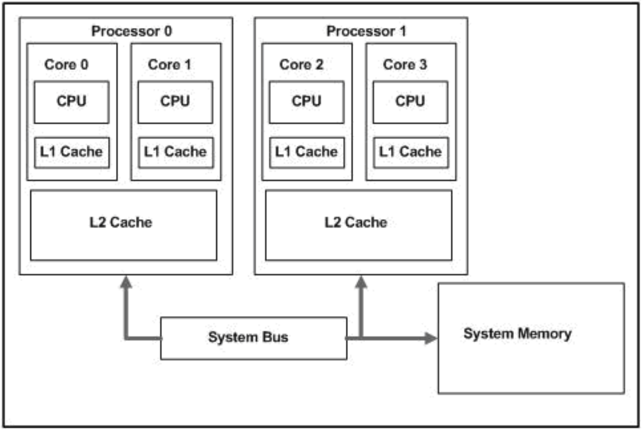
Introduction to parallel programming via OpenMP

Voronin K.

July 28, 2017

Contents

“Parallel” comes from ...



Amdahl's law

Gives theoretical speedup formula:

$$S_{\text{latency}} = \frac{1}{a + \frac{1-a}{p}}$$

where

- S_{latency} is the theoretical speedup of the algorithm,
- $a = \dfrac{\text{number of operations done in parallel}}{\text{total number of operations}} < 1$,
- p is the number of parallel workers.

Amdahl's law

It follows that

- $S_{\text{latency}}(s) \leq \frac{1}{a}$
- $\lim_{s \to \infty} S_{\text{latency}}(s) = \frac{1}{a}$

- Doesn't take into account many other factors: memory, various overheads, ...
- But can be used as a first estimate!

Example: dot product

Dot product

Given vectors x and y of length N compute

$$c = x \cdot y = \sum_{i=1}^{N} x_i y_i$$

where x_i, y_j - coordinates of x and y

Questions

- What is the total number of operations?
- What is the maximum speedup?
- What about memory?

Amdahl's law (modified)

$$S_{\text{latency}} = \frac{1}{a + \frac{1-a}{p} + c}$$

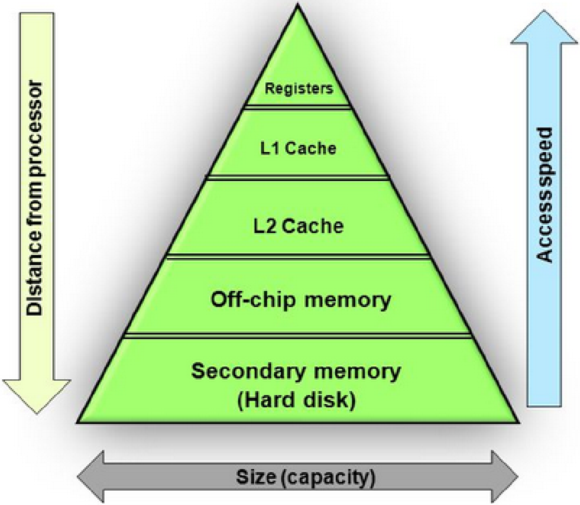
where:

- S_{latency} is the theoretical speedup of the execution of the whole task,
- $a = \dfrac{\text{number of operations done in parallel}}{\text{total number of operations}} < 1$,
- p is the number of parallel workers,
- c is the communication overhead (nonlinear!).

Main characteristics:

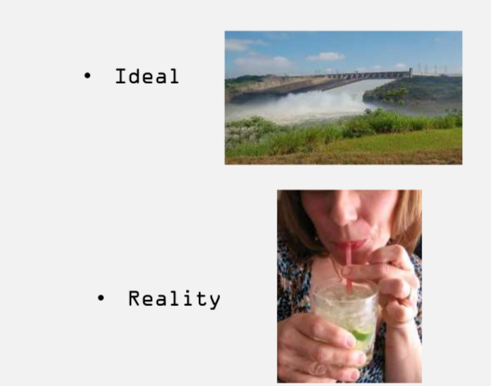
- flops = floating point operations per second
- memory access speed
- memory latency = amount of time to satisfy an individual memory request
- memory bandwidth = how much data a memory channel can transfer at once

Memory access speed



Memory bandwidth

True story (for graphical chips at least)



You cannot optimize more after you've reached the maximum memory bandwidth!

Measuring memory bandwidth

Theoretical bandwidth:
number of RAM sticks $\times$ bus width $\times$ memory clock frequency for one stick.

Experimental bandwidth (simplest variant):

```
void write_memory_loop(void* array, size_t size) {
  size_t* carray = (size_t*) array;
  size_t i;
  for (i = 0; i < size / sizeof(size_t); i++) {
    carray[i] = 1;
  }
}
```

Experimental bandwidth: accessed memory / time
Taken from:

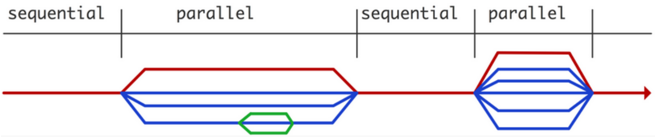
1) http://codearcana.com/posts/2013/05/18/achieving-maximum-memory-bandwidth.html

2) https://github.com/awreece/memory-bandwidth-demo

OpenMP

Two main concepts of OpenMP:

- the use of *threads*.
- *fork/join* model of parallelism.



Website: http://www.openmp.org/
Alternatives working on shared memory: TBB, Cilk (high-level), Pthreads (low-level)

OpenMP

Let's try "hello openmp world".

```
#pragma omp parallel private(nthreads, tid)
{
  tid = omp_get_thread_num();
  printf("Hello OpenMP world from thread = %d\n"
  if (tid == 0)
  {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads)
  }
}
```

Compiling:

- GNU: -fopenmp
- Intel: -openmp

Environment variable: export $OMP\_NUM\_THREADS = N$

OpenMP

Main tool: adding preprocessor directives #*pragma omp* tells the compiler how to set up threads.

Syntax

#pragma omp directive-name [clause[[,] clause]...] new- line
#pragma omp parallel

Memory specification

Default: variables declared outside parallel regions are shared, internal are private.

OpenMP

Example: vector dot product

Implementation 1

```
  sum = 0.0;
#pragma omp parallel for
  for (int i=0; i<N; ++i)
  {
    sum += a[i]*b[i];
  }
```

Wrong result!

Why? It's a race condition. All threads are trying to update sum at the same time. Let's correct that!

Implementation 2

```
  sum = 0.0;
#pragma omp parallel for
  for (int i=0; i<N; ++i)
  {
#pragma omp atomic
    sum += a[i]*b[i];
  }
```

OpenMP

If there is a special construction - it might be good idea to use that!

Implementation 3

```
  sum = 0.0;
#pragma omp parallel for reduction(+:sum)
  for (int i=0; i<N; ++i)
  {
    sum += a[i]*b[i];
  }
```

OpenMP

Warning: don't use more programming threads than your hardware allows!

Implementation 4

```
  sum = 0.0;
#pragma omp parallel for reduction(+:sum) \
num_threads(8)
  for (int i=0; i<N; ++i)
  {
    sum += a[i]*b[i];
  }
```

Less "defaults" - less errors! Define variable types explicitly.

Implementation 5

```
  sum = 0.0;
#pragma omp parallel for reduction(+:sum) \
shared(a,b,N)
  for (int i=0; i<N; ++i)
  {
    sum += a[i]*b[i];
  }
```

Let's try to add more flexibility. Dynamic should be good. Or ...?

Implementation 6

```
  sum = 0.0;
#pragma omp parallel for reduction(+:sum) \
  shared(a,b) schedule(dynamic)
  for (int i=0; i<N; ++i)
  {
    sum += a[i]*b[i];
  }
```

Think of how memory is read for a moment. Do you have any idea how to improve? Use blocks

Implementation 7

```
  sum=0.0;
  int M=32;
#pragma omp parallel for reduction(+:sum) \
 shared(a,b)
  for (int i=0; i<N/M; i++)
  {
    for (int j=i*M; j<(i+1)*M; j++)
    {
      sum+=a[j]*b[j];
    }
  }
```

OpenMP

Just an alternative.
Implementation 8

```
#pragma omp parallel sections
  {
#pragma omp section
    {
      for (int i=0; i<N/2; i++)
      {
        sum1+=a[i]*b[i];
      }
    }
#pragma omp section
    {
      for (int i=N/2; i<N; i++)
      {
        sum2+=a[i]*b[i];
      }
    }
  }
```

OpenMP

Things to learn:

- *#pragma omp atomic*
- *#pragma omp flush*
- *#pragma omp critical*
- *#pragma omp barrier*
- *#pragma omp single/master*
- KMP_AFFINITY
- locks and mutexes
- false sharing
- OpenMP tasks

OpenMP in Python

Bad news (AFAIK)

Python is not a friend with OpenMP because of the GIL = General Interpreter Lock. Python has GIL so only one thread is allowed to run at a given time.

Good news (AFAIK)

There are some Python extensions which can handle OpenMP: SciPy (to some extent), CPython, weave, other C extensions...

Thanks for coming!