High-performance computing on distributed-memory architecture

Xing Cai

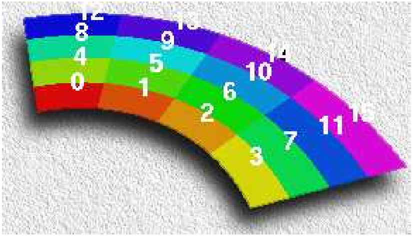
Simula Research Laboratory

Dept. of Informatics, University of Oslo

Winter School on Parallel Computing
Geilo
January 20–25, 2008

Outline

1. Overview of HPC
2. Introduction to MPI
3. Programming examples
4. High-level parallelization via DD



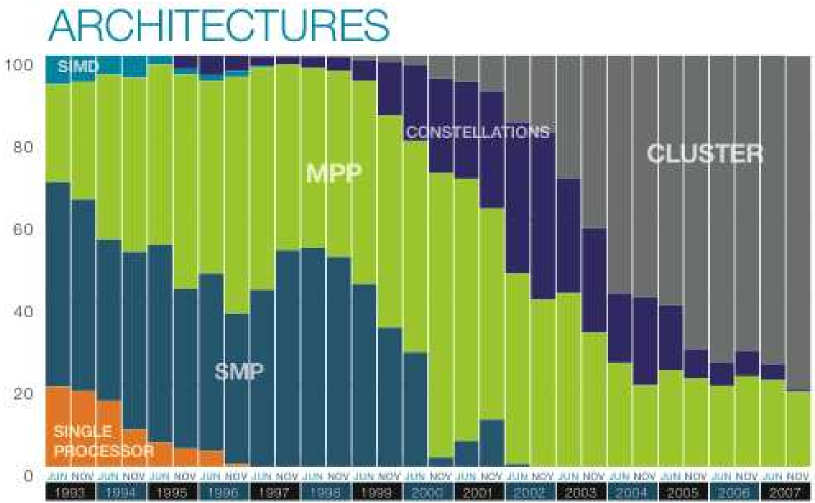
List of Topics

Motivation

- Nowadays, HPC refers to the use of parallel computers
- Memory performance is the No.1 limiting factor for scientific computing
  - size
  - speed
- Most parallel platforms have some level of distributed memory
  - distributed-memory MPP systems (tightly integrated)
  - commodity clusters
  - constellations
- Good utilization of distributed memory requires appropriate parallel algorithms and matching implementation

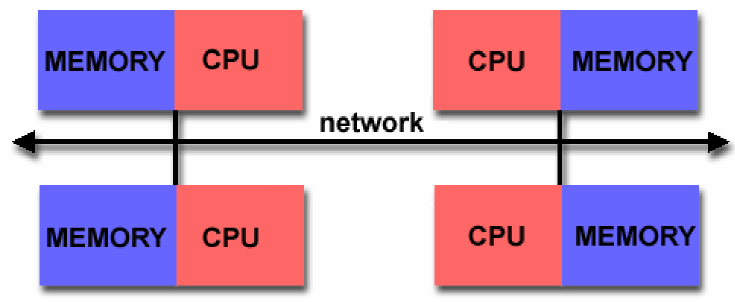
In this lecture, we will focus on distribued memory

Architecture development of Top500 list



http://www.top500.org

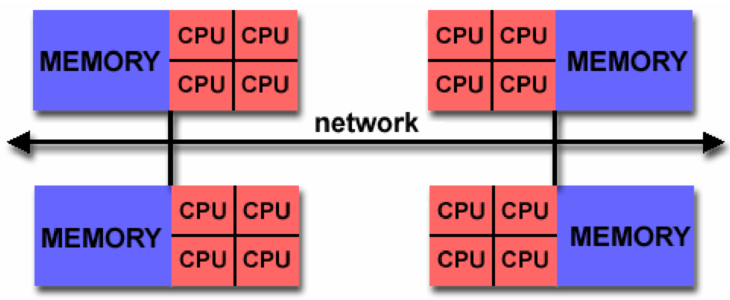
Distributed memory



A schematic view of distributed memory

Plot obtained from `https://computing.llnl.gov/tutorials/parallel_comp/`

Hybrid distributed-shared memory



A schematic view of hybrid distributed-shared memory

Plot obtained from `https://computing.llnl.gov/tutorials/parallel_comp/`

Main features of distributed memory

- Individual memory units share no physical storage
- Exchange of info is through explicit communication
- Messing passing is the de-facto programming style for distributed memory
- A programmer is often responsible for many details
  - identification of parallelism
  - design of parallel algorithm and data structure
  - breakup of tasks/data/subdomains
  - load balancing
  - insertion of communication commands

List of Topics

MPI (message passing interface)

MPI is a library standard for programming distributed memory

- MPI implementation(s) available on almost every major parallel platform (also on shared-memory machines)
- Portability, good performance & functionality
- Collaborative computing by a group of individual processes
- Each process has its own local memory
- Explicit message passing enables information exchange and collaboration between processes

More info: `http://www-unix.mcs.anl.gov/mpi/`

MPI basics

- The MPI specification is a combination of MPI-1 and MPI-2
- MPI-1 defines a collection of 120+ commands
- MPI-2 is an extension of MPI-1 to handle "difficult" issues
- MPI has language bindings for F77, C and C++
- There also exist, e.g., several MPI modules in Python (more user-friendly)
- Knowledge of entire MPI is not necessary

MPI language bindings

C binding

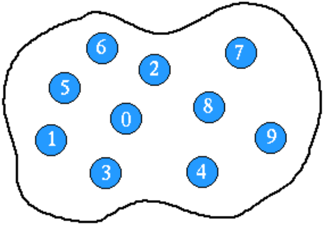
```
#include <mpi.h>

rc = MPI_Xxxxx(parameter, ... )
```

Fortran binding

```
include 'mpif.h'

CALL MPI_XXXXX(parameter,..., ierr)
```

MPI communicator

MPI_COMM_WORLD



- An MPI communicator: a "communication universe" for a group of processes
- `MPI_COMM_WORLD` – name of the default MPI communicator, i.e., the collection of all processes
- Each process in a communicator is identified by its rank
- Almost every MPI command needs to provide a communicator as input argument

MPI process rank

- Each process has a unique rank, i.e. an integer identifier, within a communicator
- The rank value is between 0 and `#procs-1`
- The rank value is used to distinguish one process from another
- Commands `MPI_Comm_size` & `MPI_Comm_rank` are very useful
- Example

```
int size, my_rank;
MPI_Comm_size (MPI_COMM_WORLD, &size);
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

if (my_rank==0) {
   ...
}
```

MPI "Hello-world" example

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

int main (int nargs, char** args)
{
  int size, my_rank;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  printf("Hello world, I've rank %d out of %d procs.\n",
         my_rank,size);
  MPI_Finalize ();
  return 0;
}
```

MPI "Hello-world" example (cont'd)

- Compilation example: `mpicc hello.c`
- Parallel execution example: `mpirun -np 4 a.out`
- Order of output from the processes is not determined, may vary from execution to execution

```
Hello world, I've rank 2 out of 4 procs.
Hello world, I've rank 1 out of 4 procs.
Hello world, I've rank 3 out of 4 procs.
Hello world, I've rank 0 out of 4 procs.
```

The mental picture of parallel execution

The same MPI program is executed concurrently on each process

Process 0	Process 1	· · ·	Process P-1

Process 0:

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

int main (int nargs, char** args)
{
  int size, my_rank;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  printf("Hello world, I've rank %d out of %d procs.\n",
         my_rank,size);
  MPI_Finalize ();
  return 0;
}
```

Process 1:

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

int main (int nargs, char** args)
{
  int size, my_rank;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  printf("Hello world, I've rank %d out of %d procs.\n",
         my_rank,size);
  MPI_Finalize ();
  return 0;
}
```

Process P-1:

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

int main (int nargs, char** args)
{
  int size, my_rank;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  printf("Hello world, I've rank %d out of %d procs.\n",
         my_rank,size);
  MPI_Finalize ();
  return 0;
}
```

MPI point-to-point communication

- Participation of two different processes
- Several different types of send and receive commands
  - Blocking/non-blocking send
  - Blocking/non-blocking receive
  - Four modes of send operations
  - Combined send/receive

Standard MPI_send/MPI_recv

- To send a message

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm);
```

- To receive a message

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm,
             MPI_Status *status);
```

An MPI message is an array of data elements "inside an envelope"

- *Data*: start address of the message buffer, counter of elements in the buffer, data type
- *Envelope*: source/destination process, message tag, communicator

Example of MPI_send/MPI_recv

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

int main (int nargs, char** args)
{
  int size, my_rank, flag;
  MPI_Status status;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

  if (my_rank>0)
    MPI_Recv (&flag, 1, MPI_INT,
              my_rank-1, 100, MPI_COMM_WORLD, &status);

  printf("Hello world, I've rank %d out of %d procs.\n",my_rank,siz

  if (my_rank<size-1)
    MPI_Send (&my_rank, 1, MPI_INT,
              my_rank+1, 100, MPI_COMM_WORLD);

  MPI_Finalize ();
  return 0;
```

Example of MPI_send/MPI_recv (cont´d)

Process 0 Process 1 · · · Process P-1

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

int main (int nargs, char** args)
{
  int size, my_rank, flag;
  MPI_Status status;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

  if (my_rank>0)
    MPI_Recv (&flag, 1, MPI_INT,
              my_rank-1, 100, MPI_COMM_WORLD, &status);

  printf("Hello world, I've rank %d out of %d procs.\n",my_rank,size);

  if (my_rank<size-1)
    MPI_Send (&my_rank, 1, MPI_INT,
              my_rank+1, 100, MPI_COMM_WORLD);

  MPI_Finalize ();
  return 0;
}
```

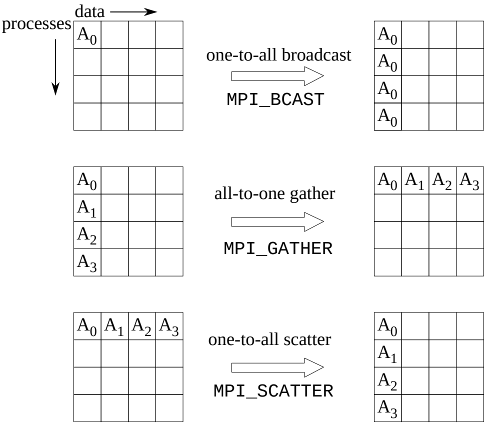
```
#include <stdio.h>
#include <mpi.h>

int main (int nargs, char** args)
{
  int size, my_rank, flag;
  MPI_Status status;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

  if (my_rank>0)
    MPI_Recv (&flag, 1, MPI_INT,
              my_rank-1, 100, MPI_COMM_WORLD, &status);

  printf("Hello world, I've rank %d out of %d procs.\n",my_rank,size);

  if (my_rank<size-1)
    MPI_Send (&my_rank, 1, MPI_INT,
              my_rank+1, 100, MPI_COMM_WORLD);

  MPI_Finalize ();
  return 0;
}
```

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

int main (int nargs, char** args)
{
  int size, my_rank, flag;
  MPI_Status status;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

  if (my_rank>0)
    MPI_Recv (&flag, 1, MPI_INT,
              my_rank-1, 100, MPI_COMM_WORLD, &status);

  printf("Hello world, I've rank %d out of %d procs.\n",my_rank,size);

  if (my_rank<size-1)
    MPI_Send (&my_rank, 1, MPI_INT,
              my_rank+1, 100, MPI_COMM_WORLD);

  MPI_Finalize ();
  return 0;
}
```

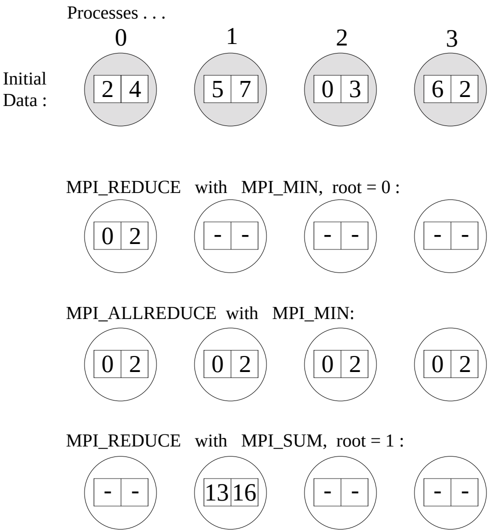
- Enforcement of ordered output by passing around a "semaphore", using MPI_send and MPI_recv
- Successful message passover requires a matching pair of MPI_send and MPI_recv

MPI collective communication

A collective operation involves *all* the processes in a communicator: (1) synchronization (2) data movement (3) collective computation



Collective communication (cont´d)



MPI example of collective communication

Inner-product between two vectors: $c = \sum_{i=1}^{n} a(i)b(i)$

```
MPI_Comm_size (MPI_COMM_WORLD, &num_procs);
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

my_start = n/num_procs*my_rank;
my_stop = n/num_procs*(my_rank+1);

my_c = 0.;
for (i=my_start; i<my_stop; i++)
  my_c = my_c + (a[i] * b[i]);

MPI_Allreduce (&my_c, &c, 1, MPI_DOUBLE,
               MPI_SUM, MPI_COMM_WORLD);
```

List of Topics

Parallel programming overview

- Decide a "breakup" of the global problem
  - functional decomposition – a set of concurrent tasks
  - data parallelism – sub-arrays, sub-loops, sub-domains
- Choose a parallel algorithm (e.g. based on modifying a serial algorithm)
- Design local data structure, if needed
- Standard serial programming plus insertion of MPI calls

Calculation of π

Want to numerically approximate the value of π

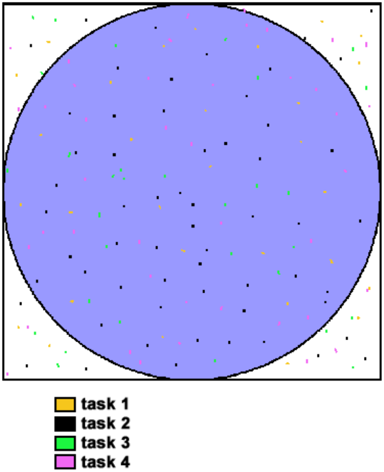
- Area of a circle: $A = \pi R^2$
- Area of the largest circle that fits into the unit square: $\frac{\pi}{4}$, because $R = \frac{1}{2}$
- Estimate of the area of the circle $\Rightarrow$ estimate of π
- How?
  - Throw a number of random points into the unit square
  - Count the percentage of points that lie in the circle by

$$\left((x-\frac{1}{2})^2 + (y-\frac{1}{2})^2\right) \leq \frac{1}{4}$$

  - The percentage is an estimate of the area of the circle
- $\pi \approx 4\,A$

Parallel calculation of π

```
num = npoints/P;
my_circle_pts = 0;

for (j=1; j<=num; j++) {
  generate random 0<=x,y<=1
  if (x,y) inside circle
    my_circle_pts += 1
}

MPI_Allreduce(&my_circle_pts,
              &total_count,
              1,MPI_INT,MPI_SUM,
              MPI_COMM_WORLD);

pi = 4.0*total_count/npoints;
```



The issue of load balancing

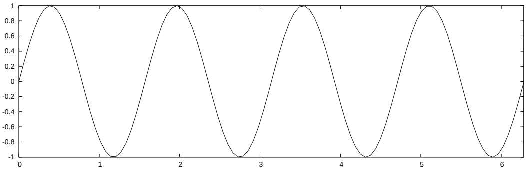
What if `npoints` is not divisible by `P`?

- Simple solution of load balancing

```
num = npoints/P;
if (my_rank < (npoints%P))
  num += 1;
```

- Load balancing is very important for performance
- Homogeneous processes should have as even disbribution of work load as possible
- (Dynamic) load balancing is nontrivial for real-world parallel computation

Example: 1D standard wave equation



Consider the 1D wave equation:

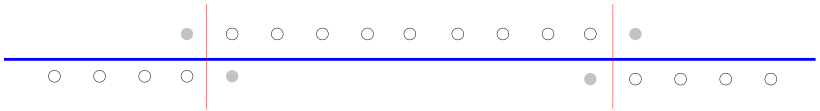
$$
\begin{aligned}
\frac{\partial^2 u}{\partial t^2} &= \gamma^2 \frac{\partial^2 u}{\partial x^2}, \quad x \in (0,1), \; t > 0, \\
u(0,t) &= U_L, \\
u(1,t) &= U_R, \\
u(x,0) &= f(x), \\
\frac{\partial}{\partial t} u(x,0) &= 0\,.
\end{aligned}
$$

Explicit FDM for 1D wave equation

Define time step Δt, spatial cell Δx, and $C = \gamma \Delta t / \Delta x$,

$$
\begin{aligned}
u_i^0 &= f(x_i), \quad i = 0, \ldots, n+1, \\
u_i^{-1} &= u_i^0 + \frac{1}{2} C^2 (u_{i+1}^0 - 2u_i^0 + u_{i-1}^0), \quad i = 1, \ldots, n \\
u_i^{k+1} &= 2u_i^k - u_i^{k-1} + C^2 (u_{i+1}^k - 2u_i^k + u_{i-1}^k), \\
&\qquad\qquad\qquad i = 1, \ldots, n, \ k \geq 0, \\
u_0^{k+1} &= U_L, \quad k \geq 0, \\
u_{n+1}^{k+1} &= U_R, \quad k \geq 0.
\end{aligned}
$$

Each processor computes on a subinterval



- The global domain is partitioned into subdomains
- Each subdomain has a set of inner points, plus 2 ghost points shared with neighboring subdomains
- First, u_i^{k+1} is updated on the inner points
- Then values on the leftmost and rightmost inner points are sent to the left and right neighbors
- Values from neighbors are received for the left and right ghost points

Multi-dimensional standard wave equation

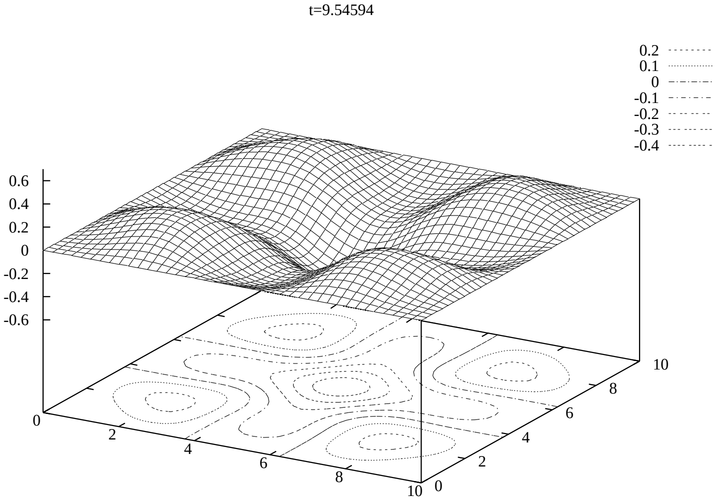
$$\frac{\partial^2 u}{\partial t^2} = \nabla \cdot \left(c^2(\mathbf{x})\nabla u\right) + f(\mathbf{x}, t)$$

- 2nd-order centered differences in time and space
- $\Rightarrow$ explicit scheme (point-wise update):

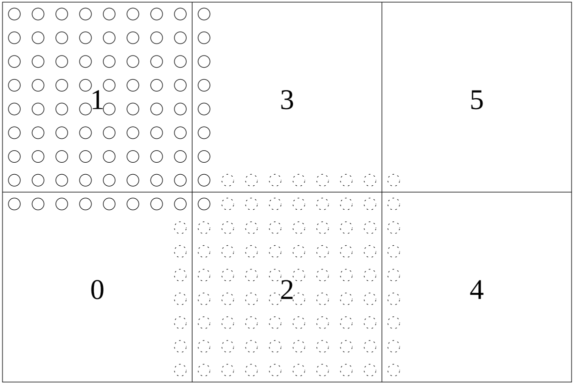
$$u_{i,j}^{k+1} = S(u_{i,j\pm1}^{k}, u_{i\pm1,j}^{k}, u_{i,j}^{k}, u_{i,j}^{k-1}, \mathbf{x}_{i,j}, t_k)$$

- Can compute all new $u_{i,j}^{k+1}$ values independently
- Parallelism arises from subdomain decomposition

Let us look at the parallel algorithm in 2D

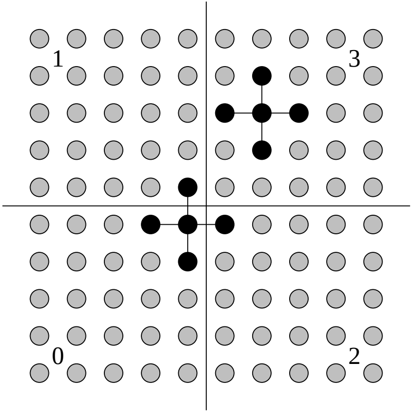


Partitioning of a rectangular 2D domain into subdomains



Each subdomain has a set of inner points, plus a set of ghost points shared with neighboring subdomains

Parallel algorithm for 2D wave equation

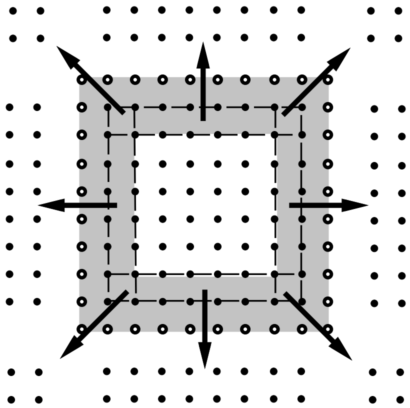


- First compute $u_{i,j}^{k+1}$ on inner points
- Then send point values to neighbors
- Then receive values at ghost points from neighbors

Python as an alternative to C for MPI programming

- MPI calls in C/Fortran are low level, easy to introduce bugs
- Python provides more high-level/Matlab-like programming
- Same logical steps as in the C code, but simpler syntax
- Python is slow, but fast enough to manage a few MPI calls

The pypar module



- Pypar (by O. Nielsen) offers a high-level interface to a subset of MPI
- Arbitrary Python objects can be sent via MPI
- Very efficient treatment of NumPy arrays
- Alternative tool: PyMPI (by P. Miller)

Python code snippets for communication

- Prepare the outgoing message:

```
upper_x_out_msg = u[nx-1,:,:]
```

  (efficient 2D array as slice reference)

- Exchange messages:

```
pypar.send(upper_x_out_msg, upper_x_neighbor_id,
           bypass=True)

pypar.receive(upper_x_neighbor_id, buffer=x_in_buffer,
              bypass=True)
```

- Extract the incoming message:

```
u[nx,:,:] = x_in_buffer
```

More detailed parallel Python code (1)

```
from RectPartitioner import partitioner  # generic!!

t = 0
while t <= tstop:
    t_old = t; t += dt

    # update all inner points (or call C/F77 for this):
    u[1:nx,1:ny] = -um2[1:nx,1:ny] + 2*um[1:nx,1:ny] +
      Cx2*(um[0:nx-1,1:ny] - 2*um[1:nx,1:ny] + um[2:nx+1,1:ny]) +
      Cy2*(um[1:nx,0:ny-1] - 2*um[1:nx,1:ny] + um[1:nx,2:ny+1]) +
      dt2*source(x[i], y[j], t_old);

    partitioner.update_internal_boundary (u)
```

More detailed parallel Python code (2)

```
def update_internal_boundary (self, solution_array):
    # communicate in the x-direction first
    if lower_x_neigh>-1:
        self.out_lower_buffers[0] = solution_array[1,:]
        pypar.send(self.out_lower_buffers[0], lower_x_neigh,
                   use_buffer=True, bypass=True)

    if upper_x_neigh>-1:
        self.in_upper_buffers[0] =
        pypar.receive(upper_x_neigh, buffer=self.in_upper_buffer[0]
                      bypass=True)
        solution_array[nx,:] = self.in_upper_buffers[0]
        self.out_upper_buffers[0] = solution_array[nx-1,:]
        pypar.send(self.out_upper_buffers[0], upper_x_neigh,
                   use_buffer=True, bypass=True)

    if lower_x_neigh>-1:
        self.in_lower_buffers[0] =
        pypar.receive(lower_x_neigh, buffer=self.in_lower_buffer[0]
                      bypass=True)
        solution_array[0,:] = self.in_lower_buffers[0]

    # communicate in the y-direction afterwards
```

Generic skeleton of PDE solvers

- Nonlinear PDEs: a series of linearized problems per time step
- A time stepping scheme for the temporal discretization
- At each time step: spatial discretization on a computational mesh $\mathcal{T}$
- Explicit schemes: point-wise update (inherent parallelism)
- Implicit schemes: need to solve *linear systems* $\mathbf{Ax} = \mathbf{b}$

Direct solvers of $\mathbf{Ax} = \mathbf{b}$ are hard to parallelize, however, many iterative Solvers are well suited for parallel computing

Jacobi iteration: slow, but easy to parallelize

$$\mathbf{A} = \{a_{ij}\}, \quad x_i^k = \left(b_i - \sum_{j<i} a_{ij} x_j^{k-1} - \sum_{j>i} a_{ij} x_j^{k-1} \right) / a_{ii}$$

- A new x_i^k value only depends on old x_i^{k-1} values
- $\Rightarrow$ The values x_i^k can be updated concurrently!
- Same parallelization strategy as for the explicit PDE solvers:
  - Each processor updates all its inner points
  - Communication needed between neighbors for updating ghost boundary points

Krylov subspace solvers: Conjugate Gradients

Suitable for symmetric and positive definite matrices
$(\mathbf{A}^T = \mathbf{A},\ \mathbf{v}^T\mathbf{A}\mathbf{v} > 0,\ \forall \mathbf{v} \neq 0)$

Initially: $r = b - Ax$, $p = r$, $\pi^0_{r,r} = (r,r)$
Iterations:

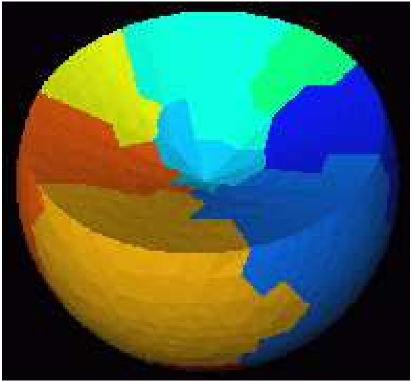
$w = Ap$	*matrix-vector product*
$M^{-1}w = w$	*solve preconditioning system*
$\pi_{p,w} = (p,w)$	*inner product*
$\xi = \pi^0_{r,r}/\pi_{p,w}$	
$x = x + \xi p$	*vector addition*
$r = r - \xi w$	*vector addition*
$\pi^1_{r,r} = (r,r)$	*inner product*
$\beta = \pi^1_{r,r}/\pi^0_{r,r}$	
$p = r + \beta p$	*vector addition*
$\pi^0_{r,r} = \pi^1_{r,r}$	

Observations

- Computational kernels of Krylov subspace solvers:
  - vector additions
  - inner products
  - matrix-vector product
- Parallelization of Krylov solvers thus needs
  - parallel vector addition
  - parallel inner product
  - parallel matrix-vector product
  - (parallel preconditioner)

Subdomain-based parallelization

Global domain $\Omega \longrightarrow \{\Omega_s\}_{s=1}^{P}$, global grid $\mathcal{T} \longrightarrow \{\mathcal{T}_s\}$, internal boundary of Ω_s: $\partial\Omega_s \backslash \partial\Omega$



Distributed matrices and vectors

- Each processor is assigned with a subdomain Ω_s and the associated subdomain mesh $\mathcal{T}_s$
- Each processor *independently* carries out spatial discretization on $\mathcal{T}_s$, giving rise to $\mathbf{A}_s$ and $\mathbf{b}_s$ (no communication needed)
- A global matrix $\mathbf{A}$ is distributed as $\{\mathbf{A}_s\}_{s=1}^{P}$
- A global vector $\mathbf{b}$ is distributed as $\{\mathbf{b}_s\}_{s=1}^{P}$
- The rows of $\mathbf{A}$ are distributed
- Each subdomain is responsible for a few rows in $\mathbf{A}$

Distributed matrices and vectors; FDM

- Subdomains arise from dividing the mesh points
- Each subdomain owns its computational points exclusively
- Layer(s) of ghost boundary points around each subdomain
- Rows of $\mathbf{A}_s$ correspond to the computational points in Ω_s, no overlap

Distributed matrices and vectors; FEM

- Denote the global finite element mesh by $\mathcal{T}$
- Mesh partitioning distributes the elements
- Each subdomain is a subset of the elements in $\mathcal{T}$
- Rows of $\mathbf{A}_s$ may overlap between neighbors
- If there's one layer of overlapping elements between neighbors, points on the internal boundaries work as ghost points (as usual)

Parallel vector addition

Global operation:

$$\mathbf{w} = \mathbf{u} + \mathbf{v}$$

Parallel implementation:

- $\mathbf{w}_s = \mathbf{u}_s + \mathbf{v}_s$ on each subdomain
- Only distributed vectors are involved
- No communication is needed

Parallel inner product

Global operation:

$$c = \mathbf{u} \cdot \mathbf{v} = \sum u_i \, v_i \quad i \in \text{ all points in } \mathcal{T}$$

Parallel implementation:

- Partial result on subdomain Ω_s:
  $c_s = \sum u_{s,i} v_{s,i} \quad i \in$ computational points in $\mathcal{T}_s$
- Global add: $c = c_1 + c_2 + \ldots + c_P$
- All-to-all communication (`MPI_Allreduce`) $\Rightarrow$ c is available on all subdomains

Parallel matrix-vector product

Global operation:

$$\mathbf{v} = \mathbf{A}\mathbf{u}$$

Parallel implementation:

- $\mathbf{v}_s = \mathbf{A}_s\mathbf{u}_s$ on Ω_s
- Ghost points in $\mathbf{v}_s$ have to ask neighbors for values
- One-to-one communication between each pair of neighboring subdomains (`MPI_Send`/`MPI_Recv`)

Some remarks

Domain partitioning $\Rightarrow$ data decomposition $\Rightarrow$ work division $\Rightarrow$ parallelism

- Linear algebra operations in an implicit PDE solver are parallelized using subdomains
- All matrices and vectors are distributed according to the subdomain partitioning $\{\Omega_s\}$
- No global matrices and vectors are stored on a single processor
- Work on Ω_s:
  - Mostly serial operations on subdomain matrices/vectors
  - Communication is needed between chunks of serial operations

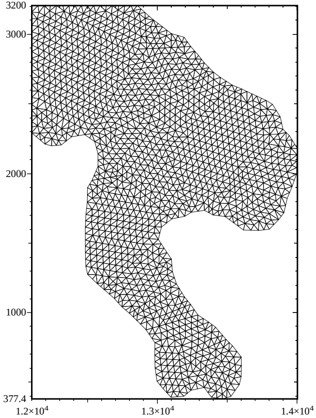
Many libraries for parallel linear algebra

Some parallel libraries for linear algebra and linear systems

- ACTS (tools collection, unified interfaces)
- ScaLAPACK (F77)
- PETSc (C)
- Trilinos (C++)
- UG (C)
- A++/P++ (C++)
- Diffpack (C++)

Finite element mesh partitioning can be easy or difficult

- When a global mesh $\mathcal{T}$ exists for Ω, domain partitioning reduces to mesh partitioning
- For *structured* global box-shaped meshes, mesh partitioning is quite easy
- For *unstructured* finite element meshes, mesh partitioning is non-trivial



Objectives for partitioning

Objectives

- Subdomains have approximately the same amount of elements and points
- Low cost of inter-subdomain communication:
  - # neighbors per subdomain is small
  - # shared points between neighbors is small

Partitioning an unstructured finite element mesh is a nontrivial load balancing problem

Overview of partitioning algorithms

- Geometric algorithms (using mesh point coordinates):
  - Recursive bisections
  - Space-filling curve approaches
- Graph-based algorithms (using connectivity info):
  - Greedy partitioning
  - Spectral partitioning
  - Multilevel partitioning
- Best choice: multilevel graph-based partitioning algorithms (Metis/ParMetis package)

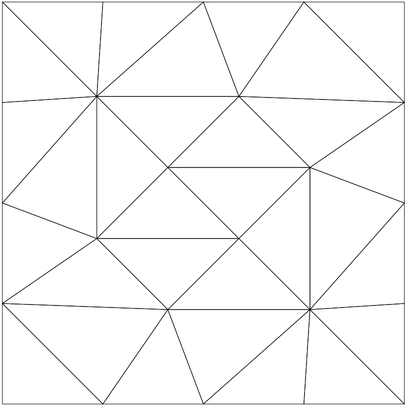
Graph-based partitioning algorithms

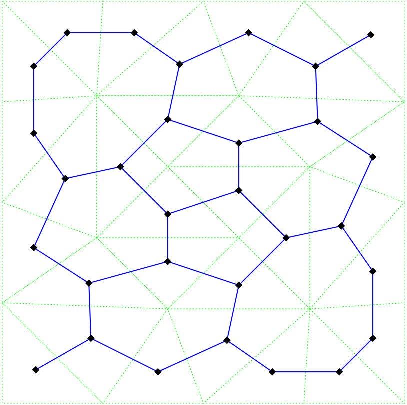
- Graph partitioning is a well-studied problem, many algorithms exist
- Mesh partitioning is similar to graph partitioning (However, not identical!)
- Easy to translate a mesh to a graph
- The graph partitioning result is projected back to the mesh to produce the subdomains

The graph partitioning problem

- A graph $G = (V, E)$ is a set of vertices and a set of edges, both with individual weights, one edge connects two vertices
- P-way partitioning of G: divide V into P subsets of vertices, V_1, V_2, ..., V_P, where
  - all subsets have (almost) the same summed vertex weights
  - summed weights of edges that stride between the subsets—**edge cut**—is minimized

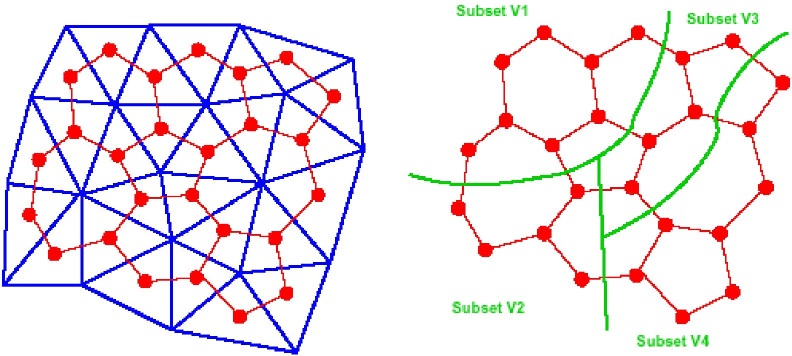
From a mesh to a graph





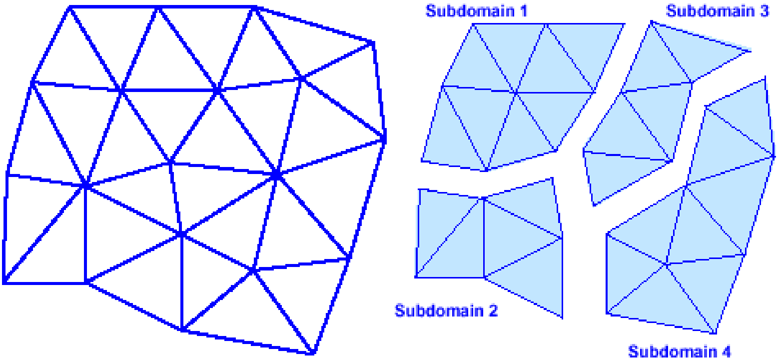
Each element becomes a vertex in the resulting graph. Whether or not an edge between two vertices depends on "neighbor-ship",

A partitioning example



A dual graph is first built on the basis of the mesh. The graph is then partitioned.

A partitioning example (cont'd)



The graph partitioning result is mapped back to the mesh and gives rise to the subdomains.

Multilevel graph partitioning

Efficient and flexible with three phases:

- Coarsening phase: a recursive process that generates a sequence of subsequently coarser graphs $G^0, G^1, \ldots G^m$
- Initial partition phase: the coarsest graph G^m is divided into P subsets
- Uncoarsening phase: the partitions of G^m is projected backward to G^0, while the partitions are adjusted for improvement along the way

Examples of public-domain software: `Jostle` & `Metis`

List of Topics

About parallel PDE solvers

- Programming a new PDE solver can be relatively easy
  - start with partitioning the global mesh $\Rightarrow$ subdomain meshes
  - parallel discrtetization $\Rightarrow$ distributed matrices/vectors
  - use parallel linear algebra libraries (PETSc, Trilinos, etc.)
- Parallelizing an existing serial PDE can be hard
  - low-level loops may not be readily parallelizable
- Special numerical components may also be hard to parallelize
  - not available in standard parallel libraries

Need a user-friendly parallelization for the latter two situations

Programming objectives

A general and flexible programming framework is desired

- extensive reuse of serial PDE software
- simple programming effort by the user
- possibility of hybrid features in different local areas

Mathematical methods based on domain decomposition

- Global solution domain is decomposed into subdomains:

$$\Omega = \cup_{s=1}^{P} \Omega_s$$

- Solving a global PDE on $\Omega \Rightarrow$ iteratively and repeatedly solving the smaller subdomain problems on Ω_s, $1 \leq s \leq P$
- The artificial condition on the internal boundary of each Ω_s is updated iteratively
- The subdomain solutions are "patched together" to give a global approximate solution

More on mathematical DD methods

- Efficient methods for solving PDEs
- Flexible treatment of local features in a global problem
- Many variants of mathematical DD methods
  - overlapping DD
  - non-overlapping DD
- Work as both stand-alone PDE solver and preconditioner
- Well suited for parallel computing

Alternating Schwarz algorithm

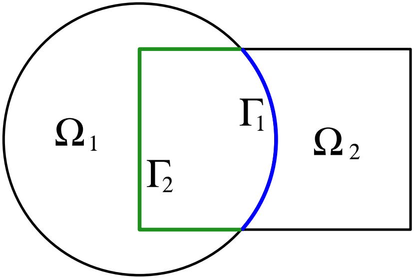
The very first DD method for

$$
\begin{aligned}
-\nabla^2 u &= f \quad \text{in } \Omega = \Omega_1 \cup \Omega_2 \\
u &= g \quad \text{on } \partial\Omega
\end{aligned}
$$

For $n = 1, 2, \ldots$ until convergence

$$
\begin{aligned}
-\nabla^2 u_1^n &= f_1 \quad \text{in } \Omega_1, \\
u_1^n &= g \quad \text{on } \partial\Omega_1 \backslash \Gamma_1, \\
u_1^n &= u_2^{n-1}|_{\Gamma_1} \quad \text{on } \Gamma_1.
\end{aligned}
$$

$$
\begin{aligned}
-\nabla^2 u_2^n &= f_2 \quad \text{in } \Omega_2, \\
u_2^n &= g \quad \text{on } \partial\Omega_2 \backslash \Gamma_2, \\
u_2^n &= u_1^n|_{\Gamma_2} \quad \text{on } \Gamma_2.
\end{aligned}
$$



Additive Schwarz method

- One particular overlapping DD method for many subdomains
- Original PDE in Ω: $L_\Omega u_\Omega = f_\Omega$ (i.e., $u_\Omega = L_\Omega^{-1} f_\Omega$)
- Additive Schwarz iterations $\Rightarrow$ concurrent work all Ω_s:

$$u_{\Omega_s}^{k+1} = L_{\Omega_s}^{-1} f_{\Omega_s}(u_\Omega^k) \text{ in } \Omega_s,$$

$$u_{\Omega_s}^{k+1} = u_\Omega^k \text{ on } \partial\Omega_s,$$

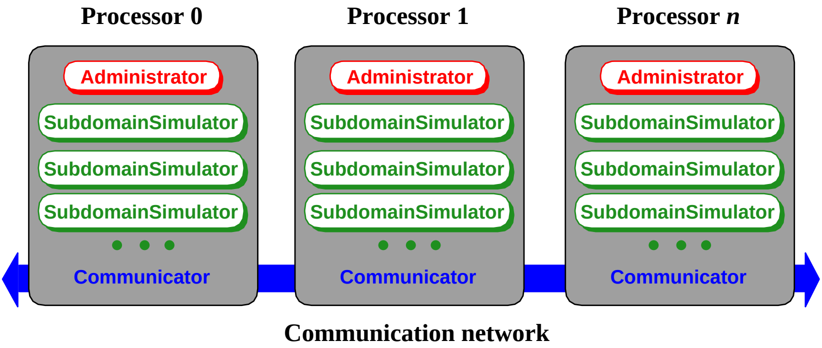
where u_Ω^k is a "global composition" of latest subdomain approximations $\{u_{\Omega_s}^k\}$

  - during each iteration a subdomain independently updates its local solution
  - exchange of local solutions between neighboring subdomains at end of each iteration

More on additive Schwarz

- Simple algorithmic structure
- Straightforward for parallelization
  - serial local discretization on Ω_s
  - serial subdomain solver on Ω_s
  - communication needed to compose the global solution
- The numerical strategy is *generic*
- Can be implemented as a parallel library
- Possibility of having different features among subdomains
  - different mathematical models
  - different numerical methods
  - different mesh types and resolutions
  - different serial code

A generic software framework



- Object-oriented programming
- `Administrator`, `SubdomainSolver` and `Communicator` are programmed as generic classes *once and for all*
- Re-usable for parallelizing many different PDE solvers
- Can hide communication details from user

Parallelizing a serial PDE solver in C++

- An existing serial PDE solver as `class MySolver`
- New implementation work task 1:
  `class My_SubdSolver : public SubdomainSolver, public MySolver`
  - Double inheritance
  - Implement the generic functions of `SubdomainSolver` by calling/extending functions of `MySolver`
  - Mostly code reuse, little new programming
- New implementation work task 2:
  `class My_Administrator : public Administrator`
  - Extend `Administrator` to handle problem-specific details
  - Mostly "cut and paste", little new programming
- Both implementation tasks are small and easy

Summary on programming parallel PDE solvers

- Subdomains give a natural way of parallelizing PDE solvers
- Discretization is embarrasingly parallel $\Rightarrow$ distributed matrices/vectors
- Linear-algebra operations are easily parallelized
- Additive Schwarz approach may be useful if
  - special parallel preconditioners are desired, and/or
  - high-level parallelization of legacy PDE code is desired, and/or
  - a parallel hybrid PDE solver is desired
- Most of the parallelization work is generic
- Languages like C++ and Python help to produce user-friendly parallel libraries

Concluding remarks

- Distributed memory is present in most parallel systems
- Message passing is used to program distributed memory
  - full user control
  - good performance
  - however many low-level details
- Use existing parallel numerical libraries if possible
- High-level parallelization is achievable
- Hybrid parallelism is possible by using SMP/Multicore for each subdomain