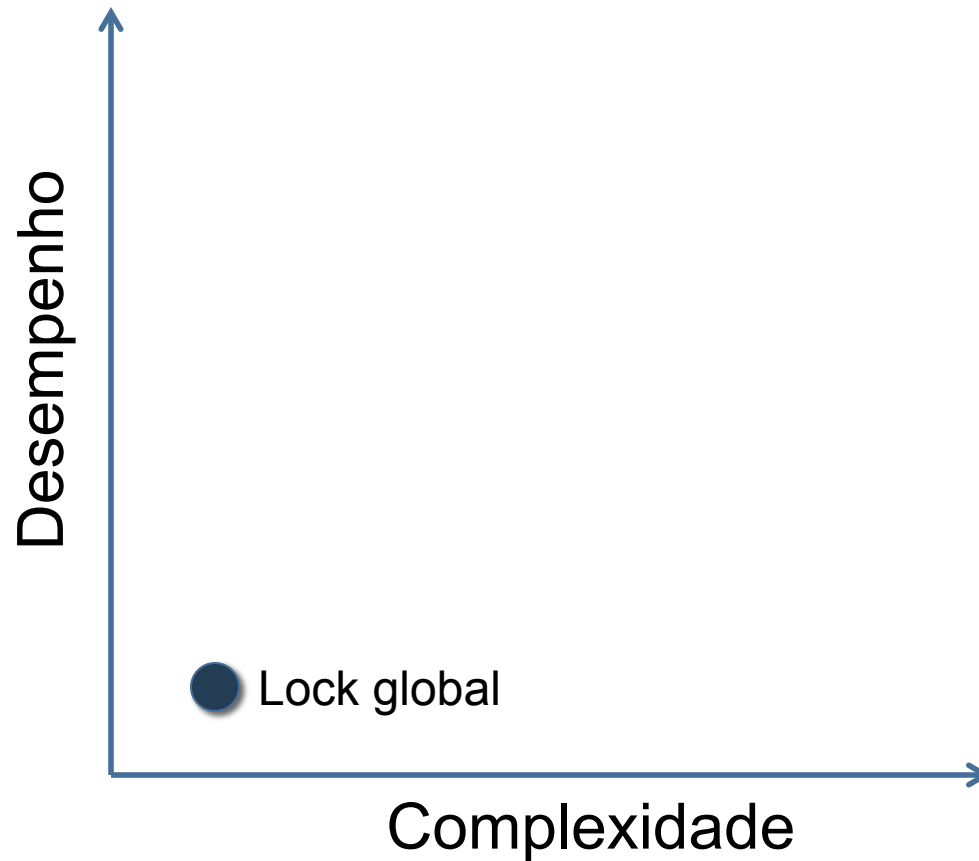


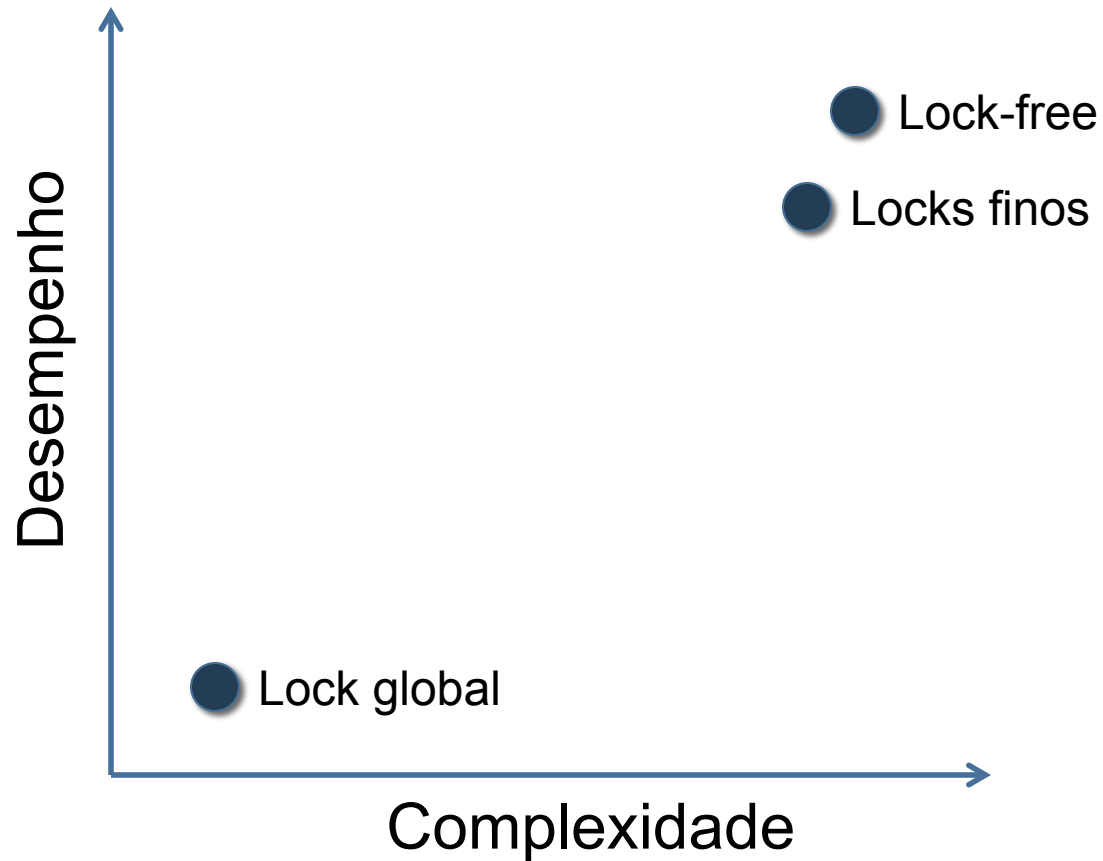
Memória Transacional

Alexandro Baldassin
Programação Concorrente – 2sem, 2016

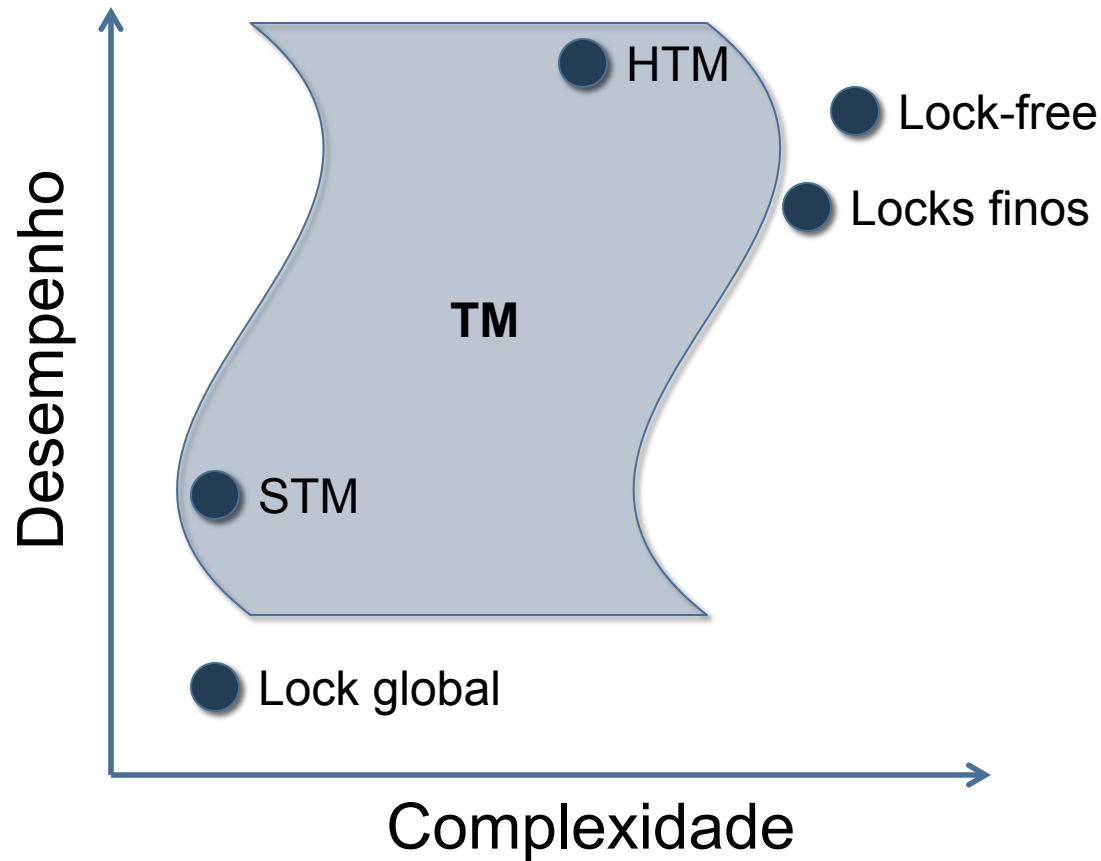
Programação concorrente



Programação concorrente



Programação concorrente

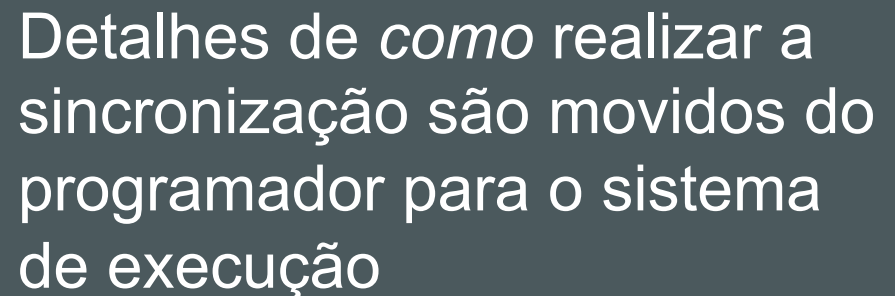


Memória Transacional (TM)

- No modelo transacional, programadores usam o conceito de **transação** como abstração
 - **A**tomicidade
 - **C**onsistência
 - **I**solamento
- Vantagens
 - Nível de abstração maior
 - Potencial ganho de desempenho
 - Dependente de implementação (visto mais adiante)
 - Composição de código

Memória Transacional (TM)

- No modelo transacional, programadores usam o conceito de **transação** como abstração
 - **A**tomicidade
 - **C**onsistência
 - **I**solamento
- Vantagens
 - Nível de abstração maior
 - Potencial ganho de desempenho
 - Dependente de implementação (visto mais adiante)
 - Composição de código



Detalhes de *como* realizar a sincronização são movidos do programador para o sistema de execução

Blocos atômicos

- O termo **bloco atômico** geralmente é usado quando o enfoque é sobre o suporte em linguagens
- Memória transacional (TM) é uma forma de se implementar blocos atômicos
- Nomenclatura ainda não consolidada!

Programando com blocos atômicos

- Programador delimita a região que deve ser executada atomicamente
 - Exemplo com lista ligada
- Sistema de execução (pode ser hardware ou software) cuida de garantir atomicidade, isolamento e consistência



```
public boolean add(int item) {
    Node pred, curr;
    boolean valid = false;

    atomic {
        pred = head;
        curr = pred.next;
        while (curr.key < item) {
            pred = curr;
            curr = curr.next;
        }
        if (item != curr.key) {
            Node node = new Node(item);
            node.next = curr;
            pred.next = node;
            valid = true;
        }
    }

    if (valid) return true;
    return false;
}
```


Suporte transacional no GCC 4.7

- Suporte experimental a TM existe no GCC a partir da versão 4.7 (abril de 2012)
 - As construções adicionadas à linguagem são baseadas no documento “*Draft Specification of Transactional Language Constructs for C++*”, versão 1.1
- <http://gcc.gnu.org/wiki/TransactionalMemory>

*“The support is experimental. In particular, this also means that several parts of the implementation are not yet optimized. **If you observe performance that is lower than expected, you should not assume that transactional memory is inherently slow; instead, please just file a bug.**”*

Construções GCC

- Principais construções
 - `__transaction_atomic {...}`
 - `__transaction_relaxed {...}`
 - `__transaction_cancel`
- Anotações
 - `__attribute__((transaction_safe))`
 - `__attribute__((transaction_callable))`
 - `__attribute__((transaction_pure))`
- Opção **-fgnu-tm** deve ser passada ao compilador

Exemplo com GCC – lista ligada

```
int list_add(list_node_t *head, int item)
{
    list_node_t *pred, *curr;

    __transaction_atomic {
        pred = head;
        curr = head->next;
        while (curr->key < item) {
            pred = curr;
            curr = curr->next;
        }

        list_node_t *node = (list_node_t *)malloc(sizeof(list_node_t));
        node->key = item;
        node->next = curr;
        pred->next = node;
    }
    return 1;
}
```

Demais operações implementadas da
mesma forma

Detalhes da geração e execução

- O compilador gera duas versões para cada rotina especificada com o atributo **transaction_safe**
 - A versão transacional é usada quando a rotina é chamada dentro de uma transação
- O código gerado é *linkado* com uma biblioteca de runtime chamada **libitm**
 - Essa biblioteca pode ser substituída em tempo de execução
 - Permite que diferentes implementações possam ser avaliadas de forma simples
 - Especificação segue basicamente a ABI proposta pela Intel
 - *Intel Transactional Memory Compiler and Runtime Application Binary Interface*, revisão 1.1 – maio de 2009

Teste de “stress” da lista

```
void *list_exercise(void *arg)
{
    int operations = (int)arg;

    int add_or_remove, chance, value, last_value = 0;

    add_or_remove = 1; /* 1 - add, 0 - remove */
    while (operations--)
    {
        chance = (int)(erand48(seed)*100);
        value = (int)(erand48(seed)*RANGE*2);

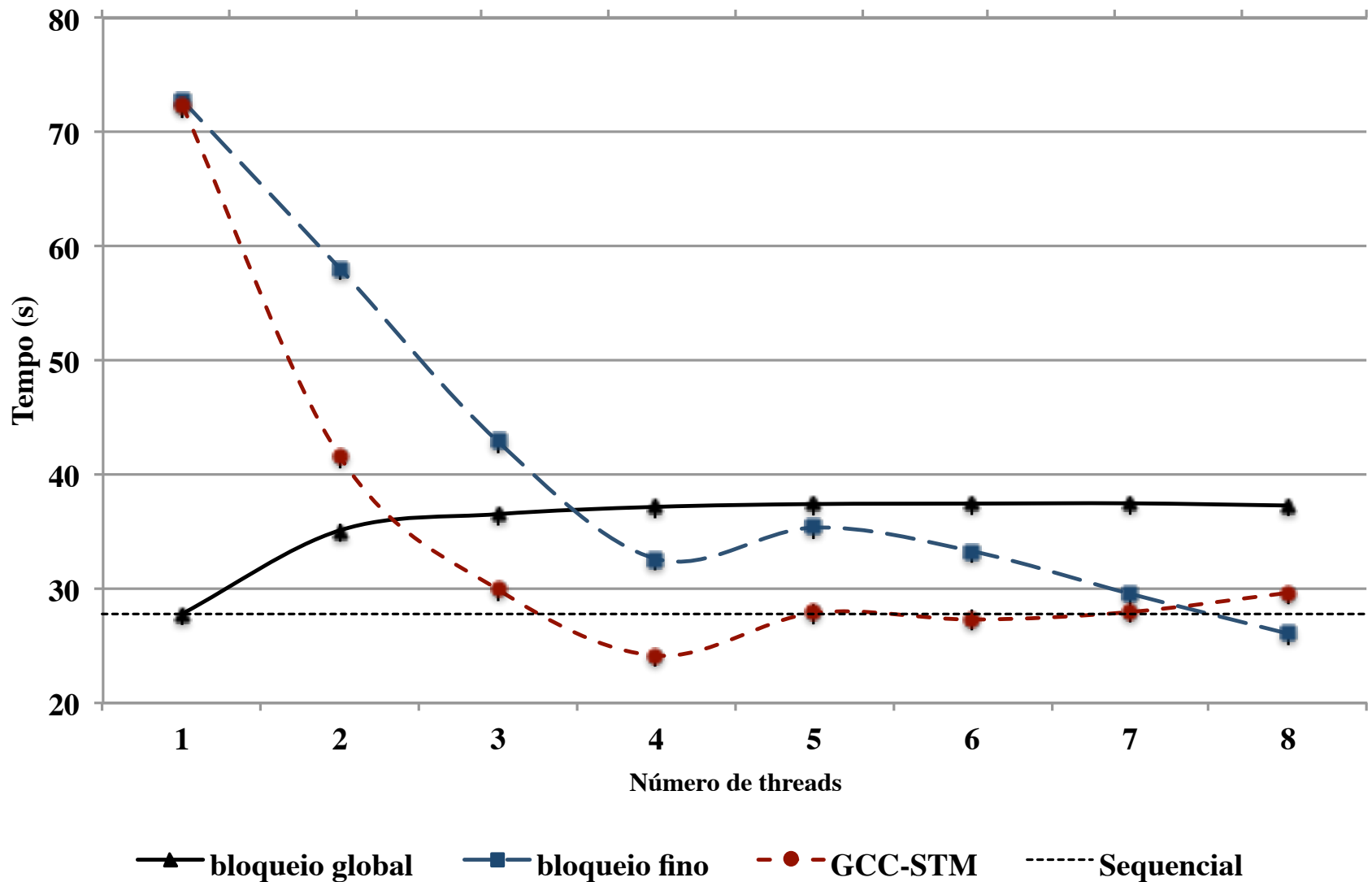
        if (chance <= UPD_RATE) {
            if (add_or_remove) {
                list_add(linked_list, value);
                last_value = value;
            }
            else list_remove(linked_list, last_value);
            add_or_remove ^= 1;
        }
        else list_contain(linked_list, value);
    }
}
```

Todas as threads executam a mesma rotina

Alguns resultados

- Máquina: Intel(R) Core(TM) i7-2600K [3.4GHz, 8GB RAM]
- Tamanho do conjunto: 10.000 elementos
- Número total de operações: 1.000.000

Taxa de atualização de 20%



Taxa de atualização de 50%

