Епифанцев Егор Витальевич, группа 9305

Тема: «Реализация разделяемых структур данных в модели MPI RMA»

Цель работы: Разработка распределенного связного списка, стека и очереди в модели удаленного доступа к памяти (MPI RMA).

Объект и предмет исследования: объект - разделяемые структуры данных. Предмет исследования - связный список с использованием блокировок, неблокирующая очередь Майкла и Скотта и неблокирующий стек Трайбера.

Техническое задание: Структуры должны быть реализованы на языке программирования Си с использованием библиотеки OpenMPI и корректно выполняться на кластере.

Актуальность темы

В настоящее время вычислительные кластеры применяются во многих областях науки, например, в физике, химии, астрономии, биологии, фармакологии для решения сложных вычислительных задач и моделирования. Такие системы используют модель распределенной памяти, в отличие от обычных ПК. Распределенная же память приводит к созданию распределенных структур данных.

Блокирующая синхронизация

Примитивы синхронизации:

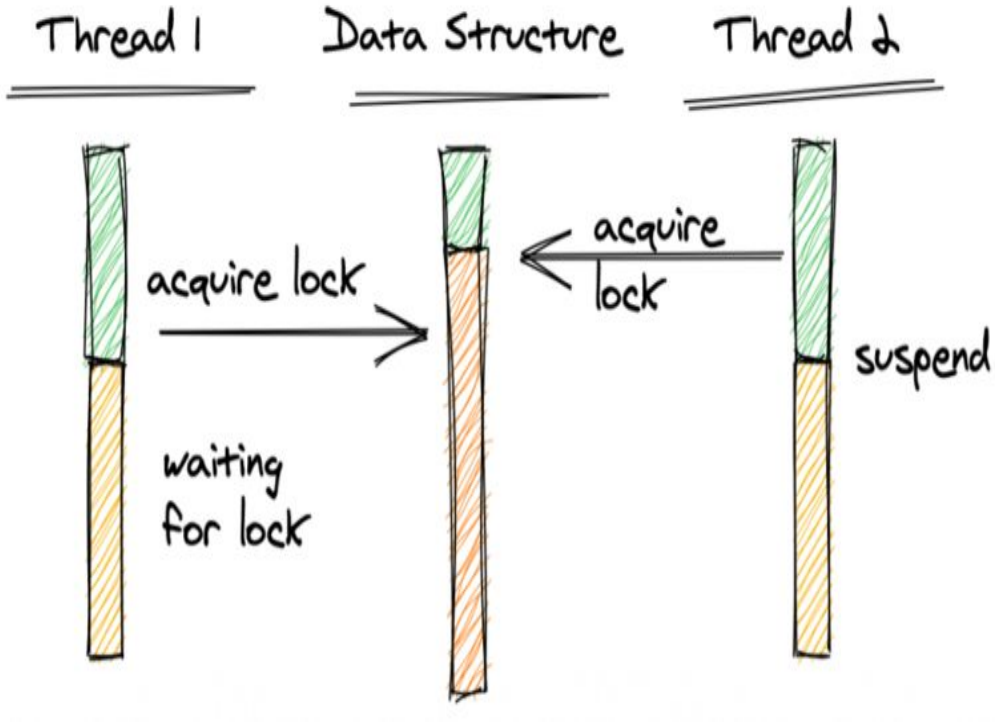
- Мьютексы;
- Семафоры;
- Спинлоки.

Достоинства:

- Простота реализации алг-мов

Недостатки:

- Низкая масштабируемость;
- Низкая отказоустойчивость.



Неблокирующая синхронизация

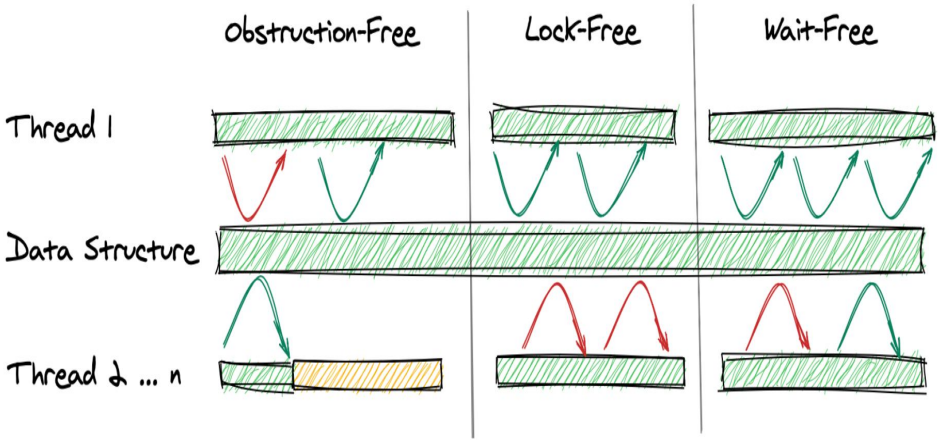
Атомарные операции (особенно CAS)

Достоинства:

- Высокая отказоустойчивость;
- Высокая масштабируемость;

Недостатки:

- Сложность построения алгоритмов;
- Большое количество атомарных операций

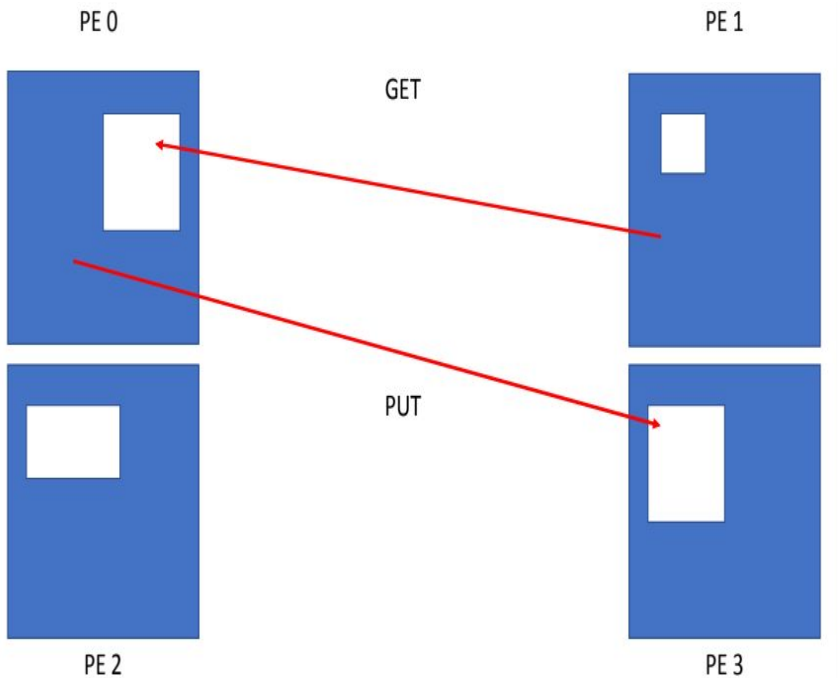



```
function cas(p: pointer to int, old: int, new: int) is
    if *p ≠ old
        return false

    *p ← new

    return true
```

Модель MPI RMA

MPI RMA - модель программирования, реализованная в библиотеки MPI, которая предоставляет возможность прямого доступа к памяти удаленных процессов без необходимости явного обмена сообщениями.

- Окна - сегменты памяти для межпроцессного взаимодействия.
- Эпохи - части программы, внутри которых происходит синхронизация
- RMA-операции - используются внутри эпох для межпроцессного обмена данными



Пассивная синхронизация

MPI RMA предоставляет два варианта синхронизации - активная и пассивная. При активной синхронизации оба процесса оказываются вовлечены в процесс синхронизации. При пассивной же только один (origin process). В работе используется пассивный метод синхронизации, так как он имеет меньше накладных расходов.

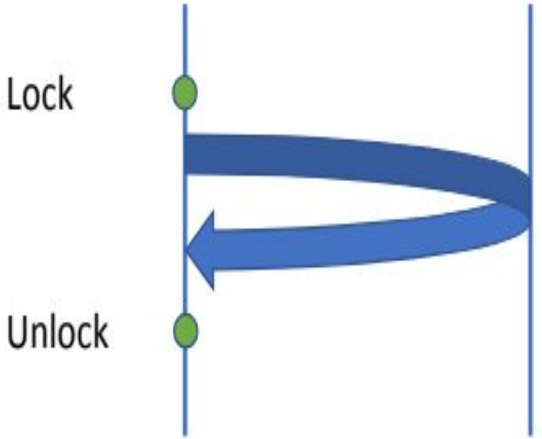
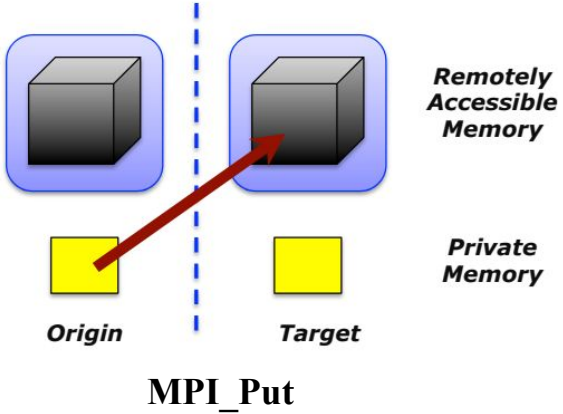


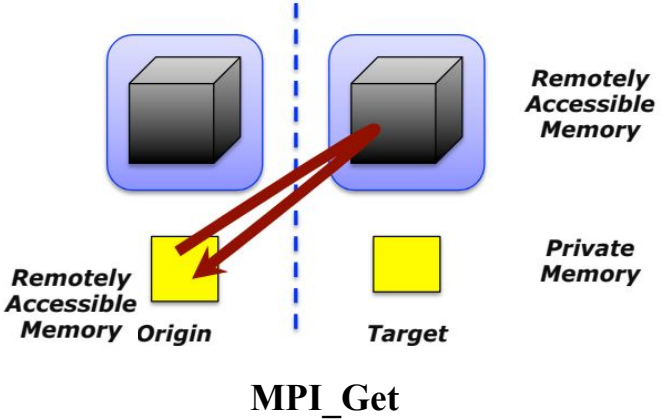
Схема пассивной синхронизации

RMA-операции

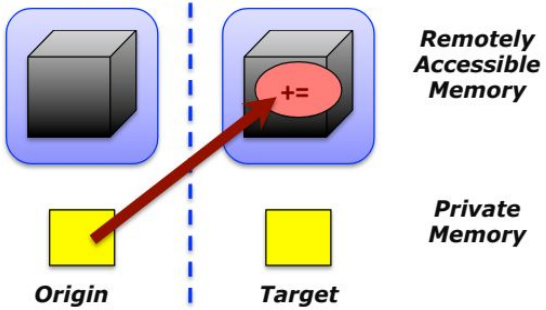
- MPI_Put (неблокирующая)
- MPI_Get (неблокирующая)
- MPI_Compare_and_swap (атомарная)
- MPI_Fetch_and_op (атомарная)
- MPI_Accumulate (атомарная)
- MPI_Get_accumulate (атомарная)



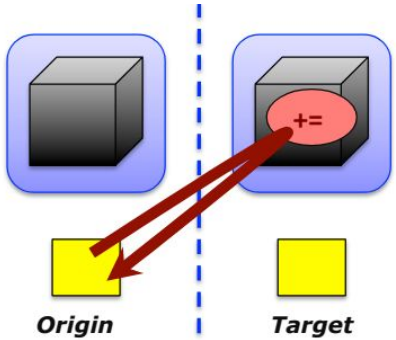
MPI_Put



MPI_Get



MPI_Accumulate



MPI_Get_Accumulate

Неблокирующая очередь Майкла и Скотта

```
1: function ENQUE(val, rank, q, win)
2:     tmpTail ← nullPtr
3:     tailNext ← nullPtr
4:     newNode ← allocElem(val, rank, win)
5:     while True do
6:         tmpTail ← getTail(q, win)
7:         CAS(tmpTail, nullPtr, newNode, result)
8:         Flush(tmpTail.rank, win)
9:         if result == nullPtr then
10:            CAS(q.tail, tmpTail, newNode, result)
11:            return
12:        else
13:            tailNext ← getTail(q, win)
14:            CAS(q.tail, tmpTail, tailNext, result)
15:            Flush(0, win)
16:        end if
17:    end while
18: end function
```

```
1: function DEQUEUE(q, win)
2:     while True do
3:         head ← getHead(q, win)
4:         tail ← getTail(q, win)
5:         afterHead ← getNextHead(head, win)
6:         if tail == head then
7:             if afterHead == nullPtr then
8:                 return
9:             else
10:                CAS(q.tail, tail, afterHead, result)
11:                Flush(0, win)
12:            end if
13:        else
14:            CAS(q.head, head, afterHead, result)
15:            Flush(0, win)
16:            if result == head then
17:                return
18:            end if
19:        end if
20:    end while
21: end function
```

Неблокирующий стек Трайбера

```
1: function PUSH(val, rank, s, win)
2:     curHead ← nullPtr
3:     newHead ← nullPtr
4:     result ← nullPtr
5:     newHead ← allocElem(val, win)
6:     while True do
7:         curHead ← getHead(s, win)
8:         changeNext(curHead, newHead, win)
9:         CAS(s.head, curHead, newHead, result)
10:        Flush(s.head.rank, win)
11:        if result == curHead then
12:            return
13:        end if
14:    end while
15: end function
```

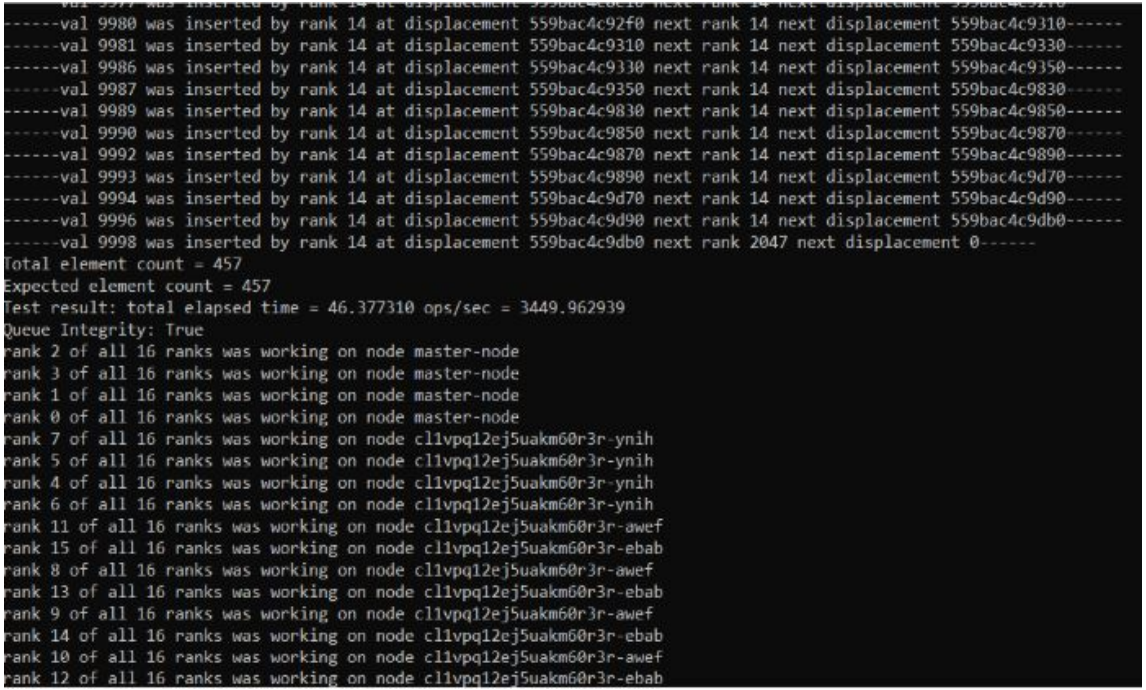
```
1: function POP(s, win)
2:     curHead ← nullPtr
3:     nextHead ← nullPtr
4:     result ← nullPtr
5:     while True do
6:         curHead ← getHead(s, win)
7:         if curHead == nullPtr then
8:             return
9:         end if
10:        nextHead ← getNextHead(curHead, win)
11:        CAS(s.head, curHead, nextHead, result)
12:        Flush(s.head.rank, win)
13:        if result == curHead then
14:            return
15:        end if
16:    end while
17: end function
```

Тестирование на кластере

Экспериментальное исследование проводилось на вычислительном кластере с 4 вычислительными узлами. При этом на каждом из узлов находилось по 1 4-ядерному процессору линейки Intel Xeon с базовой частотой 2 ГГц и максимальной частотой 3.2 ГГц. В качестве MPI-библиотеки использовалась OpenMPI 4.1.2.

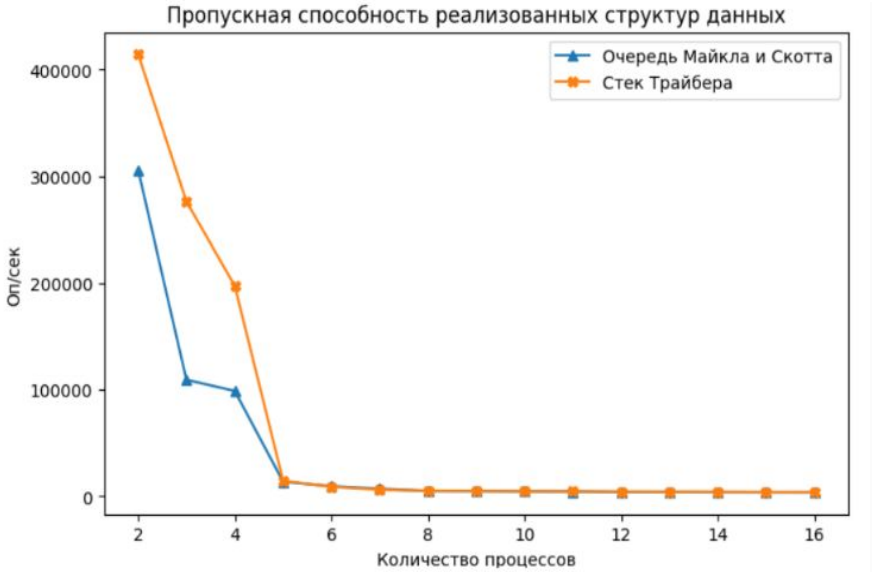
```
processor       : 7
vendor_id       : GenuineIntel
cpu family      : 6
model           : 106
model name      : Intel Xeon Processor (Icelake)
stepping        : 0
microcode       : 0x1
cpu MHz         : 1995.312
cache size      : 16384 KB
physical id     : 0
siblings        : 8
core id         : 3
cpu cores       : 4
apicid          : 7
initial apicid  : 7
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
```

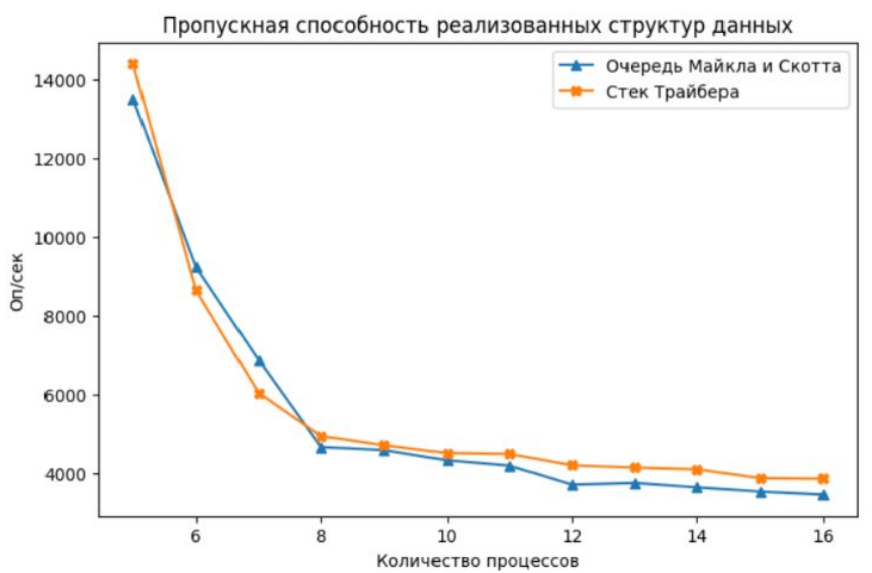
Тестирование на кластере



Пример теста очереди на кластере

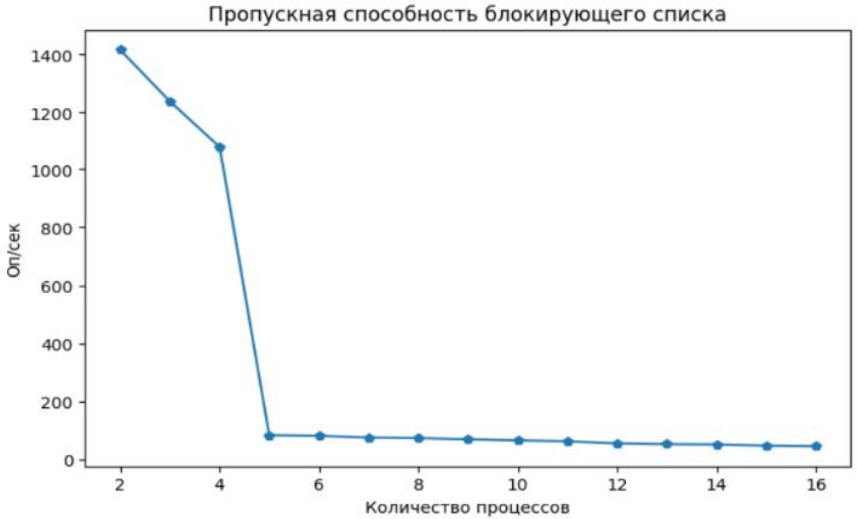
Тестирование на кластере

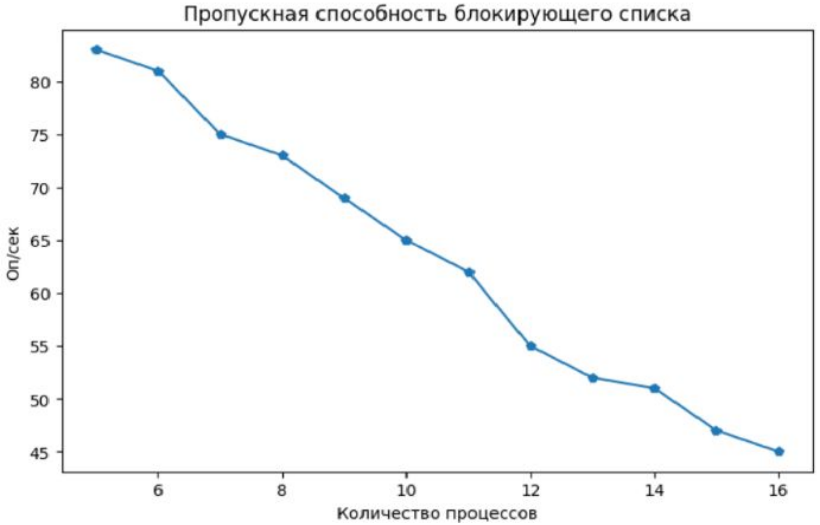




Каждый процесс производил по 10000 операций вставки/удаления (тип операции выбирался равновероятно)

Тестирование на кластере





Изначально корневой процесс генерировал список длиной 512 элементов, затем каждый из процессов производил по 512 операций вставки/удаления (тип операции выбирался равновероятно)

Заключение

Результаты: разработаны 3 структуры данных в модели MPI RMA. Для каждой из структур данных были проведены тесты на вычислительном кластере, которые показали, что очередь и стек достаточно хорошо масштабируются как минимум до 16 процессов. Реализованные структуры могут найти применение на вычислительных системах с распределенной памятью, например, для реализации более сложных структур данных (очередь с приоритетом, развернутый связный список и т.д.) или для хранения и упорядочивания данных, полученных в ходе вычислений.

Дальнейшие перспективы развития темы:

- В неблокирующих структурах отойти от идеи централизации головы и хвоста структур в памяти корневого процесса;
- Реализовать неблокирующий связный список, например, список Харриса.