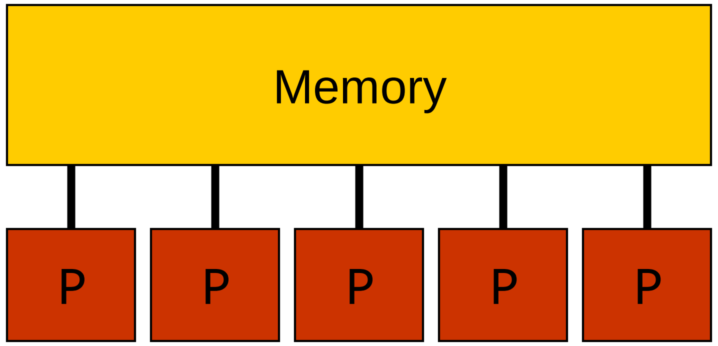
Shared Memory Programming

(I)Shared Memory Programming Model

Shared memory programming with threads.

- Global memory

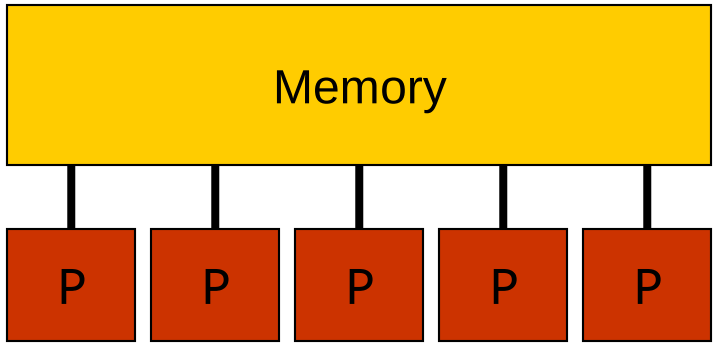


- Parallelization by threads

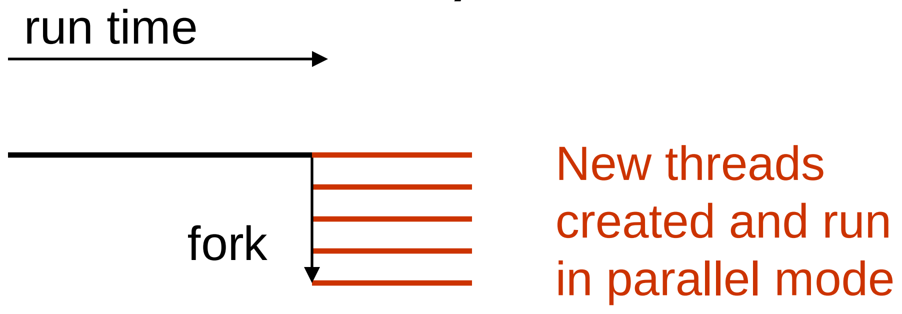
Master thread runs in serial mode

Shared memory parallel programming with threads

- Global memory

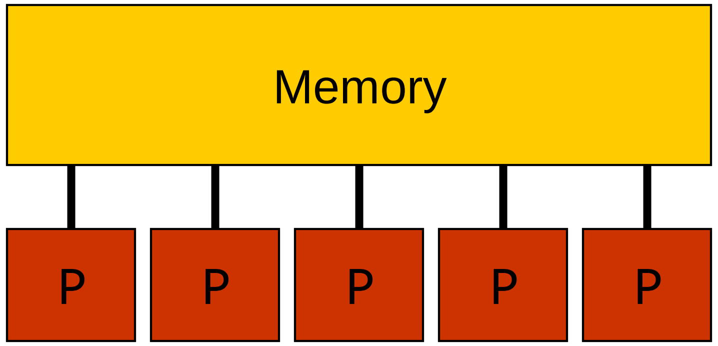


- Parallelization by threads

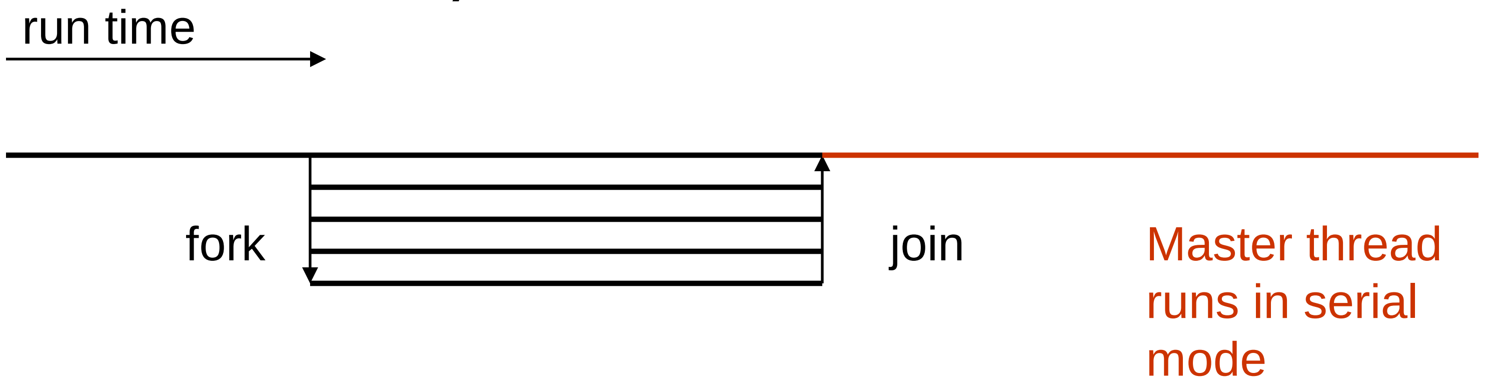


Shared memory parallel programming with threads

- Global memory

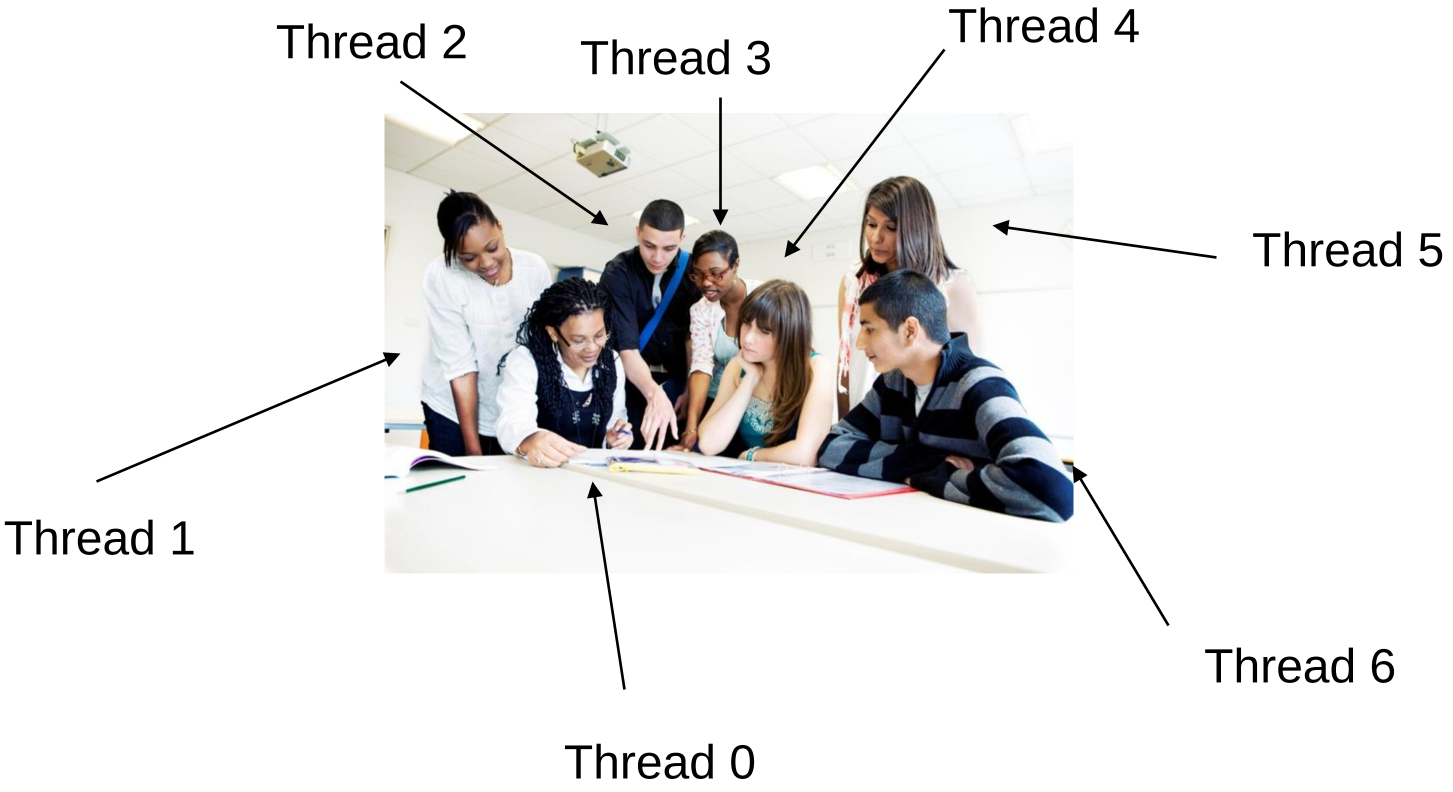


- Parallelization by threads



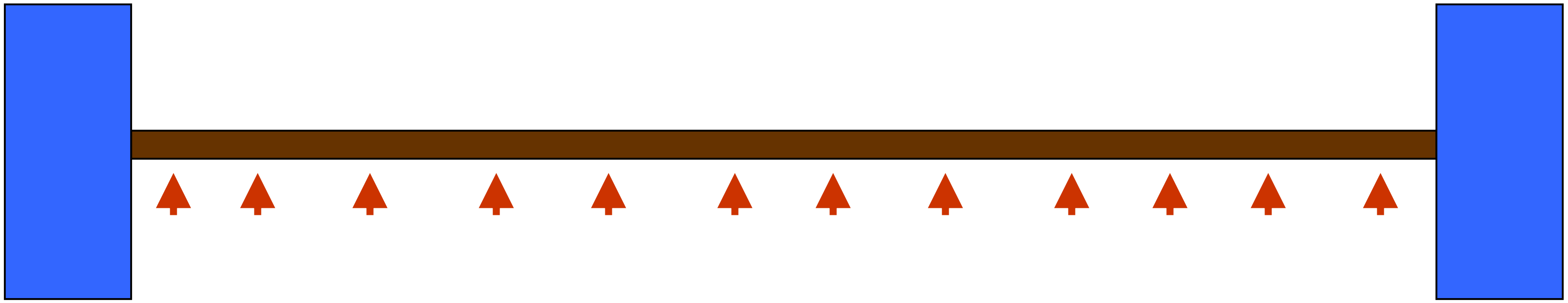
Shared memory parallel programming with threads

Multiple workers at the same shared white board or workspace.



(II) Heat Equation by Finite Differences

Simple example problem: temperature profile in a rod

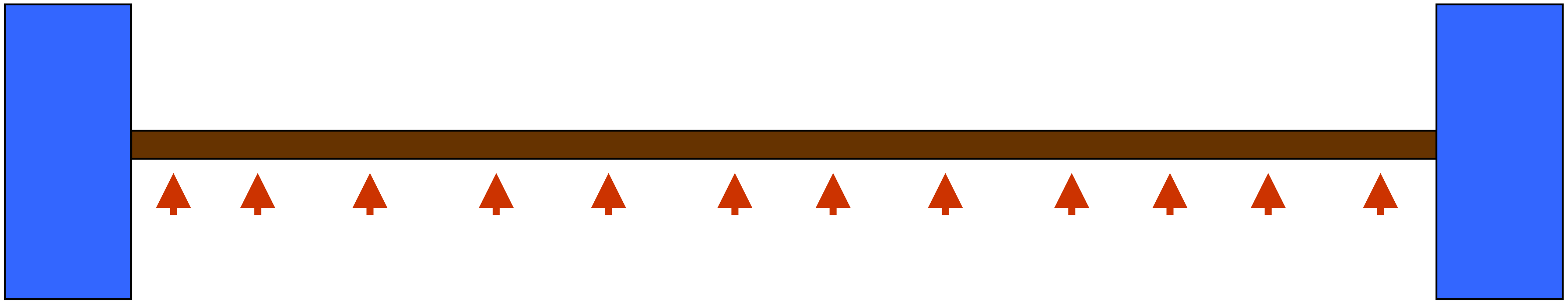


- Rod heated uniformly by burners, ends held at constant temperature.
- Idealized equations

$$\frac{\partial u(x,t)}{\partial t} - \frac{\partial^2 u(x,t)}{\partial x^2} = 1; \qquad 0 < x < 1; \qquad 0 < t < 2$$

$$u(0,t) = u(1,t) = 0; \qquad u(x,0) = 0$$

Simple example problem: temperature profile in a rod [Steady State]

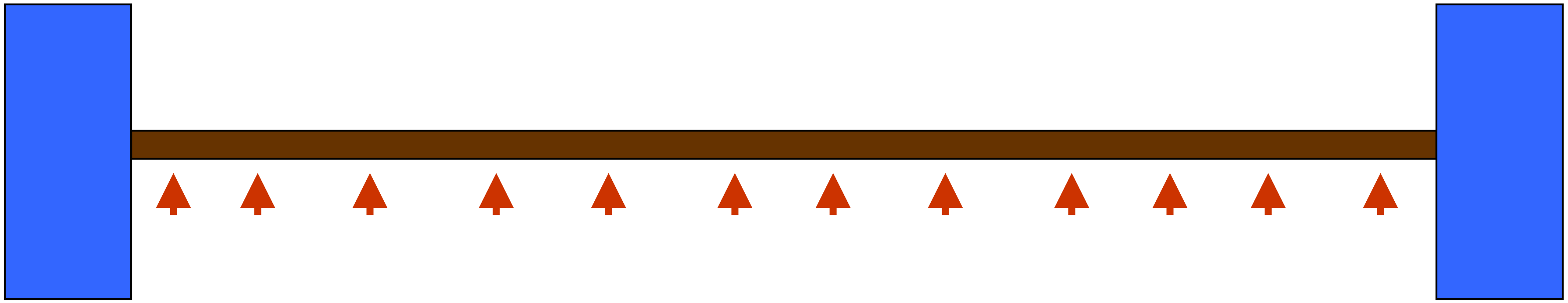


- Rod heated uniformly by burners, ends held at constant temperature.
- Idealized equations

$$\overset{0}{\cancel{\frac{\partial u(x,t)}{\partial t}}} - \frac{\partial^2 u(x,t)}{\partial x^2} = 1; \qquad 0 < x < 1; \qquad 0 < t < 2$$

$$u(0,t) = u(1,t) = 0; \qquad u(x,0) = 0$$

Simple example problem: temperature profile in a rod [Steady State]

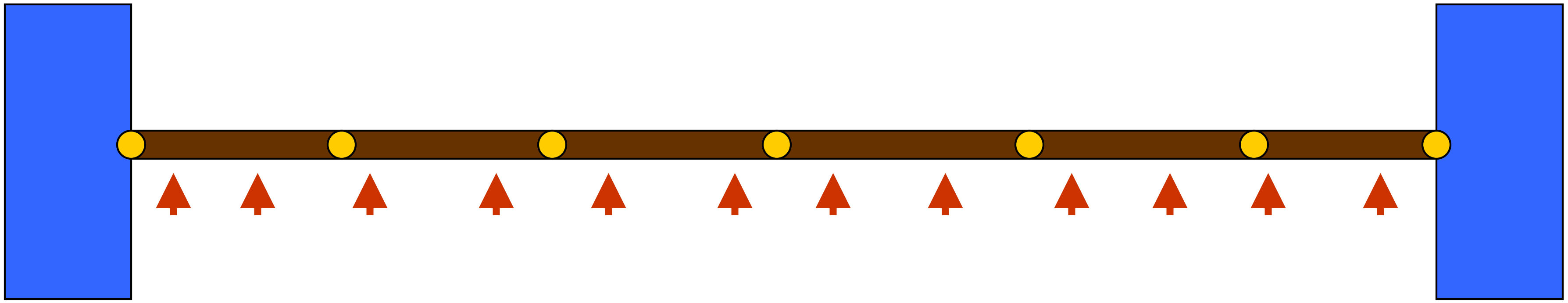


- Rod heated uniformly by burners, ends held at constant temperature.
- Idealized equations

$$-\frac{\partial^2 U(x)}{\partial x^2} = 1; \qquad 0 < x < 1;$$

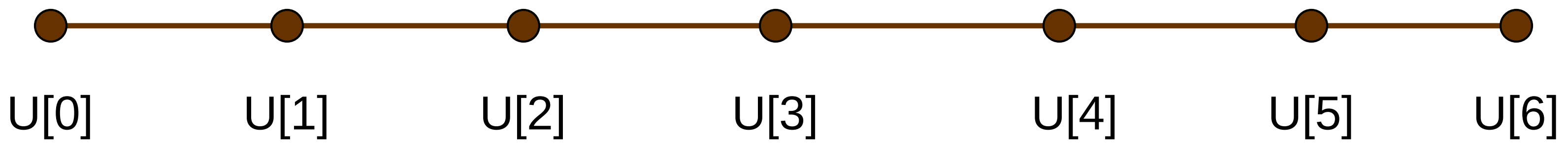
$$U(0) = U(1) = 0;$$

Simple example problem: temperature profile in a rod [Steady State]



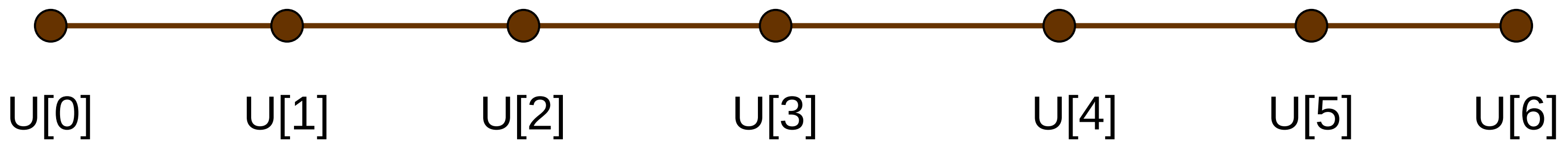
- Rather than looking for a solution as a continuous function, like sin(x), compute solution U at a discrete set of spatial points.

Simple example problem: temperature profile in a rod [Steady State]



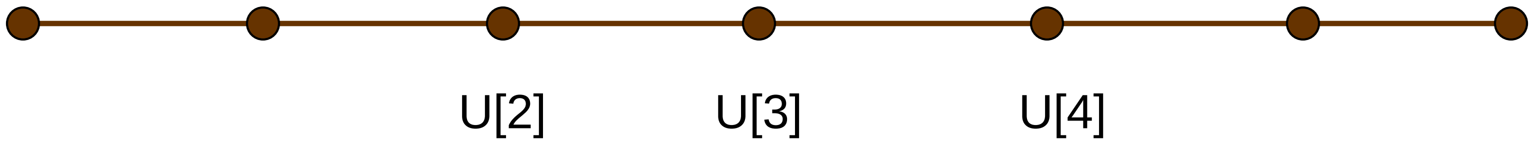
- Rather than looking for a solution as a continuous function, like sin(x), compute solution U at a discrete set of spatial points.

Discretizing: going from the continuum (PDE) to the discrete (algebraic)



- Discretizing the problem also involves converting the derivatives into algebraic difference equations

Discretizing: going from the continuum (PDE) to the discrete (algebraic)



- Taylor's theorem

$$U(x+h) = U(x) + U'(x)h + \frac{1}{2}U''(x)h^2 + \frac{1}{6}U'''(x)h^3 + O(h^4)$$

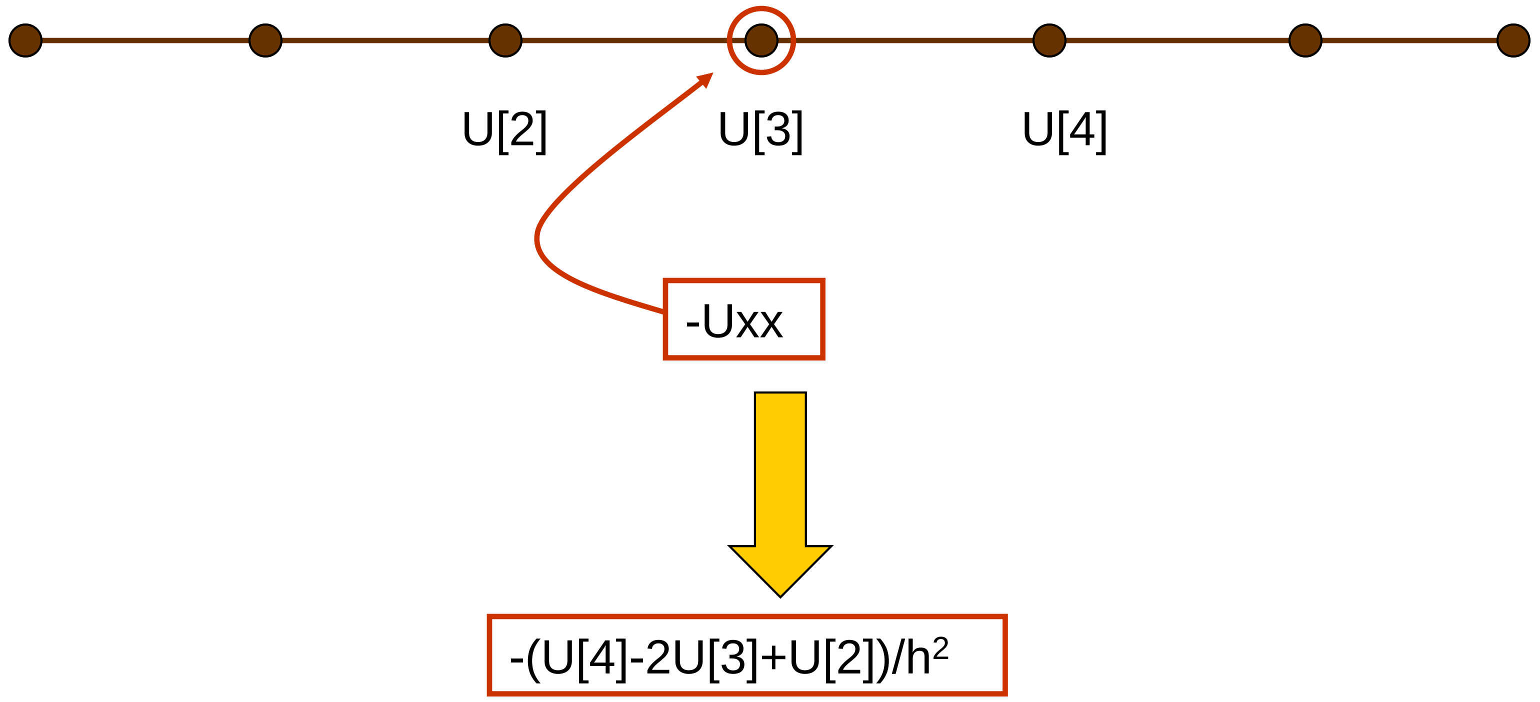
$$U(x-h) = U(x) - U'(x)h + \frac{1}{2}U''(x)h^2 - \frac{1}{6}U'''(x)h^3 + O(h^4)$$

- Adding and solving for second derivative

$$U(x+h) + U(x-h) = 2U(x) + U''(x)h^2 + O(h^4)$$

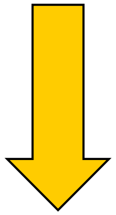
$$U''(x) = \frac{1}{h^2}\left[U(x+h) - 2U(x) + U(x-h)\right] + O(h^2)$$

Discretizing: going from the continuum (PDE) to the discrete (algebraic)

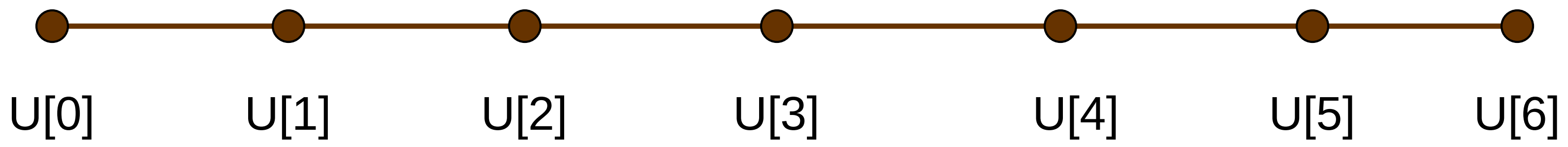


Forward Difference method, replace derivatives by differences

$$-\frac{\partial^2 U(x)}{\partial x^2} = 1;$$



$$-\frac{U\,[i-1] - 2U\,[i] + U\,[i+1]}{h^2} = 1$$



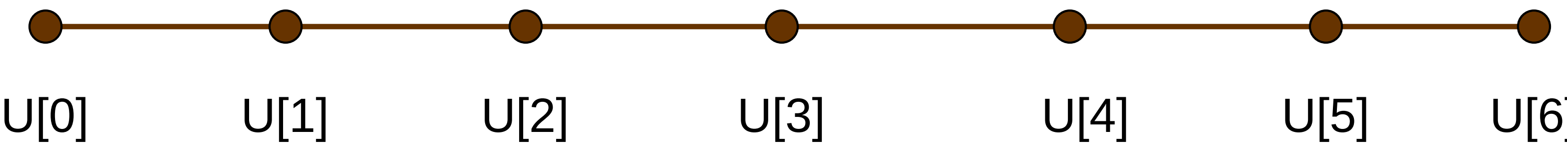
Forward Difference method, replace derivatives by differences

$$-\frac{\partial^2 U(x)}{\partial x^2} = 1;$$

$$-\frac{U[i-1] - 2U[i] + U[i+1]}{h^2} = 1$$

Solve for U[i]

$$U[i] = \frac{1}{2}\left[h^2 + U[i-1] + U[i+1]\right]$$

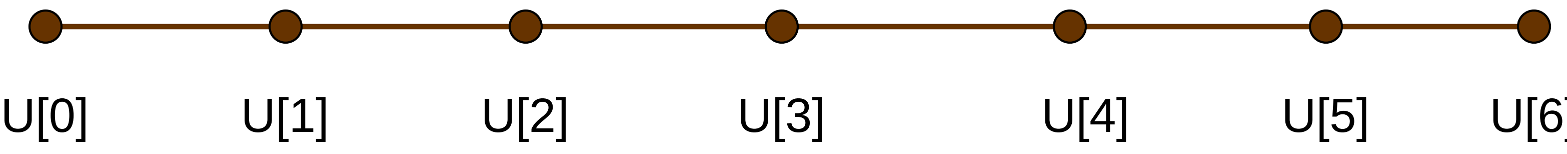


Jacobi method, guess at a solution, and update via finite difference formula

$$U[i] = \frac{1}{2}\left[h^2 + U[i-1] + U[i+1]\right]$$

$$\Downarrow$$

$$Unew[i] = \frac{1}{2}\left[h^2 + Uold[i-1] + Uold[i+1]\right]$$



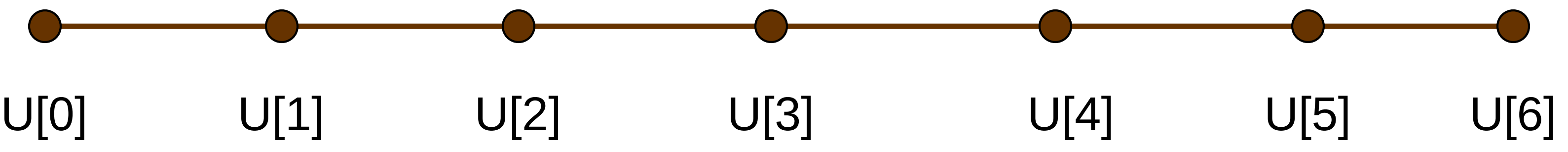
Jacobi method, guess at a solution, and update via finite difference formula

$U[i]=$

Keep going …
- Replace Uold by Unew
- Update again (called an iteration or step)

If this process converges (U stops changing), we'll have …

$$Unew[i]=\frac{1}{2}\left[h^2+Uold[i-1]+Uold[i+1]\right]$$



Jacobi method, guess at a solution, and update via finite difference formula

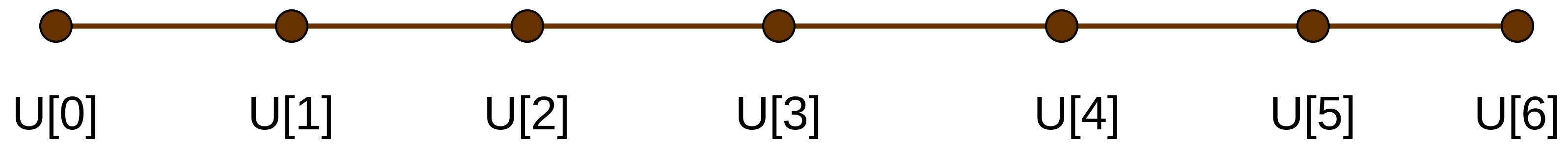
$U[i] =$

Keep going …

- Replace Uold by Unew
- Update again

If this process converges (U stops changing), we'll have a solution

$$U[i] = \frac{1}{2}\left[h^2 + U[i-1] + U[i+1]\right]$$



Serial Code Fragment: Jacobi

```
double          u_new[N+2], u_old[N+2];

u_old[0]=0.0, u_old[N+1]=0.0;
u_new[0]=0.0, u_new[N+1]=0.0;

h=1.0/(N+1),h2=h*h;

for (step=0; step<num_steps; step++) {
 for (i=1; i<=N; i++) }
  /* compute new temp from formula */
  u_new[i]=0.5*(h2+u_old[i+1]+u_old[i-1]);
 }
 for (i=1; i<=N;  i++) {
  u_old[i]=u_new[i];
 }
}
```

(III) Heat Equation by Finite Differences + OpenMP

OpenMP is one standard for shared memory programming.

- Write most of code in C, C++, FORTRAN but add OpenMP compiler directives to control threads.
- Allows incremental parallelization
  - Profile serial code
  - Mark for parallelization those loops that take the most time
- Still have to *think* to make sure marked loops can be executed in parallel.

Jacobi Open MP

```
# include <omp.h>
double         u_new[N+2], u_old[N+2];
u_old[0]=0.0, u_old[N+1]=0.0;
u_new[0]=0.0, u_new[N+1]=0.0;
h=1.0/(N+1),h2=h*h;

for (step=0; step<num_steps; step++) {
#pragma omp parallel for
 for (i=1; i<=N; i++) }
  /* compute new temp from formula */
       u_new[i]=0.5*(h2+u_old[i+1]+u_old[i-1]);
 }
#pragma omp parallel for
 for (i=1; i<=N;  i++) {
  u_old[i]=u_new[i];
 }
}
```

Jacobi Open MP

```
# include <omp.h>
double        u_new[N+2], u_old[N+2];
u_old[0]=0.0, u_old[N+1]=0.0;
u_new[0]=0.0, u_new[N+
h=1.0/(N+1),h2=h*h;

for (step=0; step<num_steps; step++) {
#pragma omp parallel for
 for (i=1; i<=N; i++) }
  /* compute new temp from formula */
       u_new[i]=0.5*(h2+u_old[i+1]+u_old[i-1]);
}
#pragma omp parallel for
 for (i=1; i<=N;  i++) {
  u_old[i]=u_new[i];
 }
}
```

Instructs the compiler to execute the immediately following for loop in parallel.

OpenMP: parallel for

- To allow compiler parallelize the loop, control clause must have canonical shape.

```
for(i = start; i     < end;          i++)
                     >               ++i
                     <=              i--
                     >=              --i
                                     i = i - inc
                                     i -= inc
                                     i = i + inc

                                     i += inc
```

OpenMP: parallel for

- To allow compiler parallelize the loop, loop body can’t contain statements that allow loop to exit prematurely.
  - No `break`
  - No `return`
  - No `exit`
  - No `goto` statements to labels outside the loop

OpenMP: parallel for

- Only mark as parallel a loop if there are no dependencies.
- The results should not depend on the order in which the loop is executed

OpenMP: parallel for

- Only mark as parallel a loop if there are no dependencies.
- The results should not depend on the order in which the loop is executed

```
for (i = 0; i < n; i++)
{
        C[i] = A[i] + B[i];
}
```

OK to parallelize
Loop index k's computation in independent of all others

OpenMP: parallel for

- Only mark as parallel a loop if there are no dependencies.
- The results should not depend on the order in which the loop is executed

```
for (i = 1; i < n; i++)
{
        c[i] = A[i] + c[i-1];
}
```

Not OK to parallelize
Loop index k's computation in dependent on result from k-1

Internally the loop iterations are divided among threads

```
#pragma omp parallel for
for (i = 0; i < n; i++)
{
        C[i] = A[i] + B[i];
}
```

OpenMP compiler will generate code like below:

```
int this_thread = omp_get_thread_num()
int num_threads = omp_get_num_threads();
int my_start = (this_thread  ) * n / num_threads;
int my_end   = (this_thread+1) * n / num_threads;
for (i = my_start; i < my_end; i++)

{
      C[i] = A[i] + B[i];
}
```

OpenMP: shared and private variables

- A ***shared variable*** has the same address in every thread (there's only one version)
  - All threads can access shared variables
- A ***private variable*** has a different address in each thread (there's a version for each thread)
  - A thread cannot access a private variable of another thread
- Default for the **`parallel for`** pragma
  - All variables are shared except for the loop index which is private.

Jacobi Open MP

```
# include <omp.h>
double       u_new[N+2], u_old[N+2];
u_old[0]=0.0, u_old[N+1]=0.0;
u_new[0]=0.0, u_new[N+
h=1.0/(N+1),h2=h*h;

for (step=0; step<num_steps; step++) {
#pragma omp parallel for
 for (i=1; i<=N; i++) }
  /* compute new temp from formula */
       u_new[i]=0.5*(h2+u_old[i+1]+u_old[i-1]);
 }
#pragma omp parallel for
 for (i=1; i<=N;  i++) {
  u_old[i]=u_new[i];
 }
}
```

Each thread will get its own, private “i” variable

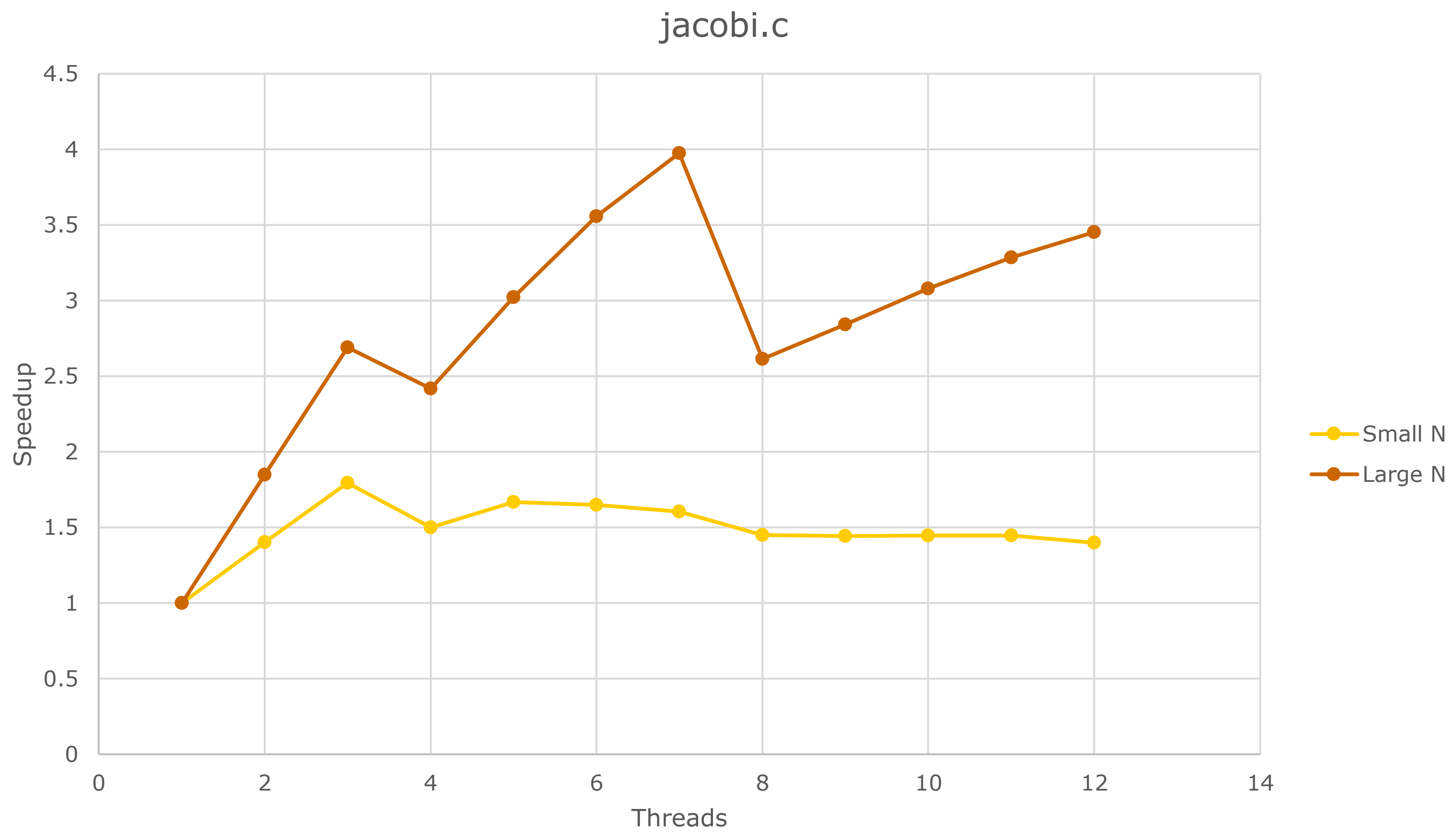
OpenMP: how many threads to use?

- void omp_set_num_threads (int t)
  - Sets the number of threads to be used in parallel sections.
  - Can also be controlled by the environment variable OMP_NUM_THREADS

Runtime (sec) for Jacobi OpenMP code on my quad core desktop

OMP num threads	Small N=16000	Large N=160000
1	16.16	182.88
2	11.54	98.92
3	9.01	67.99
4	10.77	75.63
5	9.69	60.49
6	9.80	51.42
7	10.07	46.00
8	11.16	69.97
9	11.2	64.36
10	11.18	59.40
11	11.17	55.67
12	11.55	52.99

Runtime (sec) for Jacobi OpenMP code on my quad core desktop



(IV) OpenMP

Codes on CANVAS

Parallel pragma alone

- The parallel pragma starts a parallel region.
- This starts (forks) a team of threads all of which execute the region. Each thread assigned an id: 0,1, …, num_threads-1
- Implicit barrier at end of parallel region, threads wait until all finish
- After the region, threads join back to one.

```
#pragma omp parallel
{
        printf(“Hello!\n”);
}
```

Output:
One print from each thread