Parallel software

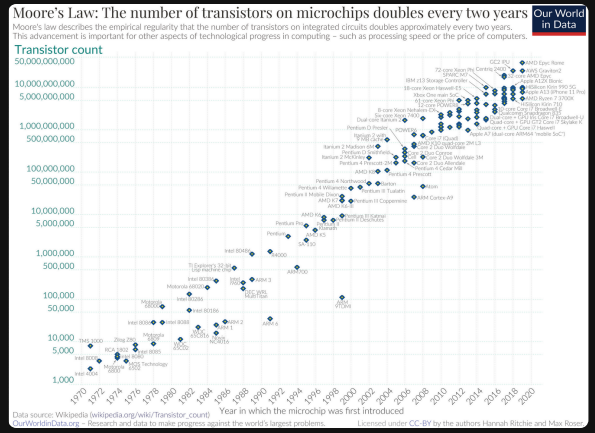
About Software

- At this point in 2026, most programs have some support for parallelism
- However, many programs are still written to run serially

> JavaScript, one of the worlds most popular languages, is single-threaded*

*Browsers and Runtimes have APIs to let JS have parallelism

This is a problem because we can't rely on hardware and compilers to provide a **steady increase** in application performance anymore



SPMD

We'll mainly focus on *single program, multiple data*, or **SPMD**, programs.

Instead of running a different program on each core,

> SPMD have one executable that acts like multiple different programs

It does this through conditionals

- Note that this means it can do **both** task parallelism and data parallelism

```
1  if (Im thread 0) {
2      do task A
3  } else if (Im thread 1) {
4      do task B
5  } else {
6      do task C
7  }
```

Parallelization

Sometimes getting good parallel performance is easy. Recall the two array addition example

```
1  double x[N], y[N], z[N];
2  for (int i = 0; i < N; i++) {
3      z[i] = x[i] + y[i];
4  }
```

Since the process of converting a program into parallel is called **parallelization**.

Programs that can be parallelized by simply dividing the work is sometimes called embarrassingly parallel.

Shared memory

communication in shared memory systems

Recall that shared memory systems can have *shared* and *private* variables

Communication is primarily done through shared variables

Dynamic and static threads

Most shared-memory programs use **dynamic threads**.

In this system

- there is a **master thread**, and
- a (possibly empty) collection of **worker threads**.

The master thread waits for work, then forks a worker thread to do the work

When the worker thread completes, it terminates and joins the master thread

This results in more efficient use of system resources, since only working threads use resources

Static threads

An alternative to the dynamic paradigm is the **static thread** paradigm.

In this system:

- all of the threads are forked after any needed setup by the master thread
- and the threads run until all the work is completed.

After the threads join the master thread, the master thread may do some cleanup, and then it also terminates.

This **may** be less efficient: if a thread is idle, its resources can't be freed.

But forking and joining threads can be fairly time-consuming operations, so there's potentially a performance gain.

Non determinism

In any MIMD system with async cores, especially shared memory systems, it is likely that there will be **nondeterminism**.

> if a given input can result in different outputs.

Usually occurs because multiple threads have varied execution timing and so if we assume two threads, with `7` and `19` as `my_x`

```
1   printf("Thread %d > my_x = %d\n", my_rank, my_x);
```

Could output

```
1   Thread 0 > my_x = 7
2   Thread 1 > my_x = 19
```

Depending on which thread executes first during that run

This is also called a **race condition**

Non Determinism

Usually, non determinism isn't that large of a problem and the order of outputs doesn't matter

However, race conditions can lead to very large problems that aren't easily detected

youtube.com/watch?v=41Gv-zzICIQ

Atomic operations

To prevent nondeterminism, we want certain operations to be *"atomic"*

Meaning that they are indivisible and uninterruptible, where after the thread has completed the operation, it appears that no other thread has executed during that time

There are a few ways of doing this, one is simply making sure that **only one** thread can execute a block at a time

The section of code is called a **critical section** and we use **mutual exclusion** to ensure that only one thread can execute the critical section at a time

Mutex

One of the simplest ways of doing so is using a **mutual exclusion lock**, or **mutex**, or simply

> lock

marking a block of code with a lock "protects" that section, and the thread must *have the mutex*. And when it's done, it *releases* the mutex

```
1  my_val = get_val(my_rank);
2  Lock(&add_my_val_lock);
3  x += my_val;
4  Unlock(&add_my_val_lock);
```

Given `process 0` and `process 1`, which one will execute the critical section first?

Mutex

A mutex **enforces** serialization for the critical section,

which means we want as few critical sections as possible

Alternatives to mutexes

- Busy-waiting

Simply have a thread enter a **loop** which tests a condition

```
my_val = get_val(my_rank);
if (my_rank == 1) {
    while (!ok_for_1) {}
    x += my_val;
}
if (my_rank == 0) {
    ok_for_1 = true;
}
```

- Semaphores

Usually consisting of a `counter` and the operations `wait` and `signal`

The combination of these two operations can be used to ensure mutual exclusion

```
counter = 1 // printer available

Thread A: wait() → c becomes 0 → print
Thread B: wait() → c = 0 → wait
Thread C: wait() → c = 0 → wait

Thread A finishes → signal() → counter = 1
Thread B or C can now proceed
```

Thread safety

Most of the time, you can use any functions you like in a parallel program.

However, some functions were written *specifically* for use in serial programs,

For `C`, any function that static local variables, like the function `strtok` which splits a string into sub stirngs could cause problems

`strtok` internally uses a static variable

```c
char s[] = "a,b,c";
char *token = strtok(s, ",");
while (token != NULL) {
    printf("%s\n", token);
    token = strtok(NULL, ",");
}
```

If multiple threads call `strtok` at the same time, they will interfere with each other

This is an example of a function that is not **thread safe**

Distributed memory

communication in distributed memory systems

In distributed memory, each process only has access to its own memory

Communication is done by sending and receiving messages

Note that while distributed memory systems are traditionally clusters of computers, they can also be on a single computer with multiple processors

Message passing

At minimum, message passing systems need two operations:

- send
- receive

```
1  char message[100];
2
3  my_rank = Get_rank();
4  if (my_rank == 1) {
5      sprintf(message, "from 1");
6      Send(message, MSG_CHAR, 100, 0);
7  } else if (my_rank == 0) {
8      Receive(message, MSG_CHAR, 100, 1);
9      printf("Process 0 received message: %s\n", message);
10 }
```

1. this is an SPMD program, same executable for both processes
2. `message` refers to *different* blocks of memory
3. we're assuming `proc 0` can write to stdout

Behavior of send and receive

Depending on the API, the implementation, and the system, `Send` and `Receive` can have different behaviors

The most common and simplest is **blocking**

- `send` the message
- `receive` the message

But the `recieve`r will **block** until the message arrives

There are a few other ways to communicate, like *collective* communication, such as **broadcast**.

Or a **reduction**, where all processes contribute a value to a single result, like a sum

The most widely used message passing API is **MPI**

One sided communication

One process must call a send function, and that send needs to be matched by a receive function in another process.

one-sided communication allows a single process to set the value of a variable in another process

Usually done by **directly** modifying the memory of another process

This makes it *slightly easier*, reduce overhead from syncing, and reduce overhead from one of the functions.

In practice, it's difficult to realize these benefits, and since there isn't any *explicit* communication, debugging is more difficult