Комплекс программ для точного и гарантированного приближенного решения задач линейного программирования в среде MPI

А.В. Панюков, В.В. Горбик

В работе рассмотрены проблематика получения гарантированного решения задач линейного программирования, методы увеличения точности вычислений. Для решения задач используются библиотеки дробно-рациональных вычислений и интервальные типы данных с плавающей точкой фиксированной длины. Приводятся методы их адаптации к MPI и демонстрируется эффективность их применения. В работе представлены результаты вычислительных экспериментов на основе реализаций параллельных версий алгоритмов решения задач систем линейных уравнений и задач линейного программирования.

1. Введение

В настоящее время широко распространены не основанные на доказательствах предрассудки, порождающие ошибки в расчетах: (1) распространение свойства ассоциативности операций сложения и умножения в поле действительных чисел на конечное множество машинных "действительных" чисел; (2) распространение свойства непрерывной зависимости от параметров решения системы, полученной после «эквивалентных» преобразований, на исходную систему. Вычисления, приписывающие несуществующие свойства объектам численного анализа, являются бездоказательными. Отмеченный недостаток имеют популярные коммерческие пакеты MatLab, MathCad и т.п., а также свободно распространяемый пакет SciLab. Использование в вычислениях разного числа процессоров во многих случаях дает существенно различающиеся результаты, демонстрируя необходимость доказательных вычислений.

Потенциал имеющихся пакетов, поддерживающих символические вычисления, не позволяет решать реальные проблемы математического и имитационного моделирования. Возможность обеспечения вычислений с произвольной точностью в программах пользователя дает библиотека GMP. Однако библиотека GMP не предоставляет своим объектам возможность их использования в параллельных вычислениях.

Ориентация на применение многопроцессорных вычислительных систем в составе персональных компьютеров или рабочих станций (параллельные вычисления) и на применение сетевых технологий (распределенные вычисления) требует разработки новых параллельных методов их решения. Они должны быть лишены недостатков «традиционных» методов таких, например, как последовательный характер вычислений. Анализ способов распараллеливания показывает эффективность распараллеливания «по информации». Поэтому, весьма перспективной становится SPMD-технология программирования (Single Program – Multiple Data). При этой технологии вычислительный процесс строится на основе единственной программы, запускаемой на всех процессорах вычислительной системы или на многих станциях локальной сети. Копии программы могут выполняться по разным ветвям алгоритма, обрабатывая подмножества данных. Неизбежна синхронизация во времени и при обработке общих данных. Данная идеология используется в стандарте MPI (Message Passing Interface) [1–2]. Такая технология параллельного программирования и обусловила разработку соответствующих методов. В то же время не отрицаются известные традиционные методы, сокращающие общее число операций и исключающие перебор. Иной методологический подход открывает дорогу к решению задач большой размерности и эффективной параллельной работе многих процессоров.

Целью исследования является развитие программного обеспечения безошибочных дробно-рациональных и гарантированных приближенных вычислений для параллельных и распределенных вычислительных систем. В работе рассматривается использование типов данных mpq_t и mpf_t из библиотеки GNU MP [3], а также построенного на основе GNU MP интервального типа mpfi_t из библиотеки MPFI [4]. Важным аспектом при этом является возможность и эф-

фективность адаптации данных типов к многопроцессорной среде. Интерфейс MPI уже на протяжении длительного времени является негласным стандартом при построении распределенных вычислительных систем. В данной работе рассмотрены способы интеграции типов точных и интервальных вычислений к MPI на основе сериализации объектов и переопределения размещения в памяти. Для целей безошибочных вычислений была выбрана библиотека GMP в виду того, что она является открытой разработкой, входит в дистрибутивы GNU/Linux и имеет хорошую производительность. Проект MPFI также является открытым и расширяет возможности GMP; добавляя интервальные вычисления на основе типов данных с плавающей точкой переменной длины.

2. Методы повышения точности вычислений

Пакет gmp содержит открытую библиотеку GNU MP для точных арифметических вычислений: операций над целыми числами со знаком, рациональными числами и числами с плавающей точкой. Библиотека GNU MP разработана для быстрой работы, как для больших, так и малых операндов. Она работает быстро, поскольку использует целые слова как базовый тип, применяет быстрые алгоритмы в зависимости от размера операндов, имеет оптимизированный ассемблерный код под многие типы процессоров и совмещает скорость с простотой и элегантностью выполнения операций.

2.1 Вычисления без округлений. Тип mpq_t

С помощью типа mpq_t реализованы точные вычисления с дробями. mpq_t построен на основе структур и функций C-библиотеки, числителем и знаменателем являются структуры mpz_struct, которые содержат:

- размер выделенной области памяти;
- размер занятой области памяти;
- указатель на массив представляющий число.

Фрагменты листинга с объявлением mpq_t представлены на рис. 1.

```
//  FILE: gmp.h
//Definition of mpq_t
#ifdef __GMP_SHORT_LIMB
  typedef unsigned int
mp_limb_t;
#else
#ifdef _LONG_LONG_LIMB
  typedef unsigned long long int
mp_limb_t;
#else
typedef unsigned long int
mp_limb_t;
#endif
#endif
```

```
typedef struct
{
  int _mp_alloc;
  int _mp_size;
  mp_limb_t *_mp_d;
} __mpz_struct;

typedef struct
{
  __mpz_struct _mp_num;
  __mpz_struct _mp_den;
} __mpq_struct;

typedef __mpq_struct mpq_t[1];
```

Рис. 1. Фрагмент листинга объявления типа mpq_t

Над типом mpq_t в библиотеке определено около 40 функций. Кроме того, можно применять любые функции над целыми числами для числителя и знаменателя отдельно.

2.1.1 Адаптация типа mpq_t к mpi

Эффективную передачу типа mpq_t в среде mpi можно осуществить с помощью неполной сериализации. Подробности реализации и оценка эффективности представлены в предыдущих работах [5–6].

2.2 Вычисления с заданной точностью. Интервальные вычисления

В случае, когда задачи требуют таких объемов расчетов, при которых не предоставляется возможным проводить вычисления точно, а погрешности решения задач стандартными аппаратными типами данных выходят за пределы допустимого, можно использовать типы данных с плавающей точкой увеличенной точности. Эффективные реализации таких производных типов данных, в отличие от дробно-рациональных, сравнимы по скорости вычислений с аппаратным (при аналогичной длине мантиссы), при этом позволяют произвольно динамически увеличить точность. Одним из таких типов является mpf_t (multiple precision floating-point). Вычисления с типом данных mpf_t являются приближенными, а погрешность не учитывается. mpf_t можно применять в алгоритмах, когда возможно построить процедуру верификации результата, и увеличивать длину мантиссы (точность вычислений) при необходимости. Для получения гарантированного решения можно использовать тип mpfi (из одноименной библиотеки multiple precision floating-point interval library [4]).

Библиотека mpfi основана на gnu mp и mpfr (multiple-precision floating-point with correct rounding [7]). Вместо единого числа в mpfi используется пара чисел с плавающей точкой заданной длины (имеющих тип mpfr), представляющих интервал, внутри которого лежит реальное значение.

В отличие от mpf, mpfi позволяет получить гарантированные (за счет использования интервальных вычислений) и точно определенные результаты (за счет использования правильных округлений по стандарту IEEE 754, реализованные в mpfr). На рис. 2 представлены структуры типов mpf_t и mpfi_t (для 64-битных архитектур).

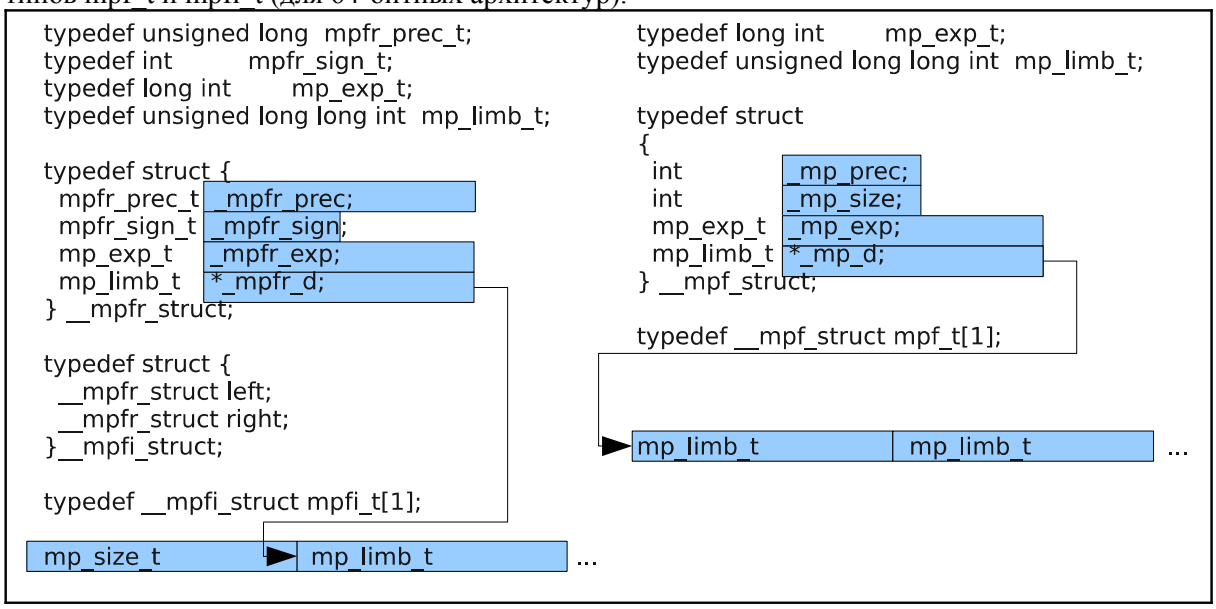

Рис. 2. Типы данных mpf и mpfi

2.2.1 Адаптация типов mpf_t и mpfi_t к mpi

Сложность эффективной адаптации производных типов динамической длины к mpi заключается в том, что в mpi нет стандартных средств для передачи типов:

- располагающихся в памяти непоследовательно,
- изменяющих длину блоков данных в процессе выполнения программы.

Рассмотрим структуру типов данных mpf и mpfi (рис. 2). Мантиссы хранится вне базовой структуры. В *mpf* на мантиссу указывает *_mp_d*. В случае с *mpfi* указатели *_mpfr_d* указывают на вторые элементы выделенных блоков памяти (начало мантиссы), в первых элементах хранится текущая длина мантиссы. Таким образом, данные в памяти лежат вразброс, и на основе стандартного представления mpf и mpfi невозможно определить *mpi* тип данных. Однако, эффективную передачу mpf и mpfi можно осуществить двумя способами:

- неполная сериализация,
- упорядочивание распределения в памяти.

В случае неполной сериализации достаточно последовательно упаковывать необходимые поля, закрашенные на рис. 3.

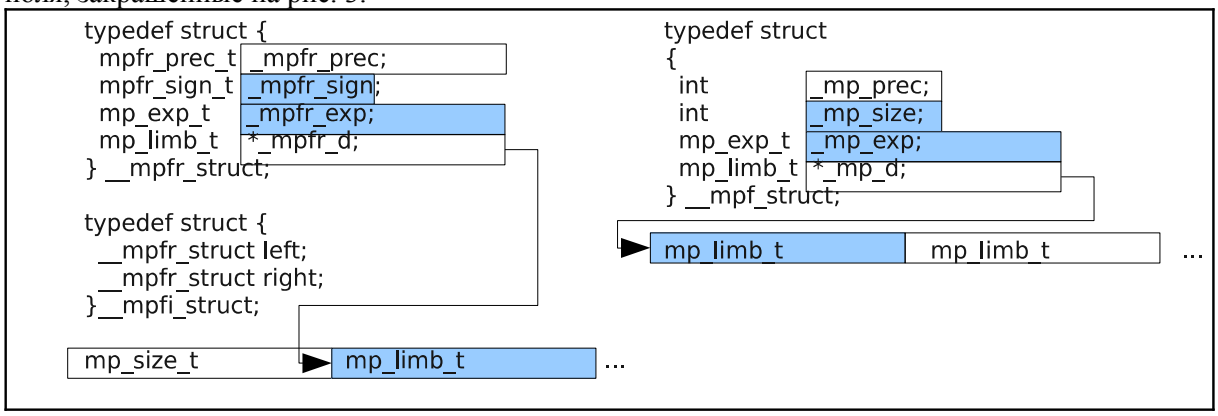

Рис. 3. Сериализация типов mpf и mpfi

В зависимости от алгоритма, если на стороне приемника не известна заданная точность отправителя, в список очевидно следует включить поля *_mp_prec* (*_mpfr_prec*). В некоторых случаях, можно передавать только *_mp_size* задействованных элементов массива мантиссы (в общем случае *_mp_size* совпадает с полной длиной *_mp_prec*).

Данный подход имеет очевидный минус, заключающийся в том, что приходится реализовывать функции передачи *mpi* на основе передачи массива байт (в случае неполной передачи мантиссы — массива байт заранее неизвестной длины). Некой прозрачности можно добиться, переопределив стандартные функции теми, которые будут проверять mpi тип (для *mpf* и *mpfi* можно взять не занятые) и выполнять передачу mpf (mpfi) или же вызывать стандартную функцию.

2.2.2 Перераспределение в памяти

Данный подход имеет смысл в случае, когда алгоритм оперирует числами заданной точности, которая не изменяется в процессе вычислений. Используя макросы приведенные на рис. 4 можно определить производные от *mpf* и *mpfi* типы *mpfn* и *mpfin*.

```
#define mpfi_type(prec) \
typedef struct \
{ \
__mpfr_struct left; \
__mpfr_struct right; \
mp_limb_t _mp_d_left[L(prec)+ 1];\
mp_limb_t _mp_d_right[L(prec)+1];\
} __mpfi_struct##prec; \
typedef __mpfi_struct##prec \
mpfi##prec##_t[1]; \
MPI_Datatype MPI_mpfi##prec;
```

```
#define mpf_type(prec) \
typedef struct \
{ \
int _mp_prec; \
int _mp_size; \
mp_exp_t _mp_exp; \
mp_limb_t *_mp_d; \
mp_limb_t _mp_d_real[L(prec)+ 1]; \
} __mpf_struct##prec; \
typedef __mpf_struct##prec \
mpf##prec##_t[1]; \
MPI_Datatype MPI_mpf##prec;
```

Рис. 4. Макросы определения производных типов mpfi и mpf

Очевидно, что структуры типов *mpfn* и *mpfin*, а также массивы объектов данных типов будут располагаться в памяти последовательно (не учитывая выравнивание). При этом, все операции, (кроме инициализации и задания точности) доступные для *mpf* (*mpfi*), могут быть без ограничений применены к *mpfn* (*mpfin*). Процедуры инициализации не принципиально отличается от оригинальных. Вместо выделения памяти под мантиссу требуется просто инициализировать указатели *_mp_d* (*left._mpfr_d* и *right._mpfr_d*) соответствующими адресами статически выделенной мантиссы.

Значительный положительный фактор данного подхода заключается в том, что на основе типов *mpfn* и *mpfin* (для каждой заданной точности) можно легко объявить тип данных *mpi*, и далее использовать все *mpi* функции доступные для *mpi*-производных типов данных без ограничений.

На рис. 5 представлены структуры *mpfn* и *mpfin* с точки зрения *mpi* (для 64-битных архитектур).

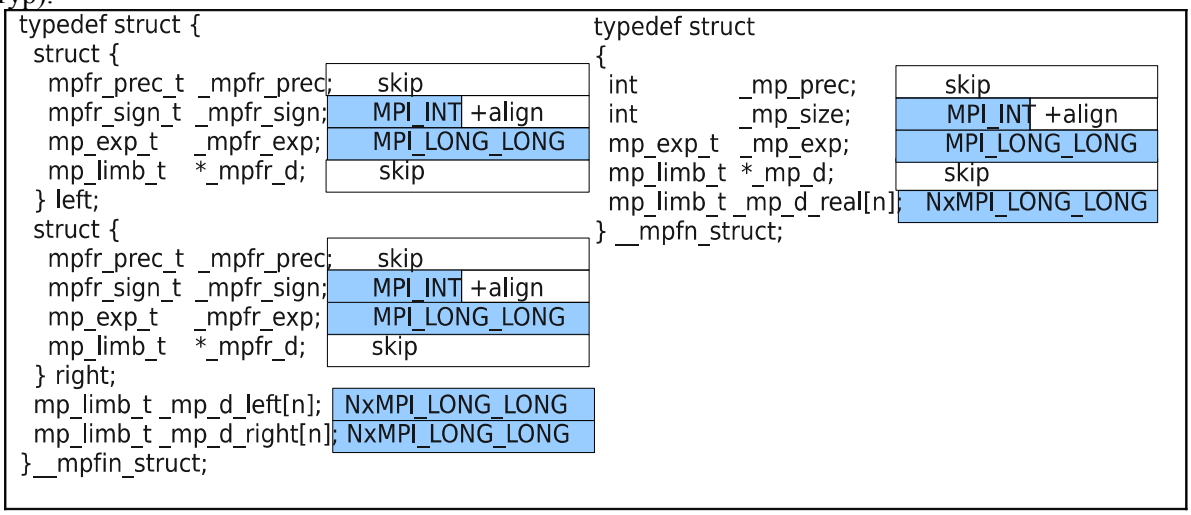

Рис. 5. Поля включаемые в описание mpi-типов

Если объявить *mpi* тип данных таким образом, как показано на рис. 4, указывая пропускать неокрашенные области, то указатели на мантиссы на стороне приемника переписываться не будут.

Еще один важный вопрос по данному подходу — как перераспределение влияет на производительность. В таблице 1 приведено сравнение попаданий в кэш исходных и модифицированных типов. Сравнение производилось на процессоре с кэш второго уровня 2 Мб (32 Кб первого уровня) на задаче решения системы уравнений методом Жордана-Гаусса малой размерности в 30 уравнений с точностью 192 бит. Временные характеристики приведены в разделе вычислительный эксперимент.

Таблица 1. Сравнение кэш-попаданий модифицированных типов.

	mpfi	mpfin	mpf	mpfn
I refs:	43366674	41850711	13762859	13601951
I1 misses:	23822	21566	10938	10205
L2i misses:	3456	3447	3236	3228
D refs:	18095665	17450618	5076607	5012224
D1 misses:	84427	65950	46662	38923
L2d misses:	13121	12126	9886	9646
L2 refs:	108249	87516	57600	49128
L2 misses:	16577	15573	13122	12874

На первый взгляд из обобщенных тестов следует, что показатели кэш-попаданий в модифицированных типах лучше. Однако, при более детальном рассмотрении по конкретным функциям, оказывается, что кэш-попадания лучше для простых функций (сравнение, сложение, вычитание и т.д.), но несколько хуже для действий умножения и деления. С ростом размерности задачи до 3000 уравнений и увеличением точности до 1024-2048 бит наблюдается незначительное превосходство исходных типов.

3. Параллельная версия симплекс метода

В случае с табличным симплекс методом, в связи со спецификой вычислений, удачнее будет декомпозиция по столбцам (рис. 6.)

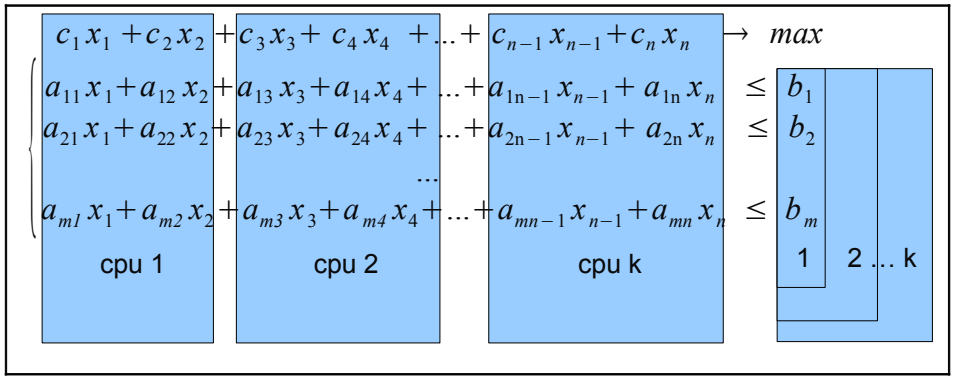

Рис. 6. Декомпозиция симплекс таблицы по процессорам

В данном случае все столбцы, коэффициенты целевой функции, делятся в равных пропорциях между процессами, вектор базисных переменных и правых частей рассылаются всем и обрабатываются отдельно. Тогда распараллеленный алгоритм будет следующим.

1. Из небазисных коэффициентов целевой функции выбрать на основе одного из критериев ведущий столбец (каждый процесс выбирает из имеющихся у него столбцов). При этом если все коэффициенты положительны – минимум найден.

2. Глобальный обмен между процессами значениями, полученными на шаге 1 и выбор оптимального.

3. Осуществляется только в процессе, «победившем» на шаге 2. Выбор, какую переменную убрать из базиса (выбор строки). Если все коэффициенты в выбранном ведущем столбце неположительные, то задача не неограниченна.

4. Процесс, «победивший» на шаге 2, глобально рассылает номера переменных вошедших и вышедших из базиса, а также ведущий столбец.

5. Каждый процесс осуществляет построение новой канонической формы по правилам симплекс метода на имеющихся у него столбцах.

Из вышесказанного можно сделать вывод, что алгоритм не сильно усложняется, и при этом использовано минимум коммуникаций, что должно привести к равномерной загрузке системы и высокой эффективности распараллеливания. Основная трудность возникает при введении процедуры порождения базисного плана. Если просто добавить необходимые искусственные переменные, а после окончания первого этапа соответствующие столбцы отбросить, то нагрузка будет не равномерной. Проблема может быть решена двумя способами:

1. Перераспределением столбцов после первого этапа, что сложно и накладно.

2. Формированием матрицы изначально таким образом, чтобы каждому процессу досталось примерно равное количество столбцов исходной задачи и столбцов вводимых на этапе порождения начального базисного допустимого решения. При этом необходимо, чтобы каждый процесс знал, сколько столбцов отбрасывать после завершения первого этапа.

В случае с модифицированным симплекс методом, изначальная матрица должна быть известна всем процессам, т.к. заранее не известно какая переменная войдет в базис и каким процессом будет обрабатываться. Кроме того, итерационная процедура основана на том, что на каждом шаге известны:

- базисные переменные и их значения;
- обращение базиса;
- соответствующие базису симплекс множители.

Дополнительные накладные расходы на коммуникацию при пересчетах приводят к тому, что не удается создать эффективную параллельную версию алгоритма [8].

3.1 Алгоритм параллельной версии симплекс-метода

Изложенный подход к распараллеливанию симплекс метода, был реализован в виде ***MPI*** программы ***plinpex*** *(parallel lineal exact solver)*, использующей для точных, дробно рациональных вычислений библиотеку ***GNU MP*** или интервальные вычисления на основе ***MPFI*** (задается опциями компиляции).

В качестве среды разработки использовался набор утилит ***GNU/gcc 4.4.2***, отладчик ***GNU/gdb***, профилер ***GNU/valgrind***. Написание, тестирование и вычислительный эксперимент проводились на операционной системе ***Gentoo GNU/Linux***, на платформе ***x86_64***.

Рассмотрим ключевые моменты алгоритма работы ***plinpex***.

1. После запуска параллельных версий программы и инициализации среды ***MPI***, каждая из них идентифицирует себя по рангу.

2. Процесс с рангом 0 (часто его называет корневой), производит считывание входного файла задачи линейного программирования в формате ***MPS*** [9]. Остальные процессы переходят на шаг 3.

2.1. Последовательное считывание секций входного файла, запоминание имен переменных (они выводятся при распечатке решения), инициализация и заполнение матрицы коэффициентов и коэффициентов целевой функции, значений столбца свободных членов.

2.2. Расширение матрицы искусственными переменными для приведения задачи к нормальной форме и дополнительными переменными, необходимыми для порождения базисного плана. Инициализация вектора переменных, входящих в базис.

3. Происходит синхронизация процессов, после чего, одновременно вызывается основной метод ***Solve()***.

3.1. Широковещательная передача от корневого процесса общих параметров задачи, таких как:

- количество строк и столбцов;
- количество основных, искусственных и дополнительных переменных.

3.2. Каждый процесс, зная свой ранг и общее количество процессов, вычисляет принадлежащие ему основные столбцы и дополнительные (необходимого для порождения допустимого базисного плана).

3.3. Все процессы, кроме корневого, инициализируют ресурсы для приема матрицы коэффициентов, вектора свободных членов, искусственной и основной целевых функций, вектора переменных входящих в базис.

3.4. Широковещательная передача от корневого процесса вектора свободных членов и вектора переменных входящих в базис.

3.5. Поочередная рассылка корневым процессом основных и искусственных столбцов матрицы всем остальным процессам.

3.6. Синхронизация всех процессов. Перед основным циклом итеративной процедуры симплекс метода.

3.6.1. Каждый процесс выбирает один столбец из имеющихся у него столбцов на основе правила Данцига (минимальный отрицательный коэффициент целевой функции).

3.6.2. С помощью вызова ***MPI_Allreduce*** находится минимальный отрицательный коэффициент целевой функции среди всех процессов и ранг процесса, у которого находится данный столбец (назовем его ведущий процесс).

3.6.3. Если на шаге 3.6.2. среди всех столбцов не оказалось отрицательных коэффициентов целевой функции, то данный этап решения закончен, иначе переход на шаг 3.6.4.

3.6.3.1. Если закончен первый этап порождения допустимого базисного решения, то все процессы исключают из дальнейших вычислений искусственные столбцы, и заменяют искусственную целевую функцию на основную.

3.6.3.2. Если закончен второй этап, то решение закончено, переход на шаг 4.

3.6.4. Ведущий процесс выбирает переменную исключаемую из базиса.

3.6.5. Ведущий процесс широковещательно рассылает индексы переменных вошедших и вышедших из базиса, а также ведущий столбец.

3.6.6. Каждый процесс осуществляет построение новой канонической формы по правилам симплекс метода на имеющихся у него столбцах.

3.6.7. Каждый процесс проверяет, принадлежат ли ему переменные, исключаемые или вошедшие в базис, и изменяет при необходимости вектор базисных переменных.

3.6.8. Итерация закончена, переход на шаг 3.6.1.

4. Процесс с рангом 0 выводит результаты решения задачи.

5. Завершение функционирования среды ***MPI*** и завершение параллельных процессов.

4. Вычислительный эксперимент

Вычислительный эксперимент производился на кластере "Скиф-Урал" Южно-Уральского государственного университета. Краткие технические характеристики представлены в таблице 2.

Таблица 2. Технические характеристики вычислительной платформы.

Тип процессора (на 1 blade)	2 четырехъядерных Intel Xeon E5472 3.0 GHz
Оперативная память (на 1 blade)	8 GB
Тип системной сети	InfiniBand (20Gbit/s, макс. задержка 2 мкс)
Операционная система	SUSE Linux Enterprise Server 10 x86_64

4.1 Оценка эффективности распараллеливания типов данных с плавающей точкой

Для вычисленного эксперимента использовалась параллельная версия алгоритма Жордана-Гаусса, адаптированная для вычислений с типами *mpf* (*mpfn*) и *mpfi* (*mpfin*). Эффективность распараллеливания представлена на на рис. 7.

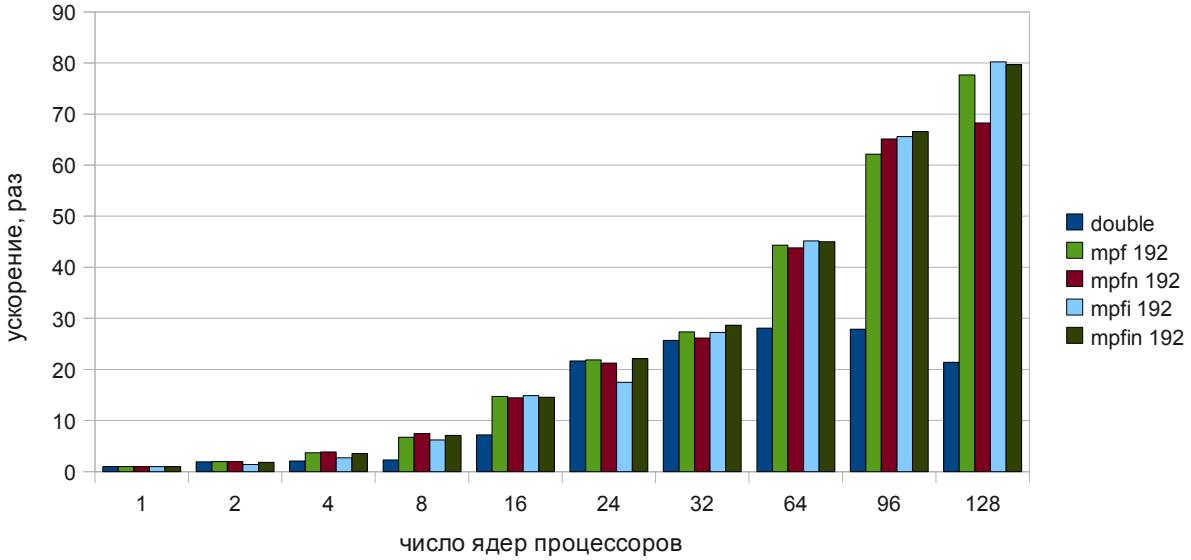

Рис. 7. Эффективность распараллеливания

Стоит отметить, что с увеличением точности (длины мантиссы) эффективность распараллеливания возрастает.

Таблица 3. Результаты численного эксперимента.

	double	mpf			mpfn			mpfi	mpfin
	53	64	192	704	64	192	704	192	192
1	43.04	1250.99	1817.42	6203.19	1238.5	1796.03	6219.7	3943.99	3817.11
2	22.06	627.28	913.78	3106.4	617.31	910.5	3132.24	2747.88	2073
4	20.41	315.35	489.06	1561.31	312.38	463.75	1573.92	1464.61	1072.29
8	18.83	171.44	269.83	795.95	166.03	239.94	804.95	634.02	537.36
16	5.98	86.91	123.35	413.09	85.19	124.01	410.43	265.07	261.93
24	1.98	60.12	83.07	325.72	61.5	84.44	277.53	225.7	172.57
32	1.68	47.7	66.41	212.97	47.3	68.72	220.01	144.66	133.1
64	1.53	27.9	40.98	135.55	28.06	40.97	123.79	87.36	84.79
96	1.54	19.85	29.25	92.61	19.88	27.57	84.55	60.14	57.31
128	2.01	16.86	23.41	77.1	18.53	26.31	76.18	49.15	47.9

4.2 Результаты решения ЗЛП

В качестве входных данных для вычислительного эксперимента использовались задачи линейного программирования из библиотеки Netlib [10]. В ней собраны сложные задачи нередко используемые для тестирования программных комплексов, решающих ЗЛП.

Таблица 4. ЗЛП отобранные из библиотеки Netlib.

Название задачи	Кол-во ограничений	Кол-во переменных	Кол-во ненулевых элементов	Оптимальное значение
SCSD6	148	1350	5666	5,0500000077E+01
SHARE1B	118	225	1182	-7,6589318579E+04
SCTAP1	301	480	2052	1,4122500000E+03

Данные задачи были выбраны по причине разного вида матриц. В ***SCSD6*** количество переменных существенно превышает количество ограничений, в то время как ***SCTAP1*** имеет форму близкую к квадратной. На рис. 8 – 13 приведены результаты экспериментов.

Рис. 8. Время счета на разном количестве процессоров для задачи SCSD6

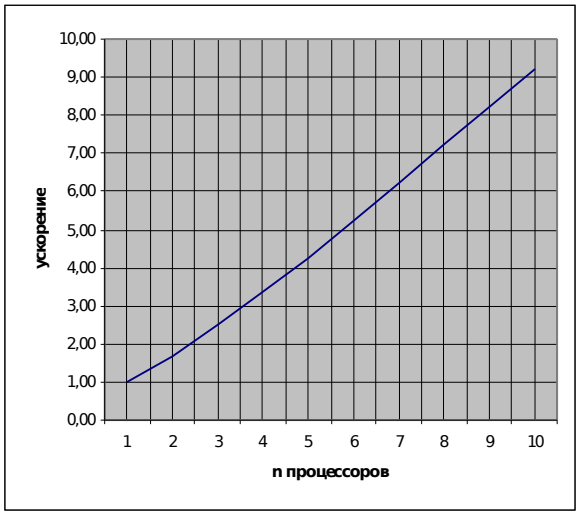

Рис. 9. График ускорения счета при введении дополнительных процессоров для задачи SCSD6

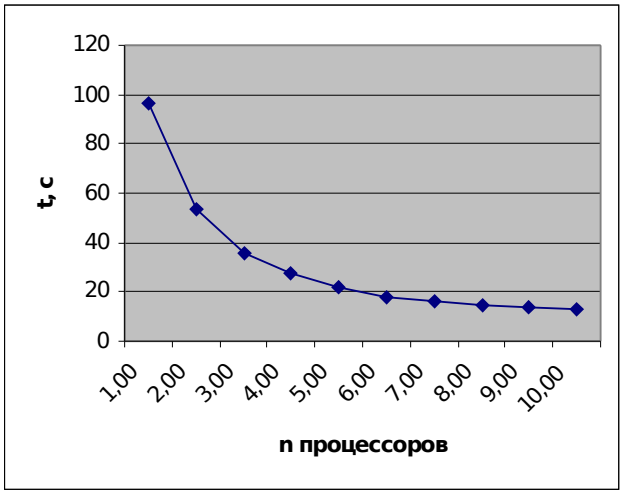

Рис. 10. Время счета на разном количестве процессоров для задачи SHARE1B

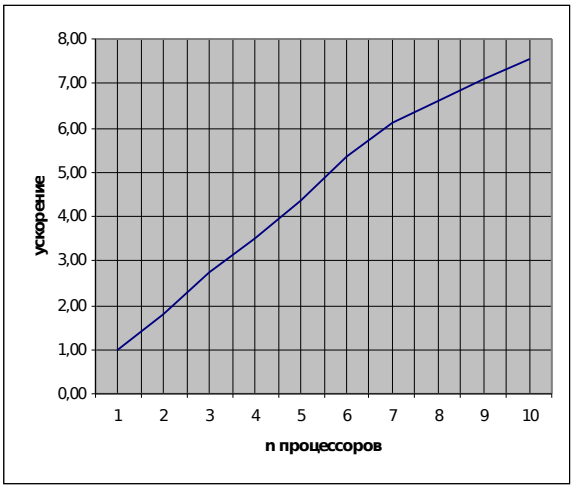

Рис. 11. График ускорения счета при введении дополнительных процессоров для задачи SHARE1B

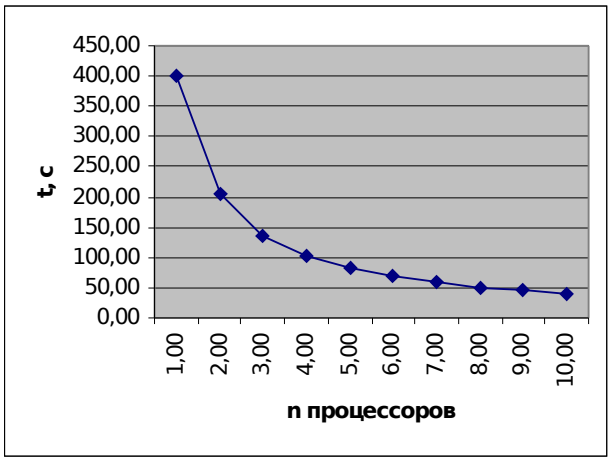

Рис. 12. Время счета на разном количестве процессоров для задачи SCTAP1

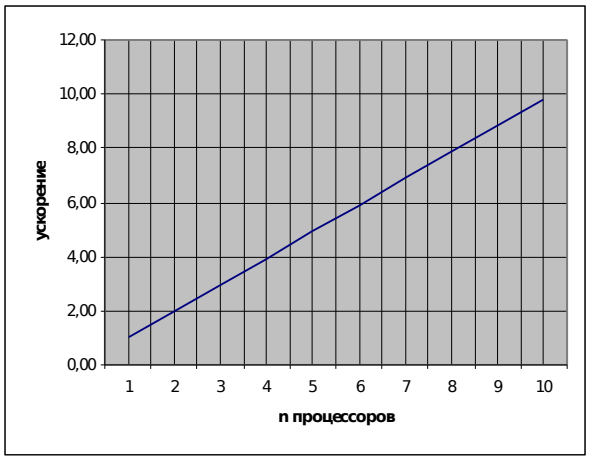

Рис. 13. График ускорения счета при введении дополнительных процессоров для задачи SCTAP1

5. Заключение

В данной статье были предложены методы для эффективного проведения расчетов с плавающей точкой заданной точности (с использованием библиотеки *gnu mp* и *mpfi*) в среде *mpi*. При решении подобных задач положительный эффект от распараллеливания состоит не только в ускорении вычислений, но и возможности решать задачи большей размерности, т.к. достаточно легко достичь границ, когда матрица целиком не будет умещаться в оперативной памяти одного узла.

В основе вычислений всех коммерческих программ лежат типы данных с плавающей точкой, поэтому они не могут гарантировать высокую точность решения. Исключением из ряда продуктов, считающих приближенно, являются две открытых реализации симплекс метода, использующих в вычислениях библиотеку точных вычислений GNU MP, что неизбежно ведет к существенному увеличению времени счета. Однако они не используют преимущества параллельного программирования, методы которого, при умелом использовании, позволяют сократить время расчетов и, следовательно, расширить круг решаемых задач. Для проведения научных расчетов, а также исследования методов параллельного и распределенного программирования, на кафедре Экономико-Математических Методов и Статистики был собран кластер (на основе имеющихся вычислительный средств учебной лаборатории). Данный кластер основан на свободно распространяемой реализации стандарта MPI – MPICH, и позволяет использовать преимущества и простоту данной библиотеки в программных реализациях. Для возможности использования классов точных вычислений в многопроцессорной среде MPI, был использован описанный в статье способ адаптации на основе сериализации объектов. Основной задачей было построение программы Plinpex решения задачи линейного программирования, основанной на параллельном алгоритме и использующей вычисления без округлений. Разработанный распараллеленный алгоритм использует всего две широковещательные коммуникации на каждой итерации основного цикла программы, что приводит к высокой оценке качества распараллеленного алгоритма.

Проведенный эксперимент показал, что реализованный метод эффективен на задачах разной размерности и соотношением количества ограничений к количеству переменных. В результате проведенного эксперимента можно сделать выводы о высокой эффективности распараллеленного алгоритма симплекс метода. Насколько позволяют оценить имеющиеся вычислительные средства, эффективность распараллеливания близка к 100% (на количестве узлов до 10). Однако общее время выполнения алгоритма достаточно велико, и чтобы расширить диапазон решаемых задач, требуется дополнительная доработка в сторону оптимизации вычислений на узлах.

Литература

1. Интернет-Университет Информационных Технологий: [http://www.intuit.ru/department/se/parallprog/4/1.html], 2007.
2. MPI: A Message Passing Interface Standard: [http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html], 12.06.1995.
3. Релиз 4.3.0 библиотеки GMP – GNU Multiple Precision Arithmetic Library: [http://gmplib.org/gmp4.3.html], 13.05.2009.
4. MPFI library: [http://perso.ens-lyon.fr/nathalie.revol/software.html], 17.04.2008.
5. Панюков А.В., Германенко М.И., Горбик В.В. Распараллеливание алгоритмов решения систем линейных алгебраических уравнений с применением вычислений без округлений // Параллельные вычислительные технологии (ПаВТ'2007), 2007. – Т.2. – С. 238–249.
6. Панюков А.В., Горбик В.В. Безошибочное решение задач линейного программирования на многопроцессорных системах // Параллельные вычислительные технологии (ПаВТ'2008), 2008. –С. 364-369.
7. MPFR library: [http://www.mpfr.org/], 7.11.2009.
8. Yarmish G. G. A Distributed Implementation of the Simplex Method// UMI Dissertations Publishing, 2001.
9. MPS format: [http://softlib.cs.rice.edu/pub/miplib/mps_format], 9.07.2008.
10. Netlib collection: [ftp://netlib2.cs.utk.edu/lp/data], 1996.