GCC/UPC 4.0
”Flexible Heap” Design Overview

Gary Funck, Intrepid Technology
Email: gary@intrepid.com
Date: September 5, 2006
Version: 1.0

Background

The GCC/UPC 4.0 runtime implements a facility known as a “flexible heap”. The flexible heap facility removes the need for the user to specify the maximum amount of memory required for allocation from the UPC language defined heap facility. The *–fupc-heap-N* (alternatively *–heap N*) runtime switch is no longer required. The *–fupc-heap-N* switch now controls only the initial amount of heap allocated to the program.

Requirements

The design of the flexible heap implementation is influenced by the following requirements and factors:

1. The UPC language specifies that a shared pointer can be cast into a thread local pointer (a conventional “C” pointer) as long as the referenced shared object has *affinity* to the thread that is casting the reference to a local pointer. This implies that any part of a dynamically allocated shared object with affinity to thread T must be mapped contiguously into the local address space of T at the moment of the initial cast to a local pointer. The reference must remain valid until T terminates (or the object is freed).
2. In order to preserve the local contiguity prescribed by requirement 1, above, the heap manager must not allow objects that were previously allocated to be coalesced into a larger object, if that object is not contiguous in the address space of the thread that has affinity to the referenced allocation.
3. Each thread should operate independently of all other threads to the maximum degree possible. Thus, the data structures used to track references into the global shared address must be thread local unless there is a requirement that the data structure be shared across all threads.

4. Since the maximum amount of space required by the UPC program is not known until the program executes, a flexible method of obtaining the required space is required. The use of an underlying file in the file system is seen as a desirable implementation option, with mapping to the file provided by the *mmap()* system call. The MAP_ANONYMOUS facility previously used by the GCC/UPC runtime is equally flexible, but often its maximum size is limited, and subject to operating system resource limitations.
5. The flexible heap facility must be accessible both from the process-based and the pthreads-based GCC/UPC runtime libraries. The design should change as little as possible when implemented in either the process-based or pthreads-based runtime libraries.
6. The flexible heap implementation should be efficient and minimize any additional overhead, when compared to the previous fixed-heap design.
7. The use of *mmap()* to map local regions of a thread's memory onto the global shared virtual address space may lead to a situation where the UPC program runs into a limit on the maximum permitted number of mapped regions supported by the operating system. The design and implementation should provide configuration options to control the maximum required number of mmap'd regions, and the default settings should support a wide range of applications while consuming only a moderate level of operating system resources.

The Paged Global Shared Virtual Address Space

UPC shared objects are mapped into a paged virtual address space, where each page is accessed by its Global Page Number (GPN). Global page numbers begin at zero and increase to the maximum number of allocated pages (minus one). The UPC runtime uses a page size that is much larger (typically in the 4 Megabyte to 32 Megabyte range) than the underlying operating system supported page size (which is typically in the 4 Kilobyte to 32 Kilobyte range). Global page number 0 corresponds to block zero in the underlying mmap()'d file, and so on.

In a fixed heap implementation, a given thread's contribution to the global shared virtual address space is given by the following equation:

```
Thread_Local_Size = Static_Shared_Object_Size + Maximum_Heap_Size
```

To simplify the mapping from a (thread, offset) address to (global page number, page offset) address, the Thread_Local_Size value is rounded up to the nearest multiple of the UPC runtime's page size. For fixed heaps, the calculation is simple:

```
Global_Page_Number = ((THREADS * Thread_Local_Size) + Offset) / Page_Size
Global_Page_Offset = Offset mod Page_Size
```

Flexible heaps complicate the calculation above, because each thread no longer has a statically defined contribution to the global shared address space. As each UPC thread allocates more shared memory, a thread's local contribution increases. For efficiency reasons, the per thread contribution increases by a series of "chunks", where each chunk is a multiple of the page size, and at a minimum is large enough to contain a given newly allocated object in its entirety. The requirement that any newly allocated object is completely contained in a chunk ensures that a per-thread slice of that object can be contiguously mapped into the memory of the thread that has affinity to that slice of the shared object. (Note that the chunk size discussed above is the per-thread chunk size.)

Although each page within a per-thread chunk must be contiguous in the global shared address space, individual per-thread chunks will be spread across the virtual address space as new chunks are allocated. Thus, the global page associated with page N+1 in a thread's virtual address space may be at some arbitrary distance away from page N in the thread's virtual address space.

The UPC runtime uses an array known at the Global Page Table (GPT) to map an address given in the (Thread, Thread_Offset) form into the (global page number, page offset) form used to locate the addressed object's data inside the memory mapped file associated with the UPC global shared virtual memory space. The mapping calculation is given by:

```
Thread_PN = Thread_Offset / Page_Size
Page_Offset = Thread_Offset mod Page_Size
Global_Page_Number = GPT[Thread_PN * THREADS + Thread]
```

The GPT is allocated by the GCC/UPC runtime before the UPC main program is executed; it is sized to accommodate the maximum number of per-thread address bits specified in the configuration file, *upc_config.h*. Here are the configuration parameters used by the default 64-bit runtime configuration:

```
/* On 64-bit machines, use page size of 32M (25 bits) and a max per thread
   offset of 128G (38 bits).  This leaves 13 bits for the per thread
   number of pages.  */
#define UPC_VM_OFFSET_BITS 25
#define UPC_VM_MAX_PAGES_PER_THREAD (1 << (38 - UPC_VM_OFFSET_BITS))
/* Derive some VM specific constants. */
#define UPC_VM_PAGE_MASK (UPC_VM_MAX_PAGES_PER_THREAD - 1)
#define UPC_VM_PAGE_SIZE (1 << UPC_VM_OFFSET_BITS)
#define UPC_VM_OFFSET_MASK (UPC_VM_PAGE_SIZE - 1)
```

Using the calculation described above, the UPC runtime can map an address of the form (thread, offset) into (global page number, page offset), which can in turn be used to locate the data in the operating system file that underlies the UPC program's shared data virtual memory space. However, in order to access the data using machine level instructions, this global page must be mapped into the referencing thread's address space. This mapping is done via the *mmap()* system call.

To efficiently implement the mapping between (global page number, page offset) and (local mmapp'd page base address, page offset), the UPC runtime uses two per-thread data structures:

1. The Local Page Table (LPT) is used to map pages in shared objects that have affinity to the currently executing thread:
   `Page_Base_Addr = LPT[Thread_PN]`
2. The Global Map Table is used to reference shared objects with affinity to threads other than the currently executing thread. The GMT hashes the Global Page Number into a row in the table. Each row in the table implements an associative set, where the first element in the set is the most recently used.

The relevant program logic is shown in the code excerpt below:

```
if (t == MYTHREAD)
  {
    /* A local reference:
       Refer to the Local Page Table to find the proper mapping.  */
    page_base = lpt[pn];
  }
else
  {
    /* A global reference to another thread's storage:
       Refer to the cached map entries in the Global Map Table.  */
    page_base = upc_vm_map_global_page (t, pn);
  }
p_offset = (offset & UPC_VM_OFFSET_MASK);
addr = page_base + p_offset;
return addr;
```

Entries in the LPT are never unmapped (via an *munmap* call), because shared data with affinity to the currently executing thread may be addressed by a regular "C" pointer by casting its shared address into a local address. There is no such requirement for references to shared data with affinity to other threads, however. Thus, mappings recorded in the GMT may come and go.

(continued on next page)

Although the mappings via the LPT and GMT are relatively efficient, the UPC runtime speeds up the address calculation further, by caching the two most recent address mapping calculations. The code that implements this additional level of caching follows:

```
offset = ((size_t) p.vaddr - (size_t) UPC_SHARED_SECTION_START);
p_offset = offset & UPC_VM_OFFSET_MASK;
pn = (offset >> UPC_VM_OFFSET_BITS) & UPC_VM_PAGE_MASK;
this_page = (pn << SHARED_PTR_THREAD_SIZE) | p.thread;
if (this_page == __upc_page1_ref)
  addr = __upc_page1_base + p_offset;
else if (this_page == __upc_page2_ref)
  addr = __upc_page2_base + p_offset;
else
  {
    addr = __upc_vm_map_addr (p);
    __upc_page2_ref = __upc_page1_ref;
    __upc_page2_base = __upc_page1_base;
    __upc_page1_ref = this_page;
    __upc_page1_base =  addr - p_offset;
  }
```

Per-Thread Update of Allocated Memory Size

Whenever the amount of currently allocated space is insufficient to satisfy a dynamic memory allocation request, the global shared memory available to each UPC thread is increased. The global shared memory is increased by appending additional blocks to the file underlying the global shared memory. The file size is increased by the product of THREADS and the per thread number of newly allocated pages.

The maximum number of pages available to each thread is maintained in a global structure that can be accessed by all threads. Accesses to this global *cur_page_alloc* value must be serialized to avoid race conditions. Thus, it relatively expensive for a thread to frequently access this value. To avoid unnecessary accesses to the globally maintained value, the UPC runtime uses a per thread copy of the value which is guaranteed to always be less than or equal to the current global value, as a result of the fact that the shared global address space is never decreased. The per thread copy of the *cur_page_alloc* value is updated whenever a reference is made to a page whose thread-relative page number is not less than the current locally maintained current page allocation value.

When the UPC runtime determines that the number of pages available to the currently executing thread has increased, it maps the pages now available to the thread and updates the Local Page Table (LPT). The UPC runtime is careful to batch sequences of pages that have contiguous global page numbers into a single mmap() call. This insures that the newly mapped region will be contiguous in the local address space of the thread that has affinity to those pages and also helps decrease the number of separate memory mapped regions within the Unix

process underlying the UPC thread (or associated with the pthreads mapped to UPC threads in a pthreads based implementation).

Heap Allocation Sequence Numbers: A Tool to Make Sure that Free Does Not Combine Non-Contiguous Memory Regions

Any slice of a shared object returned by the UPC language defined dynamic memory allocation functions with affinity to a given thread must be contiguous in that thread's local address space. This property insures that valid casts from a shared pointer to a local pointer operate in the UPC language prescribed manner.

The GCC/UPC heap implementation uses a sequence number that is associated with each set of pages that are added to the thread's available virtual memory, in response to a dynamic memory allocation library call. The *upc_free()* function will not coalesce entries in the free space list that have different sequence numbers. This ensures that every per-thread slice of a newly allocated shared object will be mapped into a contiguous region in the local address space of the thread with affinity to that slice.

The UPC Heap Allocation Functions are Written in the UPC Language

The UPC dynamic memory allocation routines maintain the heap data structures in UPC shared memory. This ensures that the space needed to manage the heap will grow naturally as new virtual memory space is allocated to the UPC program. Further, placing the heap data structures in shared memory simplifies the transformation of free space into allocated space – the pointer to the free space entry selected for allocation is simply incremented by a fixed offset to then point to the newly allocated space. The *upc_free()* operation finds the control information associated with the object that is to be freed by simply decrementing the pointer to the object.

In the GCC/UPC version 3 fixed heap implementation, all of shared memory could be easily accessed because the entire shared memory region was mapped into each UPC thread's address space. Now that the mapping is more complicated, re-implementing the heap allocator in UPC makes sense because it hides the details of the shared memory virtualization and maintains the implementation in a form that is easier to understand and maintain.

The flexible heap allocation routines are written in UPC, and at present the heap allocation module is the only GCC/UPC runtime library source file that is written in UPC. The UPC heap allocation library functions depend upon the UPC virtual memory implementation, and at the same time also interact with the VM mechanism whenever requests are made to add new pages to the UPC program's global shared virtual memory address space. Thus, care must be taken in the implementation to avoid circular dependencies both during initialization and during normal execution of the UPC program. The heap allocation routines must be careful to use a safe subset of UPC facilities and library functions because some library functions implicitly rely upon UPC's dynamic memory allocation functions (UPC language defined locks for example).

Paging and the UPC Memory-to-Memory Library Functions

The flexible heap implementation virtualizes the UPC program's shared memory address space. This VM implementation breaks the address space up into pages that are subsequently mapped into the referencing thread's local address space as required.

Dividing the UPC global shared memory into a series of potentially separately mapped pages complicates the implementation of the memory to memory operations, defined in the UPC library. The memory-to-memory UPC library functions, such as *upc_memget()* and *upc_memput()*, now must be written in a way that that memory copy operations will not cross VM page boundaries. Alternatively the memory-to-memory operations could consult the page tables to determine if particular sequences of pages that are contiguous in the virtual address space also happen to be contiguous in the thread's address space. However, since the UPC page size has been configured by default to be fairly large, page crossings should be infrequent, and a simpler algorithm is appropriate.

For example, the implementation of *upc_memput()* is shown below:

```
for (;;)
  {
    char *destp = (char *)upc_sptr_to_addr (dest);
    size_t offset = ((size_t)dest.vaddr - (size_t)UPC_SHARED_SECTION_START);
    size_t p_offset = (offset & UPC_VM_OFFSET_MASK);
    size_t n_copy = min (UPC_VM_PAGE_SIZE - p_offset, n);
    memcpy (destp, src, n_copy);
    n -= n_copy;
    if (!n)
      break;
    dest.vaddr += n_copy;
    src += n_copy;
  }
```

Above, the *upc_sptr_to_addr()* function converts a shared pointer into a regular C pointer that points to the data associated with the designated shared memory address. The paged nature of the UPC virtual memory implementation ensures

that all bytes from the location in the VM page identified by the translated shared pointer can be directly addressed. The *upc_memput()* implementation above efficiently makes use the fact that the UPC shared memory is mapped into fixed size physical pages. The runtime also ensures that at least the two most recently mapped pages are mapped into the currently executing thread's local address space.

The *upc_memcpy()* function implements a copy between two shared objects and places a special requirement upon the runtime. The runtime must guarantee that the two most VM memory mappings are valid. This guarantee avoids the need to utilize an intermediate local memory buffer to effect the copy.

Judicious Use of Inlining

To further improve efficiency, and to factor out common code, the UPC runtime uses GCC's *inline* specifier to selectively inline internal library support functions called from frequently executed runtime library routines. An example is shown in the UPC shared memory access routine below:

```
/* To speed things up, the last two unique (page, thread)
   lookups are cached.  Caller must validate the pointer
   'p' (check for NULL, etc.) before calling this routine. */
static inline
void *
upc_sptr_to_addr (upc_shared_ptr_t p)
{
  /* … */
  if (this_page == __upc_page1_ref)
    addr = __upc_page1_base + p_offset;
  else if (this_page == __upc_page2_ref)
    addr = __upc_page2_base + p_offset;
  else
    {
      addr = __upc_vm_map_addr (p);
      /* … */
    }
  return addr;
}

static inline
void *
upc_access_sptr_to_addr (upc_shared_ptr_t p)
{
  if (IS_NULL_SHARED (p))
    __upc_fatal (UPC_NULL_ACCESS_MSG);
  if (p.thread >= THREADS)
    __upc_fatal (UPC_INVALID_THREAD_IN_ADDR_MSG);
  return upc_sptr_to_addr (p);
}

u_intQI_t
__getqi2 (upc_shared_ptr_t p)
{
  const u_intQI_t *addr = (u_intQI_t *) upc_access_sptr_to_addr (p);
  return *addr;
}
```

Above, the code generated for the *getqi2()* function will usually follow the fast path through the referenced inlined functions, and infrequently call the external *__upc_vm_map_addr()* function. Thus, the remote access implemented by this function will typically require only a single procedure call to the library access routine itself.

It is anticipated that a similar technique for inlining the fast path through the memory access routines will be utilized in an improved runtime implementation that initially targets the Berkeley runtime API on the Cray XT3 (and its successors). Once any issues related to inlining those UPC library functions have been ironed out, the GCC/UPC runtime will be upgraded in a similar fashion.

Pthreads Implications

The user can direct GCC/UPC to utilize POSIX threads (pthreads) in the implementation of each UPC thread. This implementation choice is selected by passing the *–fupc-pthreads-model-tls* switch to the GCC/UPC compiler. The pthreads based implementation runs each UPC thread within the context a its own POSIX thread (pthread).

The GCC/UPC flexible heap runtime design and implementation does not change in any significant way, when executing in an environment where each UPC thread is mapped to a pthread. Each UPC thread will maintain its own thread local Local Page Table, and Global Map Table. An alternative implementation, where pages accessed via each UPC thread's Local Page Table (because the LPT's of all threads can be directly addressed by all pthreads) might offer some efficiencies at the cost of increasing the need to serialize accesses to the LPT's of other threads.

In the present design, a given UPC thread will map pages with affinity to other threads via the hashed Global Map Table. In a pthreads environment, it may be that one or more UPC threads map a particular thread's page in their own GMT's. Further the mapping to the same page will be recorded in the LPT of the thread that has affinity to that page. Since Linux and most Unix'es permit this sort of map aliasing, pthreads based UPC programs will operate correctly, though perhaps not optimally using the current design. The main benefit of the design is that it remains the same in both the process-based and pthreads-based implementtions.

Limitations

The previous GCC/UPC version 3 compiler took advantage of the fact that the entire global memory region was directly mapped into each UPC thread's address space, and that a location identified by a shared pointer could be mapped directly into a locally accessible memory location by a simple calculation involving only a multiply and an add. The GCC/UPC version 3 compiler would generate code that performed this address mapping and memory access directly. Alternatively, the user could direct the compiler to generate calls to the runtime library instead by compiling with the *–fupc-libcall* switch, but the *–fno-upc-libcall* option was generally preferred because it eliminated the runtime library call overhead.

GCC/UPC version 4 translates all remote get and put operations into calls to the runtime library and does not generate explicit code to implement accesses to shared memory. The flexible heap implementation is sufficiently complex, and subject to change that it doesn't make sense for the compiler to generate explicit code to implement the remote get and put operations. Thus, at present, external library routines will always be called.

Due to the call overhead, and more complicated memory mapping, it is likely that UPC programs (compiled with GCC/UPC version 4) that make frequent reference to shared memory will run (sometimes noticeably) more slowly than the same programs compiled with GCC/UPC version 3.

Benefits

The flexible heap facility removes the requirement that the user supply an explicit upper limit on the size of the UPC dynamic memory allocation runtime heap. Further, by removing the restriction that the entire shared address space must be mapped into each UPC thread's address space, it should be possible to increase the amount of memory available to each thread, and thus make it possible for a UPC program compiled by GCC/UPC version 4, and linked with its runtime, to handle larger problems than those compiled with version 3. This will be especially noticeable on platforms that support only 32 bit address spaces.

The lack of directly generated code for UPC's remote access operations will be offset in the future by moving many of the frequently executed runtime procedures into a pre-included header file, where they will be declared as inline functions.

-- End --