Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON n° 8512

CENTRE NATIONAL
DE LA RECHERCHE
SCIENTIFIQUE

SPI

An efficient and transparent thread migration scheme in the PM2 runtime system

Gabriel Antoniu
Luc Bougé
Raymond Namyst

January 1999

Research Report N° 1999-08

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`

INRIA

An efficient and transparent thread migration scheme in the PM2 runtime system

Gabriel Antoniu
Luc Bougé
Raymond Namyst

January 1999

Abstract

This paper describes a new *iso-address* approach to the dynamic allocation of data in a multithreaded runtime system with thread migration capability. The system guarantees that the migrated threads and their associated static data are relocated exactly at the same virtual address on the destination nodes, so that no post-migration processing is needed to keep pointers valid. In the experiments reported, a thread can be migrated in less than 75 μs.

Citation. A slightly abridged version of this report has been published in the Proceedings of the 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP '99) [1]. Please, mention this reference in any citation.

Keywords: Thread migration, iso-address allocation, PM2, multithreading.

Résumé

Nous décrivons une nouvelle technique d'allocation dynamique de mémoire (approche que nous appellons *iso-adresse*) conçue et implémentée dans l'environnement multithread PM2, qui propose la migration de processus légers. Le système garantit que les threads et leurs données associées sont relogés exactement à la même adresse sur le nœud destinataire. Ainsi, aucun traitement post-migration n'est nécessaire pour préserver la validité des pointeurs. Dans les expériences présentées, un thread peut être migré en moins de 75 μs.

Citation. Une version légèrement abrégée de ce rapport a été publiée dans les actes du *3rd Workshop on Runtime Systems for Parallel Programming* (RTSPP '99) [1]. Merci de mentionner cette référence dans les citations.

Mots-clés: Thread, processus léger, migration, allocation iso-adresse, PM2, multithreading.

An efficient and transparent thread migration scheme in the PM2 runtime system

Gabriel Antoniu* Luc Bougé* Raymond Namyst*

26th January 1999

Contents

---

*LIP, ENS Lyon, 46 Allée d'Italie, F-69364 Lyon Cedex 07, France. Contact: {Gabriel.Antoniu,Luc.Bouge,Raymond.Namyst}@ens-lyon.fr. This work has been supported by the INRIA ResCapA Research Coordinated Action, the NSF/INRIA *C*IT* Cooperative Research Grant, the CNRS ARP Research Program and the ReMaP Project, INRIA Rhône-Alpes.

1 Introduction

Why use threads? *Lightweight processes* (or *threads*) have proven useful to implement massively parallel activities in both shared-memory and distributed-memory systems, thanks to the efficiency of their management. Thread creation, destruction and context switching are tens to thousands of times as fast as the corresponding operations for traditional, *heavy* processes. For instance, in the PM2 multithreaded runtime system, a thread context switch takes 0.5 μs. This efficiency is mainly due to the small size of the thread resources (a few kilobytes, compared to several megabytes for traditional processes).

Using threads for load balancing In order to obtain an efficient execution of a distributed parallel application, computation needs to be evenly distributed among the processors of the underlying architecture. Sometimes, a "good" initial distribution may suffice, but this is not the case of irregular applications with unpredictible behaviour. In such cases, *dynamic* load balancing is necessary: tasks get reassigned from the overloaded nodes to the underloaded ones. This can be done in an elegant, transparent way by using *task migration*. Migrating *heavy* processes may be straightforward, but inefficient, because of the large size of the resources to be transfered. Using *threads* instead of *processes* has considerable advantages. First, thread migration is much more efficient (less than 75 μs in PM2 on our experimental plateform). Second, thread-based systems may support fine-grain parallelism, allowing for better load balancing.

The iso-address approach Migrating a thread usually means moving its control flow (i.e. the thread stack and descriptor), but sometimes may also mean moving its *static data*. A thread may deal with arbitrarily complex data structures (for instance, linked lists or trees, built up with pointers) ant it is crucial to keep all these pointers valid when the thread moves to another node. One way consists in updating these references once the thread has been copied in memory on the destination node. A much more satisfactory solution is to guarantee that the copy may be made while keeping the same virtual address. Post-migration reference update is then no longer necessary. The idea is to provide the programmer with a memory allocation function which guaranees that the virtual address range returned is kept free at all nodes but the allocator. The object (i.e. a thread stack or a private data structure belonging to the thread) can then always be migrated at the same virtual address. This problem is currently subject to several research efforts [8, 5, 9, 14].

Objective of this work Our interest in iso-address allocation and migration stems from data-parallel compiling. Consequently, our study focuses on the case in which data are not shared: they belong to some unique thread and thus have to follow it on migration. Our iso-address allocator has been implemented in the PM2 multithreaded runtime system [11], which serves as a runtime support for two data-parallel compilers [2]. We target applications having to execute on *homogeneous* clusters of workstations or PCs interconnected by a high-speed network (e.g., Myrinet [10]).

This paper is structured as follows. In Section 2, we give a quick description of PM2: a multithreaded runtime system providing thread migration. An overview of our iso-address approach is given in Section 3 and some implementation details are presented in Section 4. Section 5 shows some performance figures. Finally, Section 6 summarizes our main results and points out what we intend to address in the near future.

Related works

Thread migration has already been implemented in several multithreaded systems. In Ariadne [9], threads are migrated to get close to the remote data they use. Static data never move. On migration, the thread stack is relocated at a usually different address on the destination node, so that pointers need to be updated. As shown in Section 2.2, several problems cannot be solved by this approach. In Millipede [8], thread migration is directed by a load balancing module integrated in the system, whereas static data get moved only when they get accessed by remote threads. The threads and their data are always relocated at the same virtual addresses on all nodes. Yet, thread creation is expensive, therefore the number of concurrent threads is statically fixed at initialization. In both systems mentioned above, data are shared and can be accessed by more than one thread. UPVM [5] provides thread migration for PVM aplications, in order to support load balancing. Threads have private heaps, for private dynamic allocations. Thread creation is expensive in this system, too, since it is carried out by means of a global synchronization. Besides, thread private heaps are fixed at thread creation, so that the amount of data that a thread can allocate is limited.

2 PM2: a multithreaded runtime system with thread migration

PM2 is a multithreaded runtime system especially designed to serve as a runtime support for highly parallel irregular applications. In such applications, threads may need to start or terminate at arbitrary moments during the execution. At the same time, the system has to efficiently cope with a large number of concurrent threads. Therefore, PM2 provides very efficient primitives to handle these operations: creation, destruction and context switching. A distinctive feature of PM2 is thread migration. Since the execution of irregular applications may lead to severe load imbalances, thread migration can be used to support the implementation of load balancing policies based on dynamic activity redistribution.

2.1 The basic migration scheme

In a PM2 application, there is a single (heavy) process running at each node and each such process may contain tens of thousands of threads. We often identify this container process with the node running it. At the simplest level, a PM2 thread is an execution flow managing a set of *resources*, i.e., its state descriptor and its private execution stack. The code to be executed by the threads is replicated on each node (SPMD approach) and is not part of the thread. Again, we emphasize that we do not consider the aspects of data sharing between threads in this paper, nor the problem of a thread using global process resources such as files, network interfaces, etc. In this setting, migrating a thread simply means moving the thread *resources* from the (heavy) process running on the local node to another (heavy) process located on some remote node. In PM2, the migration operation is carried out in three main steps (Figure 1):

1. The thread gets stopped (*frozen)* and its resources get copied to a communication buffer. The memory area storing the resources is set free.

2. The buffer contents get sent to the destination node through the network.

3. An adequate memory area is allocated on the destination node, the thread resources are copied into it, and the thread execution is resumed.

In PM2, any thread may "decide" to migrate to another node at any arbitrary point during its execution. It may also be *preemptively* migrated by another thread running on the same node. This

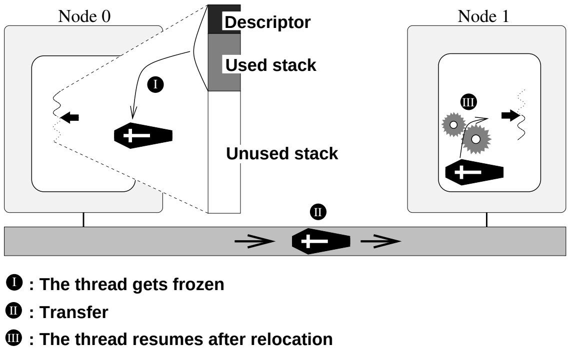

Figure 1: Thread migration in PM2

latter property is essential, since it ensures that application threads may be *transparently* migrated across the nodes. Consequently, a *generic* module implemented *outside* the running application could balance the load by migrating the application threads. The threads are unaware of their being migrated and keep on running irrespective of their location.

Source code:

```
void p1()
{
  int x;

  x = 1;
  pm2_printf("value = %d\n", x);
  pm2_migrate(marcel_self(), 1);
  pm2_printf("value = %d\n", x);
}
```

Execution:

```
[node0] value = 1
[node1] value = 1
```

Figure 2: Thread migration without pointers.

Source code:

```
void p2()
{
  int x;
  int *ptr = &x;

  x = 1;
  pm2_printf("value = %d\n", *ptr);
  pm2_migrate(marcel_self(), 1);
  pm2_printf("value = %d\n", *ptr);
}
```

Execution:

```
[node0] value = 1
Segmentation fault
```

Figure 3: Thread migration in the presence of pointers to stack data

An example of thread migration is given on Figure 2. Assume that a thread running on node 0 calls procedure `p1`. The thread declares a local variable `x`, writes the value 1 to this variable, then prints it. Next, the thread migrates to node 1 and prints the value of the variable `x` again. At run time, we can see that the value 1 is displayed in both cases, before and after migration. The local variable `x` gets automatically moved to node 1, since it is stored in the thread stack.

A difficulty turns up as soon as a migrating thread makes use of pointers. Such a situation is illustrated on Figure 3. Here, the thread which calls `p2` reads variable `x` through pointer `ptr`. After migration, there is no guarantee that variable `x` is still located at address `ptr` and the execution (most probably!) fails.

2.2 Thread migration and pointers

One way to tackle this problem is to update all references to stack data after migration, before the thread resumes its execution by adding some offset to all pointers. Two categories of pointers to stack data require such post-migration processing: the *implicit* pointers generated by the compiler in order to chain the stack frames and the *explicit* pointers used by the programmer. The former may be identified using some knowledge about the way they are generated by the compiler, whereas the latter need to be explicitly declared to the system, in order to enable their update after migration. Such an approach was implemented in the early versions of PM2, which provided primitives to register/unregister user-level pointers. When a thread moved to another node, all its registered pointers were updated (Figure 4).

Source code:

```
void p2()
{
  int x;
  int *ptr = &x;
  unsigned int key;
  key = pm2_register_pointer(&ptr);
  x = 1;
  pm2_printf("value = %d\n", *ptr);
  pm2_migrate(marcel_self(), 1);
  pm2_printf("value = %d\n", *ptr);
  pm2_unregister_pointer(key);
}
```

Execution:

```
[node0] value = 1
[node1] value = 1
```

Figure 4: Thread migration with registered pointers

Source code:

```
void p3 ()
{
  int *t =
   (int *)malloc (100 * sizeof(int));

  t[10] = 1;
  pm2_printf("value = %d\n", t[10]);
  pm2_migrate(marcel_self(), 1);
  pm2_printf("value = %d\n", t[10]);
}
```

Execution:

```
[node0] value = 1
Segmentation fault
```

Figure 5: Thread migration with pointers to heap data

Clearly, this approach does not extend to complex applications. Moreover it does not cope with resources located outside of the stack, such as heap data dynamically allocated by the malloc primitive of the C language. Figure 5 shows a thread which calls malloc to allocate some memory area, writes (potentially large) data into this area, migrates, and eventually tries to read at the same virtual address. The program obviously fails, since the allocated data has not been migrated.

One way to solve this problem consists in remallocating the data on the destination node. In this case, the programmer has to explicitly handle the data packing and unpacking, and to manage the pointer updating as the allocation address are usually different from the original one. As in the case of pointers to stack data, this approach cannot be used for arbitrarily complex applications making use of a large number of pointers to heap data. Moreover, this approach cannot cope with compiler-generated pointers in case optimization options are used, since such pointers are not registered and cannot be updated. Fundamental compiler optimizations such as using pointers instead of indices to scan large arrays are thus forbidden.

3 Our approach: the isomalloc memory allocator

3.1 General overview

A much better approach to the problem described in the previous section is to provide a mechanism able to guarantee that *both* the stack and the private, dynamically allocated data of a thread can be migrated and remallocated at the same virtual address on the destination node (*iso-address allocation and migration*). The idea is to *locally* allocate storage areas in a system-wide, *globally* consistent way. The allocation mechanism must guarantee that each range of virtual addresses at which memory has been mmapped at some node is kept free on all the other nodes. Such an approach has several advantages.

Simplicity The migration mechanism is simplified, because no post-migration pointer update is necessary any longer.

Transparency *Applications* may make free use of pointers without having to take into account possible problems related to thread migration. User-level pointers are always guaranteed to be safe.

Portability No *compiler* knowledge about the thread stack structure is required, since the stack contents remains exactly the same after migration. In particular, compiler-generated pointers are migration-safe, too. Consequently, any compiler may be used and compiler optimizations are allowed.

Preemptiveness Preemptive migration is possible, given that no assumption is made about the thread state at migration time.

The isomalloc allocation mechanism relies on a few basic principles. These rules ensure that each node may use its globally reserved memory without having to "inform" the other nodes. We thus avoid any synchronization when allocating memory to threads.

1. The physical execution environment is assumed to be *homogeneous* (same type of processor, same operating system). Moreover, all nodes have the same memory mapping: the same binary code is loaded on each of them at the same virtual address (so that no code needs getting moved upon migration). The (unique) process stack is also located at the same virtual address on all nodes.
2. On each node, all iso-address allocations take place within a special address range called *iso-address area*. We have located it between the process stack and the heap (Figure 6). This zone corresponds to the same virtual range on all nodes.
3. Separate ranges of virtual addresses within the iso-address area are *globally reserved* for each node, so that each address may be used by a single node at a time.
4. The *actual* memory allocation is carried out *locally*, within an address range belonging to the node on which the allocation request is made.

3.2 The slot layer

In this improved view, a PM2 thread is an execution flow managing a set of *resources*, i.e., its state descriptor, its private execution stack, and a series a dynamically allocated sub-areas within the iso-address area. Let us introduce some terminology at this point for the sake of clarity. An *address slot* is a range of virtual addresses within the iso-address area. A slot is *free* if no memory has been mmapped at this address. Otherwise, it is *busy*, and we say that memory has been *allocated* in this slot. Then, data may be stored within this slot of virtual addresses. The iso-address discipline guarantees that a slot which is busy on a node is guaranteed to remain free on any other node.

Our goal is to design the management policy so as to avoid inter-node synchronization as far as possible and to remain compatible with the heap management mechanisms of the container (heavy)

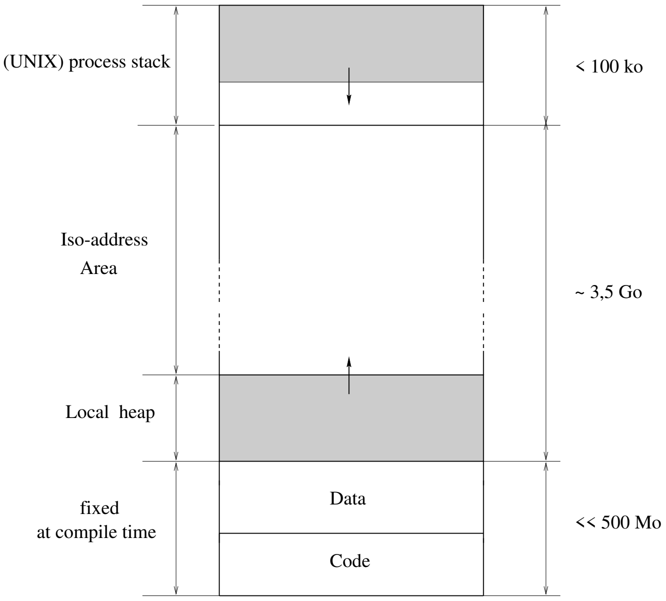

Figure 6: All nodes have the same memory mapping. In particular, the iso-address area covers the same virtual address range on all nodes

process. To manage slots in a consistent system-wide manner, it is convenient to give them a uniform size, very much like memory pages at the node level. The choice of this size is obviously crucial and we will discuss it later. We introduce again some terminology. At any point, exactly *one* agent, a node or a thread, is responsible for managing a given slot. It is the *owner* of the slot. The slots owned by a node or a thread are called its *private* slots. A slot owner is responsible for mmapping or unmmaping memory at this slot of addresses, and reading or writing data. Nobody but the owner is allowed to use the slot.

At initialization time, each slot is owned by a unique *node* and is free. When a thread is created, the local node *gives* the thread a slot to store its initial resources: this slot is from now on owned by the thread. When isomallocating data dynamically, a thread acquires additional slots from the local node. Notice that all this change of ownership do not require any synchronization between nodes whatsoever. A thread is associated with the list of its private slots where it stores its resources. On migration, these slots migrate along with the thread, which still owns them after the migration, though the memory is allocated at another node. At any point, a thread may release slots. They are then given to the node the thread is currently visiting. This node may be different from the node from which they have been acquired. On dying, a thread releases all the slots it currently owns. This *slot life cycle* is illustrated on Figure 7. Observe that the size of the slots and their initial distribution among nodes is completely irrelevant at this point. We will discuss the choice of this distribution later.

In our current implementation, the slot size is fixed to a medium value of 64 kB, that is 16 virtual memory pages in our implementation. It is large enough to contain the initial resources of a PM2 thread: its descriptor and its execution stack. This ensures that thread creation remains a simple, local operation, irrespective of the slot distribution pattern, since a single slot is needed. A more detailed description of our slot management policy is given in Section 4.

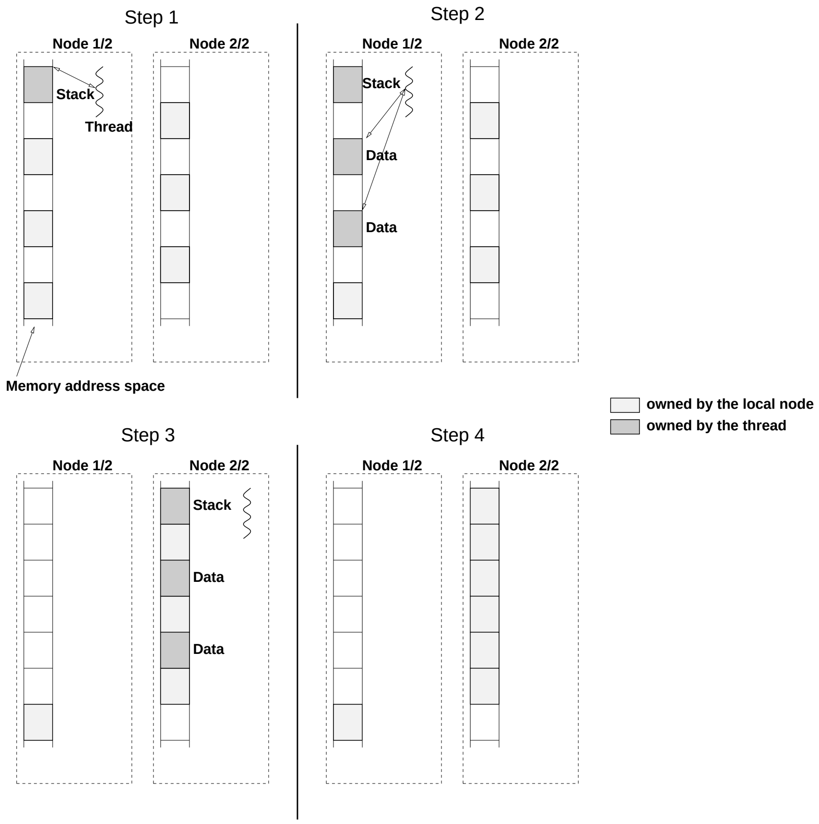

Figure 7: Slot ownership may change due to migration. In this example, a thread is created and acquires a slot owned by the local node to store its stack (Step 1). The thread acquires other slots from the local node, to store its private data (Step 2). The thread migrates along with its slots (Step 3). The thread dies and its slots are acquired by the destination node (Step 4).

3.3 The block layer

Since our goal is to provide an allocation function compatible with the malloc C primitive, the isomalloc allocator has been refined so as to cope with arbitrarily-sized zones of memory. This leads to a new concept: the block. A *medium-sized* slot may contain multiple *small-sized* blocks. Therefore, our allocator is structured in two layers:

1. The lower slot layer, responsible for slot management (allocation, release and chaining in the private list of a thread);
2. The upper block layer, responsible for multiple allocations made by a thread within the slots it currently onws.

Thus, our allocation primitive is a malloc-like function managing block allocations within a list of slots. PM2 calls the block layer primitives for the private allocation/release operations required by the threads, given that arbitrarily-sized blocks need to be handled. On the contrary, for thread resources allocations, the slot layer is directly called, since a stack is always stored in a whole slot. As explained in the previous subsection, the slot size was especially chosen to fit the thread stack size.

A slot may contain multiple small-sized blocks. Conversely, when large request are to be handled, a block may stretch over multiple contiguous slots. If the current local node owns the necessary number of *contiguous* slots, this allocation is carried out the same way as a simple, single-slot allocation. The set of contiguous slots is simply merged into a large slot. Otherwise, the node has to enter a negotiation with other nodes to *buy* from them the necessary set of *contiguous* slots. As such an operation involves synchronization and mutual exclusion, it is clearly much more expensive than "usual", local allocations. Everything has to be done to keep it exceptional. It is of course possible to increase the slot size defined at initialization. It is much more efficient to adjust the initial distribution of slots so as to favor the contiguity of the slots owned by nodes. We discuss these aspects further in Section 4.1.

3.4 The programming interface

The PM2 high-level programming interface provides two primitives by means of which threads may allocate (respectively release) memory in the iso-address area: pm2_isomalloc and pm2_isofree. These primitives have the same prototype as the classic C functions malloc and free:

```
void *pm2_isomalloc(size_t size);
void pm2_isofree(void *addr);
```

A thread must call pm2_isomalloc instead of malloc to allocate memory for private, non-shared data that are required to migrate with the thread. PM2 *guarantees* that all data stored at addresses returned by pm2_isomalloc follow the calling thread in case of migration. All addresses allocated by pm2_isomalloc have to be set free through a call to pm2_isofree. Using these primitives ensures that all references to the address areas handled by them remain valid and that accesses to the corresponding data are migration-safe. Migration is thus transparent and the migrating threads may use pointers in an arbitrary way.

An example of code using pm2_isomalloc is given in figure 8. Let us suppose that the procedure p4 is called by a thread running on node 0. The thread allocates memory blocks in the iso-address zone through successive calls to pm2_isomalloc and creates a linked list. Then, the thread begins to traverse the list while printing its elements. When the 101st element is reached, the thread migrates to node 1 and continues the traversal. As we can notice in figure 9, the first 100 list elements are displayed on node 0, whereas the next ones are displayed on node 1. All pointers in the list are still valid after migration, since PM2 guarantees that all blocks allocated by pm2_isomalloc migrate with the thread and keep the same virtual addresses.

4 Implementation details

4.1 Basic requirements

In order to implement our iso-address allocation strategy, we had to address the following points.

Iso-address area A specific part of the virtual space has to be dedicated to iso-address memory allocations on all nodes. To this purpose, we defined an *iso-address area* situated between the process stack and the heap (Figure 6). This is possible since all nodes are binary compatible and run by the same version of the operating system.

Global reservation, local allocation The iso-address area is divided into fixed-size virtual address slots, each of which is given to a unique node at initialization. To implement this *global*

```
#define NB_ELEMENTS 100000
#define NB_ITERATIONS 20000

typedef struct _item {int value;  struct _item *next;} item;

[...]
void p4() {
 int j; item *head, *ptr;

 /* Create a list. */
 head = NULL;
 for (j = 0; j < NB_ELEMENTS; j++) {
  ptr = (item *) pm2_isomalloc(sizeof(item));
  ptr->value = j * 2 + 1;   /* For example */
  ptr->next = head; head = ptr;
 }
 pm2_printf("I am thread %p\n", marcel_self());

 [...]
 /* Print the list elements.  */
 j = 0; ptr = head;
 while(ptr != NULL) {
  if (j = 100) { /* Migrate! */
   pm2_printf("Initializing migration from node %d\n", pm2_self());
   pm2_migrate(marcel_self(), 1);
   pm2_printf("Arrived at node %d\n", pm2_self());
  }
  pm2_printf("Element %d = %d\n", j, ptr->value);
  ptr = ptr->next; j++;
 }
}
```

Figure 8: Sample code using pm2_isomalloc. Procedure p4 is called by a thread initially running on node 0. After having allocated a few blocks in the iso-address area and constructed a linked list, the thread starts traversing the list. Arrived at element 100, the thread migrates to node 1 and continues the traversal.

reservation, each node is provided with a private bitmap which identifies the slots owned by the node (see 4.2). The initial slot distribution pattern must ensure that no slot is shared by several nodes. On each node, actual, *local allocations* may only take place at the slots owned by the caller. Memory allocation is done using the mmap primitive, which allows for memory allocation at specified virtual addresses.

Slot distribution Initially, slots are distributed among the nodes according to some user-defined distribution pattern which may be chosen so as to meet the needs of the application. This choice should be made such that most allocations be local and negotiations are as seldom possible. In our current implementation, slots are assigned to nodes in a round-robin fashion: slot i belongs to node $i \bmod p$ in a p-node configuration. This choice has been made for simplicity, but it behaves rather poorly for multi-slot allocations. Nothing prevents the user from choosing other distributions. For instance, instead of distributing single slots cyclically among the nodes, one may distribute series of contiguous slots (*block-cyclic* distribution). An extreme choice is to split the iso-adress area into p sub-areas, one for each node, but these scheme is not advisable if the heap of the container process needs to grow in unpredictable

```
info%pm2load example1
[node0] I am thread eeff0020
[node0] Element 0 = 1
[node0] Element 1 = 3
[...]
[node0] Element 99 = 199
[node0] Initializing migration
        from node 0
[node1] Arrived at node 1
[node1] Element 100 = 201
[node1] Element 101 = 203
[node1] Element 102 = 205
```

Figure 9: Execution trace for the code in Figure 8. The list traversal starts on node 0 and continues on node 1. Using malloc instead of pm2_isomalloc would result in a memory access error (Figure 10), since the list is not migrated with the thread in this case

```
info%pm2load example2
[node0] I am thread eeff0020
[node0] Element 0 = 1
[node0] Element 1 = 3
[...]
[node0] Element 99 = 199
[node0] Initializing migration
        from node 0
[node1] Arrived at node 1
[node1] Element 100 = -1797270816
[node1] Element 101 = 57654
Segmentation fault
```

Figure 10: If the call to `pm2_isomalloc` is replaced by a call to `malloc` in the code given in figure 8, an error occurs when the thread tries to access its list after the migration

ways. Observe that nothing prevents the system from triggering at any point a *global negotiation* phase, where all nodes would simply exchange their (free) slots to maximize the contiguity,

Slot size As previously explained, the slot size was chosen so as to fit a thread stack and was fixed to 64 kB, that is 16 pages. Thus, thread creation is a local operation (i.e., no negotiation is needed) irrespective of the slot distribution, since a single slot is required. This is also valid for all allocations of blocks smaller than a slot. As for larger allocations, details are given in Section 4.4.

4.2 Managing slots

Each node keeps track of its private slots by means of a private bitmap. Each bit in this bitmap corresponds to a slot in the iso-address zone. Given that this zone is typically as large as 3.5 GB and that a slot corresponds to 64 kB, the size of such a bitmap amounts to 7 kB. In each bitmap, the bits are set to 1 if they correspond to slots owned by the local node, otherwise they are set to 0. If a bit is set to 1, the corresponding slot is free. If it is set to 0, the slot belongs either to another node (and it is necessarily free) or to some local or remote thread.

When a slot request is issued by a thread (for instance, when a thread is created or when it requires additional storage area), one of the slots owned by the local node is given to the thread and the corresponding bit is set to 0 in the local bitmap. The slot does not belong to the local node any more. When a slot is released by a thread (due to dynamic release or to thread death), the corresponding bit *in the current local bitmap* is set to 1. Observe that the bitmaps do not undergo any change on thread migration, since the migrating slots keep being owned by the thread and the corresponding bits keep their 0-value on all nodes. Notice also that, due to migration, a slot may be allocated on a node and released on another, so that the destination node may eventually acquire slots that it did not possess initially.

Threads manage their private slots in a double-linked list (Figure 11). This is in contrast with nodes which manage their private slots by means of a bitmap. Chaining the slots owned by a thread makes it much easier to manipulate them on migration. Actually, chaining is carried out by means of pointers stored in the slot headers. Given that the slot contents get copied at the same virtual address in case of migration, these pointers remain valid and the chaining is thus preserved. As with user-level pointers, no post-migration processing is necessary: an iso-address copy is enough.

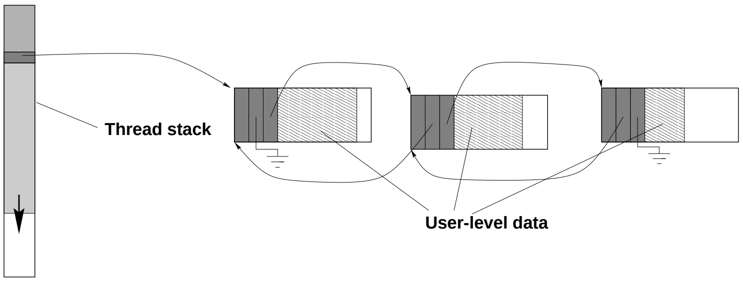

Figure 11: Each thread keeps its private slots in a double-linked list

4.3 Allocating blocks

In contrast to the traditional malloc/free primitives, which deal with dynamic allocations in a contiguous heap, pm2_isomalloc and pm2-isofree manage allocations of arbitrarily-sized blocks within a list of discontinuous slots. Each slot contains a double-linked list of free blocks. Blocks have headers storing their size, as well as pointers to the neighboring blocks in the list.

Block allocations are carried out as follows. When a thread requires some additional storage space, its slots are searched for a large enough free block. In the current implementation, a first-fit strategy is used, but other strategies could be considered as well, especially if fragmentation is to be kept low. If no suitable block is found, a new free slot belonging to the current local node is acquired by the thread. It gets attached to its slot list. Then, a new block is allocated in this new slot. This scheme works for all requests for blocks smaller than the slot size, as long as the node owns at least one slot.

4.4 Coping with large-block allocations

To ensure the compatibility with malloc and free, our allocator can also cope with arbitrarily-sized block requests, larger than a slot. In order to satisfy such requests, the key point is to make up a larger slot out of n regular, contiguous slots and to allocate the block inside this new slot (where n is the smallest number of contiguous slots that would be necessary). For this purpose, the following steps are accomplished.

1. The slot bitmap of the local node is scanned, in order to find the necessary number of contiguous slots. A first-fit strategy is used. If this search is successful, the corresponding slots are given to the thread, which uses them to build up a large slot. This large slot gets attached to the slot list of the thread.

2. If the search fails, a global negotiation phase among all the nodes is launched. The initiating nodes behave as follows.

    (a) Enter a system-wide critical section. No other node is allowed to modify its slot bitmap within this section. (It may still run its code and allocate/free *blocks*, as long as no *slot* management is necessary.)

(b) Gather the local bitmaps of all nodes.

(c) Compute an *global or* taking all bitmaps as operands.

(d) Search for the first series n contiguous available slots in this *global* bitmap and "buy" the non-local slots. It suffices to mark these slots are marked with "1" in the bitmap of the requesting node and "0" in the bitmap of their original owner node.

(e) Send back the updated bitmaps to their respective nodes.

(f) Exit the system-wide critical section.

Notice that the same algorithm may be used if a node has run out of slots. It simply enables a node to buy slots from some other nodes.

A global negotiation is obviously an expensive operation, because of the global communication required. It should therefore be kept as exceptional as possible. Two main factors have an impact on the frequency of these negotiations: the slot size and the initial slot distribution. Since all single-slot allocations are guaranteed to be local, the slot must be large enough to avoid multiple-slot allocations as much as possible. On the other hand, even for such allocations, negotiation may be avoided if the necessary number of contiguous slots are locally available. It is therefore important to choose a "good" initial slot distribution, in order to avoid negotiations even more. Observe that there is no restriction whatsoever on the initial distribution.

Notice also that the manipulation of the bitmaps on the local node may be completely arbitrary. It is in particular possible for the local node to to take advantage of a negotiation phase to "pre-buy" slots in prevision of foreseeable large allocation requests. It is also possible to completely restructure the slot distribution at the system level, for instance by grouping contiguous free slots as much as possible on the various nodes. The only requirement is that each slot present in the bitmaps must finally belong to exactly one node.

5 Performance and optimizations

We present here some results obtained on our PoPC cluster. Each node consists of a 200 MHz PentiumPro processor. The operating system is Linux 2.0.36. The nodes are interconnected by a Myrinet network from Myricom [10] accessed through the BIP low-level communication interface [13].

The time needed to migrate a thread with no static data between two nodes is less than 75 μs. It was measured by means of a thread ping-pong between two nodes. This time includes packing the thread resources, transferring them over the network, allocating the memory on the destination node and unpacking the resources. Notice that no post-migration processing whatsoever is needed thanks to our iso-address approach. This time should be compared to the 150 μs reported for the migration of a null thread in Active Threads [14]. This performance figure is partly due to the very efficient Madeleine communication layer used by PM2 [3].

Using the `pm2_isomalloc` function instead of the usual `malloc` induces a non-significant overhead for the requests of blocks larger than one slot, as shown in Figure 12. This overhead is mainly due to the negotiation automatically required by any multi-slot allocation when the slots are distributed in a round-robin way (which is the case in our experiment). This negotiation takes 255 μs in a 2-node configuration when using BIP/Myrinet. If the underlying architecture provides more than 2 nodes, another 165 μs should be added per extra node. Notice that, for large allocations, this overhead is small and rather insignificant compared to the total allocation time (see Figure 12, bottom). We can thus conclude that our approach scales well.

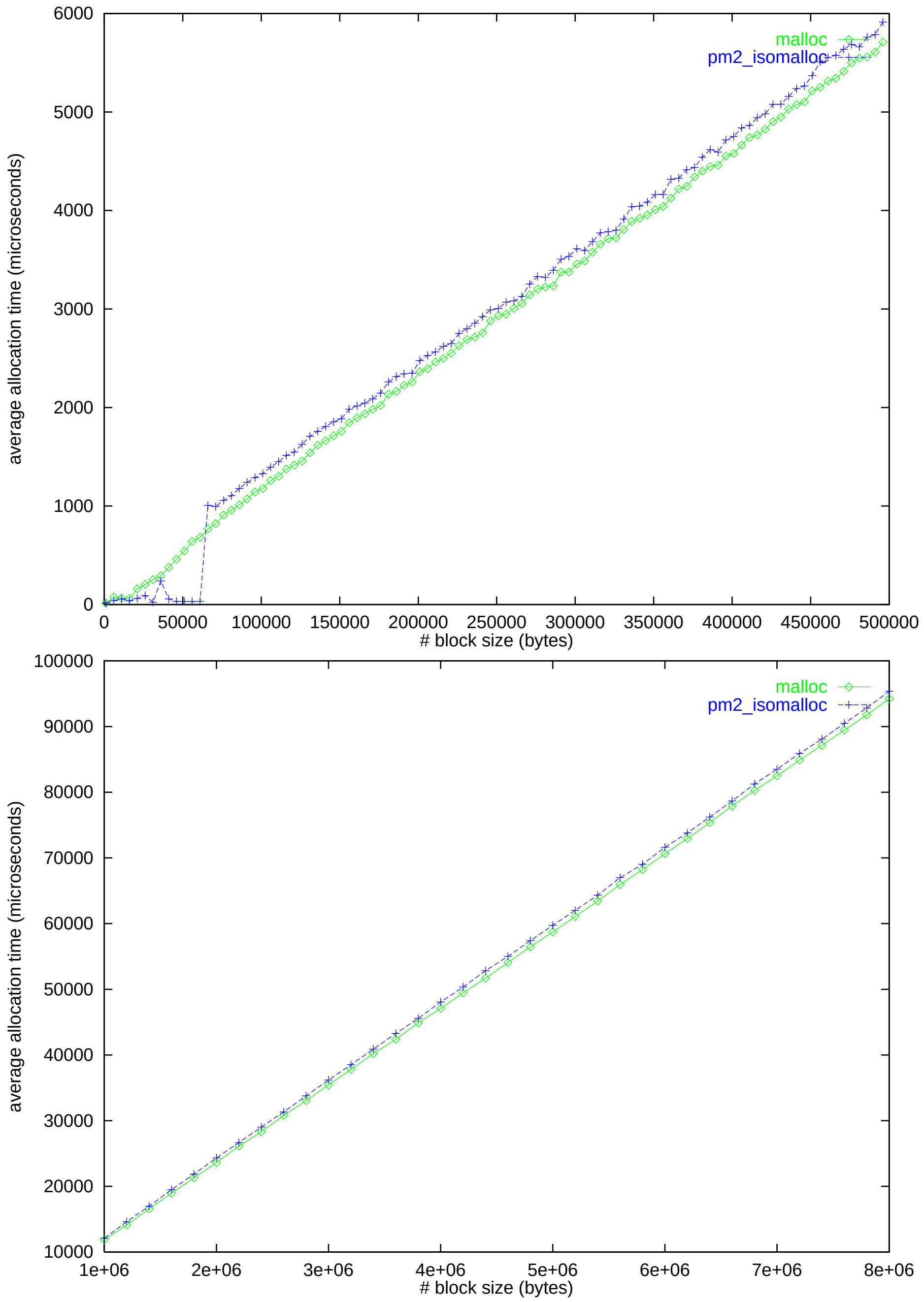

Figure 12: Compared performance of `malloc` and `pm2_isomalloc` for respectively small and large requests in a 2-node configuration.

6 Conclusion and future work

To validate our approach, we have integrated the iso-address allocation primitives in the runtime libraries used by two data-parallel compilers [4, 7]. These compilers have been previously modified, in order to generate multithreaded code for PM2 [12, 2]. Thanks to our new allocator, the runtime code responsible for thread migration was significantly simplified. Given that pre- and post-migration processing were reduced, we could notice an improvement of our virtual processor migration time. We are currently working on these aspects.

A number of optimizations have been considered on top of the general scheme presented. Instead of unmmapping a slot each time it is released, we keep a number of mmapped empty slots in a process-wide cache. This saves the mmapping time at the next slot allocation. When migrating a *slot* attached to a thread, it is sufficient to send its internally allocated *blocks*. Additional details on the current implementation and a downloadable version can be found at http://www.ens-lyon.fr/~rnamyst/pm2.html.

References

[1] G. Antoniu, L. Bougé, and R. Namyst. An efficient and transparent thread migration scheme in the PM2 runtime system. In *Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP '99)*, Lect. Notes in Computer Science, San Juan, Puerto Rico, April 1999. Springer-Verlag. Held as part of IPPS/SPDP 1999, IEEE/ACM. To appear.

[2] L. Bougé, P. Hatcher, R. Namyst, and C. Perez. Multithreaded code generation for a HPF data-parallel compiler. In *Proc. 1998 Int. Conf. Parallel Architectures and Compilation Techniques (PACT'98)*, ENST, Paris, France, October 1998. Preliminary version available at ftp://ftp.lip.ens-lyon.fr/pub/LIP/Rapports/RR/RR1998/RR1998-43.ps.Z.

[3] L. Bougé, J-F. Méhaut, and R. Namyst. Madeleine: an efficient and portable communication interface for multithreaded environments. In *Proc. 1998 Int. Conf. Parallel Architectures and Compilation Techniques (PACT'98)*, pages 240–247, ENST, Paris, France, October 1998. IFIP WG 10.3 and IEEE. Preliminary version avaiable at ftp://ftp.lip.ens-lyon.fr/pub/LIP/Rapports/RR/RR1998/RR1998-26.ps.Z.

[4] Th. Brandes. *Adaptor (HPF compilation system), developped at GMD-SCAI.* Available at http://www.gmd.de/SCAI/lab/adaptor/adaptor_home.html.

[5] J. Casas, R. Konuru, S. W. Otto, R. Prouty, and J. Walpole. Adaptive load migration systems for PVM. In *Proc. Supercomputing '94*, pages 390–399, Washington, D. C., November 1994. Available at http://www.mcs.vuw.ac.nz/~pmar/refs.html#R545.

[6] D. Cronk, M. Haines, and P. Mehrotra. Thread migration in the presence of pointers. In *Proc. Mini-track on Multithreaded Systems, 30th Intl Conf. on System Sciences*, Hawaii, January 1997. Available at URL http://www.cs.uwyo.edu/~haines/research/chant.

[7] P. J. Hatcher. UNH C*. Available at