СОДЕРЖАНИЕ

Введение 3

ЧАСТЬ 1. ИЗ ЧЕГО СОСТОИТ СУПЕРКОМПЬЮТЕР 13
Глава 1. Что такое суперкомпьютер? 14
Глава 2. Аппаратное обеспечение 27
Глава 3. Программное обеспечение 40
Глава 4. Управление ресурсами кластера 68

ЧАСТЬ 2. ПРИМЕНЕНИЕ СУПЕРКОМПЬЮТЕРА 81
Глава 5. Технологии и парадигмы параллельного программирования 82

ЧАСТЬ 3. СТРОИМ ПРОСТЕЙШИЙ КЛАСТЕР 127
Глава 6. Выбор рабочей станции – однопроцессорный и многопроцессорный вариант 128
Глава 7. Выбор трансляторов 132
Глава 8. Выбор коммуникационного компонента 134
Глава 9. Суп из топора. Как построить кластер, если денег на аппаратуру нет вообще 154
Глава 10. Системы управления ресурсами (краткий обзор) 166
Глава 11. Суперкомпьютеры МВС-1000 – история и обзор 174
Глава 12. Что дальше? 182

ПРИЛОЖЕНИЯ 187
Приложение 1. Инструкция прикладному программисту по коммуникационной среде TCP Router для МВС-1000/16 188
Приложение 2. CDA – реализация распределенных массивов с теневыми гранями в ScaLAPACK 198
Приложение 3. Текст нагрузочного теста mpitnet 221
Приложение 4. Нагрузочный тест NFS 221
Приложение 5. Асинхронный вариант MPICH и LAM 222

Литература 226

Предметный указатель 229

Введение

Такие названия, как «суперкомпьютер», «параллельная система», «кластер» последнее время употребляются часто, что не избавляет нас от необходимости с самого начала строго определить соответствующие понятия. Без этого мы не сможем даже обозначить круг читателей, которым адресована книга. Вдобавок, смысл привычных понятий со временем часто меняется. Так, например, автор пару лет назад с удивлением узнал, что занимается он, оказывается, «специализированными» (то есть не универсальными) вычислительными системами.

В самом деле, кому сейчас придет в голову назвать «универсальным» компьютер, с которого нельзя ни письма отправить, ни «посидеть в чате», ни распечатать бухгалтерскую ведомость? Еще 20 лет назад общепринятое представление об универсальном компьютере было другим. При этом и тогда, и сейчас вряд ли кому-нибудь пришло бы в голову начинать книгу для профессионалов компьютерной отрасли объяснением, что такое универсальный компьютер. Также совсем недавно довелось встретить студента профильной специальности, который искренне полагал, что «суперкомпьютер» – это «такой хороший компьютер, ну просто супер», то есть не знал, что данное слово является термином. Итак, приступим к возможно более строгому определению смысла слов, с которыми нам предстоит далее сталкиваться постоянно. Первое упоминание определяемого термина будем при этом выделять ***жирным курсивом***.

Общеизвестно, что набор строгих формальных определений, как правило, довольно плохо запоминается и понимается, особенно, если этих определений около десятка и более – а ведь нам придется сейчас определить именно такое количество базовых понятий, чтобы потом легко ими оперировать. Для этого совместим определение основных терминов с кратким историческим обзором: когда и почему появлялось то или иное понятие.

Суперкомпьютером мы будем называть вычислительную установку мелкосерийного или штучного выпуска, многократно превосходящую по вычислительной мощности массово выпускаемые компьютеры. Таким образом, под суперкомпьютером мы не будем понимать машину, быстродействие которой «не ниже X миллиардов (или триллионов, или чего-то еще) операций в секунду». Конкретная, численно выражаемая грань «уже суперкомпьютерного» быстродействия, в действительности, довольно размыта и к тому же быстро смещается по мере непрерывного прогресса в области производства процессоров. Отличительных признаков определяемого таким образом суперкомпьютера, на наш взгляд, два:

- это **не** изделие действительно массового выпуска, а следовательно, и при его изготовлении, и при его применении используются совсем не массовые технологии, более дорогие и, возможно, менее удобные, чем «общепринятый ширпотреб»,
- это машина, ориентированная на **вычисления**, на заметное, хотя бы на порядок, снижение времени выполнения сложных расчетов, по сравнению с обычными компьютерами.

Понятие суперкомпьютера появилось, фактически, одновременно с началом серийного выпуска компьютеров. Отражает оно вполне понятное желание применить для решения особо сложных вычислительных задач те технологии, которые, по тем или иным причинам, не стали «конвейерными». На протяжении первых примерно 30 лет с момента появления компьютеров речь шла почти исключительно о технологиях производства процессоров. Более простые, лучше освоенные и менее дорогие технологии применялись при серийном выпуске, в то время как для «избранных» задач строились «избранные» процессоры – гораздо более быстрые, но и более дорогие в производстве и «капризные» в эксплуатации. Таким образом, примерно до середины 80-х можно было с высокой степенью точности утверждать, что суперкомпьютер – это компьютер, оснащенный специальным, особо мощным (и потому – особо дорогим) процессором.

Ситуация радикально изменилась буквально за 2–3 года, когда прогресс микроэлектронных технологий позволил преодолеть рубеж, условно говоря, миллиона транзисторов на одном кристалле, что, очень грубо, соответствует уровню сложности процессора i486. Стало ясно, что технологиям изготовления процессоров из отдельных элементов пришел конец – отныне все процессоры будут только однокристальными микропроцессорами. В 1989 году фирма Intel выпустила однокристальный микропроцессор i860, сравнявшийся по производительности с действовавшим тогда же суперкомпьютером Cray-1 [2]. Поскольку производство микросхем, по природе своей, является конвейерным и довольно дешевым, преодоление этого рубежа означало конец эры особо мощных процессоров. Просто любую мыслимую технологию повышения производительности процессора стало возможным реализовать в кристалле и довести до крупносерийного выпуска за считанные месяцы. Процессоры, грубо говоря, перестали делиться на «обычные» и «особо мощные».

В этих условиях для повышения быстродействия сверх «массово доступного» уровня, то есть для создания суперкомпьютера, остался единственный путь – путь объединения многих процессоров для параллельного решения одной особо сложной задачи. С этого времени и до наших дней мы можем, таким образом, уверенно ставить знак равенства между **суперком-**

пьютером и ***параллельной (или многопроцессорной) вычислительной системой*** [1].

Можно отметить, как минимум, два интересных для нас результата этой «первой суперкомпьютерной революции».

- во-первых, ключевой технологией в **разработке** суперкомпьютеров стала (вместо технологии разработки процессора) технология комплексирования многих процессоров в единую систему,
- во-вторых, суперкомпьютер стал архитектурно отличаться от традиционных машин массового выпуска гораздо сильнее, чем раньше, а это уже не могло не отразиться на технологиях **применения**. Для того, чтобы параллельная машина дала скоростной выигрыш, надо написать параллельную программу, а это совсем не то же самое, что написать программу для традиционной машины.

Примерно до середины 90-х развитие многопроцессорных суперкомпьютеров шло по двум параллельным направлениям.

Магистральное направление было связано с построением из микросхем массового выпуска специализированных многопроцессорных систем. Главным вопросом при построении таких систем был вопрос о способе комплексирования процессоров.

Наиболее удобным для программиста был бы, конечно, способ объединения многих процессоров на единой, симметрично адресуемой **общей памяти**. Такие компьютеры получили название ***SMP-систем*** (SMP – Symmetric Multi Processing). Главное достоинство SMP-систем с точки зрения программиста состоит в том, что для организации параллельной работы требуется поделить между процессорами лишь вычисления, но не данные: данные и так принадлежат «всем на равных». К сожалению, этот самый удобный для программиста способ является, одновременно, и самым дорогим в аппаратной реализации. Процессор общается с собственной памятью весьма интенсивно, и для того, чтобы память могла обслуживать потребности многих процессоров, она должна быть либо очень сложной и дорогой, либо довольно медленной с точки зрения каждого из процессоров (как правило, имеют место обе эти особенности). Когда число процессоров вырастает до десятков, задача вообще становится почти неразрешимой технически. Таким образом, область применения SMP-подхода – дорогие, не очень большие по числу узлов системы, довольно удобные для программирования.

Менее удобным для программиста, но на порядок более дешевым, допускающим потенциально бесконечное масштабирование, оказался подход на основе объединения отдельных, самостоятельных ЭВМ специализированными каналами связи. Говоря об отдельных ЭВМ, мы имеем в виду не

конструкцию, а лишь логику построения системы. В действительности, конечно же, использовались не ЭВМ общего назначения, с отдельным экраном, клавиатурой и блоком питания, а специализированные вычислительные модули, компактные, оснащенные не применяемой в системах общего назначения специализированной связной аппаратурой. Модули эти разрабатывались специально для использования в качестве деталей конкретного суперкомпьютера и, зачастую, ни в каком другом качестве использованы быть не могли. При этом логически каждый такой модуль представлял из себя полноценную ЭВМ, то есть имел в своем составе отдельный процессор, отдельную память и выполнял совершенно отдельную, независимую от других программу, оперирующую своими локальными данными. Для организации взаимодействия модулей между собой в каждой такой программе следовало явно закодировать акты передачи данных по каналам связи.

Такие системы, в отличие от SMP, получили название ***MPP*** (MPP – Massively Parallel Processing, название подчеркивает теоретически неограниченную масштабируемость систем этого класса) [1, 3]. Наиболее логически завершенного вида концепция систем MPP достигла в **транспьютерных системах**, к сожалению, не получивших впоследствии дальнейшего развития [8, 9].

Промежуточное положение между системами SMP и MPP занимают системы ***NUMA*** (NUMA – Non-uniform Memory Access). Это системы, в которых общая для многих процессоров память присутствует (аппаратно или логически), но по своим архитектурным свойствам (быстродействию, прежде всего) столь сильно отличается от локальной памяти процессора, что это необходимо явно учитывать при программировании. И по цене, и по стилю программирования эти системы ближе к SMP, чем к MPP [1].

Второе направление развития параллельных суперкомпьютеров, не связанное с разработкой специализированных MPP-систем, на рассматриваемом нами этапе не было основным и не подразумевало каких-либо разработок в области аппаратуры вообще. Логически MPP-система мало отличается от обычной локальной сети, а локальные сети в описываемый период переживали довольно бурное развитие. Возникало вполне естественное желание попытаться использовать локальную сеть как параллельный суперкомпьютер – например, в периоды ночных простоев. Все построение суперкомпьютера при этой технологии сводится к разработке соответствующего программного обеспечения.

Такие локальные сети, используемые в качестве многопроцессорных вычислительных систем, стали называть ***кластерами рабочих станций*** или просто «кластерами». Когда-то уровень развития аппаратуры локальных сетей был довольно низок. По этой причине кластеры сильно проигрывали «на-

стоящим, железным» суперкомпьютерам в своих коммуникационных возможностях. На практике это означало, что класс успешно решаемых на кластерах задач был гораздо уже, чем на «настоящих» MPP. В самом деле, чем теснее способны взаимодействовать узлы многопроцессорной системы, тем больше шансов на успех имеет программист, пытающийся наладить одновременную, совместную работу нескольких процессоров над одной задачей.

Во второй половине 90-х грянула «вторая суперкомпьютерная революция», связанная, в первую очередь, с двумя достижениями в области аппаратуры крупносерийного выпуска. Повсеместное оснащение персональных компьютеров высокоскоростной шиной PCI [12], с одной стороны, и переход на дешевую, но довольно быструю (причем коммутируемую) сеть Fast Ethernet, с другой, привели к тому, что кластеры «догнали» специализированные MPP-системы по своим коммуникационным возможностям. При этом среди производителей специализированного коммуникационного оборудования, такого, которое раньше применялось только в составе MPP-систем, стало принято оформлять свои изделия в виде стандартных плат PCI, устанавливаемых в компьютеры общего назначения, а не в виде наборов микросхем, из которых требовалось изготавливать специализированные модули.

В этих условиях изготовление MPP-систем из специализированных, только для этой цели спроектированных и изготовленных, модулей потеряло всякий смысл. За счет некоторых потерь в плотности компоновки стало возможно изготовить MPP-систему не из специальных модулей, а из стандартных персональных компьютеров, купленных в магазине и при необходимости оснастить эти «модули из магазина» очень высокопроизводительной, специализированной коммуникационной средой, так как соответствующие устройства тоже можно было приобрести. При этом можно было быть совершенно уверенным, что, если только такое изделие существует, по формату оно является обычной PCI-платой, которую можно вставить в стандартный персональный компьютер.

Разница между MPP и кластером, таким образом, из области аппаратной сместилась в область чисто организационную [13].

Локальные сети, специально собранные для использования в качестве многопроцессорной вычислительной системы, компактно размещенные в одном или нескольких шкафах, стали называть ***кластерами выделенных (dedicated) рабочих станций***, подчеркивая этим названием, что узлы такой сети **выделены** для работы в составе суперкомпьютера и не используются для расчетов зарплаты, деловой переписки или компьютерных игр.

В то же время, кластеры в старом смысле этого слова не прекратили своего существования, только теперь их стали называть ***кластерами невыделенных рабочих станций***.

В отличие от традиционных (невыделенных) кластеров, кластеры выделенных рабочих станций стали оснащать программным обеспечением, ориентированным исключительно на управление установкой как единым целым: так, на грамотно организованном выделенном кластере отдельный узел обычно не имеет адреса в Интернете и не может использоваться как самостоятельный компьютер общего назначения. Это позволяет нам говорить о кластерах выделенных рабочих станций как о полноценных MPP-системах, отличающихся от MPP-систем предыдущего поколения лишь аппаратной архитектурой, но не порядком использования.

В кластерах невыделенных рабочих станций, напротив, программное обеспечение ориентировано на использование набора совершенно самостоятельных машин, при необходимости, не только по прямому назначению, но еще и в качестве вычислительных узлов параллельной вычислительной системы. На развитии систем SMP и NUMA вторая суперкомпьютерная революция практически не отразилась, эти системы продолжали свое развитие в классе очень дорогих и, как правило, не очень больших машин. MPP-системы из специализированных модулей практически исчезли в новых разработках, уступив место кластерам выделенных рабочих станций. Впрочем, многие ранее построенные MPP-системы, тем не менее, продолжают эксплуатироваться [1, 3, 13, 14, 20, 24].

Наконец, ценовой разрыв между системами на базе специализированной аппаратуры, с одной стороны, и кластерами выделенных рабочих станций, с другой, составляет, при сопоставимом числе процессоров, примерно два порядка и продолжает нарастать. Этим обусловлено повсеместное (и нарастающее) распространение кластеров выделенных рабочих станций в качестве единственной на сегодня технологии построения сравнительно дешевых и доступных суперкомпьютеров.

Сложившаяся в ходе этой «второй суперкомпьютерной революции» структура мирового парка суперкомпьютеров, существующая сегодня, судя по всему, сохранится в обозримом будущем.

Таким образом, слова «кластер», «параллельная система» и «суперкомпьютер» в значительной степени являются синонимами, но не «по определению», а в силу того цикла технологической эволюции аппаратуры, который мы кратко рассмотрели. Следует отметить, что словом «кластер» принято обозначать, помимо параллельной вычислительной системы, вообще всякое образование из нескольких компьютеров, объединенных для решения единой задачи. Например, если сервер системы резервирования авиа-

билетов реализован на базе локальной сети, с целью повышения производительности и отказоустойчивости, то это – кластер. Далее в настоящей работе мы будем, если явно не оговорено противное, понимать под словом «кластер» именно параллельную вычислительную систему, построенную по кластерной технологии, а не всякий кластер вообще.

Из всего сказанного выше достаточно полно вырисовывается представление о **круге тех читателей**, кому могла бы быть интересна эта книга.

Прежде всего, суперкомпьютер – это компьютер в изначальном, «древнем» смысле этого слова, то есть машина для массовых, трудоемких **вычислений**. Одна из первых популярных журнальных статей о параллельных системах, помнится, начиналась с фразы: «Если ваше представление о необходимом быстродействии не простирается далее удвоенного быстродействия самого лучшего «Пентиума» из магазина, остальную часть статьи можете не читать». Хотя быстродействие «Пентиума из магазина» выросло с тех пор, пожалуй, раз в 10, суть утверждения осталась вполне справедливой. Второе, на чем хотелось бы особо заострить внимание – это довольно существенная трудоемкость освоения суперкомпьютера по сравнению с традиционной машиной.

Мы покажем, что методов автоматического превращения программ для традиционных компьютеров в программы для параллельных машин, дающие при исполнении скоростной выигрыш, не просто не существует, а не может существовать в принципе, по крайней мере, в обозримом будущем. Следовательно, решившийся на освоение суперкомпьютера пользователь должен быть морально готов к тому, что придется учиться, тратить время и силы. Это относится даже к тем пользователям, которые не пишут программ сами, а занимаются исключительно эксплуатацией готовых прикладных программ, купленных «под ключ» – хотя, конечно, в заметно меньшей степени, чем к программистам.

Таким образом, эта книга для тех, кому надо **много и быстро считать**, причем «надо» настолько сильно, что они готовы вложить в ускорение счета заметный объем рабочего времени и собственных сил. Под массовыми вычислениями мы понимаем, в первую очередь, инженерно-технические расчеты (гидродинамические, прочностные – словом, численное решение задач математической физики во всех видах и вариантах). Кроме того, суперкомпьютеры применяются (и довольно широко) в вычислительно трудоемких задачах обработки данных нечислового характера, например, в приложениях из области data mining, но это – отдельная тема, которой мы в этой книге касаться не будем. Читатель этой книги – скорее всего, инженер-математик: из тех, кого математики называют «программистами», а системные программисты – «математиками».

Книга не предполагает каких-либо углубленных, узкопрофессиональных знаний в области системного программирования или организации вычислений. Однако и учебником по программистскому «ликбезу» она безусловно не является. Предполагается, что читатель знаком с обликом традиционных вычислительных машин на уровне грамотного прикладного программиста. Изложение адресовано, в первую очередь, именно программистам, хотя будет полезно и тем, кто пользуется готовыми прикладными программами для кластеров. Специфика рассматриваемых параллельных систем такова, что даже для успешного использования готовых прикладных систем «под ключ» также требуется некоторая степень понимания того, как работает машина.

Конечно, книга в значительной своей части адресована системным администраторам и системным программистам параллельных систем. Впрочем, прикладным программистам, использующим небольшие параллельные системы, зачастую приходится администрировать их самим. С другой стороны, опыт показывает, что нет совершенно ничего сверхъестественного не только в том, чтобы самому администрировать кластер, но и в том, чтобы при необходимости самому его построить.

Здесь уместно было бы остановиться на одном широко распространенном среди новичков суперкомпьютерного сообщества заблуждении. Речь идет о расхожем мнении, что теперь, с наступлением «кластерного» этапа в суперкомпьютерных технологиях, для изготовления суперкомпьютера достаточно решимости сколотить прочный стеллаж, вмещающий необходимое число системных блоков. К сожалению, это не просто не совсем верно – это **совсем не верно**. В полном соответствии с диалектическим принципом перехода количества в качество суперкомпьютер не есть простая сумма составляющих его готовых компонентов. Тем опытным системным администраторам, которые не раз монтировали локальные сети (и никогда – кластеры) и которые уже готовы возмутиться по поводу последнего утверждения, особенно рекомендуется прочитать эту книгу до конца.

Хотелось бы также остановиться на стиле изложения, принятом в этой книге. Эта книга не является (и не стремиться стать) «самозамкнутым» сборником рецептов, который необходим и достаточен для получения практического результата. Автор предполагает, что читатель бегло читает по-английски и уверенно находит в Интернете ответы на любые конкретные, частные вопросы. Цель настоящего изложения – систематично изложить, по возможности, весь спектр таких вопросов, особо обращая внимание на взаимосвязи и взаимозависимости. При этом отнюдь не ставится цель освещения каждого из таких частных вопросов – даже «в сокращенном виде».

Приведем конкретный пример. Основным инструментом параллельного программирования является библиотека MPI [6, 7]. Конечно, мы подробнейшим образом остановимся как на ее принципиальных возможностях, так и на особенностях доступных реализаций. Но списка подпрограмм MPI с пояснениями, что означает каждый из формальных параметров, в этой книге Вы не найдете. Их следует искать в Интернете. Также тщетно было бы ожидать рекомендаций по конкретным маркам оборудования, за исключением случаев, когда некоторая сетевая технология представлена на рынке единственным производителем соответствующих изделий.

Несколько слов о рекомендуемом порядке поиска дополнительной информации по конкретным темам, затрагиваемым в данной книге. Отсылка в Интернет вместо рекомендации прочитать тот или иной конкретный учебник – не отписка недобросовестного автора, а всего лишь констатация двух экспериментальных фактов.

Во-первых, сам автор по мере накопления тех знаний, которыми он рискнул в итоге поделиться с читателями, «бумажными» учебниками практически не пользовался. Во всяком случае, для самого поиска «бумажного» учебника Интернет – не самое плохое место.

Во-вторых, излагаемые в этой книге сведения по природе своей очень динамичны. Не только изданные традиционным способом книги, но и сетевые источники знаний в рассматриваемой области стремительно стареют. Даже замена традиционных ссылок на «бумажные» издания ссылками на тот или иной сетевой ресурс не решает проблемы – сетевые ресурсы могут появляться и исчезать. По этой причине в конце каждого из разделов приводятся как ссылки на относящиеся к затронутой теме сетевые ресурсы, которые автор полагает более или менее стабильными, так и (что более важно) поисковые фразы, по которым следует искать в Интернете информацию на затронутую тему. Предполагается, что читатель будет использовать автоматически индексирующие Сеть поисковые машины, например, Google или Yandex, а не сайты с занесением ссылок по требованию, вроде Yahoo.

Среди всех рекомендуемых интернет-ресурсов следует, безусловно, выделить один, который рекомендуется практически по всем рассматриваемым темам, и сравнимых с которым по информативности автор не знает. Это, конечно, портал **www.parallel.ru**, в особенности – раздел «Технологии».

Автор полностью отдает себе отчет в том, что такой «не вполне рецептурный» стиль подачи материала может кого-то раздражать. Конечно, было бы гораздо удобнее раскрыть книгу на соответствующей странице, четко выполнить по пунктам все рекомендации и получить работоспособный суперкомпьютер. Но, к сожалению, таким способом можно только установить

новый драйвер звуковой карты в Windows. Суперкомпьютер – это не массовая технология, и тем, кто рассчитывает воспользоваться параллельной системой, не понимая, как она работает («пусть оно распараллелится хоть как-нибудь»), настоящая книга не рекомендуется. Таким пользователям может помочь, до некоторой степени, поставленная «под ключ» система класса SMP (примерная стоимость – 100 тыс. долларов на узел).

Подводя итог, обозначим круг предполагаемых читателей этой книги как **всех тех, кому интересно или необходимо понять, как и почему строятся и работают** параллельные системы.

Рекомендуемые поисковые фразы:

`supercomputer taxonomy`

ЧАСТЬ I

Из чего состоит суперкомпьютер

Глава 1. Что такое суперкомпьютер?

Рассказывая о таком многогранном явлении, как суперкомпьютер, всякий раз испытываешь серьезные трудности, продумывая порядок изложения. Наиболее привлекательный для автора – в силу своей простоты – способ заключается в индуктивном представлении материала. В самом деле, как бы много или мало ни знал читатель о существе рассматриваемой проблематики, очевидно, для него не является новостью то, что большинство современных параллельных вычислительных систем – это кластеры рабочих станций. Раз кластеры, значит, строятся они по технологии системной интеграции, из готовых аппаратных компонентов общего пользования и, преимущественно, из готовых же компонент программных. Число этих компонентов конечно. Изучим их все по очереди, с некоторой степенью конкретности, в последней главе – проинтегрируем и все поймем. Более всего подкупает в этом способе его конкретность и, образно говоря, «рецептурность»: всякий раз, говоря о чем-то новом, мы опираемся на твердый фундамент понятий, изученных ранее. Изложение очень конкретно – никаких общих схем, никаких оставленных на потом «темных мест».

Не менее силен, пожалуй, соблазн академического стиля, который правильнее было бы назвать дедуктивным, то есть строго следующим схеме «сверху вниз». По этой схеме, в первой главе суперкомпьютер определяется как «упорядоченный набор множеств устройств, осуществляющих переработку данных, с одной стороны, и их передачу – с другой…» – очевидно, многим приходилось сталкиваться и с образцами такого стиля, согласно которому умопомрачительное количество общих схем вкладывается друг в друга до бесконечности, и лишь в последних главах вдруг становится понятно, что тот самый «X», о котором мы «сформулировали ряд лемм», – это просто сетевая карточка.

Оба способа весьма эффективны – в том смысле, что каждый из них, несомненно, достигает двух следующих эффектов:

- при беглом знакомстве с текстом у читателя остается самое искреннее высокое мнение о солидности профессиональной подготовки автора (нередко соответствующее действительности),
- при подробном знакомстве с текстом, особенно с его частями, повествующими о более или менее известных читателю конкретных проблемах, достигается эффект стойкого отвращения.

Последнее вполне объяснимо. В случае строго восходящего стиля изложения, мы до самого конца так и не узнаем, зачем нам изучать тот или иной конкретный механизм. Возникает необозримая мешанина конкретных сведений, слить которые в общую схему не легче, чем понять основы программирования, исходя из знания вольт-амперной характеристики транзистора (а ведь

кроме транзисторов и конденсаторов в компьютере, как известно, ничего существенного нет). В случае же изложения строго нисходящего читателю, скорее всего, просто не удастся преодолеть желание уснуть задолго до того, как тугая спираль общих понятий начнет раскручиваться в обратном направлении, «зарядившись» конкретными значениями вместо общих концепций.

Как же структурировать весьма не простую систему знаний, более, кстати, сложную и запутанную, чем древовидная структура, подразумеваемая каждым из упомянутых методов? На взгляд автора, выход здесь всего один. Если хочешь кому-то что-то объяснить, вспомни, как это объясняли тебе, и постарайся не повторять ошибок своих учителей. Так мы и поступим.

К общему пониманию того, что представляет собой параллельная вычислительная система, автор настоящей работы пришел на практическом опыте (не без неоценимой помощи гораздо более опытных коллег, конечно) еще тогда, когда можно было построить серьезную установку, так и не узнав о существовании магического слова «кластер». По этой причине, соблазн изложения «снизу вверх» нам вовсе не грозит. В конце концов, кластер – понятие технологическое, и строим мы машины сегодня из готовых компонентов общего назначения не потому, что нам некуда девать эти самые компоненты, а потому лишь, что поступать так дешевле и быстрее, чем разрабатывать специализированное «железо». Попросту говоря, у нас есть некая **цель**, и достигаем мы ее наиболее доступными из имеющихся **средств**.

Вот это, пожалуй, и послужит нам ключом к структуризации изложения – будем стараться все время **идти от цели**, подбирая для ее реализации те средства, которые готовы предложить нам современные информационные технологии, и изучая эти средства, на некоторую разумную глубину, по мере надобности, уже нами осознанной.

Однако в этом случае нам просто совершенно необходимо начать с **достаточно конкретного** понимания того, для чего строится суперкомпьютер, что он делает. Пока мы знаем (см. Введение), что суперкомпьютер – это ***система для выполнения параллельных программ***, причем система эта построена, в силу непреодолимых технических и финансовых ограничений, по совершенно конкретной архитектуре. Мы знаем, что это за архитектура: речь идет о наборе отдельных компьютеров, каждый со своим процессором и своей памятью, объединенных средой передачи данных. Пока мы не поймем, как в принципе пишутся и работают программы для таких систем, вряд ли есть смысл двигаться дальше.

Приступим к выяснению того, что такое параллельная программа. Возьмем несложную последовательную программу и попытаемся переписать ее для выполнения на параллельной вычислительной системе. Не будем нарушать

традиций – выберем в качестве примера программу численного решения двумерной краевой задачи для уравнения теплопроводности методом простой итерации с использованием явного итерационного алгоритма. Текст такой программы приводится ниже.

```
#include <stdio.h>
/***/
/* Размеры сетки: */
#define MX 200
#define MY 200
/* Число итераций: */
#define NITER 100
/* Массив значений функции: */
       static float  f[MX][MY];
/* Массив приращений на шаге: */
       static float df[MX][MY];
/***/
       int main( int argc, char **argv )
{
       int i, j, n;
/***/
        printf( "Heat conduction task on %d by %d grid\n",
               MX, MY );
       fflush( stdout );
/* Начальные условия: */
       for ( i = 0; i < MX; i++ )
        {
          for ( j = 0; j < MY; j++ )
           {
            f[i][j] = df[i][j] = 0.0;
            if ( (i == 0)
              || (j == 0)
              || (i == (MX-1))
              || (j == (MY-1)) ) f[i][j] = 1.0;
           }
         }
/* Цикл итераций: */
       for ( n = 0; n < NITER; n++ )
        {
         printf( "Iteration %d\n", n );
          fflush( stdout );
/* Шаг итерационного процесса: */
/***/
/* Цикл вычисления приращений: */
          for ( i = 1; i < (MX-1); i++ )
           {
            for ( j = 1; j < (MY-1); j++ )
             {
              df[i][j] = ( f[i][j+1] + f[i][j-1] + f[i-1][j] +
f[i+1][j] )
```

```
                        * 0.25 - f[i][j];
            }
          }
/* Цикл вычисления новых значений функции: */
          for ( i = 1; i < (MX-1); i++ )
          {
            for ( j = 1; j < (MY-1); j++ )
            {
             f[i][j] += df[i][j];
            }
          }
/* Конец шага итерационного процесса */
        }
      return 0;
}
```

Пример 1.1. Модельная последовательная программа

Перед нами стоит задача ускорить выполнение этой программы, заставив несколько процессоров совместно (причем одновременно) выполнить записанные в ней вычисления. При этом мы не забываем, что каждый процессор оснащен своей памятью – значит, нам надо поделить между процессорами не только работу, но и обрабатываемые данные.

Проще всего делить работу, когда в программе имеются ***независимые вычисления***, то есть такие, которые, хотя и выполняются последовательно, поскольку на традиционных машинах вообще все выполняется последовательно, но в действительности не используют результаты друг друга, то есть могли бы выполняться в произвольном порядке без ущерба для смысла программы [3]. Иными словами, нам надо постараться найти циклы с независимыми витками, последовательная организация которых – не более чем дань «однопроцессорной традиции», и «раскидать» эти витки по разным процессорам.

Циклов в нашей программе, исключая задание начальных значений, всего пять. Внешний цикл (цикл итераций), очевидно, не является циклом с независимыми витками. В самом деле, нелегко выполнить следующую итерацию, не дождавшись завершения предыдущей. Значит, если и есть в этой программе независимые вычисления, кроются они внутри отдельной итерации. В этом смысле весьма интересны вложенные пары циклов вычисления приращений, с одной стороны, и значений функции – с другой. Легко видеть, что внутренние витки каждого из этих вложенных циклов совсем не зависят друг от друга. Если мы будем вычислять значения df[i][j], в первом случае, и f[i][j], во втором, не в том порядке, как это написано в программе, а в любом другом, суть дела от этого никак не изменится. При вычислении на очередной итерации значения f[5][7] нам совершенно не важно, вычислено ли уже новое значение для f[79][103] или это только планируется сделать.

Коль скоро каждая из этих пар вложенных циклов имеет независимые витки, ничто не мешает нам поручить выполнение этих витков, вообще говоря, разным процессорам. Как именно делить работу? Совершенно не важно (пока), главное, чтобы поровну: иначе менее загруженные процессоры будут на каждой итерации простаивать, ожидая более загруженного.

Располагай мы системой SMP (с общей памятью), задачу распараллеливания на этом месте можно было бы считать, в принципе, решенной. Однако перед нами стоит более сложная задача: нам надо поделить между процессорами не только работу, но и данные. Проще всего выполнить это для цикла получения новых значений функции (второго по тексту программы), тело которого состоит из единственного оператора f[i][j] += df[i][j]. В самом деле, как бы мы ни поделили элементы f и df между процессорами, если мы сделаем это единообразно, так, чтобы для всякой пары значений i и j f[i][j] и df[i][j] находились на одном процессоре, то все будет работать.

Хуже обстоит дело с первым циклом, в котором получаются значения приращений. Легко видеть, что для вычисления значения df[i][j] нам необходимо помимо значения f[i][j], иметь также значения f из соседних ячеек сетки – например, f[i-1][j]. Как бы мы ни поделили наши массивы между процессорами, периодически будут возникать ситуации, когда необходимые некоторому процессору элементы массива f находятся на другом процессоре и, значит, должны быть переданы оттуда при помощи сети передачи данных. Довольно очевидно, что, чем меньше будет таких ситуаций, тем легче нашим нескольким процессорам будет совместно решать поставленную перед ними задачу. Простейший способ достичь заметного уменьшения числа таких доступов к чужим данным – разделить массивы f и df между процессорами сплошными группами строк:

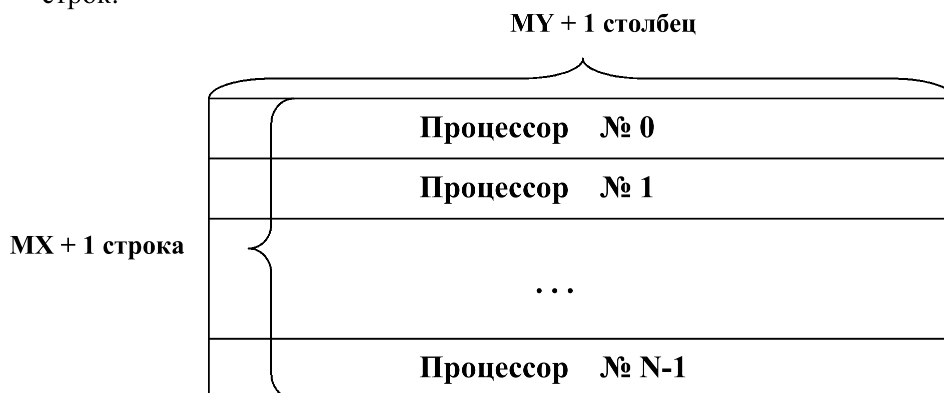

Рис. 1.1. Распределение массивов f и df по процессорам. Всего процессоров – N

При таком распределении данных (и одновременно работы) программам, выполняющимся на разных процессорах, придется обмениваться только копиями верхнего и нижнего краев полоски (перед выполнением каждой итерации). До тех пор, пока полоска не станет слишком узкой (из-за увеличения количества используемых процессоров по сравнению с размером используемой сетки), мы будем иметь заметное преобладание объема вычислений над объемом пересылок, к чему из общих соображений и следует стремиться.

Очевидно, в программе, выполняющейся на отдельном процессоре, надо будет предусмотреть в массивах f и df по одной «лишней» строке (сверху и снизу) для приема от «соседних» процессоров копий недостающих строк матрицы (точнее, это потребуется только для массива f, но, поскольку df должен быть распределен идентично f, отведем лишние строки в обоих):

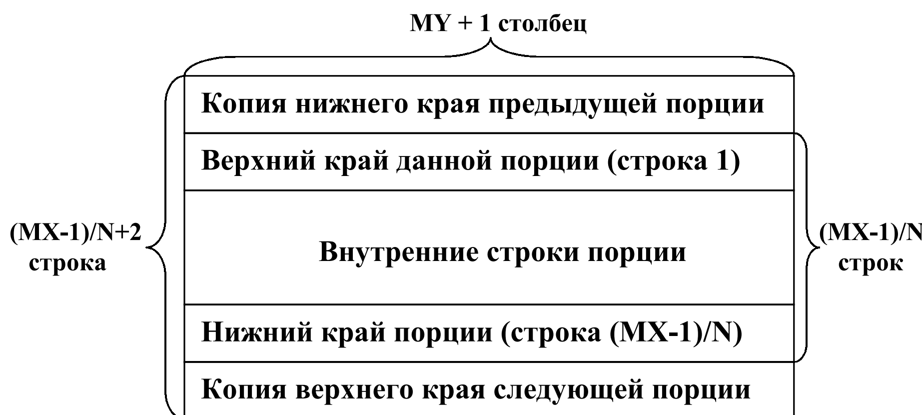

Рис. 1.2. Порция массива f, расположенная на одном процессоре

Столь же очевидно, что программа для нулевого процессора не обменивается с соседом «сверху», а программа для последнего – с соседом «снизу» («сверху» и «снизу» – в смысле рис. 1.2). В этих случаях в соответствующих «лишних» строках находятся просто граничные условия, которые не меняются в ходе расчета.

Проведя эти рассуждения, мы неявно постулировали несколько достаточно важных утверждений, на которых имеет смысл остановиться.

1. На содержательном уровне процесс превращения последовательной программы в параллельную состоит в решении трех задач:

1) как поделить между процессорами работу,

2) как поделить между процессорами данные,

3) как организовать обмен данными, которые в процессе вычислений нужны одному процессору, в то время как хранятся на другом.

Совокупность ответов на эти вопросы для конкретной программы (алгоритма) будем называть ***схемой параллелизма*** для данного алгоритма.

2. Для реализации схемы параллелизма необходимо написать для каждого из участвующих в параллельной работе процессоров по отдельной программе, а потом все их вместе запустить, обеспечив предварительно возможность взаимодействия таких программ путем обмена данными. Такая программа, выполняющаяся на отдельном процессоре, рассчитанная на взаимодействие с себе подобными в рамках совместного решения одной задачи, называется ***ветвью параллельной программы***, а параллельная программа – это всего лишь совокупность ветвей. Ветви как-то именуются (в нашем случае – пронумерованы подряд с нуля).

3. Как правило, ветви одной параллельной программы очень похожи, но не идентичны (см. выше замечание об отсутствии некоторых обменов в нулевом и последнем процессорах).

Теперь нам предстоит написать текст ветви (часто для краткости именно ветвь называют «параллельной программой»). Конечно, никто и ничто не запрещает программисту написать для выполнения на 100 процессорах 100 отдельных исходных текстов, оттранслировать и запустить их все одновременно на 100 процессорах, но это довольно утомительно. Тем более, что ветви для разных процессоров, как мы знаем, довольно похожи. Проще (и так поступают почти всегда) написать единственную программу – обобщенную ветвь для процессора k, где значение k заранее не известно, и запустить одну и ту же программу на каждом из 100 процессоров одновременно. Конечно, в тексте такой программы должна быть предусмотрена проверка, на каком именно процессоре данный экземпляр программы выполняется, и модификация собственных действий в зависимости от конкретного значения k.

Разумный вопрос: откуда ветвь узнает собственный номер? Примерно оттуда же, откуда программа «узнает» свои начальные данные, когда получает их из файла. То программное обеспечение, которое стоит за всеми этими непростыми действиями (одновременный запуск ста экземпляров программы на ста разных процессорах, обеспечение впоследствии их взаимодействия путем обмена данными и т. п.) должно предоставить ветви эту информацию в виде дополнительных исходных данных, подобно тому как, обращаясь к специальным системным функциям, обычная программа может узнать время суток или номер версии операционной системы, хотя в число исходных данных, подготовленных пользователем, эти значения и не входят.

Мы снова неявно постулировали ряд важных утверждений, которые имеет смысл привести в явном виде.

1. Очень часто (хотя и не всегда) все ветви одной параллельной программы реализуются одной программой (одним исходным текстом и одним исполняемым файлом). По этой причине такой текст часто даже называют «параллельной программой», что, строго говоря, не верно.

2. Для того, чтобы ветви могли быть одновременно запущены на надлежащем числе процессоров и успешно взаимодействовали, системное программное обеспечение должно решить, как минимум, следующие задачи:

1) согласованный запуск указанного пользователем числа экземпляров программы (программ) на соответствующем числе процессоров,

2) предоставление каждой ветви по ее требованию информации о номере процессора, на который попала эта ветвь, и общем количестве процессоров, на которых произведен запуск,

3) предоставление ветви возможности обмена данными с другими ветвями по ее (ветви) требованию.

Аппаратура связи между процессорами, используемая для решения первой задачи, называется ***сетью управления*** параллельной вычислительной установки. Аппаратура связи, решающая третью задачу, называется ***сетью коммуникаций*** (давая эти определения, мы не утверждаем, что это обязательно разные сети).

Программный компонент, решающий эти три задачи, называется ***базовым программным обеспечением выполнения параллельных программ***. Это базовое ПО подразделяется, в свою очередь, на ***базовую систему запуска*** (грубо говоря, это та команда, которую набирает на клавиатуре пользователь, чтобы параллельная программа начала выполняться) и ***базовую коммуникационную библиотеку*** (тот набор функций, к которым обращается ветвь параллельной программы, чтобы узнать собственный номер или послать сообщение другой ветви).

Вернемся к написанию текста ветви параллельной программы. Мы не сможем выполнить эту задачу до конца, пока не выберем конкретную коммуникационную библиотеку и не изучим ее основных возможностей. К счастью, до конца нам сейчас писать ее и не требуется – «спрячем» обращения к конкретным функциям коммуникационной библиотеки в небольшом числе функций, написание которых отложим. Сейчас важно понять, насколько усложнится основной текст, а не как именно называется функция опроса собственного номера. С учетом этой оговорки, текст ветви параллельной программы выглядит так:

```
#include <stdio.h>
/***/
/* Размеры сетки: */
#define MX 200
#define MY 200
/* Число итераций: */
#define NITER 100
/* Порция массива значений функции на данном процессоре: */
       static float  f[MX][MY];
/* Порция массива приращений на шаге: */
       static float df[MX][MY];
/* Собственный номер ветви, число ветвей, фактический */
/* размер порции массивов: */
       int my_number, n_of_nodes, mx, my;
/***/
       int main( int argc, char **argv )
{
       int i, j, n, nprev, nnext;
/***/
/* Обращаемся к коммуникационной библиотеке: Кто я? Сколько нас? */
/* Реальные обращения к конкретной библиотеке скрыты в функции */
/* get_node_data, которую нам еще предстоит написать, причем в */
/* расчете на конкретную коммуникационную библиотеку: */
        get_node_data( &argc, &argv, &my_number, &n_of_nodes );
/* Не спрашивайте, зачем этой программе передаются ссылки на */
/* argc и argv, – в свое время мы об этом узнаем. */
/* Запоминаем номера предыдущей и следующей ветвей – нам они */
/* понадобятся для организации обменов данными. */
/* Если нет предыдущей или нет следующей – запоминаем (-1): */
        if ( my_number > 0 )
        nprev = my_number – 1;
       else
        nprev = -1;
       if ( my_number < (n_of_nodes-1) )
        nnext = my_number + 1;
       else
        nnext = my_number + 1;
/* Сколько строк матрицы нам реально досталось? Это было задано */
/* пользователем при запуске, нам надо именно на столько частей */
/* порезать массив. Будем считать, что все делится нацело: */
       mx = MX/n_of_nodes;
       my = MY;
       if ( !my_number )
        {
/* Печатаем только из нулевой ветви – зачем нам N одинаковых печатей?
*/
          printf( "Heat conduction task on %d by %d grid by %d proces-
sors\n",
                 mx*n_of_nodes, my, n_of_nodes );
        }
       fflush( stdout );
```

```
/* Начальные условия. По хорошему, надо было бы их задать */
/* только на верхнем краю в нулевой ветви и на нижнем в */
/* последней, но для простоты зададим везде. Там, где они */
/* не нужны, они попадут в места, предназначенные для */
/* обменов краями, и при первом же обмене все равно будут */
/* затерты: */
       for ( i = 0; i < mx; i++ )
        {
         for ( j = 0; j < my; j++ )
          {
           f[i][j] = df[i][j] = 0.0;
           if ( (i == 0)
             || (j == 0)
             || (i == (mx-1))
             || (j == (my-1)) ) f[i][j] = 1.0;
          }
        }
/* Цикл итераций: */
       for ( n = 0; n < NITER; n++ )
        {
/* Обращаемся к коммуникационной библиотеке: обмениваемся краями */
/* Реальные обращения к конкретной библиотеке скрыты в функции */
/* exchange, которую нам еще предстоит написать, причем в */
/* расчете на конкретную коммуникационную библиотеку. */
/* аргументы функции exchange: */
/* - длина сообщения */
/* данные для обмена с предыдущей ветвью: */
/* - номер ветви (-1), если нет предыдущей */
/* - что слать */
/* - куда принимать */
/* то же для следующей ветви */
/* всего 7 аргументов */
         exchange(
                  my - 2,
                    nprev, &f[1][1],          // Посылка
                           &f[0][1],          // Прием
                    nnext, &f[mx-2][1], // Посылка
                           &f[mx-1][1] // Прием
                  );
         if ( !my_number )
          {
          printf( "Iteration %d\n", n );
           fflush( stdout );
          }
/* Шаг итерационного процесса: */
/***/
/* Цикл вычисления приращений: */
         for ( i = 1; i < (mx-1); i++ )
          {
           for ( j = 1; j < (my-1); j++ )
```

```
            {
             df[i][j] = ( f[i][j+1] + f[i][j-1] + f[i-1][j] +
f[i+1][j] )
                        * 0.25 - f[i][j];
            }
          }
/* Цикл вычисления новых значений функции: */
          for ( i = 1; i < (mx-1); i++ )
          {
            for ( j = 1; j < (my-1); j++ )
            {
             f[i][j] += df[i][j];
            }
          }
/* Конец шага итерационного процесса */
        }
      return 0;
}
```

Пример 1.2. Модельная параллельная программа – функция main

Мы видим, что по сравнению с последовательным прототипом программа если и удлинилась существенно, то разве что за счет комментариев. В целом ее общая логика сохранилась. В программе присутствуют обращения к двум функциям, в которых «спрятаны» вызовы функций коммуникационной библиотеки. В отношении первой из них – *get_node_data* – все более или менее ясно. Внутреннее строение этой функции, в любом случае, тривиально, и мы просто изобрели собственное, ни на что не похожее имя, чтобы не «привязываться» к конкретной коммуникационной библиотеке. В отношении функции *exchange* ясности гораздо меньше. Ясно, что, помимо знаний о конкретных именах функций конкретной коммуникационной библиотеки, текст функции *exchange* должен содержать в себе некую нетривиальную логику, а значит нуждается в дальнейшем объяснении. Мы ведь не знаем пока не только имен конкретных функций и порядка аргументов в них, но и вообще ничего не знаем о том, как принято в коммуникационных библиотеках записывать обмены данными между процессорами.

Впрочем, сейчас нам это и не важно. Представляется, что приведенная здесь часть текста программы отражает главное – схему параллелизма. В ней написано, как поделены между ветвями данные и работа, как ветвь настраивается на общее число ветвей, задаваемое пользователем при запуске программы на счет, и на собственный номер. Более того, с учетом комментариев можно считать, что в нашем тексте написано, где (в каком месте программы) и какие именно данные перемещаются между ветвями (в полном соответствии с рисунками 1 и 2, приведенными выше). Большего нам в этой главе и не нужно, а к

более подробному рассмотрению этого примера мы еще не раз вернемся в последующих главах.

Что мы можем сказать о составе и внутренней структуре аппаратных и программных средств суперкомпьютера, опираясь на только что приведенный пример?

Начнем с состава аппаратуры. На примере мы узнали, что параллельная вычислительная установка состоит из некоторого количества отдельных компьютеров, взаимодействующих между собой по двум (с логической точки зрения) сетям: сети управления и сети коммуникаций. Мы также понимаем, что в реальных программах, в отличие от рассмотренной нами программы модельной, обязательно присутствует ввод исходных данных и вывод результатов – значит, нужна некоторая общая для всех система файлового ввода/вывода. Более подробному обзору этих составных частей аппаратуры суперкомпьютера посвящена Глава 2.

О программном обеспечении мы тоже можем сказать довольно много. Очевидно, ветвь параллельной программы – это, прежде всего, обычная программа, пользующаяся услугами операционной системы отдельного процессора, на котором она запущена. Кроме того, необходима реализация на основе этой ОС отдельного процессора базовой системы выполнения параллельных программ, которая, в свою очередь, состоит из базовых средств запуска и коммуникационной библиотеки. Конечно, от ОС отдельного процессора требуется, чтобы на ней можно было достаточно эффективно реализовать базовую систему выполнения параллельных программ и соответствующие механизмы универсальных ОС, используемые в такой реализации, должны быть рассмотрены. С другой стороны, сама базовая система выполнения параллельных программ, на каком бы фундаменте она ни стояла, должна быть реализована наилучшим образом, и есть смысл посмотреть, из чего она состоит, и как эти части устроены. Этому посвящена Глава 3.

Понятно, что на одном суперкомпьютере одновременно выполняются, вообще говоря, несколько задач, подобных рассмотренной выше. Как они «разбираются», какие процессоры можно занимать под выполнение ветвей, а какие уже заняты? Эта проблематика рассматривается в Главе 4.

Глава 5, выделенная в отдельную часть книги, повествует о том, какие существуют средства для разработки (а не выполнения) параллельных программ. Проводя аналогию с традиционным, последовательным, программированием, можно сказать, что для понимания, что такое вообще программа, лучше всего приводить примеры на Си, Фортране или даже языке ассемблера (реальном или вымышленном, как в монографии Кнута). Однако сущест-

вует ведь и Пролог, и специализированные языки (вроде языка SQL-запросов). В параллельном программировании ситуация аналогичная.

Наконец, Глава 6 и последующие имеют «рецептурный» характер. Они посвящены, преимущественно, систематизации тех «подводных камней», с которыми сталкиваются разработчики и пользователи кластеров, а также приемам выбора наилучших решений по использованию готовых компонентов в конкретных ситуациях.

Рекомендуемые поисковые фразы:

`parallel programming fundamentals`

Глава 2. Аппаратное обеспечение

Для начала рассмотрим общую схему. Пусть нам необходимо построить локальную сеть из ***вычислительных узлов*** (в количестве, допустим, 15 штук) и ***управляющей машины***. На вычислительных узлах будут выполняться параллельные программы, которые проще всего писать, исходя из равной загрузки узлов работой. По этой причине все сервисные функции, в том числе подготовку программ и данных, лучше возложить на отдельный компьютер – управляющую машину, а не на один из узлов. Все узлы и управляющая машина должны быть связаны между собой ***сетью управления***, при помощи которой управляющая машина взаимодействует с узлами, а узлы, при необходимости, друг с другом.

Кроме того, узлы должны быть связаны между собой ***сетью коммуникаций***, посредством которой узлы обмениваются данными в процессе совместного решения единой задачи. Управляющая машина не обязательно должна быть включена в сеть коммуникаций. От последней как минимум требуется максимально возможная производительность. Ведь чем теснее связаны узлы, чем меньше ограничений накладывает сеть на интенсивность общения узлов между собой, тем проще будет программисту построить параллельный алгоритм, дающий максимальный скоростной выигрыш от параллельного выполнения.

Мы здесь не утверждаем, что физических локальных сетей, объединяющих узлы и управляющую машину, должно быть именно две (хотя в кластерах МВС-1000 [10, 11], например, дело обстоит именно так). В самом общем случае сеть управления и сеть коммуникаций – сущности виртуальные. Физически эти два набора функций могут быть возложены как на одну локальную сеть, так и, например, на четыре (см. ниже).

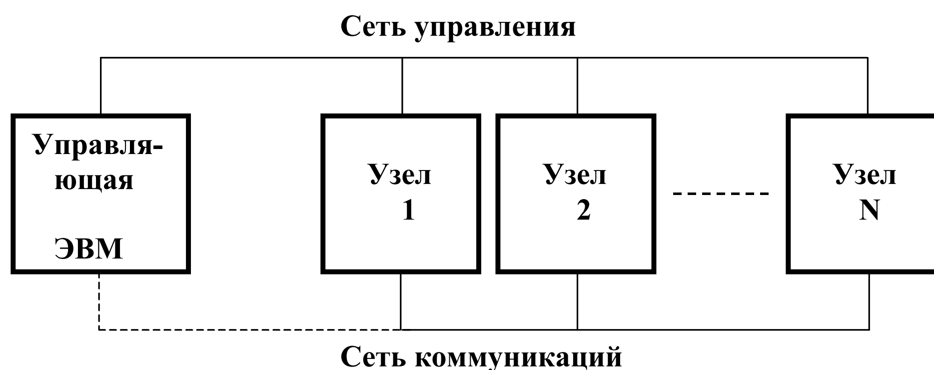

Рис. 2.1. Общая структура суперкомпьютера

2.1. Рабочая станция узла

Для простоты будем считать, что в этом качестве используется обычный Intel-совместимый персональный компьютер, хотя, конечно, возможны и другие варианты. Клавиатура и монитор узлу не требуются – разве что для первоначальной установки ОС и поиска «хитрых» неисправностей. Локальный диск весьма желателен – это упрощает не только начальную загрузку, но и многие другие сервисные действия. Будем считать, что он есть. Весьма желательно также, чтобы узлы были близки по производительности, а управляющая машина совпадала с узлами по системе команд процессора.

Первое, что нам предстоит сделать с узлами, – это объединить их средой связи, способной выполнять функции сетей управления и коммуникаций. Подавляющее большинство современных материнских плат имеет встроенный разъем для подключения локальной сети Fast Ethernet. Опыт показывает, что возложение всех сетевых функций на единственную физическую сеть Fast Ethernet не желательно и что следует обеспечить возможность подключения хотя бы одного дополнительного сетевого адаптера.

В целом рабочая станция узла как таковая – наиболее беспроблемная часть проекта кластерной машины, чего нельзя сказать о сетевом компоненте, к рассмотрению которого мы и переходим.

2.2. Коммуникационная среда – баланс производительности и стоимости

Разброс цен и производительностей коммуникационного оборудования, применяемого в современных кластерах, значительно превосходит один десятичный порядок. Поэтому начнем наш обзор с выяснения того, из чего складывается эта самая производительность. Для этого построим очень простую грубую, но во многих отношениях полезную модель коммуникационной среды кластера.

Базовое понятие нашей модели – обмен «точка–точка» между двумя произвольными узлами: например, узел 7 посылает узлу 9 сообщение длиной 100 байт, а узел 9 его принимает. Прежде всего, примем гипотезу **симметрии** обменов «точка–точка»: будем считать, что с точки зрения производительности обмены между любыми двумя узлами равноценны. В отношении транспьютероподобных систем, популярных лет 7–8 назад, подобное допущение было бы весьма далеко от истины: в этих системах узлы соединялись попарно отдельными линиями связи в образования типа решеток, гиперкубов и им подобных [2, 4, 8, 9, 11]. При этом, конечно, показатели эффективности обменов между узлами решающим образом зависели от того, сколько «пересадок» делало сообщение на своем пути. В отношении современных технологий локальных сетей, применяемых при построении класте-

ров, напротив, наше допущение довольно реалистично, поскольку все они строятся либо на основе центрального коммутатора, либо (реже) на основе общей шины. Неоднородностью внутренней структуры коммутатора или каскада из небольшого числа коммутаторов можно с некоторой степенью точности пренебречь.

Производительность обмена «точка–точка» зависит от того, является ли этот обмен единственным в сети или одновременно с ним выполняются другие обмены. Не только обмены с одними и теми же участниками (например, две одновременных посылки из узлов 3 и 5 в узел 11), но и обмены между разными парами узлов (3-й посылает 5-му, а в это же время 9-й посылает 8-му) могут «мешать» друг другу.

Наконец, при выполнении обмена (приема или передачи сообщения) узлом процесс разделяется на два элементарных акта: запуск обмена и проверка завершения. В то время, когда обмен запущен, но еще не завершился, узел вправе выполнять некоторые расчеты. Вычислительная производительность узла на фоне выполняющегося обмена может быть ниже, чем обычно, поскольку идущий асинхронно обмен потребляет ресурсы узла: нагружает шину памяти, отвлекает процессор прерываниями и т. п.

Конечно, модель коммуникационной среды, построенная с учетом только этих соображений, далека от полноты и точности, но для целей формулирования критериев эффективности мы этими соображениями ограничимся.

2.2.1. Критерии эффективности коммуникационной среды

Производительность, латентность и цена обмена

Производительностью канала «точка–точка» между узлами A и B будем называть количество данных, передаваемых по каналу в единицу времени в среднем за некоторый большой промежуток времени, скажем, за минуту. Производительность канала можно представить себе как среднюю скорость передачи данных. Очевидно, производительность канала сильно зависит от того, какой длины сообщения используются при передаче данных, то есть от того, как часто канал «останавливается», чтобы потом снова «разогнаться».

Пусть узел A передает узлу B сообщение длиной X байт, и при этом никаких других обменов в сети не происходит. Время T, затрачиваемое на такую передачу, довольно точно оценивается формулой:

$$T = SX + L,$$

где L не зависит от X.

В этой формуле, очевидно, S есть ***пропускная способность канала «точка–точка» на пустой сети*** или ***мгновенная скорость*** передачи данных. S измеряется в (мега)байтах в секунду.

Величина L, в свою очередь, представляет собой ***время запуска обмена***, не зависящее от длины сообщения, и измеряется в (микро)секундах. На профессиональном жаргоне принято называть эту величину ***латентностью***, и мы будем впредь поступать так же, чтобы не нарушать традицию.

Иногда удобно оперировать ***латентностью, приведенной к скорости***, или ***ценой обмена***, которую мы обозначим как P:

$$P = L\,S.$$

Эта величина измеряется в байтах и имеет несколько полезных «физических» интерпретаций. Прежде всего, цена обмена – это число байт, которое канал «точка–точка» мог бы передать за время своего запуска, если бы «умел» запускаться мгновенно. Иными словами, за счет «инертности» канала к каждому передаваемому им сообщению «как бы добавляется», с точки зрения скорости передачи, P байт.

Таким образом, производительность канала зависит от длин сообщений, используемых при передаче данных. Если X много больше P, то есть длина сообщений много больше цены обмена, производительность близка к пропускной способности. Напротив, если X много меньше P, производительность практически полностью определяется латентностью, а не пропускной способностью. Наконец, при X равном P производительность равна в точности половине пропускной способности канала. Тем самым, **цена обмена – это такая длина сообщения, при использовании которой производительность канала равна половине его пропускной способности**.

В итоге, мы сформулировали два независимых критерия эффективности: пропускную способность канала «точка – точка» на пустой сети и латентность, и один производный критерий – цену обмена. Очевидно, сеть тем лучше, чем выше пропускная способность, и чем ниже латентность. Так, в SMP-системе (машине с общей, симметрично адресуемой памятью) пропускная способность бесконечна, а латентность теоретически равна нулю.

Полнота сети

Полнота сети есть мера того, насколько несколько одновременно происходящих обменов «мешают» друг другу. Простейшее определение полноты сети – это понятие ***бисекционной полноты***: сеть называется бисекционно полной, если **любые** обмены между **разными** парами узлов, происходящие одновременно и в любом количестве, **совсем** не мешают друг другу. Или, что то же самое, при лю-

бом разделении сети пополам (бисекции) пропускная способность потока данных из одной половины в другую есть сумма пропускных способностей независимых каналов, ведущих из одной половины сети в другую [1, 3]. Типичный пример бисекционно неполной сети – сеть на базе двух однородных коммутаторов, объединенных между собой единственной линией.

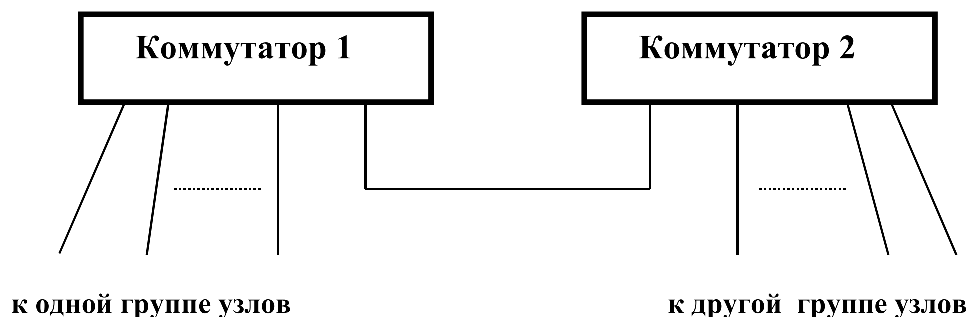

Рис. 2.2. Бисекционно неполная сеть

Следует отметить, что при практической оценке пропускной способности реальной сети в режиме интенсивных перекрестных обменов модель бисекционной полноты недостаточна. Если сеть построена на базе центрального коммутатора (а так бывает чаще всего), то реальное время выполнения серии одновременных обменов, некоторые из которых пересекаются по участвующим процессорам, зависит от особенностей внутренней реализации коммутатора. Достаточно типична ситуация, когда два коммутатора разных моделей, каждый из которых реализует попарно независимые обмены без потери быстродействия, сильно отличаются по времени выполнения одной и той же тестовой серии частично пересекающихся обменов.

Информацию о внутренней логике поведения коммутатора, по которой можно было бы предсказать его скоростные характеристики в этом режиме, добыть обычно не удается: производители ее не сообщают. При этом в реальных задачах почти всегда имеет место именно такой режим, когда одни и те же процессоры участвуют одновременно в нескольких разных обменах. Поскольку достоверной информации о внутренней логике коммутатора у нас нет, практически нецелесообразно строить на эту тему какие-то сложные формализованные модели, основанные на выдуманных допущениях. При необходимости оценить качество коммутатора или сравнить несколько коммутаторов между собой разумно воспользоваться каким-либо простым эталонным тестом. Такой тест должен циклически, много раз подряд осуществлять обмены всех узлов со всеми сообщениями фиксированной длины X, причем X должно быть заметно больше цены обмена.

Все обмены на одном узле должны запускаться одновременно, чтобы исключить их зависимость друг от друга по порядку выполнения. Если среднее время выполнения витка такого теста равно T, то легко подсчитать суммарный объем переданных за это время **одним узлом** данных D:

$$D = X(N - 1),$$

где N – число процессоров. Если бы сеть была идеальна, то есть никакой обмен не мешал бы никакому другому, этот объем данных мог бы быть передан из узла за время T_i:

$$T_i = D/S = (X(N - 1))/S.$$

Считая канал, связывающий узел с сетью, дуплексным, то есть способным одновременно с передачей принимать данные с той же скоростью, и помня, что узлы работают одновременно, видим, что T_i представляет собой теоретически идеальное время срабатывания витка нашего теста. В действительности, тест будет выполнять виток за экспериментально измеренное время T_e, очевидно, большее, чем T_i. Поделив одно на другое, получим ***экспериментальный коэффициент полноты*** F_e:

$$F_e = T_i/T_e.$$

Вне всякого сомнения, приведенное здесь определение экспериментального коэффициента полноты нельзя назвать определением в математическом смысле этого слова. Мы сделали слишком много не формализованных допущений. Конечно, значение коэффициента будет при прочих равных условиях зависеть и от длины сообщения X, и от особенностей тестовой программы. Однако приведенная здесь процедура экспериментальной оценки полноты сети обладает одним несомненным достоинством: она позволяет на практике измерить буквально, на сколько процентов наша сеть является идеальной с точки зрения ее полноты. Опыт показывает, что для сравнения между собой различных сетевых коммутаторов такой способ измерения весьма полезен. Конечно, если полученные на разных коммутаторах значения F_e отличаются на четверть, это мало о чем говорит, но на практике не редкостью являются отличия в несколько раз.

Потребность сети в вычислительной мощности

Пусть некоторое вычисление – например, перемножение двух матриц – занимает время T_m. Выполним то же самое вычисление на фоне запущенного обмена, время завершения которого заведомо превосходит T_m. Теперь это же вычисление займет время T_c. Коэффициент замедления счета на фоне обмена C вычисляется по формуле:

$$C = T_m/T_c.$$

Отметим, что для «более или менее разумной» сети C должно мало отличаться от единицы, то есть сеть не должна заметно потреблять вычислительную мощность узла.

В итоге мы сформулировали четыре независимых критерия эффективности сети, важных с точки зрения ее практического использования. Таковыми являются:

- пропускная способность канала «точка–точка»,
- латентность,
- экспериментальный коэффициент полноты,
- потребность сети в вычислительной мощности.

Рекомендуемые поисковые фразы:

`parallel computer communication system`

2.2.2. Обзор современных сетевых решений

Вооружившись формулировками критериев эффективности, мы можем теперь сравнить между собой присутствующие в настоящее время на рынке сетевые технологии. Выбор подходящей сетевой технологии (технологий) при сооружении кластера – пожалуй, наиболее ответственное проектное решение. Вычислительная производительность кластера грубо оценивается, как сумма вычислительной производительности его узлов. Однако очевидно, что реальная его производительность при выполнении расчетов сильно зависит от эффективности сети.

Если сеть очень «слабая», класс прикладных задач, при решении которых реальная производительность будет хоть сколько-нибудь приближаться к сумме производительностей узлов, будет исчезающе узок. С другой стороны, удельная стоимость сетевого оборудования (из расчета на узел) варьируется в зависимости от выбора конкретной сетевой технологии от нескольких долларов до 3–4 тысяч. Установив слишком «мощную» сеть, разработчик кластера, тем самым, автоматически отказывается от возможности удвоить, а то и утроить число узлов за те же деньги. Если класс задач, для решения которых строится кластер, заведомо позволяет использовать более дешевую сеть, установка более дорогой сети была бы ошибкой. Чтобы выбрать верное для каждого конкретного случая решение, то есть соблюсти баланс между стоимостью и производительностью сети, надо знать, из чего выбирать.

«Классический» (10-мегабитный) вариант Ethernet

Этот вид сети, с которого и начиналось нынешнее триумфальное шествие сетевых технологий, устарела, и сегодня практически не применяется. Для нее ха-

рактерны: пропускная способность около 1 мегабайта в секунду, латентность на уровне сотен микросекунд и построение сети, как правило, в виде общей шины. Последнее означает очень низкие показатели полноты, поскольку в каждый момент времени только один узел во всей сети может передавать. Потребность в вычислительной мощности для современных процессоров практически не заметна. С выходом значительно более мощных сетевых технологий на ценовой уровень в десятки и менее долларов за узел, смысл применения классического варианта Ethernet в кластерах выделенных рабочих станций пропал окончательно.

100-мегабитный вариант Ethernet

Технология, обычно называемая Fast Ethernet, является сегодня наиболее применяемой локальной сетью и успешно используется в недорогих кластерах выделенных рабочих станций, а также для вспомогательных целей и в более мощных параллельных системах. Пропускная способность этой сети – около 11,5 мегабайт в секунду, в дуплексном режиме и ввод, и вывод происходят одновременно с этой скоростью. Латентность довольно высока: в зависимости от используемого процессора и, в особенности, конкретной модели сетевого адаптера, она может варьироваться в пределах от 40 до 70 микросекунд. Потребность в вычислительной мощности пренебрежимо мала. Характеристики полноты целиком определяются используемым коммутатором и варьируются в широких пределах.

Наиболее простой и дешевый вариант Fast Ethernet – соединение узлов через концентратор (hub). Логически такое соединение эквивалентно общей шине – в каждый момент времени только один узел может передавать, остальные должны либо слушать, либо ждать, пока сеть освободится. В офисных локальных сетях такой вариант применяется достаточно широко, но в кластерах выделенных рабочих станций он неприемлем. Требуется использовать сетевой коммутатор (switch), позволяющий нескольким узлам передавать одновременно путем коммутации кадров (пакетов). Смысл коммутации в том, чтобы кадры доходили только до тех узлов, которым они предназначены, при этом независимые обмены могут не мешать друг другу. По экспериментальному коэффициенту полноты, измеренному одним и тем же тестом, различные коммутаторы отличаются очень сильно, иногда в 3–4 раза. Коммутатор, который можно рекомендовать для применения в кластере выделенных рабочих станций, должен быть бисекционно полным, то есть выполнять любое число пар независимых обменов без потери производительности и иметь экспериментальный коэффициент полноты около 30%. Отметим, что не все коммутаторы Fast Ethernet вообще работоспособны в характерных для кластера режимах интенсивных нагрузок. Грамотно построенный тест, используемый для измерения экспериментального коэффициента полноты, проверяет заодно и интегральную работоспособность коммутатора. Если этот тест «подвисает», такой коммутатор использовать нельзя.

Из всех применяемых в кластерах сетевых технологий Fast Ethernet – наиболее дешевая. Цена сетевого адаптера часто равна нулю, поскольку некоторые современные материнские платы имеют встроенный адаптер. Впрочем, качественный внешний адаптер вряд ли будет дороже 30–40 долларов (никогда не покупайте самый дешевый – это верный способ собрать не работающую установку!). Цена коммутатора из расчета на узел – десятки долларов и менее.

1000-мегабитный вариант Ethernet

Эта технология во всем сходна с Fast Ethernet, отличаясь от нее в десять раз более высокой частотой передачи. К сожалению, в десять раз лучшей по сравнению с Fast Ethernet ее назвать трудно.

Пропускная способность обычно не превышает 60–80 мегабайт в секунду, латентность зачастую в полтора–два раза **выше**, чем у Fast Ethernet (что удивительно, но факт). Потребность в вычислительной мощности заметна, в отношении полноты справедливы все те замечания, которые касались Fast Ethernet. Цена адаптера еще год–полтора назад достигала 200–300 долларов, а цена хорошего коммутатора в расчете на узел – около 700 долларов. Учитывая, что оборудование с гораздо большей пропускной способностью и многократно меньшей латентностью можно было приобрести уже за две тысячи долларов на узел, применение этой технологии представлялось достаточно спорным решением. В последнее время ситуация существенно изменилась, адаптеры подешевели, а коммутаторы подешевели многократно. Это может означать, что технология станет привлекательной в качестве «промежуточной» между фактически бесплатной, но медленной Fast Ethernet и гораздо более быстрыми, но дорогими решениями, разработанными специально для кластеров.

Myrinet

Эта технология [21] представлена изделием единственного производителя – Myricom, Inc. Весьма подробную информацию о нем можно получить на сайте производителя. Здесь ограничимся кратким обзором.

Технология изначально ориентирована на кластеры выделенных рабочих станций. В отличие от Ethernet, не предусматривается, например, наличие длинных (в сотни метров) соединений. Пропускная способность определяется фактически скоростью взаимодействия сетевого адаптера с памятью узла по шине PCI, а не скоростью передачи данных в линии, которая достигает двух гигабит в секунду по каждому из направлений в дуплексном режиме. На реально применяемых сегодня материнских платах вполне можно рассчитывать на 160–180 мегабайт в секунду. Латентность составляет 16–18 микросекунд, потребность в вычислительной мощности мала. Обязательно использование коммутатора, позволяющего получать ра-

зумные значения экспериментального коэффициента полноты даже для очень больших (сотни процессоров) установок. Цена сетевого адаптера – около тысячи долларов, цена коммутатора из расчета на узел – примерно 600 долларов. Myricom обещает в самое ближайшее время значительно снизить цену адаптера и перейти на выпуск коммутаторов, способных коммутировать также и соединения Gigabit Ethernet. Если это произойдет, данная технология, без сомнения, сможет рассматриваться в качестве основной для суперкомпьютеров средней ценовой группы. Впрочем, уже сегодня технология Myrinet очень популярна.

SCI

Эта технология так же представлена продукцией единственного производителя – Dolphin Interconnect Solutions, Inc. Сеть имеет примерно такие же, как Myrinet, показатели эффективности, за исключением одного – латентности. По этому показателю SCI – безусловный чемпион: латентность заметно ниже 10 микросекунд (заявленный минимум – 3 микросекунды). В отличие от Myrinet, предел масштабируемости – 128 узлов. Стоимость из расчета на узел примерно соответствует показателям Myrinet и целиком сосредоточена в сетевом адаптере и кабелях: коммутаторы в этой сети практически не применяются, поскольку она построена на базе очень высокоскоростной общей шины. Физически шина реализуется как несколько шин из довольно дорогостоящих кабелей, и выбираемая топология влияет как на показатели полноты, так и на цену.

Прочие технологии

Из прочих технологий следует упомянуть, прежде всего, Giganet [1] и Quadrics. Ограничимся упоминанием по следующей причине. Главный принцип кластерной технологии построения суперкомпьютера – использование максимально готовых, то есть уже опробованных компонентов. По состоянию на сегодня рассмотренные технологии, безусловно, являются наиболее распространенными, все остальное по сравнению с ними – экзотика. В каких случаях применение «экзотики» оправданно? Вероятно, лишь в двух:

- предлагается решение гораздо более производительное, чем все широко распространенные,
- предлагается решение гораздо более дешевое.

Ни одна из прочих известных на сегодня сетевых технологий этими свойствами не обладает. И Giganet, и Quadrics имеют сравнимые с Myrinet и SCI показатели производительности, при гораздо более высокой цене и, что важно, меньшей распространенности. Конечно, эти решения используются профессиональными разработчиками дорогостоящих, собираемых по заказу систем – возможно, в силу традиции, или в силу каких-то существенных именно для этих систем второсте-

пенных преимуществ. Мы же имеем в виду более или менее малобюджетный, самодельный суперкомпьютер, так что для нас использование, скажем, Quadrics вместо Myrinet – непозволительная роскошь.

Из приведенного обзора очевидна резкая «ступенька» в цене и производительности доступных сетевых технологий. В самом деле, сеть с производительностью около 10 мегабайт в секунду и латентностью около 60 микросекунд можно получить почти бесплатно. Сеть с производительностью в 10–15 раз выше и гораздо меньшей латентностью можно получить уже по цене 1,5–2 тысячи долларов за узел. Промежуточного по цене и производительности варианта не просматривается, за исключением Gigabit Ethernet, популярность которой, видимо, резко возрастет в ближайшее время за счет заметного падения цен, в основном – на гигабитные коммутаторы. Впрочем, довольно высокая латентность, свойственная этому оборудованию, ставит ряд непростых вопросов в области программного обеспечения.

Рекомендуемые поисковые фразы:

`parallel cluster network hardware`
`Ethernet fundamentals explained`
`sci dolphin home page`

Рекомендуемые сетевые ресурсы:

`www.myri.com`
`www.dolphinics.com`

2.3. Организация внешней памяти

Пока не получил отражения тот очевидный факт, что параллельные программы, требующие очень большого быстродействия, вероятно, обрабатывают весьма солидные объемы данных, хранимые на дисках. При этом в кластере доступ к этим данным нужен всем узлам. Реализация доступа узлов к единому хранилищу файлов – задача, имеющая самое непосредственное отношение к вопросам организации коммуникационной среды. Рассмотрим кратко основные применяемые в кластерах режимы использования дисковой памяти именно под этим углом зрения, обращая основное внимание на то, какое использование коммуникационной среды подразумевает тот или иной режим.

Общий файловый сервер

Это наиболее простой в реализации и использовании режим. Файлы с исходными данными и результатами расчетов размещаются на едином файловом

сервере, роль которого может выполнять как управляющая ЭВМ кластера, так и отдельная серверная ЭВМ. Для организации доступа к этим файлам со стороны узлов используются стандартные приемы сетевого доступа к удаленным файловым системам, например, протокол NFS. Реализация сетевого доступа возлагается на сеть управления – впрочем, возможно и выделение отдельной ***сети ввода/вывода*** специально для этих целей.

Основное достоинство этого способа – простота использования. В самом деле, легко сконфигурировать NFS таким образом, чтобы пути к домашним директориям пользователей на управляющей машине и каждом из узлов совпадали. Тогда, готовя программы и данные на управляющей машине, пользователь не должен задумываться о том, как ветвь параллельной программы «доберется» до них на узле: пути ко всем пользовательским файлам на всех машинах совпадают автоматически.

Основной недостаток этого способа связан с проблемами производительности файлового ввода/вывода. Причем дело не столько в том, что единственному файловому серверу трудно обслужить потребности многих узлов из-за ограниченности быстродействия его диска (дисков), сколько в том, что сетевому входу файлового сервера трудно обслужить заказы от многих узлов. Например, при одновременной записи данных из всех узлов возникает классическая сетевая перегрузка по сценарию «все передают в одну точку». Если пропускная способность сетевого канала файлового сервера многократно выше, чем пропускная способность каналов узлов, это может несколько улучшить ситуацию. Впрочем, для организации сетевой перегрузки типа «все передают в одну точку» вполне достаточно, чтобы суммарная пропускная способность каналов узлов была всего лишь вдвое выше, чем пропускная способность серверного канала. При этом ситуация будет тем **хуже**, чем выше производительность самого сервера по обмену с собственными дисками.

В силу очевидных удобств для использования, вытекающих из симметрии доступа к единой файловой системе, этот способ доступа к внешней памяти обычно применяется в кластерах в качестве основного, но не всегда единственного.

Использование локальных дисков

Очевидная альтернатива общему файловому серверу – использование локальных дисков узлов. Этот способ наиболее привлекателен с точки зрения производительности файлового ввода/вывода, поскольку работа ветви параллельной программы с локальными дисками узла вообще никак не нагружает ни одну из сетей, а производительность файлового ввода/вывода максимальна. В то же время, такой способ организации внешней памяти имеет целый ряд недостатков, например:

- Файлы разбросаны по разным машинам. Так, файл исходных данных, подготовленный на управляющей машине, уже не является автоматически доступным на узле: его надо туда переписать. Аналогично результаты расчетов, полученные на локальных дисках, для анализа или последующей обработки необходимо передать на управляющую машину.
- Однократно использовав для своего выполнения некоторый набор узлов, программа пользователя оказывается «привязанной» при последующих пусках именно к этому набору узлов, если только файлы, хранящиеся в узлах, необходимы для продолжения счета.

Использование распределенной файловой системы

Распределенная файловая система – это программный компонент, позволяющий организовать единую файловую систему на нескольких машинах, связанных сетью. Такими машинами могут быть как несколько специально выделенных машин-компонентов распределенного файлового сервера, так и некоторые (или все) узлы.

Распределенная файловая система способна обеспечить все логические преимущества общего файлового сервера, не создавая, по крайней мере теоретически, свойственной единому файловому серверу «точки сгущения» сетевого доступа. По производительности эта схема является промежуточным вариантом между общим файловым сервером и локальными дисками. Основные недостатки такого, на первый взгляд, привлекательного решения лежат в сфере надежности, отказоустойчивости и пригодности к администрированию. Подробнее эти проблемы будут рассмотрены ниже, при обсуждении средств управления ресурсами кластера.

Использование специализированной аппаратуры

Наиболее привлекательное с точки зрения как эффективности, так и простоты использования решение – использование аппаратуры класса SAN (Storage Area Network), специально предназначенной для симметричного доступа большого числа компьютеров к единой файловой системе с максимально высокой производительностью. При этом используется сервер, снабженный специальной сетью, используемой исключительно для доступа специально подключаемых к этой сети компьютеров к файловой системе сервера. К сожалению, цена таких решений зачастую многократно превышает цену остальной аппаратуры, из которой состоит кластер [23].

Рекомендуемые поисковые фразы:

`parallel file system`