System Programming: Threads

Raymond Namyst

Dept. of Computer Science

University of Bordeaux, France

https://gforgeron.gitlab.io/progsys/

Communication between processes

- Processes have private address spaces
  - They don’t seem to share any data
    - Actually, they do (mostly in read-only mode, e.g. code)

- Exchanging data between processes is painful… and slow!
  - BTW: Signals are not aimed at communicating rich information
  - Pipes: system calls are slow

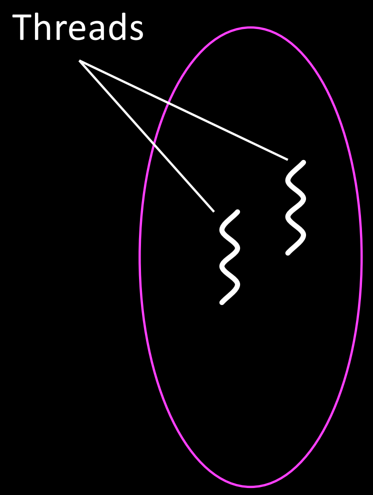
- Except with mmap…

Address space and execution flow

- Many applications spawn multiple processes to speed up execution
  - Perform many I/O intensive tasks concurrently
  - Perform tasks in parallel over multicore architectures
- But process creation/destruction is slow
  - Memory allocation + deallocation + initialization
- We only want to start a new activity
  - Sharing data is bonus

Threads

- Threads = Execution flow
- Process = Thread + Address Space
- Several threads can share the same address space



Threads

Process featuring 2 threads

Our first "hello thread" program

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


void *thread_func (void *arg)
{
  printf ("%s from thread!\n", arg);

  return NULL;
}
```

```c
int main (int argc, char *argv[])
{
  pthread_t pid;
  pthread_create (&pid, NULL, thread_func, "Hello");

  printf ("Hello from main\n");

  return 0;
}
```

Our first “hello thread” program

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


void *thread_func (void *arg)
{
  printf ("%s from thread!\n", arg);

  return NULL;
}
```

```c
int main (int argc, char *argv[])
{
  pthread_t pid;
  pthread_create (&pid, NULL, thread_func, "Hello");

  printf ("Hello from main\n");

  pthread_join (pid, NULL);

  return 0;
}
```

Creating a group of threads

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int NBTHREADS = 10;

void *thread_func (void *arg)
{
  int me = arg;

  printf ("Hello from thread %d\n", me);

  return NULL;
}
```

```
int main (int argc, char *argv[])
{
  if (argc > 1)
    NBTHREADS = atoi (argv[1]);

  pthread_t pids[NBTHREADS];

  for (int i = 0; i < NBTHREADS; i++)
    pthread_create (&pids[i], NULL, thread_func, i);

  printf ("Hello from main\n");

  for (int i = 0; i < NBTHREADS; i++)
    pthread_join (pids[i], NULL);

  return 0;
}
```

Creating a group of threads

- Useful when decomposing computation is smaller parts
    - Each thread must decide which part it should address
        - Easier if threads are numbered [0..N-1]
        - See ”spin” kernel, under the EasyPAP environment

Parallelizing computations

- The ”spin” kernel involves independent computations on the elements of an array
  - Trivially parallel
- Our first work distribution strategy assigns horizontal stripes of (approximately the same number of) pixels to threads

- TODO: extend spin.c!

Parallelizing computations

- The "*spin*" kernel involves independent computations on the elements of an array
  - Trivially parallel

- Our first work distribution strategy assigns horizontal stripes of (approximately the same number of) pixels to threads

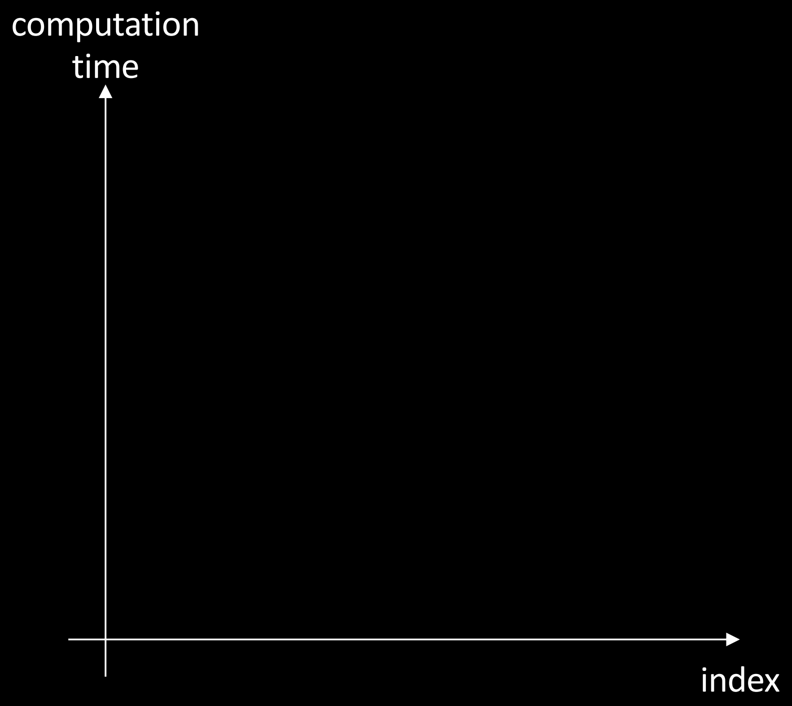
```
void *thread_starter (void *arg)
{
  …

  for (int i = line; i < line + slice; i++)
    for (int j = 0; j < DIM; j++)
      cur_img (i, j) = compute_color (i, j);

  return NULL;
}
```

Parallelizing computations

- Why did we choose a static *block* distribution?



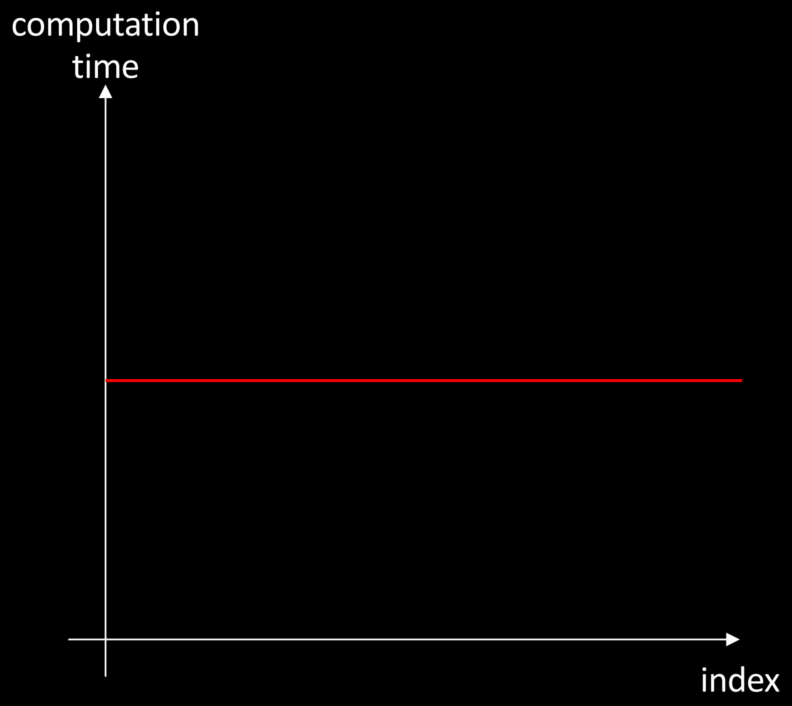
Parallelizing computations

- Why did we choose a static *block* distribution?
  - Because we assumed that the computation time of “compute_color” is constant
    - I.e. does not depend on (i, j)

- Let us consider a 1D example

```
float tab [MAX];

for (int i = 0; i < MAX; i++)
  tab [i] = f (i);
```



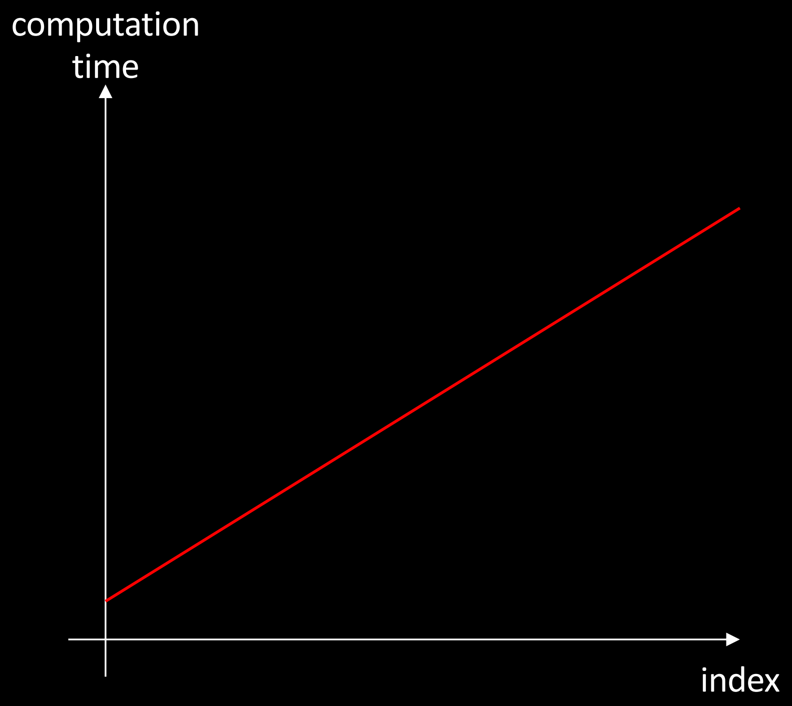
Parallelizing computations

- Let us consider a 1D example

```
float tab [MAX];

for (int i = 0; i < MAX; i++)
  tab [i] = f (i);
```

- What if the computation time is linearly increasing?



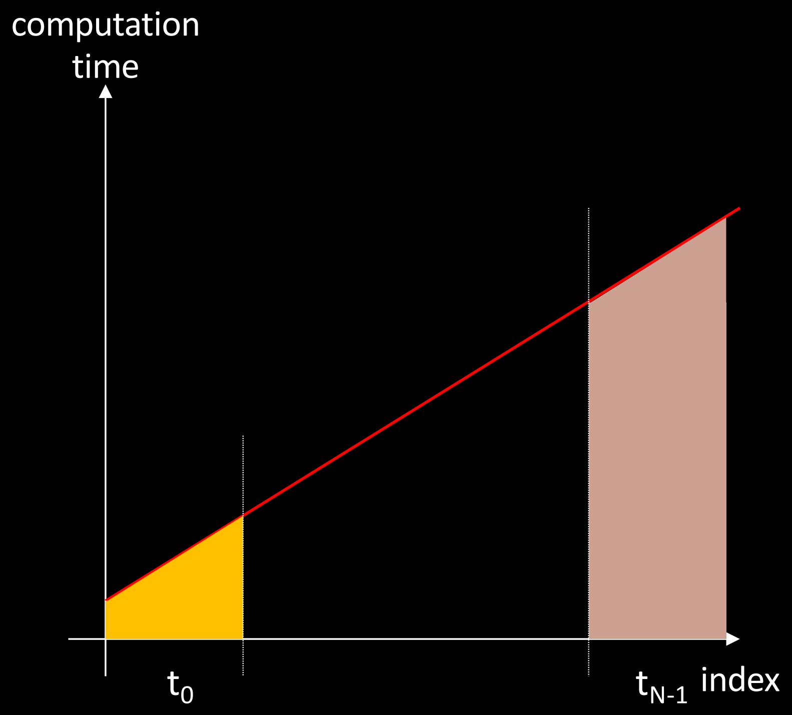
Parallelizing computations

- Let us consider a 1D example

```
float tab [MAX];

for (int i = 0; i < MAX; i++)
  tab [i] = f (i);
```

- What if the computation time is linearly increasing?
  - Our block distribution is no longer relevant



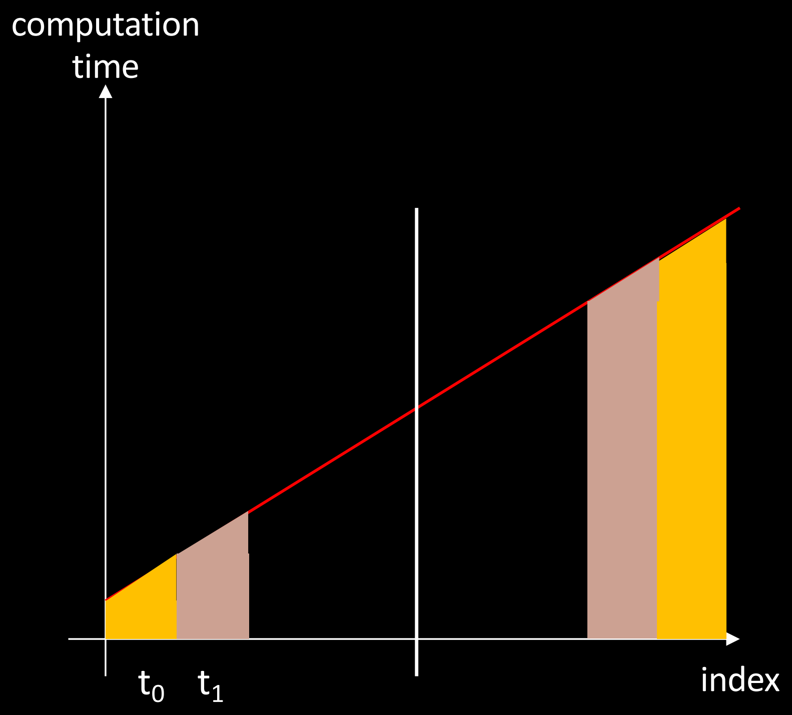
Parallelizing computations

- Let us consider a 1D example

```
float tab [MAX];

for (int i = 0; i < MAX; i++)
  tab [i] = f (i);
```

- What if the computation time is linearly increasing?
  - Our block distribution is no longer relevant
    - Well, using a mirror block distribution assigning two blocks per thread would work…



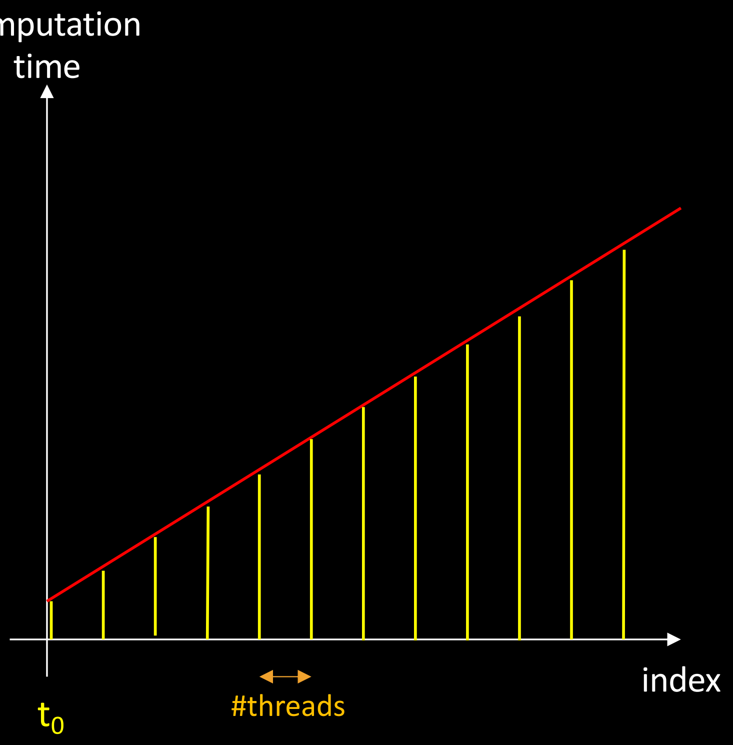
Parallelizing computations

- Let us consider a 1D example

```
float tab [MAX];

for (int i = 0; i < MAX; i++)
  tab [i] = f (i);
```

- What if the computation time is linearly increasing?
  - A cyclic distribution of indexes would be a good option



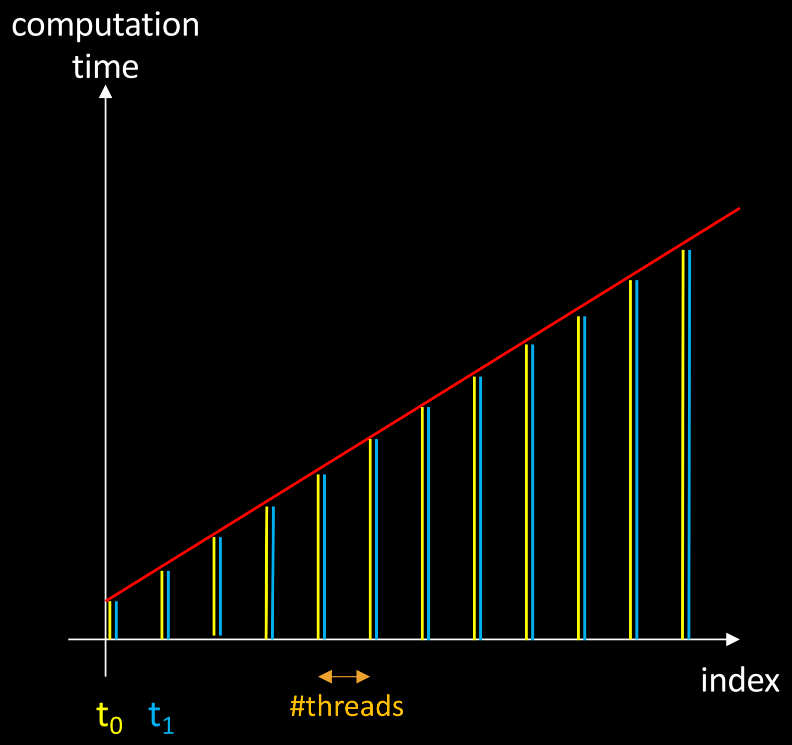
Parallelizing computations

- Let us consider a 1D example

```
float tab [MAX];

for (int i = 0; i < MAX; i++)
  tab [i] = f (i);
```

- What if the computation time is linearly increasing?
  - A cyclic distribution of indexes would be a good option



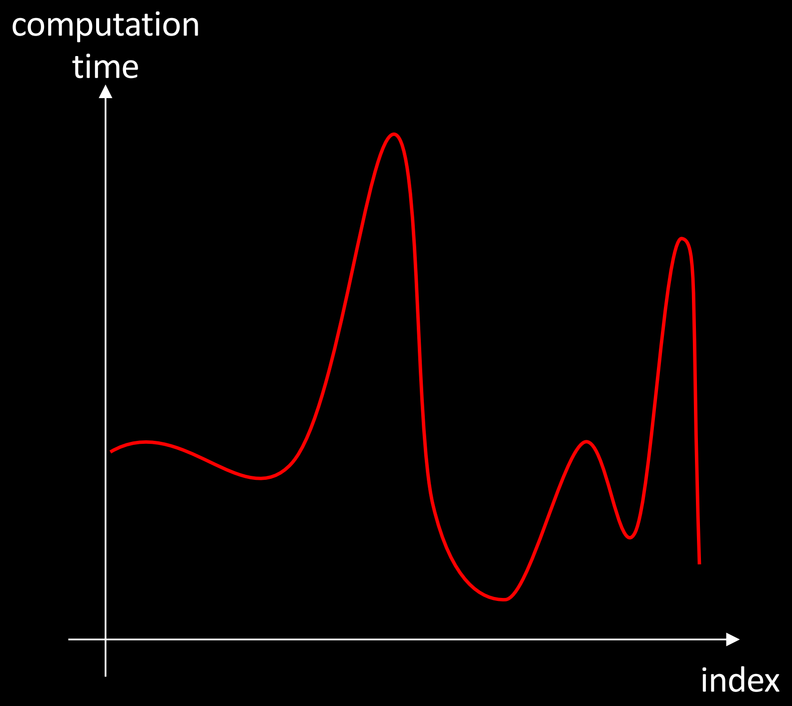
Parallelizing computations

- Let us consider a 1D example

```
float tab [MAX];

for (int i = 0; i < MAX; i++)
  tab [i] = f (i);
```

- What if the computation time is unpredictable?
  - Even the cyclic strategy may fail


computation time
index