



OPTIMIZATIONS CHEATS

Parallel Programming

Wissam HLAYHEL

OPTIMIZATION CHEATS

Use private/local variable

Let each parallel task works on private data as much as possible, and delay pushing result to shared memory to the end. Avoiding memory access by using local (register allocated) variables is also very beneficial.

Avoid false sharing

Even Task data are not logically shared, they can share same physical memory cache line. Each time a task write a data, it will invalidate the whole cache line on others cache. We can use some data padding to distance task data and avoid false sharing.

Use performant synchronization

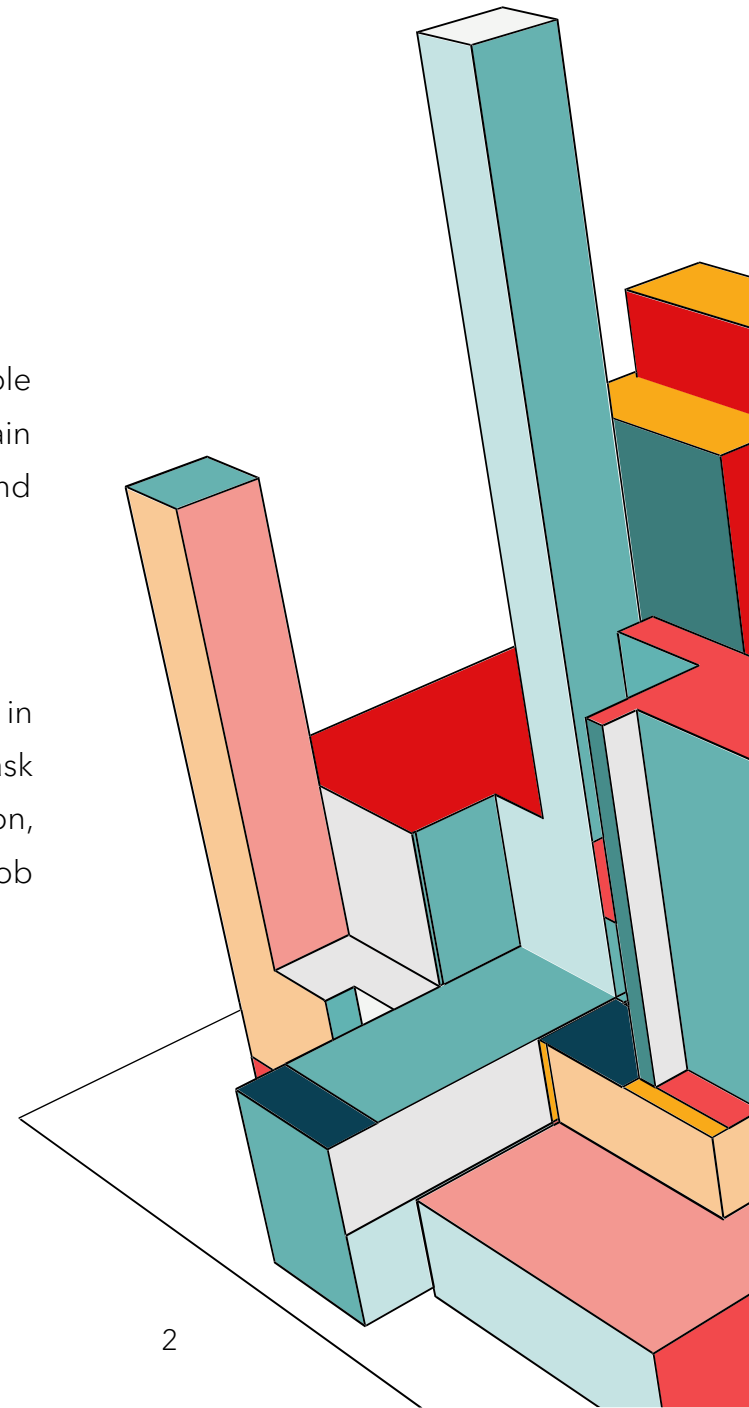
Synchronization is a parallelism bottleneck, it must be limited to minimum, and divided granularly. We must also use performant constructs (for example POSIX mutex are more performant than semaphore).

Use Cache Blocking Technique

Try to keep your data in cache and take whole benefic of them before they get urged to main memory. Optimize your code to use temporal and special memory locations.

Load balancing

Job to be processed can not always be divided in equivalent time sub-jobs. Statical Task decomposition in this case is not a best solution, try load balancing using dynamical sub-job allocation.



TESTS CONFIGURATION

Tests in next slides are done on a PC with following configuration (with 4 threads when applicable)

MacBook Pro (Retina, 15-inch, Mid 2015)
Processor 2.2 GHz Quad-Core Intel Core i7
Memory 16 GB 1600 MHz DDR3
Startup Disk SSD Disk

USE PRIVATE/LOCAL VARIABLE

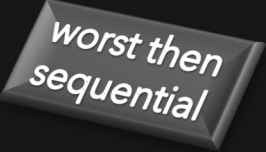
* Parallel counting of value 3 occurrences in an array with multi-threads (each handles a sub-set of the array)

Solution 1: using global shared counter

```
int count = 0;
int *array;
pthread_mutex_t mutex;

void *thread_counter_shared(void *data)
{
    int *idx = (int *)data;
    int id = *idx;

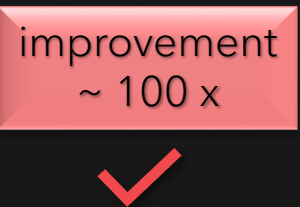
    int length_per_thread = N / t;
    int start = id * length_per_thread;
    for (int i = start; i < start + length_per_thread; i++)
    {
        if (array[i] == 3)
        {
            pthread_mutex_lock(&mutex);
            count += 1;
            pthread_mutex_unlock(&mutex);
        }
    }
    return (NULL);
}
```



Solution 2 : using local counter per thread, then global summing

```
void *thread_counter_local(void *data)
{
    int *idx = (int *)data;
    int id = *idx;
    int local_counter = 0;

    int length_per_thread = N / t;
    int start = id * length_per_thread;
    for (int i = start; i < start + length_per_thread; i++)
    {
        if (array[i] == 3)
        {
            local_counter++;
        }
    }
    pthread_mutex_lock(&mutex);
    count += local_counter;
    pthread_mutex_unlock(&mutex);
    return (NULL);
}
```



AVOID FALSE SHARING

* Alternative solution to previous counter 3 program, that use private counter per thread, and shows effect of false sharing



improvement
~ 5 x

Solution 1: using global array of separate counters

```
int part_counter[t];
void *thread_counter_private_falseShared(void *data)
{
    int *idx = (int *)data;
    int id = *idx;

    part_counter[id] = 0;

    int length_per_thread = N / t;
    int start = id * length_per_thread;
    for (int i = start; i < start + length_per_thread; i++)
    {
        if (array[i] == 3)
        {
            part_counter[id]++;
        }
    }

    pthread_mutex_lock(&mutex);
    count += part_counter[id];
    pthread_mutex_unlock(&mutex);
    return (NULL);
}
```

Solution 2 : Adding a padding of 64 bytes to array elements

```
#define PAD 16
int part_counter_padded[t * PAD];
void *thread_counter_private_padded(void *data)
{
    int *idx = (int *)data;
    int id = *idx;

    part_counter[id] = 0;

    int length_per_thread = N / t;
    int start = id * length_per_thread;
    for (int i = start; i < start + length_per_thread; i++)
    {
        if (array[i] == 3)
        {
            part_counter[id * PAD]++;
        }
    }

    pthread_mutex_lock(&mutex);
    count += part_counter[id * PAD];
    pthread_mutex_unlock(&mutex);
    return (NULL);
}
```

USE CACHE BLOCKING TECHNIQUE

* Calculating the power 2 of a Matrix N x N



improvement
greater than 2 x

Solution 1: Classical 3 loops

```
for (int i = 0; i < N; i++)
{
    for (int j = 0; j < N; j++)
    {
        for (int k = 0; k < N; k++)
        {
            power_matrix[i][j] = power_matrix[i][j] + matrix[i][k] * matrix[k][j];
        }
    }
}
```

OUTPUT DEBUG CONSOLE TERMINAL JUPYTER tcsh

MacBook-Pro:CS6447-Parallel_prog/Experiments/pthread_task] wissamhlayhel% ./a.out
000000 , Execution time 6925 millis

Solution 2 : Switching internal loop is very beneficial due to blocking cache lines till finishing using them.

```
for (int i = 0; i < N; i++)
{
    for (int k = 0; k < N; k++)
    {
        for (int j = 0; j < N; j++)
        {
            power_matrix[i][j] = power_matrix[i][j] + matrix[i][k] * matrix[k][j];
        }
    }
}
```

OUTPUT DEBUG CONSOLE TERMINAL JUPYTER tcsh

MacBook-Pro:CS6447-Parallel_prog/Experiments/pthread_task] wissamhlayhel% ./a.out
000000 , Execution time 3030 millis

LOAD BALANCING

✓ improvement
of 1.7 x

Doing calculation on lines of matrix with variable length and processing time.

```
void allocate_init()
{
    matrix = malloc(N * sizeof(int *));
    vect = malloc(N * sizeof(int));
    int m;
    for (int i = 0; i < N; i++)
    {
        m = 100+i*50 ;/* (1 + rand() % 10);
        matrix[i] = malloc(m * sizeof(int));
        for (int j = 0; j < m - 1; j++)
            matrix[i][j] = rand()%10000;
        matrix[i][m - 1] = -1; // terminator value
    }
}
```

Solution 1: Static balancing

```
void *calculate_task(void *data)
{
    int *idx = (int *)data;
    int id = *idx;
    int start_idx = id * (N / T);
    int end_idx = start_idx + N / T - 1;
    if (id == (T - 1))
        end_idx = N - 1;
    for (int i = start_idx; i <= end_idx; i++)
    {
        int count = 0;
        for (int j = 0; matrix[i][j] != -1; j++)
        {
            if (is_prime(matrix[i][j]))
                count++;
        }
        vect[i] = count;
    }
    return NULL;
}
```

Solution 2: Dynamic Balancing

```
void *calculate_task_dyna(void *data)
{
    while (1)
    {
        int i;
        pthread_mutex_lock(&lock);
        i = counter;
        counter++;
        pthread_mutex_unlock(&lock);
        if (i >= N)
            break;
        int count = 0;
        for (int j = 0; matrix[i][j] != -1; j++)
        {
            if (is_prime(matrix[i][j]))
                count++;
        }
        vect[i] = count;
    }
    return NULL;
}
```

USE PERFORMANT SYNCHRONIZATION

- Use Mutex rather than Semaphore for mutual exclusion

Solution 1: protect shared counter with semaphore

```
sem_t semaphore;
void *thread_semaphore(void *data)
{
    for (int i = 0; i < 100000; i++)
    {
        sem_wait(&semaphore);
        shared_counter++;
        sem_post(&semaphore);
    }
    return NULL;
}
```

Solution 2 : protect shared counter with mutex

```
pthread_mutex_t mutex;
void *thread_mutex(void *data)
{
    for (int i = 0; i < 100000; i++)
    {
        pthread_mutex_lock(&mutex);
        shared_counter++;
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}
```

~ 3-4 x faster

- Do not use same mutex for independent mutual exclusion sections (like different shared variable)