Profiling, optimization, parallelization

David Barina

April 7, 2020

Motivation

Why parallelize?

Reduce computation time

- real-time applications (control systems, games, ...)
- ensuring the usefulness of results (simulation, prediction, ...)
- processing of incoming data on time
- ensuring user comfort

Reduce amount of energy needed

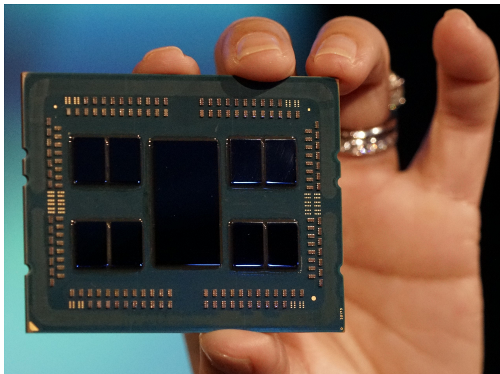
- mobile and energy restricted devices
- high-performance computing (HPC)

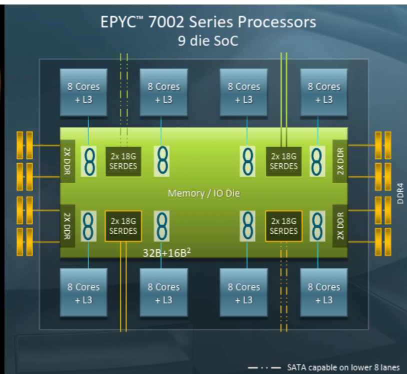
Why not?

Return on investment (ROI)

Snapdragon 865

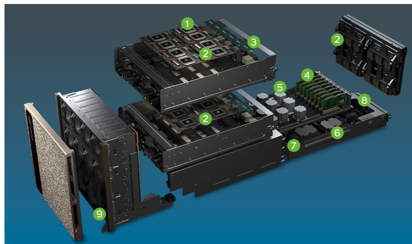
AMD EPYC 7002 Series

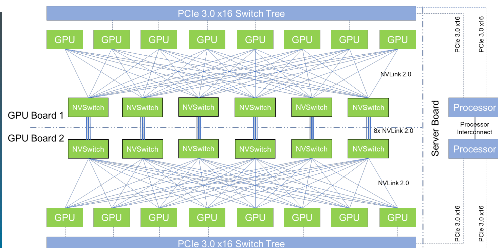




- 64 cores / 128 threads
- 256MB L3 cache

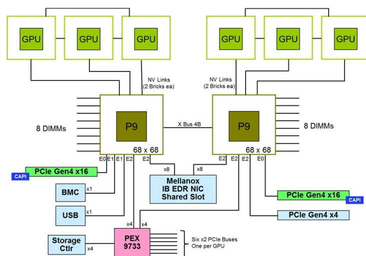
NVIDIA DGX-2





- $16 \times$ NVIDIA Tesla V100
- GPU memory: 512 GB (16×32 GB)

Summit (supercomputer)



- 4 608 nodes (machines)
- GPUs: 27 648 NVIDIA Volta V100s (6/node)
- Memory: $>$ 10 PB DDR4

CPU core

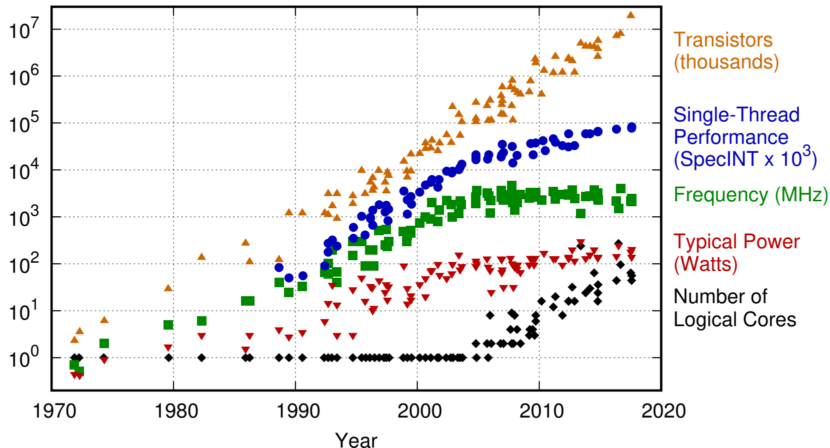
Hidden parallelism

- up to 5 instructions per cycle
- up to 224 μ-operations
- 7 parallel units

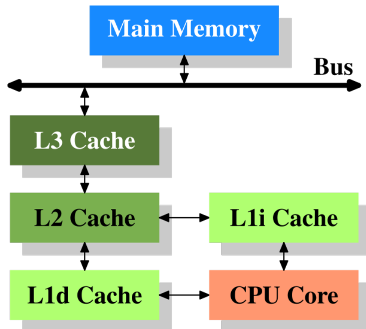
Explicit parallelism

- SMT (Hyper-threading)
- vector operations (SIMD)
- multiple cores with shared memory (MIMD)

Evolution



CPU cache



Modern hardware

- Mobile CPU
  - 64-bit ARM (AArch64)
  - up to 8 cores/threads (often 4+4)
  - up to 2.5 GHz
  - cache: 2 MB L2 cache
- Desktop CPU
  - 64-bit x86 (AMD64)
  - up to 16 cores / 32 threads
  - up to 4.5 GHz
  - cache: 256 kB/core L2 + 64 MB L3
- Server CPU
  - 64-bit x86 (AMD64)
  - up to 64 cores / 128 threads
  - up to 2.35 GHz
  - cache: 512 kB/core L2 + 256 MB L3

Optimization

- appropriate algorithm (asymptotic complexity)
- profiling
- compiler options
- compiler intrinsics (SIMD)
- parallelization

Big-O notation

Used to classify algorithms according to how their running time or space requirements grow as the input size grows

$$f(n) = \mathcal{O}(g(n))$$

- $|f|$ is bounded above by g (up to constant factor) asymptotically
- best, worst, and average cases

Big-O notation

notation	name
$\mathcal{O}(1)$	constant
$\mathcal{O}(\log n)$	logarithmic
$\mathcal{O}(n)$	linear
$\mathcal{O}(n \log n)$	linearithmic
$\mathcal{O}(n^2)$	quadratic
$\mathcal{O}(n^3)$	cubic
$\mathcal{O}(n^c)$	polynomial
$2^{\mathcal{O}(n)}$	exponential
$\mathcal{O}(n!)$	factorial

Big-O notation

bogo sort

- $T_{\text{worst}} = \mathcal{O}(\infty)$
- $T_{\text{avg}} = \mathcal{O}(n\, n!)$
- $S_{\text{worst}} = \mathcal{O}(1)$

while array is not in order **do**
    shuffle array
end while

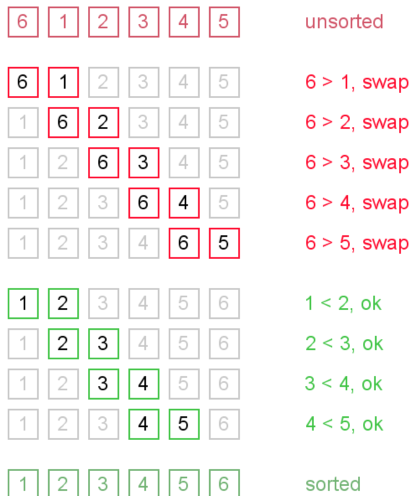
Big-O notation

bogo sort

- $T_{\text{worst}} = \mathcal{O}(\infty)$
- $T_{\text{avg}} = \mathcal{O}(n\, n!)$
- $S_{\text{worst}} = \mathcal{O}(1)$

bubble sort

- $T_{\text{worst}} = \mathcal{O}(n^2)$
- $T_{\text{avg}} = \mathcal{O}(n^2)$
- $S_{\text{worst}} = \mathcal{O}(1)$



Big-O notation

bogo sort

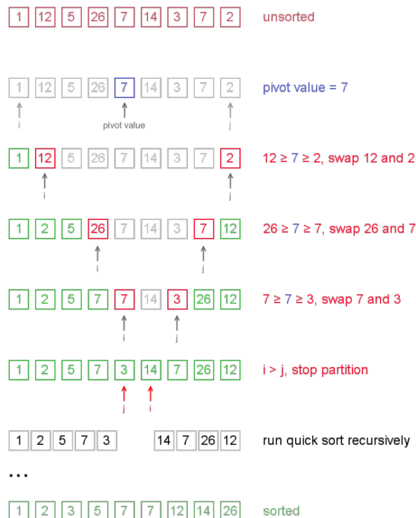
- $T_{\text{worst}} = \mathcal{O}(\infty)$
- $T_{\text{avg}} = \mathcal{O}(n\, n!)$
- $S_{\text{worst}} = \mathcal{O}(1)$

bubble sort

- $T_{\text{worst}} = \mathcal{O}(n^2)$
- $T_{\text{avg}} = \mathcal{O}(n^2)$
- $S_{\text{worst}} = \mathcal{O}(1)$

quick sort

- $T_{\text{worst}} = \mathcal{O}(n^2)$
- $T_{\text{avg}} = \mathcal{O}(n \log n)$
- $S_{\text{worst}} = \mathcal{O}(\log n)$



Big-O notation

bubble sort

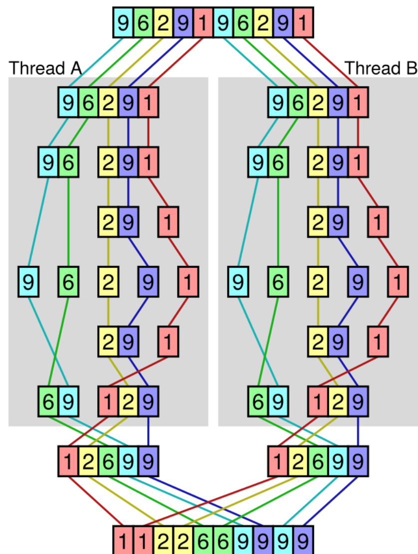
- $T_{\text{worst}} = \mathcal{O}(n^2)$
- $T_{\text{avg}} = \mathcal{O}(n^2)$
- $S_{\text{worst}} = \mathcal{O}(1)$

quick sort

- $T_{\text{worst}} = \mathcal{O}(n^2)$
- $T_{\text{avg}} = \mathcal{O}(n \log n)$
- $S_{\text{worst}} = \mathcal{O}(\log n)$

merge sort

- $T_{\text{worst}} = \mathcal{O}(n \log n)$
- $T_{\text{avg}} = \mathcal{O}(n \log n)$
- $S_{\text{worst}} = \mathcal{O}(n)$



Big-O notation

$n!$ 2^n n^2 $n \log_2 n$ n $\sqrt{n}$ $\log_2 n$ 1

N: 0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100

n: 0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100

Optimize options

-march

`-march=skylake`, `-march=znver1`, `-march=native`

-O

`-O2`, `-O3`, `-Og`

-m

`-msse3`, `-mavx`, `-mavx2`

Debugging

`-Og -g -rdynamic`

Branch predictor

```
#define N 100000000

int arr[N];

// fill arr[n] with random numbers 1..255

clock_t start = clock();

for (size_t n = 1; n < N; ++n) {
        if (arr[n] >= 128) {
                r *= arr[n];
        }
}

printf("time␣=␣%f␣secs\n", (clock() - start) / (float)
    CLOCKS_PER_SEC);
```

Branch predictor

Processing a sorted array is faster than processing an unsorted array

array	time
unsorted	0.420355 secs
sorted	0.077027 secs

```
gcc -std=c99 -march=native -O3
```

CPU cache size

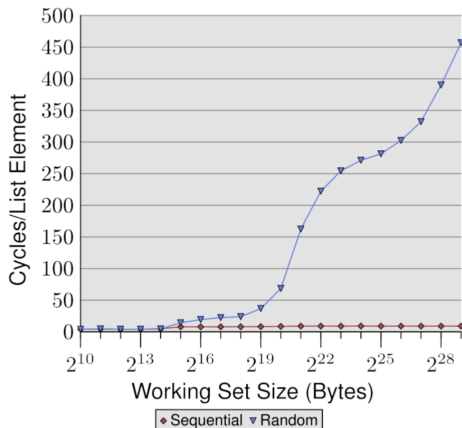
Sequential access

```
for (size_t n = 0; n < STEPS; ++n) {
        arr[n % size]++;
}
```

Random access

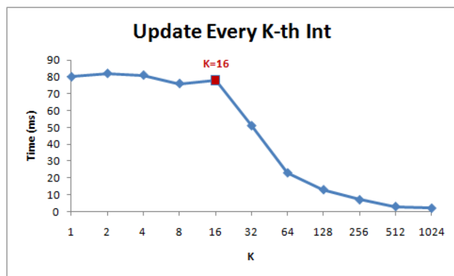
```
for (size_t n = 0; n < STEPS; ++n) {
        arr[rand() % size]++;
}
```

CPU cache size



CPU cache line

```
for (size_t n = 0; n < LENGTH; n += K) {
        arr[n] *= 3;
}
```



CPU cache associativity

```
#define N (16 * 1024)

int arr[N][N];
```

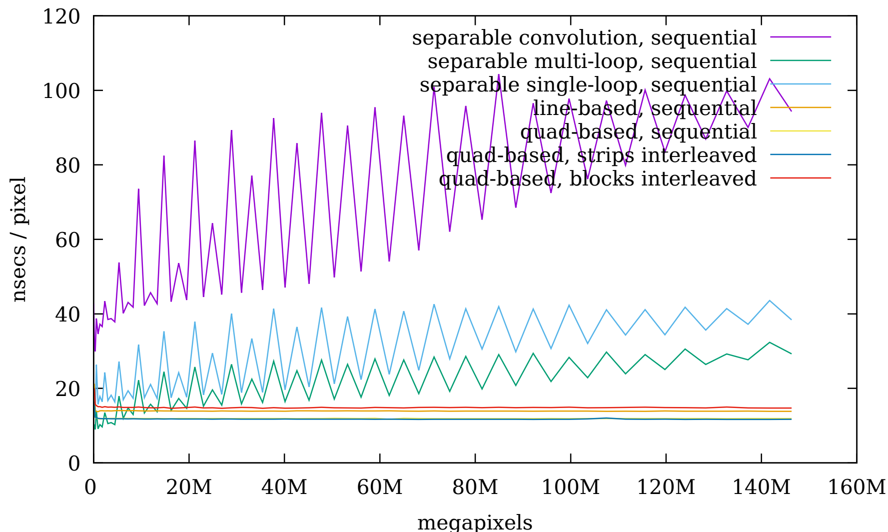
```
for (size_t y = 0; y < N; ++y) {
        for (size_t x = 0; x < N; ++x) {
                arr[y][x]++;
        }
}
```

```
0.16user 0.29system 0:00.46elapsed 100%CPU
```

```
for (size_t x = 0; x < N; ++x) {
        for (size_t y = 0; y < N; ++y) {
                arr[y][x]++;
        }
}
```

```
3.13user 0.28system 0:03.41elapsed 100%CPU
```

CPU cache effects



Profiling

Data obtained include

- annotated source listing (number of executions, % time)
- call graph (number of calls per function, % time)
- number of L1, L2, L3 cache accesses, hit/miss rates
- other counters (branch predictor statistics)

Tools: Callgrind, Cachegrind, VTune, gprof, ...

Callgrind

Information about the number of instructions executed

1. `valgrind --tool=callgrind your-program`
2. `callgrind_annotate [options] callgrind.out.<pid>`

```
          .   int divisible_ui_p(unsigned int m, unsigned int a)
          .   {
244,213,636       while (m > a) {
 72,111,818               m += a;
288,447,272               m >>= __builtin_ctz(m);
          .       }
          .
 49,995,000       if (m == a) {
      6,736               return 1;
          .       }
          .
 99,976,528       return 0;
      6,736   }
```

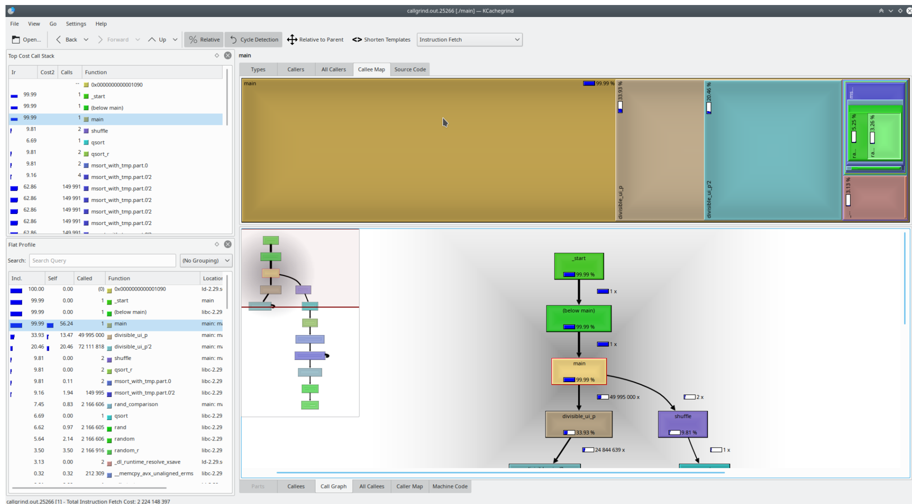
Cachegrind

Information about the CPU cache utilization

1. `valgrind --tool=cachegrind your-program`
2. `cg_annotate [options] cachegrind.out.<pid>`

```
Ir          I1mr ILmr Dr
          .    .    .           .  int divisible_ui_p(unsigned
          .    .    .           .  {
          .    .    .           .          do {
122,701,334    0    0           0                  m += a;
368,104,002    0    0           0                  m >>=
245,402,668    0    0           0          } while (m > a);
          .    .    .           .
 49,995,000    0    0           0          if (m == a) {
      6,736    0    0           0                  return 1;
          .    .    .           .          }
          .    .    .           .
 99,976,528    0    0  49,988,264          return 0;
      6,736    0    0       6,736  }
```

KCachegrind

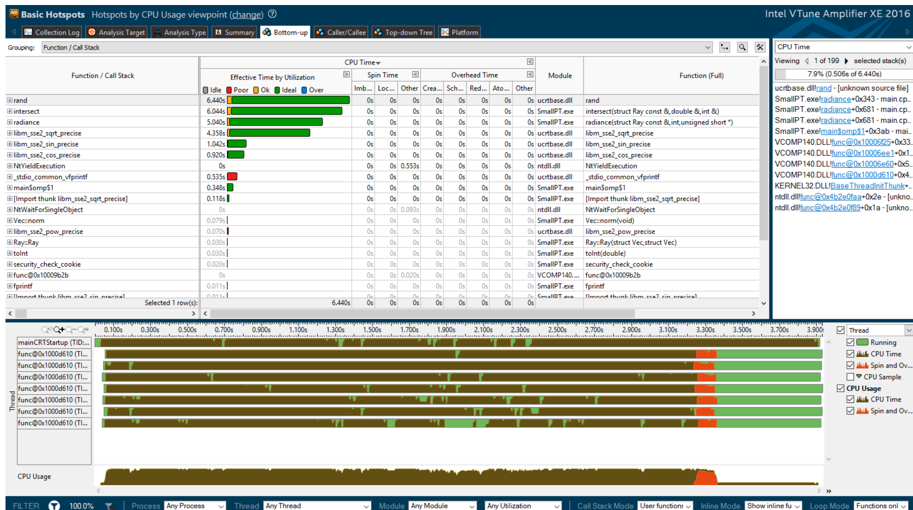


gprof

1. `gcc myprog.c -g -pg -o myprog`
2. `./myprog`
3. `gprof [options] myprog`

```
           2 ->{
                 register ulg c;

                 static ulg crc = (ulg)0xffffffffL;

           2 ->  if (s == NULL) {
           1 ->      c = 0xffffffffL;
           1 ->  } else {
           1 ->      c = crc;
           1 ->      if (n) do {
       26312 ->          c = crc_32_tab[...];
26312,1,26311 ->     } while (--n);
                 }
           2 ->  crc = c;
           2 ->  return c ^ 0xffffffffL;
           2 ->}
```

VTune



Profile Guided Optimization (PGO)

1. `gcc -march=native -O3 [...]`

   ```
   \time ./x enwik8
   1.18user 0.07system 0:01.25elapsed 100%CPU
   ```

2. `gcc -march=native -O3 -fprofile-generate [...]`
3. `./x enwik6`
4. `gcc -march=native -O3 -fprofile-use [...]`

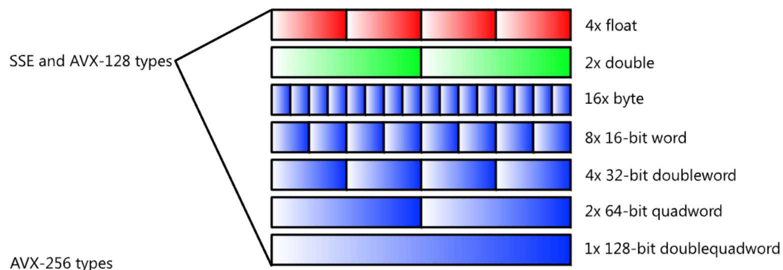
   ```
   \time ./x enwik8
   0.96user 0.07system 0:01.03elapsed 100%CPU
   ```

Link time optimization (LTO)

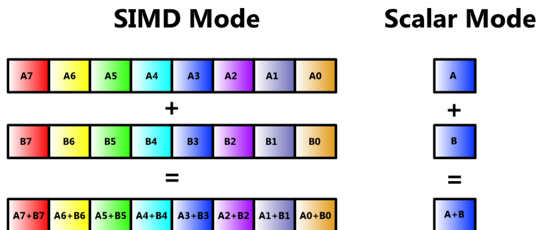
Extends the scope of interprocedural analysis from single source file to whole program

1. `gcc -flto -c f1.c`
2. `gcc -flto -c f2.c`
3. `gcc -flto -o f f1.o f2.o`

SIMD extensions



SIMD extensions



Data types

```
__m128 f; // { float, float, float, float }
__m128d d; // { double, double }
__m128i i; // 16 8-bit, 8 16-bit, 4 32-bit, or 2 64-bit ints
```

SIMD extensions

Example

```
__m128 r0, r1;

r1 = _mm_setzero_ps();

for (int i = (n >> 2); i > 0; --i)
{
        r0 = _mm_load_ps(in);
        in += 4;
        r1 = _mm_unpackhi_ps(r1, r0);
        r0 = _mm_shuffle_ps(r0, r1, 0x61);
        r0 = _mm_shuffle_ps(r0, r0, 0xC6);
        _mm_store_ps(out, r0);
        out += 4;
}
```

GCC vector extensions

```
typedef int v4si __attribute__ ((vector_size (16)));

v4si a, b, c;

c = a + b;

a = b + 1;      /* a = b + { 1, 1, 1, 1 }; */
a = 2 * b;      /* a = { 2, 2, 2, 2 } * b; */

c = a > b;      /* { 0, 0, -1, 0 } */
```

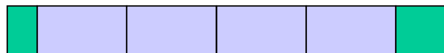
OpenMP

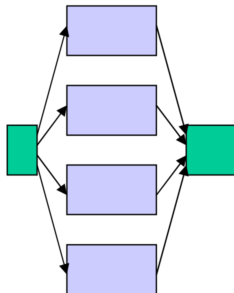
OpenMP 4.0+ (`#pragma omp simd ...`)

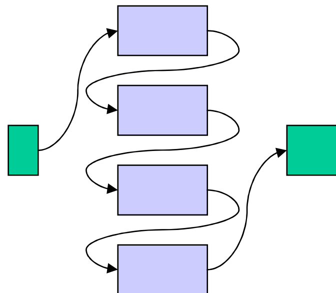
```
// add a[] and b[]
#pragma omp simd aligned(a,b,c:16)
for (int i = 0; i < N; ++i) {
        c[i] = a[i] + b[i];
}
```

```
// sum of arr[]
#pragma omp simd reduction(+:s)
for (int i = 0; i < N; ++i) {
        s += arr[i];
}
```

Parallel computing

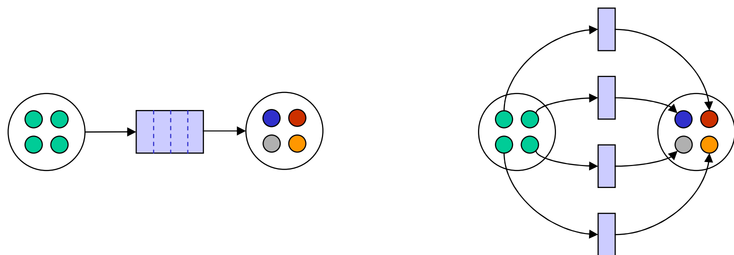






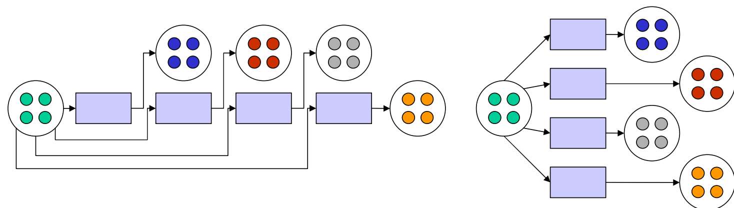
- divide the computation into several parts
- minimize dependencies and communication between these parts
- compute each part independently

Data parallelism



- performing the same operation on different (independent) data
- data can be divided between threads and processed in parallel
- parallelization limited by the amount of data

Task parallelism



- performing different (independent) operations on the same data
- each thread executes a different task
- parallelization limited by the number of independent tasks

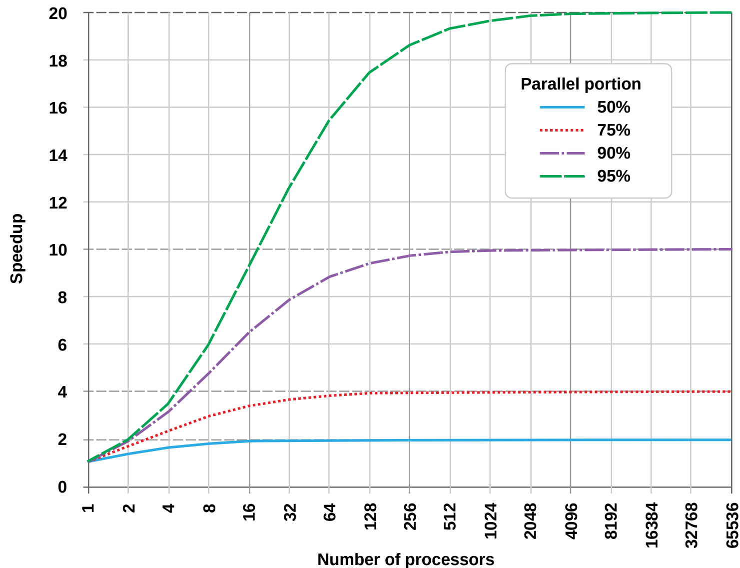
Amdahl's law

The expected speedup of the parallel code over the serial code

$$S(n) = \frac{1}{(1-P) + P/n}$$

- using n processors
- P is the proportion of a program that can be made parallel
- $(1-P)$ is the portion of that cannot be parallelized

Amdahl's law



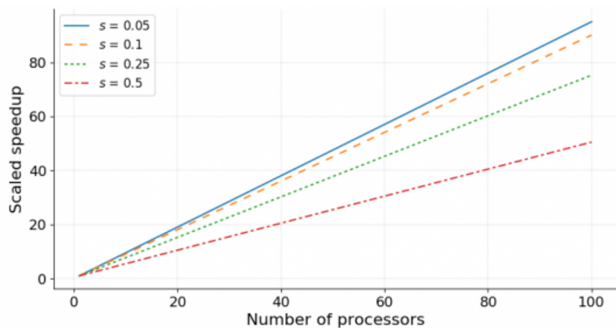
Gustafson's law

The expected speedup of the parallel code over the serial code

$$S(n) = (1 - P) + nP$$

- using n processors
- P is the proportion of a program that can be made parallel
- $(1 - P)$ is the portion of that cannot be parallelized

Gustafson's law



Parallelization levels

- instructions
  - hidden parallelism
  - SIMD instructions
- threads
  - stream of instructions that can be managed independently by the operating system
  - inside a single process, share the same memory space
  - OpenMP, pthreads, ...
- processes
  - operating system primitives, executed by one or multiple threads
  - separate memory space, the possibility to use shared memory
  - creation and switching is slower than in the case of threads
  - OpenMPI, ...
- computers (nodes)
  - processes running on physically separated machines
  - separated memory spaces (cannot be shared effectively)
  - OpenMPI, ...

POSIX threads

```
#include <pthread.h>

void *thread(void *arg)
{
        /* ... */
}

void foo()
{
        pthread_t tid;

        pthread_create(&tid, NULL, *thread, NULL);

        /* ... */

        pthread_join(tid, NULL);
}
```

OpenMP

- standardized thread extension of C/C++ languages
- support for data as well as task parallelism
- defines the behavior of variables in relation to threads
- defines synchronization and atomic primitives
- directives `#pragma omp ...` and functions from `omp.h`

```
#pragma omp parallel
{
	/* this block is executed by multiple threads */
}
```

OpenMP

```
#pragma omp parallel
{
        int id = omp_get_thread_num();
        int np = omp_get_num_threads();
        printf("thread␣%d␣of␣%d\n", id, np);
}
```

```
#pragma omp parallel for
for (int i = 0; i < N; ++i) {
        c[i] = a[i] + b[i];
}
```

```
int x = 0;
#pragma omp parallel for reduction(+:x)
for (int i = 0; i < N; ++i) {
        x = x + c[i];
}
```

OpenMP

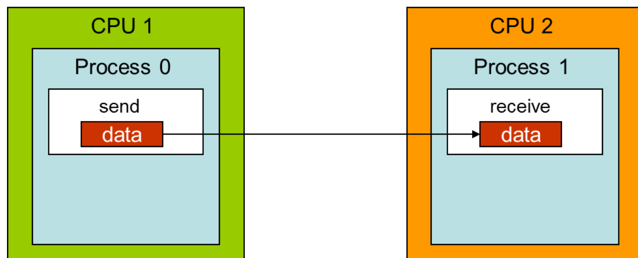
```
#pragma omp parallel sections
{
	#pragma omp section
	{
		printf("thread␣%d\n", omp_get_thread_num());
	}

	#pragma omp section
	{
		printf("thread␣%d\n", omp_get_thread_num());
	}
}
```

Output:

```
thread 0
thread 1
```

OpenMPI

- library for process communication and synchronization
- separate memory spaces
- message-based communication model
- communication via shared memory or network (ssh)



Portable Batch System (PBS)

- supercomputers
- submitting jobs (or job arrays)
- jobs are executed on multiple nodes (machines)
- specifying requested resources (memory, GPUs)

```
# run 1000 instances
qsub -J 1-1000 my_script.sh
```

OpenCL

OpenCL code

```
__kernel void worker(__global int *arr)
{
        /* thread */
}
```

Host code

```
clCreateProgramWithSource(context, 1, (const char **)&
   program_string, (const size_t *)&program_length, NULL);
clBuildProgram(program, 1, &dev_id[i], options, NULL, NULL);

clCreateKernel(program, "worker", NULL);

clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&arr);
clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, &
   problem_size, NULL, 0, NULL, NULL);
```

Sources

- U. Drepper. *What Every Programmer Should Know About Memory*, Red Hat, 2007
- Intel Intrinsics Guide
- Stack Overflow