A Brief Introduction to OpenMP

Yifan He

09-12-2010

Why Multi-core?

Additional transistors afforded by Moore's Law

Single core: maximize clock frequency , more ILP support, etc.?

- Problems of Single Core:
  - Power/Heat Dissipation issues (Frequency Wall)
  - Instruction-Level-Parallelism (ILP Wall)
  - Memory Wall

Why Multi-core?

Additional transistors afforded by Moore's Law

Single core: maximize clock frequency , More ILP support, etc.?

Multiple cores on a single silicon, and processing simultaneously

Performance scaling through parallel processing of Multi-Core!

How to migrate the single-core code to the multi-core processor conveniently?

Outlines

- About OpenMP
- “Hello World” example
- OpenMP programming model
- Step-by-step demo
- Application for the assignment

Why OpenMP?

- What we would like to have:
  - ✓ Automatic parallelization of sequential code
  - ✓ No additional parallelization effort for development, maintenance, etc.

OpenMP as a programming interface:

- ► Compiler directives `#pragma omp parallel`
- ► Library functions `omp_get_num_threads()`
- ► Environment variables `OMP_NUM_THREADS = 4`

"Hello World"

```
#include "omp.h"
void main()
{
   #pragma omp parallel
   {
      printf(" hello world \n");
   }
}
```

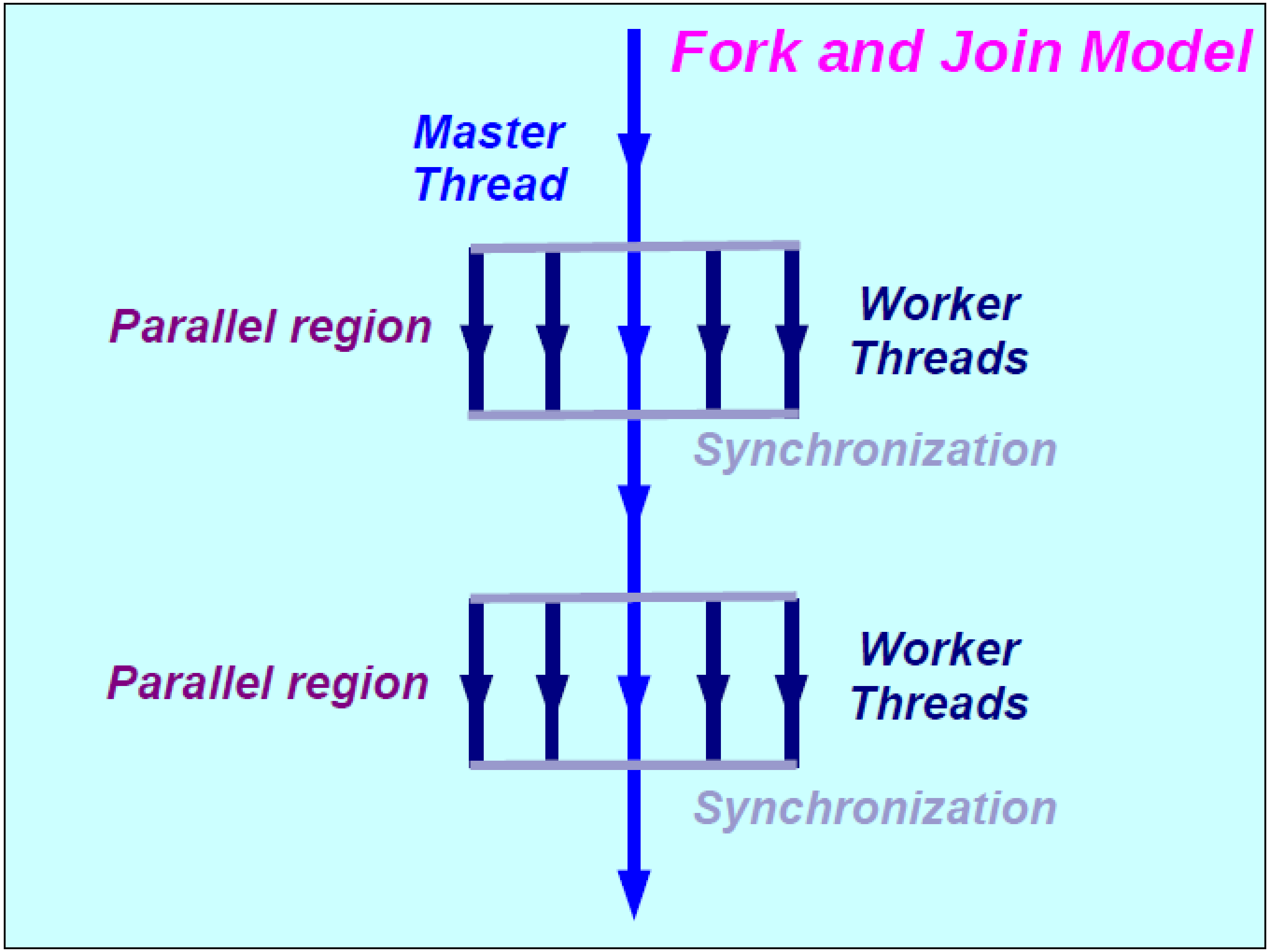
Compiler Directive

```
$ gcc  hello_world.c -o hello_world
$ ./hello_world
hello world
```

Environment Variable

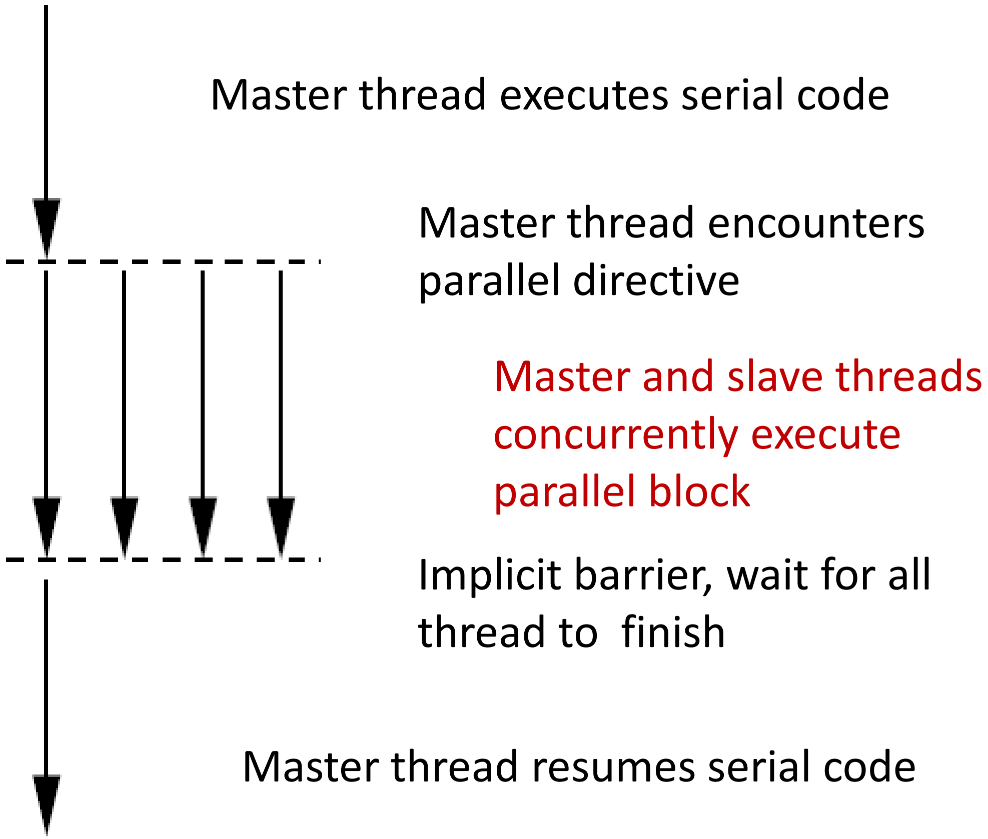
```
$ export OMP_NUM_THREADS=4
$ gcc -fopenmp hello_world.c
   -o hello_world_omp
$ ./hello_world_omp
hello world
hello world
hello world
hello world
```

Execution Model

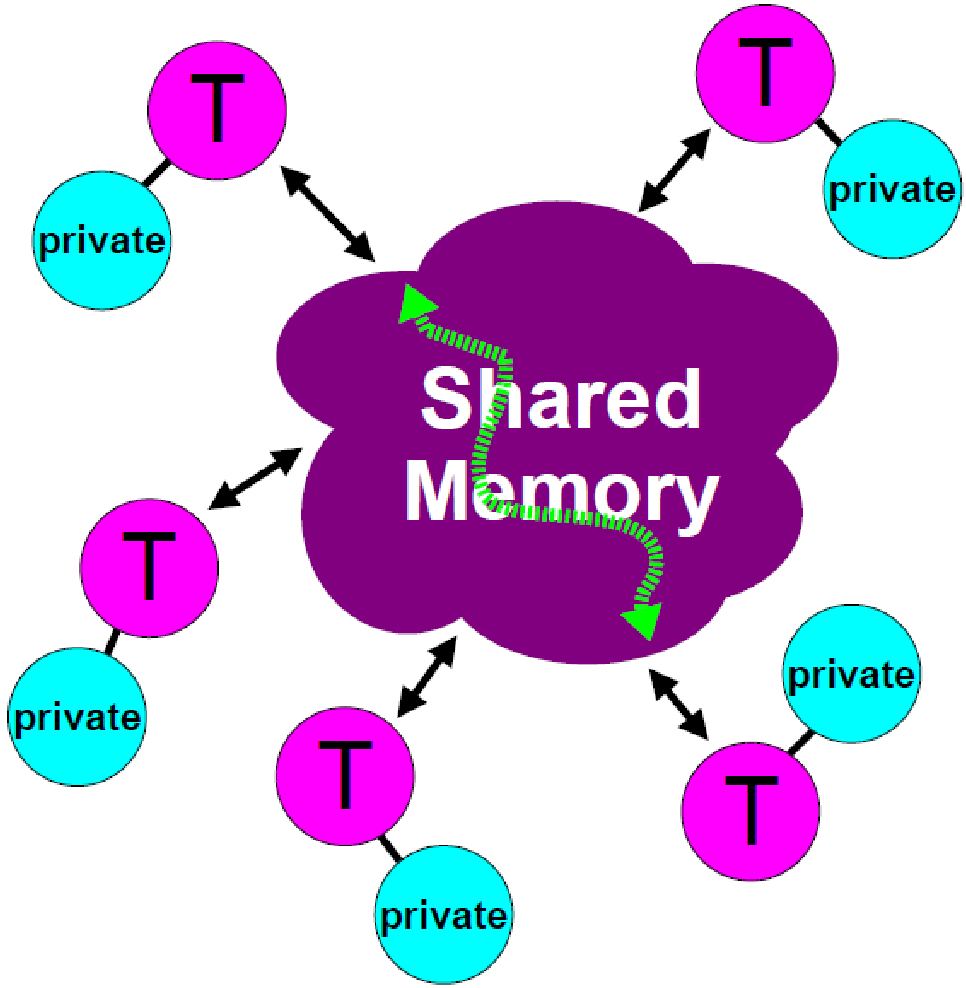


Execution Model

```
#include "omp.h"
void main()
{
    ...    // serial code
   #pragma omp parallel
   {
     printf(" hello world \n");
   }
   ...     // serial code
}
```

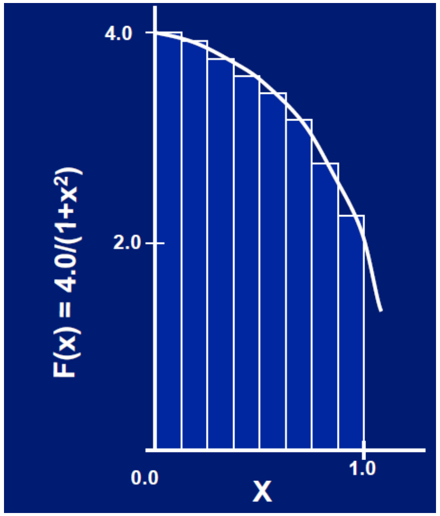


OpenMP Memory Model



Step-by-Step Demo: calculate π

```
static long num_steps = 100000000;
double step;

void main ()
{
  int i;
  double x, pi, sum = 0.0;

  step = 1.0/(double) num_steps;

  for ( i = 1; i <= num_steps; i++ ) {
    x = (i - 0.5) * step;
    sum = sum + 4.0/(1.0 + x * x);
  }

  pi = step * sum;
  printf("\n pi with %d steps is %f \n", num_steps, pi);
}
```



Mathematically,

$$\pi = \int_0^1 \frac{4.0}{(1+x^2)} dx$$

$$\pi \approx \sum_{i=0}^{num\_steps} F(x_i) * \Delta x$$

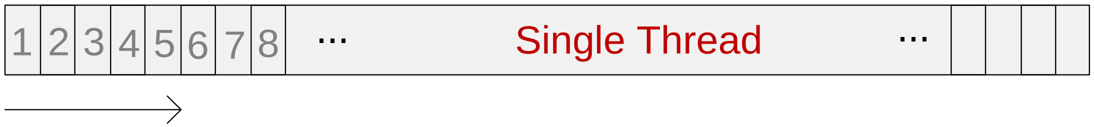
```
$ ./pi
pi = 3.141593,  in 0.953144 Sec.
```

Processing time

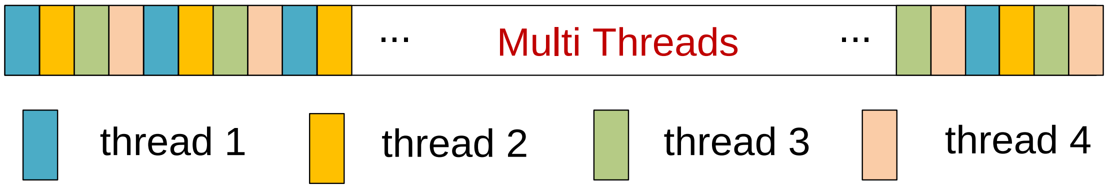
OpenMP: using SPMD pattern

```
static long num_steps = 100000000;

for ( i = 1; i <= num_steps; i++ ) {
  x = (i - 0.5) * step;
  sum = sum + 4.0/(1.0 + x * x);
}
```

Total Workload (num_steps = 100,000,000)



Total Workload (num_steps = 100,000,000)



OpenMP: using SPMD pattern

```
static long num_steps = 100000000;
double partial_sum[numthreads];
```

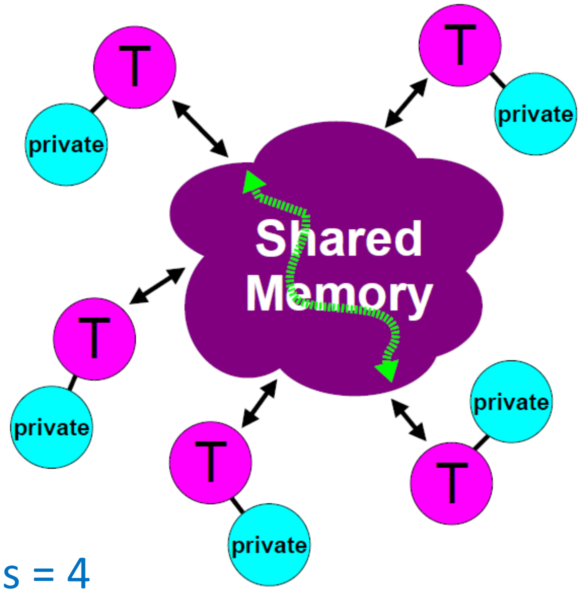
//# numthreads = 4

```
#pragma omp parallel
{
   int i;
   double x;
   int id = omp_get_thread_num();

   partial_sum[id] = 0.0;

   for (i = id; i < num_steps; i += numthreads) {
     x = (i + 0.5) * step;
     partial_sum[id] = partial_sum[id] + 4.0/(1.0 + x * x);
   }
}

full_sum = 0.0;
for( i = 0; i < numthreads; i++)
   full sum += partial sum[i];
```

```
$ export OMP_NUM_THREADS=4
$ ./pi_spmd_simple
pi = 3.141593,  in 4.412447 S
```

Processing Time:
from 0.953144 Sec. (Single thread)
to **4.412447 Sec. (4 threads)!!!**
Why?

False-Sharing




```
double partial_sum[numthreads];   //# numthreads = 4

partial_sum[id] = 0.0;

for (i = id; i < num_steps; i += numthreads) {
  x = (i + 0.5) * step;
  partial_sum[id] = partial_sum[id] + 4.0/(1.0 + x * x);
}
```

Each thread has its own *partial_sum[id]*   *(id = 1 for thread 1, …, id = 4 for thread 4).*

However, since it's defined as an array, the partial sums happen to be in consecutive memory locations, and be loaded into the *same cache line*.

Remove False-Sharing

```
static long num_steps = 100000000;  //# numthreads = 4

#pragma omp parallel
{
  int i;
  double x;
  int id = omp_get_thread_num();

  double partial_sum = 0.0;

  for ( i = id; i < num_steps; i += numthreads ) {
    x = (i + 0.5) * step;
    partial_sum = partial_sum + 4.0/(1.0 + x * x);
  }


  #pragma omp critical
  full_sum += partial_sum;
}
```

```
$ ./pi_spmd_no_false_sharing
pi = 3.141593,  in 0.253590 Sec.
```

Compiler Directive, indicate that it's a critical region. Check the learning material for detail

OpenMP: *loop*

```
static long num_steps = 100000000;  //# numthreads = 4

#pragma omp parallel
{
  #pragma omp for private(x) reduction(+:sum)
  for( i = 1; i <= num_steps; i++ ){
    x = (i - 0.5) * step;
    sum = sum + 4.0/(1.0 + x * x);
  }
}
```

***reduction:* Check the learning material for detail**

```
$ ./pi_loop
pi = 3.141593,  in 0.245648 S
```

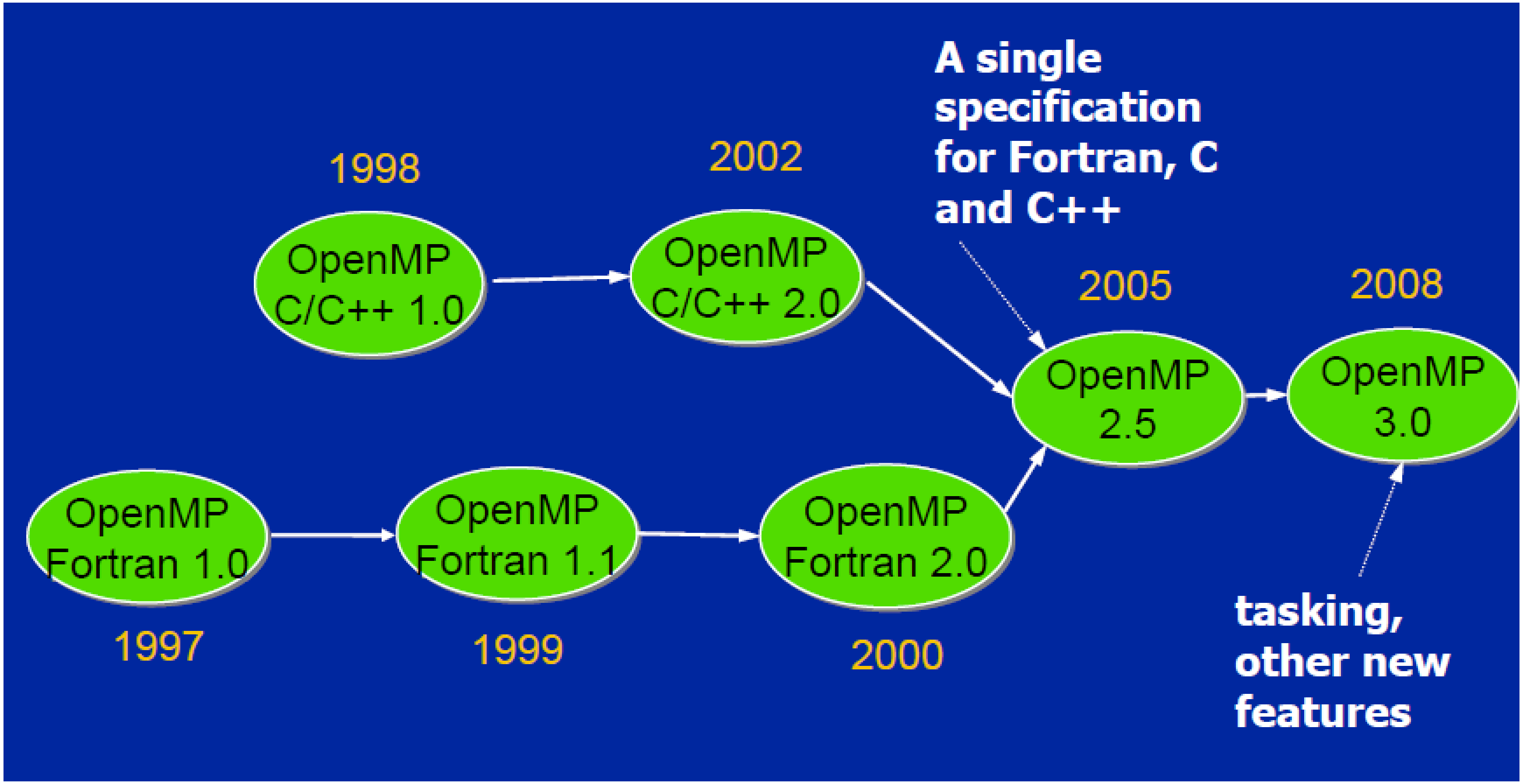
What We Learned

- Parallelism does not always guarantee performance improvement
- Assess data dependences is the difficult part

Other Important Contents

- *Variable Type*: shard, private, firstprivate, etc.
- *Synchronization*: atomic, ordered, barrier, etc.
- *Scheduling*: static, dynamic, guided
- *…*

OpenMP Release History

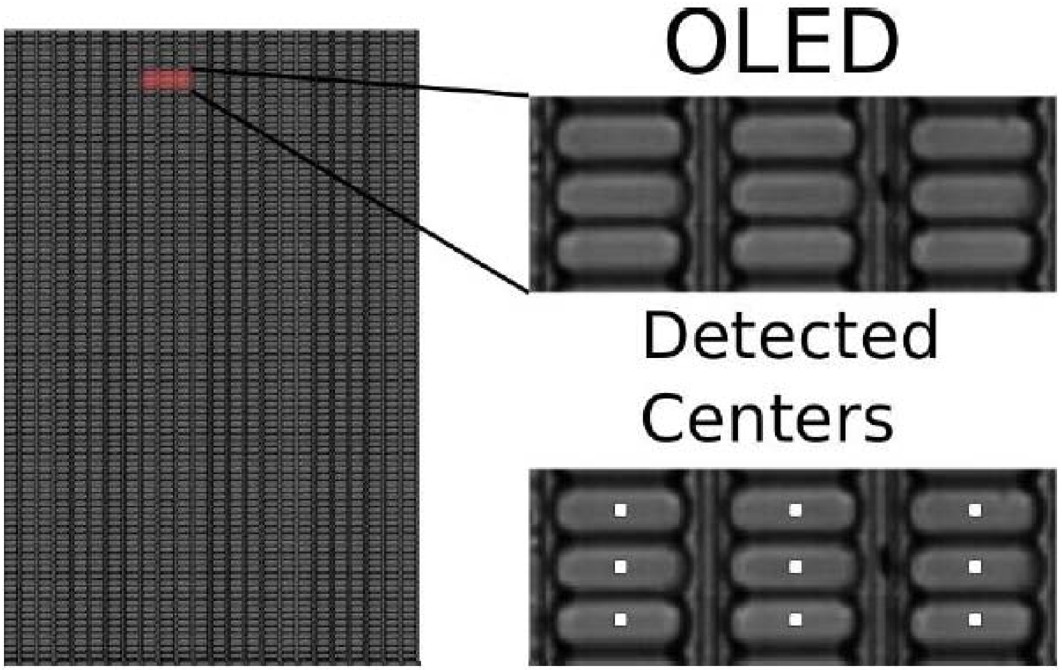


Is de-facto standard!

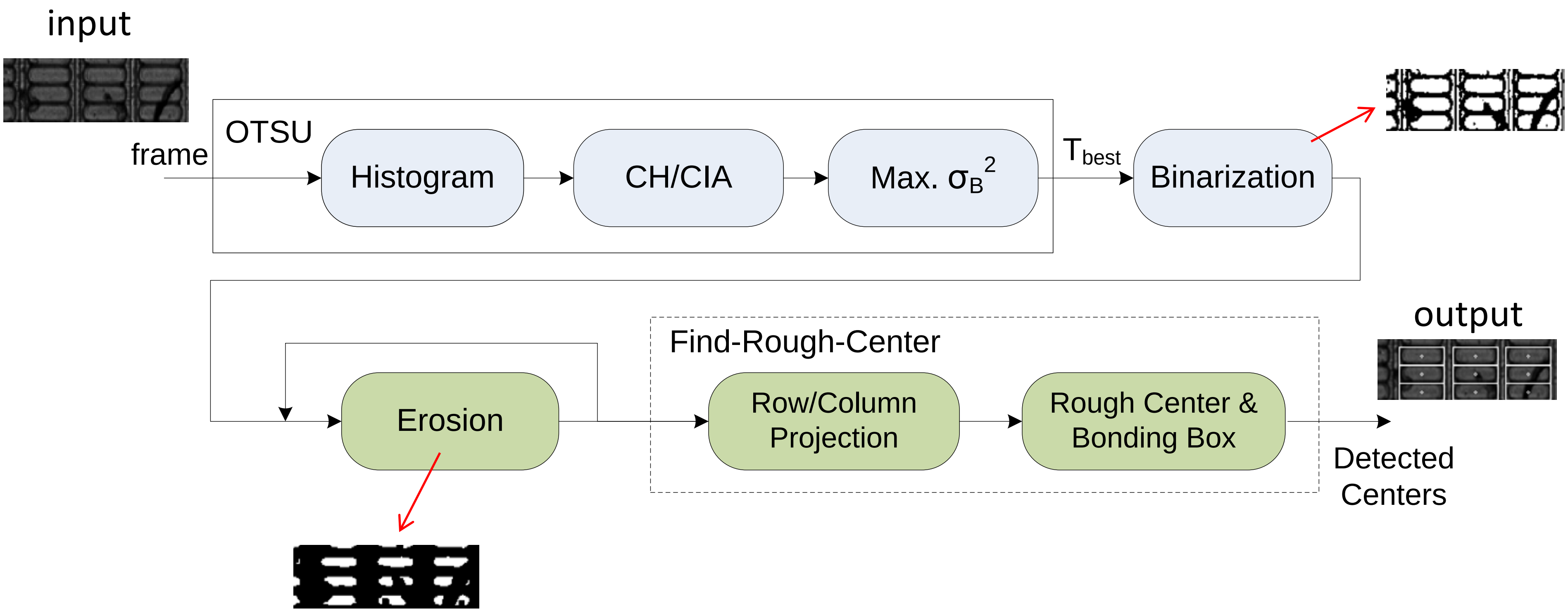
Assignment: *Application*

- From industrial OLED-Printing application

Organic-Light-Emitting-Diode (OLED) substrate localization



Assignment: *Application*



Assignment Website

- sites.google.com/site/omp5md00

Refernces:

[1] Tim Mattson and Larry Meadows, “Hands-On Introduction to OpenMP”

[2] Ruud van der Pas, “An Overview of OpenMP”, 2009

[3] Clemens Grelck, “Low-Level Multi-Core Programming with OpenMP”, 2010