QNX: кластерные вычисления

Олег Цилюрик

В данной статье на примерах конкретных рабочих программ показано, насколько просто организовать параллельную работу нескольких сетевых узлов над единым вычислительным процессом, используя специфические особенности операционной системы реального времени QNX. Такая организация может быть использована для весьма существенного наращивания производительности вычислительных систем и применима для достаточно широкого круга практических задач. Отмечается свойственное QNX отсутствие условий для инверсии приоритетов в распределённой системе, что особенно важно для обеспечения надёжного функционирования систем реального времени.

Общие соображения

Идея многопроцессорной обработки как способа повышения общей эффективности вычислений родилась давно. Однако прежде чем попытаться распределить вычисления между N процессорами, нужно отчётливо понимать, что не любой вычислительный процесс получит какие-либо преимущества от реализации на многопроцессорных архитектурах. Для этого он должен быть достаточно «хорошо распараллеливаемым». Какие же классы задач предполагают такой вычислительный процесс? Это, как правило, задачи с вычислениями, многократно повторяемыми при вариациях некоторых начальных условий в каждом цикле. Кроме того, желательно, чтобы в таких задачах параметры последующих циклов вычисления имели бы минимально выраженную зависимость от результатов предыдущих циклов («итерационность»).

Как это ни странно, достаточно широкие классы задач оказываются в определённой мере «хорошо распараллеливаемыми». Приведём в качестве примера некоторые из них.

- Восстановление криптографированного текста с помощью всех возможных ключей шифрования и выбор наилучшего результата.
- Поиск в больших объёмах данных по ключевым признакам или по их сложным комбинациям.
- Прочностные расчёты, реализация метода конечных элементов, задачи гидро- и электродинамики сплошных сред.
- Проверка комбинаторно синтезируемых гипотез и идентификация отметок, полученных пространственно разнесёнными приёмниками, в системах радио- и гидролокации.
- Задачи баллистики.
- Обработка изображений, например идентификация дактилоскопических отпечатков или анализ аэрокосмических снимков.
- Множественное вычисление целевой функции в процедурах многомерной нелинейной оптимизации.
- Реализация нестационарного метода статистического моделирования для газовой динамики и кинетической теории газов.
- Обеспечение высокой надёжности систем баз данных.
- Большинство задач поиска вариантов в пошаговых игровых программах.

Легко заметить, что степень успешности распараллеливания обратно зависима от того, как сильно исходные данные последующих циклов вычислений зависят от предыдущих.

За годы эволюции идеи распределённой обработки сложились в различные её реализационные механизмы, каждый из которых в той или иной степени соответствует двум основным моделям (рис. 1): сильносвязанные многопроцессорные системы (системы с симметричной многопроцессорной обработкой — SMP) и слабосвязанные многопроцессорные системы (кластерные системы).

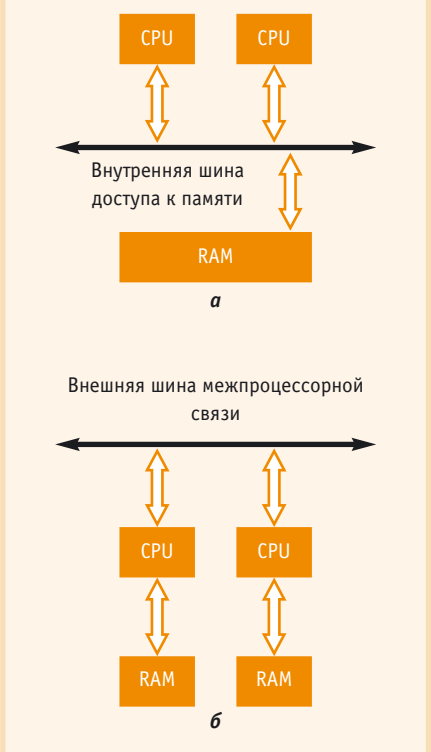

Рис. 1. Сильносвязанная (*а*) и слабосвязанная (*б*) многопроцессорные архитектуры

В SMP-системах N обрабатывающих процессоров разделяют общие поля внешних устройств и, что особенно важно, поле оперативной памяти. Для реализации этой модели необходимо использовать специализированные архитектуры взаимодействия процессоров и специальные наборы системных микросхем (chipset). В таких архитектурах оптимальным механизмом распределения работы между

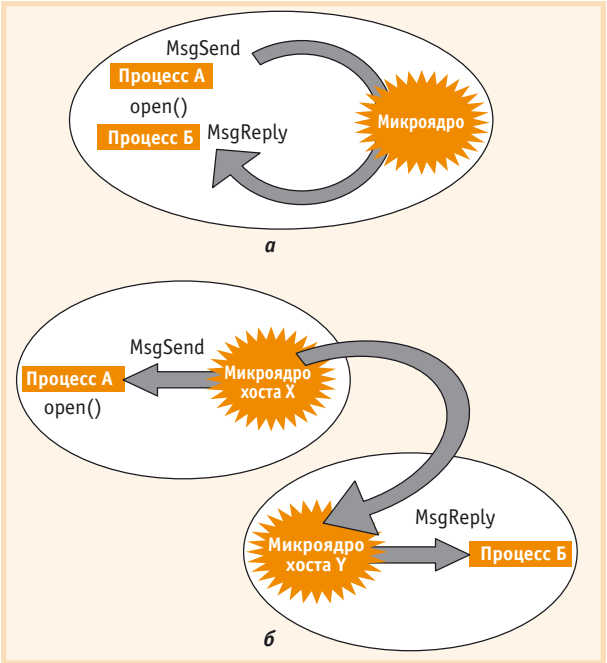

Рис. 2. Если в «микроядерной» архитектуре стандартные операции POSIX (например, open) выполняются посылкой микроядром сообщений от одного процесса к другому (*а*), ничто не препятствует тому, чтобы, используя абсолютно тот же механизм, выполнить запрос к процессу, выполняющемуся на совершенно другом сетевом хосте (*б*), — в этом принципиальное отличие «микроядерных» ОС (QNX, NextStep, Darwin и др.) от традиционных ОС с монолитным ядром (Windows, Linux и др.)

параллельными ветвями представляется разделение на уровне потоков (thread). Последовательно наблюдались массовая реализация механизмов thread в аппаратных платформах (начало 90-х), поддержка абстракций thread в операционных системах (1994-1996) и отражение их в стандартах POSIX (конец 90-х). Одной из самых существенных технических трудностей при построении таких архитектур является необходимость обеспечения когерентности данных в локальных устройствах кэш-памяти каждого из процессоров.

В кластерных системах предполагается, что каждый из узлов является типовой вычислительной архитектурой со своим процессором, оперативной памятью, каналами ввода-вывода и т.д., а кооперация узлов осуществляется через некоторые каналы передачи данных между ними. В такой архитектуре распараллеливание работ реализуется на уровне процессов, каждый из которых выполняется на своём узле вычислительной структуры.

И та и другая модель имеет как свои преимущества, так и свои недостатки, баланс которых может существенно смещаться в зависимости от класса решаемых задач. Все прочие многопроцессорные архитектуры могут рассматриваться как линейная комбинация решений, почерпнутых из этих двух моделей.

Из приведённых общих соображений уже должно быть достаточно понятно, что если SMP-структуры пригодны только для наращивания производительности системы, то кластерные системы, кроме того, могут быть использованы и для повышения «живучести» системы в применениях, где должна быть обеспечена высокая надёжность. Действительно, в N-процессорном кластере при выходе из строя любого количества хостов до (N-1) система может сохранять работоспособность (правда, обычно со снижением общей производительности) за счёт перераспределения загрузки между оставшимися хостами.

Общее описание проекта

Идея того, что QNX-хосты, объединённые сетью QNET, сами по себе уже являются полноценным многомашинным кластером, появляется у каждого, кто только знакомится с необычной и остроумной организацией операционных систем (ОС), использующих принцип микроядра и обмена сообщениями (рис. 2). Действительно,

- каждый QNX-хост в сети обладает собственным микроядром ОС;
- микроядро QNX с одинаковой лёгкостью обменивается сообщениями уровня микроядра по схеме Send–Receive–Reply как с процессами на своём собственном хосте, так и с процессами на удалённых хостах;
- стоит «нагрузить» сообщения уровня микроядра целевой информацией для взаимодействия разнесённых частей распределённого приложения... и кластер готов.

Идея «кластерности», уже заложенная в архитектуру QNX, достаточно широко используется разработчиками систем на практике, но в несколько другом аспекте: на N хостах сети QNET размещают функционально **различные** части единой системы, которые могут легко взаимодействовать между собой. Меня же заинтересовала идея использования симметричного кластера: отдельные части единой задачи разбросать между **идентичными** процессами на хостах.

Сравните простоту такой модели с необходимостью реализации взаимодействия (скажем, «над слоем» TCP/IP) между составными частями кластерного приложения в традиционных ОС! Забегая вперёд, скажу по собственному опыту, что у представленного далее проекта трудоёмкость (объём) реализации оказывается на 1 или 2 порядка ниже, чем у аналогичного проекта, скажем, в Linux или в Windows.

В предложенном к рассмотрению проекте показана возможная реализация кластерных вычислений в системе из нескольких универсальных компьютеров, работающих под управлением ОС QNX (версия, начиная с QNX Momentics 6.2) и объединённых сетью QNET. Для реализации QNET достаточно объединения хостов любого рода сетью Ethernet или даже низкоскоростными каналами с протоколом IP.

Собственно, проект содержит две задачи (программы): а) целевая задача, которую предстоит решить кластерному вычислителю; б) задача, организующая решение целевой задачи на кластере, то есть распараллеливающая её исполнение. Рассмотрим их по порядку.

Целевая задача

В качестве целевой задачи следовало выбрать задачу из числа тех, которые хорошо распараллеливаются и примеры которых приведены в начале статьи. Для демонстрации кластерной обработки я выбрал задачу криптоанализа – поиска ключа шифрования, которым криптографирован неизвестный текст, образец которого мы имеем. В целом эта затея очень напоминает хорошо известный проект Distributed.net (www.distributed.net), с той лишь разницей, что в нём анализируют криптостойкость известных и хорошо себя зарекомендовавших алгоритмов, а я использую простейший алгоритм криптографирования — XOR-свёртку с неизвестным ключом.

На сайте журнала по адресу www.cta.ru/archive/3-2004 приводится полный текст работающей программной

реализации проекта ***cluster-114.tgz*** *версии 1.14. Эта версия отличается от ранее представленных в Internet: например, скорость кодирования-декодирования увеличена более чем в 6 раз, что позволяет экспериментировать с более длинными ключами.*

Кроме того, для тех, кто не располагает установленной системой QNX, в данном архиве хранится полный исходный код проекта ***Appendix.doc*****.**

Для проведения испытаний в первую очередь понадобится собственно программа начального шифрования произвольного фрагмента текста – программа подготовки исходных тестовых последовательностей. В проекте это программа **codec**. Поскольку она осуществляет основную операцию шифрования (XOR-свёртку), а операция дешифрования для этого метода симметрична, то очень бегло рассмотрим, что и как она делает. Программа **codec** (файл **codec.cpp**) принимает три параметра в командной строке, например так:

```
#./codec s.txt d.txt key,
```

где

`s.txt` – имя исходного текстового файла;

`d.txt` – имя файла той же длины, в который будет записана результирующая последовательность;

`key` – имя файла, содержащего байтовую последовательность ключа.

Длина ключа определяется непосредственно из длины файла ключа.

Реально в прилагаемом проекте эта и все далее приводимые команды задаются в виде:

```
#time nice –n2 ./codec s.txt d.txt key
```

Это позволяет, во-первых, фиксировать системное время выполнения программы, а во-вторых, запускать задачу с приоритетом, несколько ниже принимаемого по умолчанию в ОС (в QNX это 10), что препятствует «омертвлению» системы на достаточно продолжительное время выполнения задач, значительно загружающих процессор (то есть отслеживать реакции пользователя с приоритетом, несколько выше приоритета загружающих ресурсы вычислительных задач).

В тексте **codec.cpp**, кроме обработки параметров, нет значащих операторов, за исключением двух строк, в которых и производится считывание ключа из файла в специальную структуру `key` и кодирование исходной текстовой последовательности `inp`:

```
key k( argv[ 3 ] );
char *out = k.code( inp, slen );
```

Всё, что связано со структурой `key` и процессом шифрования-дешифрования, записано в файле **coder.h.** Объект класса `key` — это байтовая последовательность ключа `(_Uint8t*)` и её длина. Далее в классе переопределён ряд традиционных операций (инициализация, присвоение, сравнения, вывод в поток и т.д.). Определена операция `rshift` — нахождение ключа, «сдвинутого» относительно исходного в сторону увеличения (напомню, ключ может быть достаточно длинным, более того – «произвольной» длины, и арифметические «+» и «–» к нему неприменимы).

Из целевых операций в классе `key` определена операция кодирования текстовой последовательности `s` длины `n`:

```
char* key::code(char *s,unsigned long n){
  char *r = new char [ n ];
  if( r == NULL ) return NULL;
  for( unsigned long i = 0; i < n; i++ )
    r[i]=(char)( (_Uint8t)s[i] ^ *(p +i%k) );
  return r;
};
```

Кроме класса `key`, в **coder.h** определена только единственная операция – тестирование полученной декодированием байтовой последовательности на принадлежность к текстовым строкам и на признак непревышения длины слов (расстояния между такими символами-разделителями, как пробел, табуляция, перенос строки и т.п.) в результирующем тексте константы

```
MAX_WORD:
bool test(const char src[],unsigned long srclen){
 const char *p=src;
 for(unsigned long i=0,j=0;i<srclen;i++,j++,p++){
  if(j > MAX_WORD )return false;
  if( *p > ' ' && *p <= '~' )continue;
  if(*p==' '||*p=='\n'||*p=='\t'||*p=='\r'){
    j = 0; continue;
  };
  return false;
 };
 return true;
};
```

Для упрощения отработки выбран именно симметричный алгоритм шифрования-дешифрования; двукратное применение программы codec должно возвращать нас к исходному виду шифруемого файла:

```
#./codec s.txt d.txt key
#./codec d.txt r.txt key
```

После таких манипуляций файлы `s.txt` и `r.txt` должны оказаться абсолютно идентичными.

Кроме того, в проекте представлена задача **single** — однопроцессорный вариант того, что мы предполагаем далее разложить на узлы кластера, то есть задачи поиска приемлемых ключей дешифрования. Для поиска ключа дешифрования используется простой линейный перебор всех возможных значений ключа заданной длины. Программа (файл **single.cpp**) выполняется с двумя аргументами, в качестве которых используются имя исходного (дешифрируемого) файла и длина ключа (точно в таком же формате будет запускаться и её многопроцессорный аналог):

```
#./single d.txt 2
```

Результаты работы этой программы будут сравниваться с результатами работы её многопроцессорного аналога **master** (они даже имеют аналогичный по форме вывод). Но самое главное, для чего данная программа просто необходима, так это для сравнения временных характеристик однопроцессорного и многопроцессорного исполнения:

```
#time single d.txt 2
#time master d.txt 2
```

Целесообразно заглянуть в текст программы **single.cpp**, чтобы позже к этому не возвращаться в более сложном многопроцессорном исполнении. Наибольший интерес представляет ядро программы:

```
key bkey(keylen),ckey(keylen);
while(ckey.next()!=bkey){
 char *out=ckey.code(inp,slen);
 if(test(out,slen)) cout<<ckey;
 delete out;
};
```

Из текста программы видно, что текущее значение ключа (`ckey`), перебираемое в цикле и применяемое к декодированию байтовой последовательности в буфере, сравнивается с начальным значением `bkey` (здесь это последовательность «\0», а в общем случае она может быть произвольной). Когда весь диапазон возможных значений перебран, процесс завершается.

Приведём некоторые итоговые комментарии, чтобы более не возвращаться к рассмотрению целевой задачи.

1. Понятие критерия принадлежности декодированного результата к интересующему нас множеству (то, что делает функция **test**) — ключевое понятие всякого дешифрования. Приведенная мной простейшая функция анализирует полученный результат по принципу: каждый байт результирующей последовательности должен принадлежать к множеству англоязычных печатных символов (латинские литеры, цифры, знаки препинания, символы пробела и табуляции, перевод и возврат каретки). Естественно, что такая критериальная функция забракует русскоязычные тексты! Более того, она может признать приемлемыми несколько результатов: один для истинного ключа и ещё несколько — для ложных, возвращающих «белиберду», но «англоязычную». Применённая в проекте критериальная функция – «байтовая», то есть она принимает или забраковывает один очередной байт без учёта какого-либо его контекста. При реальном дешифровании, после побайтового использования подобной критериальной функции, к ограниченному подмножеству отобранных «кандидатов» должна применятся «контекстная» критериальная функция (статистика литер, анализ длины слов, разделяемых пробелами или знаками пунктуации и т.д.).
2. Ранее по тексту я уже употреблял термины «символьная последовательность» и «байтовая последовательность», и ещё неоднократно эти термины будут упоминаться далее. Чем они отличаются в данном проекте? Практически ничем (я работаю с байтовым представлением символа, **unicode** в этом проекте не использую), кроме того, что к «байтовым последовательностям» нельзя применять ни одну из функций группы **str...** — внутри «байтовой» строки вполне допустим значащий символ «\0»!
3. Сразу хочу подчеркнуть, что функции **code** и **test** сделаны наихудшими с точки зрения эффективности. Функция **code** для каждой операции динамически выделяет буфер результата; кроме того, для любого значения ключа она сначала делает полную дешифрацию и только после этого результат передаётся критериальной функции **test**. Если кого-то заинтересует эффективная реализация, то это должно быть нечто следующее:

```
bool test(_Uint8t b){return b=='\n'||b
=='\t'||b== '\r'||(b>=' '&&b<=~');};
```

```
bool key::code(_Uint8t*s,unsigned long
n,_Uint8t*d,bool(t*)(_Uint8t)){
 for(unsigned long i=0; i<n;i++)if(!
*t(d[i]=(s[i]^*(p+i%k))))return
false;
 return true;
};
```

Поскольку оценивалось и сравнивалось только время выполнения, то неэффективные реализации меня более чем устраивали, так как они требовали более продолжительного и поэтому легче фиксируемого времени.

Организация кластера

Наконец мы подошли к самому интересному! Теперь проделаем аналогичные манипуляции, но распределив работу между всеми доступными хостами в сети QNET. Идея состоит в следующем:

- с помощью инициирующей программы **master** на каждом хосте сети (включая и тот, на котором выполняется **master**) запустить другую автономную программу **agent**, но передать ей только часть работы, соответствующую определённому диапазону ключей («от и до»), которые должен отработать этот хост [1];
- хорошо бы ещё, чтобы программа **master** выделяла хостам диапазон обработки ключей не «поровну», а предварительно «попросив» каждый хост сообщить производительность своего процессора и раздав работу пропорционально сообщённой производительности каждого, то есть «по справедливости» (Д. Алексеев «Получение системной информации» [1];
- программа **master** должна синхронизироваться и дождаться завершения каждой из запущенных программ **agent** (потому и разумно на своём хосте запустить **agent** – чего же ждать попусту?);
- далее программа **master** должна получить (собрать) результаты работы всех хостов и представить их на вывод, как это делала программа **single**.

Из такой постановки уже видно, что программы **master** и **agent** должны достаточно плотно кооперироваться и пересылать друг другу данные в обоих направлениях, пользуясь транспортным механизмом сообщений уровня микроядра. Какой самый простой, отработанный и высокоуровневый механизм обработки сообщений микроядра? — Конечно же, менеджер ресурса, реализующий технологию, которая в QNX отработана для написания драйверов устройств и псевдоустройств (см. «Writing Resource Manager» в документации QNX Momentics 6.2.1). То есть в качестве **agent** (это и есть псевдоустройство) мы пишем менеджер ресурса. Причём это тот не столь частый случай в практике, когда нам абсолютно не нужен многопотоковый (multithread) менеджер и нас вполне устроит менеджер однопотоковый.

Как всегда с любым менеджером ресурсов, первейшим вопросом после принятия решения о его написании является следующий: какие операции он будет обрабатывать? В данном случае я определил это так (решение было принято весьма произвольно; возможно, лучше было бы возложить весь обмен **master** и **agent** на `devctl()`, но такое предположение появилось у меня только на завершающей стадии проекта):

- задания «порции» работы **master** будет осуществлять операцией `write()`, причём в качестве буфера

данных master будет передавать программе **agent** последовательности двух ключей (начало и конец диапазона), то есть если работа идёт с ключами длины `keylen`, `write()` будет осуществлять передачу буфера длины `2*keylen`;

- ожидать получения результата от **agent** программа **master** будет на обычной блокирующей операции `read()`, а **agent** возвратит буфер длины `N*keylen`, где `N` может иметь и нулевое значение (вспомним, что из-за нечёткости критериев приемлемых результатов работы может быть 0, 1 и более);
- поскольку master должен ожидать `read()` от многих хостов, то последовательность `write()-read()` для каждого доступного хоста должна быть выполнена в отдельном потоке (забегая вперёд, отмечу, что синхронизация потока далее будет сделана на барьере `pthread_barrier_t`);
- программе **master** для работы с **agent** нужны ещё некоторые вспомогательные операции приёма-передачи информации, все они сделаны на `devctl()` и определены в comand.h:
  - `DCMD_CLUST_STS` – по этой команде **master** передаёт программам **agent** сетевое (то есть в форме `/net/host/…`) имя файла-источника для декодирования; получив эту команду, каждая программа **agent** средствами QNET загружает содержимое источника к себе в буфер и далее в источнике не нуждается (направление передачи данных – к **agent**);
  - `DCMD_CLUST_STK` — команда, которой master передаёт текущую использующуюся длину ключа (направление передачи данных – к **agent**);
  - `DCMD_CLUST_GTF` – команда запроса частоты (производительности) процессора, на котором работает **agent** (направление передачи данных – от **agent**).

С процедурами взаимодействия всё ясно, строим менеджер.

Менеджер находится в файле **agent.cpp**, и в нём нет совершенно ничего интересного (типовой dispatch-менеджер, скопированный из HELP QNX), за исключением нескольких деталей:

- менеджер **agent** регистрирует префикс пути `/dev/agent` на своём хосте;
- **agent** является не только менеджером ресурса; это fork-программа, которая своим дочерним процессом создаёт менеджер и остаётся активной, а родительский процесс благополучно завершается;
- как уже понятно, в менеджере устанавливаются три обработчика для операций `read()`, `write()`, `devctl()`; все обработчики сообщений находятся не в основном файле **(agent.cpp)**, а в отдельном файле обработчиков **(agefun.h—agefun.cpp)**.

Всё остальное уже предельно просто. Некоторых минимальных комментариев заслуживает только текст запускающей программы **master** (**master.cpp**).

Начиная работу, **master** читает QNET-каталог `/net` (это каталог по умолчанию QNET; если вы используете другой, то вам придётся несколько исхитриться). Если он не находит `/net`, то это означает, что `npm-qnet.so` просто не «подмонтирован» к `io-net` (то есть сеть QNET не запущена), и мой **master** пытается подправить ситуацию (но это — дело вкуса).

Далее программа перечитывает содержимое `/net`. Для определённости описания будем считать, что она там находит имена хостов: alpha, beta, gamma. Будем считать, что alpha соответствует имени локального хоста, на котором и запущена программа **master**.

Программа строит односвязный список хостов (`class SutList`), в котором каждый доступный хост будет представлен элементом списка (`class Sutelite`). В этом есть достаточно глубокий смысл! Первоначально я, пересчитав хосты в `/net`, создавал динамический массив хостов, но потом перешёл к динамическому списку, который позволяет более рационально использовать ресурсы памяти. Поясню почему. QNET — «устойчивая» сеть (в отличие, скажем, от IP), в которой крайне просто сигнализируется потеря канала по коду возврата `read()` (кто пытался обрабатывать разнообразные ошибки канала в TCP – меня поймёт, об UDP я просто не хочу говорить...). Последующие `open()` позволяют восстановить трафик сразу же после восстановления канала (это очень важно, и все эти свойства QNET я перепроверял и тестировал сам). А значит, при использовании динамического списка программа **master** может поддерживать в нём только «актуальные» хосты: вы

можете добавлять или убирать хосты «на ходу», но кластер будет распределять работы только среди тех из них, которые ему реально сейчас доступны.

Далее **master** выполняет команду `on` для каждого имени хоста в его связном списке с целью запустить менеджер ресурса **agent** на соответствующем хосте. Здесь есть одна тонкость: первоначально я хотел «заставить» **master** выполнить «дословно» команду `on f<host> agent`. Однако, как выяснилось, эта команда благополучно выполняется на всех хостах, кроме... собственного, локального (в нашем случае – alpha). Достаточно продолжительные эксперименты меня ни к чему не привели, и эту особенность команды `on` я пока могу относить только к «артефактам»... Поэтому, перебирая хосты, **master** анализирует их «локальность» и для своего хоста выполняет команду `on -n<host>` **agent** (ключиком отличается). Конечно, я мог бы для локального хоста выполнять просто `./agent`, но меня всегда привлекает симметричность – она обязательно потом где-то скажется.

В своей программе **master** я действительно использую операторы типа `sytem "on -nalpha agent"`, а не `spawn()`. Это выглядит грубо, но сознательно сделано для простоты и наглядности. Как от `system "on …"` перейти к `spawn()`, прекрасно описано в статье Д. Алексеева [1].

После выполнения команды `on` на каждом хосте в списке **master** появляется «драйвер» `/dev/agent`.

Программа master выполняет операцию `open()` поочерёдно применительно ко всем именам `/net/<host>/dev/agent` и сохраняет файловые дескрипторы в соответствующих элементах `Sutelite`. Всё! Связь установлена, и я могу делать с хостами всё, что вздумается!

Программа **master** по «списку» `devctl( DCMD_CLUST_GTF, …)` запрашивает тактовую частоту процессоров, на которых работают хосты, программы **agent** отвечают ей, выполнив на своих процессорах `SYSPAGE_ENTRY ( qtime )->cycles_per_sec …`

Master выполняет для всех поочерёдно `devctl( DCMD_CLUST_STS, …)` и `devctl( DCMD_CLUST_STK, …)`, сообщая всем хостам: обрабатываем файл такой-то, с такой-то длиной ключа.

После этого в цикле запускаются потоки (thread), для каждого из хостов выполняющие операцию `write()` (диапазон ключей поиска устанавливается согласно производительности хоста), и каждый поток ожидает результатов этого поиска на операции `read()`.

После синхронизации на барьере (barrier) с завершением работы всех хостов программа **master** собирает результаты и представляет их на вывод.

Всё.

Схематично хронологию распределённого вычислительного процесса иллюстрирует рис. 3.

Весь текст программного обеспечения многомашинного кластера — порядка 300 (!) строк кода (**agefun.cpp** + **master.cpp**). Всё остальное — либо целевая задача **coder.h**, либо типовой менеджер ресурса **agent.cpp** из HELP, но даже с учётом этих компонентов полный объём — всего около 690 строк.

Кластер и живучесть системы

Работая над данным проектом, я отчётливо понял, что чуть ли не текстуально тот же кластер может быть использован в качестве основы для по-

строения систем с высокой живучестью, сохраняющих функциональность при своей аппаратной деградации (выходе из строя отдельных узлов кластера).

В чём «тонкое место» приведённого в проекте кластера? Именно в программе **master**, которая является синхронизатором работы системы (напомню, во время активной работы хостов сам синхронизатор пассивен, и на его локальном хосте работает **agent** так же, как и на всех других хостах)!

Однако ничто не препятствует запуску некоторого подобия **master** на всех без исключения узлах кластера, и чтобы при этом только одна программа **master** из всех была активной, а все остальные (неактивные) пассивно выполняли бы прослушивание её команд с целью кратчайшей реакции на обнаружение отсутствия работоспособной программы **master** в системе.

Какая из программ **master** будет активной, должно определяться в результате некоторой состязательной процедуры, которая проводится при начальной загрузке системы и при обнаружении (по тайм-ауту), что текущая активная программа **master** неработоспособна. Например, каждая программа **master** на различных хостах может уведомлять прочие хосты о своей готовности; было бы хорошо, если временные задержки активации хостов заведомо различались. Здесь можно предложить рассмотреть некоторые «смешные» способы, например, поскольку кластер — сугубо сетевое творение, использовать в качестве задержки активизации хоста... MAC-адрес его сетевого адаптера (вот уж точно не совпадет с другими!).

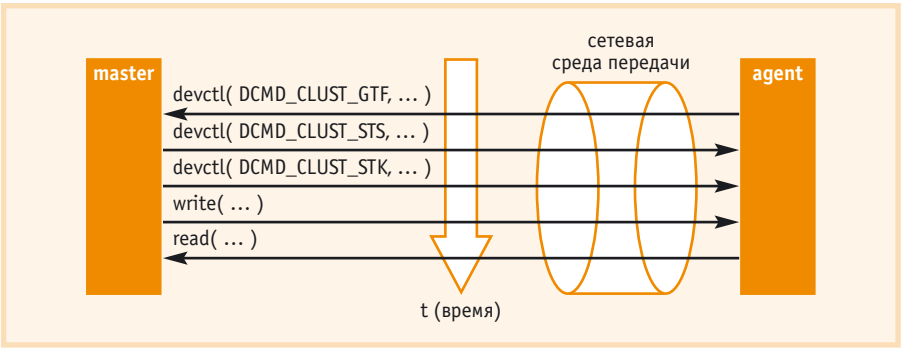

Рис. 3. Хронология распределённого вычислительного процесса

Такая динамическая кластерная система могла бы стать особенно интересной именно в QNX-исполнении, учитывая embedded-возможности этой ОС. Каждый модуль кластера вполне мог бы реализовываться на каком-либо умеренно мощном компьютере, скажем, SBC формата PC/104 (Advantech, Octagon Systems, Fastwel, Diamond Systems, Lippert и др.) с процессором AMD 5x86. Всё программное обеспечение как QNX, так и целевого кластера — в DiskOnChip. Вся взаимосвязь модуля с кластером осуществлялась бы посредством гальванически развязанного Ethernet, чтобы модули могли произвольно «на ходу» как добавляться, так и изыматься из кластера. А программное обеспечение кластера достаточно быстро и легко

```
# ./all
Coder, vers.1.14
key = <7a,30>
    0.19s real     0.01s user     0.00s system
Coder, vers.1.14
key = <7a,30>
    0.17s real     0.01s user     0.00s system
QNX home 6.2.1 2003/01/08-14:50:46est x86pc x86
Single Decoder, vers.1.14
find:   <7a,30>
find 1 keys
   13.04s real     8.56s user     0.26s system
slay: Unable to find process 'agent'
QNX home 6.2.1 2003/01/08-14:50:46est x86pc x86
Multi Cluster Decoder, vers.1.14
There are hosts into cluster:
- /net/home/dev/agent : open    -- fr. = 534636300
- /net/rtp/dev/agent : open     -- fr. = 451178900
Number of hosts in cluster = 2
home:   <7a,30>
rtp: not found
find 1 keys
    9.51s real     0.04s user     0.01s system
#
```

Рис. 4. Результаты выполнения задач на одном и двух процессорах соответственно, для случая кодирования ключом длиной 2 байта

```
# cd /root/Cluster/114/3
# ./all
Coder, vers.1.14
key = <41,30,7a>
    0.06s real     0.00s user     0.00s system
Coder, vers.1.14
key = <41,30,7a>
    0.05s real     0.01s user     0.00s system
Single Decoder, vers.1.14
find:   <41,30,7a>
find 1 keys
  154.91s real   154.57s user     0.00s system
QNX home 6.2.1 2003/01/08-14:50:46est x86pc x86
Multi Cluster Decoder, vers.1.14
There are hosts into cluster:
- /net/home/dev/agent : open    -- fr. = 534636300
- /net/rtp/dev/agent : exist    -- fr. = 451178900
Number of hosts in cluster = 2
home:   <41,30,7a>
rtp: not found
find 1 keys
   98.10s real     0.02s user     0.00s system
#
```

Рис. 5. Результаты выполнения задач на одном и двух процессорах соответственно, для случая кодирования ключом длиной 3 байта

смогло бы адаптироваться к числу имеющихся «по факту» хостов в системе.

Как это выглядит

Кратко рассмотрим результаты выполнения программ.

На рис. 4 показаны результаты выполнения последовательности кодирования и сравнительного восстановления задачами, работающими на одном и двух процессорах соответственно, для случая кодирования ключом длиной 2 байта. Время выполнения задач на двух процессорах составляет примерно 70% от времени выполнения на одном процессоре. Первоначально можно было предположить, что это соотношение составит 50%, однако оно оказывается больше из-за неизбежных в первом случае потерь времени на операции обмена.

На рис. 5 показаны результаты выполнения этих же задач при прочих равных условиях, но для ключа длиной 3 байта. Время двухпроцессорной обработки составляет здесь уже 62% от времени обработки однопроцессорной – сказывается тот фактор, что за счёт большей продолжительности основных операций уменьшается относительная доля начальных подготовительных операций.

С увеличением длины ключа время выполнения задачи растёт по экспоненциальному закону.

Замечания в заключение

Основная цель проделанной работы — показать практический способ организации многопроцессорной обработки, которая использует уникальные свойства операционных систем, построенных на базе микроядра с обменом сообщениями. Кроме того, важно было на примере реального проекта продемонстрировать, какими минимальными трудозатратами это обеспечивается. Причём в представленном проекте кластерная обработка реализовывалась, не выходя за пределы архитектуры универсальных компьютеров общего назначения.

Кластерная обработка реализуема и в иных ОС (Windows, Linux, Sun Solaris и др.). Для этого существуют программные пакеты общего применения (например, интерфейс MPICH 1.2). При этом транспортными механизмами выступают общеизвестные сетевые протоколы: TCP/IP, NetBEUI и др. В представленном проекте показано применение специализированного сетевого протокола обмена сообщениями QNET. Уже существенно позже написания основного текста данной статьи выявилась принципиальная особенность использования этого протокола, которая крайне важна для систем реального времени. Речь идёт о наследовании приоритетов и инверсии приоритетов. Детальное рассмотрение проблемы можно найти в моей статье «Инверсия приоритетов и реальное время» [1].

Дело в том, что в многомашинных архитектурах, связанных на основе стандартных транспортных протоколов (TCP/IP), каждый процесс-сателлит на удалённом хосте выполняется на своём уровне приоритета, заданном при его запуске. А это значит, что нет никакого механизма, препятствующего возникновению инверсии приоритетов на хосте, в результате чего может нарушиться строгая последовательность диспетчеризации процессов. В системах для ответственных применений это может иметь критические последствия. В описываемой же системе на основе сообщений QNET сохраняется свойственное QNX наследование приоритетов, даже если процессы клиента и сервера физически разделены между хостами. В результате вся сеть процессов, выполняющихся в составе кластерной задачи, ведёт себя как единое целое («дышит»), повышая или понижая свои приоритеты синхронно с приоритетом запрашивающей задачи (в описанном проекте это **master**, но и этот процесс может работать, обслуживая запросы более высокого уровня, например GUI-модуля). Как следствие, даже в многомашинной системе не возникают условия для инверсии приоритетов, что обеспечивает высокую надёжность функционирования всей системы.

Конечно, можно реализовать некую подобную систему наследования приоритетов и в «прикладном» сетевом слое над транспортным механизмом TCP/IP, передавая информацию о приоритете выполнения запроса в его теле. Однако такая реализация сама по себе сложна и громоздка не менее чем реализация собственно кластерного взаимодействия. В описанном же проекте это является встроенным исходным свойством механизмов QNET. ●

Литература

1. Практика работы с QNX/ Д. Алексеев, Е. Видревич, А. Волков и др.- М.: Ком-Бук, 2004.

Автор — сотрудник ООО «Лот»
E-mail: olej@front.ru