基于 pthreads+queue 实现 spmc channel 提供给应用程序的接口的设计文档

hfcao[1]

中国科学技术大学 计算机科学与技术学院

December 27, 2016

[1]caobenzhi0915@gmail.com

目录

1 使用 spmc channel 编写的应用程序用到的接口和数据结构

1.1 dedup-DetMP 和 DMR 等应用程序中用到的接口

1. 在我们使用 spmc channel 来编写 dedup,DMR(DMR 是我们课题组做的一个确定性的 MapReduce) 等应用程序时，均使用到了 spmc channel 提供的一些接口。这些接口总结如表1所示。

注:在 spmc channel 的实现中，表中列出的 spmcchan_t 的类型为 int，spaceid_t 和 tid_t 的类型均为 uint8_t。这里我们记使用 spmc channel 实现的 dedup 为 dedup-DetMP, 用 pthreads+queue 实现的 dedup 为 dedup-pthreads。

表 1: dedup-DetMP 等应用程序使用到的 spmc channel 提供的接口

接口名称	接口描述
void spmc_init(int nthreads)	初始化 SPMC 环境，nthreads 指明了内部创建的 space 的数目。
void spmc_destroy()	这个接口用于注销 spmc 环境。
spaceid_t thread_alloc(tid_t gtid)	分配一个相对编号为 gtid 的线程
tid_t thread_start(spaceid_t idx, void *(*fn)(void *), void *arg)	派发一个 space 编号为 idx 的线程去做 fn 任务，arg 是传递给 fn 函数的参数。
void thread_join(spaceid_t idx)	等待 space 编号为 idx 的线程完成任务
int chan_alloc(int nino)	分配 nino 个 channel, 返回分配的 channel 的编号
int chan_setprod(spmcchan_t cino, spaceid_t dsid, bool cons)	将相对编号为 dsid 的线程设置为编号为 cino 的 channel 的生产者，cons 表示当这个线程写完之后是否成为它的消费者。
int chan_setcons(spmcchan_t cino, spaceid_t dsid)	将相对编号为 dsid 的线程设置为编号为 cino 的 channel 的消费者
chansize_t chan_send(spmcchan_t cino, void *buf, size_t nbyte)	将起始地址为 buf 的数据发送到编号为 cino 的 channel 中，发送的消息大小为 nbyte 字节。
size_t chan_recv(spmcchan_t cino, void *buf)	从编号为 cino 的 channel 中接收消息，接收的消息放在 buf 开始的内存中。

2. 对于 DMR 而言，除了使用了表1列出的接口外，它还额外使用了表2列出的接口，使用这些接口的原因见表中的接口描述部分。

表 2: DMR 额外向应用程序提供的接口

接口名称	接口描述
void spmc_set_copyargs(int flag, int sz)	应用程序调用这个接口，以向 DMR 表明某个阶段的线程产生的 key 或者 value 指向私有堆面阶段的线程需要 key 或者 value 指针指向的内容。由于 DMR 底层是基于 DetMP 来实现 DetMP 中每个线程是私有的，为了让某个线程可以看见另外一个线程更新的堆变量，我们 这个接口。
void spmc_set_shareargs(void *addr, int sz)	应用程序调用这个接口，以向 DMR 表明某个全局变量在 MapReduce 过程中会发生改 MapReduce 中派发的线程会更新这个全局变量)，而主线程在 MapReduce 调用结束后需要 个全局变量的值。DMR 底层基于的 DetMP 使用了进程模拟线程的方式来达到空间隔离， 个进程更新的全局变量会变成私有的区域，而父进程看不见。为了让父进程可以看见子进程 值，我们提供了这个接口。

2 基于 pthreads+queue 实现 spmc channel 提供的接口

我们基于 pthreads+queue 来实现 spmc channel 的思路为 ：

代码 1: data struct of queue

```
typedef struct queue {
  int head, tail;//head and tail of queue.
  void ** data;//存储数据项，每个数据项为指针，即 void* 类型。
  int size;//队列的大小
  pthread_mutex_t mutex;//和 empty,full 一样，均用于控制线程的并发访问
  pthread_cond_t empty, full;
}queue_t;
```

- 定义一个全局的数据结构来保存 queue 和线程 id 等信息。
- 复用表1和表2中提供的接口，但是给出基于 pthread+queue 的实现。

下面先给出 queue 的数据结构定义和它提供的接口、保存 queue 和线程 id 信息的数据结构定义，然后给出 spmc channel 给出的接口和 queue 给出的接口的映射关系。

2.1 queue 和保存全局信息的数据结构定义以及 pthreads+queue 提供的接口

queue 的数据结构定义： queue 的数据结构定义如代码1所示。队列的每个元素为指针类型，即 void * 类型。

queue 提供的接口： queue 提供的接口如表3所示，主要操作为队列的初始化，入队，出队。

表 3: queue 提供的接口

queue 提供的接口名称	描述
void queue_init(struct queue * que, int size)	初始化一个容量为 size 的队列 que。
void *enqueue(struct queue* que, void *addr)	addr 为指向数据项的指针，enqueue 将 addr 指针存放到队列 que 中。
void *dequeue(struct queue * que)	从队列 que 中取出一个元素，返回值是一个指向数据项的指针。

保存线程和队列信息的数据结构定义　代码2列出了保存线程信息和队列信息的数据结构的定义。在具体实现时，要定义 global_info_t 类型的全局变量，记为 global_vars。global_vars 中的每个 queue 和 DetMP 中的 channel 是一一对应的，具体表现为编号为 cino 的 channel 对应着 global_vars.que[cino] 队列。同时,DetMP 的中每个线程的 spaceid 和 pthreads 情况下的 pthreadid 也是一一对应的，具体表现为 DetMP 中编号为 spaceid 的线程对应着 pthreads 中 global_vars.tids[spaceid] 线程。

spmc DetMP 中定义的数据类型的转换： spmc 中定义了一些数据类型，对于这些数据类型的处理如下:

- spmcchan_t: 使用宏定义让其变成 int 类型。#define spmcchan_t int。
- tid_t: 使用宏定义让其变成 pthread_t 类型。 #define tid pthread_t。

代码 2: ‘保存线程和队列信息的数据结构‘

```
typedef struct global_info {
  pthread_t* tids;//保存创建的所有线程编号
  int nth;//记录当前分配的线程数目
  struct queue *que[MAX_QUEUENUM];//保存创建的所有队列信息,
  //MAX_QUENUM 是一个宏, 它表示为分配的队列的最大的数目，初始值为 1024
  int nqueues; //que 数组中已经分配的队列的编号
}global_info_t;
```

2.2 基于 pthreads+queue 实现 DetMP 和 DMR 提供的接口:

我们可以保证原来使用 DetMP 进行编写的代码不变的情况下，再额外添加一个 DetMP-pthreads.c.c 文件，DetMP-pthreads.c.c 文件中对 DetMP 提供的接口给出了 pthreads+queue 的实现。表4列出了对于表1,2给出的接口怎么基于 pthreads+queue 来实现。

表 4: 基于 pthreads+queue 实现 DetMP 和 DMR 提供的接口

spmc channel 提供的接口名称	用 **pthreads+queue** 实现时对应的转换
void spmc_init(int nthreads)	接口不变，内部实现为初始化全局变量 global_vars，主要包括根据传入的参数 nthreads 给 t 员变量开辟大小为 sizeof(pthread_t)*nthreads 的空间，将 nth 和 nqueues 置为零。global_va 见2.1的介绍。
void spmc_destroy()	接口不变，内部实现为注销在 spmc_init 接口中初始化的全局变量。
spaceid_t thread_alloc(tid_t gtid)	接口不变，接口的语义仍然返回线程的相对 id。实现时，返回 global_vars 中的 nth 的当前值 pthread+queue 实现获得的线程编号和使用 spmc DetMP 获得的线程编号是相等的。
tid_t thread_start(spaceid_t idx, void *(*fn)(void *), void *arg)	转换成 pthread_create，具体可以使用宏定义。如 #define thread_start(pthread_create(&(global_vars.tids[a]),NULL,b,c)
thread_join(spaceid_t idx)	转换成 pthread_join，具体的实现可以用宏定义 #define thread_join(a pthread_join(global_vars.tids[a], b)
int chan_alloc(int nino)	内部返回一个队列的编号，即其在 global_vars 中的 que 数组中的编号。队列的编号和 channe 号是相等的。
int chan_setprod(spmcchan_t cino, spaceid_t dsid, bool cons)	接口不变，内部实现为空.
int chan_setcons(spmcchan_t cino, spaceid_t dsid)	接口不变，内部实现为空即可
chansize_t chan_send(spmcchan_t cino, void *buf, size_t nbyte);	接口不变，接口的实现中将 buf 的数据放入到编号为 cino 的队列 (即 global_vars 的 que[cino] 中，具体操作可以调用 enqueue 来完成操作。
size_t chan_recv(spmcchan_t cino, void *buf);	内部实现为从全局编号为 cino 的队列 (即 global_vars 的 que[cino] 队列) 中取数据到 buf 中，以使用宏定义，如 #define chan_recv(a, b) (b = dequeue(global_vars.que[cino]))
void spmc_set_copyargs(int flag, int sz)	接口不变，内部实现为空即可。因为在 pthreads 共享内存的情况下，从堆中分配出来的变量 的，而不像 DMR 的情况下是私有的。
void spmc_set_shareargs(void *addr, int sz)	接口不变，内部实现为空即可。因为在 pthreads 共享内存的情况下，全局变量是所有线程都可

2.3 基于 pthreads+queue 实现的参数设置

1. 目前初始化队列时，一个队列的容量设置为 1024*1024*sizeof(void*)。这个值为 dedup-pthreads 版本中设置的值。在做 dedup-DetMP 和 dedup-pthreads 对比实验时，应该 DetMP 中 channel 的大小和队列的大小设置成相同的值。

代码 4: Heap Operation Pattern

```
DataProcess(...){
  ...
  anchor = (char*)malloc(...);
  chan_send(args->w_ports[qid], anchor, p-anchor);
  ...
  free(anchor);
}
FindAllAnchors(...){
  ...
  chan_recv(args->r_ports[qid], anchor);
  ...
}
```

2.4 基于 pthreads+queue 实现 detmp 接口在实现上遇到的问题

问题 1.chan_send 的实现需要拷贝发送者发送过来的数据。

虽然我们用 pthreads+queue 来实现 detmp 的接口时，线程之间是共享内存的。但是在 chan_send 的内部实现时仍然需要从堆中分配出空间，并将发送者发送过来的数据拷贝到这个分配的空间中。chan_send 的代码如清单3所示。

代码 3: Implementation of chan_send

```
void chan_send(spmcchan_t cino, void *addr, size_t sz){
  char *data = (char*)malloc(sz);//从堆中开辟一个空间去
  //存放发送者发送过来的数据
  memcpy(data, addr, sz);
  enqueue(&(global_vars.que[cino]), data);//将发送者发送过来的数据入队
}
```

这样实现的原因有两个：

1)dedup-detmp 实现的代码中，存在一个线程发送给另一个线程的数据是栈变量的情况。为了表述方便，我们这里假设线程 A 发送一个栈变量给线程 B。在这种情况下，如果不保存发送过来的变量的内容，而只是保存变量的地址，则有可能会出现在 B 线程读取这个变量的时候，这个变量已经被 A 线程释放了, 因为栈变量在函数调用结束时会被释放掉。

2) 在 dedup 程序中存在如下模式的代码，见代码4。 在代码4中，DataProcess 线程从堆中分配空间，并将它赋值给 anchor, 然后使用 chan_send 将 anchor 的内容发送给 FindAllAnchors 线程，并最后调用 free 函数将其分配的空间释放掉。而 FindAllAnchors 线程则使用 chan_recv 来接收 anchor 的内容。

从有这种访问模式的代码来看，如果在用 pthreads+queue 来实现 detmp 的接口时不拷贝发送数据的内容而是仅传递指针，则有可能会出现 FindAllAnchors 在访问数据的时候，该空间已经被 DataProcess 线程调用 free 函数给释放掉了。