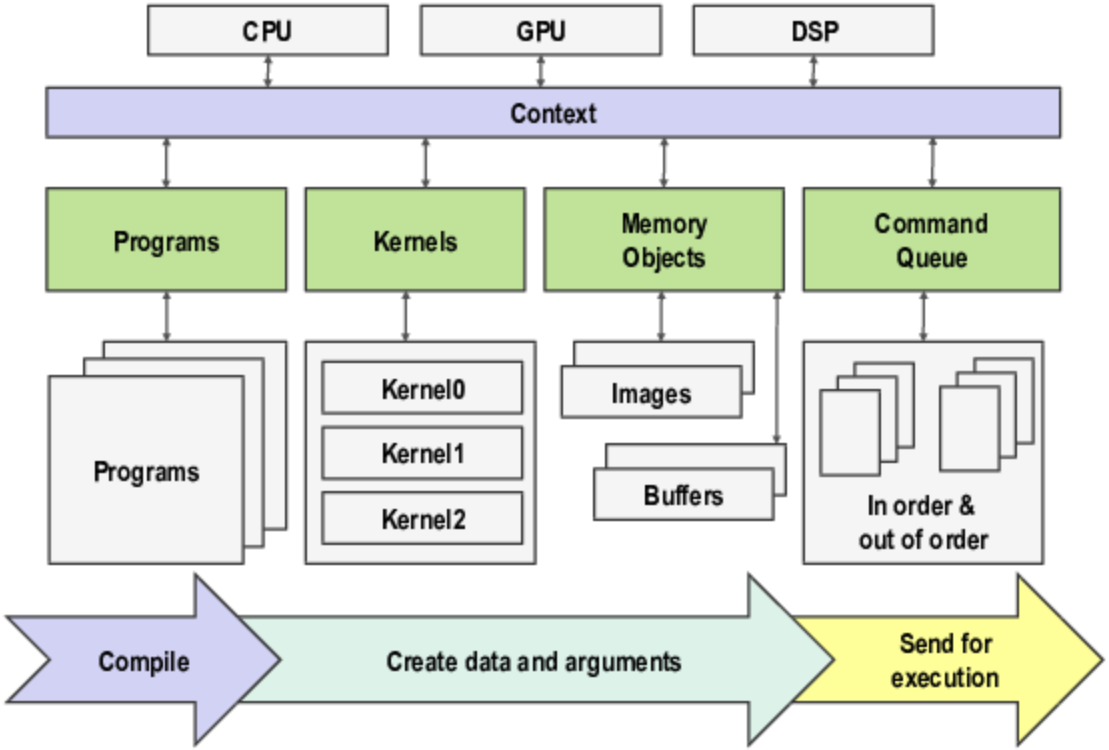
Programming Kernels: OpenCL C

- **Derived from ISO C99**
  - But without some C99 features such as standard C99 headers, function pointers, recursion, variable length arrays, and bit fields
- **Language Features Added**
  - Work-items and workgroups
  - Vector types
  - Synchronization
  - Address space qualifiers
- **Also includes a large set of built-in functions**
  - Image manipulation
  - Work-item manipulation,
  - Math functions, etc.

KHRONOS GROUP

https://www.khronos.org/assets/uploads/developers/library/2012-pan-pacific-road-show-June/OpenCL-Details-Taiwan_June-2012.pdf

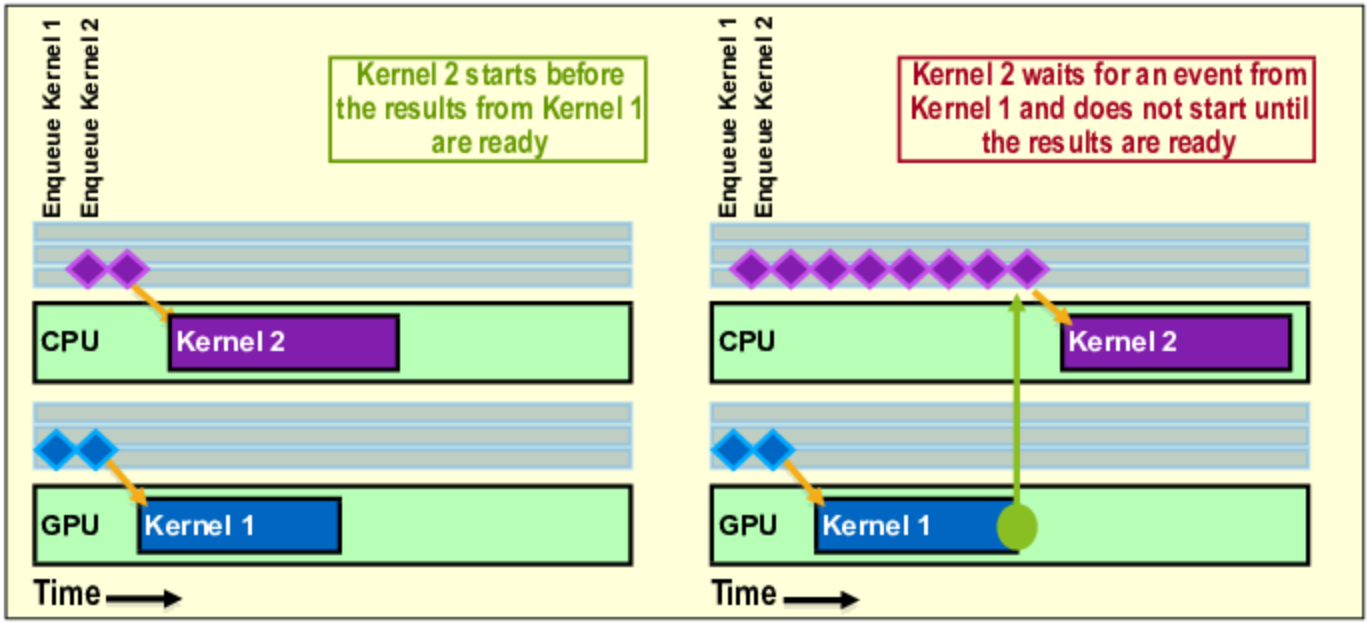
Creating an OpenCL Program



https://www.khronos.org/assets/uploads/developers/library/2012-pan-pacific-road-show-June/OpenCL-Details-Taiwan_June-2012.pdf

Synchronization: Queues & Events

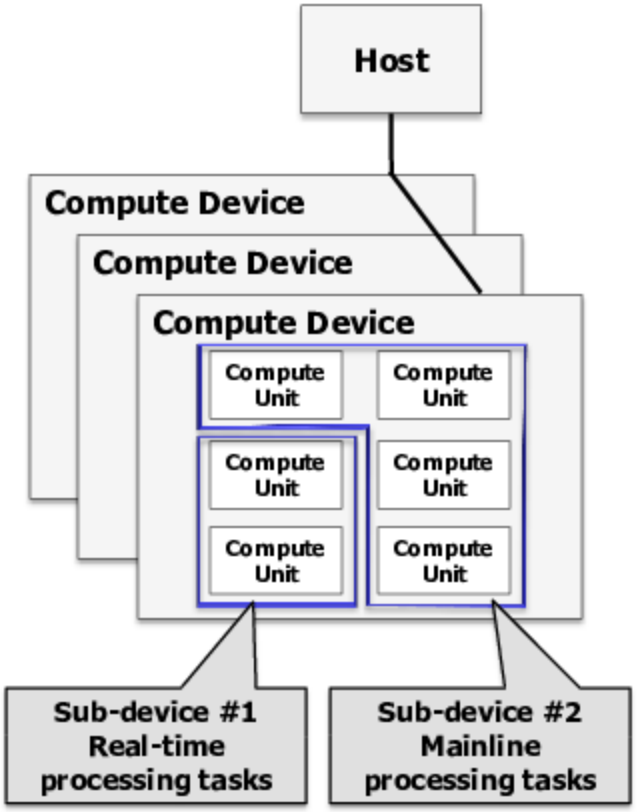
- Events can be used to synchronize kernel executions between queues
- Example: 2 queues with 2 devices



https://www.khronos.org/assets/uploads/developers/library/2012-pan-pacific-road-show-June/OpenCL-Details-Taiwan_June-2012.pdf

Partitioning Devices

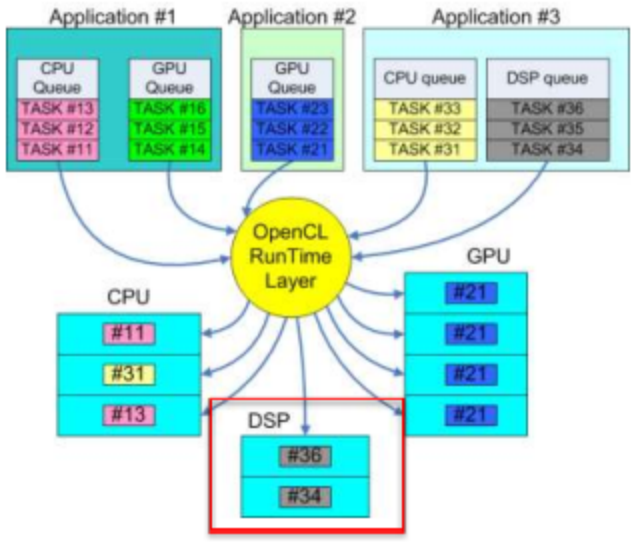
- **Devices can be partitioned into sub-devices**
  - More control over how computation is assigned to compute units
- **Sub-devices may be used just like a normal device**
  - Create contexts, building programs, further partitioning and creating command-queues
- **Three ways to partition a device**
  - Split into equal-size groups
  - Provide list of group sizes
  - Group devices sharing a part of a cache hierarchy



https://www.khronos.org/assets/uploads/developers/library/2012-pan-pacific-road-show-June/OpenCL-Details-Taiwan_June-2012.pdf

Custom Devices and Built-in Kernels

- **Embedded platforms often contain specialized hardware and firmware**
  - That cannot support OpenCL C
- **Built-in kernels can represent these hardware and firmware capabilities**
  - Such as video encode/decode
- **Hardware can be integrated and controlled from the OpenCL framework**
  - Can enqueue built-in kernels to custom devices alongside OpenCL kernels
- **FPGAs are one example of device that can expose built-in kernels**
  - Latest FPGAs can support full OpenCL C as well
- **OpenCL becomes a powerful coordinating framework for diverse resources**
  - Programmable and non-programmable devices controlled by one run-time

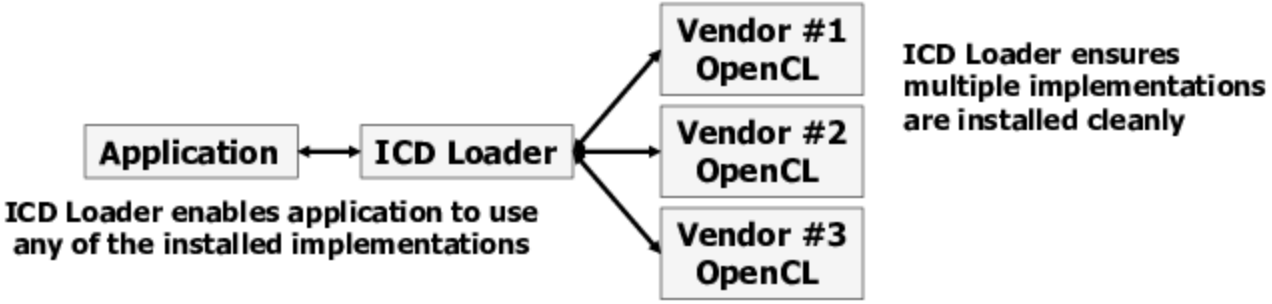


Built-in kernels enable control of specialized processors and hardware from OpenCL run-time

https://www.khronos.org/assets/uploads/developers/library/2012-pan-pacific-road-show-June/OpenCL-Details-Taiwan_June-2012.pdf

Installable Client Driver

- **Analogous to OpenGL ICDs in use for many years**
  - Used to handle multiple OpenGL implementations installed on a system
- **Optional extension**
  - Platform vendor will choose whether to use ICD mechanisms
- **Khronos OpenCL installable client driver loader**
  - Exposes multiple separate vendor installable client drivers (Vendor ICDs)
- **Application can access all vendor implementations**
  - The ICD Loader acts as a de-multiplexor



https://www.khronos.org/assets/uploads/developers/library/2012-pan-pacific-road-show-June/OpenCL-Details-Taiwan_June-2012.pdf

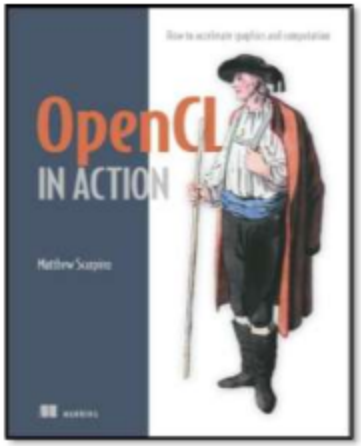
OpenCL Desktop Implementations

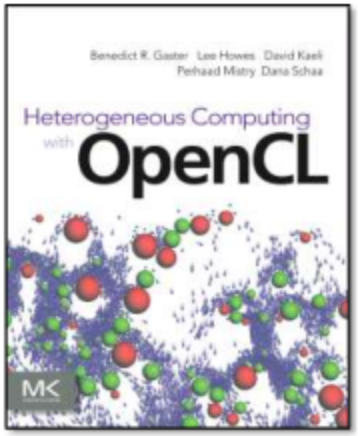
- http://developer.amd.com/zones/OpenCLZone/
- http://software.intel.com/en-us/articles/opencl-sdk/
- http://developer.nvidia.com/opencl

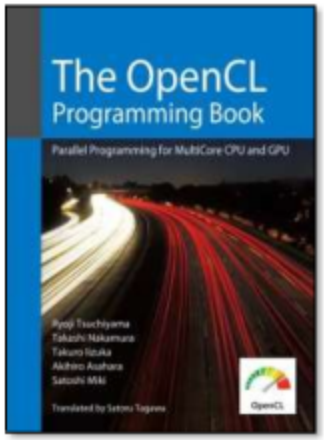
https://www.khronos.org/assets/uploads/developers/library/2012-pan-pacific-road-show-June/OpenCL-Details-Taiwan_June-2012.pdf

OpenCL Books – Available Now!

- **OpenCL Programming Guide - The "Red Book" of OpenCL**
  - http://www.amazon.com/OpenCL-Programming-Guide-Aaftab-Munshi/dp/0321749642
- **OpenCL in Action**
  - http://www.amazon.com/OpenCL-Action-Accelerate-Graphics-Computations/dp/1617290173/
- **Heterogeneous Computing with OpenCL**
  - http://www.amazon.com/Heterogeneous-Computing-with-OpenCL-ebook/dp/B005JRHYUS
- **The OpenCL Programming Book**
  - http://www.fixstars.com/en/opencl/book/







https://www.khronos.org/assets/uploads/developers/library/2012-pan-pacific-road-show-June/OpenCL-Details-Taiwan_June-2012.pdf

OpenCL: Definitions

Devices: Collection of OpenCL devices used by the host

Kernels: The OpenCL functions that run on OpenCL devices

Program Objects: The program source and executable that implements the kernel

Memory Objects: A set of memory objects visible to the host and the OpenCL devices. Memory objects contain values that can be operated on by instances of a kernel.

OpenCL: Definitions...

Global Memory: This memory region permits read/write access to all work-items in all work-groups. Work-items can read from or write to any element of a memory object. Reads and writes to global memory may be cached depending on the capabilities of the device.

Constant Memory: A region of global memory that remains constant during the execution of a kernel. The host allocates and initializes memory objects placed into constant memory.

Local Memory: Local Memory: A memory region local to a work-group. This memory region can be used to allocate variables that are shared by all work-items in that work-group. It may be implemented as dedicated regions of memory on the OpenCL device. Alternatively, the local memory region may be mapped onto sections of the global memory.

Private Memory: A region of memory private to a work-item. Variables defined in one work-item's private memory are not visible to another work-item.

Memory Consistency

OpenCL uses a relaxed consistency memory model; i.e. the state of memory visible to a workitem is not guaranteed to be consistent across the collection of work-items at all times.

Within a work-item memory has load / store consistency. Local memory is consistent across work-items in a single work-group at a work-group barrier. Global memory is consistent across work-items in a single work-group at a work-group barrier, but there are no guarantees of memory consistency between different work-groups executing a kernel.

Memory consistency for memory objects shared between enqueued commands is enforced at a synchronization point.

For more information see the very readable OpenCL Standard document by the Khronos Group.

More remarks

- The current standard version is 2.2. However, implementations are not yet widely spread.
- The most widely used standard is 1.2 (NVIDIA only supports 1.2). Intel and AMD also support OpenCL 2.
- In this module we focus on OpenCL 1.2.
- Cuda is very dominant in the HPC market. In this module we use OpenCL since it runs on a variety of devices and its abstract device model makes a transition to Cuda easy (use PyCuda for Cuda from Python).
- In the future OpenCL will merge with the Vulkan API for fast computer graphics (already used by many games). This will provide a unified graphics and compute API.

Distributed Memory Parallelization

- Thread based parallelization only works for shared memory
- All threads run within the same process on the same physical node
- Different approach needed for communication between processes on the same and different nodes
- Standard framework is MPI (Message Passing Interface)

A brief overview of MPI

- MPI-1 standard published in 1994
- MPI-2 came in 1996. Added parallel I/O, one-sided communication, dynamic process management
- Most MPI programs only require a handful of MPI commands
- MPI is mainly a distributed memory concept. Some shared memory features introduced in MPI-3
- MPI is tremendously successful. Virtually every distributed HPC code uses it for communication
- It scales well from desktops to peta-scale architectures
- The main used implementations are MPICH, OpenMPI and Intel MPI
- MPI is directly callable from Fortran, C/C++. But has bindings for other languages (including Python)

MPI Hello World

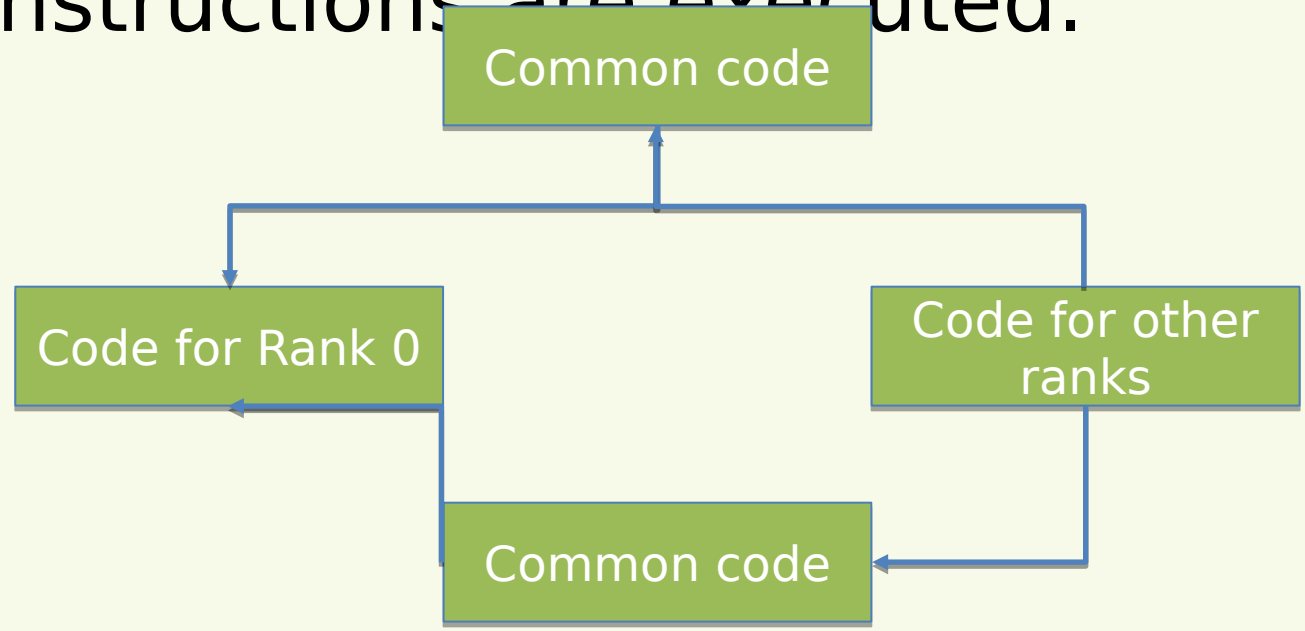
```c
#include <mpi.h>
#include <stdio.h>

int main() {
  // Initialize the MPI environment
  MPI_Init(NULL, NULL);

  // Get the number of processes
  int world_size;
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);

  // Get the rank of the process
  int world_rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

  // Get the name of the processor
  char processor_name[MPI_MAX_PROCESSOR_NAME];
  int name_len;
  MPI_Get_processor_name(processor_name, &name_len);

  // Print off a hello world message
  printf("Hello world from processor %s, rank %d"
         " out of %d processors\n",
         processor_name, world_rank, world_size);

  // Finalize the MPI environment.
  MPI_Finalize();
}
```

- Compile with
  mpic++ ./hello.cpp
- Run with
- mpirun –n 4 ./a.out

Hello world from processor Timos-MBP-2, rank 0 out of 4 processors
Hello world from processor Timos-MBP-2, rank 1 out of 4 processors
Hello world from processor Timos-MBP-2, rank 2 out of 4 processors
Hello world from processor Timos-MBP-2, rank 3 out of 4 processors

The MPI Model

- Typically one program for all MPI processes
- Depending on the MPI rank different instructions are executed.



Sending and receiving data

```cpp
#include <iostream>
#include <mpi.h>

int main() {

  int my_number;
  int other_number;

  MPI_Init(NULL, NULL);

  // Get the number of processes
  int world_size;
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);

  // Get the rank of the process
  int world_rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

  int error;

  if (world_rank == 0) {
    my_number = 42;
    error = MPI_Send(&my_number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
  }

  if (world_rank == 1) {
    MPI_Status status;
    error = MPI_Recv(&other_number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    std::cout << "I have received from rank 0 the number " << other_number
              << "." << std::endl;
  }
}
```

Send data

Receive data

- Run with
  mpirun –n 2 ./send_receive

More on MPI soon via the mpi4py interface

Introducing mpi4py

```
1   #!/usr/bin/env python
2   """
3   Parallel Hello World
4   """
5
6   from mpi4py import MPI
7   import sys
8
9   size = MPI.COMM_WORLD.Get_size()
10  rank = MPI.COMM_WORLD.Get_rank()
11  name = MPI.Get_processor_name()
12
13  sys.stdout.write(
14      "Hello, World! I am process %d of %d on %s.\n"
15      % (rank, size, name))
16
```

mpirun -n 2 python hello.py

Sending data

```
1   from mpi4py import MPI
2   import numpy as np
3   import sys
4
5   comm = MPI.COMM_WORLD
6
7   size = comm.Get_size()
8   rank = comm.Get_rank()
9
10  sendbuf = np.empty(1, dtype='i')
11  recvbuf = np.empty(1, dtype='i')
12
13  sendbuf[:] = rank
14
15  if rank == 0:
16      comm.Send(sendbuf, rank + 1)
17  elif rank > 0 and rank < size-1:
18      comm.Recv(recvbuf, rank - 1)
19      comm.Send(sendbuf, rank + 1)
20  else:
21      comm.Recv(recvbuf, rank - 1)
22
23  if rank > 0:
24      sys.stdout.write(
25          "Process %d received data %d.\n"
26          % (rank, recvbuf[0]))
27
```

- Each rank sends data to its successor
- Need to make sure that send and receive operations do not block each other
- Better alternative: Use non-blocking send/receive with Isend/Irecv

Non-blocking send

```python
from mpi4py import MPI
import numpy as np
import sys

comm = MPI.COMM_WORLD

size = comm.Get_size()
rank = comm.Get_rank()

sendbuf = np.empty(1, dtype='i')
recvbuf = np.empty(1, dtype='i')

sendbuf[:] = rank

if rank < size - 1:
    comm.Isend(sendbuf, rank + 1)

if rank > 0:
    comm.Recv(recvbuf, rank - 1)

comm.Barrier()

if rank > 0:
    sys.stdout.write(
        "Process %d received data %d.\n"
        % (rank, recvbuf[0]))

```

- Each rank sends data to its successor
- Send is non-blocking. The process does not wait for the actual send to take place.
- We enforce all process to finish the send/receive operations with a barrier

A note on types

- Communication commands in mpi4py can start with a large letter (e.g. Send) or a small letter (send)
- If we use a large letter native data types are used. Compatible Numpy types are automatically recognized
- With smalll letter commands can send any objects, but executation slow due to necessary serialization of object.

Scatter data

```python
"""Scatter data across all processes."""

from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

sendbuf = None
if rank == 0:
    sendbuf = np.empty([size, 10], dtype='i')
    sendbuf.T[:,:] = range(size)
recvbuf = np.empty(10, dtype='i')
comm.Scatter(sendbuf, recvbuf, root=0)
assert np.allclose(recvbuf, rank)

```

```
>>> sendbuf
array([[0, 0, 0, 0, 0],
       [1, 1, 1, 1, 1],
       [2, 2, 2, 2, 2]],
       dtype=int32)

On rank 1 have
>>> recvbuf
array([1, 1, 1, 1, 1],
dtype=int32)
```

- Given an input array sendbuf. Distribute the array elements onto all processes.
- For unequal chunk sizes uses Scatterv
- Note that memory buffers need already exist!

Reduce data

```
"""Scatter data across all processes."""

from mpi4py import MPI
import numpy as np
import sys

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

sendbuf = np.empty(1, dtype='i')
sendbuf[:] = 1

recvbuf = np.empty(1, dtype='i')

comm.Allreduce(sendbuf, recvbuf, MPI.SUM)

sys.stdout.write(
    "Process %d has data %d.\n"
    % (rank, recvbuf[0]))

```

- Sum data across all processes.
- Every process gets the sum of the data.
- Different types of reducers possible.

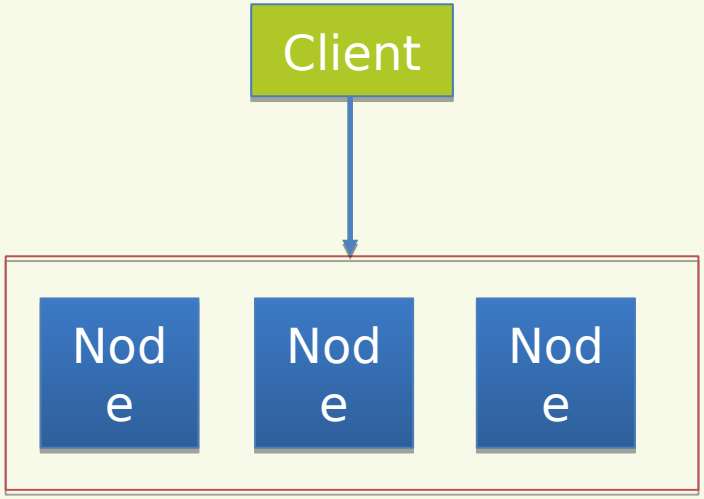
More remarks on MPI

- Have given a small sample of MPI operations
- A range of sophisticated MPI communication and process management operations available
- Many are supported by mpi4py
- In practice we often only need a few (send/receive and global communication)

IPython clusters

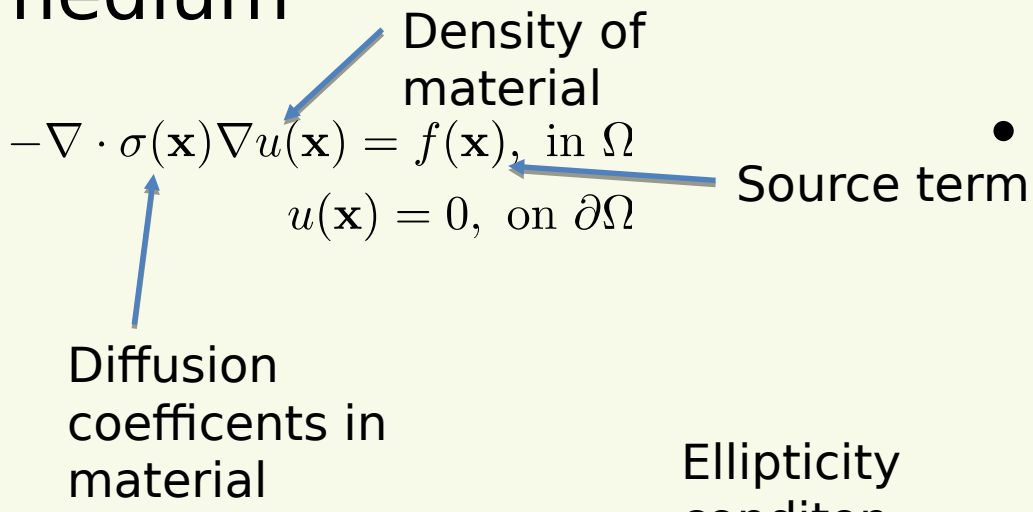
- Supercomputing on your fingertips using Ipyparallel
- Have a local Python client that communicates with an HPC resource
- More info on

  https://ipyparallel.readthedocs.io/en/latest/



Modeling a large-scale PDE

Diffusion in a complex medium



- Diffusion of a pollutant in soil
- Often diffusion coefficient a random parameter

Ellipticity conditon:

$$\langle \sigma(\mathbf{x})\mathbf{y}, \mathbf{y} \rangle \geq C \|\mathbf{y}\|_2^2, \ \forall \mathbf{y} \in \mathbf{R}^3, \ C > 0$$

Deriving the weak form

Multiply with test function in $H_0^1(\Omega)$

$$-\nabla \cdot \sigma(\mathbf{x}) \nabla u(\mathbf{x}) v(\mathbf{x}) = f(\mathbf{x}) v(\mathbf{x})$$

Integration by parts gives

$$\int_\Omega [\sigma(\mathbf{x}) \nabla u(\mathbf{x})] \cdot \nabla v(\mathbf{x}) d\mathbf{x} - \int_{\partial\Omega} \sigma(\mathbf{x}) \frac{\partial u(\mathbf{x})}{\partial n(\mathbf{x})} v(\mathbf{x}) ds(\mathbf{x}) = \int_\Omega f(\mathbf{x}) v(\mathbf{x})$$

Apply the boundary condition to arrive at

$$\int_\Omega [\sigma(\mathbf{x}) \nabla u(\mathbf{x})] \cdot \nabla v(\mathbf{x}) d\mathbf{x} = \int_\Omega f(\mathbf{x}) v(\mathbf{x})$$

Weak form of the partial differential equation

Deriving the weak form...

Find $u \in H_0^1(\Omega)$ such that

$$\int_\Omega [\sigma(\mathbf{x})\nabla u(\mathbf{x})] \cdot \nabla v(\mathbf{x}) d\mathbf{x} = \int_\Omega f(\mathbf{x}) v(\mathbf{x})$$

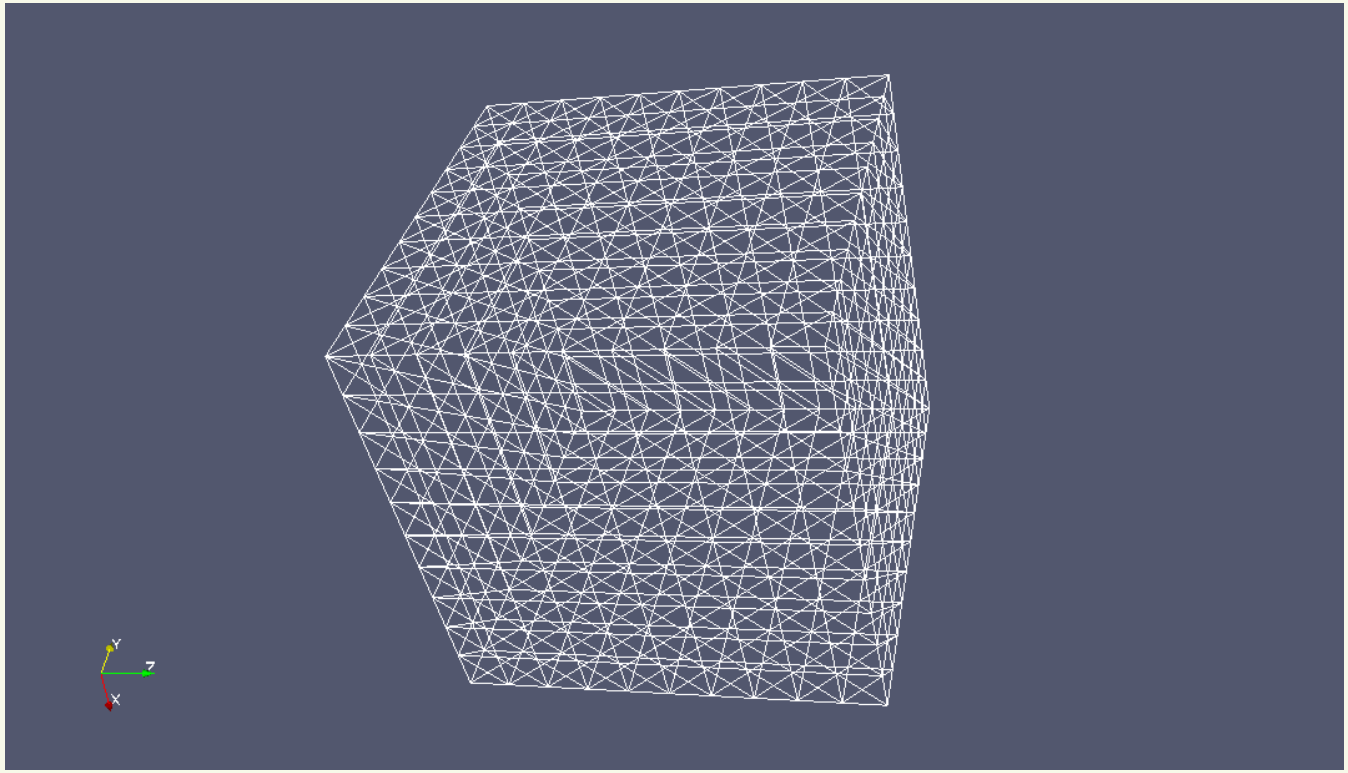
for all $v \in H_0^1(\Omega)$.

Weak form of the partial differential equation

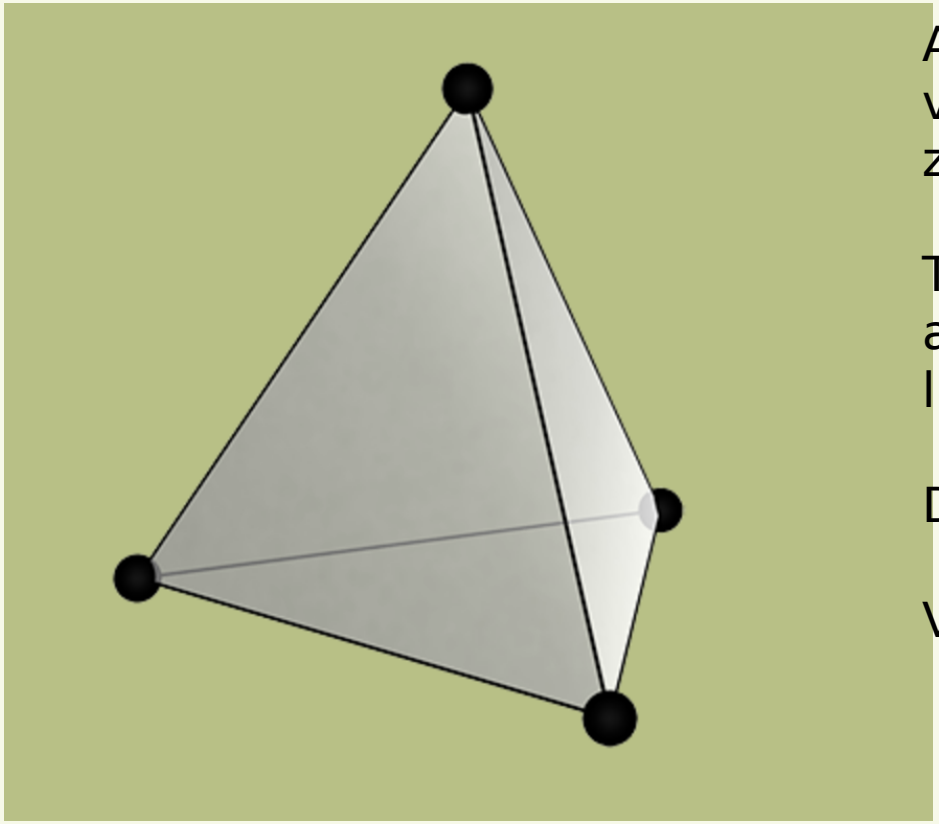
- Create a computational grid of the domain
- Define a discrete function space over the domain
- Assemble the equation on the discrete function space
- Solve the resulting linear system of equations

Creating a computational grid

```
from dolfin import *
Mesh = UnitCubeMesh(10,
10, 10)
```



Standard P1 Elements



A single P1 basis function has the value 1 at one node and the value zero at all other nodes.

The set of all P1 basis functions over a grid span a space of piecewise linear, continuous basis functions.

Define a P1 function space in Fenics:

V = FunctionSpace(mesh, "Lagrange"

The discrete weak form

- Denote by $H^1_{0,h}(\Omega)$ the finite dimensional space spanned by P1 basis functions over a given grid with element diameter h.
- Discretize the variational problem by restricting it into this finite dimensional space.

$$\text{Find } u \in H^1_{0,h}(\Omega) \text{ such that}$$

$$\int_\Omega [\sigma(\mathbf{x})\nabla u(\mathbf{x})] \cdot \nabla v(\mathbf{x}) d\mathbf{x} = \int_\Omega f(\mathbf{x}) v(\mathbf{x})$$

$$\text{for all } v \in H^1_{0,h}(\Omega).$$

Define

$$a(u,v) := \int_\Omega [\sigma(\mathbf{x})\nabla u(\mathbf{x})] \cdot \nabla v(\mathbf{x}) d\mathbf{x}$$

Obtaining a matrix

Let ψ_j be the P1 basis function associated with the jth node in the grid.

We can therefore write

$$H^1_{0,h}(\Omega) := \mathrm{span}\{\psi_1, \ldots, \psi_N\}$$

Any function ψ in $H^1_{0,h}(\Omega)$ can be written as

$$\psi = \sum_{j=1}^{N} x_j \Psi_j$$

for some coefficient vector x.

Define $A \in \mathbb{R}^{N \times N}$ by $A_{i,j} = a(\psi_j, \psi_i)$ and $\overline{f}_j = \int_\Omega f(\mathbf{x})\psi_j(\mathbf{x})d\mathbf{x}$

The variational problem is equivalent to

$$\text{Find } x \in \mathbb{R}^N \text{ such that}$$

$$y^T A x = y^T \overline{f}$$

$$\text{for all } y \in \mathbb{R}^N.$$

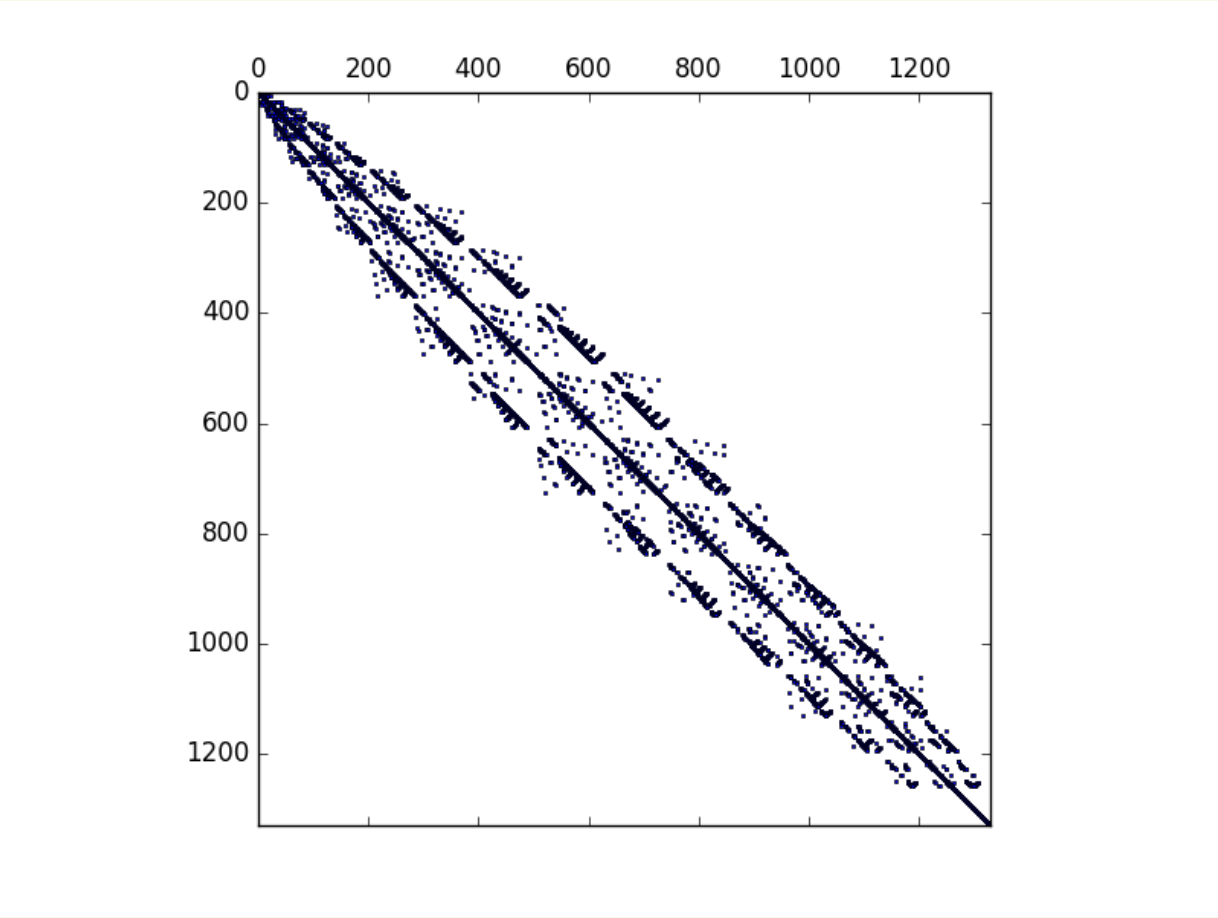
Obtaining a matrix...

This is equivalent to the linear system of equations

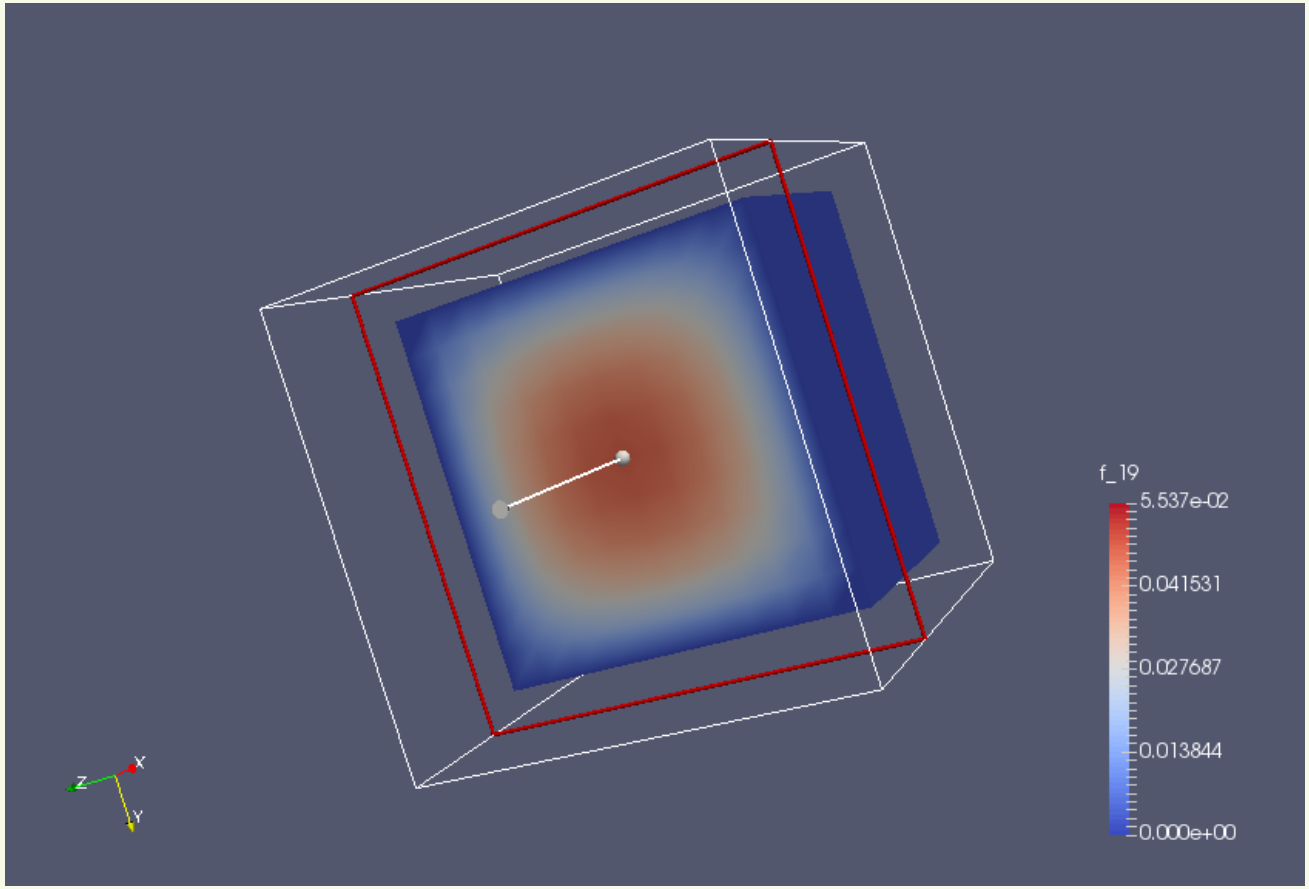
$$Ax = \overline{f}$$

- The system matrix is sparse, that is only a small percentage of the elements is nonzero.
- For smaller problem sizes a direct solution is possible. For larger problems iterative solvers need to be used.
- Very large-scale practical applications can have billions of unknowns.

Obtaining a matrix...



The complete solution



An example FeniCS Script

```
1   from dolfin import *
2
3   mesh = UnitCubeMesh(10, 10, 10)
4
5   V = FunctionSpace(mesh, "Lagrange", 1)
6   def boundary(x):
7       return (x[0] < DOLFIN_EPS or x[0] > 1 - DOLFIN_EPS
8           or x[1] < DOLFIN_EPS or x[1] > 1 - DOLFIN_EPS
9           or x[2] < DOLFIN_EPS or x[2] > 1 - DOLFIN_EPS)
10
11  u0 = Constant(0)
12
13  bc = DirichletBC(V, u0, boundary)
14
15  u = TrialFunction(V)
16  v = TestFunction(V)
17  f = Expression('1', degree=1)
18
19  a = inner(grad(u), grad(v)) * dx
20  rhs = f * v * dx
21
22  A = assemble(a)
23  b = assemble(rhs)
24  bc.apply(A, b)
25  sol = Vector()
26  solve(A, sol, b, 'lu')
27  output_file = File('poisson.pvd')
28  output_file << Function(V, sol)
29
30
```

Define the mesh

Specify boundary region (for complex domains other ways are possible)

Definition of the weak form

Assemble the operators and apply boundary condition

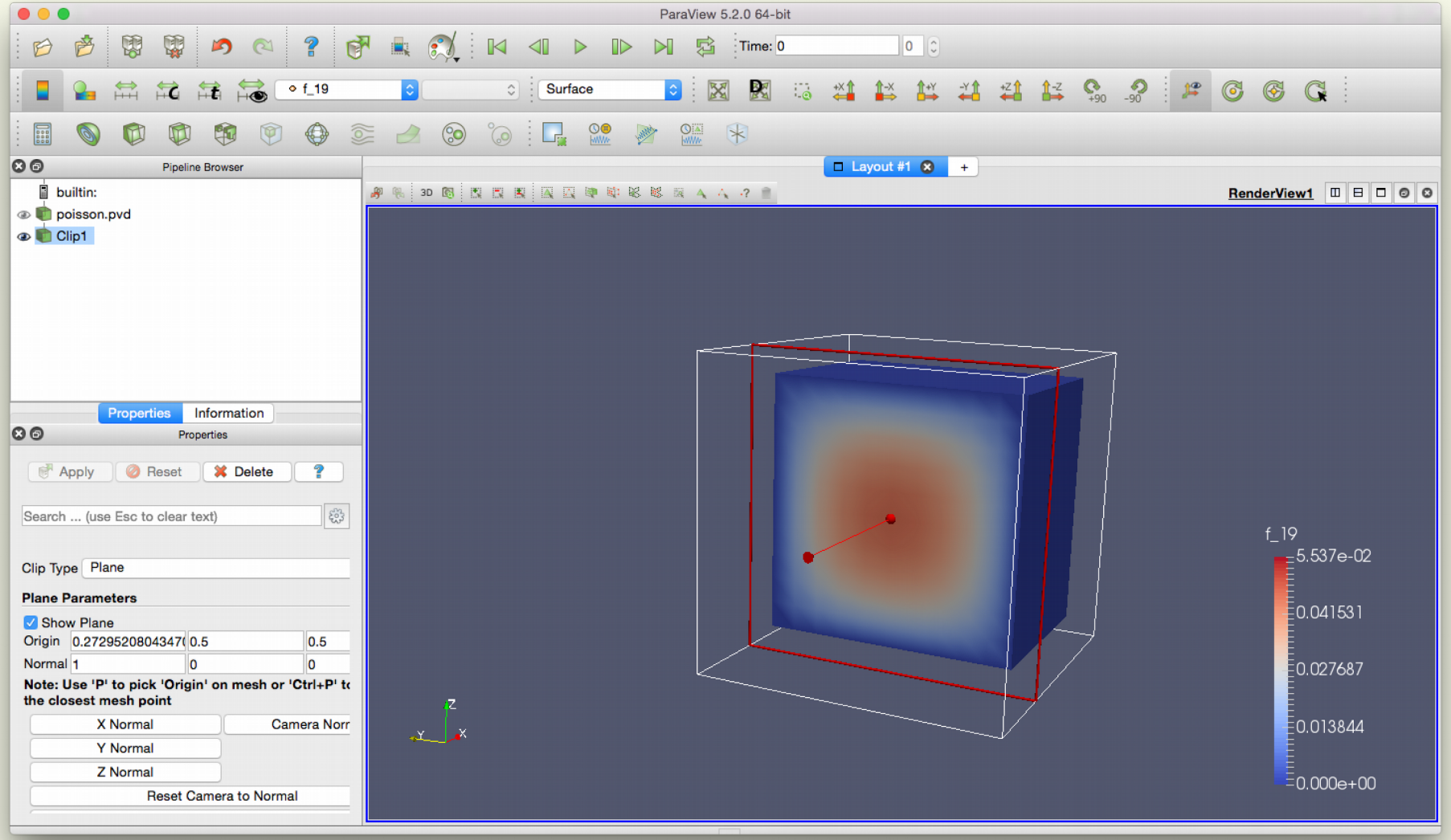
Solve

Write out the solution

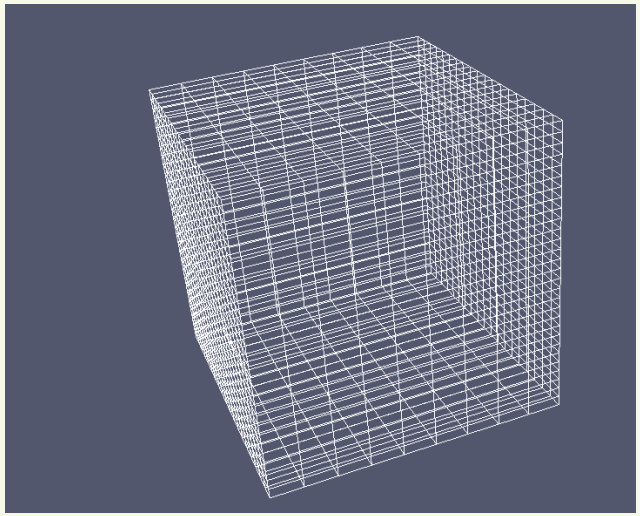
Visualising with VTK

- VTK is widely used data format for storing meshes and data over meshes
- Parallel operations possible
- Developed by Kitware
- Bindings available for all major programming languages (including Python)
- Example viewers: Paraview, VisIt, Mayavi2, etc.

Paraview

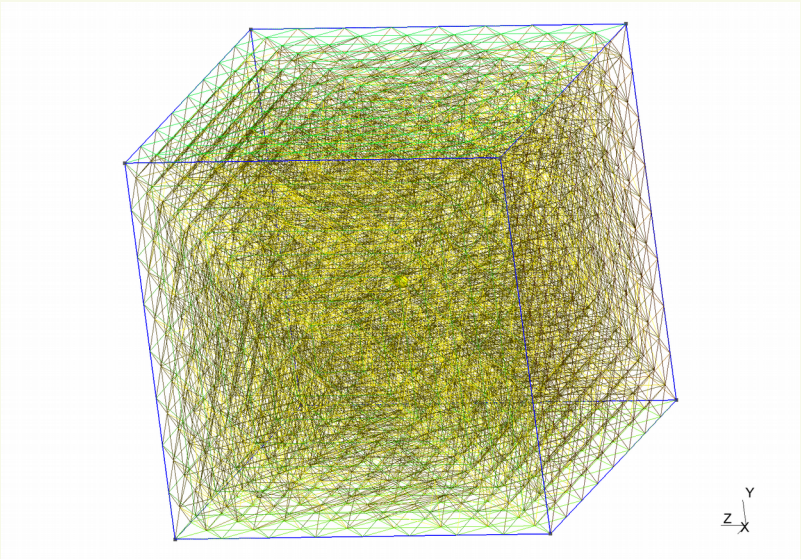

ParaView 5.2.0 64-bit
Time: 0
f_19
Surface
Pipeline Browser
builtin:
poisson.pvd
Clip1
Properties
Information
Properties
Apply
Reset
Delete
Search ... (use Esc to clear text)
Clip Type
Plane
Plane Parameters
Show Plane
Origin
0.2729520804347(
0.5
0.5
Normal
1
0
0
Note: Use 'P' to pick 'Origin' on mesh or 'Ctrl+P' t
the closest mesh point
X Normal
Camera Norr
Y Normal
Z Normal
Reset Camera to Normal
Layout #1
RenderView1
f_19
5.537e-02
0.041531
0.027687
0.013844
0.000e+00

Computational Grids



- Each point uniquely identified by index triplet (i, j, k)
- Fast algorithms due to regular structure (in particular good for GPUs)
- Used in many finite difference applications
- Only simple geometries possible

- Need to store points and connectivity explicitly
- Perfect for generic unstructed geometries
- Allows refinement towards complex geometric features



Unstructured grid data structures

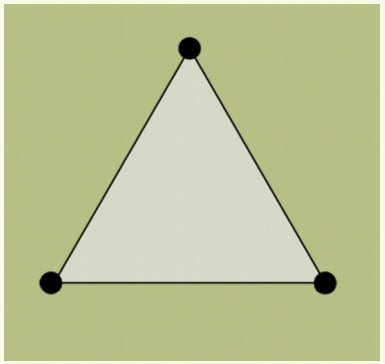
- Typically two sets of data:
  - A point structure that stores the nodal information
  - An element structure that stores the connectivity between points

```
4 13 12 10 8
4 10 12 11 8
4 9 10 11 8
4 13 10 9 8
4 5 13 6 9
4 13 7 6 10
4 7 4 12 13
4 5 9 1 8
4 0 1 11 8
4 5 4 13 8
4 3 7 12 10
4 6 10 2 9
4 4 0 12 8
4 10 3 2 11
```

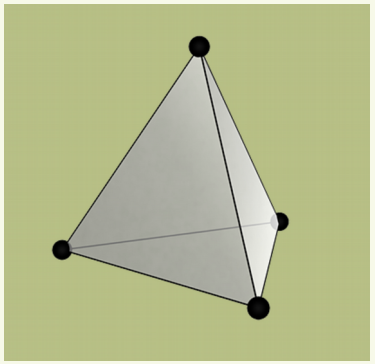
```
# vtk DataFile Version 2.0
untitled, Created by Gmsh
ASCII
DATASET UNSTRUCTURED_GRID
POINTS 14 double
0 0 0
1 0 0
1 1 0
0 1 0
0 0 1
1 0 1
1 1 1
0 1 1
0.5 0 0.5
1 0.5 0.5
0.5 1 0.5
0.5 0.5 0
0 0.5 0.5
0.5 0.5 1

CELLS 68 268
1 0
1 1
1 2
1 3
1 4
1 5
1 6
1 7
2 0 4
2 4 5
2 5 1
2 1 0
```

Frequent element types

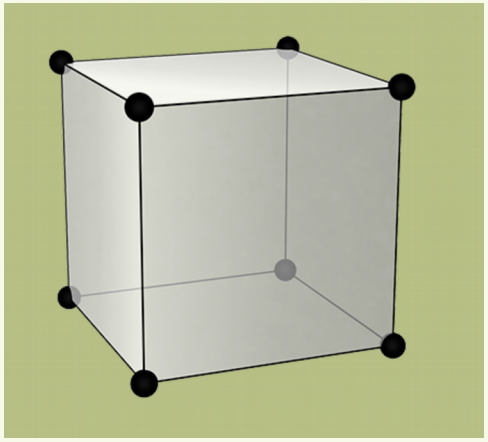
Line



Triangle



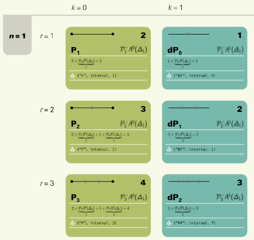
Tetrahedron



Cube

- These are lowest order elements
- Higher order variants available
- Many more element types exist
- See www.femtable.org

Periodic Table of the Finite Elements



$\mathcal{P}_r^- \Lambda^k$

The shape function space for $\mathcal{P}_r^-\Lambda^k$ is

$$\mathcal{P}_{r-1}\Lambda^k + \kappa\mathcal{P}_{r-1}\Lambda^{k+1},$$

where κ is the Koszul differential.[7] It includes the full polynomial space $\mathcal{P}_{r-1}\Lambda^k$, is included in $\mathcal{P}_r\Lambda^k$, and has dimension

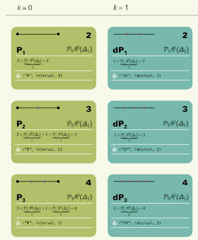
$$\dim \mathcal{P}_r^-\Lambda^k(\Delta_n) = \binom{r+n}{r+k}\binom{r+k-1}{k}.$$

The degrees of freedom are given on faces f of dimension $d \geq k$ by moments of the trace weighted by a full polynomial space:

$$u \mapsto \int_f (\mathrm{tr}_f\, u) \wedge q, \quad q \in \mathcal{P}_{r+k-d-1}\Lambda^{d-k}(f).$$

The spaces with constant degree r form a complex:

$$\mathcal{P}_r^-\Lambda^0 \xrightarrow{\mathrm{d}} \mathcal{P}_r^-\Lambda^1 \xrightarrow{\mathrm{d}} \cdots \xrightarrow{\mathrm{d}} \mathcal{P}_r^-\Lambda^n.$$



$\mathcal{P}_r \Lambda^k$

The shape function space for $\mathcal{P}_r\Lambda^k$ consists of all differential k-forms with polynomial coefficients of degree at most r, and has dimension

$$\dim \mathcal{P}_r\Lambda^k(\Delta_n) = \binom{r+n}{r+k}\binom{r+k}{k}.$$

The degrees of freedom are given on faces f of dimension $d \geq k$ by moments of the trace weighted by a $\mathcal{P}_r^-$ space:

$$u \mapsto \int_f (\mathrm{tr}_f\, u) \wedge q, \quad q \in \mathcal{P}_{r+k-d}^-\Lambda^{d-k}(f).$$

The spaces with decreasing degree r form a complex:

$$\mathcal{P}_r\Lambda^0 \xrightarrow{\mathrm{d}} \mathcal{P}_{r-1}\Lambda^1 \xrightarrow{\mathrm{d}} \cdots \xrightarrow{\mathrm{d}} \mathcal{P}_{r-n}\Lambda^n.$$

$\mathcal{Q}_r^- \Lambda^k$

This family is constructed from the complex of 1-dimensional finite elements using a tensor product construction.[10] The shape function space on the unit cube $\square_n = I^n$ is given by

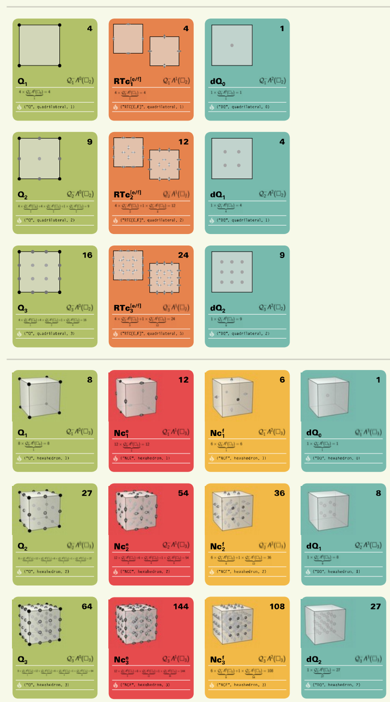
$$\bigoplus_{\sigma\in\Sigma(k,n)} \left[\bigotimes_{i=1}^{n} \mathcal{P}_{r-\delta_{i,\sigma}}(I)\right] \mathrm{d}x^{\sigma_1}\wedge\cdots\wedge \mathrm{d}x^{\sigma_k},$$

where $\Sigma(k,n)$ denotes the increasing maps $\{1,\dots,k\} \to \{1,\dots,n\}$. Its dimension is $\dim \mathcal{Q}_r^-\Lambda^k(\square_n) = \binom{n}{k}r^k(r+1)^{n-k}$. The degrees of freedom are

$$u \mapsto \int_f (\mathrm{tr}_f\, u) \wedge q, \quad q \in \mathcal{Q}_{r-1}^-\Lambda^{d-k}(f).$$

The spaces with constant degree r form a complex:

$$\mathcal{Q}_r^-\Lambda^0 \xrightarrow{\mathrm{d}} \mathcal{Q}_r^-\Lambda^1 \xrightarrow{\mathrm{d}} \cdots \xrightarrow{\mathrm{d}} \mathcal{Q}_r^-\Lambda^n.$$



$\mathcal{S}_r \Lambda^k$

The shape function space for $\mathcal{S}_r\Lambda^k$ is given by

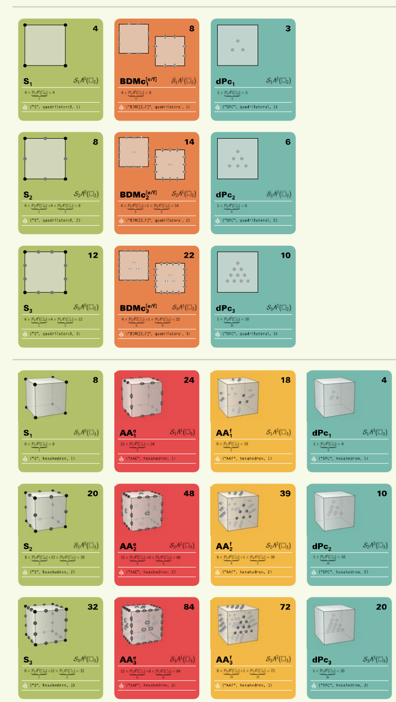
$$\mathcal{P}_r\Lambda^k \oplus \bigoplus_{\ell\geq 1}[\kappa\mathcal{H}_{r+\ell-1,\ell}\Lambda^{k+1} \oplus \mathrm{d}\kappa\mathcal{H}_{r+\ell,\ell}\Lambda^k],$$

where $\mathcal{H}_{r,l}\Lambda^k$ consists of homogeneous polynomial k-forms of degree r which are linear and undifferentiated in at least ℓ variables.[11] Its dimension is $\dim \mathcal{S}_r\Lambda^k(\square_n) = \sum_{d\geq k} 2^{n-d}\binom{n}{d}\binom{r-d+2k}{d}\binom{d}{k}$. The degrees of freedom are

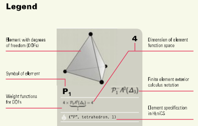
$$u \mapsto \int_f (\mathrm{tr}_f\, u) \wedge q, \quad q \in \mathcal{P}_{r-2(d-k)}\Lambda^{d-k}(f).$$

The spaces with decreasing degree r form a complex:

$$\mathcal{S}_r\Lambda^0 \xrightarrow{\mathrm{d}} \mathcal{S}_{r-1}\Lambda^1 \xrightarrow{\mathrm{d}} \cdots \xrightarrow{\mathrm{d}} \mathcal{S}_{r-n}\Lambda^n.$$



Legend



$$\dim \mathcal{P}_r^-\Lambda^k(\Delta_n) = \binom{r+n}{r+k}\binom{r+k-1}{k}$$

	k	r = 1	2	3	4	5	6	7
n = 1	0	2	3	4	5	6	7	8
	1	1	2	3	4	5	6	7
n = 2	0	3	6	10	15	21	28	36
	1	3	8	15	24	35	48	63
	2	1	3	6	10	15	21	28
n = 3	0	4	10	20	35	56	84	120
	1	6	20	45	84	140	216	315
	2	4	15	36	70	120	189	280
	3	1	4	10	20	35	56	84
n = 4	0	5	15	35	70	126	210	330
	1	10	40	105	224	420	720	1155
	2	10	45	126	280	540	945	1540
	3	5	24	70	160	315	560	924
	4	1	5	15	35	70	126	210

$$\dim \mathcal{P}_r\Lambda^k(\Delta_n) = \binom{r+n}{r+k}\binom{r+k}{k}$$

	k	r = 1	2	3	4	5	6	7
n = 1	0	2	3	4	5	6	7	8
	1	2	3	4	5	6	7	8
n = 2	0	3	6	10	15	21	28	36
	1	6	12	20	30	42	56	72
	2	3	6	10	15	21	28	36
n = 3	0	4	10	20	35	56	84	120
	1	12	30	60	105	168	252	360
	2	12	30	60	105	168	252	360
	3	4	10	20	35	56	84	120
n = 4	0	5	15	35	70	126	210	330
	1	20	60	140	280	504	840	1320
	2	30	90	210	420	756	1260	1980
	3	20	60	140	280	504	840	1320
	4	5	15	35	70	126	210	330

$$\dim \mathcal{Q}_r^-\Lambda^k(\square_n) = \binom{n}{k}r^k(r+1)^{n-k}$$

	k	r = 1	2	3	4	5	6	7
n = 1	0	2	3	4	5	6	7	8
	1	1	2	3	4	5	6	7
n = 2	0	4	9	16	25	36	49	64
	1	4	12	24	40	60	84	112
	2	1	4	9	16	25	36	49
n = 3	0	8	27	64	125	216	343	512
	1	12	54	144	300	540	882	1344
	2	6	36	108	240	450	756	1176
	3	1	8	27	64	125	216	343
n = 4	0	16	81	256	625	1296	2401	4096
	1	32	216	768	2000	4320	8232	14336
	2	24	216	864	2400	5400	10584	18816
	3	8	96	432	1280	3000	6048	10976
	4	1	16	81	256	625	1296	2401

$$\dim \mathcal{S}_r\Lambda^k(\square_n) = \sum_{d=k}^{\min(n,\lfloor r/2\rfloor+k)} 2^{n-d}\binom{n}{d}\binom{r-d+2k}{d}\binom{d}{k}$$

	k	r = 1	2	3	4	5	6	7
n = 1	0	2	3	4	5	6	7	8
	1	2	3	4	5	6	7	8
n = 2	0	4	8	12	17	23	30	38
	1	8	14	22	32	44	58	74
	2	3	6	10	15	21	28	36
n = 3	0	8	20	32	50	74	105	144
	1	24	48	84	135	204	294	408
	2	18	39	72	120	186	273	384
	3	4	10	20	35	56	84	120
n = 4	0	16	48	80	136	216	328	480
	1	64	144	272	472	768	1188	1764
	2	72	168	336	606	1014	1602	2418
	3	32	84	180	340	588	952	1464
	4	5	15	35	70	126	210	330

Finite elements

The table presents the primary spaces of finite elements for the discretization of the fundamental operators of vector calculus: the gradient, curl, and divergence. A finite element space is a space of piecewise polynomial functions on a domain determined by: (1) a mesh of the domain into polyhedral cells called *elements*, (2) a finite dimensional space of polynomial functions on each element called the *shape functions*, and (3) a unisolvent set of functionals on the shape functions of each element called *degrees of freedom* (DOFs), each DOF being associated to a (generalized) face of the element, and specifying a quantity which takes a single value for all elements sharing the face. The element diagrams depict the DOFs and their association to faces.

The spaces $\mathcal{P}_r^-\Lambda^k$ and $\mathcal{P}_r\Lambda^k$ depicted on the left half of the table are the two primary families of finite element spaces for meshes of simplices, and the spaces $\mathcal{Q}_r^-\Lambda^k$ and $\mathcal{S}_r\Lambda^k$ on the right side are for meshes of cubes or boxes. Each is defined in any dimension $n \geq 1$ for each value of the polynomial degree $r \geq 1$, and each value of $0 \leq k \leq n$. The parameter k refers to the operator: the spaces consist of differential k-forms which belong to the domain of the kth exterior derivative. Thus for $k = 0$, the spaces discretize the Sobolev space H^1, the domain of the gradient operator; for $k = 1$, they discretize $H(\mathrm{curl})$, the domain of the curl; for $k = n-1$ they discretize $H(\mathrm{div})$, the domain of the divergence; and for $k = n$, they discretize L^2.

The spaces $\mathcal{P}_r^-\Lambda^0$ and $\mathcal{P}_r\Lambda^0$, which coincide, are the earliest finite elements, going back in the case $r = 1$ of linear elements to *Courant*,[1] and collectively referred to as *the Lagrange elements*. The spaces $\mathcal{P}_{r+1}^-\Lambda^n$ and $\mathcal{P}_r\Lambda^n$, which also coincide, are the discontinuous *Galerkin* elements, consisting of piecewise polynomials with no interelement continuity imposed, first introduced by *Reed and Hill*.[2] The space $\mathcal{P}_r^-\Lambda^1$ in 2 dimensions was introduced by *Raviart and Thomas*[3] and generalized to the 3-dimensional spaces $\mathcal{P}_r^-\Lambda^1$ and $\mathcal{P}_r^-\Lambda^2$ by *Nédélec*,[4] while $\mathcal{P}_r\Lambda^1$ is due to *Brezzi, Douglas and Marini*[5] in 2 dimensions, its generalization to 3 dimensions again due to *Nédélec*.[6] The unified treatment and notation of the $\mathcal{P}_r^-\Lambda^k$ and $\mathcal{P}_r\Lambda^k$ families is due to *Arnold, Falk and Winther* as part of *finite element exterior calculus*,[7] extending earlier work of *Hiptmair* for the $\mathcal{P}_r^-\Lambda^k$ family.[8] The space $\mathcal{P}_1^-\Lambda^k$ is the span of the *elementary forms* introduced by *Whitney*.[9]

The family $\mathcal{Q}_r^-\Lambda^k$ of cubical elements can be derived from the 1-dimensional Lagrange and discontinuous Galerkin elements by a tensor product construction detailed by *Arnold, Boffi and Bonizzoni*,[10] but for the most part were presented individually along with the corresponding simplicial elements in the papers mentioned. The second cubical family $\mathcal{S}_r\Lambda^k$ is due to *Arnold and Awanou*.[11]

The finite elements in this table have been implemented as part of the *FEniCS Project*.[12,13,14] Each may be referenced in the *Unified Form Language* (UFL)[15] by giving its family, shape, and degree, with the family as shown on the table. For example, the space $\mathcal{P}_3\Lambda^2(\Delta_3)$ may be referred to in UFL as:

```
FiniteElement("N2F", tetrahedron, 3)
```

Alternatively, the elements may be accessed in a uniform fashion as:

```
FiniteElement("P-", shape, r, k)
FiniteElement("P",  shape, r, k)
FiniteElement("Q-", shape, r, k)
FiniteElement("S",  shape, r, k)
```

for $\mathcal{P}_r^-\Lambda^k$, $\mathcal{P}_r\Lambda^k$, $\mathcal{Q}_r^-\Lambda^k$, and $\mathcal{S}_r\Lambda^k$, respectively.

References

1. R. Courant, *Bulletin of the American Mathematical Society 49*, 1943.
2. W. H. Reed and T. R. Hill, *Los Alamos report LA-UR-73-479*, 1973.
3. P. A. Raviart and J. M. Thomas, *Lecture Notes in Mathematics 606*, Springer, 1977.
4. J. C. Nédélec, *Numerische Mathematik 35*, 1980.
5. F. Brezzi, J. Douglas Jr., and L. D. Marini, *Numerische Mathematik 47*, 1985.
6. J. C. Nédélec, *Numerische Mathematik 50*, 1986.
7. D. N. Arnold, R. S. Falk, and R. Winther, *Acta Numerica 15*, 2006.
8. R. Hiptmair, *Mathematics of Computation 68*, 1999.
9. H. Whitney, *Geometric Integration Theory*, 1957.
10. D. N. Arnold, D. Boffi, and F. Bonizzoni, *Numerische Mathematik*, 2014.
11. D. N. Arnold and G. Awanou, *Mathematics of Computation*, 2013.
12. A. Logg, K.-A. Mardal, and G. N. Wells (eds.), *Automated Solution of Differential Equations by the Finite Element Method*, Springer, 2012.
13. R. C. Kirby, *ACM Transactions on Mathematical Software 30*, 2004.
14. A. Logg and G. N. Wells, *ACM Transactions on Mathematical Software 37*, 2010.
15. M. Alnæs, A. Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells, *ACM Transactions on Mathematical Software 40*, 2014.

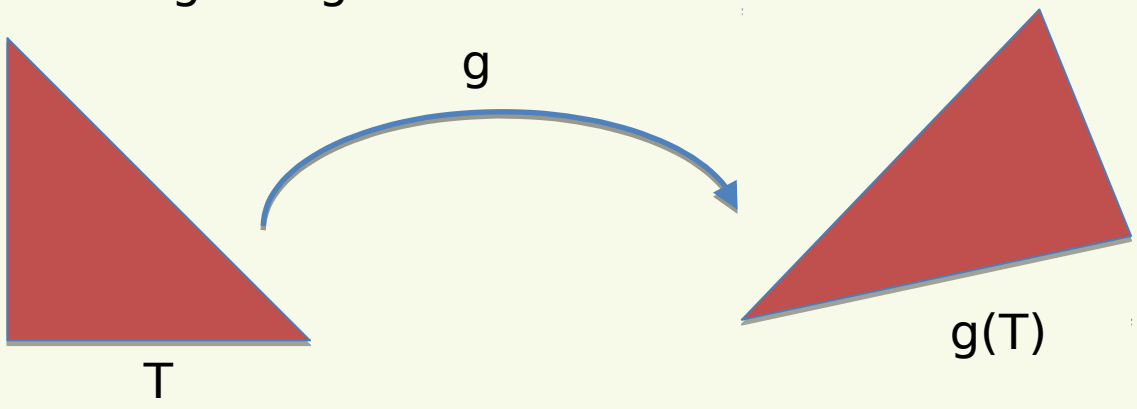
Concept and scientific content: Douglas N. Arnold (University of Minnesota) and Anders Logg (Chalmers University of Technology). Graphic design: Mattias Schläger. The production of this poster has been supported by Simula Research Laboratory and is partially based on work supported by the U.S. National Science Foundation under grant DMS-1115291. Findings do not necessarily represent the views of Simula or of the NSF.

femtable.org

Evaluating integrals over grids

How can we efficiently evaluate $\int_\Omega [\sigma(\mathbf{x})\nabla u(\mathbf{x})] \cdot \nabla v(\mathbf{x}) d\mathbf{x}$

over a given grid?



Reference Element

Physical Element

For simple flat triangles have: $g(x) = Jx + b$

Evaluating integrals over grids...

Integration by parts formula:

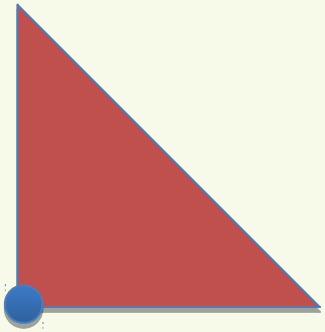
$$\int_{g(T)} f(\mathbf{y})d\mathbf{y} = \int_T f(g(\mathbf{x}))|\det(J(\mathbf{x}))|d\mathbf{x}$$

Jacobian (derivative) of g

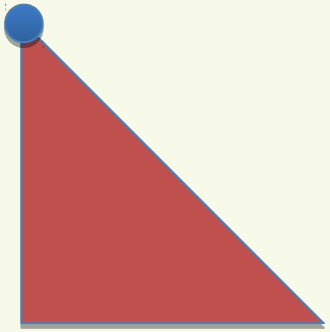
$$\int_\Omega [\sigma(\mathbf{y})\nabla u(\mathbf{y})]\cdot\nabla v(\mathbf{y})d\mathbf{y} = \sum_T \int_T \left[\sigma(g_T(\mathbf{x}))J_T^{-T}(\mathbf{x})\nabla\hat{u}_T(\mathbf{x})\right]\cdot J_T^{-T}\nabla\hat{u}_T(\mathbf{x})|\det(J_T(\mathbf{x}))|d\mathbf{x}$$

- Only integrals over triangles
- All quantities expressed in coordinates with respect to reference element
- Only need quadrature rule for reference element

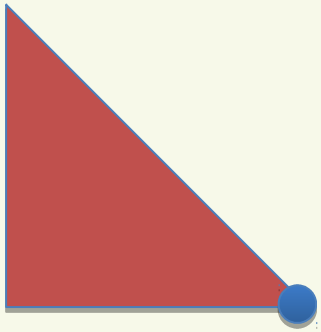
Linear shape functions



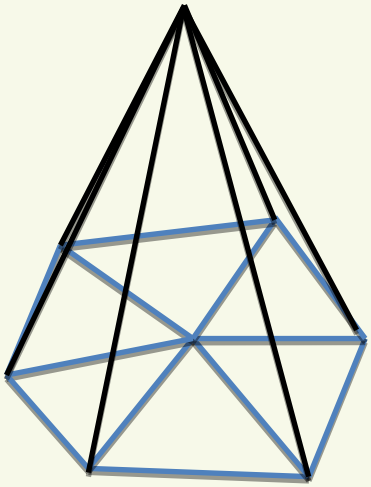
$u_0(x,y) = 1 - x - y$



$u_1(x,y) = y$



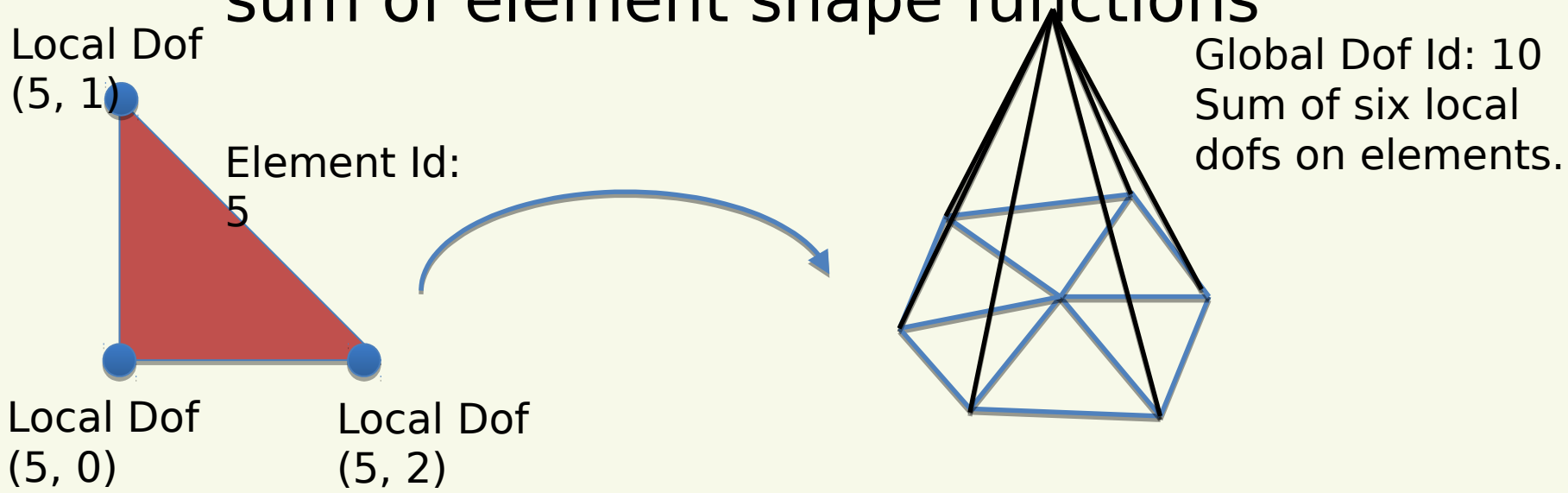
$u_2(x,y) = x$



- Each linear basis function is a linear combination of shape functions on each element.
- There are three different shape functions on the reference element.

Dof maps

- Two different types of degrees of freedom
  - Local element shape functions indexed by element id and local id
  - Global P1 basis function consisting of a sum of element shape functions



Dof Maps…

- Local-To-Global Map:
  - Element id -> A list of global indices that the local shape functions sum into
- Global-To-Local Map:
  - Global Id -> local shape functions. A list of local shape functions whose sum is the global ID.

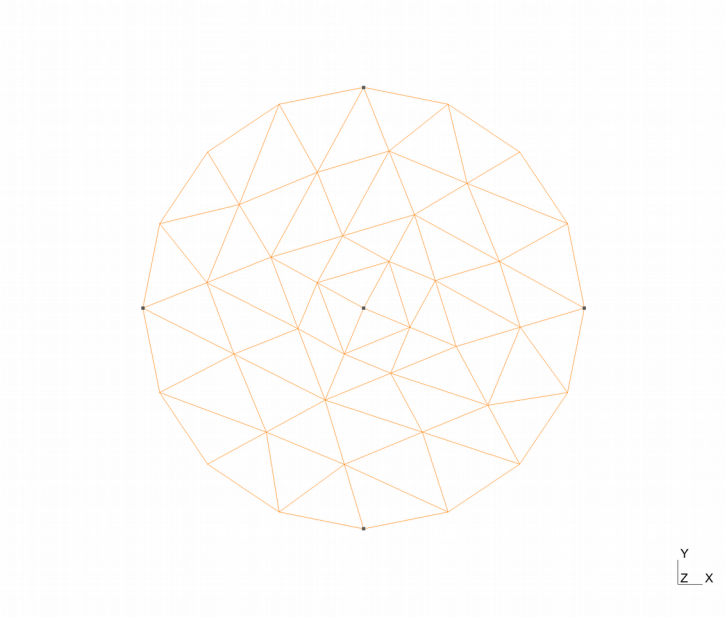
```
V = FunctionSpace(mesh,
“Lagrange”, 1)
dofmap = V.dofmap()

# Local to Global Dofs for element
0:
dofmap.cell_dofs(0)
```

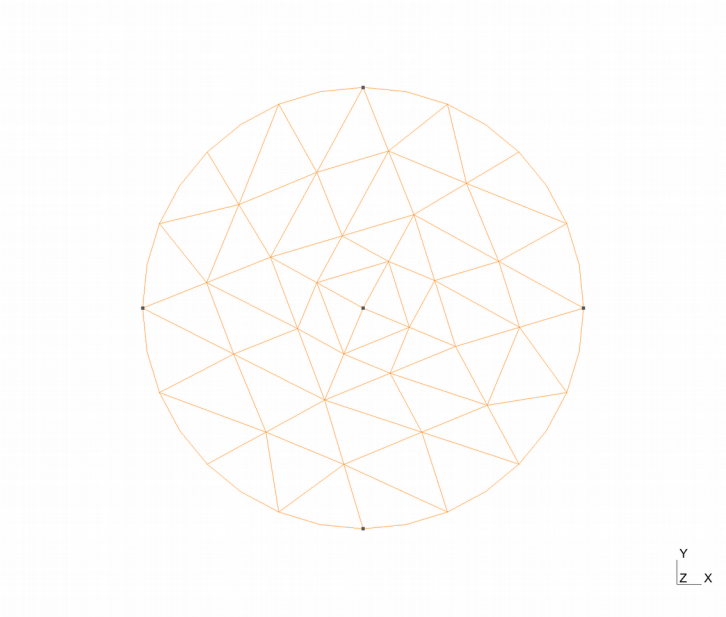
Assembling the global matrix

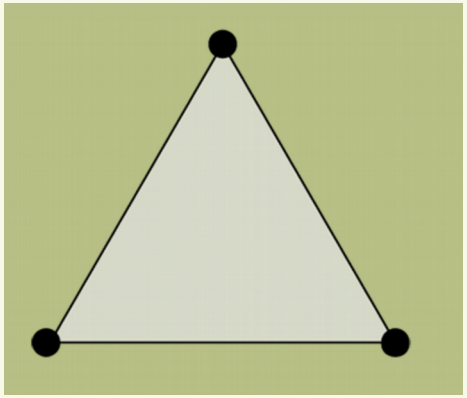
```
for cell in list of cells:
      for i in local_dofs:
            for j in local dofs:
                  Evaluate a(local_dof_j, local_dof_i)
                        A[local_to_global(i), local_to_global(j)] +=
a(local_dof_j, local_dof_i)
```

- In practice all weak forms on an element are assembled first and then sumed into the global matrix to cache geometric information from the reference map
- Difficult to implement efficiently using Python as fast for loops needed
- Use Numba/PyOpenCL or similar packages to speed up assembly or code directly in C/C++
- Have not yet talked about parallel assembly. Will do this later.

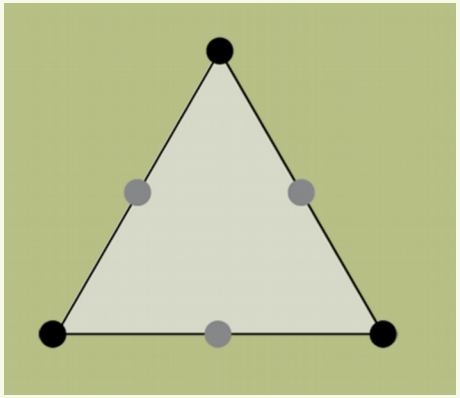
From low to high order





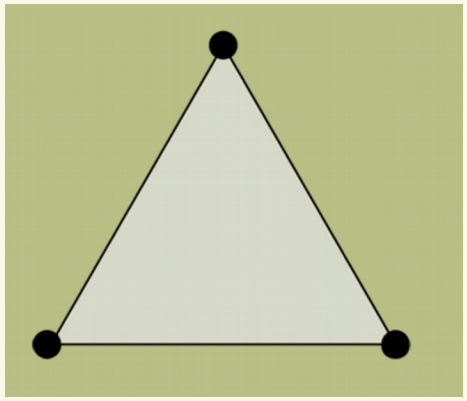


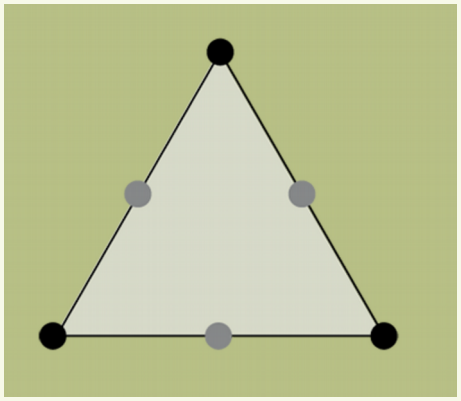
First order elements

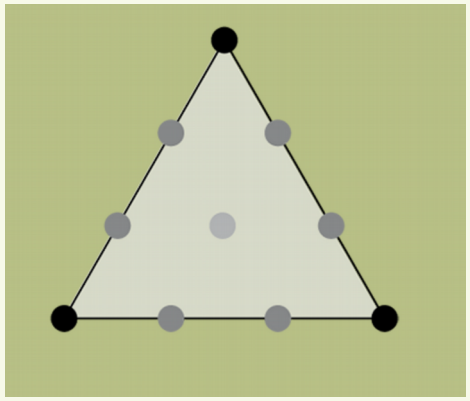


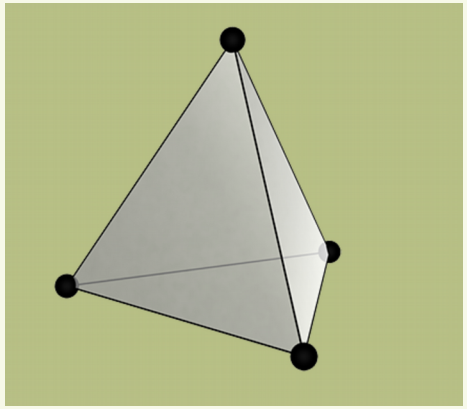
Second order elements

From low to high order...

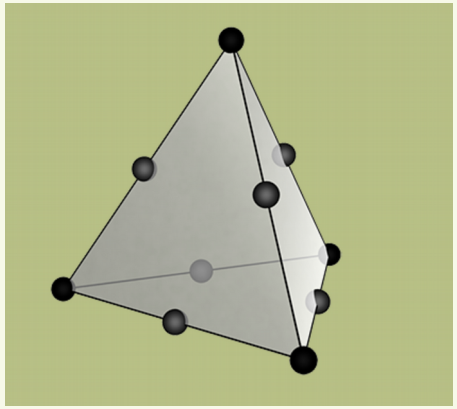




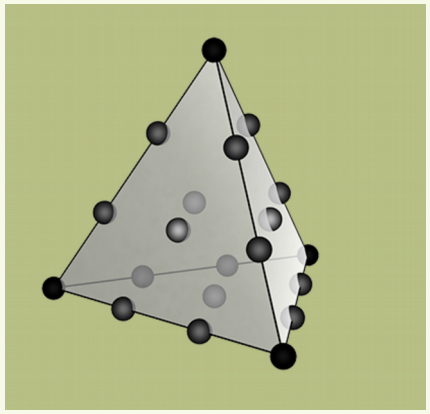




Linear



Quadrat
ic

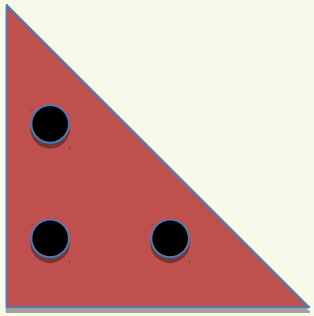


Cubi
c

Quadrature on the reference element

Approximate integrals over reference element by quadrature rule

$$\int_T f(\mathbf{x})d\mathbf{x} \approx \sum_{j=1}^{N} f(\mathbf{x}_j)\omega_j$$



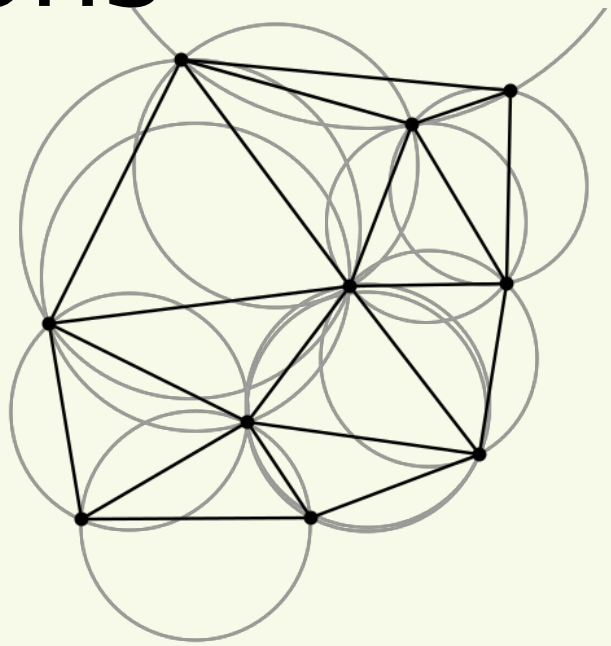
Symmetric Gauss Triangle quadrature rule:

- Find nodes and weights on the reference triangle such that as high polynomials as possible are integrated exactly
- Nodes and weights must be symmetric
- Detailed descriptions and node definitions for example contained in
- http://math2.uncc.edu/~shaodeng/TEACHING/math5172/Lectures/Lect_15.PDF

Triangulation in two dimensions

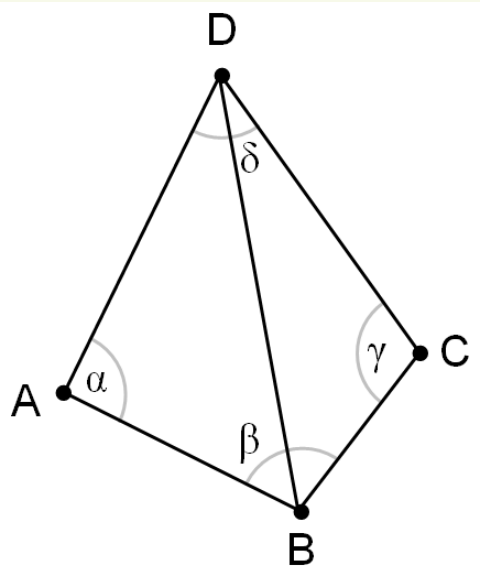
A set of triangles over a point set is called a triangulation if:

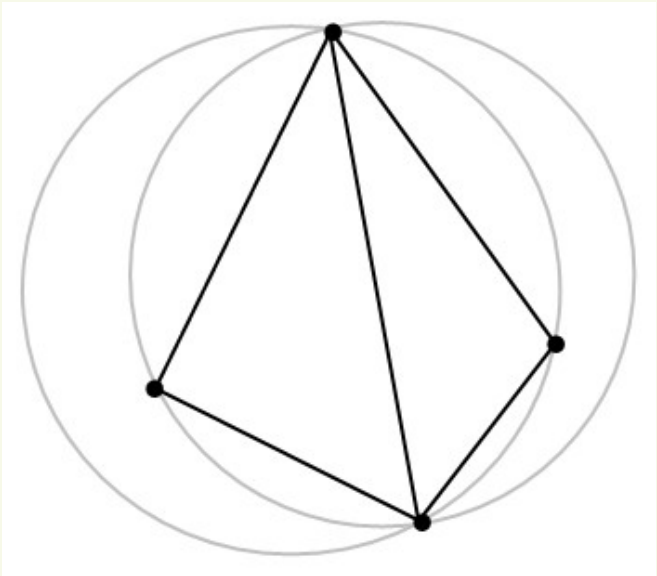
1. The union of the triangles equals the convex hull of the point set.
2. The union of all triangle vertices equals the point set.
3. Two distinct triangles share either a vertex, an edge or nothing. In particular, they do not overlap.

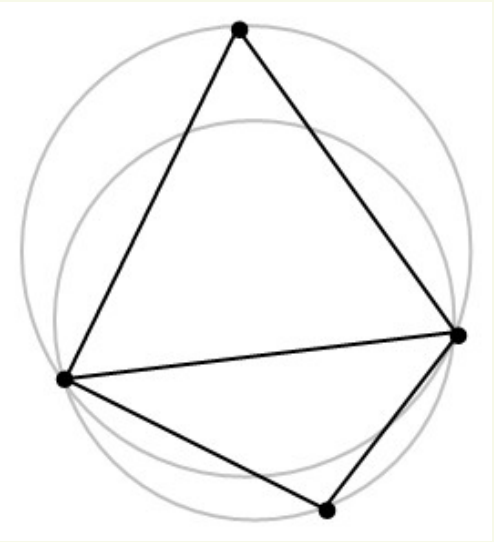


Delaunay Triangulation: A triangulation is called a Delaunay
triangulation if the circumcircle of every triangle is empty, that is no point is contained inside the circumcircle.

Delaunay Property

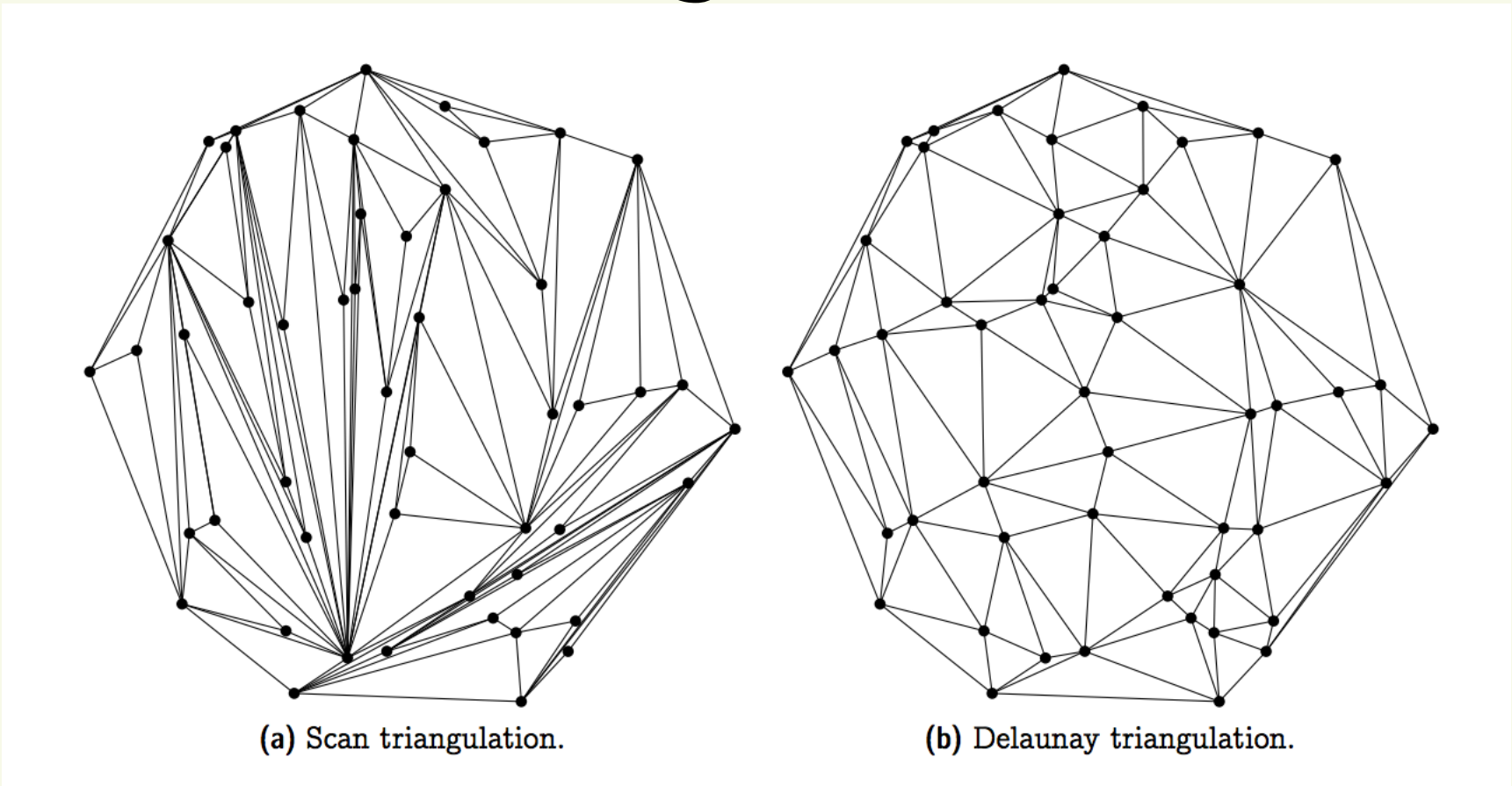






- An edge is called locally Delaunay if it has an open circumdisk containing no vertex of its adjacent triangles.
- The edge BD is not locally Delaunay.
- Flip BD into a new edge AC to become locally Delauny
- Flip Algorithm: Given any triangulation. Perform flips until all egdges are locally Delaunay. The resulting triangulation is Delaunay.

Quality of Delaunay triangulation



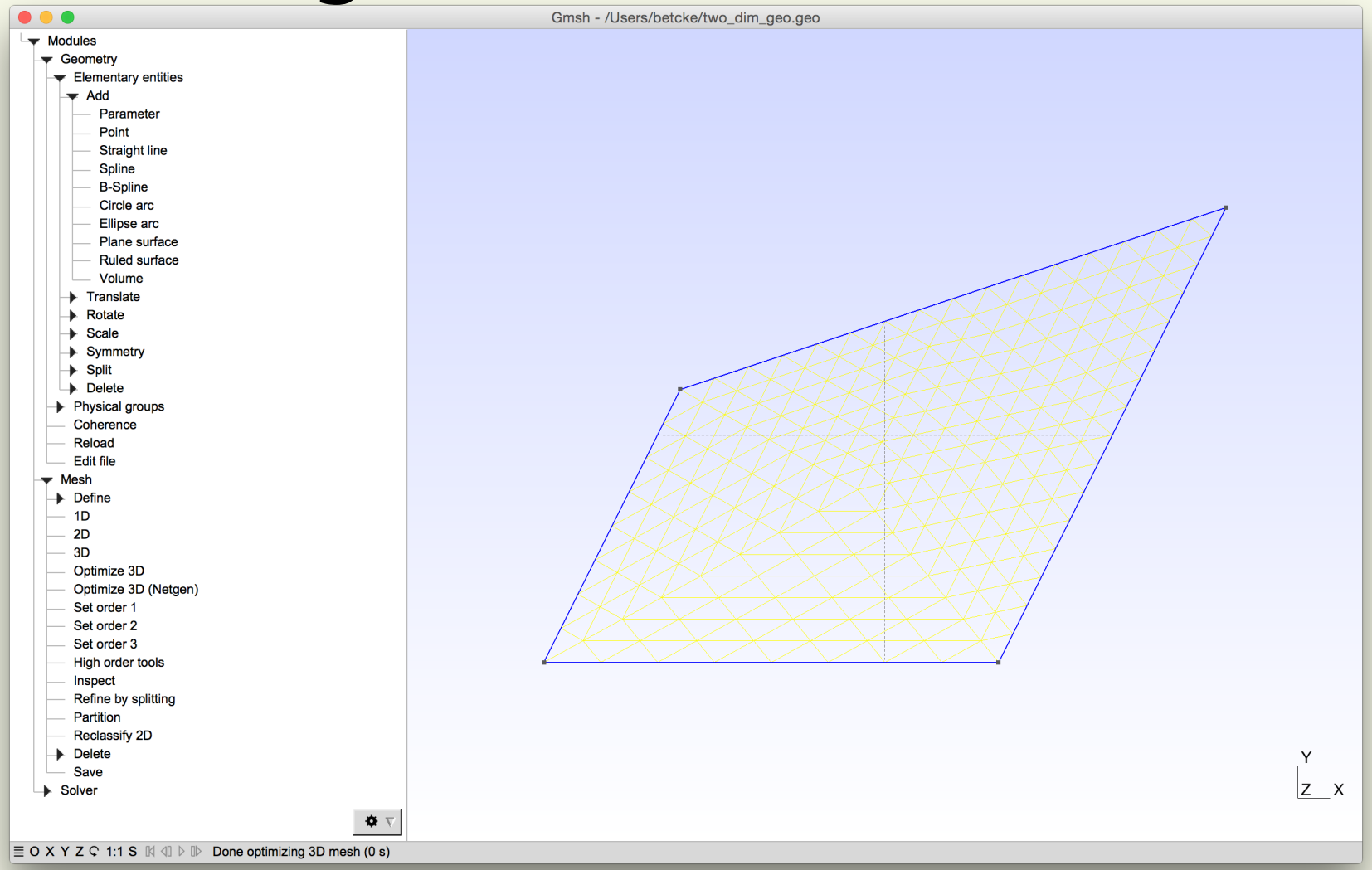
(a) Scan triangulation.

(b) Delaunay triangulation.

- Every Delaunay triangulation maximises the minimum angle over all possible triangulations.

http://www.ti.inf.ethz.ch/ew/Lehre/CG13/lecture/Chapter%206.pdf, p.3

Triangulation with Gmsh



http://gmsh.info

Triangulation in 3d

- Can extend Delaunay triangulation to tetrahedra (now working with spheres instead of circles)
- Good overview contained in:
- M. Bern, D. Eppstein, *Mesh generation and optimal triangulation*, 1995,
- https://www.ics.uci.edu/~eppstein/pubs/BerEpp-CEG-95.pdf
- Gmsh is great tool for triangulation in three dimensions.

Creation of regular grids

```
import vtk
import numpy as np

grid = vtk.vtkStructuredGrid()
points = vtk.vtkPoints()
writer = vtk.vtkStructuredGridWriter()

dims = [10, 20, 30]
data   = np.mgrid[0:1:dims[0] * 1j,
                  0:1:dims[1] * 1j,
                  0:1:dims[2] * 1j]

grid.SetDimensions(dims)

for k in range(dims[2]):
    for j in range(dims[1]):
        for i in range(dims[0]):
            points.InsertNextPoint(
                data[0, i, j, k], data[1, i, j, k], data[2, i, j, k])


grid.SetPoints(points)

writer.SetFileName('structured_grid.vtk')
writer.SetInputData(grid)
writer.Write()

```

- Can create regular grids directly with VTK
- Applications in finite differences and finite volume methods (see examples towards the end of term)

Mesh partitioning

- Large-Scale Grids do not fit onto a single compute node
- Need to distribute grids across multiple node
- Want to minimize the interfaces between partitions to avoid
nication

Consider equivalent graph partitioning problem. Each element corresponds to a vertex. If two elements share a common edge (or face in 3d) then the corresponding vertices in the graph are connected by an edge.

http://license.umn.edu/technologies/z09041_parmetis-mesh-graph-partitioning-algorithm

Graph partitioning problem

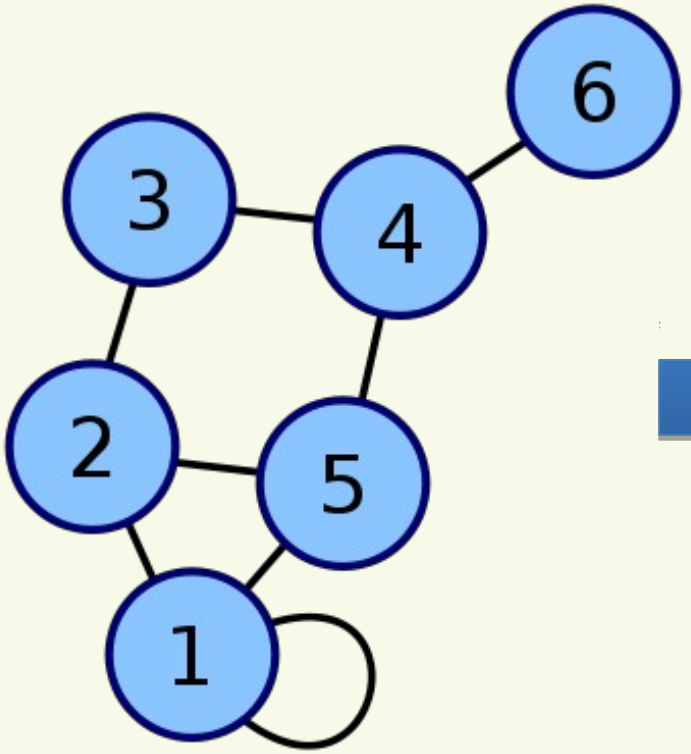
Definition (Graph): A graph is a tuple (V, E)
consisting of a set of vertices V and a set E of edges (v_i, v_j) between vertices.
Given a subset S of the vertices V.
Define

$$\partial S = \{(i,j) \in E : i \in S \wedge j \notin S\}$$

Minimum balanced cut:

$$\rho(S) = \min_S \frac{|\partial S|}{|S|}$$

Graph Adjacency matrix



Various conventions
how to deal with loops

$$\begin{pmatrix} 2 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

https://en.wikipedia.org/wiki/Adjacency_matrix

Graph Laplacian

Definition (Graph Laplacian):

$$L = D - A$$

$$D_{ij} = \begin{cases} \sum_\ell A_{i\ell}, & i = j \\ 0, & i \neq j \end{cases}$$

Incidence Matrix

Adjacency Matrix

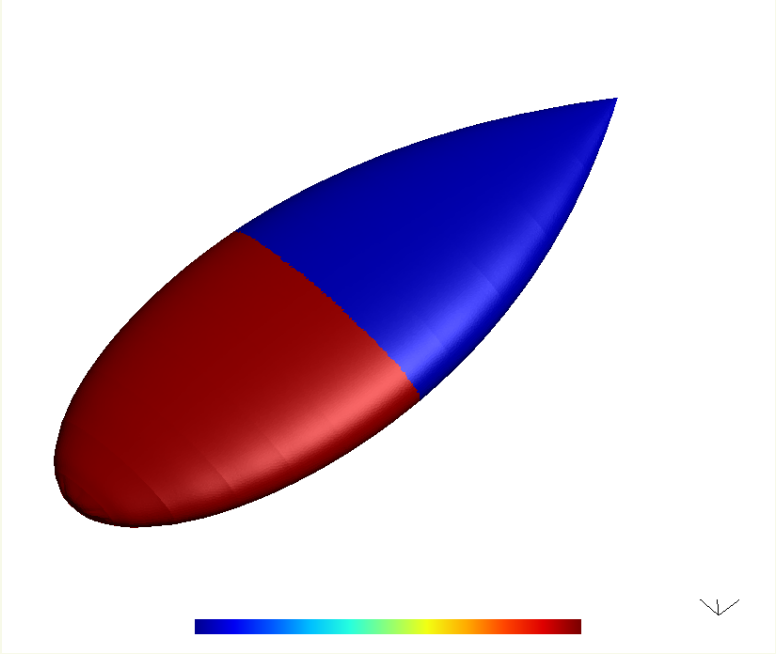
Properties:
L is symmetric and positve semidefinite with

$$0 = \lambda_1 < \lambda_2 \leq \lambda_3 \leq \dots$$

- if L is connected.
- The number of zero eigenvalues equals the number of connected components
- We can use the first nonzero eigenvalue and corresponding eigenvector for graph partitioning

Computing a 2-Partition

- Assemble the graph Laplacian L
- Compute the first nonzero eigenvalue of L and the corresponding eigenvector v.
- The positive entries of v correspond to one subdomain and the negative entries to the other subdomain.
- The eigenvector v is called Fiedler vector



Partitioning of a surface mesh arising from the Fiedler vector.

FEniCS Example

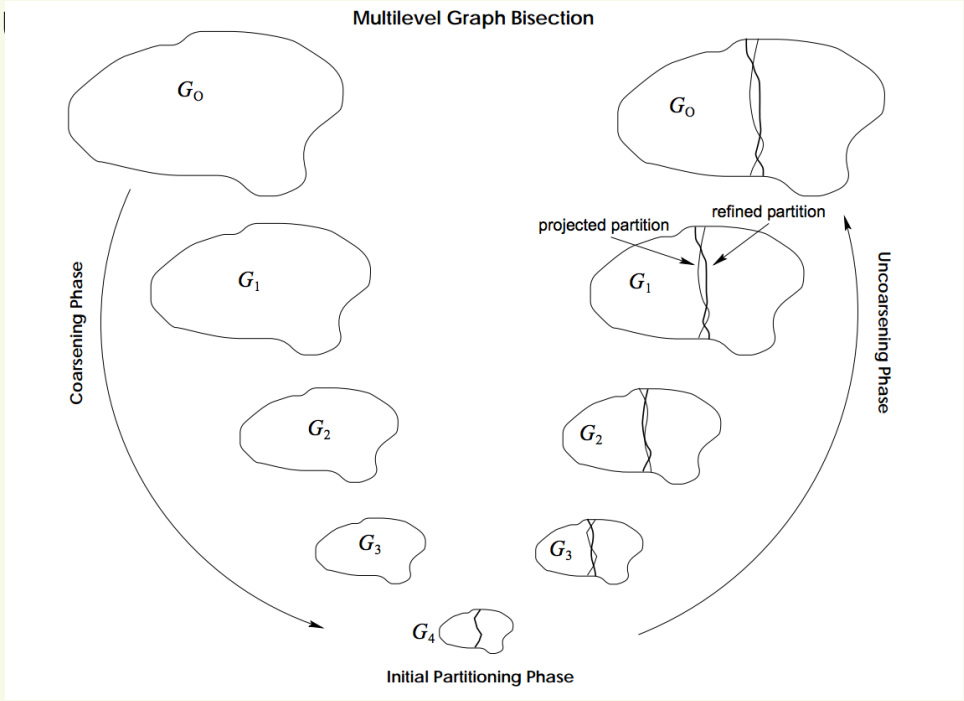
```python
import numpy as np
import scipy.sparse
import scipy.sparse.linalg

from dolfin import *
from matplotlib import pyplot as plt

# Create the mesh
mesh = UnitCubeMesh(10, 10, 10)

# Export the cells
cells = mesh.cells()

# Get the element to vertices map
indices = cells.flatten()
indptr = 4 * np.arange(cells.shape[0] + 1)
e2v = scipy.sparse.csr_matrix((np.ones(len(indices)), indices, indptr))

# Now compute the adjacency matrix
items = (e2v.dot(e2v.T)).todok().items()
items = [item for item in items if (2.5 < item[1] and item[1] < 3.5) ]
row_ind = [item[0][0] for item in items]
col_ind = [item[0][1] for item in items]
adjacency = scipy.sparse.coo_matrix((np.ones(len(row_ind)), (row_ind, col_ind))).tocsr()

# Compute the Graph Laplacian
e = np.ones(adjacency.shape[0])
d = adjacency.dot(e)
D = scipy.sparse.spdiags(d, 0, adjacency.shape[0], adjacency.shape[0])
L = D - adjacency

# Compute the Fiedler vector
lam, vec = scipy.sparse.linalg.eigs(L, 2, sigma=0)
fiedler = np.real(vec[:, 1])

# Visualize the resulting partitioning
V = FunctionSpace(mesh, "DP", 0)
u = Function(V)
u.vector()[:] = np.ascontiguousarray(np.sign(fiedler))
f = File('sol.pvd')
f << u
```

Heuristic explanation

- The graph Laplacian acts like a discrete Laplace operator
- The eigenfunction of the first nonzero eigenvalue has one nodal line that separates a domain into a positive and a negative part
- The smaller the first nonzero eigenvalue the less connected a graph
- For more than one partition consider the first k eigenvectors.

Multilevel Partitioning

- Computing the eigenvectors corresponding to a large graph is expensive.
- Idea: Use a multilevel technique by first coarsening the graph and computing the partitioning on a small s



G. Karypis and V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM J. Sci. Comp. 20, pp. 359—392 (1998)

Multilevel Partitioning...

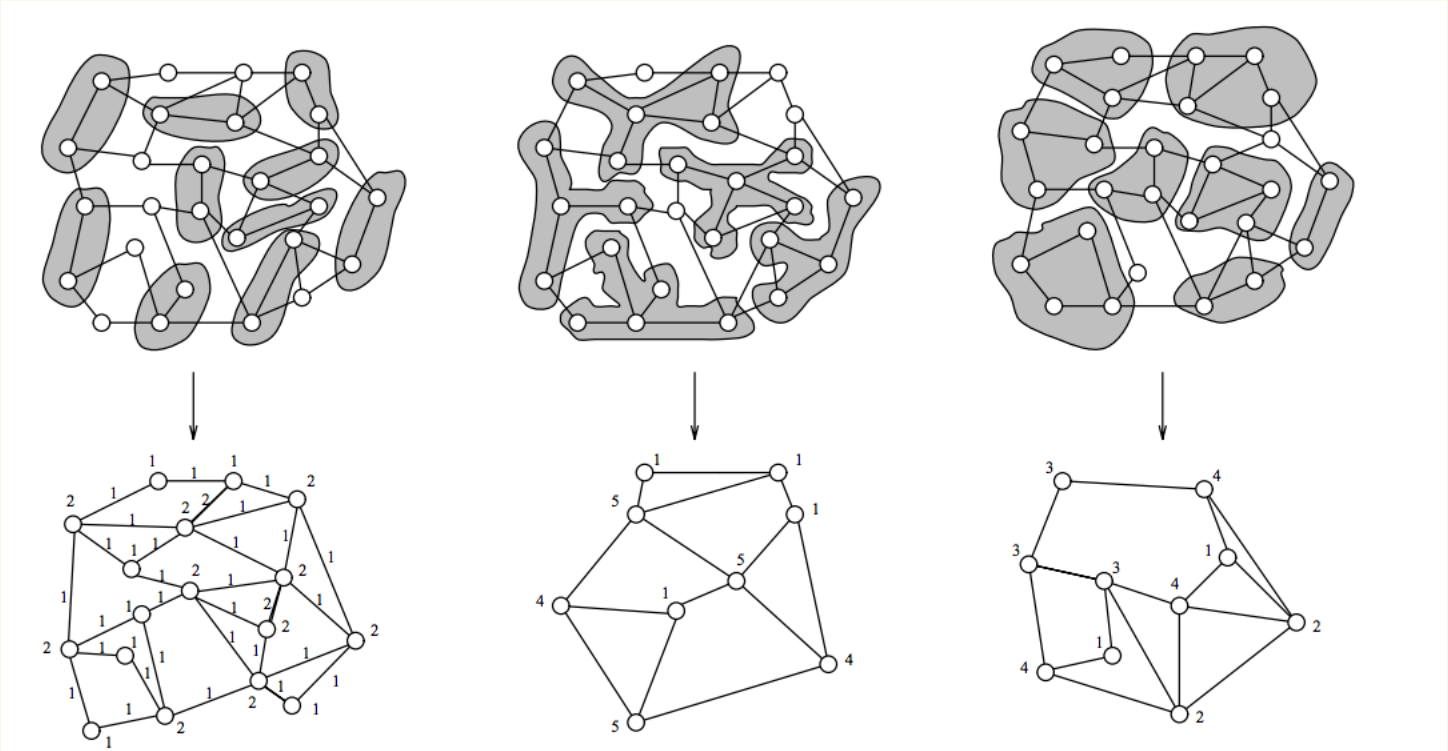


FIG. 2. *Different ways to coarsen a graph.*

G. Karypis and V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM J. Sci. Comp. 20, pp. 359—392 (1998)

Graph partitioning software

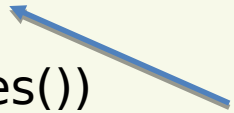
- METIS (http://glaros.dtc.umn.edu/gkhome/metis/metis/overview), pymetis available (https://mathema.tician.de/software/pymetis/)
-
- Scotch (https://www.labri.fr/perso/pelegrin/scotch/)
-
- Chaco (http://www3.cs.stonybrook.edu/~algorith/implement/chaco/implement.shtml)

Grid Management Software

- Dune-Grid (www.dune-project.org)
  - Extensive package for grid based algorithms
  - Supports various grid managers, large-scale parallel grids, refinement, etc.
- STK (https://trilinos.org/packages/stk/)
  - Grid manager of Trilinos
  - Large-Scale Parallel grids
- FEniCS (www.fenicsproject.org)
  - High-quality grid manager as part of the FEniCS distribution
  - Well integrated into FEniCS

Sparse Matrices

```
from dolfin import *
mesh = UnitCubeMesh(100, 100,
100)
print(mesh.num_vertices())
```



Creates mesh with over 1m vertices.

- Storage of the resulting matrix in dense format would take around 8 TeraBytes of RAM.
- However, almost all entries are zero.
- We should only store nonzero entries.
- Algebra to deal with matrices in sparse storage format required
- Software Support: Petsc, Trilinos (Epetra/Tpetra), Scipy
- In the following we will use Y. Saad, Iterative Methods for Sparse Linear Systems, 2nd Edition (freely available for download)

Sparse Formats

$$\begin{bmatrix} 1 & 0 & 0 & 2 & 0 \\ 3 & 4 & 0 & 5 & 0 \\ 6 & 0 & 7 & 8 & 9 \\ 0 & 0 & 10 & 11 & 0 \\ 0 & 0 & 0 & 0 & 12 \end{bmatrix}$$

1	2	3	4	5	6	7	8	9	10	11	12

data

0	0	1	1	1	2	2	2	2	3	3	4

Row indices

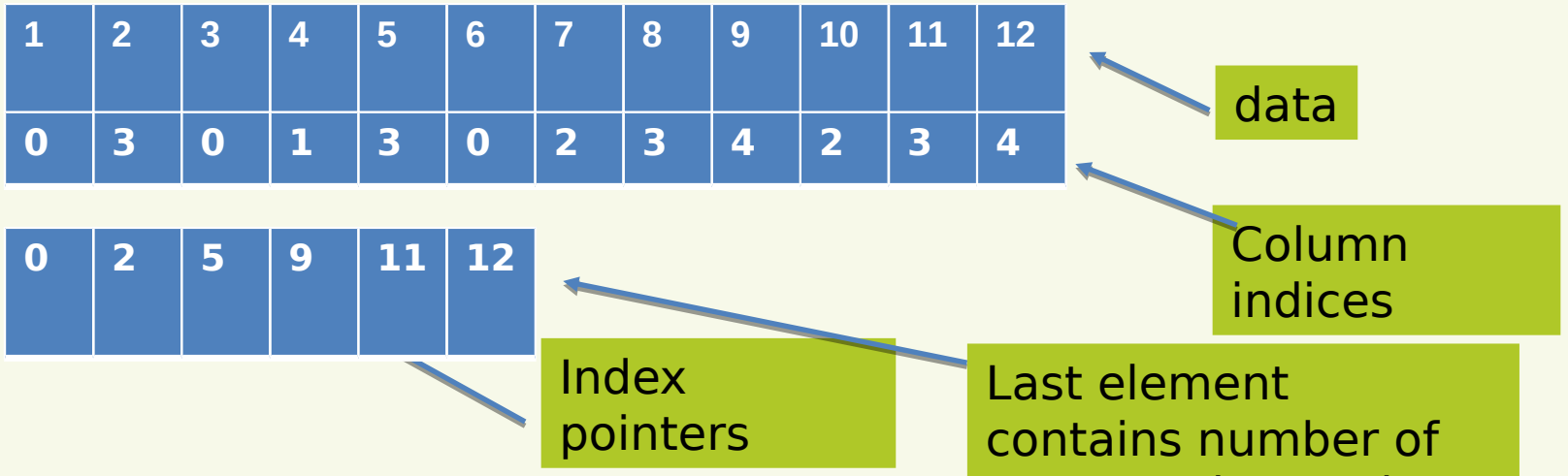
0	3	0	1	3	0	2	3	4	2	3	4

Column indices

- Simple coordinate list
- Easy to insert elements
- Not memory efficient

Compressed Sparse Row Format (CSR)

$$\begin{bmatrix} 1 & 0 & 0 & 2 & 0 \\ 3 & 4 & 0 & 5 & 0 \\ 6 & 0 & 7 & 8 & 9 \\ 0 & 0 & 10 & 11 & 0 \\ 0 & 0 & 0 & 0 & 12 \end{bmatrix}$$



- The nonzero col indices in the jth row are contained in JA[IA[j]: IA[j+1] - 1]
- Variant is compressed sparse column format (CSC)

Y. Saad, Iterative Methods for Sparse Linear Systems, 2nd Edition, p. 93

Matrix-Vector product

```
def matvec(x, data, indices, indptr):
    """Matvec of a matrix in CSR format with
vector x"""
    y = numpy.zeros(x.shape[0],
dtype='float64')
    n = len(indptr) - 1

    for i in range(n):
        y[i] = data[ indptr[i]: indptr[i+1] ].dot(
            x[indices[indptr[i]: indptr[i+1]]])
```

- Very easy to parallelize by distributing matrix rows among the processes
- Efficient vectorized operations

LU Decomposition

Want to solve

$$Ax = b$$

for a general n x n matrix A and an arbitrary right-hand side vector b.

Idea: Represent A as a product of two simpler matrices.

$$A = LU$$

and solve

$$LUx = b$$

instead by first solving $Ly = b$ and then $Ux = y$

.

How do we get L and U?

LU Decomposition...

Reviewing Gaussian Elimination...

Consider the system $Ax = b$. We write it in the form

$$\left[\begin{array}{cccc|c} a_{11} & a_{12} & \dots & a_{1n} & b_1 \\ a_{21} & a_{22} & \dots & a_{2n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} & b_n \end{array}\right]$$

Now subtract $\ell_{21} := a_{21}/a_{11}$ times the first row from the second row to obtain

$$\left[\begin{array}{cccc|c} a_{11} & a_{12} & \dots & a_{1n} & b_1 \\ 0 & a_{22} - \ell_{21}a_{12} & \dots & a_{2n} - \ell_{21}a_{1n} & b_2 - \ell_{21}b_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} & b_n \end{array}\right]$$

LU Decomposition…

Continuing this way we obtain an upper triangular matrix U of the form

$$\left[\begin{array}{cccc|c} * & * & \dots & * & b_1 \\ 0 & * & \dots & * & * \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ * & * & \dots & * & * \end{array}\right].$$

The trick is to represent the subtraction of a multiple of the ith row from the jth row as a matrix vector product of the form

$$L_{ij}A := \left[I - \ell_{ij} e_i e_j^T\right] A$$

LU Decomposition...

We obtain therefore an upper triangular matrix by the sequence of operations

$$\begin{aligned} U &:= L_{n,n-1} \cdots \\ &\quad \ldots L_{42} L_{32} L_{n1} \ldots L_{31} L_{21} A \end{aligned}$$

The matrix L is therefore obtained as

$$\begin{aligned} L &:= (L_{n,n-1} \ldots L_{42} L_{32} L_{n1} \ldots L_{31} L_{21})^{-1} \\ &= L_{21}^{-1} L_{31}^{-1} \ldots L_{n1}^{-1} L_{32}^{-1} L_{42}^{-1} \ldots L_{n,n-1}^{-1}. \end{aligned}$$

A direct computation shows that

$$\begin{aligned} L &= L_{21}^{-1} L_{31}^{-1} \ldots L_{n1}^{-1} L_{32}^{-1} L_{42}^{-1} \ldots L_{n,n-1}^{-1} \\ &= (I + \ell_{21} e_2 e_1^T)(I + \ell_{31} e_3 e_1^T) \ldots \\ &= \begin{bmatrix} 1 & 0 & \ldots & 0 \\ \ell_{21} & 1 & \ldots & 0 \\ & & \ddots & \vdots \\ \ldots & \ldots & & \\ \ell_{n1} & \ell_{n2} & \ldots & 1 \end{bmatrix} \end{aligned}$$

LU Decomposition…

We can now solve with L using simple forward substitution

$$Ly = \begin{bmatrix} 1 & 0 & \dots & 0 \\ \ell_{21} & 1 & \dots & 0 \\ & & \ddots & \vdots \\ \dots & \dots & & 1 \\ \ell_{n1} & \ell_{n2} & \dots & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

$$y_i = b_i - \sum_{j=1}^{i-1} \ell_{ij} y_j;\ i = 1, \dots, n$$

Correspondingly can solve for U by using backward substitution.

Remarks

- LU decomposition has a complexity of O(n^3) for general matrices of dimension n
- In practice, we use use pivoting strategy to reorder the rows during the computation to improve the stability of the LU decomposition.
- LU has a wonderful property. For certain sparse matrices its running time reduces from O(n^3) to O(n).

Cholesky Decomposition

A matrix A is called symmetric positive definite (SPD) if A is symmetric and

$$x^T Ax > 0$$

- for all nonzero x.
- SPD matrices are very common. Essentially, they can be interpreted as energy functionals. Solving linear systems with SPD matrices amounts to finding minimum energy states of a system.
- For SPD matrices a special variant of Gaussian Elimination is possible that adheres to the structure of SPD matrices, namely we can decompose A into

  $$A = C^T C$$

  with C upper triangular. This can be achieved in half the operations as required for the standard LU decomposition.

LU for sparse matrices

A matrix is called banded if

$$a_{ij} \neq 0 \text{ if and only if } i - m_L \leq j \leq i + m_U$$

The number

$$m_L + m_U + 1$$

is called the bandwidth of A.

Thm: The lower bandwidth of L in the LU decomposition A = LU is m_L and the upper bandwidth in U is m_U.

Thm: Let A be an n x n matrix. If the bandwidth is constant O(1) for growing n. Then the complexity of the LU decomposition is O(n).

Reorderings

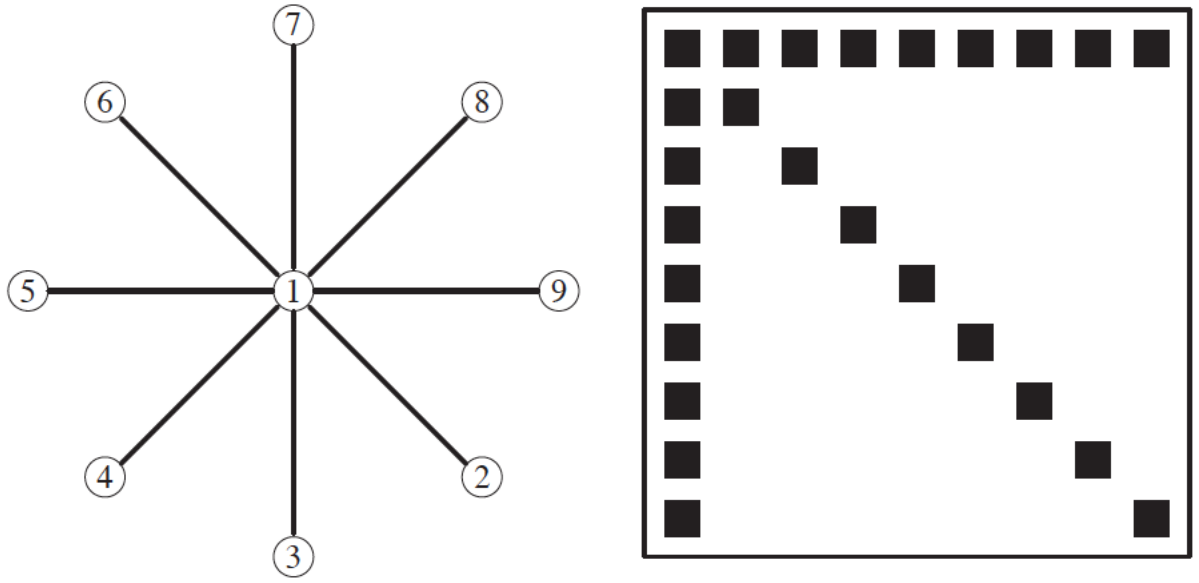


Figure 3.4: Pattern of a 9×9 arrow matrix and its adjacency graph.

Performing Gaussian Elimination fills up the matrix.

Reorderings...

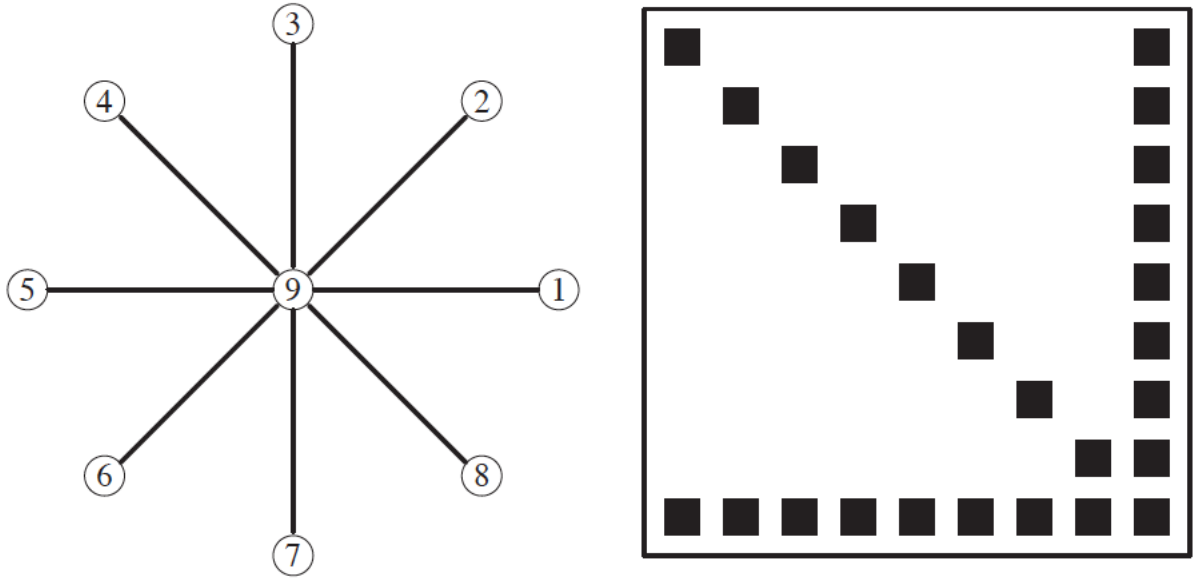


Figure 3.5: Adjacency graph and matrix obtained from above figure after permuting the nodes in reverse order.

A simple reordering produces no fill-in.

Permutations

Definition: Let A be a matrix and $\pi = \{i_1, i_2, \ldots, i_n\}$ a given permutation of the set $\{1, 2, \ldots, n\}$. Then

$$A_{\pi,*} = \{a_{\pi(i),j}\}_{i=1,\ldots,n;j=1,\ldots,m}$$
$$A_{*,\pi} = \{a_{i,\pi(j)}\}_{i=1,\ldots,n;j=1,\ldots,m}$$

are called row, respectively, column perturbation of A.

$$P_\pi = I_{\pi,*},\ P_\pi^T = I_{*,\pi} \qquad A_{\pi,*} = I_{\pi,*}A = P_\pi A \qquad A_{*,\pi} = AI_{*,\pi} = AP_\pi^T$$

Permutations

Example 3.1. Consider, for example, the linear system $Ax = b$ where

$$A = \begin{pmatrix} a_{11} & 0 & a_{13} & 0 \\ 0 & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & 0 \\ 0 & a_{42} & 0 & a_{44} \end{pmatrix}$$

and $\pi = \{1, 3, 2, 4\}$, then the (column-) permuted linear system is

$$\begin{pmatrix} a_{11} & a_{13} & 0 & 0 \\ 0 & a_{23} & a_{22} & a_{24} \\ a_{31} & a_{33} & a_{32} & 0 \\ 0 & 0 & a_{42} & a_{44} \end{pmatrix} \begin{pmatrix} x_1 \\ x_3 \\ x_2 \\ x_4 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix}.$$

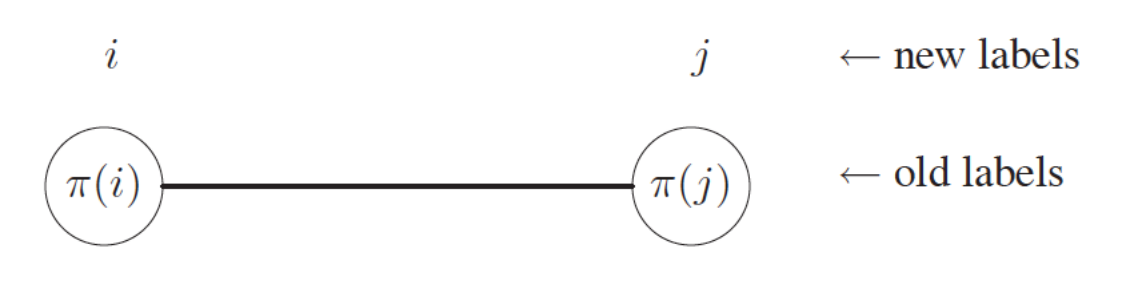
Symmetric permutation

A symmetric permutation of a matrix A is defined as

$$A' = A_{\pi,\pi}$$

We have $A'_{ij} = a_{\pi(i),\pi(j)}$.

Symmetric permutations do not change the adjacency graph. They just relabel the vertices.



Reducing bandwidth by Level-Set Reordering

- Reorder a graph by traversing level-sets
- A level set is defined recursively as the set of all neighbors of the previous level-set
- We start with a single node

Reordering by Breadth-First Search

1. *Initialize* $S = \{v\}, seen = 1, \pi(seen) = v$; *Mark* v;
2. *While* $seen < n$ *Do*
3. $S_{new} = \emptyset$;
4. *For each node* v *in* S *do*
5. *For each unmarked* w *in adj*(v) *do*
6. *Add* w *to* S_{new};
7. *Mark* w;
8. $\pi(++seen) = w$;
9. *EndDo*
10. $S := S_{new}$
11. *EndDo*
12. *EndWhile*

In what order should be traverse the adjacent vertices ?

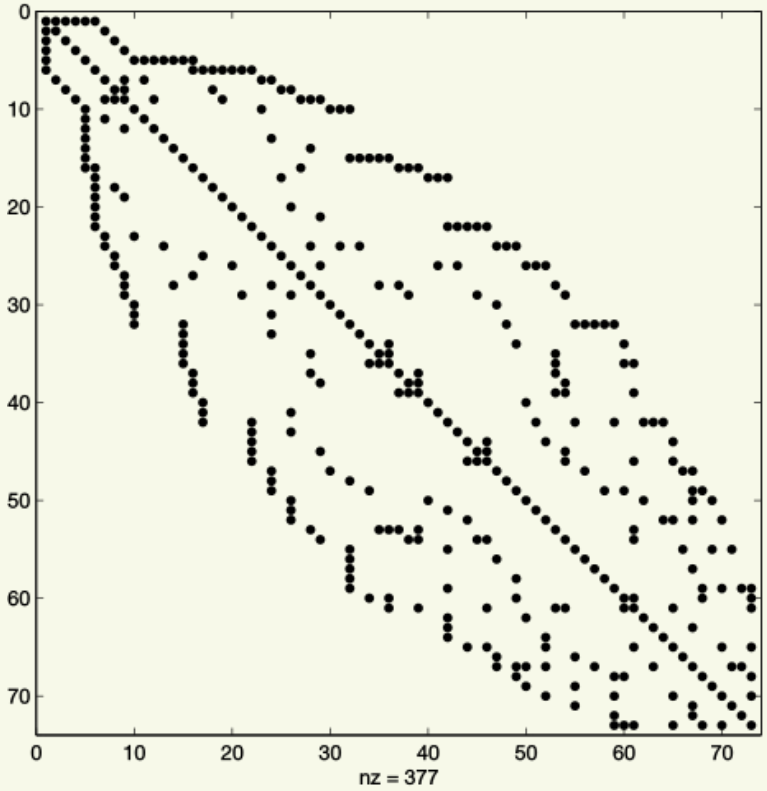
Cuthill-McKee Algorithm

0. *Find an intial node v for the traversal*
1. *Initialize* $S = \{v\}, seen = 1, \pi(seen) = v$; *Mark* v;
2. *While* $seen < n$ *Do*
3. $S_{new} = \emptyset$;
4. *For each node* v *Do:*
5. $\pi(++seen) = v$;
6. *For each unmarked* w *in adj*(v), *going from lowest to highest degree Do:*
7. *Add* w *to* S_{new};
8. *Mark* w;
9. *EndDo*
10. $S := S_{new}$
11. *EndDo*
12. *EndWhile*

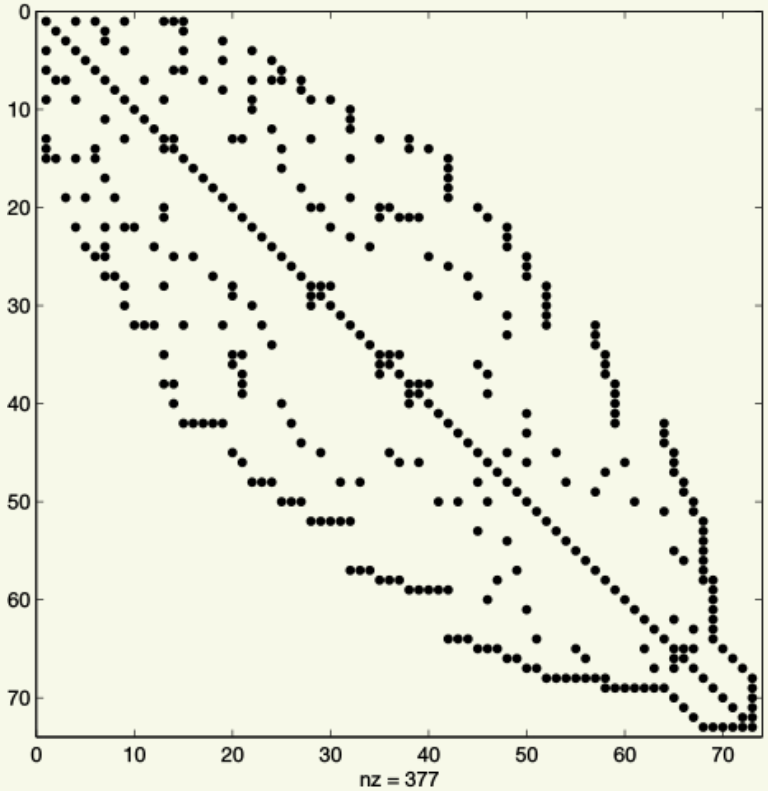
Traverse from lowest to highest degree

Reverse Cuthill-Mckee:
Use Cuthill-McKee and then reverse the ordering. Often reduced the fill-in better than Cuthill-McKee.

Cuthill-Mckee and Reverse Cuthill McKee



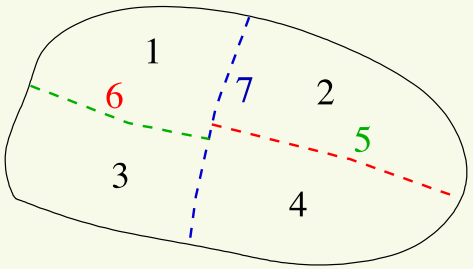
Cuthill-McKee

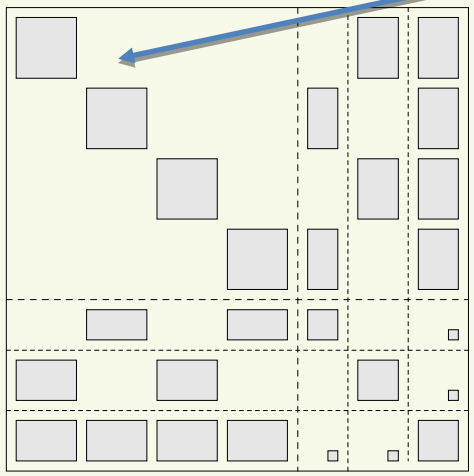


Reverse Cuthill-McKee

Nested Dissection

- Reordering for SPD matrices that allows the parallelization of the Cholesky decomposition.
- Idea: Split a graph into two separated subgraphs and a separator layer that connects the subgraphs. Recursively continue for each subgraph.
- Label nodes in the separator after nodes in the subgraphs.





Can perform Cholesky decomposition independently in each diagonal block

Shopping for matrices in the Matrix Market

- http://math.nist.gov/MatrixMarket/
- Huge collection of sparse matrices with search features for a wide range of characteristics
- Ideal to test your own algorithms
- Scipy.io can directly read matrix market format. Routines for Matlab/C/Fortran can be downloaded

Sparse matrix packages

- Matlab has excellent built-in sparse matrix format, using UMFPack as sparse direct solver
- Scipy has decent sparse-matrix support (good enough for many things, not as extensive as Matlab). Default solver is SuperLU
- Intel MKL can deal with sparse matrices and provides a built-in version of Pardiso, a non-free high-performance solver
- PETSc/Trilinos have outstanding parallel sparse matrix support

Scipy sparse matrix formats

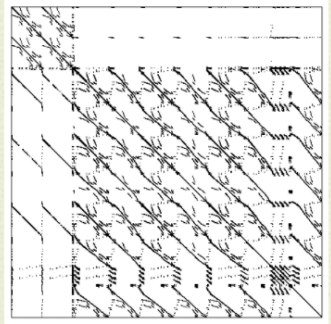
- bsr_matrix: A blockwise sparse format for sparse matrices with dense subblocks. These often appear in FEM with high-order basis functions
- coo_matrix: Sparse matrix in triplet format (i, j, data). Multiple entries are summed up when converting to CSR format, making it ideal for FEM assembly
- csc_matrix: Compressed sparse column format
- csr_matrix: Compresse sparse row format
- dia_matrix: Ideal for matrices where data is given along diagonals
- dok_matrix: Dictionary of keys. Allows O(1) assignment operations and then conversion to coo format (note: Does not allow creation of duplicate entries).
- lil_matrix: Linked-list format for row-wise construction.

Sparse Direct Solvers - Mumps

- MPI Parallel Multifrontal Solver
- Widely used in applications
- Open-Source License
- http://mumps.enseeiht.fr/

MUMPS : A PARALLEL SPARSE DIRECT SOLVER

MUMPS Main Features



- Solution of large linear systems with
  symmetric positive definite matrices
  general symmetric matrices
  general unsymmetric matrices
- Version for **complex arithmetic**
- **Parallel** factorization and solve phases
  (uniprocessor version also available)
- **Iterative refinement** and **backward error analysis**
- Various matrix input formats
  assembled, distributed, elemental format
- **Partial factorization** and **Schur complement matrix** (centralized or 2D block-cyclic)
- **Interfaces to MUMPS**: Fortran, C, Matlab and Scilab
- Several **reorderings** interfaced: AMD, QMAD, AMF, PORD, METIS, PARMETIS, SCOTCH, PT-SCOTCH

Sparse Direct Solvers - SuiteSparse

- Multi-Threaded parallel multifrontal solvers
- Integrated into Matlab
- Widely used

SuiteSparse is a suite of sparse matrix algorithms, including:

- UMFPACK: multifrontal LU factorization. Appears as LU and x=A\b in MATLAB.
- CHOLMOD: supernodal Cholesky. Appears as CHOL and x=A\b in MATLAB. Used in Google Ceres. Now with CUDA acceleration, in collaboration with NVIDIA.
- SPQR: multifrontal QR. Appears as QR and x=A\b in MATLAB. CUDA acceleration just released (October 10, 2014, SuiteSparse 4.4.0), and submitted to ACM Trans. Math. Software
- KLU and BTF: sparse LU factorization, well-suited for circuit simulation. Appears in Xyce by Sandia, and many commercial circuit simulation packages.
- Ordering methods (AMD, CAMD, COLAMD, and CCOLAMD). AMD and COLAMD appear in MATLAB.
- CSparse and CXSparse: a concise sparse Cholesky factorization package for my SIAM book.
- UFget: a MATLAB interface for the UF Sparse Matrix Collection
- spqr_rank: a MATLAB package for reliable sparse rank detection, null set bases, pseudoinverse solutions, and basic solutions.
- Factorize: an object-oriented solver for MATLAB (a reusable backslash).
- SSMULT and SFMULT: sparse matrix multiplication. Appears as C=A*B in MATLAB.
- ... and many other packages.

Sparse direct solvers - SuperLU

- General purpose sparse direct multi-threaded and MPI enabled solver
- Single-Core version default sparse solver in Scipy

SuperLU is a general purpose library for the direct solution of large, sparse, nonsymmetric systems of linear equations. The library is written in C and is callable from either C or Fortran program. It uses MPI, OpenMP and CUDA to support various forms of parallelism. It supports both real and complex datatypes, both single and double precision, and 64-bit integer indexing. The library routines performs an LU decomposition with partial pivoting and triangular system solves through forward and back substitution. The LU factorization routines can handle non-square matrices but the triangular solves are performed only for square matrices. The matrix columns may be preordered (before factorization) either through library or user supplied routines. This preordering for sparsity is completely separate from the factorization. Working precision iterative refinement subroutines are provided for improved backward stability. Routines are also provided to equilibrate the system, estimate the condition number, calculate the relative backward error, and estimate error bounds for the refined solutions.
Serial SuperLU package also contains ILU routines, using numerical threshold-based dropping, with partial pivoting (ILUTP).

SuperLU package comes in three different flavors:

- SuperLU for sequential machines (code documentation in HTML)
- SuperLU_MT for shared memory parallel machines
- SuperLU_DIST for distributed memory (code documentation in HTML)
  The target machines for SuperLU_DIST are the highly parallel distributed memory hybrid systems. The numerical factorization routines are already implemented for hybrid systems with multiple GPUs. Further work will be needed to implement the other phases of the algorithms on the hybrid systems and to enhance strong scaling to extreme scale.

Sparse direct solvers - Pardiso

- Highly efficient sparse direct multithreaded and MPI solver
- Two versions:
  - MKL Pardiso (part of the Intel MKL Library)
  - Pardiso (http://www.pardiso-project.org)
- Pardiso is not free software
- Academic licenses from Pardiso
- available for free
- MKL Pardiso less advanced than
- Pardiso

Iterative Solvers

- Sparse direct solvers become infeasibly expensive for big problems with complicated connectivity that creates a large amount of fill-in for Gaussian elimination
- Idea: Develop robust iterative solvers that compute not the exact solution, but an approximate solution to a certain given accuracy.
- Modern developments: Combine sparse direct and iterative solvers (more later)

Splitting Methods

- Splitting Methods are among the earliest and most simple iterative methods.
- By themselves not efficient, but very good as preconditioners or as solvers within a multigrid framework.

We want to solve

$$Ax = b$$

Idea: Split A into two matrices M and N such that

$$A = M - N$$

$$x = M^{-1}b + M^{-1}Nx$$

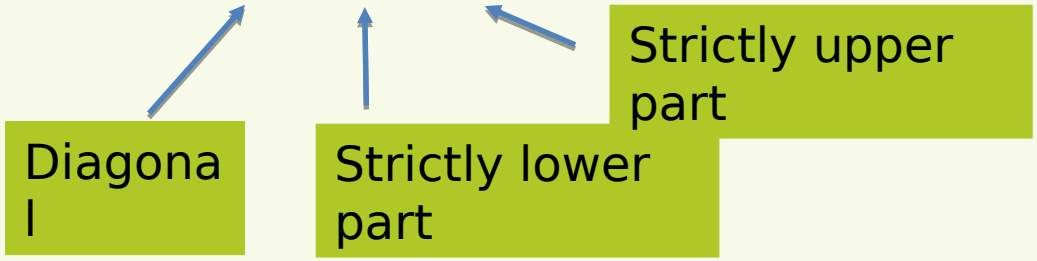
We consider this equation as the fixed point iteration

$$x_{k+1} = M^{-1}b + M^{-1}Nx_k$$

Splitting Methods...

Let

$$A = D - E - F$$



Jacobi:

$$x_{k+1} = D^{-1}(E + F)x_k + D^{-1}b$$

Gauss-Seidel:

$$x_{k+1} = (D - E)^{-1}Fx_k + (D - E)^{-1}b$$

SOR:

$$x_{k+1} = (D - \omega E)^{-1}\left[\omega F + (1 - \omega)D\right]x_k + \omega(D - \omega E)^{-1}b$$

SOR Splitting;

$$\omega A = (D - \omega E) - (\omega F + (1 - \omega)D)$$

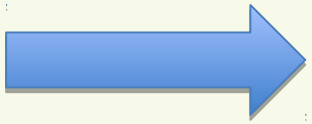
Convergence of splitting methods

General iteration scheme:

$$x_{k+1} = Gx_k + f$$

Exact solution satisfies

$$x^* = Gx^* + f$$



$$x_{k+1} - x^* = G(x_k - x^*) = G^k(x_0 - x^*)$$

Take norms to obtain

$$\|x_{k+1} - x^*\| \leq \|G\|^k \|(x_0 - x^*)\|$$

Thm: The scheme converges if $\|G\| < 1$

Since we did not specify the norm a more intrinsic argumentation shows that it is enough for the largest eigenvalue of G to be of magnitude smaller than 1.

Convergence of splitting methods...

A splitting method is a special case of this iteration scheme with $G = M^{-1}N$.

A matrix A is **strictly** diagonally dominant if

$$|a_{jj}| > \sum_{i=1, i\neq j}^{n} |a_{ij}|, \ j = 1, \ldots, n.$$

It is called **irreducibly diagonally dominant** if A is irreducible and

$$|a_{jj}| \geq \sum_{i=1, i\neq j}^{i=1} |a_{ij}|, \ j = 1, \ldots, n$$

with strict inequality for at least one j.

Thm: If A is strictly diagonally dominant or irreducibly diagonallly dominant
Jacobi and Gauss-Seidel converge for any given starting vector.

Summary

- Splitting methods not competitive by themselves.
- However, they can be used as preconditioners (more later)…
- …or as smoothers in multigrid methods (again, later).
- Even if we don’t use splitting methods directly any more they still play an important role.

Krylov Subspace Methods

Motivation: Consider the simple iteration

$$x_{k+1} = (I - A)x_k + b$$

If the iteration converges, it converges to the solution of $Ax = b$

.

We have

$$\begin{aligned} x_1 &= b \\ x_2 &= (I - A)b + b = 2b - Ab \\ x_3 &= (I - A)x_2 + b = 3b - 3Ab + A^2 b \\ x_k &= P_{k-1}(A)b \end{aligned}$$

$P_{k-1}(A) = \alpha_0 b + \alpha_1 Ab + \cdots + \alpha_{k-1} A^{k-1} b$

is a polynomial in A with coefficients $\alpha_0, \ldots, \alpha_{k-1}$.
Ideally, we would choose the coefficients such that
$P_{k-1}(A)b \approx A^{-1}b$ in some way.

Krylov Subspace Methods

The kth **Krylov subspace** $\mathcal{K}_k(A, b)$ is defined as

$$\mathcal{K}_k(A, b) = \text{span}\{b, Ab, \ldots, A^{k-1}b\}.$$

By

$$K_k = \begin{bmatrix} b & Ab & A^2b & \ldots & A^{k-1}b \end{bmatrix}$$

we denote the corresponding **Krylov matrix**.

How do we choose $x_k = P_{k-1}(A)b$ from the k-th Krylov subspace? Define $r_k = b - Ax_k$ to be the corresponding residual. We define the following Krylov subspace methods by the condition on the residual.

- Conjugate Gradients: The residual must be orthogonal to the k-th Krylov subspace.
- GMRES/MINRES: The norm of the residual is minimized over the k-th Krylov subspace.
- BICG: The residual is orthogonal to the Krylov subspace associated with A^T.

Arnoldi method

We want to create an orthogonal basis for the k-th Krylov subspace.

$q_1 = b / ||b||$
for j = 1, ..., k - 1
    $t = A\, q_j$
    for i = 1, ..., j
        $h_{i,j} = q_i^T t$
        $t = t - h_{ij}\, q_i$
    $h_{j+1,j} = ||t||$
    $q_{j+1} = t / h_{j+1,j}$
end

Multiply to get next vector

Orthogonalize against previous vectors

Normalize

$$A \begin{bmatrix} q_1 & \dots & q_j \end{bmatrix} = \begin{bmatrix} q_1 & \dots & q_j & q_{j+1} \end{bmatrix} \begin{bmatrix} h_{1,1} & h_{1,2} & \dots & h_{1,j} \\ h_{2,1} & h_{2,2} & \dots & h_{2,j} \\ & & & \vdots \\ 0 & h_{3,2} & \dots & \vdots \\ 0 & 0 & \dots & h_{j+1,j} \end{bmatrix}, \; j = 1, \dots, k-1$$

Arnoldi method...

Rewrite the Arnoldi algorithm as

$$AQ_j = Q_j H_j + h_{j+1,j} q_{j+1} e_j^T$$

$$Q_j = \begin{bmatrix} q_1 & \dots & q_j \end{bmatrix}, \; H_j = \begin{bmatrix} h_{1,1} & h_{1,2} & \dots & h_{1,j} \\ h_{2,1} & h_{2,2} & \dots & h_{2,j} \\ & \ddots & \dots & \\ 0 & & & h_{j,j} \end{bmatrix}$$

Orthogonal columns

Hessenberg matrix

By construction the columns of Q_j form an orthogonal basis of the j-th Krylov subspace.

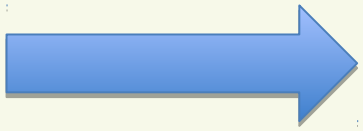
FOM – Full Orthogonalization Method

Consider the linear system $Ax = b$ with initial approximation x_0 .
We rewrite the linear system as $Ay = b - Ax_0 = r_0$

for $x = x_0 + y$. We now define the Krylov subspace with starting vector r_0 and write $x_k = x_0 + Q_k \tilde{y}_k$

We now want to find $\tilde{y}_k$ such that $r_k \perp \mathcal{K}_k(A, r_0)$, or equivalently

$$Q_k^T(b - Ax_k) = Q_k^T(b - Ax_k - AQ_k\tilde{y}_k) = Q_k^T(r_0 - AQ_k\tilde{y}_k) = 0.$$



$$Q_k^T A Q_k \tilde{y}_k = Q_k^T r_0$$

equivalently $H_k \tilde{y}_k = h_{11} e_1$

FOM – Full Orthogonalization Method

We have the following simple result for the residual

$$\begin{aligned} r_k &= b - Ax_k \\ &= b - Ax_0 - AQ_k\tilde{y}_k \\ &= h_{11}q_1 - Q_kH_k\tilde{y}_k - h_{k+1,k}e_k^T\tilde{y}_kq_{k+1} \\ &= h_{11}q_1 - h_{11}Q_ke_1 - h_{k+1,k}e_k^T\tilde{y}_kq_{k+1} \\ &= -h_{k+1,k}e_k^T\tilde{y}_kq_{k+1}. \end{aligned}$$

It follows that .

$$\|r_k\| = |h_{k+1,k}$$

In practice, the full orthogonalization method is rarely used. Instead,
the standard solver for many problems is GMRES - Generalized Minimum Residuals.

GMRES

The idea of GMRES is simple:

- Perform an Arnoldi iteration
- In each step of the Arnoldi iteration minimize the residual

$$\|Ax_k - b\|_2 = \|A(x_0 + Q_k\tilde{y}_k - b\|_2 = \|AQ_k\tilde{y}_k - r_0\|_2.$$

From the structure of the Arnoldi iteration we have equivalently

$$\begin{aligned}\|AQ_k\tilde{y}_k - r_0\|_2 &= \|Q_kH_k\tilde{y}_k - r_0 + h_{k+1,k}q_{k+1}e_k^T\tilde{y}_k\|_2 \\ &= \| \begin{bmatrix} H_k \\ h_{k+1,k}e_k^T \end{bmatrix} \tilde{y}_k - \|r_0\|_2 e_1\|_2\end{aligned}$$

In each step minimize this expression.

GMRES as Approximation Problem

$$\begin{aligned}\|r_k\|_2 &= \|r_0 - AQ\tilde{y}_k\|_2 \\ &= \|r_0 - AP_{k-1}(A)r_0\|_2 \\ &= \|(I - AP_{k-1}(A))r_0\|_2\end{aligned}$$

Polynomial of degree k such that P(0) = 1.

GMRES Approximation Problem: Find a polynomial P_k with P(0) = 1 such that

$$\|P(A)r_0\|_2 = \text{minimum}$$

GMRES Convergence

A simple observation:

$$\|r_{k+1}\|_2 \leq \|r_k\|_2$$

The reason is that GMRES minimises the residual over Krylov subspaces of growing dimension k.

Expressing the residual in terms of polynomials we obtain

$$\frac{\|r_k\|}{\|r_0\|} \leq \inf_{p_k \in \mathcal{P}_k} \|p_k(A)\|$$

Space of all polynomials of maximum degree k with $p_n(0) = 1$.

GMRES Convergence...

Let A be diagonalizable with

$$A = V\Lambda V^{-1}.$$

At step k of the GMRES iteration, the residual r_k satisfies

$$\frac{\|r_k\|}{\|r_0\|} \leq \inf_{p_k \in \mathcal{P}_k} \|p_k(A)\| \leq \|V\| \cdot \|V^{-1}\| \inf_{p_k \in \mathcal{P}_k} \|p_k\|_{\Lambda(A)}$$

Rule of thumb: If the eigenvalues are bounded away from zero and not too far apart from each other then GMRES will converge quickly.

GMRES Convergence...

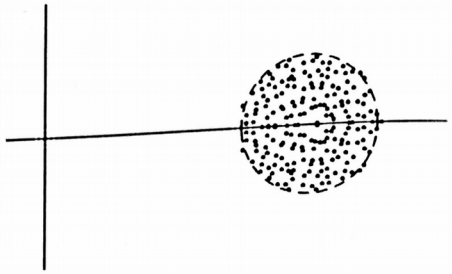


Figure 35.2. *Eigenvalues of the* 200×200 *matrix* A *of* (35.17). *The dashed curve is the circle of radius* $1/2$ *with center* $z = 2$ *in* $\mathbb{C}$. *The eigenvalues a approximately uniformly distributed within this disk.*

GMRES for a matrix with tightly clustered spectrum around center z=2. We can observe fast exponential convergence.

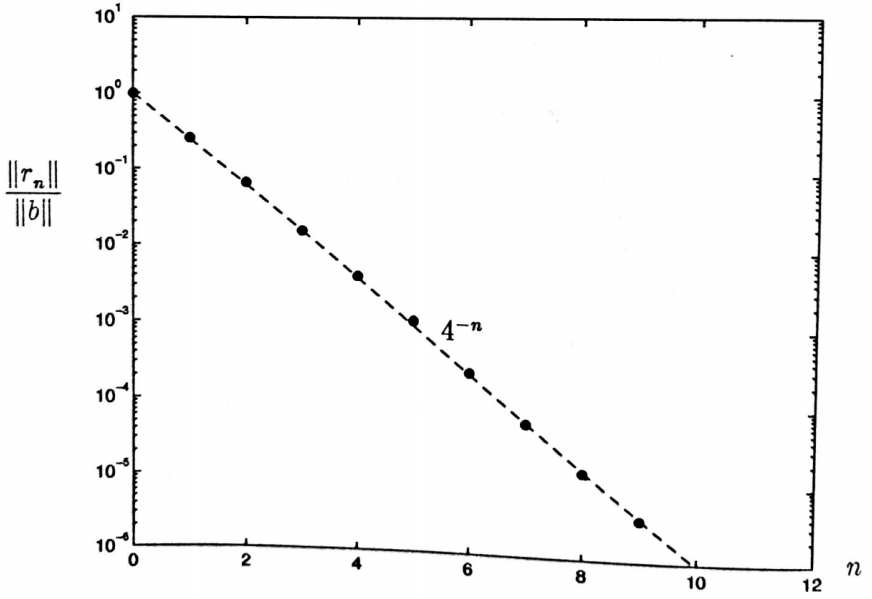


Figure 35.3. *GMRES convergence curve for the same matrix* A. *This rapid, steady convergence is illustrative of Krylov subspace iterations under ideal circumstances, when* A *is a well-behaved (or well-preconditioned) matrix.*

L. N. Trefethen, D. Bau, *Numerical Linear Algebra*, Lecture 35.

GMRES Convergence...

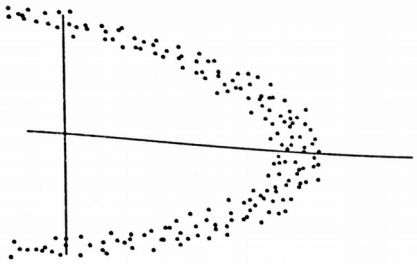


Figure 35.4. *Eigenvalues of a* 200×200 *matrix, like that of* (35.17) *except with a modified diagonal. Now the eigenvalues surround the origin on one side.*

GMRES for a matrix, where the eigenvalues surround the origin. GMRES still converges, but very slowly.

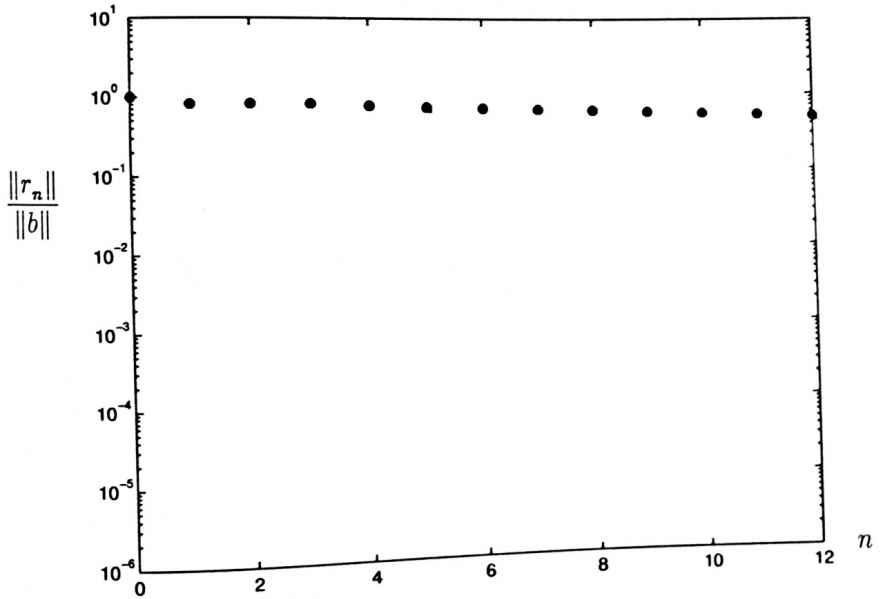


Figure 35.5. *GMRES convergence curve for the matrix of Figure* 35.4. *The convergence has slowed down greatly. When an iterative method stagnates like this, it is time to look for a better preconditioner.*

L. N. Trefethen, D. Bau, *Numerical Linear Algebra*, Lecture 35.

Symmetric Problems

Let now the matrix A be symmetric.

$$A = A^T$$

From the symmetry we obtain

$$\begin{aligned} H_k^T &= \left[Q_k^T A Q_k\right]^T \\ &= Q_k^T A^T Q_k \\ &= Q_k^T A Q_k \\ &= H_k \end{aligned}$$

Hence, H is symmetric. Since it is also upper Hessenberg, it must be tridiagonal.

The Lanczos Algorithm

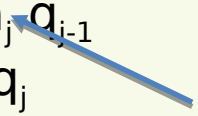
We have seen that H is symmetric and tridiagonal. Therefore, we have

$$H_k =: T_k = \begin{bmatrix} \alpha_1 & \beta_2 & & \\ \beta_2 & \alpha_2 & \beta_3 & \\ & \ddots & \ddots & \ddots \\ & \beta_{k-1} & \alpha_{k-1} & \beta_k \\ & & \beta_k & \alpha_k \end{bmatrix}.$$

This structure simplifies the Arnoldi method significantly. Instead of all previous vectors we only need to orthogonalize against the last two vectors.

The Lanczos Algorithm…

$q_1 = b / ||b||$, $beta_1 = 0$,
$q_0 = 0$
for $j = 1, \ldots, k - 1$
$w = A q_j - beta_j q_{j-1}$
$alpha_j = w_j^T q_j$
$w_j = w_j - alpha_j q_j$
$beta_{j+1} = \|w_j\|$
$q_{j+1} = w_j / beta_{j+1}$
end



Only orthogonalize against the two previous vectors.

Much cheaper than Arnoldi. We do not need to keep all previous vectors in memory, but only two.

For symmetric matrices we can apply GMRES with the short recursion and arrive at MINRES, a cheap iterative solver for symmetric linear systems of equations.

Conjugate Gradients

We now assume that A is symmetric positive definite.

Define the residuals such that

$$r_k \perp \mathcal{K}_k(A, r_0).$$

We have $r_k = b - Ax_k$. Hence, $r_k \in \mathcal{K}_{k+1}(A, r_0)$ However, since also $r_k \perp \mathcal{K}_k(A, r_0)$ it follows that r_k shows in the same direction as q_{k+1} .
We have

$$r_i^T r_k = 0, \ i < k$$

Define

$$\Delta r = r_k - r_{k-1}, \ \Delta x = x_k - x_{k-1}$$

It follows that

$$(x_i - x_{i-1})^T (r_k - r_{k-1}) = 0, \ i < k$$

$$r_k - r_{k-1} = (b - Ax_k) - (b - Ax_{k-1}) = -A(x_k - x_{k-1})$$

$$(x_i - x_{i-1})^T A(x_k - x_{k-1}) = 0, \ i < k$$

Conjugate Gradients...

Start with $d_0 = r_0 = b - Ax_0$ and x_0.

1. $\alpha_k = \frac{r_{k-1}^T r_{k-1}}{d_{k-1}^T A d_{k-1}}$
2. $x_k = x_{k-1} + \alpha_k d_{k-1}$
3. $r_k = r_{k-1} - \alpha_k A d_{k-1}$
4. $\beta_k = r_k^T r_k / r_{k-1}^T r_{k-1}$
5. $d_k = r_k + \beta_k d_{k-1}$

Update solution

Update residual

Update search direction

- In each step one matvec is necessary
- CG is only stable for symmetric positive definite matrices
- Very interesting behavior in finite precision!

CG as optimization method

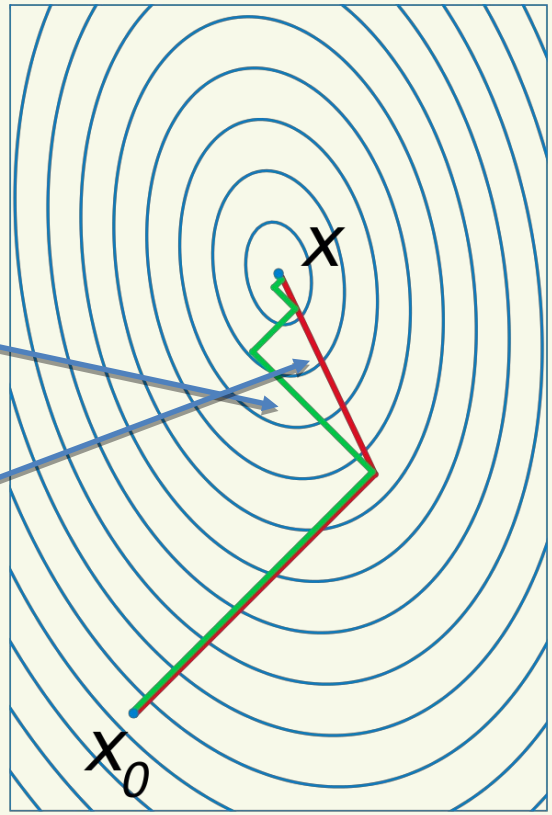
CG can be interpreted as a line search numerical optimization method to minimize the function

$$f(x) = \frac{1}{2}x^T Ax - x^T b$$

using a sequence of conjugate directions.

Line search with orthogonal directions.

Line search with conjugate directions.



This interpretation of CG can be generalized to nonlinear optimization problems.

What method to use?

- Symmetric (Hermitian) Positive Definite: CG is the best choice
- Symmetric (Hermitian): Use Minres
- Nonsymmetric: If there are not too many iterations use GMRES. If memory and large number of iterations are a concern BiCGStab is a frequently used solver.
- Lots of other solvers exist (QMR, CGLS, etc.)
- None of these work well if the problem is ill-conditioned. In that case, preconditioning is necessary (to come…)

Iterative Solvers in Scipy

Iterative methods for linear equation systems:

bicg(A, b[, x0, tol, maxiter, xtype, M, ...])	Use BIConjugate Gradient iteration to solve A x = b
bicgstab(A, b[, x0, tol, maxiter, xtype, M, ...])	Use BIConjugate Gradient STABilized iteration to solve A x = b
cg(A, b[, x0, tol, maxiter, xtype, M, callback])	Use Conjugate Gradient iteration to solve A x = b
cgs(A, b[, x0, tol, maxiter, xtype, M, callback])	Use Conjugate Gradient Squared iteration to solve A x = b
gmres(A, b[, x0, tol, restart, maxiter, ...])	Use Generalized Minimal RESidual iteration to solve A x = b.
lgmres(A, b[, x0, tol, maxiter, M, ...])	Solve a matrix equation using the LGMRES algorithm.
minres(A, b[, x0, shift, tol, maxiter, ...])	Use MINimum RESidual iteration to solve Ax=b
qmr(A, b[, x0, tol, maxiter, xtype, M1, M2, ...])	Use Quasi-Minimal Residual iteration to solve A x = b

Iterative methods for least-squares problems:

lsqr(A, b[, damp, atol, btol, conlim, ...])	Find the least-squares solution to a large, sparse, linear system of equations.
lsmr(A, b[, damp, atol, btol, conlim, ...])	Iterative solver for least-squares problems.

Preconditioning

An iterative solver for

$$Ax = b$$

converges best it the eigenvalues of A are tightly clustered away from zero. Most practical problems do not satisfy this property, e.g. FEM discretization of Laplace operator.

Preconditioning. Let $P \approx A^{-1}$. P is called a preconditioner. We now solve instead

$$PAx = Pb.$$

Preconditioning...

- The preconditioner P needs not be explicitly given as matrix, but should be easy to evaluate.
- The best preconditioner is the inverse of A. But evaluating it is as expensive as solving the original system.
- Analytic Preconditioners: Derive P from a simpler to solve PDE problem that approximates the original problem in some some sense.
- Algebraic Preconditioner: Algebraically derive an easy to invert approximation of A.

Splitting Preconditioners

$$x_{k+1} = M^{-1}b + M^{-1}Nx_k$$

Substitute $N = M - A$ to obtain

$$\begin{aligned} x_{k+1} &= M^{-1}b + M^{-1}(M - A)x_k \\ &= M^{-1}b + (I - M^{-1}A)x_k \end{aligned}$$

In the limit we have

$$x = M^{-1}b + (I - M^{-1}A)x$$

or equivalently

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

We can solve this system directly with a Krylov subspace method.

ILU – Incomplete LU Decomposition

ALGORITHM **10.2** *Gaussian Elimination – IKJ Variant*

1. For $i = 2, \ldots, n$ Do:
2. For $k = 1, \ldots, i-1$ Do:
3. $a_{ik} := a_{ik}/a_{kk}$
4. For $j = k+1, \ldots, n$ Do:
5. $a_{ij} := a_{ij} - a_{ik} * a_{kj}$
6. EndDo
7. EndDo
8. EndDo

- Based on the IKJ Variant of Gaussian Elimination
- Only update an entry if it is not part of a predetermined zero pattern
- ILU(0): Use the zero pattern of original matrix – no fill-in possible
- Use resulting approximate LU Decomposition as Preconditioner

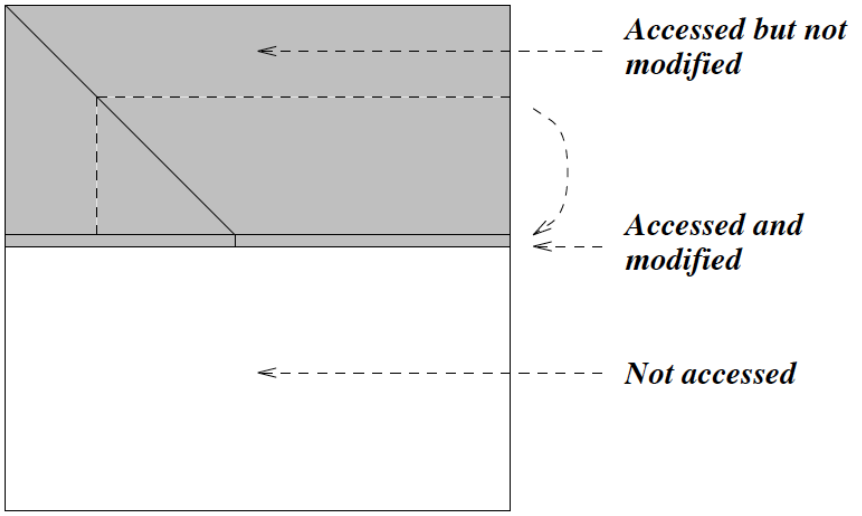
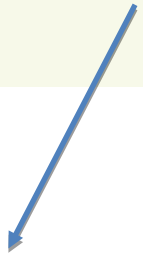


Figure 10.1: IKJ variant of the LU factorization.

ILU(0)

Non-Zero Pattern of A



ALGORITHM **10.4** *ILU(0)*

1. For $i = 2, \ldots, n$ Do:
2. For $k = 1, \ldots, i-1$ and for $(i,k) \in NZ(A)$ Do:
3. Compute $a_{ik} = a_{ik}/a_{kk}$
4. For $j = k+1, \ldots, n$ and for $(i,j) \in NZ(A)$, Do:
5. Compute $a_{ij} := a_{ij} - a_{ik}a_{kj}$.
6. EndDo
7. EndDo

ILU(P)

Definition 10.5 *The initial level of fill of an element a_{ij} of a sparse matrix A is defined by*

$$lev_{ij} = \begin{cases} 0 & \text{if } a_{ij} \neq 0, \text{or } i = j \\ \infty & \text{otherwise.} \end{cases}$$

Each time this element is modified in line 5 of Algorithm 10.2, its level of fill must be updated by

$$lev_{ij} = \min\{lev_{ij}, lev_{ik} + lev_{kj} + 1\}. \qquad (10.17)$$

ALGORITHM **10.5** *ILU(p)*

1. *For all nonzero elements a_{ij} define $lev(a_{ij}) = 0$*
2. *For $i = 2, \ldots, n$ Do:*
3. *For each $k = 1, \ldots, i-1$ and for $lev(a_{ik}) \leq p$ Do:*
4. *Compute $a_{ik} := a_{ik}/a_{kk}$*
5. *Compute $a_{i*} := a_{i*} - a_{ik}a_{k*}$.*
6. *Update the levels of fill of the nonzero $a_{i,j}$'s using (10.17)*
7. *EndDo*
8. *Replace any element in row i with $lev(a_{ij}) > p$ by zero*
9. *EndDo*

Multigrid Methods

- PDE adapted solver technique for non-oscillatory problems
- Grid independent convergence for iterative solvers
- Multigrid methods are tremendously successful and have impact far beyond just iterative solver techniques

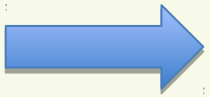
A model problem

$$-u''(x) = f(x),\ x \in (0,1)$$
$$u(0) = u(1) = 0$$

Choose n equally spaced points

$$x_i = i \times h,\ i = 0, \ldots, n+1,\ h = 1/(n+1)$$

and approximate $u''(x_i) \approx \dfrac{x_{i+1} - 2x_i + x_{i-1}}{h^2}$



$$Ax = b$$

$$A = \begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{bmatrix}, \quad b = h^2 \begin{bmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_{n-1}) \\ f(x_n) \end{bmatrix}$$

Eigenvalues and Eigenvectors

We will analyze the behavior of iterative methods by considering the eigenvalues and eigenvectors of A.

$$Av_k = \lambda_k v_k$$

$$\lambda_k = 4\sin^2\frac{\theta_k}{2},\ \theta_k = \frac{k\pi}{n+1}$$

$$v_k = \sin(k\pi x_i)$$

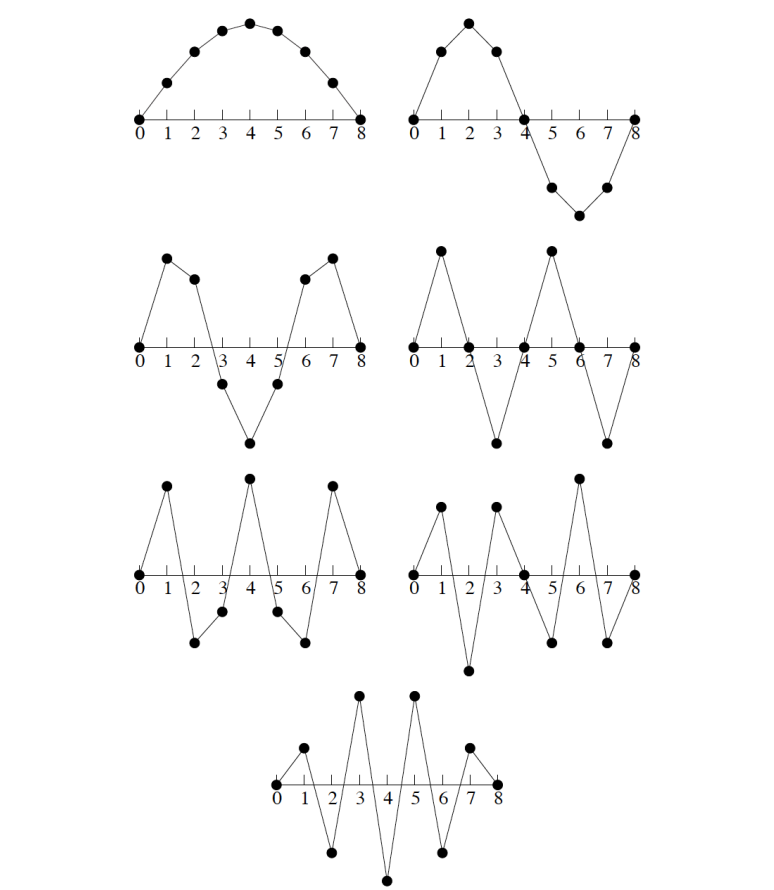


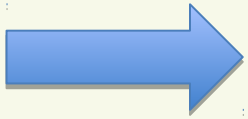
Figure 13.1: The seven eigenfunctions of the discretized one-dimensional Laplacean when $n = 7$.

Richardson's iteration

$$x_{j+1} = x_j + \omega(b - Ax_j)$$
$$= (I - \omega A)x_j + \omega b$$

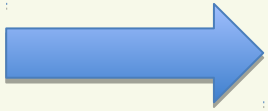
Bounds for eigenvalues μ_i of iteration matrix $M_\omega = I - \omega A$.

$$1 - \omega\lambda_n \leq \mu_i \leq 1 - \omega\lambda_1$$



$$\omega < \frac{2}{\lambda_n}$$

Let $\gamma > \lambda_n$ and choose $\omega = \frac{1}{\gamma}$.



Richardson's iteration converges.

Error of Richardson's iteration

Error in iteration j:

$$e_j \equiv x^* - x_j$$

Limit of Richardson's iteration.

Expand the error e_0 in the basis of eigenvectors of A.

$$e_0 = \sum_{k=1}^{n} \xi_k v_k$$

We obtain

$$e_j = M_\omega^j e_0 = \sum_{k=1}^{n} \left(1 - \frac{\lambda_k}{\gamma}\right)^j \xi_k v_k$$

Each error component is reduced by $\left(1 - \frac{\lambda_k}{\gamma}\right)^j$.

Error of Richardson's iteration...

Choose $\gamma = 4$. In the model problem the reduction coefficient for the error with respect to the smallest eigenvalue is

$$1 - \sin^2 \frac{\pi}{2(n+1)} \approx 1 - \left(\frac{\pi h}{2}\right)^2 = 1 - \mathcal{O}(h^2).$$

As h decreases the rate of convergence becomes arbitrarily slow.
However, the oscillatory part (error components associated with eigenvalues k > n / 2) are decreased rapidly since in that case.

$$\mu_k = 1 - \sin^2 \frac{k\pi}{2(n+1)} = \cos^2 \frac{k\pi}{2(n+1)} \leq \frac{1}{2}.$$

Reduction coefficients for model problem

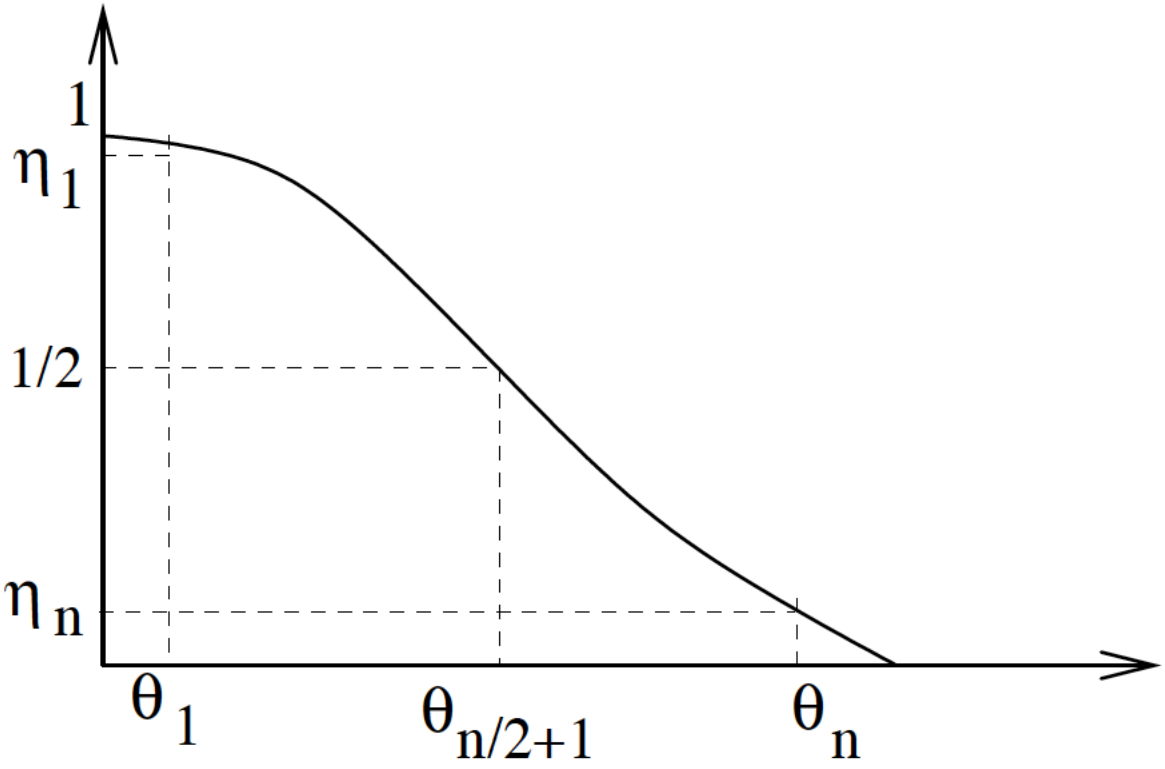


Figure 13.2: Reduction coefficients for Richardson's method applied to the 1-D model problem

Coarse Grid Problem

Assume that n is odd. Denote by Ω_{2h} the grid obtained by going from h to 2h. We have

$$x_i^{2h} = i(2h),\ x_i^{2h} = x_{2i}^h$$

and therefore

$$v_k(x_{2i}^h) = \sin(k\pi x_{2i}^h) = sin(k\pi x_i^{2h}) = v_k^{2h}(x_i^{2h})$$

for $k \leq n/2$.

Some of the non-oscillatory modes on the fine grid become oscillatory modes on the coarse grid.

Multigrid idea: Eliminate high-frequency error components on the fine grid and then move to coarse grid to reduce low-frequency errors.

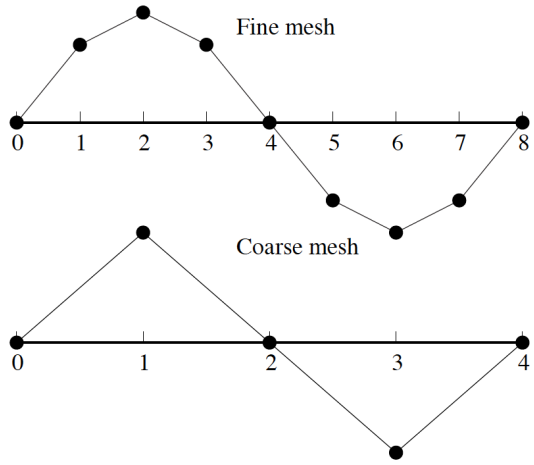


Figure 13.3: The mode w_2 on a fine grid $(n = 7)$ and a coarse grid $(n = 3)$.

Moving between fine grid and coarse grid

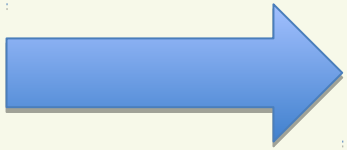
Fine grid: Ω^h , Coarse grid:Ω^H ,
H=2h

Prolongation Operator

$$I_H^h : \Omega_H \to \Omega_h$$

We assume that n is odd and define the prolongation operation $v^h = I_{2h}^h v^{2h}$ by

$$\begin{cases} v_{2j}^h &= v_j^{2h} \\ v_{2j+1}^h &= (v_j^{2h} + v_{j+1}^{2h})/2 \end{cases} \quad \text{for} \quad j = 0, \ldots, \frac{n+1}{2} .$$



Matrix form

$$v^h = \frac{1}{2} \begin{bmatrix} 1 & & & & \\ 2 & & & & \\ 1 & 1 & & & \\ & 2 & & & \\ & 1 & 1 & & \\ & & \vdots & & \\ & & \vdots & & \\ & & 1 & 1 & \\ & & & 2 & \\ & & & 1 & \end{bmatrix} v^{2h} .$$

Moving between fine grid and coarse grid...

Restriction of a function v^h onto v^{2h} :

$$v_j^{2h} = \frac{1}{4}\left(v_{2j-1}^h + 2v_{2j}^h + v_{2j+1}^h\right)$$

$$I_h^{2h} = \frac{1}{4}\begin{bmatrix} 1 & 2 & 1 & & & & & & & & \\ & & 1 & 2 & 1 & & & & & & \\ & & & & 1 & 2 & 1 & & & & \\ & & & & \cdots & \cdots & \cdots & & & & \\ & & & & & & & 1 & 2 & 1 \end{bmatrix}.$$

It follows that

$$I_{2h}^h = 2(I_h^{2h})^T$$

The Multigrid Cycle

We want to solve

$$A_h u^h = f^h$$

on the fine grid.

Define the smoothing operation

$$u_\nu^h = \text{smooth}^\nu(A_h, u_0^h, f_h)$$

It applies ν steps of a smoother on the linear system

$$A_h u^h = f^h$$

The smoothing operation is given by

$$u_{j+1}^h = S_h u_j^h + g^h \quad \Longleftrightarrow \quad u_{j+1}^h = u_j^h + B_h(f^h - A_h u_j^h)$$

$$S_h \equiv I - B_h A_h, \quad B_h \equiv (I - S_h)^{-1} A_h^{-1}, \quad g^h \equiv B_h f^h$$

For Richardson's iteration: $B_h = \omega I, \quad S_h = I - \omega A$

The Multigrid Cycle...

After the smoothing step for the error and residual have

$$e_\nu^h = (S_h)^\nu e_0^h = (I - B_h A_h)^\nu e_0^h, \quad r_h^\nu = (I - A_h B_h)^\nu r_0^h$$

Two Grid Cycle

1.	*Pre-smooth:*	$u^h := \texttt{smooth}^{\nu_1}(A_h, u_0^h, f^h)$
2.	*Get residual:*	$r^h = f^h - A_h u^h$
3.	*Coarsen:*	$r^H = I_h^H r^h$
4.	*Solve:*	$A_H \delta^H = r^H$
5.	*Correct:*	$u^h := u^h + I_H^h \delta^H$
6.	*Post-smooth:*	$u^h := \texttt{smooth}^{\nu_2}(A_h, u^h, f^h)$

The Multigrid Cycle

We want to write a single Two-Grid cycle as an iteration of the form

$$u_{new}^h = M_h u_0^h + g_{M_h}.$$

We obtain

$$u_{new}^h = S_h^{\nu_2} \left[S_h^{\nu_1} u_0^h + I_H^h A_H^{-1} I_h^H (-A_h S_h^{\nu_1} u_0^h) \right]$$

and therefore

$$M_H = S_h^{\nu_2} \left[I - I_H^h A_H^{-1} I_h^H A_h \right] S_h^{\nu_1}.$$

We define

$$T_h^H = I - I_H^h A_H^{-1} I_h^H A_h.$$

This matrix is called the coarse grid correction.

V-Cycle

ALGORITHM **13.3** $u^h = \text{V-cycle}(A_h, u_0^h, f^h)$

1. *Pre-smooth:* $u^h := \texttt{smooth}^{\nu_1}(A_h, u_0^h, f^h)$
2. *Get residual:* $r^h = f^h - A_h u^h$
3. *Coarsen:* $r^H = I_h^H r^h$
4. *If* $(H == h_0)$
5. *Solve:* $A_H \delta^H = r^H$
6. *Else*
7. *Recursion:* $\delta^H = \text{V-cycle}(A_H, 0, r^H)$
8. *EndIf*
9. *Correct:* $u^h := u^h + I_H^h \delta^H$
10. *Post-smooth:* $u^h := \texttt{smooth}^{\nu_2}(A_h, u^h, f^h)$
11. *Return* u^h

We recursively apply the Two-Cycle to solve the coarse grid system.

General Multigrid cycle

ALGORITHM **13.4** $u^h = MG\,(A_h, u_0^h, f^h, \nu_1, \nu_2, \gamma)$

1. *Pre-smooth:* $u^h := \texttt{smooth}^{\nu_1}(A_h, u_0^h, f^h)$
2. *Get residual:* $r^h = f^h - A_h u^h$
3. *Coarsen:* $r^H = I_h^H r^h$
4. *If* $(H == h_0)$
5. *Solve:* $A_H \delta^H = r^H$
6. *Else*
7. *Recursion:* $\delta^H = MG^{\gamma}(A_H, 0, r^H, \nu_1, \nu_2, \gamma)$
8. *EndIf*
9. *Correct:* $u^h := u^h + I_H^h \delta^H$
10. *Post-smooth:* $u^h := \texttt{smooth}^{\nu_2}(A_h, u^h, f^h)$
11. *Return* u^h

Number of repetitions.

V-Cycle: $\gamma = 1$
W-Cycle: $\gamma = 2$

General Multigrid cycle...

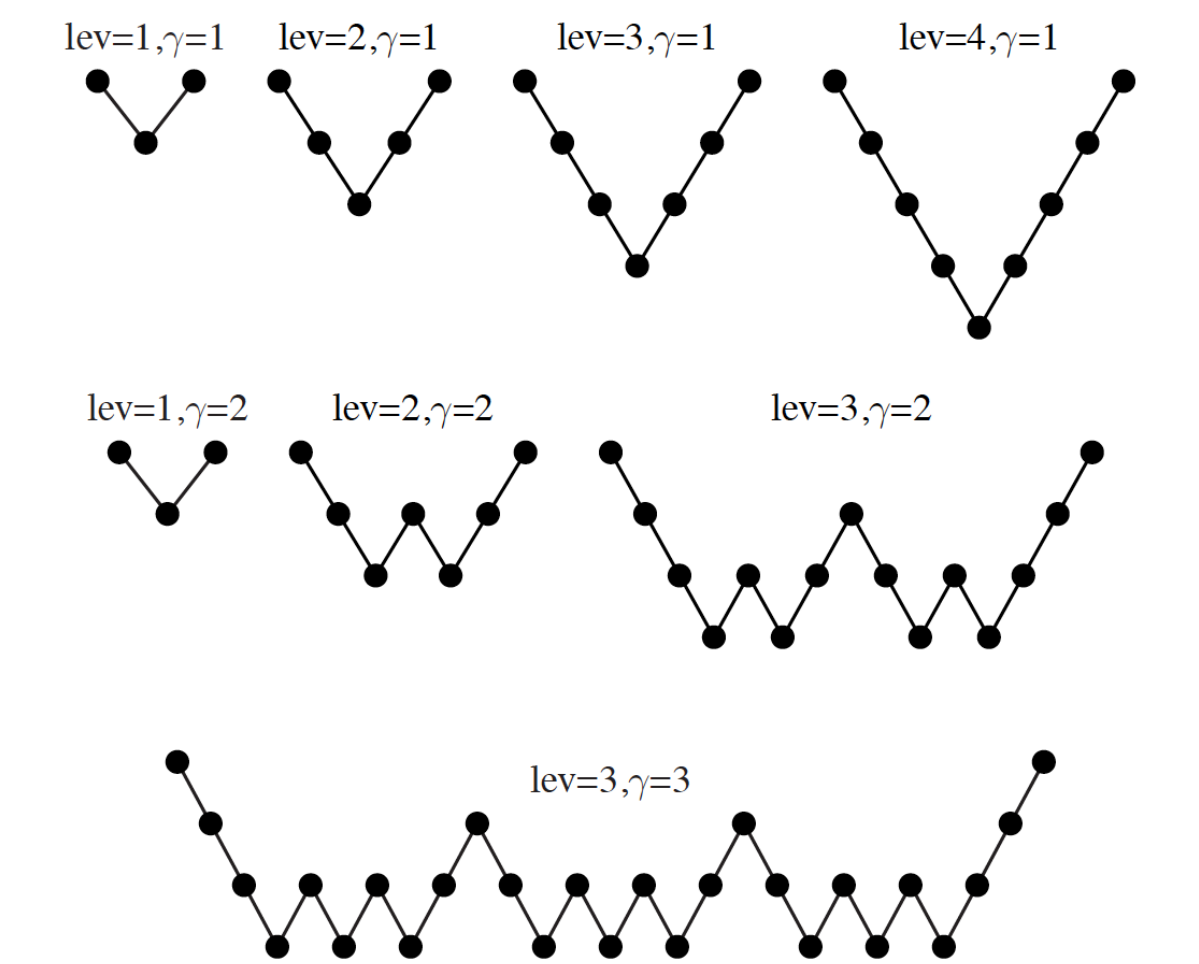


Figure 13.6: Representations of Various V-cycles and W-cycles

Algebraic Multigrid

- Multigrid requires a sequence of meshes and restriction and prolongation operators between them.
- In practice this is often not available.
- We would like to have a Multigrid like technique that works directly on the matrix without requiring a hierarchy of meshes.

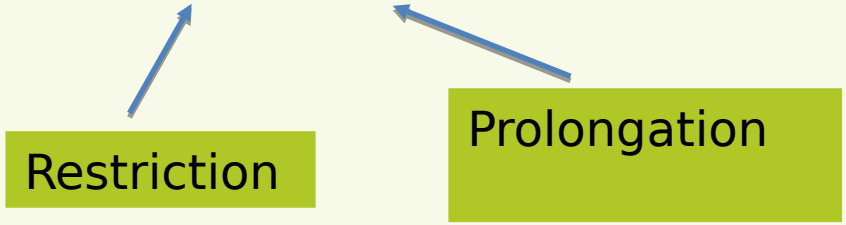
Multigrid ingredients

Generically, multigrid methods require two ingredients:

1. A way to define a coarse subspace X_H from a fine subspace X_h .
2. An interpolation operator I_h^H .

Coarse grid problem

$$\left[I_h^H\right]^T A_h I_h^H u_H = \left[I_h^H\right]^T f_h$$



AMG Ideas

For smooth error s can show heuristically that in Multigrid

$$\sum_{j \neq i} \frac{|a_{ij}|}{|a_{ii}|} \left(\frac{s_i - s_j}{s_i} \right)^2 \ll 1$$

If $\frac{|a_{ij}|}{|a_{ii}|}$ is large then the corresponding error $s_i - s_j$

must be small, that is the error s changes slowly along strongly connected nodes in the adjacency graph.

Given threshold parameter

Weak connection: $|a_{ij}|/|a_{ii}| < \sigma$

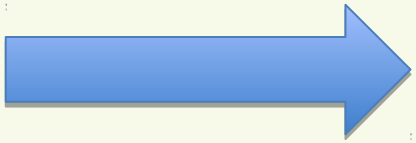
$|a_{ij}|/|a_{ii}| > \sigma$

Strong connection:

Around a given node i split up the neighbors into coarse grid nodes, strong connections and weak connections and treat those separately.

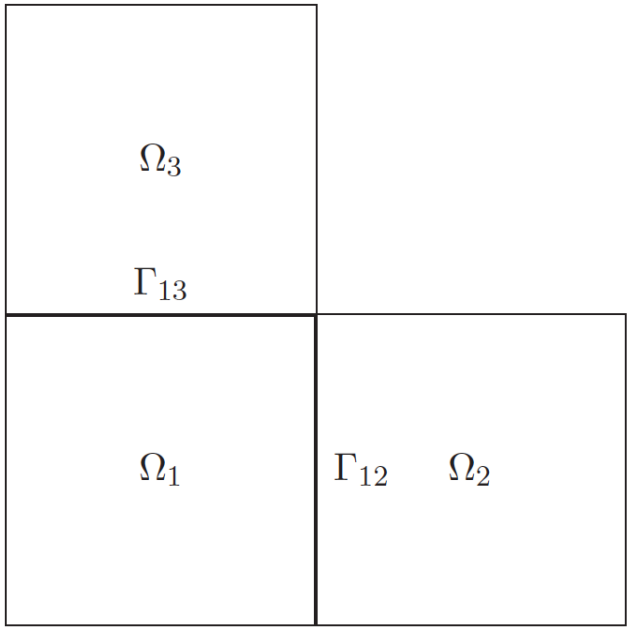
Multigrid and Krylov solvers

- We can use Multigrid directly as solver
- Alternative to Krylov subspace methods
- Works extremely well for Poisson type problems
- In practice very often use AMG as preconditioner for a Krylov subspace method



Best of both worlds!!

Domain Decomposition



Model Problem

$$-\Delta u = f, \text{ in } \Omega$$
$$u = u_\Gamma, \text{ on } \Gamma$$

- Split complicated geometry into easier subdomains
- Allow parallel solves on the subdomains

$$\Omega = \bigcup_{i=1}^{s} \Omega_i$$

Partitioning

$$\begin{bmatrix} B_1 & & & & E_1 \\ & B_2 & & & E_2 \\ & & \ddots & & \vdots \\ & & & B_s & E_s \\ F_1 & F_2 & \dots & F_s & C \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_s \\ y \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_s \\ g \end{bmatrix}$$

Interior dofs

Local stiffness matrices

Interface dofs

$$A \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} f \\ g \end{bmatrix}$$

$$A = \begin{bmatrix} B & E \\ F & C \end{bmatrix}$$

- Sort dofs by interior and interface dofs
- Obtain an arrowhead structure
- We will reduce this system to a problem purely defined on the interfaces

Partitioning...

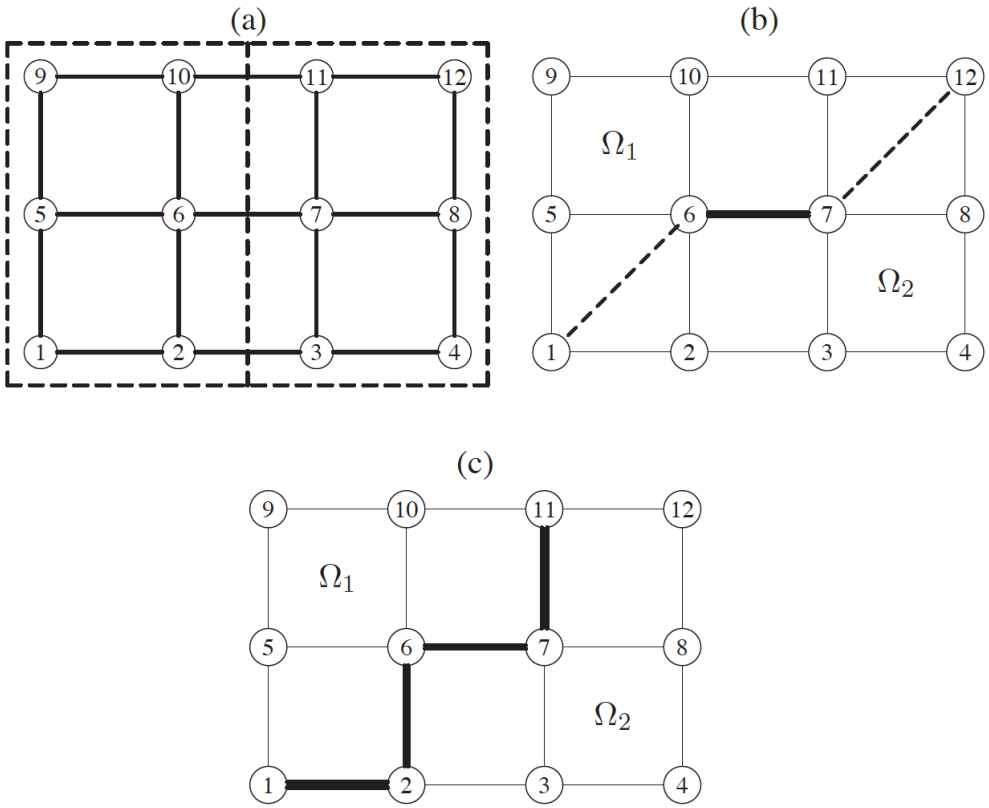


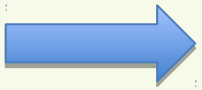
Figure 14.2: (a) Vertex-based, (b) edge-based, and (c) element-based partitioning of a 4×3 mesh into two subregions.

Domain Decomposition Types

- Type of Partitioning: Along edges, elements or vertices
- Overlap: Should subdomains overlap or not
- Treatment of interfaces: Schur complements, interface value updates,...
- Subdomain solution: Approximate solver or direct solver, local preconditioning...

Schur complements

$$\begin{bmatrix} B & E \\ F & C \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} f \\ g \end{bmatrix}$$



$$x = B^{-1}(f - Ey)$$

Reduced system

$$(C - FB^{-1}E)y = g - FB^{-1}f$$

Define the Schur complement

$$S = C - FB^{-1}E$$

With have $E' = B^{-1}E, \; f' = B^{-1}f$

Subdomain PDE solves

$$(C - FE')y = g - Ff'$$

$$x = f' - E'y$$

Parallel solve for interface dofs

Properties of the Schur Complement

Block LU Decomposition gives

$$\begin{bmatrix} B & E \\ F & C \end{bmatrix} = \begin{bmatrix} I & 0 \\ FB^{-1} & I \end{bmatrix} \begin{bmatrix} B & E \\ 0 & S \end{bmatrix}$$

Schur Complement

$$\begin{bmatrix} B & E \\ F & C \end{bmatrix}^{-1} = \begin{bmatrix} B^{-1} & -B^{-1}ES^{-1} \\ 0 & S^{-1} \end{bmatrix} \begin{bmatrix} I & 0 \\ -FB^{-1} & I \end{bmatrix}$$

$$= \begin{bmatrix} B^{-1} + B^{-1}ES^{-1}FB^{-1} & -B^{-1}ES^{-1} \\ -S^{-1}FB^{-1} & S^{-1} \end{bmatrix}$$

The inverse of the Schur complement is the bottom right entry of the inverse of A.

Properties of the Schur Complement...

Let A be nonsingular and partitioned as before, such that the submatrix B is nonsingular and let R_y be the restriction operator onto the interface variables, that is

$$R_y \begin{bmatrix} x \\ y \end{bmatrix} = y.$$

The following properties are true:

- The Schur complement matrix S is nonsingular
- If A is symmetric pos. definite, then so is S
- For any y,

$$S^{-1}y = R_y A^{-1} \begin{bmatrix} 0 \\ y \end{bmatrix}.$$

Schur complement for FEM partitioning

Variational Problem

$$\text{Find } u \in V_h \text{ such that } a(u,v) = (f,v) \text{ for all } v \in V_h.$$

$$a(u,v) = \int_\Omega \nabla u \cdot \nabla v.$$

Assume disjoint subdomains.
Then

$$a_i(u,v) = \sum_{\Omega_i} \nabla u \cdot \nabla v$$

Schur complement for FEM partitioning...

Assume standard P1 nodal basis functions.
Sort dofs according to interface nodes and interior nodes to obtain.

$$\begin{bmatrix} B_1 & & & & E_1 \\ & B_2 & & & E_2 \\ & & \ddots & & \vdots \\ & & & B_s & E_s \\ F_1 & F_2 & \dots & F_s & C \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_s \\ y \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_s \\ g \end{bmatrix}$$

On ith subdomain obtain local stiffness matrix

$$A_i = \begin{bmatrix} B_i & E_i \\ F_i & C_i \end{bmatrix}.$$

This matrix contains only contributions from elements in this subdomain.

Schur complement for FEM partitioning...

For Schur complement S now have

$$
\begin{aligned}
S &= C - FB^{-1}E \\
&= C - \sum_{i=1}^{s} F_i B_i^{-1} E_i \\
&= \sum_{i=1}^{s} C_i - \sum_{i=1}^{s} F_i B_i^{-1} E_i \\
&= \sum_{i=1}^{s} \left[C_i - F_i B_i^{-1} E_i \right].
\end{aligned}
$$

Can assemble global Schur complement S from
local Schur complements

$$S_i = C_i - F_i B_i^{-1} E_i.$$

Schwarz methods

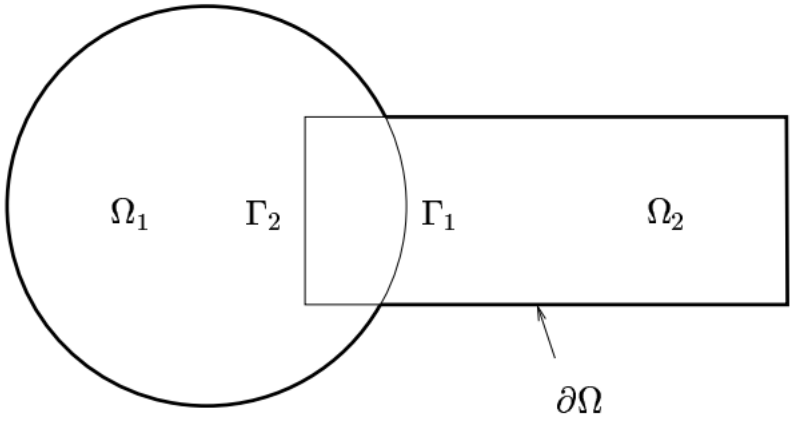


FIGURE 2.1. *The first domain decomposition method was introduced by Schwarz for a complicated domain, composed of two simple ones, namely a disk and a rectangle.*

Here, consider overlapping Schwarz methods, where the subdomains overlap.

M. Gander, *Schwarz Methods in the Course of Time*

Schwarz Alternating Procedure

$$-\Delta u_1^{n+1} = f_1, \text{ in } \Omega, \qquad -\Delta u_2^{n+1} = f_2, \text{ in } \Omega,$$
$$u_1^{n+1} = u_2^n, \text{ on } \Gamma_1 \qquad u_2^{n+1} = u_1^{n+1}, \text{ on } \Gamma_1$$

Matrix form:

$$\begin{bmatrix} A_1 & A_{12} \\ A_{21} & A_2 \end{bmatrix} \begin{bmatrix} u_1^{n+1} \\ u_2^n \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} \qquad \begin{bmatrix} A_1 & A_{12} \\ A_{21} & A_2 \end{bmatrix} \begin{bmatrix} u_1^{n+1} \\ u_2^{n+1} \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix}$$

$$A_1 u_1^{n+1} = f_1 - A_{12} u_2^n \qquad A_2 u_2^{n+1} = f_2 - A_{21} u^{n+1}$$

Schwarz Alternating Procedure...

Residual Form

$$A_1\delta_1^n = f_1 - A_{12}u_2^n - A_1u_1^n =: r_1^n \qquad A_2\delta_2^n = f_2 - A_{21}u_1^{n+1} - A_2u_2^n =: r_2^n$$

$$u_1^{n+1} = u_1^n + \delta_1^n \qquad u_2^{n+1} = u_2^n + \delta_2^n$$

Parallel Schwarz Method

$$A_1\delta_1^n = f_1 - A_{12}u_2^n - A_1u_1^n =: r_1^n \qquad A_2\delta_2^n = f_2 - A_{21}u_1^n - A_2u_2^n =: r_2^n$$

$$u_1^{n+1} = u_1^n + \delta_1^n \qquad u_2^{n+1} = u_2^n + \delta_2^n$$

Algebraic Schwarz Methods

Define restriction operators

$$R_1 = \begin{bmatrix} 1 & & \\ & \ddots & \\ & & 1 \end{bmatrix}, \quad R_2 = \begin{bmatrix} & 1 & \\ & & \ddots \\ & & & 1 \end{bmatrix},$$

and projections

$$A_j = R_j A R_j^T, \ j = 1, 2$$

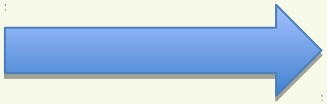
Multiplicative Schwarz Method

$$u^{n+\frac{1}{2}} = u^n + R_1^T A_1^{-1} R_1 (f - A u^n)$$
$$u^{n+1} = u^{n+\frac{1}{2}} + R_2^T A_2^{-1} R_2 (f - A u^{n+\frac{1}{2}})$$

Multiplicative Schwarz

Without overlap obtain

$$\begin{bmatrix} A_1 & 0 \\ A_{21} & A_2 \end{bmatrix} \begin{bmatrix} u_1^{n+1} \\ u_2^{n+1} \end{bmatrix} = \begin{bmatrix} 0 & -A_{12} \\ 0 & 0 \end{bmatrix} \begin{bmatrix} u_1^{n} \\ u_2^{n} \end{bmatrix} + \begin{bmatrix} f_1 \\ f_2 \end{bmatrix}$$



Gauss-Seidel Iteration

Using suitable boundary conditions the multiplicative Schwarz method is identical to the Schwarz alternating procedure, even for overlapping boundaries.

For s subdomains multiplicative Schwarz takes the form

$$u^{n+\frac{j}{s}} = u^{n+\frac{j-1}{s}} + R_j^T A_j^{-1} R_j \left(f - A u^{n+\frac{j-1}{s}} \right), \; j = 1, 2, \ldots, s$$

Multiplicative Schwarz...

Subtracting the iterates from the exact solution and defining the error vector d results in

$$\begin{aligned} u - u^{n+\frac{j}{s}} &= u - u^{n+\frac{j-1}{s}} - R_j^T A_j^{-1} R_j A \left(u - u^{n+\frac{j-1}{s}} \right), \\ d^{n+\frac{j}{s}} &= d^{n+\frac{j-1}{s}} - R_j^T A_j^{-1} R_j A d^{n+\frac{j-1}{s}}, \ j = 1, 2, \ldots, s \\ &= (I - P_j)\, d^{n+\frac{j-1}{s}} \end{aligned}$$

Hence,

$$d^{n+1} = Q_s d^n$$

with

$$Q_s = (I - P_s) \ldots (I - P_1).$$

From $u - u^{n+1} = Q_s(u - u^n)$ obtain

$$u^{n+1} = Q_s u^n + (I - Q_s)u.$$

Multiplicative Schwarz...

Remember the fixed point iteration

$$u^{n+1} = M^{-1}f + (I - M^{-1}A)u^n$$

In the limit have

$$u = M^{-1}f + (I - M^{-1}A)u$$

or equivalently

$$M^{-1}Au = M^{-1}f.$$

Multiplicative Schwarz : $u^{n+1} = Q_s u^n + (I - Q_s)u.$

We therefore obtain

$$M^{-1}A = I - Q_s$$
$$M^{-1}f = (I - Q_s)u = (I - Q_s)A^{-1}f$$

Multiplicative Schwarz...

Define

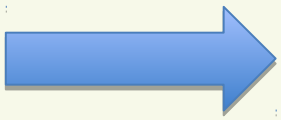
$$Z_i = I - Q_i, \quad M_i = Z_i A^{-1}, \quad T_i = P_i A^{-1} = R_i^T A_i^{-1} R_i$$

We have $M^{-1} = M_s, \quad M^{-1} A = Z_s$

with the recurrence relation

$$Z_1 = P_1 \qquad M_1 = T_1$$

$$Z_i = Z_{i-1} + P_i(I - Z_{i-1}) \qquad M_i = M_{i-1} + T_i(I - AM_{i-1})$$



$$Z_k = \sum_{j=1}^{i} P_j Q_{j-1}, \quad Q_0 \equiv I$$

Multiplicative Schwarz Preconditioning

Solve the preconditioning multiplicative Schwarz system

$$(I - Q_s)u = (I - Q_s)A^{-1}f$$

by a Krylov subspace solver such as GMRES.

ALGORITHM **14.4** *Multiplicative Schwarz Preconditioner*

1. *Input:* v; *Output:* $z = M^{-1}v$.
2. $z := T_1 v$
3. *For* $i = 2, \ldots, s$ *Do:*
4. $\quad z := z + T_i(v - Az)$
5. *EndDo*

Only solves on the subdomains are necessary!

ALGORITHM **14.5** *Multiplicative Schwarz Preconditioned Operator*

1. *Input:* v, *Output:* $z = M^{-1}Av$.
2. $z := P_1 v$
3. *For* $i = 2, \ldots, s$ *Do*
4. $\quad z := z + P_i(v - z)$
5. *EndDo*

Additive Schwarz

The idea of additive Schwarz is similar to the parallel Schwarz alternating method. The new iteration is obtained as

$$u^{n+1} = u^n + \sum_{j=1}^{s} R_j A^{-1} R_j^T (f - Au^n).$$

Note: This method does not converge in the overlap!

We can still use additive Schwarz as an effective preconditioner though.

$$M^{-1}Au = M^{-1}f$$

$$M^{-1}A = \sum_{j=1}^{s} P_j$$

$$M^{-1} = \sum_{j=1}^{s} T_j$$

Additive Schwarz

1. For $i = 1, \ldots, s$ Do
2. Compute $\delta_i = R_i^T A_i^{-1} R_i (b - Ax)$
3. EndDo
4. $x_{new} = x + \sum_{i=1}^{s} \delta_i$

ALGORITHM **14.6** *Additive Schwarz Preconditioner*

1. *Input:* v; *Output:* $z = M^{-1} v$.
2. For $i = 1, \ldots, s$ Do:
3. Compute $z_i := T_i v$
4. EndDo
5. Compute $z := z_1 + z_2 \ldots + z_s$.

Schur Complement Preconditioners

Let the linear system be given as

$$\begin{bmatrix} B & E \\ F & C \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} f \\ g \end{bmatrix}$$

Schur complement system

$$(C - FB^{-1}E)y = g - FB^{-1}f$$

How can we efficiently solve this Schur complement system with the left-hand side

$$S = C - FB^{-1}E$$

Applying S to a vector:

1. Compute $v' = Ev$,
2. Solve $Bz = v'$
3. Compute $w = Cv - Fz$.

Need a good preconditioner for the Schur complement.

Induced Preconditioner

Restriction Operator

$$R_y \begin{bmatrix} x \\ y \end{bmatrix} = y.$$

Let P be a good preconditioner for the original matrix A.
We define the induced Schur complement preconditioner as

$$S^{-1} \approx R_y P R_y^T.$$

Let $P^{-1} = L_A U_A$ be an ILU preconditioner for A. Then the preconditioner P_S for S induced by P is given by

$$P_S^{-1} = L_S U_S, \quad L_S = R_y L_A R_y^T, \quad U_S = R_y U_A R_y^T.$$

An introduction to PETSc

- Scalable software library for the solution of algebraic systems arising from PDEs
- Developed in C. But very good Python interface available (petsc4py)
- FEniCS by default uses PETSc as linear algebra backend

This and the following slides incorporate information and graphs from the PETSc tutorial available at https://www.mcs.anl.gov/petsc/documentation/tutorials/PetscTu06.pdf

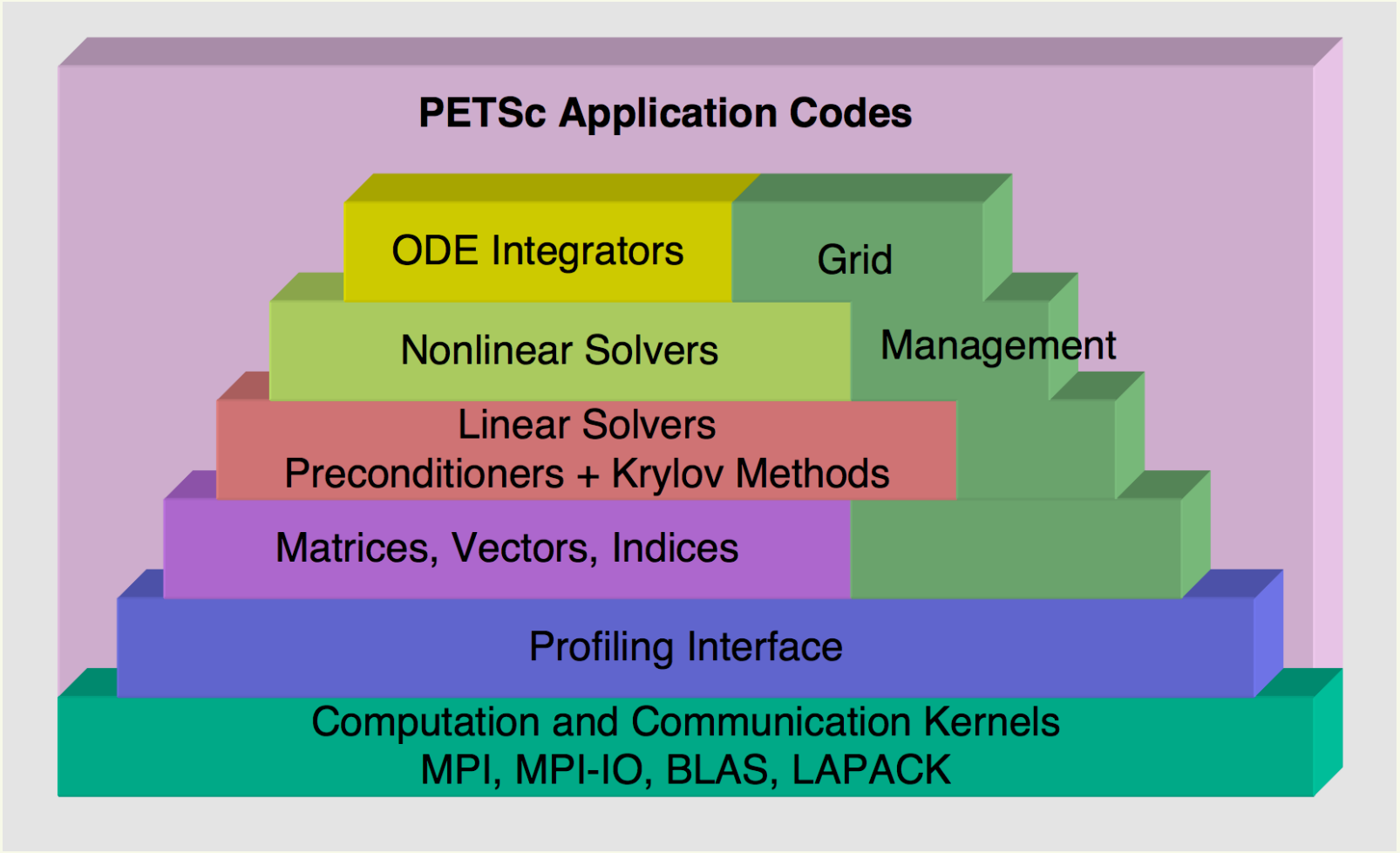
Main features

- Interfaces to a large range of numerical packages
- Parallel vector/array operations
- A range of parallel matrix formats
- A huge selection of built-in iterative solvers
- Nonlinear solvers
- Some ODE integration capabilities
- Interfaces to Algebraic Multigrid
- PETSc is used for massive Peta-Scale applications

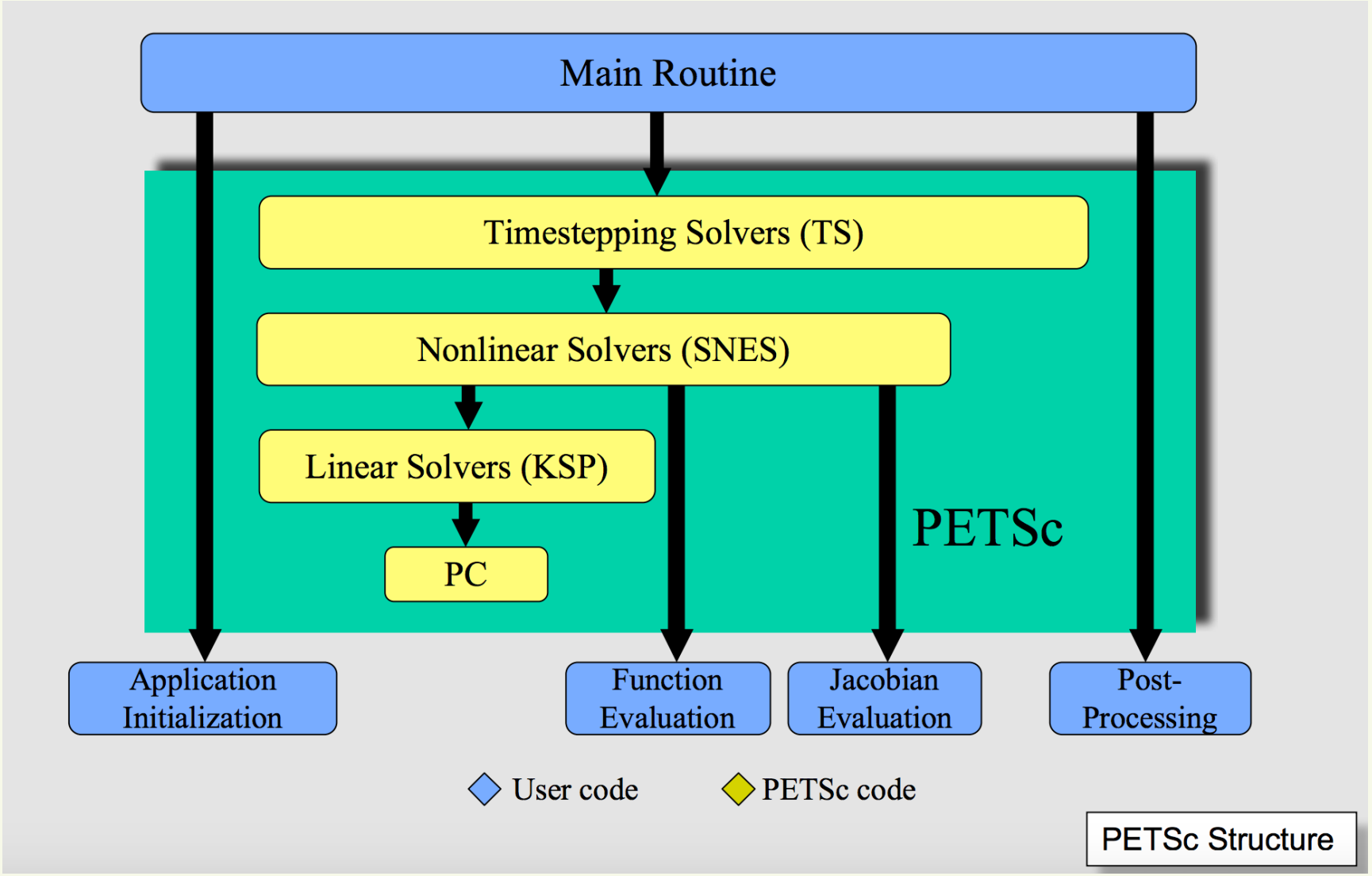
Feature Matrix

- https://www.mcs.anl.gov/petsc/documentation/linearsolvertable.html

PETSc Structure



Overview of a PDE Simulation



Vectors

- Sequential and Parallel Vectors.
- Parallel vectors are distributed across all processes.

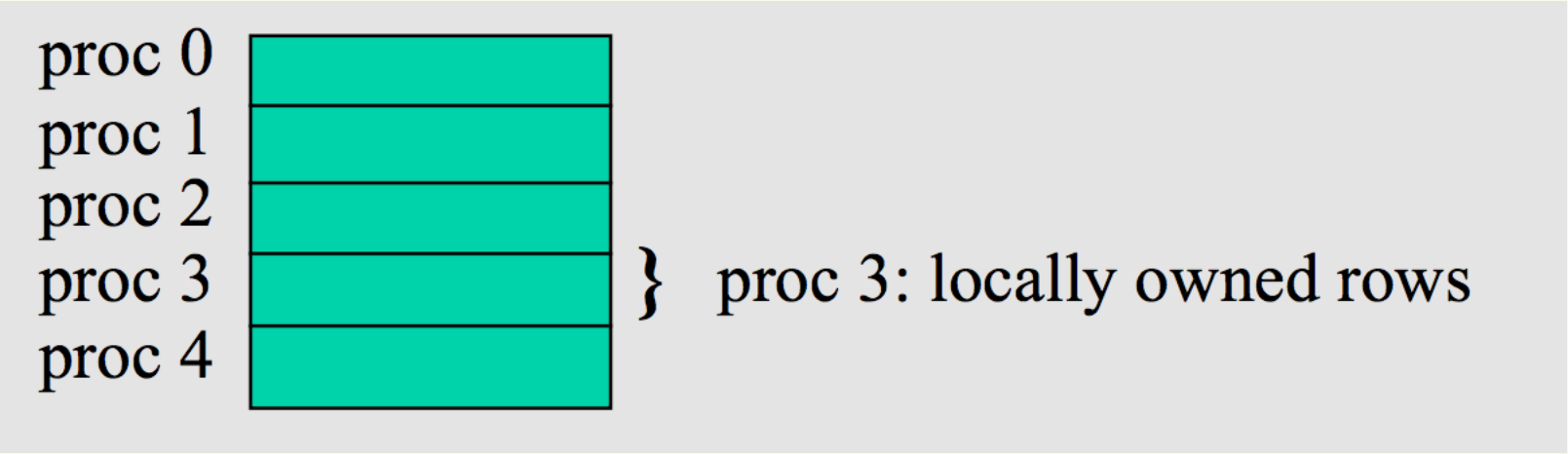
```
# Creating a vector in Python

from petsc4py import PETSc
v_mpi = PETSc.createMPI(10)
v_seq = PETSc.createSeq(10)
```

```
# Get the local indices

s, e = v.getOwnershipRange()
```

```
# Assign all local elements of
the vector the value 1

with v as a:
    a[:] = 1
```

Matrices

Rows are distributed among processes



A = PETSc.Mat().createAIJ([m, n], nnz=10)

Upper bound on nonzeros per row

Can create matrix also directly from CSR data.

To finalize assembly and communicate data use
A.assemblyBegin()
A.AssemblyEnd()

The KSP Solver Module

```
#Initialize a solver for GMRES with ILU
Preconditioner

ksp = PETSc.KSP().create()
ksp.setType('gmres')
ksp.setOperators(A)
ksp.getPC().setType('ilu')
ksp.setFromOptions()
ksp.solve(b, x)
```

Disable iterations. Only use preconditioner

```
#Initialize a solver for a direct LU solve

ksp = PETSc.KSP().create()
ksp.setType('preonly')
ksp.setOperators(A)
ksp.getPC().setType('lu')
ksp.setFromOptions()
ksp.solve(b, x)
```

Trilinos

- Sandia Based C++ Library for computational PDEs.
- More functionality than PETSc, but less stable development cycle (more disruptive changes over time)
- https://trilinos.org

Some Trilinos Packages

Epetra (Legacy distributed linear algebra module)	Tpetra (Templated distributed linear algebra module)	Thyra (Linear Algebra adapters to external codes)
Amesos(2) (Direct solver interface)	Belos (Iterative solver interface)	Intrepid (PDE Discretization Library)
Nox (Nonlinear Solvers)	ML (Multigrid Library)	PyTrilinos (Python interface to Legacy Libraries)

PyTrilinos – Direct Solver Example

```
#! /usr/bin/env python
from PyTrilinos import Amesos, Triutils, Epetra

Comm = Epetra.PyComm()
Map, A, x, b, Exact = Triutils.ReadHB("fidap035.rua", Comm)

Problem = Epetra.LinearProblem(A, x, b);
Factory = Amesos.Factory()
SolverType = "MUMPS"
Solver = Factory.Create(SolverType, Problem)
AmesosList = {
  “MaxProcs”: 2,
  "PrintStatus": True
}
Solver.SetParameters(AmesosList)
Solver.SymbolicFactorization()
Solver.NumericFactorization()
Solver.Solve()
```

All solvers can be accessed in parallel through a Python script with no effort

```
$ mpirun -np 4 python my-script.py
```

See https://trilinos.org/oldsite/packages/pytrilinos/PARA06-pytrilinos.pdf

PyTrilinos – Iterative Solver Example

```
#! /usr/bin/env python
from PyTrilinos import AztecOO, Triutils, Epetra

Comm = Epetra.PyComm()
Map, A, x, b, Exact = Triutils.ReadHB("fidap035.rua", Comm)

Solver = AztecOO.AztecOO(A, x, b)
Solver.SetAztecOption(AztecOO.AZ_solver, AztecOO.AZ_cg)
Solver.SetAztecOption(AztecOO.AZ_precond,
                      AztecOO.AZ_dom_decomp)
Solver.SetAztecOption(AztecOO.AZ_subdomain_solve,
                      AztecOO.AZ_icc)
Solver.SetAztecOption(AztecOO.AZ_graph_fill, 1)

Solver.Iterate(1550, 1e-5)
```

```
$ mpirun -np 4 python my-script.py
```

See https://trilinos.org/oldsite/packages/pytrilinos/PARA06-pytrilinos.pdf

Finite Difference Methods

- We will give an overview on solving stationary and time-domain problems using finite difference methods
- We will not go deeply into the Mathematical Theory
- Focus on tools and implementation
- Books:
  - K.W. Morton and D.F. Mayers, Numerical Solution of Partial Differential Equations
  - H. P. Langtangen and S.Linge, Finite Difference Computing with PDEs – A modern software approach (available online)

1D Model Problem

Consider the 1d oscillator

$$u'' + \omega^2 u = 0$$
$$u(0) = U_0$$
$$u'(0) = 0$$

Exact solution

$$u(t) = U_0 \cos(\omega t)$$

$$u''(t_n) + \omega^2 u(t_n) = 0,\ n = 1, \ldots, N$$
$$u''(t_n) \approx \frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2}$$
$$\frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} = -\omega^2 u^n$$
$$\frac{u^1 - u^{-1}}{2\Delta t} = 0$$

1. Discretize the domain
2. Satisfy the equation at discrete time points
3. Replace derivatives by finite differences
4. Formulate algorithm

Time-Stepping Algorithm

Solution values computed by the following iteration:

$$u^{n+1} = 2u^n - u^{n-1} - \Delta t^2 \omega^2 u^n$$

From initial conditions have

$$u^{-1} = u^1$$

and therefore

$$u^1 = 2u^0 - u^1 - \Delta t^2 \omega^2 u^0 = u^0 - \frac{1}{2}\Delta t^2 \omega^2 u^0.$$

Analysis

Assume that

$$u^n = U_0 A^n, \quad A = e^{i\tilde{\omega}\Delta t}$$

Unknown numerical wavenumber

$$\begin{aligned}
\frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} &= U_0 \frac{A^{n+1} - 2A^n + A^{n-1}}{\Delta t^2} \\
&= \frac{U_0}{\Delta t^2}\left(e^{i\tilde{\omega}(t_n+\Delta t)} - 2e^{i\tilde{\omega}t_n} + e^{i\tilde{\omega}(t_n-\Delta t)}\right) \\
&= U_0 e^{i\tilde{\omega}t_n}\frac{2}{\Delta t^2}\left(\cos(\tilde{\omega}\Delta t) - 1\right) \\
&= -U_0 e^{i\tilde{\omega}t_n}\frac{4}{\Delta t^2}\sin^2\left(\frac{\tilde{\omega}\Delta t}{2}\right)
\end{aligned}$$

We obtain

$$-U_0 e^{i\tilde{\omega}t_n}\frac{4}{\Delta t^2}\sin^2\left(\frac{\tilde{\omega}\Delta t}{2}\right) + \omega^2 U_0 e^{i\tilde{\omega}t_n} = 0.$$

Analysis...

We obtain

$$\tilde{\omega} = \pm \frac{2}{\Delta t} \sin^{-1}\left(\frac{\omega \Delta t}{2}\right).$$

Taylor series expansion gives

$$\tilde{\omega} = \omega\left(1 + \frac{1}{24}\omega^2 \Delta t^2\right) + \mathcal{O}\left(\Delta t^4\right).$$

Need to keep $\omega \Delta t$ small. In practice choose between 20 and 30 points per wavelength, even more for longer simulations.

Global error

Error at nth time step

$$e^n = \frac{1}{24} n \omega^3 \Delta t^3$$

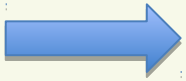
Error until time T

$$\|e^n\|_{l^2} = \frac{1}{24}\sqrt{T^3} 3 \omega^3 \Delta t^2$$

First-Order Schemes

First order formulation of oscillator

$$u' = v$$
$$v' = -\omega^2 u$$



$$\vec{u}' = A\vec{u}, \quad \vec{u} = \begin{bmatrix} u \\ v \end{bmatrix}, \quad A = \begin{bmatrix} 0 & 1 \\ -\omega^2 & 0 \end{bmatrix}$$

Euler's method

$$\vec{u}^{n+1} = (I + \Delta t A)\vec{u}^n \qquad u^{n+1} = 2u^n - u^{n-1} - \Delta t^2 \omega^2 u^{n-1}$$

Implicit Euler

$$(I - \Delta h A)\vec{u}^{n+1} = u^n \qquad \frac{u^{n+1} - 2u^n + u^{n+1}}{\Delta t^2} = -\omega^2 u^{n+1}$$

Crank-Nicholson

$$(I - \frac{\Delta t}{2} A)\vec{u}^{n+1} = (I + \frac{\Delta t}{2} A)\vec{u}^n \qquad \frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} = -\omega^2 u^n + \mathcal{O}\left(\Delta t^2\right)$$

Wave Equation

$$u_{tt} = c^2 u_{xx}$$
$$u(x,0) = U_0$$
$$u_t(x,0) = 0$$
$$u(0,t) = 0$$
$$u(L,t) = 0$$

- Use equispaced points in space with spacing Δx .
- Use equispaced points in time with spacing Δt .
- $u_i^n := u(x_i, t_n)$

Finite difference discretization

$$\frac{u_i^{n+1} - 2u_i^n + u_i^{n-1}}{\Delta t^2} = c^2 \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2}$$

Recursion

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + C^2 \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2}$$

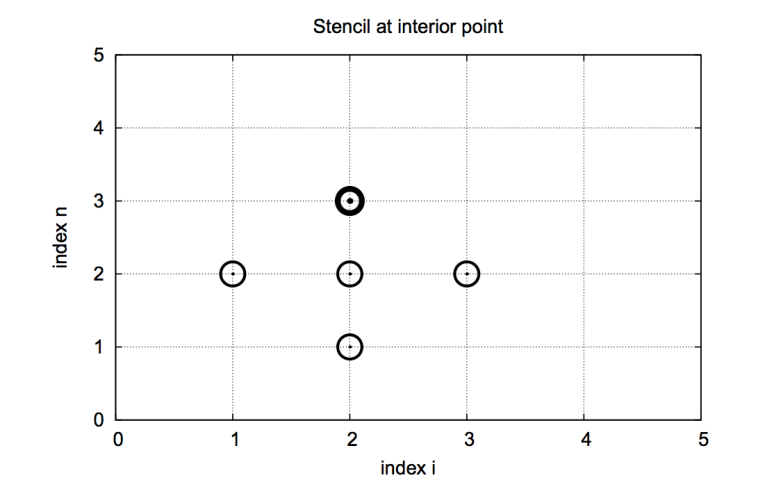
$$C = c\frac{\Delta t}{\Delta x}$$

Courant Number

Initial Conditions:

$$u_i^{-1} = u_i^1$$
$$u_i^0 = U_0(x_i)$$

Stencil

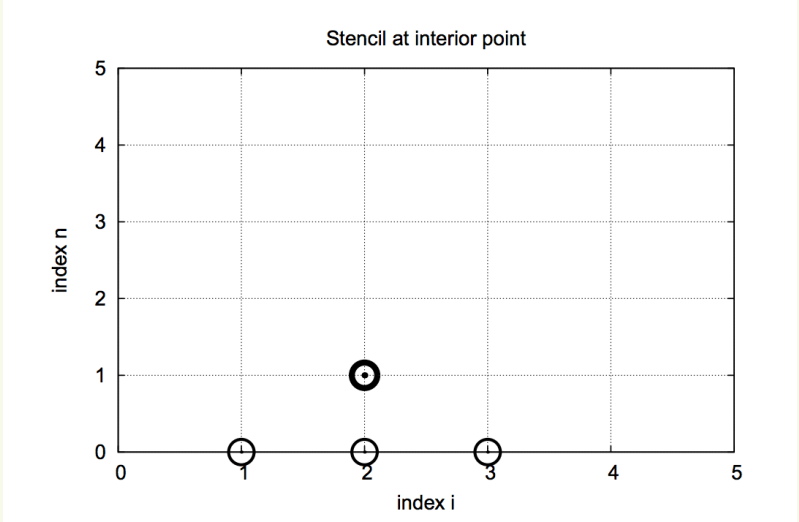


Space-Time dependency of discrete solution.

CFL Condition: The numerical domain of dependence must contain the analytic domain of dependence.

$$c\frac{\Delta t}{\Delta x} \leq 1$$

Stencil for the initial condition



Wave Equation in Multiple Dimensions

$$u_{tt} = c^2 \Delta u$$

$$\Delta u = u_{xx} + u_{yy}, \ (d = 2)$$

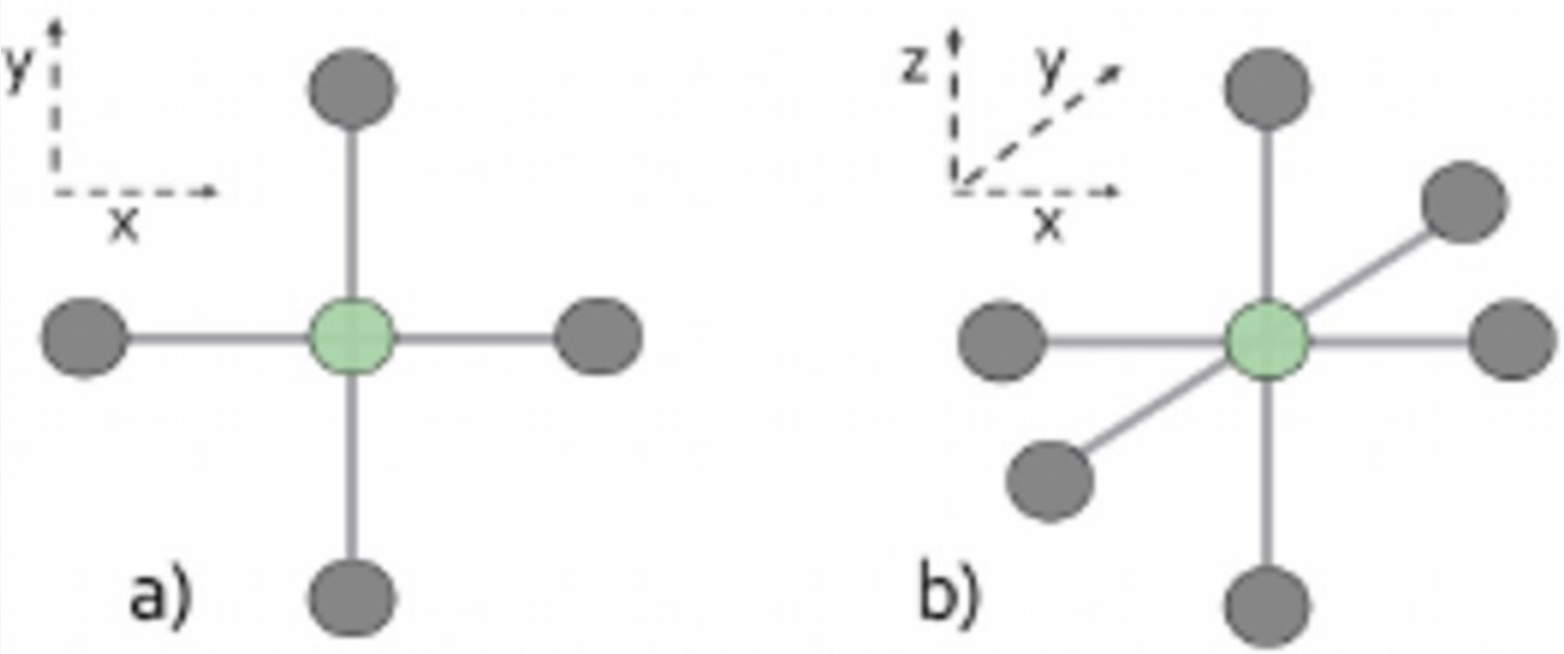
$$\Delta u = u_{xx} + u_{yy} + u_{zz}, \ (d = 3)$$

+ initial and boundary conditions.

Discretization in three space dimensions.

$$u^{n+1}_{i,j,k} = 2u^n_{i,j,k} - u^{n-1}_{i,j,k} + c^2 \Delta t^2 \left[D_x D_x u + D_y D_y u + D_z D_z u \right]^n_{i,j,k}$$

Stencils



5 point stencil

7 point stencil

D. Unat, X. Cai, S. B. Baden, Mint
: Realizing CUDA performance in 3D stencil methods with annotated C

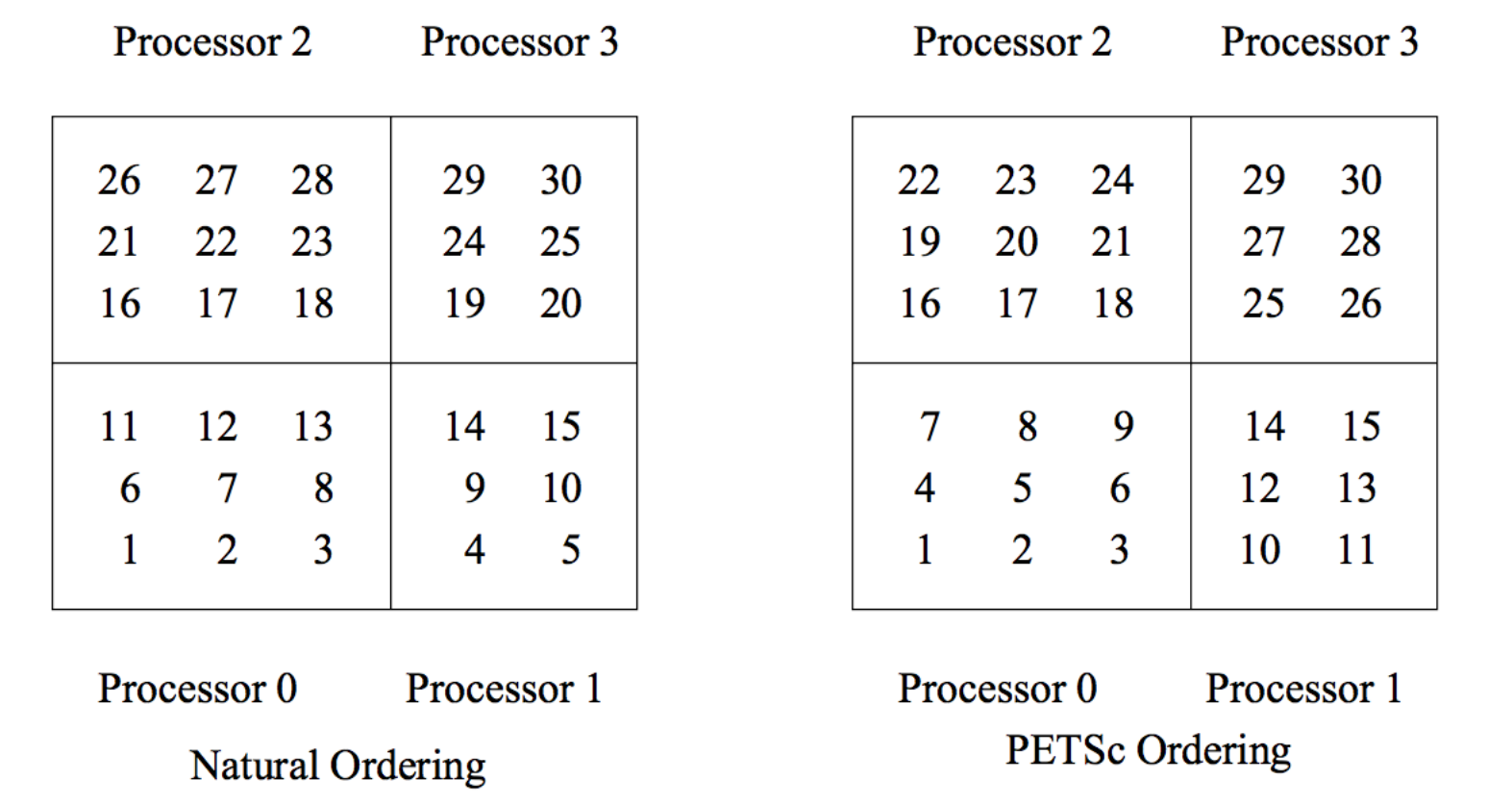
Regular grids in PETSc

- PETSc allows creation of regular grid in 1, 2 or 3 space dimensions.
- Grids are normal vectors, but support grid based indexing.
- Ghost cells from neighboring processes are automatically synchronized.

```
# Creating a global 2d grid

da  =  PETSc.DA().create([10, 10])
```

How to distribute grids in parallel?

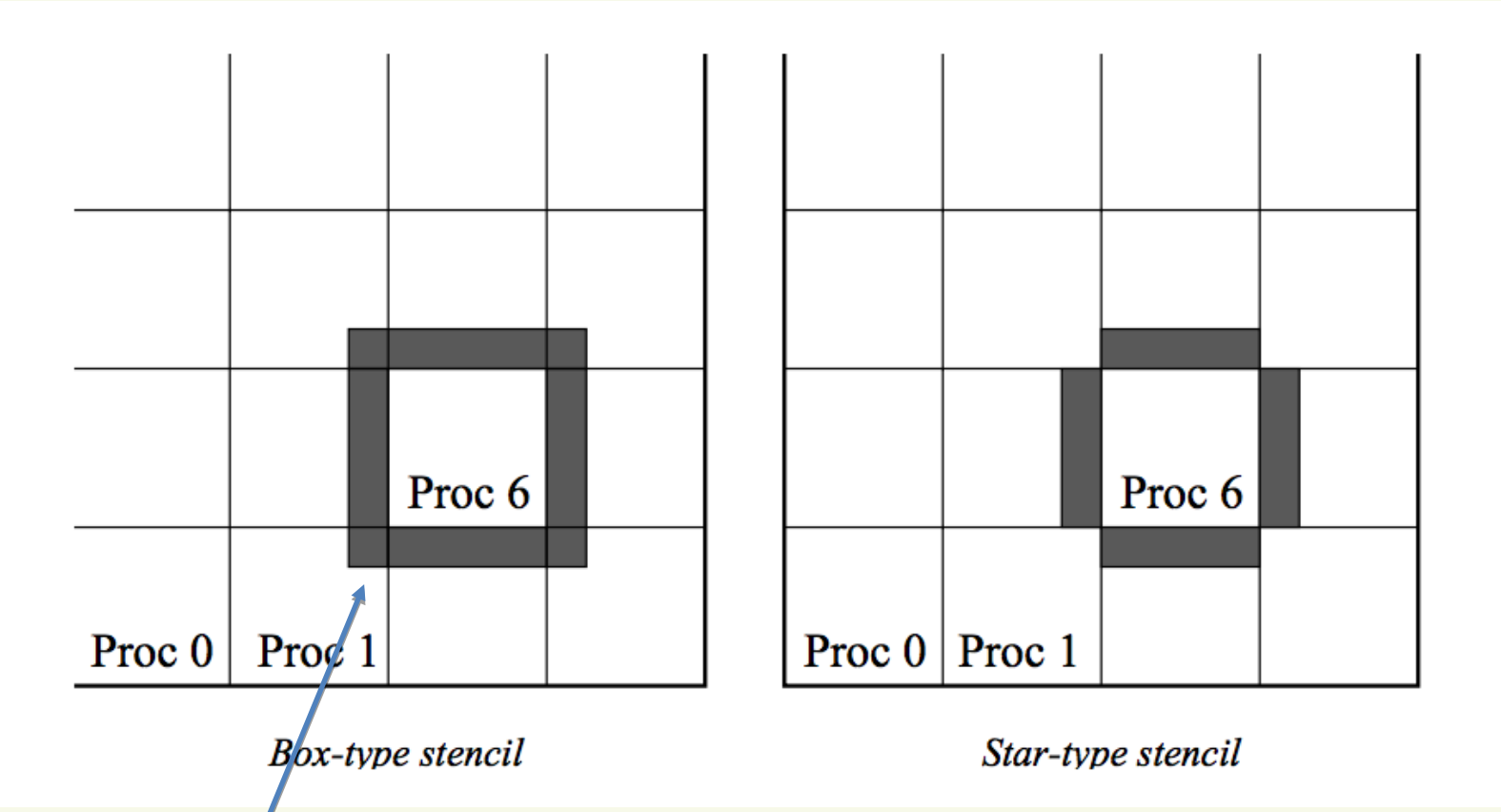
Distributed Arrays



Array distributed on a 2x 2 process grid

PETSc manual, http://www.mcs.anl.gov/petsc/petsc-current/docs/manual.pdf

PETSc Stencil Types



Ghost elements will automatically be handled

Use Star-type pencil for standard 5 or 7 point Finite Difference stencil

PETSc manual, http://www.mcs.anl.gov/petsc/petsc-current/docs/manual.pdf