МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»

Институт компьютерных наук и кибербезопасности
Высшая школа технологий искусственного интеллекта
Направление 02.03.01 Математика и компьютерные науки

Отчет по лабораторной работе

по дисциплине «Архитектура суперкомпьютеров»

Разработка приложения, моделирующего работу механизма передачи сообщения в коммуникационной сети суперкомпьютера.

Обучающийся: _____ Гладков И.А.

Руководитель: _____ Чуватов М.В.

«_____»_____20___г.

Санкт-Петербург, 2024

Содержание

Введение

Отчет представляет собой описание выполненной лабораторной работы по дисциплине «Архитектура суперкомпьютеров». Лабораторная работа включает в себя реализацию приложения, моделирующего работу механизма передачи сообщения в коммуникационной сети суперкомпьютера. В приложении реализована функция отправки сообщения, которая вызывается в формате (АДРЕС_ИСТОЧНИКА ДЛИНА_СООБЩЕНИЯ АДРЕС_ПОЛУЧАТЕЛЯ) и которая включает в себя функцию вычисления маршрута передачи сообщения с целью минимизации времени, необходимого для передачи. ДЛИНА_СООБЩЕНИЯ – это число шагов, требуемых для его передачи по свободному каналу. Также реализована возможность продвигаться по шагам, и в начале каждого шага пользователь может указать от 0 до 10 заданий на пересылку сообщений с произвольными адресами источника и получателя. Важной особенностью является штраф за каждую пересылку через промежуточный узел или через коммутатор в 1 дополнительный шаг времени. В качестве использованной топологии была выбрана топология Dragonfly со следующими характеристиками:

- Число групп: 11
- Коммутаторов в группе: 5
- Узлов на коммутатор: 3
- Пропускная способность внутри группы: 4
- Пропускная способность между группами: 2

Работа была выполнена в среде Visual Studio 2022 на языке программирования C++.

1 Математическое описание

1.1 Коммуникационные сети суперкомпьютеров

Коммуникационные сети суперкомпьютеров — это критически важная часть архитектуры, которая обеспечивает связь между отдельными узлами (процессорами, серверами, вычислительными блоками) для выполнения параллельных вычислений. Эти сети должны быть чрезвычайно быстрыми и эффективными, чтобы минимизировать задержки и обеспечить максимальную пропускную способность для обмена данными.

Основные компоненты коммуникационной сети:

1. **Узлы** — это вычислительные элементы (процессоры, ядра или серверы), которые обмениваются данными. Каждый узел имеет сетевые адаптеры, через которые они подключены к сети.

2. **Коммутаторы** — это устройства, которые управляют потоком данных между узлами, перенаправляя данные от одного узла к другому по оптимальному пути.

3. **Маршрутизация** — процесс определения путей передачи данных между узлами через коммутаторы. Эффективные алгоритмы маршрутизации играют ключевую роль в минимизации задержек.

4. **Сетевые интерфейсы** — это интерфейсы, через которые узлы подключаются к сети (например, InfiniBand, Ethernet, или специализированные сети).

Ключевые характеристики коммуникационных сетей:

- **Пропускная способность** (Bandwidth) — максимальное количество данных, которое может передаваться по сети за единицу времени.

- **Задержка** (Latency) — время, которое требуется для передачи данных от одного узла к другому.

- **Прямое соединение** (Direct Connection) — возможность узлов обмениваться данными напрямую без участия коммутаторов.

- **Трафик** (Traffic Pattern) — типы передач данных, например, all-to-all (все ко всем), one-to-all (один ко всем), one-to-one (один к одному).

- **Отказоустойчивость** (Fault Tolerance) — сети суперкомпьютеров проектируются таким образом, чтобы справляться с отказами отдельных узлов или соединений без серьезного влияния на общую производительность.

1.2 Сетевые топологии

Сетевые топологии — это схема организации связи между узлами в вычислительной системе, такие как суперкомпьютеры, серверы или рабочие станции. Топология сети определяет, как данные передаются между узлами, насколько эффективно сеть масштабируется, и насколько быстро она обрабатывает запросы. Выбор топологии влияет на производительность, устойчивость к отказам и стоимость создания и поддержания сети.

Основные типы топологий сетей, используемых в сети суперкомпьютеров:

1. **Полносвязная топология** – каждый узел сети напрямую соединён со всеми другими узлами. Это минимизирует задержки при передаче данных, так как каждый узел может обмениваться данными без участия промежуточных узлов.

2. **Fat-tree** (Толстое дерево) – древовидная топология с высокой пропускной способностью на каждом уровне, благодаря расширенным (“жирным”) каналам между уровнями. У каждого уровня дерева есть более широкие каналы для соединения с более высоким уровнем. В этой топологии узлы на нижнем уровне связаны с узлами верхних уровней через несколько уровней коммутаторов.

3. **Тор** представляет собой двумерную или трёхмерную сетку узлов, в которой каждый узел соединён с ближайшими соседями, а крайние узлы соединяются друг с другом, создавая “кольцевую” структуру. В трёхмерной версии каждый узел связан с соседями по трём направлениям (например, x, y, z).

4. **Dragonfly** — это иерархическая сеть, где узлы объединены в группы, каждая из которых представляет собой почти полносвязную структуру. Между группами создаются ограниченные глобальные соединения, что снижает количество коммутаторов и кабелей, требуемых для межгрупповой связи.

5. **Гиперкуб** — это топология, в которой каждый узел представляет вершину многомерного куба. В двумерной версии (2D) каждый узел соединён с двумя соседями, а в трёхмерной версии (3D) узлы соединены с тремя соседями. С увеличением измерений количество узлов и соединений растёт экспоненциально.

6. **Butterfly** – данные передаются через несколько уровней узлов, где каждый уровень имеет узлы, соединённые с узлами следующего уровня по заранее определённой схеме, напоминающей "крылья бабочки". Это позволяет эффективно распределять данные между узлами через минимальное количество уровней.

1.3 Топология Dragonfly

Топология Dragonfly — это современная топология, разработанная для высокопроизводительных вычислительных систем, таких как суперкомпьютеры, с целью обеспечения

высокой производительности при минимальных затратах на межузловую коммуникацию. Dragonfly сочетает идеи из «Fat-tree», «Torus» и других топологий для создания эффективной, высокомасштабируемой сети.

Основные черты Dragonfly:

1. Иерархическая структура:
   - Группы: Сеть разбита на группы, внутри которых узлы соединены полносвязно или почти полносвязно. Группы могут состоять из подгрупп узлов, связанных между собой с высокой пропускной способностью.
   - Межгрупповые соединения: Узлы из разных групп соединены меньшим числом связей через специальные маршрутизаторы или коммутаторы, обеспечивая глобальные связи между группами.
2. Минимизация числа соединений: Каждая группа связана только с небольшим количеством других групп, что снижает общие требования к числу межгрупповых соединений и позволяет строить сети с тысячами или даже миллионами узлов.
3. Эффективное распределение нагрузки: Топология Dragonfly использует адаптивные алгоритмы маршрутизации, чтобы избежать перегрузок в межгрупповых каналах и сбалансировать нагрузку между узлами и группами.

Преимущества топологии Dragonfly:

- Отличная масштабируемость: Топология поддерживает большие кластеры с тысячами узлов, при этом минимизируя количество межгрупповых соединений.
- Высокая пропускная способность: Полносвязные группы и прямые соединения между группами обеспечивают высокую скорость передачи данных.
- Снижение задержек: Адаптивная маршрутизация позволяет минимизировать задержки и сбалансировать нагрузку.

Структура топологии Dragonfly представлена на рисунке 1.

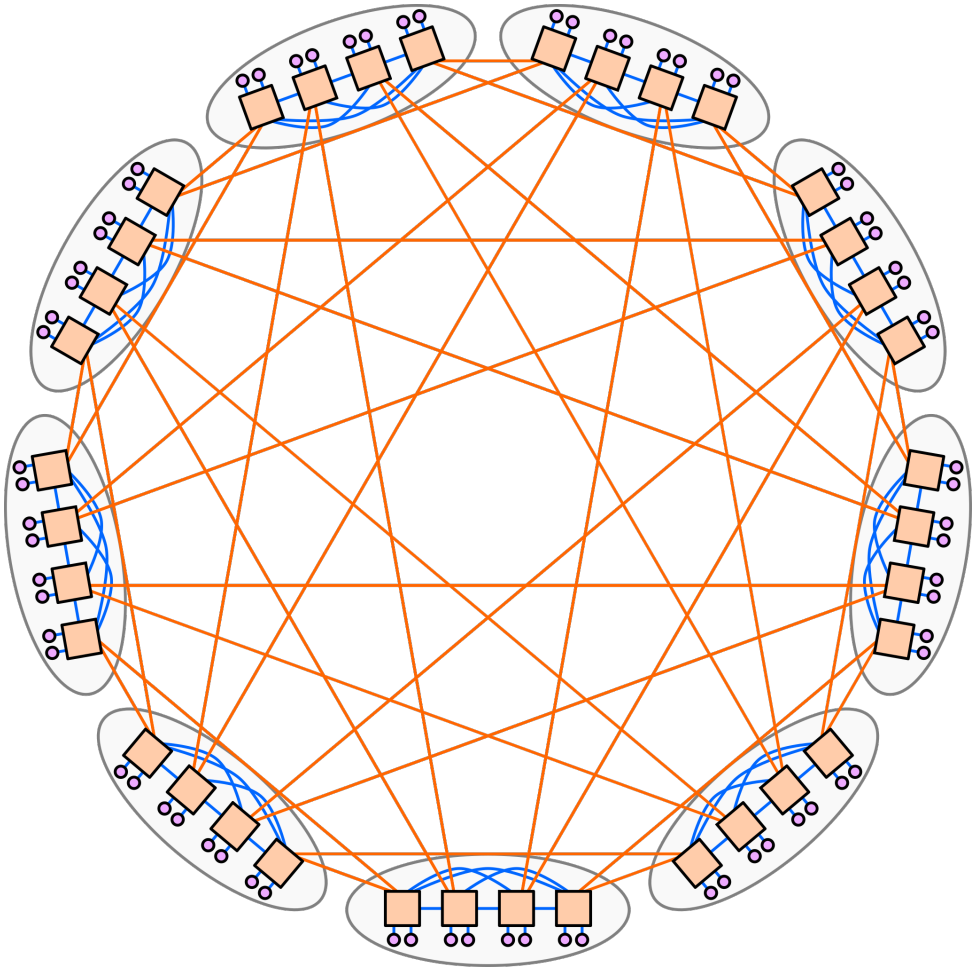

Рис. 1. Структура топологии Dragonfly

2 Особенности реализации

2.1 Класс Network

Класс `Network` предназначен для моделирования сетевой структуры заданной в варианте топологии, состоящей из нескольких групп, коммутаторов и узлов. Основная цель класса — управление сетевыми ресурсами, такими как пропускная способность внутри группы и между группами, а также обработка сообщений в сети.

Поля класса Network:

- `GROUPS` — количество групп в сети.
- `COMMUTATORS` — количество коммутаторов в каждой группе.
- `NODES` — количество узлов, подключенных к каждому коммутатору.
- `BANDWIDTH_IN_GROUP` — пропускная способность внутри группы.
- `BANDWIDTH_BETWEEN_GROUP` — пропускная способность между группами.
- `messages` — вектор, содержащий сообщения, обрабатываемые сетью.
- `status` — вектор строк, описывающих статус каждого сообщения.
- `path` — вектор векторов, содержащий пути передачи сообщений.
- `reserv` — вектор, описывающий количество зарезервированных ресурсов на каждом шаге передачи сообщения.
- `reminder` — вектор, содержащий информацию о том, сколько данных осталось передать между вершинами.
- `matrix_bandwidth` — матрица пропускных способностей сети.
- `matrix_load` — матрица загруженности сети.

2.2 Конструктор класса Network

Класс Network имеет параметризованный конструктор, который позволяет задать количество групп, коммутаторов, узлов, а также пропускную способность внутри группы и между группами. Помимо этого, он строит матрицу пропускных способностей и матрицу загруженности для заданной сети.

Реализация конструктора представлена в листинге 1.

Листинг 1. Конструктор класса Network

```
1       Network::Network(int GROUPS, int COMMUTATORS, int NODES, int
    BANDWIDTH_IN_GROUP, int BANDWIDTH_BETWEEN_GROUP)  {
2           this->GROUPS = GROUPS;
3           this->COMMUTATORS = COMMUTATORS;
4           this->NODES = NODES;
5           this->BANDWIDTH_IN_GROUP = BANDWIDTH_IN_GROUP;
6           this->BANDWIDTH_BETWEEN_GROUP = BANDWIDTH_BETWEEN_GROUP;
7
8           count_commutators = GROUPS * COMMUTATORS;
9           count_nodes = count_commutators * NODES;
10          count_vertex = count_commutators + count_nodes;
11
12          vector<vector<int>> new_bandwidth(count_vertex, vector<int>(
    count_vertex, 0));
13          matrix_bandwidth = new_bandwidth;
14
15
16          for (int i = 0, ii=count_nodes, k = NODES; i < count_nodes;i++)
     {
17              k--;
18              matrix_bandwidth[i][ii] = BANDWIDTH_IN_GROUP; //пропускная
    способность в группе
19              matrix_bandwidth[ii][i] = BANDWIDTH_IN_GROUP;
20              if (k == 0) {
21                  ii++;
22                  k = NODES;
23              }
24          }
25
26
27          for (int i = 0, back=COMMUTATORS-1, group=GROUPS-1; i <
    COMMUTATORS; i++, back--, group-=GROUPS/COMMUTATORS) {
28
29
30              for (int ii = count_nodes + i, group_right=group ; ii <
    count_vertex; ii+=COMMUTATORS, group_right++) {
31                  if (group_right >= GROUPS) {
32                      group_right -= GROUPS;
33                  }
34                  for (int iii = 0, current_group = group_right; iii <
    GROUPS/COMMUTATORS; iii++) {
35                      current_group = (group_right - iii) < 0 ? GROUPS +
```

```
    (group_right - iii) : group_right - iii;
36                      int buf = count_nodes + current_group * COMMUTATORS
     + back;
37                      matrix_bandwidth[ii][count_nodes + current_group *
    COMMUTATORS + back] = BANDWIDTH_BETWEEN_GROUP;//пропускная способность м
    ежду группами
38                      matrix_bandwidth[count_nodes + current_group *
    COMMUTATORS + back][ii] = BANDWIDTH_BETWEEN_GROUP;
39                  }
40              }
41          }
42
43          for (int i = 0; i < GROUPS; i++) {
44              for (int ii = count_nodes + i * COMMUTATORS; ii <
    count_nodes + (i + 1) * COMMUTATORS; ii++) {
45                  for (int iii = ii; iii < count_nodes + (i + 1) *
    COMMUTATORS; iii++) {
46                      if (ii != iii) {
47                          matrix_bandwidth[ii][iii] = BANDWIDTH_IN_GROUP;
    //пропускная способность в группе
48                          matrix_bandwidth[iii][ii] = BANDWIDTH_IN_GROUP;
49                      }
50                  }
51              }
52          }
53
54          matrix_load = matrix_bandwidth;
55      }
56
```

2.3 Модифицированный алгоритм Дейкстры

Метод `Dijkstra_algorythm` реализует модифицированный алгоритм Дейкстры для поиска кратчайшего пути в сети с учетом загруженности соединений и штрафов за пересечение промежуточных узлов.

Метод принимает параметр `message` — вектор из трех элементов: `message[0]` — исходный узел, `message[1]` — длина сообщения, `message[2]` — конечный узел.

Реализация алгоритма представлена в листинге 2.

Листинг 2. Модифицированный алгоритм Дейкстры

```
1       vector<int> Network::Dijkstra_algorythm(vector<int> message) {
2           int index_from = message[0];
3           int index_to = message[2];
```

```
            int cost = message[1];

            int count = 0;
            vector<int> vertex(count_vertex);
            for (int i = 0; i < vertex.size(); i++) {
                vertex[i] = INT_MAX;
            }
            vector<bool> vertex_check(count_vertex, false);
            vector<vector<int>> route(count_vertex);
            vertex[index_from] = 0;

            int current;
            while (CheckBoolVector(vertex_check) == false) {
                current = Min_Index(vertex, vertex_check, cost) ;
                if (current == -1)
                break;
                vertex_check[current] = true;

                for (int i = 0; i < matrix_load[current].size(); i++) {
                    if (matrix_load[current][i] != 0) {
                        int buf = cost / matrix_load[current][i];
                        buf = cost % matrix_load[current][i]==0? buf :buf +
 1;
                        if (!(current == index_from || current == index_to)
) {
                            buf++; //прибавляем штраф
                        }
                        bool condition = vertex[current] + buf <= vertex[i
];
                        if (condition) {

                            vertex_check[i] = false;
                            route[i].clear();
                            route[i] = route[current];
                            route[i].push_back(current);
                            vertex[i] = vertex[current] + buf;
                        }
                    }
                }
            }
            for (int i = 0; i < vertex.size(); i++) {
                if (vertex[i] == INT_MAX || vertex[i] == INT_MAX + 1)
                vertex[i] = 0;
            }
```

```
45              for (int i = 0; i < route.size(); i++) {
46                  if (route[i].size() != 0)
47                  route[i].push_back(i);
48              }
49              vector<int> current_route;
50              for (int i = 0; i < route[index_to].size() - 1; i++) {
51                  int index_1;
52                  int index_2;
53                  int cost_inner;
54                  int broad;
55                  if (i != route[index_to].size() - 2) {
56                      index_1 = route[index_to][i];
57                      index_2 = route[index_to][i + 1];
58                      cost_inner = matrix_load[index_1][index_2];
59
60                      broad = cost / cost_inner;
61                      broad = cost % cost_inner == 0 ? broad : broad+1;
62                      for (int ii = 0; ii < broad; ii++) {
63                          current_route.push_back(index_1);
64                          current_route.push_back(index_2);
65                      }
66
67                      //добавим штраф
68                      current_route.push_back(index_2);
69                      current_route.push_back(index_2);
70                  }
71                  else {
72                      cost_inner = this->BANDWIDTH_IN_GROUP;
73                      index_1 = route[index_to][i];
74                      index_2 = route[index_to][i+1];
75
76                      broad = cost / cost_inner;
77                      broad = cost % cost_inner == 0 ? broad : broad+1;
78                      for (int ii = 0; ii < broad; ii++) {
79                          current_route.push_back(index_1);
80                          current_route.push_back(index_2);
81                      }
82                  }
83              }
84              return current_route;
85          }
86
87
```

2.4 Добавление нового сообщения

Метод `adding_msg` отвечает за добавление сообщения в систему. Он запрашивает у пользователя ввод данных о сообщении в формате:

```
АДРЕС_ИСТОЧНИКА ДЛИНА_СООБЩЕНИЯ АДРЕС_ПОЛУЧАТЕЛЯ
```

где адреса находятся в пределах от 0 до `count_nodes - 1` включительно. Метод проверяет на корректность вводимых данных и вызывает метод AddMessage, который добавляет сообщение в вектор `messages`, вызывает алгоритм Дейкстры, чтобы получить путь для передачи сообщения, устанавливает статус для нового сообщения “Сообщение ожидает отправки”. Также метод определяет количество отправляемых данных, проверяя, возможно ли передать все данные по текущему пути, и обновляет матрицу нагрузки `matrix_load`.

Реализация методов `adding_msg` и представлена листинге 3.

Листинг 3. Добавление нового сообщения

```
1         void Network::adding_msg() {
2             string input;
3             int dep, length, arr;
4             char space1, space2;
5             cout << "\nВведите сообщение в формате АДРЕС_ИСТОЧНИКА ДЛИНА_СО
    ОБЩЕНИЯ АДРЕС_ПОЛУЧАТЕЛЯ, " << endl;
6             cout << "где адреса находятся в пределах от 0 до " <<
    count_nodes - 1 << " включительно" << endl;
7
8             do {
9                 int flag_out1 = false;
10                int flag_out2 = false;
11
12                getline(cin, input);
13                istringstream iss(input);
14
15                if (iss >> dep >> length >> arr) {
16
17                    if (dep == arr) {
18                        cout << "Введите различные узел отправления и прибы
    тия" << endl;
19                    }
20                    else if (length <= 0) {
21                        cout << "Длина сообщения должна быть положительной!
    " << endl;
22                    }
23                    else if (dep >= 0 && dep < count_nodes && arr >= 0 &&
```

```
    arr < count_nodes && iss.eof()) {
24                      flag_out1 = true;
25                  }
26                  else {
27                      cout << "Некорретный ввод! Попробуйте снова" <<
    endl;
28                  }
29              }
30              else cout << "Некорретный ввод! Попробуйте снова" << endl;
31
32              if (messages.size() == 0) {
33                  flag_out2 = true;
34              }
35              else {
36                  for (int i = 0; i < messages.size(); i++) {
37                      if (dep != messages[i][0] ||
38                      length != messages[i][1] ||
39                      arr != messages[i][2]) {
40                          flag_out2 = true;
41                      }
42                  }
43              }
44              if (flag_out2 == false) {
45                  cout << "Такое собщение уже существует! Попробуйте снов
    a" << endl;
46              }
47              if (flag_out1 && flag_out2)
48              break;
49          } while (true);
50
51          this->AddMessage({ dep, length, arr });
52      }
53      void Network::AddMessage(vector<int> message) {
54          vector <int> path;
55          this->messages.push_back(message);
56          if (Can_Make_Route(message))
57          path = Dijkstra_algorythm(message);
58          this->path.push_back(path);
59          this->status.push_back({ "Сообщение ожидает отправки" });
60          this->reminder.push_back(message[1]);
61
62          if (Can_Make_Route(message)) {
63              int can_send = message[1] < matrix_load[path[0]][path[1]] ?
     message[1] : matrix_load[path[0]][path[1]];
```

```
64                  this->reserv.push_back(can_send);
65                  matrix_load[path[0]][path[1]] -= can_send;
66              }
67              else
68              this->reserv.push_back(0);
69          }
70
```

2.5 Метод NextStep

Метод `NextStep` выполняет шаг передачи сообщений в сети. Он проходит по всем сообщениям и обрабатывает их в зависимости от текущего состояния. Основные действия метода:

- Итерирует по всем сообщениям в векторе `messages`.
- Если сообщение уже передается, проверяет, в каком состоянии находится передача:
  - Если сообщение находится в одном узле из-за штрафа, проверяет наличие свободного пути для передачи. Если путь доступен, вызывает алгоритм Дейкстры для нахождения нового маршрута и обновляет соответствующие параметры (остаток сообщения, матрицу нагрузки).
  - Если сообщение передается из одного узла в другой, проверяет доступную пропускную способность и обновляет матрицы нагрузки, резерв и остаток сообщения.
- После обработки каждого сообщения проверяет, нужно ли удалить текущий шаг маршрута. Если сообщение успешно доставлено, оно удаляется из всех векторов, а в противном случае — просто обновляется статус.
- В конце метода восстанавливает матрицу нагрузки до первоначального состояния с помощью `matrix_bandwidth`.

Реализация метода представлена листинге 4.

Листинг 4. Метод NextStep

```
1
2       void Network::NextStep() {
3           for (int i = 0; i < messages.size(); i++) {
4               if (status[i].size() == 0) {//рассматриваются ситуации когд
    а сообщение уже передается
5
6                   int local_from = path[i][0]; //точка в которой сейчас н
    аходится сообщение
```

```
7                   int local_to = path[i][1]; //точка в которую передается
    шаг на данном шаге
8                   int index_from = messages[i][0]; //отправитель
9                   int index_to = messages[i][2]; //получатель
10                  int message_cost = messages[i][1]; //длина сообщения
11
12                  if (local_from == path[i][1]) {// ситуация когда сообще
    ние находится в одном узле из за штрафа (то есть никуда не передается)
13                      reserv[i] = 0;
14                      bool check = false;
15                      for (int ii = 0; ii < count_vertex; ii++) {
16                          if (matrix_load[ii][index_to] != 0) {
17                              check = true;
18                              break;
19                          }
20                      }
21                      if (check) {
22                          vector<int> new_path = Dijkstra_algorythm({
    local_from, message_cost, index_to });
23                          new_path.insert(new_path.begin(), { path[i][0],
    path[i][1] });
24                          path[i] = new_path;
25                          reminder[i] = message_cost;
26
27                          int can_send = message_cost < matrix_load[path[
    i][2]][path[i][3]] ? message_cost : matrix_load[path[i][2]][path[i][3]];
28
29                          reserv[i] = can_send;
30                          matrix_load[path[i][2]][path[i][3]] -= can_send
    ;
31                          matrix_load[path[i][3]][path[i][2]] -= can_send
    ;
32                      }
33                      else {
34                          string status = "Ожидает освобождения пути. Нах
    одится в " + to_string(local_from) + " коммутаторе";
35                          this->status[i].push_back({ status });
36                      }
37                  }
38                  else { //ситуация когда передается сообщение из одного
    узда в другой
39                      //пропускная способность изменяется
40
41                      if (matrix_load[local_from][local_to] > 0) {
```

```
42                          int can_send = reminder[i] < matrix_load[
    local_from][local_to] ? reminder[i] : matrix_load[local_from][local_to];
43
44                          int need_steps = reminder[i] / can_send;
45                          need_steps = reminder[i] % can_send == 0 ?
    need_steps : need_steps + 1;
46
47                          matrix_load[local_from][local_to] -= can_send;
48                          matrix_load[local_to][local_from] -= can_send;
49                          reserv[i] = can_send;
50
51                          if (path[i].size() > 2) {//сообщение может изме
    ниться если пропускная способность увеличилась
52
53                              int old_steps = 0;//считаем сколько было ша
    гов
54                              for (int ii = 0; ii < path[i].size() - 2;
    ii += 2) {
55                                  if (local_from == path[i][ii] &&
    local_to == path[i][ii + 1]) {
56                                      old_steps++;
57                                  }
58                                  else {
59                                      break;
60                                  }
61                              }
62                              if (need_steps< old_steps) {
63
64                                  for (int iii = 0; iii < old_steps -
    need_steps; iii++) {
65                                      path[i].erase(path[i].begin());
66                                      path[i].erase(path[i].begin());
67                                  }
68
69                              }
70                          }
71                          reminder[i] -= can_send;
72                      }
73                      else {
74                          reminder[i] -= reserv[i];
75                      }
76
77                  }
78                  //в конце шага нам нужно удалить данный шаг
```

```
79                      if (path[i].size() > 2) {
80                          if (reserv[i] == 0) {
81                              matrix_load[path[i][0]][path[i][1]] += reserv[i
    ];
82                              matrix_load[path[i][1]][path[i][0]] += reserv[i
    ];
83                          }
84                          path[i].erase(path[i].begin());
85                          path[i].erase(path[i].begin());
86
87                      }
88                      else {
89                          messages.erase(messages.begin() + i);
90                          matrix_load[path[i][0]][path[i][1]] += reserv[i];
91                          matrix_load[path[i][1]][path[i][0]] += reserv[i];
92                          path.erase(path.begin() + i);
93                          status.erase(status.begin() + i);
94                          reminder.erase(reminder.begin() + i);
95                          reserv.erase(reserv.begin() + i);
96                          i--;
97                      }
98                  }
99                  else if(path[i].size()>0 && path[i][0]!= messages[i][0]) {
    // когда в штрафе, а затем ожидание происходит
100
101                     int local_from = path[i][0]; //точка в которой сейчас н
    аходится сообщение
102                     int local_to = path[i][1]; //точка в которую передается
     шаг на данном шаге
103                     int index_from = messages[i][0]; //отправитель
104                     int index_to = messages[i][2]; //получатель
105                     int message_cost = messages[i][1]; //длина сообщения
106
107                     reserv[i] = 0;
108                     bool check = false;
109                     for (int ii = 0; ii < count_vertex; ii++) {
110                         if (matrix_load[ii][index_to] != 0) {
111                             check = true;
112                             break;
113                         }
114                     }
115                     if (check) {
116                         vector<int> new_path = Dijkstra_algorythm({
    local_from, message_cost, index_to });
```

```
117                         new_path.insert(new_path.begin(), { path[i][0],path
    [i][1] });
118                         path[i] = new_path;
119                         reminder[i] = message_cost;
120 
121                         int can_send = message_cost < matrix_load[path[i
    ][2]][path[i][3]] ? message_cost : matrix_load[path[i][2]][path[i][3]];
122 
123                         reserv[i] = can_send;
124                         matrix_load[path[i][2]][path[i][3]] -= can_send;
125                         matrix_load[path[i][3]][path[i][2]] -= can_send;
126                     }
127                 }
128             }
129             for (int i = 0; i < messages.size(); i++) {
130 
131                 if (status[i].size() != 0) {
132                     if (path[i].size() > 0 && path[i][0] != messages[i][0])
     { //когда ожидает освобождения пути
133                         bool check = false;
134                         for (int ii = 0; ii < count_vertex; ii++) {
135                             if (matrix_load[ii][messages[i][2]] != 0) {
136                                 check = true;
137                                 break;
138                             }
139                         }
140                         if (check) {
141                             status[i].erase(status[i].begin());
142                         }
143                     }
144                     else {
145                         if (Can_Make_Route(messages[i])) {//это для начальн
    ых этапов
146                             status[i].erase(status[i].begin());
147                             if (reserv[i] == 0) {
148                                 path[i] = Dijkstra_algorythm(messages[i]);
149                                 int can_send = messages[i][1] < matrix_load
    [path[i][0]][path[i][1]] ? messages[i][1] : matrix_load[path[i][0]][path
    [i][1]];
150                                 reserv[i] = can_send;
151                                 matrix_load[path[i][0]][path[i][1]] -=
    can_send;
152                                 matrix_load[path[i][1]][path[i][0]] -=
    can_send;
```

```
                                }
                            }
                        }
                    }
                }
                matrix_load = matrix_bandwidth;
            }


```

2.6 Метод Can_Make_Route

Метод `Can_Make_Route` проверяет, возможно ли создать маршрут для заданного сообщения. Он выполняет следующие действия:

- Итерирует по всем сообщениям в векторе `messages` и проверяет, совпадает ли текущее сообщение с любым из предыдущих сообщений. Если найдено совпадение и резерв больше нуля, устанавливается флаг `flag1`.
- Проверяет наличие свободных маршрутов в матрице нагрузки для узла отправителя. Если хотя бы один узел имеет доступную пропускную способность, устанавливается флаг `flag2`.
- Возвращает `true`, если выполняется хотя бы одно из условий (флаги `flag1` или `flag2` равны `true`).

Реализация метода представлена листинге 5.

Листинг 5. Метод Can_Make_RouteStep

```
bool Network::Can_Make_Route(vector<int> message) {
    bool flag1 = false;
    bool flag2 = false;
    bool flag3 = false;
    for (int i = 0; i < messages.size(); i++) {
        if (message[0] == messages[i][0] &&
        message[1] == messages[i][1] &&
        message[2] == messages[i][2]){

            if (i < reserv.size() && reserv[i] > 0) {
                flag1 = true;
            }
        }
    }
    for (int i = 0; i < count_vertex; i++) {
```

```
16                  if (matrix_load[message[0]][i])
17                      flag2 = true;
18              }
19              return flag1 || flag2;
20          }
21
```

3 Пример работы программы

```
Программа, моделирующая работу механизма передачи сообщения в коммуникационной сети суперкомпьютера

Выберите студента:
[1] Алексей Шихалев {13, 4, 3, 4, 3}
[2] Игорь Гладков {11, 5, 3, 4, 2}
[3] Никита Ромашко {10, 3, 4, 3, 3}
[4] Георгий Золоев {7, 6, 3, 2, 3}
[0] Выход из программы
2

Количество групп: 11
Количество коммутаторов в группе: 5
Количество узлов на коммутатор: 3
Пропускная способность внутри группы: 4
Пропускная способность между группами: 2


---

Список сообщений:
Нет сообщений для пересылки!

Выберите действие:
[1] Отправить сообщения (до 10 штук)
[0] Сменить студента
1
Введите количество сообщений, которое хотите отправить (от 0 до 10): 2

Введите сообщение в формате АДРЕС_ИСТОЧНИКА ДЛИНА_СООБЩЕНИЯ АДРЕС_ПОЛУЧАТЕЛЯ,
где адреса находятся в пределах от 0 до 164 включительно
0 3 4

Введите сообщение в формате АДРЕС_ИСТОЧНИКА ДЛИНА_СООБЩЕНИЯ АДРЕС_ПОЛУЧАТЕЛЯ,
где адреса находятся в пределах от 0 до 164 включительно
5 1 15
---

Список сообщений:

```

```
{0,3,4} : Сообщение ожидает отправки
{5,1,15} : Сообщение ожидает отправки

Выберите действие:
[1] Отправить сообщения (до 10 штук)
[2] Следующий шаг
[0] Сменить студента
2
```

```
Следующий шаг.
Список сообщений:

{0,3,4} : Из узла 0 в коммутатор 165 передано 3/3
{5,1,15} : Из узла 5 в коммутатор 166 передано 1/1

Выберите действие:
[1] Отправить сообщения (до 10 штук)
[2] Следующий шаг
[0] Сменить студента
2
```

```
Следующий шаг.
Список сообщений:

{0,3,4} : Штраф. Находится в 165 коммутаторе
{5,1,15} : Штраф. Находится в 166 коммутаторе

Выберите действие:
[1] Отправить сообщения (до 10 штук)
[2] Следующий шаг
[0] Сменить студента
2
```

```
Следующий шаг.
Список сообщений:

{0,3,4} : Из коммутатора 165 в коммутатор 166 передано 3/3
{5,1,15} : Из коммутатора 166 в коммутатор 169 передано 1/1

Выберите действие:
[1] Отправить сообщения (до 10 штук)
```

```
[2] Следующий шаг
[0] Сменить студента
2
```

```
Следующий шаг.
Список сообщений:

{0,3,4} : Штраф. Находится в 166 коммутаторе
{5,1,15} : Штраф. Находится в 169 коммутаторе

Выберите действие:
[1] Отправить сообщения (до 10 штук)
[2] Следующий шаг
[0] Сменить студента
2
```

```
Следующий шаг.
Список сообщений:

{0,3,4} : Из коммутатора 166 в узел 4 передано 3/3
{5,1,15} : Из коммутатора 169 в коммутатор 170 передано 1/1

Выберите действие:
[1] Отправить сообщения (до 10 штук)
[2] Следующий шаг
[0] Сменить студента
2
```

```
Следующий шаг.
Список сообщений:

{5,1,15} : Штраф. Находится в 170 коммутаторе

Выберите действие:
[1] Отправить сообщения (до 10 штук)
[2] Следующий шаг
[0] Сменить студента
2
```

```
Следующий шаг.
```

```
Список сообщений:

{5,1,15} : Из коммутатора 170 в узел 15 передано 1/1

Выберите действие:
[1] Отправить сообщения (до 10 штук)
[2] Следующий шаг
[0] Сменить студента
2
```

```
Следующий шаг.
Список сообщений:

Нет сообщений для пересылки!

Выберите действие:
[1] Отправить сообщения (до 10 штук)
[0] Сменить студента
```

Заключение

В процессе выполнения работы было реализовано консольное приложение, моделирующее работу механизма передачи сообщения в коммуникационной сети суперкомпьютера. Все поставленные задачи были выполнены:

1. Заданная конфигурация соединений предложенной топологии была описана с помощью матриц смежности и пропускных способностей.

2. Реализана функция отправки сообщения, которая вызывается пользователем и включает в себя функцию вычисления маршрута передачи сообщения с целью минимизации времени, необходимого для передачи. Формат сообщения: “АДРЕС_ИСТОЧНИКА ДЛИНА_СООБЩЕНИЯ АДРЕС_ПОЛУЧАТЕЛЯ”.

3. Реализована возможность продвигаться по шагам, и в начале каждого шага пользователь может указать от 0 до 10 заданий на пересылку сообщений с произвольными адресами источника и получателя. ДЛИНА_СООБЩЕНИЯ — это есть число шагов (количество единиц времени), требуемых для его передачи по свободному каналу. Важной особенностью является штраф за каждую пересылку через промежуточный узел или через коммутатор в 1 дополнительный шаг времени.

Работа была выполнена в среде Visual Studio 2022 на языке программирования C++.

Список литературы

[1] В. Олифер, Н. Олифер. Компьютерные сети. Принципы, технологии, протоколы. — Питер, 2013. — С. 55. — 944 с. — 3000 экз.

[2] Russell J. Super-Connecting the Supercomputers: Innovations Through Network Topologies [Электронный ресурс]. HPCwire. 2019. URL: https://www.hpcwire.com/2019/07/15/super-connecting-the-supercomputers-innovations-through-network-topologies/ (дата обращения: 25.09.2024).

[3] Huawei Network. Dragonfly Topology | Test It, Believe It Series for Data Center Networks [Видео]. YouTube, 2019. URL: https://www.youtube.com/watch?v=atKMrPmTOXY (дата обращения: 25.09.2024).