Министерство образования и науки Российской Федерации

Федеральное агентство по образованию

ГОУ Владимирский государственный университет

Кафедра вычислительной техники

РЕШЕНИЕ СЛАУ С ИСПОЛЬЗОВАНИЕМ КЛАСТЕРНОЙ СИСТЕМЫ

ВЛГУ. 230100. 12. 04. 00. ПЗ

студент гр. ИМ-205

_____ Н. В. Морев

”____”_____ 2005 г

2005

Содержание

1 Введение

Сегодня вычислительный мир находится в состоянии ожидания чего-то необычного: с одной стороны, появилась уникальная возможность создавать недорогие установки суперкомпьютерного класса — вычислительные кластеры, а с другой, есть устойчивое сомнение в серьезности этого направления. Однако при всех «за» и «против» постоянным обитателям списка Top500 — компаниям Cray, Sun, Hewlett-Packard и другим уже пришлось потесниться, пропустив 11 кластерных установок в элитные ряды чемпионов.

Сразу оговоримся, в компьютерной литературе понятие «кластер» употребляется в различных значениях, в частности, «кластерная» технология используется для повышения надежности серверов баз данных или web-серверов. В данной статье мы будем говорить только о вычислительных кластерах, используемых в качестве доступной альтернативы традиционным «числовым молотилкам» — суперкомпьютерам.

Если говорить кратко, то вычислительный кластер — это совокупность компьютеров, объединенных в рамках некоторой сети для решения одной задачи. В качестве вычислительных узлов обычно используются доступные на рынке однопроцессорные компьютеры, двух- или четырехпроцессорные SMP-серверы. Каждый узел работает под управлением своей копии операционной системы, в качестве которой чаще всего используются стандартные операционные системы: Linux, NT, Solaris и т.п. Состав и мощность узлов может меняться даже в рамках одного кластера, давая возможность создавать неоднородные системы. Выбор конкретной коммуникационной среды определяется многими факторами: особенностями класса решаемых задач, доступным финансированием, необходимостью последующего расширения кластера и т.п. Возможно включение в конфигурацию специализированных компьютеров, например, файл-сервера, и, как правило, предоставлена возможность удаленного доступа на кластер через Internet.

Ясно, что простор для творчества при проектировании кластеров огромен. Рассматривая крайние точки, кластером можно считать как пару ПК, связанных локальной 10-мегабитной сетью Ethernet, так и вычислительную систему, создаваемую в рамках проекта Cplant в Национальной лаборатории Sandia: 1400 рабочих станций на базе процессоров Alpha, связанных высокоскоростной сетью Myrinet. Чтобы немного почувствовать масштаб и технологию существующих систем, сделаем краткий обзор наиболее интересных современных кластерных установок. Кластерные проекты

Один из первых проектов, давший имя целому классу параллельных систем — кластеры Beowulf — возник в центре NASA Goddard Space Flight Center для поддержки необходимыми вычислительными ресурсами проекта Earth and Space Sciences. Проект Beowulf стартовал летом 1994 года, и вскоре был собран 16-процессорный кластер на

процессорах Intel 486DX4/100 МГц. На каждом узле было установлено по 16 Мбайт оперативной памяти и по 3 сетевых Ethernet-адаптера. Для работы в такой конфигурации были разработаны специальные драйверы, распределяющие трафик между доступными сетевыми картами.

Позже в GSFC был собран кластер theHIVE — Highly-parallel Integrated Virtual Environment (http://newton.gsfc.nasa.gov/thehive/).Этот кластер состоит из четырех подкластеров E, B, G, и DL, объединяя 332 процессора и два выделенных хост-узла. Все узлы данного кластера работают под управлением RedHat Linux.

В 1998 году в Лос-Аламосской национальной лаборатории астрофизик Майкл Уоррен и другие ученые из группы теоретической астрофизики построили суперкомпьютер Avalon, который представляет собой Linux-кластер на базе процессоров Alpha 21164A с тактовой частотой 533 МГц. Первоначально Avalon состоял из 68 процессоров, затем был расширен до 140. В каждом узле установлено по 256 Мбайт оперативной памяти, жесткий диск на 3 Гбайт и сетевой адаптер Fast Ethernet. Общая стоимость проекта Avalon составила 313 тыс. долл., а показанная им производительность на тесте LINPACK — 47,7 GFLOPS, позволила ему занять 114 место в 12-й редакции списка Top500 рядом с 152-процессорной системой IBM RS/6000 SP. В том же 1998 году на самой престижной конференции в области высокопроизводительных вычислений Supercomputing'98 создатели Avalon представили доклад «Avalon: An Alpha/Linux Cluster Achieves 10 Gflops for $150k», получивший первую премию в номинации «наилучшее отношение цена/производительность».

В апреле текущего года в рамках проекта AC3 в Корнелльском Университете для биомедицинских исследований был установлен кластер Velocity+, состоящий из 64 узлов с двумя процессорами Pentium III/733 МГц и 2 Гбайт оперативной памяти каждый и с общей дисковой памятью 27 Гбайт. Узлы работают под управлением Windows 2000 и объединены сетью cLAN компании Giganet.

Проект Lots of Boxes on Shelfes (LoBoS, http://www.lobos.nih.gov) реализован в Национальном Институте здоровья США в апреле 1997 года и интересен использованием в качестве коммуникационной среды технологии Gigabit Ethernet. Сначала кластер состоял из 47 узлов с двумя процессорами Pentium Pro/200 МГц, 128 Мбайт оперативной памяти и диском на 1,2 Гбайт на каждом узле. В 1998 году был реализован следующий этап проекта — LoBoS2, в ходе которого узлы были преобразованы в настольные компьютеры с сохранением объединения в кластер. Сейчас LoBoS2 состоит из 100 вычислительных узлов, содержащих по два процессора Pentium II/450 МГц, 256 Мбайт оперативной и 9 Гбайт дисковой памяти. Дополнительно к кластеру подключены 4 управляющих компьютера с общим RAID-массивом емкостью 1,2 Тбайт.

Интересная разработка появилась недавно в Университете штата Кентукки — кластер KLAT2 (Kentucky Linux Athlon Testbed 2, http://parallel.ru/news/kentucky_klat2.html). Система KLAT2 состоит из 64 бездисковых узлов с процессорами AMD Athlon/700 МГц и оперативной памятью 128 Мбайт на каждом. Программное обеспечение, компиляторы и математические библиотеки (SCALAPACK, BLACS и ATLAS) были доработаны для эффективного использования технологии 3DNow! процессоров AMD, что позволило увеличить производительность. Значительный интерес представляет и использованное сетевое решение, названное «Flat Neighbourghood Network» (FNN). В каждом узле установлено четыре сетевых адаптера Fast Ethernet от Smartlink, а узлы соединяются с помощью девяти 32-портовых коммутаторов. При этом для любых двух узлов всегда есть прямое соединение через один из коммутаторов, но нет необходимости в соединении всех узлов через единый коммутатор. Благодаря оптимизации программного обеспечения под архитектуру AMD и топологии FNN удалось добиться рекордного соотношения цена/производительность — 650 долл. за 1 GFLOPS.

Идея разбиения кластера на разделы получила интересное воплощение в проекте Chiba City (http://parallel.ru/news/anl_chibacity.html), реализованном в Аргоннской Национальной лаборатории. Главный раздел содержит 256 вычислительных узлов, на каждом из которых установлено два процессора Pentium III/500 МГц, 512 Мбайт оперативной памяти и локальный диск емкостью 9 Гбайт. Кроме вычислительного раздела в систему входят раздел визуализации (32 персональных компьютера IBM Intellistation с графическими платами Matrox Millenium G400, 512 Мбайт оперативной памяти и дисками 300 Гбайт), раздел хранения данных (8 серверов IBM Netfinity 7000 с процессорами Xeon/500 МГц и дисками по 300 Гбайт) и управляющий раздел (12 компьютеров IBM Netfinity 500). Все они объединены сетью Myrinet, которая используется для поддержки параллельных приложений, а также сетями Gigabit Ethernet и Fast Ethernet для управляющих и служебных целей. Все разделы делятся на «города» (town) по 32 компьютера. Каждый из них имеет своего «мэра», который локально обслуживает свой «город», снижая нагрузку на служебную сеть и обеспечивая быстрый доступ к локальным ресурсам.

Таким образом видно, что различных вариантов построения кластеров очень много. Одно из основных различий лежит в используемой сетевой технологии, выбор которой определяется, прежде всего, классом решаемых задач.

Первоначально Beowulf-кластеры строились на базе обычной 10-мегабитной сети Ethernet. Сегодня чаще всего используется сеть Fast Ethernet, как правило, на базе коммутаторов, основное достоинство которого — низкая стоимость оборудования (40-50 долл. за сетевой адаптер и менее 100 долл. за один порт коммутатора). Однако большие накладные расходы на передачу сообщений в рамках Fast Ethernet приводят к серьезным

ограничениям на спектр задач, которые можно эффективно решать на таком кластере. Если от кластера требуется большая универсальность, то нужно переходить на другие, более производительные коммуникационные технологии.

Какими же числовыми характеристиками выражается производительность коммуникационных сетей в кластерных системах? Основных характеристик две: латентность — время начальной задержки при посылке сообщений и пропускная способность сети, определяющая скорость передачи информации по каналам связи. При этом важны не столько пиковые характеристики, заявляемые производителем, сколько реальные, достигаемые на уровне пользовательских приложений, например, на уровне MPI-приложений. В частности, после вызова пользователем функции посылки сообщения Send() сообщение последовательно пройдет через целый набор слоев, определяемых особенностями организации программного обеспечения и аппаратуры, прежде, чем покинуть процессор — отсюда и вариации на тему латентности. Кстати, наличие латентности определяет и тот факт, что максимальная скорость передачи по сети не может быть достигнута на сообщениях с небольшой длиной.

Скорость передачи данных по сети в рамках технологий Fast Ethernet и Scalable Coherent Interface (SCI) зависит от длины сообщения. Для Fast Ethernet характерна большая величина латентности — 160–180 мкс, в то время как латентность для SCI это величина около 5,6 мкс. Максимальная скорость передачи для этих же технологий 10 Мбайт/с и 80 Мбайт/с соответственно.

Сейчас на рынке представлено несколько коммуникационных технологий «гигабитного» уровня. Прежде всего, это Gigabit Ethernet — развитие Fast Ethernet. По соображениям стоимости, производительности и масштабируемости Gigabit Ethernet редко используется в кластерных решениях и обычно предпочтение отдается технологиям Myrinet или SCI. Таблица 1 содержит краткое сравнительное описание технологий Fast Ethernet, SCI, Myrinet, cLAN и ServerNet, используемых в существующих кластерах.

Любопытно то, что на современном рынке представлено не так много поставщиков готовых кластерных решений. По всей видимости, это связано с доступностью комплектующих, легкостью построения самих систем, значительной ориентацией на свободно распространяемое ПО, а значит и проблематичностью последующей поддержки со стороны производителя. Среди наиболее известных поставщиков стоит отметить SGI, VALinux и Scali Computer.

Летом прошлого года Корнелльским университетом был основан Консорциум по кластерным технологиям (Advanced Cluster Computing Consortium — AC3), основная цель которого — координация работ в области кластерных технологий и помощь в осуществлении разработок в данной области. Ведущими компаниями, обеспечивающими инфра-

структуру консорциума, стали Dell, Intel и Microsoft. Среди других членов можно назвать Аргоннскую национальную лабораторию, Нью-Йоркский, Корнелльский и Колумбийский университеты, компании Compaq, Giganet, IBM, Kuck & Associates и другие.

В России всегда была высока потребность в высокопроизводительных вычислительных ресурсах, и относительно низкая стоимость кластерных проектов послужила серьезным толчком к широкому распространению подобных решений в нашей стране. Одним из первых появился кластер «Паритет», собранный в ИВВиБД и состоящий из восьми процессоров Pentium II, связанных сетью Myrinet. В 1999 году вариант кластерного решения на основе сети SCI был апробирован в НИЦЭВТ, который, по сути дела, и был пионером использования технологии SCI для построения параллельных систем в России.

Благодаря Федеральной целевой программе «Интеграция» ожидается появление большого числа высокопроизводительных кластеров в российских образовательных и научно-исследовательских организациях. Это и понятно, поскольку использование высокопроизводительных вычислительных ресурсов жизненно необходимо для получения качественно новых результатов и поддержания выполняемых фундаментальных научных исследований на мировом уровне.

Первой ласточкой в этом направлении стал высокопроизводительный кластер на базе коммуникационной сети SCI, установленный в Научно-исследовательском вычислительном центре Московского государственного университета (http://parallel.ru/cluster/). Кластер НИВЦ включает 12 двухпроцессорных серверов «Эксимер» на базе Intel Pentium III/500 МГц, в общей сложности 24 процессора с суммарной пиковой производительностью 12 млрд. операций в секунду. Общая стоимость системы — около 40 тыс. долл. или примерно 3,33 тыс. за 1 GFLOPS.

Вычислительные узлы кластера соединены однонаправленными каналами сети SCI в двумерный тор 3x4 и одновременно подключены к центральному серверу через вспомогательную сеть Fast Ethernet и коммутатор 3Com Superstack. Сеть SCI — это ядро кластера, делающее данную систему уникальной вычислительной установкой суперкомпьютерного класса, ориентированной на широкий класс задач. Максимальная скорость обмена данными по сети SCI в приложениях пользователя составляет более 80 Мбайт/с, а время латентности около 5,6 мкс. При построении данного вычислительного кластера использовалось интегрированное решение Wulfkit, разработанное компаниями Dolphin Interconnect Solutions и Scali Computer (Норвегия).

Основным средством параллельного программирования на кластере является MPI (Message Passing Interface) версии ScaMPI 1.9.1. На тесте LINPACK при решении системы линейных уравнений с матрицей размера 16000x16000 реально полученная производительность составила более 5,7 GFLOPS. На тестах пакета NPB производительность

кластера сравнима, а иногда и превосходит производительность суперкомпьютеров семейства Cray T3E с тем же самым числом процессоров.

Основная область применения вычислительного кластера НИВЦ МГУ — это поддержка фундаментальных научных исследований и учебного процесса. Важно отметить, что несмотря на неизбежные затраты на освоение новой вычислительной техники и новых технологий программирования, использование подобных параллельных систем зачастую дает уникальную возможность для получения принципиально новых научных результатов.

Из других интересных проектов следует отметить решение, реализованное в Санкт-Петербургском университете на базе технологии Fast Ethernet (http://www.ptc.spbu.ru): собранные кластеры могут использоваться и как полноценные независимые учебные классы, и как единая вычислительная установка, решающая единую задачу. В Самарском научном центре пошли по пути создания неоднородного вычислительного кластера, в составе которого работают компьютеры на базе процессоров Alpha и Pentium III. В Санкт-Петербургском техническом университете собирается установка на основе процессоров Alpha и сети Myrinet без использования локальных дисков на вычислительных узлах. В Уфимском государственном авиационном техническом университете проектируется кластер на базе двенадцати Alpha-станций, сети Fast Ethernet и ОС Linux.

Это перечисление можно продолжать и дальше, тем более, что число подобных систем растет очень быстро. Но основная идея понятна: каждый делает те акценты и собирает именно ту конфигурацию, которая лучше отвечает его потребностям, возможностям и желаниям.

Анализируя аргументы сторонников и противников кластеров сегодня можно с уверенностью констатировать два факта. Ясно то, что с помощью высокопроизводительных кластеров найден эффективный способ решения большого класса задач. Однако ясно и то, что пока кластеры ни в коей мере не являются вычислительной панацеей — у традиционных суперкомпьютеров практически все показатели намного лучше. . . кроме стоимости.

2 Способы организации параллельных вычислительных систем (программные средства)

Под параллельными вычислительными системами понимаются системы, которые позволяют распределять задачи или процессы между несколькими процессорами. Обычно параллельный кластер — это группа ЭВМ, специально выделенных для разделения ресурсов между собой. Кластер можно построить даже из всего двух компьютеров, но из-за того, что компьютерная техника сегодня относительно дешева, встречаются и кластеры из десятков, сотен и даже тысяч компьютеров (например, в Google работает кластер из 8000 машин).

Иногда параллельный кластер называют Beowulf-кластером. Этот термин обозначает вычислительный кластер, построенный на основе общедоступного аппаратного обеспечения и бесплатных открытых программных средств.

Для организации кластеров наиболее часто используются следующие программные средства: MPI, MOSIX, PVM.

Для выполнения курсового проекта была выбрана технология MPI в виде одной из ее реализаций MPIch версии 1.2.6. Этот выбор был обоснован следующими причинами:

- MPI — самый распространенный стандарт для организации параллельных вычислительных систем.
- Установка MPI проще, чем установка остальных систем, т. к. не требуется переустанавливать полностью систему и не требуется полностью занимать сервер под узел кластера, он может выполнять эту функцию параллельно с остальными задачами.
- По стандарту MPI написано много книг и пособий как на русском так и на английском языке.

2.1 PVM

Основой вычислительной среды кластера Beowulf является параллельная вируталь-ная машина PVM. PVM (Параллельная Виртуальная Машина) — это пакет программ, который позволяет использовать связанный в локальную сеть набор разнородных компьютеров, работающих под операционной системой Unix, как один большой параллельный компьютер. Таким образом, проблема больших вычислений может быть весьма эффективно решена за счет использования совокупной мощности и памяти большого числа компьютеров. Пакет программ PVM легко переносится на любую платформу. Исходные тексты, свободно распространяемые netlib, был скомпилирован на компьютерах начиная от laptop и до CRAY.

Параллельную виртуальную машину можно определить как часть средств реального вычислительного комплекса (процессоры, память, периферийные устройства и т.д.), предназначенную для выполнения множества задач, участвующих в получении общего результата вычислений. В общем случае число задач может превосходить число процессоров, включенных в PVM. Кроме того, в состав PVM можно включать довольно разнородные вычислительные машины, несовместимые по системам команд и форматам данных. Иначе говоря, Параллельной Виртуальной Машиной может стать как отдельно взятый ПК, так и локальная сеть, включающая в себя суперкомпьютеры с параллельной архитектурой, универсальные ЭВМ, графические рабочие станции и все те же маломощные ПК. Важно лишь, чтобы о включаемых в PVM вычислительных средствах имелась информация в используемом программном обеспечении PVM. Благодаря этому программному обеспечению пользователь может считать, что он общается с одной вычислительной машиной, в которой возможно параллельное выполнение множества задач.

PVM позволяет пользователям использовать существующие аппаратные средства, для решения намного более сложных задач при минимальной дополнительной стоимости. Сотни исследовательских групп во всем мире используют PVM, чтобы решить важные научные, технические, и медицинские проблемы, а так же используют PVM как образовательный инструмент, для преподавания параллельного программирования. В настоящее время, PVM стал де факто стандартом для распределенных вычислений.

Главная цель использования PVM — это повышение скорости вычислений за счет их параллельного выполнения. Функционирование PVM основано на механизмах обмена информацией между задачами, выполняемыми в ее среде. В этом отношении наиболее удобно реализовывать PVM в рамках многопроцессорного вычислительного комплекса, выделив виртуальной машине несколько процессоров и общее или индивидуальные (в зависимости от условий) ОЗУ. Использование PVM доспустимо как на многопроцессорных компьютерах (SMP) так и на вычислительных комплексах, построенных по кластерной технологии. При использовании PVM, как правило, значительно упрощаются проблемы быстрого информационного обмена между задачами, а также проблемы согласования форматов представления данных между задачами, выполняемыми на разных процессорах

Эффективное программирование для PVM начинается с того, что алгоритм вычислений следует адаптировать к составу PVM и к ее характеристикам. Это очень творческая задача, которая во многих случаях должна решаться программистом. Кроме задачи распараллеливания вычислений с необходимостью возникает и задача управления вычислительным процессом, координации действий задач — участников этого процесса. Иногда

для управления приходится создавать специальную задачу, которая сама не участвуя в вычислениях, обеспечивает согласованную работу остальных задач — вычислителей.

Ранее вскользь упоминалось, что при параллельных вычислениях необходимо программировать специальные действия по координации работы задач, такие как процессы запуска задач на процессорах кластера, управление обменом данных между задачами и пр. Также следует четко определить «область деятельности» для каждой задачи.

Наиболее простой и популярный способ организации параллельного счета выглядит следующим образом. Сначала запускается одна задача (master), которая в коллективе задач будет играть функции координатора работ. Эта задача производит некоторые подготовительные действия, например инициализация начальных условий, после чего запускает остальные задачи (slaves), которым может соответствовать либо тот же исполняемый файл, либо разные исполняемые файлы. Такой вариант организации параллельных вычислений предпочтительнее при усложнении логики управления вычислительным процессом, а также когда алгоритмы, реализованные в разных задачах, существенно различаются или имеется большой объем операций (например, ввода-вывода), которые обслуживают вычислительный процесс в целом.

2.2 MOSIX

MOSIX — системное ПО для ядер UNIX-like ОС, таких как Linux, состоящее из адаптивных алгоритмов разделения ресурсов. Это позволяет множеству однопроцессорных (UP — Uni-Processors) и SMP узлам запускаться для работы в замкнутой кооперации. Алгоритмы разделения ресурсов MOSIX разработаны в соответствии с использованием ресурсов узлов в режиме реального времени. Это достигается миграцией процессов с одного узла на другой, преимущественно и прозрачно, для балансировки загрузки (load-balancing) и предотвращения истощения памяти. Целью — увеличение суммарной производительности и создание удобной многопользовательской и разделяемой по времени среды для запуска последовательных и параллельных приложений. Стандартное время выполнения среды MOSIX представляет собой СС, в котором глобальные ресурсы кластера доступны каждому узлу. При запрещении автоматической миграции процессов, пользователь может переключить конфигурацию в простой СС или другой single user MPP режим (single user Massively Parallel Processing — обработка данных с массовым параллелизмом в однопользовательском режиме).

Текущая реализация MOSIX разработана для СС на основе рабочих станций X86/Pentium, обоих типов (UP и SMP), соединённых стандартной ЛВС. Возможна конфигурация, основывающаяся на маленьких кластерах с PC, соединенными Ethernet, либо на основе производительных систем с большим числом узлов на основе high-end Pentium

SMP серверов соединенных посредством Gigabit LAN, например.

Технология MOSIX состоит из двух составляющих: механизма PPM и набора алгоритмов для адаптивного разделения ресурсов. Обе составляющие реализуются на уровне ядра, используя загружаемые модули, не модифицирующие интерфейс ядра, таким образом, оставаясь полностью прозрачным для уровня приложений, тем более для уровня пользователя.

PMM может мигрировать любой процесс, в любое время, на любой доступный узел кластера. Обычно, миграция основывается на информации предоставляемой одним из алгоритмов разделения ресурсов, но пользователи могут аннулировать любой автоматический выбор системы миграции своего процесса и выполнит то же самое при ручном управлении. Также ручная миграция может инициироваться процессом синхронно, или по явному запросу другого процесса любого пользователя (или суперпользователя). Ручная миграция процессов может быть полезна для представления конкретных режимов (policy) или для тестирования различных планирующих алгоритмов (scheduling algorithms). Необходимо также отметить, что суперпользователь имеет дополнительные привилегии относительно PPM, например, определение установок общей политики и списка доступных узлов кластера для миграции процессов.

MOSIX не имеет центрального управления и отношений master-slave между узлами: каждый узел может оперировать как автономная система и это делает весь контроль независимым. Такое решение позволяет динамическую конфигурацию, когда узлы способны подсоединяться или отключаться от сети с минимальными нарушениями работоспособности. Алгоритмы масштабирования обеспечивают, что система запустится как на большой конфигурации, так и на малой. Масштабирование достигается введением случайности в алгоритмах контроля системы, где каждый узел основывается на выборе частичных знаний о состоянии на других узлах и делает неравномерным попытку определить общее состояние кластера или конкретного узла. Например, в вероятностной информации распространенных алгоритмов, каждый узел посылает через определённые промежутки времени информацию о доступных ресурсах на случайно выбранную группу других узлов. Одновременно это все поддерживает маленькое "окошко"с наиболее свежеполученной информацией. Эта сема поддерживает масштабирование, равномерная информация распространяется и динамически конфигурируется.

2.3 MPI

MPI расшифровывается как «Message passing interface» («Интерфейс передачи сообщений»). MPI — это стандарт на программный инструментарий для обеспечения связи между отдельными процессами параллельной задачи. MPI предоставляет программисту

единый механизм взаимодействия процессов внутри параллельно исполняемой задачи независимо от машинной архитектуры (однопроцессорные, многопроцессорные с общей или раздельной памятью), взаимного расположения процессов (на одном физическом процессоре или на разных) и API операционной системы. Программа, использующая MPI, легко отлаживается и переносится на другие платформы, часто для этого достаточно простой перекомпиляции исходного текста программы.

Основное отличие стандарта MPI от его предшественников (p4, PVM) — понятие коммуникатора. Все операции синхронизации и передачи сообщений локализуются внутри коммуникатора. С коммуникатором связывается группа процессов. В частности, все коллективные операции вызываются одновременно на всех процессах, входящих в эту группу. Поскольку взаимодействие между процессами инкапсулируется внутри коммуникатора, на базе MPI можно создавать библиотеки параллельных программ.

В настоящее время разными коллективами разработчиков написано несколько программных пакетов, удовлетворяющих спецификации MPI, в частности: MPICH, LAM, HPVM и так далее. В двух словах охарактеризуем наиболее распространенные из этих пакетов. Если говорить о LAM, то основное достоинство этого пакета — богатые отладочные возможности. Трассировка обращений к MPI и анализ состояния параллельной программы после аварийного завершения делают процесс отладки менее тяжелым и более продуктивным. С другой стороны, пакет MPICH более мобилен, следуя простым инструкциям можно перенести MPI на новую платформу (например с Linux на Windows или наоборот). Для этого потребуется необходимо написать лишь несколько драйверов нижнего уровня.

MPI — это хорошо стандартизованный механизм для построения программ по модели обмена сообщениями. Существуют стандартные «привязки» MPI к языкам C/C++, Fortran 77/90. Имеются и бесплатные и коммерческие реализации почти для всех суперкомпьютерных платформ, а также для High Performance кластерных систем, построенных на узлах с ОС Unix, Linux и Windows. В настоящее время MPI — наиболее широко используемый и динамично развивающийся интерфейс из своего класса. За стандартизацию MPI отвечает MPI Forum[1]. В новой версии стандарта 2.0 описано большое число новых интересных механизмов и процедур для организации функционирования параллельных программ: динамическое управление процессами, односторонние коммуникации (Put/Get), параллельные I/O. Но к сожалению, пока нет полных готовых реализаций этой версии стандарта, хотя часть из нововведений уже активно используется.

При запуске задачи создается группа из P процессов. Группа идентифицируется целочисленным дескриптором (коммуникатором). Внутри группы процессы нумеруются от 0

[1] http://www.mpi-forum.org

до $P-1$. В ходе решения задачи исходная группа (ей присвоено имя `MPI_COMM_WORLD`) может делиться на подгруппы, подгруппы могут объединяться в новую группу, имеющую свой коммуникатор. Таким образом, процесс имеет возможность одновременно принадлежать нескольким группам процессов. Каждому процессу доступен свой номер внутри любой группы, членом которой он является.

Поведение всех процессов описывается одной и той же программой. Межпроцессные коммуникации в ней программируются явно с использованием библиотеки MPI, которая и диктует стандарт программирования. Квазиодновременный запуск исходной группы процессов производится средствами операционной системы. При этом P определяется желанием пользователя, а отнюдь не количеством доступных процессоров!

Таким образом, в программе «под одной крышей» закодировано поведение всех процессов. В этом и заключена парадигма программирования Single Program — Multiple Data (SPMD).

Обычно поведение процессов одинаково для всех, кроме одного, который выполняет координирующие функции, не отказываясь, впрочем, взять на себя и часть общей работы. В качестве координатора обычно выбирают процесс с номером 0.

3 Методы решения СЛАУ

Система линейных алгебраических уравнений (СЛАУ) — это набор из n линейных уравнений с k неизвестными переменными. СЛАУ могут быть представлены в матричной форме, как матричное уравнение:

$$A\vec{x} = \vec{b},$$

где A — матрица коэфициентов, $\vec{x}$ — вектор-столбец переменных и $\vec{b}$ — вектор-столбец постоянных значений.

Если $k < n$, то система не имеет решения. Если $k = n$ и матрица A невырождена, то система имеет единственное решение, состоящее из n переменных. В частности, если можно получить матрицу A^{-1}, обратную по отношению к A, то существует единственное решение:

$$\vec{x} = A^{-1}\vec{b}.$$

Если $\vec{b} = \vec{0}$, то решение $\vec{x} = \vec{0}$.

Основная задача вычислительной алгебры — решение систем линейных алгебраических уравнений (СЛАУ):

$$\begin{aligned}
&a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1,\\
&a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2,\\
&\ldots\\
&a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n.
\end{aligned}$$

Предполагается, что матрица A неособенная, т. е. $\det A \neq 0$, и решение единственно.

3.1 Прямые и итерационные методы

Численные методы решения СЛАУ делятся на две большие группы: прямые и итерационные. Прямые методы при отсутствии ошибок округления за конечное число арифметических операций позволяют получить точное решение x^*. В итерационных методах задается начальное приближение x_0 и строится последовательность $\{x_k\} \to_{k\to\infty} x^*$, где k — номер итерации. В действительности итерационный процесс прекращается, как только x_k становится достаточно близким к x^*. Итерационные методы привлекательнее с точки зрения объема вычислений и требуемой памяти, когда решаются системы с матрицами высокой размерности. При небольших порядках системы используют прямые методы либо прямые методы в сочетании с итерационными методами.

Рассмотрим два метода решения СЛАУ — Якоби и Гаусса–Зейделя. Метод Якоби рассмотрен в силу того, что вычисления компонент вектора внутри одной итерации независимы друг от друга и параллелизм очевиден, поэтому легко написать MPI программу.

Метод Гаусса–Зейделя более эффективен, поскольку требует заметно меньше итераций. Высокая сходимость к решению в этом методе достигается за счет выбора матрицы расщепления, лучше аппроксимирующей матрицу A. Однако в методе Гаусса–Зейделя вычисление каждой компоненты вектора зависит от компонент вектора, вычисленных на этой же итерации.

3.2 Метод простой итерации (метод Якоби)

Пусть требуется решить систему

$$Ax = b; x, b \in \mathbf{R}_n.$$

Представим $A = A^- + D + A^+$, где D — диагональная матрица с диагональными членами матрицы A; A^- — часть матрицы A, лежащая ниже центральной диагонали; A^+ — часть матрицы A, лежащая выше центральной диагонали. Тогда

$$(A^- + D + A^+)x = b,$$

или

$$Dx = -(A^- + A^+)x + b.$$

Запишем итерационный метод в виде

$$Dx^{k+1} = -(A^- + A^+)x^k + b; k = 0, 1, 2, \ldots.$$

Разрешим его относительно x^{k+1}:

$$x^{k+1} = -D^{-1}(A^- + A^+)x^k + D^{-1}b; k = 0, 1, \ldots$$

$$x_i^{k+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^k - \sum_{j=i+1}^{n} a_{ij} x_j^k \right); i = 1, \ldots, n. \quad (1)$$

Нетрудно убедиться, что метод Якоби в координатной форме есть не что иное, как разрешение каждого из уравнений системы относительно одной из компонент вектора. Из первого уравнения системы выражается x_1 и его значение принимается за значение x_1^{k+1}. Из второго уравнения определяется x_2 и его значение принимается за x_2^{k+1} и т. д. Переменные в правой части этих соотношений при этом полагаются равными их значениям на предыдущей итерации.

3.3 Метод Гаусса–Зейделя

Пусть решаемая система представлена в виде

$$(A^- + D + A^+)x = b.$$

Итерационная схема Гаусса–Зейделя следует из этого представления системы:

$$(A^- + D)x^{k+1} = b - A^+x^k; k = 0, 1, 2, \ldots,$$

или

$$Dx^{k+1} = -(A^-x^{k+1} - A^+x^k + b; k = 0, 1, 2, \ldots.$$

Приведем метод Гаусса–Зейделя к стандартному виду:

$$x^{k+1} = -(A^- + D)^{-1}A^+x^k + (A^- + D)^{-1}b; k = 0, 1, \ldots.$$

Стандартная форма метода позволяет выписать его итерационную матрицу и провести над ней очевидные преобразования:

$$B = -(A^- + D)^{-1}A^+ = -(A^- + D)^{-1}(A - D - A^-) = I - (A^- + D)^{-1}A.$$

Представим метод Гаусса–Зейделя в координатной форме для системы общего вида:

$$x_i^{k+1} = \left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{k+1} - \sum_{j=i+1}^{n} a_{ij}x_j^k\right)/a_{ii}; i = 1, \ldots, n; k = 0, 1, 2, \ldots. \quad (2)$$

Координатная форма метода Гаусса–Зейделя отличается от координатной формы метода Якоби лишь тем, что первая сумма в правой части итерационной формулы содержит компоненты вектора решения не на k-й, а на $(k + 1)$-й итерации.

4 Разработка алгоритма

4.1 Последовательный алгоритм метода простой итерации

Для реализации метода простой итерации должны быть заданы матрица СЛАУ A, вектор свободных членов B, начальное приближение вектора X, точность вычислений ε. Тогда новые значения вектора X вычисляются по формуле (1).

Критерий завершения процесса вычислений

$$\|x^{k+1} - x^k\| = \max |x_i^{k+1} - x_i^k| < \varepsilon, 1 \leq i \leq n,$$

где x^k — приближенное значение решения на k-м шаге численного метода.

В основном цикле процедуры сохраняется значение вектора, вычисленное на предыдущей итерации цикла, вычисляется новое значение вектора, затем вычисляется норма погрешности. Если достигнута необходимая точность вычислений, осуществляется выход из цикла.

4.2 Параллельный алгоритм метода простой итерации

Следующая система уравнений описывает метод простой итерации:

$$\begin{aligned} X_1^{k+1} &= f_1(x_1^k, x_2^k, \ldots, x_n^k), \\ X_2^{k+1} &= f_2(x_1^k, x_2^k, \ldots, x_n^k), \\ &\ldots \\ X_n^{k+1} &= f_n(x_1^k, x_2^k, \ldots, x_n^k). \end{aligned}$$

Вычисления каждой координаты вектора зависят от значений вектора, вычисленных на предыдущей итерации, и не зависят между собой. Поэтому, естественно, для параллельной реализации можно предложить следующий алгоритм.

Количество координат вектора, которые вычисляются в каждом процессе, определим следующим образом.

```
size = MATR_SIZE/numprocs + (MATR_SIZE % numprocs > myid ? 1 : 0);
```

где `size` — количество координат вектора, вычисляемых в данном процессе, `myid` – – собственный номер процесса, `numprocs` — количество процессов приложения, `MATR_SIZE` — размерность вектора.

Тогда каждый процесс может вычислять свое количество `size` новых значений координат вектора. После этого процессы могут обменяться вновь вычисленными значениями, что позволит главному процессу произвести оценку точности вычислений.

Прежде всего определяется номер элемента вектора `first`, с которого вычисляются новые значения в каждом процессе. Переменная `first` будет в общем случае иметь

разные значения, так как количество элементов вектора size различно в процессах. Вычислить значение переменной удобно вызовом коллективной функции MPI_Scan:

```
MPI_Scan(&size, &first,1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
first-= size;
```

Затем в цикле (аналогично последовательному варианту) сохраняется значение вектора, вычисленного на предыдущей итерации; вызовом процедуры вычисляются новые значения вектора, осуществляется рассылка вычисленных значений вектора от всех всем.

```
MPI_Allgatherv(&X[first], size, MPI_DOUBLE, X, sendcounts, displs, MPI_DOUBLE, ...
    MPI_COMM_WORLD );
```

Для выполнения коллективной функции MPI_Allgatherv необходимо заполнить массивы sendcounts и displs (заполнение массивов лучше выполнить до цикла). После обмена каждый процесс имеет новый вектор X. В корневом процессе вычисляется норма погрешности и сравнивается с заданной. Условие выхода из цикла $Result \neq 0$, поэтому корневой процесс рассылает значение Result всем процессам:

```
MPI_Bcast(&Result, 1, MPI_INT, Root, MPI_COMM_WORLD);
```

В корневом процессе происходит задание размерности исходной системы MATR_SIZE, заполняется матрица значений системы AB (матрица и столбец свободных членов), начальное приближение вектора решения X, точность вычислений Error. После инициализации MPI, определения количества процессов в приложении numprocs, собственного номера процесса myid каждый процесс получает от корневого процесса заданную размерность исходной матрицы, точность вычислений, начальное приближение вектора X:

```
MPI_Bcast(&MATR_SIZE, 1, MPI_INT, Root, MPI_COMM_WORLD);
MPI_Bcast(&Error, 1, MPI_DOUBLE, Root, MPI_COMM_WORLD);
MPI_Bcast(X, MATR_SIZE, MPI_DOUBLE, Root, MPI_COMM_WORLD);
```

После этого каждый процесс определяет количество координат вектора size, которые будут вычисляться в данном процессе (разные процессы, возможно, будут иметь разные значения size):

```
size = (MATR_SIZE/numprocs)+((MATR_SIZE % numprocs) > myid ? 1 : 0 );
```

Далее необходимо разделить исходную матрицу AB по процессам: по size строк матрицы в каждый процесс:

```
MPI_Scatterv(AB, sendcounts, displs, MPI_DOUBLE, A, (MATR_SIZE+1) * size, MPI_DOUBLE,Root...
    , MPI_COMM_WORLD );
```

Для выполнения функции MPI_Scatterv необходимо заполнить массивы sendcounts, displs. В первом из них находится количество элементов матрицы

SIZE каждого процесса, во втором — расстояния между элементами вектора, распределенного по процессам, причем

```
SIZE=(MATR_SIZE+1)*size;
```

где MATR_SIZE+1 — количество элементов в строке матрицы, size — количество строк матрицы в процессе. Тогда заполнение первого массива sendcounts выполняется вызовом функции:

```
MPI_Gather(&SIZE, 1, MPI_INT, sendcounts, 1, MPI_INT, Root, MPI_COMM_WORLD);
```

4.3 Параллельный алгоритм метода Гаусса–Зейделя

Отличие метода Гаусса–Зейделя от метода простой итерации заключается в том, что новые значения вектора вычисляются не только на основании значений предыдущей итерации, но и с использованием значений уже вычисленных на данной итерации (формула (2)).

Следующая система уравнений описывает метод Гаусса-Зейделя.

$$
\begin{aligned}
X_1^{k+1} &= f_1(x_1^k, x_2^k, \ldots, x_n^k) \\
X_2^{k+1} &= f_2(x_1^{k+1}, x_2^k, \ldots, x_n^k) \\
&\ldots \\
X_n^{k+1} &= f_n(x_1^{k+1}, x_2^{k+1}, \ldots, x_{n-1}^{k+1}, x_n^k)
\end{aligned}
$$

Вычисления каждой координаты вектора зависят от значений, вычисленных на предыдущей итерации, и значений координат вектора вычисленных на данной итерации. Поэтому нельзя реализовывать параллельный алгоритм, аналогичный методу простой итерации: каждый процесс не может начинать вычисления пока, не закончит вычисления предыдущий процесс.

Можно предложить следующий модифицированный метод Гаусса–Зейделя для параллельной реализации. Разделим вычисления координат вектора по процессам аналогично методу простой итерации. Будем в каждом процессе вычислять свое количество координат вектора по методу Гаусса-Зейделя, используя только вычисленные значения вектора данного процесса. Различие в параллельной реализации по сравнению с методом простой итерации заключается только в процедуре вычисления значений вектора.

5 Разработка программы

Проверка правильности работы программы осуществлялась сравнением результатов, которые она выдает с результатами полученными в пакете Matlab и с результатами выполнения других программ решения СЛАУ, вычисляющих решение другими методами (метод Гаусса, метода Гаусса-Зейделя).

Предварительно была написана обычная последовательная версия программы. Затем она была модифицирована для выполнения в среде кластера MPI.

Программа принимает в качестве входных параметров размерность системы уравнений и матрицу коэфициентов СЛАУ. На выходе программы выдает время выполнения отдельных частей программы, количество итераций и вектор решения системы. Требуемая ошибка решения системы задается непосредственно в программе на этапе компиляции.

Количество процессов задается не в самой программе, а в качестве параметра скрипта, с помощью которого программа запускается в среде MPI. Выполнение программы происходит следующим образом: программа автоматически одновременно запускается на всех узлах системы, при этом каждому процессу присваивается свой уникальный номер, по которому она может идентифицировать номер своего процесса. По ходу выполнения этих параллельных процессов, они могут обмениваться различными данными, требуемыми в процессе решения задачи. Для этого и предназначены большинство функций MPI.

5.1 Выбор языка программирования

Разработка программы велась в среде Unix-подобной операционной системы FreeBSD для процессоров архитектуры Intel IA-32. Языком программирования для написания программы был выбран C. Выбор языка обусловлен тем, что библиотеки MPI в стандартной поставке идут только для двух языков — C и Fortran. При этом C является более понятным и знакомым языком программирования. Компиляция программы осуществлялась с помощью специального компилятора `mpicc`, подготавливающего программу для выполнения в среде MPI.

5.2 Анализ и выбор методов декомпозиции

Распараллеливание программ сводится к процессу декомпозиции задачи на независимые процессы, которые не требуют последовательного исполнения и могут, соответственно, быть выполнены на разных процессорах независимо друг от друга.

Варианты декомпозиции существует три основных варианта декомпозиции: простая декомпозиция (trival), функциональная (functional) и декомпозиция данных. Вопрос об ис-

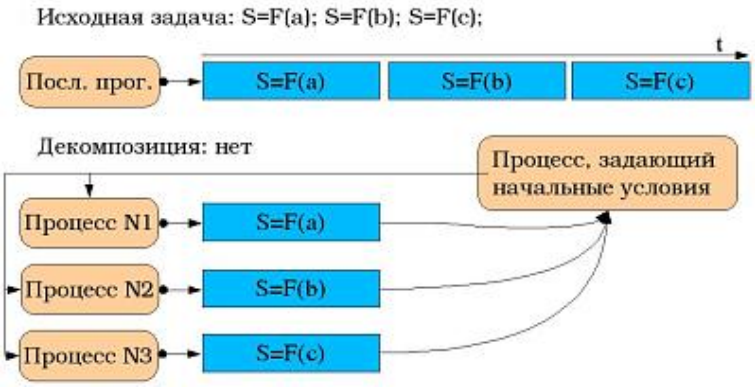

Рисунок 1. Тривиальная декомпозиция

пользовании того или иного типа декомпозици при написании параллельной программы решается исходя из структуры самой задачи. Причем, в зависимости от условий, можно использовать сразу несколько типов.

5.2.1 Тривиальная декомпозиция

Как следует из названия, тривиальная декомпозиция наиболее простой тип декомпозиции. Применяется она в том случае, когда различные копии линейного кода могут исполняться независимо друг от друга и не зависят от результатов, полученых в процессе счета других копий кода. Проиллюстрировать подобный вариант можно на примере решения задачи методом перебора или Монте-Карло. В этом случае одна и та же программа, получив различные начальные параметры, может быть запущена на различных процессорах кластера. Как легко заметить, программирование таких параллельных процессов ничем не отличается обычного программирования на последовательном компьютере, за исключением маленького участка кода, отвечающего за запуск копий программы на процессорах кластера и затем ожидающего окончания счета запущеных программ.

5.2.2 Функциональная декомпозиция

При функциональной декомпозиции исходная задача разбивается на ряд последовательных действий, которые могут быть выполнены на различных узлах кластера не зависимо от промежуточных результатов, но строго последовательно.

Предположим наша задача сводится к применению некоего функционального оператора к большому массиву данных: `S[i]=F(a[i])`. Предположим также, что выполнение функции F над одним элементом массива занимает достаточно большое время и нам хотелось бы это время сократить. В этом случае мы можем попытаться представить исходную функцию как композицию нескольких фунуций: `S(a[i])=I(H(R(a[i]))`. Произведя декомпозицию мы получим систему последовательных задач:

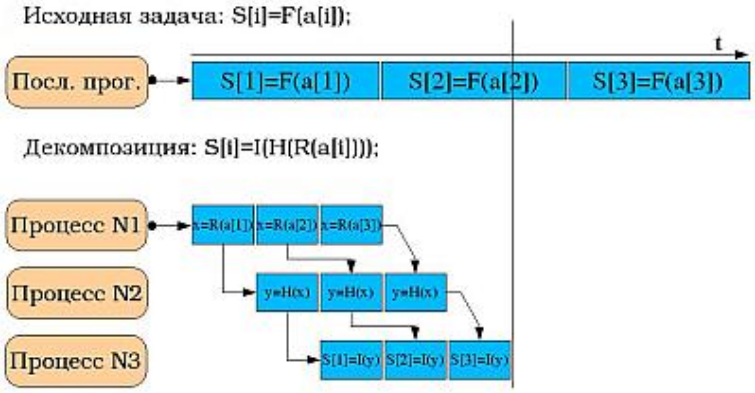

Рисунок 2. Функциональная декомпозиция

```
  x=r(a[i]);
y=h(x);
b[i]=i(y);
```

Каждая из этих задач может быть выполнена на отдельном узле кластера. Как можно заметить общее время обработки одного элемента массива a[i] в результате не изменяется, а скорее немного увеличивается за счет межпроцессорных пересылок. Однако общее время обработки всего массива заметно снижается за счет того, что в нашем примере одновременно идет обработка сразу трех элементов массива.

У данного метода декомпозиции есть пара особенностей, о которых надо помнить.

Первая особенность сосотоит в том, что выход кластера на максимальную эффективность происходит не сразу после запуска задачи, а постепенно, по мере того, как происходит частичная обработка первого элемента массива. Второй и третий процессоры в нашем примере, которые отвечают за выполнение функций `g(x)` и `f(y)`, будут простаивать до тех пор, пока не закончится выполнение функции `h(a[1])` на первом процессоре. Третий процессор будет простаивать до окончания выполнения функции `g(a[1])`. По аналогичному сценарию, только в зеркальном отображении, происходит окончание работы.

Вторая особенность заключается в выборе декомпозированных функций h,g,f. Для уменьшения времени простоя процессоров в ожидании следующей порции работы необходимо таким образом подбирать декомпозированные функции, чтобы время их работы было примерно одинаковым.

По приведенному сценарию данные обрабатываются в режиме конвеера. На программиста, выбравшего функциональный тип декомпозиции задачи, ложится обязанность не только по выбору декомпозированных функций, но и по организации работы параллельных частей программы в режиме конвеера, то есть правильно организовать процедуры

получения исходных данных от предыдущего процесса и передачи обработаных данных следующему процессу.

5.2.3 Декомпозиция данных

В отличие от функциональной декомпозиции, когда между процессорами распределяются различные задачи, декомпозиция данных предполагает выполнение на каждом процессоре одной и той же задачи, но над разными наборами данных. Части данных первоначально распределены между процессорами, которые обрабатывают их, после чего результаты суммируются некоторым образом в одном месте (обычно на консоли кластера). Данные должны быть распределены так, чтобы объем работы для каждого процесора был примерно одинаков, то есть декомпозиция должна быть сбалансированной. В случае дисбаланса эффективность работы кластера может быть снижена.

В случае, когда область данных задачи может быть разбита на отдельные непересекающиеся области, вычисления в которых могут идти независимо, мы имеем регулярную декомпозицию.

5.2.4 Выбор метода декомпозиции

В качестве метода декомпозиции была выбрана тривиальная декомпозия, т.к.:

- Данный метод дает наилучшую эффективность распараллеливания;
- Тривиальная декомпозиция очень хорошо согласуется с выбранным алгоритмом решения СЛАУ.

Кроме того при подсчете нормы вектора погрешности текущего решения используется декомпозиция данных, т. к. результаты от всех процессов собираются в корневом и принимается решение о дальнейшем уточнении результата или останове вычислений.

5.3 Текст программы

Листинг 1. Текст программы решения СЛАУ

```
#include <stdlib.h>
#include <math.h>
#include <mpi.h>
#include "util2.h"

#define ROOT (0)

void jacobi_iter(double *A, double *x, double *x_old, int n_part, int n, int first)
{
```

```
    int i, j;
    double sum;

    for (i=0; i<n_part; i++) {
        sum = 0;
        for (j=0; j<i+first; j++) {
            sum += A[i*(n+1)+j] * x_old[j];
        }

        for (j=i+first+1; j<n; j++) {
            sum += A[i*(n+1)+j] * x_old[j];
        }

        x[i+first] = (A[i*(n+1)+n] - sum) / A[i*(n+1)+i+first];
    }
}

unsigned jacobi_solve(double *A, double *x, double e, int n, int myid, int n_part, int ...
    numprocs)
{
    double *x_old;
    int i, iter = 0, first;
    double d_norm, d_val;
    int *sendcnts, *displs;

    displs = (int*)malloc(numprocs*sizeof(int));
    sendcnts = (int*)malloc(numprocs*sizeof(int));

    MPI_Scan(&n_part, &first, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
    first -= n_part;

    MPI_Allgather(&n_part, 1, MPI_INT, sendcnts, 1, MPI_INT, MPI_COMM_WORLD);
    displs[0] = 0;
    for (i=1; i<numprocs; i++) displs[i] = displs[i-1] + sendcnts[i-1];

    x_old = new_vector(n);

    do {
        iter++;

        memcpy(x_old, x, sizeof(double)*n);

        jacobi_iter(A, x, x_old, n_part, n, first);

// TODO     Gatherv?
```

```
        MPI_Allgatherv(x+first, n_part, MPI_DOUBLE, x, sendcnts, displs, MPI_DOUBLE, ...
            MPI_COMM_WORLD);

        d_norm = 0;
        if (myid == ROOT) {
            for (i=0; i<n; i++) {
                d_val = fabs(x[i] - x_old[i]);
                if (d_norm < d_val) d_norm = d_val;
            }
        }

        MPI_Bcast(&d_norm, 1, MPI_DOUBLE, ROOT, MPI_COMM_WORLD);
    } while (e < d_norm);

    free(x_old);

    return iter;
}

int main(int argc, char **argv)
{
    double *A, *A_part, *x;
    int i, n, n_part, part_size, iter, myid, numprocs;
    double error = 0.1e-10;
    int *sendcnts, *displs;
    double t_start, t_end;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int namelen;


    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Get_processor_name(processor_name, &namelen);

    printf("process %i on %s\n", myid, processor_name);

    scanf("%u", &n);
    MPI_Bcast(&n, 1, MPI_INT, ROOT, MPI_COMM_WORLD);
    MPI_Bcast(&error, 1, MPI_DOUBLE, ROOT, MPI_COMM_WORLD);

    if (myid == ROOT) {
        t_start = MPI_Wtime();
        A = new_matrix(n, n+1);
        read_matrix(A, n, n+1);
```

```
        t_end = MPI_Wtime();
        printf("Time to input matrix: %f sec\n", t_end - t_start);
    }

    x = new_vector(n);
    MPI_Bcast(x, n, MPI_DOUBLE, ROOT, MPI_COMM_WORLD);

    n_part = (n/numprocs) + (n%numprocs > myid ? 1 : 0);
//  printf("process: %i; num of rows: %i\n", myid, n_part);

    A_part = new_matrix(n_part, n+1);

    displs = (int*)malloc(numprocs*sizeof(int));
    sendcnts = (int*)malloc(numprocs*sizeof(int));

    part_size = (n+1) * n_part;
    MPI_Gather(&part_size, 1, MPI_INT, sendcnts, 1, MPI_INT, ROOT, MPI_COMM_WORLD);

    displs[0] = 0;
    for (i=1; i<numprocs; i++) displs[i] = displs[i-1] + sendcnts[i-1];

/*
if (myid == ROOT) {
    printf("#%i, %p: ", myid, A_part); write_matrix(A_part, n_part, n+1);
//  printf("part_size #%i - %i\n", myid, part_size);
    printf("sendcnts #%i: ", myid); for (i=0; i<numprocs; i++) printf("%i ", sendcnts[i])...
        ; printf("\n");
    printf("displs #%i: ", myid); for (i=0; i<numprocs; i++) printf("%i ", displs[i]); ...
        printf("\n");
    printf("A #%i: ", myid); write_matrix(A, n, n+1);
}
*/

    t_start = MPI_Wtime();
    MPI_Scatterv(A, sendcnts, displs, MPI_DOUBLE, A_part, part_size, MPI_DOUBLE, ROOT, ...
        MPI_COMM_WORLD);
//  printf("#%i: ", myid); write_matrix(A_part, n_part, n+1);
    t_end = MPI_Wtime();
    if (myid == ROOT) printf("Time to scatter matrix to all processors: %f sec\n", t_end...
         - t_start);

    t_start = MPI_Wtime();
    iter = jacobi_solve(A_part, x, error, n, myid, n_part, numprocs);
    t_end = MPI_Wtime();
    if (myid == ROOT) printf("Time to solve equation: %f sec\n", t_end - t_start);
```

```
    if (myid == ROOT) {
        printf("iter %i\n", iter);
        printf("ans = \n");
        write_vector(x, n);
    }

    if (myid == ROOT) {
        free_matrix(A);
    }
    free(x);
    free_matrix(A_part);
    free(sendcnts);
    free(displs);

    MPI_Finalize();

    return EXIT_SUCCESS;
}
```

6 Выбор аппаратных средств

Для построения кластера из 4-х узлов были взяты имеющиеся в распоряжении серверные компьютеры на основе архитектуры Intel IA-32, объединенные локальной сетью по технологии Ethernet 100Base-TX с пропускной способностью 100 Мбит/с.

Опишем аппаратную конфигурацию узлов кластера:

storm ЦП: Intel Pentium 4 2.40GHz Память: 512 МБ

proxy ЦП: Intel Xeon 3.06GHz Память: 2 ГБ

host2 Intel Pentium III 1133MHz Память: 512 МБ

ns ЦП: Intel Xeon 2.40GHz Память: 2 ГБ

Основным (Master) узлом является storm, остальные являются дочерними (Slave). Задача запускается оператором с основного узла с помощью специального скрипта `mpirun`, входящего в комплект MPIch. Это скрипт по протоколу ssh удаленно заходит на остальные узлы и выполняет задачу там. После этого одна и та же программа параллельно исполняется на нескольких узлах и использует обмен сообщениями для решения общей задачи. Каждому процессу выделяется его уникальный номер, по которому он может отличать себя от других и выполнять определенные характерные только для него действия. Чаще всего это требуется для корневого (Root) процесса, который производит ввод информации, сбор данных (для вычисления погрешности и выдачи результата) и рассылку

информации остальным процессам.

Способ использования оперативной памяти и виды кэш памяти задаются операционной системой, поверх которой запускается кластер, и архитектурой используемого оборудования. Помимо этого, память на каждом узле кластера своя собственная, не разделяемая с остальными узлами. Таким образом кластер представляет собой MIMD-систему. Для обмена данными между узлами используются встроенные в программу сообщения, которые пересылаются между узлами по обычной сети, построенной на основе стека протоколов TCP/IP. Кэш память, использующаяся в системе задействована на этапах ввода матрицы и вычислений. На этапе ввода матрицы используется кэширование операционной системой считанных с диска данных в оперативной памяти компьютера. На этапе вычислений используется внутренняя кэш-память процессора, которая позволяет совершать меньше обращений к оперативной памяти в процессе вычисления. Соответственно поддержание когерентности кэш памяти производится в первом случае операционной системой, а во втором процессором. В обоих случаях могут использоваться различные способы и они являются скрытыми от программиста и пользователя.

Так как в разработанной системе отсутствует разделяемая память, то требуются некоторые способы передачи данных между процессорами. Эти способы регламентируются стандартом MPI, в котором перечислены различные подпрограммы, которые программист может использовать с этой целью. Низкоуровневая реализация обмена сообщениями не регламентируется в стандарте и оговаривается для конкретных реализаций, которые могут быть приспособлены для выполнения в различных программно-аппаратных средах (MS Windows NT, Unix, с использованием TCP/IP, с использованием SCI, на процессорах Intel, на процессорах MIPS и т. д.)

В выбранной мной в данном курсовом проекте реализации MPI — MPIch, — обмен данными между процессорами производится с помощью обмена сообщениями по сети, на основе протоколов TCP/IP. Канальный уровень передачи сообщений реализован в виде сети стандарта Ethernet 100Base-TX, позволяющей передавать данные со скоростью до 100 Мбит/с.

Схема организации блоков и связей представлена на плакате.

7 Экспериментальное исследование программы

7.1 Тестирование программы

Для проверки правильности работы программы было проведено решение трех примеров и сопоставление полученных результатов с результатами, полученными для таких же матриц в математическом пакете Matlab.

Кроме того были проверены результаты работы программы на матрицах более высокой размерности ($1000 \times 1000, \ldots, 5000 \times 5000$). Они также совпадают с полученными в пакете Matlab, но здесь не приводятся, т.к. занимают довольно большой объем.

Далее приводятся результаты тестирования:

7.1.1 Тест 1

Листинг 2. Matlab

```
>> a

a =
    4.9501    0.8913    0.8214    0.9218
    0.2311    4.7621    0.4447    0.7382
    0.6068    0.4565    4.6154    0.1763
    0.4860    0.0185    0.7919    4.4057

>> b'

ans =
    0.9355    0.9169    0.4103    0.8936

>> (a\b)'

ans =
    0.1193    0.1541    0.0511    0.1799
```

Листинг 3. Программа

```
storm# mpirun -np 2 -stdin test1.in ./jacobi
Authentication successful.
process 0 on storm
Time to input matrix: 0.000024 sec
Time to scatter matrix to all processors: 0.000024 sec
Time to solve equation: 0.047424 sec
iter 24
ans =
  0.119269   0.154103   0.051106   0.179838
```

```
process 1 on proxy
```

7.1.2 Тест 2

Листинг 4. Matlab

```
>> [a b]
ans =

    5.0579    0.2028    0.0153    0.4186    0.8381    0.5028
    0.3529    5.1987    0.7468    0.8462    0.0196    0.7095
    0.8132    0.6038    5.4451    0.5252    0.6813    0.4289
    0.0099    0.2722    0.9318    5.2026    0.3795    0.3046
    0.1389    0.1988    0.4660    0.6721    5.8318    0.1897

>> (a\b)'
ans =

    0.0881    0.1168    0.0463    0.0427    0.0178
```

Листинг 5. Программа

```
storm# mpirun -np 2 -stdin test2.in ./jacobi
Authentication successful.
process 0 on storm
Time to input matrix: 0.000037 sec
Time to scatter matrix to all processors: 0.000058 sec
Time to solve equation: 0.056563 sec
iter 23
ans =
  0.088098   0.116831   0.046309   0.042673   0.017829
process 1 on proxy
```

7.1.3 Тест 3

Листинг 6. Matlab

```
>> [a b]

ans =
    6.1934    0.3784    0.8216    0.3412    0.3704    0.7948    0.2714
    0.6822    6.8600    0.6449    0.5341    0.7027    0.9568    0.2523
    0.3028    0.8537    6.8180    0.7271    0.5466    0.5226    0.8757
    0.5417    0.5936    0.6602    6.3093    0.4449    0.8801    0.7373
    0.1509    0.4966    0.3420    0.8385    6.6946    0.1730    0.1365
    0.6979    0.8998    0.2897    0.5681    0.6213    6.9797    0.0118

>> (a\b)'
```

```
ans =
    0.0238    0.0178    0.1154    0.1033    0.0001   -0.0162
```

Листинг 7. Программа

```
storm# mpirun -np 2 -stdin test3.in ./jacobi
Authentication successful.
process 0 on storm
Time to input matrix: 0.000035 sec
Time to scatter matrix to all processors: 0.000052 sec
Time to solve equation: 0.098751 sec
iter 29
ans =
  0.023809   0.017766   0.115370   0.103321   0.000119  -0.016189
process 1 on proxy
```

Таким образом, по результатам тестирования, можно сделать вывод, что программа работает правильно на выбранных случайных наборах коэфициентов.

7.2 Исследование программы

Цель экспериментального исследования программы — запустить программу на кластере и исследовать зависимости времени выполнения от параметров: количество одновременно запущеных процессов (от 1 до 4), размерность матрицы (от 1000×1000 до 5000×5000).

Для эксперимента было подготовлено 5 случайно сгенерированных в пакете Matlab матриц коэфициентов. Тестирование проводилось в ночное время — время минимальной загрузки серверов. Для проведения тестирования был написан пакетный файл на языке командной оболочки Unix.

Листинг 8. run_all.sh

```
#!/bin/sh

for j in 1 2 3 4 5
do
        for i in 1 2 3 4
        do
                echo `date` ": run on $i processors ${j}000x${j}000 matrix"
                echo "./run_4_tests.sh $i ../matr_${j}000.in"
                ./run_4_tests.sh $i ../matr_${j}000.in
        done
done
```

Листинг 9. run_4_tests.sh

```
#!/bin/sh
```

Таблица 1. Время считывания матрицы

	1000	2000	3000	4000	5000
1	1.63141675	8.29959375	29.81600225	50.604064	57.60872775
2	1.052793	4.187841	20.26942625	42.0447345	58.211761
3	1.18045925	4.690126	21.52936	40.4394535	55.86067675
4	1.14388275	4.76836	30.81043775	38.23159825	53.91873275

Таблица 2. Время рассылки матрицы процессам

	1000	2000	3000	4000	5000
1	0.02917875	1.34563725	2.46606525	0.7835735	17.0997205
2	0.6252465	2.61815625	5.31395075	9.31598425	13.23164825
3	0.80715175	3.207169	6.361523	10.6709245	15.28715225
4	0.979049	4.05991325	9.390454	12.70651625	18.72131275

```
np=$1
stdin=$2

for i in 1 2 3 4
do
        echo "run #$i"
        echo "/usr/local/bin/mpirun -np $np -stdin $stdin ./jacobi"
        /usr/local/bin/mpirun -np $np -stdin $stdin ./jacobi |grep Time
done
```

Первый скрипт запускает второй скрипт для различных сочетаний количества процессов и размерности матрицы. Второй скрипт запускает программу вычисления СЛАУ с заданными параметрами четыре раза, для дальнейшего усреднения полученного результата.

Измерения времени выполнения частей программы производились с помощью стандартной функции MPI `MPI_Wtime()`, предоставляющей высокоточный счетчик времени, возвращающий время в виде дробного числа секунд.

Затем полученные результаты были просмотрены и запуски программы, время которых значительно выделялось из общей закономерности, были повторены. После этого результаты были обработаны и сведены в общие таблицы.

Графики зависимостей, построенные по полученным таблицам, представлены на плакате. По полученным результатам можно сделать следующие выводы:

- Ввиду большого размера матриц коэфициентов и того, что они хранились на сетевом диске, их ввод в память программы занимает довольно большое время. Необходимость использования сетевого диска была обусловлена тем, что среда MPI требует,

Таблица 3. Время решения СЛАУ

	1000	2000	3000	4000	5000
1	0.61992425	2.376641	4.80698275	7.7232665	9.1183235
2	0.29486875	1.102585	2.0276365	3.4203035	5.34110275
3	0.267597	0.84580775	1.666024	2.40782975	3.2665045
4	0.243515	0.798622	1.71639975	2.209172	3.6425425

Таблица 4. Ускорение решения

	1000	2000	3000	4000	5000
1	1	1	1	1	1
2	2.102373514	2.155517262	2.370732007	2.258064672	1.707198668
3	2.316633781	2.809906861	2.885302223	3.207563367	2.791462097
4	2.545733322	2.975927285	2.800619582	3.496000538	2.503285411

чтобы на всех узлах кластера файловая система была синхронизирована.

- Можно заметить, что в последовательности нескольких экспериментов с одной и той же матрицей, первое считывание занимает больше времени, чем последующие. Это связано с алгоритмами операционной системы, кэширующими и оптимизирующими операции ввода-вывода.
- Время считывания матрицы не зависит от количества процессоров, т. к. считывание производится только в одном главном процессе. Время зависит только от размера считываемого файла. Зависимость близка к линейной.
- Время рассылки частей матрицы процессам для последующей обработки, увеличивается как с увеличением размерности матрицы, так и с увеличением числа процессов.
- Непосредственно время решения системы сравнительно мало относительно вспомогательных операций ввода-вывода. Это обусловлено спецификой задачи и тем, что арифметические операции в процессоре выполняются гораздо быстрее, чем операции ввода-вывода и пересылки по сети.
- Время решения системы увеличивается с увеличением размерности матрицы. Сложность алгоритма метода Якоби $O(n^2)$, что подтверждается и результатами выполнения эксперимента.
- При увеличении количества процессов время выполнения программы снижается. Данный алгоритм очень хорошо распараллеливается, поэтому в наиболее удачных случаях видно, как скорость вычисления решений СЛАУ пропорционально соответ-

ствует количеству процессов. Однако данное соответствие не точное в силу закона Амдала и погрешностей измерения, вызванных нестабильной загрузкой узлов системы побочными задачами.

8 Заключение

В ходе выполнения данной работы была установлена кластерная вычислительная система из 4 узлов, на основе технологии MPI, и написана программа для этой системы, вычисляющая корни системы линейных алгебраических уравнений.

Выполнение работы и исследование результатов эксперимента показало, что задача построения кластера не является очень сложной и вполне по силам обычным студентам. Однако это не значит, что в этой области отсутствуют объекты для изучения. Достаточно сложными являются вопросы нахождения эффективных алгоритмов для распараллеливания задач, оптимальной настройки вычислительной среды для получения максимальных результатов и т. д.

Список литературы

[1] Танненбаум Э. С. Архитектура компьютера. 4-е изд. Питер: 2005. 704 с.

[2] Цилькер Б. Я., Орлов С. А. Организация ЭВМ и систем: учебник для вузов. Питер: 2005. 672 с.

[3] Танненбаум Э. С., Ван Стеен М. Распределенные системы. Принципы и парадигмы. Питер: 2003. 880 с.

[4] MathWorld. (*http://mathworld.wolfram.com/*, 25.10.2005)

[5] Приклонский В. И. Численные методы: программа лекционного курса. (*http://fiziki.uniyar.ac.ru/educate/lectures/digital.html*, 25.10.2005)

[6] Шпаковский Г. И., Серикова Н. В. Программирование для многопроцессорных систем в стандарте MPI. Минск, БГУ: 2002.