системы и самого компьютера ему не подвластна. Тем не менее, имеется возможность повысить эффективность и этих этапов. Связана она с выбором подходящих форм алгоритма и программы, поскольку разные программы показывают разную эффективность. Принимая во внимание особенности конкретной вычислительной системы, можно попытаться найти подходящий вариант программы. Вопрос состоит только в том, на основе каких знаний осуществлять данный выбор. Все эти обстоятельства важно понять, запомнить и постоянно учитывать на практике.

ЛЕКЦИЯ 2

Большие задачи и программирование

Содержание: *интересы специалистов и программирование, предельно сложные задачи, совершенствование техники и программирование, преемственность программных наработок, переносимость программного обеспечения, отсутствие гарантий качества компиляции, простые примеры, необходимость изучения структуры алгоритмов*.

Вроде бы все ясно. Если алгоритм разработан, вычислительная система определена, то нужно лишь выбрать один из языков программирования на этой системе и можно приступать к самому процессу написания программ. Конечно, для большой задачи этот процесс потребует какого-то времени, возможно, даже немалого, могут возникнуть трудности с отладкой. Однако в нем не видны какие-то места, требующие серьезных размышлений.

Такое мнение распространено довольно широко. Можно указать две характерные группы специалистов, в той или иной степени его поддерживающие. Во-первых, это высококлассные специалисты, решающие однотипные по структуре задачи в течение длительного периода. Они настолько хорошо знают свою предметную область, что многие трудности решения больших задач преодолевают за счет выбора подходящей реструктуризации вычислительного процесса. Но подобных специалистов не так много. И, во-вторых, это специалисты, имеющие некоторый опыт решения небольших или средних задач и лишь приступающие к большим задачам. Прежний опыт им ничего не говорит о том, с какими новыми трудностями они могут встретиться. Таких специалистов очень много.

В обеих группах можно выделить специалистов, для которых большие задачи приходится решать *эпизодически*. Например, при численной проверке какой-либо частной гипотезы в процессе изучения той или иной проблемы. Если задача не требует предельного использования ресурсов вычислительной системы, то в этих случаях совсем *не обязательно* заботиться об эффективности функционирования составленной программы. Но есть немало

специалистов, для которых исключительно важно решать задачи не только максимально *быстро*, но и максимально *большого размера*. Для них сама возможность решать подобные задачи нередко оказывается зависящей от того, насколько эффективно будут работать создаваемые программы.

При любой вычислительной технике существовали и будут существовать задачи, требующие предельного использования всех ресурсов. Характерной их особенностью является очень большой объем вычислений. Чтобы решать эти задачи за приемлемое время, необходимо добиваться такого качества программирования, при котором программы должны реально выполняться со скоростью, *соизмеримой с пиковой*. По отношению к используемой вычислительной технике большими считаются именно такие задачи и именно при их решении начинают возникать проблемы с программированием.

Переход к более совершенной технике неизбежно сопровождается пересмотром сложившихся принципов программирования, особенно программирования с максимальным использованием имеющихся технических ресурсов. Под влиянием конкуренции и ряда других факторов конструктора вычислительной техники стараются выпускать новые модели как можно раньше. Штатное программное обеспечение для них к этому моменту всегда оказывается достаточно сырым и пригодно лишь для того, чтобы была возможность работать. После ввода новой техники начинается обкатка программного обеспечения и накопление опыта его использования, создаются новые языки и системы программирования, разрабатываются новые технологии решения больших задач. Все это занимает большой период времени.

Опыт освоения вычислительной техники показывает, что она меняется довольно часто. При этом совсем не обязательно, что одни и те же большие задачи на новой технике будут решаться более эффективно, чем на старой. По крайней мере, в первый период. Конечно, что-то из старого опыта переносится в новые условия, но очень многое теряется. Поэтому решение предельно сложных задач представляет процесс *постоянной адаптации* численных методов и программ к требованиям и возможностям новых вычислительных систем.

Приведем некоторые примеры. В середине 60-х годов прошлого столетия в нашей стране были широко распространены вычислительные машины типа М-20. По тем временам это были вполне современные компьютеры, обладающие производительностью около 20 тыс. операций в секунду. Однако их оперативная память была малой и позволяла решать системы линейных алгебраических уравнений с плотной матрицей порядка всего лишь 50–100. Требования практики заставляли искать эффективные способы решения систем более высокого порядка. Несколько бóльшая память имелась на магнитных барабанах. Но доступ к ней был существенно более медленным, чем к оперативной памяти. Магнитные барабаны играли тогда такую же роль, какую сейчас в персональном компьютере играют жесткие диски.

Естественно, с поправкой на объем хранимой информации. Под этот тип медленной памяти была разработана специальная *блочная* технология решения больших алгебраических задач. Она позволяла с использованием только 300 слов оперативной памяти решать системы практически любого порядка. Точнее, такого порядка, при котором матрица и правая часть могли целиком разместиться на магнитных барабанах. При этом системы решались почти столь же быстро, как если бы вся информация о них на самом деле была размещена в оперативной памяти. Созданные на основе данной технологии программы были весьма эффективны. В частности, на машинах типа М-20 они позволяли решать системы 200-го порядка всего за 9 минут. Для машин типа М-20 подобные системы являлись экстремальными задачами.

В конце 60-х годов появилась машина БЭСМ-6. Для того времени это была одна из лучших вычислительных машин в Европе. Она обладала производительностью около одного миллиона операций в секунду, т. е. была быстрее машин типа М-20 примерно в 50 раз. Среди математического обеспечения машин БЭСМ-6 были и стандартные программы для решения системы линейных алгебраических уравнений большого порядка с использованием памяти на магнитных барабанах. Ожидалось, что с их помощью можно решать большие системы, если и не в 50, то хотя бы в несколько раз быстрее. Однако практика показала удивительные результаты. Система 200-го порядка решалась за 20 минут! Прошли годы, прежде чем на машине БЭСМ-6 появились сопоставимые по качеству программы.

Заметим, что похожие проблемы являются актуальными и для многих современных компьютеров. Сравните производительность своего персонального компьютера и машины М-20. Теперь решите на своем компьютере систему линейных алгебраических уравнений 200-го порядка методом Гаусса и измерьте время ее решения. Далее сравните отношение производительностей и отношение времен решения систем для обоих компьютеров. Скорее всего, это сравнение будет не в пользу вашего компьютера. Если же система большого порядка будет решаться на персональном компьютере с использованием жесткого диска в качестве памяти или на любой распределенной вычислительной системе, то результат сравнения окажется еще хуже. Поэтому даже для самой современной вычислительной техники эффективное использование памяти большого объема остается достаточно трудным делом.

Эффективность решения любых задач зависит от качества штатного программного обеспечения, в первую очередь, компиляторов и операционных систем. Очень важно иметь достоверные прогнозы его развития, поскольку ошибки в прогнозах могут привести к значительным издержкам в смежных сферах деятельности.

Вот один из примеров. Создание машинно-независимых языков программирования привело к появлению на рубеже 50-х — 60-х годов прошлого столетия замечательной по своей красоте идеи. Традиционные

описания алгоритмов в книгах нельзя без предварительного исследования перекладывать на любой компьютерный язык. Как правило, они не точны, содержат много недомолвок, допускают неоднозначность толкования и т. п. В частности, из-за предполагаемых свойств ассоциативности, коммутативности и дистрибутивности операций над числами в формулах отсутствуют многие скобки, определяющие порядок выполнения операций. Поэтому в действительности книжные описания содержат целое множество алгоритмов, на котором разброс отдельных свойств может быть очень большим. В результате трудоемких исследований из данного множества выбираются несколько алгоритмов, обладающих лучшими характеристиками, и именно они программируются.

В разработку алгоритма всегда вкладывается большой исследовательский труд. Чтобы избавить других специалистов от необходимости этот труд повторять, результаты его выполнения надо бы сохранять. Любой алгоритм описывается абсолютно точно в любой программе. Поэтому программы на машинно-независимых языках высокого уровня как раз удобны для выполнения функций хранения. Идея состояла в том, чтобы на таких языках накапливать багаж хорошо отработанных алгоритмов и программ, а на каждом компьютере иметь компиляторы, переводящие программы с этих языков в эффективный машинный код. При этом вроде бы удавалось решить сразу несколько важных проблем. Во-первых, сокращалось *дублирование* в программировании. Во-вторых, гарантировалось, что используются *лучшие* программы. И, в-третьих, автоматически решался вопрос о *переносе* программ с компьютера одного строения на компьютер другого строения. Вопрос, который был и остается весьма актуальным. При реализации данной идеи функции по адаптации программ к особенностям конкретных компьютеров брали на себя компиляторы. Как следствие, пользователь освобождался от необходимости знать устройство и принципы функционирования как компьютеров, так и компиляторов.

В первые годы идея развивалась и реализовывалась очень бурно. Издавались многочисленные коллекции хорошо отработанных алгоритмов и программ из самых разных прикладных областей. Был налажен широкий обмен ими, в том числе, на международном уровне. Программы действительно легко переносились с одних компьютеров на другие. Постепенно пользователи стали отходить от детального изучения компьютеров и компиляторов, ограничиваясь разработкой программ на машинно-независимых языках высокого уровня. Однако радужные надежды на длительный эффект от реализации обсуждаемой идеи довольно скоро стали меркнуть. Причина оказалась простой и связана с несовершенством работы компиляторов. Конечно, речь не идет о том, что они создают неправильные коды, хотя такое тоже случается. Дело в другом – разработчики штатных компиляторов *не давали никаких гарантий*, касающихся эффективности реализации получаемых кодов. Поэтому установить эффективность использования готовой программы на конкретном

компьютере можно было только опытным путем. По мере усложнения архитектуры компьютеров разброс в эффективностях программ становился все больше. В конце концов, идея накапливать коллекции хорошо отработанных алгоритмов и эффективных в реализации программ потеряла свою практическую привлекательность. А сколько было потрачено усилий на ее воплощение! Ведь она владела умами многих математиков и программистов не менее десяти лет.

Усложнение архитектуры вычислительных систем, в частности, увеличение числа процессоров, использование кэш-памяти и др., привело к тому, что эффективность решения даже простых задач стала сильно зависеть от стиля программирования или, точнее, от формы записи алгоритма.

Рассмотрим два очень "простых" примера. Известно, что пиковая производительность одного процессора компьютера CRAY Y-MP C90 составляет 960 Mflop/s. Однако на фортранной программе

```
DO k = 1, 1000
 DO j = 1, 40
  DO i = 1, 40
   A(i,j,k) = A(i-1,j,k)+B(j,k)+B(j,k)
  END DO
 END DO
END DO
```

компьютер показывает реальную производительность всего лишь 20 Mflop/s. Тем не менее, на почти такой же программе

```
DO i = 1, 40, 2
 DO j = 1, 40
  DO k = 1, 1000
   A(i,j,k) = A(i-1,j,k)+2·B(j,k)
   A(i+1,j,k) = A(i,j,k)+2·B(j,k),
  END DO
 END DO
END DO,
```

реализующей *тот же самый* алгоритм, производительность достигает уже 700 Mflop/s. На программе

```
DO i = 1, n
 DO j = 1, n
  U(i+j) = U(2n+1–i–j)
 END DO
END DO
```

на всех многопроцессорных системах с общей памятью, на которых компилятор *сам распределял работу между процессорами*, реальная производительность при любом значении параметра n не превышала производительности одного процессора. Но на почти такой же программе

```
DO i = 1, n
  DO j = 1, n–i
   U(i+j) = U(2n+1–i–j)
  END DO
  DO j = n–i+1, n
   U(i+j) = U(2n+1–i–j)
  END DO
END DO,
```

реализующей *тот же самый* алгоритм, производительность повышается с ростом n. При кажущейся простоте этих примеров, совсем не просто объяснить, почему так сильно меняется реальная производительность в зависимости от формы записи алгоритмов. Отметим лишь, что в первом примере компилятор не смог оптимально использовать кэш-память, в третьем примере ни один компилятор не смог распознать независимые ветви вычислений.

Уже с 60-х годов прошлого столетия все наиболее мощные вычислительные системы стали создаваться как многопроцессорные. Для использования таких систем начали разрабатываться специализированные языки и системы программирования, обобщенно называемые средствами параллельного программирования. К настоящему времени их набралось очень много. Даже поверхностный анализ приводит к появлению списка из более 100 наименований [1]. В случае последовательных компьютеров и систем с малым числом процессоров такого обилия базовых средств программирования не было.

Этот факт говорит о том, что в конструировании языков и систем программирования для многопроцессорных систем появились какие-то новые трудности, которых не было раньше. Одна из них видна сразу. Многопроцессорные системы создаются, в первую очередь, с целью ускоренного решения очень больших задач. Чтобы задача решалась быстро, все процессоры бóльшую часть времени должны быть заняты выполнением полезной работы. Операции, выполняемые в один и тот же момент, не могут быть связаны информационно. Поэтому для обеспечения высокой скорости реализации программ необходимо задавать независимые ветви вычислений. Но кто или что будет это делать?

Безусловно, в идеале выделение независимых ветвей вычислений должно было бы осуществляться без участия человека. Попытки поручить выполнение такой работы компиляторам неоднократно предпринимались на первых

многопроцессорных системах. Тем не менее, здесь не удалось достичь большого успеха. Сама по себе задача анализа структуры программ исключительно сложна и во многих отношениях является NP-полной. По этой причине для ее решения в компиляторах применялись простые и, как следствие, весьма несовершенные технологии. Поэтому создаваемые машинные коды часто оказывались не эффективными. Иногда даже для внешне очень простых программ компиляторы просто не могли определить независимые ветви, как это случилось, например, на рассмотренном выше примере двойного цикла.

Стоит заметить, что все первые серийно выпускаемые многопроцессорные системы имели относительно небольшое число процессоров и общую оперативную память, обеспечивающую быстрый доступ для любого процессора. В этих условиях задача оптимизации кодов программ становилась значительно проще и разработчики компиляторов хотя бы как-то могли ее решить. Но когда стали появляться системы с большим числом процессоров и, к тому же, с распределенной памятью, технологии анализа программ оказались настолько сложными, что разработчики компиляторов окончательно отказались от некоторых видов оптимизации создаваемых кодов программ. В первую очередь, от оптимизации по числу процессоров и использованию распределенной памяти. Поскольку такая оптимизация необходима и кто-то ее все же должен проводить, забота о ней не сразу, в некоторой опосредованной форме, но была переложена на пользователей. Это означает, что явно или неявно, но при подготовке задачи к решению на любой современной многопроцессорной вычислительной системе пользователю всегда придется самому обнаруживать, указывать и использовать дополнительную информацию о независимых ветвях вычислений, распределении массивов данных по модулям памяти, организации обменов информацией между ними и, возможно, много еще о чем. Характер дополнительной информации и форма ее представления определяются особенностями архитектуры вычислительной системы и используемого языка программирования. Тем не менее, для выбора правильной стратегии освоения современной многопроцессорной вычислительной техники нужно понимать следующее.

В языках программирования через дополнительную информацию осуществляется передача компилятору для оптимизации машинного кода каких-то свойств структуры данных и связей между отдельными операциями во всей совокупности используемых алгоритмов. Никакой другой функции в языках программирования дополнительная информации не выполняет.

Как уже отмечалось, проблема переносимости программ с одной вычислительной системы на другую не решена даже для компьютеров относительно простой архитектуры. Нет никаких предпосылок думать, что она будет решена в обозримом будущем на основе создания каких-то новых языков и систем программирования. В конечном счете, на вычислительной

технике решаются задачи. И только хорошее знание структуры задачи и алгоритмов поможет решать задачи эффективно. Для больших задач это особенно важно.

ЛЕКЦИЯ 3

Компьютеры и параллельные формы алгоритмов

Содержание: *абстрактная модель последовательного компьютера, влияние последовательных вычислений, развитие параллелизма в компьютерах, концепция неограниченного параллелизма, граф алгоритма, необходимость новых сведений о структуре алгоритмов, параллельная форма алгоритма, абстрактная модель параллельной системы.*

Несмотря на огромное фактическое разнообразие, все существующие компьютеры и вычислительные системы с точки зрения пользователя условно можно разделить на две большие группы: последовательные и параллельные.

В простейшей интерпретации последовательные компьютеры выглядят следующим образом. Имеются два основных устройства. Одно из них, называемое *процессором* (центральным процессором, решающим устройством, арифметико-логическим устройством и т.п.), предназначено для выполнения некоторого ограниченного набора простых операций. В набор операций обычно входят сложение, вычитание и умножение чисел, логические операции над отдельными разрядами и их последовательностями, операции над символами и многое другое. Наборы операций, выполняемые процессорами разных компьютеров, могут отличаться как частично, так и полностью. Другое устройство, называемое *памятью* (запоминающим устройством и т.п.), предназначено для хранения всей информации, необходимой для организации работы процессора. Процессор является активным устройством, т.е. он имеет возможность преобразовывать информацию. Память является пассивным устройством, т.е. она не имеет такой возможности. Процессор и память связаны между собой *каналами* обмена информацией.