Распределение вычислений для многоядерных/многопроцессорных вычислителей на основе системы виртуальных машин языка параллельного программирования Caper

С.Р. Вартанов

ООО "Курс-АС1"

В статье рассмотрены методы распараллеливания вычислений в многоядерной вычислительной среде на платформе виртуальных машин языка императивного параллельного программирования Caper. Даны необходимые определения языка в части структур программ и средств императивного вызова псевдопараллельных легковесных процессов. Кратко описана схема работы виртуальной машины языка VM-Caper. Рассмотрены вопросы, связанные с использованием многоядерности процессоров и многопоточности ОС. Описаны решения как на уровне изменений в инструкциях вызова параллельных процессов, компиляции, так и при их интерпретациях виртуальной машиной. Представленное решение основывается на запуске комплексов виртуальных машин VM-Caper на ядрах процессора и в потоках ОС с распределением по ним псевдопараллельных процессов. Выявлены закономерности между количеством ядер и потоков ОС и производительностью вычислений. В связи, с учетом эвристик, заданы разные механизмы распределения виртуальных машин в потоках операционной системы, так же как и различные варианты распределения наборов легковесных процессов по запущенным VM-Caper. Приведены фрагменты программы, имитирующей работу клеточных автоматов, и демонстрирующие возможности подготовки и исполнения вычислений с несколькими миллионами процессов на однопроцессорном многоядерном вычислителе.

Ключевые слова: распараллеливание вычислений, язык параллельного программирования Caper, легковесные процессы, императивное распараллеливание.

1. Введение

Коротко о качествах и особенностях языка Caper. Язык параллельного программирования Caper [1,2] – язык, ориентированный на представление именно параллельных вычислений как структурно, так и разнообразными средствами императивного запуска реально- и псевдо- параллельных вычислений, отслеживания их выполнения, синхронизации и пр. Распараллеливание осуществляется на уровне процедур – базовых логических единиц всякой программы на Caper.

Язык обеспечивает модульную структуру программ с возможностями организации эффективных оверлейных вычислений, а также с возможностями миграции модулей в многомашинных ассоциациях и их исполнения на выбранных вычислителях. Соответственно, обеспечиваются возможности динамического связывания загружаемых модулей программы.

Язык поддерживает взаимодействие с исполнимым кодом, подготовленном на других языках, посредством динамических библиотек, причем как стандартными средствами ОС, так и посредством специально разработанной парадигмы и соответствующими ей служебными модулями.

Исходные коды на языке компилируются в промежуточный код (т.н. код псевдоассемблера) по аналогии с известным подходами, принятыми в языках Java, C# и др. Исполняется откомпилированный исходный код посредством виртуальной машины языка VM-Caper, обеспечивающей псевдопараллельное исполнение программ, управление событиями и пр. VM-Caper обладает встроенным динамическим компилятором (по традиции - jitter), позволяющим компилировать исходные коды в отдельных модулях динамически в процессе вычислений.

С появлением многоядерных процессоров возникла задача увеличения производительности исполнения псевдопараллельных программ. Еще в 4-ой версии языка (а каждая версия приобретала новые качественные свойства) все запущенные параллельные легковесные процессы Caper исполнялись в псевдопараллельном режиме на кооперативной основе на единственной виртуальной машине.

В данной статье представлены принципы и механизмы распределения псевдопараллельных вычислений по ядрам/потокам ОС с целью использования средств аппаратного параллелизма. С этими целями в части 2 статьи даны необходимые здесь определения языка Caper, в том числе, по структуре программ и средствам императивного параллельного вызова псевдопараллельных легковесных процессов. Кратко описана работа VM-Caper.

В части 3 рассмотрены проблемы, связанные с исполнением параллельных программ в многопоточной/многоядерной среде, и описаны решения как на уровне изменений в инструкции вызова параллельных процессов, так и на уровне виртуальной машины языка. Представлено решение на основе запуска комплексов виртуальных машин VM-Caper на ядрах процессора и потоками ОС, с распределением по ним псевдопараллельных процессов.

В 4-ой части статьи приведены ключевые фрагменты подготовки и запуска параллельных процессов в программе, имитирующей работу клеточных автоматов с более чем миллионом параллельных процессов.

Завершает статью Заключение, в котором отмечены задачи, которые программировались на языке Caper, области применения языка, и отдельные перспективы его развития. Указана и ссылка на сайт, на котором размещены некоторые подробности приложений, разработанных на Caper, описания языка и др.

2. Основные понятия и свойства языка Caper

В математической интерпретации языка блоком команд назван массив, каждый элемент которого является либо инструкцией языка, либо другим блоком команд. Блок команд является элементарной логической, именуемой и указываемой в программе единицей. Блоки команд разнотипны и являются группирующим носителем не только совокупностей инструкций, но и разного рода данных. И одновременно элементарной процедурной единицей, которая может быть вызвана на выполнение как в последовательном, так и в параллельном режиме.

Всякий программный модуль языка Caper – это файл с исходным кодом языка. По итогам компиляции исходного модуля формируется файл с объектным кодом (псевдоассемблер VM-Caper), который интерпретируется как блок (процедура) с логическим именем "MAIN". Всякий объектный модуль может стать стартовым для исполнения программы или быть загруженным в любой момент времени в процессе исполнения программы вычислений с назначаемым именем. Т.е. один и тот же модуль может быть загружен многократно под разными именами, что позволяет использовать каждую его копию специфически, исходя из текущих нужд вычислений.

С появлением многоядерных процессоров возникла проблема увеличения производительности исполнения псевдопараллельных программ. Еще в 4-ой версии языка (а каждая версия приобретала новые качественные свойства) все запущенные параллельные легковесные процессы Caper исполнялись в псевдопараллельном режиме на кооперативной основе, а в язык были внедрены средства массовых параллельных стартов [3] на основе единственной виртуальной машины (см. Рис. 1). В новой же 5-ой версии были исследованы принципы распределения псевдопараллельных вычислений по ядрам ОС с целью задействования аппаратных ресурсов на основе множества виртуальных машин (см. Рис. 2).

Блок $B = <c_1, c_2, ,,, c_n>$, где каждое c_i, $i=(1,n)$, $n < \infty$, является либо инструкцией, либо аналогичным блоком. Пусть $P = \{ b_j \}$ – конечное множество блоков, встреченных в определении программы (избегая ненужной здесь теоретизации – в конечном наборе программных модулей), и $j = (1,K)$. M_t – мультимножество, сформированное на момент t работы VM-Caper из кратных вхождений блоков из P.

На первом шаге $M_0 = \{ b_0 \}$ – состоит из единственного блока MAIN, поданного на вход виртуальной машины. Далее, с помощью инструкций языка на различных шагах вычислений осуществляются операции кратного включения (т.е. включение, в том числе, множества одноименных блоков, или их исключение): $M_x = M_{x-1} + \{ b: b \text{ из } P \}$ или $M_x = M_{x-1} - \{ b: b \text{ из } M_{x-1} \}$ с

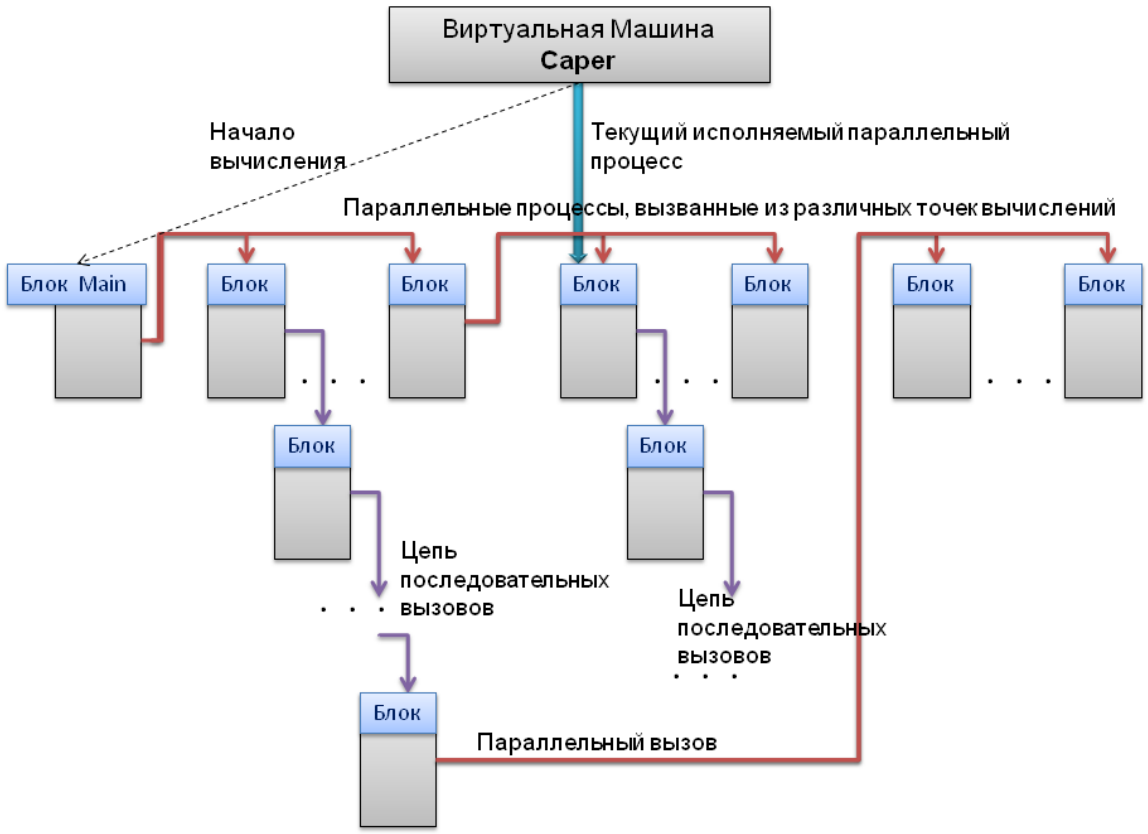

Рис. 1. Схема работы одной VM-Caper

помощью инструкций DO языка, включающих в M указания на блоки ('+' и '–' – операции объединения мультимножеств и исключения элементов из них по определенным правилам, с указанием конкретных копий блоков), и описанных ниже. Исключение из M осуществляется естественным завершением параллельных процессов или инструкциями их принудительного завершения. Если $M_x = \emptyset$, то работа VM-Caper и вычисление завершается.

Виртуальная машина языка – интерпретатор, который работает с мультимножеством M, распределяя время исполнения каждого блока из M и последовательно исполняя его команды: на шаге n выбирается очередной блок из M_n, согласно указателю текущей инструкции IP блока интерпретируется указываемая команда, счетчик IP сдвигается на следующую, и т.д., пока VM не встретит в блоке команду возврата управления в VM, или не истечет квант времени, отведенный на фрагмент работы данного блока, и которая, в свою очередь, не переключится на следующий блок из M_n (см. Рис.1). Команды возврата в VM расставляются компилятором языка руководствуясь специальными дескрипторами, определяемыми программистом, или по умолчанию. Порядок исполнения команд блоков может быть нарушен событиями, сгруппированными в классы по типу, и возникающими в программе, в VM и в операционной системе.

Заметим, что с позиций теории автоматов можно выстраивать и исследовать истории вычислений не только по памяти, но и по последовательности множеств M_t, по состояниям входящих в них блоков и др.

В языке "блок" представляется как набор инструкций языка, выделенный логическими скобками:

block <имя блока> [[**static**] (parm1, parm2, … , parmN)]
<инструкция>
…
<инструкция>
endeblock

где parm1, parm2, … , parmN – формальные параметры, а присутствие static указывает на способ передачи фактических параметров. Отметим, что существует еще несколько форм определения блоков, связанные как с режимами их исполнения, являющимися разновариантными аналогами критических секций, так и с формированием пулов данных.

Блоки в Caper для последовательного исполнения могут быть вызваны традиционным для большинства языков программирования способом через указание имени блока и перечислением через запятую фактических параметров вызова:

SomeBlock(1, 'abc', 3*x, SomeBlock2(), etc)

А вот параллельный запуск процедур осуществляется посредством разноформатного оператора DO, в котором с помощью специальных ключевых слов и дескрипторов можно указать:

- способ запуска параллельных процессов: синхронный, с ожиданием их завершения в точке вызова, или асинхронный, без оного;
- прямым перечислением имен запускаемых параллельно блоков с их параметрами, или средствами косвенного представления, в том числе, заданием массивов структур, в которых заданы указания блоков и их параметров; здесь возможны указания практически всех вариантов параллельных схем исполнения процессов: SPSD, SPMD, MPSD, MPMD по Флинну [4].
- прямым предписанием виртуальной машине исполнять параллельные процессы с их распределением по различным потокам/ядрам; описание данного механизма и является целью данной статьи.

Форма 1 (запуск перечислением блоков с параметрами):

do [**synch** | **asynch**] [**ext** | **inn**] $bl_1,bl_2,...,bl_k$ [**with** $quant_1,quant_2,...,quant_k$]
[**within** $med_1,med_2,...,med_k$]

Форма 2 (запуск перечислением массивов):

do [**synch** | **asynch**] [**ext** | **inn**] **array** $aname_1, aname_2,..., aname_K$
[**with** $quant_1,quant_2,...,quant_k$] [**within** $med_1,med_2,...,med_k$]

$quant_1,quant_2,...,quant_k$ - определяют режим исполнения параллельных процессов с квантованием времени.
$med_1,med_2,...,med_k$ - указания вычислителей, на которых должны выполняться параллельные процессы (пока не реализовано).
bl1,bl2,...,blk - список дескрипторов вызова блоков команд (процедур), имеющие следующие возможные формы:
форма MPMD:
<указатель блока >[([<параметр 1>, <параметр 2>, . . . , <параметр N>])]

либо форму MPSD:
(<указатель блока 1 > , <указатель блока 2 >, . . . , <указатель блока K >)
(<параметр 1>, <параметр 2>, . . . , <параметр N>)

либо форму SPMD:
<указатель блока> (<параметр 1,1>, <параметр 1,2>, . . . , <параметр 1,N>) …
(<параметр M,1>, <параметр M,2>, . . . , <параметр M,N>)

Примеры архитектурных форм и схем запуска параллельных процессов:
do synch proc1(x1, x2), proc1(y1, y2), proc2(), proc3, proc4(a, b, c, d) – схема MPMD.
Параллельный старт перечисленных процедурных блоков. proc1 стартует дважды с разными параметрами.

do asynch (proc1, proc2, proc3) (x1, x2, x3) – схема MPSD
Параллельный старт proc1, proc2, proc3 с общим параметрическим полем (x1, x2, x3).

do asynch proc1 (x1, x2) (y1, y2) (z1, z2)
Параллельный старт процедуры proc1 с параметрами (x1, x2), (y1, y2), (z1, z2) – схема SPMD.

Возможны смешанные записи:

do asynch (proc1, proc2, proc3) (x1, x2, x3) , proc4 (x1, x2) (y1, y2) (z1, z2), proc5 (), =>
proc2(a, b, c), (proc6, proc7) (x4, x5, x6) – схема MPMD.

Caper обладает средствами представления блоков с параметрами в виде классов данных (структур), которые компилируются и интерпретируются особым образом, и которые предназначены для предварительного, до вызова, задания фактических параметров для каждой вызываемой процедуры. Возможно создание массивов таких структур, которые могут указывать на различные процедурные блоки с различными параметрами, параллельный старт которых может быть необходим.

Форма 2 с **array** $aname_1$, $aname_2$,..., $aname_K$ предназначена для одновременного старта всех процессов, описанных в структурах из массивов $aname_1$, $aname_2$,..., $aname_K$. В целом, это делается так:

prototype <тип возвращаемого значения> <имя блока> [(<тип значения параметра> <имя параметра> {, <тип значения параметра> <имя параметра> })] определяет класс процедур, возвращающих значения указанного типа, и имеющих параметры с указанных типов. Класс процедур получает имя <имя блока> - порожденный тип. К примеру, инструкцией (всякая инструкция размещается на одной логической строке; ';*' – начало комментария на строке после инструкции; '=>' – символ продолжения инструкции на следующую физическую строку текста с возможностью записи комментария после него):

prototype <**int**> SomeBlock (<**float**> fl_coef, <**int**> db_offset, <**byte**> symbol)

порождается новый тип SomeBlock, а следующим оператором (build – оператор динамического создания составных данных по описанным переменным; ключевое слово private ограничивает область видимости переменных):

build private < SomeBlock > smBlk

осуществляется построение структуры

```
struct SomeBlock{          =>
    <int>      return,     => элемент, в который размещается значение выполнения блока
    <float>    fl_coef,    => параметр
    <int>      db_offset,  => параметр
    <byte>     bt_symbol,  => параметр
    <block>    blk_body,   => ссылка на исполнимый блок
    . . .
}
```

ссылка на которую размещается в переменную smBlk. Многоточие означает, что в структуре создаются еще несколько скрытых членов, к части из которых программист имеет доступ, а к части – нет. Далее, мы можем осуществлять заполнение структуры (':=' – символ присвоения):

```
smBlk.fl_coef      :=  1.2
smBlk.db_offset    :=  7
smBlk.bt_symbol    :=  'A'
smBlk.blk_body     :=  SomeProc  ;* SomeProc - определенный ранее иными средствами прототи-
                                 ;* пирования реальный процедурный блок с одноименными
                                 ;* с prototype параметрами и возвращаемым типом
```

и записать вызов smBlk(). При этом будет вызван процедурный блок SomeProc с заданными в структуре параметрами. При определенных условиях возможно не заполнять самостоятельно структуру, а просто записать вызов smBlk(1.2, 7, 'A'), по которому компилятор сам встроит инструкции заполнения элементов структуры.

Записью

build private < SomeBlock >[] arrSmBlk(10)

создается массив структур типа SomeBlock из 10 элементов в данном случае, к которым можно обращаться и заполнять

```
arrSmBlk[1].fl_coef       :=  1.2
arrSmBlk[1].db_offset     :=  7
arrSmBlk[1].bt_symbol     :=  'A'
arrSmBlk[1].blk_body      :=  SomeProc

arrSmBlk[2].fl_coef       :=  1.5
arrSmBlk[2].db_offset     :=  10
arrSmBlk[2].bt_symbol     :=  'B'
arrSmBlk[2].blk_body      :=  SomeProc2 ;* Возможно указать иной процедурный блок того же
                                        ;* класса
```

и т.д.
Заполнив массив или набор массивов разнотипных процедур возможно вызывать параллельный старт множества процедур, заданных во всех перечисленных массивах:

```
build private  <SomeBlock>[] arrSmBlk( 10000 ), < SomeBlock2> [5000] arrSmBlk2
. . .                  ;* заполняем массивы arrSmBlk и arrSmBlk2
do asynch array   arrSmBlk, arrSmBlk2    ;* будет запущено 15000 параллельных процессов.
```

При do asynch процедура, из которой осуществляется вызов, не останавливается в ожидании завершения вызванных процессов и продолжает исполняться вместе с запущенными, и останавливается при do synch до момента их завершения.

Существует способ обусловленного запуска для обеих форм:
if <условие> <**do**-запись>, где <**do**-запись> - запись одной из двух форм определения оператора DO.

Чем характерен подход, при котором единовременно за один такт вычисления запускается множество параллельных процессов, а не по одному, как это свойственно большинству языков программирования, так это тем, что автоматически решаются проблемы, связанные с т.н. "аксиомами" Деннинга[5].

Так, фактически, снимаются проблемы устойчивости параллельных стартов и существенно упрощается контроль за однозначностью вычислений. VM-Caper безусловно выполняется требование одной из "аксиом Деннинга": "аксиома конечной задержки", т.е. требование запуска параллельного процесса за заранее определенное и ограниченное время.

3. Производительность: проблема и решения

Как и для всех систем с параллелизмом в Caper так же актуальны вопросы времени старта новых параллельных процессов, объемов выделяемых ресурсов, времени переключения между процессами, времени терминирования и высвобождения ресурсов. В частности, всякое обращение к подсистеме управления задачами ОС для старта новых процессов (потоков) является "дорогостоящим" по времени, в то время как в Caper это время измеряется на порядки меньшими величинами.

При рассмотрении способов использования новых архитектурных возможностей учитывались особенности операционной системы (MS Windows) и фактор того, что она не гарантирует исполнение своих потоков на заявленных ядрах. Более того, "настойчивость" программы в этом вопросе может приводить к аварийному завершению приложения, что, как правило, связано с перенастройкой аппаратных кэшей.

В целом, решение напрашивалось: при превышении определенного количества параллельных процессов Caper для одной VM запускать в новом потоке ОС очередную копию виртуальной машины с очередной порцией процессов. Заметим, что управление задачами и потоками MS Windows придерживаются логики равномерного распределения нагрузки по ядрам процессора, т.е. оставалось полагаться на "здравую" логику распределения потоков, реализованную в ОС.

В этой связи проектировались следующие возможности программирования:

- автоматический запуск новой копии VM-Caper при превышении пороговой нагрузки на текущую VM по количеству обслуживаемых процессов;
- принудительный запуск указываемых к запуску параллельных процессов на новой копии VM;
- принудительное сохранение обслуживания новых параллельных процессов на текущей VM.

Так, если на данный момент вычислений работают машины V_1, V_2, …, V_p, а M_1, M_2, …, M_p – соответствующие им наборы исполняемых блоков – параллельных процессов. Если в V_j , $1 \le j \le p$, в выполняемом блоке согласно оператору DO требуется параллельный старт новых блоков L = { b: b из P }, и если мощность мультимножеств $|M_j| + |L| > D$, где D – некая вычисляемая пороговая величина процессов для одной виртуальной машины, то V_j посредством средств общего управления виртуальными машинами пытается стартовать новую копию VM-Caper V_{p+1} в новом потоке ОС (см. Рис.2).

В последней версии Caper предоставляется возможность с помощью ключевого слова EXT прямо предписывать текущей V_j осуществить запуск множества блоков L на новой копии VM-Caper V_{p+1} с $M_{p+1} = L$. К примеру,
do asynch ext proc1(x1,x2),proc1(y1,y2),proc2(),proc3,proc4(a,b,c,d)
предписывает запустить копию VM-Caper для исполнения proc1,proc1,proc2,proc3,proc4.

Однако EXT носит декларативный, а не императивный характер: управление виртуальными машинами либо выполнит это предписание, либо нет исходя из внутренних эвристик. Т.е. если запуск новой VM-Caper не приводит к улучшению производительности "по мнению" системы управления комплексом VM, то дополнительная VM-Caper не стартует, а происходит расширение набора M_j до $M_j + L$.

В частности, после ряда тестирований для MS Windows было замечено, что оптимальным числом копий виртуальной машины является величина, ограниченная (порядок) 2.4 ×N, где N – количество ядер, при наличии режима гиперпоточности ("hyper-threading") и 1.3×N без оного.

Предписание INN требует исполнения новых параллельных процессов Caper-а на текущей VM, блокируя распределение новых процессов по другим VMedu-Caper.

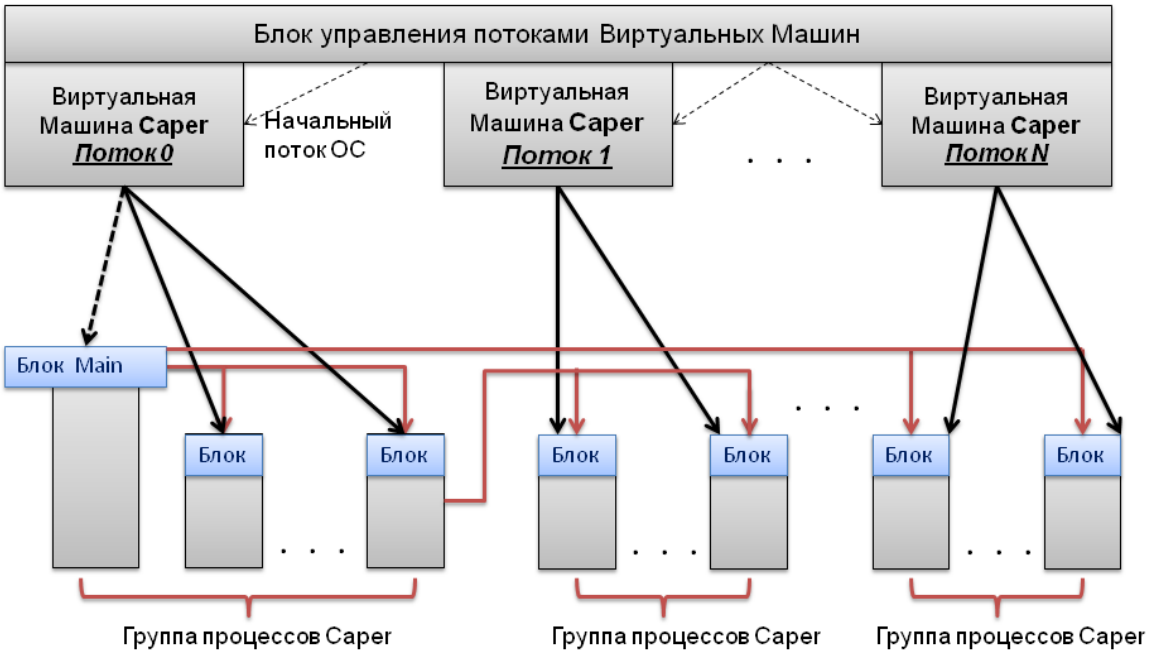

Рис. 2. Множество виртуальных машин с псевдопараллелизмом

Заметим, что к разнообразным средствам контроля и синхронизации параллельных процессов для одной VM-Caper, добавлены средства контроля и синхронизации работы множества виртуальных машин. Так, включены средства приостановки с последующим включением работы той или иной VM, ее удаления вместе со всеми процессами, ожидания событий для выбранных VM и пр.

4. Программирование параллельных вычислений на примере клеточных автоматов

В приведенном ниже примере представлен вариант обработки изображений в логике клеточных автоматов с назначением каждому пикселу – клетке процедуры – автомата, который будет изменять значение пиксела исходя из значений его окрестности.

Для этого определяется прототип процедуры, изображение фрагментируется исходя из количества ядер процессора и свойства гиперпоточности (к примеру, в случае восьми ядер с гиперпоточностью изображение делится на 19 фрагментов строк), для каждого фрагмента формируется массив процедур на основе прототипа с указанием для каждой процедуры координат обрабатываемого пиксела. При этом процедуры могут отличаться по способу обработки окрестностей.

По каждому фрагменту – набору строк изображения, кроме первого, и соответствующему ему массиву структур стартует собственная виртуальная машина. Что касается первого фрагмента, то он по требованию INN исполняется на текущей VM.

```
* Прототип с двойственной интерпретацией: интерпретируется компиляторм
* и как описатель класса процедур, и как класс-структура, членами которой
* являются формальные параметры процедуры.
* '=>' - знак продолжения логической строки на следующую физическую;
* '*' -знак начала комментария на строке
* В <…> указываются типы данных; жирным выделены базовые типы
prototype <word> CellProc(     =>
          <int>  posY,         => позиция клетки по вертикали
          <int>  posX,         => позиция клетки по горизонтали
          <int>  rank,         => ранг окрестности
          <var>  value,        => текущее значение пиксела
          <var>  oldValue,     => старое значение пиксела
          <int>  myStepCounter => количество шагов итерации
                          )

* Структура, содержащая массив описателей клеточных процессов для фрагмента
* изображения со строки lineFrom до строки (включительно) lineTo
struct arrCellsDesc{  =>
          <CellProc>[] arrOfCells, => массив клеток
          <int> lineFrom,          => начиная с номера строки картинки
          <int> lineTo,            => по номер строки картинки
          <int> processesNumber,   => количество процессов
          <int> startVariant       => здесь - номера потока-ядра
                  }

* imgWidth – ширина и imgHeigth – высота изображения
* imgArr   - битовое поле изображения
* processorsNumber – количество допустимых  потоков с VM и одновременно
* количество фрагментов строк изображения
* sizePart - количество строк фрагмента: imgHeigth/processorsNumber

* строит масСив размером processorsNumber описателей массивов клеток
build private <arrCellsDesc>[] arrDesc( processorsNumber )

* Для всех ядер готовим массивы структур-описателей параллельных процессов
* Caper-а и стартуем поток с виртуальной машиной для этой группы процессов
i := 0
while (i+=1) <= processorsNumber
  from := (i-1)*sizePart                ;* для строк с from
  to   := min( imgHeigth, from+sizePart );* до to;

* descNum - количество параллельных процессов для фрагмента
  descNum := (to - from)*imgWidth
```

```
  arrDesc[i].processesNumber := descNum

* создать массив описателей клеточных процессов фрагмента строк
  build private <CellProc>[] arrOfCells( descNum )

* ссылка на него сохраняется в структуре описания потока с VM
  arrDesc[i].arrOfCells := arrOfCells

* инициализировать массив (процедура не описана)
  ArrOfCellsIni( arrOfCells, from, to )

  switch i
    case 1    ;* если первый поток с VM, исполняющей текущий процесс
      arrDesc[i].startVariant  := 1  ;* указать внутренний номер потока
* по массиву описателей запустить на текущей VM клеточные процессы и про
* должить
      do asynch inn array arrOfCells
      break

    case  ;* аналог default: для потоков с остальными номерами (2 и выше)
      arrDesc[i].startVariant  := i

* запустить новую копию VM-Caper и клеточные процессы
* Управление VM-Caper может не выполнить требование и запустит процессы на
* текущем ядре
      do asynch ext array arrOfCells

* по asynch текущая процедура продолжает выполняться вместе с запущенными

   ends  ;* конец switch

   allProcNumber += descNum ;* суммируем количество запущенных процессов
endw
```

В примере исходного кода не приведены средства контроля при синхронной или асинхронной интерпретации собственно клеточного автомата, хотя об их существовании можно догадаться по члену myStepCounter в прототипе CellProc. При синхронной интерпретации каждая процедура формирования нового состояния клетки (пиксела) фиксирует в myStepCounter номер осуществленного шага итерации, что позволяет самой же процедуре средствами Caper перевести себя в состояние ожидания до завершения шага итерации всеми прочими процессами, привязанными к другим клеткам.

Возможны и варианты, отличные от приведенного в примере, по подготовке и запускам параллельных процессов. Так, можно все планируемые параллельные процессы (а не порциями, как в примере) представить в едином массиве arrOfCells, запустить их без прямого предписания с помощью

do asynch array arrOfCells

положившись при этом на разбиения массива на порции для нескольких VM-Caper исходя из внутренней логики управления виртуальными машинами. Caper позволяет программисту самому устанавливать размер порций для каждой VM.

В качестве тестовых данных – клеточных пространств - использовались изображения разных размеров, в том числе размером 1024x1024, где каждый пиксел рассматривался в качестве клетки. Т.е. запускалось одновременно 1048577 процессов (один - управляющий) при 2 Гб адресуемой оперативной памяти на восьмиядерном процессоре с режимом гиперпоточности. Распределение процессов осуществлялось равномерно между14-ью – 26-ью копиями VM-Caper, запускаемых потоками ОС (большее, чем 18-20 количество потоков приводило к заметному падению общей производительности). Потенциал же количества параллельных процессов Caper при отмеченном объеме оперативной памяти оценивается в 1.5-1.6 миллиона.

5. Заключение

Итак, в статье рассмотрены проблемы стартов больших массивов легковесных псевдопараллельных процессов. Рассмотрены и описаны решения на основе использования набора виртуальных машин языка Caper с возможностями императивных и декларативных инициирований разбиений массивов параллельных процессов. В язык имплементированы средства контроля и синхронизации не только параллельных процессов, но и виртуальных машин, реализующих их в различных потоках ОС.

Приведен фрагмент исходного кода, демонстрирующего простоту организации параллелизма средствами языка.

Язык Caper применялся в приложениях, связанных с несколькими классами задач. В частности, в области информационно-поисковых систем, при решении задачи Эратосфена на основе параллельных алгоритмов, но, в основном, в области обработки изображений и распознавания образов на основе параллельного алгоритма кластеризации [6].

С подробностями свойств языка программирования Caper и с приложениями, реализованными на нем, можно ознакомиться на сайте http://course-as.ru/co_caper.html

Наконец, при переводе Caper на 64-битовую платформу и создании определенных модификаций языка, ускоряющих исполнение программ, можно ожидать решений со многими миллионами параллельных процессов, исполняемых на одном PC. В планах и развитие VM-Caper для исполнения программ в многомашинной среде (кластеры, обычные вычислительные сети и др.).

Литература

1. Вартанов С.Р. Язык программирования CAPER. Препринт 97-5. Национальная Академия Наук Украины, Институт Кибернетики им. Глушкова. Киев, 1997, 27 с.
2. Vartanov S.R. On Parallel Programming Language Caper. Lect. Notes in Computer Sci., HCPN-2001, P. 565-568. DOI 10.1007/3-540-48228-8_61
3. Vartanov S.R. A Mass Parallel Starts In Parallel Programming Language Caper. В сб.: Информационные технологии и управление. Том 4-1, 2006, С. 10-18.
4. Denning P.J. On the determinacy of Schemata. – In: Rec. of the Project MAC Conf. on Concurrent Systems and Parallel Computations. Wood Hole (Mass), 1970, June, P.143 – 147.
5. Flynn M.J. Toward more efficient computer organizations. – In: AFIPS Conf. Proc., vol. 40: Proc. Spring Joint Comp. Conf., 1972, P. 1211-1217.
6. Vartanov S.R. Parallel Programming Methods in Caper Language and its Application in Image Processing. In: SCI2002/ISAS2002, Orlando, USA, 2002, vol. XI, P.1059-1071