(HPC)
High Performance Coding & Computing

Alberto Dassatti - 2023

What's next

You now know how to optimize your code.

Bit Hacks, caching, branch prediction and SIMD have no secrets for you.
You know how to write code in such a way your compiler enables the most advanced optimizations.

What's next

You now know how to optimize your code.

Bit Hacks, caching, branch prediction and SIMD have no secrets for you.
You know how to write code in such a way your compiler enables the most advanced optimizations.

What if all of this is not enough?

What's next

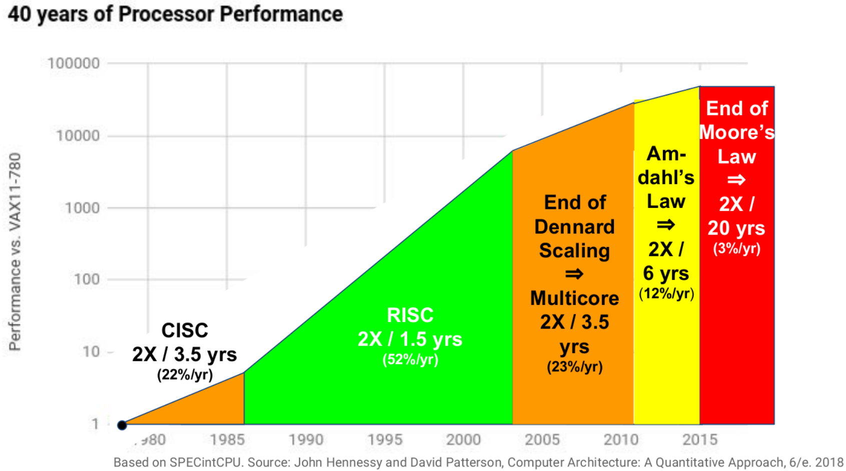
You now know how to optimize your code.

Bit Hacks, caching, branch prediction and SIMD have no secrets for you.
You know how to write code in such a way your compiler enables the most advanced optimizations.

What if all of this is not enough?

Parallel hardware is obiquitus: let's go parallel!

End of Growth of Performance?



[Reference.]

Parallelism vs Concurrency

Tasks are parallel when they execute at the same time.

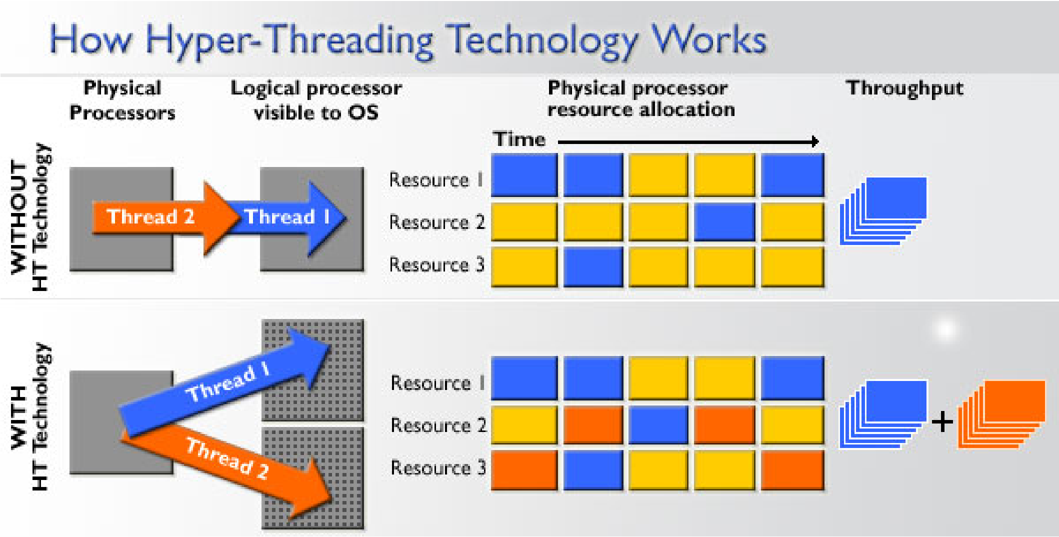
Tasks are concurrent when their execution order is not predetermined.

PARALLEL MACHINES

Shared-Memory Hardware
or
Distributed-Memory Hardware

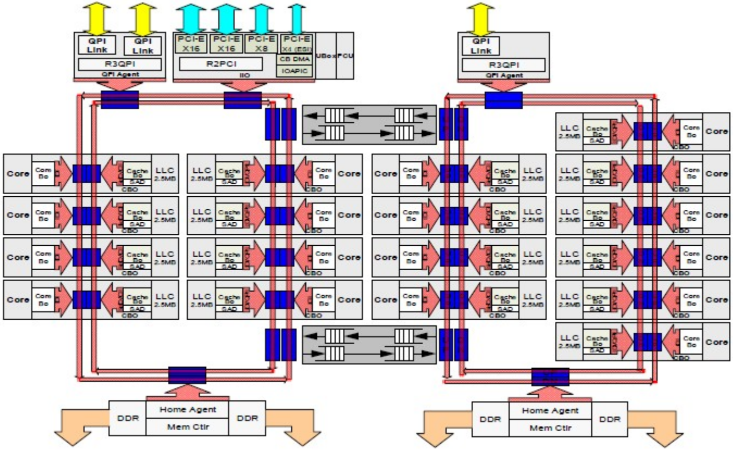
Shared-Memory machines have HW Cache Coherence, the semantic and implementation of wich is extremely complex and out of scope for our purposes.

Distributed-Memory machines use Message passing (MPI) to transfer data and distribute job on several nodes.



[source and extra Reference]

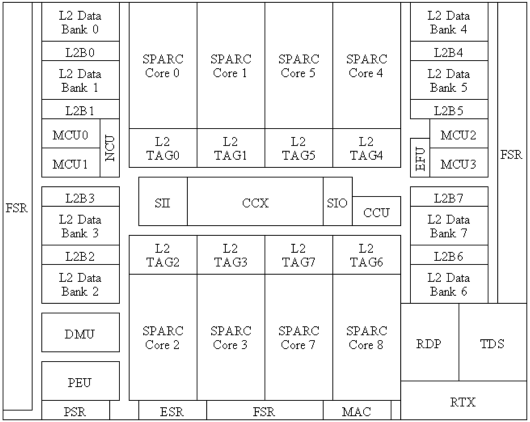
XEON E7-8890 V4



up to 24 cores.

[source]

ULTRASPARC T2

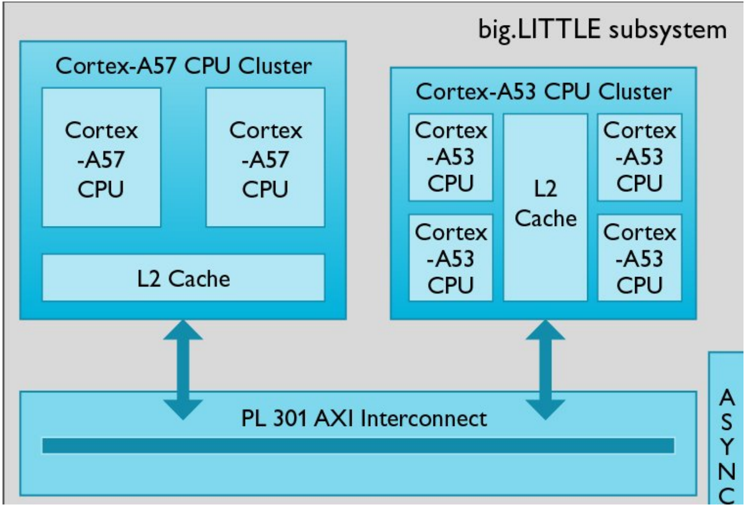


Niagra 2 / UltraSPARC T2 / OpenSPARC T2 - Die Micrograph Diagram (davidhalko)

up to 8 cores, 8 threads each.

[source]

Cortex A72



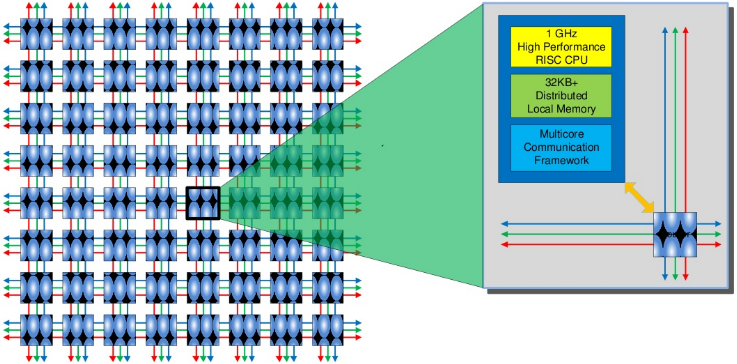
up to 8 cores.

[Reference]

Parallela Board

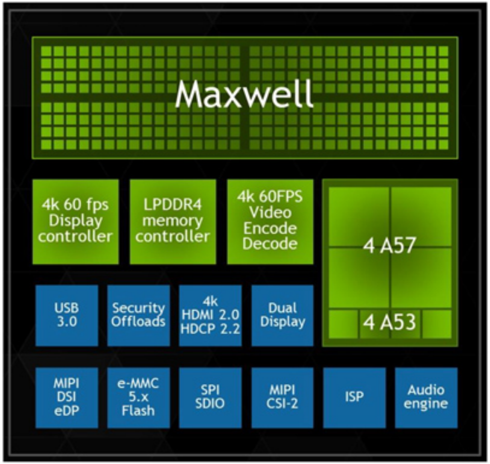
[Reference.]

Parallela Board



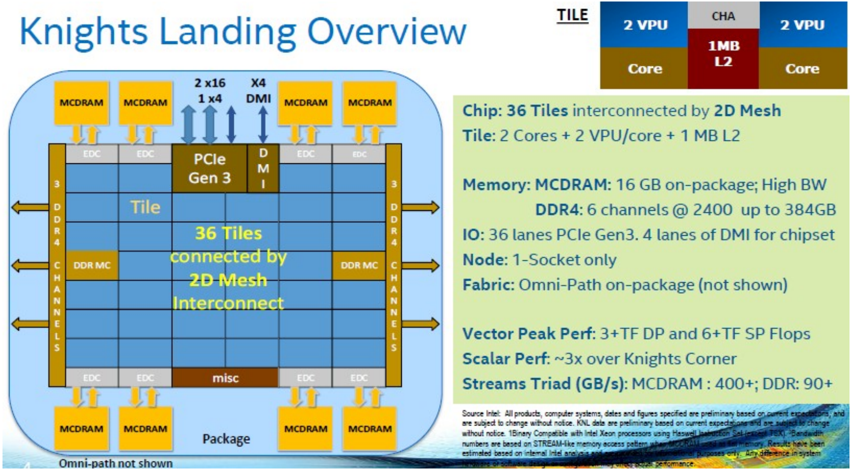
[Reference.]

NVIDIA X1



[Reference.]

XEON PHI



[Reference.]

Hardware

There are many parallel machines and many more are coming, but the question is still there:

Hardware

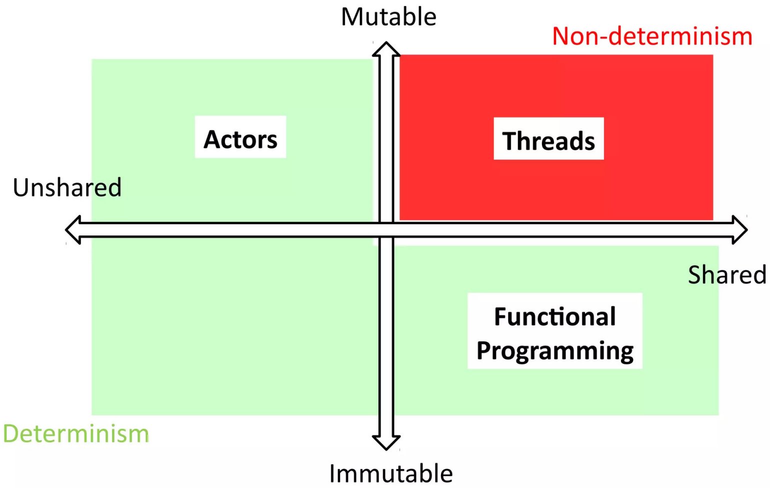
There are many parallel machines and many more are coming, but the question is still there:
how we program them?

Hardware

There are many parallel machines and many more are coming, but the question is still there:
how we program them?

You already know `pthreads`. Can they be the answer? Why?

The state quadrants



Parallelism

Using parallelism means doing more tasks (work) in parallel.
Main difficulties:

1. data dependency (linked list traversal).
2. syncronization (mutex, semaphores, data races, deadlocks, etc).
3. problem formulation (a less performant serail algorithm may be easier to parallelize).
4. compilers (and in general tools) do not help much (as of now).
5. memory sub-system is critical (SMP, NUMA, rDMA).
6. DSL and languages can have a better impact on parallel programming, C/C++ do not shine here.

Suggested reading: *"Is Parallel Programming Hard, And, If So, What Can You Do About It?"*, by Paul E. McKenney (pfd).

Parallelism: good news

There are many frameworks (languages, libraries, etc.) that helps the developper.

Our objective is knowing the existing options and making some experience with (some of) them.

PARALLELISM: GOOD NEWS

- `pthread` (all low level details are exposed to the programmer).
- OpenMP (`#pragma` based).
- OpenACC (`#pragma` based).
- Threading Building Blocks (Intel TBB C++).
- Cilk Plus (fork-join parallelism, now Intel, Tapir MIT open source).
- OpenCL.
- CUDA (only NVIDIA).
- SYCL.
- DSL (break computation and schedule apart)

- MPI (non shared memory machines).

Parallelism: bad news

There are many frameworks (languages, libraries, etc.) that helps the developper,

but

Parallelism: bad news

There are many frameworks (languages, libraries, etc.) that helps the developper,

but

they are incompatible, closed source, quickly evolving or abbandoned, complex, miss tooling, are not portable, etc. We will only focus on:

- `pthread`: POSIX compliant, high availability.
- OpenMP: mature and stable, well supported by open tooling.

Before Programming some theory

what are the limits to parallelization?

Before Programming some theory

what are the limits to parallelization?

Most programs have a sequential part and a parallel part.

AMDAHL'S LAW

`S`: fraction of serial runtime in a serial execution.
`P`: fraction of parallel runtime in a serial execution.

$$S + P = 1 \tag{1}$$

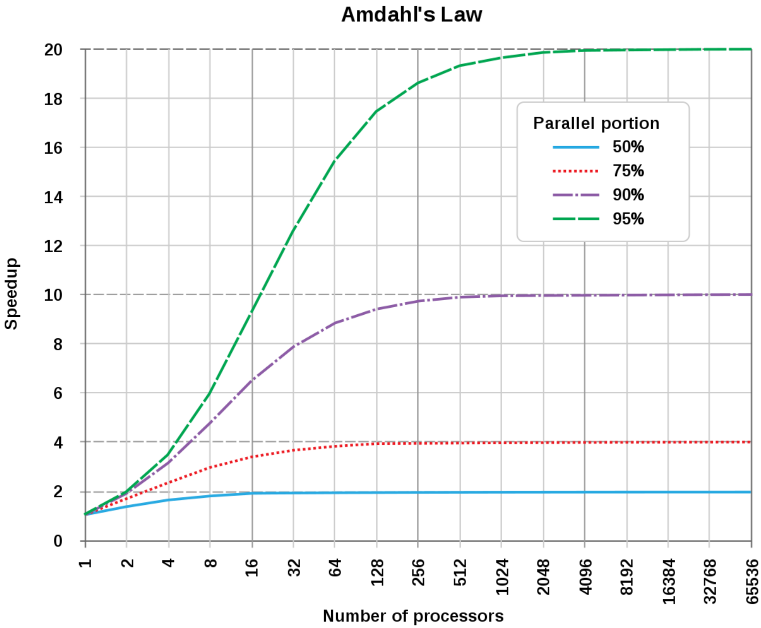
T_s : time for the program to run in serial
N: number of processors/parallel executions
T_p : time for the program to run in parallel

$$T_p = T_s(S + \frac{P}{N}) \tag{2}$$

Amdahl's Law

If no overhead (never the case, but could be low) is needed to parallelize, the speedup can be expressed as:

$$
\begin{aligned}
speedup &= \frac{T_s}{T_p} \\
&= \frac{T_s}{T_S \cdot (S + \frac{P}{N})} \\
&= \frac{1}{S + \frac{P}{N}}
\end{aligned}
$$


Amdahl's Law
Speedup
Number of processors
Parallel portion
50%
75%
90%
95%
0
2
4
6
8
10
12
14
16
18
20
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536

Amdahl's Law

$$
\begin{aligned}
speedup &= \frac{T_s}{T_p} \\
&= \frac{T_s}{T_S \cdot (S + \frac{P}{N})} \\
&= \frac{1}{S + \frac{P}{N}} \\
&= \frac{1}{(1-P) + \frac{P}{N}}
\end{aligned}
$$

or

$$
max\_speedup = \frac{1}{(1-P)} \tag{3}
$$

Thread & Process

Modern OSes give us two ways of describing parallelism and concurrency: threads and processes.

Process

- Private address space (virtual memory).
- One private stack.
- Private resources.
- High overhead (`fork` and switch).

Thread

- Shared address space (parent virtual memory).
- Multiple private stacks.
- Shared resources.
- Low overhead (`pthread_create` and switch without MMU involevement).

Thread

There are three main threading models:

- Hardware Thread (hyperthreading or cores).
- Kernel Thread (POSIX threading model). Available in C++11 and C11, [1].
- Green Thread or user level threading.
    - Quick context switches, no need for system call.
    - Cannot use multiple processors, only for concurrency.
    - GNU Portable Threads, GO, Ruby, etc., list.

Windows 7 use an hybrid model. It has the benefit of the two worlds, but has increased complexity.

Each task (process/thread) can be associated with a set of processors (Processor Affinity, `taskset`).

PTHREAD

Some aspects of `pthread` you may be interested in with respect of performance:

- Detached or joinable state
- Scheduling inheritance
- Scheduling policy
- Scheduling parameters
- Scheduling contention scope
- Stack size
- Stack address
- Stack guard (overflow) size

Make sure the libraries you use are thread-safe! (`glibc` reentrant functions are.)

IO

If you are blocked by I/O, may be files, may be sockets, Asynchronous I/O could be the answer.

It's not easy to implement correctly and usually the OS does a great job for you, but nobody knows your task better than you do.

```
fd = open (... , O_NONBLOCK );
read (...); // returns instantly !
close ( fd );
```

Modern linux includes io uring, go and check it out!

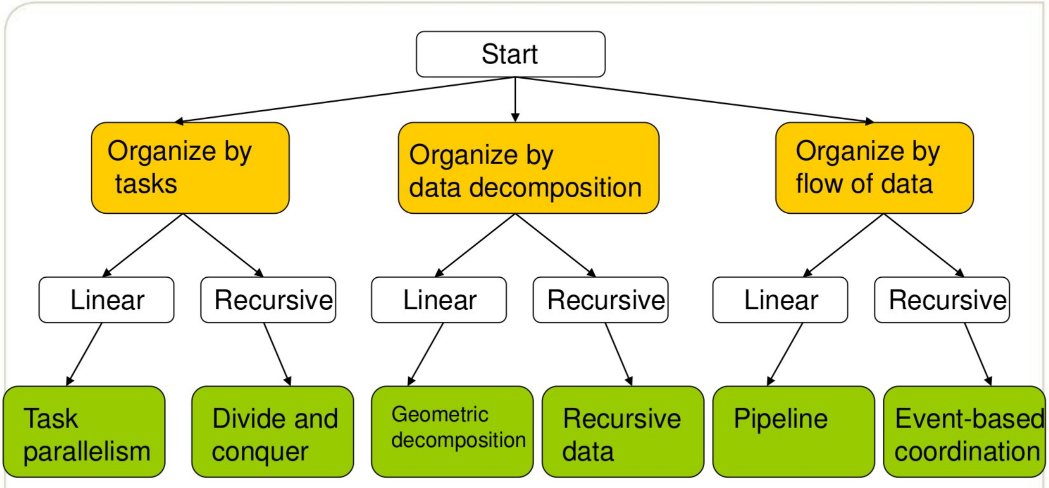
Few words on Synchronization

You know what a `mutex` is and how to use it. There are some useful fucntions if your focus is performance:

- try-lock.
- `pthread_spin_lock`: repeatedly try the lock and will not put the thread to sleep.
- if reads can happen in parallel, as long as there's no write, `pthread_rwlock_rdlock` and `pthread_rwlock_wrlock`.
- `pthread_barrier`: Allows you to ensure that (some subset of) a collection of threads all reach the barrier before returning.
- Atomics! [1],[2].
- (not really synchronization but) do not forget `volatile`.

But much better than all of them... Lock-Free Algorithms!

Parallelization Patterns



source.

THREAD POOLS

Instead of creating threads, destroying them and recreating them, you can use a thread pool. It creates `n` threads; you just push work onto them.

Only question is: How many threads should you create? (Experiment to find out). For an example implementation look at GLib.

For more…

- Memory consistency models.
- Work stealing scheduler.
- Lock free algorithms and data structures.
- Heterogeneus systems.