МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ» (НГУ)

Кафедра параллельных вычислений

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА**

Курныш Юрий Владимирович

# Сравнительный анализ и реализация алгоритмов нерегулярной перефрагментации данных

Направление подготовки 230100.62 ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА

**Руководитель**

Киреев С. Е.
б/с, ст. преподаватель

…………………………..
(подпись, дата)

**Автор**

Курныш Ю. В.
ФИТ, 7203

..………………..
(подпись, дата)

Новосибирск, 2011 г.

# Содержание

ВВЕДЕНИЕ ........ 3
1 Постановка задачи перефрагментации данных ........ 5
2 Обзор решений задачи перефрагментации данных ........ 7
 2.1 Теоретический анализ задачи ........ 7
 2.2 Обзор эвристических алгоритмов ........ 7
3 Алгоритмы решения задачи ........ 9
 3.1 Алгоритм DRC ........ 9
 3.2 Модификация алгоритма DRC (sDRC) ........ 10
 3.3 Алгоритм GGP ........ 10
4 Сравнительное тестирование алгоритмов ........ 14
 4.1 Сравнение времен работы алгоритмов ........ 15
 4.2 Сравнение времен выполнения обменов ........ 16
 4.3 Сравнение суммарных временных затрат на организацию обменов ........ 18
Заключение ........ 20
Литература ........ 21
Приложение А. Листинги программ тестирования алгоритмов. ........ 22

# ВВЕДЕНИЕ

В настоящее время большие вычислительные мощности, представленные различными параллельными компьютерными системами (многопроцессорные серверы, компьютерные сети, суперкомпьютеры, грид-системы и т.п.), стали доступны широкому кругу пользователей. Это позволяет за приемлемое время решать задачи с большим объемом данных и вычислений, которые раньше невозможно было реализовать на последовательных компьютерах. Но возможности параллельных вычислений непосредственно связаны со сложностями параллельного программирования, которые в свете расширенного круга пользователей встают с особой остротой. Параллельной программе приходится управлять во времени системами, достигающими тысяч процессоров и терабайт оперативной памяти.

Одной из основных трудностей, возникающих перед программистом, является работа с распределёнными данными. Каждый параллельный алгоритм требует особого распределения данных по вычислительным узлам. При использовании различных параллельных численных алгоритмов на разных стадиях вычислений приходится выполнять перефрагментацию данных между стадиями. Под перефрагментацией мы понимаем перераспределение данных между узлами распределенной вычислительной системы. Правильная перефрагментация помогает сбалансировать вычислительную нагрузку, увеличить локальность данных и уменьшить межузловые коммуникации в процессе вычислений. Некоторые языки программирования, например, High Performance Fortran [2], поддерживают примитивы перераспределения данных. Перефрагментация данных делится на две категории: регулярная и нерегулярная. При регулярной перефрагментации рассматривается распределение данных по процессорам блоками одинакового размера. При нерегулярной перефрагментации, которая обсуждается в настоящей работе, рассматривается распределение данных по процессорам блоками различного размера.

На практике даже в хорошо изученном и освоенном последовательном программировании отношение «полезной» производительности процессора к его пиковой производительности превышает 50 процентов лишь для библиотечных функций, разработанных специалистами. Обычные пользовательские вычислительные программы реализуют лишь 10-20 процентов от возможностей процессора. В параллельном программировании эта величина еще значительно меньше, в том числе из-за накладных расходов на обмены данными. Поэтому вопрос об эффективной перефрагментации данных занимает одно из главных мест. Это и определяет актуальность темы работы.

Существует проблема поиска оптимального алгоритма перефрагментации данных. Если регулярная перефрагментация данных – полиномиально разрешимый случай, то нерегулярная перефрагментация данных – NP-трудная задача. Для решения NP-трудных задач за реальное время обычно используются эвристические алгоритмы.

Целью настоящей работы является:

- обзор эвристических алгоритмов нерегулярной перефрагментации данных,
- сравнение алгоритмов по теоретическим характеристикам и по результатам тестирования на реальных вычислительных системах.

# 1 Постановка задачи перефрагментации данных

Есть N вычислительных узлов, входящих в состав распределенной вычислительной системы. Заданы два распределения данных по узлам: начальное и конечное (рис. 1). Требуется построить схему передач данных, оптимальную по некоторому критерию (например, время и стоимость пересылок).

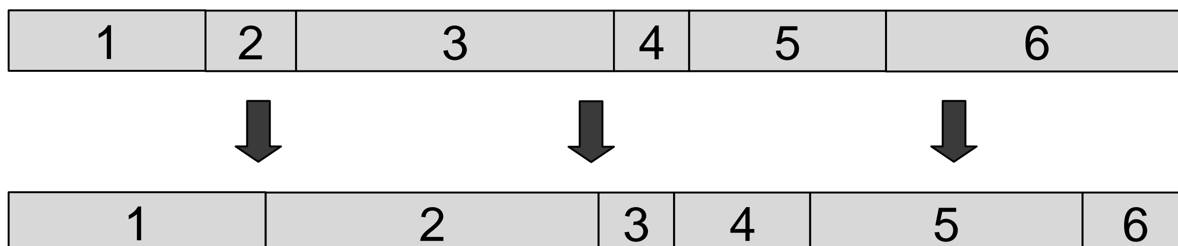



Рис. 1. Пример начального и конечного распределения данных по узлам.

Сформулируем задачу в следующем виде. Множество всех пересылок данных можно представить в виде взвешенного ориентированного двудольного графа G=(A, B, E), где вершины A=B={1 ,..., N} – множество узлов вычислительной системы, рёбра $E \subset \{1,...,N\} \times \{1,...,N\}$ – множество пересылок (рис. 2). Вес ребра $(v_1, v_2)$ – время передачи сообщения от узла $v_1$ к узлу $v_2$. Время инициализации передач не учитывается. Мы считаем, что коммуникационная среда имеет ограниченную пропускную способность и не может одновременно выполнять некоторые передачи. Поэтому возникает задача планирования обменов. Введём следующее ограничение коммуникационной среды: будем предполагать, что узел в один момент времени может участвовать только в одном приеме и в одной отправке. Обмены, которые не могут выполняться одновременно, будем называть конфликтующими.

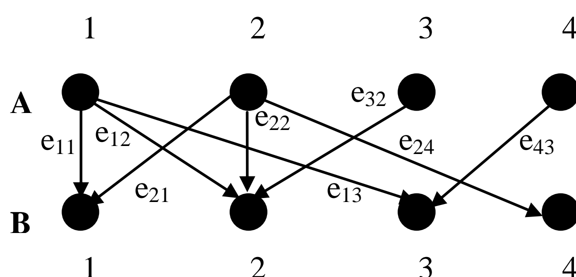



Рис. 2. Пример графа коммуникаций.

Требуется построить расписание обменов таким образом, чтобы суммарное время, затраченное на организацию обменов, было минимальным. Решение будем представлять в виде списков обменов для каждого узла.

К алгоритмам планирования может быть предъявлен ряд требований:

– *Универсальность алгоритма.* Алгоритм должен решать задачу для любого графа коммуникаций.
– *Высокая эффективность алгоритма.* Под эффективностью алгоритма можно, например, понимать величину: $E = T_U / (T_A + T_U)$, где $T_A$ – время работы алгоритма, $T_U$ – время работы остальной части параллельной программы (в которой данный алгоритм используется) без алгоритма.
– *Хорошая масштабируемость алгоритма*. Алгоритм должен сохранять достаточную эффективность при росте числа процессоров. Данное требование следует из требования масштабируемости самих параллельных программ, где данный алгоритм будет использоваться.

# 2 Обзор решений задачи перефрагментации данных

## 2.1 Теоретический анализ задачи

В статье [6] показано, что задача планирования на двудольных графах является NP-трудной. Для её решения за реальное время требуются эвристические алгоритмы.

Продемонстрируем, что неэвристические алгоритмы являются слишком трудоемкими. Для нахождения точного решения поставленной задачи можно выполнить полный перебор всех вариантов расписаний обменов. Сложность полного перебора по всем перестановкам на множестве сообщений имеет порядок $O(|E|!)$. Например, для числа вершин $N=100$ и числа рёбер $|E|=1000$ число операций будет порядка $4{,}024 \cdot 10^{567}$. Это значит, что при производительности процессора 6 млрд. операций в секунду задача бы решалась за $6{,}7 \cdot 10^{557}$ секунд ($2{,}12 \cdot 10^{550}$ лет).

Другим способом поиска точного решения может быть метод динамического программирования. Сложность алгоритма $O(2^{|E|})$, т. к. решение исходной задачи строится из решений подзадач с меньшим числом рёбер. Это значит, что та же задача при той же производительности процессора решалась бы более чем за $1{,}6 \cdot 10^{115}$ секунд ($5 \cdot 10^{107}$ лет).

## 2.2 Обзор эвристических алгоритмов

Был выполнен обзор работ, в которых ставилась задача перефрагментации данных. Было выделено несколько подходов к решению данной проблемы.

Один из используемых подходов состоит в том, что перефрагментация данных вообще не происходит. Вместо этого выполняется миграция процессов [1].

В ряде работ существует задача перефрагментации данных, но проблема планирования обменов не рассматривается. Например, в библиотеке KeLP [7] передача происходит за две стадии: 1) асинхронная отправка всех сообщений, 2) асинхронный прием всех сообщений. Порядок отправки специально не оптимизируется. При такой схеме обменов неизбежна конкуренция потоков данных за канал связи, что может привести к дополнительным накладным расходам.

Большое число работ посвящено планированию обменов для некоторых специальных случаев графов коммуникаций. Многие из них рассматривает ограниченную задачу планирования обменов только для ступенчатой матрицы обменов. Такая матрица обменов получается, например, в результате применения оператора REDISTRIBUTE в языке HPF [2]. Ниже рассмотрены несколько алгоритмов, разработанных с учетом этого ограничения.

- В работе [4] представлен алгоритм Divide-and-Conquer, который состоит из двух частей: 1) разделение исходной задачи на подзадачи, которые похожи на исходную

задачу, но с меньшим размером; 2) рекурсивное решение этих подзадач, построение решения исходной задачи путём комбинирования решений этих подзадач. Данный алгоритм даёт точное решение задачи.

- В работе [3] представлен List algorithm, который основан на том, что оптимальное решение равно максимальной мощности строк и столбцов матрицы смежности графа коммуникаций. Он состоит из двух стадий: на стадии планирования находится приближенное решение задачи, на стадии перераспределения найденное решение улучшается. Весь алгоритм даёт точное решение задачи.
- Алгоритм DRC (Degree Reduction and Coloring), представленный в работе [5], основан на поиске хроматического индекса графа коммуникаций. Он состоит из двух стадий: 1) стадии уменьшения степени графа до второй и 2) стадии 2-раскраски. В отличие от других алгоритмов, он может быть использован и для решения задачи в более общей постановке.

Еще один алгоритм для специального случая обменов вида All-to-all (для полного графа коммуникаций) представлен в работе [8].

Наконец, существуют алгоритмы, которые решают более общую задачу планирования. Алгоритм GGP (Generic Graph Peeling) [6] решает задачу планирования передач данных между кластерами при наличии канала связи с ограниченной пропускной способностью (рис. 3). В ходе работы алгоритма граф коммуникаций трансформируются в регулярный (по весу) граф, из которого путем поиска максимальных паросочетаний формируются стадии обменов.

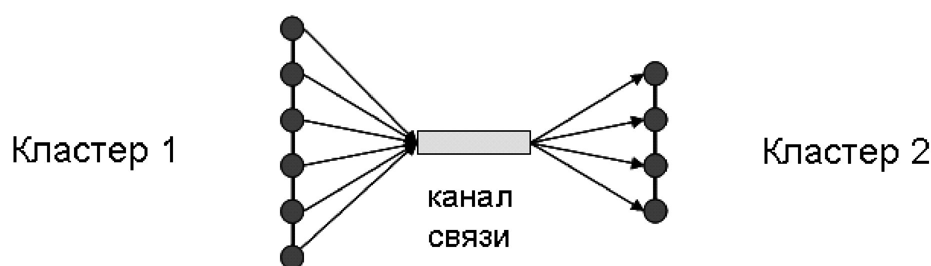
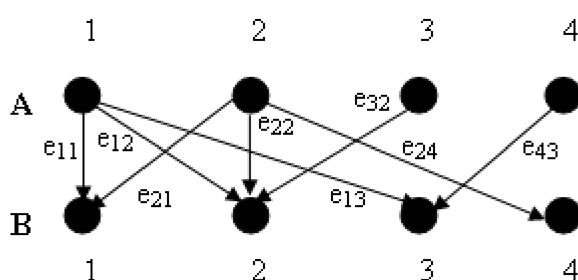



Рис. 3. Задача планирования передач данных между кластерами

Для дальнейшего рассмотрения были выбраны следующие алгоритмы: DRC и GGP. Алгоритм GGP решает более общую задачу, в том числе и задачу в приведенной в главе 2 постановке. Алгоритм DRC хотя и был разработан для решения более узкой задачи, но подходит и для решения задачи в постановке, которая требуется в настоящей работе.

# 3 Алгоритмы решения задачи

Рассматриваемые далее алгоритмы находят псевдооптимальное решение задачи планирования обменов в виде расписания обменов. Расписание имеет вид последовательности шагов (рис. 4б). Каждый шаг содержит множество неконфликтующих обменов между узлами. Будем предполагать, что узел в один момент времени может участвовать только в одном приеме и в одной отправке. Обмены, которые не могут выполняться одновременно, будем называть конфликтующими.




(а)

| Расписание обменов | |
|---|---|
| Шаг 1: | $e_{11}$, $e_{22}$, $e_{43}$ |
| Шаг 2: | $e_{12}$, $e_{24}$, $e_{32}$ |
| Шаг 3: | $e_{21}$, $e_{13}$ |

(б)

Рис. 4. Вход и выход алгоритмов построения расписаний: граф коммуникаций (а) и расписание обменов (б).

## 3.1 Алгоритм DRC

Далее приведен алгоритм DRC, как он представлен в работе [5], но с незначительной модификацией, позволяющей применить его для произвольного графа коммуникаций. Вход алгоритма: граф коммуникаций G, выход: расписание обменов (рис. 4).

Алгоритм DRC.

1. Генерация списка сообщений.
2. step = 0.
3. Отсортировать список сообщений в порядке убывания веса ребра.
4. *Этап уменьшения степени графа.* Повторять, пока deg(G)>2:
   4.1. Выбрать из списка первое слева сообщение (с максимальным весом) без конфликта, удалить его из графа и списка и включить в шаг с номером step.
   4.2. step=step+1.
5. *Этап раскраски графа.* Повторять, пока остались нераскрашенные сообщения:
   5.1. Выбрать нераскрашенное сообщение с максимальным весом.
   5.2. Выполнить 2-раскраску компоненты связности, в которую входит выбранное сообщение.

6. Сформировать из раскрашенных вершин две последние стадии обменов.

Здесь и далее deg(G) – степень графа G. Алгоритм DRC находит псевдооптимальное решение задачи минимизации времени обменов для любого графа коммуникаций. Сложность алгоритма O(|E| log |E|+N|E|), т.к. она складывается из сложности алгоритма сортировки и сложности N проходов отсортированного списка. Объем памяти, используемой алгоритмом, имеет порядок $O(N^2)$.

## 3.2 Модификация алгоритма DRC (sDRC)

Можно выполнить следующую модификацию алгоритма DRC, которая заключается в следующем: этап раскраски не выполняется, а этап уменьшения степени графа продолжается до полного исчерпания рёбер. Полученный алгоритм будем называть sDRC (simplified DRC). Данная модификация будет иметь смысл в случае большого числа компонент связности остаточного графа степени 2. В данном случае модифицированный алгоритм предположительно будет работать быстрее. Его недостатком является то, что он может давать большее число стадий обмена, и, следовательно, менее оптимальное решение задачи.

Алгоритм sDRC.

1. Генерация списка сообщений.
2. step = 0.
3. Отсортировать список сообщений в порядке убывания веса ребра.
4. Повторять, пока deg(G)>0:
   - 4.1. Выбрать из списка первое слева сообщение (с максимальным весом) без конфликта, удалить его из графа и списка и включить в шаг с номером step.
   - 4.2. step=step+1.

Характеристики модифицированного алгоритма те же, что и у исходного алгоритма DRC.

## 3.3 Алгоритм GGP

Введем следующие определения и обозначения.

- $w_G(v) = \sum_{(v,w)\in E} w_G((v,w))$ – вес вершины v графа G (сумма весов всех инцидентных ей рёбер).
- Граф называется k-регулярным по весу, если $\forall v \in V$ w(v) = k.
- P(G) – суммарный вес графа G (сумма весов всех ребер графа G).
- W(G) – максимальный вес вершины в графе G.
- mw(v) = P(G) / k - w(v) – остаточный вес вершины v.

Далее приведен алгоритм GGP, как он представлен в работе [6]. Вход алгоритма: граф коммуникаций G. Выход алгоритма: множество паросочетаний S.

Алгоритма GGP

1. Построить граф $H = (A, B, E, w_H)$, где $\forall e \in E\ w_H(e) = \lceil w_G(e) \rceil$.
2. Построить граф $I = (V_{1I}, V_{2I}, E_I, w_I)$, где $P(I)/k \geq W(I)$ и $P(I)/k \in N$, k – параметр алгоритма (некоторое целое число), следующим образом:

   $\varphi = \max(W(H), \lceil P(H)/k \rceil)$.

   $\delta = \lceil (\varphi k - P(H))/W(H) \rceil$.

   $V_{1I}=\{v_1',..., v_{n1+\delta}'\}$, $V_{2I}=\{u_1',..., u_{n2+\delta}'\}$ – вершины нового графа I соответствуют вершинам графа H.

   $E_I = E_1 \cup E_2$, где

   $E_1=\{(v_i', u_j')| (v_i, u_j) \in E , i\in[1, n_1]. j\in[1, n_2]\}\ \forall (v_i', u_j') \in E_1\ w_I((v_i', u_j'))= w_H(v_i, u_j)$

   Если $\delta=0$:

   $E_2=\varnothing$

   иначе

   $E_2=\{(v_{n1+i}',u_{n2+i}')| i\in[1, \delta]\}$

   Если $\delta\neq1$:

   $\forall\ i\in[1, \delta-1], w_I((v_{n1+i}',u_{n2+i}'))=W(H)$.

   Если $(\varphi k- P(H)) \bmod W(H) \neq 0$:

   $w_I((v_{n1+\delta}',u_{n2+\delta}'))=(\varphi k- P(H)) \bmod W(H)$

   иначе

   $w_I((v_{n1+\delta}',u_{n2+\delta}'))=W(H)$
3. Преобразовать I в P(I)/k-регулярный по весу граф $J=(V_{1J}, V_{2J}, E_J, w_J)$ (процедура преобразования представлена ниже).
4. $S=\varnothing$.
5. Повторять, пока $E_J\neq\varnothing$:

   5.1. Выбрать совершенное паросочетание M в J.

   5.2. Изменить $w_M$ так, чтобы $\forall e\in M\ w_M(e)=s(M)$.

   5.3. Добавить M к S, к множеству паросочетаний.

   5.4. $\forall e\in M$ заменить $w_j(e)$ на $w_j(e)-w_M(e)$.

   5.5. Удалить из $E_J$ рёбра с весом 0.
6. Удалить рёбра в S, которые не принадлежат E.

Процедура преобразования графа I в граф J.

1. Скопировать граф I в J.
2. Ввести переменную cn – номер вершины. Первоначально значение cn неопределено.
3. Повторять для всех $s \in V_{1J}$:
   3.1. Вычислить mw(s).
   3.2. Если cn неопределено или mw(cn)=0:
      3.2.1. Добавить узел n к множеству $V_{2J}$.
      3.2.2. cn=n.
      3.2.3. Добавить ребро (s, cn) к $E_J$ с $w_J$(s, cn)=nw(s).
      иначе
      3.2.4. Если mw(cn)≥mw(s):
         3.2.4.1. Добавить ребро (s, cn) к $E_J$ с $w_J$(s, cn)=mw(s).
         иначе
         3.2.4.2. Добавить ребро (s, cn) к $E_J$ с $w_J$(s, cn)=mw(cn).
         3.2.4.3. Добавить узел n к множеству $V_{2J}$.
         3.2.4.4. cn=n.
         3.2.4.5. Добавить ребро (s, cn) к $E_J$ с $w_J$(s, cn)=mw(s).
4. Выполнить действия из пункта 3 для узлов из $V_{2J}$.

Для поиска совершенного паросочетания в пункте 5.1 используется венгерский метод [9], представленный ниже. Введем несколько определений:

- Ненасыщенная вершина – вершина, не покрытая паросочетанием.
- Чередующаяся цепь – цепь, в которой чередуются рёбра из паросочетания и рёбра не из паросочетания.

Алгоритм венгерского метода поиска совершенного паросочетания:

1. Перебираем рёбра в любом порядке. Каждое очередное ребро включается в паросочетание в том случае, если оно не смежно другим рёбрам паросочетания. В результате этого шага получили некоторое случайное паросочетание.
2. Во всем графе ищем ненасыщенные вершины. Если таких вершин нет, или такая вершина одна, то алгоритм закончен.
3. Ищем чередующуюся цепь в графе, соединяющую две ненасыщенные вершины. Если такой нет, то алгоритм закончен.
4. Исключаем из найденной цепи рёбра с текущим паросочетанием, а из паросочетания исключаем рёбра, входящие в цепь. Включаем в паросочетание оставшиеся в цепи рёбра. Возвращаемся к шагу 2.

Паросочетания, образующие множество S, являются шагами в расписании. Эти шаги можем выполнять в любом порядке. Сложность алгоритма GGP имеет порядок $O((N+|E|)^2\sqrt{N})$. Объем памяти, используемой алгоритмом, имеет порядок $O(N^3)$. Алгоритм имеет относительную гарантированную точность[1] 8/3.

[1] Относительная гарантированная точность алгоритма оптимизации – отношение решения алгоритма к оптимальному решению.

# 4 Сравнительное тестирование алгоритмов

Для оценки рассмотренных алгоритмов было выполнено их сравнительное тестирование. При тестировании варьировались такие параметры, как число процессоров, число пересылок, объем пересылаемой информации, способ выполнения обменов. Сравнивалось время работы алгоритма и время выполнения обменов по расписаниям, полученным в результате работы алгоритма. Все алгоритмы также сравнивались с вариантом, когда планирование не выполняется.

Все тесты были проведены на кластере МВС-100К (г. Москва). Основные характеристики кластера представлены в таблице 1.

Таблица 1. Характеристики кластера МВС-100К

| Количество ядер | 8576 |
|---|---|
| Тип процессоров | Intel(R) Xeon(R) CPU E5450 3GHz,<br>Intel(R) Xeon(R) CPU 5365 3GHz |
| Число ядер в узле | 8 |
| Объем оперативной памяти узла, Гб | 8 |
| Коммуникационная сеть | InfiniBand 4x DDR |

Будем рассматривать три варианта выполнения обменов: «No Barrier», «Barrier Send», «Barrier Send/Recv»:

- В варианте «No Barrier» одновременно происходит асинхронный запуск всех команд приёма и отправки, потом ожидание завершения всех команд (рис. 5а).
- В варианте «Barrier Send» одновременно запускаются все команды асинхронного приёма, затем на каждой стадии выполняются соответствующие ей команды асинхронной отправки и ожидание их завершения. После выполнения всех стадий происходит ожидание всех приёмов (рис. 5б).
- В варианте «Barrier Send/Recv» на каждой стадии выполняются соответствующие ей команды асинхронных приёма и отправки и ожидание завершения этих команд (рис. 5в).

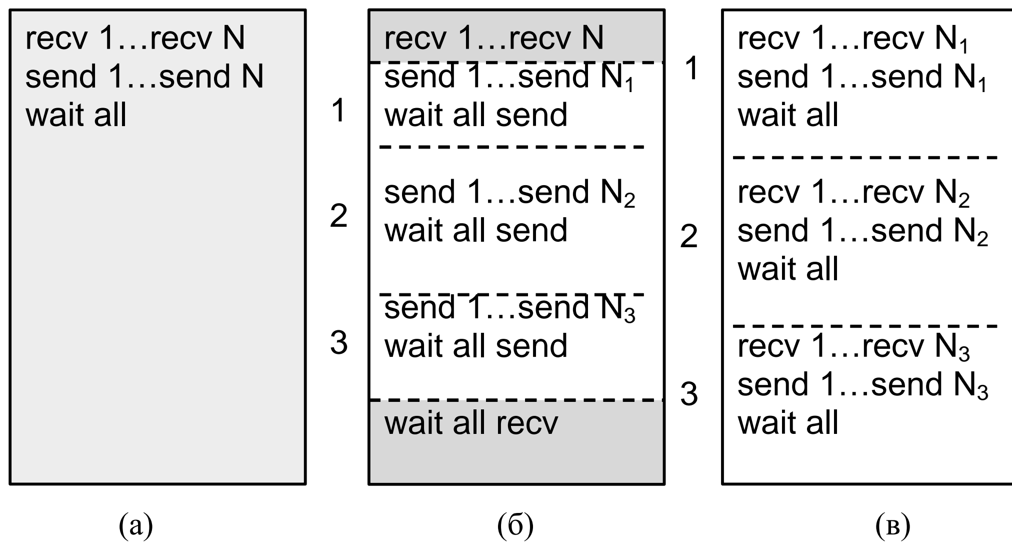


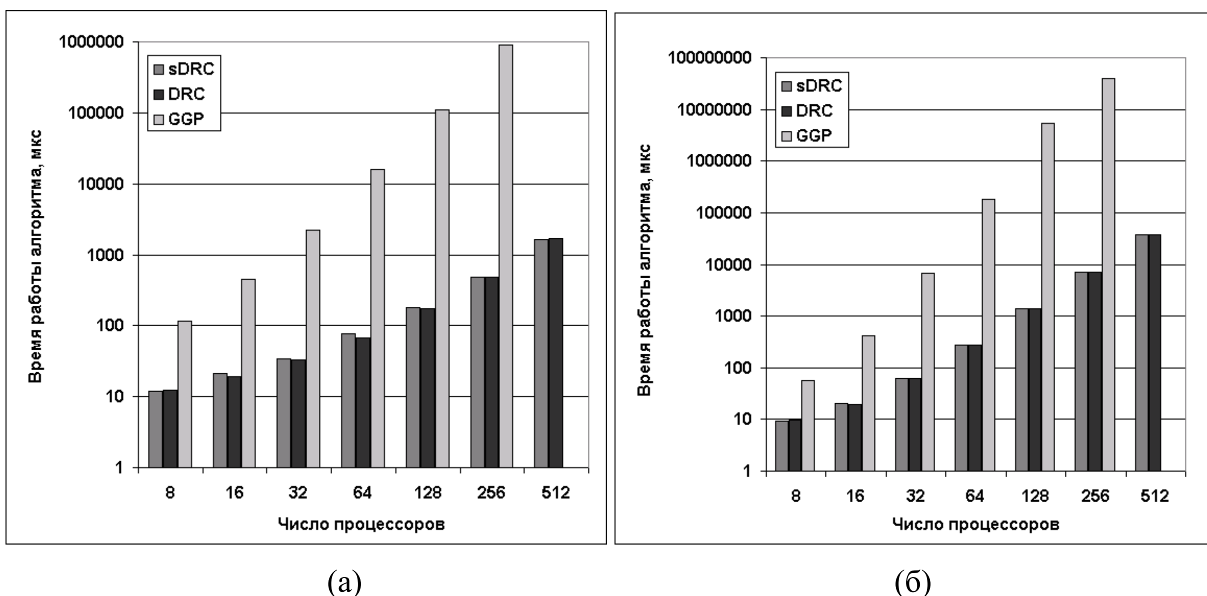

Рис. 5. Варианты выполнения обменов: а) «No Barrier», б) «Barrier Send», в) «Barrier Send/Recv».

Тестирование проводилось следующим образом. Случайным образом генерировалось большое число (101) графов коммуникаций с одинаковыми параметрами: число вершин, число рёбер, суммарный объём передаваемых данных. Для каждого графа коммуникаций выполнялось построение расписаний различными алгоритмами. Для каждого расписания производились обмены тремя различными вариантами (рис. 5). В качестве результата для данного алгоритма, данного способа выполнения обменов и данного набора параметров графа коммуникаций бралось время, осредненное по всем таким графам.

## 4.1 Сравнение времен работы алгоритмов

На рис. 6 представлено время работы алгоритмов для различного числа процессоров и различного числа пересылок.

Результаты показывают, что алгоритм GGP тратит существенно больше времени на построение расписания обменов. Это соответствует теоретическим оценкам, представленным в главе 4. Алгоритмы DRC и sDRC показали примерно одинаковое время построения расписания.



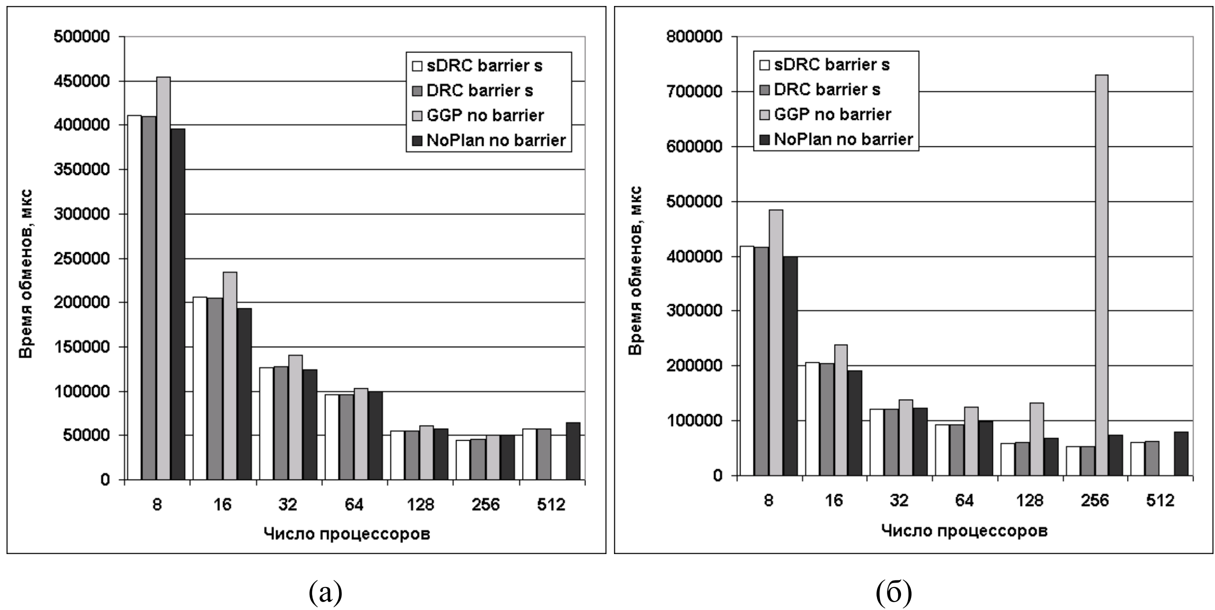

(а) (б)

Рис. 6. Время построения расписания различными алгоритмами для различного числа процессоров N и числа пересылок |E|: а) |E| = 4N, б) $|E| = N^2/4$.

## 4.2 Сравнение времен выполнения обменов

На рис. 7 и 8 представлено время выполнения обменов при построении расписания различными алгоритмами и без планирования для различного числа процессоров и различного числа пересылок. Для каждого алгоритма были исследованы различные способы выполнения обменов. На рисунках представлены только лучшие результаты для каждого алгоритма. На рис. 7 представлены результаты для суммарного объёма пересылаемых данных 512 MB. На рис. 8 представлены результаты для суммарного объёма пересылаемых данных 1 GB.

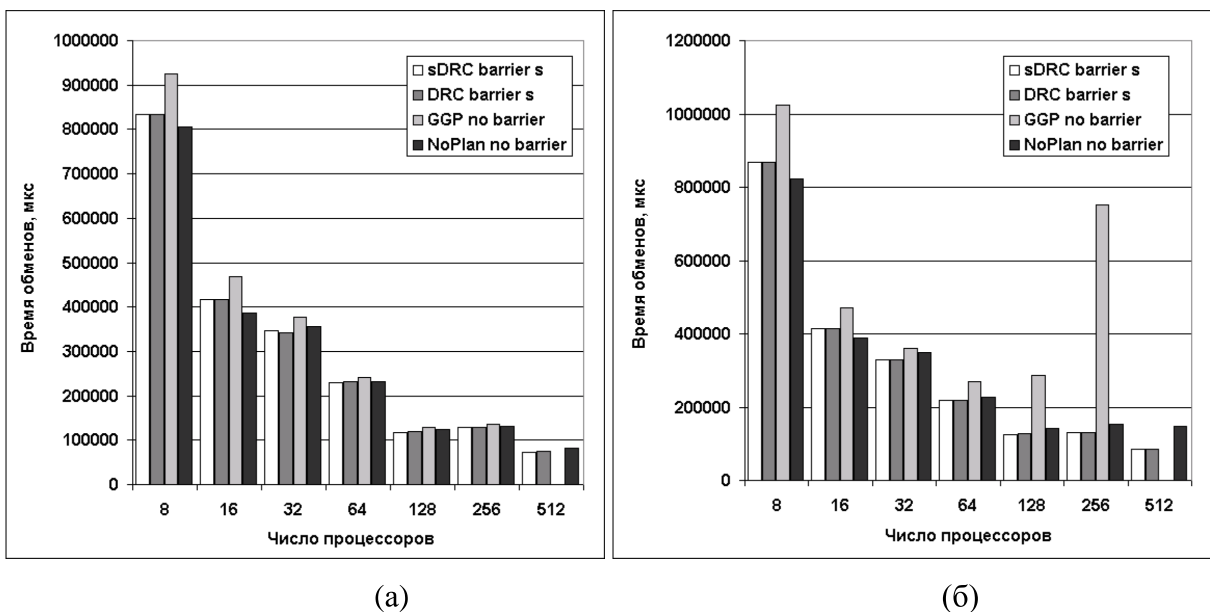
При увеличении числа процессоров увеличивается число сообщений, при этом размер сообщений в среднем пропорционально уменьшается. По результатам тестирования время выполнения обменов падает с ростом числа процессоров до 256, после чего время начинает расти из-за накладных расходов на передачу. Это значит, что при использовании до 256 процессоров решающую роль играет объем передаваемых сообщений, а при превышении числа в 256 процессоров – число передаваемых сообщений.




(а) (б)

Рис. 7. Время выполнения обменов с помощью различных алгоритмов для различного числа процессоров N и числа пересылок |E|: а) |E| = 4N, б) $|E| = N^2/4$. Суммарный объём данных 512 Мб.

Алгоритм GGP показывает самое большое время обменов, т.к. число сообщений, генерируемых этим алгоритмом, значительно увеличивается из-за их разбиения на части. Действительно, при большом числе пересылок (рис. 7б) начиная уже с 256 процессоров время обменов с помощью алгоритма GGP резко возрастает. Данное явление объясняется тем, что мы не учитываем время инициализации передач.



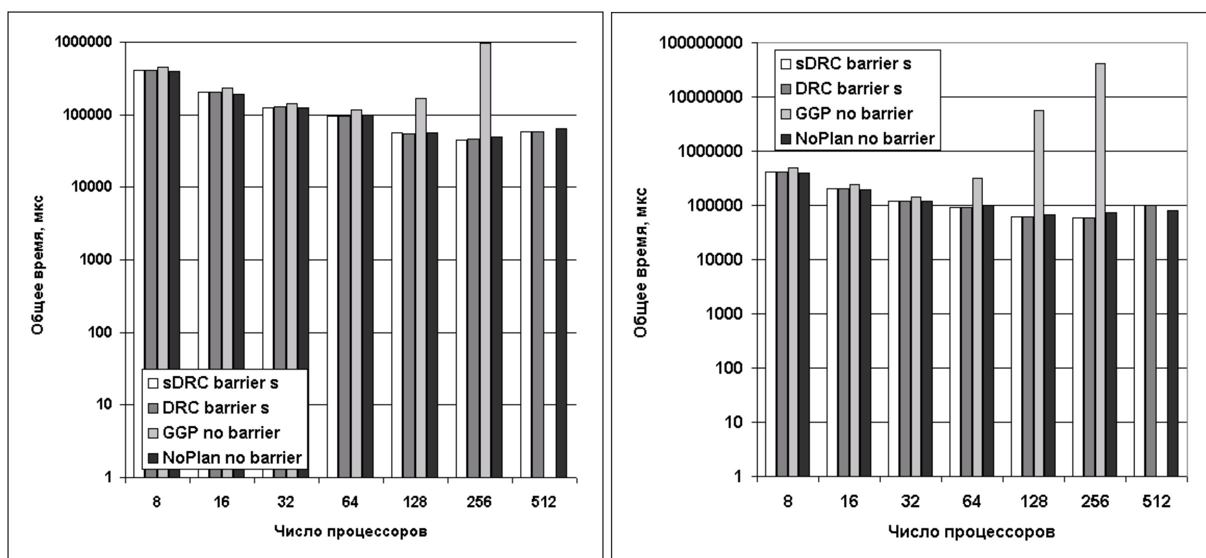

(а) (б)

Рис. 8. Время выполнения обменов с помощью различных алгоритмов для различного числа процессоров N и числа пересылок |E|: а) |E| = 4N, б) $|E| = N^2/4$. Суммарный объём данных 1 Гб.

Как при объёме данных 512 MB, так и при 1 GB, для малого числа процессоров (до 64) вариант обменов без планирования работает быстрее. При большом числе процессоров алгоритмы DRC и sDRC дают лучшие результаты. Причем, отличия между ними незначительны. Результаты для объёмов данных 512 Мб и 1 Гб во многом похожи, за исключением масштаба времени (времена отличаются почти в 2 раза).

## 4.3 Сравнение суммарных временных затрат на организацию обменов

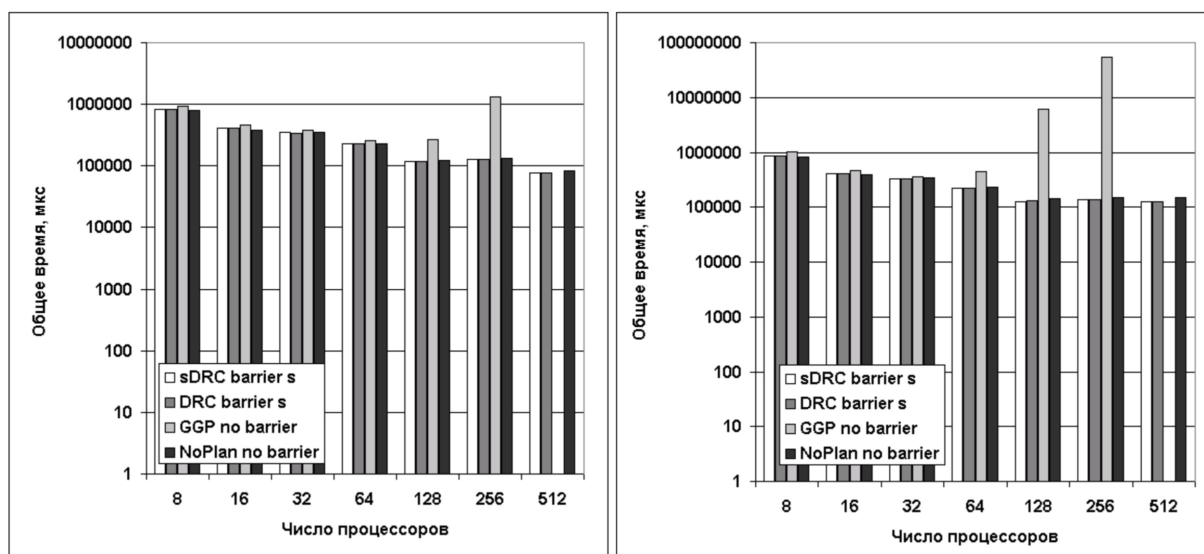
На рис. 9 и 10 представлено общее время на организацию обменов, которое складывается из времени построения расписания различными алгоритмами и времени обменов для различного числа процессоров и различного числа пересылок.




(а) (б)

Рис. 9. Общее время на организацию обменов с помощью различных алгоритмов для различного числа процессоров N и числа пересылок |E|: а) |E| = 4N, б) $|E| = N^2/4$.

Суммарный объём данных 512 Мб.



(а) (б)

Рис. 10. Общее время на организацию обменов с помощью различных алгоритмов для различного числа процессоров N и числа пересылок |E|: а) $|E| = 4N$, б) $|E| = N^2/4$. Суммарный объём данных 1 Гб.

По результатам тестирования можно сделать следующие выводы:

- На малом числе процессоров лучше не использовать планирование и не выделять стадии обменов, а использовать вариант обменов «No Barrier». Таким образом передача данных реализована, например, в системе KeLP [7].
- Про большом числе процессоров в большинстве случаев лучше всего работают алгоритмы sDRC и DRC с разделением всех операций отправки по стадиям (вариант «Barrier Send»).
- Алгоритм GGP в рассмотренных ограничениях показал плохие результаты. Для их улучшения требуется учитывать время инициализации передачи данных. Кроме того, сам алгоритм обладает плохой масштабируемостью.

## Заключение

В работе была рассмотрена проблема поиска оптимального алгоритма построения расписания нерегулярной перефрагментации данных. Сформулирован ряд требований к алгоритму перефрагментации. Выполнен обзор работ, посвященных проблеме перефрагментации. Выбраны алгоритмы, удовлетворяющие сформулированным требованиям. Реализованы следующие алгоритмы построения расписания обменов: DRC, GGP и модификация алгоритма DRC. Проведено сравнительное тестирование реализованных алгоритмов. По результатам тестирования сделаны выводы.

В дальнейшем работа может быть продолжена в магистратуре по следующим направлениям:

- Анализ и тестирование алгоритмов с учетом затрат на инициализацию передач данных.
- Использование реализованных алгоритмов в реальных задачах численного моделирования.

## Литература

1. Christian Perez. Load balancing HPF programs by migrating virtual processors. Ecole Normale Supérieure de Lyon. October 1996.
2. High Performance Fortran Forum, http://hpff.rice.edu/
3. Hui Wang, Minyi Guo, Daming Wei. Message Scheduling for Irregular Data Redistribution in Parallelizing Compilers. IEICE TRANS. INF. & SYST., VOL E89-D, NO. 2 FEBRUARY 2006, p. 418-424.
4. Hui Wang, Minyi Guo, Sushil K. Prasad, Yi Pan, Wenxi Chen. An Efficient Algorithm for Irregular Redistributions in Parallelizing Compilers. ISPA 2003, LNCS 2745, p. 76–87.
5. Kun-Ming Yu, Yi-Lin Tsai. An Efficient Scheduling Algorithm for Irregular Data Redistribution. IAENG International Journal of Applied Mathematics, 36:1, IJAM_36_1_6.
6. Johanne Cohen, Emmanuel Jeannot, Nicolas Padoy, Frederic Wagner. Messages Scheduling for Parallel Data Redistribution between Clusters. IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, VOL. 17, NO. 10, OCTOBER 2006, p. 1163-1174.
7. Stephen. J. Fink, Scott B. Badden. Runtime Support for Multi-Tier Programming of Block-Structured Applications on SMP-Clusters. ISCOPE, December 1997.
8. Wenheng Liu, Cho-Li Wang, Viktor K. Prasanna. Portable and Scalable Algorithm for Irregular All-to-All Communication. Journal of Parallel and Distributed Computing 62, 2002, p. 1493–1526.
9. Емеличев В. А. Лекции по теории графов. / В. А. Емеличев, О. И. Мельников, В. И. Сарванов, Р. И. Тышевич. – М., Наука, 1990. – 384 с.

# Приложение А. Листинги программ тестирования алгоритмов.

(обязательное)

Листинг А.1. Программа для сравнительного тестирования алгоритмов нерегулярной перефрагментации данных.

```
1	#include <stdio.h>
2	#include <stdlib.h>
3	#include <string.h>
4	#include <sys/time.h>
5	#include <time.h>
6	#include <limits.h>
7	#include <mpi.h>
8	#ifndef M
9	#define M 16
10	#endif
11	#define K M
12	struct A
13	{
14	   unsigned int v1;
15	   unsigned int v2;
16	   unsigned int e;
17	   unsigned int tmp;
18	};
19	unsigned int G[M][M];
20	unsigned int nsteps[M*M];
21	unsigned int vsteps1[M*M][M];
22	unsigned int vsteps2[M*M][M];
23	unsigned int wsteps[M*M];
24	struct timeval tv1,tv2,dtv;
25	struct timezone tz;
26	void time_start() { gettimeofday(&tv1, &tz); }
27	long time_stop()
28	{
29	   gettimeofday(&tv2, &tz);
30	   dtv.tv_sec= tv2.tv_sec -tv1.tv_sec;
31	   dtv.tv_usec=tv2.tv_usec-tv1.tv_usec;
32	   if(dtv.tv_usec<0) { dtv.tv_sec--; dtv.tv_usec+=1000000; }
33	   return dtv.tv_sec*1000000+dtv.tv_usec;
34	}
35	void build(int G[], int nv, int ne, int W)
36	{
37	   int i, k, s=0, tmp[ne];
38	   float r;
39	   for(k=0;k<ne;k++)
40	   {
41	         srand(time(NULL));
42	         i=rand()%(nv*nv);
43	         while(G[i]!=0)
44	         i=rand()%(nv*nv);
45	         tmp[k]=i;
```

```
46              G[i]=rand()%W+1;
47              s+=G[i];
48          }
49          for(i=0;i<nv*nv;i++)
50                  if(G[i]!=0)
51                  {
52                  r=(float)G[i]*W/s;
53                  if((int)r!=0)
54                          r=1.f;
55                  G[i]=r;
56                  }
57          s=0;
58          for(i=0;i<nv;i++)
59          s+=G[i];
60          if(W-s>0)
61          for(k=0;k<W-s;k++)
62          {
63              i=rand()%ne;
64                  G[tmp[i]]++;
65          }
66          else
67          for(k=0;k>W-s;k--)
68          {
69              i=rand()%ne;
70                  while(G[tmp[i]]<=1)
71                      i=rand()%ne;
72                  G[tmp[i]]--;
73          }
74      }
75      int match(int v, int* stack, int *mt, int J[], int nv, int nv2)
76      {
77          register int to;
78          if(stack[v]) return 0;
79          stack[v]=1;
80          to=0;
81          while(to<nv2)
82          {
83                  for(to;J[v*4*nv+to]==0 && to<nv2;to++);
84                  if(J[v*4*nv+to]!=0)
85                  if(mt[to]==-1 || match(mt[to], stack, mt, J, nv, nv2))
86                  {
87                      mt[to]=v;
88                      return 1;
89                  }
90                  to++;
91          }
92          return 0;
93      }
94      long exchange2(int G[], int rank, int nv, int ne, int nsteps[], int vsteps1[], int vsteps2[],
95      int wsteps[])
96      {
97          long t1;
```

```
98          int i, j, so[nv], ro[nv], ir, is;
99          char *sdata[nv], *ddata[nv];
100         MPI_Request ss[nv*nv], rs[nv*nv];
101         memset(so, 0, nv*sizeof(int));
102         memset(ro, 0, nv*sizeof(int));
103         MPI_Bcast(G, nv*nv, MPI_INT, 0, MPI_COMM_WORLD);
104         MPI_Bcast(vsteps1, nv*nv*nv, MPI_INT, 0, MPI_COMM_WORLD);
105         MPI_Bcast(vsteps2, nv*nv*nv, MPI_INT, 0, MPI_COMM_WORLD);
106         MPI_Bcast(nsteps, nv*nv, MPI_INT, 0, MPI_COMM_WORLD);
107         MPI_Bcast(wsteps, nv*nv, MPI_INT, 0, MPI_COMM_WORLD);
108         ir=0;
109         is=0;
110         for(i=0;i<nv;i++)
111         {
112               if(G[rank*nv+i]!=0)
113               {
114                  sdata[i]=malloc(G[rank*nv+i]);
115                  srand(time(NULL));
116                  memset(sdata[i],(char)rand(),G[rank*nv+i]);
117               }
118         }
119         for(i=0;i<nv;i++)
120         {
121               if(G[i*nv+rank]!=0)
122                  ddata[i]=malloc(G[i*nv+rank]);
123         }
124         i=0;
125         time_start();
126         while(nsteps[i]!=0)
127         {
128               for(j=0;j<nsteps[i];j++)
129               {
130                  if(vsteps2[i*nv+j]==rank)
131                  {
132                        MPI_Irecv(ddata[vsteps1[i*nv+j]]+ro[vsteps1[i*nv+j]], wsteps[i],
133      MPI_CHAR, vsteps1[i*nv+j], 40, MPI_COMM_WORLD, &rs[ir]);
134                        ir++;
135                        ro[vsteps1[i*nv+j]]+=wsteps[i];
136                  }
137               }
138               i++;
139         }
140         i=0;
141         while(nsteps[i]!=0)
142         {
143               for(j=0;j<nsteps[i];j++)
144               {
145                  if(vsteps1[i*nv+j]==rank)
146                  {
147                        MPI_Isend(sdata[vsteps2[i*nv+j]]+so[vsteps2[i*nv+j]], wsteps[i],
148      MPI_CHAR, vsteps2[i*nv+j], 40, MPI_COMM_WORLD, &ss[is]);
149                        is++;
```

```
150                     so[vsteps2[i*nv+j]]+=wsteps[i];
151                 }
152             }
153             i++;
154         }
155         MPI_Waitall(ir, rs, MPI_STATUSES_IGNORE);
156         MPI_Waitall(is, ss, MPI_STATUSES_IGNORE);
157         t1=time_stop();
158         for(i=0;i<nv;i++)
159         {
160             if(G[rank*nv+i]!=0)
161                 free(sdata[i]);
162             if(G[i*nv+rank]!=0)
163                 free(ddata[i]);
164         }
165         return t1;
166     }
167     long exchange2sr(int G[], int rank, int nv, int ne, int nsteps[], int vsteps1[], int vsteps2[],
168     int wsteps[])
169     {
170         long t1;
171         int i, j, so[nv], ro[nv], ir, is;
172         char *sdata[nv], *ddata[nv];
173         MPI_Request ss[nv*nv], rs[nv*nv];
174         memset(so, 0, nv*sizeof(int));
175         memset(ro, 0, nv*sizeof(int));
176         MPI_Bcast(G, nv*nv, MPI_INT, 0, MPI_COMM_WORLD);
177         MPI_Bcast(vsteps1, nv*nv*nv, MPI_INT, 0, MPI_COMM_WORLD);
178         MPI_Bcast(vsteps2, nv*nv*nv, MPI_INT, 0, MPI_COMM_WORLD);
179         MPI_Bcast(nsteps, nv*nv, MPI_INT, 0, MPI_COMM_WORLD);
180         MPI_Bcast(wsteps, nv*nv, MPI_INT, 0, MPI_COMM_WORLD);
181         for(i=0;i<nv;i++)
182         {
183             if(G[rank*nv+i]!=0)
184             {
185                 sdata[i]=malloc(G[rank*nv+i]);
186                 srand(time(NULL));
187                 memset(sdata[i],(char)rand(),G[rank*nv+i]);
188             }
189         }
190         for(i=0;i<nv;i++)
191         {
192             if(G[i*nv+rank]!=0)
193                 ddata[i]=malloc(G[i*nv+rank]);
194         }
195         i=0;
196         time_start();
197         while(nsteps[i]!=0)
198         {
199             ir=0;
200             is=0;
201             for(j=0;j<nsteps[i];j++)
```

```
202             {
203                if(vsteps2[i*nv+j]==rank)
204                {
205                      MPI_Irecv(ddata[vsteps1[i*nv+j]]+ro[vsteps1[i*nv+j]], wsteps[i],
206   MPI_CHAR, vsteps1[i*nv+j], 40, MPI_COMM_WORLD, &rs[ir]);
207                      ir++;
208                      ro[vsteps1[i*nv+j]]+=wsteps[i];
209                }
210                if(vsteps1[i*nv+j]==rank)
211                {
212                      MPI_Isend(sdata[vsteps2[i*nv+j]]+so[vsteps2[i*nv+j]], wsteps[i],
213   MPI_CHAR, vsteps2[i*nv+j], 40, MPI_COMM_WORLD, &ss[is]);
214                      is++;
215                      so[vsteps2[i*nv+j]]+=wsteps[i];
216                }
217             }
218             MPI_Waitall(ir, rs, MPI_STATUSES_IGNORE);
219             MPI_Waitall(is, ss, MPI_STATUSES_IGNORE);
220             i++;
221       }
222       t1=time_stop();
223       for(i=0;i<nv;i++)
224       {
225             if(G[rank*nv+i]!=0)
226                free(sdata[i]);
227             if(G[i*nv+rank]!=0)
228                free(ddata[i]);
229       }
230       return t1;
231   }
232   long exchange2s(int G[], int rank, int nv, int ne, int nsteps[], int vsteps1[], int vsteps2[],
233   int wsteps[])
234   {
235       long t1;
236       int i, j, so[nv], ro[nv], ir, is;
237       char *sdata[nv], *ddata[nv];
238       MPI_Status st;
239       MPI_Request ss[nv*nv], rs[nv*nv];
240       memset(so, 0, nv*sizeof(int));
241       memset(ro, 0, nv*sizeof(int));
242       MPI_Bcast(G, nv*nv, MPI_INT, 0, MPI_COMM_WORLD);
243       MPI_Bcast(vsteps1, nv*nv*nv, MPI_INT, 0, MPI_COMM_WORLD);
244       MPI_Bcast(vsteps2, nv*nv*nv, MPI_INT, 0, MPI_COMM_WORLD);
245       MPI_Bcast(nsteps, nv*nv, MPI_INT, 0, MPI_COMM_WORLD);
246       MPI_Bcast(wsteps, nv*nv, MPI_INT, 0, MPI_COMM_WORLD);
247       for(i=0;i<nv;i++)
248       {
249             if(G[rank*nv+i]!=0)
250             {
251                sdata[i]=malloc(G[rank*nv+i]);
252                srand(time(NULL));
253                memset(sdata[i],(char)rand(),G[rank*nv+i]);
```

```
254                 }
255           }
256           for(i=0;i<nv;i++)
257           {
258                 if(G[i*nv+rank]!=0)
259                    ddata[i]=malloc(G[i*nv+rank]);
260           }
261           i=0;
262           ir=0;
263           time_start();
264           while(nsteps[i]!=0)
265           {
266                 for(j=0;j<nsteps[i];j++)
267                 {
268                    if(vsteps2[i*nv+j]==rank)
269                    {
270                          MPI_Irecv(ddata[vsteps1[i*nv+j]]+ro[vsteps1[i*nv+j]], wsteps[i],
271        MPI_CHAR, vsteps1[i*nv+j], 40, MPI_COMM_WORLD, &rs[ir]);
272                          ir++;
273                          ro[vsteps1[i*nv+j]]+=wsteps[i];
274                    }
275                 }
276                 i++;
277           }
278           i=0;
279           while(nsteps[i]!=0)
280           {
281                 is=0;
282                 for(j=0;j<nsteps[i];j++)
283                 {
284                    if(vsteps1[i*nv+j]==rank)
285                    {
286                          MPI_Isend(sdata[vsteps2[i*nv+j]]+so[vsteps2[i*nv+j]], wsteps[i],
287        MPI_CHAR, vsteps2[i*nv+j], 40, MPI_COMM_WORLD, &ss[is]);
288                          is++;
289                          so[vsteps2[i*nv+j]]+=wsteps[i];
290                    }
291                    MPI_Waitall(is, ss, MPI_STATUSES_IGNORE);
292                 }
293                 i++;
294           }
295           MPI_Waitall(ir, rs, MPI_STATUSES_IGNORE);
296           MPI_Barrier(MPI_COMM_WORLD);
297           t1=time_stop();
298           for(i=0;i<nv;i++)
299           {
300                 if(G[rank*nv+i]!=0)
301                    free(sdata[i]);
302                 if(G[i*nv+rank]!=0)
303                    free(ddata[i]);
304           }
305           return t1;
```

```
306     }
307     void schedule3(int G[], int nv, int ne, int nsteps[], int vsteps1[], int vsteps2[], int
308     wsteps[])
309     {
310        int i, j, k, m, cn, P, W, I[2*nv][2*nv], J[4*nv][4*nv], fi, delta, nv1, nv2, q, w1, w2,
311     used1[4*nv], used2[4*nv], stack[8*nv], max, deg, step, mt[4*nv];
312        memset(I, 0, 4*nv*nv*sizeof(int));
313        memset(J, 0, 16*nv*nv*sizeof(int));
314        memset(stack, 0, 8*nv*sizeof(int));
315        time_start();
316        P=0;
317        W=0;
318        for(i=0;i<nv;i++)
319              for(j=0;j<nv;j++)
320                 P=P+G[i*nv+j];
321        for(i=0;i<nv;i++)
322        {
323              w1=0;
324              for(j=0;j<nv;j++)
325                 w1+=G[i*nv+j];
326              W=((W>w1)?W:w1);
327        }
328        for(i=0;i<nv;i++)
329        {
330              w2=0;
331              for(j=0;j<nv;j++)
332                 w2+=G[j*nv+i];
333              W=((W>w2)?W:w2);
334        }
335        fi=((W>(P/K+1))?W:(P/K+1));
336        delta=(fi*K-P)/W+1;
337        for(i=0;i<nv;i++)
338        {
339              for(j=0;j<nv;j++)
340              {
341                 I[i][j]=G[i*nv+j];
342              }
343        }
344        if(delta!=1)
345        {
346              for(i=0;i<delta-1;i++)
347              {
348                 I[nv+i][nv+i]=W;
349              }
350        }
351        if((fi*K-P)%W!=0)
352              I[nv+delta-1][nv+delta-1]=(fi*K-P)%W;
353        else
354              I[nv+delta-1][nv+delta-1]=W;
355        nv1=nv2=nv+delta;
356        P=0;
357        for(i=0;i<nv+delta;i++)
```

```
358         {
359               for(j=0;j<nv+delta;j++)
360               {
361                  P=P+I[i][j];
362               }
363         }
364         for(i=0;i<nv+delta;i++)
365         {
366               for(j=0;j<nv+delta;j++)
367               {
368                  J[i][j]=I[i][j];
369               }
370         }
371         cn=-1;
372         for(i=0;i<nv+delta;i++)
373         {
374                  w2=0;
375                  for(j=0;j<nv2;j++)
376                        w2=w2+J[i][j];
377               if(cn!=-1)
378               {
379                  w1=0;
380                  for(j=0;j<nv1;j++)
381                        w1=w1+J[j][cn-1];
382               }
383               if(cn==-1 || P/K-w1==0)
384               {
385                  cn=++nv2;
386                  J[i][cn-1]=P/K-w2;
387               }
388               else
389               {
390                  if(w1>=w2)
391                        J[i][cn-1]=P/K-w2;
392                  else
393                  {
394                  J[i][cn-1]=P/K-w1;
395                  cn=++nv2;
396                  J[i][cn-1]=P/K-w2;
397                  }
398               }
399         }
400         for(i=0;i<nv+delta;i++)
401         {
402                  w2=0;
403                  for(j=0;j<nv1;j++)
404                        w2=w2+J[j][i];
405               if(cn!=-1)
406               {
407                  w1=0;
408                  for(j=0;j<nv1;j++)
409                        w1=w1+J[cn-1][j];
```

```
410             }
411             if(cn==-1 || P/K-w1==0)
412             {
413                cn=++nv1;
414                J[cn-1][i]=P/K-w2;
415             }
416             else
417             {
418                if(w1>=w2)
419                      J[cn-1][i]=w2;
420                else
421                {
422                J[cn-1][i]=w1;
423                cn=++nv1;
424                J[cn-1][i]=P/K-w2;
425                }
426             }
427       }
428       max=0;
429       for(i=0;i<nv1;i++)
430       {
431             deg=0;
432             for(j=0;j<nv2;j++)
433             {
434                if(J[i][j]!=0)
435                      deg++;
436             }
437             if(max<deg)
438             {
439                max=deg;
440             }
441       }
442       step=0;
443       while(max>0)
444       {
445             memset(used1, 0, nv1*sizeof(int));
446             memset(used2, 0, nv2*sizeof(int));
447             nsteps[step]=0;
448             memset(mt, -1, 4*nv*sizeof(int));
449             for(i=0;i<nv1;i++)
450             {
451             for(j=0;j<nv2;j++)
452             {
453                if(J[i][j]!=0)
454                {
455                if(!used1[i] && !used2[j])
456                {
457                mt[j]=i;
458                used1[i]=1;
459                used2[j]=1;
460                }
461                }
```

```
462             }
463             }
464             for(j=0;j<nv1;j++)
465             {
466             if(used1[j])
467                continue;
468             memset(stack, 0, 4*nv*sizeof(int));
469             match(j, stack, mt, J, nv, nv2);
470             }
471             w1=INT_MAX;
472             for(i=0;i<4*nv;i++)
473             {
474             if(mt[i]!=-1)
475                w1=((w1<J[mt[i]][i])?w1:J[mt[i]][i]);
476             }
477             wsteps[step]=w1;
478             for(i=0;i<4*nv;i++)
479             if(mt[i]!=-1)
480                J[mt[i]][i]-=w1;
481             max=0;
482             for(i=0;i<nv;i++)
483             {
484                if(mt[i]!=-1 && mt[i]<nv)
485                {
486                      vsteps1[step*nv+nsteps[step]]=mt[i];
487                      vsteps2[step*nv+nsteps[step]]=i;
488                      nsteps[step]++;
489                }
490             }
491             for(i=0;i<nv1;i++)
492          {
493                deg=0;
494                for(j=0;j<nv2;j++)
495                {
496                      if(J[i][j]!=0)
497                         deg++;
498                }
499                if(max<deg)
500                {
501                      max=deg;
502                }
503             }
504             if(nsteps[step]!=0) step++;
505       }
506       printf("%ld\t", time_stop());
507    }
508    long exchange3(int G[], int rank, int nv, int ne, int nsteps[], int vsteps1[], int vsteps2[])
509    {
510       long t1;
511       int i, j, ir, is;
512       MPI_Request ss[ne], rs[ne];
513       char *sdata[nv], *ddata[nv];
```

```
514         MPI_Bcast(G, nv*nv, MPI_INT, 0, MPI_COMM_WORLD);
515         ir=0;
516         is=0;
517         for(i=0;i<nv;i++)
518         {
519               if(G[rank*nv+i]!=0)
520               {
521                  sdata[i]=malloc(G[rank*nv+i]);
522                  srand(time(NULL));
523                  memset(sdata[i],(char)rand(),G[rank*nv+i]);
524               }
525         }
526         for(i=0;i<nv;i++)
527         {
528               if(G[i*nv+rank]!=0)
529                  ddata[i]=malloc(G[i*nv+rank]);
530         }
531         i=0;
532         MPI_Barrier(MPI_COMM_WORLD);
533         time_start();
534         for(i=0;i<nv;i++)
535         {
536               if(G[i*nv+rank]!=0)
537               {
538                  MPI_Irecv(ddata[i], G[i*nv+rank], MPI_CHAR, i, 40,
539      MPI_COMM_WORLD, &rs[ir]);
540                  ir++;
541               }
542         }
543         for(i=0;i<nv;i++)
544         {
545               if(G[rank*nv+i]!=0)
546               {
547                  MPI_Isend(sdata[i], G[rank*nv+i], MPI_CHAR, i, 40,
548      MPI_COMM_WORLD, &ss[is]);
549                  is++;
550               }
551         }
552         MPI_Waitall(ir, rs, MPI_STATUSES_IGNORE);
553         MPI_Waitall(is, ss, MPI_STATUSES_IGNORE);
554         MPI_Barrier(MPI_COMM_WORLD);
555         t1=time_stop();
556         for(i=0;i<nv;i++)
557         {
558               if(G[rank*nv+i]!=0)
559                  free(sdata[i]);
560               if(G[i*nv+rank]!=0)
561                  free(ddata[i]);
562         }
563         return t1;
564      }
565      long exchange3s(int G[], int rank, int nv, int ne, int nsteps[], int vsteps1[], int vsteps2[])
```

```
566     {
567         long t1;
568         int i, j, ir;
569         MPI_Request ss[ne], rs[ne];
570         char *sdata[nv], *ddata[nv];
571         MPI_Bcast(G, nv*nv, MPI_INT, 0, MPI_COMM_WORLD);
572         ir=0;
573         for(i=0;i<nv;i++)
574         {
575                 if(G[rank*nv+i]!=0)
576                 {
577                     sdata[i]=malloc(G[rank*nv+i]);
578                     srand(time(NULL));
579                     memset(sdata[i],(char)rand(), G[rank*nv+i]);
580                 }
581         }
582         for(i=0;i<nv;i++)
583         {
584                 if(G[i*nv+rank]!=0)
585                     ddata[i]=malloc(G[i*nv+rank]);
586         }
587         i=0;
588         MPI_Barrier(MPI_COMM_WORLD);
589         time_start();
590         for(i=0;i<nv;i++)
591         {
592                 if(G[i*nv+rank]!=0)
593                 {
594                     MPI_Irecv(ddata[i], G[i*nv+rank], MPI_CHAR, i, 40,
595     MPI_COMM_WORLD, &rs[ir]);
596                     ir++;
597                 }
598         }
599         for(i=0;i<nv;i++)
600         {
601                 if(G[rank*nv+i]!=0)
602                 {
603                     MPI_Isend(sdata[i], G[rank*nv+i], MPI_CHAR, i, 40,
604     MPI_COMM_WORLD, &ss[i]);
605                     MPI_Wait(&ss[i], MPI_STATUSES_IGNORE);
606                 }
607         }
608         MPI_Waitall(ir, rs, MPI_STATUSES_IGNORE);
609         MPI_Barrier(MPI_COMM_WORLD);
610         t1=time_stop();
611         for(i=0;i<nv;i++)
612         {
613                 if(G[rank*nv+i]!=0)
614                     free(sdata[i]);
615                 if(G[i*nv+rank]!=0)
616                     free(ddata[i]);
617         }
```

```
618        return t1;
619     }
620     long exchange3sr(int G[], int rank, int nv, int ne, int nsteps[], int vsteps1[], int vsteps2[])
621     {
622        long t1;
623        int i, j;
624        MPI_Status st;
625        MPI_Request ss[ne], rs[ne];
626        char *sdata[nv], *ddata[nv];
627        MPI_Bcast(G, nv*nv, MPI_INT, 0, MPI_COMM_WORLD);
628        for(i=0;i<nv;i++)
629        {
630              if(G[rank*nv+i]!=0)
631              {
632                 sdata[i]=malloc(G[rank*nv+i]);
633                 srand(time(NULL));
634                 memset(sdata[i], (char)rand(), G[rank*nv+i]);
635              }
636        }
637        for(i=0;i<nv;i++)
638        {
639              if(G[i*nv+rank]!=0)
640                 ddata[i]=malloc(G[i*nv+rank]);
641        }
642        i=0;
643        MPI_Barrier(MPI_COMM_WORLD);
644        time_start();
645        for(i=0;i<nv;i++)
646        {
647              if(G[i*nv+rank]!=0)
648                 MPI_Irecv(ddata[i], G[i*nv+rank], MPI_CHAR, i, 40,
649     MPI_COMM_WORLD, &rs[i]);
650              if(G[rank*nv+i]!=0)
651                 MPI_Isend(sdata[i], G[rank*nv+i], MPI_CHAR, i, 40,
652     MPI_COMM_WORLD, &ss[i]);
653              if(G[i*nv+rank]!=0)
654                 MPI_Wait(&rs[i], &st);
655              if(G[rank*nv+i]!=0)
656                 MPI_Wait(&ss[i], &st);
657        }
658        MPI_Barrier(MPI_COMM_WORLD);
659        t1=time_stop();
660        for(i=0;i<nv;i++)
661        {
662              if(G[rank*nv+i]!=0)
663                 free(sdata[i]);
664              if(G[i*nv+rank]!=0)
665                 free(ddata[i]);
666        }
667        return t1;
668     }
669     long exchange(int G[], int rank, int nv, int ne, int nsteps[], int vsteps1[], int vsteps2[])
```

```
670     {
671        long t1;
672        int i, j, ir, is;
673        MPI_Request ss[ne], rs[ne];
674        char *sdata[nv], *ddata[nv];
675        MPI_Bcast(G, nv*nv, MPI_INT, 0, MPI_COMM_WORLD);
676        MPI_Bcast(vsteps1, nv*nv, MPI_INT, 0, MPI_COMM_WORLD);
677        MPI_Bcast(vsteps2, nv*nv, MPI_INT, 0, MPI_COMM_WORLD);
678        MPI_Bcast(nsteps, nv, MPI_INT, 0, MPI_COMM_WORLD);
679        ir=0;
680        is=0;
681        for(i=0;i<nv;i++)
682        {
683              if(G[rank*nv+i]!=0)
684              {
685                 sdata[i]=malloc(G[rank*nv+i]);
686                 srand(time(NULL));
687                 memset(sdata[i],(char)rand(), G[rank*nv+i]);
688              }
689        }
690        for(i=0;i<nv;i++)
691        {
692              if(G[i*nv+rank]!=0)
693                 ddata[i]=malloc(G[i*nv+rank]);
694        }
695        i=0;
696        MPI_Barrier(MPI_COMM_WORLD);
697        time_start();
698        while(nsteps[i]!=0)
699        {
700              for(j=0;j<nsteps[i];j++)
701              {
702                 if(vsteps2[i*nv+j]==rank)
703                 {
704                       MPI_Irecv(ddata[vsteps1[i*nv+j]], G[vsteps1[i*nv+j]*nv+rank],
705     MPI_CHAR, vsteps1[i*nv+j], 40, MPI_COMM_WORLD, &rs[ir]);
706                       ir++;
707                 }
708              }
709              i++;
710        }
711        i=0;
712        while(nsteps[i]!=0)
713        {
714              for(j=0;j<nsteps[i];j++)
715              {
716                 if(vsteps1[i*nv+j]==rank)
717                 {
718                       MPI_Isend(sdata[vsteps2[i*nv+j]], G[rank*nv+vsteps2[i*nv+j]],
719     MPI_CHAR, vsteps2[i*nv+j], 40, MPI_COMM_WORLD, &ss[is]);
720                       is++;
721                 }
```

```
722                 }
723                 i++;
724             }
725             MPI_Waitall(ir, rs, MPI_STATUSES_IGNORE);
726             MPI_Waitall(is, ss, MPI_STATUSES_IGNORE);
727             MPI_Barrier(MPI_COMM_WORLD);
728             t1=time_stop();
729             for(i=0;i<nv;i++)
730             {
731                 if(G[rank*nv+i]!=0)
732                     free(sdata[i]);
733                 if(G[i*nv+rank]!=0)
734                     free(ddata[i]);
735             }
736             return t1;
737         }
738         long exchanges(int G[], int rank, int nv, int ne, int nsteps[], int vsteps1[], int vsteps2[])
739         {
740             long t1;
741             int i, j, ir, is;
742             MPI_Request ss[ne], rs[ne];
743             char *sdata[nv], *ddata[nv];
744             MPI_Bcast(G, nv*nv, MPI_INT, 0, MPI_COMM_WORLD);
745             MPI_Bcast(vsteps1, nv*nv, MPI_INT, 0, MPI_COMM_WORLD);
746             MPI_Bcast(vsteps2, nv*nv, MPI_INT, 0, MPI_COMM_WORLD);
747             MPI_Bcast(nsteps, nv, MPI_INT, 0, MPI_COMM_WORLD);
748             for(i=0;i<nv;i++)
749             {
750                 if(G[rank*nv+i]!=0)
751                 {
752                     sdata[i]=malloc(G[rank*nv+i]);
753                     srand(time(NULL));
754                     memset(sdata[i],(char)rand(), G[rank*nv+i]);
755                 }
756             }
757             for(i=0;i<nv;i++)
758             {
759                 if(G[i*nv+rank]!=0)
760                     ddata[i]=malloc(G[i*nv+rank]);
761             }
762             i=0;
763             MPI_Barrier(MPI_COMM_WORLD);
764             ir=0;
765             time_start();
766             while(nsteps[i]!=0)
767             {
768                 for(j=0;j<nsteps[i];j++)
769                 {
770                     if(vsteps2[i*nv+j]==rank)
771                     {
772                             MPI_Irecv(ddata[vsteps1[i*nv+j]], G[vsteps1[i*nv+j]*nv+rank],
773         MPI_CHAR, vsteps1[i*nv+j], 40, MPI_COMM_WORLD, &rs[ir]);
```

```
774                     ir++;
775                 }
776             }
777             i++;
778         }
779         i=0;
780         while(nsteps[i]!=0)
781         {
782             is=0;
783             for(j=0;j<nsteps[i];j++)
784             {
785                 if(vsteps1[i*nv+j]==rank)
786                 {
787                     MPI_Isend(sdata[vsteps2[i*nv+j]], G[rank*nv+vsteps2[i*nv+j]],
788     MPI_CHAR, vsteps2[i*nv+j], 40, MPI_COMM_WORLD, &ss[is]);
789                     is++;
790                 }
791             }
792             MPI_Waitall(is, ss, MPI_STATUSES_IGNORE);
793             i++;
794         }
795         MPI_Waitall(ir, rs, MPI_STATUSES_IGNORE);
796         t1=time_stop();
797         for(i=0;i<nv;i++)
798         {
799             if(G[rank*nv+i]!=0)
800                 free(sdata[i]);
801             if(G[i*nv+rank]!=0)
802                 free(ddata[i]);
803         }
804         return t1;
805     }
806     long exchangesr(int G[], int rank, int nv, int ne, int nsteps[], int vsteps1[], int vsteps2[])
807     {
808         long t1;
809         int i, j, ir, is;
810         MPI_Request ss[ne], rs[ne];
811         char *sdata[nv], *ddata[nv];
812         MPI_Bcast(G, nv*nv, MPI_INT, 0, MPI_COMM_WORLD);
813         MPI_Bcast(vsteps1, nv*nv, MPI_INT, 0, MPI_COMM_WORLD);
814         MPI_Bcast(vsteps2, nv*nv, MPI_INT, 0, MPI_COMM_WORLD);
815         MPI_Bcast(nsteps, nv, MPI_INT, 0, MPI_COMM_WORLD);
816         for(i=0;i<nv;i++)
817         {
818             if(G[rank*nv+i]!=0)
819             {
820                 sdata[i]=malloc(G[rank*nv+i]);
821                 srand(time(NULL));
822                 memset(sdata[i],(char)rand(), G[rank*nv+i]);
823             }
824         }
825         for(i=0;i<nv;i++)
```

```
826         {
827             if(G[i*nv+rank]!=0)
828                 ddata[i]=malloc(G[i*nv+rank]);
829         }
830         i=0;
831         MPI_Barrier(MPI_COMM_WORLD);
832         time_start();
833         while(nsteps[i]!=0)
834         {
835             ir=0;
836             is=0;
837             for(j=0;j<nsteps[i];j++)
838             {
839                 if(vsteps2[i*nv+j]==rank)
840                 {
841                     MPI_Irecv(ddata[vsteps1[i*nv+j]], G[vsteps1[i*nv+j]*nv+rank],
842     MPI_CHAR, vsteps1[i*nv+j], 40, MPI_COMM_WORLD, &rs[ir]);
843                     ir++;
844                 }
845                 if(vsteps1[i*nv+j]==rank)
846                 {
847                     MPI_Isend(sdata[vsteps2[i*nv+j]], G[rank*nv+vsteps2[i*nv+j]],
848     MPI_CHAR, vsteps2[i*nv+j], 40, MPI_COMM_WORLD, &ss[is]);
849                     is++;
850                 }
851             }
852             MPI_Waitall(ir, rs, MPI_STATUSES_IGNORE);
853             MPI_Waitall(is, ss, MPI_STATUSES_IGNORE);
854             i++;
855         }
856         MPI_Barrier(MPI_COMM_WORLD);
857         t1=time_stop();
858         for(i=0;i<nv;i++)
859         {
860             if(G[rank*nv+i]!=0)
861                 free(sdata[i]);
862             if(G[i*nv+rank]!=0)
863                 free(ddata[i]);
864         }
865         return t1;
866     }
867     int cmp(void *a, void *b)
868     {
869         return (*(struct A*)a).e<(*(struct A*)b).e;
870     }
871     void schedule1(int G[], int nv, int ne, int nsteps[], int vsteps1[], int vsteps2[])
872     {
873         int i, j, k, deg[2][nv], max, step, used1[nv], used2[nv], tmp1, tmp2;
874         struct A *sorted;
875         time_start();
876         step=0;
877         for(i=0;i<nv;i++)
```

```
878         {
879                deg[0][i]=0;
880                for(j=0;j<nv;j++)
881                {
882                   if(G[i*nv+j]!=0)
883                          deg[0][i]++;
884                }
885                if(step<deg[0][i])
886                {
887                   step=deg[0][i];
888                }
889         }
890         for(i=0;i<nv;i++)
891         {
892                deg[1][i]=0;
893                for(j=0;j<nv;j++)
894                {
895                   if(G[j*nv+i]!=0)
896                          deg[1][i]++;
897                }
898                if(step<deg[1][i])
899                {
900                   step=deg[1][i];
901                }
902         }
903         max=step;
904         sorted=malloc(ne*sizeof(struct A));
905         k=0;
906         for(i=0;i<nv;i++)
907         {
908                for(j=0;j<nv;j++)
909                {
910                if(G[i*nv+j]!=0)
911                {
912                   sorted[k].v1=i;
913                   sorted[k].v2=j;
914                   sorted[k].e=G[i*nv+j];
915                   sorted[k].tmp=0;
916                   k++;
917                }
918                }
919         }
920         qsort(sorted, ne, sizeof(struct A), cmp);
921         for(step=0;max>0;step++)
922         {
923                memset(used1, 0, nv*sizeof(int));
924                memset(used2, 0, nv*sizeof(int));
925                for(k=0;k<ne;k++)
926                {
927                   if(sorted[k].tmp==0)
928                   {
929                   tmp1=sorted[k].v1;
```

```
930                tmp2=sorted[k].v2;
931                if(!used1[tmp1] && !used2[tmp2])
932                {
933                vsteps1[step*nv+nsteps[step]]=tmp1;
934                vsteps2[step*nv+nsteps[step]]=tmp2;
935                nsteps[step]++;
936                used1[tmp1]=1;
937                used2[tmp2]=1;
938                sorted[k].tmp=1;
939                deg[0][tmp1]--;
940                deg[1][tmp2]--;
941                }
942                }
943            }
944            max=0;
945            for(i=0;i<nv;i++)
946            {
947                if(max<deg[0][i])
948                {
949                        max=deg[0][i];
950                }
951            }
952            for(i=0;i<nv;i++)
953            {
954                if(max<deg[1][i])
955                {
956                        max=deg[1][i];
957                }
958            }
959        }
960        printf("%ld\t", time_stop());
961        free(sorted);
962    }
963    void schedule2(int G[], int nv, int ne, int nsteps[], int vsteps1[], int vsteps2[])
964    {
965        int i, j, k, q, v, max, deg[2][nv], step, used1[nv], used2[nv], tmp1, tmp2;
966        struct A *sorted;
967        time_start();
968        step=0;
969        for(i=0;i<nv;i++)
970        {
971            deg[0][i]=0;
972            for(j=0;j<nv;j++)
973            {
974                if(G[i*nv+j]!=0)
975                        deg[0][i]++;
976            }
977            if(step<deg[0][i])
978            {
979                step=deg[0][i];
980            }
981        }
```

```
982         for(i=0;i<nv;i++)
983         {
984                 deg[1][i]=0;
985                 for(j=0;j<nv;j++)
986                 {
987                     if(G[j*nv+i]!=0)
988                             deg[1][i]++;
989                 }
990                 if(step<deg[1][i])
991                 {
992                     step=deg[1][i];
993                 }
994         }
995         max=step;
996         sorted=malloc(ne*sizeof(struct A));
997         k=0;
998         for(i=0;i<nv;i++)
999         {
1000                for(j=0;j<nv;j++)
1001                {
1002                    if(G[i*nv+j]!=0)
1003                    {
1004                    sorted[k].v1=i;
1005                    sorted[k].v2=j;
1006                    sorted[k].e=G[i*nv+j];
1007                    sorted[k].tmp=0;
1008                    k++;
1009                    }
1010                }
1011        }
1012        qsort(sorted, ne, sizeof(struct A), cmp);
1013        for(step=0;max>2;step++)
1014        {
1015                memset(used1, 0, nv*sizeof(int));
1016                memset(used2, 0, nv*sizeof(int));
1017                for(k=0;k<ne;k++)
1018                {
1019                    if(sorted[k].tmp==0)
1020                    {
1021                    tmp1=sorted[k].v1;
1022                    tmp2=sorted[k].v2;
1023                    if(!used1[tmp1] && !used2[tmp2])
1024                    {
1025                    vsteps1[step*nv+nsteps[step]]=tmp1;
1026                    vsteps2[step*nv+nsteps[step]]=tmp2;
1027                    nsteps[step]++;
1028                    used1[tmp1]=1;
1029                    used2[tmp2]=1;
1030                    deg[0][tmp1]--;
1031                    deg[1][tmp2]--;
1032                    sorted[k].tmp=1;
1033                    }
```

```
1034                }
1035            }
1036            max=0;
1037            for(i=0;i<nv;i++)
1038            {
1039                if(max<deg[0][i])
1040                {
1041                        max=deg[0][i];
1042                }
1043            }
1044            for(i=0;i<nv;i++)
1045            {
1046                if(max<deg[1][i])
1047                {
1048                        max=deg[1][i];
1049                }
1050            }
1051        }
1052        j=0;
1053        for(k=0;k<ne;k++)
1054        {
1055            if(!sorted[k].tmp)
1056            {
1057                sorted[j]=sorted[k];
1058                j++;
1059            }
1060        }
1061        while(max>0)
1062        {
1063            i=0;
1064            q=0;
1065            while(deg[0][i]!=1 && i!=nv)
1066                i++;
1067            if(i==nv)
1068            {
1069            i=0;
1070            q=1;
1071            while(deg[1][i]!=1 && i!=nv)
1072                i++;
1073            }
1074            if(i==nv)
1075            {
1076            i=0;
1077            q=0;
1078            while(deg[0][i]!=2)
1079                i++;
1080            }
1081            do
1082            {
1083                j=0;
1084                if(q==0)
1085                {
```

```
1086                  while(sorted[j].v1!=i || sorted[j].tmp)
1087                        j++;
1088                  }
1089                  else
1090                  {
1091                  while(sorted[j].v2!=i || sorted[j].tmp)
1092                        j++;
1093                  }
1094                  v=((q==0)?sorted[j].v2:sorted[j].v1);
1095                  vsteps1[(step+q)*nv+nsteps[step+q]]=((q==0)?i:v);
1096                  vsteps2[(step+q)*nv+nsteps[step+q]]=((q==0)?v:i);
1097                  nsteps[step+q]++;
1098                  if(q==0)
1099                  {
1100                        deg[0][i]--;
1101                        deg[1][v]--;
1102                  }
1103                  else
1104                  {
1105                        deg[0][v]--;
1106                        deg[1][i]--;
1107                  }
1108                  i=v;
1109                  sorted[j].tmp=1;
1110                  q=((q==0)?1:0);
1111               }
1112               while(deg[q][i]==1);
1113               max=0;
1114            for(i=0;i<nv;i++)
1115            {
1116                  if(max<deg[0][i])
1117                  {
1118                        max=deg[0][i];
1119                  }
1120               }
1121      }
1122      printf("%ld\t", time_stop());
1123      free(sorted);
1124   }
1125   int main(int argc, char **argv)
1126   {
1127      int i, j, size, rank, i1;
1128      long t1, t2;
1129      MPI_Init(&argc, &argv);
1130      MPI_Comm_size(MPI_COMM_WORLD, &size);
1131      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
1132      printf("%d\n", size);
1133      for(i1=0;i1<=100;i1++)
1134      {
1135      for(i=0;i<M*M;i++)
1136            nsteps[i]=0;
1137      if(rank==0)
```

```
1138	{
1139	memset(G, 0, M*M*sizeof(int));
1140	build(G, M, 4*M, 536870912);
1141	schedule1(G, M, 4*M, nsteps, vsteps1, vsteps2);
1142	i=0;
1143	}
1144	for(i=0;i<M;i++)
1145	{
1146		MPI_Irecv(&j, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &rs[i]);
1147		MPI_Isend(&i, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &ss[i]);
1148		MPI_Wait(&rs[i], &st);
1149		MPI_Wait(&ss[i], &st);
1150	}
1151	MPI_Barrier(MPI_COMM_WORLD);
1152	t1=exchange(G, rank, M, 4*M, nsteps, vsteps1, vsteps2);
1153	MPI_Allreduce(&t1, &t2, 1, MPI_LONG, MPI_MAX, MPI_COMM_WORLD);
1154	if(rank==0) printf("%ld\t", t2);
1155	for(i=0;i<M;i++)
1156	{
1157		MPI_Irecv(&j, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &rs[i]);
1158		MPI_Isend(&i, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &ss[i]);
1159		MPI_Wait(&rs[i], &st);
1160		MPI_Wait(&ss[i], &st);
1161	}
1162	MPI_Barrier(MPI_COMM_WORLD);
1163	t1=exchanges(G, rank, M, 4*M, nsteps, vsteps1, vsteps2);
1164	MPI_Allreduce(&t1, &t2, 1, MPI_LONG, MPI_MAX, MPI_COMM_WORLD);
1165	if(rank==0) printf("%ld\t", t2);
1166	for(i=0;i<M;i++)
1167	{
1168		MPI_Irecv(&j, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &rs[i]);
1169		MPI_Isend(&i, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &ss[i]);
1170		MPI_Wait(&rs[i], &st);
1171		MPI_Wait(&ss[i], &st);
1172	}
1173	MPI_Barrier(MPI_COMM_WORLD);
1174	t1=exchangesr(G, rank, M, 4*M, nsteps, vsteps1, vsteps2);
1175	MPI_Allreduce(&t1, &t2, 1, MPI_LONG, MPI_MAX, MPI_COMM_WORLD);
1176	if(rank==0) printf("%ld\t", t2);
1177	MPI_Barrier(MPI_COMM_WORLD);
1178	if(rank==0)
1179	{
1180	for(i=0;i<M*M;i++)
1181	     nsteps[i]=0;
1182	schedule2(G, M, 4*M, nsteps, vsteps1, vsteps2);
1183	}
1184	for(i=0;i<M;i++)
1185	{
1186		MPI_Irecv(&j, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &rs[i]);
1187		MPI_Isend(&i, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &ss[i]);
1188		MPI_Wait(&rs[i], &st);
1189		MPI_Wait(&ss[i], &st);
```

```
1190        }
1191        MPI_Barrier(MPI_COMM_WORLD);
1192        t1=exchange(G, rank, M, 4*M, nsteps, vsteps1, vsteps2);
1193        MPI_Allreduce(&t1, &t2, 1, MPI_LONG, MPI_MAX, MPI_COMM_WORLD);
1194        if(rank==0) printf("%ld\t", t2);
1195        for(i=0;i<M;i++)
1196        {
1197                MPI_Irecv(&j, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &rs[i]);
1198                MPI_Isend(&i, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &ss[i]);
1199                MPI_Wait(&rs[i], &st);
1200                MPI_Wait(&ss[i], &st);
1201        }
1202        MPI_Barrier(MPI_COMM_WORLD);
1203        t1=exchanges(G, rank, M, 4*M, nsteps, vsteps1, vsteps2);
1204        MPI_Allreduce(&t1, &t2, 1, MPI_LONG, MPI_MAX, MPI_COMM_WORLD);
1205        if(rank==0) printf("%ld\t", t2);
1206        for(i=0;i<M;i++)
1207        {
1208                MPI_Irecv(&j, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &rs[i]);
1209                MPI_Isend(&i, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &ss[i]);
1210                MPI_Wait(&rs[i], &st);
1211                MPI_Wait(&ss[i], &st);
1212        }
1213        MPI_Barrier(MPI_COMM_WORLD);
1214        t1=exchangesr(G, rank, M, 4*M, nsteps, vsteps1, vsteps2);
1215        MPI_Allreduce(&t1, &t2, 1, MPI_LONG, MPI_MAX, MPI_COMM_WORLD);
1216        if(rank==0) printf("%ld\t", t2);
1217        if(M<=256)
1218        {
1219        if(rank==0)
1220        {
1221        for(i=0;i<M*M;i++)
1222             nsteps[i]=0;
1223        schedule3(G, M, 4*M, nsteps, vsteps1, vsteps2, wsteps);
1224        }
1225        for(i=0;i<M;i++)
1226        {
1227                MPI_Irecv(&j, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &rs[i]);
1228                MPI_Isend(&i, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &ss[i]);
1229                MPI_Wait(&rs[i], &st);
1230                MPI_Wait(&ss[i], &st);
1231        }
1232        MPI_Barrier(MPI_COMM_WORLD);
1233        t1=exchange2(G, rank, M, 4*M, nsteps, vsteps1, vsteps2, wsteps);
1234        MPI_Allreduce(&t1, &t2, 1, MPI_LONG, MPI_MAX, MPI_COMM_WORLD);
1235        if(rank==0) printf("%ld\t", t2);
1236        for(i=0;i<M;i++)
1237        {
1238                MPI_Irecv(&j, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &rs[i]);
1239                MPI_Isend(&i, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &ss[i]);
1240                MPI_Wait(&rs[i], &st);
1241                MPI_Wait(&ss[i], &st);
```

```
1242        }
1243        MPI_Barrier(MPI_COMM_WORLD);
1244        t1=exchange2s(G, rank, M, 4*M, nsteps, vsteps1, vsteps2, wsteps);
1245        MPI_Allreduce(&t1, &t2, 1, MPI_LONG, MPI_MAX, MPI_COMM_WORLD);
1246        if(rank==0) printf("%ld\t", t2);
1247        for(i=0;i<M;i++)
1248        {
1249                MPI_Irecv(&j, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &rs[i]);
1250                MPI_Isend(&i, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &ss[i]);
1251                MPI_Wait(&rs[i], &st);
1252                MPI_Wait(&ss[i], &st);
1253        }
1254        MPI_Barrier(MPI_COMM_WORLD);
1255        t1=exchange2sr(G, rank, M, 4*M, nsteps, vsteps1, vsteps2, wsteps);
1256        MPI_Allreduce(&t1, &t2, 1, MPI_LONG, MPI_MAX, MPI_COMM_WORLD);
1257        if(rank==0) printf("%ld\t", t2);
1258        }
1259        for(i=0;i<M;i++)
1260        {
1261                MPI_Irecv(&j, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &rs[i]);
1262                MPI_Isend(&i, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &ss[i]);
1263                MPI_Wait(&rs[i], &st);
1264                MPI_Wait(&ss[i], &st);
1265        }
1266        MPI_Barrier(MPI_COMM_WORLD);
1267        t1=exchange3(G, rank, M, 4*M, nsteps, vsteps1, vsteps2);
1268        MPI_Allreduce(&t1, &t2, 1, MPI_LONG, MPI_MAX, MPI_COMM_WORLD);
1269        if(rank==0) printf("%ld\t", t2);
1270        for(i=0;i<M;i++)
1271        {
1272                MPI_Irecv(&j, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &rs[i]);
1273                MPI_Isend(&i, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &ss[i]);
1274                MPI_Wait(&rs[i], &st);
1275                MPI_Wait(&ss[i], &st);
1276        }
1277        MPI_Barrier(MPI_COMM_WORLD);
1278        t1=exchange3s(G, rank, M, 4*M, nsteps, vsteps1, vsteps2);
1279        MPI_Allreduce(&t1, &t2, 1, MPI_LONG, MPI_MAX, MPI_COMM_WORLD);
1280        if(rank==0) printf("%ld\t", t2);
1281        for(i=0;i<M;i++)
1282        {
1283                MPI_Irecv(&j, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &rs[i]);
1284                MPI_Isend(&i, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &ss[i]);
1285                MPI_Wait(&rs[i], &st);
1286                MPI_Wait(&ss[i], &st);
1287        }
1288        MPI_Barrier(MPI_COMM_WORLD);
1289        t1=exchange3sr(G, rank, M, 4*M, nsteps, vsteps1, vsteps2);
1290        MPI_Allreduce(&t1, &t2, 1, MPI_LONG, MPI_MAX, MPI_COMM_WORLD);
1291        if(rank==0) printf("%ld\t", t2);
1292        if(rank==0) printf("\t");
1293        if(rank==0)
```

```
1294        {
1295        for(i=0;i<M*M;i++)
1296              nsteps[i]=0;
1297        memset(G, 0, M*M*sizeof(int));
1298        build(G, M, M*M/4, 536870912);
1299        schedule1(G, M, M*M/4, nsteps, vsteps1, vsteps2);
1300        }
1301        for(i=0;i<M;i++)
1302        {
1303                MPI_Irecv(&j, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &rs[i]);
1304                MPI_Isend(&i, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &ss[i]);
1305                MPI_Wait(&rs[i], &st);
1306                MPI_Wait(&ss[i], &st);
1307        }
1308        MPI_Barrier(MPI_COMM_WORLD);
1309        t1=exchange(G, rank, M, M*M/4, nsteps, vsteps1, vsteps2);
1310        MPI_Allreduce(&t1, &t2, 1, MPI_LONG, MPI_MAX, MPI_COMM_WORLD);
1311        if(rank==0) printf("%ld\t", t2);
1312        for(i=0;i<M;i++)
1313        {
1314                MPI_Irecv(&j, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &rs[i]);
1315                MPI_Isend(&i, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &ss[i]);
1316                MPI_Wait(&rs[i], &st);
1317                MPI_Wait(&ss[i], &st);
1318        }
1319        MPI_Barrier(MPI_COMM_WORLD);
1320        t1=exchanges(G, rank, M, M*M/4, nsteps, vsteps1, vsteps2);
1321        MPI_Allreduce(&t1, &t2, 1, MPI_LONG, MPI_MAX, MPI_COMM_WORLD);
1322        if(rank==0) printf("%ld\t", t2);
1323        for(i=0;i<M;i++)
1324        {
1325                MPI_Irecv(&j, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &rs[i]);
1326                MPI_Isend(&i, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &ss[i]);
1327                MPI_Wait(&rs[i], &st);
1328                MPI_Wait(&ss[i], &st);
1329        }
1330        MPI_Barrier(MPI_COMM_WORLD);
1331        t1=exchangesr(G, rank, M, M*M/4, nsteps, vsteps1, vsteps2);
1332        MPI_Allreduce(&t1, &t2, 1, MPI_LONG, MPI_MAX, MPI_COMM_WORLD);
1333        if(rank==0) printf("%ld\t", t2);
1334        MPI_Barrier(MPI_COMM_WORLD);
1335        if(rank==0)
1336        {
1337        for(i=0;i<M*M;i++)
1338              nsteps[i]=0;
1339        schedule2(G, M, M*M/4, nsteps, vsteps1, vsteps2);
1340        }
1341        for(i=0;i<M;i++)
1342        {
1343                MPI_Irecv(&j, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &rs[i]);
1344                MPI_Isend(&i, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &ss[i]);
1345                MPI_Wait(&rs[i], &st);
```

```
1346                MPI_Wait(&ss[i], &st);
1347        }
1348        MPI_Barrier(MPI_COMM_WORLD);
1349        t1=exchange(G, rank, M, M*M/4, nsteps, vsteps1, vsteps2);
1350        MPI_Allreduce(&t1, &t2, 1, MPI_LONG, MPI_MAX, MPI_COMM_WORLD);
1351        if(rank==0) printf("%ld\t", t2);
1352        for(i=0;i<M;i++)
1353        {
1354                MPI_Irecv(&j, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &rs[i]);
1355                MPI_Isend(&i, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &ss[i]);
1356                MPI_Wait(&rs[i], &st);
1357                MPI_Wait(&ss[i], &st);
1358        }
1359        MPI_Barrier(MPI_COMM_WORLD);
1360        t1=exchanges(G, rank, M, M*M/4, nsteps, vsteps1, vsteps2);
1361        MPI_Allreduce(&t1, &t2, 1, MPI_LONG, MPI_MAX, MPI_COMM_WORLD);
1362        if(rank==0) printf("%ld\t", t2);
1363        for(i=0;i<M;i++)
1364        {
1365                MPI_Irecv(&j, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &rs[i]);
1366                MPI_Isend(&i, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &ss[i]);
1367                MPI_Wait(&rs[i], &st);
1368                MPI_Wait(&ss[i], &st);
1369        }
1370        MPI_Barrier(MPI_COMM_WORLD);
1371        t1=exchangesr(G, rank, M, M*M/4, nsteps, vsteps1, vsteps2);
1372        MPI_Allreduce(&t1, &t2, 1, MPI_LONG, MPI_MAX, MPI_COMM_WORLD);
1373        if(rank==0)printf("%ld\t", t2);
1374        for(i=0;i<M;i++)
1375        {
1376                MPI_Irecv(&j, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &rs[i]);
1377                MPI_Isend(&i, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &ss[i]);
1378                MPI_Wait(&rs[i], &st);
1379                MPI_Wait(&ss[i], &st);
1380        }
1381        MPI_Barrier(MPI_COMM_WORLD);
1382        if(M<=256)
1383        {
1384        if(rank==0)
1385        {
1386        for(i=0;i<M*M;i++)
1387            nsteps[i]=0;
1388        schedule3(G, M, M*M/4, nsteps, vsteps1, vsteps2, wsteps);
1389        }
1390        for(i=0;i<M;i++)
1391        {
1392                MPI_Irecv(&j, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &rs[i]);
1393                MPI_Isend(&i, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &ss[i]);
1394                MPI_Wait(&rs[i], &st);
1395                MPI_Wait(&ss[i], &st);
1396        }
1397        MPI_Barrier(MPI_COMM_WORLD);
```

```
1398        t1=exchange2(G, rank, M, M*M/4, nsteps, vsteps1, vsteps2, wsteps);
1399        MPI_Allreduce(&t1, &t2, 1, MPI_LONG, MPI_MAX, MPI_COMM_WORLD);
1400        if(rank==0) printf("%ld\t", t2);
1401        for(i=0;i<M;i++)
1402        {
1403                MPI_Irecv(&j, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &rs[i]);
1404                MPI_Isend(&i, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &ss[i]);
1405                MPI_Wait(&rs[i], &st);
1406                MPI_Wait(&ss[i], &st);
1407        }
1408        MPI_Barrier(MPI_COMM_WORLD);
1409        t1=exchange2s(G, rank, M, M*M/4, nsteps, vsteps1, vsteps2, wsteps);
1410        MPI_Allreduce(&t1, &t2, 1, MPI_LONG, MPI_MAX, MPI_COMM_WORLD);
1411        if(rank==0) printf("%ld\t", t2);
1412        for(i=0;i<M;i++)
1413        {
1414                MPI_Irecv(&j, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &rs[i]);
1415                MPI_Isend(&i, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &ss[i]);
1416                MPI_Wait(&rs[i], &st);
1417                MPI_Wait(&ss[i], &st);
1418        }
1419        MPI_Barrier(MPI_COMM_WORLD);
1420        t1=exchange2sr(G, rank, M, M*M/4, nsteps, vsteps1, vsteps2, wsteps);
1421        MPI_Allreduce(&t1, &t2, 1, MPI_LONG, MPI_MAX, MPI_COMM_WORLD);
1422        if(rank==0)printf("%ld\t", t2);
1423        }
1424        for(i=0;i<M;i++)
1425        {
1426                MPI_Irecv(&j, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &rs[i]);
1427                MPI_Isend(&i, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &ss[i]);
1428                MPI_Wait(&rs[i], &st);
1429                MPI_Wait(&ss[i], &st);
1430        }
1431        MPI_Barrier(MPI_COMM_WORLD);
1432        t1=exchange3(G, rank, M, M*M/4, nsteps, vsteps1, vsteps2);
1433        MPI_Allreduce(&t1, &t2, 1, MPI_LONG, MPI_MAX, MPI_COMM_WORLD);
1434        if(rank==0)printf("%ld\t", t2);
1435        for(i=0;i<M;i++)
1436        {
1437                MPI_Irecv(&j, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &rs[i]);
1438                MPI_Isend(&i, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &ss[i]);
1439                MPI_Wait(&rs[i], &st);
1440                MPI_Wait(&ss[i], &st);
1441        }
1442        MPI_Barrier(MPI_COMM_WORLD);
1443        t1=exchange3s(G, rank, M, M*M/4, nsteps, vsteps1, vsteps2);
1444        MPI_Allreduce(&t1, &t2, 1, MPI_LONG, MPI_MAX, MPI_COMM_WORLD);
1445        if(rank==0)printf("%ld\t", t2);
1446        for(i=0;i<M;i++)
1447        {
1448                MPI_Irecv(&j, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &rs[i]);
1449                MPI_Isend(&i, 1, MPI_INT, i, 40, MPI_COMM_WORLD, &ss[i]);
```

```
1450                MPI_Wait(&rs[i], &st);
1451                MPI_Wait(&ss[i], &st);
1452        }
1453        MPI_Barrier(MPI_COMM_WORLD);
1454        t1=exchange3sr(G, rank, M, M*M/4, nsteps, vsteps1, vsteps2);
1455        MPI_Allreduce(&t1, &t2, 1, MPI_LONG, MPI_MAX, MPI_COMM_WORLD);
1456        if(rank==0)printf("%ld\t", t2);
1457        if(rank==0)printf("\n");
1458        }
1459        MPI_Finalize();
1460        return 0;
1461    }
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
```

**Комментарии к листингу**

Описание рабочих данных и параметров программы:

- В строках 8-10 определяется число процессоров M.
- В строке 11 определяется параметр K – ограничение на пропускную способность канала для алгоритма GGP. В настоящей работе его значение было взято максимальным.
- В строках 12-18 определяется структура A, содержащая информацию о ребре, входящем в расписание. Она включает поля: вес ребра (e), начало ребра (v1), конец ребра (v2), флаг использования ребра (tmp).
- В строках 19-23 объявляются: G – матрица графа коммуникаций, vsteps1 –массив вершин-источников в расписании обменов (первое измерение - номер стадии обмена, второе - номер позиции в данной стадии), vsteps2 – массив вершин-приемников в расписании обменов, nsteps – массив количеств рёбер по стадиям, wsteps – массив весов стадий для алгоритма GGP.

Описание функций

- Функция build (строки 35-74) случайным образом генерирует граф коммуникаций G.
- Функция match (строки 75-93) ищет максимальное паросочетание в графе.
- Функции exchange2 (строки 94-166), exchange2sr (строки 167-231), exchange2s (строки 232-306) проводят обмены для алгоритма GGP по вариантам NoBarrier, Barrier Send/Recv, Barrier Send соответственно.
- Функция schedule3 (строки 307-507) строит расписание обменов по алгоритму GGP.
- Функции exchange3 (строки 508-564), exchange3s (строки 565-619), exchange3sr (строки 620-668) проводят обмены без планирования по вариантам NoBarrier, Barrier Send, Barrier Send/Recv соответственно.
- Функции exchange(строки 669-737), exchanges (строки 738-805), exchangesr (строки 806-866) проводят обмены для алгоритмов sDRC и DRC по вариантам NoBarrier, Barrier Send, Barrier Send/Recv соответственно.
- Функция cmp (строки 867-870) задаёт правило сортировки для элементов структуры A.
- Функции schedule1 и schedule2 строят расписание по алгоритмам sDRC и DRC соответственно.
- Функция main (сама тестирующая программа) состоит из цикла от 0 до 100. На каждой итерации цикла генерируются графы с различным числом рёбер. Для каждого числа рёбер запускаются функции построения расписании. Для каждого алгоритма запускаются соответствующие функции обменов. Перед каждым выполнением обменов предварительно происходит "разогрев" линий связи – обмен всех процессоров со всеми.