cores: 4

logical processors: 8

private: variable inside and outside parallel environment are different
=> inside parallel environment, need to specify value of this variable

- any variable created inside ll region is automatically private

first private: in parallel environment, variable is initialized with value outside of environment but result will not copy outside of environment

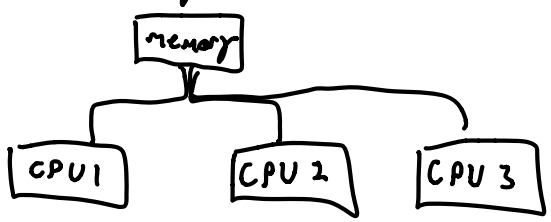
last private: variable must be initialized inside parallel environment result will be copied to value outside parallel env.

shared: shared outside and inside of parallel environment

Notes on parallel programming and open MP

parallel architectures and programming models

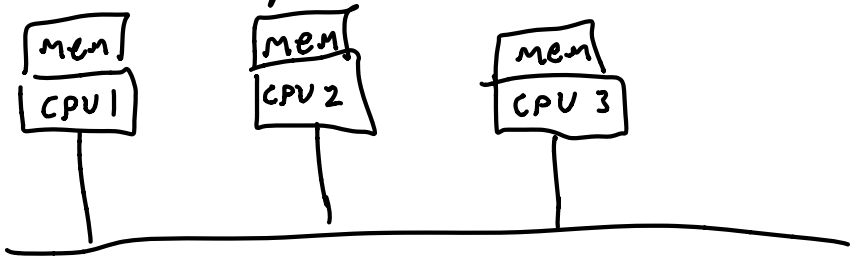
shared memory



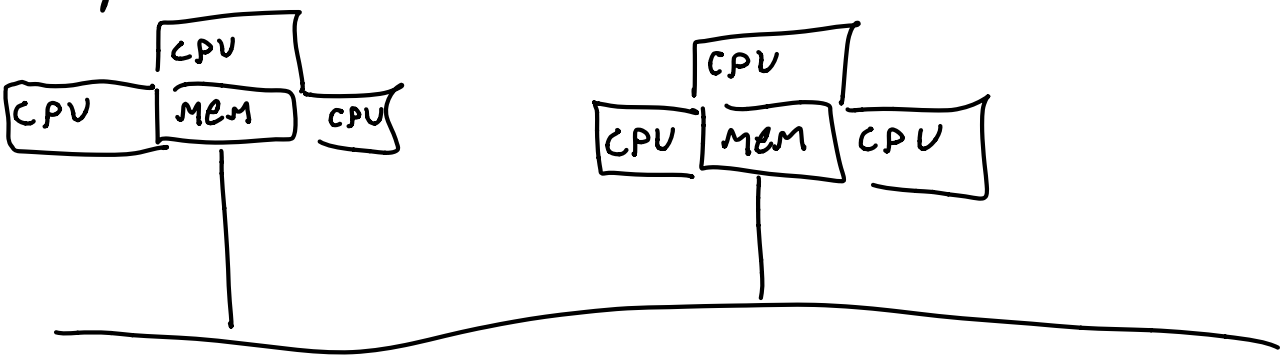
issue: not scalable, limit to # of CPUs

=> this is what openmp works with

distributed memory

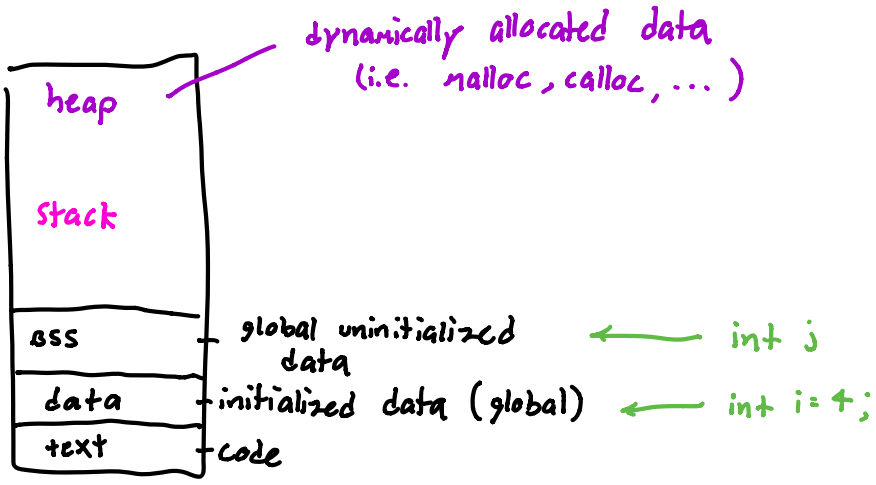


hybrid



Thread info

when you run a code, it has access to some memory that is divided into regions




```
function main
{
  int x;
   =
   =
   g(x);
   =
}

g(int x) {
   =
   int y
   =
}
```

program counter stores the line we are currently computing... but when g(x) is called, counter must go down to bottom line and the location of counter before it started searching for g is saved in STACK

STACK also stores local variables i.e. variables created in functions

multiple threads and multitasking

each thread computes independent data

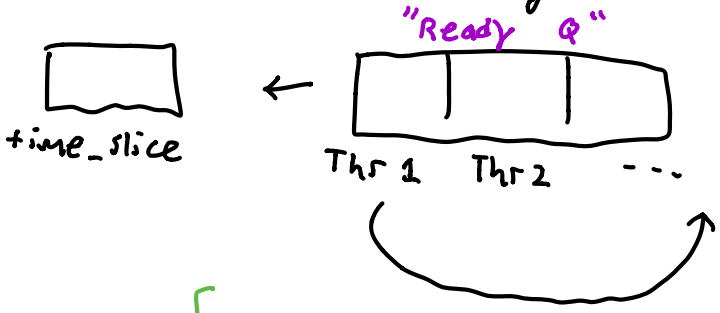
=> each thread has its own stack

context switching

each processor has fixed time_slice

each thread can only be continued for a max time of time_slice

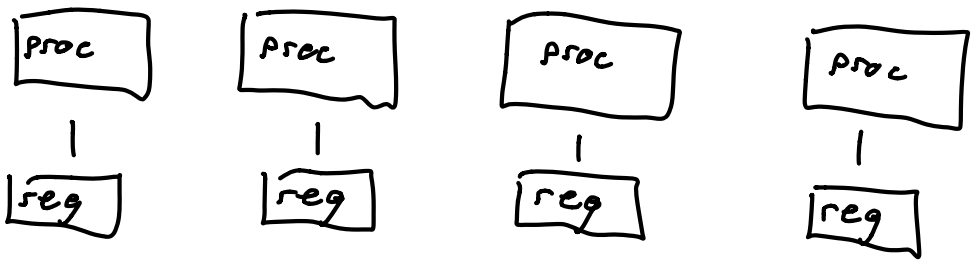
then next thread in line gets computed



context switching

- once time slice is up, goes to end of queue
- when this happens, all the registers are copied into local space associated with that thread
- Then processor picks up the registers for next scheduled thread

multicore processing



each processor has its own register set

Basic thread functions

each thread has an ID (starting with zero)

```
int tid = omp_get_thread_num();
```

current thread $\in \{0, 1, 2, \ldots, 7\}$

```
int numt = omp_get_num_threads();
```

total # of threads = 8

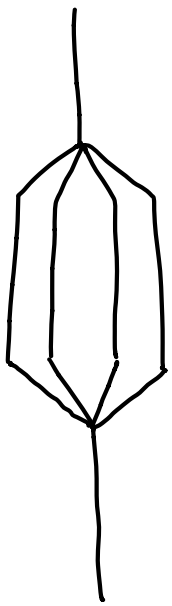
atomic operation: appear as if nothing else is happening within that operation

*if I have a print statement in each of the threads, the prints could happen at the same time
(i.e. print Hello wo Helrld lo world)

but there are ways to make this atomic => locks, mutual exclusion

Forking

going from a master thread to launching additional threads



called "fork-join model

Sequential code and consistency model

compilers sometimes move around lines of code when computing for optimization purposes

```
X = a
for (i=0; i<n; ++
  ~
  ~
```

sometimes will move this if not in for loop

sequential consistency model

processor 1	processor 2	processor 3
line 1	line 1	line 1
line 2	line 2	line 2
line 3	line 3	line 3

operations of each individual processor appear in the order specified by its program

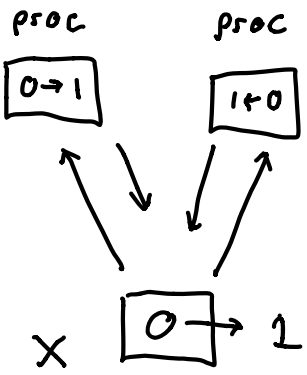
i.e.

line 1
line 2
line 1
line 1
line 2
line 3
line 3
⋮

too stringent and aren't used in practice

Race Condition

correctness of the program depends on how the threads are scheduled



but we wanted 2

want x = 2
but use x+=1 twice

private versus shared variables

private: each processor has its own copy and each copy is only accessible to a specific processor

shared: same memory location for all threads

critical section: section of the code that can only be executed one thread at a time

Linear Algebra

BLAS (Basic Linear Algebra Subroutines)

BLAS-1	vector, vector-vector
BLAS-2	matrix-vector
BLAS-3	matrix-matrix

scheduling

schedule (dynamic / static , chunk_size)

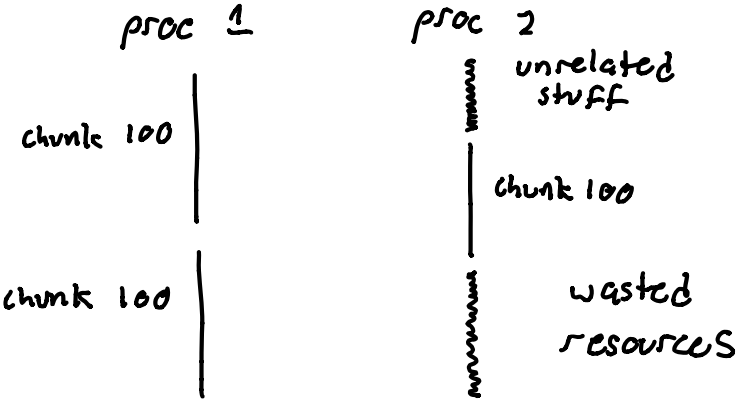
static: all threads scheduled during compile time (before run-time)

dynamic: some threads scheduled during run time

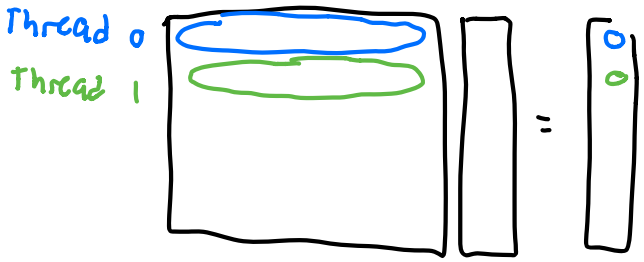
chunk_size: number of threads in a row that will all be assigned to the same processor

* dynamic scheduling better because you can't typically know when each processor will finish its work nor how long each chunk will take
* dont want small chunk_size so we can take advantage of cashe but dont want it too big because then I'll have processors that are free for a long time waiting for the last chunk to finish

i.e.



matrix-vector operations



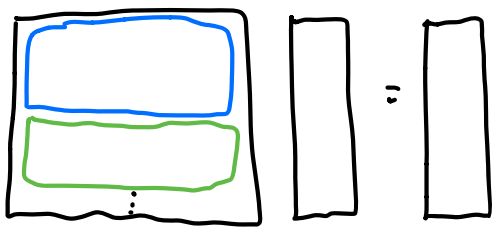
optimal approach for C because of row-major order, doesn't need to lock

* Divide the work given to different threads given the output

Also in C, data is stored in Row-major order

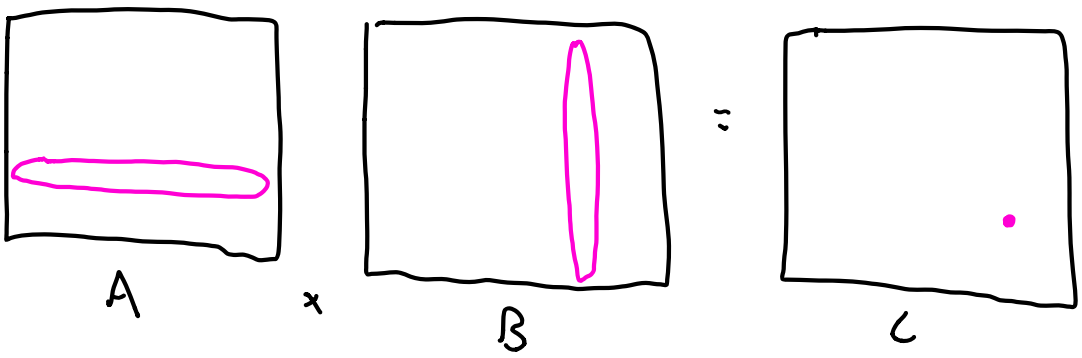
Best to store results $\begin{matrix}0\\0\end{matrix}$ in local variables and update it at the end because of "false sharing" where the caches for the two threads are so close, the caches both store data for location $\begin{matrix}0\\0\end{matrix}$ so computer thinks it has to reload data

Better solution:



assign multiple contiguous rows to same thread

matrix-matrix multiply

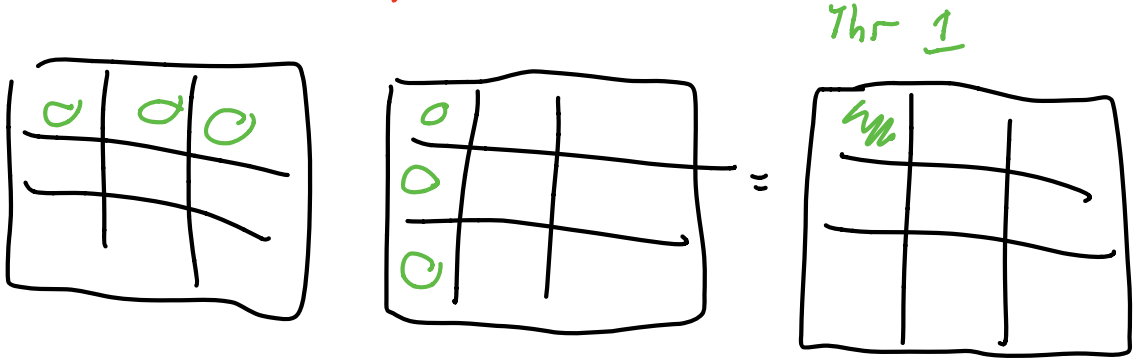


all $\in \mathbb{R}^{n \times n}$

```
#pragma omp parallel for collapse (2)
    for (;
        for (;
            for (k
                c[i,j] += A[i,k] * B[k,j];
```

will choose i,j pair for each thread

But to take full advantage of cashe



Make into submatrices that are b×b s.t. 3 b×b matrices can fit into the cashe at the same time

Openmp tasks

piece of code that doesn't have to be executed right now, just whenever you have time

Tasks were introduced in openmp 3.0
but VS uses openmp 2.0

```
#pragma omp parallel
{
    int j;
    j = omp_get_thread_num (j);
```

```
# pragma omp task
{
    int k = omp_get_thread_num();
    printf ("%d, %d", j, k);
```

initial thread that executes first few lines of parallel region and also puts the task into task queue

thread that ends up executing this task

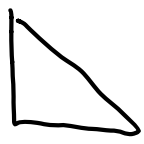
LU factorization

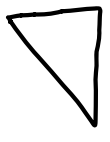
$A = LU$

$Ax = b$

$LUx = b$

$\underbrace{Ux}_{x'} = b$

$Lx' = b$  $\begin{bmatrix} x' \end{bmatrix} = \begin{bmatrix} b \end{bmatrix}$ Triangular Solve Vectors (TRSV)

$Ux = x'$  $\begin{bmatrix} x \end{bmatrix} = \begin{bmatrix} x' \end{bmatrix}$ TRSV

} BLAS

But how do we get L, U? Let's do example

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 3 & 1 & 4 \\ 5 & 3 & 1 \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, \quad b = \begin{bmatrix} 14 \\ 17 \\ 14 \end{bmatrix}$$

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 & 3 \\ 3 & 1 & 4 \\ 5 & 3 & 1 \end{bmatrix}$$

$R_2 \to R_2 - 3R_1$

$$= \begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 0 & -5 & -5 \\ 5 & 3 & 1 \end{bmatrix} \quad R_3 \to R_3 - 5 \cdot R_1$$

$$= \begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 5 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 0 & -5 & -5 \\ 0 & -7 & -14 \end{bmatrix} \quad R_3 \to R_3 - \frac{7}{5} R_2$$

$$= \begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 5 & \frac{7}{5} & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 0 & -5 & -5 \\ 0 & 0 & -7 \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 5 & 7/5 & 1 \end{bmatrix} \begin{bmatrix} x_1' \\ x_2' \\ x_3' \end{bmatrix} = \begin{bmatrix} 14 \\ 17 \\ 14 \end{bmatrix}$$

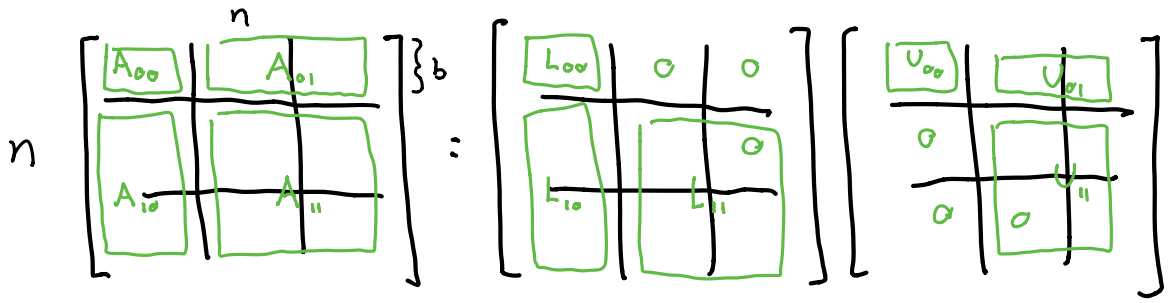
$$\begin{bmatrix} x_1' \\ x_2' \\ x_3' \end{bmatrix} = \begin{bmatrix} 14 \\ -25 \\ -21 \end{bmatrix}$$

$$70 - 35 + x_3' = 14$$
$$35 + x_3' = 14$$
$$x_3' = -21$$

$$\Rightarrow \begin{bmatrix} 1 & 2 & 3 \\ 0 & -5 & -5 \\ 0 & 0 & -7 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 14 \\ -25 \\ -21 \end{bmatrix}$$

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

How do we implement this in parallel? Blocking!



$A_{00} = L_{00} U_{00}$

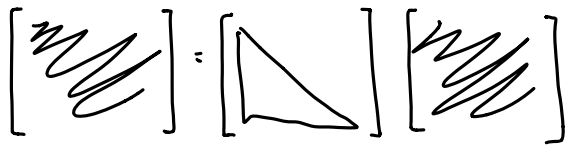
$A_{01} = L_{00} U_{01}$

$A_{10} = L_{01} U_{00}$

$A_{11} = L_{10} U_{01} + L_{11} U_{11}$

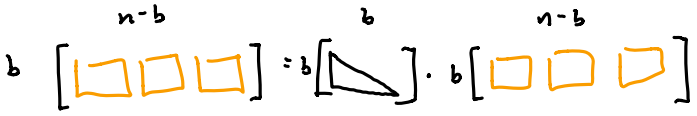
1) compute L_{00}, U_{00} by factorizing L_{00}, U_{00} (easy - shown above)

2) compute U_{01} using $A_{01} = L_{00} U_{01}$



if these were square, this is simply a triangular solve with a matrix TRSM

But they look more like this



But we can break into b×b blocks then use TRSM methods

3) compute L_{10} using $A_{10} = L_{10} U_{00}$ (same as step 2)

4) $L_{11} U_{11} = A_{11} - L_{10} U_{01} = A_{11}'$

in this step I just need to solve A_{11}'

5) Recursively solve $A_{11}' = L_{11} U_{11}$ untill A_{11}' is of size b×b

how to parallelize:
- steps 2, 3 can happen at the same time and even within those individual steps, the b×b blocks can be computed using different threads
- step 4 is most computationally heavy but we know how to parallelize

pragma omp atomic

- acts like critical section but only works for simple operations (increment, addition, ...)
- essentially puts a lock the memory address of whatever is being updated