Система для распределённых вычислений RiDE

Михаил Олегович Бахтерев <mike@0xfb.imm.uran.ru>
Павел Александрович Васёв <pavel.vasev@gmail.com>
Илья Алексеевич Альбрехт <ilya.albrekht@gmail.com>
Алексей Юрьевич Казанцев <ajk.xyz@gmail.com>

V Международная научная конференция ‘Параллельные Вычислительные Технологии’, Москва 2011

Решаемые RiDE проблемы

- Задача согласованного использования неоднородного вычислительного оборудования в распределённой системе является довольно трудоёмкой. Программисту может потребоваться нетривиально скомбинировать вычисления на различных процессорах при помощи:

  `libdevice + OpenCL + libNUMA + OpenMP + (MPI | SHMEM) + GRID`

  `libdevice` – например, библиотека для доступа к специально созданной для вычисления структуре на ПЛИС (допустим, FFT).
- Ещё сложнее, когда данные нерегулярны (web-серверы с интенсивной загрузкой).
- Поэтому цели:
  - снизить до минимума ручное управление памятью и передачами данных, желательно обойтись вообще без системных вызовов;
  - получить инфраструктуру, в которую можно однообразно вписывать различные устройства, в том числе и ПЛИС.

Желанные дополнительные технологические возможности

1. Максимальная эффективность:
   - учёт распределения данных по системе;
   - избежание ненужного копирования данных, особенно при запусках на NUMA модулях.
2. Устойчивость к отказам и отключению оборудования.
3. Распространение расчёта на подключаемое оборудование.
4. Работа в больших, подобных BOINC, конфигурациях с устойчивостью к ненадёжным хостам.

2 и 3 могут обеспечить приостановку и восстановление вычисления.

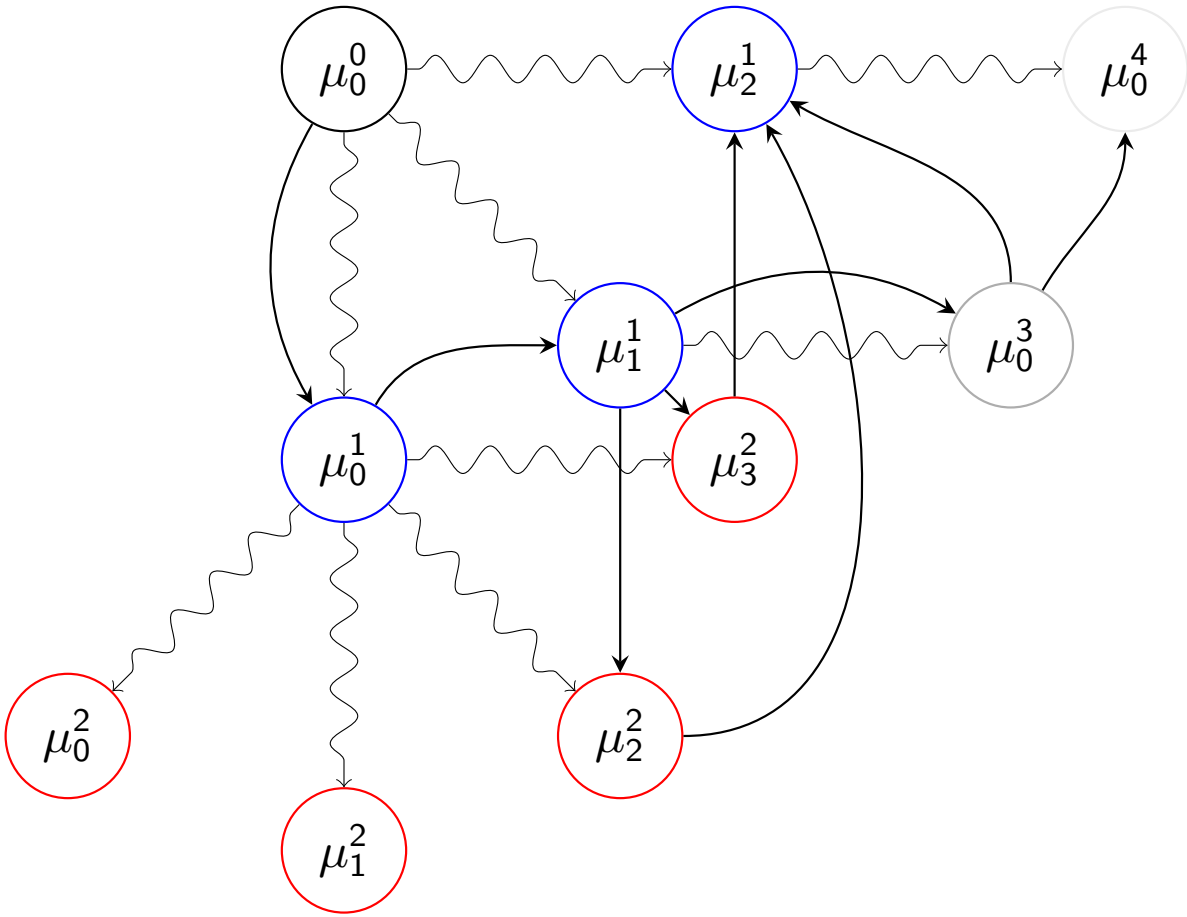
Важно подобрать правильный API

Правила или μ-замыкания (μέλλον):

- формально – структура из двоичных строк $(S_{ij};\ s_i;\ c_k;\ R_l;\ r_l)$
  - S_{ij}: имена исходных данных, μ-замыкание активируется при $\bigvee_j \bigwedge_i \delta(S_{ij})$, $\delta(n) \vdash$ ‘сформирован блок данных с именем n’ (сложно сформулировать точно: время не линейно; через существования одной из цепочек активации, возможно, здесь решётка (?));
  - c_k: описание вычислительных кодов;
  - R_l: имена блоков данных, которые будут сформированы в результате успешной активации μ-замыканиями;
  - s_i и r_l: служебные, локальные имена, для связывания данных с программами из $\{c_k\}$;
- в коде – например, такая синтаксическая конструкция

```
R.0 R.1 = (S.0.0 S.0.1 S.0.2) (S.1.0 S.1.1 S.1.2) :
  r.0 r.1 = (linux.'gcc -O P p.c && ./P' opencl.'pkernel.cl')
    s.0 s.1 s.2
```

M-замыкания в деле



Ребро в графе потока данных: (R_l, μ_k^n)

Освобождение ресурсов

В подобных системах проблемой может быть освобождение памяти.

- Каждый блок данных может понадобиться в неизвестном будущее (а какое-то будущее вычисления известно). Классическая сборка мусора не сработает.
- Вкладываемые друг в дружку области видимости (аналогичные стеку) излишне ограничивают варианты потоков данных.

В RiDE проблема решена при помощи выполнения системной процедуры unlink с продолжением:

```
() = S.0 S.1 S.2 S.3 : system.'unlink S.0 S.1 S.2 S.3;
  S.0 S.1 : s.0 s.1 = newstep'
```

Это, кроме решения проблемы с освобождением памяти, позволяет экономно и эффективно автоматически разворачивать циклы:

```
while(B.1 = V.1 V.2 : b.1 = 'cmp <' v.1 v.2) (
  V.1 = ...;  V.2 = ... )
```

Умножение матриц: μ-замыкания схемы вычисления

- загрузка и разбиение матриц на блоки

```
pR.0 : r = 'zeromtx N M';
A : a = home.load; pA.i.j = A : p = pick a;
B : a = home.load; pB.j.i = B : p = pick a;
```

- получение произведения двух блоков:

```
pmR.i.j.k = pA.i.k pB.k.j : r = mul a b;
```

- инициализация и запуск процессов суммирования для элементов $R_{i,j}$:

```
pR.i.j.0 = pmR.i.j.0 : a = system.link b;
pR.i.j.(k + 1) = pR.i.j.k pmR.i.j.(k + 1) : r = add a b;
```

- сбор результатов:

```
pR.i.j = pR.i.j.bdimK : a = system.link b;
pR.(i * j + 1) = pR.(i * j) pR.i.j : r = collect a b;
R = pR.(bdimM * bdimN) : a = system b
```

- структуры данных могут остаться и распределёнными.

Умножение матриц: код mul, язык C99

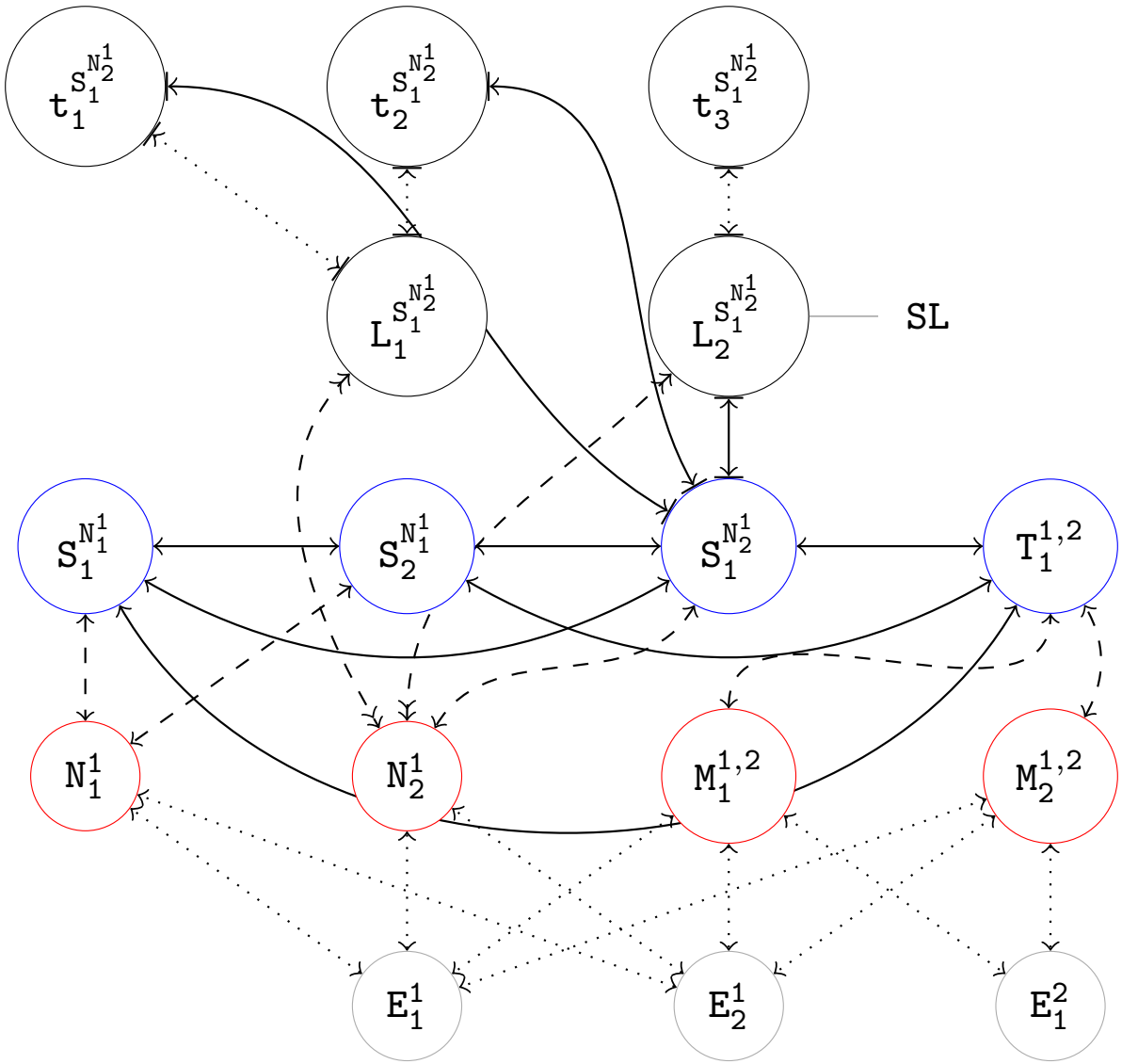
- ```
  int main(int argc, char** argv) {
    mblock* a = rget("a")->data;
    mblock* b = rget("b")->data;
    int rsz = mbsize(a->rows, b->cols);
    mblock* r = malloc(rsz);
    if(!r && a->cols != b->rows) { exit(error); }

    r->rows = a->rows;
    r->cols = b->cols;
    int M = a->cols;
    for(int i = 0; i < a->rows; i += 1) {
      for(int j = 0; j < b->cols; j += 1) {
        r->vals[i * M + j] = 0;
        for(int k = 0; k < M; k += 1) {
          r->vals[i * M + j] =
            a->vals[i * M + k] * b->vals[k * M + j]; } } }

    rput("r", r, rsz); return 0; }
  ```

- Вычислительный код может быть написан, оптимизирован, и повторно использован; похоже на идеологию UNIX: утилиты, файлы и оболочка
  ```

Архитектура



Основные необходимости

- Нужно отслеживать множество событий публикаций μ-замыканий и данных, возникающих по всей распределённой системе.
- Нужно принимать решение о активациях μ-замыканий и запуске задач.
- Централизованное решение простое (уже создан прототип), но:
  - потенциальное бутылочное горлышко;
  - единственная критическая точка отказа системы.

Формулировка в виде задачи поиска

M-замыкание можно рассматривать как поисковый запрос. Выполнение программ при активации μ-замыканий приводит к публикации данных: к обновлению массива данных для поиска; и к публикации новых замыканий: к обновлению множества поисковых запросов. Как осуществлять поиск?

- DHT, как решение поисковой задачи, имеет недостатки:
  - каждое имя блока данных в запросе и при публикации нужно обрабатывать отдельно, нет пакетного режима;
  - большие накладные расходы.
- Поисковый робот Echo:
  - узлы – клиенты (как браузер в Яndex) и сайты (как http-сервер) для поисковой машины;
  - возможен пакетный режим;
  - возможны сложные запросы;
  - возможна балансировка и планирование процедур поиска;
  - возможно обойтись без робота, используя некоторые из p2p-технологий, но робот удобен в размышлениях.

Основная проблема поиска

- Поиск относительно хорошо работает в рамках одного кластера. Но мы стремимся к работе в многокластерных конфигурациях.
- А самая сложная поисковая задача: проверить корректность для μ-замыкания $(\cdot;\ \cdot;\ \cdot;\ R_l;\ \cdot)$. Оно не должно разрушать граф потока данных. Дуги в этом графе суть (R_l, μ^i_j), а это свойство может быть разрушено, если одно из R_l окажется не уникальным во всём вычислении, что может произойти в случае ошибки.

Достижение инварианта

Задача поиска решается, если мультиузлы, выполняя проверочный поисковый запрос, пытаются достичь состояния, когда все мультиузлы знают, что все мультиузлы в графе мультикластера зелёные, то есть, подтверждают корректность μ-замыкания. Сложные случаи в доказательстве корректности:

- отключение кластера, что требует ввести в рассмотрение кластер-0, содержащий процесс – терминал пользователя;
- ввод кластера в вычисление.

Это всё похоже на:

- on-the-fly алгоритм сборки мусора, предложенный Дейкстрой, Лампортом et al;
- протокол OSPF.

Светлое будущее

Прототип подтвердил адекватность модели, за ним, естественно, следует продолжение:

- небольшая доработка семантики μ-замыканий $(S_{ij};\ \cdot;\ \cdot;\ R_l;\ \cdot)$:
  - автоматическая очистка памяти на основе escape-анализа;
  - разделение автоматически генерируемых уникальных идентификаторов и текстовых меток данных, поиск по последним;
  - это всё как оптимизация (снижение вычислительной сложности процедур поиска), так и новые возможности по управлению потоком данных;
- один узел и множество областей запуска;
- однокластерная конфигурация со множеством узлов;
- мультикластерная конфигурация;
- вычислительная среда класса BOINC;
- большая распределённая файловая система с поддержкой событий (полезно для high load);
- или даже :) internet - 2 с векторным, распределённым, гипертекстовым FIDO.