TDA - Fila

Andressa Andrade, Guilherme Bernal, Renata Antunes e Rodrigo Fernandes

Operações básicas

- queue_alloc → Aloca memoria para fila
- queue_free → Libera espaço na memoria
- queue_push → Adiciona um item no fim da fila
- queue_shift → Remove frente da fila

Operações opcionais

- queue_sort → Ordena fila
- queue_front → Exibe o primeiro elemento
- queue_last → Exibe o último elemento
- queue_each → Exibe a fila inteira
- queue_find → Descobre a posição do elemento, caso este exista
- queue_nth → Busca elemento por índice
- queue_size → Verificar o tamanho da fila

Queue.h

```
struct queue_t {
    struct queue_block_t* first;
    struct queue_block_t* last;
    unsigned size;
    unsigned space;
};

struct queue_block_t {
    struct queue_block_t* next;
    void* data[BLOCK_SIZE];
};
```

Implementação da Fila com lista encadeada

```
queue* queue_alloc(void);
void queue_free(queue* qu);
void  queue_push(queue* qu, void* el);
void* queue_shift(queue* qu);
```

Operações básicas

```
void queue_sort(queue* qu, int(*comp)(void*, void*));
void* queue_front(queue* qu);
void* queue_last(queue* qu);
void* queue_nth(queue* qu, unsigned i);
unsigned queue_find(queue* qu,int(*test)(void*));
void queue_each(queue* qu, void(*action)(void*));
```

Operações opcionais

Complexidade

O(1)

Criar fila vazia
Adicionar elemento
Remover frente
Ler frente
Ler trás
Verificar tamanho da fila

O(n)

Liberar fila
Buscar elemento em posição x
Descobrir posição de elemento
Exibir fila inteira

O(n logn)

Ordenar por Quick Sort

Queue.c

básicas

```
queue* queue_alloc(void) {
    queue* qu = malloc(sizeof(queue));
    memset(qu, 0, sizeof(queue));
    return qu;
}
```

```
void queue_free(queue* qu) {
        struct queue_block_t* current = qu->first;
          while (current != qu->last)
         free(current++);
         free(qu);
}
```

```
static void add_block(queue* qu) {
    struct queue_block_t* blk = malloc(sizeof(structqueue_block_t));
    memset(blk, 0, sizeof(struct queue_block_t));
    if (qu->last)
          qu->last->next = blk;
          qu->last = blk;
    if (qu->first == 0)
        qu->first = blk;
  }
```

```
static void remove_block(queue* qu) {
    struct queue_block_t* blk = qu->first;
    qu->first = blk->next;
    if (qu->first == 0)
        qu->last = 0;
    free(blk);
    qu->space = 0;
}
```

Queue.c

básicas

```
void queue_push(queue* qu, void* el) {
    if ((qu->space+qu->size) % BLOCK_SIZE == 0)
        add_block(qu);
        qu->last->data[(qu->space + qu->size) % BLOCK_SIZE] =
el;
        ++qu->size;
}

void* queue_shift(queue* qu) {
    void* front = queue_front(qu);
    --qu->size;
    ++qu->space;
    if (qu->space == BLOCK_SIZE)
        remove_block(qu);
    return front;
}
```

```
static struct queue_block_t* advance_block(struct queue_block_t*source,
unsigned i)
 {
    while (i >= BLOCK_SIZE)
    {
        source = source->next;
        i -= BLOCK_SIZE;
    }
    return source;
}
```

Queue.c

opcionais

```
unsigned queue_size(queue* qu) {
    return qu->size;
}

void* queue_front(queue* qu) {
    return qu->first->data[qu->space];
}

void* queue_last(queue* qu) {
    return qu->last->data[(qu->space + qu->size - 1) % BLOCK_SIZE];
}

void print_element(void* el) {
    printf("%ld ", (long)el);
}
```

```
void queue_each(queue* qu, void(*action)(void*)) {
    struct queue_block_t* block = qu->first;
    unsigned i = qu->space;
    unsigned index = 0;
    while (index < qu->size) {
        // Call the function
        action(block->data[i]);
        ++index;
        ++i;
        if (i >= BLOCK_SIZE) {
            block = block->next;
            i -= BLOCK_SIZE;
        }
    }
}
```

Queue.c

opcionais

```
static void** element_pointer(queue* qu, unsigned i) {
    i += qu->space;
    struct queue_block_t* block = advance_block(qu->first,i);
    return &block->data[i % BLOCK_SIZE];
}

void* queue_nth(queue* qu, unsigned i) {
    return *element_pointer(qu, i);
}

unsigned queue_find(queue* qu, int test(void*)){
    if (qu->size == 0)
        return -1;
        struct queue_block_t* block = qu->first;
    unsigned i = qu->space;
        unsigned index = 0;
```

```
while (!test(block->data[i])) {
                ++i;
                 ++index;
               if (index == qu->size)
            return -1;
            if (i >= BLOCK_SIZE) {
                    block = block->next;
                    i -= BLOCK_SIZE;
        }
    }
    return index;
}
    swap(element_pointer(qu, start+size-1),
&indexblock->data[index]);
    return index2;
}
```

Queue.c

opcionais

```
static int partition(queue* qu, int start, int size, int(*comp)
(void*, void*)) {
    swap(element_pointer(qu, start+size/2), element_pointer
(qu, start+size-1));

    void* pivot = queue_nth(qu, start+size-1);
    struct queue_block_t* iblock = advance_block(qu->first,
start);
    int i = start % BLOCK_SIZE;
    struct queue_block_t* indexblock = iblock;
    int index = i;
    int index2 = 0;
```

```
    for (int ii = 0; ii < size-1; ++ii) {
        if (comp(iblock->data[i], pivot)) {
                swap(&iblock->data[i], &indexblock-
>data[index]);
                ++index;
                ++index2;
        if (index >= BLOCK_SIZE) {
                indexblock = indexblock->next;
                index -= BLOCK_SIZE;
            }
        }
                ++i;
        if (i >= BLOCK_SIZE) {
            iblock = iblock->next;
            i -= BLOCK_SIZE;
        }
    }
```

Queue.c

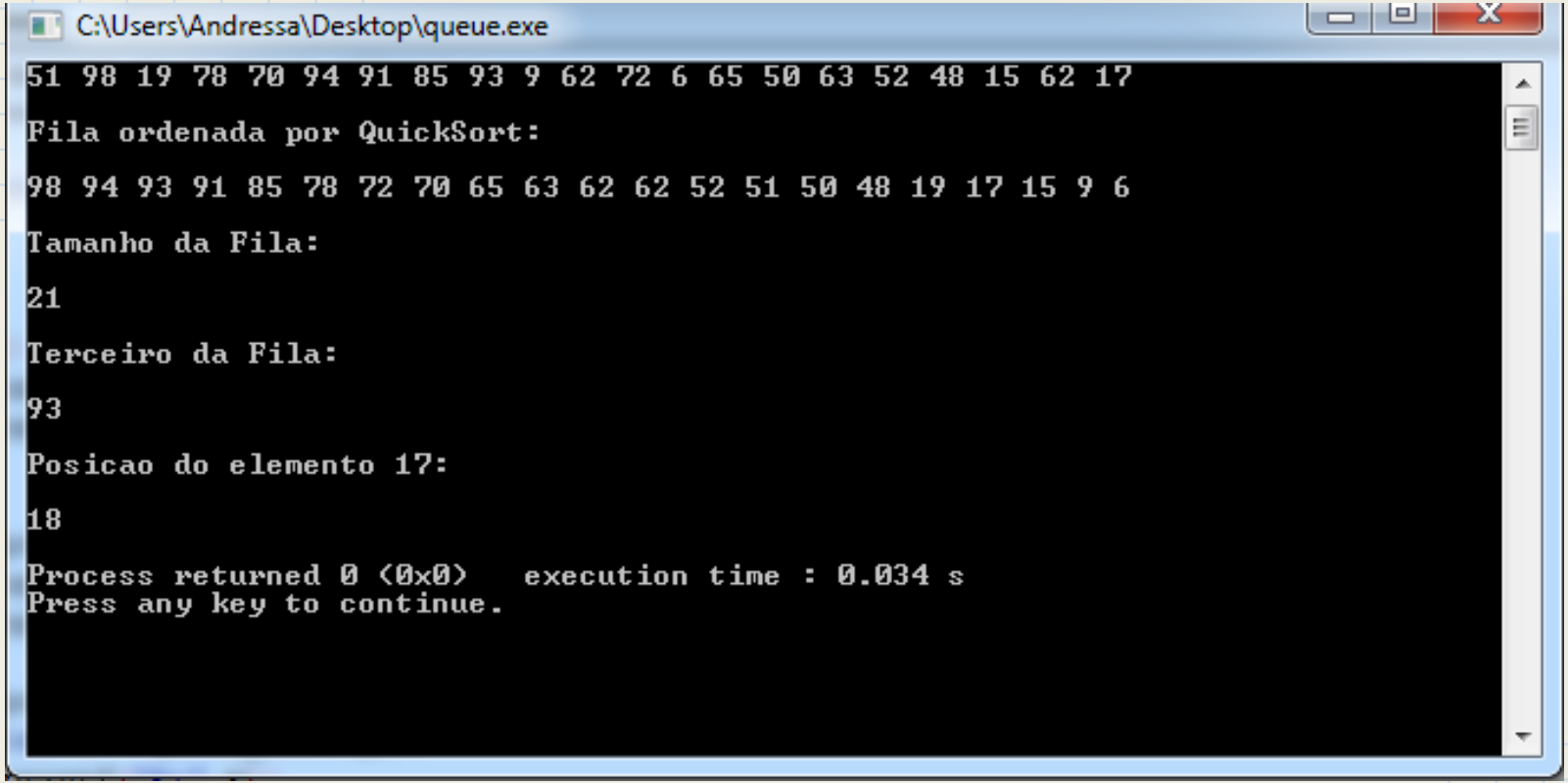
opcionais

```
static void quick_sort(queue* qu, int start, int size, int
(*comp)(void*, void*))
{
    if (size <= 1) return;
    int halfsz = partition(qu, start, size, comp);
    quick_sort(qu, start, halfsz, comp);
    quick_sort(qu, start+halfsz+1, size-halfsz-1, comp);
}


void queue_sort(queue* qu, int(*comp)(void*, void*)) {
    quick_sort(qu, qu->space, qu->size, comp);
}
```

```
static void swap(void** a, void** b) {
    void* tmp = *a;
    *a = *b;
    *b = tmp;
}

int sort_op(void* a, void* b) {
    return (long)a > (long)b;
}
```

```
int main(void)
{
    queue* qu = queue_alloc();
    srand(time(NULL));

    for (int i = 0; i < 20; ++i) {
        int c;
        c = rand() % 100;
        queue_push(qu, (void*)(long)c);
    }
    queue_push(qu, (void*)(long)17);
    queue_each(qu, print_element);

    printf("\n\nFila ordenada por QuickSort:\n\n");
    queue_sort(qu, sort_op);
    queue_each(qu, print_element);

    printf("\n\nTamanho da Fila:\n\n");
    int x = queue_size(qu);
    printf("%i",x);

    printf("\n\nTerceiro da Fila:\n\n");
    int y = queue_nth(qu, 2);
    printf("%i",y);

    printf("\n\nPosicao do elemento 17:\n\n");
    int test(void* a) {
            return (long)a == 17;
    }
    int z = queue_find(qu, test);
    printf("%i\n",z+1);

    for (int i = 0; i < 21; ++i) {
            queue_shift(qu);
    }

    queue_free(qu);
}
```

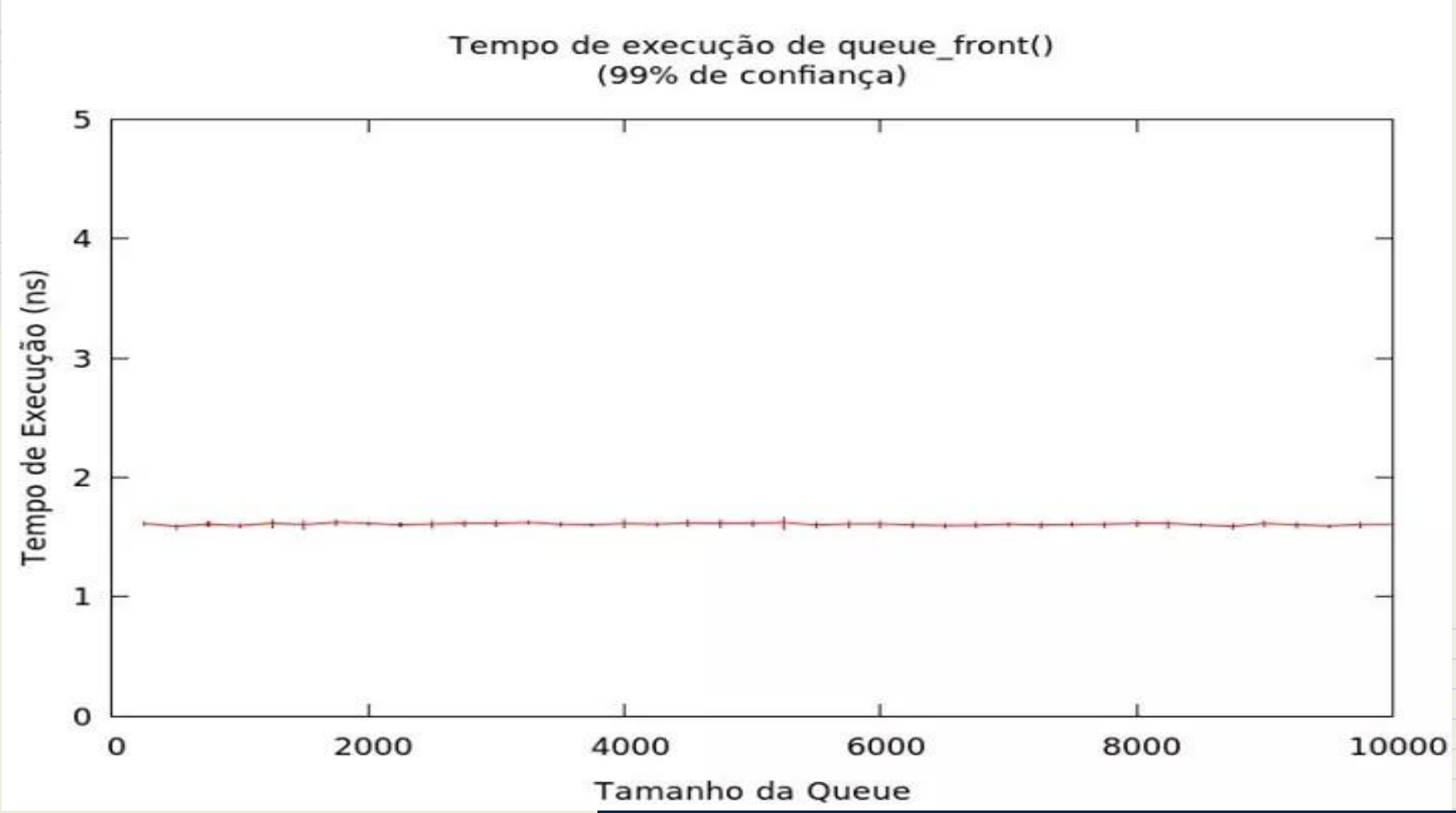
Main.c


C:\Users\Andressa\Desktop\queue.exe
51 98 19 78 70 94 91 85 93 9 62 72 6 65 50 63 52 48 15 62 17
Fila ordenada por QuickSort:
98 94 93 91 85 78 72 70 65 63 62 62 52 51 50 48 19 17 15 9 6
Tamanho da Fila:
21
Terceiro da Fila:
93
Posicao do elemento 17:
18
Process returned 0 (0x0)   execution time : 0.034 s
Press any key to continue.

Main.c

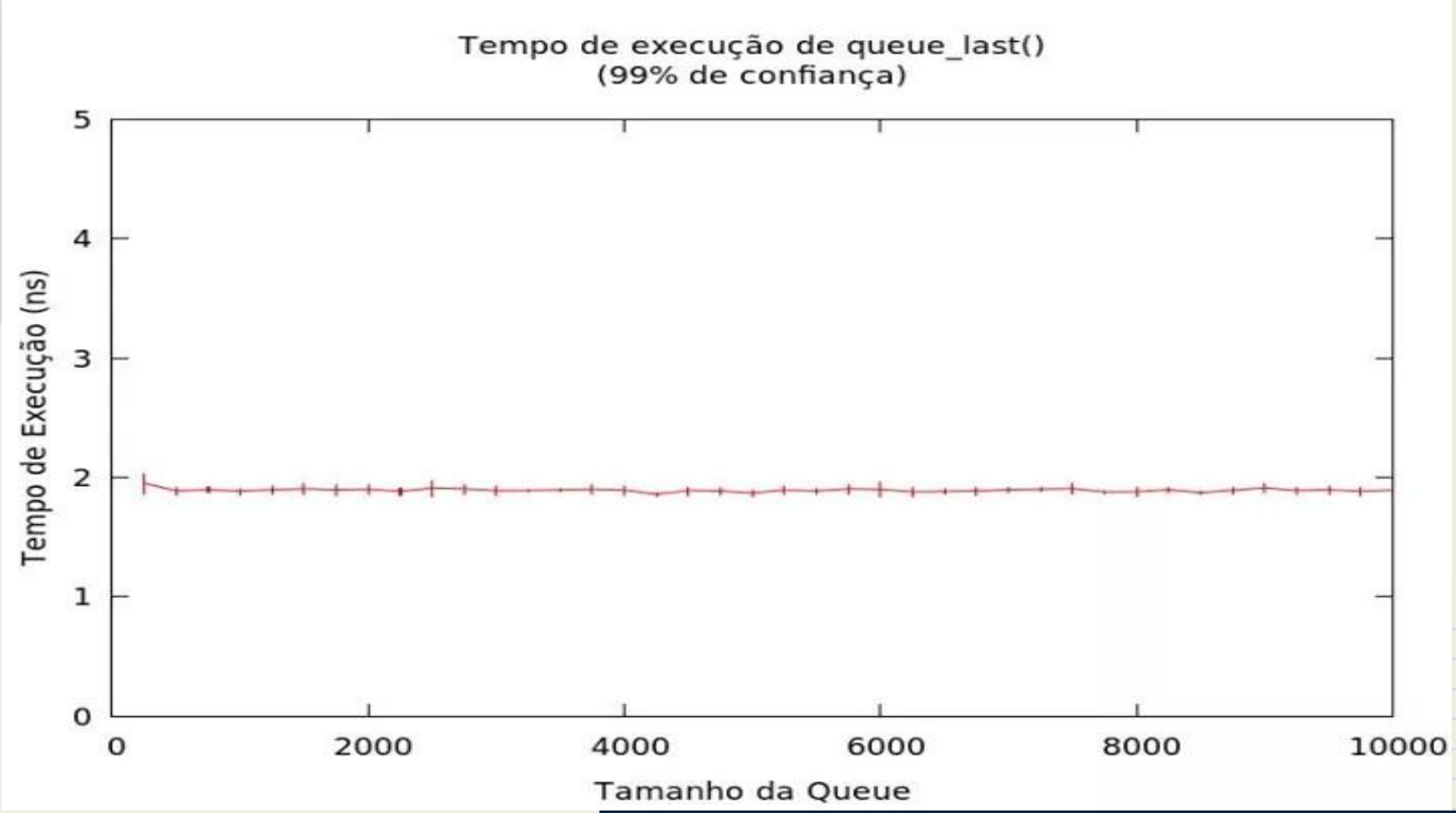
Gráficos da Complexidade

Operações opcionais



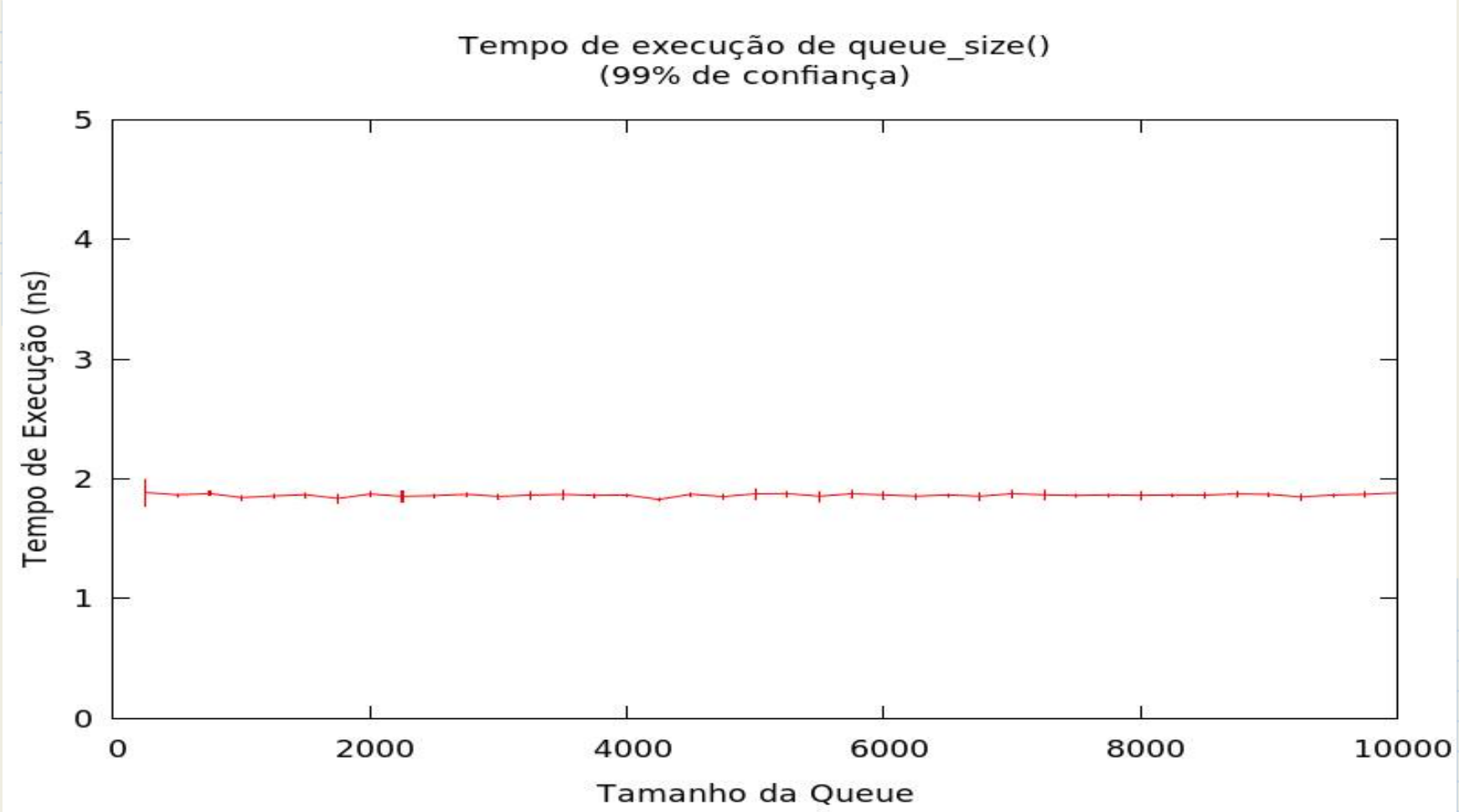
*Block size = 120

Queue_front



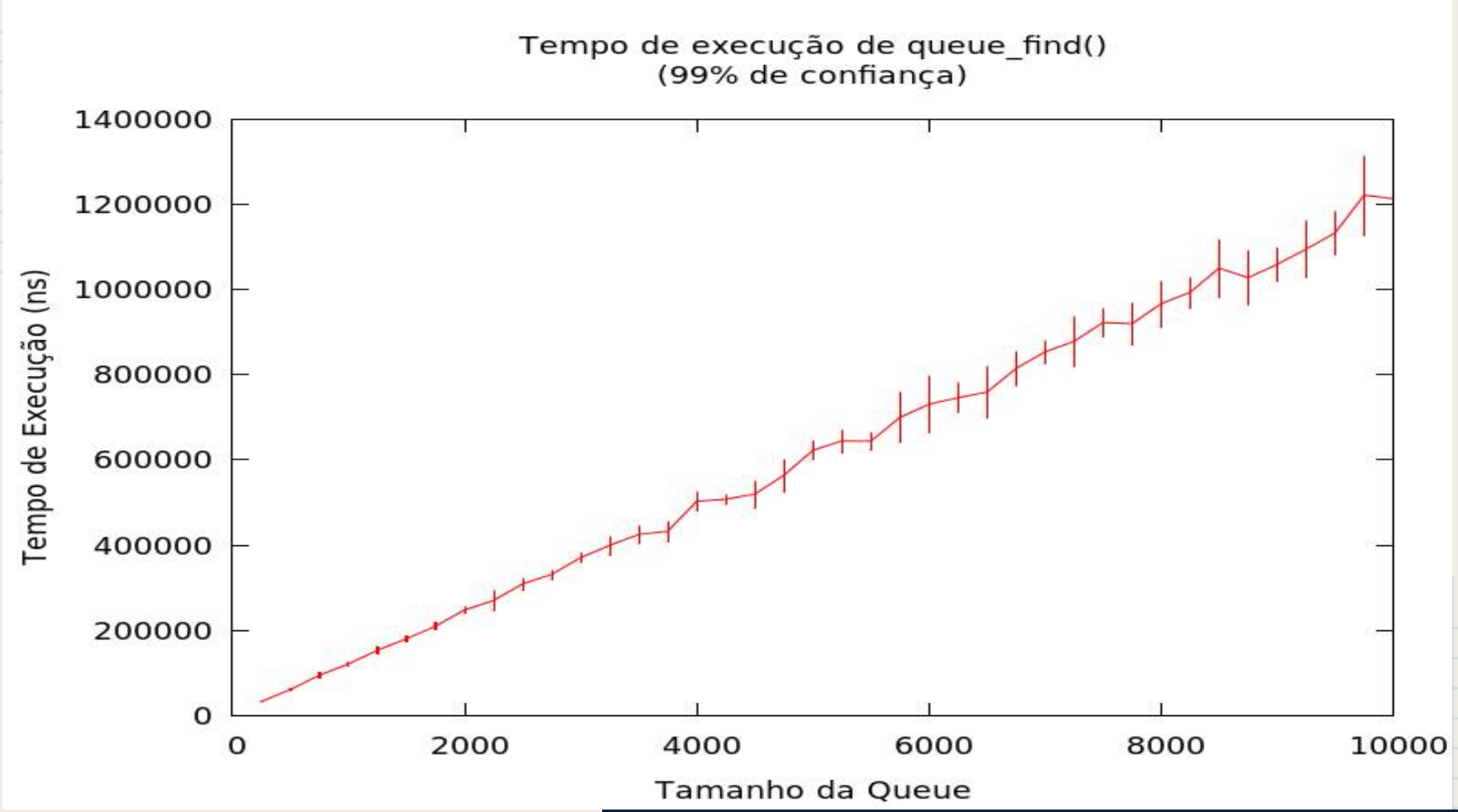
*Block size = 120

Queue_last



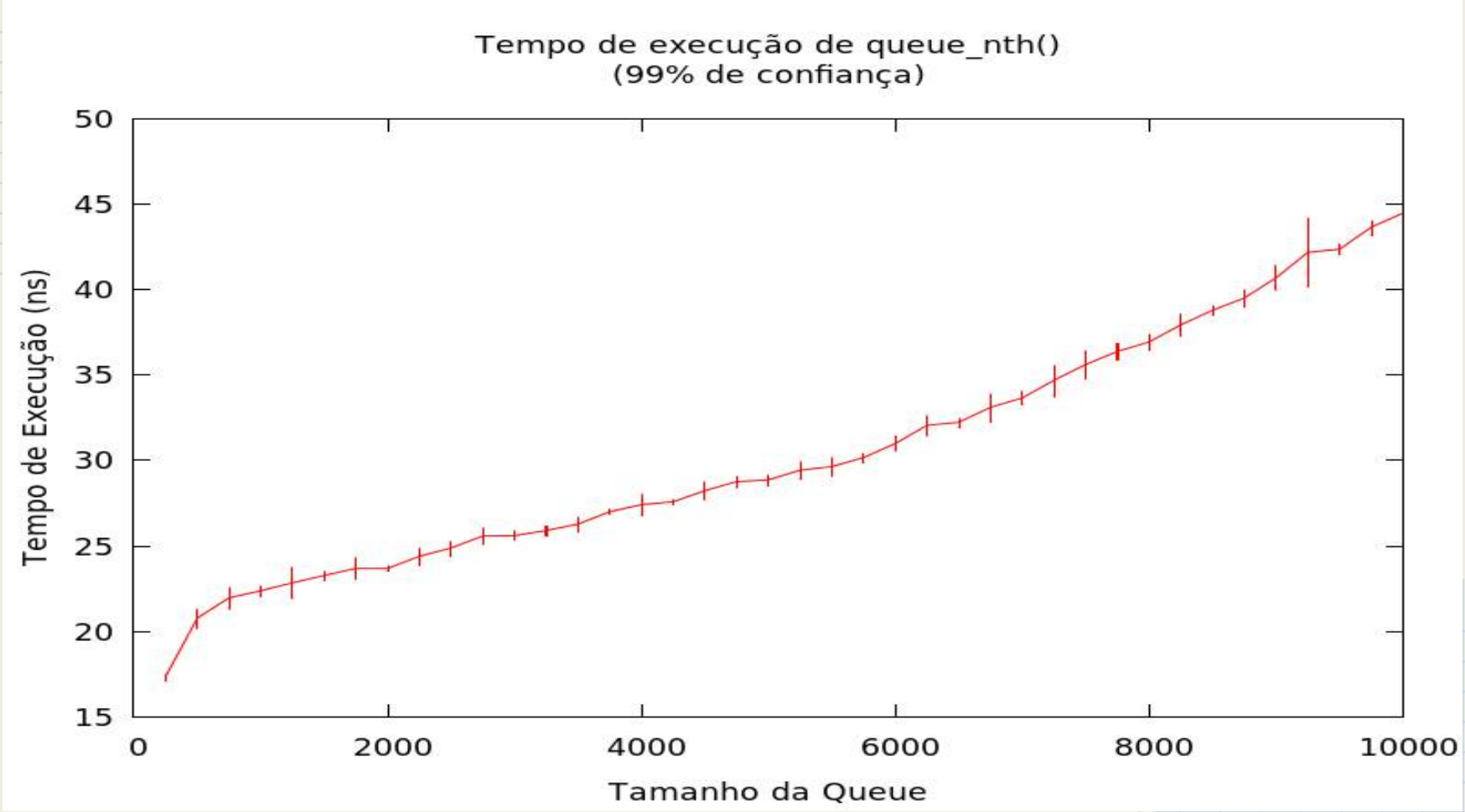
*Block size = 120

Queue_size



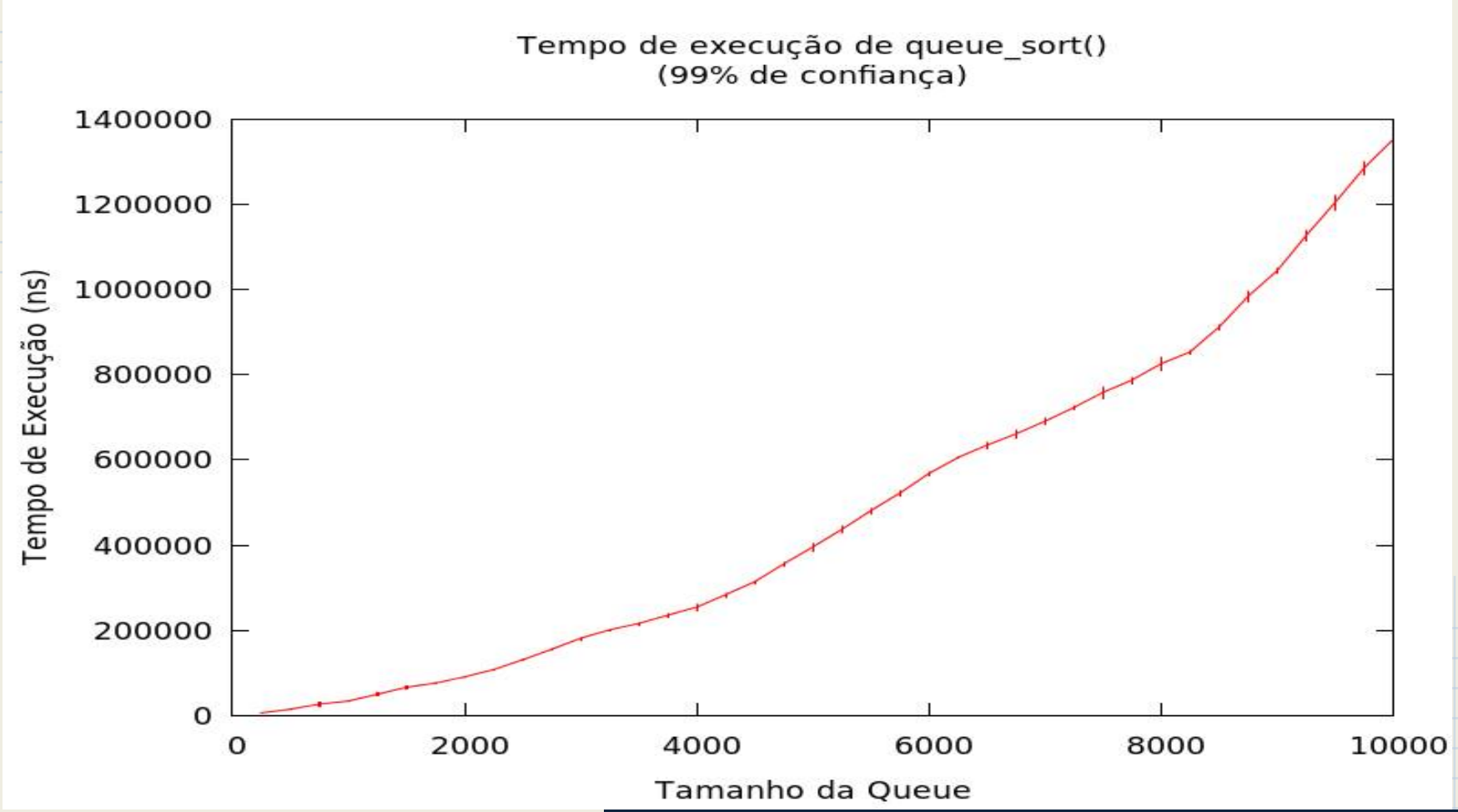
*Block size = 120

Queue_find



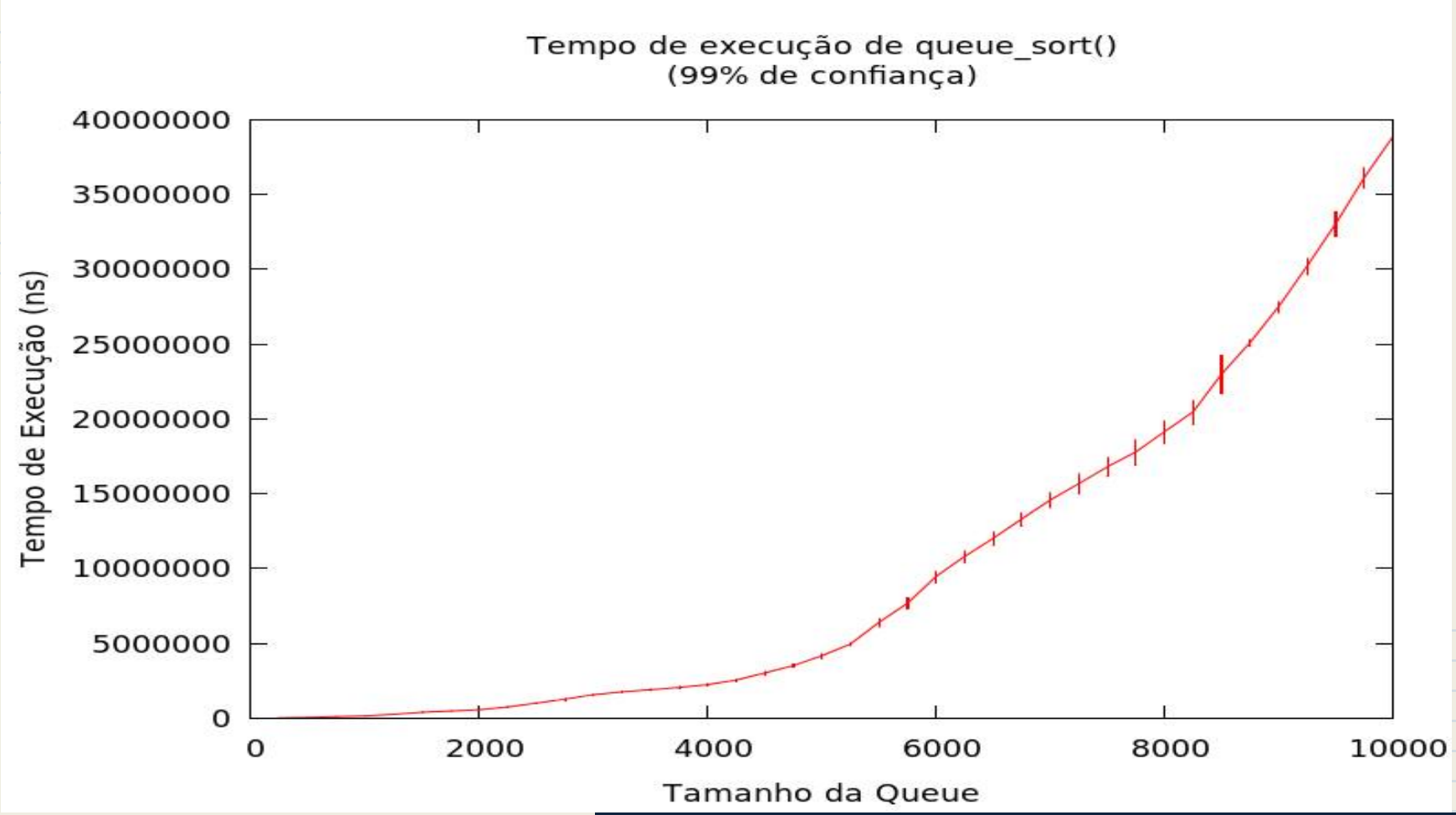
*Block size = 120

Queue_nth



*Block size = 120

Queue_sort



*Block size = 10

Queue_sort